

Page No.: NEW LINK
Date: / /

- ## Types

- (Modular)

#	Procedural oriented Progr.	Object-oriented prog. (OOP)
→	Program is written as sequence of instructions eg: recipes	→ Program is an interaction of functions b/w objects
(1)	Top down approach	(1) Bottom up approach
(2)	Data is not secure	(2) Helps in wrapping data and functions in a class.
(3)	Code is interdependent	(3) Helps building secure programs.
(4)	It doesn't have proper way for hiding data.	(4) Code is modular
(5)	Reuse difficult	(5) Can be extended for reuse. C++, Java, Python
(6)	Need less memory	(6) Need more memory than POP
		(7) Systematic.

Code editor: VS code

Programming languages

- serve as middle-man of sorts (interpreter)
- Translate our instructions into machine code

↓
The series of 0's and 1's
that the computer can
understand.

- Each language is unique in how they operate
 - Java/Python - General purpose
 - HTML/CSS - Designed for specific tasks
 - Each language also ~~converts~~ varies in how powerful it is
 - JS isn't used for big problems but,
 - Java & Python are
 - Each language also has an attribute known as power or level
 - Basically how similar it is to machine code
 - Low-level programming languages: Assembly or C.
 - High-level programming languages: Java or Python
- * Lower the level ⇒ More similar to machine code

How we write code?

- We use IDE's (Integrated Development Environments) to write code
 - A place to write, run, and debug code and also convert it to machine code.

→ IDE's are like any other program on our computer except ~~for~~ used for facilitation of code

- Examples- NetBeans, IntelliJ, Visual studios



→ In addition to a place to write code, IDE's also include

- Built in Error-checking
 - ↳ For when code doesn't go right
- Auto-fill for frequently used words
- Project Hierarchy.

→ Each language has its own set of rules you must follow within an IDE

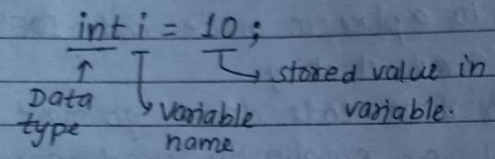
- We can't simply type rubbish in an IDE, convert it to machine code and expect the computer to understand. We need syntax.

Learning languages-

- Learning a computer language can be very similar to learning a real language
 - All prog. languages have a set of rules we must follow when writing code in that language, just like in real languages
 - In computer science it is called syntax similar to grammar in real life languages.

What's the syntax?

- Rules we must follow to a tee if we want our program to run correctly
- Syntax for each programming languages is unique.
- Breaking programming rules will result in an error.



<code>int variable = 3;</code>	<code>x = 3</code>	<code>var x = 3</code>
↓ Java	↓ Python	↓ Javascript

integer

In Java, we specify which type of variable we are defining, and also a semicolon at the end of line	In python, we just type what we want to create	In Javascript, we specify we are making a variable, but we don't define what type of variable.
---	--	--

- Syntax matters when learning a new language
- The goal of the program was the same, but all three languages shown took different approaches

★ If we forget one semicolon or misplace a character, the entire program will not run and send us back a Syntax error.

Real-life examples

- ① Let's Eat, Grandma! → Forgetting a comma misinterprets entire sentence
- ② Let's Eat Grandma! →

- ★ Another cool thing about IDE's is that they will let us know if ~~what~~ and when there are syntax errors in our code
 - parts of our code which do not follow the rules
 - Usually underlined in red. ~~like above~~

How do we get information from computers?

- how do we know if our program worked?

⇒ The Console

- Programmers keep track of their progress by looking at the console.
 - A text interface within the computer that programmers can use for a variety of purposes
- The main use of the console is to output text from the program using code.
 - More specifically a print statement
 - Prints text to the console for the programmers view pleasure.
- To use the print statement, simply instruct the console to print, and then include whatever we want to be printed inside the parenthesis

Eg: In Python, Console:
`print("Hello World")` → Hello world

→ Print statement is also used for viewing and interpreting the computer's output from a program

Eg:

- Computing $4+3$ will print nothing to the console

Console:

$4+3$

- Printing $4+3$ however, will print 7 to the console

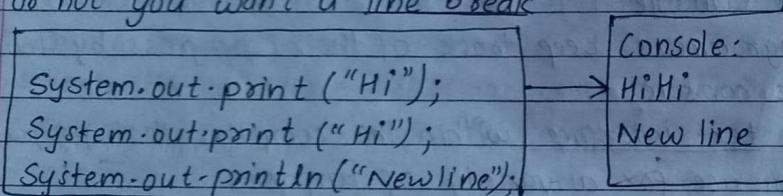
`Print(4+3)`

Console:

7

⇒ Print statement will vary depending on the language

Eg:- In Java, there are actually multiple versions of the print statement depending on whether or not you want a line break



The console is mainly a Developer Tool

→ • Not usually meant to be used and interacted with by the end user except in very abstract cases.

• Text-based games/simple programs

→ Tends to be hidden away behind the scenes

→ Don't try to and implement the console in the final product.

Console: tells how our program is working behind the scene.

What can computer's do? ⇒ Math

- The computer already knows how to do simple arithmetic
→ Addⁿ, subtractⁿ, Multiplicatⁿ & Division
- You'll be able to print the result of any math operation in an IDE that you may install
- May seem useless, but comes in handy extremely often.

Computer are ~~be~~ dumb.

⇒ Building a calculator app

• We'd need to utilize this functionality in order to correctly display the answer to an arithmetic problem.

- Most programming languages has an additional operator known as modulus (%)
- Allows us to get the remainder of a divisional operation.

Eg:-

`Print(10 % 3)`

Console:

1

- In the case where there isn't a remainder. The computer will simply print/return 0.

Eg:-

`Print(50 % 2)`

console:

0

- Evens and Odds

• If a number modulo 2 is 0 → even

• If a number modulo 2 is 1 → odd

What can computer's do: Strings-

→ Strings are another way of saying text

→ "Hello world"

• "A"

• Anything enclosed by quotation marks is denoted as a string.

→

Print("Game over, 4 was your final score.")

Console: Game over, 4 was your final score.

⇒ But score is not fix, it can change ~~each~~ each time we play integral value.

★ Using concatenation ⇒ Adding strings together
eg:

then

Print("Game over," + 4 + "was your final score.")

Console: Game over, 4 was your final score.

⇒ Adding spaces-

print("Game over," + 4 + "was your final score.")

↓

Console: Game over, 4 was your final score.

print("Game over," + 4 + "was your final score.")

↓

Console: Game over, 4 was your final score.

Difference b/w 4 and "4"

• 4 in quotation marks ("4") is treated as a STRING

• 4 without quotation marks (4) is treated as an INTEGER

Variables

• Something that can store information.

• Can be referenced and manipulated

Cardboard Box Examples-

Cardboard Box store items.

Variables store information

• Each variable has a type, a name, and a piece of information stored inside of it.

- Name is simply a name for the variable.

• Types of variables-

- Many types of variables, but we will be focusing on primitive type variables

• Integers, booleans, floats, doubles, strings and chars.

4	False	π	4.0	"Hi"	"A"
Integer	Boolean	Float	Double	String	Char's

* Integer's

• A variable that can store an integer value.

- 2, 147, 483, 648

• Can't and will not hold any decimal values.

Booleans

Can store a value of either true or false.

- can only hold true or false

No other types of information

Floats and doubles

Both are types of floating point data types

Can store numbers with decimal places

Float variables: can store up to 32 bits of information

Double variables: can store up to 64 bits of information

Strings-

Useful for displaying text and storing input information

information the user puts into our program

Also useful for outputting information in a readable

format for the user.

eg: "Hi" "Computer Science" "a"

Char's:

Char → Character

Each hold one character

Useful when a programmer wants to read one button

press of one character in a string without using

a string variable

Eg: Game controlled by keyboard

eg: "#", "x", "a", ".", " "

Tip: We can store char's in a string variable, but not strings longer than 1 character in a char variable.

Why are variables so useful-

• Often times you're going to want to keep track of

things such as user's name or score

- By creating a variable we can store this information

in that variable and then reference, add to or

modify it.

"Sean"	10
↑	↑
Name	Score

• Other important uses for variables

- Taking input from the user

- Making our program have variability

- Have it change based on certain factors

- Manipulating variables is necessary for many tasks in programming.

How do we manipulate variables?

• What happens when we define a variable?

→ creates a little space in memory that stores your variable name and its contents.

→ Manipulating strings

• While you can't subtract, multiply, divide or modulo strings, you are able to add them.

str 1	str 2	dual string
"Hello"	+	"There"
		= "Hello There"

system.out.print(str 1 + str 2)

• Naming variables-

- There is one big rule when naming variables
 - They must be one continuous string.
- Most programmers name variables according to camel case
 - ↳ Don't capitalize the first word, but capitalize the first letter of all words after it.

eg: thePlayerScoreBeforeTheFinalBoss

↳ Easy to read

the playerscorebefore the final boss

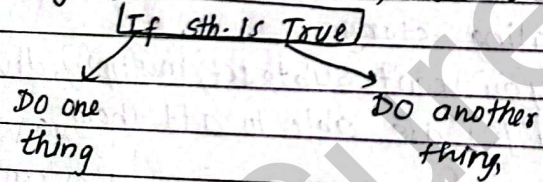
↳ hard to read.

What are conditional statements?

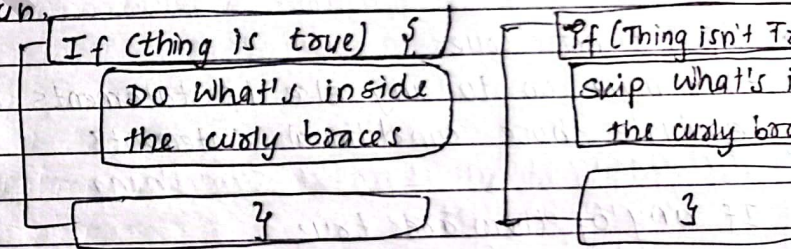
- Red lines → our code will not be going down this path.
- Green lines → our code will be going down this path.

→ The most common/basic cond. statement is the If statement.

- If something is True do this, otherwise do sth. else.



- Most programming languages use braces {}
 - whatever is inside the braces will be evaluated and either true or false.
- If the statement is true, whatever is enclosed in a set of curly braces directly after the if state will run.



eg:-

```
if (name == "Steven") {
  print("Hello Steven")
}
```

```
if (score > 5) {
  print("Nice Score!")
}
```

- Each condition is evaluated as a boolean
 - True or False.
- The Else-if statement will only be evaluated if the if (or if-else) statement was bypassed due to it being false.

- Running through If-Else statements-
 - Check the initial statement
 - If it's true we run that segment of code and move on with the program

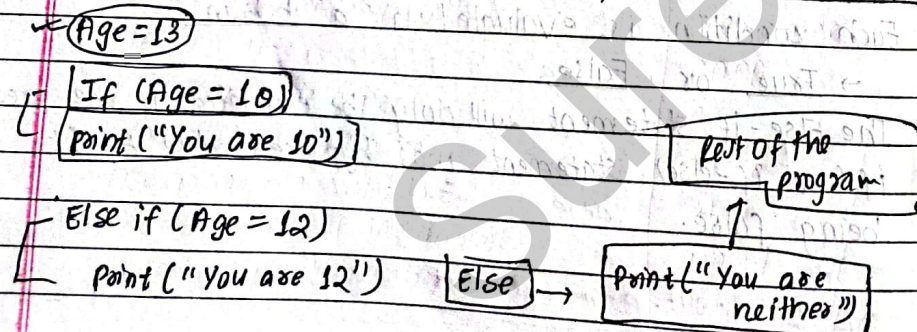
- If it's not true
 - We move on to any else if statements and evaluate those conditional statements

- If any of them are true
 - Run that segment of code and move on.

→ The else statement-

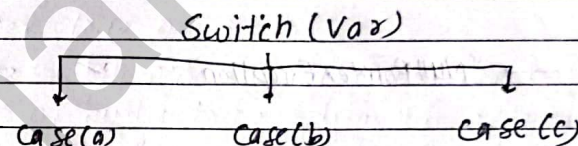
The else statement comes after an if/ if-else statement and will always carry out instructions.

- carries out instructions if all previous cases are not true.



* The switch statement-

- An easily collapsible way to write many if-else statements.
- You input a variable, then determine which "cases" that could be.



- Start with a colon (:)
- Each switch statement also includes a "default" case (Else statement)

* The usefulness of conditional statements-

- Adds variability to programming
- - Program runs differently based on user input
- If a user does something, we want to be able to adapt accordingly.
- Without, a program would run the same way every time.

What are arrays?

: Where do variables fail?

- Variables are very good at storing singular bits of information.

As a result, they are unable to hold more than one piece of data.

Name → "Null Pointer Exception" ✓

Name → "Steven"
Name → "Sean"

- An array is a list of something

- can be Integers
- Can be Strings
- Can even be other arrays

- All information in an array is related

- Like columns in Google sheets.

Eg: Start-up company-

- Start-up company with 100,000 users
- Every time a person wants to create a new username, we want to check to see if the account name has already been taken

usernames
AAAA
aaaa
aaron
alex
arnold
...

* Referencing arrays-

- The single most important thing to note about arrays is how we reference each element inside of them.

- In programming we use Indexes

- That number "place" in the array

- Indexing for arrays begins at 0

Eg:

Numbers	1	2	3	4	5	6	7	8	9	10
	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
Index	0	1	2	3	4	5	6	7	8	9

- If you slip up and accidentally reference the 10th elements in this array

- Would result in an "Out of Bounds" error since you are actually trying to reference the 9th element

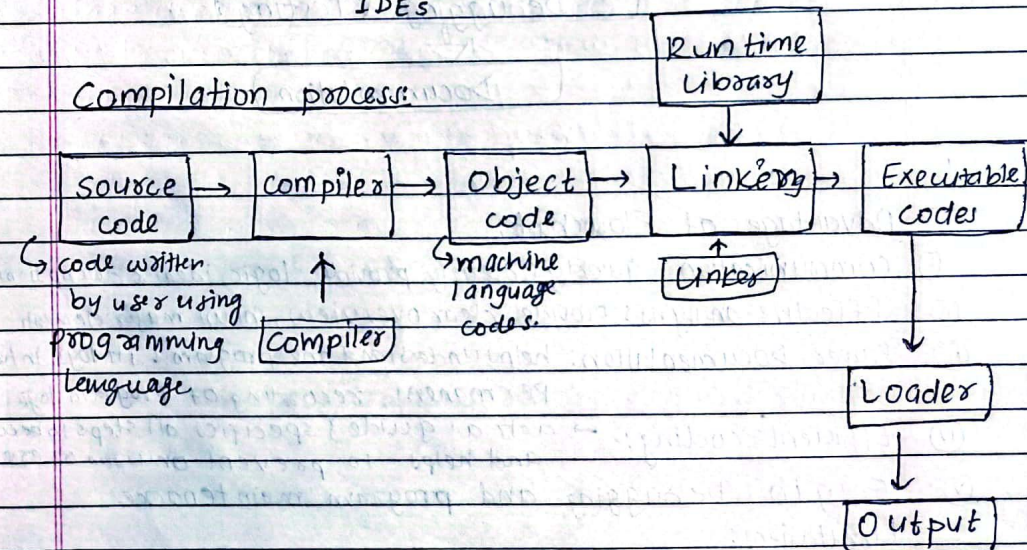
Steps involved in coding to execution

- (i) Program writer (also called coder) creates .C source files which is then passed to compiler producing .obj or object file.
- (ii) .obj Object files are passed to linker/loader which includes all necessary library files to produce .exe executable files.
- (iii) .exe is the executable file which is then loaded into the memory to be executed by the user. User can run this executable file anytime to perform higher tasks required.

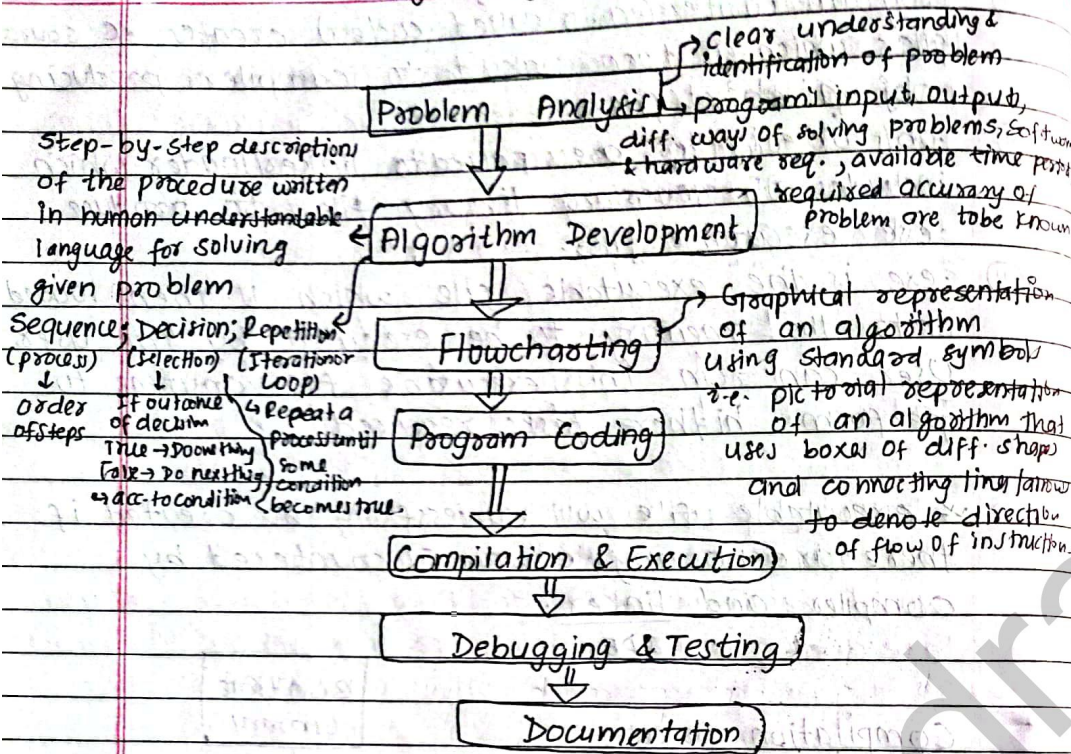
✶ executable file will successfully be created if there are no any errors encountered by compiler and linker.

IDEs

Compilation process:



Problem solving Using computer



Advantages of Flowchart:-

- (i) **Communication**: quickly & clearly provides logic, ideas & descriptions of algo.
- (ii) **Effective analysis**: provides clear overview, shows major elements.
- (iii) **Proper Documentation**: helps understand the program & its logic in future, permanent recording of program logic.
- (iv) **Efficient coding**: acts as a guide, specifies all steps to be coded and helps to prevent omission or error.
- (v) **Easy in Debugging and program maintenance**.

Limitation:

- (i) **Complex logic**: For complex logic it becomes complex & clumsy.
- (ii) **Difficulty in alteration & modifications** → need to be redrawn completely.

While loop → checks condition → True → Do process
False → stops.

If.. then.. goto loop → Repeat a process until the given condition is False.

Repeat loop → Repeat a process until the given condition becomes True.

Program coding:

Coding is the process of transferring paper works like algorithm and flowchart into computer program following the well defined syntaxes used in programming language like C, C++, Java, etc.
Only 20-30% of total task is done to solve problem using computer.

Compilation & Execution:

Compilation: Process of changing HLL into MLL

- done by using special software: Compiler
- Tests if program contains syntax error or n
- If error is present → cannot compile the code.

Debugging & Testing.

Debugging: discovery & correction of programming errors.

→ error might appear during compilation, linking or execution.

Semantic error → logical error
↳ generally captured while testing the output & verifying the result.

Program documentation

↳ systematic writing of activities from problem analysis to debugging/testing phases including operational information and presenting it as a report.
↳ description of program and its logic written to support understanding of program in future & for other programmers.

Why C Programming?

- 50 years old +
- top 10 PL every year
- fast
- Most of the OS, Mac, Windows's most part are written using C
- Ruby, C Python, Java initial compiler.
- use to write drivers programs for devices like tablet, printer, etc.
- use to develop games on area where latency is very important.

Basic structure of C program.

- Program's execution starts from main function
- all statements are terminated with semicolon
- Instructions are case sensitive
- Instructions are executed in same order in which they are written.

↳ Shortcut for comment: `ctrl + /`

↳ Syntax for printf (output): `printf(" ", arguments);`
Control String

↳ `%` → only for integer.
→ Hgn is same as of numerator.

↳ `int a = 3.6` Float 3.6 is demoted to '3(int)' as a not able to store float.
`float a = 8;` a will store 8.0

↳ else block in if-else statement is optional.

↳ while loop: `while (condition)`
{
 // code
}
} block keeps executing as long as condition is true.

eg: `int a = 1;` ⇒ initialization
`while (a < 11)` ⇒ condition
{
 `printf("%d", a);`
 `a++;` ⇒ updation.
}
↳ loop counter can be integer or float.

Enter $\Rightarrow$ is also a character.

Do while loop

Syntax:

do {

// code;

// code;

} while(condition);

Do while works similar to while loop

Note:

(i) while loop $\rightarrow$ checks condition & execute code

(ii) Do-while $\rightarrow$ executes code & checks condition.

Do-while $\rightarrow$ while loop which executes at least once

(3) For loop

Syntax:

for (initialize; test/condition; updation)

{

// code;

// code;

}

setting loop counter to initialize value.

increment or decrement

Whenever 'break' is encountered inside the loop, the controls are sent outside the loop.

continue statement skips the particular iteration of the loop.

Function prototype,
Function call,
Function definition

* When a function is called from main(), the main function falls asleep and get temporarily suspended. During this time, the control goes to the function being called. When the function body is done executing, main() resumes.

Every function gets called directly or indirectly from main().

Reasons to use function

avoid rewriting the same logic.

to keep track what we are doing in a program.

to test and check logic independently.

* Passing value to function

eg: int sum(int a, int b)

$\Rightarrow$ Above prototype means the sum is a function which takes value 'a' (of type int) and 'b' (of type int) and return a value of type

Parameters: values or variables place holder in funcⁿ definiⁿ

Arguments: Actual values passed to the funcⁿ to make a call.

Function returns one value at a time

$\text{sqrt}() \rightarrow \sqrt{\quad} \rightarrow \langle \text{math.h} \rangle$

Page No.: NEW LINK
Date: / /

* Pointers store addresses rather than values.
↳ stores address of another variable.

Declare pointer

Syntax

`int *j;` // declare a variable j of type int - pointer
`j = &i;` // store address of i in j.

Pointer can be char, float type as well.

* pointer to a pointer

Just like j is pointing to i or storing the address of i, we can have another variable k, which can store the address of j.

Q. What will be the type of k? ⇒ pointer to pointer

`int **k;`

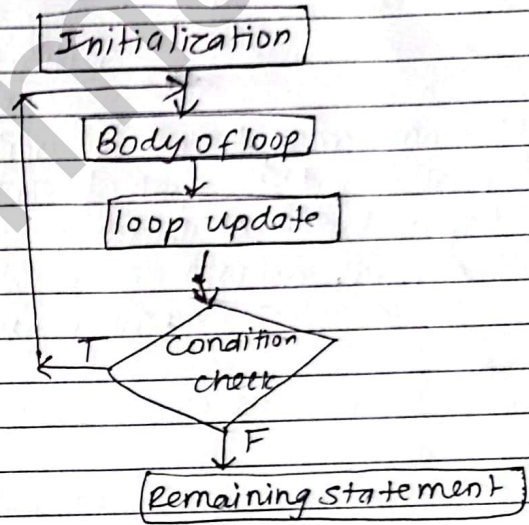
`k = &j;`

i	j	k
72	87994	87998
87994	87988	88004
int	int*	int**

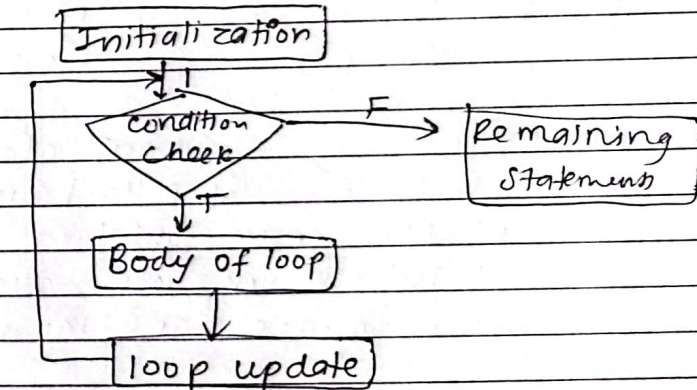
We can go further one level and create a variable of type `int ***` to store the address of k. We mostly use `int*` and `int**` some times in real-world programs.

Page No.: NEW LINK
Date: / /

do-while loop (Exit controlled loop)



for/while loop (Entry controlled loop)



S. No.

Date

Title

Page
NoTeacher's
Sign /
Remarks

Surendra Sharma

Programming

078 BCE 178

www.surendrasharma0001.com.npUnit - 1

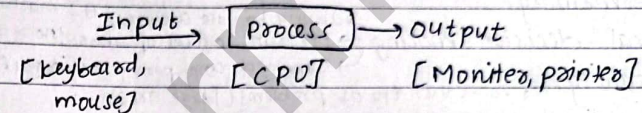
classmate

Date

Page

Computer

→ An ^{programmable} electronic device, that accepts data and instructions, process data and execute the instructions and displays the output.

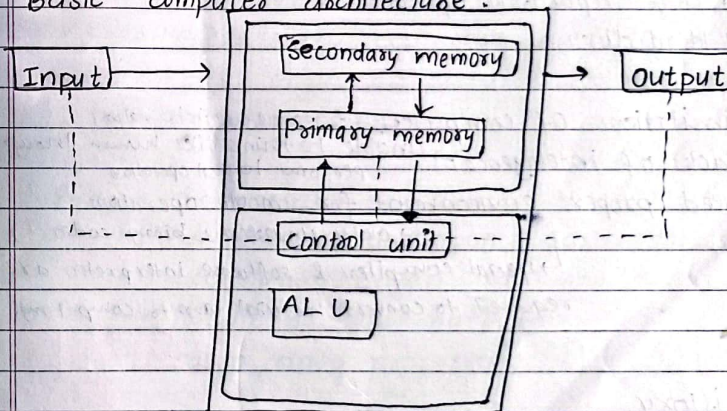


Components of computer

(a) Hardware: Physical component

(b) Software: computer programs

Basic computer architecture:



Characteristics of a computer

High speed

- Millions of instructions per second.

Reliability & accuracy ^{error checking system} → precise; works on electrical pulses
^{store data temporarily & permanently.}

Large memory

Logical decision making ^{ability to take decisions; AI machines} → perform logical operations ($no > =, < =$)
^{capable of comparing text, images & audio}

Versatility → solve variety of problems (large area)

Automation → perform operations automatically

Diligence → never feels tiredness

→ perform single or multiple task without losing precision

Temporary memory

→ RAM

→ erased when power is off.

Permanent memory - permanently store

→ large repository of data.

→ Hard disk

Limitations of computers

Lack of intelligence → unable to think like human though perform logical operations

Need proper environment for smooth operation.

→ only understands binary codes

→ special compilers & software interpreters are required to convert natural lang. to comp. lang.

History

Abacus - 5000 years ago

→ manual counting device

→ beads & rods.

1642 → French mathematician Blaise Pascal

→ Pascaline

→ Telephone dial: revolved to make calculation.

1801 → French weaver Joseph-marie Jacquard

→ Jacquard loom

→ Punched cards

→ movement of needles, threads & fabric

1842 → Charles Babbage

I. Difference engine

II. Analytical engine

→ General purpose: add, subtract, divide & multiply

Ada Augusta Lovelace

→ punched cards could be used to instruct

Babbage's engine to perform repetitive task.

→ First programmer.

1890 → Dr Herman Hollerith

→ Electro-mechanical punch card tabulating machine

→ Hand punch: enter data into card

→ Tabulating dial: output

→ Modern data processing.

First modern digital computer - Atanasoff-Berry computer

→ built in Iowa state university

→ Professor John Atanasoff and a graduated student Clifford Berry

→ Binary arithmetic, parallel processing, storage

→ Store 3000 bits

→ 750 pounds

- ✓ 1944 → First electro mechanical computer
 - ↳ Mark I
 - Harvard university professor Howard Aiken
- ✓ 1946 → First large-scale digital computer
 - ↳ Electronic numerical integrator & calculator
 - ↳ John Mauchly & John presper Eckert
 - ↳ ENIAC
- ✓ 1954 → IBM
- ✓ 1959 → Dr. Grace Hopper
 - ↳ Cobol programming language
 - ↳ General purpose english like programming language
 - ↳ Pioneer to find first real 'Bug'

Generations of computer

- ① First generation (1942-1955) → UNIVAC, EDVAC, ENIAC
 - vacuum tubes (glass bulb & wires)
 - Large, expensive, require huge amt. of electricity
 - ENIAC (Electronic Numerical Integrator And Calculator)
 - relied on machine language; could solve only one problem at a time
- ② Second generation (1956-1964): IBM 1620, IBM 7090
 - Transistors instead than vacuum tubes
 - ↳ smaller; semiconductor material that amplifies a signal or opens or closes a circuit.
 - smaller, faster & cheaper
 - relied on punched cards for input and printouts for output.
 - assembly language.
- ③ Third generation (1965-1975)
 - Integrated circuit (IC)
 - faster, cheaper & smaller than 2nd gen
 - IC: silicon chip that embed an electronic circuit, which comprises several components such as transistors, diodes & resistors
 - Punched card replaced by keyboard and monitor interfaced with OS which allowed device to run diff. applications at one time.
 - IBM 370, CDC 7600

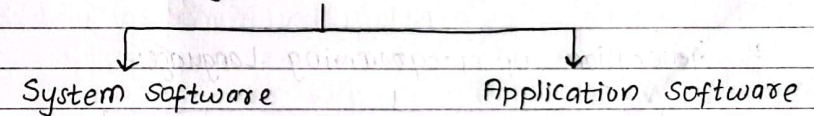
→ can do assembly level prog to high level programming
 ↳ Middle Level Language
 ↳ Features of both low level (can access memory directly using pointers) & High level (user-friendly)

- ④ Fourth generations (1975 - the present)
 - Microprocessors: small silicon chip on which thousands of circuits are placed by use of LSI & VLSI (Very Large Scale Integration)
 - OS like MSDOS, MS Windows with Graphical User Interface (GUI) & user friendly applications → Pentium II, Pentium III/IV.
- ⑤ Fifth-generation (the future)
 - most advanced
 - can think on their own: Artificial Intelligence (AI)
 - Robots
 - VLSI technology: more powerful & faster than microprocessor
 - AI: voice recognition system, parallel processing, super computing robotics, soon to achieve natural language processing.

Software

- Software are the computer programs
- Sequence of instruction designed to direct a computer to perform certain functions.

Types of Software



- ✓ System software: It is the computer programs that operate, manage and control computer hardware.
 - This type of software directly interact with computer hardware.
- ✓ Application software
 - The computer programs that perform specific task.
 - Application sw are managed by system sw.

Users

Application Software Eg: word, Excel, browser, etc.

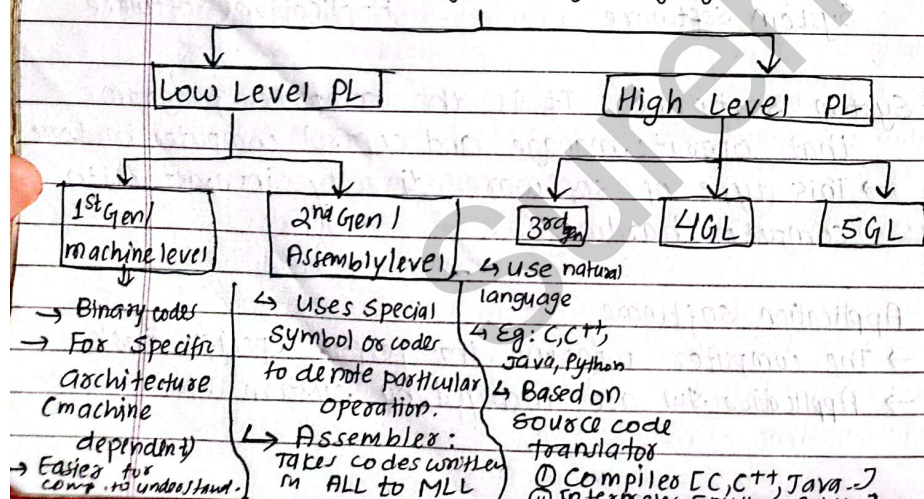
System Software Eg: - OS, device drivers

Computer Hardware

Features of software:

- (i) **Functionality** - should perform its task
- (ii) **Usability** - user friendly
- (iii) **Portability** - portable to different system with diff. architecture
- (iv) **Efficiency** - memory & CPU efficient (low memory consumption)
- (v) **Maintainability** - maintain i.e. add new features to existing software.
Sub → bugs

Generation of programming Language:-



3GL

code

```

int main() {
    int a=2, b=3, res;
    res = a+b;
    printf("%d", res);
    return 0;
}

```

↳ C-source code.
test.c

Python ^{source} Code

```

a = 2
b = 2
res = a+b
print(res)
test.py

```

→ Compiler scans whole source code at a time and converts it into object code.

↳ difficult debugging ↳ Fast ↳ Not memory efficient

→ Interpreter executes one instruction at a time and directly convert it into machine code.

↳ easier debugging.

↳ memory efficient: as no intermediary code is generated.

4GL

→ It is called as non-procedural PL

→ Example: SQL (Structured Query Language)

→ typically for processing large volumes of data

→ database queries, data manipulation, analysis & reporting

5GL

↳ uses natural language

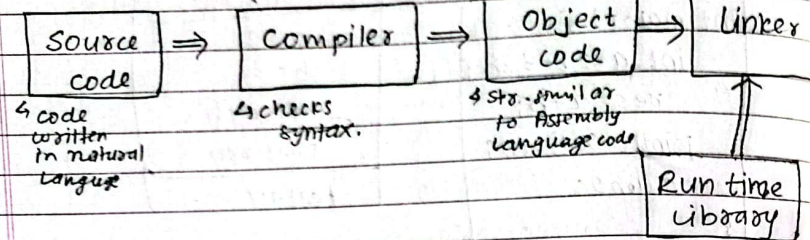
↳ uses AI (Artificial Intelligence)

↳ NLP (Natural Language programming processing)

↳ ML (Machine learning)

↳ computer vision eg: Google assistant

Compilation process:



Problem solving using computers

→ Systematic process to design, develop and implement software system by using computer based tools.

Software: collection of computer programs

computer programs: set of well defined instructions.

Steps in software development process:

- ① Problem Analysis
- ② Program design (Algorithm, Flowchart, Pseudocode)
- ③ Coding / Programming
- ④ Compilation and execution
- ⑤ Debugging and Testing
- ⑥ Program documentation

① Problem Analysis

- * specifying the objectives
- * Identify the output
- * Specify the input
- * Specify the process
- * Feasibility study
- * Documentation (Software Requirement Specification SRS)

② Program design:

- ① Algorithm
- ② Flowchart
- ③ Pseudocode

① Algorithm

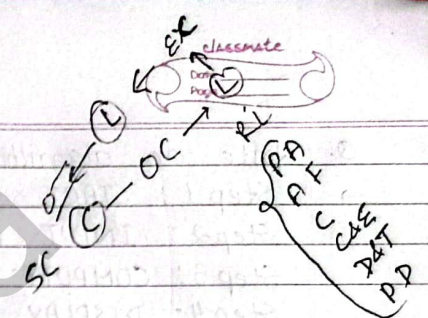
→ Algorithm is a well-defined procedure to solve a problem using finite number of steps.
→ in human understandable form
→ It is step-wise descriptions of solution to a problem.

* Features of algorithm:

- It should have finite number of steps
- It may or may not have Input specified
- It should produce an output
- It should not contain any ambiguity (instructions should be easy to understand by all)
- You can have n-number of algorithm for a program.
- All the steps should be executed in finite time.

* Algorithm design guidelines:-

- Algorithm should start with START/BEGIN and ends with STOP/END
- Accept input using INPUT/GET/READ
- Display output using OUTPUT/PRINT/DISPLAY
- To compute the result use keyword COMPUTE/PROC



Q. Write an algorithm to find sum of two numbers.

→ Step 1: START

Step 2: INPUT two numbers : a, b

Step 3: COMPUTE: $Sum = a + b$

Step 4: DISPLAY sum

Step 5: STOP

Q. Write an algorithm to check whether the given number is positive or negative.

→ Step 1: START

Step 2: INPUT a number : a

Step 3: If $(a < 0)$

DISPLAY "Negative number"

else

DISPLAY "Positive number"

Step 4: STOP

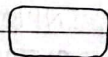
Flowchart

→ Flowchart is a graphical (pictorial) representation of an algorithm using standard symbols.

→ It uses different symbols to denote the flow of execution.

* Standard symbols used in flowchart

① START/STOP:



START

STOP

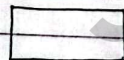
② INPUT/OUTPUT:



INPUT

OUTPUT

③ COMPUTE:



COMPUTE

$c \text{ num} > 0 ?$

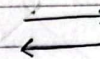
④ Decision



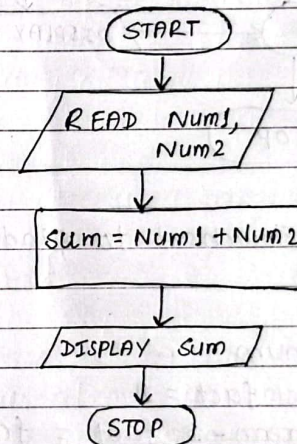
⑤ Connector:



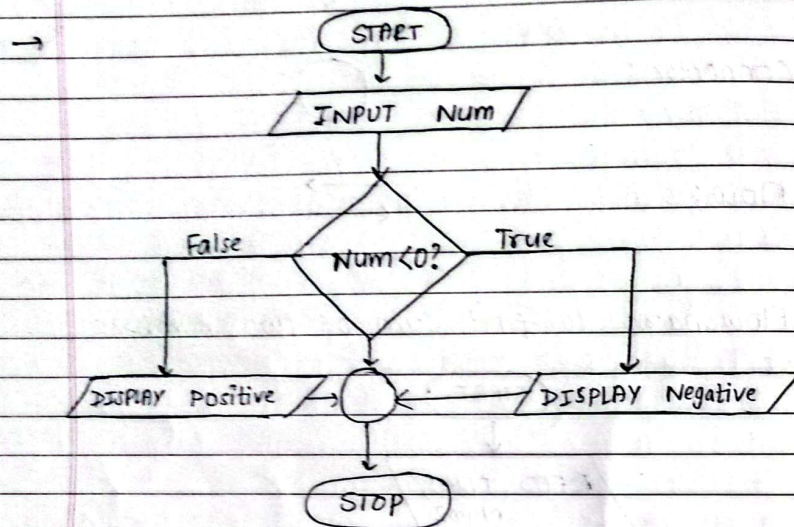
⑥ Flow:



Q. Flowchart to find sum of two numbers



9. Flowchart to whether given number is even or odd positive or negative.



9. Algorithm and Flowchart to find factorial.

Algorithm:

→ Step 1: START

Step 2: INPUT num

Step 3: Initialize fact = 1

Step 4: ~~num = num - 1~~ fact = fact * num

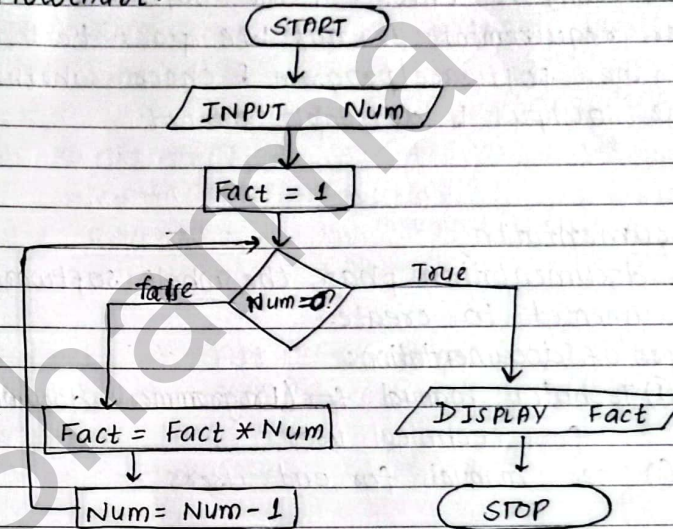
Step 5: num = num - 1

Step 6: If num = 0 GOTO step 7
else GOTO step 4

Step 7: DISPLAY fact as factorial of number num

Step 8: STOP

Flowchart:



③ Coding

→ In this process, a programmer writes the computer programs using programming language like C, C++, Python, etc.

④ Compilation & Execution

The written code is compiled using compiler & is checked if there are syntax errors or not. Compilation is completed then object code is linked with library function & is loaded into memory for execution.

⑤ Debugging and testing

→ Debugging is process of finding errors in the source code. If any error is found, then it is corrected and the program is passed for testing.

→ In testing we check if the software meets user requirements or not. We pass test data to the software/program & check whether the output is desirable or not.

⑥ Documentation

→ In documentation phase, the whole software is documented to create.

Types of documentation:

- ① Technical manual ~~for~~ (Programmer's documentation) for technical user
- ② User manual for end users.

→ Introduction to C-programming Language

header file (.h)

P1.
 Header file (preprocessor directive)
 #include <stdio.h>
 #include <conio.h> → console input/output { clrscr, getch }
 int main()
 {
 clrscr(); → after/below declaration statements
 printf("Hello World"); → clears output screen (where previous program output generated)
 getch(); → Argument inside parenthesis of function
 return 0; → display the program till it gets character
 }
 Output ⇒ Alt F9 (Compile)
 ⇒ Ctrl F9 (Run)
 Output: Hello World

return: 0 → successful termination
 other integers → Unsuccessful termination

Alphabets → words → sentences → paragraphs
 ↓ ↓ ↓ ↓
 [A-Z] Keywords
 [a-z] → variables → Instructions → Program
 [0-9] constant
 Special symbols

Variables

- Variable is a name given to memory location.
- The value of location keeps on changing.

constants

- constants are actual value.

→

Constant

Primary constant
 → Integer constant,
 Real constant,
 Character constant

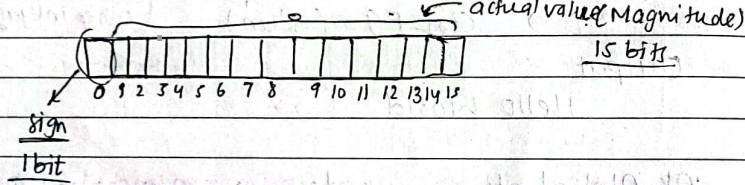
Secondary constant
 → Array,
 Structure,
 Union, etc.

Storage depends on computer architecture

* Integer constant

- Integer constants must have at least one digit
- require 16 bits of storage
- whole numbers i.e. non-fractional
- spaces, comma, and decimal points are not allowed.
- Integer constants can be +ve or -ve.
- Range of integer values depends upon size of compiler.

16 bit: -32768 to 32767



Eg:- 10, 20, -2, -3, etc.
→ 2 byte

* Real constants (float)

- Real constants are used to denote decimal values
- size of float is 4 byte [Though depends on computer architecture]
- size of double is 8 byte (For high precision)

* Character constants:

- Character constants are single letter digit or special symbol enclosed within single quote ' '.
- Memory required: 1 byte
- A → 65 (ASCII) a → 97
- Z → 90 (ASCII) z → 122
- 0 → 48 9 → 57

ASCII → American Standard Code for Information Interchange

Q. Learn about ASCII value for special characters

Variable declaration

← ~~data~~ syntax:

<data type> <variable name>;

Eg:

int age; → Reserve (int) size of memory space and name the space as "age"

char gender;

float weight;

double wt - of - atom;

2 byte

age ← variable name

2 byte ← size

101040020 ← address.

int age; → Declaration int age = 10;

age = 10; → Initialization

Rules to declare variable name:

- variables name consists of any combination of alphabets, digits and underscore (_).

ab12_ ✓

ab12\$ ✗ (invalid)

- variable name should start with letter i.e. alphabet or underscore.

ab → ✓

ab1 → ✓

lab → ✗

-ab ✗

ab_ ✓

a_b ✓

a b x x

- ⇒ Length of variable should not exceed 31 characters.

- variable names should not be keywords.

- Variable names are case sensitive i.e. uppercase & lowercase are different.

→ spaces are not allowed.

Keywords

→ Keywords are reserved names
(predefined words for C programming)

→ There are 32 keywords in C.

Eg: auto, case, const, default, enum, signed, etc.

#

→ `int a = 10, b = 20; ✓`

→ `int a, b, c;
a = b = c = 10; ✓`

→ `int a = b = c = 10; ✗`

Need to remember following, while writing instructions:

→ All the variables must be declared

→ All the instructions, must be written in order in which it should be executed.

→ All the instructions, must be terminated by semi-colon (;)

→ Always use small case letters while writing ~~the~~ instructions.

→

Comment Lines

Preprocessor Directives (collection of special statements that are executed at beginning of compilation process)

Main()

{

Statement 1;

Statement 2;

Statement 3;

}

"Beem is good"
printf("Beem is good\n");
//n

10th In
classmate
Date
Page

Q WAP to add two numbers:

include <stdio.h>

include <conio.h>

int main()

{

int x, y, res;

x = 10, y = 20;

res = x + y;

printf("The sum of %d and %d is %d", x, y, res);

getch();

return 0;

}

Syntax of printf function:

printf("<format string>", <list of variables>);

Format string:-

%d → int

%c → char

%f → float

ch-5

Input and output functions in C

(a) Formatted IO - uses format string

(b) Unformatted IO → do not use format string.

(a) Formatted Output/write operation

Syntax → Performed by using printf() function

printf("<format string>", <list of variables>);

eg:-

int a = 10, b = 20;

float x = 20.05;

char ch = 'c';

printf("%d %d %f %c", a, b, x, ch);

Output:

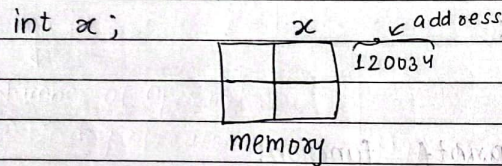
10 20 20.05 c

(b) Formatted Input/Read operation

Syntax: $\rightarrow$ scanf()

scanf("<format string>", <list of variables with the & address>);

& $\rightarrow$ address of operator (ampersand)



$\rightarrow$ To store the value entered by user at address of x

&x $\rightarrow$ address

x $\rightarrow$ garbage value

scanf("%d", &x);

$\rightarrow$ #include <stdio.h>

#include <conio.h>

void main()

{

char ch;

printf("Enter a character:");

scanf("%c", &ch);

printf("You entered: %c\n", ch);

getch();

}

Output:

Enter a character:

A

You entered: A

(b) Unformatted Input output

Input:

getchar() / getch()

Output:

putchar() / putch()

Eg: void main()

{

char ch;

printf("Enter a character: \n");

ch = getchar();

putchar(ch);

}

Preprocessor Directives:

$\rightarrow$ preprocessor directives are special instructions, which are processed before the source is passed to compiler.

$\rightarrow$ all begins with # (hashtag)

Eg: #include $\rightarrow$ File inclusion

#define PI 3.1415 $\rightarrow$ define symbolic constant PI as 3.1415

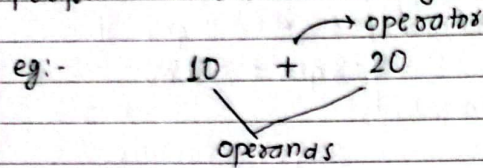
#define TRUE 1

#define FALSE 0

Division: $\text{int}/\text{int} = \text{int}$ • $\text{float}/\text{float} = \text{float}$ • Deno not equal to 0
 $\text{int}/\text{float} = \text{float}$ • $\text{float}/\text{int} = \text{float}$

Operators in C programming

- Operators are symbols that act upon operands
- operands are values
- perform mathematical or logical manipulations.



On the basis: Number of Operand	Utility/Function.
(a) Unary operators C programming supports 3 unary operators → require only <u>one operand</u> eg: ++, --, -, ~ → used in decision making eg: of exp. involving unary operators: ++a, --a, -a, ~a unary plus	(i) Arithmetic operators → +, -, *, /, % modulo or remainder operators.
(b) Binary operators → require <u>two operands</u> eg: +, -, *, /, %, <, >	(ii) Relational operators → <, >, <=, >=, ==, != → used to compare two similar operands. → used in decision making.
(c) Ternary operators → require three operands → eg: Conditional operator ? : Syntax: $\text{value} = \text{exp1} ? \text{exp2} : \text{exp3}$ Conditional	(iii) Logical operators → used to compare or evaluate logical and relational expressions → &&, , ! AND OR NOT → operands of logical operator must be either Boolean value (1 or 0) or expressions that produce Boolean value.

class(c) → after declaration statement

abs() → absolute value
 classmate
 Date: _____
 Page: _____

- output is always either 1 (true) or 0 (false)
- && → produce true if both operands are true. otherwise false
- || → produce true if at least one of operands is true otherwise false.
- ! → produce true if operand is false & produce false if operand is true.

(iv) Assignment operator → '='

→ assign result of an expression to a variable

→ Shorthand/Arithmetic assignment operator

+ = ⇒ a + = b means a = a + b

- = ⇒ a - = b means a = a - b

* = ⇒ a * = b means a = a * b

/ = ⇒ a / = b means a = a / b

% = ⇒ a % = b means a = a % b

(v) Increment and decrement Operators.

→ used to increase the value of operand by 1: Increment oper

→ prefix notations: ++a and --a

→ operator before operand

→ first incremented or decremented then u

→ postfix notations: a++ and a--

→ operator after operand.

eg: $b = a++$ $b = ++a$
 $b = a;$ $a = a++;$
 $a = a++;$ $b = a;$
 → previous value of variable is used the variable is incremented or decremented

(vi) Conditional operators

→ ? :

→ takes three operands so, also called ternary operator

Syntax: $\text{Value} = \text{exp1} ? \text{exp2} : \text{exp3}$

i.e. If exp1 is true, exp2 is final value,
 (else) / exp1 is false exp3 is final value.

Hierarchy of data type:

- double
- float
- long
- int
- char/short

Modulus operator → %
(remains operator)
only used for Integer

for (int

value = exp1? exp2 : exp3 is eqv to following

if-else structure.

if (exp 1)

variable = exp2;

else

variable = exp3;

Code for relational operator

#include <stdio.h>

#include <conio.h>

int main()

{

int a=2, b=2, c=3;

printf("%d\n", (a=b));

printf("%d\n", (a<b));

printf("%d\n", (a!=c));

getch(); // for turbo c++

return 0;

}

Code for logical operator: WAP to print greatest number among

#include <stdio.h>

#include <conio.h>

int main { int a, b, c; a=2, b=3, c=4;

if (a>b && a>c) {

printf("%d is the greatest number", a);

}

else if (b>a && b>c) {

printf("%d is the greatest number", b);

else {

printf("%d is the greatest number", c);

return 0;

In C programming,

→ 0 is considered false value
and any other number except zero is true value.

Q WAP to find largest number among three numbers using conditional operator.

→ #include <stdio.h>

#include <conio.h>

int main () {

int a=5, b=8, c=4; largest;

largest = (a>b)?(a>c?a:c):(b>c?b:c);

printf("%d is the largest", largest);

return;

getch();

return 0;

}

^ → caret

~ → tilde

(Vii) Bitwise operators

→ Bitwise operator, operates on binary bits.

→ Following are bitwise operators:

(i) >> (right shift)

(ii) << (left shift)

(iii) & (Binary and)

(iv) | (Binary or)

(v) ^ (XOR)

(vi) ~ (compliment)

^ → caret

~ → tilde

2³=8

Left shift

int num=8; res;

res = num << 2;

printf("%d", res);

3 2 1 0
□ □ 1 0 0 0 = 8

shift left by 2 bits

5 4 3 2 1 0
1 0 0 0 0 0 send place

when you left shift 8 by 2 bits

we get 32

32 = 2⁵

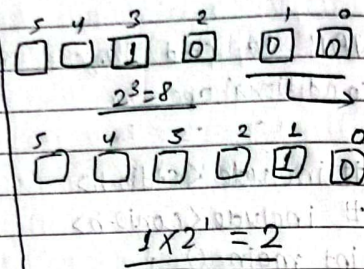
Left shift
multiplication operation

Right shift
Division operation

Right shift

int num = 8;
num >> 2;

$$8/2 = 8/4 = 2$$

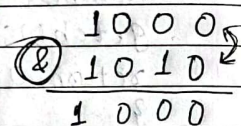


Binary And (&)

int num1 = 8, num2 = 10, res;

res = num1 & num2;
printf("%d", res);

Output
8

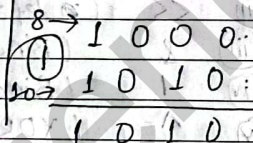


Binary OR (|)

int num1 = 8, num2 = 10, res;

res = num1 | num2;
printf("%d", res);

Output
10



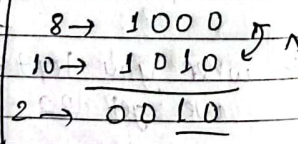
oeling

XOR (^) : Same → 0, Diff. → 1

int num1 = 8, num2 = 10, res;

res = num1 ^ num2;
printf("%d", res);

Output
2

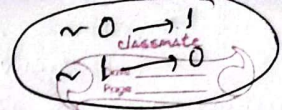


Unary operator

~ (complement)

int num1 = 8, num2 = 10, res;
res = num1 ~ num2;
printf("%d", res);

Output
7



viii) Special operator;

@ size of ()

→ size of () operator returns, number of memory bytes required by values or data types:

Syntax: size of (constant | variable | data type) Argument
printf("Memory req. of int is: %d bytes",

size of (int));

→ operand may be constant, variable or data type qualifier.

⑥ comma operator : evaluated from left to right & value of rightmost expression is the value of combined expression

Operator precedence and Associativity

1. Brackets → (), [] () → Parenthesis, [] → Array index, Left to Right

2. Unary operators (&, *, +, -, ++, --, sizeof, ~, !) → Right to left

3. Arithmetic operator i. *, /, % ii. +, - } → Left to right

4. Left shift (<<), Right shift (>>) → Left to right

5. Relational operator i. <, >, <=, >= ii. ==, != } → Left to right

6. Bit wise operator (i) & (ii) ^ (iii) | } → Left to right

7. Logical operator i. &&, || } → Left to right

8. Conditional: ? : } → Right to left

9. Assignment operator: } → Right to left.
 =, *, /=, %, +=, -=, &=, ^=, |=, <<=, >>=

10. Comma operators (,) } → Left to right.

Summary:
 Bracket
 Unary
 Arithmetic
 Relational
 Logical
 Assignment

eg: $x = 20, y = 5$

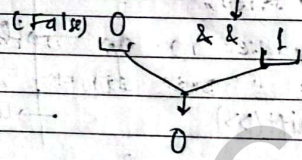
$x = 10 + 15 \ \&\& \ y < 10$

$x = 25 \ \&\& \ y < 10$

$5 < 10$

$x = 25 \ \&\& \ 1$ (True)

$20 = 25$



eg: $\text{int } a = 0, \text{res};$
 $\text{res} = a++ + ++a;$

← Associativity

$a++ + ++a$

$a = 0$

$a++ \rightarrow a = 1$ increase then use

$a++ \rightarrow a = 1$ use

$a = 2$ increase

$1 + 1 = 2$

eg: $\text{int } a = 0;$
 $\text{printf}(\text{"\%d \&\& \%d \&\& \%d \n"}, a, ++a, a++);$

① $a = 0$ use
 $a++ \rightarrow a = 1$ increase

$++a \rightarrow a = 2$ increase
 ② $a = 2$ use

$a \rightarrow a = 2$ use

Type conversion

→ You can convert values in one data type to another which is called type conversion/type casting.

Type casting

Implicit typecast
 (Automatic)

$2 \times 0.5 = 1.0$
 $\frac{2}{5.0} = 0.4$

Explicit typecast
 (Forced typecast)

$\frac{2}{5} = 0$

Explicit typecast

Syntax: $\rightarrow$ (destination type) source value

eg: (float) 2 $\Rightarrow$ 2.0

(float) 2/5 $\Rightarrow$ 0.4

Decision making statements:

* If statement $\rightarrow$

Syntax -

```
if (expression) {
    statement 1;
    statement 2;
    statement 3;
    !
    statement n;
}
```

$\rightarrow$ If given expression evaluates to TRUE(1), the if block will be executed

* If-else executed statement $\rightarrow$

Syntax -

```
if (expression)
{
    statement 1;
    statement 2;
    !
    statement n;
}
else
{
    statement 1;
    statement n;
}
```

$\rightarrow$ If block

$\rightarrow$ else block

In C $\rightarrow$ All non-zero are $\rightarrow$ TRUE

- $\rightarrow$ If expression evaluates TRUE(1), if block will be executed
- $\rightarrow$ otherwise, else block will be executed.

eg: WAP to check whether number is even or odd.

```
#include <stdio.h>
#include <conio.h>
int main ()
{
    int num;
    printf("Enter a number: ");
    scanf ("%d", &num);
    if (num % 2 == 0)
    {
        printf("Even");
    }
    else
    {
        printf("Odd");
    }
    getch();
    return 0;
}
```


Q WAP to based on condition given below

Purchase amount (Rs)	Discount (%)
Less than 5000	10%
More than or equal to 5000	15%

```
#include <stdio.h>
#include <conio.h>
int main()
{
    float amount, discount, pay-amount;
    printf("Enter the purchase price : ");
    scanf("%f", &amount);
    if (amount < 5000)
    {
        printf("Payable amount = 0.1 * amount;");
        printf("Discount amount = %f\n", discount);
        pay-amount = amount - discount;
        printf("Payable amount = %f\n", pay-amount);
    }
    else
    {
        discount = 0.15 * amount;
        printf("Discount amount = %f\n", discount);
        pay-amount = amount - discount;
        printf("Payable amount = %f\n", pay-amount);
    }
    getch();
    return 0;
}
```

③ If... else if... else statement

Syntax:

```
→ if (expn 1)
{
    // statement 1;
}
else if (expn 2)
{
    // statement 2;
}
...
else if (expn n)
{
    // statement n;
}
else
{
    // default;
}
```

Q. WAP to check whether given number is positive, negative or zero.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int num;
    printf("enter a number : ");
    scanf("%d", &num);
    if (num > 0)
    {
        printf("%d is positive", num);
    }
}
```



```

else if ( num < 0 )
{
    printf("%d is negative", num);
}
else
{
    printf("%d is zero", num);
}
getch();
return 0;
}

```

Q. WAP to find largest number among three given numbers. (using if-else if)

```

→ #include <stdio.h>
#include <conio.h>
int main()
{
    int a, b, c, largest;
    printf("enter three numbers:");
    scanf("%d %d %d", &a, &b, &c);
    if (a > b && a > c)
    {
        largest = a;
    }
    else if (b > c && b > a)
    {
        largest = b;
    }
    else
    {
        largest = c;
    }
}

```

```

printf("%d is largest", largest);
getch();
return 0;
}

```

Nested if-else

→ You can write, any number of if-else statement, within given if-else statement.

Syntax:

```

if (expression)
{
    if (cond 1)
    {
        // statement
    }
    else
    {
        // statement
    }
}
else
{
    if (cond n 2)
    {
        // statement
    }
    else
    {
        // statement
    }
}
}

```


Q. WAP to find largest no among three given numbers using nested if)

→ Syntax:

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

```
#include <conio.h>
```

```
int main()
```

```
{
```

```
int a, b, c, largest;
```

```
printf("enter three numbers:");
```

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

```
if (a > b)
```

```
{
```

```
if (a > c)
```

```
{
```

```
largest = a;
```

```
}
```

```
else
```

```
{
```

```
largest = c;
```

```
}
```

```
}
```

```
else {
```

```
if (b > c)
```

```
{
```

```
largest = b;
```

```
} else { largest = c;
```

```
printf("%d is largest", largest);
```

```
getch();
```

```
return 0;
```

```
}
```

Switch

→ It is also decision making statement

→ It is used to program menu-based application.

Syntax

```
switch (expression)
```

```
{
```

```
case c1: // do something
```

```
break;
```

```
case c2: // do something
```

```
break;
```

```
default: // do something
```

```
}
```

Q. WAP to show Thank you for 1
Welcome for 2

```
→ #include <stdio.h>
```

```
#include <conio.h>
```

```
int main()
```

```
{
```

```
int choice;
```

```
printf("PRESS 1 or 2"); // choice → integer const
```

```
scanf("%d", &choice); // or character constant
```

```
switch (choice)
```

```
{
```

```
case 1: printf("Thank you");
```

```
break;
```

```
case 2: printf("Welcome");
```

```
break;
```

```
default: printf("Invalid choice");
```

```
break;
```

```
}
```

```
getch();
```

```
return 0;
```

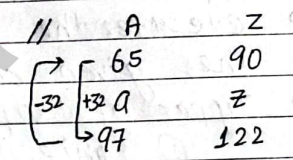
```
}
```


Q. WAP to check, whether user has entered vowel letter or consonant letter;

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char ch;
    printf("enter any small cased alphabet:");
    scanf("%c", &ch);
    switch (ch)
    {
        case 'a': printf("vowel"); break;
        case 'e': printf("vowel"); break;
        case 'i': printf("vowel"); break;
        case 'o': printf("vowel"); break;
        case 'u': printf("vowel"); break;
        default: printf("consonant"); break;
    }
    getch();
    return 0;
}
```

Q. Write a program to convert upper case to lower case and vice versa.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char ch;
    printf("enter a character:");
    scanf("%c", &ch);
    if (ch >= 65 && ch <= 90)
    {
        ch += 32; // ch = ch + 32
        printf("%c", ch);
    }
    else
    {
        ch -= 32; // ch = ch - 32
        printf("%c", ch);
    }
    getch();
    return 0;
}
```



Looping statements

→ Looping statements are used to perform repetitive task.

→ executes the statements in the loop until the given condition is false.

→ Types of loop:

(i) while loop

(ii) for loop

(iii) do-while loop

① while loop

Syntax:-

initialize the counter; // eg: i = 0

while (expression) true or false (condition)

{

statement 1;

statement 2;

statement n;

increment/decrement loop counter; // updation

}

Q. WAP to print hello world 5 times.

#include <stdio.h>

#include <conio.h>

int main()

{

int i;

i = 0

while (i < 5)

{

printf("hello world\n");

i++; // i = i + 1

}

getch();

return 0;

}

i	i < 5	O/P
0	T	hello world
1	T	hello world
2	T	hello world
3	T	hello world
4	T	hello world
5	F	comes out of loop.

Q. WAP to count number of digits in given number.

eg: 523 / 10 → 52 rem → 3
52 / 10 → 5 rem → 2
5 / 10 → 0 rem → 5

#include <stdio.h>

#include <conio.h>

int main()

{

int num = 5246, counter;

counter = 0;

while (num > 0)

{

num = num / 10;

counter++; // counter = counter + 1

}

printf("number of digits = %d", counter);

}

OR

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int num, counter;
    printf("enter the number : ");
    scanf("%d", &num);
    num = num/10;
    counter = 1;
    while (num > 0) // num != 0
    {
        counter++;
        num = num/10;
    }
    printf("NO of digits is %d", counter);
    getch();
    return 0;
}
```

Q. WAP to find sum of digits of given number

```
→ #include <stdio.h>
#include <conio.h>
int main()
{
    int num, remaindercounter, sum;
    printf("enter the number : ");
    scanf("%d", &num);
    counter = 0; sum = 0;
    while (num != 0)
    {
        remaindernum = num % 10; // 572 % 10 = 2
        sum = 0 + 2 = 2 sum = sum + remainder;
        num = num / 10; // 572 / 10 = 57
    }
}
```

; → statement terminate

```
printf("sum of digit = %d \n", sumremainder);
getch();
return 0;
}
```

ii) for loop

syntax:

for (initialize; condition check; update)

{

Statement 1;

Statement 2;

Statement 3;

}

Q. WAP to print all the numbers from 1 to 5.

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

```
#include <conio.h>
```

```
int main()
```

```
{
```

```
int i = 1;
```

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

```
{
```

```
printf("%d \n", i);
```

```
}
```

```
getch();
```

```
return 0;
```

```
}
```

	572	57	5
rem	2	7	5
S	0+2	0+7	0+5
temp	57	5	
tem	=0	✓	✓

③ do-while loop

Syntax:

```

initialize;
do
{
    statement 1;
    statement 2;
    ...
    statement n;
    update the counter;
}
while (condition);

```

eg:

```

int main()
{
    i = 0;
do while
{
    printf("Hello world");
    i++;
}
while (i < 0);

```

Output

Hello world

✗

```

int main()
{
    i = 0;
while
{
    printf("Hello world");
    i++;
}

```

O/P

Break and continue

Break statement

→ Break statement is used to abruptly terminate the execution.

eg1

```

int main()
{
    int i;
    for (i = 0; i < 5; i++)
    {
        printf("%d\n", i);
        if (i == 2)
        {
            break;
        }
    }
}

```

Output

0
1
2

④ WAP to find first three even numbers in the given range.

→ #include <stdio.h>

#include <conio.h>

```

int main()
{
    int a, b, i; count = 0;
    printf("Enter the range : ");
    scanf("%d %d", &a, &b);
    for (i = a; i <= b; i++)
    {

```

if (i % 2 == 0)

{

printf("%d\n", i);

count++;

}

if (count == 3)

{

break;

}

}

```
getch();
return 0;
```

}

Continue Statement

→ continue statements are used to skip the execution.

Q. WAP to print 1 2 4 5 6 8 9 10

```
eg: int main()
{
    int i;
    for (i = 1; i <= 10; i++)
    {
        if (i == 3 || i == 7)
        {
            continue;
        }
        else
        {
            printf("%d\n", i);
        }
    }
}
```

Q. WAP to check whether the given number is prime or not.

```
→ #include <stdio.h>
#include <conio.h>
int main() {
    int a, count = 0;
    printf("enter the number");
    scanf("%d", &a);
```

```
for (i = 1; i <= a; i++)
{
```

```
    if (a % i == 0)
    {
```

```
        count ++
```

```
    }
```

```
}
```

```
if (count == 2)
```

```
{
    printf("%d is prime number", a);
```

```
}
```

```
else
```

```
{
    printf("%d is not prime", a);
```

```
}
```

```
getch();
```

```
return 0;
```

```
}
```

Q. WAP to check whether given number is palind or not.

→ #include <stdio.h>

#include <conio.h>

```
int main()
```

```
{
    int rem, num, rev = 0;
```

```
    printf("Enter the number: ");
```

```
    scanf("%d", &num);
```

```
    int temp = num;
```

```
    while (temp != 0)
```

```
{
```

```
        rem = temp % 10;
```

```
        rev = rev * 10 + rem;
```

```
        temp = temp / 10
```

```
}
```


if (rev == num)
{
printf("%d is palindrome");
}
else
{
printf("%d is not palindrome");
}
getch();
return 0;
}

Nested loop:

Code	O/P
int i, j; for (i = 1; i < 4; i++) { for (j = 1; j < 5; j++) { printf("%d %d\n", i, j); } }	<pre> 1 1 1 2 1 3 1 4 2 1 2 2 2 3 2 4 3 1 3 2 3 3 3 4 </pre>

say num = 0
 $1 + \frac{2}{3} + \frac{3}{6} + \frac{4}{10}$ (each) =

Q WAP to check whether given number is armstrong or not.

```

#include <stdio.h>
#include <math.h>
#include <conio.h>
int main()
{
    int num, no_of_digits, temp, sum, count;
    printf("Enter the number:");
    scanf("%d", &num);
    temp = num;
    while (temp != 0)
    {
        temp = temp / 10;
        no_of_digits++;
    }
    temp = num;
    while (temp > 0)
    {
        temp = temp % 10;
        sum = sum + pow(temp, no_of_digits);
        temp = temp / 10;
    }
    if (sum == num)
    {
        printf("%d is armstrong", num);
    }
    else
    {
        printf("%d is not armstrong", num);
    }
    getch();
    return 0;
}

```


Function

→ Function is a **self-contained block of statements**, that is used to perform specific tasks.

Types of function

Build-in function

User defined function.

Syntax of function/Way of using function

#include

#include <stdio.h>

Function prototype

int display();

int main()

int main()

{

{

int a=5;

display();

Function call

return 0;

return 0;

}

}

Function definition

int display()

{

printf("%d is required number", a);

}

Built-in function / Library function.

→ Is a predefined function which are already, written, compiled & placed in C library.

→ are defined in particular header file.

→ Example: printf(), scanf(), getch(), sqrt() etc.

Way of writing main function

int main()

{

;

;

;

;

return 0;

}

void main()

{

;

;

;

;

return 0;

}

return-type

function

(args)

name

statement 1;

statement 2;

}

Q. WAP to calculate sum of two numbers.

→ #include <stdio.h>

#include <conio.h>

int sum(int a, int b);

int main()

{

int x, y, res;

printf("Enter 2 numbers:");

scanf("%d %d", &x, &y);

res = sum(x, y);

printf("sum is %d", res);

return 0;

}

int sum(int a, int b)

{

int res;

res = a + b;

return res;

}

OR

void sum()

{

int a, b, res;

printf("enter 2 numbers:");

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

res = a + b;

printf("sum = %d", res);

return 0;

}

void sum();

int main()

{

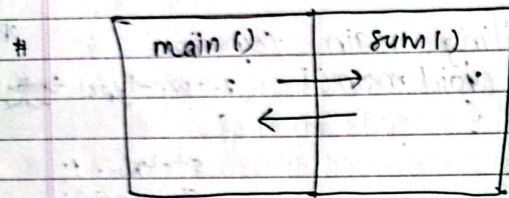
sum();

return 0;

}

Advantages of function:

- ① Write once, call many times.
- ② less code
- ③ code can be broken into smaller parts which increases readability.



calling function communicates with called function, using "arguments".

called function communicates with calling function, using "return" statements.

Q. WAP to check whether given number armstrong or unit, using function.

```

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

int digit_checker(int num);
int armstrong_checker(int no_of_digits, int num);

int main()
{
    int num, no_of_digits;

    printf("Enter the number:");
    scanf("%d", &num);
    no_of_digits = digit_checker(num);
  
```

$sum = 0$
 $sum = sum + pow(rem, no_of_digits)$

```

armstrong_checker(no_of_digits, num);
return 0;
}
  
```

```

int digit_checker(int num)
{
  
```

```

    int counter = 0;
  
```

```

    do
  
```

```

    {
  
```

```

        num = num / 10;
  
```

```

        counter++;
  
```

```

    } while (num != 0);
  
```

```

    return counter;
  
```

```

}
  
```

```

int armstrong_checker(int no_of_digits, int num)
{
  
```

```

    int sum = 0, rem, Actual_num;
  
```

```

    Actual_num = num;
  
```

```

    do
  
```

```

        rem = num % 10;
  
```

```

        sum = sum + pow(rem, no_of_digits);
  
```

```

        num = num / 10;
  
```

```

    }
  
```

```

    while (num != 0);
  
```

```

    if (Actual_num == sum)
  
```

```

    {
  
```

```

        printf("Armstrong number\n");
  
```

```

    }
  
```

```

    else
  
```

```

    {
  
```

```

        printf("Not Armstrong number\n");
  
```

```

    }
  
```

```

    return 0;
  
```


Q. WAP to swap ~~number~~ values of two given numbers.

```
#include <stdio.h>
#include <conio.h>
void swap(int num-1, int num-2);
int main()
{
    int a=10, b=20;
    swap(a, b);
    return 0;
}

void swap(int num-1, num int num-2)
{
    int temp;
    temp = a num-1;
    num-1 = num-2;
    num-2 = temp;
    printf("a = num-1 %d and b = %d",
           num-1, num-2);
}
```

Pointer

→ Pointer is a variable that holds address of another variable.

initialising int a, *b; // a is normal variable and *b is pointer variable

a = 20; value
b = &a; // b stores address of a
printf("%d", a); a 10
printf("%u", b); ↑ 120034 ← address of a
printf("%d", *b); variable name
 & → address of
 &a → address of a
 ↓
 120034

Output

10
120034
10
120034
a 10
b 120034
120034
120038
address of (a)

* → value at address

*b *(120034)
value at address 120034

eg:

```
char ch, *cptr;
ch = 'a';
cptr = &ch;
```

printf("value of ch = %c", ch); → a
printf("Address of ch = %u", &ch); → 110020
printf("Address of ch = %u", cptr); → 110020
printf("Value of ch = %c", *(&ch)); → a
printf("Value of ch = %c", *cptr); → a

ch 'a' cptr 110020
110020 180034

int a;

compiler → Reserve (int) size of memory
and reserve space in memory location
and give name 'a' to the location.

Q WAP to swap the numbers (call by reference)

→ #include <stdio.h>

#include <conio.h>

void swap(int *a, int *b);

int main()

{

int x = 10, y = 20;

swap(&x, &y);

printf("After swap x = %d, y = %d\n", x, y);

return 0;

}

void swap(int *a, int *b);

{

int temp;

temp = *a;

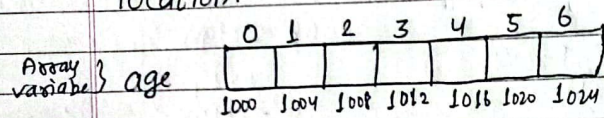
*a = *b;

*b = temp;

}

Array

- Array is collection of elements of similar data types.
- Array stores values at contiguous memory location.



int age[7];

7 memory location of size int is reserved
Total memory allocated = $7 * \text{sizeof(int)} = 7 * 4 = 28$ bytes

- Index of array starts from zero (0) to size of array - 1.
- Array is collection of homogenous data-types.

Syntax:

⇒ data-type name-of-array [size];

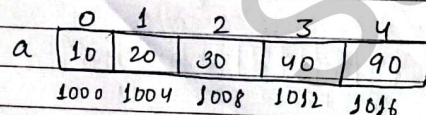
eg:

double price[10];

↑
size

Reserve $10 * \text{sizeof(double)}$ byte of memory location & call location as price.

eg: int a[5]; ← initialization



a[0] = 10;
a[1] = 20;
a[2] = 30;
a[3] = 40;
a[4] = 90;

← array declaration

Array declaration: (Alternative way)

char ch[] = {'a', 'e', 'i', 'o', 'u'};

eg: int a[100], n, i; } Taking size of array from user

```
printf("number of element: ");
scanf("%d", &n);
printf("Enter %d elements", n);
for (i = 0; i < n; i++)
{
    scanf("%d", &a[i]);
}
```

- First address of array i.e. address of 1st element of array is also called base address.

&a[0] ≡ a

↓ ↓

1000 1000

Ⓐ = ○

int a[n] { C programming doesn't allow dynamic array declaration }

eg: `int a[20], *ptr;`
`ptr = a;`
`*ptr = 10;`
`ptr++;`
`*ptr = 20;`
`ptr++;`
`*ptr = 30;`

ptr → stores address of first element i.e. `a[0]`.

Diagram: `a` is an array of 20 elements. The first three elements are 10, 20, 30. `ptr` points to the first element (10).

Q. WAP to add all elements of array.

```
#include <stdio.h>
int main()
{
    int a[100], n, i, sum = 0;
    printf("Enter size of array: ");
    scanf("%d", &n);
    printf("Enter %d elements", n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    for (i = 0; i < n; i++)
    {
        sum = sum + a[i];
    }
    printf("sum = %d", sum);
    return 0;
}
```

Diagram: `n = 4`, `sum = 79`. Array `a` contains [10, -1, 20, 50].

Calculation:
`i = 0` | `1000 + 0 = 1000` ⇒ `* (1000) = 10`
`i = 1` | `1000 + 1 * 2 byte = 1002` ⇒ `* (1002) = -1`
`i = 2` | `1000 + 2 * 2 byte = 1004` ⇒ `* (1004) = 20`

Q. Using concept of address/pointer.

```
#include <stdio.h>
int main()
{
    int a[100], n, i, sum = 0;
    printf("Enter size of array: ");
    scanf("%d", &n);
    printf("Enter %d elements", n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    for (i = 0; i < n; i++)
    {
        sum = sum + (*a + i);
    }
    printf("sum = %d", sum);
    return 0;
}
```

Q. WAP to add all elements of array, passing array to function.

```
#include <stdio.h>
void calculate-sum (int a[], int n);
```

```
int main()
{
    int a[100], n, i, sum = 0;
    printf("Enter size of array: ");
    scanf("%d", &n);
```


$\begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix}$ $a[1] < a[2]$

```

1 // point f ("Enter x-d elements", n);
2 for (i=0; i<n; i++)
3     scanf("%d", &a[i]);
4 calculate-sum(a, n);
5 return 0;
6
7 void calculate-sum (int a[], int size)
8 {
9     int i, sum=0;
10    for (i=0; i<size; i++)
11        sum = sum + a[i];
12    printf("sum = %d", sum);
13 }
  
```

Q. WAP to sort the elements in ascending order.

```

1 #include <stdio.h>
2 void sort (int a[], int size);
3 int main()
4 {
5     int a[100], n, i;
6     printf("Enter size of array:");
7     scanf("%d", &n);
8     printf("Enter %d elements :", n);
9     for (i=0; i<n; i++)
10         scanf("%d", &a[i]);
11     sort(a, n);
  
```

```

1 printf("sorted array are:");
2 for (i=0; i<n; i++)
3     printf("%d \t", a[i]);
4 printf("\n");
5 void sort (int a[], int size)
6 {
7     int temp, i, j;
8     for (i=0; i<size; i++)
9     {
10        for (j=i+1; j<size; j++)
11        {
12            if (a[i] > a[j])
13            {
14                temp = a[i];
15                a[i] = a[j];
16                a[j] = temp;
17            }
18        }
19    }
  
```


2D-array

→ 2 Dimensional array is also called as matrix.

Syntax:

datatype variable [row_size][column_size];

Example;

Create a Integer matrix of size: 2x3

```
int mat[2][3];
```

	0	1	2
0			
1			

mat[0][0]	mat[0][1]	mat[0][2]
mat[1][0]	mat[1][1]	mat[1][2]

Q. WAP to add 2 arrays & store in 3rd array.

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

```
int main()
```

```
{
```

```
int a[20][20], b[20][20], c[20][20], i, j, r, col;
```

```
printf("Enter size of matrix");
```

```
scanf("%d %d", &r, &col);
```

```
printf("Enter %d elements of first matrix", r*col);
```

```
for (i=0; i<r; i++)
```

```
{
```

```
for (j=0; j<col; j++)
```

```
{
```

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

```
}
```

```
printf("Enter %d elements of second matrix", r*col);
```

```
for (i=0; i<r; i++)
```

```
{
```

```
for (j=0; j<col; j++)
```

```
{
```

```
scanf("%d", &b[i][j]);
```

```
}
```

```
}
```

```
for (i=0; i<r; i++)
```

```
{
```

```
for (j=0; j<col; j++)
```

```
{
```

```
c[i][j] = a[i][j] + b[i][j];
```

```
}
```

```
}
```

```
printf("resultant matrix\n");
```

```
for (i=0; i<r; i++)
```

```
{
```

```
for (j=0; j<col; j++)
```

```
{
```

```
printf("%d\t", c[i][j]);
```

```
}
```

```
}
```

```
}
```

```
}
```

```
return 0;
```

```
}
```


String

→ String constant is character array, terminated by NULL character ('0').

```
char ch[] = {'a', 'b', 'c', 'd'};
```

0	1	2	3
'a'	'b'	'c'	'd'

```
char ch[] = {'a', 'b', 'c', 'd', '\0'};
```

Print hello string

Syntax:

```
char msg[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

OR

```
char msg[] = "hello" // In this case,
```

adds a null

character automatically

" " → String
" " → character

%s → format specifier for string

Q. WAP to read & display name

→ #include <stdio.h>

#include <conio.h>

int main()

{ char name[100];

printf("Enter your name :");

scanf("%s", name);

printf("My name is: %s", name);

return 0;

}

OR

int main()

{

char name[100];

printf("Enter your name :");

gets(name);

printf("My name is : ");

puts(name);

}

OR

int main()

{

char name[100];

printf("Enter your name :");

scanf("%[^\n]", name);

printf("Name is :");

puts(name);

}

Q. WAP to find length of given string.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int len = 0, i;
    char name[] = "thank you";
    i = 0;
    while (name[i] != '\0')
    {
        len++;
        i++;
    }
    printf("Length of %s is: %d", name, len);
    return 0;
}
```

Index → 0 1 2 3 4 5 6 7 8 9
 string → 't' 'h' 'a' 'n' 'k' ' ' 'y' 'o' 'u' '\0'
 address → 10 11 12 13 14 15 16 17 18 19

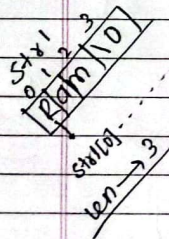
OR Using pointers,

```
int main()
{
    int len = 0;
    char name[] = "thank you";
    char *ptr;
    ptr = name; // ptr = &name[0]
    while (*ptr != '\0')
    {
        len++;
        ptr++;
    }
}
```

printf("length of %s is: %d\n", name,

len);

}



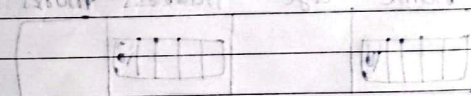
Q. WAP to copy given string.

```
#include <stdio.h>
int main()
{
    char src[] = "abc";
    char dest[100];
    char *p, *q;
    p = src; // p = &src[0] ← base address
    q = dest; // q = &dest[0] ← base address
    while (*p != '\0')
    {
        *q = *p;
        q++;
        p++;
    }
}
```

*q = 10; // * (next contiguous address after last character of string) = 10.
 ↓
 To make string

```
printf("%s", src);
printf("%s", dest);
return 0;
```

}



- Q. Diff. build in function for all the available built-in-function (string).

Structure in C-Programming

- Structure is used to group values of different data-types.
- By using structure we can create user defined datatype.

Syntax:

```
struct structure-name
```

```
{
```

```
    datatype var1;
```

```
    datatype var2;
```

```
    datatype varn;
```

```
}
```

eg: struct student

```
{
```

```
    char name[10];
```

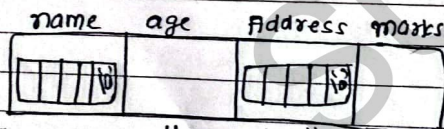
```
    int age;
```

```
    char address[20];
```

```
    float marks;
```

```
}
```

eg: struct student s;



```
s.name = "Suwendra";
```

```
s.age = 19;
```

```
s.address = "Jhapa";
```

```
s.marks = 85.56;
```

OR

```
struct student P = {"Ram", 10, "Kathmandu", 92.5};
```

- Q. WAP to create Employee, structure and display his/her info.

```
→ #include <stdio.h>
```

```
struct Employee
```

```
{
```

```
    char name[100];
```

```
    int age;
```

```
    float salary;
```

```
};
```

```
int main()
```

```
{
```

```
    struct employee e1;
```

```
    printf("Enter information of an employee");
```

```
    scanf("%s%d%f", &e1.name, &e1.age, &e1.salary);
```

OR e1.name

OR e1.salary

```
    printf("Following are the details");
```

```
    printf("\n Name = %s", e1.name);
```

```
    printf("\n Age = %d", e1.age);
```

```
    printf("\n Salary = %f", e1.salary);
```

```
    return 0;
```

```
}
```


Q. WAP to pass structure into function:

→ #include <stdio.h>

void display(struct employee q);

struct employee

{
char name[100];

int age;

float salary;

}

int main()

{

struct employee e1;

printf("Enter information of employee:");

scanf("%s %d %f", &e1.name, &e1.age,
&e1.salary);

display(e1);

return 0;

}

void display(struct Employee q)

{

printf("Following are the details");

printf("\n name = %s", q.name);

printf("\n Age = %d", q.age);

printf("\n salary = %f", q.salary);

}

Q. How to use pointer in structure:

→ struct employee e, *ptr;

ptr = &e;

ptr -> age;

ptr -> name;

ptr -> salary;

→ #include <stdio.h>

~~struct employee~~

void display(struct employee *q)

struct employee

{

char name[100];

int age;

float salary;

}

int main()

{

struct Employee e1;

printf("Enter information of an employee");

scanf("%s %d %f", &e1.name, &e1.age,
&e1.salary);

display(&e1);

return 0;

}

void display(struct Employee *q)

{

printf("Following are the details");

printf("\n name = %s", q -> name);

printf("\n Age = %d", q -> age);

printf("\n salary = %f", q -> salary);

}

Assignment:

Q. WAP to store, following information of given number of students in a class:-

name $\rightarrow$ string
Age $\rightarrow$ int
Marks obtained in 5 different subjects
Assume: Total marks is 500

- Write a function to calculate the percentage score of each student.
- Display the information on the basis of obtained percentage score (i.e. Rank wise).

Two Dimensional Array

- Syntax: `datatype var_name [row][column];`
total data $\Rightarrow$ row $\times$ column
- 2-D array is also called collection of 1-D array.

eg: `int a[2][3];`

		0	1	2
	i			
a	j	$a[0][0]$	$a[0][1]$	$a[0][2]$
		$a[1][0]$	$a[1][1]$	$a[1][2]$

In memory,

	$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[1][0]$	$a[1][1]$	$a[1][2]$
a	10	20	30	40	50	60
	1000	1002	1004	1006	1008	1010

$\&a[0][0]$	$\&a[1][0]$
$\downarrow$	$\downarrow$
1000	1006

For 1-D

`int a[] = {10, 20, 30};`

`int a`

For 2-D

`int a[][3] = {10, 20, 30, 40, 50, 60};`

OR

`int a[][3] = { {10, 20, 30},
 {40, 50, 60} };`

Q. WAP to read and display the matrix.

```
#include <stdio.h>
#include <conio.h>
void read(int a[][10], int m, int n);
void display(int a[][10], int m, int n);

int main()
{
    int a[10][10], m, n;
    printf("Enter size of matrix : ");
    scanf("%d %d", &m, &n);
    read(a, m, n);
    display(a, m, n);
    return 0;
}

void read(int a[][10], int m, int n)
{
    int i, j;
    printf("enter %d elements", m*n);
    for (i=0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
}

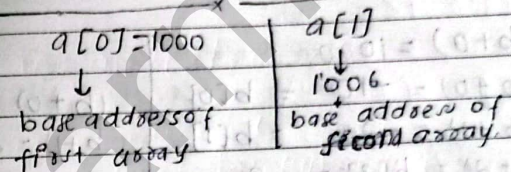
void display(int a[][10], int m, int n)
{
    int i, j;
    for (i=0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
            printf("%d\t", a[i][j]);
        }
        printf("\n");
    }
}
```

Two dimensional array and pointer

```
int a[3][3] = {10, 20, 30, 40, 50, 60};
```

@

a	10	20	30	40	50	60
	1000	1002	1004	1006	1008	1010



$a[0] \equiv \&a[0][0] \rightarrow$ base address of 1st array

$a[0] + 0 = 1000$	$a[1] = 1006$
$a[0] + 1 = 1002$	$a[1] + 0 = 1006$
$a[0] + 2 = 1004$	$a[1] + 1 = 1006 + 1 * \text{sizeof(int)}$
	$= 1008$
	$a[1] + 2 = 1010$

$a \rightarrow$ gives address of base address
(first element of array)

$a[i][j] = (i * \text{sizeof(int)} + j * \text{sizeof(int)})$
 $a[i][j] = (i * 4 + j * 4)$

For 1-D

int b[3];

	0	1	2
b	-10	-20	-30
	1000	1002	1004

$(b+0) = 1000$		$(b+0) = \&b[0]$
$*(b+0) = -10 = b[0]$		$(b+1) = \&b[1]$
$*(b+1) = -20 = b[1]$		$(b+2) = \&b[2]$
$*(b+2) = -30 = b[2]$		

For 2-D array

int a[2][3] = {10, 20, 30, 40, 50, 60}

	0,0	0,1	0,2	1,0	1,1	1,2
a	10	20	30	40	50	60
	1000	1002	1004	1006	1008	1010
	a[0]			a[1]		

$a[0] + 0 = 1000$	$a[i] + j \rightarrow \text{address}$
$a[0] + 1 = 1002$	$*(a+i) + j \rightarrow \text{address}$

$*(a[i] + j) \rightarrow \text{value}$
$*(a[i] + j) \rightarrow \text{value}$

Pointer to 2-D array

Q. syntax for declaration of 2-D array using pointer notation:

data-type (*ptr-variable)[size2];

instead of data-type array[size1][size2]; ← normal array declaration.

Eg: int (*p)[3];

Q. WAP to read & display matrix by using pointer & function.

→ #include <stdio.h>
#include <conio.h>

int main()

```

{
    int a[10][10], m, n;
    printf("Enter size of matrix");
    scanf("%d %d", &m, &n);
    read(a, m, n);
    display(a, m, n);
    return 0;
}

```

```

void read(int (*a)[10], int m, int n)
{
    int i, j;
    printf("Enter %d elements", m*n);
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        {
            scanf("%d", (*(a+i)+j));
        }
    }
}

```



```
void display(int (*a)[10], int m, int n)
{
```

```
    int i, j;
```

```
    for (i=0; i<m; i++)
```

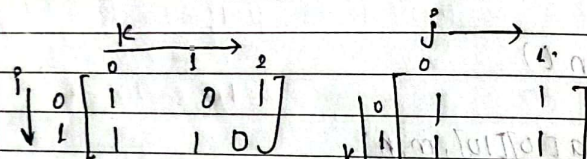
```
    {
        for (j=0; j<n; j++)
```

```
        {
            printf("%d\t", (*a[i]+j));
```

```
        }
```

```
    }
    printf("\n");
}
```

Matrix Multiplication



m x n

p x q

n=p

i=0

i++

j=0

k=0

i → row of 1st
k → column of 1st / row of 2nd
j → column of 2nd
inc. j for continue loop
k++ After k=2

sum=0

sum = sum + a[i][k] * b[k][j];

sum

Code

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

```
int main()
```

```
{
```

```
    int a[10][10], b[10][10], c[10][10]; m, p, n, q;
```

```
    printf("Enter size of first matrix");
```

```
    scanf("%d %d", &m, &n);
```

```
    printf("Enter size of second matrix");
```

```
    scanf("%d %d", &p, &q);
```

```
    if (n != p)
```

```
    {
```

```
        printf("Not possible");
```

```
    }
```

```
    else
```

```
    {
```

```
        read(a, m, n);
```

```
        read(b, p, q);
```

```
        multiply(a, b, c, m, n, p, q);
```

```
        display(c, m, q);
```

```
    }
```

```
    return 0;
```

```
}
```

```
void read(int (*a)[10], int m, int n)
```

```
{
```

```
    int i, j;
```

```
    printf("Enter %d elements", m*n);
```

```
    for (i=0; i<m; i++)
```

```
    {
```

```
        for (j=0; j<n; j++)
```

```
        {
            scanf("%d", &a[i][j]);
```

```
        }
```

```
}
```



```
void multiply(int a[][10], int b[][10],
             cint[][10], int m, int n, int p, int q)
```

```
{
    int j, k, sum = 0;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < q; j++)
        {
            for (k = 0; k < n; k++)
            {
                sum = sum + a[i][k] * b[k][j];
            }
            c[i][j] = sum;
            sum = 0;
        }
    }
}
```

Matrix Multiplication using pointers

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

```
#include <conio.h>
```

```
void read(int (*a)[10], int m, int n);
```

```
void multiply(int (*a)[10], int (*b)[10], int (*mul)[10], int m,
             int n, int q);
```

```
void display(int (*mul)[10], int m, int q);
```

```
int main()
```

```
{
```

```
    int a[10][10], b[10][10], mul[10][10], m, n, p, q;
```

```
    printf("Enter order of first matrix m*n:");
```

```
    scanf("%d %d", &m, &n);
```

```
    printf("Enter order of second matrix p*q:");
```

```
    scanf("%d %d", &p, &q);
```

```
    if (n != p)
```

```
    {
```

```
        printf("Matrix multiplication not possible\n");
```

```
    }
```

```
    else
```

```
    {
```

```
        printf("Enter %d elements of first matrix:\n", m*n);
```

```
        read(a, m, n);
```

```
        printf("Enter %d elements of second matrix:\n", p*q);
```

```
        read(b, p, q);
```

```
        multiply(a, b, mul, m, n, q);
```

```
        printf("The matrix after multiplication is:\n");
```

```
        display(mul, m, q);
```

```
    }
```

```
    getch();
```

```
    return 0;
```

```
}
```



```

void read (int (*a)[10], int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            printf("Enter element at index [%d]: ", i+j);
            scanf("%d", *(a+i)+j);
}

```

```

void multiply (int (*a)[10], int (*b)[10], int m, int n, int q)
{
    int i, j, k;
    for (i = 0; i < m; i++)
        for (k = 0; k < q; k++)
        {
            *(*(mul+i)+k) = 0;
            for (j = 0; j < n; j++)
                *(*(mul+i)+k) += *(*(a+i)+j) * *(*(b+j)+k);
        }
}

```

```

void display (int (*mul)[10], int m, int q)
{
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < q; j++)
            printf("%d\t", *(*(mul+i)+j));
        printf("\n");
}

```

File handling in C

To open file:

```

FILE *fptr-variable;
fptr-variable = fopen(<path>, <mode>);

```

The RAM is volatile and its content is lost once the program terminates. In order to persist the data forever we use files.

Mode:

"r" → read mode (file should exist)

"w" → write mode (creates new file if it does not exist)

"a" → append mode (data is added to end of file)

Q. WAP to read content of a file

Hello! How
are you?

check.txt

→ #include <stdio.h>

#include <stdlib.h>

int main ()

{ char ch;

FILE *fp;

fp = fopen("check.txt", "r");

EOF → terminated

```
if (fp == NULL)
```

```
{
```

```
exit(0);
```

```
}
```

```
else
```

```
{
```

```
while (1)
```

```
{
```

```
ch = fgetc(fp);
```

```
if (ch == EOF)
```

```
{
```

```
break;
```

```
}
```

```
else
```

```
{
```

```
printf("%c", ch);
```

```
}
```

```
}
```

```
fclose(fp);
```

```
}
```

Q. WAP to copy content of a file to another file.

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

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

```
int main()
```

```
{
```

```
char ch;
```

```
FILE *fp, *dp;
```

```
fp = fopen("check.txt", "r");
```

```
dp = fopen("abc.txt", "w");
```

```
if (fp == NULL || dp == NULL)
```

```
{
```

```
exit(0);
```

```
}
```

```
else
```

```
{
```

```
while (1)
```

```
{
```

```
ch = fgetc(fp);
```

```
if (ch == EOF)
```

```
{
```

```
break;
```

```
}
```

```
else
```

```
{
```

```
fputc(ch, dp);
```

```
}
```

```
}
```

```
fclose(fp);
```

```
fclose(dp);
```

```
}
```

abc.txt

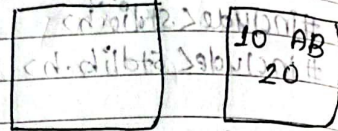
check.txt

Hello! How are you?

Q.

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

```
FILE *fp, *dp;
int a, b;
char str[100];
fp = fopen("src.txt", "r");
dp = fopen("dest.txt", "w");
if (fp == NULL || dp == NULL)
{
    exit(0);
}
else
{
    fscanf(fp, "%d %d", &a, &b);
    fprintf(dp, "%d %d", a, b);
}
```



Assessment Qn.

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

```
struct student
{
    char name[100];
    float marks;
};
```

```
int main()
{
    float max = 0;
    FILE *fp;
    int i;
    struct student s[5];
    fp = fopen("record.txt", "w+");
```

```
for(i = 0; i < 5; i++)
```

```
{
    printf("Enter name & marks for student %d", i+1);
```

```
scanf("%s %f", s[i].name, &s[i].marks);
```

```
}
```

```
for(i = 0; i < 5; i++)
```

```
{
    fprintf(fp, "%s\t%f\n", s[i].name, s[i].marks);
}
```



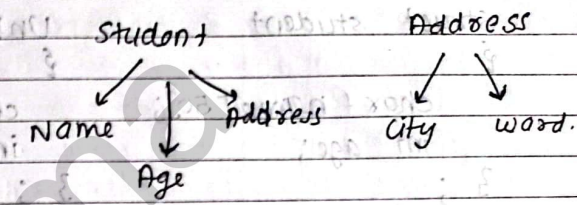
```

rewind(fp);
fscanf(fp, "%s%f", q.name, &q.marks);
max-m = q.marks;
line = 1;
while (fscanf(fp, "%s%f", q.name, &q.marks)) {
    line++;
    if (q.marks > max-m)
        max-m = q.marks;
}
line = 1;
while (fscanf(fp, "%s%f", q.name, &q.marks)) {
    if (line == 1-max)
        printf("Detail of Topper: ");
        printf("Name = %s Marks = %f\n", q.name, q.marks);
    line++;
}

```

return 0;

Nested structure



```

struct Address
{
    char city[100];
    int ward;
};

struct student
{
    char name;
    int Age;
    struct address addr;
};

struct student s;
s.name = "abc";
s.age = 10;
s.addr.city = "Pokhara";
s.addr.ward = 10;

```


Structure
 struct student
 {
 char name[50];
 int age;
 };

Union
 Union student
 {
 char name[50];
 int age;
 };

50x1 byte + 2 byte
 = 52 byte

50x1 byte space.

FORTAN

- FORMula TRANslation
- compiler based language.

Structure of Fortran

comment	label	Extend	
1	2-5	6	7-72
c			73-80
c			fortran code to add two numbers: PROGRAM ADD integer a b, c write(*,*) "Enter two numbers" read(*,*) a, b c = a + b write(*,*) "sum = ", c end stop

datatypes
 Integer,
 real,
 complex,
 character

write operation: printf() ≈ write(*,*)
 read operation: scanf() ≈ read(*,*)

1st * → standard input/output
 2nd * → format specifier.

** → Power

2**8 ⇒ 8

Relation Operator

C	Fortran
<	.lt.
>	.gt.
<=	.le.
>=	.ge.
=	.eq.
≠	.ne.

Logical Operator

C	Fortran
&&	.and.
	.or.
!	.not.

Boolean:

- TRUE
- FALSE

Q. WAP to check whether number is even or odd.

1	2-5	6	7-72	73-80
C			check whether number	
C			is even or odd	
			PROGRAM EVENODD	
			integer num	
			write(*,*) "Enter a number"	
			read(*,*) num	
			if (mod(num, 2) .eq. 0) then	
			goto 100	
			else	
			goto 200	
			endif	
	100		write(*,*) "Even"	
			stop	
	200		write(*,*) "Odd"	
			stop	
			end	

Finding area of circle:

```

Program areaofcircle
real r, pi, area
pi = 3.1415
write(*,*) "Enter radiw Of circle!"
read(*,*) r
area = pi*(r**2)
write(*, 400) "Area = ", area.
400 format (A4, F10.2)
endProgram areaofcircle.

```

should be done at beginning

Finding sum of two numbers

```

Program add
integer a, b, sum
write(*,*) "Enter two numbers!"
read(*,*) a, b
sum = a + b
write(*,*) "sum = ", sum
endProgram add

```

stop & break.

FORTRAN

- ↳ acronym for FORMula TRANslation
- ↳ first high level programming language
- ↳ used for mathematical, and scientific application.

↳ Fortran 77 : most popular version developed in IBM 90, 95, 66, II, IV

Characteristics

- ↳ numerical capacity & I/O facilities are more powerful than others.
- ✓ ↳ dominating language for mathematical computing.
- ✓ ↳ simple to understand
- ✓ ↳ easy to learn
- ✓ ↳ FORTRAN program more portable than others
- ✓ ↳ Machine independent (easy to translate program from one machine to another)
- ✓ ↳ standardized by ANSI and ISO } standard organizations.
- ✓ ↳ rich built in library functions.

Limitations

- ↳ struct & class not available in F77
- ↳ NO Dynamic memory Allocation DMAPX
- ↳ code: difficult to extend, maintain & reuse
- ↳ poor string handling capabilities
- ↳ loop controls are limited. Use of goto statements is high.
- ↳ not a free-format language : strict set of rules for writing source code.

column position rule
// Program to display "Hello World"

1	2-5	6	7-72	73-80
c			program Hello World	
			write (*,*) 'Hello World'	
			stop	
			end	

Note: we can't start writing instructions from very beginning of line

1: comment {for comment c or * else leave blank}

2-5: label {write some text to denote or label the particular statement}

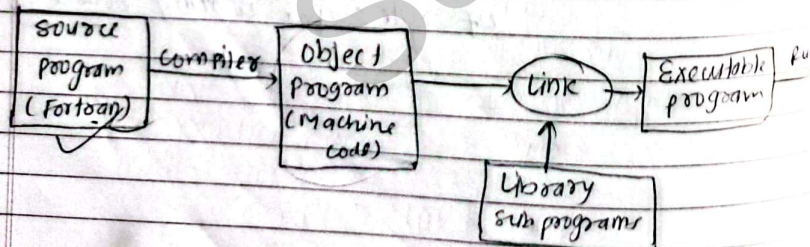
6: continuation of previous line {if statement doesn't fit into 66 available columns i.e. (7-72) columns}

{ '+' sign used for continuation but any character can be used as continuation character }

7-72: statements & actual instructions or statements for execution

73-80: sequence numbers (rarely used)

Steps for writing and executing of FORTRAN



FORTRAN compiler: converts source code to object code

After compilation,

Linker: takes the set of object modules produced by compiler and links them all together into executable image

After compilation & linking;

executable program are generated and loaded into memory to run to produce corresponding output.

Structure of FORTRAN program

↳ consists main program and subprogram

Structure of main program

✓ (A) Program name : begins with keyword program which helps to assign name to program

✓ (B) Declaration statement
↳ declare variables & parameters required in program (define)

(C) Statements : actual statements

(D) Stop : (optional) but recommended to terminate program with stop statement to emphasize flow stop here

(E) End : inform compiler about ending of program

Pn (P)

DS →

St →

SOP)

E.

#	Character set
---	---------------

4 set of characters used to form words, numbers and expressions in FORTRAN

4) combination of these character form
FORTRAN program

4 con sirt3

(i) Letters

26 upper case letter : A-Z 2 Alphabetic

26 lowercase letters: a-z } characters

(ii) Digital

numbers : 0-9

slash left parentheses

(iii) special characters

' - blank, '=' , '+' , '-' , '*' , '/' , '(' , ')' , ','

'\$', '()', ':', '!', '-', '_', ']'

~~apoptosis~~

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----

under_score

double quote

↓
column

(iv) escape sequence

4 non-printing characters used in FORTRAN

↳ (\) backslash followed by letter or digit

Escape sequence	Meaning
\n	Newline
\t	Tab
\b	Backspace
\f	Form feed
\o	Null
\'	Single quote
\"	Double quote
\\	Slash.

Data types, variables and constants

(1) Data types

4 types of data in programming environment

• Integer : +ve & -ve integral (non-fractional) number

↳ occupy 4 bytes in memory in RAM

Stores integers from -2^{31} to 2^{31}

- Real : +ve & -ve number with fractional part

↳ 4 bytes

$$4.2 \times 10^{-38} \text{ to } 3.4 \times 10^{38}$$

- Double Precision: $\rightarrow$ Deal with twice storage space & greater precision

↳ 8 bytes

$$L, 2.2 \times 10^{-308} \text{ to } 1.8 \times 10^{308}$$

- complex: ordered pairs of REAL data:

real & imaginary components

↳ $2 \times 4 = 8$ bytes

- Logical: BOOLEAN data representing TRUE or FALSE

FALSE (ie. either 0 or 1)

4. 1 bit

- Character: one alphanumeric character

↳ one byte

↳ character * N $\Rightarrow$ multiple character

Working with Nchoso

1 N byte

~~Large~~ memory in RAM

Note: Number of bytes occupied machine dependent

Q. Program to read two complex number and display their result

→ c program variables
complex a, b, c
write(*,*) 'Enter two complex numbers:'
read(*,*) a, b
c = a + b
write(*,*) 'The sum of two complex is:', c
STOP
END

⇒ Enter two complex numbers:

(1,3)

(2,5)

The sum of two complex is (3.0000,8.0000)

Q. read string and display on screen

c program variables
character (LEN = 20) name
write(*,*) 'Enter your name:'
read(*,*) name
write(*,*) 'Your name is', name
STOP
END

⇒ Enter your name : Surender
Your name is Surender

Note: space or $\downarrow$ terminates.
For reading name with space modify
read statement as
read(*, '(A)') name.

(2) Variables

↳ Identifier with a name, data type and value
↳ type depends upon either data type
provided at time of its declaration

(i.e. explicit definition) or first character
in its name (i.e. in case of implicit definition)

↳ symbolic name of data item

↳ variable cannot be used or referred unless
it has been defined ✓

• Implicit definition

↳ can define variable without explicit data
in variable declaration statement

↳ if not explicitly specified, data type of a
item is determined by first character of
name

↳ By default

symbolic name beginning with I, J, K, L, I
(uppercase or lowercase) ⇒ integer data type

all other letters ⇒ real data type

• Explicit definition

defining variable using following syntax

type - name variable - name

eg: integer a, b, c

(3) Constants

↳ value doesn't change during program
execution

(i) Integer constant: + or - sign followed by or

eg: -420

for positive number: + is optional

(ii) Real constants

↳ contains decimal point or an exponent (or both)

↳ In FORTRAN: "E" ⇒ "times 10 to the power of"

eg: 2.5, .0001,

+13.456E4
↳ +13.456 × 10⁴

(iii) Complex constants

↳ two real or integer constants separated by comma & enclosed in a pair of parentheses

↳ first number: real component

second number: imaginary component

eg: (2.14, -3.67)

(+1E2, 0.105)

(iv) Logical constants

↳ two possible logical constants

↳ expressed as .TRUE. and .FALSE.

↳ data at each end to distinguish the special forms from words TRUE and FALSE

Symbolic name

(v) Character constants

↳ consist of string of character enclosed in a pair of apostrophes which act as quotation marks

↳ within quotation marks, any set of characters available in character set are permitted

eg: 'A', 'Rom', 'I am here'

Operators and expressions

• operator

↳ symbol that operates on a certain data type or data item

↳ used to perform mathematical and logical manipulations

• operands

↳ data items that operators act upon



operator

• expression

↳ combination of variable, constants and operation written acc. to syntax of the language

eg:

8+9 a+b*c a>b

Types of operators

(1) Arithmetic operator

(2) Relational

(3) Logical

(4) Assignment

(5) Concatenation

A ✓
R ✓
L ✓
A ✓
C ✓

1) Arithmetic operator

- ↳ used to evaluate numeric expression
- ↳ evaluation produces numeric value
- Arithmetic $\Rightarrow$ formed by using one or more arithmetic expressions operators together with arithmetic operators and parentheses

Arithmetic operands: must identify values of type integer, real, double precision or complex

Associativity	Precedence	Operation	Symbol	FORTRAN Expr.	Arithmetic Expr.
right to left	highest	Exponentiation	$**$	$A**B$	a^b
left to right	high	Multiplication	$*$	$A*B$	$a \times b$
		Division	$/$	A/B	$a \div b$
left to right	low	Addition	$+$	$A+B$	$a+b$
		Subtraction	$-$	$A-B$	$a-b$
		Unary minus	$-$	$-A$	$-a$

Operator precedence and Associativity

- ↳ precedence: used to determine how an expression involving more than one operator is evaluated.

↳ distinct levels of precedence

↳ operator at higher level of precedence

↳ evaluated first

operator of same level of precedence: evaluated from 'left to right' or 'right to left' (depending on the level)

associativity property of an operator

Q. Write output of program

Program precedence

real x, y, z , result

$x = 10$

$y = 6.5$

$z = 15.4$

result = $y + z * x ** 2$

write (*, 900) 'The value of result = ', result

900 format (A20, F10.2)

Stop

End

$\rightarrow$ The value of result is 1546.50

Step 1: $x ** 2 = 10 ** 2 = 10^2 = 100$

Step 2: $z * x ** 2 = 15.4 * 100 = 1540$

Step 3: $y + z * x ** 2 = 6.5 + 1540 = 1546.50$

Division rule

Result of division operator (i.e. $/$) depends upon the types of operands.

Both integer operands $\rightarrow$ integer division

Else $\rightarrow$ real arithmetic division

i.e. Integer/Integer $\rightarrow$ Integer

real/real

Integer/real

real/integer

eg: ~~10/4~~ integer a, b, c

real c

$a = 10$

$b = 4$

$c = a/b$

write (*, *) c

2.000

Integer $\frac{10}{4}$
not 2.5
 $c \rightarrow$ real
so 2.0000

- (3) Relational operator
 ↳ used to compare two similar operands & depending upon relations take some actions.
 ↳ compare left hand side operand with right hand side operand.
 ↳ gives either 1 (if condition is true) or 0 (if condition is false)

FORTRAN operator	Common operator	Meaning	Example	Output (a=20, b=6)
• LT.	<	less than	a.LT.b	0
• GT.	>	greater than	a.GT.b	1
• LE.	<=	less than or equal to	a.LE.b	0
• GE.	>=	greater than or equal to	a.GE.b	1
• EQ.	=	equal to	a.EQ.b	0
• NE.	!=	Not equal to	a.NE.b	1

- (4) Logical operator
 ↳ used to compare or evaluate logical and relational expressions.
 ↳ operands must produce either 1 (true) or 0 (false).
 ↳ whole result: 0 or 1
 ↳ • AND, • OR, • NOT.

A	B	• NOT A	A • AND • B	A • OR • B
• TRUE	• TRUE	• FALSE	• TRUE	• TRUE
• TRUE	• FALSE	• FALSE	• FALSE	• TRUE
• FALSE	• TRUE	• TRUE	• FALSE	• TRUE
• FALSE	• FALSE	• TRUE	• FALSE	• FALSE

- (4) String concatenation operator
 ↳ Character strings can be joined together in a single
 ↳ represented by double forward slash //
 ↳ also known as character operator

eg:
 (Q) Read first name & family name from user and concatenate two strings to display full name.
 c program string concatenation
 character from 10, lname * 10, name * 20
 write(*,*) 'What is your first name?'
 read(*,*) fname
 write(*,*) 'What is your last name?'
 read(*,*) lname
 name = fname // ' ' // lname
 write(*,*) 'Your full name is', name
 stop
 end

- (5) Assignment operators
 ↳ Assign result of expression to a variable
 ↳ assignment operator → '='
 ↳ precedence → least
 ↳ Associativity → ~~left to right~~ right to left

Library Functions

Function → Two types → User-defined

↳ Library (built-in function)

can be directly called in program to make program development easy.

$y = x + x^2 + x^3 + \dots$
 mat(1), mat(2), mat(n)
 mat n=1

Library functions in FORTRAN:

Abs()
 Min()
 Max()
 Sqrt()
 Sin()
 Cos()
 Tan()
 Atan()
 Exp()
 Log()
 Read()
 Write()
 Mod()

read(UNIT=unit no, FMT=format label for)

Formatted and Unformatted I/O

I/O Function
 Used to handle input and output activities

Two functions for I/O:

read() and write()

read() → used for transferring data from input device to main memory of computer

write() → used for transferring data from main memory of computer (i.e. program) to output device

Syntax:

read(UNIT=unit no, FMT=format no) list of variables separated by comma

write(UNIT=unit no, FMT=format no) list of variables separated by comma

1 2 4 3 6 7
 12 3 5 6 7 8

Format Statement

Label FORMAT(Fmt sp1, Fmt sp2, ...)
 Integer rem. width
 beg. space

I - Integer

A - text string

D - Double precision, exp.

E - real no. in exp.

F - real, fix

x - hor. skip(space) skip some columns in data

Y - vert. skip(new line) no. of columns to skip

F → F w.d

decimal real data in decimal form
 precision (no. of digit to be displayed after decimal)

E → E w.d
 total field width including mantissa (i.e. sign exponent)

decimal width i.e. precision of mantissa
 real no in exponential form

0.12 E+03

A → character type.

A w
 width (opt)

(len)

skip later char - len

Not given → size of var. in ch. statement will be

con. as width

T format → output format sp.
col from which output must start

Tn

Aw

NX

Fw.d

Ew.d

Iw

Quote → output print a i t i x inside quotes
normal flow → seq. cons.

Control st → alters flow. u {

Decision makst → dec. wh. to ex. cer. st. or not.

Loop → repeatedly for cer. no. of time, until par. con. is sat.

Decision making stat.

(i) Unconditional go to: Control trans. to label without check.

Go to label-name

✓ label of ex. st. app. in same program

Go to 400

400 stn u

(ii) Computed GO TO

↳ transfer pr. control to one of sev. st. dep. upon value of int. exp.

GO TO ($n_1, n_2, n_3, \dots, n_k$), i
integer var. (evaluates i & goes to the label)

$i = 1 \rightarrow n_1, i = 2 \rightarrow n_2, \dots$

(iii) Logical if stat

✓ checks log. cond. & transfer prog. contr. acc. to

Sum	If (cond) then	go to <u>label</u> else-if-else
	else	car on
	endif	be us

(iv) Arithmetic If stat

↳ transfer prog. control to par. st. dep. upon the value of exp.

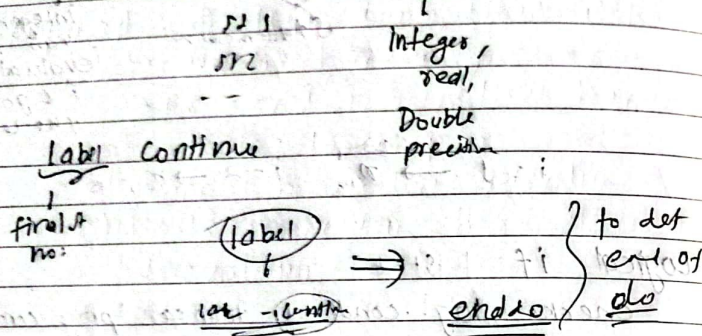
if (expression), n_1, n_2, n_3
valid order in exp. parent
-ve → n_1
0 → n_2
+ve → n_3

Loop:

(i) Do loop → repeat ex. of some stmt. pre-def. in controlled loop

Syntax

Do [label,] loop-until-var = int-val, final-val [Step-in]



(ii) Do while loop

execute set of st. repeatedly while the sp. cond. is true.

DO while (cond-exp) body enddo
OR Do label while (cond-exp) body label continue

(iii) Do until loop → term cond. → end of loop
do st -- until (cond-st) if (cond) goto label

Array → group of related data items sharing common name
→ individual data: elements
same data type

ID: array-name (array-length) → 1 - array-length
OR array-name (lower bound: upper bound) → LB - UB

* C-Middle level language → can do assembly level programming to high level programming

↳ Low level (can access memory directly using pointers)
↳ High level (user friendly)

* compiler/Interpreter: HLL → MLL
C → .obj
source file → executable file or binary file

* Preprocessor directive: line included in program which gives pre-information to the compiler regarding the library function used, and its relevant code to be executed.

process before compilation of any other source code.
↳ start with (#) sp.
<#> <directive name> (identifier)

↳ special instructions which are executed before code passes through compilation process.

* IDE → Integrated Development Environment

↳ handy piece of software that acts as a text editor, debugger and compiler all in one

↳ designed to make coding easier for developers.

↳ a place to write, run, and debug code and also convert it into machine code.

eg: NetBeans, Visual Studio

Character set: set of characters that are used to form words, numbers and expression.

- (i) letters and alphabets (ii) digits
- (iii) special characters (iv) white spaces

Keywords: predefined words in C PL

- have fixed meaning
- reserved words that have special meaning to compiler
- cannot be used as identifiers
- eg: auto, break, return, short, static, typedef, volatile, etc

Identifiers: word used to identify name of variables, function, array, pointer and symbolic constant

- combination of letter, digit, underscore
- no white space
- 31 characters
- no keywords
- case sensitive

$\rightarrow -2^{15}$ to $2^{15}-1$

signed int $\rightarrow$ both +ve & -ve, By default all int are signed
unsigned int $\rightarrow$ only +ve, have to be declared explicitly.

In 16 bit
1 bit to store sign (+ or -) $\rightarrow$ no need to reserve 1 bit for sign
 0 to $2^{16}-1$

ASCII: American Standard Code for Information Interchange

- equivalent numerical value of each character

A-Z $\rightarrow$ 65-90

A-a $\rightarrow$ 97

a-z $\rightarrow$ 97-122

User defined data type: type definition

- allows user to define identifier that would represent existing data type.

typedef $\rightarrow$ give new name to existing data type.

typedef <existing-data-type> <new-name-for-existing-data-type>

eg: typedef int integer.

Derived data types: derived from primary data types

eg: structure, union, enumeration, ~~int, float, char~~

Constant: quantity which doesn't change during execution

Decimal integer constant: +ve or -ve digit

Octal integer constant: leading 0 (zero)
digits $\rightarrow$ 0-7

eg: 073, 0, 0456

hexadecimal integer constant: leading '0x' or '0X'
digit $\rightarrow$ 0-9

alphabet $\rightarrow$ A-F

eg: 0x345, 0x123

Real constant: Floating point constant
 (i) Fractional form constant: at least one digit & a decimal point
 ↳ +ve or -ve
 default: +ve
 eg: 456.0, -43.54

(ii) exponential form

mantissa ^{digit before} (e) ^{digit after} exponent

eg:

$+3.2e5 \Rightarrow [+3.2 \times 10^5]$

$9.1e-5 \Rightarrow 9.1 \times 10^{-5}$

$-2.2e-6 \Rightarrow -2.2 \times 10^{-6}$

Escape sequence: non-printing characters inc.

backslash (\) followed by letter or digit

\' → for single quote

\" → for double quote

\\ → for backslash

\b → delete one character to the left of current cursor position (backspace)

%% → for %

Symbolic constant

#define Symbolic-constant-name <value>
 no blank space between # & define
 blank space between Symbolic-constant-name & <value>
 Identifier
 constant

eg:

#define PI 3.1428

Tokens: basic elements recognized by C compiler.
 ↳ source-program text that cannot be broken down into component elements by compiler.

Keywords, constants, identifiers, string, special symbol.

%X → displays integers in hexadecimal and capital Alphabet format

%x → displays integers in hexadecimal and small alphabet format.

Formatted input: scanf()

scanf("control string", list of variables preceded by &);

control string

[white space-character] [ordinary-character] % [field- [conversion character]

width error
 ↓
 max no. of character to be read
 %d, %f, %c, %s, %i, %e, %E, %g, %G

for printf

↳ If no. of char in cor data item < specified field width.

data item will be preceded by enough leading blank to fill specified field.

↳ If no. of char in the data item > specified field width,

entire data item will be displayed

printf

% [flag] [field width] [precision] [modifier] [conversion specifier]
'-' → indicate data item to be left justified
Flag: '0' → leading 0s to appear instead of leading blank.

Formatted I/O → receive input in fixed format and display output in fixed format
→ works slower than unformatted I/O as unformatted console I/O do not have overheads of formatting the input or output
→ library functions for formatted I/O works for all datatypes
Unformatted I/O: for char & string only.

scanf(): need to press Enter key before the function can digest what we have typed.

getch() & getche(): read single character the instant it is typed without waiting for enter key to be hit.

getch(): Reads character you typed without displaying/echoing on screen.

'e' in getche(): it echoes/display the character you typed to screen.

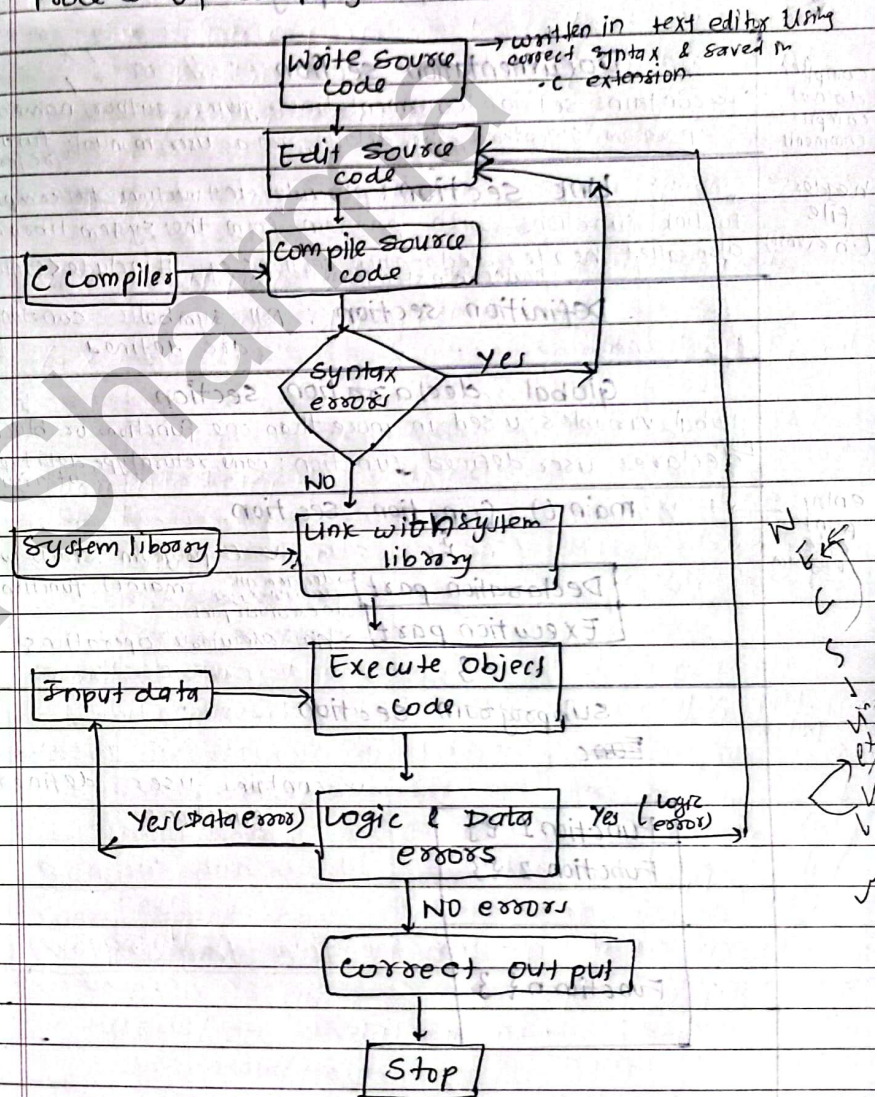
getchar(): similar to getch but requires enter key to be typed following the character you typed.

scanf(): not possible to read multiword string into a single variable directly.
(space, tab, are not acceptable)

gets(): gets string from keyboard; terminated with enter key, space and tab are acceptable.

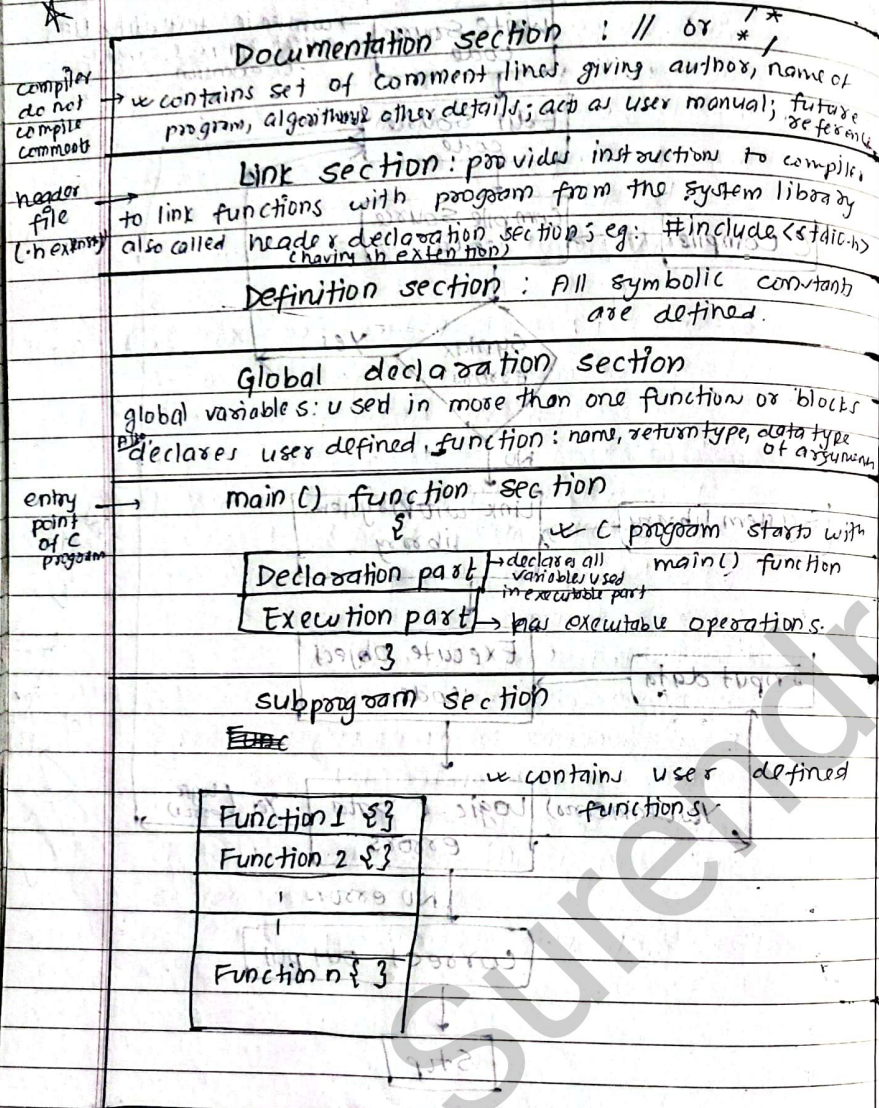
while: entry - controlled loop
do while: exit - controlled loop.

Process of compiling and running a C program



printf ("<format> string", ...);
 % → only for integer.

Basic structure of C program



operators having equal precedence
 evaluated using associativity
 value from

signed char: 0 to 255 ✓
 unsigned char: -128 to 127 ✓

Data type summary

Data type	Category	Range	Memory space	Format specifier (conversion factor)
	Short signed	-32768 to 32767	2	%d or %i ✓
	Short unsigned	0 to 65535	2	%u ✓
int	Signed	-32768 to 32767	2	%d ✓
	Unsigned	0 to 65535	2	%u
	long signed	-2 ³¹ to (2 ³¹ -1)	4	%ld ✓
	long unsigned	0 to 2 ³²	4	%lu ✓
	hexadecimal			%x
	unsigned octal			%o
float	Float	1.2 × 10 ⁻³⁸ to 3.4 × 10 ⁺³⁸	4	%f
double	Double	1.7 × 10 ⁻³⁰⁸ to 3.4 × 10 ⁺³⁰⁸	8	%lf
	long double	3.4 × 10 ⁻⁴⁹³² to 1.1 × 10 ⁺⁴⁹³²	10	%Lf
char	Signed char	-128 to 127	1	%c
	unsigned char	0 to 255	1	%c

typedef → give new name to existing data type

Variable declaration → before using variable in a program

- ↳ the type of variable should be declared or define
- ↳ compiler will allocate memory space in computer as required by the data type
- ↳ syntax: data-type var-name1, var-name2, ...

- ★ gives name of variable & its type
- ↳ allocates appropriate memory space acc to data type.

Rules:

- Var. name should start with letter or underscore only
 - not a keyword
 - case sensitive
 - white-space not allowed b/w the characters of variable
- (2) No two variables of same name are allowed to be declared in same scope.

Character constant 'A' is not same as string constant "A"
"5+3" is string rather than arithmetic operation.

type conversion

Operators: symbol that operates on single or multiple data items.

★ Expression: combination of variables, constants and operators written according to syntax of the language

eg: $8+9$ $a+b*c$ $a>b$ $a*b/3$

operators

{ $y = a + b$ } expressions.

operand

Relational operators

↳ compare left hand side operator with their right hand side operator for less than, greater than, less than and equal to - -

↳ output of relational expression is either 1 (if condition is true)

or 0 (if condition is false)

% (Modulus) → both operands must be integer

number of digits: say (num)

int num, no. of digits

★ Number of digits: $\log_{10}(\text{num}) + 1$

Control statement: statement which alter flow of execution of program.

Two types:

(i) Decision making statements	Loop statements or repeating statements
if statement	for loop
if... else statement	while loop
if... else if statement	do... while loop
Nested if... else statement	
Switch statement	

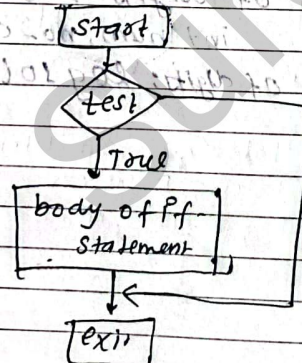
(i) Decision making statements:

- Test a condition and allow to execute some statements on the basis of result of the test i.e. either true or false.
- have capability to decide whether specified statements to be executed or not (so called decision making).

(i) if statements

- first evaluates the expression and then execute the statements within the block if the evaluated exp. is true.

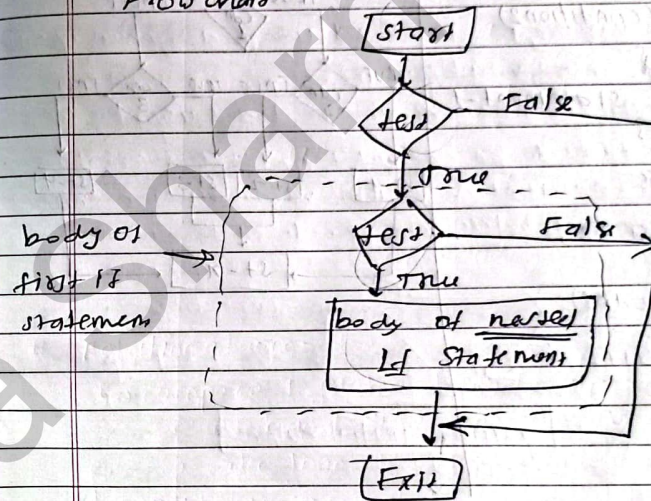
Flowchart:



(ii) Nested if statement

- when one if statement is written within body of another if.
- can declare several if statements using nested if.

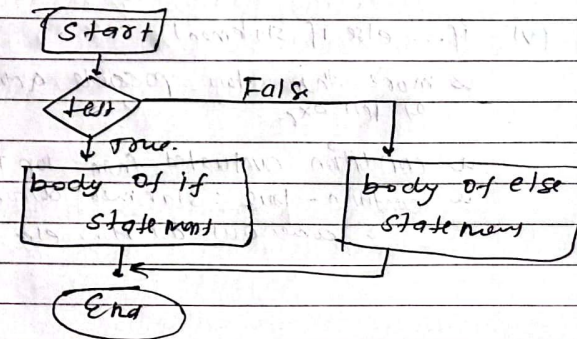
Flowchart



(iii) if... else statement

- used when there are two possible actions.
- one happens when a test condition is true
- other when it is false

Flowchart



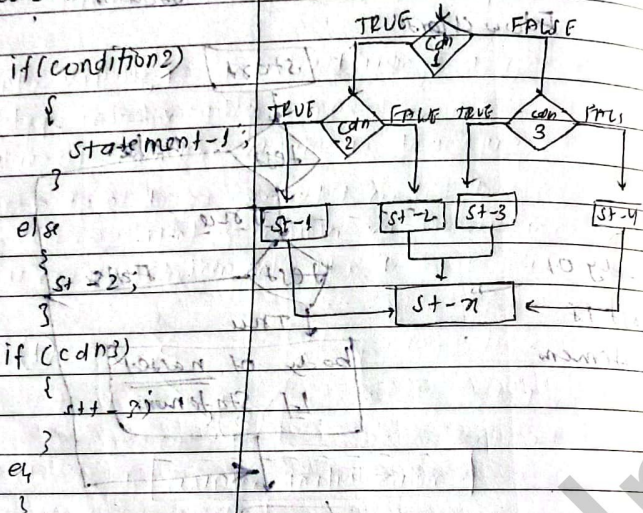
num%2 == 0 → even
else → odd

classmate
Date _____
Page _____

(iv) Nested if-else
↳ if-else written inside body of another if-else (either in body of if or body of else)

Syntax:
if (condition1)

Flowchart:



★ For every if statement corresponding else statement is required in nested-if to avoid dangling effect

↳ if without else is wrong
↳ else without if is wrong

(v) If... else if statement

↳ more than two possible actions depending upon outcome of test exp.

↳ condition evaluated from top to bottom.

↳ condition - true : statement associated to it are executed & comes out of it. else is ladder.

(n1 > n2)
n1 > n2
n1 < n2
else n2 < n1

classmate
Date _____
Page _____

Syntax:

if (con 1)

{
ST-1;
}

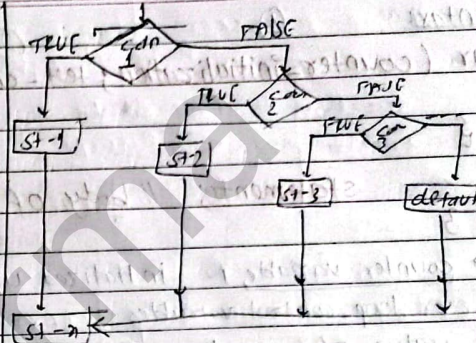
elseif (con 2)

{
ST-2;
}

else

{
default stat;
}

ST-X;



Loop or iteration or repeating construct

★ ↳ block of statements which are repeatedly executed for a certain number of times until a particular condition is satisfied

↳ execute certain block of code repeatedly till a condition exp. is true.

↳ exp → false: control transfer outside the loop.

Loop
(Two segments)

control segment
(decides whether body is to be executed or not)

body

eg: Printing Hello world five times

Types

(1) for loop

★ ↳ execute block of statement number of times.

↳ when number of repetitions is known in advance for loop is more efficient

↳ also known as determinate or definite loop

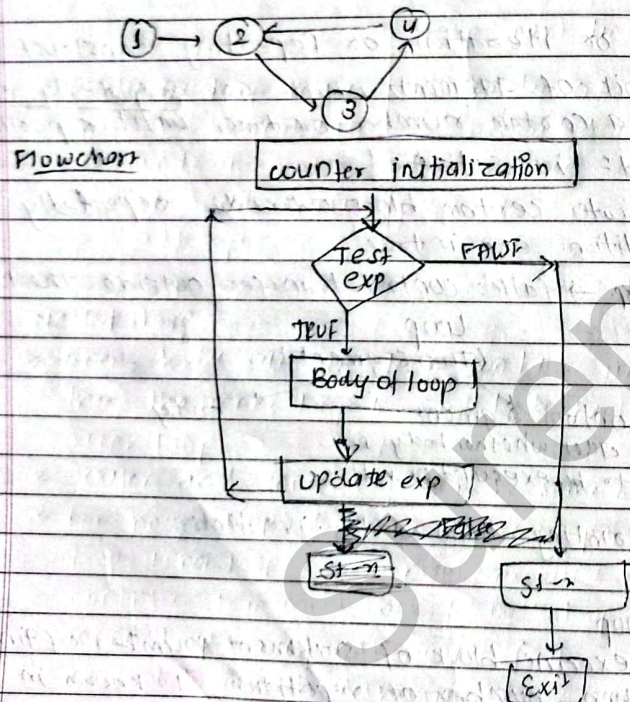
Syntax:

```

for (counter-initialization; test-cond①; updation②)
{
    statements; // body of for loop ③
}

```

- ① → counter variable is initialized using assignment operator or loop-control variable
- ② value of counter-variable is tested using test cond.
- ③ After body of loop is executed, control transfers back to for statement & counter variable is incremented or decremented then only it is to be tested.



(2) while loop

Syntax:

```

while (test cond①)
{
    body of loop;
    updation;
}

```

eg: Printing no from 1 to 10

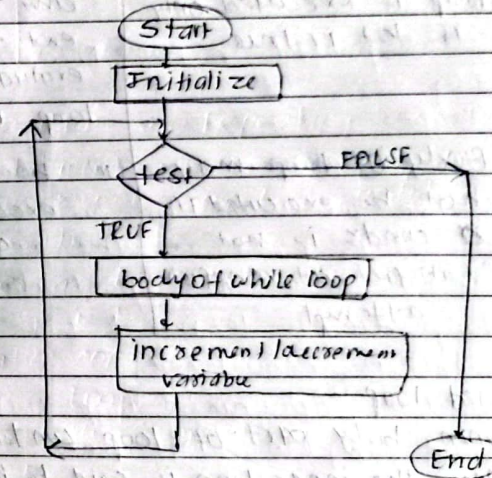
```

int i = 0;
while (i <= 10)
{
    printf("%d", i);
    i++;
}

```

we when frequency of repetition is not known in advance while loop is suitable.

Flowchart:



we for loop → while loop we

we while loop → for loop is not efficiently

(3) do-while loop

Syntax:

```

do
{
    // Statement of body of loop;
} while (test - cond①);

```

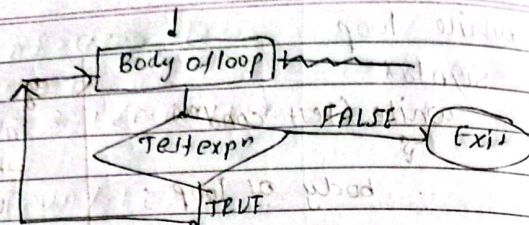
eg: char ch;

```

do
{
    ch = getch();
    printf("You typed: %c", ch);
    printf("Do you want to exit? Y/N");
} while (ch != 'Y');

```


Flowchart



we

while

- (i) entry-controlled loop
- (ii) test condⁿ evaluated first and body of loop is executed only if test is true.

- (iii) Body of loop may not be executed if condⁿ is not satisfied at very first attempt

(iv) Nested loop

we when body part of loop contains another loop then the inner loop is said to be nested with outer loop

we outer and inner loop may be same or diff. type.

we for nested loop; for each value or pair of outer

loop, inner loop is completely executed then only control is transferred back to outer loop

we inner loop → operates/moves fast

we nested loop ≈ hour & minute hand of clock

eg: multiplication table of 5-10

do-while

- (i) exit-controlled loop
- (ii) Body of loop is executed first without checking any condition and at end the condition is evaluated for rep. of loop, next time

- (iii) body of loop is executed at least once.

eg: for (ctr initialization; test condⁿ; updation)

{

for (ctr initialize; test; updation)

{

state;

}

inner for loop

outer loop

break and continue statements.

(i) break

↳ terminates execution of loop and control is transferred

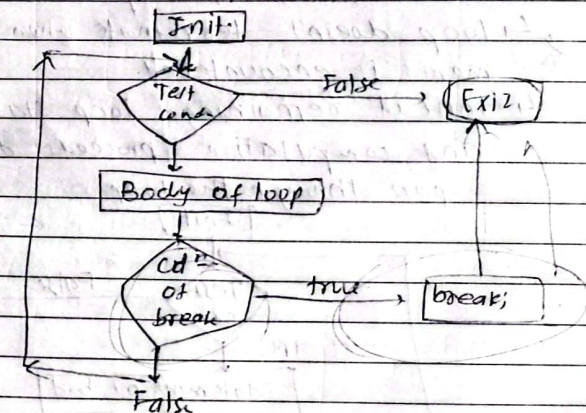
★ to the statement immediately following the loop

↳ Generally, loop terminated if its test condⁿ is false.

↳ If loop is to be terminated instantly without checking test condⁿ break statement is used.

↳ used in switch statement

Flowchart: break with for loop

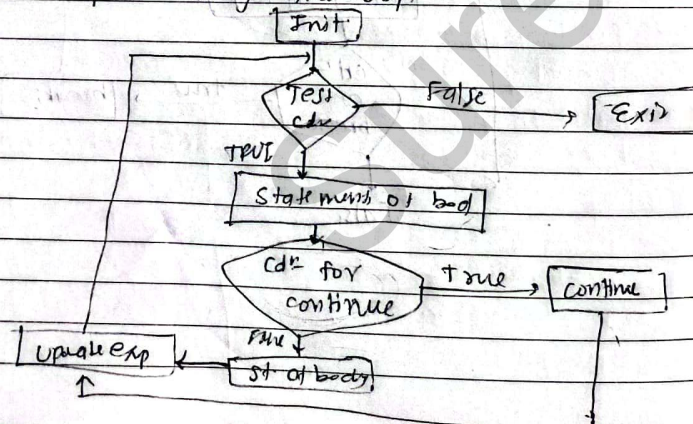


eg: prime no. using break

```
for (i = 2; i < num; i++)
{
    if (num % i == 0)
    {
        printf("It is NOT prime number!!!");
        break;
    }
}
if (i == num)
{
    printf("It is prime Number!!!");
}
```

(2) Continue statement

- ↳ when we want to continue running loop, but stop processing the remainder of the code in the body for particular iteration.
- ↳ loop doesn't terminate when continue statement is encountered.
- ↳ instead remaining loop statements are skipped and computation proceeds directly to next pass through the loop.



eg: displaying only even no

```
for (i = 1; i <= num; i++)
{
    if (i % 2 != 0)
    {
        continue;
    }
    printf("%d ", i);
}
```

Break

- (i) break loop or iteration
- (ii) used in switch case loop
- (iii) key word used is "break"
- (iv) Breaks loops and allows coming out from it.
- (v) control is transferred outside the loop.

Continue

- (i) continues the loop or iteration
- (ii) not used with switch case.
- (iii) keyword used is "continue"
- (iv) Allows iterating in the loop.
- (v) control remain in the same loop.

goto statement

- ↳ alter normal sequence of program execution unconditionally transferring control to some other part of the program.

- ↳ transfer control to labeled statements.

Syntax for goto statement:

goto label;

Syntax for labeled statement:

label: statements;

we avoided a make code very hard to read.

Use: we jump to end of loop under certain cond

↳ jump completely out of loop under certain cond, terminating execution of a loop.

we return; after label statement

digit: $\frac{num}{10}$
48-57

switch statements

- ↳ when number of options available and one is to be selected on basis of some criteria.
- ↳ allows user to choose a statement among several alternatives.
- ↳ compares a var. or exp with diff. constants (literals).
- ↳ if equal to a case constant, the statements all are executed.

↳ constants → char or int type only.

syntax:

switch (variable or exp.)

{

case caseconst 1:

statements;

break;

case caseconst 2:

statements;

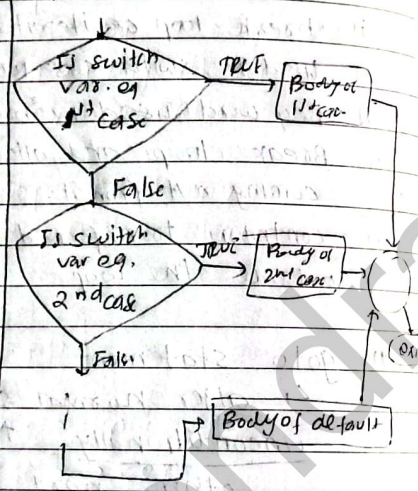
break;

default:

statements;

}

Flowchart:



rem = num % 10;

(first digit = num / pow(10, no-of-digits-1))

num = num - digit * divisor;

divisor *= 10;

Function

↳ self-contained block of statements that performs a particular specified job in a program.

Advantages:-

- (1) Manageability: Code for each job or activity is placed in individual funcⁿ so testing & debugging become easy and efficient.
- (2) Code Reusability: single funcⁿ can be used/called multiple times.
- (3) Non-Repeated programming: Avoids rewriting.
- (4) Logical clarity: main program calls function call rather than counting no. of lines.
- (5) Easy to divide work among programmers.

↳ user defined function.

main(): entry point of every program.

↳ calling function

other funcⁿ: called funcⁿ { main(): name of funcⁿ is defined or fixed by language.

Library (Built-in) function: function already written, compiled and placed in C library and are not required to be written by a programmer again.

↳ Built-in functions defined within particular header file.

argument in function call → parameter
 " in " prototype/definition → Formal parameter

we no statement within function body is executed after the return statement

Recursive function

Problem

- (i) Problem could be written or defined in term of its previous result
- (ii) Problem statement must include a stopping condition :

If number of values provided while initialization is less than the size of array; zero is assigned to the remaining elements.

Single operation, which involves entire arrays, are not permitted in C.

Assignment, comparison operation involving two arrays must be carried out on element-by-element basis.

[!flag == flag == 0]

In C, Truth → non-zero
 False → zero

if (5) {
 //
 }

ternary operator (?:) can be nested.

variable count; Also called loop counter or index variable

In place of condition in conditional

Statements/loops there can be expression
 As long as expression evaluates to non-zero expression within the body will be executed.

☆ -32768 to 32767
 int i = 32767

i++;
 printf("%d", i); → -32768

we can put a pair of braces around any individual statement or set of statements without affecting the execution of program.

while statement: condition check is only done in while statement

for statement: initialization, updation & testing of loop counter is done in for statement itself

Even though we not include one or more of initialization, testing & updation statement ; should still

(i=1); be given

eg: for (; i <= 10;)

{
 printf("x.d\n", i);
 i = i + 1;
 }

break: when break is encountered inside any loop, control automatically passes to the first statement after the loop.
 takes execution control out of the loop.

switch: control statement that allows us to make decisions from number of choices

cases can be in any order in switch.

char or int values in case

once a case is satisfied the control simply falls through the case till it doesn't encounter break statement.

only char & int constants or an exp that evaluates to one of these constants are only allowed.

switch versus if-else ladder

- float exp. cannot be tested using switch.
- cases can never have variable expression in switch (No eg: it is wrong to say $a+3$;))
- Multiple cases cannot use same expression.
- ~~switch~~ switch works faster than an equivalent if-else ladder (if more than two cases are involved)

goto

- ↳ violates the normal flow of execution
- ↳ so need to be avoided.
- ↳ difficult to trace the control flow of program making the program hard to understand & hard to modify.
- ↳ reduces readability of the program
- ↳ poor programming practice.

Adv: can alter the normal sequence of program execution so it gives the power to jump to any part of program.

Function: self contained block of statements that perform particular task.

C program: → collection of functions.

Using function & hiring a person to do a specific job.

(nested function is not supported in C)

A function cannot be defined in another function.

Argument: mechanism used to convey information to the function is the 'argument'.

- ↳ also called parameters.
- arguments in calling function: actual arguments.
- arguments in called function: formal arguments.

↳ Any number of arguments can be passed to a function being called. However, the type, order and number of the actual and formal parameters must always be same.

↳ function can return only one value at a time.
↳ On executing the return statement it immediately transfers the control back to the calling function.

↳ scope of a variable is local to the function in which it is defined

↳ C's calling convention is from right to left. (, → associativity: right to left)

↳ stdio.h contains prototype of all I/O-function
↳ C function by default return int value (if not mention return type)

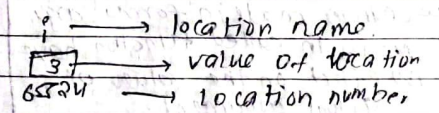
↳ prototype declaration ^{can be} done in main () ^{for calling} however advisable to do in beginning of the program.

declaration | what actually happen,

int i = 3;

above declaration tells C compiler to

- (1) Reserve space in memory to hold the integer value.
- (2) Associate the name i with the memory location
- (3) Store the value 3 in this location



%u → print address (unsigned int)
Data structures → ways of organizing data

& → 'address of' * → 'value at address'
↳ indirection operator

int *j;

tells compiler that j will be used to store the address of an integer value.

j → pointer to int (integer pointer)
int *j → value at the address contained in j

pointer variables → variables capable of holding address

↳ addresses are always whole number
↳ pointers would always contain whole number

eg: char *ch → doesn't mean that ch is going to contain character value
↳ rather means that ch is going to contain the address of a char value

call by value

we sending values of the arguments

we each of the actual arguments in calling function is copied into corresponding formal arguments of called function.

we changes made in formal arguments in called function have no effect on the values of actual argument.

we can return only one value at a time

call by reference

we sending addresses of the arguments

we address of actual arguments in the calling function are copied into formal arguments of the called function.

we Using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them.

we Using this we can return more value at a time. ^{indirectly}

Call by reference → If want actual parameter to be changed in calling function.
state → last in first out (LIFO) data structure

Recursion: function calling itself

↳ i.e. statements within the body of a function call the same function.

↳ For functions which can be expressed in the form of itself.

Function declaration → specifies what is return type of the function and the types of parameters it accepts.

Function definition → defines body of the function.

[*arr][10] = arr[10]

long int ≈ long

short int ≈ short

Sometimes when the constant is small enough to be an int, but still we want to give it as much storage as long. In such case we add suffix 'L' or 'l' at the end of number

eg: 23L

char ch = 'A';

Here it gets stored in ch is the binary equivalent of the ASCII value of 'A' (i.e. bin of 65)

ch = 127;

ch++

ch → 128 instead of 127.

%f

%lf

%Lf

float < double < long double

4 bytes

8 bytes

10 bytes

char ;
↳ unsigned char;
↳ %c

void pointer → pointer that has no data type associated with it
 general purpose pointer → can easily be typecasted to any pointer type

short signed int → %d	Size depends on bits of compiler (16-bit compiler) (32-bit compiler)
short unsigned int → %u	
signed int → %d	
unsigned int → %u	
long signed int → %ld	
long unsigned int → %lu	

Four storage classes in C

Properties	Automatic Storage class	Register storage class	Static Storage class	External storage class
Storage →	Memory	CPU registers (so can be accessed faster)	Memory	Memory
Default initial value (if not initialized)	An unpredictable value, called garbage value.	Garbage value	zero	zero
Scope →	Local to the block in which variable is defined.	Local to the block in which the variable is defined.	Local to the block in which variable is declared.	Global
Life →	Till the control remains within the block in which the variable is defined.	Till the control remains within the block in which the variable is defined.	Value of persists b/w different function calls.	As long as the program's execution doesn't come to an end.
Keyword →	<u>auto</u>	<u>register</u>	<u>static</u>	<u>extern</u>
Block:	{ ; enclosed by curly braces }	→ (loop counted) value stored in CPU register is accessed faster than stored in memory. so, variable used at many places in program it is better to declare storage class as register.	(do not disappear when function is no longer active) i.e. their value persists.	outside all the functions.
Example:	auto int i;	register int i;		→ if the variable are being used in almost all functions.

extension int y; ⇒ declaration } A variable can be declared several times but defined only once.
 int y = 31; ⇒ definition
 → when conflict arises b/w local & global variable (scope)
 ↳ local variable gets precedence over global variable.

Preprocessor: program which processes our source program before passed to the compiler.
 C-Preprocessor ⇒ not part of compiler.
 Preprocessor directive → Features offered by preprocessor.

begin with # symbol
 Types: #define (macro template) PI 3.14 (macro expansion)
 (a) Macro Expansion directive

eg: #define PI 3.14
 ↳ During preprocessing, the preprocessor replaces every occurrence of PI in the program with 3.14
 ↳ replaces every macro template in the program with appropriate macro expansion.
 ↳ not terminated by semi colon (;)
 ↳ usually macro template written in capital letter
 ↳ makes program easier to read & recognize & reveal purpose.
 ↳ can make changes to values of constant at all places in program just by making change in #define directive.
 ↳ #define directive → could replace condition or even an entire C statement.

#define AREA(x) (3.14 * x * x)
 ↳ Macros → takes more space.
 ↳ operate much like functions.
 ↳ define preprocessor variables.

#NULL → built in constant that has value of zero.
0 can also be value of pointer which is same as zero.

(b) File inclusion

- ↳ this directive causes one file to be included in another.
- Preprocessor command for file inclusion.

#include "filename" → fetch from current directory.

(OR)

#include <filename> → fetch from standard system directory.

- ↳ files to be included should have .h extension.
- .h stands for 'header file'.

eg: (i) prototypes of all mathematical related functions are stored in header file math.h

(ii) prototypes of console input/output functions are stored in conio.h

Flow of C program

Pre processing → Compilation → Assembly → Linking

#endif → specifies end of #if

#pragma → to issue some special commands to compiler

#define → also called macros/substitutes

• #include → include one header file in another

• #undef → undefines preprocessor macros

• #ifdef → checks if sth is defined or not

& defined → returns true & defined else → returns false

• #ifndef → if not defined → returns true

• #if → if any compile time condition is true

• #else → Alternative of #if

• #elif → using else if in same statement

0 to n-1
contiguous memory locations

Array: Collection of similar elements

• Array declaration: what kind of array, its name and how large the array

int mark[30] → size/dimension of array
↑ data type of variable
↑ name of variable
[] square braces: tell compiler we are dealing with array.

Call by value & Call by reference: Passing array elements to function

Every time pointer is incremented it points to the next location of its type.

Array elements are always stored in contiguous memory locations.

Accessing array elements by using pointers is always faster than accessing them by subscript. address of zeroth element (called the base address) can be passed by just passing the name of the array.

1-D
 $\&\text{num}[0] \equiv \text{num}$ ← base address
 $*(\text{num}+0) \equiv * \text{num} \equiv \text{num}[0]$ ← base element / zeroth element
 $\text{num}[i] \equiv *(\text{num}+i)$ ← value of ith element.

2D array → matrix of collection of one-d arrays placed one below another

In memory, whether 1-D or 2-D, both array elements are stored in contiguous memory location.

Each row of 2-D array → 1-D array.

2-D
 $\text{sc}[0] \equiv \&\text{sc}[0][0]$ → address of zeroth 1-D array
 $\text{sc}[1] \equiv \&\text{sc}[1][0]$ → address of first 1-D array
 $\&\text{sc}[2][1] \equiv \text{sc}[2]+1 \equiv *(\text{sc}+2)+1$ → address of element of second row & 1st column

Subroutine or procedure

Function $\rightarrow$ Point to code, not data
pointer, when you want to pass address of a function to another function.

$S[R][C] \equiv *(S[R]+1) \equiv *(*(S+2)+1) \rightarrow$ value of element at second row & 1st column

$\hookrightarrow S[i][j] \equiv *(S[i]+j) \equiv *(*(S+i)+j)$

$\hookrightarrow \&S[i][j] \equiv S[i]+j \equiv *(S+i)+j$

$\hookrightarrow$ Pointer to an array {2-D}

`int S[R][C];`

`int (*p)[C];` $\rightarrow$ parenthesis in declaration of ptr necessary

contains address of zeroth element

`p = &S[0]`

OR `p = &S[1]`

$\rightarrow$ Absence of parenthesis makes p an array of $\leq$ integer pointers.

Passing 2-D arrays to a function.

`int main()`

{

`int a[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 6 };`

`display(a, 3, 4);`

`show(a, 3, 4);`

`print(a, 3, 4);`

}

`display(int *q, int row, int col)`

{

`int i, j;`

`for (i = 0; i < row; i++)`

`{ for (j = 0; j < col; j++)`

`{ printf("%d", *(q + i * col + j));`

`printf("\n");`

}

}

`show(int (*q)[4], int row, int col)`

{

`int i, j;`

`int *p;`

`for (i = 0; i < row; i++)`

`{ p = q + i;`

`for (j = 0; j < col; j++)`

`{ printf("%d", *(p + j));`

`printf("\n");`

}

`print(int q[][4], int row, int col)`

{

`int i, j;`

`for (i = 0; i < row; i++)`

`{ for (j = 0; j < col; j++)`

`{ printf("%d", q[i][j]);`

`printf("\n");`

}

}

void pointer

Generic pointer

$\rightarrow$ which has no association with it. $\rightarrow$ pointer to anything.

classmate
Date _____
Page _____

String: Array of characters terminated by a null character ('0')

ASCII value of '\0' $\rightarrow$ 0

ASCII value of '0' $\rightarrow$ 48

Initializing

`char name[] = "RAM";` $\leftarrow$ C inserts null character automatically

OR `char name[] = { 'R', 'A', 'M', '\0' };`

Similar to integer array character array can be accessed as:

`name[i]  $\equiv$  *(name + i)`

`strlen(str)`;

$\rightarrow$ what gets passed to the function `strlen()` is the base address of string and not the string itself.

www.surendrasharma0001.com.np