Computer Introduction

What is Computer?

→ A computer is an electronic device designed to perform a wide variety of tasks by following a set of instructions, or programs. It processes data by taking input, applying computations, and producing output, making it a powerful tool for solving complex problems and automating tasks.

→ This is an electronic device that processes, stores and retrieves data.

→ It follows a set of instructions, called programs, to perform tasks such as calculations, data management and running applications.

→ At its core, a computer operates using binary code (0s and 1s), but high-level programming languages like C, C++, Java and Python are used to write human-readable instructions that are translated into Machine code (0s and 1s).

High-Level languge

↓

Compiler / Interpreter

↓

Assembly Language

↓

Assembler

↓

Machine Code

1. High-Level Language: e.g. C, C++, Python, Java
2. Compiler / Interpreter: Converts the high-level language to assembly language or directly to machine code.
3. Assembly Language: Low level language closely related to machine instructions.
4. Assembler: Converts assembly language to machine code
5. Machine Code: Binary code → 0s and 1s, the language understood by the computer hardware

---

Programming Language Introduction

What is Programming Language?

→ A programming language is a formal language used to write instructions that a computer can execute.

→ It allows programmers to communicate with computers by providing specific commands and logic that computer follows to perform tasks.

Why are we using programming Languages?

1. Human Readability:

→ Programming languages bridge gap between human logic and machine code.

→ Computers only understand binary (0s and 1s), programming languages allows us to write instructions

in a more understandable, structured way.

2. Efficiency:

Programming languages help automate task, perform calculations, and process data faster and more efficiently than humans can manually.

3. Reusability:

Code written in programming languages can be reused, modified, and scaled across different applications or platforms, and making software development more efficient.

4. Problem solving:

With programming languages, we can solve complex problems by creating algorithms, and logic that instruct computers on what to do

---

Introduction of C Programming Language

→ C is a general purpose, high-level programming language developed inthe early 1970s by Dennis Ritchie at Bell Labs.

History and Origins of C:

Who found C?

→ Dennis Ritchie developed C in 1972 at AT&T's Bell Laboratories.

→ Ritchie and his colleagues needed a more flexible and powerful language for system development, particularly for Unix Operating System.

How did C come about?

→ Before C language like B and BCPL (Basic Combined Programming Language) were used for system programming.

→ However, these languages had limitations in terms of flexibility and performance

→ Dennis Ritchie improved upon the B language to create C, which addressed these limitations.

Why was C created?

→ C was especially designed to rewrite the Unix operating system, which was originally written in assembly language

→ The main goal were to create a language that was portable, meaning it could run on different machines, and efficient enough to perform low-level tasks, such as managing memory and hardware

Why the name C?

→ The name C was choosen because it directly followed the language B, which was a predecessor created by Ken thompson. since C was an important over B, it naturally took the next letter in the alphabet.

Mother and Father of C:

Father of C: The 'father' of C is Dennis Ritchie who designed and implemented the language.

Mother of C: The 'mother' of C could be considered Ken Thompson's B language, from which C evolved.

Some example Applications of C:

1. operating system.
2. system-level Programming
3. Embedded systems
4. General software Development
5. Portable Code

The unique feature of C:

→ one of the biggest unique feature of C is its close relationship with hardware while maintaing high-level language characteristics.

→ C allows direct manipulation of memory using pointers.

Why is C still important today?

→ Eventhough many modern languages have been developed, C remains crucial in areas like embedded systems, operating systems, and system software development because of its low-level access to memory, efficieny and speed.

→ Many newer languages like C++, Java and Python have borrowed key concepts from C, making it a foundation language in computer science

---

Pseudocode

What is Pseudocode?

→ pseudocode is a simplified, informal way of writing a program's logic using plain, human-readable language.

→ It doesn't follow the strict syntax rules of any specific programming language.

→ Instead, Pseudocode use a mix of natural language and programming constructs like loops, conditions, and variables to represent how the logic will work.

→ While pseudocode cannot be directly compiled or executed by a computer, it serves as an intermediate between the problem description and the final implementation in a programming language.

Why are we using Pseudocode?

1. To simplify complex logic
2. Improves communication
3. Focus on the logic.

Purpose of using Pseudocode:

1. Algorithm planning:

It helps in planning the structure and flow of a program before writing the actual code.

2. Improving clarity:

By using simple language, pseudocode makes it easy to understand the logic of a program

3. Guideline for code Implementation:

It provides a roadmap for the actual coding process, ensuring that programmers know exactly what needs to be done.

Benefits of Using Pseudocode in Programming:

1. language - Independent.
2. Time-saving.
3. Easy Debugging
4. Better Understanding
5. Improves problem - solving.

Example of Pseudocode:

```
START
  Input two numbers, A and B
  IF A is greater the B
     Display "A is the largest".
  ELSE
     Display "B is the largest"
END
```

Algorithm

What is an algorithm?

An algorithm is simply a step-by-step set of instructions to solve a problem or complete a task

Why use algorithms?

→ Algorithm helps us solve problems more efficiently and systematically.

→ It give a clear path to follow, just like how a GPS guides you to a destination

Purpose of algorithms in Programming:

1. They break down complex tasks into simple steps.
2. They help make programs faster and easier to understand

How algorithms work:

1. Step-by-step: Each task is breaking into small steps.
2. Order matters: Steps must be followed in the correct order for the algorithm to work properly.
3. Repeatable: Every time you follow the steps, you will get the same result.

Some real-time examples for making an algorithm:

① Making a sandwich:

→ Take two slices of bread.
→ Spread butter on one slice
→ Put a slice of cheese on the buttered bread
→ Cover with second slice of bread.
→ Cut the sandwich into two halves.

→ It is a step-by-step process that guarantees the task is done correctly.

② Finding the largest number:

you are given three numbers: 3, 7 and 5. You need to find the largest one.

→ Compare 3 and 7 : 7 is larger
→ compare 7 and 5 7 is larger
→ Result: the largest number is 7.

③ Tying shoelaces:

→ Cross the two laces over each other.
→ Pull one lace through the loop created
→ Tighten the laces.
→ Make a loop with each lace.
→ Cross the loops and pull one through the other loop.
→ Tighten the knot.

Flowchart

Flowchart?

→ Flowchart are visual diagrams that map out the steps or decisions in a process.

→ Imagine it like a blueprint for understanding how something works.

→ Making it easier to analyze and communicate the logic behind it.

→ In programming, flowcharts help programmers plan out their code before they start typing.

→ They show the sequence of actions, decisions and loops, making it easier to spot potential errors or areas for improvement before any code is written.

key components of flowcharts:

1. Start / End (Oval):

→ This symbol represents the beginning or end of a process.

→ Always used at the start and the conclusion of the flowchart.

2. Process / Action (Rectangle):

→ Represents a specific task or action to be performed.

→ Used for actions like calculations, assignments, or events that occur in a process.

3. Decision (diamond):

→ Indicates a decision point where different outcomes are possible

→ You have to choose between two or more paths based on a condition

→ Examples: "Is the water hot?" (Yes/No) - each path leads to a different result.

4. Arrows (connecting lines):

→ shows the flow of control or direction from one step to another

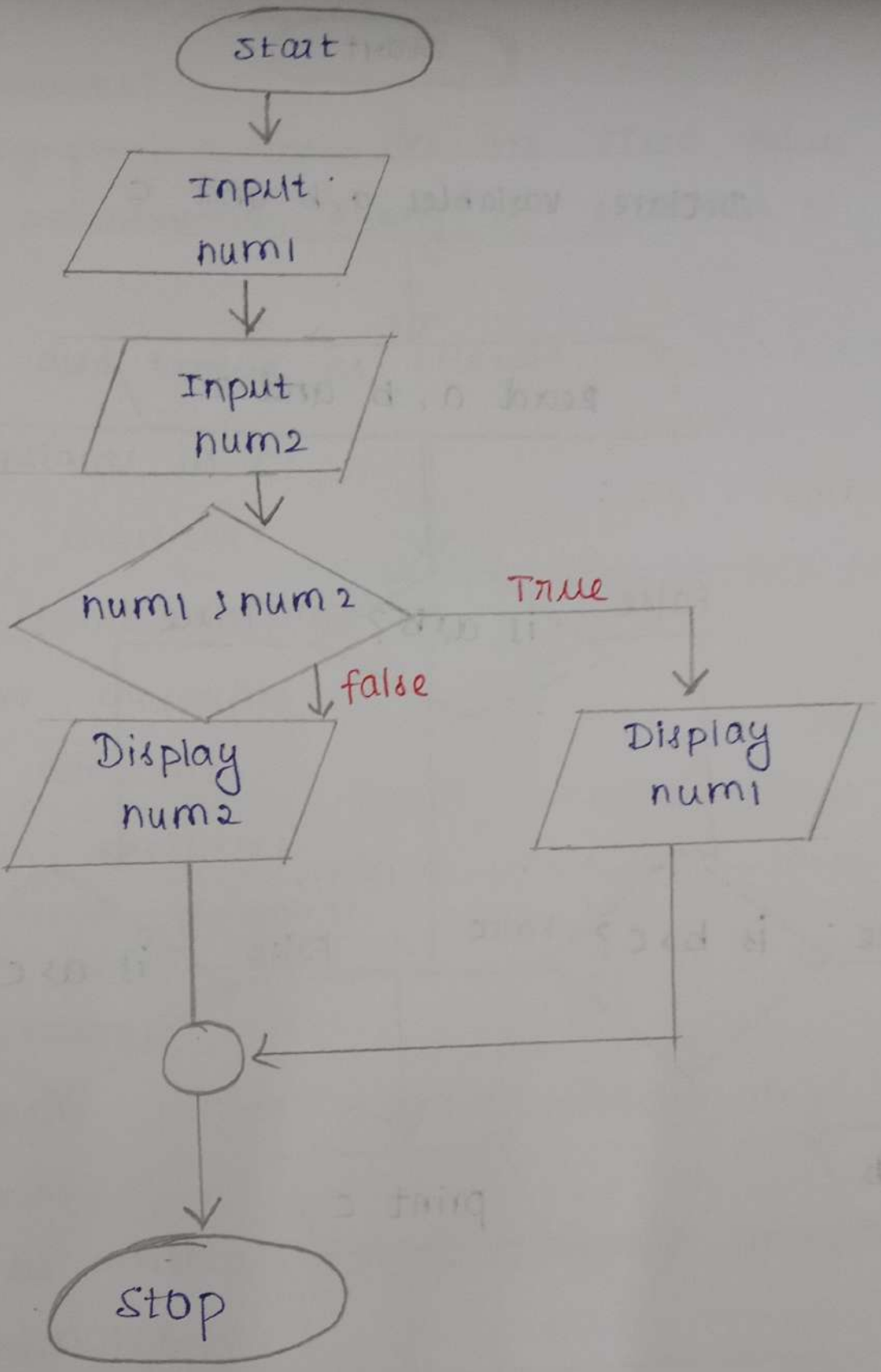
→ Used to connect shapes and represent the order of execution

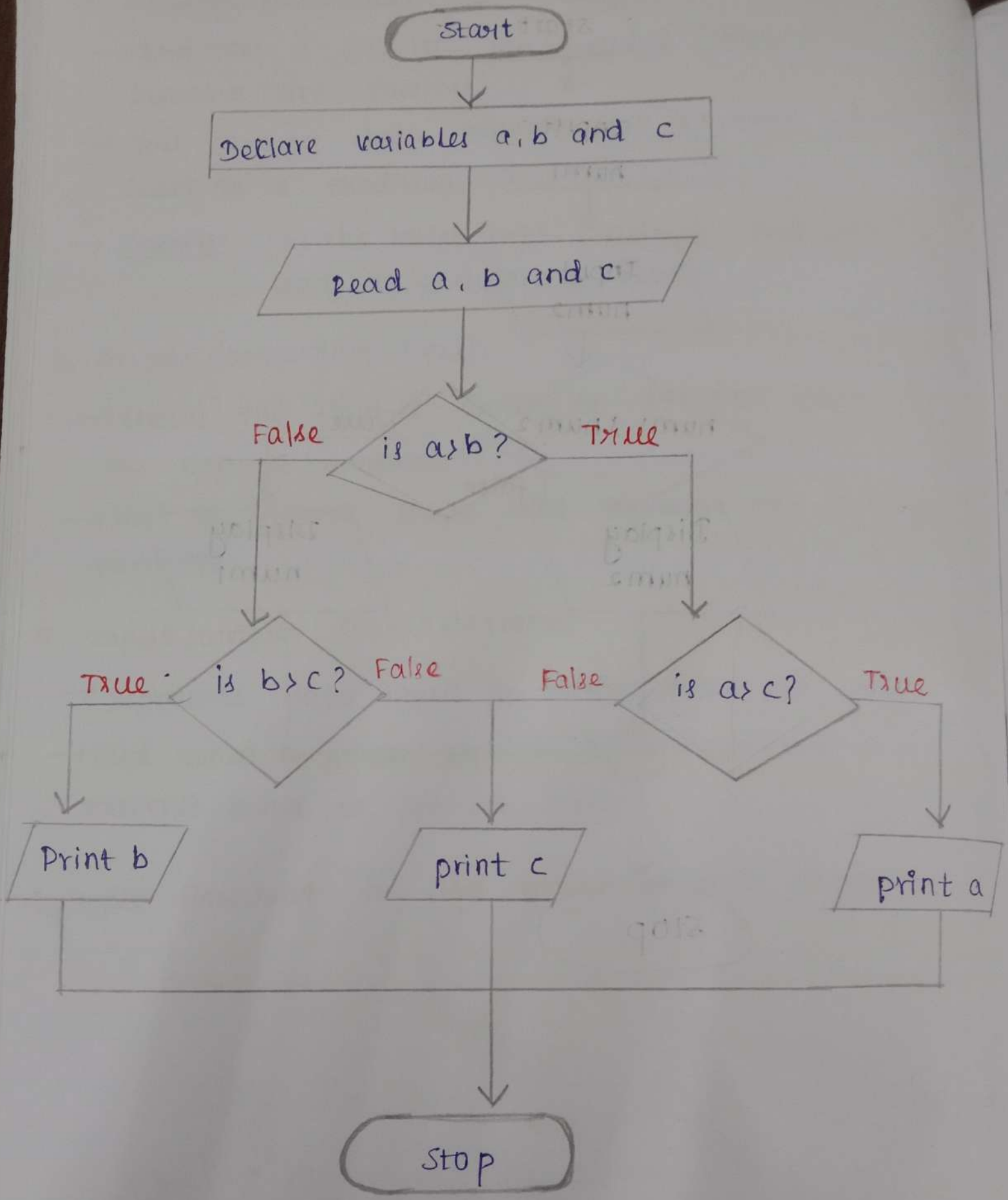
5. Input/output (Parallelogram):

→ Represents data input or output from the process.

→ Used when a process requires data from an external source or produces data

A simple flowchart to find largest of two numbers:


Start
Input num1
Input num2
num1 > num2
True
false
Display num2
Display num1
Stop


Start
Declare variables a, b and c
Read a, b and c
is a>b?
False
True
is b>c?
True
False
is a>c?
False
True
Print b
print c
print a
Stop

Constants

What is constants?

→ In C programming, constants are fixed values that cannot be changed during the execution of a program.

→ They are also known as Literals.

Types of Constants in C:

1. Integer constants
2. Floating point constants
3. Character constants
4. String constants
5. Symbolic constants.
6. Enumeration constants.

① Integer constants:

→ These represents whole numbers without any decimal point.

→ They can be written in three different number system: decimal (base 10), octal (base 8), hexadecimal (base 16).

Examples:

decimal constants: 10, 0, -45

Octal constants: 012 (represents 10 in decimal, octal constants start with 0)

Hexadecimal constant: 0X1A (represents 26 in decimal; hex constants starts with 0x).

In C,

```
int decimal = 10;   // decimal
int octal   = 012;  // octal
int hex     = 0x1A; // hexadecimal
```

② . Floating point constants:

→ It represents real numbers (numbers with fractional parts).

→ A floating point constant can be written using decimal notation or scientific notation (exponential).

Examples:

→ Decimal notation: 3.1415, 0.006, -0.05, 35.65.

→ scientific notation: 2.5e3, 2.e61

[which represents 2.5 * 10^3 or 2500]

In C,

```
float pi = 3.1415;  // decimal notation
float exp = 2.5e3;  // scientific notation (2500).
```

③. Character constants:

→ A single character enclosed in single quotes. (' ')

→ It represents a single character value in the ASCII character set.

Examples: 'A', 'a', 'b', 'z', '\n' (newline character).

In C,

```
char letter = 'A';  // character constant
char newline = '\n'; // newline constant.
```

④. String Constants:

→ There are sequence of characters enclosed in double quotes (" ").

→ They represents an array of characters.

Examples:

"Hello, World!", "C Programming".

In C,

```
char greeting[] = "Hello, World!";
```

⑤. Symbolic Constants:

→ There are constants defined by the #define preprocessor directive or using the const keyword.

→ They allow us to define meaningful names for constants, which helps with code readability and mainability.

Using #define:

```
#define PI 3.14159
#define MAX_SIZE 100

int main() {
    int radius = 5;
    float area = PI * radius * radius;
}
```

Using const:

```
const int maxSize = 100;
const float pi    = 3.14159;

int main() {
    int radius = 5;
    float area = pi * radius * radius;
}
```

⑥. Enumeration Constants:

→ These are user-defined constants that represent integer value.

→ Enumeration constants are defined using the enum keyword.

Examples: In C,

```
enum week {sunday, Monday, Wednesday, Thursday,
           Friday, Saturday}.
```

```
int main() {
    enum week today = Wednesday ; // Wednesday is
}                                 assigned integer value 3.
```

Constant Type	Example
Integer const	10, 0x1A, 012
Floating-point constant	3.14, 2.5e3
Character constant	'A', '\n'
String constant	"Hello, World!"
Symbolic constant	#define PI 3.14 (or) const pi = 3.14
Enumeration constant	enum week {sunday, Monday}

Operators

→ In C, 45 operators there.

→ In C programming, operators are symbols that tell the compiler to perform specific mathematical, logical or bitwise operations.

→ They work with operands (values or variables) to produce a result.

a + b
↳ + is an operator, that adds [a and b]
↓
operands.

→ C has 45 operators, divided into several categories based on their functionality:

1. Arithmetic Operators : 5 operators
2. Relational operators : 6 operators
3. Logical operators : 3 operators
4. Bitwise operators : 6 operators
5. Assigment operators : 6 operators
6. Increment / Decrement operators : 2 operators
7. Conditional operator : 1 operator (Ternary)
8. Special operator : 3 operators. [e.g. size of, comma, pointer difference].
9. Shift operator : 2 operators
10. Miscellaneous operators : 10 (includes &, *, size of, etc).

① Arithmetic operators [5 operators]:

operator	Description
+	Addition
-	subtraction
*	Multiplication
/	Division
%	Modulus (reminder).

→ These operators are used for basic mathematical operations.

② Relational operators [6 operators]:

→ These operators used to compare two values and return a boolean result [true or false].

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

③. Logical operators (3 operators):

→ these operators are used for logical operations

Operator	Description
&&	Logical AND
!	Logical NOT (negation).

④. Bitwise operators (6 operators):

→ These operators perform bit-level operations on integers.

Operator	Description
&	Bitwise AND
^	Bitwise XOR. [exclusive OR]
~	Bitwise NOT (complement)
<<	Left shift.
>>	right sift.

⑤. Assignement operators:

→ These operators used to assign values to variables.

Operator	Description
=	Assignment
+=	Add and assign
-=	subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign

(6). Increment / Decrement operators (2 operators).

→ These operators used to increase or decrease the value of a variable by 1.

Operator	Description
++	Increment by 1
--	Decrement by 1

(7). Conditional (Ternary) operator (1 operator):

→ This is a compact form of an if-else statement.

Operators	Description
?:	Ternary operator (condition? expr1: expr2)

⑧. special character operators (3 operators):

Operator	Description
&	Address-of operator
*	Deference operator
sizeof	sizeof operator

⑨. Shift operators (2 operators):

Operator	Desciption
<<	left shift (shifts bits to left)
>>	right shift (shifts bits to right)

⑩. Miscellaneous Operator (10 operators):

operator	Description
?:	conditional (Ternary) operator
sizeof	sizeof operator
,	Comma operator
&	Address of operator
*	Difference operator
.	structure/union member access operator (dot)
->	structure/union member access through pointer (arrow)
(type)	Typecasting operator
[]	Array subscript operator.

operator	Description
()	function call operator
.*	Pointer-to-member operator
->*	pointer-to-member operator through object pointer

operator	Description
()	function call operator
.*	Pointer-to-member operator
->*	pointer-to-member operator through object pointer

Assignment operator

→ Assignment operator takes 2 arguments.

→ The left side argument must be a variable

→ The right side argument can be a constant or an expression or a variable

→ Left side argument is also called L-value.

→ The job of assignment operator is to assign right side value to left side variable

→ After assignment the entire expression will be replacement with the resultant.

Examples:

1. a = 5+2;

a → variable → left side var

5+2 → value → right side value

a = 7;

a
[7]

2. $a = 5+2-3;$
$a = 7-3;$
$a = 4;$

3. $a = 5+\underline{2 * 3};$
$a = 5+6;$
$a = 11;$

4. $a = (5+2) * 3;$
$a = 7 * 3;$
$a = 21;$

→ ( ) is not operator, helps to change priority.

→ Whenever we notice, multiple operators in a single statement in order to know which operator will execute first, we go to priority.

→ If multiple operators are having same priority in order to know which operator will execute first we go to order of presidence, order of associativity, order of evaluation. (Left to right).

→ paranthesis don't have priority it helps to change the priority.

→ In python, () acts as operator (grouping operator).

int – int → int

int – float → float

float – int → float

float – float → float

① $a = \frac{-2.0}{5}$ fl $= -0.4$
int

② $a = \frac{-2}{5.0} = -0.4$

③ $a = \frac{-2.0}{-5.0} = 0.4$

④ $a = \frac{-2}{5} = 0$

⑤ $a = -5/2 = -2$

⑥. $a = 5/2.0 = 2.5$

⑦. $a = 5.0/2 = 2.5$

⑧. $a = -5/-2 = 2$

⑨. $a = \frac{-2.0}{-5} = 0.4$

⑩. $a = -5.0/2 = -2.5$

Arithmetic operator

→ Arithmetic operators are used to perform basic mathematical operations on variables or constants.

→ These operators allow you to carry out tasks like addition, subtraction, multiplication, division and more

Maths	C-lang
+ → sum	+ → addition
- → difference	- → subtraction
* → product	* → multiplication
÷ → quotient	/ → division
÷ → Reminder	% → modulus.

① In C,

```
int sum = 5+3;
int diff = 10-7;
int product = 4*6;
int quotient = 20/4;
int reminder = 17%3;
```

② $a = 5 + 2 * 3 / 2 - 1;$

$a = 5 + 6 / 2 - 1;$

$a = 5 + 3 - 1;$

$a = 8 - 1;$

$a = 7;$ based on priority

divisor → 7 | 36 → divident, 35 | 1 → reminder; 5 → quotient

③. $a = 10 + 3 * 4 / 2 - 2;$

$a = 10 + 12/2 - 2;$

$a = 10 + 6 - 2;$

$a = 16 - 2;$

$a = 14;$

'C'-tokens

→ In C, tokens are the smallest elements or building blocks of a C program.

→ The compiler breaks the source code into these tokens to understand the process of the program.

→ Every statement or expression in a C-program made up of tokens.

→ Every individual meaningful unit is called C-token

→ The main purpose of tokens is to provide structure to the source code.

→ The compiler uses these tokens to interpret the programmer's intent and generate the appropriate machine code.

→ C-tokens can be

Keywords - void, int, if, else, return

Identifiers - main, printf

Operators - +, -, **, &&, ==

Seperators - ; {}, [ ]

Constants - "hello", "Hi!"

In, C,

```
#include <stdio.h>
int main() {
    int a = 5, b = 10;
    int sum = a+b;
    printf("sum is: %d\n", sum);
    return 0;
}
```

→ In this code tokens are,

Keywords: int, return

Identifiers: main, a, b, printf, sum

Constants : 5, 10, 0

Operators : =, +, , , ;

Seperators : (), {}, ;

Strings : "sum is : %d\n"

printf

→ printf will print first argument as it on the screen except for the format specifier.

→ If it is a format specifier, printf will try to map with next respective argument and if it is correct match printf will print the value and if it is not a correct match and nothing to map with. printf will print garbage value

→ First argument must be always enclosed within a pair of double quotes " "

→ If there are multiple arguments, they must be sperated with ' , ' comma.

→ printf is a function call.

→ printf cannot convert the data, it can only format the data.

int → %d [format specifier]
char → %c
float → %f

Examples:

printf ("Hello"); → Hello

printf ("%d %d %d", 10, 20, 30); → 10 20 30

printf ("%d,%d,%d", 10, 20, 30); → 10, 20, 30

printf ("%d-%d-%d", 10, 20, 30); → 10-20-30

printf ("%d %c %f", 10, 'a', 5.5); → 10 a 5.5

printf ("%d %d %d", 10, 20); → 10 20 G.V [Garbage value]

printf ("%d %d %d", 10); → 10 G.V G.V

printf ("%d %d %d"); → G.V G.V G.V

printf ("%d", 5.5); → G.V

printf ("%d", 5/2); → 2

printf ("%d", 10/2); → 5

printf ("%f", 5.0%2); ⟶ compilation error : illegal use of floating point.

printf ("%f", 5/2); ⟶ G.V

printf ("%d", 5+2, 5, 10); ⟶ 7

printf ("%d + %d", 5, 2); ⟶ 5+2

printf ("%d + %d = %d", 5, 2); ⟶ 5+2 = G.V

printf ("%d + %d = %d"); ⟶ G.V + G.V = G.V

printf ("%f", 5%2); ⟶ G.V

printf ("%d %d %d", "%d %d", "%d", 1, 2, 3);

G.V G.V 1

printf ("5+2 = 5", 1, 2); ⟶ 5+2 = 5

printf ("5 + %d = 5"); ⟶ 5 + G.V = 5

printf ("5 + %d = 5", 5); ⟶ 5 + 5 = 5

printf ("hello", "people"); ⟶ hello

ts = 5000

printf (ts); ⟶ CE : Exp syntax

printf ("ts"); ⟶ ts

printf ("%d", ts); ⟶ 5000

printf ("ts = %d"); ⟶ ts = G.V

printf ("ts = %d + 1000", ts); ts = 5000 + 1000

printf ("ts = %d", ts+1000); ⟶ ts = 6000.

→ Compilation takes place from top to bottom
→ Execution takes place from main function.

int → 2 bytes
char → 1 byte
float → 4 bytes

① . void main() {

int a; → definition → allocate memory.

a = 5;

printf("%d", a);

} o/p: 5

→ Data types gives the size information to compiler.

② . void main()

{

float f; → definition.

f = 5.5; → variable (left side)
→ value (right side)

printf("%f", f);
mapping

}

o/p: 5.5

③ . void main()

{

char ch;

ch = 'a';

printf("%c", ch);

}

o/p: a.

(4)
```
void main()
{
    float a;
    a = 5/2;
    printf("%f", a);
}
```

int - int → int

o/p = 2.0

(5).
```
void main()
{
    int a;
    a = 5.0/2;
    printf("%d", a);
}
```

o/p = 2

(6).
```
void main()
{
    char ch;
    ch = 5.0%2;
    printf("%c", ch);
}
```

Compilation Error: illegal use of floating point.

→ In C-language, we can assign any type of data to any type of variable, internally compiler will take care.

→ In C and C++, we can assign any type of data to any type of variable, so they are called as loosely typed Language.

→ In Java we cannot assign any type of data to any type of variable, so Java is also called as strictly typed Language.

→ In Python, we can assign any type of data to any type of variable, so ~~java~~ python is no typed language (or) loosely typed language.

→ C, C++, Java are also called statically typed language, because int a; → definition is only statement is responsible to allocate memory.

→ Python is also called as dynamically typed language.

Scanf

→ In C, scanf function is used to take input from the user.

→ It reads formatted data from the standard input (typically the keyboard) and stores the value inthe variable provided.

→ It allows users to input various types of data such as integers, floating point numbers, characters, and strings during the execution of program.

Syntax:

```
scanf ("format-specifier", &variable);
```

format-specifier: It defines the type of data that will be input.

&variable: The address-of operator & is used before the variable to store the input data in the appropriate memory location.

Common format specifiers:

%d → Integer

%f → Float

%c → Character

%s → String

%lf → Double

%u → Unsigned Integer.

Examples:

① .
```
void main()
{
    int a;
    scanf("%d", &a);
    printf("%d", a);
}
```

O/P: 10 → input from user
10 → output.

② .
```
void main()
{
    int b;
    scanf("%d", &b);
    printf("%d", b);
}
```

O/P: 20
20

③ .
```
void main()
{
    int a;
    int b;
    char ch;
    float f;
    scanf("%d", &a);
    scanf("%d", &b);
```

```
    scanf("%c", &ch);
    scanf("%f", &f);
    printf("%d %d %c %f", a, b, ch, f);
}
```

O/P: 10 → a
20 → b
a → ch
10.5 → f
} → input from user.

1020a10.5 } → output

Variable - syntax:

Definition vs Decleration:

→ Definition handles with memory whereas decleration don't handle with memory.

→ The purpose of data type is to tell the size information to the compiler.

① int a;
int b; ✓
int c;

② int a, b, c; ✓

③ int a1, b1, c1; ✓

④ int 1a, 1b, 1c; ✗

⑤ int _a, _b, _c; ✓

⑥ int a_, c_; ✓

⑦ int a_b, b_c; ✓

⑧ int _; ✓

⑨ int a, a; ✗ CE

(10). int a, A; ✓

(11) int a=a; ✓

(12) int a=b=c=10; ✗

(13) int a, b, c=a=b=10; ✓

10

a=10, b=10, c=10

(14) int a,b,c;

a = b = c=10; ✓

order of associativity.

(15) int if; ✗

(16). int a+b; ✗

(17) int 1a_, 2a_; ✗

(18) int 10; ✗

(19). float 3.5; ✗

(20) char 'a', 'b'; ✗

(21) int char int; int char; ✓

(22). int INT, CHAR; ✓

(23) float -1, -2; ✓

(24) int abcdefghilmno12345; ✓

(25) int hello mum; ✗

(26) int hello_world; ✓

(27) int hellobaby; ✓

(28) int a=27/10/2023; ✓

Implicit Typecasting

→ Implicit typecasting in C, also known as automatic type conversion, when the compiler automatically converts one data to another without the programmer's explicit instruction.

→ This usually happens when performing operations between different types.

→ The compiler converts the smaller data type to larger data type.

→ Implicit typecasting simplifies operations between variables of different data types.

→ The compiler ensures that smaller data types are converted to larger data types.

Examples:

①
```
void main() {
    int a = 5;
    float result;
    result = a;
    printf("%f", result); // 5.0000
    printf("%d", a);      // 5

}
```

a: int → result: float

result = a

② .
```
void main()
{
  float a = 7.5;
  double result;
  result = a;
  printf("%.lf", result); // 7.500000
}
```

③ .
```
void main()
{
  int a = 95;
  char letter;
  letter = a;
  printf("%c", letter);
}
```

```
void main()
{
  ~~int~~ char letter = 'A';
  int result;
  result = letter;
  printf("%d", result);
}
```

O/P : 65

Ascii value of A

④ .
```
~~int~~ void main()
{
  int a = 5;
  float b = 7.5;
  float sum = a+b;
  printf("%f", sum);
}
```

O/P: 12.5

a	b
5	7.5

sum

5.0 + 7.5 = 12.5

$\frac{\text{int}}{\text{float}}$ = float.

problems:

1. Write a code to print your name, roll number, CGPA, gender, date of birth.

```
#include <stdio.h>
int main()
{
  char name[] = "Harish";
  int rollno = 12;
  float cgpa = 7.5;
  char gender = 'M';
  char dateOfBirth[] = "24/9/2000";
  printf ("%s\n", name);
  printf ("%d\n", rollno);
  printf ("%f\n", cgpa);
  printf ("%c\n", gender);
  printf ("%s", dateOfBirth);
  return 0;
}
```

a collection of characters will be a string (word).

"\n" → to get a newline

O/P: Harish
12
7.5
M
24/9/2000.

---

2. Given radius in decimal point, write a code to find diameter, are and circumference of the circle.

sample input1: 2.3 [radius].

sample input 2: 6

```
#include<stdio.h>  #define PI 3.14159
```

↳ we can use PI instead of π.

```
int main()
{
  float radius;
  scanf("%f", &radius);
  float diameter = 2 * radius;
  float area = PI * radius * radius;
  float circumference = 2 * PI * radius;
  printf("Diameter = %f\n", diameter);
  printf("Area = %f\n", area);
  printf("Circumference = %f\n", circumference);
  return 0;
}
```

O/P: 2.3

Diameter = 4.600000

Area = 16.619025

Circumference = 14.451326.

6

Diameter = 12.000000

Area = 113.097335

Circumference = 37.699111

(3) Given with temperature 'x' in Celsius the challange is to convert the given temperature to Fahrenheit and display it:

```
#include<stdio.h>
int main()
{
 float celsius;
 scanf("%f", &celsius);
 float fahrenheit = (celsius * 9/5) + 32;
 printf("%f", fahrenheit);
 return 0;
}
```

O/P:

```
100.00      //celsius
212.000000  //fahrenheit
-40.        //celsius
-40.000000  //fahrenhiet
```

---

(4). Write a code to convert specified days into years, weeks and days.

```
#include<stdio.h>
int main()
{
  int days, years, weeks, remaining_days;
  scanf("Number of days: %d", &days);

  years = days / 365;
  remaining_days = days % 365;
  weeks = remaining_days / 7;
  remaining_days = remaining_days % 7;
```

365 | 1329 → 3

```
return 0;
}
```

O/P:

number of days = 1329 //input

3 //years

33 // weeks

3 // days

---

(5). Given a 4 digit number, split that into two half, manage them such that the left half and right half of the number get swapped.

```
#include<stdio.h>
int main()
{
  int num, leftHalf, rightHalf, swappedNum;
  scanf("%d", &num);
  leftHalf = num/100; //first two digits
  rightHalf = num%100; //last two digits
  swappedNum = (rightHalf *100)+leftHalf; //swap and merge two halves
  printf("%d", swappedNum);
  return 0;
}
```

O/P:

1234 //num

3412 //swapped and merged

Daily Task:

① Write a code that accepts length and breath of the rectangle the task is to print area and perimeter of the rectangle:

```
#include<stdio.h>
int main()
{
  int length, breath;
  scanf("%d %d", &length, &breath);
  int area = length * breath;
  int perimeter = 2.*(length+breath);
  printf("Area: %d", length);
  printf("Perimeter: %d", breath);
  return 0;
}
```

O/P: 5 2 // length and breath
Area : 10
Perimeter: 14

---

② Write a code that converts the given inches to centimeter;

```
#include<stdio.h>
int main()
{
  int inch;
  scanf("%d", &inch);
  float cm = inch * 2.54;
  printf("%f", cm);
  return 0;
}
```

O/P:
8 //inch
20.320000 //cm.

③. Write a code that takes number of seconds as the input and converts to hh:mm:ss format:

```
#include<stdio.h>
int main()
{
  int total_seconds;
  scanf("%d", &total_seconds);
  int hour, minutes, seconds;
  hours = total_seconds / 3600; // 14325/3600
  minutes = (total_seconds % 3600) /60; //(14325 % 3600) /60

  seconds = total_seconds % 60; // 14325 % 60.

  ~~printf("%d:~~
  printf("%02d : %02d : %02d", hours, minutes, seconds);
  return 0;
}
```

14325 // total_seconds.

O/P: 03:58:45 // h:m:s

---

④. Given n1, n2 where n2 is larger than n1, your task is to find the next multiple of n1 which is greater than n2.

```
#include<stdio.h>
#defind MAX_SIZE 100
int main()
{
  int n1, n2;
  scanf("%d %d", &n1, &n2);
```

```
for(int i=1; i<MAX-SIZE; i++)
{ n1 = n1 * i;
  if(n1>n2)
  { printf("%d", n1);
    break;
  }
}
return 0;
}
```

O/P: 4 2~~4~~ //n1, n2

24. //next greater of n1

~~5~~ 21 // n1 n2

30 // next greater of n1

Relational operator

Maths	C-Language
$>$	$>$
$<$	$<$
$\geq$	$>=$
$\leq$	$<=$
$=$	$==$
$\neq$	$!=$

→ Relational operators will help to establish relation between exactly $\boxed{\text{two arguments.}}$

→ The resultant of relational operators will be true (T) or False(F), but in C-language we don't have boolean data type, so we represent T=1 & F=0

Boolean datatype:

	C	C++	Java	Python
5>2	1	true	true	True
5<0	0	false	false	False

→ The resultant of relational operators in 'c' will always be 0 (or) 1.

Examples:

5>1 → 1

5<2 → 0

5>=5 → 1

5>=2 → 1

5>=6 → 0

5<=2 → 0

5<=6 → 1

5<=10 → 1

5!=5 → 0

5==5 → 1

5=5 → C.E : L-value req.

5==10 → 0

8.5<8.2 → 0

-9>-6 → 0

-3.75>-3.5 → 0

5==5.0 → 1

5!=5 → 0

5!=4 → 1

5 == 4 ⟶ 0

5 == 5.5 ⟶ 0

① .

```
void main()
{
  int a;
  a = 4 > 3 > 2;
  printf("%d", a);
}
```

1 > 2 => 0

O/P: 0 [4>3>2]

Maths	C-language
a>b	4>3 3>2
b>c	4>3>2
a>c	↓
a>b>c	logical error
4>3 3>2	[relational operator take a relation between two arguments only].
4>2 True	
4>3>2 True	

② . a = 4 + 3 > 2 + 7;

a = 7 > 2 + 7;

a = 7 > 9;

a = 0

③ . a = b > 3 + 5 < 10;

a = 6 > 8 < 10;

0 < 10

a = 1

④. void main()

```
{
  int a;
  a = 4>3>2;
  printf("%d", a);
}
```

$1>2 \rightarrow 0$

O/P: 0

→ If we try to establish relational more than two argument using relational operators we get logical error.

→ To avoid this logical error or to establish relational between more than two argument we go to logical operators.

⑤. a = 2+3>4>1+5;

a = 5>4>6;

a = 1>6;

a = 0;

⑥. a = 6>3+2>5;

a = 6>5>5;

a = 1>5

a = 0;

⑦. a = 5+3*5 >= 25 % 2 < = 8 != 5 - 3 * 2 >= 8 + 3 != 5;

a = 5+15 >= 1 < = 8 != 5-6 >= 8 + 3 != 5;

a = 20 >= 1 < = 8 != -1 >= 11 != 5;

a = 1 < = 8 != -1 >= 11 != 5;

a = 1 != -1 >= 11 != 5;

a = 1 != 0 != 5;

a = 1 != 5;

O/P: 1

a = 1

8. void main()

```
{
    int a=5;
    a=5>1<2;
    printf("%d", a);
}
```

5>1<2
1<2
3 arguments

o/p: logical error.

Logical operator

1. Logical AND (&&)
2. Logical OR (||)
3. Logical NOT (!)

a1	a2	&& AND	|| OR	!a1	!a2
zero	zero	0	0	1	1
zero	non-zero	0	1	1	0
non zero	non zero	1	1	0	0
non zero	zero	0	1	0	1

Examples

!10 = 0 [!non-zero = 0]

!0 = 1 [!zero = 1]

!0 = 1

&&:

0 && 1 ⇒ 0

0 && 0 ⇒ 0

1 && 0 ⇒ 0

1 && 1 ⇒ 1

Examples:

0 && 5 ⇒ 0

0 && 0 ⇒ 0

10 && 0 ⇒ 0

5 && 10 ⇒ 1

-5 && -10 ⇒ 1

10 &&
&& 5
&&

C.E
Expression
syntax

||:

1 || 0 ⇒ 1

0 || 1 ⇒ 1

1 || 1 ⇒ 1

0 || 0 ⇒ 0

Examples:

5 || 0 ⇒ 1

0 || 10 ⇒ 1

5 || 5 ⇒ 1

0 || 0 ⇒ 0

!:

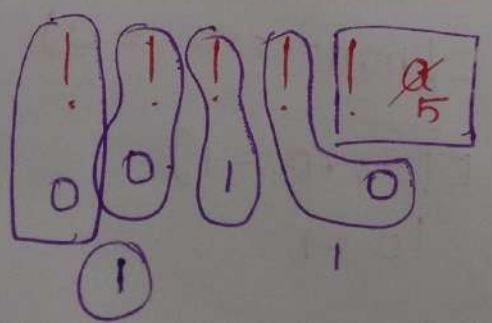
!5 ⇒ 0

!0 ⇒ 1

① . void main()

```
{
    int a=5;
    !!!!!a;
    printf("%d", a);
}
```

O/P: 5



if a = !!!!!a;

O/P = 0

② . void main()

```
{ int a;
  a = 4>3 && 3>2;
  printf("%d", a);
}
```

O|P: 1

③ . a = 5>2 && 1 + 5*2 || 0;

a = 5>2 && 1 + 10 || 0;

a = 5>2 && 11 || 0;

a = 1 && 11 || 0;

a = 1 || 0;

a = 1.

④ . void main ()

```
{
  int a;
  a = 5%5==5 || 10==1 || 5>10;
```

a = 0 || 0 || 0;

a = 0 || 0;

a = 0.

⑤ . a = 5%5==1 && 5==5.5 || 5!=5.0 || 5.0!=5.0;

a = 1 && 0 || 0 || 0;

a = 0 || 0 || 0;

a = 0 || 0;

a = 0.

if - else

→ if-else statements in C is used to perform conditional branching in code.

→ if-else statements enable your program to make decisions.

→ If our condition is true, $\boxed{\text{if-body}}$ will execute. if false, $\boxed{\text{else-body}}$ will execute.

syntax:

```
if (condition)
{
    // code to execute if condition is true
}
else {
    // code to execute if condition is false
}
```

```
① if (condition)
{
  ② if body
}
else {
  ③ else body
}
④
```

T : ① ② ④

F : ① ③ ④

```
void main()
{
printf("A");
printf("B");

if (3>2)
{
printf("C");
printf("D");
}
printf("E");
printf("F");
}
```

① T: ABCDEF

F: ABEF

② if(3>2) $\boxed{\{\}}$ if not there

→ whenever programmer is not providing {} to if conditional constant, compiler will consider upto 1st ; as if's body.

T: ABCDEF

F: ABDEF

→ In c-language every non-zero treated as true ⇒ 1

→ In c-language every zero is treated as false ⇒ 0.

Examples:

if(3>2) → 1

if(3<2) → 0

if(3) → 1

if(0) → 0

if(3+2) → 1
5

if(!2+3) → 1
0+3
3

if(!(2+3)) → 0
!5

if(5*0) → 0
0

if( ) → error
↳ condition mandatory

if(non-zero) → 1
if(zero) → 0

if → conditional construct.

Note: else keyword must be written immediately after if body. otherwise we will get an error telling else without previous if.

Example problems

①. Write a program to find maximum of two integers using if conditional construct.

```
void main()
{
    int a, b, max;
    a = 5; b=10;
    if (a>b);
      max = a;
    if (b>a);
      max = b;
    printf("%d", max);
}
```

O/P: 10 //max = b

a b

5 10

max (if a>b)

5

max (if b>a)

10

②. Write a program to find maximum of two integers using one if conditional construct:

```
void main()
{
    int a,b, max;
    a = 5; b=10;
    if (a>b)
    max = a;
```

a b

5 10

max

10

b

```
else {
    max = b;
}
printf("%d", max);
}
```

o/p: 10 // max = b.

another way:

```
void main()
{
    int a, b, max;
    max = b;
    if (a>b)
     max = a;
    printf("%d", max);
}
```

o/p: b or 10 (if declared).

a , b, max

if (a>b)

max

a

③. Write a program to find maximum of two integers using only one if conditional construct and without using max variable (temp variable):

```
void main()
{
    int a = 5, b = 10;
    if (a>b)
    {
      b = a; }
      printf("%d", b);
    }
}
```

o/p: 10.

(4). Write a program to find maximum of two integers using only one if conditional construct and without using max variable (temp variable) and without changing the original value of a and b

```
void main()
{
  int a, b;
   a = 5; b = 10;
  if (a > b)
  {
    printf("%d", a);
  exit(0);
  }
  printf("%d", b);
}
```

→ Whenever the compiler ~~com~~ encounters the keyword exit, compiler will take the control outside the program.

→ Whenever compiler encounters return keyword the control will come out of the current function.

→ This is the reason exit is also called as program terminator and return is also called as function terminator.

(5). Write a program to find maximum of two integers without using if conditional construct.

$a > b * a + b > a * b$

```
void main()
{
  int a, b, max;
  a = 5; b = 10;
  max = (a>b) * a + (b>a) * b;
  printf ("%d", max);
}
```

---

(6). Write a program to find maximum of two integers without using comparison operation.

relational & logical

```
void main()
{
  int a, b, max;
  a = 5, b = 10;
  max = (a+b + abs (a-b))/2;
  printf ("%d", max);
}
```

$a = -9, \quad b = -5$

$max = (-9 - 5 + abs(-9+5))/2$

$max = (-9 - 5 + 4)/2$

$max = -10/2$

$= -5.$

Flow control - conditinal operator

→ Conditional operator in c, also known as the ternary operator, a way to evaluate a condition and return one of two values based on whether condition is true or false

syntax:

```
condition ? expression1 : expression2;
```

condition: An expression that evaluates to true (nonzero) or false (zero).

expression 1: Executed if the condition is true

expression 2: Executed if the condition is false

Note:

→ The primary purpose of the conditional operator is to shorten simple if-else statements into a single line.

→ It is useful when you want to assign a value or execute a small operation based on the condition without writing full if-else blocks.

syntax:

```
arg1 ? arg2 : arg3;   → replace with arg2 (T)
arg1 ? arg2 : arg3;   → replace with arg3 (F)
```

Examples:

1. $a = 5 > 3 \; ? \; 5 : 3; \longrightarrow 5$ (T)

2. $a = 5 < 3 \; ? \; 5 : 3; \longrightarrow 3$ (F)

3. $a = 5 \; ? \; 6 : 0; \longrightarrow 6$ (T)

4. $a = 0 \; ? \; 5 : 6; \longrightarrow 6$ (F)

5. $a = -5 \; ? \; -10 : -3; \longrightarrow -10$ (T)

6. $a = 5.5 \; ? \; 6.3 : 5.4; \longrightarrow 6.3$ (T)

7. $a = 0.75 \; ? \; 0.5 : 5.5; \longrightarrow 0.5$ (T)

8. $a = 0.0 \; ? \; 0.5 : 5.5; \longrightarrow 5.5$ (F)

9. $a = 5 \; ? \; 5 : 6; \longrightarrow$ C.E → Expression syntax

10. $a = 5 \; : \; ? \; ;$

12. $a = : \; 5 \; ? \; ;$

13. $a = \; : \; ? \; ;$

→ C.E → Expression syntax

11. Write a program to find maximum of two integers using conditional operators:

2 int

⇒ Max = a>b ? a : b;

3 int

⇒ Max = a>b && a>c ? a :
b>c ? b :
c;

4int

```
⟹ Max = a>b && a>b && a>d ? a :
        b>c && b>d ? b :
            c>d ? c :
                d ;
```

```
a = 4>3 ? 6>8 ? 10 : 3>8 ? 4>1 ? 20 : 30 :
    4>7 ? 40 : 50 : 6>8 ? 60 : 70;
```

Step 1: check if number of '?' is equal to no of ':' if yes it is valid if not we get compilation error telling Expression syntax

Step 2: Identify the ':' and map th ':' with Left nearest '?'

Step 3: Depending on argument 1 if condition is true the entire expression will be replaced with the second argument (from ? till mapped colon).

→ if condition is false then the entire expression will be replaced with third argument (from : till ;)

```
a = 6>8 ? 10 : 3>8 ? 4>1 ? 20 : 30 : 4>7 ?
                                  40 : 50;
```

a = 3>8 ? 4>1 ? 20 : 30 : 4>7 ? 40 : 50;

a = 4>7 ? 40 : 50;

$\boxed{a = 50;}$

① .
```
if (s1)
{
  a = s2;
}
else
{
  a = s3;
}
```

$\boxed{a = s1 ? a2 : a3;}$

② .
```
if (s1)
{
  a = s1;
}
else
{
  if (s3)
    a = s4;
  else
    a = s5;
}
```

$\boxed{a = s1 ? s2 : s3 ? s4 : s5;}$

③ -
```
if (s1)
{
  if (s2)
  a = s3;
  else
  a = s4;
}
else
{
  a = s5;
}
```

$\boxed{a = s1 ? s2 ? s3 : s4 : s5;}$

→ We can convert every conditional operator statement to if-else conditional construct but vice versa is not true always.

→ We can convert every if-else conditional construct to conditional operator only & only if, if and else body is having exactly one statement.

Examples:

① Find the maximum of two numbers:

```
#include <stdio.h>
int main()
{
    int a = 10, b = 20;
    int max = (a>b) ? a : b;
    printf("max is: %d", max);
    return 0;
}
```

O/P: max is 20.

② Check if a number is Even or odd:

```
#include<stdio.h>
void main()
{
    int num = 15;
    char result = (num % 2 == 0) ? "Even" : "odd";
    printf("num is: %s", result);
}
```

O/P: num is odd

③. Check if a num is Positive or negative:

```
#include<stdio.h>
void main()
{
  int num = -10;
  char result = (num >= 0) ? "positive" : "negative";
  printf("num is: %s", result);
}
```

O/P: num is Negative

④. Find the Absolute Value of a Number:

```
#include<stdio.h>
void main()
{
  int num = -5;
  int absolute = (num < 0) ? -num; num;
  printf("the absolute value is: %d", absolute);
}
```

$-(-5) = 5$

O/P: The absolute value is: 5

⑤. Check for a Passing Grade:

```
#include<stdio.h>
void main()
{
  int grade = 75;
  char result = (grade >= 50) ? "Pass" : "Fail";
  printf("The grade is = %s", result);
}
```

O/P: The grade is Pass.

Loops

① Write a program to print 11111

```
printf("%d", 1)
printf("%d", 1)
printf("%d", 1)
printf("%d", 1)
printf("%d", 1)
```

Olp: 11111

→ Whenever we notice a piece of code is repeated continously inorder to avoid code continous repeatition we go to a concept called loops

→ Whenever we notice a peice of code, that is loop itself is repeated in order to avoid loop continous repeatition we go to a concept called nested loops.

→ Whenever we notice a piece of code is repeated randomly in order to avoid random repeatition we go to a concept called function.

→ Whenever we notice a function itself is repeated in order to avoid function repeatition we go to a concept called Recursion

→ Whenever we notice on recursion we get run time error telling recursion stack overflow or to reduce the no of function calls or to reduce the no of stack frames or activation records or to optimize recursion we go to a concept called dynamic program (dp).

dynamic Programming	= recursion + memoization

Loops:

1. for
2. while
3. do-while

① while loop:

Syntax:

```
while (condition)
{
  while body
}
```

Examples:

① a = 1;

```
while (a <= 5)
{
    while body;
    a = a - -1;
}
```

a: 5 4 3 2 1 0 -1 -2 -3

o/p: ∞ times

② a = 1;

```
while (a <= 5)
{
    printf("%d", 1);
    a = a + 1;
}
```

6 <= 5 F
5 <= 5
4 <= 5
3 <= 5
2 <= 5
1 <= 5

a: 1 2 3 4 5 6

come out (stopped running loop)

o/p: 11111

③ a = 5;

```
while (a <= 1)
{
    a = a - 1;
}
```

5 <= 1 False

a: 5 4 3

o/p: 0 times (won't get inside while loop even).

④ a = 5;

```
while (a >= 1)
{
    printf("%d", 1);
    a = a - 1;
}
```

0 >= 1 False
1 >= 1
2 >= 1
3 >= 1
4 >= 1
5 >= 1

a: 5 4 3 2 1

o/p: 11111

```
printf("%d", 1)
printf("%d", 2)
printf("%d", 3)
printf("%d", 4)
printf("%d", 5)
```

⑤.

```
a = 5;
while (a >= 1)
{
  printf("%d", 6 - a);
  a = a - 1;
}
```

0 >= 1 F
1 >= 1
2 >= 1
3 >= 1
4 >= 1

6 - 5 = 1
6 - 4 = 2
6 - 3 = 3
6 - 2 = 4
6 - 1 = 5

a: ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0

o/p: 12345

⑥.

```
a = 1;
while (a <= 5)
{
  printf("%d", a);
  a = a + 1;
}
```

1 <= 5
2 <= 5
3 <= 5
4 <= 5
5 <= 5
6 <= 5 F

a: ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6

o/p: 12345

(7).

```
a = 10;
while (a >= 1)
{
  printf("%d", 6 - a/2);
  a = a - 2;
}
```

a: 10 → 8 → 6 → 4 → 2 → 0

10 >= 1
8 >= 1
6 >= 1
4 >= 1
2 >= 1
0 >= 1 F

6 - 10/2
6 - 5 = 1
6 - 8/2 ⇒ 6 - 4 = 2
6 - 6/2 ⇒ 6 - 3 = 3
6 - 4/2 ⇒ 6 - 2 = 4
6 - 2/2 ⇒ 6 - 1 = 5

O/P = 12345

---

(8).

```
a = 1;
while (a <= 5)
{
  if (a % 2 == 0)
    printf("0");
  else
    printf("1");
  a = a+1;
}
```

a: 1 → 2 → 3 → 4 → 5 → 6

1 % 2 = 0

1 <= 5
2 <= 5
3 <= 5
4 <= 5
5 <= 5
6 <= 5 F

O/P: 10101.

---

(9).

```
a = 1;
while (a <= 5)
{
  printf("%d", a % 2);
  a = a+1;
}
```

O/P: 10101

a % 2 → gives reminder

① Write a program to print 12345 (5 times)

O/P: 1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5

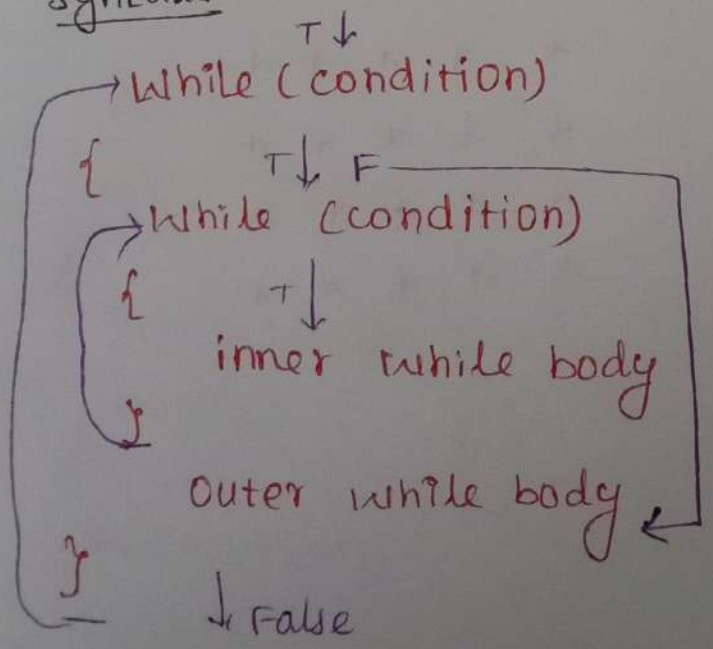
```
a=1;
while (a<=5)
{
 printf("%d", a);
 a=a+1;
}
```

5 times repeat to get above output

Nested While loop:

→ Whenever we notice a piece of code, that is loop itself is repeated in order to avoid loop continous repeatition, we go to a concept called Nested loop.

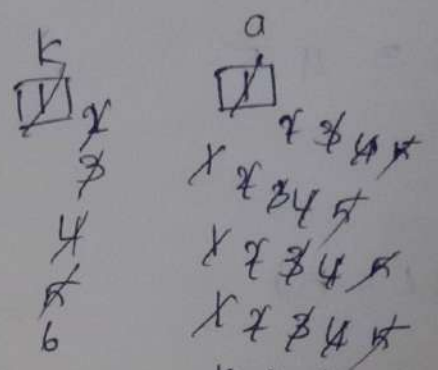
Syntaxs



k=1; → no of lines or rows

```
while (k<=5)
{
    a=1; → no of elements or columns
while (a<=5)
{
  printf("%d", a);
  a=a+1;
}

  printf("\n");
  k=k+1;
}
```



O/P:
```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

---

(2).
```
k=1;
while(k<=5)
{
  a=1;
while (a<=5)
{
  printf("*");
  a=a+1;
}

printf("\n");
k=k+1;
}
```

O/P:
```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

③.

```
k=1;
while (k<=5)
{
  a=5;
  while (a<=5)
  {
    printf("%d", b-a);
    a=a-1;
  }
  printf("\n");
  k=k+1;
}
```

a<=5

a>=1

b-5
b-4
b-3
b-2
b-1

k	a
1	5
2	5 4 3 2 1
3	5 4 3 2 1
4	5 4 3 2 1
5	5 4 3 2 1
6	5 4 3 2 1

O/P:

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

④.

```
k=1;
while(k<=5)
{
  a=1;
  while(a>=5)
  {
    printf("%d", b-a);
    a=a+1;
  }
  printf("\n");
  k=k+1;
}
```

b-1
b-2
b-3
b-4
b-5

k	a
1	1 2 3 4 5
2	1 2 3 4 5
3	1 2 3 4 5
4	1 2 3 4 5
5	1 2 3 4 5
6	

O/P:

```
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
```

1. printf("%d", 100); ⟶ 100
2. printf("%d", 100.5); ⟶ G.V
3. printf("%d", 'a'); ⟶ 97
4. printf("%d", 97); ⟶ 97
5. printf("%d", 97.5); ⟶ G.V

→ Every [integers] and [characters] ~~vato~~ will be internally stored in binary format.

→ Every [float] data will be internally stored in IEEE754 format

%d %i %x %X %o → int
%c → char
%f → float

int, char → Binary
float → IEEE754

%d %i %c ⟶ binary ⟶ value
%d %i %c ⟶ IEEE754 ⟶ G.V
%f ⟶ binary ⟶ G.V
%d ⟶ IEEE754 ⟶ value

1. printf("%d", 97); ⟶ 97

2. printf("%d", 'a'); ⟶ 97

3. printf("%d", 97.5); ⟶ G.V

4. printf("%c", 97); ⟶ a

5. printf("%c", a); ⟶ a

6. printf("%c", 97.5); ⟶ G.V

7. printf("%f", 97); ⟶ G.V

8. printf("%f", 'a'); ⟶ G.V

9. printf("%f", 97.5); ⟶ 97.5.

Example Problems:

①
```
k=1;
while (k<=5)
{

    a=1;
  while(a<=5)
  {
    printf("%c", 64+a);
    a=a+1;
  }
  printf("\n");
  k=k+1;
}
```

O/P:

A	B	C	D	E
A	B	C	D	E
A	B	C	D	E
A	B	C	D	E
A	B	C	D	E

② .

```
k=1
while(k<=5)
{ a=1;
  while(a<=k)
  { printf("%d", a);
    a=a+1;
  }
  printf("\n");
  k=k+1;
}
```

O/P:

1				
1	2			
1	2	3		
1	2	3	4	
1	2	3	4	5

③ .

```
k=1;
while(k<=5)
{
    a=1;
  while(a<=5)
{
  printf("%d", k);
  a=a+1;
}
printf("\n");
k=k+1;
}
```

O/P:

1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4
5	5	5	5	5

④ :

```
k=1;
while(k<=5)
{
  a=5;
  while(a>=1)
{
  printf("%d", 6-k);
  a=a-1;
}
  printf("\n"); k=k+1; }
```

O/P:

5	5	5	5	5
4	4	4	4	4
3	3	3	3	3
2	2	2	2	2
1	1	1	1	1

⑤.

```
k=5;
while(k>=1)
{
    a=5;
while (a>=k)
{
printf("%d", a);
a=a-1;
}
printf("\n");
k=k-1;
}
```

o/p:

5				
5	4			
5	4	3		
5	4	3	2	
5	4	3	2	1

k: ~~5~~ 4 a: ~~5~~ 4

a → 5 to k
k → 5 to 1
~~printf(a)~~

⑥.

```
k=1;
while(k<=5)
{
    a=5;
    while(a>=k)
{
    printf("%d", a);
    a=a-1;
}
    printf("\n");
    k=k+1;
}
```

o/p:

5	4	3	2	1
5	4	3	2	
5	4	3		
5	4			
5				

k: ~~1~~ 2 a: ~~5~~ ~~4~~ ~~3~~ ~~2~~ 1

k: 1 a: 1 ⇒ k

⑦.

```
k=1
while(k<=5)
{
    a=k;
    while(a>=1)
{
    printf("%d", a);
    a=a-1;
}
    printf("\n");
    k=k+1; }
```

o/p:

1				
2	1			
3	2	1		
4	3	2	1	
5	4	3	2	1

⑧.

```
k=1;
while(k<=5)
{
  a=1;
  while(a<=5)
  {
    if(k==1 || k==5 || a==1 || a==5)
    {  printf("*");
    }
    else {
      printf("  ");
    }
      a=a+1;
  }
  printf("\n");
  k=k+1;
}
```

O/P:

*	*	*	*	*
*				*
*				*
*				*
*	*	*	*	*

a ⟶ 1 to 5

k ⟶ 1 to 5

printf ⟶ *

(9).

```
k=1;
while(k<=5)
{
    a=1;
    while(a<=5)
{
    if(a==k)
{
    printf("*"); }
else {
    printf(" ");
}
    a=a+1;
}
printf("\n");
  k=k+1;
}
```

o/p:

*				
	*			
		*		
			*	
				*

a → 1 to k

k → 1 to 5

---

(10).

```
k=1;
while(k<=5)
{
    a=1;
    while(a<=5)
{
  if(a+k==6)
printf("*");
else
    printf(" ");
a=a+1; }
printf("\n");
  k=k+1;
}
```

o/p:

				*
			*	
		*		
	*			
*				

a → 1 to 5

k → 1 to 5

For Loop

Syntax:

```
for ( initialization ; condition ; re-initialization)
{
    for body
}
```

① . for ( ) → Compilation Error: Expression syntax

{

}

→ Not valid.

Example Problems:

①. Write a program to print 12345 using for loop:

```
void main()
{
    int a;
    for(a=1; a<=5; a++)
    {
        printf("%d", a);
    }
}
```

a = a+1 (or)

a: [5] ✗ ✗ ✗ ✗ 5

O/P: 1 2 3 4 5

②. void main()

```
{
   int a,b;
for (a=1, b=10;  a<=b;  a=a+1, b=b-1)
{
   printf("%d %d \n", a,b);
}
 printf("%d %d \n", a+10, b+10);
}
```

o/p:

```
1   10
2   9
3   8
4   7
5   6
16  15
```

③. void main()

```
{
   int a=1;
for(printf("A"); a<=3; b=b+1)
{
   printf("%d %d\n", a,b);
}
 printf("%d %d \n", a+10, b+10);
}
```

o/p: 1 1 1 2 1 3 ACBCBCB.

④. void main()

```
{
   int a,b;
 for(a=b=1; a<=b; b=b+1)
 {
   printf("%d %d \n", a,b);
 }
 printf("%d %d \n", a+10, b+10);
}
```

1<=5
1<=4
1<=3
1<=2

O/P:

1	1
1	2
1	3
1	4
1	5
⋮	⋮
1	b+1

a b
[1] [~~1~~] ~~1~~ ~~2~~ ~~3~~ ~~4~~ 5

⑤. void main()

```
{
   int a, b;
 for(a=b=1; a; if(a>=0) printf("students"))
 {
   a=b<=3;
   printf("Hi");
 }
   printf("Hello");
}
```

→ conditional construct,
→ not reinitialization

O/P: Expression : syntax

⑥. for (initialization ; condition ; re-initialization)

```
{
   for body;
}
```

```
for (statement1 ; statement2; statement3)
{
   for body;
}
```

⑥. for(; ;)

```
{
  printf("Hi students");
}
```

O/P: Hi students ..... ∞ times.

⑦. while( )

```
{
 printf("Hi guys");
}
```

O/P: Expression syntax.

①. What is the difference between for loop and while loop?

while loop	for loop
whenever, we don't know number of iterations in advance we go to while loop	whenever we know number of iterations in advance we go to for loop.
→ Examples: linked list	→ Example: Array

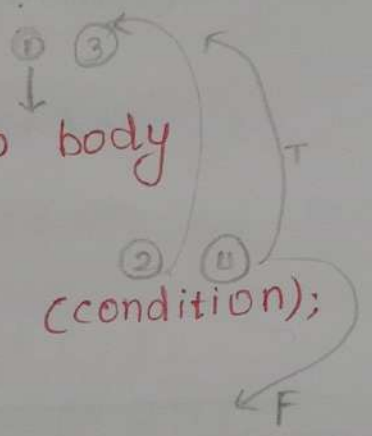
Do-while loop:

Whenever we wanted to execute a piece of code atleast for one time irrespective of the condition we go to Do-while loop.

Do - is keyword.
while - is keyword
Do-while → Not keyword.

syntax:

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



① . Write a program to print 1 2 3 4 5 using do-while loop:

```
void main()
{
    int a=1;
    do
    {
      printf("%d", a);
      a=a+1;
    }
    while (a<=5); }
```

a
1 2 3 4 5 6

o/p:

1	2	3	4	5

```
a=1;
② while (a>2)
  {
    printf("Hello");
  }
```

O/p: NO output

```
a=1;
do{
printf("Hello");
}
while(a>2);
```

O/p: Hello

condition is false → while → 0 time.
do-while → 1 time

③ .
```
void main()
{
  int a;
  a=1;
do
  printf("Hello guys");
while (a<=5)
}
```

O/p: Hello guys..... ∞ times.

④ .
```
void main()
{
  int a;
  a = 1;
do
  printf("Hi");
  a = a+1;
while (a<=5);
}
```

O/p: Compilation Error:
do without while

→ Immediately after end of 'do', while keyword must be present. If not present we get compilation error telling do without while

Break

→ Whenever compiler encounter the keyword 'break' the control will come out of the current loop or switch

→ break keyword can be used only inside loop or switch

① .
```
void main()
{
  int a;
  a=1;
while (a<=5)
{
 printf("%d", a);
if ( a==3)
 break;
 a=a+1;
}}
```

O/P: 1 2 3

→ We cannot use break outside the loop (while, for, do-while) and switch if we use, then we will get compilation error.

②. void main()

```
{
   printf("A");
   printf("B");

 if (3>2)

 {  printf("C");
    break;
   printf("D");
 }
  else
  {
   printf("E");
   printf("F"); }
   printf("G");
}
```

O/p: compilation error : break outside loop or switch

③ · void main()

```
{
  int n =147;
  int flag =0;
for (int i=2; i<=sqrt(n); i++)
{
  if (n%i ==0)
{
    flag =1;
    break;
}
```

0 ==flag

```
if (flag ==0)
printf("Prime number");
else
printf("not a prime number");
}
```

0 ==flag

0 =flag

error

L-value required

Continue

→ whenever we encounter the keyword continue compiler will skip the current iteration and continue with the next iteration.

→ continue can be used only inside loop.

① .
```
void main()
{
  int a=0;
  while (a<10)
  {
    a=a+1;
  if (a>3 && a<8)
  continue;
 printf("%d", a);
}
}
```

a

~~1 2 3 4~~

O/P: 1 2 3 8 9 10

---

② .
```
void main()
{
  int a=1;
  while (a<=10)
  {
   printf("%d", a);
  if (a==4)
   continue;
   a=a+1;
}}
```

O/P:

1 2 3 4 4 4 ... ∞ times

③. void main()

```
{
  printf("A");
 if (3>2)
 printf("B");
  continue;
 a = a+1;
}}
```

o/p: compilation error : continue outside loop.

while (condition)
{
continue;
⋮
}
	for(; ;)
	{
	continue;
	⋮
	}
		do
		{ continue;
		⋮
		}
		while (condition);

~~go to~~

goto:

→ goto is a keyword

→ whenever we encounter the keyword goto the control will be taken from 1 place to another place unconditinally.

→ goto label cannot be constant.

→ goto can be used multiple times in a single program.

① void main ()

```
{
  printf("A");
  printf("B");
  goto last; → label
  printf("C");
  printf("D");  → won't be executed.
last;
  printf("E");
  printf("F");
}
→ forward jump.
```

o/p:

A B E F

② void main ()

```
{
  int a =1;
  printf("A");
  start;
  printf("B");
  if(a<=3)
  {
    printf("C");
    a=a+1;
  goto start;
  printf("D");
}
```

→ backward jump

o/p:

A B C B C B C B D

③. Write a program to print from 1 to 100 without using loop:

```
void main()
{
 int a=1;
 label1;
 if(a<=100)
 {
  printf("%d", a);
  a = a+1;
 goto label1;
}}
```

O/P:

1 2 3 4 100

---

④.

```
void main()
{
  printf("A");
  goto end;
  printf("B");
}
```

O/P: compilation Error. label "end".

---

⑤.

```
void main()
{
  printf("A");
  guys :
  printf("B");
}
```

O/P: A B

⑥. void main()

```
{
  printf("A");
  goto here;
  printf("B");
  start;
  printf("C");
  here;
  printf("D");
  goto start;
  printf("E");
}
```

O/P:

ADCDCD.....

---

⑦. void main()

```
{
  printf("A");
  goto 130;
  printf("B");
  130;
  printf("C");
}
```

O/P: compilation error:

Expression syntax

---

⑧. void main()

```
{
  printf("A");
  goto here;
  printf("B");
  here;
  printf("C");
}
```

C.E:

O/P: Expression syntax

Increment & Decrement

→ In C language, increment and decrement are unary operators used to increase or decrease the value of a variable by 1.

1. Increment operator (++):

→ Increases the value of a variable by 1.

① . Pre-increment : ++x

→ increment first, then uses (replace) the value

② . Post-increment : x++

→ uses (replace) the value first, then increments.

---

② . Decrement operator (--):

→ Decreases the value of a variable by 1.

① . Pre-decrement : --x

→ decrement first, then uses (replace) the value

② . Post-decrement : x--

→ uses (replace) the value first, then decrements.

→ Incrementing or decrementing a variable is faster than using addition or subtraction

→ often used in loops like for and while to iterate over arrays or repeat actions.

① void main ()

```
{
  int a;
  a=5;
  b = ++a; → increment and replace
  printf ("%d %d", a,b);
}
```

a	b
~~5~~ 6	6

o/p: 6 6

---

② void main ()

```
{
  int a,b;
  a=5;
  b = a++; → replace and increment
  printf ("%d %d", a,b);
}
```

a	b
~~5~~ 6	5

o/p: 6 5

---

③ void main ()

```
{
  int a, b;
  a=5;
  b = --a; → decrement and replace
  printf ("%d %d", a,b);
}
```

a	b
~~5~~ 4	4

o/p: 4 4

④. void main()

```
{
  int a, b;
  a = 5;
  b = a--;
  printf("%d %d", a, b);
}
```

→ replace and decrement

a: ~~5~~ 4 b: 5

O/P: 4 5

⑤. void main()

```
{
  int a = 5;
  ++a;
  printf("%d", a);
}
```

O/P: 6

⑥. void main()

```
{
  int a = 5;
  a++;
  printf("%d", a);
}
```

O/P: 6

→ ++a
a++
a = a + 1

→ all three are 100% same

⑦. void main()

```
{
  int a = 5;
  --a;
  printf("%d", a);
}
```

O/P: 4

⑧. void main()

```
{
  int a = 5;
  a--;
  printf("%d", a);
}
```

O/P: 4

--a
a--
a = a - 1

→ all three are 100% same

⑨. void main()

```
{ int a=5;
  printf("%d", ++a);
}
```

a: ~~5~~ 6

o/p: 6

⑩. void main()

```
{ int a=5;
  printf("%d", a++);
}
```

a: 5

o/p: 5

⑪. void main()

```
{ int a=5;
  printf("%d", --a);
}
```

a: ~~5~~ 4

o/p: 4

⑫. void main()

```
{ int a=5;
  printf("%d", a--);
}
```

a: ~~5~~ 4

o/p: 5

⑬. void main()

```
{ int a;
  a = ++100;
  printf("%d", a);
}
```

o/p: ERROR

Compilation Error → L-value required.

⑭. void main()

```
{ float f;
  f = 3.14;
  printf("%f", ++f);
}
```

4.14

o/p: 4.14

⑮. void main()

```
{ int a;
  a = --5;
  printf("%d", a);
}
```

a: 5

unary minus

o/p: 5

⑯. void main()

```
{ int a=5;
  ++(++(++(++a)));
  printf("%d", a);
}
```

o/p: ERROR, C.E-L-value required

(17). void main()

```
{
    int a=10;
    if (a++ >10)
    printf(" hello %d", a);
    else
    printf(" hai %d", a);
}
```

O/P: hai 11

(18). void main()

```
{
    int a;
    a=1;
    while(a++ <=1)
    while(a++ <=2)
    printf("%d", a);
}
```

a: 1 ~~2~~ ~~3~~ ~~4~~ 5

O/P: 5

(19). void main()

```
{
    int a;
    for(a=1; a++ <=1; a++)
    {
    for(a++; a++ <=6; a++)
    {
        a++;
    }
    }
    printf("%d", a);
}
```

O/P: 12

a: 1 ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ ~~7~~ ~~8~~ ~~9~~ ~~10~~ ~~11~~ 12

(20). void main()

```
{
    int a=1;
    do
    {
        ++a;
    printf("%d", a);
    if (a++ <=5)
        continue;
    printf("%d", a);
    }
    while(a++ <=10);
}
```

O/P: 2 5 8 9 11 12

Switch

→ Based on the switch arguement go to the matched case and execute upto break or end of switch

→ If there is no matched case go to default case and execute upto break or end of switch

→ If there is no default and matched case then come out of the switch

→ If the normal case is executed then there is a chance of executing default case also and vice versa

→ Default case can be return written only where in the program inside switch

→ All the case order as our wish and we can write in any order.

→ All the cases must be integral constant and variables and float values, duplicate case values are not allowed. case : Int only

→ Continue keyword cannot be used inside switch

Syntax:

```
switch (choice)
{

}
```

①. void main()

```
{
  int a = 2;
  switch (a)
  {
   case 1: printf("A");
           break;
   case 2: printf("B");
           break;
   case 3: printf("C");
           break;
   default: printf("D");
  }
}
```

O/P : B

②. void main()

```
{
  int a = 2;
  switch (a)
  {
   case 3: printf("C");
   case 1: printf("A");
   case 2: printf("B");
   case 0: printf("D");
   case -5: printf("E");
  }
}
```

O/P : BDE

③. void main()

```
{
  int a = 5;
  switch (a)
  {
   case 7: printf("A");
   case 4: printf("B");
   case 5: printf("C");
   case 21: printf("D");
        continue;
   default: printf("E");
  }
}
```

O/P : ERROR → continue outside loop.

---

④. void main()

```
{
  int a = 5;
  switch (a)
  {
   case 7 : printf("A");
   case a : printf("B");
   case -5 : printf("C");
   case 0 : printf("D");
  }
}
```

O/P : compilation Error - constant Expression required.

(5). void main()

```
{
  int a=5;
  switch (a)
  {
    case 10-5 : printf("A");
    case 10*2 : printf("B");
    case 10/2 : printf("C");
    case 5-1 : printf("D");
    default : printf("E");
  }
}
```

o/p : C.E : duplicate case label.

---

(6). void main()

```
{
  int a=5;
  switch (a)
  {
    case 1*1 : printf("A");
    case 5/2 : printf("B");
    case 5*1 : printf("C");
    case 5*2 : printf("D");
    default : printf("D");
  }}
```

o/p : C D

FLOW CONTROL

Conditional

Selective

- if
- if-else
- conditional (? : ;)
- switch

Iterative

- while
- for
- do-while

Unconditional

- goto
- continue
- break
- return

Short circuit operators

→ Logical AND & Logical OR are also called as short circuit operators.

→ If logical AND first argument is 0 we should not go to second argument, the resultant will be 0.

→ If logical OR first argument is 1, we should not go to second argument, the resultant will be 1.

→ In logical AND & logical OR, we are able to decide the resultant based upon the first argument and by bypassing the second argument and that is the reason logical AND & logical OR are also called as short circuit operators.

→ In c language, there is no order of presidence rule for operands except when applied with the following four operators.

1. logical AND
2. logical OR
3. conditional (? : ;)
4. operator (,) (seperator).

→ The order of presidence rule for operands when applied with above operator is from left to right.

①. $a = b = c = 1;$

$d = (\overset{2}{++a} \;\&\&\; \overset{2}{++b}) \;\&\&\; \overset{2}{++c};$

$d = (\overset{2}{++a} \;\&\&\; \overset{2}{++b}) \;||\; ++c;$

$d = \overset{2}{++a} \;||\; (++b \;\&\&\; ++c);$

$d = (\overset{2}{++a} \;||\; ++b) \;||\; ++c;$

a	b	c	d
~~1~~ 2	~~1~~ 2	~~1~~ 2	1
~~1~~ 2	~~1~~ 2	1	1
~~1~~ 2	1	1	1
~~1~~ 2	1	1	1

O/P:

a	b	c	d
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

---

②. $a = b = c = 1;$

$d = a{+}{+} \;\&\&\; b{+}{+} \;\&\&\; c{+}{+};$

$d = a{+}{+} \;\&\&\; b{+}{+} \;||\; c{+}{+};$

$d = a{+}{+} \;||\; b{+}{+} \;\&\&\; c{+}{+};$

$d = a{+}{+} \;||\; b{+}{+} \;||\; c{+}{+};$

a	b	c	d
1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

---

③. $a = b = c = 1;$

$d = (\overset{0}{--a} \;\&\&\; \overset{0}{--b}) \;\&\&\; \overset{0}{--c};$

$d = (\overset{0}{--a} \;\&\&\; \overset{0}{--b}) \;||\; \overset{0}{--c};$

$d = \overset{0}{--a} \;||\; (\overset{0}{--b} \;\&\&\; \overset{0}{--c});$

$d = (\overset{0}{--a} \;||\; \overset{0}{--b}) \;||\; \overset{0}{--c};$

a	b	c	d
1	1	1	0
1	1	1	0
1	1	1	0
1	1	1	0

(4). a = b = c = 0;

d = a-- && b-- && c--;

d = a-- && b-- || c--;

d = a-- || b-- && c--;

d = a-- || b-- || c--;

a	b	c	d
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

(5). void main()

```
{
  int a, b;
  a = b = 10;
  while (a++ <= 13 && b++ <= 13)
  printf ("%d %d\n", a, b);
}
printf ("%d %d", a+10, b+10);
}
```

O/p: 11 11
12 12
13 13
14 14
25 24

(6). void main()

```
{
  int a, b;
  a = b = 10;
  while (a++ <= 13 || b++ <= 13)
{ printf ("%d %d\n", a, b); }
  printf ("%d %d", a+10, b+10);
}
```

O/p: 11 10
12 10
13 10
14 10
15 10
16 11
16 12
17 13
18 14
29 25

Data types

int a ; definition
↳ datatype

→ In C language, data types defines the type and size of data a variable can hold, as well as the operations that can be performed on that data.

→ Data types determine the memory allocation, the type of operations allowed, and the method for representing values in memory.

→ Data types help the compiler allocate the appropriate amount of memory for each variable.

→ Data types prevent invalid operations by restricting the kind of data that can be stored in a variable.

→ Example: you cannot assign a decimal number to an int variable without explicit type casting.

→ The type of data stored determines the operations that can be performed on the variable.

auto → auto, static, extern, register → storage classes

short → short, long

signed → signed, unsigned

int a;

Auto - auto, static, extern, register

short - short, long

signed - signed, unsigned.

Type qualifiers - const, volatile

signed → MSB → negative bit

Unsigned → MSB → positive bit

[MSB → Most significant Bit]

→ If MSB is 0 then signed and unsigned are Equal.

→ if MSB is 0 → +ve

→ if MSB is 1 → -ve.

Signed	Unsigned
1. MSB : -ve bit	1. MSB : +ve bit
2. size : 4 bits	2. size : 4 bits
3. min : -8	3. min : 0
4. max : 7	4. max : 15
5. range : -8 to 7	5. range : 0 to 15

range:

-2^{n-1} to $2^{n-1}-1$ (n-bits)

-2^{4-1} to $2^{4-1}-1$

-2^3 to 2^3-1

-2^3	2^2	2^1	2^0	
-8	4	2	1	
~~0~~ 0	0	0	0	
~~0~~ 0	0	1	0	
~~0~~ 0	1			

-8	4	2	1		
0	0	0	0	0	
0	0	0	1	1	
0	0	1	0	2	
0	0	1	1	3	
0	1	0	0	4	inc
0	1	0	1	5	
0	1	1	0	6	
0	1	1	1	$\boxed{7}$ →	Max
1	0	0	0	-8 →	Min
1	0	0	1	-7	
1	0	1	0	-6	
1	0	1	1	-5	inc
1	1	0	0	-4	
1	1	0	1	-3	
1	1	1	0	-2	
1	1	1	1	-1	

range:

0 to 2^n-1 (n bits)

Binary Representation

Max : 15 ; 1111

2^3	2^2	2^1	2^0		
8	4	2	1		
0	0	0	0	0	
0	0	0	1	1	
0	0	1	0	2	
0	0	1	1	3	
0	1	0	0	4	
0	1	0	1	5	
0	1	1	0	6	
0	1	1	1	7	
1	0	0	0	8	
1	0	0	1	9	
1	0	1	0	10	
1	0	1	1	11	
1	1	0	0	12	
1	1	0	1	13	
1	1	1	0	14	max
1	1	1	1	15	

	Signed	Unsigned
min - binary value	1 All 0's	All 0's
max - binary value	0 All 1's	All 1's
range	-2^{n-1} to $2^{n-1}-1$	0 to 2^n-1
All 1's	-1	Max
MSB	-ve	+ve

Range:

int - 2 bytes - 16 bits

Signed	Unsigned
-2^{n-1} to $2^{n-1}-1$	0 to 2^n-1
-2^{15} to $2^{15}-1$	0 to $2^{16}-1$
-32768 to 32767	0 to 65535

char - 1 byte - 8 bits

signed	Unsigned
-2^{n-1} to $2^{n-1}-1$	0 to 2^n-1
-2^7 to 2^7-1	0 to 2^8-1
-128 to 127	0 to 255

→ Every character constant have its equivalent integer constant.

→ character constant is called as ASCII code

ASCII → American Standard Code for Information Interchange.

→ And its equalent integer constant is also called as ASCII value

ASCII code	ASCII value
'a'	97
'b'	98
'z'	122
'A'	65
'B'	66
'Z'	90

Examples:

① void main()

```
{
    char ch;
    ch = 100.5;
    printf("%c", ch);
}
```

ch

100.5
↓
d

a - 97
b - 98
c - 99
d - 100
⋮

o/p: d

② . void main()

```
{
  char ch;
  ch = 100;
  printf("%c", ch);
}
```

ch

100 → d

O/p: d

③ . printf("%c", 100); → d

④ . printf("%c", 100.5); → d

⑤ . printf("%c", 'd'); → d

⑥ . printf("%d", 0x13); → 19

⑦ . printf("%f", 100.75); → 100.75

⑧ . printf("%f", 100); → G.V

⑨ . void main()

```
{
  char ch;
  ch = 65;
  printf("%c", ch);
}
```

ch

65

%c → A

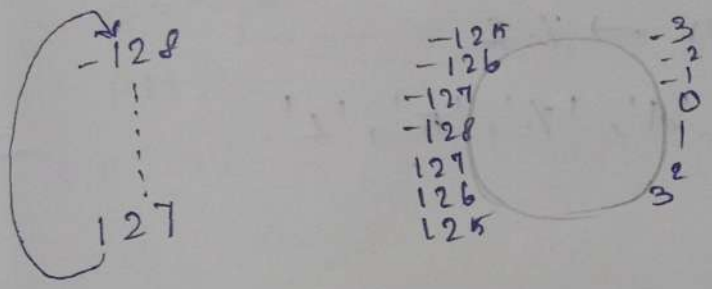
O/p: A

⑩ . void main()

```
{
  char ch;
  ch = 128;
  printf("%d", ch);
}
```

O/p: −128

	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
127	0	1	1	1	1	1	1	1
+1	0	0	0	0	0	0	0	1
	1	0	0	0	0	0	0	0



→ We can assign any type of data to any type of variable, Internally compiler will take care.

→ Here ch is of character datatype (signed char) where we are trying to assign 128 which is outside the range.

→ But character variable ch can only hold any data which is inside the range and also only the data which is of 8 bits (1 byte)

→ Whenever we try to assign any data which is outside the range then we go to butterfly diagram which helps us to take the data whithin the range.

Unsigned char (or) char unsigned

size → 8 bits (1 byte)

range → 0 to 255

format specifier → %c

Example → 'a', 'A', '%', '5', '2'.

1. void main()

```
{
    unsigned char ch;
    ch = -1;
    printf("%d", ch);
}
```

O/P : 255

Signed char : It stores values from -128 to 127.

Unsigned char : It stores values from 0 to 255.

→ In C, char can be either signed or unsigned, and their value ranges differ based on whether they are signed or unsigned.

signed char : Range : -128 to 127

-128 — 0 — 127

negative — positive

Negative Range : From -128 to -1

Positive Range : From 0 to 127.

unsigned char: (Range: 0 to 255)

0 ——— 255

only positive values

① . a = -128
↳ a++ → 127
a++;

o/p: 127

② . b = 127
↳ b++ → -128
b++;

o/p: -128

Input	output
0	0
-1	255
-2	254
-3	253
-4	252
-5	251
255	255
256	0
257	1
258	2

→ short, signed int, short int
→ short signed, int short, signed
→ short, signed int, int signed short
→ signed int short, short signed int
→ int short signed, signed short int, short int signed.

→ Among 3 either of them are not there, Others will be taken by default.

size → 16 bits (2 bytes)

format specifier → %d (or) %i

Range → -32768 to 32767

Example : 5 -5 0

Input	Output
32768	-32768
32769	-32767
32770	-32766
-32769	+32767
-32770	+32766
-32768	+32768

---

① .
```
void main()
{
    short s;
    s = 32768;
    printf("%d", s);
}
```

32767

O/P: -32768

---

②.
```
void main()
{
    short s;
    s = 200 * 200;
    if(s == 200 *200)
{
    printf("hello");
}
    else
{
    printf("hai");
}
```

-ve 40000

O/P: Hai

③. void main()

s
$\boxed{1}$ 2

{ short s;

s = 1;

o/p: -32767.

while (s++ > = 1)

{ printf("%d" s);

}

→ unsigned int, unsigned short int, unsigned short, short unsigned, unsigned

size → 16 bits (2 bytes)

range → 0 to 65535

format specifier → %u

Examples: 5U, 100U, 0XFU.

---

①. void main()

{ unsigned short u;

u = -1;

printf("%d", u);

o/p: $\boxed{-1}$

}

printf("%u", u); o/p: $\boxed{65535}$.

Input	output	Input	output
-1	65535	65538	2
-2	65534	65539	3
-3	65535		
-10	65525		
65536	0		
65537	1		

① .
```
void main()
{
  signed s = -2;
  unsigned u = 3;
  if (s > = u)
  printf("hello");
  else
  printf("hai");
}
```
→ 65534 > = 3

O/P: Hai

②.
```
void main()
{
  unsigned short u;
for(u = s; u > = 0; u = u - 1)
{
  printf("%u", u);
}}
```

O/P: 5 4 3 2 1 0, 65535, ... ∞

Type Promotionality Rule: Whenever we perform any operation, compiler will promote lower data type to heigher order datatype

Long int, signed long int, signed long, long

size → 4 bytes (32 bits)

range → -2147483648 to 2147483647

format specifier → %ld, %li

Example: 5l, 5L, 0x123l.

Unsigned long int, unsigned long, long unsigned int

size → 4 bytes (32 bits)

range → 0 to 2494967295

format specifier → %lu (or) %ul

Function

→ Whenever we notice a piece of code is repeated randomly in order to avoid code random repeatation, we go to a concept called Function.

→ The purpose of function is code reusage (writing the code in one place and reusing that code in multiple place).

Advantages:

1. Debugging will be easy
2. Maintanance easy
3. sharing will be easy.

```
void main()
{

}
```

→ return type
→ function name,
function definition / main function definition

→ () → function call operator fun()

→ Whenever, we encounter a function call then only the function definition is executed.

→ Whenever we encounter a function call the control will go to respective function definition. → void fun(){ }

→ once after executing the function call (or) function definition get back to the respective function call. (main).

→ operating system is responsible to call the main function.

→ We can call any function from any function. [We can also call main function].

Examples

```
void test()
{
  printf("Hi test");
}
void main()
{
  printf("Hi main");
  test();
}
```

→ function definition (or) test function definition (or) test function

→ function definition (or) main function (or) main function definition

→ function calling (or) function invoking (or) test function calling.

o/p: Hi main
Hi test

→ fun-name () { } → function definition

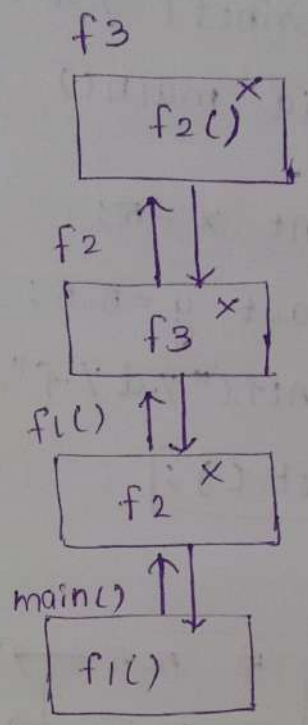
→ fun-name(); → function call.

→ Whenever a function call is executed the controll will go to respective function definition and once function definition completed (once we encounter end of function definition) we get back to the function from where it was called

→ once main is executed (once we encounter end of main function) program execution is completed.

Examples:

①.
```
void f3()
{
printf("Hi f3\n");}
void f2()
{
printf("I am f2\n");
f3();
printf("Hi f2\n");}
void f1()
{
printf("Hi f1\n");
f2();
printf("I am f1\n");
}
void main()
{
printf("Hi main\n");
f1();
printf("I am main");
}
```



O/p: Hi main
Hi f1
I am f2
Hi f3
Hi f2
I am f1
I am main

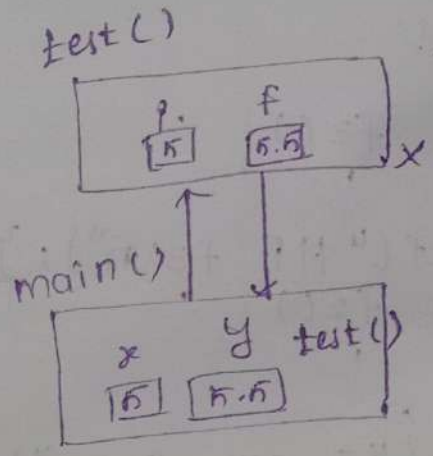
```
f1();
f2();
f3();
```
→ function calls

```
void f1(){ }
void f2(){}
void f3(){}
```
→ function definition

② . void test (int i, float f) → dummy variable

fun-name

```
{
  printf("%d %f", i, f);
void main()
{
  int x = 5;
  float y = 5.5;
  printf("%d %f", x, y);
  test ();
}
```



O/P: 5 5.5 → from main
5 5.5 → from test()

---

③ . test (x, y); ✓

int float

test (5, 5.5); → 5, 5.5

test (5, 5); → 5 5.0

test (5.5, 5.5) → 5 5.5

test (5/2, 5/2); → 2 2.0

test (5.0/2, 5.0/2); → 2 2.5

test (5%2, 5%2); → 1 1.0

test (5%2, 5%2); → 1 1.0

test (5.0%2, 5.0%2); → C.E → illegal use of floating point.

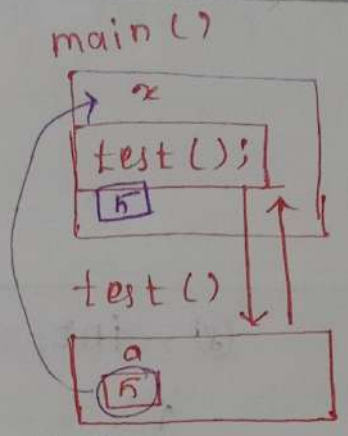
test (5, 2, 3); C.E → too many parameters / extra parameters.

```
test (5);
test ();
```
→ too few parameters.

```
test (,);
test (5, );
test (, 6);
```
→ Compilation Error: Expression syntax.

---

(4).
```
int test()
{
  int a = 5;
  return a; }
void main()
{
  int x;
  x = test();
  printf("%d", x);
}
```



o/p: 5

→ function call will be replaced with the returned value.

→ return is also called as function terminator (keyword).

---

(5).
```
int test()
{
  float f = 3.75;
  return f;
}
void main()
{
  float f;
  f = test();
  Printf("%f", f);
}
```

→ return type → int

o/p: ~~3.75~~ 3.0

↳ because test() function return type is int.

⑥.

```
float test()
{
  int a = 5;
  return a;
}
void main()
{
  float f;
  f = test();
  printf("%f", f);
}
```

return type

o/p : 5.0

⑦.

```
float test()
{
  float f = 3.15;
  return f;
}
void main()
{
  float f;
  f = test();
  printf("%f", f);
}
```

3.15

⑧.

```
int test()
{
  3 > 2 ?
  return 3 : return 2;
}
void main()
{
  int a;
  a = test();
  printf("%d", a);
}
```

o/p: Compilation Error;
Expression syntax

→ We cannot use return statement in operator.

→ But we can use operator in return statement.

Notes:

→ True (or) False function call must be always terminated with ';'

Example: return true;
return false;

→ Whenever a function call is part of any statement it need not be terminated with ';'

---

(9). void main()

```
{
    int a;
    a = printf("hello");
    b = printf("%d", a);
    printf("%d", b);
}
```

no of characters hello → 5

a: hello

5 → no of char

b: 5

O/P: 5 1

→ printf is a function call
→ It will return number of characters.
→ The entire function call will be replaced with the return value (no. of char).
→ printf is also called as built-in function (or) library function (or) standard function (or) predefined function.

---

(10). Write a program to find the given number is positive or negative without using if, arithmetic operator, relational operator.

a>0 && printf("positive") || printf("negative

(11).
```
void students()
{
printf("hello guys");
}
void main()
{
students();
}
```
O/P: hello guys

(12).
```
void main()
{
guys();
}
void guys()
{
printf("hello guys");
}
```
O/P: Compilation Error: redecleration to guys().

(13). Write a program to find the given number is odd or even

```
a % 2 == 0 &&
printf("even")||
printf("odd");
```

→ Whenever function calls is below and function definition is above, there is no problem.

→ Whenever function call is above and function definition is below compilation error will occur, because compiler will assume afterwords in the future, the return type is coming with type int & if we use any other type we will get compilation error.

(14).
```
int test()
{ return ;
}
void main()
{
  int a;
  a = test();
print("%d", a);
}
```

o/p: Garbage value

warning: function must return a value.

(15).
```
void test()
{
  return 5;
}
void main()
{
  int a;
  a = test();
  printf("%d", a);
}
```

o/p: Compilation Error: Expression syntax.

→ function decleration inside the { } is called as local function decleration.

→ function decleration outside the { } is called as global function decleration.

→ variable defined inside the { } is called as local variable.

→ variable defined outside the { } is called as global variable

→ The default value of global variable is 0 and default value of local variable is G.V

Storage classes

→ Defining any variable inside {} is called local variable.

→ The default value of local variable is Garbage value.

→ The default storage class of local variable is auto.

→ Defining any variable outside {} is called global variable.

→ The default value of global variable is 0.

→ The default storage class of global variable is extern.

Storage class	scope (accessibility / visibility)	life (alive/dead)	default value
auto	within the body	within the body	Garbage value
static	within the body	entire program	0
extern	entire program	entire program	0
register	within the body	within the body	Garbage value

①. void main()

```
{
  int a;
  a=10;
  {
  int b;
  b=20;
  printf("%d", b); //20
  }
  printf("%d", a); //10
  printf("%d", b);
}
```

→ life → auto → within the body

b is a local variable, so default storage class is auto.

o/p: Compilation error : undefined variable 'b'

---

②. NOTE:

→ We cannot define same variable name within the same scope

→ We can define same variable name within different scope.

---

②. void main()

```
{ int a=10;
  goto inside;
  {
   int a=20;
   {
    inside:
   printf("%d", a);
   }
}}
```

a: G.V 10

a: G.V

o/p: Garbage value.

③.

```
void func()
{
  auto int num=10;
  printf("%d", num);
}

void main()
{
   func();
}
```

o/p: 10

---

④.

```
void main()
{
  auto int a=5;
  printf("%d", a);
{
  auto int a=10;
  printf("%d", a);
}
printf("%d", a);
}
```

	a	a	a
o/p:	5	10	5

---

⑤.

```
void func1()
{ auto int a =10;
  printf("%d", a);
}
void func2()
{ auto int b=20;
  printf("%d", b);
}
void main(){
   func1();
   func2();
   func1(); }
```

	a	b	a
o/p:	5	10	5

Static storage class:

Scope:	Within the body
Life:	Entire program
Default value:	~~Garb~~ 0.

Examples:

①
```
void counter()
{
    static int count=0;
    count++;
    printf("%d", count);
}
void main()
{
    counter();
    counter();
    counter();
}
```

O/p: 1 2 3

---

②.
```
void increment()
{ static int num=1;
  num++;
  printf("num incremented: %d\n", num);
}
void display()
{ static int num=1;
  printf("Display num %d\n", num);
}
void main()
{
  increment();
  display();
  increment();
  display();
}
```

num ① [1] 2 num ② [1] 3

num [1]

O/p: num incremented: 2
Display num: 1
num incremented: 3
Display num: 1

③. void test()

```
{
  static int i=0;
  i++;
  printf("%d\n", i);
}
void main()
{
  for(int j=0; j<3; j++)
    test();
}
}
```

O/p: 1
2
3

④. static int globalVar = 100;

```
void printGlobalVar()
{
  printf("Global static var: %d\n", globalVar);
}
void main()
{
  printGlobalVar();
}
```

O/p: Global static var: 100.

⑤. void initializeOnce()

```
{
  static int num=10;
  printf("%d\n", num);
  num+=10;
}
void main()
{
  initializeOnce();
  initializeOnce();
  initializeOnce();
}
```

O/p: 10
20
30.

num
~~10~~
~~20~~
30

Extern storage class

Scope: Entire program
Life: Entire Program.
Default value: 0

Examples:

① - void main()

```
{
  extern int a;
  a=10;
  printf("%d", a);
}
```

o/p: linker error.

---

test.c

```
void main()
{
 test();
}
```

compiler → test.obj: object code → linker → linker error.

→ for windows → linker error: Undefined symbol test()

→ for linux → linker error: undefined reference test().

test.c

```
void main()
{
 extern int a;
 a=10;
 printf("%d", a);
}
```

compiler → test.obj: object code → linker → linker error

→ To avoid linker errors when using the extern storage class in C,

1. Declare with extern in files where the variable or function is used.

2. Define the variable / function in exactly one source file without extern

3. Include the declaration in a shared header file if used across multiple files.

4. Link all source files together during compilation.

5. Ensure proper scope by only defining (not declaring as extern once).

Examples:

1.
```
void main()
{
    extern int g;
    g = 10;
    printf("%d", g);
}
```

O/p: linker Error

2.
```
void main()
{
    extern int g;
    printf("%d", g);
}
int g;
```

O/p : 0.

[extern class default value = 0]

③. void main()

```
{
 extern int g;
 g=10;
 printf("%d", g);
 int g=20;
}
```

O/P: 10

④. void main()

```
{
 extern int g=10;
 printf("%d", g);
}
```

O/P: Compilation Error: Decleration cannot be initialized

⑤. void main()

```
{
 extern int g;
 g=10;
 printf("%d", g);
} int g;
```

O/P: 10

⑥. void main()

```
{
 extern int g;
 printf("%d", g);
} int g=10;
```

O/P: 10.

⑦. void main()

```
{
 extern int g;
}
```

O/P: no output.

(8).
```
void main()
{
    extern int g;
    printf("Hello");
}
```
O/P: Hello.

(9).
```
int g;
void main()
{
    printf("%d", g);
}
```
O/P: $\boxed{0}$.

(10).
```
extern int g=10; → def
void main()
{
    printf("%d", g);
}
```
O/P: 10.

Register storage Class

Scope: within the body
Life: within the body
default value: Garbage Value

→ The register storage class in C, used to suggest to the compiler that a variable should be stored in a CPU register instedd of RAM.

→ We cannot take the address of the register variable.