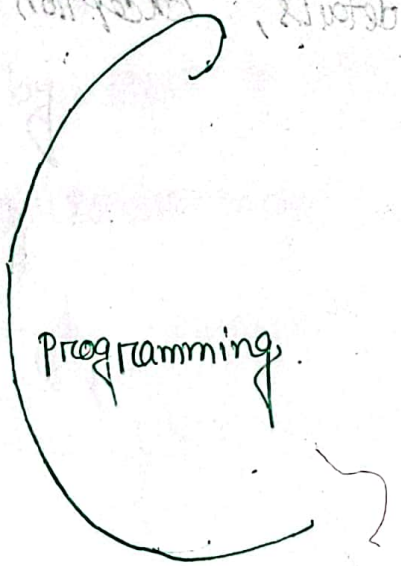
C Programming Note

PART-01



TOPICS:

① C, C++, Java র output problem.

② Simple programming.

③ Pattern matching.

OOP: JVM, variables, access modifiers, constructor, OOP র basic বিষয় গুলি details, Exception handling.

C - Programming

'C' is a general-purpose programming language that is extremely popular, simple and flexible.

↳ It is developed by Dennis Ritchie at Bell Lab in 1972

↳ Successor of 'B' Language and BCPL etc

↳ Standard by ANSI in 1989.

↳ It is very popular language, despite being old.

↳ C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

C Programming can be defined by the following ways:

1. Mother language
2. System programming language.
3. Procedure-oriented programming language.
4. Structured programming language.
5. Mid-level programming language.

1) 'C' as a mother language

↳ Most of the compilers, JVMs, kernels, etc. are written in C Language. → main element of OS.

↳ Most of the programming language follow C syntax — for example, C++, Java, C# etc.

↳ It provides the core concepts like the array, strings, functions, file handling, etc. that are being used in many languages like C++, Java, C# etc.

2) C as a system programming language

↳ A system programming language is used to create system software.

↳ C is a system programming language because it can be used to do low-level programming (for example driver and kernel)

↳ It is generally used to create hardware devices, OS, drivers, kernels etc. For example: Linux kernel is written in C.

↳ It can't be used for internet programming Java, .NET, PHP etc.

3) C as a procedural language.

↳ A procedure is known as a function, method, routine, subroutine, etc.

↳ A procedural language specifies a series of steps for the program to solve the problem.

↳ A procedural language breaks the program into functions, data structures, etc.

↳ C is a procedural language. In C, variables and function prototypes must be declared before being used.

4. Why 'C' language being considered a middle level language?

C is considered as a middle-level language because it

↳ supports the feature of both low-level and high-level languages.

↳ It can be used for both, system programming (like as operating system)

↳ As well as application programming (like as spreadsheet).

Middle-level language are more related to a machine as well as human language. So, that's why it is called "Middle-level language".

C language merges the best element of high-level language With the rule and flexibility of assembly language.
'C' allows the manipulation of bits and addresses and bytes.

Why to Learn 'C' Programming ?

N.B: Features কোনো ২টা

i. Easy to learn
ii. Structured language
iii. It produces efficient programs.
iv. It can handle low-level activities.
v. It can be compiled on a variety of computer platforms.
vi. C is very fast, compared to others

Features of C Language:

1. Simple.
2. Machine independent or Portable.
3. Mid-level programming language.
4. Structured programming language.
5. Rich Library.
6. Memory Management.
7. Fast Speed.
8. Pointers.
9. Recursion.
10. Extensible.

Applications of C Programming.

1. Operating systems
2. Language compilers
3. Assemblers
4. Text Editors.
5. Print Spoolers
6. Network drivers
7. Modern programs.
8. Databases.
9. Language Interpreters.
10. Utilities.

Difference between High-Level and Low-Level Language.

High-Level Language	Low-Level Language.
① It is programmer friendly language.	① It is a machine friendly language.
② High level language is less memory efficient.	② Low-Level language is high memory efficient.
③ It is easy to understand.	③ It is difficult to understand.
④ It is simple to debug.	④ It is complex to debug.
⑤ It is simple to maintain.	⑤ It is complex to maintain.
⑥ It is portable.	⑥ It is non-portable.
⑦ It can run on any platform.	⑦ It is machine-dependent.
⑧ It needs compiler or interpreter for translation.	⑧ It needs assembler for translation.
⑨ It is used widely for programming	⑨ It is not commonly used now a days in programming.
⑩ Example: Java, Python.	⑩ Example: Assembler and machine language.

* Mid level Language = C, C++

Hello World Examples

A 'C' Program basically consists of the following parts -

- Preprocessor Commands.
- Functions.
- Variables.
- Statements & Expressions.
- Comments.

Structure of C program:

Header / Pre-processor	# include <stdio.h>
main ()	int main () {
Variable declaration	int a = 10;
body	Printf ("%d\n", a);
Return	return 0; }

1. Header / pre-processor:

#include <stdio.h> → যেটার declare করা থাকে সব ধরনের input/output file (printf, scanf) use করার আগে header file declare না করলে function গুলো use করলে error দেখাবে

→ The #include is a preprocessor command that tells the complier to include the contents of stdio.h (standard input&output) file in the program.

2. int main () → এখান থেকে আমরা code লিখি। code-এ অনেক function use করি (scanf(), printf()) এর জন্য খুবই difficult যে কোথা থেকে code execution শুরু করবে। এজন্য main() fonction declare করি, এখান থেকে code execution শুরু করবে।

→ The execution of a C program start from the main () function.

→ Various form of main() function.

① main () → empty parenthesis indicates does not take any argument, value or parameter.
② int main () →
③ void main ()
④ main (void) → does not return any value.
⑤ void main (void)
⑥ int main (void)

3. variable declaration

```
int a = 10;
```

4. printf() fonction:

↳ printf() is a library function to Send formatted output to the Screen. In this program, printf() displays "Hello world" text on the screen.

5. \n (newline)

→ indicates new line.

6. return 0;

→ It is not mandatory to written, but it is a very good practice to ~~return~~ written (return 0)

→ return 0; statement is the "Exit status" of the program.

* This program might be called by some other program, and that program which call this particular program wants to know this status of execution.

When we write return 0; → it means it is success

Program:

```
#include <stdio.h>
int main()
{
    int a=10;
    printf("Hello World\n");
    return 0;
}
```

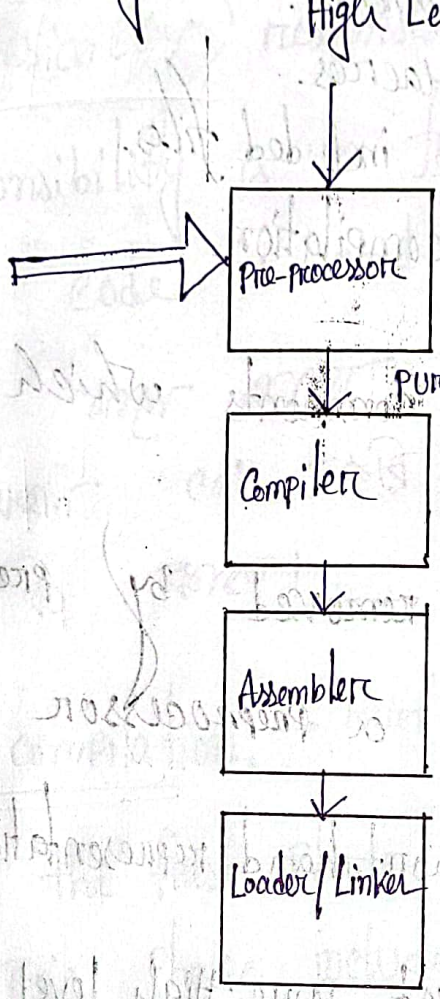
→ A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source file.

Some of 'C' Header files:

1. stddef.h → Defines several useful types and macros.
2. stdint.h → Defines exact width integer types.
3. stdio.h → Defines core input and output functions.
4. stdlib.h → defines numeric conversion functions, memory allocation.
5. string.h → Defines string handling functions.
6. math.h → Defines common mathematical functions.

Phases of Compilation:

Compilling a C program:



→ Convert High-level language to low-level language. (Assembly language)

→ Compiler is not generate executable code.

Q: What is the rule of compiler?

Ans: Take High-level language and convert to it Assembly code. Then assembler generate machine code and it is relocatable.

* This is the first phase through which source code is passed. This phase include:
  - Removal of comments.
  - Expansion of Macros.
  - Expansion of the included files.
  - conditional compilation.

→ Preprocess delete the comments which is write in the code

→ All the spaces are removed by pre-processor.

→ # include(h) is also a preprocessor directives.

→ Macros → is a short hand representation of big line.

→ After pre-processing get pure High level language.

→ Compiler convert High level code to Assembly code.

→ Assembler take assembly code and convert it to machine code (relocatable) or object code.

→ Relocatable means, let us say goto 100; → it means 100 line, but when we load it into machine 100 line may be presented in 1000.

So, it is responsibility of loader to change the addresses which are present inside the program. (13)

→ Loader basically responsible for correct the addresses.

→ Linker responsibility is to find the where the particular code is and linked it.

↳ Linker মানে হলো একটা function অন্য কোন code-এ থাকলে call করা হলে যে কাজটি হয় তাকে linker এ বলে। → figure আছে page-এ আগে।

* Phases of compilation: → briefly দিলে বুঝতে সুবিধা হবে।

Preprocessor:

① This is the first phase through which source code is passed. This phase include

- Removal of comments
- Expansion of Macros.
- Expansion of included files.
- Conditional compilation.

② The compiler converts this code into assembly code. or we can say that the 'c' compiler converts the pre-processed code into assembly code.

CamScanner

→ ③ The assembly code is converted into object code by using an assembler.

→ The name of the object file generated by the assembler is the same as the source file.

→ The extension of the object file in DOS is '.obj' and in UNIX, the extension is 'o'.

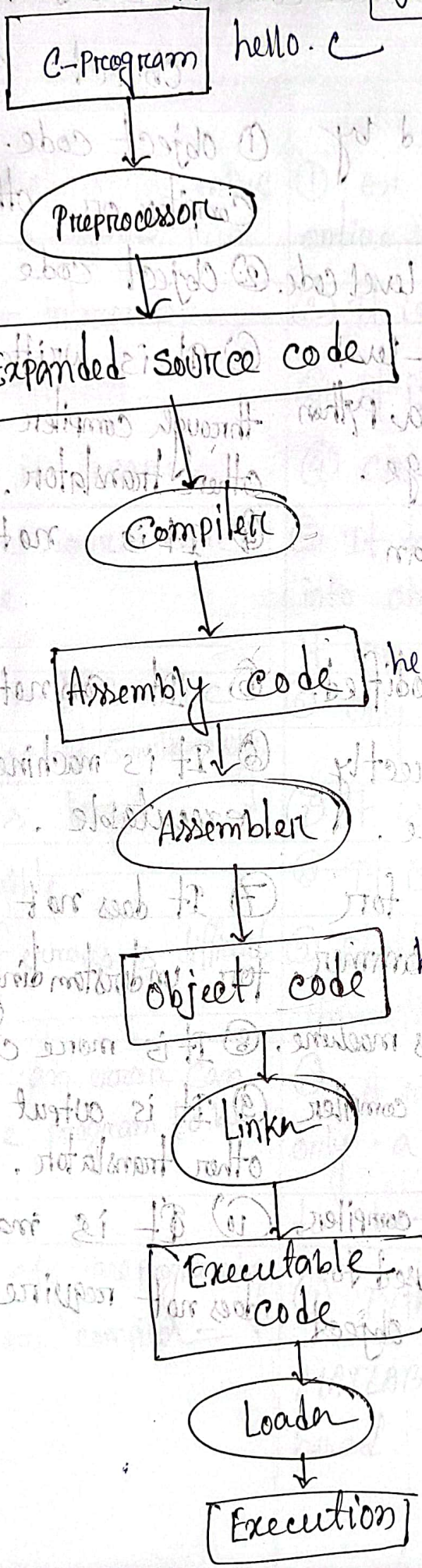
→ If the name of the source file is 'hello.c' then the name of the object file would be 'hello.obj'.

④ The main working of the linker is to combine the object code of library files with the object code of our program.

→ Sometimes the situation arises when our program refers to the functions defined in other files, then linker plays a very important role in this. It links the object code of these files to our program.

→ Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. The output of the linker is executable file.

* Flow diagram of compilation - Javatpoint



Difference between Source code and object code.

Source code	Object code.
① Source code is generated by human or programmer.	① object code is generated by Compiler or other translator.
② Source code is high level code.	② Object code is low - level code.
③ It is written in a high-level language like C, C++, Java, Python etc. or assembly language.	③ It is written in machine language through compiler or assembler or other translator.
④ Source code is human understandable.	④ It is not human understandable.
⑤ It can be easily modified.	⑤ It can not be modified.
⑥ Source code is not directly understandable by machine.	⑥ It is machine understandable and executable.
⑦ It contains comment for better understanding by programmer.	⑦ It does not contain comments for understanding by machine.
⑧ It is less close towards machine.	⑧ It is more close towards machine.
⑨ Source code is input to compiler or any other translator.	⑨ It is output of compiler or any other translator.
⑩ Language translator like compiler, assembler, interpreter are used to translate Source code to object code.	⑩ It is machine code so it does not require any translation.

Compiler Vs Interpreter

Compiler	Interpreter
① A compiler translates the entire source code in a single run.	① An interpreter translates the entire source code line by line.
② It is faster than interpreter.	② It is slower than the compiler.
③ It is more efficient.	③ It is less efficient.
④ CPU utilization is more.	④ CPU utilization is less.
⑤ It converts the source code into object code.	⑤ It does not convert source code into object code. instead it scans it line by line.
⑥ Both syntactic and semantic errors can be checked simultaneously.	⑥ Only syntactic errors are checked.
⑦ The compiler is larger.	⑦ It is smaller than compilers.
⑧ It is not flexible.	⑧ It is flexible.
⑨ Identification of errors is difficult.	⑨ Identification of errors is easier than the compiler.
⑩ A presence of an error can cause the whole program to be organized.	⑩ A presence of an error causes only a part of the program to be organized.
⑪ C, C++, C#, etc are programming languages that are compiler based.	⑪ Python, Ruby, Perl, SNOBOL, MATLAB etc are interpreter based.

Basic Concepts

Character Set:

A character set is a set of alphabets, letters and some special characters that are valid in C language.

↳ A character set in 'C' is divided into

1. Letters / Alphabets.
2. Numbers / digit.
3. Special characters.
4. White spaces (blank spaces)

① Alphabets / Letters

↳ Uppercase characters (A-Z)

↳ Lowercase characters (a-z)

② Numbers / digit

↳ All the digits from (0-9)

③ White spaces.

↳ Blank space

↳ Newline

↳ Carriage return

↳ Horizontal tab.

(4) Special Characters

~ ! # % ^ & * ( ) - + = { } [ ] < > , . / ? \ | : ; " ' @ $

Variables in C:

↳ A variable is a container (storage area) to hold data.

↳ A variable is a name of the memory location.

It is used to store data.

↳ It's value can be changed, and it can be reused many times.

↳ To indicate the storage area, each variable should be given a unique name (identifier).

Syntax to declare a variable:

```
type variable_list;
```

Example:

```
int a;
float b;
char c;
```

* Rules for defining a variable —

① A variable can have alphabets, digits, and underscore.

② A variable name can start with the alphabet, and underscore only. It can't start with a digit.

③ No whitespace is allowed within the variable name.

④ A variable name must not be any reserved word or keyword. e.g. int, float etc.

Example: _srujan, srujan_poojari, srujan812, srujan_812

* We can't declare a variable in the form.

① Srujan poojari (it contains wide space) X

② 13srujan (start with a digit) X

③ void, char, int (these are reserved words).

N.B. C is a case Sensitive language - that means. a variable named 'age' and 'AGE' are different.

* Types of variables:

1. Local variable
2. global variable
3. Static variable
4. automatic variable
5. external variable.

1. Local variable: A variable that is declared and used inside the function or block is called a local variable.

↳ It is scope is limited to function or block.

↳ It cannot be used outside the block.

Examples:

```
#include <stdio.h>
void function (){
    int x = 10; // Local variable // local variable.
}
int main ()
{ function ();
}
```

2. Global variable:

A variable that is declared outside the function or block is called a global variable.

↳ It is declared as the start of the program.

↳ It is available for all functions.

Example:

```
#include <stdio.h>
int x = 20; // global variable
void function1 ()
{ printf ("%d\n", x);
}
void function2 ()
{ printf ("%d\n", x);
}
int main(){
  function1 ();
  function2 ();
  return 0;
}
```

→ In the above code, both functions can use the global variable x as we already have global vaiables accessible by all the function.

3. Static Variable :

↳ A variable that is declared with the static keyword is called static variable.

↳ It retains its value between multiple function calls.

Examples:

```
void function1 (){
int x=10; // local variable
static int y=10; // static variable

x = x+1;
y = y+1;
printf("%d,%d", x, y);
}
```

Explanation:

↳ If you call this function many times, the local variable will print the same value for each function call, e.g. 11, 11, 11 and so on.

↳ But the static variable will print the incremented value in each function call, e.g. 11, 12, 13 and so on.

4. Automatic variable :

↳ All variables in C that are declared inside the blocks are automatic variables by default.

↳ We can explicitly declare an automatic variable using the auto keyword.

↳ It is similar to local variable.

Example:

```
#include <stdio.h>
void function ()
{
    int x=10; // local variable (also automatic)
    auto int y=20; // automatic variable
}
int main ()
{
    function ();
    return 0;
}
```

* In this above example, both x and y are automatic variables. The only difference is that variable y is explicitly declared with auto keyword.

5. External variable

↳ We can share a variable in multiple C source file by using an external variable.

↳ to declare an external variable, you need to use extern keyword.

Example:

```
extern int x = 10; // external variable (also global)

#include "myfile.h"
#include <stdio.h>
void printvalue(){
    printf("Global variable: %d", global_variable);
```

variable type not void

* void a; ✗ → incorrect

invalid
* ① a = 10
② int a;
printf("%d", a)

* "C" language case sensitive
① Num
② num
③ NUM
three are diffrent.

[Jenny]

① abc ✓
② average ✓
③ sum12 ✓
④ sum_12 ✓
⑤ sum 12 ✗
⑥ sum$12 ✗
⑦ 12sum ✗
⑧ _sum ✓
⑨ sum@12 ✗

① SimpleInterest ✓
② age ✓
③ int_type ✓
④ num12 ✓
⑤ _b ✓
⑥ SUM ✓
⑦ _Jenny ✓
⑧ Jenny's_lectures ✗
⑨ float ✗
⑩ 1ab ✗
⑪ %- ✗

Procedural Programming: is a programming paradigm, derived from structured programming based on the concept of the Procedure call.

↳ Procedures, also known as routines, subroutines, or functions simply contain a series of computational steps to be carried out.

↳ Any given procedure might be called at any point during a program's execution, including by other procedures or itself.

↳ The first major procedural programming languages appeared circa 1957 - 1964, including Fortran, ALGOL, COBOL, PL/I and BASIC.

↳ Pascal and C were published circa 1970 - 1972

Source File: The file contains the source code of the program.

↳ The file extension of any C file is .c. Like first.c

Header file: A header file is a file with extension (.h) which contains the C function declarations and macro definitions and to be shared between several source file.

⊞ Object file: An object file is a file containing object code with an extension (.o), meaning relocatable format machine code that is usually not directly executable.

* Executable file:

↳ The binary executable file is generated by the linker.

↳ Linker links the various object files to produce a binary file that can be directly executed.

Q. What are header files and what are its uses in C?

Ans:

↳ Header files are also known as library files.

↳ A header file is a file containing C declarations and macro definitions to be shared between several source files.

→ You request the use of a header file in your program by including it, with the C preprocessing directive #include.

↳ Header files contain two essential things: the definitions and prototypes of functions being used in a program.

↳ Simply put, commands that you use in 'C' programming are actually functions that are defined from within each

header files.

↳ Each header files contains a set of functions.

For example, stdio.h is a header file that contains definition and prototypes of commands like printf and scanf.

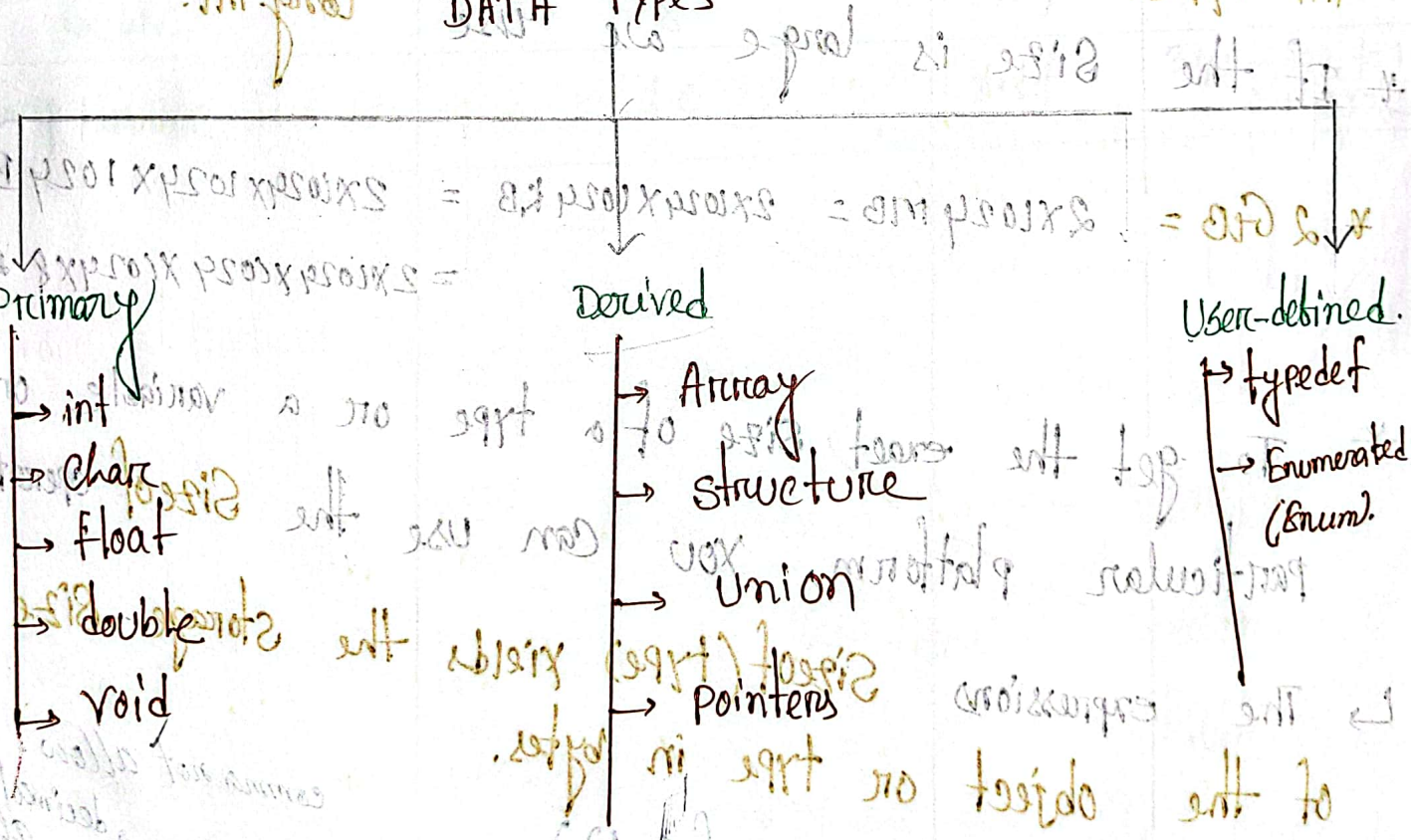
Example: stdio.h, conio.h, math.h, string.h etc.

DATA Types in C

↳ Data types in C refers to an extensive system used for declaring variables or functions of different types.

↳ The type of variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

↳ Data types tells us how much storage (memory) to be allocated to a variable.



1. Primary Data Types:

(i) int

↳ Integers are whole numbers that can have both zero, positive and negative values but no decimal values.

For example, 0, -5, 10

int → short int
int → long int
signed int
unsigned int

* If the size of the instruction is small we use short int type.
* If the size is large we use long int.

* 2 GiB = 2×1024 MiB = 2×1024×1024 KiB = 2×1024×1024×1024 Byte
= 2×1024×1024×1024×8 bits.

* To get the exact size of a type or a variable on a particular platform, you can use the sizeof operator.

↳ The expressions sizeof(type) yields the storage size of the object or type in bytes.

* printf("%lu", sizeof(int));

comma not allow.
decimal not allow.

① -250 ② 15053 ③ +2100 ④ 0 ⑤ 4,442 ⑥ -1.8
⑦ 8888888 → out of range (-32768 to 32767)

Type	Storage size (bytes)	Format specifier	Value range
int/signed int	at least 2, usually 4	%d; %i	-32,768 to 32767 or -2,147,483,648 to 2,147,483,647
unsigned int	2	%u	0 to 65535
short int	2	%hd	-32,768 to 32767
long int	4	%ld, %li	-2,147,483,648 to 2,147,483,647
char/signed char	1	%c	-128 to 127
unsigned char	1	%c	0 to 255
Float	4	%f	3.4E-38 to 3.4E+38
double	8	%lf	1.7E-308 to 1.7E+308
long double	10	%Lf	3.4E-4932 to 1.1 E+4932

2. Char Type: → stores a single character / Letter / number, or ASCII values.

↳ keyword 'char' is used for declaring character type variables.

↳ The size of the character variable is 1 byte.

Examples:

char a; 1 bytes (8 bits).

↳ signed → -128 to 127

↳ unsigned → 0 to 255

Printf("%c", 98);

output: b

3. Float and Double:

↳ float and double are used to hold real numbers.

↳ float size 4 bytes, rang -3.4e38 to + 3.4e38.

Ex: float a;

a = 10.0;

printf("%lf", salary);

↳ double size 8 bytes.

↳ long double → 10 bytes.

4. void type

↳ A void data type doesnot contain or return any value.

↳ It is mostly used for defining functions.

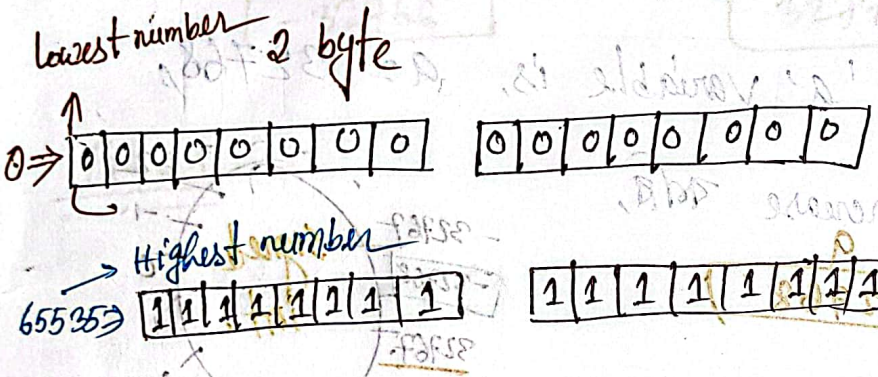
Example:

```
void displaydata();
```

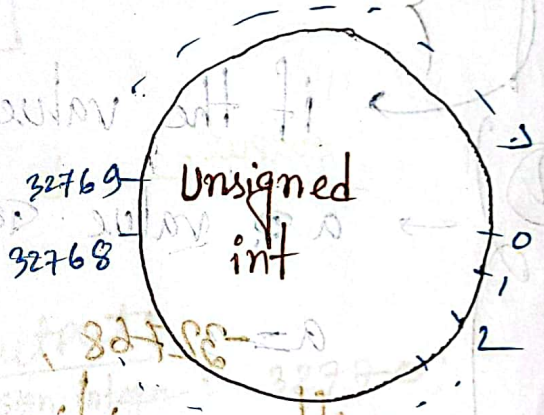
* Integers behavior:

-32768 to 32767 ⇒ signed int

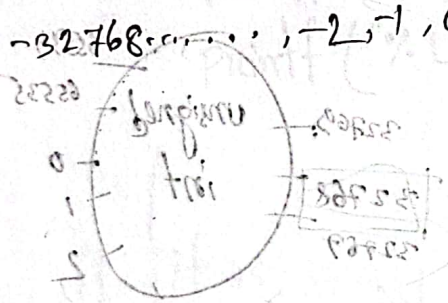
0 to 65535 ⇒ unsigned int

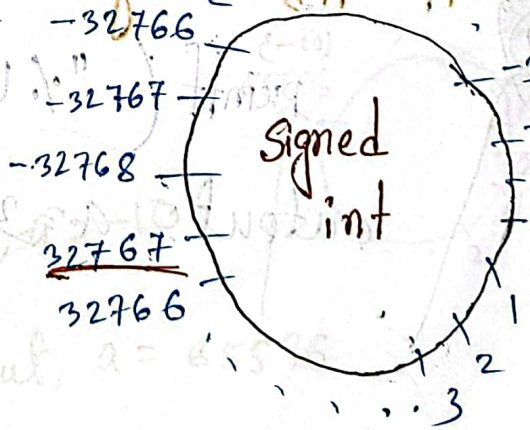


$$\frac{65535}{2} = 32768$$



(a)





(b)

Example:

if we don't write signed or unsigned before int it will be signed integer.

```
#include <stdio.h>
# include <conio.h>
void main()
{
int a = 32767;
clrscr();
printf("%d", a);
getch();
}
```

→ just allocate memory not value store

a → 32767 (1000 = memory location)

→ -32768 to 32767 signed int এর location, ∴ a = 32767

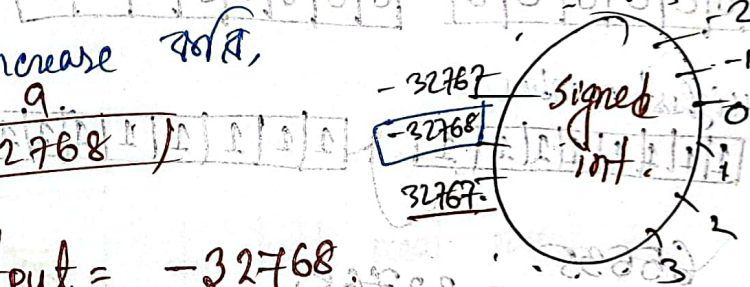
output: 32767, a → 32767

① if the value of 'a' variable is, a = 32768;

② → a এর value কে increase করবে,

a = -32768, a → -32768

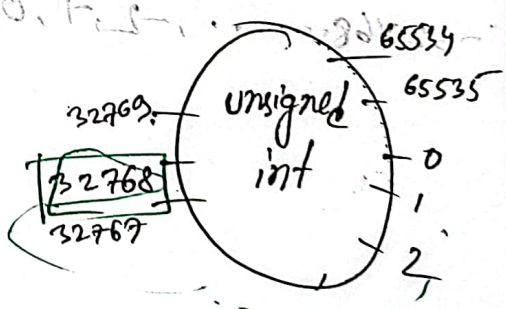
→ After printing output = -32768



③ if it's unsigned int then print

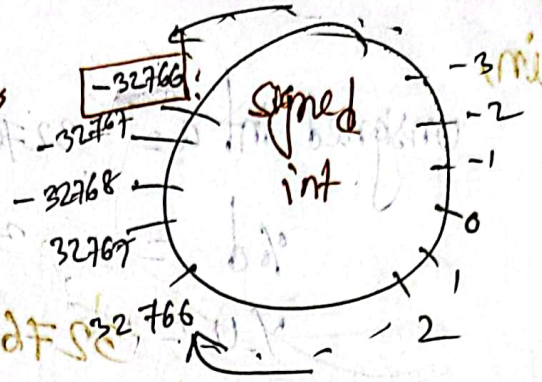
printf("%u", a); where, a = 32768

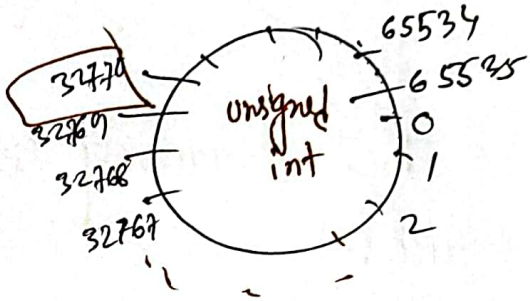
output: a = 32768.



Ex:-02

```
void main ()
{
    int a = 32770;
    printf ("%d", a);
    printf ("\n%u", a);
}
```





output:

a
-32766

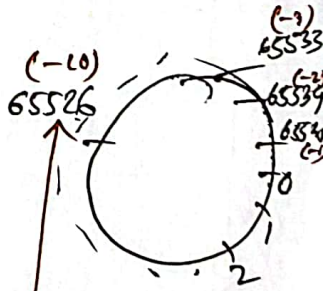
output

a
32770

Ex-03

```
Void main ()
{
    Long int a = 32770;
    printf ("%d", a);
    printf ("%u", a);
}
```

output:
sign integer : 32770;
unsigned int = 32770;



if it is, int a = -10;

%d = → output, a = -10

%u = → output, a = 65526

int a = 0,

%d = 0

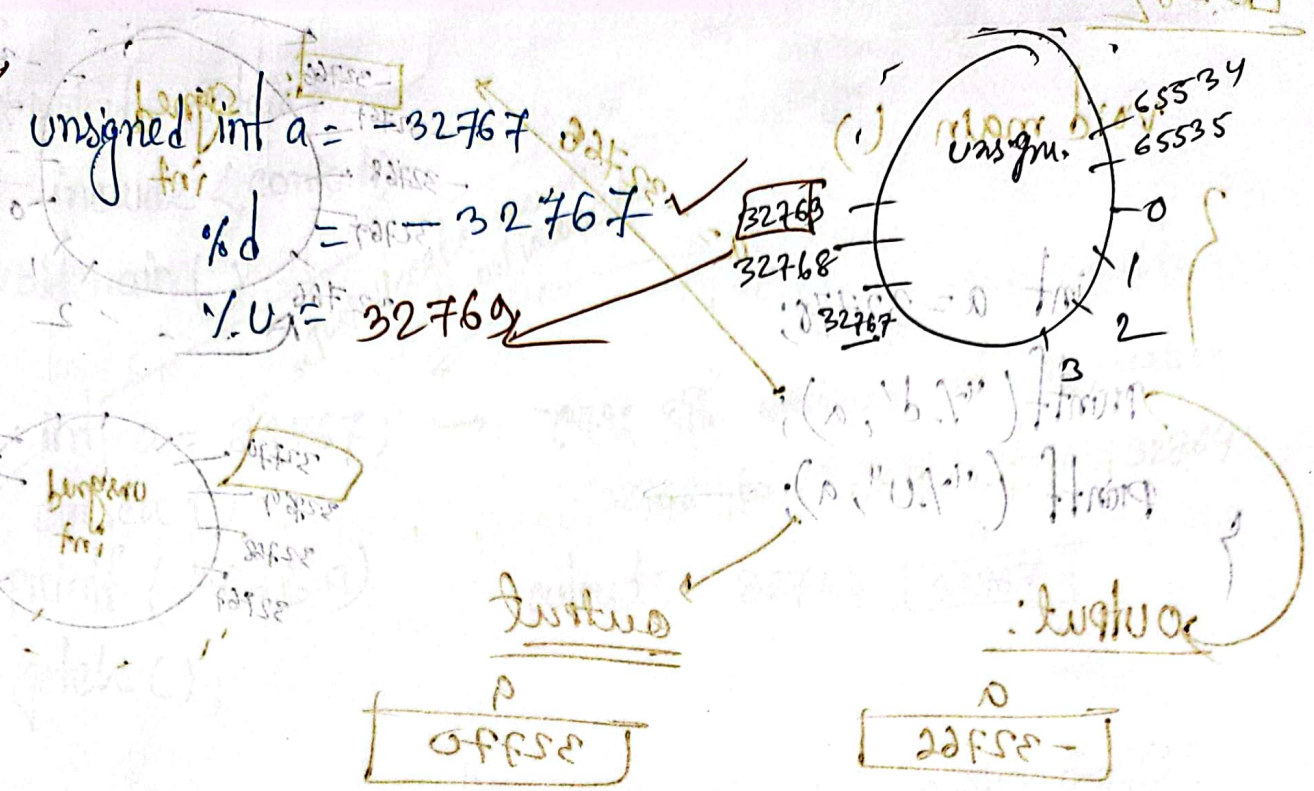
%u = 0

Again,

unsigned int a = -32767

%d = -32767

%u = 32769



C Format Specifier:

↳ The Format specifier is a string used in the formatted input and output functions.

↳ The format string determines the format of the input and output.

→ The commonly used format specifiers in printf() functions are :-

Format Specifier	Description.
%d or %i	It is used to print the signed integer value where signed integer means that the variable can hold both positive and negative values.
%u	Print the unsigned integer means variable can hold only positive value.
%o	Print octal unsigned integer, start with 0
%x	Print hexadecimal unsigned integer, start with 0x, print small letter (a,b,c...)
%X	Print hexadecimal unsigned integer, print uppercase letter
%6d	Prints as decimal number at least 6 characters wide.
%f	Print as floating point.
%6f	Print as floating point at least 6 characters wide.
%2f	Print as floating point, 2 character after decimal point.
%6.2f	Prints as floating point, at least 6 places including the digits after the decimal.
%c	Print as ASCII character.
%s	It is used to print the strings.
%ld	It is used to print the long-signed integer value,

Ex: %6.2f :-

Total width = 6

After decimal point = 2

For decimal point = 1

Before decimal point = 6 - 1 - 2 = 3.

So, ------123456 → total 6 places (3 (remaining) + 1 (for decimal point) + 2 (after decimal point))

"%6.2f" 100.1555 result are showing 100.16

"%8.2f" 100.1555 result are showing 100.16.

Inputs and outputs function :-

Function	Purpose
1. Printf	— prints formatted string.
2. Scanf	— reads formatted string
3. getchar	— reads character
4. putchar	— displays a character
5. gets	— reads a string
6. puts	— displays a string.

Note :

scanf ("%s", str); does not ~~contain~~ read string which contain white space. Hence to use multi word string use gets (str).

→ scanf is delimited by blank, gets is delimited by end of line.

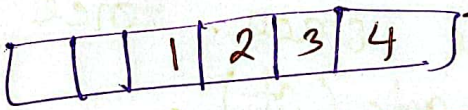
* Formatted input functions = scanf() → int, float, char type formatted input.
* Formatted output function = printf().

* Unformatted Input functions in C
  - getchar()
  - getch()
  - getche()
  - gets()

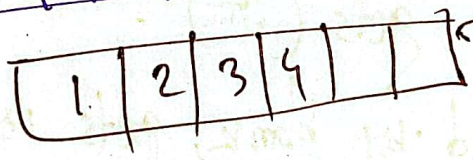
* Unformatted output functions in C
  - putchar()
  - putch()
  - puts().

* printf("%6d", a); when a = 1234

6 space (bits) ফাঁকা রাখবে

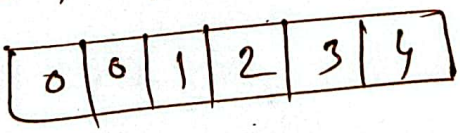


+6d হলে Right থেকে value start হবে



-6d, হলে left থেকে value start হবে

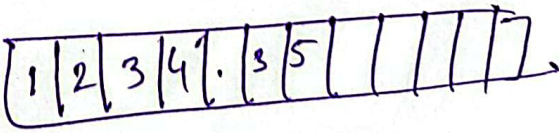
* printf("%06d", a);



* float value

a = 1234.3456;
printf("%10.2f", a);

* printf("%-10.2f", a);



Escape Sequences in C:

↳ An escape sequence in C Language is a sequence of characters that doesn't represent itself when used inside string literal or character.

↳ It is composed of two or more characters starting with backslash \. For example: \n represents new line.

* List of Escape Sequences in 'C'

Escape sequence	Meaning	Elucidation.
\n	New line	shift the cursor control to the next line.
\t	Horizontal Tab	shift the cursor ~~control to the next~~ to a couple of space.
\a	Alarm or Beep	A beep is generated indication the execution of the program to alert the user.
\r	Carriage Return	used to position the cursor to the begining of the current line.
\b	Backspace	→ escape from the normal way the characters are handled by the compiler.
\f	Form Feed.	
\v	Vertical Tab.	
\\	Backslash	
\'	single quote	
\"	Double quote	
\?	Question mark.	
\nnn	octal number	
\xhh	hexadecimal number	
\0	null Character.	

Constants: [Penny]

A constant is a value or variable that can't be changed in the program.

For examples, 10, 20, 'a', 3.4, "C programming"

Types of Constants:

1. Numeric
   1. Integer
      - decimal ((0-9) base 10.)
      - octal (0-7) Base 8
      - Hexadecimal (0-15) base 16.
   2. Floating/Real constants.
2. Characters
   1. Single character constants.
   2. String constants.

1. Integer constants:

Integer = 10, 5, 110, 010.

before zero (0) it is octal number.

- decimal = 05, 17, 24,
- Hexadecimal = 0X, 0x. (0-9, A-F).
  = 0X14
  = 0x7F

} all are hexadecimal number.

✗ 0X7G → not hexadecimal number

- 1234 → integer constants.

but 56,100 = is not an integer number.

→ cannot write comma or special character.

→ Some integers valid or invalid.

① +1234 ✓

② +56,100 ✗ → invalid (comma).

③ −123 ✓

④ $123 → ✗ → special character

⑤ 57 100 ✗ → space.

2. Floating / Real constants:

① +12.56 ✓

② −56.02 ✓

③ 12.56.0 ✗ → two decimal point

Example of some Numeric constants:

① 0 ✓

② 0X, 0x ✓

③ ✗ 25,000 → comma

④ ✗ 1234 → space.

⑤ 012 ✓

⑥ 123 ✓

⑦ 0XAF ✓

⑧ #123 ✗ → #, special character.

2. Character constants.

→ 1. Single character constants.

→ 2. String constants.

1. Single character constants:

↳ Enclosed by single quotation.

'a', 'S', '5', '/0', '1', '@',

↳ All the characters are stored by its ASCII value.

a = 97, A = 65

b = 98, B = 66.

* 5 ≠ '5' → not equal.

5 ↓ numeric const

'5' ↓ character const.

ASCII value

A-Z = 65-90

a-z = 97-122

0-9 = 48-57

special = 0-47, 58-64, 91-96.

char

Examples:

printf("%d", 'a') → print = 97 (int)

printf("%c", 97) → print = a (character)

2. String constants:

↳ Enclosed by double quotation marks.

"Jenny" "abc" "abf" "12345" → string constant.

* "a" ≠ 'a'
* "1" ≠ '1'

"Jenny\0" → Length of string is = 6

↳ Compiler added null character just to mark that it ends of the string.

Declare constants:

1. const keyword
2. #define preprocessor.

1. Const keyword

```
void main ()
{
    const int a = 10;
    printf ("The value of a is : %d", a);
    return 0;
}
```

→ if we wants to change the value of a = 15; it is not possible. a declare const keyword

output: The value of A is : 10

Else.

```
int main ()
{
    const float a = 10;
    a = 15;
    printf ("%d", a);
    return 0;
}
```

output: compile time error: cannot modify a const object.

2. # define preprocessor

define A 10

- capital letter must distinguish variable and global variable.
- body
- no semicolon
- space added.

```
int main ()
{
}
```

* Types of literals:

There are four types of literals that exist in 'C' programming.

1. Integer literal
2. Float literal
3. Character literal
4. String literal
5. Escape Sequence.

1. Integer literal:

↳ It is a numeric literal that represents only integer type values.

↳ It represents the value neither in fractional nor exponential part.

Literals:

↳ Literals are constant values assigned to the constant variables.
↳ literals represent the fixed values. that cannot be modified.
↳ It also contains memory but does not have references as variable.

For example,

```
const int = 10; → constant integer expression.
                  └→ 10 is an integer literal.
```

* Types of literals:

There are four types of literals that exist in 'C' programming.

① Integer literal
② float literal
③ character literal
④ string literal
⑤ Escape Sequence.

1. Integer literal:

↳ It is a numeric literal that represents only integer type values.
↳ It represents the value neither in fractional nor exponential part.

There are three ways / types of integer literals in C:

① decimal (base 10) → 0 to 9, exaple: 45, 67.
② octal (base 8) → 0 to 7. example, 012, 034, 058 etc..
③ hexadecimal (base 16) → 0X; 0x, 0 to 9 (A-F).

2. Floating-point literals:-

↳ A floating point literals is a numeric literal that has either a fractional form or an exponent form.

for example, -2.0
0.0000234
-0.22E-5

3. Characters literals:

↳ A character literal is created by enclosing a single character inside single quotation marks.

For examples: 'a', 'm', '2', '}' etc.

4. Escape Seauences:

↳ Sometimes, it is necessary to use characters that cannot be typed or has special meaning in 'C' programming.
For example: newline (enter), tab, question mark etc

See page NO: 45

5. String literals:

↳ A string literal represents multiple characters enclosed within double-quotes.

↳ It contains an additional character i.e. '\0' (null character), which gets automatically inserted.

↳ The null character specifies the termination of the string.

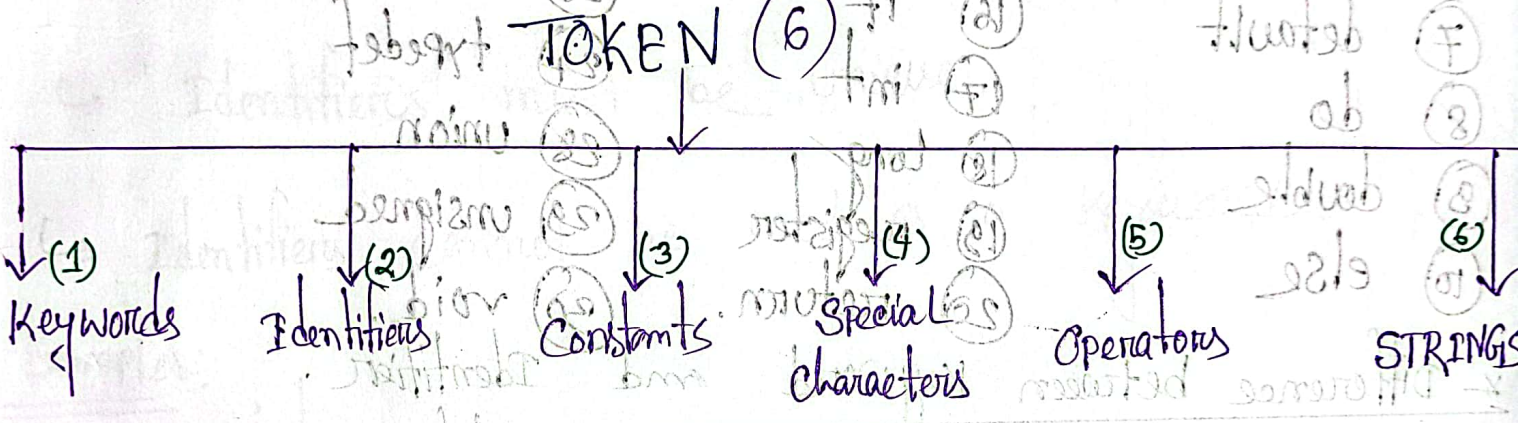
'C' TOKENS:

↳ TOKEN is the smallest unit into a 'C' program.

↳ It is each and every word and punctuation that you come across in your 'C' program.

↳ The Compiler breaks a program into the smallest possible units (Tokens) and processes to the various stages of the Compilation.

* 'C' Token is divided into Six different types.



1. Keywords:

↳ Keywords are predefined, reserved words used in programming that have special meanings to the compiler.

↳ Keywords have fixed meaning, and the meaning cannot be changed.

↳ Each keyword perform a specific function in a program.

↳ Keywords cannot be used as a variable name.

↳ keywords are written in lowercase letters.

* There are a total of 32 keywords in C.

list of keywords

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

* Difference between keyword and Identifier.

keyword	Identifier
(1) keyword is a pre-defined word.	(1) Identifier is a user-defined word.
(2) It must be written in a lowercase letter.	(2) both uppercase & lowercase
(3) Its meaning is pre-defined in the C-compiler.	(3) not defined in 'C' compiler.
(4) It is a combination of alphabetical characters.	(4) alphanumeric character.
(5) It does not contain the underscore character	(5) It can contain the underscore character.

2. Identifiers:

↳ Identifiers in C are used for naming variables, functions, array, structures

↳ Identifiers in 'C' are the user-defined words.

↳ It can be composed of uppercase letters, lowercase letters, underscore, or digits.

↳ but the starting letter should be either an underscore or an alphabet.

↳ Identifiers must be uniques.

↳ Identifiers cannot be used as a keywords.

Examples:

```
int money;
double accountBalance;
```

here, money and accountbalance are identifiers.

* Types of identifiers
  - ① Internal identifier / local variable
  - ② External identifier / global variable.

¥[Jenny]

① Sum ✓
② sum12 ✓
③ Sum_12 ✓
④ 12sumXX ✗
⑤ int float; ✗
⑥ int a; ✓
⑦ _Sum_12 ✓
⑧ Sum__12 ✓
⑨ Sum.12 'x ✗

Rules for naming Identifiers:

① A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.

② Begin with a letter or underscore (_)

③ It should not be a keyword.

④ It must contain no whitespace.

⑤ The name must be meaningful, short, and easy to ~~understand~~ read.

⑥ The length of the identifiers should not be more than 31 characters.

③ constants → Seepage 46 to 54

④ Special Characters in C:

↳ Some special characters are used in C, and they have a special meaning while cannot be used for another purpose.

① Square Brackets []:

↳ opening and closing brackets are used as array element reference.

↳ These indicate Single and multidimensional subscripts.

② Simple brackets / Parenthesis / ():

↳ It is used in function declaration and function calling.

For example, printf() is a pre-defined function.

③ Curly braces {}:

↳ It is used in the opening and closing of the code.

↳ It is also used in opening and closing of loops.

④ Comma (,):

↳ It is used for Separate more than one statements like for Separating parameters in function calls.

↳ Separating variables when printing the value of more than one variable using a single printf statement.

⑤ Colon (:):

↳ It is an operator that essentially invokes something called an initialization list.

⑥ Semicolon (;)

↳ It is known as a statement terminator.

⑦ Hash/pre-processor (#):

↳ It is used for preprocessor directive.

↳ It is basically denotes that we are using the header file.

↳ The pre-processor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.

⑧ Assignment operator (=):

↳ It is used for assign values and for the logical operation validation.

⑨ Asterisk (*):

↳ It is used to create a pointer variable and for the multiplication.

⑩ Tilde (~):

↳ It is used as a destructor to free memory

⑪ Period (.): It is used to access a member of a structure or a union.

⑥ Strings in C

↳ Strings are always represented as an array of characters having null character '\0' at the end of the String.

↳ The null character denotes the end of the String.

↳ Strings are enclosed within double quotes, characters are within Single quotes.

→ The Size of String is a number of characters that the String contains.

① Unary operator
② Binary operator
③ Ternary operator

5. Operators in 'C':

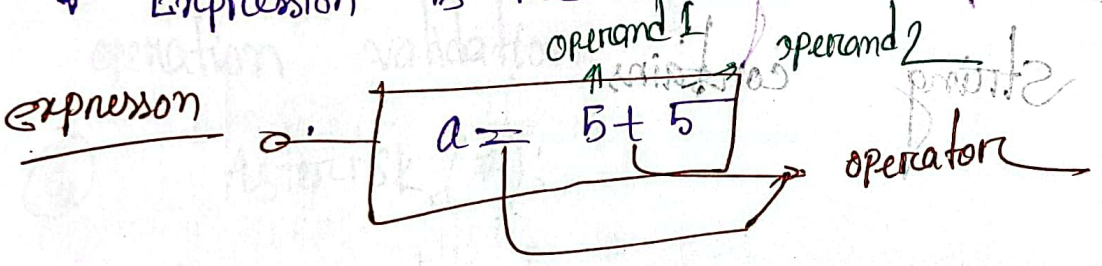
↳ An operator is a symbol that operates on a value or a variable.

↳ An operator is a symbol that tells the compiler to perform specific mathematical or logical functions.

↳ The data items on which the operators are applied are known as operands.

↳ operators are applied between operands.

* Expression is the combination of operators and operands.



* Types of operators based on operands

① Unary operator
② Binary operator
③ Ternary operator.

① Unary operators:

↳ A unary operator is an operator used to operate on a single operand to return a new value.

↳ In other words, it is an operator that updates the value of an operand or expressions value, by using the appropriate unary operators.

↳ In unary, operators have equal priority from right to Left side associativity.

✱ Types of unary operators:

1. Unary Minus (-)
2. Unary plus (+)
3. Increment (++)
4. Decrement (--)
5. Logical Negation (!)
6. Address operator (&)
7. Sizeof () operator.

1. Unary Minus (-):

↳ represented using the symbol (-).

↳ Used to changes the positive number to negative and negative to positive number.

Example:

```
int a,b;
a=5;
b=10;
c=a+(-b)
```

$= 5 + (-10) = 5 - 10$

$c = -5$

also, $a = -b$

$a = -10$

(2) Unary Increment operator (++):

↳ denoted by ++. Symbol represents the operand value is increased by 1.

↳ It can be used in two ways

(i) Pre-increment (++a)

(ii) post-increment (a++)

(i) Pre-increment: → value after 1 more print 2025,

↳ represented as (++a)

↳ which means the value of variable 'a' is increment by 1 before using operand to the expression.

For example:

```
x=10
A=++x;
printf("%d\n",A);
printf("%d\n",x);
```

output

```
A=11
x=11
```

(ii) Post Increment (a++):

↳ denoted by (a++), which means the value of 'a' is incremented by 1 after assigning the original value to the expression or another variable.

Example

```
x = 10
A = x++
printf ("%d", A);
printf ("%d", x);
```

output

```
A = 10
x = 11
```

3. Unary decrement operator (--)

↳ represented by the double minus (--) symbol, and it is used to decrease the operand value by 1 according to the decrement's types.

↳ Pre & post-decrement operator works by same procedure as increment operator.

Example

```
a = 10;
x = --a;
printf ("%d", x);
printf ("%d", a);
```

output:

```
x = 9
a = 9
```

Again

```
b = 20;
y = b--;
printf ("%d", y);
printf ("%d", b);
```

output:

```
y = 20
b = 19
```

4. Sizeof() operator:

↳ The Sizeof is a keyword used to find the size of different data types or operands like int, float, char, double etc.

Syntax:

Sizeof (data type / data_variable)

Example:

```
int x;
float y;
char ch;
double z;
printf("%d", sizeof(x));
Printf("%d", sizeof(y));
printf("%d", sizeof(ch));
printf("%d", sizeof(z));
```

output:

4
4
1
8

5. Logical Not (!) Operator:

↳ The logical not operator is used to reverse the given condition.

↳ if the operand is true, logical not operator (!) return false.
if false, (!) return true.

Example:

!(x>y)
here, x=11, y=10.
11>10
condition true
(!) return False,
output: False

!(x>y)
x=10, y=11
(10>11)
→ condition False.

output:

true.

6. Address of operator (&):

↳ Used to find the address of a variable defined in Computer memory.

3. Ternary operator / conditional operator:

↳ Conditional operator is also known as a ternary operator.

↳ The conditional statement are the decision making statements.

↳ It is represented by two symbols '?' and ':'.

→ As conditional operator works on three operands, So, it is also known as the ternary operator.

→ The behavior of ternary operator is similar to if-else statement.

Syntax: condition → যদি সত্য হয় তবে expression2 answer দেবে। যদি মিথ্যা হয় তবে expression3 answer দেবে।

```
Expression1 ? expression2 : expression3;
```

Example

```
int a=10, b=15;
X = (a<b) ? a : b;
```

10<15 answer

if, a=15, b=10; 15>10 ? a : b; answer

also write:

```
if (a<b)
{
  x=a;
}
else
{
  x=b;
}
```

2. Binary operator:

↳ The binary operator is an operator applied between two operands.

Following is the list of the binary operators

1. Arithmetic operators
2. Assignment "
3. Increment and Decrement operator
4. Logical operator
5. Relational operator / Comparison operator
6. Bitwise operator
7. Special operator

(1) Arithmetic operator: +, -, *, /, %

↳ Arithmetic operator is used to performing mathematical operations such as addition, subtraction, multiplication, division, modules etc on the given operands.

Program:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a=10, b=7;
printf("a+b=%d", a+b);
printf("a-b=%d", a-b);
```

printf("a*b = %d", a*b); = 70

printf("a/b = %d", a/b); = $\frac{10}{7}$ = 1.0 - print = 1 (int value)

printf("a%b = %d", a%b); remainder = 3

For Modulus:

① All input must be integer value

② a = -10, b = 7.

a%b = ③ -10%7 = -3

if. a = 10, b = -7.

a%b = 10/-7 = 3

* ভাগশেষের চিহ্ন সবসময় মূল্যের value অনুযায়ী হবে।

* মূলের value (-) হলে answer = (-) হবে

* " " (+) " " = (+) হবে।

* যেমন, int a = 10;
float b = 7;

যেকোন value int.
একটি float হলে,
answer হবে = float

* Precedence of arithmetic operators.

*, /, % — ①

+, - — ②

Left to Right

2. Assignment operator:

- Assignment operators are used to assign value to a variable.
- The Left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value.
- Value of the right side must be of the same data type as the variable on the Left side otherwise the compiler will raise an error.

Operator	Example	Same as
1. =	a=b	a=b
2. +=	a+=b	a=a+b
3. -=	a-=b	a=a-b
4. *=	a*=b	a=a*b
5. /=	a/=b	a=a/b
6. %=	a%=b	a=a%b

N.B: ① Precedence or associativity

Right to Left

② a= b+(c*10)

a += c*10 ✗ লিখা যাবে না। কেবল = এর আগে Same variable থাকলেই লিখা যাবে।

a= a+(c*10)

a+=(c*10) ✓

3. Increment and Decrement operator

See page No: 64 & 65 (unary operator).

→ increment, decrement-ൽ value float ആയും ആകും!

N.B:

y = --x;	y = x--;
Same as	Same as
x = x-1;	y = x;
y = x;	x = x-1;

Example:

```
void main()
{ int a=5, b, c, d;
      b = ++a;     b=6, a=6,
      c = a++;     c=6, a=7.
      d = ++a;     d=8,  a=8;
   printf("%d", a);   a=8
}
```

4. Relational operators:

→ Checks the relationship between two operands

↳ If the relation is true, it returns = 1

↳ If the relation is false, it returns = 0

→ Used in decision making and loops.

operator:

$<$
$>$
$<=$
$>=$
] Same precedence
Associativity, L → R

$==$
$!=$

Operator	Example
1. $<$	$5<3 = 0$ (False)
2. $>$	$5>3 = 1$ (true)
3. $<=$	$5<=3 = 0$ (False)
4 $>=$	$5>=3 = 1$ (true)
5. $==$	$5==3 = 0$ (False)
6. $!=$	$5!=3 = 1$ (True).

N.B. ① comparison এর ক্ষেত্রে যদি operand এ কোন, তবে expression এর কোন মান হবে।

② int, float, char এর কোন compare হয়।

③ string এ কোন relational operator কাজ করে না।

④ Float এ avoid করার try করব।

Example:

① $3 < 5 = \text{true} = 1$.

② $3 > 5 = 0$

③ $3 <= 3 = 1$

④ $3+5 < 5+5 = 1$.

⑤ $a+b < b+c$

⑥ 'a' < 'b' = 1 (true)
'a' → 97, 'b' → 98

⑦ "Basar" < "Leeforce".
↳ array of string position/address compare করা হয় না
↳ value only compare করা হয়।

Syntax:

Arithmetic Expression/operator 1 → Relational operator → Arithmetic Expression/operator 2

① $-14 > 3 = 0$

② $4.5 < 4 = 0$.

③ Format specifier এর = %d, তার answer হবে
true = 0, false = 1

Example-01

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int a=18, b=9;
  printf("%d", a<b);
  printf("%d", 'c'<'b');
}
```

if a==b, output=0
if a!=b, output=1.

= 0

= 0

'c' → 99, 'b' → 98

Example-02

a	b	c	d	e
18	10	9	10	10

```
void main()
{
  int a=18, b=9, c, d, e=10;
  c=b++;
  d=b;
  printf("%d", a<b<c>d);
  printf("%d", b==e);
  printf("%d", e+1>e);
  printf("%d", a+c==b>e<c+d);
}
```

L → R

c = b++ → c = 10, b = 10

d = b → d = 10

a<b<c>d → 0 1 0 → output = 0

b==e → 10==10 → output = 1

e+1>e → 11>10 → output = 0

a+c==b>e<c+d → 27==10>10<19

27==0<19

27==1

= 0 (output)

Arithmetic high precedence

L → R

Associativity

Higher Precedence: < > <= >=

Lower Precedence: == !=

Example – 3

```
void main()
{ int a=18, b=9, c, d, e=10, f;
  c= b++;
  d= b;
  f= a>b>d<c;
  printf("%d", f!=1);
  printf("%d", a+c==b>=e<c+d!=1);
}
```

a=18, b=10, c=9, d=10, e=10.

b++ → L→R

1 0 1

output = 0

27 == 10 >= 10 < 19 != 1

27 == 1 < 19 != 1

27 == 1 != 1

0 != 1 = 1

output

1 0

And, check in codeblocks

char R.operator char

'<' '>' '<'

74 76 0

5. Logical operators:

↳ Logical operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration.

↳ The result is true (1) or False (0).

Operator	Meaning
1. && (Logical And)	→ True only if all operands are true.
2. || (Logical OR)	→ True only if either one operand is true.
3. ! (Logical NOT)	→ True only if the operand is false.

Example: Logical AND (&&) → যদি একটা expression '0' হয় answer = 0,

```
int a=10, b=5;
1. a==b && b<a = 0(output)
2. a && b
3. 10 && 5 = 1
4. 10 && 0 = 0
```

Example: result = a>b && b!=10 && b<11 && a>5 = 1(output)

&& a<5 = 0(output)

if

2. Logical OR (||):

int a=10, b=5;

→ যেকোনো 1 হলেই answer = 1 (OR এ এরকম হয়)

① a>b || b==4; output = 1

② a<b || b==4; output: 0

3. Logical NOT (!):

→ যেকোনো value answer = 1 ('0' বাদে)

① !5 = 0

② !0 = 1

③ !(a>b) = 0

Example -1

```
main ()
{ int a=4, b=6, result;
  result = a>b && printf("jenny");
  printf("%d", result);
}
```

→ যেকোনো 0 হলেই answer = 0 (&& এর জন্য)

a<b && printf("Jenny");

output = 1

(0 বাদে সবই = 1)

Example-2

&& = higher precedence
|| = lower

```
main()
{
int a=4, b=6, result;
result = a>b && printf("Jenny") || printf("lectures");
}
```

result = 0 && printf("Jenny") || printf("lectures");
result = 0 || printf("lectures");
result = 1

output: lectures 1

Again:

① result = a>b && printf("Jenny") || printf("lectures") && printf("JK");
result = 0 || 1 && 1
result = 1

② result = a>b && printf("Jenny") || printf("lectures") || printf("JK");
result = 0 || 1 || 1
result = 1

Example- 03

```
main ()
{ int a=4, b=6, result;
  printf("%d", 4&&!0); = 1
  printf("%d", 4&&0); = 0
}
```

higher precedence. !
&&
||

Example-04

```
main ()
{ int a=10, b=5, result;
  result= (a>b) && a++;
  printf("%d", result); = 1
  printf("%d", a); = 11
}
```

output

if
result= (a<b) || a++;
result = 1
a = 11
else
result= (a>b) || a++;
result = 1
a = 10

প্রথমটি true হলে 1, তাই = 1
|| এর পরের
expression
execute হয় না।

H.W

```
int a=1, b=6,
result= a-- && ++b ; = 1, a=0 b=7
result= --a && ++b; = 0 a=0 b=6
```

দ্বিতীয়টি
execute হবে না

6. Bitwise operators:

↳ Used to perform the operations on the data at the bit-level.

↳ It is also known as bit-level programming.

↳ It consists of two digits, either 0 or 1.

→ mainly used for processing faster and saves power.

operators

&
|
^ → exclusive OR
~ → unary operator. (NOT) → one operand.
<<
>>

① &:

```
int a = 10, b = 5
a & b
10 & 5
 ↓    ↓
1010  0101
```

① দুইটা value দুইটাকে binary তে convert করতে হবে
② And করতে হবে
③ শেষে answer কে decimal করতে হবে

2^3 2^2 2^1 2^0
8 4 2 1 (int 4 bit)
1 0 1 0 → 10
0 1 0 1 → 5
(&) 0 0 0 0 = 0 (decimal)

② | = a | b

10 | b

(|) 1 1 1 1 = 15 (decimal)

③ ^ = a ^ b

= 10 ^ 5

(^) 1 1 1 1 = 15

Character value ko can 8 bits

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1
	1	1						1
		1	1	0	0	0	0	1

'a'

97 = 01100001

Precedence

1. Arithmetic operator.
2. Bitwise
3. logical.

Example:

```
int main ()
{
    int a=10, b=6;
    printf("%d", a&b);
    printf("%d", a|b);
    printf("%d", a^b);
    printf("%d", a&b && b+1||0);
}
```

	8	4	2	1
10	1	0	1	0
6	0	1	1	0
(&)	0	0	1	0 = 2
(|)	1	1	1	0 = 14
(^)	1	1	0	0 = 12

a&b && b+1||0

2 && 7||0

1||0 = 1

Again,

a&b&&b+1||0||b++;

2 7

1||0||6

1||6 = 1

NOT execute.

printf(b) = 6

1 && 6 → execute

printf(b) = 7

Part - 2

<< >> ~

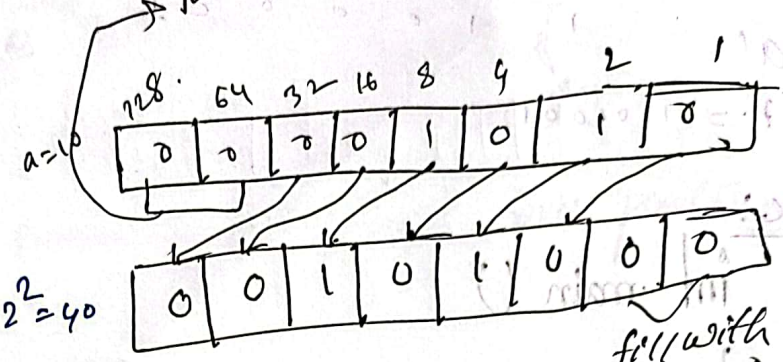
① << :- Left shift.

Var << 2

```
int a=10;
c = a<<2;
```

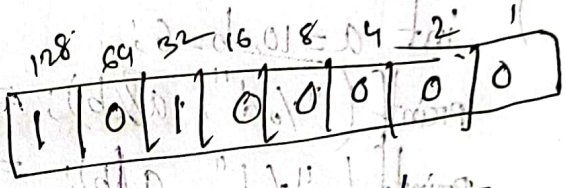
$10 \times 2^2 = 40$

c = 40



Again, c = a<<4;

c = 160



Left shift - ↑ multiply.

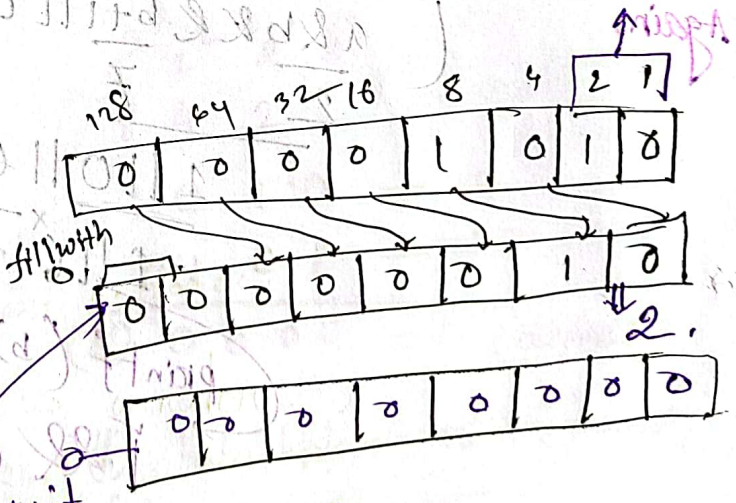
shortcut: $10 \times 2^4 = 10 \times 16 = 160$.

Var x 2^{shiftbit}

② >> :- right shift →

Var >> 2

```
int a=10;
c = a>>2;
c = 2
if, c = a>>4;
c = 0.
```



shortcut: $\frac{a}{2^{\text{shiftbit}}}$ (answer 2 or 1)

① a>>4 $= \frac{10}{2^4} = \frac{10}{16} = 0$

② a>>2 $= \frac{10}{2^2} = \frac{10}{4} = 2.$

float point ans নিয়ে হবে না

③ ~ (bitwise NOT):

```
int a = 5
b = ~a
```

8	4	2	1	
0	1	0	1	→ a(5)
1	0	1	0	→ bitwise NOT

→ b = 10

computer-এ যার মানের আগে Negative চিহ্ন থাকে।

$$\sim a = -(a+1)$$
$$= -(6) = -6$$

* if a = 10.

$$a = -(10+1)$$
$$= -11$$

* direct value,

~5
~10

H.W

a+1 << 2;
(a+2) >> 1
~(a+3)
~13 = -(14)

a << 1 → ডান দিকে
5 << 1 → same দিকে
5 এর 1 bit left shift করবে

10 >> 2
→ 10 কে 2bit right shift.

7. Special operators:

Least precedence (সবচেয়ে নিচে) (L → R) associativity ; sizeof, &, ., →

i. Comma operator (,):

↳ Comma Operators are used to link related expressions together.

```
int a, c = 5, d;
```

↳ Comma operator has the lowest precedence of any 'C' operator.

↳ Comma acts as both operator and Separator.

~~int a=10, b=5.~~

* `int a, b, c;` → declare করলে comma Separator হিসেবে কাজ করে।

```
int a;
int b;
int c;
```

* `int a=10, b=5, c=10, d;` → separator.

operator

```
int a;
a = 5,4;
```

- higher precedence → `=`
- `,` → least precedence.
- → operator.
- → এটার প্রথমে কাজ করবে

$a = 5$

* if, `a = (5,4);` → bracket থাকলে।
  1. 5 প্রথমে evaluate করবে, reject করবে
  2. answer = `a=4;`

* ✗ int a = 5,4;
  - → initialize করার সময় / declare
  - → একসাথে declare, value দিলে error দেখাবে।
  - ↓ declare করার সময় comma Separator হিসেবে কাজ করে। operator না।

output = error

inta = 5, int 4; → (incorrect)

✓ OK

* int a = (5,4); → ( ) bracket দিলে হবে
  - → 5 evaluate, reject. Separator না

a = 4

both correct

① int a;
a = 5,4;

or,

int a = (5,4); also correct

② int a;
a = (5,4);

but, int a = 5,4; (Wrong)

Example-01

int a;

① a = (printf("Jenny"), 2); → evaluated → assign a = 2

After execution

Jenny
2

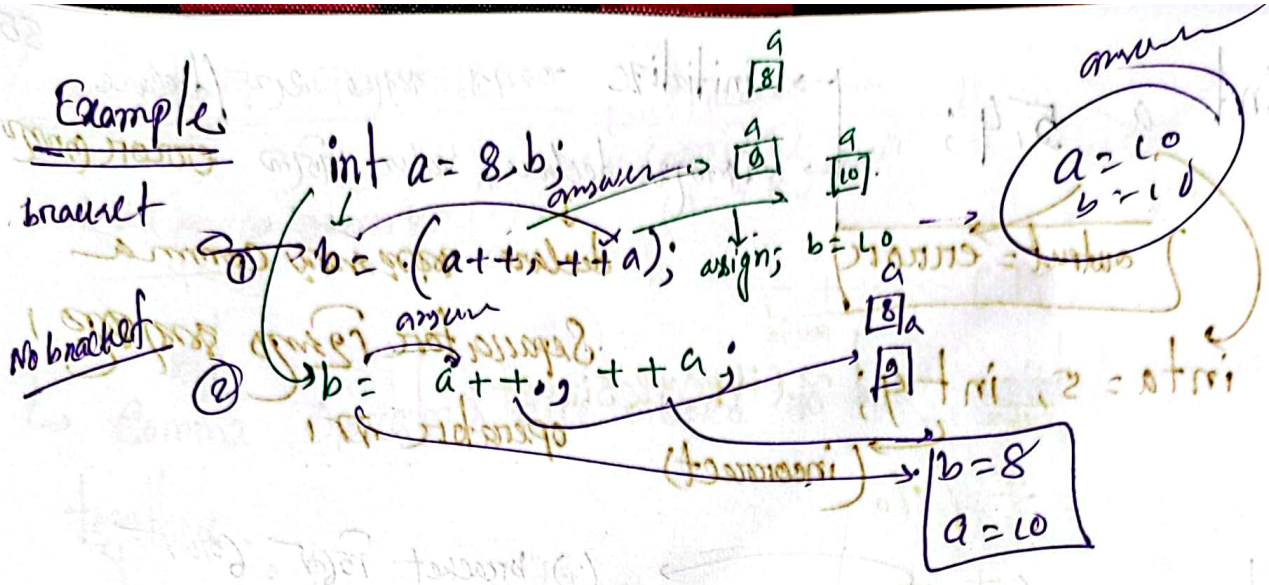
② a = (printf("Jenny"), 2,3); → Jenny evaluated, 2, evaluate. still operator (comma) assign, a = 3

a = 3

After execution

Jenny
3

Example:



H.W: b = (a++, ++a, a>>2)

* Precedence and associativity of C operators.

Category	Operators	Associativity
1. Postfix	() [] -> . ++ -- (postfix)	L → R
2. Unary	++ -- (prefix) + - ! ~ (type) * & sizeof.	R → L
3. Multiplicative	* / %	L → R
4. Additive	+ -	L → R
5. Shift	<< >>	L → R
6. Relational	< <= > >=	L → R
7. Equality	== !=	L → R
8. Bitwise AND	&	L → R
9. Bitwise XOR	^	L → R
10. Bitwise OR	|	L → R
11. Logical AND	&&	L → R
12. Logical OR	||	L → R
13. Conditional/Ternary	?:	R → L
14. Assignment	= += -= *= /= %= >>= <<= &= ^= |=	R → L
15. Comma.	,	L → R

Example:

① $2 + 3 * 4$

$2 + 12 = 14$

② $2 + 3 * 4 / 2 * 12$

$2 + 12 / 2 * 12$

$2 + 6 * 12$

$= 74$

Same precedence */

associativity L → R

<< >>

HW $a = 0, b = 1, c = 1$

$a * (5+b) / 2 - c++ * b$

③ $a = 3 * 4 \% 5 / 2$

$= 12 \% 5 / 2$

$= 2 / 2$

$= 1$

④ $a = 3 * (4 \% 5) / 2$

$= 3 * 4 / 2$

$= 12 / 2 = 6$

⑤ $a = 3 * 4 + 5 * 6$

$= 12 + 30$

$= 42$

$a = 3 * (4+5) * 6$

$= 3 * 9 * 6$

$= 162$

Control statements in C:

↳ Control statements are the statements that change the flow of execution of statements.

There are three types of control statements:

1. Conditional | Selection statements
2. Iteration | Loop statements.
3. Jump statements.

1. Conditional | Selection statement:

↳ Conditional statements in C Programming are used to make decisions based on the conditions.

↳ Conditional statements execute sequentially when there is no condition around the statements.

↳ If you put some condition for a block of statements, the execution flow may change based on the result evaluated by the condition.

This process is called decision making in 'C'

Decision making statements available in C or C++ are:

1. if Statement
2. if-else statements
3. nested if statements
4. if-else-if ladder.
5. Switch statements
6. Jump statements
   - break
   - continue
   - goto
   - return.

1. if Statement:

Syntax:

```
if (test expression/condition)
{
    // code
}
```

How if statement works?

The if statement evaluates the test expression inside the parenthesis ().

(i) If the test expression is evaluated to true, statements inside the body of if are executed.

(ii) If the test expression is evaluated to false, statements inside the body of if are not executed.

Expression is true

```
int test=5;
if (test<10)
{
// codes
}
// codes after if
```

Expression is false

```
int test= 5;
if (test>10)
{
// codes.
}
// codes after if
```

2. if-else statement:

Syntax:

```
if (test expression) {
  // run code if test expression is true.
}
else {
  // run code if test expression is false
}
```

How if..else statement works?

If the test expression is evaluated to true,

(i) statements inside the body of if are executed.

if the test expression is false.

(i) statement inside the body of else are executed.

Expression is true

```
int test=5;
if (test<10)
{
  // body of if
}
else
{
  // body of else
}
```

Expression is false

```
int test=5;
if (test>10)
{
  // body of if
}
else
{
  // body of else
}
```

3. If else-if ladder statement.

↳ The if-else-if ladder statement is an extension to the if-else statement.

↳ It is used in the scenario where there are multiple cases to be performed for different conditions.

↳ The if...else ladder allows you to check between multiple test expressions and execute different statements.

Syntax

```
if (condition 1){
    // statements
}
else if (condition 2){
    // statements
}
else if (condition 3){
    // statement.
}
:
else {
    // statements.
}
```

N.B:
যেকোন condition true হলে program execute হবে।

4. Nested if...else

↳ If is possible to include an if...else statement inside the body of another if...else statement.

Syntax:

```
if (condition)
{
    // execute
    if (condition2)
    {
        // execute
    }
    else {
        // execute code
    }
}
else {
    // execute code
}
return 0;
}
```

Jump statements:

↳ Used to interrupt the normal flow of the program or escape a particular section of the program.

↳ The main uses of jump statements are to exit the loops like for, while, do-while also switch case and executes the given or next block; skip the iterations of the loop, change the control flow to specific location, etc.

Types of Jump statement

1. Break
2. Continue
3. Goto
4. Return

1. Break statement:

↳ Break statement exists the loops like for, while, do-while immediately, brings it out of the loop, and starts executing the next block. It also terminates the switch statement.

↳ If we use the break statement in the nested loop, then first, the break statement inside the inner loop will break the inner loop. Then the outer loop will execute

as it is, which means the outer Loop remains unaffected by the break statement inside the inner Loop.

Syntax :-

```
// specific condition, Loop or switch case.
break;
```

How break works?

①
```
while (testExpression){
  // codes
  if (condition to break){
    break;
  }
  // codes
}
```

②
```
do{
  // codes
  if (condition to break){
    break;
  }
  // codes
} while (testExpression);
```

③
```
for (init; condition; ++){
  // codes
  if (condition to break){
    break;
  }
  // codes
}
```

Example-01

```
int main ()
{ int i;
for (i=0; i<10; i++)
{ printf ("%d", i);
  if (i==5)
    break;
}
printf ("came outside of Loop i=%d ", i);
}
```

Output:

0 1 2 3 4 5 came outside of Loop i=5

Example: Nested Loop

```
int main (){
int i=1, j=1;
for (i=1; i<=3; i++){
for (j=1; j<=3; j++){
printf ("%d%d \n", i, j);
if (i==2 && j==2){
break;
}}
return 0;
}
```

end of Loop

Output

```
1 1
1 2
1 3
2 1
2 2
3 1
3 2
3 3
```

2. Continue statement:

↳ continue in jump statement in C skips the specific iteration in the loop.

↳ It is similar to the break statement, but instead of terminating the whole loop, it skips the current iteration and continues from the next iteration in the same loop.

↳ Using the continue statement in the nested loop skips the inner loop's iteration only and doesn't affect the outer loop.

Syntax

// Loop statements / conditions...

Continue ; ⟶ break & continue are keyword always write in lowercase letter.

Example-1

```
int main ()
{ int i;
  for (i=1; i<=7; i++)
  { if (i==3) // continue condition
    {
      continue;
    }
    printf ("value = %d \n", i);
  }
}
```

Output

```
1
2
4
5
6
7.
```

→ 3 SKIPPED

Example-2 nested Loop

```
int main (){
int i=1; j=1;
for (i=1; i<=3; i++){
for (j=1; j<=3; j++){
if (i==2 && j==2){
  continue; // will continue loop j only.
}
printf ("%d %d \n", i, j);
}
{ // end of for Loop.
return 0;
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

3. Goto statement:

↳ The goto is one of the control statements in c/c++ that allows the jump to a labelled statement in the same function.

↳ The Labelled statement is identified using an identifier called a label.

↳ It is preceded by an identifier followed by a colon (:)

Label:

Syntax

Syntax 1	Syntax 2
goto label;	Label:
.	.
.	.
.	.
Label:	goto Label;

Program:

```
int main(){
int i, j;
for (i=1; i<5; i++){
 if (i==2)
    goto there;
   printf ("%d\n", i);
}
there:
printf ("Two");
return 0;
}
```

output:

```
1
Two
```

*Reasons to avoid goto statements.

(i) It is hard for the other person to trace the flow of the program and understand it.

(2) code hard to modify because it references the different labels, so rewriting is the only solution.

(3) it could only be used inside the same function.

4. Return statement:

↳ Return jump statement is usually used at the end or terminate it with or without a value.

↳ It takes the control from the calling function back to the main function (main function itself can also have a return).

↳ return can only be used in functions that is declared with a return type such as int, float, double, char, etc.

↳ Functions declared with void type does not return any value.

Program:

```
#include <stdio.h>
char func(int ascii){
    return ((char) ascii);
}
int main(){
int ascii;
char ch;
printf("Enter any ascii value: \n");
scanf("%d", &ascii);
ch = func(ascii);
printf("The character is: %c", ch);
return 0;
}
```

output

```
Enter any ascii value: 97
The character is: a
```

```
Enter any ascii value: 10
The character is: 
```

Switch statement in C:

↳ Switch statement in C tests the value of a variable and compares it with multiple cases.

↳ Once the case match is found, a block of statements associated with that particular case is executed.

↳ Each case in a block of a switch has a different name/number which is referred to as an identifier.

↳ The value provided by the user is compared with all the cases inside the switch block until the match is found.

↳ If a case match is NOT found, the default statement is executed, and the control goes out of the switch block.

↳ Switch statement is an alternative to if-else-if ladder.

Rules for Switch statement in C Language

i. There can be One or N numbers of cases.

ii. The values in the case must be unique

iii. Switch expression must be of an integer or character type.

iv. The case value must be an integer or character constant.

v. The case value can be used only inside the switch statement.

vi. The break statement is optional.

valid switch	Invalid switch	valid case	Invalid case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x,y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

Syntax:

```
Switch (expression)
{
  case constant1:
      // statements
  break; // optional.
  case constants2;
    // statements
  break;
  .
  .
  default
      // default statements.
}
```

Example-1 Program to create a Simple Calculator.

```
#include <stdio.h>
int main (){
  char operation;
  double n1, n2;
  printf ("Enter an operator (+,-,*,/): ");
  Scanf ("%.c, & operation);
  printf (" Enter two operands: ");
  scanf ("%lf %lf", &n1, &n2);
```

Switch (operation)

{

case '+';
printf("%.1lf + %.1lf = %.1lf", n1, n2, n1+n2);
break;

case '-';
printf("%.1lf - %.1lf = %.1lf", n1, n2, n1-n2);
break;

case '*';
printf("%.1lf * %.1lf = %.1lf", n1, n2, n1*n2);
break;

case '/';
printf("%.1lf / %.1lf = %.1lf", n1, n2, n1/n2);
break;

// operator does not match any case

default:
printf("Error! operator is not correct");

}

return 0;

}

Example-02

```
#include <stdio.h>
int main (){
    int num = 8;
    switch (num) {
        case 7:
            printf ("value is 7");
            break;
        case 8:
            printf("value is 8");
            break;
        case 9:
            printf ("value is 9");
            break;
        default:
            printf ("out of range");
            break;
    }
    return 0;
}
```

output:

value is 8

Example-3 default value print.

```
#include <stdio.h>
int main (){
 int language = 10;
   switch (language){
   case 1 :
      printf ( "C#\n");
      break;
   case 2:
      printf ("c\n";
      break;
   case 3:
      printf ( "c++\n");
      break;
   default:
      printf ("other programming language");
```

output:

other programming language

Example-4 Nested Switch in C

```
#include <stdio.h>
int main (){
    int ID = 500;
    int password = 000;
    printf (" please Enter your ID :\n");
    Scanf ("%d", &ID);
    Switch (ID) {

    Case 500:
        printf ("Enter your password: \n ");
        scanf ("%d", & Password);
        Switch (password) {

            case 000;
            printf ("Welcome Dear programmer \n");
            break;
        default:
            printf ("incorrect password");
            break;
    }
    break;
default:
    printf ("incorrect ID");
    break;
}
```

output

Please Enter your ID :
500
Enter your password :
000
Welcome Dear programmer.

Q. Why do we need a switch case?

There is one potential problem with the if-else statement which is the complexity of the program increases whenever the number of alternatives path increases.

↳ If you use multiple if-else constructs in the program a program might become difficult to read and comprehend.

↳ Sometimes it may even confuse the developer who himself wrote the program.

The solution to this problem is the switch statement.

Q. Which Loop to Select?

Selection of a Loop is always a tough task for a programmer to select a Loop do the following steps.

① . Analyze the problem and check whether it requires a pre-test or a post-test loop.

② If pre-test is required, use a while or for loop.

③ If post-test is required; use a do... while loop.

Loop in C:

↳ Looping statements in C execute the sequence of statements many times until the stated condition becomes false.

↳ A Loop in C consists of two parts, a body of Loop and a control statement.

↳ The purpose of C Loop is to repeat the same code a number of types.

Types of Loops

→ while, for Loop

① Entry/pre controlled Loop → Check condition first before execute the body statement.

② post /Exit controlled loop → a condition is checked after executing the body of a Loop.
↳ do-while Loop.

'C' Programming language provides us with three types of Loop constructs:

1. While Loop
2. do-while
3. for Loop

= Entry controlled Loop

= Exit-controlled loop

Q. Why use Loops in C

i. It provides code reusability.

ii. We do not need to write the same code again and again.

iii. We can traverse over the elements of data structures. (array or linked lists).

1. While Loop → Entry-controlled loop.

↳ In while loop, a condition is evaluated before processing a body of the loop.

↳ If a condition is true then and only then the body of a Loop is executed.

Initialization.

```
while (condition)
{
  statement 1;
  :
  statement n;
  update/modify;
  statement abc.
}
```

Syntax

```
while (condition) {
    // code to be executed.
}
```

* How while Loop works?

① While Loop evaluates the condition/text expression inside the parenthesis.

② If condition is true, body statements are executed then, condition is evaluated again.

③ The process goes on until condition is false.

④ If testExpression is false, the Loop terminates (ends).

Example-01

```
// print numbers from 1 to 5
#include <stdio.h>
int main(){
  int i=1;
  while(i<=5){
    printf("%d\n", i);
    ++i;
  }
  return 0;
}
```

output

```
1
2
3
4
5
```

Example-2

```
void main()
{
  while()
  {
    printf("hello javatpoint");
  }
}
```

output

Compile time error:
while loop can't be empty.

Example-03

```
void main ()
{
int x=10, y=2;
while( x+y-1)
{
Printf("%d%d", x--,y--);
}
}
```

output

Infinite Loop.

Infinitive while Loop:

```
While(1){
// statement
{
```

→ If the expression "passed" in while Loop results in any non-zero value then the Loop will run the infinite number of times.

2. do-while Loop:

↳ The do..while loop is similar to the while Loop with one important difference.

↳ The body of do...while Loop is executed at least once. Only then, the test expression is evaluated.

Syntax

```
do {
    // the body of the Loop.
}
while (condition/ testExpression);
```

↳ In do.....while loop, the body of a loop is always executed at least once.

↳ After the body is executed, then it checks the condition.

↳ If the condition is true, then it will again execute the body of a Loop otherwise control is transfered out of the Loop.

Example of

output

```
int main()
{
  int num=1;
  do
  {
    Printf ("%d\n", 2* num);
    num++;
  } while (num<=10);
  return 0;
}
```

2
4
6
8
10
12
14
16
18
20

* Infinitive do--- while loop

```
do {
    //statement
} while (1);
```

do-while Loop will run infinite times if we pass any non-zero value as the conditional expression.

3. for loop in C:

↳ The for loop in C language is used to iterate the statements or a part of the program several times.

↳ It's frequently used to traverse the data structures like the array and linked list.

Syntax:

```
for ( initialization statement; testExpression; updatestatement)
{
    // statement inside the body of the loop.
}
```

How for loop works?

i. Initialization statement is executed only once.

ii. Then, the test expression is evaluated. If condition is false, loop terminate.

iii. if the condition is true, body statement executed and the update expression is updated.

iv. Again the condition is evaluate.

The process goes on until the test expression is false.

Example-01 Print number 1 to 10.

```
int main () {
 int i;
for (i=1; i<11; ++i)
 { printf ("%d", i);
 }
 return 0;
}
```

output:

1 2 3 4 5 6 7 8 9 10

Example-2

```
int main()
{ int num, count, sum=0;
 Printf ("Enter positive integer:");
 Scanf ("%d", &num);
 for (count=1; count<=num; ++count)
 { sum += count;
 printf ("sum=%d", Sum);
 return 0;
}
```

output:

Enter: 10

Sum = 55

⊞ control statement in c : Jenny

* If statement:

Example-1

```
void main ()
{ int a;
  printf ("Enter a ");
  scanf ("%d", &a);
  if (a)
    printf ("Inside if block");
    printf ("value of a= %d", a);
  printf ("out of if block");
}
```

0 বাদে যা দিবো সবই true হবে → Condition true হলে

Input: 5

Output:
inside if block
~~Print~~ value of a = 5
out of if block

~~outP~~ Input: 0

Output:
value of a = 0
out of if block

N.B: ① যদি if এর পর bracket { } না দেই তাহলে ~~শুধুমাত্র টাই~~ সবসময় প্রথম line টা print হবে। বাকী গুলো if এর under-এ consider হবে না। out of if block-এ থাকবে।

② condition true হলে, { } না থাকলে, if এর পরে অনেকগুলো printf থাকলে, ~~শুধুমাত্র~~ সবগুলো printf টা print হবে এবং if এর বাইরেরও গুলো print হবে।

③ condition wrong হলে, { } না থাকলে, if এর পরে অনেকগুলো printf থাকলে, শুধু if এর পরের প্রথমটা print হবে না। বাকী সবগুলো print হবে।

③ bracket { } থাকলে, এর ভেতরের সবগুলো if এর under-এ থাকবে।

③ Curly braces (bracket) { } থাকলে condition true হলে ঐ if block এর ভেতরে যা আছে সব print হবে।

```
void main()
{ int a;
printf("Enter a");
scanf("%d", &a);
if (a)
{
 printf("inside if block");
 printf("value of a = %d", a);
}
printf("out of if block");
}
```

Input: 5
output:
inside if block
value of a = 5
out of if block

Input: 0
output: out of if block

④ if (condition); → সেমিকোলন থাকলে, if block terminate হয়ে যায়, তখন কোনো সমস্যা হয় না, পর পরে block থাকলে ও, block এর ভিতরে যা আছে সব print হবে।

এটার কোনো error দিবে না।

```
void main()
{ int a;
 printf("Enter a");
 scanf("%d", &a);
 if(a);
 { printf("inside if block");
  printf("value of a = %d", a);
 }
 printf("out of if block")
}
```

Input: 5
output:
inside if block
value of a = 5
out of if block

Input: 0
output:
inside if block
value of a = 0
out of if block

N.B.: সব print হবে। কারণ condition check হবে না।

⑤ if (age == 25) → equality operator, input ఇచ్చిన condition check అవుతుంది, but if (age = 25) → assignment operator, input ఇచ్చిన దానిని age లోకి convert చేసి, assign అవుతుంది.

①
```
void main ()
{ int age;
  scanf("%d", &age);
  if (age == 25)
  { printf ("your age is :%d", age);
    printf ("you can go coffee with me");
  }
  printf ("It's time to go home");
}
```

Input: 25
output:
your age is 25
you can go coffee with me
It's time to go home.

②
```
void main ()
{ int age;
  scanf("%d", &age);
  if (age = 25)
  { printf ("your age is : %d", age);
    printf ("you can go coffee with me");
  }
  printf ("It's time to go home");
}
```

Input: 20
output: age = 25 (assign అవుతుంది → condition true)
your age is 25
you can go coffee with me
It's time to go home

If-Else Statement

① If-else এর ক্ষেত্রে, if এর condition সত্য হলে if এর block এর ভেতরের সবগুলো print হবে। তারপর else block এর বাইরে কোনো কিছু থাকলে তা print হবে।

→ Condition মিথ্যা হলে else এর ভেতরের সবগুলো print হবে। এবং বাইরে কোনোকিছু থাকলে তা print হবে।

② If (condition); → থাকলে error দেখাবে, কারণ তারপর else আছে, if না থাকলে else থাকতে পারবে না।

```
int main ()
{ int age;
  printf ("Enter age");
  scanf ("%d", age);
  if (age>25 && age<30);
  { printf ("age: %d", age);
    printf ("\n coffee with me");
  }
  else
    { printf ("age: %d", age);
    }
  printf ("out of if-else");
}
```

output:

→ compile error.

③ যদি If (condition) এর পরে Curly braces {} না থাকে, এবং একাধিক printf() থাকলে, [illegible], কারণ else এর পূর্বে if block এর output হওয়ার অনুমোদন কিছু print করা যাবে না।

→ কারণ if-block এর পরে {} না থাকলে প্রথম শুধু একটি line -টার if-এ count করা হয়।

উদাহরণ:

```c
void main (){
 int age;
 scanf("%d", &age);
if (age>25 && age<30)
  printf ("age: %d", age);
  printf ("\n Coffee with me");
else
   printf ("age : %d", age);
   printf (" Go home");

printf ("out of if-block");
```

output: compile error.

যেহেতু if এর পরে {} নেই, এখানে age: 28 শুধু if এর জন্য print হবে but, coffee with me এই line টা else এর পূর্বে তাই এখানে error হবে। কারণ, if এর পর direct else বসাতে। if-else এর ক্ষেত্রে

Switch statement

মূলত এখানে হবে। integral

→ integer or character হবে।

scanf("%d", &a);

```
switch (expression)
{
    case value-1 :
        block statement ;
        break;        → break দিয়ে out of switch body

    case value-2 :
        st-2;
        break;

    default:
        statement;
}
```

expression:
- (a+2)
- (a+b+c)
- (1+2+3/4) → float আসলেও যদি integer answer হবে।
- (a+b) → a = int, b = float → output = error (float not allow)
- (2+2) → correct.

after switch block, printf("out of switch block");

① break যদি না দেই তাহলে desire output পাওয়া যাবে না। কারণ program যে পর্যন্ত break statement না পাবে ততক্ষণ পর্যন্ত switch block থেকে বের হবে না।

② default দিলে, উপরে কোনো [illegible]

③ case mark > 80 → Not allow.

④ case 'a' → allowed. → এর ascii value কে ধরে নিবে।

program:

```
char ch;
printf("Enter ch: ");
scanf("%c", &ch);
switch(ch)
{
  case 'a';

  case 'e':
  case 'i':
  case 'o':
  case 'u':
           printf("vowel");
           break;

  default:  printf("Not vowel");
}
```

উপরের যেখানে value মিলেছে সেখানে break দেই নাই সেহেতু case 'u' পর্যন্ত print করবে।

output:

ch = e

case e,
case i,
case o.
case 'u':
→ print হবে = vowel

Loop "Jenny"

⊞ While-Loop:

Example-01

```
void main()
{ int i=1;
  while (i==1)  → 1==1
  { printf("%d", i);
    ++i;
  }
  printf("End of program");
}
```

Equality operator → (i==1)

output:

i == 1
1 == 1
print: 1;
++i; 2 → 2==1, exit Loop.
print: End of program.

while (i=1) → Assignment operator

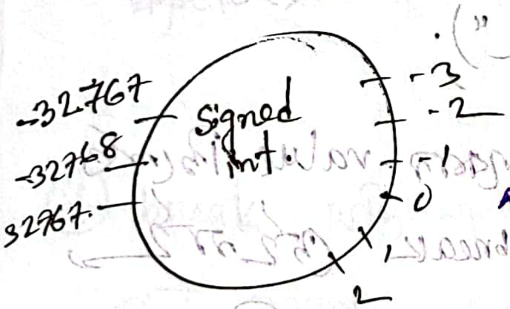
output: i = 1
print: 1
++i; 2, print: 2, 3, 4, 5, ...

⊙ Infinite Loop.

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

output: condition সবসময় (1), 0 নয় তাই সেক্ষেত্রে যত answer হবে.
∴ এটা Infinite Loop. ~~হবে না থামবে~~ কিন্তু
এটা একটা নির্দিষ্ট Range এর ভিতর value print করবে। -32768 to 32767 (int এর জন্য)

Example-02



Ex-02

```
void main ()
{ int i=10;
 while (i)
{ printf ("%d\n", i);
   ++i
}
```

output

```
10
11
12
.
.
32767
-32768
.
.
-3
-2
-1
0
```

Infinite Loop এর ক্ষেত্রে Range এর বাইরে গেলে print হয়ে যায়।

0 এর ক্ষেত্রে loop stop হয়ে যায়।

Ex-03

```
{ int i=10;
 while (i)
{ printf ("%d", i)
}
```

modifier (++i) না করলে;

output: 10101010 - - - -

Infinite Loop.

Ex-04

```
void main()
{ int i;
 while (i<=10)
{ printf ("%d", i);
   i++;
 printf ("End of program");
}
```

Initialization না দিলে Loop কখনো চলবে না।

output: End of program.

Ex-02

```
void main ()
{ int i=10;
  while (i)
{ printf ("%d\n", i);
    ++i
}
```

output

10
11
12
.
.
.
32767
-32768
.
.
-3
-2
-1
0 → 0 এর ক্ষেত্রে loop stop হবে

Infinite Loop হবে Range এর বেশি print হবে

Ex-03

```
{ int i=10;
  while (i)
{ printf ("%d", i)
}
```

modifier (++i) না করলে;

output: 10101010 - - - .

Infinite Loop.

Ex-04

```
void main()
{ int i;
  while (i<=10)
{ printf ("%d", i);
    i++;
}
  printf ("End of program");
}
```

Initialization না দিলে Loop কয়বার ঘুরবে না।

output: End of program.

Ex – 05:

```
void main ()
{int i=1;
while ( )
{
  i++
}
```

→ সংকলন হবেনা এটার compile error দিবে এটার কোন valid sentence না।

Ex – 06

```
void main ()
{ int i=0;
while (i++)
{ printf ("%d\n", i)
}
printf ("End of program");
}
```

output: i=0,

i++ means, =0 (post increment).

while (0) → loop-এ ঢুকবে না।

print করবে: End of program –

i=0

→ while (++i) → pre increment হলে

++i = 1 loop-এ ঢুকবে print হবে

i=0 হলে তখন stop হবে।

Q. while (condition); → কি হবে?

Ex-07

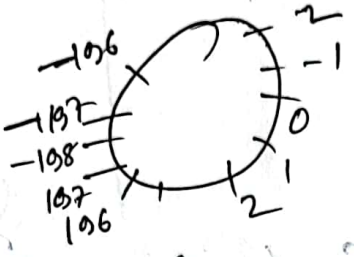
```
void main()
{ char ch = 'a';
  while(ch)
  { printf("%d", ch);
    ch++;
  }
}
```

output:- a = 97.

print: 97, 98, 99. :- _____

→ infinite loop होगा क्योंकि char का range (-128 to 127)

का बाहर print होगा।



For loop in C :- properties of for Loop.

```
for (expression 1 ; expression 2 ; expression 3)   // expression 1 = initialization
{
    // Body of Loop
}
```

N.B: for (; ;) → দুটি semicolon এখানে mandatory বাকিগুলো optional.

① initialization for (; ;) এর ভিতরে দিলেও হবে অথবা, উপরে initialize করলেও হবে। কিন্তু যদি initialize না করি কোথাও error দেখাবে।

```
void main ()
{ int i;           // initialize
for (i=1; i<=5; i++)
  {
   - -
  }
}
```
✓

```
void main ()
{ int i=1;         // initialize
for ( ; i<=5; i++)
  {
   - - .
  }
}
```
✓

```
void main.
{ int i;
for ( ; i<=5; i++){
   - - .
  }
}
```
error, no initialization.

② একাধিক initialization হতে পারে। → what happen

```
void main (){
 int i;
for (i=1, j=0; i<=5; i++)
 { printf("%d %d\n", i,j);
 }
}
```

output	
1	0
2	0
3	0
4	0
5	0

যদি, int i;
for (j=0; i<=5; i++)

③ expression 2 (condition) ফাঁকা রাখলে, condition সব সময় সত্য (true) থাকে infinite Loop হয়ে যায়।

```
void main()
{
  int i,j;
for (i=1, j=0;  ; i++) {
  {printf("%d %d\n", i,,j);
}
}
```

output: Infinite Loop, no condition

② যদি condition সবসময় সত্য থাকে তাহলে Last Condition কে termination condition হিসেবে use করা।

①

```
void main(){
   int i,j;
for (i=1,j=0; i<=5;j<3; i++)
  {printf ("%d %d\n",i,j);
}
}
```

termination condition.

output

1 0
2 0
3 0
4 0
5 0
6 0 → termination condition

simple statement

infinite Loop, j always =0

②

```
void main()
{int i,j;
for(i=1, j=0; i<=100; j<3; i++,j++)
  {printf("%d %d\n", i,j);
}}
```

output:

1 0
2 1
3 2

j<3

③ termination condition, 0 & 1 : যেকোন কিছু হতে পারে,

0 = false, 1 = true.

④ for (i=1, j=0; i<=10; i++, j++)

{

}

Guard condition;

(i) for (i=1, j=0; i<=100 && j<3; i++, j++)

for (i=1, j=0; i<=100 || j<3; i++, j++)

one condition, logical operator.

⑤ increment, decrement or কিছুই না

```
void main ()
{ int i,j;
for (i=1, j=0 ; i<=100 j<3; )
{printf( "%d %d\n", i, j);
}
```

output

j always <0,

infinite loop হবে

→ i++; i increase হবে

infinit loop হবে

→ j++ কিছুই না,

1 0 } output

1 1

1 2

Ex-03

```
void main()
{ int i;
  int a=5, b=6;
for( i= a+b; i<=100; i++)
```

(a+b)/2

i = i/2

i+2

i+a

⇒ যেকোনো কিছু হতে পারে।

Ex-04: infinite loop → condition না দিলে infinit loop হবে

```
for ( ; ; )
  { printf ("My name is Bashar");
  }
```

* 10 → for (i=10; i>0; i--)
9
8
,
1 { printf ("%d\n", i); }

Ex-05:

```c
void main (){
int i,j;
for (i=1, j=0; i<5; j<=6; i++,j++);
    { printf ("%d%d\n", i,j);
    }
}
```

সেমিকোলন দিলে যে পর্যন্ত condition false না হবে Loop এর block এর মধ্যে না ঢোকায় print-ও হবে না। সবশেষে কে general এ statement হিসেবে ধরবে।

Output:

i	j
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	7

print: i j = 8 7

Assignment

(1)

```c
void main (){
int i,j;
for (i=1, j=0; i<5; j<=6; i++);
 { printf ("%d%d\n", i,j);
   j++;
 }
}
```

output:
infinite loop

(2)

```c
void main(){
int i,j;
for (i=1, j=0; i<5, j<=6; j++);
 { printf ("%d%d\n", i,j);
   i++;
 }
}
```

i	j
1	0
1	1
1	2
1	3
1	4
1	5
1	6
1	7

Task-2:

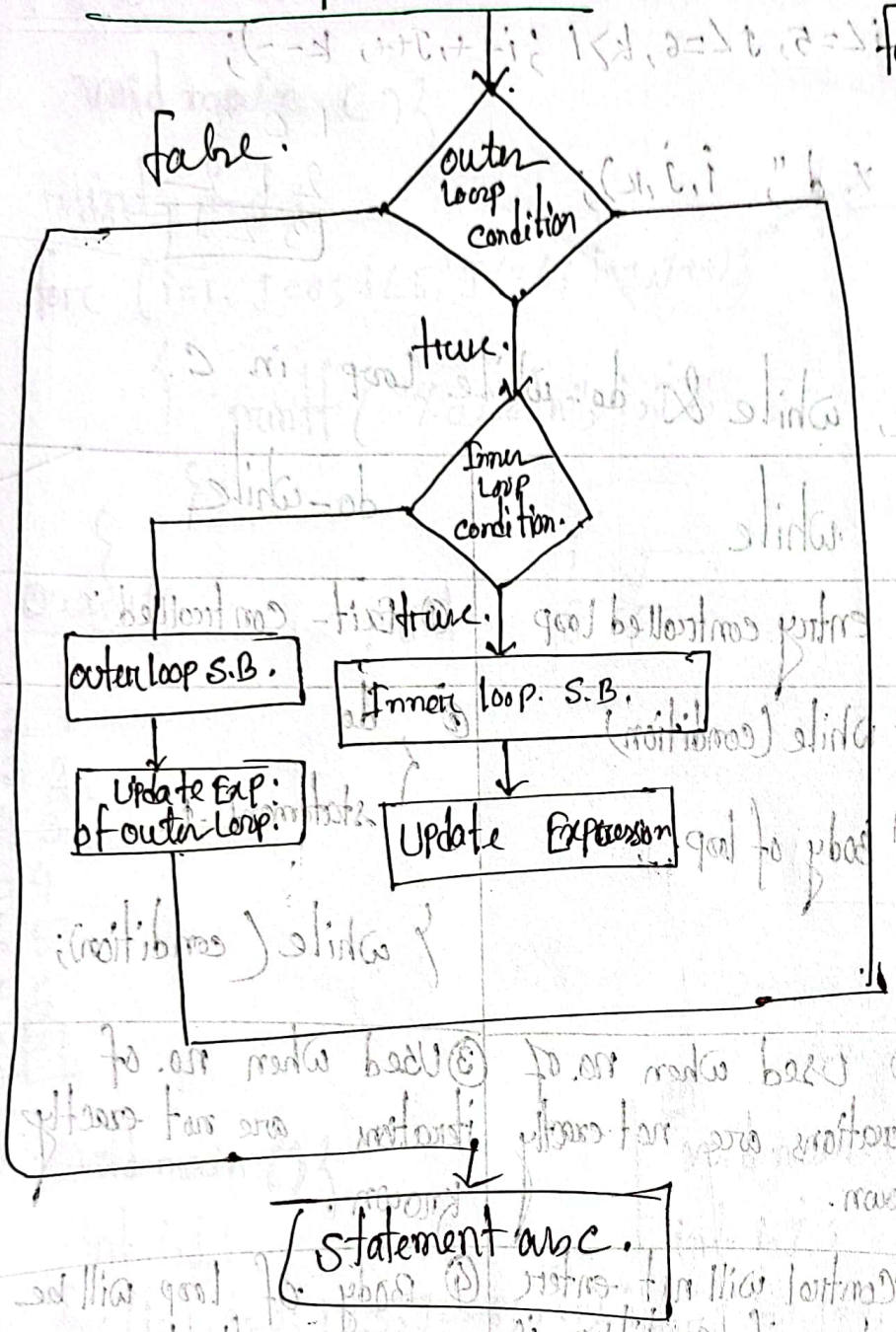
```
for(i=1, j=0, k=3; i<=5, j<=6, k>1 ; i++, j++, k--);
{ printf("%d %d %d", i,j,k);
}
```

1 0 3
2 1 2
3 2 1 → print

Q. Difference between for, while & do-while loop in C.

for	while	do-while
① Entry controlled loop	① entry controlled loop	① Exit-controlled
② for(initialization; condition; update) { Body of loop }	② while (condition) { Body of loop }	② do { statement-1; } while (condition);
③ Used when no. of iterations are known.	③ Used when no. of iterations are not exactly known.	③ Used when no. of iterations are not exactly known.
④ Control will not enter into loop if condition if false.	④ Control will not enter into loop if condition is false.	④ Body of loop will be executed atleast once.

Nested Loops in C




```
for (I; c; ↑↓)
{
  for (I, c, ↑↓↓)
  {
    Inner loop statement.
    block;
  }
  outer Loop S.B
}
statement abc.
```

Program:

```
void main ( ){
  int i, j;
for (J=1; J<=6; J++)
{
   for (i=1; i<=5; i++)
    { printf ("*");
    }
 printf ("\n");
}
```

← Execute new line

J=1, i=1,2,3,4,5
J=2, i=1,2,3,4,5
J=6; i=1,2,3,4,5

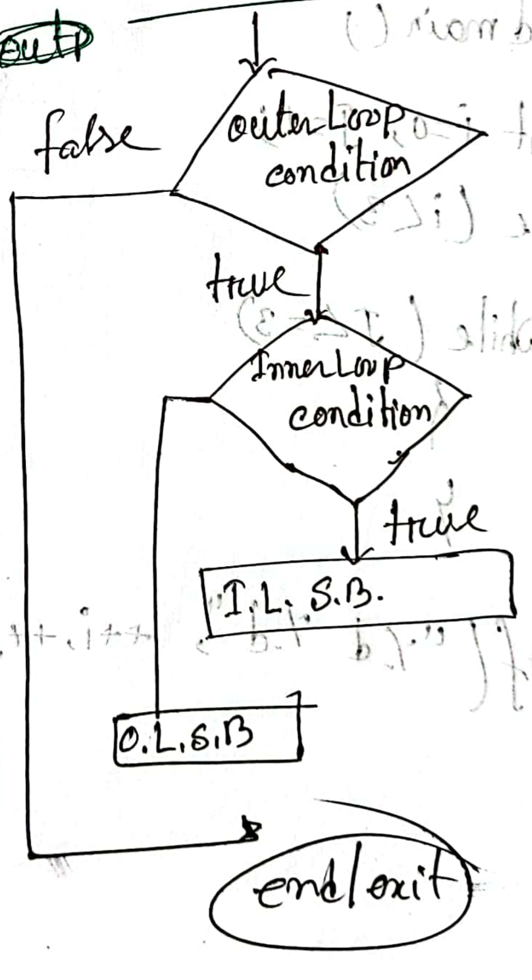
output:

```
                i
          ------------->
J=1,  *  *  *  *  *   → i=1,2,3,4,5
J=2,  *  *  *  *  *   → i=1,2,3,4,5
J=3,  *  *  *  *  *   → i=1,2,3,4,5
J=4,  *  *  *  *  *
J=5,  *  *  *  *  *
J=6,  *  *  *  *  *
```

Nested while loop in C

```
while (condition
 { while (condition)
    { Innerloop statement Block;
    }
  Outer Loop st. Block;
}
```

Flowdiagram



Program:

```
int i=0, j=1;
while (i<3)
{
  while (j<=3)
  {
    printf ("%d", j++)
  }
  printf ("%d", i++);
}
```

output

i=0, condition true, enter loop

j=1, condition true.

print: 1 2 3 0 1 2

(j++) j j

Assignment:

```
void main ()
{ int i=0, j=1;
while (i<3)
{ while (j<=3)
  {
  }
printf ("%d %d", ++i, ++j);
}
```

output

blank

Nested do-while Loop

```
int i=1, j=3
do
{
   do
   {
      print ("%d", j);
      j--;
   } while (j>0);
  i++;
 Printf("%d ", i);
while (i<4);
```

output

i j

x 3 2 1 0 -1 -2

2 3

4

output: 3 2 1 2 0 3 -1 4

Assignment

```
int i=1, j=3;
do
{
   do
   { printf("%d", --j);
   } while (j>0);
 printf ("%d", i++);
} while (i<4);
```

output

Function in C

23.10.22

Function in C programming is a reusable block of code that performs a specific task and makes a program easier to understand, test and can be easily modified without changing the calling program.

↳ Functions divide the code and modularize the program for better and effective results. (divide program)

↳ In short, a larger program is divided into various subprograms which are called as functions.

↳ Modular programming reduces the complexity in writing program.

Types of functions:

There are two types of functions in 'C' programming

i. Library functions: These are the in-built functions of 'C' library. These are already defined in 'C' header files.

Example: scanf(), printf(), gets(), puts(), ceil(), floor(), strcpy(), strcmp(), strlen(), strcat(), strlwr() etc.

(ii) User-defined functions: are the functions which are created by the 'C' programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code. It's a big advantage of 'C' programming.

Some important key point about Function

- A function receives zero (or) more parameters, performs a specific task, and returns zero or one value.
- A function is invoked by its name and parameters.
- No two functions have the same name in a single C program.
- The communication between the function and invoker is through the parameters and the return value.
- A function is independent.
- It is completely self-contained.
- It can be called at any place of your code and can be ported to another program.
- Functions make programs reusable and readable.

Note1 Function calls execute with the help of execution stack.

Execution of 'C' program starts with main function.

Main is the user-defined function.

Note-2 In C programming program start from main function. If put main() function under a main() function it will be a infinite loop.

Advantage of functions in 'C' Program

i. We can avoid rewriting same logic/code again and again in a program.

ii. We can call C functions any number of times in a program and from any place in a program.

iii. We can track a large C program easily when it is divided into multiple functions.

iv. Reusability is the main achievement of 'C' functions.

v. However, Function calling is always a overhead in a C program.

Function Aspects:

C - Programming functions are divided into three activities such as:-

i. Function declaration
ii. Function definition.
iii. Function call.

i. Function declaration:

↳ A function must be declared globally in a C program to tell the compiler about the function name, function parameters, and return type.

↳ We cannot use a function without declare the function.

↳ A function declaration is also called 'Function prototype'

Declaration

return_data_type function_name (data_type arguments);

- return_data_type: is the data type of the value function returned back to the calling statement.
- function_name: is followed by parentheses.
- Arguments: names with their data type declarations optionally are placed inside the parentheses.

Program declare a function:

```
# include <stdio.h>
/* Function declaration */
int add (int a, b);
/* End of Function declaration */
int main (){}
```

↳ Keep in mind that a function does not necessarily return a value. In this case, the keyword void is used.

ii. Function Definition:

↳ Function definition means just writing the body of a function.

↳ A body of a function consists of statements which are going to perform a specific task.

↳ A function body consists of a single or a block of statements. It is also a mandatory part of a function.

Definition

```
return_data_type function_name (parameter list){
    // body of the function;
}
```

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function

- Return_data_type:- A function may return a value. The return_type is the data type of the value the function returns.
- Function Name:- This is the actual name of the function.
- Parameters: The parameter list refers to the type, order and number of the parameters of a function. parameters are optional;
- Function body: contains a collection of statements that define what the function does.

iii. Function calling: Function can be called from anywhere in the program. when it is required in a program.

↳ Whenever we call a function, it performs an operation for which it was designed.

↳ A function call is an optional part of a program.

Calling: function_name (argument_list).

Example-of:

```
#include <stdio.h>
int add (int a, int b); // function declaration.
int main ()
{
    int a=10, b=20;
    int c= add (10,20); // function call
    printf ("Addition: %d\n", c);
}
int add (int a, int b) // function body
{
    int c;
    c= a+b;
    return c;
}
```

output:

Addition: 30.

Different aspects / Category of function calling :-

1. function without arguments and without return value.
2. Without arguments and with return value.
3. With arguments and without return value.
4. With arguments and with return value.

1. function without argument and without return value.

Example:

```
#include <stdio.h>
void printName ();
void main ()
{
  printf ("Hello");
  printName ();
}
void printName ()
{
  printf ("javatpoint");
}
```

output:

Hello
javatpoint.

(ii) Without arguments and with return value.

Example:

```
#include <stdio.h>
float area(); // prototype / declaration.
int main(){
    printf("Area of square = %f", area());
    return 0;
}
float area(){
    float ar;
    float side;
    printf("Enter the side: ");
    scanf("%f", &side);
    ar = side * side;
    return ar;
}
```

output:

```
Enter the side: 10
Area of circle = 100.000000
```

(iii) With arguments and without return value.

Example:

```
#include <stdio.h>
# include <conio.h>
void area (float length; float breadth);
int main (){
float breadth. length;
   printf ("Enter the length and breadth: ");
   scanf ("%f %f", & length, & breadth);
   area (length, breadth);
   return 0;
}
void area (float length, float breadth) {
float arc;
arc = length * breadth;
printf ("Area of Rectangle = %f", ar);
}
```

output:

outputs:

Enter the length and breadth: 10 12

Area of Rectangle = 120.000000.

iv. With arguments and Return value.

```
#include <stdio.h>
int sum(int, int); // prototype
int main() {
    int x, y, result;
    printf("Enter value of x and y: ");
    scanf("%d %d", &x, &y);
    result = sum(x,y); // calling of function.
    printf("sum: %d", result);
    return 0;
}
int sum(int a, int b) { // function definition.
    int res;
    res = a+b;
    return res;
}
```

output:

Enter value of x and y: 10 12

Sum = 22

Variable scope in C

↳ variable scope means the visibility of variables within a code of the program.

There are three places where variables can be declared in C Programming language.

i. Local variable → Inside a function or a block which is
ii. Global variable → outside of all functions.
iii. Formal parameter → In the definition of function parameters.

Initializing Local and Global variable

↳ When a local variable is defined, it is not initialized by the system, you must initialize it yourself.

↳ Global variables are defined automatically by the system

Data Type	Initial default value
int	0
char	'\0'
float	0
double	0
pointer	NULL

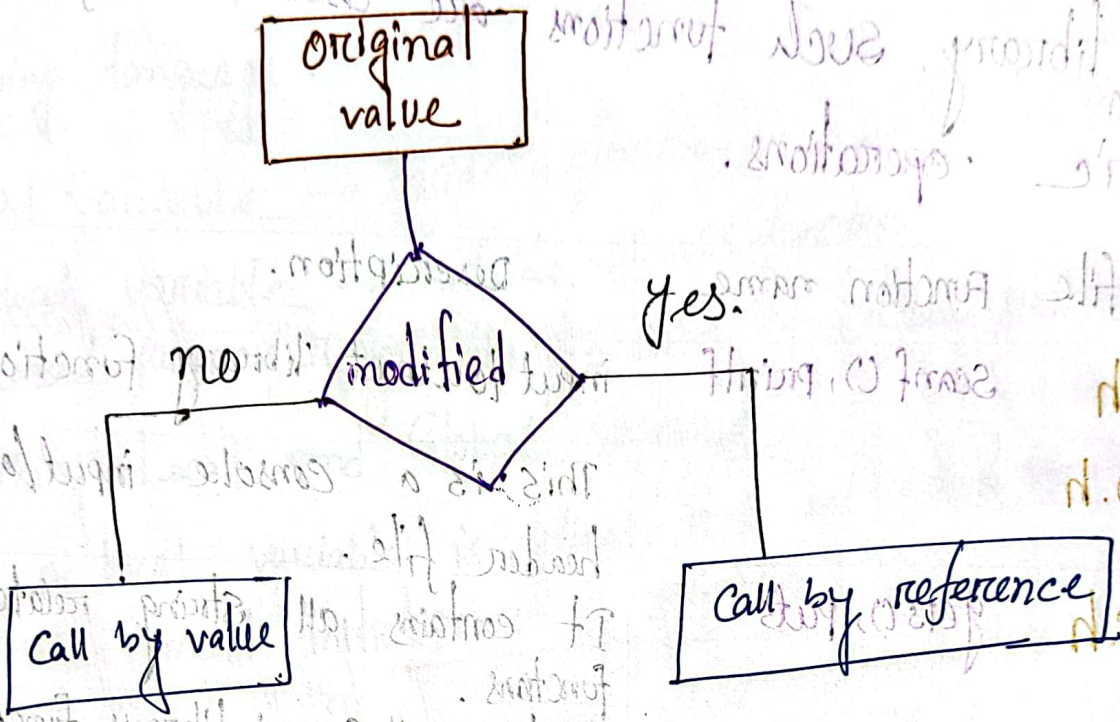
C Library functions:

↳ Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations.

Header file	Function name	Description.
stdio.h	scanf(), printf	input/output library functions.
conio.h		This is a console input/output header file.
string.h	gets(), puts	It contains all string related library functions.
stdlib.h	malloc(), calloc(), exit(), abs()	contains all general library functions.
math.h	pow(), sqrt	contains all math operations related functions.
time.h		time related functions.
ctype.h	isalnum(), isdigit(), issupper(), islower()	contains all character handling functions.
stdarg.h	variable arg	variable argument functions
signal.h		All the signal handling functions.
setjmp.h		all the jump functions.
locale.h		contains locate functions.
errno.h		contains error handling functions.
assert.h		contains diagnostic functions.

Call by value and Call by Reference in C

↳ There are two methods to pass the data into the function in C Language, i.e., call by value and call by reference.



1. Call by value in C:

↳ In call by value method, the value of the actual parameters is copied into the formal parameters.

↳ In call by value method, we can not modified the value of the actual parameter by the formal parameter.

↳ In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.

↳ The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Note: category of function calling 1, 2, 3, 4 are example of call by value.

Example:

```
#include <stdio.h>
void change (int num){
printf("Before adding value inside function num = %d\n", num);
num = num + 100;
printf(" After adding value inside function num = %d\n", num);
}
int main(){
int x = 100;
printf("Before function call x = %d\n", x);
change (x); // Passing value in function
Printf(" After function call x = %d\n", x);
return 0;
}
```

output:

Before function call x = 100
Before adding value inside function num = 100
After adding value inside function num = 200
After function call x = 100.

2. Call by reference in C:

↳ The address of the variable is passed into the function call as the actual parameter.

↳ The value of the actual parameters can be modified by changing the formal parameters, since the address of the actual parameter is passed.

↳ In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Example:

```
#include<stdio.h>
void change(int *num){
printf(" before adding value = %d\n", *num);
(*num) += 100;
printf("After adding value = %d\n", *num);
}
int main(){
  int x=100;
  printf("Before function call x = %d\n", x);
  change(&x); // passing reference in function
  printf("After function call x = %d\n", x);
return 0;
}
```

output

```
before function call x=100
before add value = 100
After add value = 200
After function call, x=200
```

Difference between call by value and reference in C.

Call by value	Call by reference.
1. A copy of the value is passed into the function.	1. An address of value is passed into the function.
2. The value of the actual parameters do not change by changing the formal parameters.	2. The values of the actual parameters do change by changing the formal parameters.
3. Actual and formal arguments are at the different memory location.	3. Actual and formal arguments are created at the same memory location.
4. void call_by_value (int a, int b)	4. void call_by_reference (int *a, int *b)

Recursion in C

↳ Any function which calls itself is called recursive function, and such function calls are recursive calls.

↳ But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Advantages of Recursion

↳ Recursion code is shorter than iterative code however it is difficult to understand.

↳ Recursion is more useful for the tasks that can be defined in terms of similar subtasks.

↳ Recursion may be applied to sorting, searching, and traversal problems.

↳ Some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial findings. etc.

Example of factorial using Recursion:

```
#include <stdio.h>
int fact (int);
int main ()
{ int n, fg;
  printf(" Enter the number whose factorial you want to calculate?");
  scanf ( "%d", &n);
  f = fact (n);
  printf ("Factorial of %d is %d\n", n, f);
}
int fact (int n) {
  if (n==0)
  { return 0;
  }
  else if (n==1)
  { return 1;
  }
  else
  { return n*fact(n-1);
  }
}
```

output:

Enter number: 5

factorial of 5 is 120

Note: Recursive function must have an if condition to force the function to return without recursive call being executed. If there is no such condition, then the function execution falls into infinite loops.

* Recursive Function

↳ A recursive function performs the tasks by dividing it into the subtasks.

↳ There is a termination condition defined in the function which is satisfied by some specific subtasks.

↳ After this, the recursion stops and the final result is returned from the function.

* Pseudocode for writing any recursive function:

```
if (test_for_base)
{
  return some_value;
}
else if (test_for_another_base)
{
  return some_another_value;
}
else
{
    // statements;
    recursive call;
}
```

Example: Fibonacci Series.

```
#include <stdio.h>
int fibonacci(int);
void main()
{ int n, f;
  printf("Enter the value of n:\n");
  scanf("%d", &n);
  f = fibonacci(n);
  printf("%d", f);
}
int fibonacci(int n)
{
  if (n==0)
  { return 0;
  }
  else if (n==1)
  { return n 1;
  }
  else
  { return fibonacci(n-1)+fibonacci(n-2);
  }
}
```

output:

① Enter the value of n: 12
= 144

② Enter value 6
= 8

Memory allocation of Recursive method

↳ Each recursive call creates a new copy of that method in the memory.

↳ Once some data is returned by the method, the copy is removed from the memory.

* Storage classes in C

↳ Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable.

There are four types of storage classes in C.

RAM storage place

1. Automatic → Default value = Garbage value, scope → Local.
2. Extern
3. static
4. Register.

Register.

2. Extern
   - Default value = zero
   - scope = Global

   Lifetime: Till the end of the main program.

3. static
   - zero
   - Local

4. Auto and register
   - Garbage value
   - local.
   - Within the function.

Difference between Recursion and Iteration.

Recursion	Iteration.
① Recursion is the process of calling a function itself within its own code.	① In Iteration, there is a repeated execution of the set of instructions.
② There is a termination condition is specified.	② The format of iteration includes initialization, condition, and increment/decrement of a variable.
③ The termination condition is defined within the recursive function.	③ The, termination condition is defined in the definition of the loop.
④ Smaller code size.	④ Larger code size.
⑤ Very high time complexity.	⑤ Relatively lower time complexity.
⑥ It is always applied to functions.	⑥ It is applied to loops.
⑦ It has to update and maintain the stack.	⑦ There is no utilization of stack.
⑧ It uses more memory as compared to iteration.	⑧ It uses less memory.
⑨ It is slower than iteration	⑨ It is faster than recursion.

'C' Array

Array: An array is defined as the collection of similar type of data items stored at contiguous memory locations.

↳ Arrays are the derived data type which can store primitive type of data such as int, char, double, float, etc.

↳ It also store derived data types; such as pointers, structures, etc.

Properties of array:

i. Each element of an array is of same data type and carries the same size. i.e. int = 4 bytes.

ii. The first element in the array is numbered 0, and the last element is one less than the total size of the array.

iii. Elements of the array can be randomly accessed.

iv. An array is also known as a subscripted variable.

v. Before using an array, its type and dimension must be declared.

vi. Individual elements accessed by index indicating relative position in collection.

vii. Index of an array must be an integer.

Advantages of array :-

i. code optimization:
ii. Ease of traversing
iii. Ease of sorting
iv. Random Access.

Disadvantage of C Array

i. Fixed Size.

Types of Array

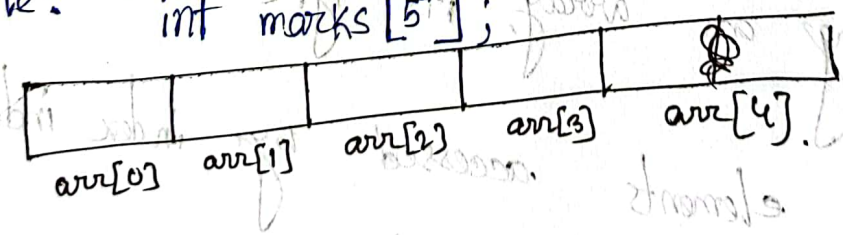
There are 2 types of array in C.

i. One Dimensional array
ii. Multi dimensional array
   - Two-dimensional array
   - Three
   - Four

i. Declaration of one-Dimensional array

data_type array_name [array_size];

For example: int marks [5];



Initialization of an array :-

data_type arr_name [arr_size] = {value1, value2, value3, ...}

Example:

int age[5] = {0, 1, 2, 3, 4}

Int age [4] = {20, 30, 40, 50}

* In such case, there is no requirement to define the size.

Example: int marks [] = {20, 30, 40, 50, 60}

Examples: one dimensional array

```
#include <stdio.h>
int main (){
  int i;
int Arrr[5] = {10,20,30,40,50};
Arrr[5] = {0};
 Arrr[0] = 10;
 Arrr[1] = 20;
 Arrr[2] = 30;
 Arrr[3] = 40;
 Arrr[4] = 50;
 for(i=0; i<5; i++)
   printf("Value of Arrr[%d] is %d \n", i, Arrr[i]);
}
```

output

value of Arrr[0] is 10
" " [1] is 20
" " [2] " 30
" " [3] " 40
" " [4] " 50.

Example-2 :-

output

```
#include<stdio.h>
int main()
{
int x[10] = {3,6,2,4,8,9},i;
for (i=0; i<10; i++)
printf("%d", x[i]);
return 0;
}
```

3 6 2 4 8 9 0 0 0 0

Note:

↳ Till the array elements are not given any specific value, they are supposed to contain garbage value.

↳ If the number of elements used for initialization is lesser than the size of array then the remaining elements are initialized with zero.

↳ Where the array is initialized with all the elements, mentioning the dimension is optional.

Two-Dimensional Array in C :-

↳ The two dimensional array can be defined as an array of arrays.

↳ The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure.

* Declaration of two dimensional Array in C.

```
data_type array_name [rows] [columns];
```

Example: `int twodimension [4] [3];`

* Initialization of 2D array in C.

```
int arr[4][3] = {{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
```

Example-01

```
#include <stdio.h>
int main (){
 int i=0, j=0;
 int arr[4][3] = {{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
  // traversing 2D array
 for (i=0; i<4; i++){
 for (j=0; j<3; j++)
  printf( "arr[%d][%d] = %d\n", i, j, arr[i][j]);
 } end of j
} end of i
 return 0; }
```

output

arr[0][0] = 1
" [0][1] = 2
[0][2] = 3
[1][0] = 2
[1][1] = 3
[1][2] = 4
[2][0] = 3
[2][1] = 4
[2][2] = 5
[3][0] = 4
[3][1] = 5
[3][2] = 6

Return an Array in C

Example: passing array to a function.

```
#include <stdio.h>
void getarray (int arr[])
{
printf ("Elements of array are: ");
for (int i=0; i<5; i++)
{
printf ("%d", arr[i]);
}
}
int main ()
{
int arr[5] = {45, 67, 34, 78, 90};
getarray (arr);
return 0;
}
```

output:

```
Element of array are:
45 67 34 78 90.
```

Example-02 passing array to a function as a pointer.

```
#include <stdio.h>
void printarray (char *arr)
{
printf ("Elements of array are: ");
for (int i=0; i<5; i++)
{
printf ("%c", arr[i]);
}
```

```
int main()
{
Char arr[5] = {'A','B','C','D','E'};
printarray (arr);
return 0;
}
```

Output: Elements of array are : A B C D E

* There are three right ways of returning an array to a function:
  1. Using dynamically allocated array
  2. Using static array
  3. Using structure.

Passing Array to Function in C

* consider the following syntax to pass an array to the function:

```
functionname (arrayname); // passing array
```

* Methods to declare a function that receives an array as an argument.

There are 3 ways to declare the function which is intended to receive an array as an argument.

First way

```
return_type function (type arrayname[])
```

→ Declaring blank subscript notation [] is the widely used technique.

Second way

return_type function(type arrayname[SIZE])

optionally, we can define size in subscript notation [].

Third way

return_type function(type * arrayname)

you can also use the concept of a pointer.

String in 'C' programming

String: The string can be defined as the one-dimensional array of characters terminated by a null ('\0').

↳ The character array or the string is used to manipulate text such as word or Sentences.

↳ Each character in the array occupies one byte of memory, and the last character must always be 0.

* The difference between a character array and a string is the string is terminated with a special character '\0'.

There are two ways to declare a string in c language.

1. By char array
2. By string literal.

1. By char array:

```
char str_name[size];
```

Example:

```
char ch[10] = {'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
```

While declaring string, size is not mandatory.

So, we can write the above code as given below.

```
char ch[] = {'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
```

11. String literal

char ch[] = "javatpoint";

or you can declare string like a pointer.

char *p = "Cloud IT BOOK"

'\0' character in C

The null character '\0' (also null terminator), is a control character with the value zero. The character has much more significance in C and it serves as a reserved character used to signify the end of a string, often called a null-terminated string.

Read String from User:

* scanf():

↳ You can use the scanf() function to read a string. The scanf() function reads the sequence of characters until it encounters a whitespace (space, newline, tab etc).

Example

```
int main()
{
char name[30];
printf("Enter name: ");
scanf("%s", name);
printf("Your name is %s.", name);
}
```

output

Input: Hello world

output: Hello.

* C gets() and puts() functions

↳ The gets() function is an input function of string. The gets() allows the users to enter the space-separated strings.

The gets() function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line is encountered. It suffers from buffer overflow, which can be avoided by using fgets(). The fgets() makes sure that not more than the maximum limit of characters are read.

↳ C puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function.

Example

```
int main()
{ char name[30];
  printf("Enter name");
  gets(name);   // read string
  printf("Name : ");
  puts(name);   // display string
}
```

output

Input: Hello world

output: Hello world.

* Passing string to a function :

```
#include <stdio.h>
void displaystring(char str[]);
int main(){
char str[50];
printf("Enter string: ");
gets(str);
displaystring(str);
return 0;
}
void displaystring(char str[]){
printf("string output: ");
puts(str);
}
```

String Functions :-

String functions	Description
strcat()	দুইটি string কে একত্রিত করার জন্য ব্যবহার করা হয়।
strcpy()	একটি string আর একটি string-এ কপি করার জন্য ব্যবহার করা হয়।
strlen()	String এর length বের করার জন্য ব্যবহার করা হয়।
strcmp()	দুইটি string কে তুলনা করার জন্য ব্যবহৃত করা হয়।
getchar()	getchar() function reads character from keyboard.
putchar()	writes a character to ~~string~~ screen.
gets()	gets() function reads string from user.
puts()	puts() function prints the string.
strstr()	returns pointer to last occurence of str2 in str1.
strlwr()	string কে ছোট হাতের বর্ণে রূপান্তর করে।
strupr()	String কে বড় হাতের বর্ণে রূপান্তর করে।
strrev()	String কে বিপরীত করে সাজায়।
strstr()	

strlen() function:-

↳ The strlen() function returns the length of the given string. It doesn't count null character '\0'

Program

```
#include <stdio.h>
#include <string.h>
int main(){
char ch[20] = {'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
printf("Length of string is: %d", strlen(ch));
return 0;
}
```

output Length of string is :10.

strcpy() function:

The strcpy(destination, source) function copies the source string in destination.

Program:

```
#include <stdio.h>
#include <string.h>
int main(){
char ch[20] = {'J', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
    char ch2[20];
    strcpy(ch2, ch);
    printf("value of second string is: %s", ch2);
    return 0;
}
```

output: value of second string: Javatpoint.

strcat() function

↳ The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

Program:

```
#include <stdio.h>
#include <string.h>
int main (){
  char ch[10] = {'h', 'e', 'l', 'l', 'o', '\0'};
  char ch2[10] = {'c', '\0'};
  strcat (ch, ch2)
  printf ("value of first string is: %s", ch);
  return 0;
}
```

output: value of first string is : helloc.

@ strcmp() function:

↳ The strcmp (first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using gets() function which reads string from the console.

program:

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

```
char str1[20], str2[20];
printf("Enter 1st string: ");
gets(str1);
gets(str2);
if(strcmp(str1, str2) == 0)
  printf("strings are equal");
else
  printf("strings are not equal");
return 0;
}
```

output:

1st string: hello
2nd string: hello
strings are equal

strrev()

↳ The strrev(string) function returns reverse of the given string.

program:

```
#include<stdio.h>
#include<string.h>
int main(){
  char str[20];
  printf("Enter string: ");
  gets(str);
  printf("string is: %s", str);
  printf("\n Reverse string is: %s", strrev(str));
  return 0; }
```

output:

```
Enter string: javatpoint
String is: javatpoint
Reverse string is: tnioptavaj.
```

strstr() function

↳ The strstr() function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

Syntax

```
char * strstr (const char *string, const char *match)
```

Program:

```
int main (){
char str[100] = "this is javatpoint with c and java";
char *sub;
sub = strstr (str, "java");
printf ("\n substring is: %s", sub);
return 0;
}
```

output:

javatpoint with c and java.

Pointer in C

↳ Pointers in C Language are a variable that stores/points the address of another variable.

↳ A pointer in is used to allocate memory dynamically i.e. at run time.

↳ The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

↳ Pointer syntax: data_type *var_name;

Example: int *p; char *p;

↳ A pointer is a variable which contains the address in memory of another variable.

↳ The unary or nomadic operator "&" gives the "address of a variable".

↳ The indirection or dereference operator "*" gives the "contents of an object pointed to by a pointer".

⊞ Key points to remember about pointers in C

↳ Normal variables stores the value whereas pointer variable the address of the variable.

↳ The content of the C pointer always be a whole number i.e. address.

↳ Always C pointer is initialized to null, i.e. int *p = null.

↳ The value of null pointer is '0'.
↳ '&' symbol is used to get the address of the variable.
↳ '*' symbol is used to get the value of the variable that the pointer is pointing to.
↳ If a pointer in C is assigned to null, it means it is pointing to nothing.
↳ Two pointers can be subtracted to know how many elements are available between these two pointers.
↳ But, pointer addition, multiplication, division are not allowed.
↳ The size of any pointer is 2 byte. (16 bit compiler)

* pointer:

↳ pointer হচ্ছে memory address এর reference. একটিকে যেমন variable হচ্ছে value রাখার container তেমনিভাবে pointer হচ্ছে memory তে value টি

* Memory access করার জন্য এবং যেকোনো address নিয়ন্ত্রণের ক্ষমতা করার জন্য C programming এ pointer ব্যবহৃত হয়।

Pointer এর বৈশিষ্ট্য লিখ?

i. Pointer variable declare করার সময় (*) চিহ্নটি pointer variable এর নামের আগে ব্যবহার করা হলেও এটি নামের অন্তর্ভুক্ত নয়।

ii. কোন pointer variable কে যে data type এ declare করা হয় সে সুবিধামত যে type এর data point করতে পারে।

iii. Declare এর পূর্বে মান নির্ধারণ করতে হয়।

iv. সাধারণ variable এর মত pointer variable declare করা হয়।

Example:

```
# include <stdio.h>
int main ()
{ int *ptr, q;
  q=50;
  ptr = &q;
printf("%d", *ptr);
return 0;
}
```

output: 50.

*p Vs **p Vs ***p meaning:

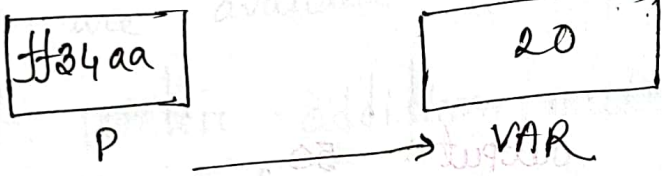
*p:

↳ *p is a pointer to a variable, as shown below.

↳ It is also called single pointer.

↳ The single pointer has two purposes:

- i. to create an array. ~~and~~
- ii. to allow a function to change its contents (pass by references).



**p:

↳ **p is a pointer to a pointer variable, also called double pointer.

↳ It is a form of multiple indirection, or a chain of pointers.

↳ When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

***p:

↳ ***p is a pointer to a double pointer, rather a pointer to a pointer to a pointer variable, as shown below.

↳ It is mostly called triple pointer.

↳ It is an even higher level of multiple indirection or pointer chaining.

↳ A triple pointer is used to traverse an array of pointers.

Pointer to pointer:

↳ We know, pointer is a variable that contains address of another variable.

↳ Now this variable address might be stored in another pointer.

↳ Thus, we now have a pointer that contains address of another pointer, known as pointer to pointer.

NULL Pointers:

↳ It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration.

↳ A pointer that is assigned NULL is called a null pointer.

↳ The NULL pointer is a constant with a value of zero defined in several standard libraries.

Consider the following program:-

```
int main()
{
  int *ptr = NULL;
  printf("The value of ptr is : %x \n", ptr);
  return 0;
}
```

Example-01:-

```
int main()
{
  int val[3] = {5, 10, 20};
  int *ptr;
  ptr = val;
  printf("%d %d %d", ptr[0], ptr[1], ptr[2]);
  ++*ptr;
  printf("%d %d %d", ptr[0], ptr[1], ptr[2]);
  printf("%d", *(++ptr));
  return 0;
}
```

output: 5 10 20 6 10 20 10

Ex-02:

```
int array[] = {6,7,8,9,0,1,2,3,4,5,6};
*p = array + 5;
printf("%d \n", p[1]);
```

output: 2

* 'C' Programming Pointer and array:

↳ 'C' programming - এ array কে pointer দ্বারা খুব সহজে প্রকাশ করা যায়।
↳ কিছু ক্ষেত্রে মধ্যে গুরুত্বপূর্ণ পার্থক্য হলো pointer value পরিবর্তনযোগ্য কিন্তু array এর address সবসময় একই থাকে। অর্থাৎ array এর address fixed থাকে।

Pointer Arithmetic in C:

↳ In pointer from pointer subtraction, the result will be an integer value.

Following arithmetic operations are possible on the pointer in C language:

- ↳ Increment
- ↳ Decrement
- ↳ Addition
- ↳ Subtraction
- ↳ Comparison.

i. Incrementing pointer in C:

↳ If we increment a pointer by 1, the pointer will start pointing to the imediate next location.

↳ We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a Loop.

The Rule to increment the pointer is given below:-

new_address = current_address + i * size_of (data type)

where, i is the number by which the pointer get increased.

32-bit

For 32-bit in variable, it will be incremented by 2 bytes.

64-bit

↳ it will be incremented by 4 bytes.

* Traversing an array by using pointer.

```
#include <stdio.h>
void main()
{ int arr[5] = {1,2,3,4,5};
  int *p = arr;
  int i;
  printf("printing array element :-\n");
  for (i=0; i<5; i++)
  { printf("%d", *(p+i));
  }
}
```

output

printing array element ...

1 2 3 4 5

ii. Decrementing pointers in C

↳ If we decrement a pointer, it will start pointing to the previous location.

The formula for decrementing the pointer is given below:-

new_address = current_address − i * size_of (data type)

iii. C - pointer addition:

↳ We can add a value to the pointer variable. The formula of adding value to pointer is given below.

new_address = current_address + (number * size_of (data type))

32-bit

↳ it will add 2* number.

64-bit

↳ it will add 4* number.

Example

```
#include <stdio.h>
int main () {
int number = 50;
int *p;
p = &number;
printf ("Address of p variable is %u \n", p);
p = p + 3;
printf (" After adding 3 : \n", p);
return 0;
}
```

output

Address of P : 32148643 00

After adding 3 : 32148631 2

(4*3) = 12

iv. Pointer subtraction:

↳ We can subtract a value from the pointer variable.

↳ Subtracting any number from the pointer will give an address.

new_address = current_address − (number * size_of(data type));

32-bit

↳ it will subtract 2* number.

64-bit

↳ it will subtract 4* number.

Example:

```
#include <stdio.h>
int main(){
int number=50;
int *p;
p= &number;
printf("Address of p variable is %u \n", p);
p= p-3;
printf("After subtracting 3: %u\n", p);
return 0;
}
```

Output

Address of p-varuable is : 32486300

After subtract 3: 32486288 (3*4=12 subtracted)

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number.

↳ If two pointers are of the same type,

Address 2 − Address 1 = (subtraction of two addresses) / size of data type which pointer points.

Ex:

```
#include <stdio.h>
void main()
{
 int i=100;
 int *p= &i;
 int *temp;
 temp = p;
  p= p+3;
 printf("pointer subtraction: %d - %d = %d", p, temp, p- temp);
}
```

output

pointer subtraction : 1030585080 − 1030585068 = 3

Illegal arithmetic with pointers.

↳ There are various operations which cannot be performed on pointers:

i. Address + Address = illegal

ii. Address * Address = illegal

iii. Address % Address = illegal

iv. Address / Address = illegal

v. Address & Address = illegal

vi. Address ^ Address = illegal

vii. Address | Address = illegal

viii. ~ Address = illegal

* Pointer to array declare

```
int arr[10];
int *p[10] = &arr;
```

* pointer to a function

```
void show (int);
void (*p)(int) = & display;
```

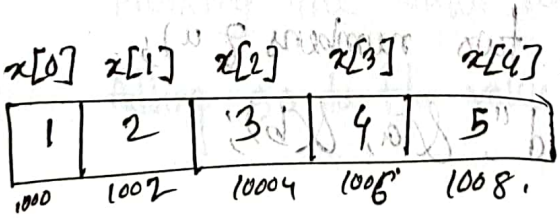
* Pointer to structure

```
struct st {
  int i;
  float f;
} ref;
struct st *p = & ref;
```

What is pointer array? Explain it with figure.

↳ Array কে যখন pointer হিসাবে declare করা হয় তখন তাকে pointer array বলে।

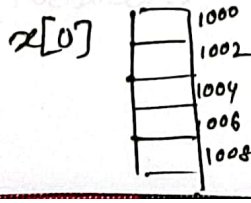
```
int x[5] = {1, 2, 3, 4, 5};
int *p = x;
```



উপরে p কে যদি pointer হিসেবে declare করা হয় তাহলে

```
p = &x[0];
```

p
1000 →



উপরে p, x[0] location point করায় এখন p এর 1000 location টি point করে আছে যার value হচ্ছে 1.

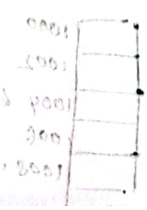
Pointer to function in C

↳ A pointer can point to a function in C. However, the declaration of the pointer variable must be the same as the function. Consider, the following example to make a pointer pointing to the function.

```
#include <stdio.h>
int addition ();
int main ()
{ int result;
   int (*ptr)();
   ptr = &addition;
  result = (*ptr)();
 printf ("The sum is %d", result);
}
int addition ()
{ int a, b;
   printf ("Enter two numbers ? ");
   scanf ("%d %d", &a, &b);
   return a+b;
}
```

Output: Enter two numbers ? 10 15

The Sum is 25



Pointers to Array of functions in C

↳ Basically, an array of the function is an array which contains the addresses of functions.

↳ In other words, the pointers to ~~the functions~~ an array of functions is a pointer pointing to an array which contains the pointers to the functions.

Example:

```
#include <stdio.h>
int show();
int showadd(int);
int (*arr[3])();
int (*(*ptr)[3])();
int main()
{
int result1;
arr[0] = show;
arr[1] = showadd;
ptr = &arr;
result1 = (*(*ptr)[0])();
printf("printing the value returned by show : %d", result1);
(*(*ptr+1))(result1);
}
```

```
int show()
{
int a = 65;
return a++;
}
int showadd(int b)
{
printf("In Adding 90 to the value returned by show: %d", b+90);
}
```

output

```
printing the value returned by show: 65
Adding 90 to the value returned by show: 155
```