C Programming (Nesco Academy)

Why C

⇒ Assembly Portable ✗

C Portable ✓

(2) It takes less line programming

in assembly

```
Move a, 2
Mov b, 3
Add a, b
Mov Result, a
```

In C

```
Result = 2+3
```

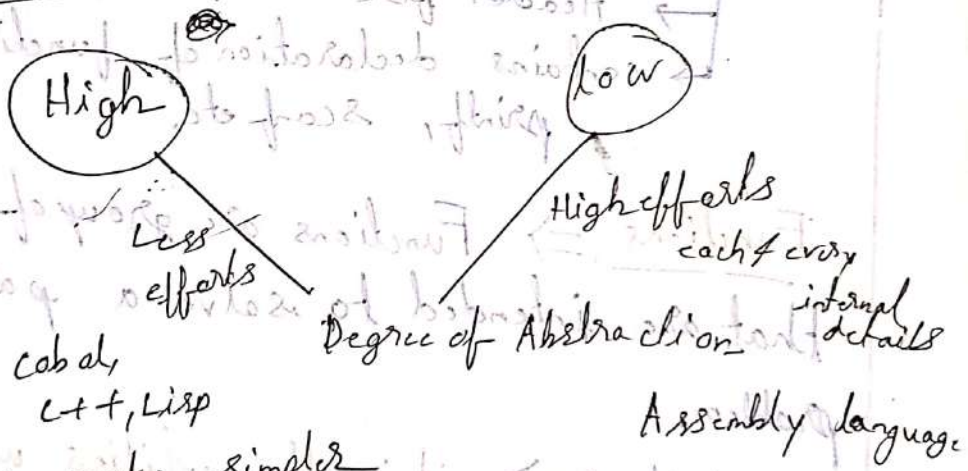
↓ compiler

Assembly code.

Feasures of C

(i) Procedural programming - divides into small

(ii) Middle level language -



it does make simpler

a) Direct access to memory through pointers

b) Bit manipulation

c) Writing assembly code within C code.

(iii) Popular choice for system level apps -

Device drivs, editor, games

(iv) Wide variety of built in function, standard Libraryies & header file.
stdio.h, scanf, printf

(v)

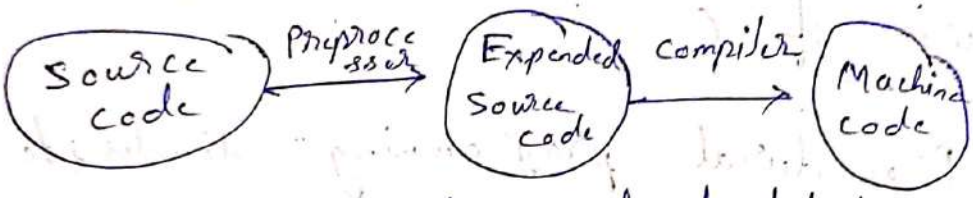
To comment in any programm we use //

#include < stdio.h> → This is called preprocessor directive.

Preprocessor:- replaces text (starting with #) with the actual content.

(*) replaces before the compilation begins

(*) output of preprocessing is expanded source code.



stdio: standard input output file

↳ Header file
↳ contains declaration of function like printf, scanf etc.

Functions :=> Functions are group of statements that are intended to solve a particular problem.

Variable :=> it is the entities used to store value.

(*) Entry point of any programme is a main function which is predefined.

(*) We can write C Program without main function

Syntax of function :⇒

Return_type name_of_function (parameter_type name of parameter, parameter_type name_of_parameter, ...)

{

set of statements;

}

* Parameters are input to that function.

Headerfiles :⇒ Contains prototype of a function or declaration. like- printf
stdio.h

standard library :⇒ Actual definations of functions like- libm.lib

Linker ⇒ it maps between declaration of function & actual defination.

Why so ⇒ Headerfiles consists of only I declaration of function that tells your compiler which function you going to use. Preprocessor combine its & produce expanded source code. Linker maps the declaration with actual defination after completation of compilation.

* if preprocessor does the work of linker it will take huge time to compile.

* if visiting card is header file then the person you met is stl.

Variables ⇒ Glass = Variable
Water = Value

⊛ you have to declare the variable before use.

Declaration → Announcing property of the variable to the compiler.

Properties - a) size of the variable.
b) Name of the variable.

Defination → allocating memory to a variable.

⊛ Most of the time declaration & defination will be done at the same time.

it differentiate one statement from another.

int var ;
- name of the variable
- Datatype - how much space a variable is going to occupy.

Initialization → Assigning value of variable at the time of declaration is called initialization.

int var = 3;

Note - initialization of a variable doesn't mean that you can't change the value later.

```
#include <stdio.h>
int main()
{
    int var = 2; // Assigning value
    var = 3; // changing the value
    int var2;
    var2 = var; // Assigning the value of one var to another.
    printf("%d", var2);
}
```

o/p → 3

Naming conventions of Variables ⇒ Composed of letters & digits

1) You can't start the variable name with digit.
   Var1 ✓ var1 ✓ 1var ✗

2) Beginning with underscore (_) is valid but not recommended.
   _ is treated as letter
   it is used for system use. It may cause problem execution.

3) C language is case sensitive. Uppercase are different from lowercase.
   Var, var, VAR, VAr, VaR are all different.
   * usually lowercase are used to name variables
   * uppercase are used for symbolic constants.

4) Most of the special characters (@, #, !, ^, $) not allowed.
   Though (_) allowed.

5) Blanks or white spaces not allowed.
   int my variable ✗
   int my_variable ✓

6) Don't use keywords (Reserved words) to name your Variables
   like - if, else, switch etc.

7) Don't use long names

Surprise test →

Fill in the valid name of the variable.

int [ ];

a) int b) @go c) 369 d) Switch e) while

Keywords are lower case (switch)

Printf :⇒ Printf is a function which prints output on the screen.

string constant

1) Printf ("Arijit");

o/p → Arijit

placeholder

2) Printf ("%d", var); 'd' means decimal

(*) : %d is a placeholder of var

o/p - 3

it prints the value of var

3) Printf ("%d %d", var1, var2);

placeholders

o/p - 3 2

(*) For each %d you have to provide variable name. if you don't provide it will give errors.

```
#include <stdio.h>
int main()
{
    int a=2, b=3;
    int r = a+b;
    Printf ("I am adding %d & %d", a, b);
    Printf ("the result is %d", r);
}
```

o/p - I am adding 2 & 3

the result is 5

Fundamental data types - Integer :=>

Integer can take 2 bytes or 4 bytes depending on the machine. (*) more size, more content holding

1 byte = 8 bits.

To know size programatically →

sizeof is an unary operator & not a function

```
#include <stdio.h>
int main()
{ printf("%d", sizeof(int)); O/P - 4
  return 0;
}
```

Range → upper & lower limit of set of data.

For example {0,1,2,3,4}

Range - 0 - 4

Decimal number system → Human understandable number system also called as base 10 number system.

Range - 0-9

$10^2, 10^1, 10^0$

532

$= 5 \times 100 + 3 \times 10 + 2 \times 1$

$= 532$

But machine can not understand the number system.

Binary number system :=> Machine understandable number system, Also called base 2 number system.

Range - 0 → 1

$2^3\ 2^2\ 2^1\ 2^0$
1 1 0 0

$= 2^3 \times 1 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 0$
$= 8 + 4 + 0 + 0 = 12$

Range of 4 bit data -

min value = 0

max value = 15

$2^3\ 2^2\ 2^1\ 2^0$
0 0 0 0

$2^3\ 2^2\ 2^1\ 2^0$
1 1 1 1

0000 → 1111

$n = 4$

$\therefore max = 2^4 - 1$
$= 15$

Formula - $\boxed{2^n - 1}$

Range of integer ⇒

1) 2 bytes ⇒ 2 bytes = 16 bits

: unsigned range - 0 to 65535 $(2^7 - 1)$

Signed range - -32768 to 32767

$(-1(2^{n-1}) \text{ to } +(2^{n-1}-1))$

2) 4 bytes ⇒ 4 bytes = 32 bits

Unsigned range 0 to 4294967295

Signed range - -2147483648 to +2147483647

Modifiers (Long, Short, signed, unsigned) :-

Long & Short are used to either less or more memory.

if integer is 4 bytes, short int may be 2 bytes

if integer is 4 bytes, long int may be 8 bytes

(X) sizeof(short) <= sizeof(int) <= sizeof(long)

Note ⇒ By default int var_name; is signed integer variable.

Unsigned int var_name; allows only positive value.

Placeholder	Modifier
%d	signed integer
%u	unsigned integer
%ld	long integer
%lu	unsigned long integer
%lld	long long integer
%llu	unsigned long long integer

What if exceeding range ?⇒

```
#include <stdio.h>
int main ()
{
    unsigned int var = 4294967295;
    printf ("%d", var);
}
```

O/P - 4294967295

But if you exceed the range -

```
#include <stdio.h>
int main ()
{
    unsigned int var = 4294967296;
    printf ("%d", var);
}
```

O/P - 0

To understand this we have to understand how the computer store the data in its memory -

3 bit unsigned range exceeding — 4 bit

2	2	2	
0	0	0	— 0
0	0	1	— 1
0	1	0	— 2
0	1	1	— 3
1	0	0	— 4
1	0	1	— 5
1	1	0	— 6
1	1	1	— 7

8 → 1 [0 0 0] → 0

9 → 1 [0 0 1] — 1

10 — 1 [0 1 0] — 2

It's like mod function

1 mod 8 = 1

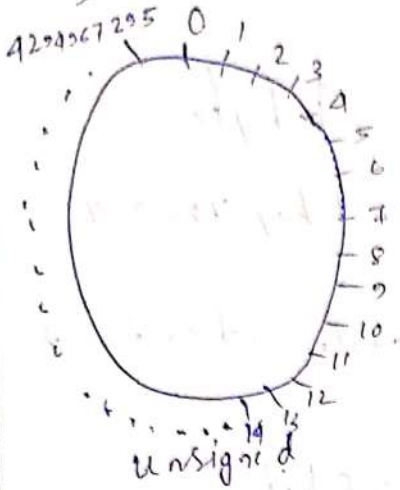
2 Mod 8 = 2

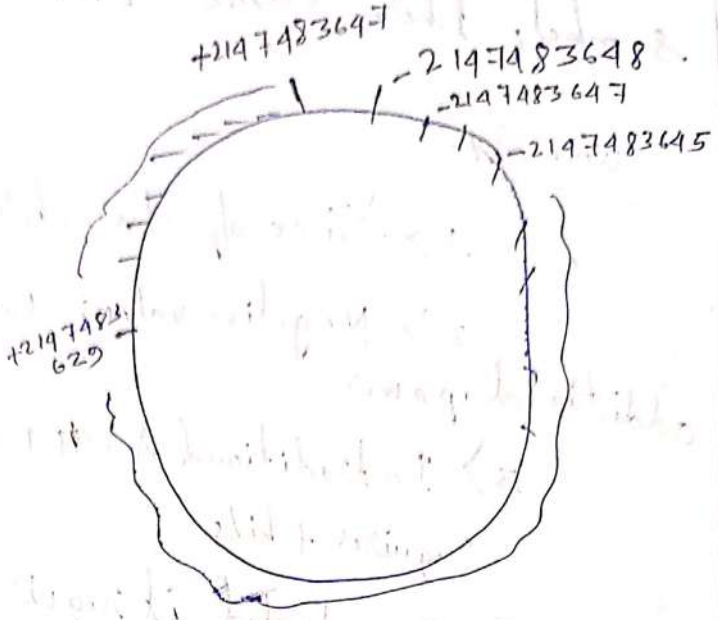
8 Mod 8 = 0

9 Mod 8 = 1

For 32 bit unsigned data — Mod_2^{32}

For n bit unsigned data Mod_2^n





Fundamental data types character :⇒

Characters are represented as 8 bits.

Common encoding scheme — ASCII.

total character 0 — 127 (signed)

ASCII requires only 7 bit to represent character.

declaration → char var_name = 'A';

it only can hold only one character at a time. You can also initialize with the integer value,

char var_name = 65;

it prints A

Size :⇒ 1 byte = 8 bits.

Range :⇒ signed — −128 → +127

unsigned 0 — 255

Extended ASCII :=> To utilize most significant bit & to represent other language symbol, there came Extended ASCII code.

Summary -

1> Size of character 1 byte.

2> Negative values won't buy you any additional power.

3> In traditional ASCII code, each character requires 7 bits

4> Incuteded it requires 8 bit

Fundamental datatypes :=>

Float, double, Long used to represent fractional numbers

Float → 4 bytes - %f (*) size of these data
Double - 8 bytes - %lf - %lf types totally depends
Long - 12 bytes - %Lf from system.

(*) Float can represent 7 digits.

(*) Double " " 16 digits

(*) Long Double " " 19 digits.

Question

(1)
```
#include<stdio.h>
int main()
{
    printf("%d", printf("%s", "Hello world!"));
    return 0;
}
```

⇒ Before giving answer to this question you have to know that

(*) %s is used to print string of character.

(*) printf not only print the ~~character~~ content on the screen. It also returns the number of characters that it successfully prints on the screen.

So first of all, Hello world! will be printed & it will return 12 as total number of the character is 12.

∴ Ans — Hello world!12

(2)
```
#include<stdio.h>
int main()
{
    printf("%10s", "Hello");
    return 0;
}
```

⇒ (*) %10s means print the character upto 10 character. As there shortage of character so first it will complete the shortage of

character with space then print the character. (14)

⇒ .....Hello (Ans)

it will print blank space till 5 character then Hello will be printed.

---

(3) #include <stdio.h>

int main()

{ char c=255;

c=c+10;

printf("%d",c);

return 0;

}

options

a) 265

b) some char

c) 7

d) 5

⇒ First of all we have to know that %d is placeholder of integer value.

* character can store value upto 255 as it occupies 1 byte = 8 bits ($2^n - 1$)

* So at first c=255. then after incrementing it became 265 which can not be stored

So the character used Modular division rule to store.

* $\frac{265}{2^n} = \frac{265}{256} = 9$

∴ it will print 9

(4) Which of the following statement is/are correct corresponding to the definition of integer.

(i) signed int i;
(ii) signed i;
(iii) unsigned i;
(iv) long i;
(v) long int i;
(vi) long long i;

a) only I & V are correct
b) only I is correct
c) All are correct
d) only IV, V, VI are correct

⇒ (c) integer is implicitly assumed. If you did not include the datatype compiler will automatically assume.

if you write signed i; it will implicitly converted into signed int i;

So all are correct

(5)
```
#include<stdio.h>
int main()
{
    unsigned i = 1; // positive
    int j = -4; // implicitly signed negative
    printf("%u", i+j);
    return 0;
}
```

a) garbage
b) -3
c) integer value depends Machine
d) None

⇒ ⊛ unsigned means it will only take positive value.

⊛ %u is the placeholder of unsigned integer.

⊛ i+j = (1+4) = -3 if there was %d then output will be -3 but as here is %u we can not represent -ve in negative format

So .in order to represent -3 in 2 complement

-3 in binary

00000000 00000000 00000000 00000011

1st complement

11111111 11111111 11111111 11111100

2nd complement => 1st complement + 1

11111111 11111111 11111111 11111101

- 4294967293

but computer to computer integer capacity varies. Some computer has the capacity of 4 byte & some has 2 byte

So answer will be — (c)

Scope of Variable

Scope = lifetime / the area under which a var is applicable or alive.

definition → A block or a region where a variable is declared, defined / used. When a block or region ends, variable is automatically destroyed.

There are 2 types of variable –

(a) Local variable.
(b) Global variable.

(a) Local variable :⇒ A variable is called local variable if it is inside any function or block. it will be called local to the function or block.

(b) Global variable → A variable is called global variable if it is outside all the function

Ex :-
```
#include<stdio.h>
int fun(); // prototype
int var = 10; // global variable.
int main()
{ int var = 3; // local to main
  printf("%d", var);
  fun();
  return 0;
}
```

```
int fun()
{
    printf("%d", var);
}
```

O/P - 3
10

(*) We can not declare or define two variable with same name within same block.

```
int main()
{
    int var = 3;
    int var = 4;
    printf("%d\n", var);
    printf("%d", var);
    return 0;
}
```

→ error: redefination of var

```
{
---
    {
    ----
    }
---
}
```

contents of outer block up to this point are visible to inner block.

content of internal block are not visible to outer block

(*) printf function will preference to print local var of that block over local var to outer block

Variable Modifier

There are four types of modifiers of variable in C

(a) auto (c) Register
(b) extern (d) static

(a) auto :=> Variables declared inside a scope by default are automatic variables.

Syntax - auto data_type var_name;

```
#include <stdio.h>
int main()
{
    int var;
    return 0;
}
```

⇒

```
#include <stdio.h>
int main()
{
    auto int var;
    printf("%d", var);
}
```

benifits - (a) it don't waste its memory.
(b) it is destroyed after execution.

(X) If you don't initialize auto variable by default it will be initialized with some garbage (random value).

(X) On the other hand global variable by default initialized to 0.

All program { global var → 0
static var → 0
auto var inside scope - garbage
extern → 0

Register - garbage value (within function)

Extern modifier ⇒

int var;

↓

Declaration + definition – allocating memory

extern int var;

↓

only declaration

(*) extern is shortname for external.

(*) used when a particular file needs to access a variable from another file.

(*) reused the variable & save the memory.

main.c

```
#include <stdio.h>
extern int a; // only declaration
int main()
{
    printf("%d", a);
    return 0;
}
```

o/p – 20

other.c

```
int a = 20;
```

```
extern int a=0;
```

↓

Compilation error we can't say extern if we initialize

(*) When we write extern no memory allocated only property is announced.

(*) multiple declaration of extern var is allowed. But not allowed for auto variable.

(*) extern variables says to compiler go outside from my scope and you will find the definition of the variable & declared.

(*) compiler believes that whatever external var says is true. Linker throws an error if it did not find the var in time of linking.

(*) When an external variable is initialized then memory for this var is allocated and it will be considered defined.

Register Modifier ⇒

Register memory → it takes less access time fast.

cache memory

main Memory } - primary Memory

Magnetic disk
tapes } - Auxilary memory

Syntax ⇒ register data_type var_name;

```
#include <stdio.h>
int main()
{
    register int var;
    return 0;
}
```

What does Register Modifier do →

(*) Register keyword hints the compiler to store a variable in register memory.

(*) This is done to reduce access time for most frequently used variable..

(*) This is the choice of compiler whether it puts the given variable in register or not.

Static Modifier ⇒ updated value

Lets first understand a program.

main.c

```
#include<stdio.h>
int main()
{
    int value;
    value = increment();
    value = increment();
    value = increment();
    printf("%d", value);
    return 0;
}
```

other.c

```
int increment()
{
    int count=0;
    count+=1;
    return count;
}
```

You might be thinking output will be 3 as we call the function three times & it incremented by 1 everytime.

But you might forget a point that

auto variable is destroyed after execution. So each time count will be initialized as 0. So the output of the above program is 1.

So to solve the problem & get our desired output = What we have to do to prevent the destroy of the memory after every execution. We can do it with static modifier.

(*) Static variable remains in memory even if it is declared within a block on the other hand automatic variable is destroyed after the complementation of function in which it was declared.

(*) Static variable if declared outside the scope of any function will act like global var but only within file it was declared.

(*) You can only assign a constant literal (value) to a static variable.

Constants :- ⇒ Once defined & cannot be modified in the code.

Defining → a) using # define.
b) Const keyword.

Syntax # define :- ⇒ # define name value
↑ Macro

Job of preprocessor (not compiler) is to...
replace Name with Value.

Ex - #include <stdio.h>

#define PI 3.14159
NAME VALUE

int main()
{
printf("%.5f", PI);
return 0;
}

O/P - 3.14159

(*) Please dont add semicolon add the end of define.

#define PI 3.14159;

(*) Always choose capital letters for NAME

because if there any variable with Macro Name it will give compilation error

#include<Stdio.h>
#define value 89
int main()
{
int value=74;
printf("%d", value);
return 0;
}

after preprocessing

#include<Stdio.h>
#define value 89
int main()
{
int 89=74;
printf("%d", value);
return 0;
}

error

(*) Whatever inside double quotes won't get replaced

(*) We can use macros like functions

```
#include <stdio.h>
#define add(x,y) x+y
int main()
{
    printf("addition : %d", add(4,3));
    return 0;
}
```

o/p - 7

(*) We can write multiple lines using \

```
#include <stdio.h>
#define greater(x,y) if (x>y)\
    printf("%d is greater than %d", x,y);\
    else\
    printf("%d is less than %d", x,y);
int main()
{
    greater (6,5);
    return 0;
}
```

o/p - 6 is greater than 5

(*) First expansion that evaluation

```
#include <stdio.h>
#define add(x,y) x+y
int main()
{
printf("result: %d", 5*add(4,3));
return 0;
}
```

⇒ 5 * 4 + 3

= 23

(*) There are some predefined Macros like __date__ and __time__ which can print current date & time.

```
#include<stdio.h>
int main()
{
printf("Date: %s\n", __DATE__);
printf("Time: %s\n", __TIME__);
return 0;
}
```

O/P — May 14 2020

Time 05:33:20

Constants using const ⇒

Syntax → const data-type varname;

We can't change a value of constant after initializing.

(*) If we try to redefine a value of a constant it will give you compilation error.

```
#include<stdio.h>
int main()
{
    const int var = 20;
    printf("%d", var);
    return 0;
}
```

o/p – 20

We can change value through pointer though with warning message.

(1)

```
int main()
{
    int var = 052;
    printf("%d", var);
    return 0;
}
```

a) 52
b) 56
c) 42
d) compile error

⇒ As there are 0 before the number it is treated as octal value

$$052 = 5\times8^1 + 2\times8^0 = 42$$

Answer will be 42

(A) if there was %.0 : then output will be
5 2

(2)

```
#include <stdio.h>
#define String "%.8\n"
#define Nesco "welcome!"
int main()
{
  printf(String, Nesco);
  return 0;
}
```

a) compile error
b) "welcome!"
c) garbage
d) welcome!

⇒ As we know preprocessor replace the macros before compilation So after preprocessing the programme will be -

```
printf("%.8\n", "welcome!");
```

O/P will be welcome!

Scanf ⇒

* Scanf stands for Scan formatted string.
* It accept character, string and numeric data from the user using standard input - keyboard.
* scanf also use format specifier as like printf -
  - %d for integer
  - %c for character
  - %s for string etc.

assigning the value.

* int var;
  scanf ("%d", &var);

Why & ? ⇒

a) While scanning the input scanf needs to store the data

b) in order to store data it need memory location.

& is also called as address-of operator.

&var → Address of Var

Example →

```
#include <stdio.h>
int main()
{ int a, b;
  printf("Enter first number");
  scanf("%d", &a);
  printf("Enter second number");
  scanf("%d", &b);
  printf("addition: %d", a+b);
  return 0;
}
```

O/P — Enter first number
3 ↵
Enter second number
4 ↵
addition - 7

---

(1)
```
#include <stdio.h>
int main()
{
  int var = 0x43FF
  print("%x", var);
  return 0;
}
```

O/P → 43FF

(2) 0x or 0X mean Hexadecimal number

Types of operator :⇒

1> Arithmetic operator :⇒

+ → addition, - → substraction, * → multiply, / → Divide, % → Modulous

2> Increment / Decrement :⇒

++ → increment, -- → Decrement

3> Relational operator :⇒

== → equal to, != → not equal to

<= → less than equal to, >= → greater than equal to

< → less than, > → greater than

(4) Logical operator :⇒

~~|| → nor~~

&& → And

|| → or

(5) Bit wise → & → bit wise and

^ → xor

| → or

~ → complement

>> → Right shift

<< → left shift

(6) Assignment :⇒

= → assignment
+= → increment then assign
-= → decrement then assign
*= → multiply then assign
/= → division then assign
%= → modulous then assign.

(7) Others →

Conditional → ? :
Address of → &
star → *
sizeof ()
, — Scope Resolution

(1) Arithmetic operator :⇒

All are binary operator means two operands need to perform operation

(#) Precedence & Associativity

Precedence	operators	Associativity
↓ Highest	*, /, %	left to Right
↓ lowest	+, -	left to Right

Note— Associativity is used only when two or more operators are of same precedence.

For example- +, -
same precedence therefore we use associativity means left most sign will be evaluated first.

Suppose, a=2, b=3, c=4

then a * b / c - here * & / are of same precedence, so left most sign * will be perform first then division will be perform.

Increment operator :=> It is used to increment the value of a variable by one.

Decrement operator :=> it is used to decrement the value of a variable by one.

increment

```
int a=5;
a++;
```
↓
a=6

Decrement

```
int a=5;
a--;
```
↓
a=4

a++; is same as a=a+1;

a--; is same as a=a-1;

(*) Both are unary operator
→ because they are applied on single operand.

(*) You can not use [rvalue] before or after increment / decrement operator

ex. (a+b)++ rvalue

++(a+b) rvalue.

(*) lvalue (left value) : simply means an object that has an identifiable location in memory (having an address).

→ in any assignment statement "lvalue" must have the capability to hold the data.

→ lvalue must be a variable as they have the capability to store the data.

→ lvalue cannot be the function like (a+b) etc

(*) rvalue (right value) : simply means an object that has no identifiable location in memory.

→ Anything which is capable of returning a constant expression or value.

→ Expression like (a+b) will return some constant value.

Question :- ⇒ What is the difference between pre-increment and post-increment operator?

⇒ Pre → means first increment / decrement then assign it to another variable.

post → means first assign it to another variable then increment / decrement

```
x = ++a;
```

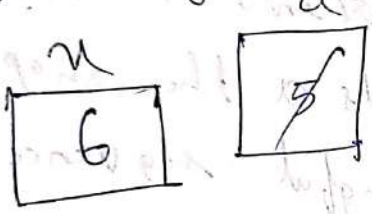
↓

first a will be increment by 1

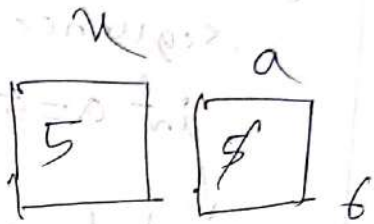
if initial a = 5

then first a = 6

then assign to x = 6




```
x = a++;
```



(1)

```
#include <stdio.h>
int main()
{ int a=4, b=3;
  printf("%d", a++ + b);
  return 0;
}
```

⇒ (3) We need to know a concept of Token generation.

(*) Lexical analysis is the first phase

(*) Lexical analyzer (Scanner) scans the whole source programme and when it finds the meaningful sequence of characters (lexemes) then it converts it into a token.

(*) Tokens: lexemes mapped into token name and attribute value

Ex- int → < keyword, int >

(*) it always matches the longest character sequence.

int a=5;

keyword	var	operator	value	
int	a	=	5	;

As it always matches longest character sequence then it scans int and convert into token.

After blankspaces it finds a then an operator but a= is not a meaningful sequence and generates a token of a

Then it generates token for =

Then it finds 5; as it is not meaningful sequence it generates token 5

(*) it can scans ++ as meaningful sequence.

So in the question a+++b [a][++][+][b]

first it scans a+ then + but as a+ is not a meaningful sequence so it generates

After a as ++ is valid operator it generates token for [++]

then as +b is not valid sequence it generate different token [+] [b]

so a = 4, b = 3 first use then increment

a+++b → a++ + b → 4+3 = 7

---

(2)

```
#include <stdio.h>
int main()
{
    int a=4, b=3;
    printf("%d", a + ++b);
    return 0;
}
```

[a] [+] [++] [b]

4 + 4

= 8

tokens [a] [+] [++] [b]

⇒ as unary operator need an operand thee ++b will be merged

a + ++b

| |

4 4

as ++b is preincrement then b will be 4 then put the value in equation

output will be 8

(3) #include <stdio.h>
int main()
{
int a=4, b=3;
printf("%d", a++++b);
return 0;
}

a) 7
b) 8
c) ,
d) error

⇒ [a] [++] [++] [+] [b]

as ++ need an operand (a++) will be merged but 2nd ++ need also an operand

(a++)++

as (a++) should be a l value means has to be a identifiable location.
But a++ is not l value
So it will produce error.

(X) int i=5;
printf("%d %d %d", i++, i++, i++);

⇒ So here all are post increment in post increment first value is assigned then incremented.

(X) According to stack Last in first out the evaluation start from right to left but printf print left to right ⇐

So if the first right expression is i++ means it first print i = 5 then incremented to 6
same happens in the other
So the output is 7, 6, 5

int i = 5;
printf("%d %d %d", ++i, ++i, ++i);

⇒ So here all are preincrement.
In preincrement first the value is incremented and print after all the equation.
So most right equation is ++i means i is incremented to 6 but not printed ∴ waiting for all the equation complete.
Same happens with the other 2.
So finally i = 8 and printf start to print from left to right
→ 8 8 8

int i = 10;
printf("%d %d %d %d\n", i++, ++i, ++i, i++);

⇒ i++, ++i, ++i, i++
↓ print 13 incremented to 14
↓ incremented to 13
↓ incremented to 13 but waiting for final
↓ print 10 incremented to 11

13 14 14 10

Relational Operator :=>

* All relational operator will return either true or false.

```
4==5          4!=5
→ false       → true
```

Logical operator :=>

* && and || are used to combine two conditions

&& returns true when all the conditions are true & returns false when any one or more than one is false.

|| returns true when one or more than one is true and returns false when all condition are false.

* ! operator is used to complement the condition under consideration

Concept of Short circuit in Logical operator :=>

Short circuit in case of &&: Simply means if there is a condition anywhere in the expression that returns false then the rest of the conditions after that will not be evaluated

```
#include<stdio.h>
int main()
{
int a=5, b=3;
int c;
c=(a<b)&&(b++);
printf("%d\n",c);
printf("%d",b);
return 0;
}
```

⇒ In the above program as $(a<b)$ is false means it will not check any other condition so b++ will not be executed

∴ c will be false (0)

b will remain 3

o/p − 0
3

(*) But if $c=(a>b)\&\&(b++)$ then first it will check $(a>b)$, as it returns true then it will check b++ and increment the value to 4

note ⇒ If an expression returns except a 1 then it will be treated as true

So c will be true & b will be 4

o/p will be 1
4

Short circuit in case of ||: Simply means if there is a condition anywhere in the expression that returns True Then the rest of the condition after that will not be evaluated

ex-

```
#include <stdio.h>
int main()
{ int a=5, b=3;
  int c;
  c = (a>b) || (b++);
  printf("c=%d\n", c);
  printf("%d", b);
```

as (a>b) is true then b++ will not be evaluated

c will be true.

b will remain same

O/P - will be 1

3

Bitwise Operator ⇒

It does bitwise manipulation.

✳ Bitwise And (&) operator →

- It takes two bits at a time and perform And operation.
- And is a binary operator
- Results of And is 1 when both bits are 1

```
7 - 0 1 1 1
4 - 0 1 0 0
-----------
  0 1 0 0
7 & 4 = 4
```

A	B	A&B
0	1	0
0	0	0
1	1	1
1	0	0

✳ Bitwise OR (|) operator →

- OR is a binary operator
- Results of OR is 0 when both bits are 0

```
7 - 0 1 1 1
4 - 0 1 0 0
-----------
  0 1 1 1
7 | 4 = 7
```

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

✳ Bitwise Not (~) operator →

- Not is a unary operator
- Results of Not is 0 when bit is 1 and 1 when bit is 0

```
7 → 0 1 1 1
-----------
8   1 0 0 0
```

~7 = 8

A	~A
0	1
1	0

Difference between bitwise And Logical operator:-

```
int main ()
{
    char x=1, y=2;
if (x & y) //false
    printf ("Result of x & y is 1");
if (x && y) //true.
    printf ("Result of x && y is 1");
return 0;
}
```

⇒ x = 1 = 0000 0001 (1)

y = 2 = 0000 0010 (2)

---

1 & 2 = 0000 0000 (0)

∴ x & y = false

but x && y = True && True

= True

Left shift operator ⇒

its a binary operator

First operand << second operand

- First operand: whose bits have been left shifted
- second operand: Decides number of places to shift the bits

(*) When bits are shifted left then trailing positions are filled with zeroes

```
#include <stdio.h>
int main ()
{
    char var=3;
    printf ("%d", var<<1);
    return 0;
}
```

3 is 0000 0011 in binary

var << 1

Left shift by 1 position

3 → 0000 0011

↓

0000 0110

↓ trailing positi.

6 in decimal

(*) Left shifting is equivalent to multiplication by 2 right operand

var = 3

var << 1

O/P : 6 [3 × 2^1]

Right shift operator ⇒

It also binary operator

First operand >> Second operand

(*) When bits are shifted right then the leading positions are filled with zeroes

```
#include<stdio.h>
int main()
{char var=3;
printf("%d", var>>1);
return 0;
}
```

var >> 1

var = 3

3 = 0000 0011

O/P - 1

0000 0001
leading zero

(*) Right shifting is equivalent to division by $2^{\text{Right operand}}$

var = 3

var >> 1

O/P - $[3/2^1]$

= 1

Bitwise XOR ^ ⇒

- It is also binary operator
- Either A is 1 or B is 1 then output is 1 but when both are 1 then B will be 0

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

7 → 0 1 1 1

4 → 0 1 0 0

0 0 1 1

7 ^ 4 = 3

```
#include <stdio.h>
int main()
{
int a=4, b=3;
a = a ^ b;
b = a ^ b;
a = a ^ b;
printf("a = %d and b = %d", a, b);
return 0;
}
```

⇒ a = 4 = 0100

b = 3 = 0011

$a = a \wedge b = \begin{array}{r} 0100 \\ 0011 \\ \hline 0111 \end{array} = 0111$

$b = a \wedge b = \begin{array}{r} 0111 \\ 0011 \\ \hline 0100 \end{array} = 0100$

$a = a \wedge b = \begin{array}{r} 0111 \\ 0100 \\ \hline 0011 \end{array} = 0011$

∴ a = 0011 = 3, b = 0100 = 4

o/p will be a = 3 and b = 4

Assignment Operator ⇒

(*) Values to a variable can be assigned using assignment operator.

(*) It requires two values L value and R-value

(*) This operator copies R value to L value.

var = 5

var → L value

= → Assignment operator

5 → R value

(*) int main()
{
char a = 7;
a ^= 5;
printf("%d", printf("%d", a+=3));
return 0;
}

a > 5
~~b > 51~~
e > 53

⇒ (*) First of all we have to remind the fact that printf not only print the output but also returns the number of character that it prints.

a = 7 = 0 1 1 1

a^=5 ⇒ a = a^5 = 7^5

0 1 1 1
0 1 0 1
---
0 0 1 0 = 2

∴ a = 2

a+=3 → a = a+3 = 2+3 = 5

∴ printf("%d", a+=3) will print 5 & return 1 as it prints only one character

∴ printf("%d", 1) it will also print 1

o/p will be 51

Conditional operator ⇒

It is a ternary operator

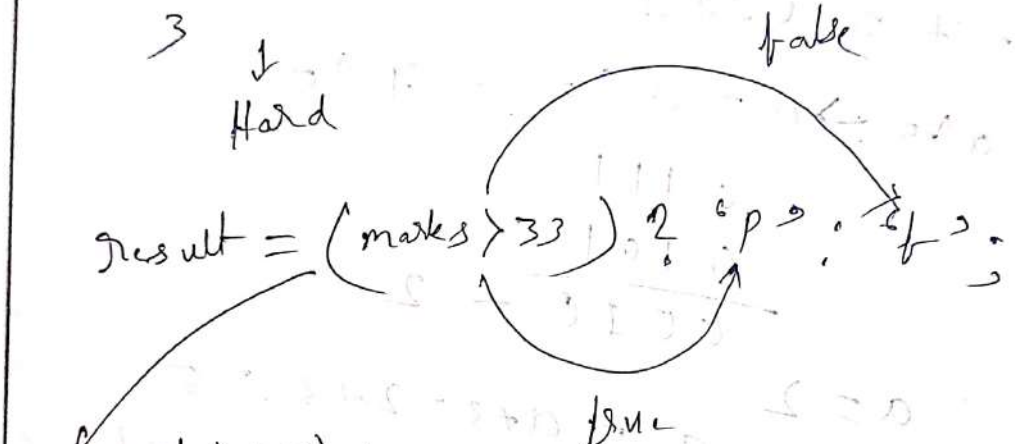
Suppose

```
char result;
int marks;
if (marks>33)
  { result = 'p';
  }
else
  {
    result = 'f';
  }
```

↓
Hard

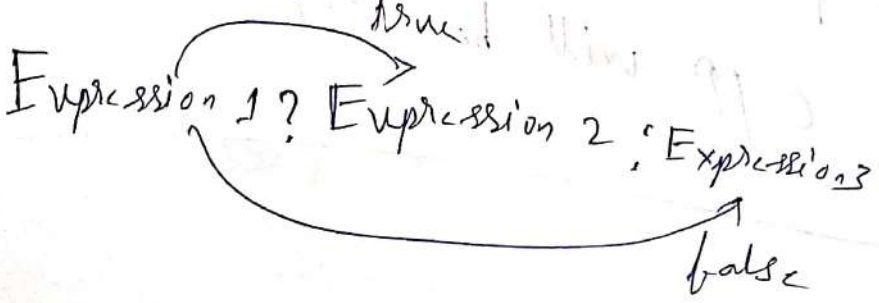
```
char results;
int marks;
result = (marks>33) ?
         'p' : 'f';
```

↓
easy



(marks>33) is an bolean expression. Therefore it will return either true or false

↳ If is a value
result is 1 value



(x)
```
#include <stdio.h>
int main()
{
    int var = 75;
    int var2 = 56;
    int num;
num = sizeof(var) ? (var2>23 ? ((var==75) ? 'A' : 0) : 0) : 0;
printf("%d", num);
return 0;
}
```

⇒ sizeof(var) is an unary operator which returns how many bytes one variable can hold. it varies from machine to machine. But it never can be 0.
∴ We all know that any number except 0 is evaluated to be true.

∴ var2 > 23 is also true
var == 75 is also true
So 'A' will be returned.
As C support auto type casting 'A' will be stored in num
∴ As in printf placeholder is %d then it will print the Ascii value of char 'A' which is 65

Ans — 65

Comma Operator ⇒

(*) Comma operator can be used as a Separator

```
int a=3, b, c;   →   int a = 3;
                     int b;
                     int c;
```

(*) Comma operator can be used as an operator

```
int a = (3, 4, 8);
printf ("%d", a);
```

Comma operator returns the right most operand in the expression and it simply evaluates the rest of the operands and finally reject them.

So here 8 will be return and assigned to a

O/p will be 8

Ex-
```
int var = ( printf ("%s\n", "Hello"), 5);
printf ("%d", var);
```

⇒ first it will evaluated the operands

So printf ("%s\n", "Hello") will be printed and rejected. 5 will be assigned to var

∴ O/p will : Hello

(*) Comma operator is having least precedence among all operator.

```
int a;
a = 3, 4, 8;
printf("%d", a);
```

⇒ As we all know comma operator has least precedence.
There also assignment operator which have more precedence.
So a=3 will be evaluated first
o/p will be 3.

(*)
```
int a=3, 4, 8;
printf("%d", a);
```

⇒ Here comma operator is behaving like a separater. Comma operator behaving like separater in — a) function call defination
b) Variable declaration
c) enum declaration

int a=3, 4, 8 is equivalent to

int a=3; int 4; int 8;

error because we can't start name with digit.

(*) int a;

a = (3, 9, 8);

printf ("%d", a);

⇒ As brackets has the highest precedence than any other operator.

8 will be assigned to a

∴ o/p will be 8

(*) #include<stdio.h>

int main()

{

int var;

int num;

num = (var=15, var+35);

printf ("%d", num);

return 0;

}

a) 15

b) 50

c) error

d) no o/p

⇒ Here as brackets has highest precedence

So var=15 will be executed then var+35 means 50 will be returned and assigned to num variable

∴ o/p will be 50

Precedence & Associativity of operator

Precedence of operators come into picture when in an expression we need to decide which operator will be evaluated first. Operator with higher precedence will be evaluated first.

Ex- 2+3*5

(2+3)*5 = 25 ✗

2+(3*5) = 17 ✓

Associativity of operators come into picture when precedence of operators are same.

Ex - 10/2*5

Left to Right - (10/2)*5 = 25 ✓

Right to left → 10/(2*5) = 1 ✗

Category	operators	Associativity
parentheses/bracket	() [] → . ++ --	Left to right
unary	! ~ ++ -- + - *, &, sizeof()	Right to left
Multiplicative	* / %	L to R
Additive	+ -	L to R
Bitwise shift	<< >>	L to R

Relational	< <= > >=	L to R
Equality	== !=	Left to Right
Bitwise And	&	L to R
Bitwise XOR	^	L to R
Bitwise OR	|	L to R
Logical And	&&	L to R
Logical OR	||	L to R
Conditional	? :	R to L
Assignment	= += *= /= %= ^=	R to L
Comma	,	L to R

Ex — int main()
{
int a;
a = fun1() + fun2();
printf("%d", a);
return 0;
}

int fun1()
{ printf("Arijit");
return 1;
}
int fun2()
{
printf("Mail");
return 1;
}

Here + has more precedence. So first + will evaluated then =

But here associativity will not come into picture as we have just one operator. Associativity will only work when we have more than one operators of same precedence

If is not defined whether fun1() will be called first or whether fun2() will be called. Behaviour is undefined and output is compiler dependent.

(X)

```
#include <stdio.h>
int main()
{
    int a=10, b=20, c=30, d=40;
    if( a<=b == d>c)
    {
        printf("True");
    }
    else
    {
        printf("False");
    }
    return 0;
}
```

⇒ As relational has more precedence than equality.

(a<=b) == (d>c)

True == True

O/P will be True.

Problem →

```
#include <stdio.h>
int main()
{
    int i = 5;
    int var = sizeof(i++);
    printf("%d %d", i, var);
    return 0;
}
```

a) 5 4
b) 6 4
c) 5 8
d) Compiler error

⇒ sizeof() operator will return 4 as int can hold 4 byte.

But the question is if the expression is evaluated or not.

So, we have to know → If the type of the operand is the variable length array type then the operand is evaluated otherwise not

O/P - 5 4

(*) int a = 1;
int b = 1;
int c = ++a || b++;
int d = b-- && --a;
printf("%d %d %d %d", d, c, b, a);

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

⇒ at first in c = ++a || b++;
++a means pre increment so a is incremented by 1 and a becomes 2
as we all know any value except 0 is true
And there c = True || b++;
We know in logical OR if one operand is true then it does not depend on other operand and result will be true.
So here comes concept of short circuit in logical or. b++ will not be evaluated.

c = 1

in the last d = b-- && --a
True && true.
b-- means post decrement b will print 1 then decremented to 0.

So: b = 0

--a means first a is decremented by 1 as previously a was 2 so a will be now 1 it will be printed.

a = 1

d = T && T = T = 1.

o/p = 1 1 0 1

Rapid Fire

(1) Sizeof operator returns size in?

⇒ a) bits
b) Bytes
c) kilobytes
d) Megabytes

→ (b) bytes

(2) Which of the following is the correct inline declaration of variables?

⇒ a) int a; b; c;
b) int a, int b, int c;
c) int a, b, c;

⇒ c) int a, b, c;

, uses as separator

(3) What does printf function returns

a) Size of integer
b) Size of character
c) number of characters printed on the screen.
d) Size of variable

⇒ (c) Number of char printed

(4) ASCII decimal range of characters A to Z is?

a) 65 – 90

b) 97 – 122

c) 100 – 127

d) 1 – 28

A – Z a) 65 – 90   a – z 97 – 122

⇒ a) 65 – 90

(5) Size of integers?

⇒ a) 32 bytes

b) 8 bytes

c) 16 bytes

d) Depends from machine to machine

d) Depends from machine to machine.

⇒ d) Depends from

(6) Consider the following variable declarations and definations in C?

i) int 3ɔ = 1;

ii) int var_3ɔ = 2;

iii) int_ = 3;

Which of the following is correct

a) both (i) & (ii) is valid

b) Only (ii) is valid

c) Both (ii) and (iii) valid

⇒ (c) Both (ii) and (iii) is valid

(7) Consider the following

int var;
extern int var;
Which is correct

a) Both statesment only declare var not define
b) Both statements declare and define the
c) statesment 1 declares & statement 2 defines
a variable
d) Statement 1 declare and define but
statement 2 just declare.

⇒ (d)

(8)
```
#include <stdio.h>
int var = 5;
int main()
{
    int var = var;
    printf("%d", var);
}
```

a) 5.
b) Compile error
c) garbage value

⇒ As var is available in global namespace
So when var is written as R value it does
not give the compilation error.
But as C cannot handle Scope Resolution

So var on the right dousnot provide any value so int var gets nothing - it makes int var = var to int var. as local variable hold garbage value so it will also give you garbage value.

(3)

```
#include <stdio.h>
int main()
{
    {
        int var = 10;
    }
    printf("%d", var);
}
```

a) 10
b) garbage value.
c) Compilation error

⇒ So as var is inside the inner block of main.
We try to access var from another inner block.
As compiler will not get access to such var it will give us compilation error.

10 #include <stdio.h>

int main()

{

unsigned int var = 10

printf("%d", ~var);

return 0;

}

a > 10 b > -10 ~~c > -11~~ d > -5

O/P –

~a = -a - 1

= -10 - 1 = -11

---

var = 10 (0000 0000 0000 1010) 4 byte

1st complement of var – (~var)

1111 1111 1111 0101 (1st complement)

%d means signed integer

MSB is 1 means negative so we have to convert to 2's complement

0000 0000 0000 1011

– 11

Conditional Constructo =>

if - else :=>

if ( is clicked)
    Application page will get opened;
else
    stay at home page;

Nested if :=>

if ( is clicked)
{ Application page will get opened;
    if (is clicked)
        play store will get opened;
}
else
    stay at home page;

if - else if :=>

if ( is clicked)
{ Applications page will be opened;
}
else if (is clicked)
{ Google chrome is opened;
}

FAQs

(*) Is it necessary to put the else parts?

⇒ The answer to this question is No and yes at the same time.

(*) Why curly braces?

⇒ if we want more than one statement to be dependent on if construct or else construct then we use curly braces.

Switch Construct ⇒

Switch is a great replacement to long elseif construct.

```
int x = 2;
switch (x)          checks if x == 1
{
    case 1: printf ("x is 1");
            break;
    case 2: printf ("x is 2");
            break;
    case 3: printf ("x is 3");
            break;
    default: printf ("default");
             break;
}
```

⇒ if none of the cases are satisfied then default will be executed.

(*) Suppose x == 1 condition is satisfied and there is no break after printf then subsequent expression will also get evaluated untill we reach the next break.

(*) In this case x == 2 is satisfied but if there is no break after printf then the statement of case 3 also executed though case 3 is not satisfied.

(*) default case can be anywhere in the programme but will evaluated after all the cases.

(*) The cases will be evaluated from Top to button.

(*) default is optional.

Facts -

(*) You are not allowed to add duplicate cases.

```
case 1 :
case 1 :
```
> - it will give duplicate case value error.

(*) Only those expressions are allowed in switch which result in an integral constant value.

```
int a=1, b=2, c=3;
switch (a+b*c)
```
allowed ✓

```
float a=1.15, b=2.0, c=3.0;
switch (a+b*c)
```
not allowed ✗

(*) Float value is not allowed as a constant value in case label. Only integer constant / constant expression allowed.

```
case 3+3:
case 3+4*5:
```
allowed ✓

```
case 3.14:
case 1.1
```
not allowed ✗

(*) Variable expressions are not allowed in case label. Although Macros are allowed.

```
int x=2; y=2, z=23;
switch (x)
{
  case y:
  case z:
```

not allowed ✗

```
#define y 2
#define z 23
int x=2
switch (x)
{
  case y:
  case z:
```

allowed ✓

Loops

Suppose you are a CEO of a MNC. You have thousands of employee. You want to send a notice to all. If you send it manually it will take huge time. So here comes Loop in rescue

```
int employee_code=1000;
While (employee_code != 0)
    { send the above notice;
      employee_code--;
    }
```

While Loop

Syntax - While (expression)
{
statements;
}

If the expression is true then the statements will be executed untill the expression is false,

Ex- int i=3;
While (i>0)
{printf (i);

o/p - 321

For loop →

Syntax- for (initialization ; condition ; increment/Decrement)
{ statements ;
}

Ex- for (i=3 ; i > 0 ; i++)
{ print (i);
}

Explanation → the first step is initialization (i=3) then it checks the condition (i > 0). If the condition is evaluated to be true then the statements are executed. Then increment / decrement happen. The same repeats untill the condition is evaluated to be false. If it is false the loop is terminated.

Do while ⇒ Syntax ⇒ do

{ statements;
} while (condition); ← important

When should i prefer do while over while?

⇒ Suppose you are asked to enter an integer untill he/she enters a value zero

```
int n;
printf ("Enter an integer \n");
scanf ("%d", &n);
while (n != 0)
{ printf ("Enter an integer \n");
  scanf ("%d", &n);
```

printf ("you are out of the loop");

using do while →

```
int n;
do
{
    printf ("Enter an integer\n");
    scanf ("%d", &n);
} while (n!=0);
printf ("You are out of the loop");
```

So whenever we wanted to execute the body of the loop atleast once then we will use do-while.

And when we want to execute the loop after checking the condition then will use while.

Break :⇒

Break is used to terminate the loop.

(*) If we encounter break then we come out with the loop without evaluating rest of the code.

Continue :⇒

Continue is used to skip the rest of code in a loop and execute the next iteration of the loop.

(-) If we use Continue in while loop then we have to use increment/decrement before Continue

(*) How many times will "Hello, world" be printed in the below programme?

```
#include<stdio.h>
int main()
{
  int i = 1024;
  for(;i;i>>=1)
    printf("Hello, world!");
  return 0;
}
```

a) 10
b) 11
c) Infinite
d) compilation error

⇒ So as i is initialized before so we simply keep the initialization step as blank then use ;

i has the value of 1024 & we know all value other than 0 is true.

i>>=1 means i = i>>1 which is right shift operator. And we know in right shift operator the value become = $\frac{\text{value of } i}{\text{right operand}}$

So here, first $i = \frac{1024}{2} = 512$

$i = 256$

$i = 128$

!

!

$i = 1$

then $i = 0$.

1024 = 10000000000

So answer will be 11

(X) What is the output of the following program fragment?

```
#include<stdio.h>
int main()
{
  int i;
  for(i=0; i<20; i++)
  {
    switch(i)
    {
      case 0: i+=5;
      case 1: i+=2;
      case 2: i+=5;
      default: i+=4;
    }
    printf("%d", i);
  }
}
```

a) 5 10 15 20
b) 7 12 17 22
c) compilation error
d) 16 21

⇒ In order to answer this question we have to understand the flow of the programme.

* At first i is initialized to 0.
* case 0 is evaluated to be true i will be increment by 5
* as after incrementation there is no break statement so the subsequents statesment (i+=2, i+=5, i+=4) will also get evaluated though other case is not satisfied.

i will be 16 after all execution.

printf will print 16 and will increment i to 17.

Again controll goes to switch statement. as none of the case is evaluated to be true so default will be evaluated. i will be 21

printf will print the 21. then i again will incremented to 22

As the condition (i < 20) is evaluated to false the loop terminates.

O/P - 16 21

(3) How many times will "Nesco" be printed on the screen?

```
int i=-5;
while (i<=5)
{
    if (i>=0)
        break;
    else{
        i++;
        continue;
    }
    printf("Nesco");
}
```

a) 10 times
b) 5 times
c) Infinite times
d) 0 times

⇒ as -5 <= 5 is true then controll will goes to while loop.

as -5 >= 0 is false then it will goes to else part.

the else part increment

happen & when it encounter continue then control jumps to condition section of while loop.

The same repeats,

i = -5	i = -4	i = -3	i = -2	i = -1
continue	continue	continue	continue	continue

now i = 0

(*) as in the if section 0 >= 0 is true then break encountered & will terminate the loop

(*) Whenever encounter printf.

∴ So O/P - is 0 times.

(*) What is the O/P of the following program.

Assuming sizeof(unsigned int) = 4

```
#include<stdio.h>
int main()
{unsigned int i = 500;
 while (i++} = 0.};
 printf ("%d", i);
 return 0;
}
```

a) infinite loop
b) 0
c) 1
d) compiler error

→ (*) Here unsigned means only positive value. As size is 4 byte we know the range 0 — 2^{32}

0 to 4294967295

And we know after 4294967295 the value will be 0 again

(*) So at first i is initialized and assigned the value of 500.

(*) next there is while loop in which there is a condition (i++ != 0). If you observe the condition you will remind this is a post increment means the value will be evaluated first then will increment happens.

Like - 500 != 0 true then i = 501

(*) As there is [;] after the while loop it means while loop has no body and the very next statement printf is not a part of it.

(*) So a time will came when i will be at 4294967295. the condition will be evaluated to be true then i will be incremented by 1. But as it will exceed the range i will be 0

(*) As 0 != 0 is false but before the termination happen i will be incremented to 1.

(*) Then printf will print the value 1

O/P will be 1

(*) What is the output of the following programme

```
#include <stdio.h>
int main()
{
    int x = 3;
    if (x == 2);   x = 0;
    if (x == 3) x++;
    else x += 2;
    printf("x = %d", x);
    return 0;
}
```

observe

a) x = 4
b) x = 2
c) compil. error
d) x = 0

⇒ (*) at first x is defined as 3

(*) if (x == 2); there is ; after that it means the next statements x = 0; is not part of it.

(*) x will be again initialized to 0

(*) 0 == 3 is false so ~~x++~~ x++ will not execute.

(*) else part will be execute and x will be 0 + 2 = 2

∴ x = 2 (o/p)

(*) What is the output of the following C program

```
int main()
{
    int i=0;
    for(printf("one\n"); i<3 && printf(""); i++)
    {
        printf("Hi!\n");
    }
    return 0;
}
```

⇒ 3

(*) In the very first line of the main function i is initialized.

(*) In the for loop first initialization executes and one will be printed. As printf returns number of character it printed. As a consiquences 4 will be returned.

(*) next in the condition checking part there is logical and operator. i<3 is true

Functions

Functions is basically a set of statements that takes inputs and perform some computation and produces output.

Syntax => return_type function_name (set_of_inputs);

(*) It is not mandatory for giving input. It can also work without any input.

Benifits :=> Reusability → once the function is defined it can be reused over and over again

Abstraction → If you are just using the function in your program then you don't have to worry about how it works inside.

Ex → scanf function

Example →

```
#include <stdio.h>
int areaOfRect (int length, int breadth)
{
    int area;
    area = length * breadth;
    return area;        // int type
}
int main()
```

```
int area = areaofRect (i, j);
printf ("%d\n", area);
i = 50, j = 10;
area = areaOfRect (i, j)
printf ("%d", area);
}
```

O/P - 50
500

Declaration ⇒ As we already know, when we declare a variable, we declare its properties to the compiler.

F.E- int var;
a) Name of variable - var
b) Type - integer

(*) Similarly the function declaration (also called function prototypes) means declaring the properties of a function to the compiler.

For example - int fun (int, int);

a) Name of function : fun
b) Return type : int
c) number of parameters : 2
d) Type of Parameter 1 : int
e) Type of Parameter 2 : int.

Note → It is not necessary to put the name of the parameters in function prototype.

int fun (int a, int b); ← must
not necessary

Ex — #include <stdio.h>
char fun(); // declaration
int main()
{
char c = fun();
printf("character is: %c", c);
}
char fun() // definition
{
return 'a';
}

O/P — character is: a

Is it always necessary to declare the functi
before using it ⇒

Not necessary but it is preferred to declare th
function before using it.

If you want to use any function bu
there is no definition of that function
before that. They you must declare the
function prototype. As compiler s can
the code from top to button, it

will produce implicit declaration of function.

But if there is defination before the calling of that function then there is no need to declare the prototype.

Function Defination ⇒ Function defination consists of block of code which is capable of performing some specific task.

For example- int add (int a, int b).

```
{
    int sum;
    sum = a+b;
    return sum;
}
```

How function works ⇒

```
#include <stdio.h>
int add (int, int); — (1)
int main()
{ int m=20, n=30, sum;
  sum = add (m,n); ---(2)
  printf ("sum is %d", sum);
}
```

```
int add (int a, int b)
    {
      return (a+b);
    }
```

③

O/P- Sum is 50

① As there is no difination of the function before calling it so we declared the prototype of the function.

* There is no need to mention names of the parameter.

② add(m, n) - this is the way you call a function.

note- while calling a function you should not mention the return type of the function. Also you should not mention the data-types of the arguments.

③ This is the way how you define a function

* it is important to mention both data type and name of parameters.

Difference between Parameter & Argument

Parameter is a variable used in declaration and defination of the function

Argument is the actual value of the parameter that gets passed to the function

Note - Parameter is called as Formal parameter

Argument is called as Actual Parameter

```
sum = add (m, n);
```

arguments
variables in calling function

```
int sum (int a, int b)
```

parameters

Call By Value

=> Here values of actual parameters will be copied to formal parameters. These two different parameters store values in different location.

```
int x=10, y=20;
fun (x, y);
```

```
int fun (int x, int y)
{ x=20;
  y=10;
```

x	y
10	20

x	y
20	10

(*) Variable x, y in function fun is local variable and when the function execution completed the variables will be destroyed. The changes you made in the variable will not get reflected back. The values of x and y will be remain same as original.

In short we just pass the value not the variable itself.

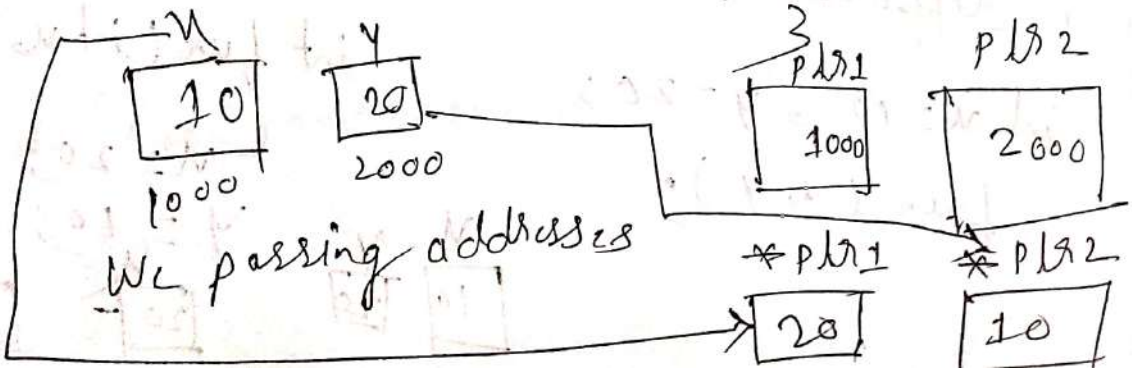
Call by Reference → Here both actual and formal parameters refers to same memory location. Therefore any changes made to the formal parameter will get reflected to actual parameter.

(*) Here instead of pass values we pass references of addresses.

```
int x=10, y=20;
fun(&x, &y);
```

&x, &y → address of operator

```
int fun(int *ptr1, int *ptr2)
{
*ptr1 = 20;
*ptr2 = 10;
```



(*) As normal variable can not hold the address so we need special type of ~~operator~~ variable called pointer variable

pointer are those variables which can hold addresses.

'*' means the value of the ptr1

* is used to access the value which is called dereference operator

(*) When we use the addresses we can change the value or do some operation and it will be reflected back to the original value.

Problems ⇒

(1)
```
int fun (int num)
{
    int count = 0;
    while (num)
    {
        count++;
        num >>= 1;
    }
    return (count);
}
```

the value returned by fun (435) is —

⇒ So here at first count is initialized at 0. as num = 435 then while (num) is true as any number other than 0 is treated as 1 (true), the loop will executed

count will be 1 and then right shift assignment operator

num >>= 1 → num = num >> 1

and we know in right shift the number is changed to num = num / 2^right operand

as here right operand is 1 then

count 1	num = 435/2^1 = 217
count 2	num = 217/2^1 = 108
count 3	num = 108/2^1 = 54
count 4	num = 54/2^1 = 27
count 5	num = 27/2^1 = 13
count 6	num = 13/2^1 = 6
count 7	num = 6/2^1 = 3
count 8	num = 3/2^1 = 1
count 9	num = 1/2^1 = 0

as while (0) is false loop will terminate. the value 9 will be returned.

---

(2)

```
int fun (int num)
{
    int count = 2;
    while (num)
    {
        count++;
        num >>= 2;
    }
    return (count);
}
```

the value returned by fun(435) is 3

16.

count = 2 | num = 935
count = 3 | num = 935/2^2 = 108
count = 4 | num = 108/2^2 = 27
count = 5 | num = 27/2^2 = 6
count = 6 | num = 6/2^2 = 1
count = 7 | num = 1/2^2 = 0

Ans is 7

17. The output of the following C program is -

```
void f1(int a, int b)
{ int c;
  c=0; a=b; b=c;
}
void f2(int *a, int *b)
{ int c;
  c=*a; *a=*b; *b=c;
}
int main()
{
  int a=4, b=5, c=6;
  f1(a,b);
  f2(&b,&c);
  return 0;
}
```

c = a - b

⇒ The calling of f1 function is useless as it is call by value and it can't change any actual parameter value.

We have to look into function f2.

Some observation -

(a) address of b and c is passed to the pointer a and b

(b) 'c' is a local variable in f2 function which holds value of pointer a means value of b [c = 5] (local to f2)

(c) Value of pointer b = value of var c (main) is assigned to value of pointer a = value of var b. [b = 6]

(d) value of pointer b is assigned to value of c (local of f2) [c = 5]

(e) a is remain same as its address is not passed. [a = 4]

∴ c — a - b = 5 - 4 - 6

= -5

(4)

```
int fun()
{
    static int num = 16;
    return num--;
}
int main()
{
    for(fun(); fun(); fun())
        printf("%d", fun());
    return 0;
}
```

What is the o/p of the C programme?

a) infinite loop
b) 13 10 7 9 1
c) 14 11 8 5 2
d) 15 12 8 5 2

⇒ Before solving ~~gv~~ question you should have knowledge of for loop

(*) Here static variable num means it is not destroyed every time after executing the function.

return num-- means it will first return value of num then decrement happen.

First initialization will happen and fun() will be called so 16 will be returned as then it will be decremented for (16; fun(); fun())

After that it will check the condition and fun() will be called

for (16; 15; fun())

So 15 is true so the control will goes to the loop and printf will execute and fun will be called and 14 will be returned.

Then incrementation (decrementation happen and fun() will be called and repeat same step until the condition is evaluated to be false.

The answer will be (c) 14 11 8 5

Static Function ⇒

Basics

(*) In C, functions are global by default. This means if we want to access the function outside from the file where it is declared, we can access it easily

(*) Now if we want to restrict this access then we make our function static by simply putting a keyword static infront of the function.

(*) Static means we cannot access the function outside the file.

(*) Reason of the same function in another file is possible. (You can create same function in another file.

file.c

```
int fun(int a, int b)
{
    int c;
    c = a+b;
    return c;
}
```

main.c

```
#include<stdio.h>
int main(void)
{
    int sum = fun(3,4);
    printf("%d", sum);
}
```

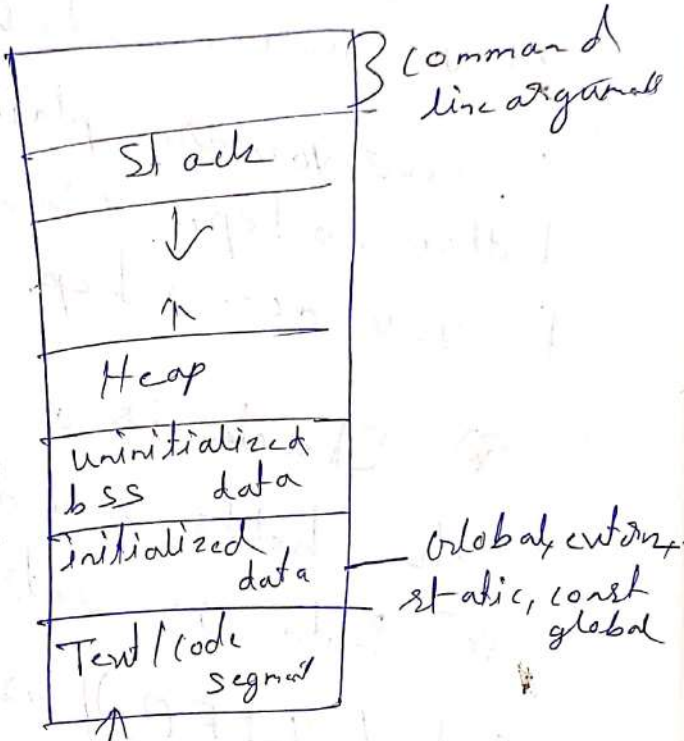
O/P → 7

But if you put static infront of fun
static int fun (int a, int b) it will
give you error - undefined reference.

Static and Dynamic Scoping ⇒

Memory Layout →

Two memory segment
1) Text / code segment
2) Data segment
   a) initialized
      i) Read only
      ii) Read write
   b) uninitialized
      (bss - Block started by symbol)
3) stack
4) Heap



* if we write static int i=0; then it will go to uninitialized because though we initialized it by default static global var is 0
* Reinitialization is not valid.
* After initialization assigning value is not allowed in global scope as

the compiler implicitly add int before Variables
in global scope

```
static int j=27;
i=45;
```
→ not allowed

```
static int i;
static int i=77;
```
→ allowed.

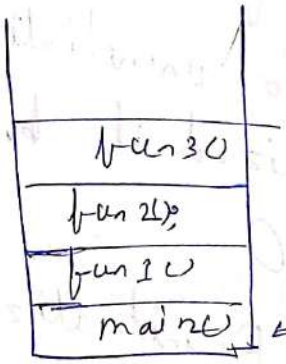
If you notice stack of book you will find whenever you are placing books you will place bottom to top.. whenever you are accessing it you access top to bottom.

(*) Stack is a container (or memory segment) which holds some data.

(*) Data is retrieved in Last in First Out (LIFO) fashion.

(*) Two operation - Push - Placing element
Pop - Retrieving element

```
main()
{
  fun1();
  return
}
fun1(){
  fun2();
  return
}
fun2(){
  fun3();
  return;
}
```



Stack (call stack)

Whenever if found return statement it will pop the function out of the stack and return to the position it left off previously.

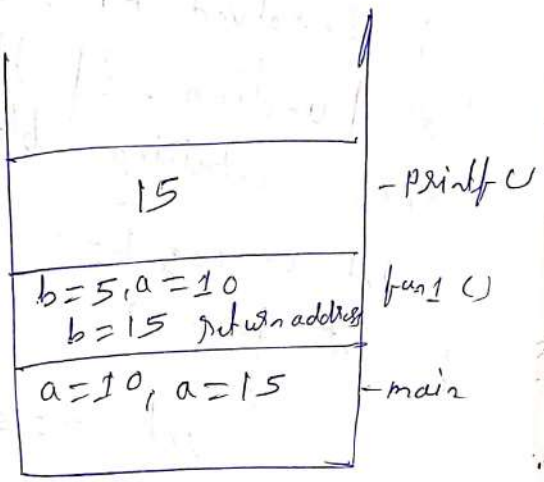
Whenever stack is empty - it indicates all function has finished their execution.

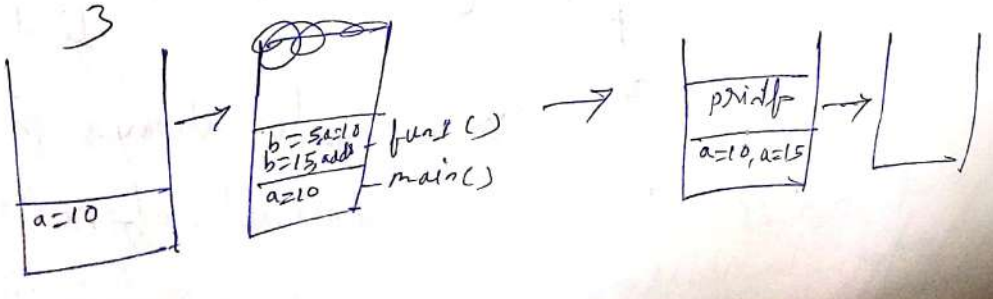
(*) Actually in call stack Activation Record maintained.

Activation Record is a portion of a stack which is generally composed of

1. Locals of the callee
2. Return address to the caller
3. Parameters of the calle

```
int main()
{
    int a=10;
    a = fun1(a);
    printf("%d", a);
}
int fun1(int a)
{
    int b = 5;
    b = b + a;
    return b;
}
```





Why Scoping ?

⇒ Scoping is necessary if you want to reuse variable names.

(*) Suppose there are millions lines of code and we will use so many variables, So according to naming convention variable names should be shorter in length. So we have minimum number of variables, we have to reuse it again and again.

```
int fun1()
{ int a=10;
}
```
scope1

```
int fun2()
{ int a=40;
}
```
scope2

Static Scoping :

⇒ Defination of a variable is resolved by searching its containing block or function. If that fails then searching the outer containing block and so on,

```
int a=10, b=20;
int fun()
{ int a=5;
  { int c;
    c = b/a;
    printf("%d", c);
  }
}
```

o/p = 4

(*) C program use static scoping

(*) As there is no a or b in definition block

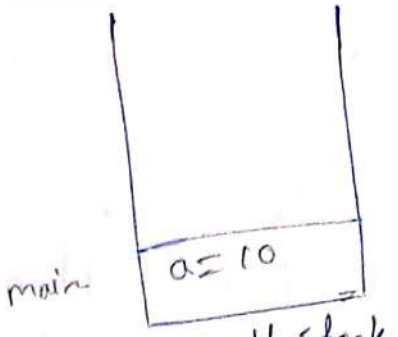
(*) In outer block we get a

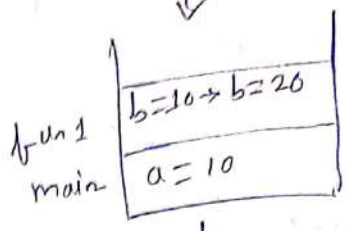
(*) Again in other block we find b

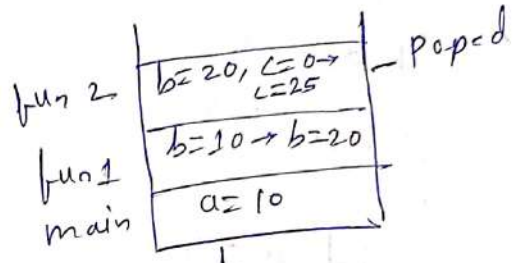
(*) Compiler always prefer Variable of nearest block

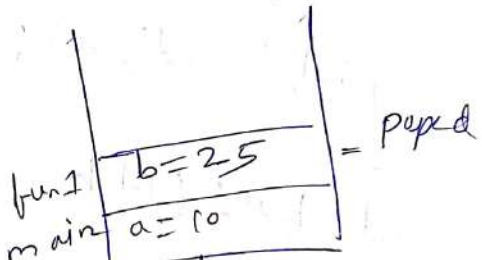
```
int fun1 (int );  // prototype
int fun2 (int );
int a = 5;
int main ()
{
    int a = 10;
    a = fun1 (a);
    printf ("%d", a);
}
int fun1 (int b)
{
    b = b+10;
    b = fun2 (b);
    return b;
}
int fun2 (int b)
{
    int c;
    c = a+b;
    return c;
}
```

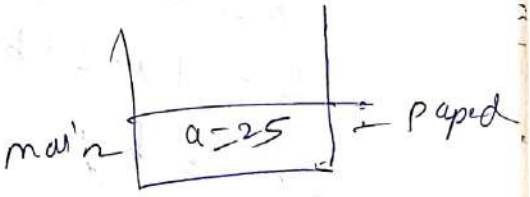
o/p - 25

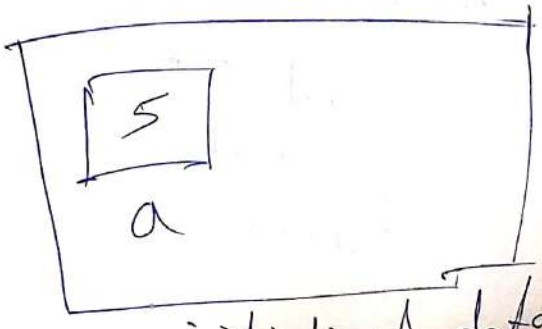






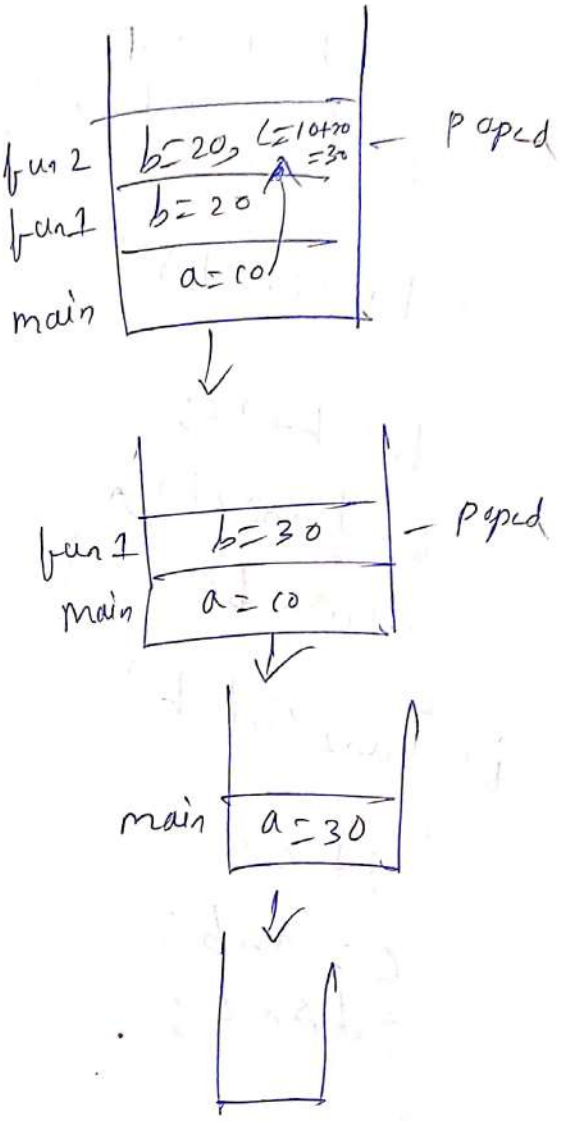






- Dynamic Scoping ⟹ Defination of a Variable is resolved by searching its containing block and if not found then searching its calling function and if still not found the function which called that calling function will be searched and so on.

```
int fun1 (int);
int fun2 (int);
int a = 5;
int main()
{
  int a = 10;
  a = fun1(a);
  printf("%d", a);
}
int fun1 (int b)
{
  b = b + 10;
  b = fun2(b);
  return b;
}
int fun2 (int b)
{ int c;
  c = a + b;
  return c;
}
```



O/P - 30

Homework Problem 0 ⇒

```
int a, b;
void print ()
{
    printf ("%d %d", a, b);
}
int fun1 ()
{
    int a, c;
    a=0; b=1; c=2;
    return c;
}
void fun2 ()
{
    int b;
    a=3; b=4;
    print ();
}
int main ()
{
    a= fun1 ();
    fun2 ();
}
```

1) with static scoping
   a) 2 4
   ~~b) 3 1~~
   c) 2 5
   d) 3 4

2) with dynamic Scoping
   a) 2 4
   b) 3 1
   c) 2 5
   ~~d) 3 4~~

⇒ 1) Note - If we don't declare the Variable in the function but assign it means we are doing this with global Variables.

The output of the program is result of printf function.

(*) As in the print function no a or b is defined compiler will goes to global scope.

(*) Global variable a was last set in fun2() to be 3

(*) Global variable b was last set in fun1() to be 1.

o/p- 3 1

2) caller function of print is fun2()
in fun2() a = 3, b = 9 so o/p will be 3 9

Important Points

(*) In most of the programming languages static scoping is followed instead of dynamic scoping.

(*) Languages Algol, Pascal, C, Haskell are statically scoped.

(*) Some languages APL, SNOBOL, LI use dynamically scopping

(4) Perl is a programming language which support both static as well as dynamic scoping.

(1) int i;
program main()
{
i = 10;
call f();
}
procedure f()
{
int i = 20;
call g();
}
procedure g()
{
print i;
}

x under static scoping
y under dynamic scoping

a) x = 10, y = 10
b) x = 20, y = 10
c) x = 10, y = 20
d) y = 20, y = 20

⇒ As i = 10 so global var i will be initialized as 10

∴ So in static scoping as there is no i in g it will go to global scope and print value of i 10

In dynamic scoping as there is no i in g it will go to its calling function and find i to be 20. So it will print 20

2) What will be the O/P of the following Pseudocode When parameters are passed by reference and dynamic scoping is assumed?

```
a = 3
void n(x)
{
  x = x * a;
  print(x);
}
void m(y)
{
  a = 1;
  a = y - a;
  n(a);
  print(a);
}
void main()
{
  m(a);
}
```

m(a) — a: 100

n(a) — a: 200

a) 6, 2

b) 6, 6

c) 4, 2

d) 4, 4

⇒ So here address of a is passed to m function.

*y = 4a

a	a
3	1
100	200

a = 3 - 1 = 2

n(200) x = 4a

x = 2 * 2 = 4

print(x) → 4

a
2
200

a
4
201

O/P - 4 4

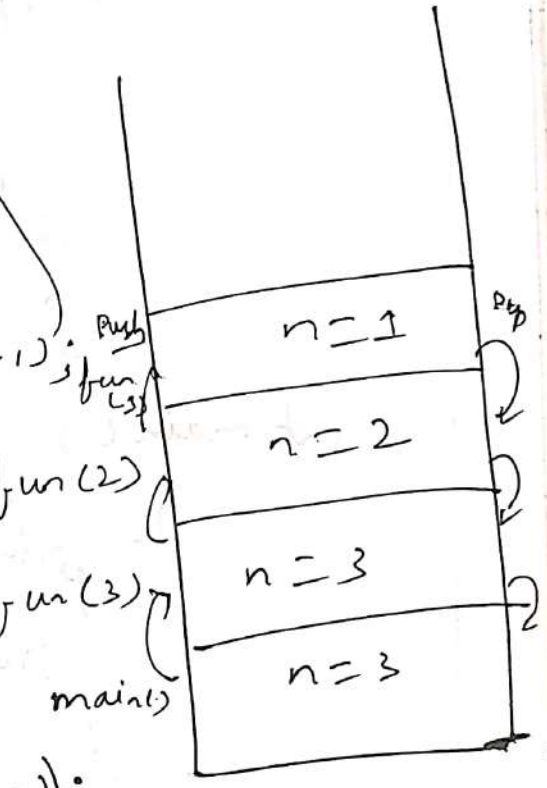
~~Array~~

Recursion

Recursion is a process in which a function calls itself directly or indirectly.

Ex- int fun()
{
:
fun();
}

```
int fun(int n)
{ if (n==1)
    return 1;
  else return 1 + fun(n-1);
}
int main()
{ int n=3;
  printf("%d", fun(n));
  return 0;
}
```



o/p - 3

fun(3) → return 1 + fun(2)

↓

return 1 + return 1 + return fun(0)

↓

return 1 + return 1 + return 1

↓

3

H.W

```
#include <stdio.h>
int fun(int n)
{ if(n==0){ return 1;
}
else return 7 + fun(n-2);
}
int main()
{ printf("%d", fun(4));
return 0;
}
```

(a) 9
(b) 7 (d) 12
(c) 15

⇒ fun(4) → 7 + fun(2) → 7 + 7 + fun(0)

→ 15

Steps to Write Recursion Program -

(i) Divide the problem in smallest sub problem
(ii) Specify the base condition to stop the recursion.

Calculate the factorial of a number

→ $5! = 5 \times 4 \times 3 \times 2 \times 1$

~~int fact (int~~

base case

fact(1) = 1

fac(2) = 2 * 1 = 2 * fact(1)

fact(3) = 3 x 2 x 1 = 3 * fac(2)

fact(4) = 4 x 3 x 2 x 1 = 4 * fact(3)

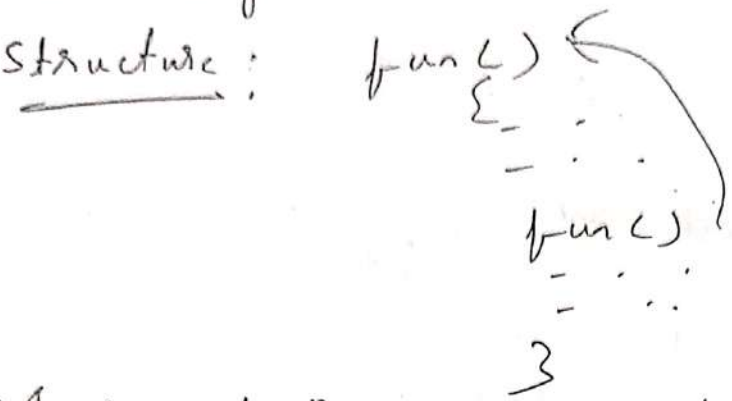
fact(n) = n * fact(n-1)

```
#include <stdio.h>
int fact (int n)
{ if (n==1)
    return 1;
  else
    return n*fact(n-1);
}
int main ( )
{ int n;
  printf ("enter no: ");
  scanf ("%d", &n);
  printf ("factorial = %d", fact(n));
```

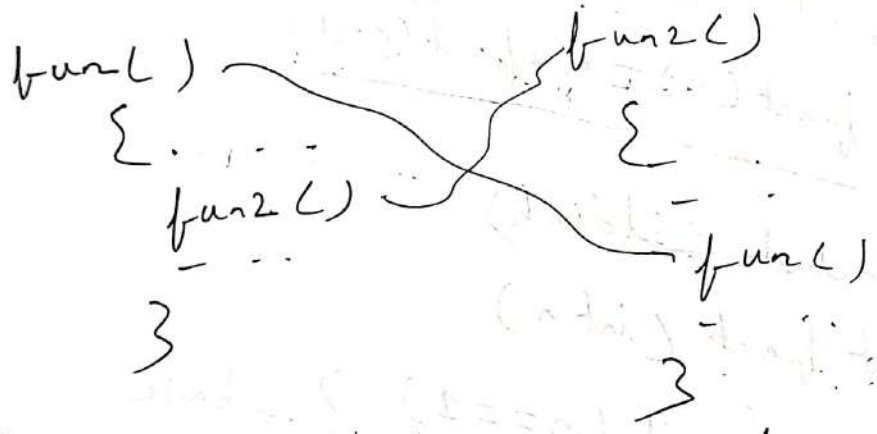
Types of Recursion

(i) Direct Recursion: A function is called direct recursive if it call the same function again.

Structure:



(ii) Indirect Recursion: A function is called indirect recursive if it calls another function and the other function call the previous funct;



Ex- WAP to print number 1 to 10 in a such way that when number is odd add 1, for even subtract 1 2 1 4 3 6 5 8 7 10 9

```
#include <stdio.h>
int n = 1
void odd (void)
{ if (n<=10)
  { printf("%d", n+1);
    n++;
even(); }
return 0;
}
void even (void)
{ if (n<=10) {
printf("%d", n-1);
n++
odd(); } return;
```

```
int main()
{ odd();
  return 0;
}
```

(iii) Tail Recursive :=> A recursive function is said to be tail Recursive if the recursive call is the last thingh done by the function. There is no need to keep record of the previous state.

```
Void fun(int n)
{ if(n==0)
    return 0;   // return back to Previous function
  else printf("%d", n);
    return fun(n-1); // last thingh
}
int main()
{
  fun(3);
  return 0;
}
```

(iv) Non tail Recursive :=> If Recursive call is not the last thing done by the function. After returning back, there is something left to evaluate

```
Void fun(int n)
    {if (n==0)
        return;
     fun(n-1);
     printf("%d", n);
    }
int main()
    {fun(3);
     return 0;
    }
```

o/p- 1 2 3

Advantage of Recursion :=>

(1) Every recursive Program can be modeled into an iterative program but recursive Program are more elegant and requires relatively less line of code.

Disadvantage

→ Recursive Program need more space (Stack space)

In iterative function call is less

(iv) Non tail Recursive ⇒ If Recursive call is not the last thing done by the function. After returning back, there is something left to evaluate

```
Void fun(int n)
{
    if (n==0)
        return;
    fun(n-1);
    printf("%d", n);
}
int main()
{
    fun(3);
    return 0;
}
```

o/p- 1 2 3

Advantage of Recursion ⇒

(i) Every recursive Program can be modeled into an iterative program but recursive Program are more elegant and requires relatively less line of code.

Disadvantage

→ Recursive Program need more space (Stack space)

In iterative function call is less

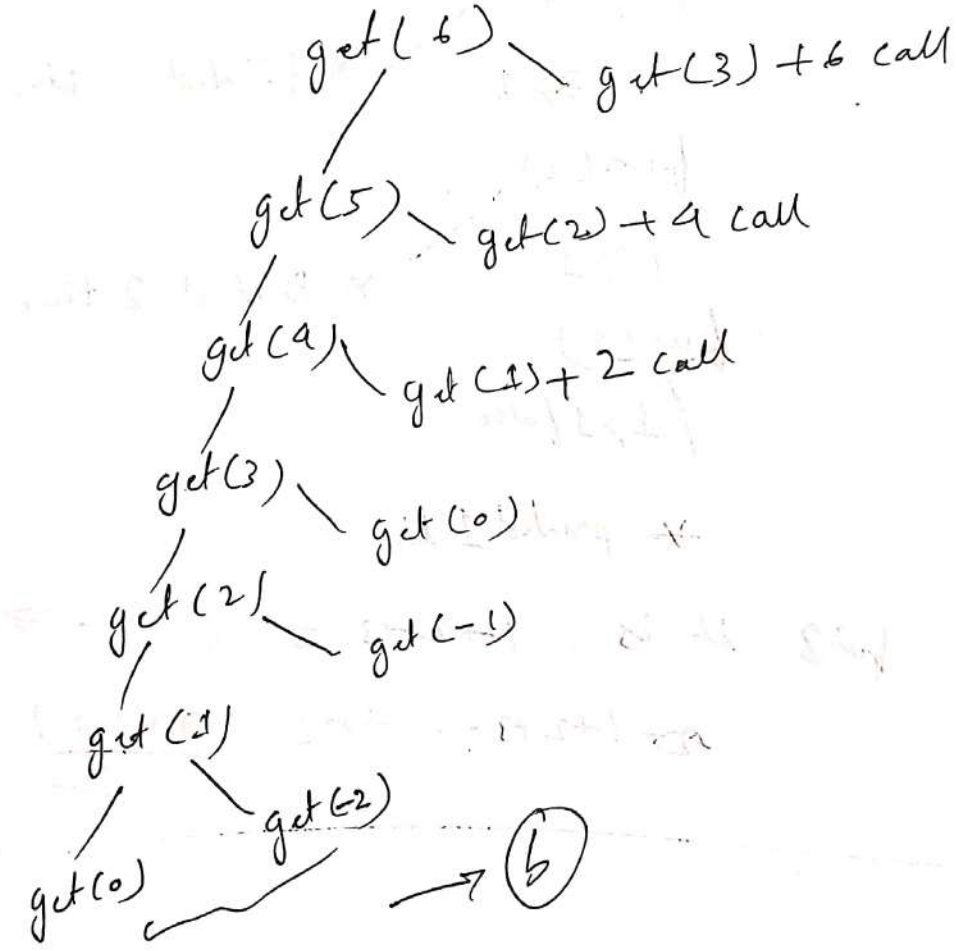
Problems :- ⇒

(1) Void get (int n)
{ if (n < 1) return;
get (n-1);
get (n-3);
printf ("%d", n);
}

if get (6) function is called in main() then how many times will the get() function be invoked before returning to the main?

(a) 15 (b) 25 (c) 35 (d) 45

⇒



(2) Void fun1(int n) how many times * will be printed.

```
{ int i=0;
  if (n>1)
      fun1 (n-1);
  for(i=0;i<n;i++)
      printf("*");
}
```

(a) n
(b) n(n+1)/2
(c) n*n
(d) none

⇒ For n=3

fun(3) → 3>1 → * Printed 3 time

fun(2) → 2>1 → * Printed 2 time

fun(1) → 1>1 false

* printed 1 tim.

for 3 it is 1+2+3 = 6 → (b)

1+2+3 .. +n = n(n+1)/2

(3)
```
int f(int j)
{ static int i=50;
  int k;
  if (i==j)
  { printf("Something");
    k=f(i);
    return 0;
  }
  else return 0;
}
```

Which of the following is true?

(a) The function return 0 for all values of j

(b) The function prints the string "Something" for all values of j

(c) The function returns 0 when j=50.

(d) The function will exhaust the runtime stack or run into an infinite loop when j=50

⇒ if j = 1, 2 or anything except 50 then if part will not executed (b) ✗

if j = 50 then Something will printed (a) ✗

if j = 50 " " " " (c) ✗

(d) ✓

(4) int fun(int n)

```
{ int u=1, k;
  if (n==1) return u;
  for (k=1; k<n; ++k)
      u = u + fun(k) * fun(n-k)
  return u;
}
```

The return value of fun(5) is

(a) 0 (b) 26 (c) 51 (d) 71

⇒ 1st Iteration k = 1

u = 1 + fun(1) * fun(4)

k = 2

u = u + fun(2) * fun(3)

k = 3

u = u + fun(3) * fun(2)

k = 4

u = u + fun(4) * fun(1)

fun(1) = 1

fun(2) = 1 + fun(1) * fun(1)

= 2

fun(3) = 1 + fun(1) * fun(2) + fun(2) * fun(1)

= 5

∴ u = 1 + fun(1) * fun(4) + fun(2) * fun(3) + fun(3) * fun(2) + fun(4) * fun(1)

= 1 + 2 * [fun(1) * fun(4) + fun(2) * fun(3)]

$fun(4) = 1 + fun(3) * fun(3) + fun(2) * fun(2) + fun(3) * fun(1)$

$= 1 + 5 + 4 + 5 = 15$

$\therefore fun(5) = 1 + 2 * (15 + 2 * 5)$

$= 51$

(5) Void count (int n)

```
{ static int d = 1;
  printf ("%d", n);
  printf (" %d", d);
  d++
  if (n > 1) count (n-1);
  printf ("%d", d);
}
int main ()
{ count (3);
}
```

(a) 3 1 2 2 1 3 4 4 4

(b) 3 1 2 1 1 1 2 2 2

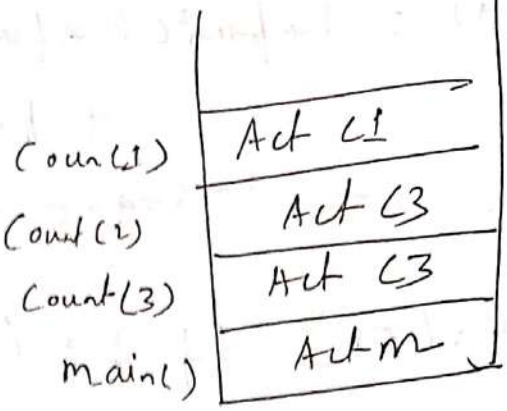
(c) 3 1 2 2 1 3 4

(d) 3 1 2 1 1 1 2

→ (a) ✓

if not Static then

3 1 2 1 1 1 2 2 2



After printing 3 1 2 2 1 3 4 we again come back to coun(2) (calling function) and the next statement (print 4) will be executed.

Coun(2), count(3) reactivated

Rapid Fire Question ⇒ iVesco Academy

(1) Which is true about static function in C?

(a) Static function are global function

(b) Static function are restricted to the files where they are declared

(c) There is no concept like static function in C

(d) none

(2) In C, it is mandatory to declare a function before use.

(a) True

(b) False

declaration - tells compiler

function name

Return type

Parameters

(3) Which keyword is used to come out of loop only for that iterations?
(a) break (b) continue (c) return (d) none.

(4) Correct function Prototype
i. int fun(int var1, int var2);
ii. int fun (int, int)
iii. fun (int, int)
(a) only i (b) only ii & iii (c) only if ii
(d) All

(5) C support dynamic Scoping
(a) yes (b) no

(6) In C parameter are always
(a) Passed by value (b) Passed by References
(c) both (d) none.

(7) Void fun(); meaning
(a) function can only be called without any parameter
(b) Function can be called with any number of parameter of any type
(c) Function can be called with any number of integer Parameter
(d) Function can be called with one integer

(8) Assume int size is 4 byte

```
#include <stdio.h>
int main()
{ printf("Hi\n");
  main();
  return 0;
}
```

(a) We can't use main() inside main()

(b) Hi will be printed $2^{31}-1$

(c) Hi would be printed untill stack overflow happens

(d) Hi will be printed once because main() inside main() is ignored.

(9) In a context of 'break', 'continue' statement which is best statement?

(a) 'break', 'continue' can be used in for, while, do while loop body & 'switch' body

(b) break, continue in loop body but only break in switch body

(c) break & continue can be loop, switch & if else body

(d) none

(10)
```
int main()
{ int i=9
  for(;i;){
      printf("Hmm");
      i--;
  }
}
```

(a) Hmm will be printed 10 times
(b) " " " 9 "
(c) Compile error (d) none.

(*) As per C standard continue can only be used in loop body

(*) In C parameter are always passed by value. Pass by reference is simulated in C by explicitly passing pointer value.

(*) C only support static scoping

(*) In C it is not mandatory to declare a function before use.
definition is Mandatory void add(int,int);

Array ⇒ An array is a data structure containing a number of data values (all of which are same type)

a | 5 | 6 | 8 | 10 | 15 | 16 | 18 | 20 |

a is an array of integer type and has 8 values

c | 'a' | 1 | 2 | 'b' | X not array type not same

→ One dimensional array

Syntax for declaring

data_type name_of_array [number of elements];

For ex- int arr[5]; // declare + definition

arr

0	1	2	3	4

definition — allocating memory

memory allocated = 5 * size of (int)

(*) The length of an array can be any positive integer constant expression

int arr[5]; ✓ int arr[-5]; ✗

int arr[5+5]; ✓ int a

int arr[5*3] ✓ int arr[a=2/3]; ✓

(*) Specifying the length of an array using Macro is considered to be an excellent practice

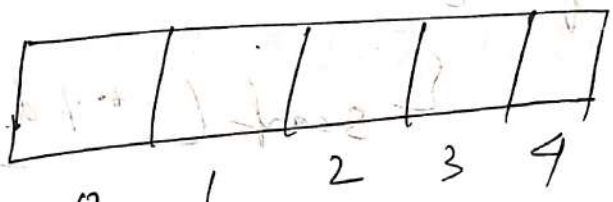
```
#define N 10
int arr[N];
```

because if we want to change length of array we have to just change one value

Accessing Elements :=>

To access array element

array_name[index]



index

arr[0] - value of first element

index range : 0 — (n-1)

Initializing 1D array =>

(i) int arr[5] = {1, 2, 5, 6, 3}

(ii) int arr[] = {1, 2, 5, 6}

(iii) int arr[5]; Mandatory

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

not preferred

(iv) int arr[5];
for(i=0; i<5; i++)
{ scanf("%d", &arr[i]); }

What if number of elements are less than the length specified?

=> int arr[10] = {1, 2, 6, 3, 0}

↓

the remaining field will be initialized as 0

{1, 2, 6, 3, 0, 0, 0, 0, 0, 0}

(X) int arr[10] = { };X ⇒ illegal

you must have specified at least one element

it is also illegal that you can not add more number of array than it ~~accepts~~ initialized.

int arr[2] = {1,2,3} —X

Designated initializing :⇒

int arr[10] = {1, 0,0,0,0, 2,3,0,0,0};

We want 1 in position 0
2 in 4
3 in 5
rest are 0

int arr[10] = {[0]=1, [5]=2, [6]=3};

[0]=0 — index - value — all others are 0

[0] ← Designater

Advantages: no need to bother about order

{[0]=1, [5]=2}

{[5]=2, [0]=1}

What if I wont mention the length

⇒ Designator could be any non-ve integer

Compiler will deduce the length of array from the largest designator in the list

```
int a[] = {[5]=90, [20]=4, [4]=3
           [45] = 78};
```

↑ largest

length will be 50

(E) you can mix both traditional & designated

```
int a[] = {1, 7, 15, [5]=90, 6, [8]
          =9};
```

if there is any clash then designator initializer will win

```
int a[] = {1, 7, 5, [2]=90};
```

at index 2 there is 5 & also 90 — 90 will be considered

equivalent

```
int a[] = {1, 7, 90};
```

④ Counting number of elements :

We know sizeof() operator returns how many bytes it occupying in Memory

: number of elements = $\frac{\text{sizeof(name\_of\_array)}}{\text{sizeof(name\_of\_array[0])}}$

(sizeof(name_of_array) → total; sizeof(name_of_array[0]) → 1 element)

```
int a[] = {1,2,3,4}
printf("%d", sizeof(a)/sizeof(a[0]));
```

⇒ 4

Multidimentional array :=>

It is an array of array

Declaration is — data_type name_of_array [size1] [size2] ... [sizen]

For —

— int a[3][4]; // two dimentional array

Total number of elements :=

a[10][20] => 10 × 20 = 200

size of a[10][20] = 200 × 4 — Dependent on machine

= 800 bytes

Constant array ⇒

Either one dimentional or multidimentional array can be made constant by starting the declaration with const.

For ex

```
const int arr[10] = {1,2,3,4,5,6,7,8,9,10};
```

⇓

We can't change any value

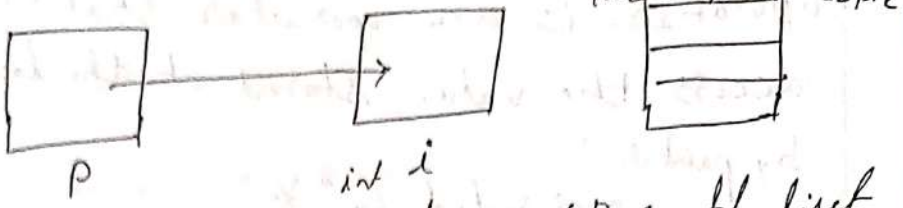
```
a[1] = 45; // error
```

Advantage

- if values are valuable then const is used.

Defination :=> Pointer

Pointer is a special variable that is capable of storing some addresses



It points the memory location where the first byte is stored.

p → 1001

Declaring & initializing :=>

General syntax → datatype *pointername

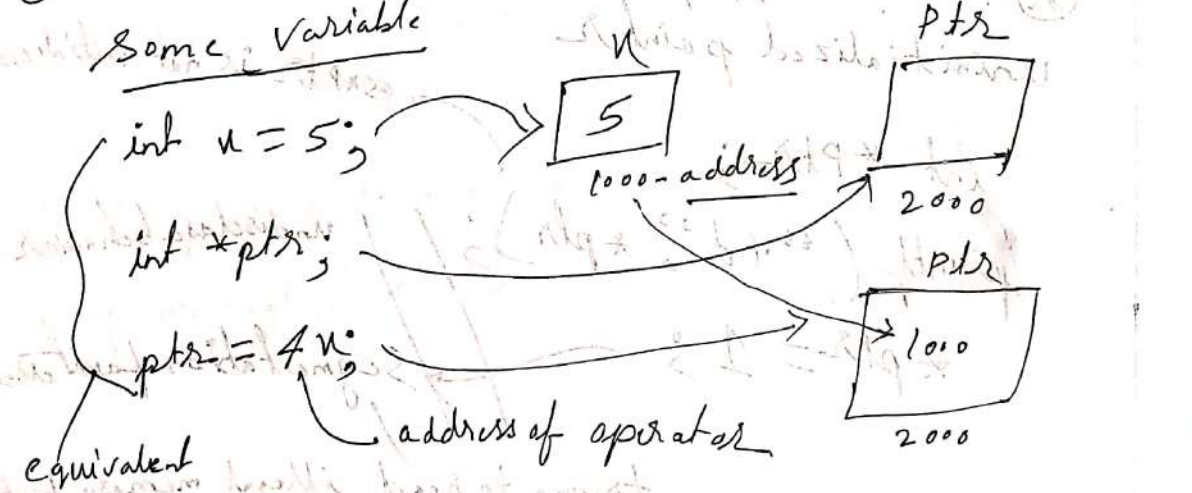
(Refers to type of value that the pointer will point to)

ex- int *ptr → ptr point to integer

char *ptr → ptr point to character.

Declaring pointer is not enough but we have to initialize it before use.

(i) One way to initialize is to assign address of some variable



ptr - address
*ptr - value of the address

Dereference of operator / Value of operator ⇒

Value of operator / indirection / dereference operator is an operator that is used to access the value stored at the location pointed by pointer

represented by "*"

```
int x = 5;
int *ptr;
 ptr = &x;
printf("%d", *ptr); // 5
```

- We can also change the value of the object pointed by pointer

```
int x = 10;
 int *ptr;
  ptr = &x;
 *ptr = 4;
printf("%d", *ptr); // 4
```

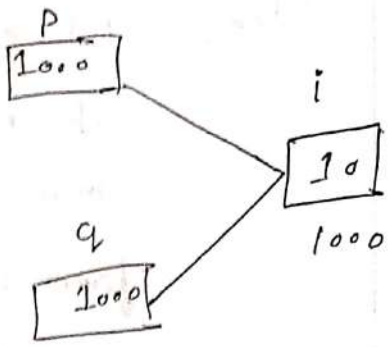
⊛ Never apply dereferencing operator to the uninitialized pointer

```
int *ptr;                          // as *ptr is not initialized
printf("%d", *ptr);   // undisclose behavior
 *ptr = 1;            // Segmentation fault error
                      // trying to read illegal memory locati.
```

Pointer assignment to another Pointer:

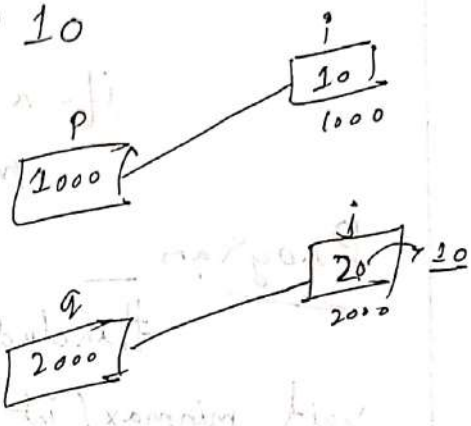
(1)

```
int i = 10;
int *p, *q; // declaring pointer
p = &i ; // initialize Pointer p
q = p ; // q is assigned address of var_i
printf ("%d %d", *p, *q); // 10 10
```



(2)

```
int i=10, j=20;
int *p, *q; // declaring
p = &i ;
q = &j; // initializing
*q = *p; // Here the value pointed by p will be copied to value pointed by q
printf ("%d %d", *p, *q); // 10 10
```



H.W problem:

```
int i = 1;
int *p = &i;
q = p;
*q = 5;
printf ("%d", *p);
```

⇒ As q is not declared then it will give error if we neglect it then O/P will be 5

Finding Largest & Smallest Elements in an array :=>

Idea -

```
int a[] = {23,45,6,98};
int min max;
min = max = a[0];
```

$len = \frac{sizeof(a)}{sizeof(a[0])}$

```
from i = 1 to 3
    if a[i] < min then
        min = a[i]
    if a[i] > max then
        max = a[i]
```

Program

```
#include <stdio.h>
void minmax(int a[], int len, int *min, int *max)
{   *min = *max = a[0]
    for (int i=1; i<len; i++)
    {   if (a[i] < *min)
            *min = a[i]
        if (a[i] > *max)
            *max = a[i]
    }
}
int main()
{
    int a[] = {23,45,6,98};
    int len, min, max;
    len = sizeof(a) / sizeof(a[0]);
    minmax(a, len, &min, &max);
    return 0; printf("%d %d", min, max); }
```

Returning Pointer :=>

```
int main()
{
    int a[] = {1,2,3,4,5}
    int n = sizeof(a) / sizeof(a[0]);
    int *mid = findmid(a, n);
    printf("%d", *mid);   // 3 (o/p)
    return 0;
}
int *findmid(int a[], int n)   // as we are returning pointer/address
{   return &a[n/2];   // a[2] = 3
}
```

Caution X: Never ever try to return the address of an automatic variable / local

```
int *fun()
{   int i=10;
    return &i;
}
int main()
{   int *p = fun();
    printf("%d", *p);
    return 0;
}
```

// error - function returns address of local variable

After returning local var is destroyed.

(1) ⊗⊗ int *p = &i;

↳ it is not an dereferencing operator it is just saying compiler that p is a pointer

✗ *p = &i; // not valid as *p is a value of an variable where &i is address, We can't assign address to some integer variable.

* has different meaning in different context

(2) What is O/P-

```
void fun (const int *p)
{
    *p = 0;
}
int main ()
{
    const int i = 10;
    fun (&i);
    return 0;
}
```

Should same

⇒ We can't reassign value of constant ~~operator~~ variable.

it will give error Message

Assignment of read only location *p

(3) How to print address of a var?

⇒ Use %p as a format specifier in printf

```
int main()
{  int i=10
   int *p=&i;
printf("The address is %p", p);//The address is
                               0x7ffd5b9a98c
}
```

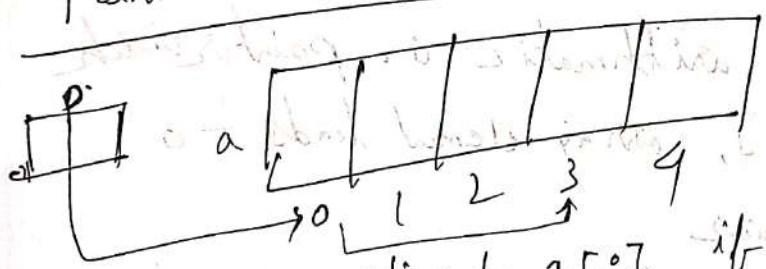
(4) If i is a variable and p points to i, which of the following expression are aliases of i

(a) *p (b) *&p (c) &p (d) *i (e) *&i

⇒ int i=10; } - given
int *p=&i; }
*p=10, &i=p → **&i=*p=i

(a) (e) ✓

Pointer Addition

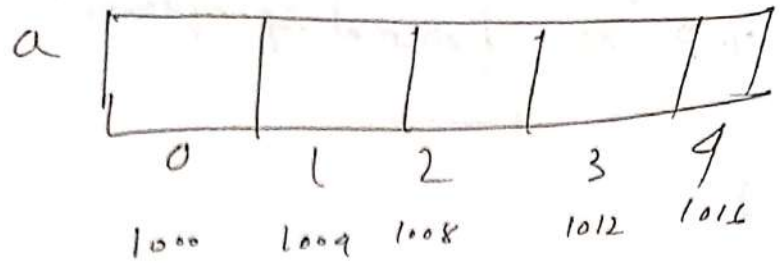


Pointer p is pointing to a[0] if we do p=p+3 then it will point to p → a[3]

(*) Initially if p points to a[i] then

p=p+j ≡ &a[i+j]

But we know integer takes 4 bytes in memory



$p = p + 1 \rightarrow p = 1000 + 1 \times 4 = 1004$

p = p+n means n shift towards right from initial address

Subtraction → p → a[i]

p = p-j ≡ &a[i-j]

p = p-j means j shift towards left from initial address

Subtraction between two pointer

$$*p - *q = \frac{4p - 4q}{\text{Size of the datatype}}$$

- Performing arithmatic on pointer which are not pointing to array element leads to undefined behavior

```
int main()
{
    int i = 10
    int *p = &i;
    printf("%d", *(p+3));
    return 0;
}
```

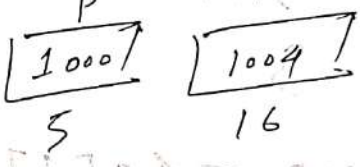
// Different o/p everytime.

* two pointer pointing to different array then performing subtraction between them leads to undefined behaviour.

Pointer increment & Post increment :=>

```
int main()
{ int a[] = {5, 16, 7, 8};
  int *p = &a[0];
  printf("%d", *(p++));
  printf("%d", *p); // p → a[1]
  return 0;
}
```

as post increment first value is printed then incremented



=> 5 16

Comparing pointer :

- Only possible when both pointer point to same array
- use relational operator & equality operator
- output: 1 true, 0 false

```
int main()
{ int a[] = {1, 2, 3, 4, 5, 6};
  int *p = &a[3], *q = &a[5];
  printf("%d", p <= q); // q is address is larger
  printf("%d", p >= q);
}
```

p → 1, q → 0

H.W o/p of the following program

```
int main()
{ int a[] = { 5,16,7,8,45,32,23,10 }
  int *p = &a[1], *q = &a[5];
  printf("%d", *(p+3));
  printf("%d", *(q-3));
  printf("%d", q-p);
  printf("%d", p<q);
  printf("%d", *p<*q);
  return 0
}
```

(a) 45 7 4 1 1
(b) 45 4 7 1 1
(c) 44 7 4 1 0
(d) 45 7 4 1 0

⇒ *p = &a[1] = 16

*(p+3) = &a[4] = 45.

*(q-3) = &a[2] = 7.

q - p = (1020 - 1004)/4 = 4

p < q — 1004 < 1020 true — 1

*p < *q — 16 < 32 true — 1

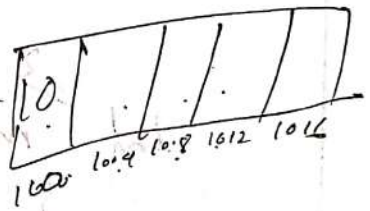
(a) ✓

Sum of array using pointer.

```
int main()
{ int a[] = { 11, 22, 36, 5, 2 };
  int sum = 0, *p;
  for (p = &a[0]; p <= &a[4]; p++)
      sum += *p;
  printf ("sum is %d", sum);
  return 0;
}
```

Fact → Name of an array can be used as a pointer to the first element of an array.

ex -
```
int main()
{ int a[5];
  *a = 10;
  printf ("%d", a[0]);
  return 0;
}
```

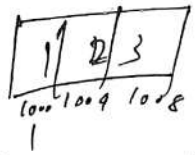


if *(a+1) = 10 then a[1] is 10

*(a+i) = n then a[i] = n

Caution - Assigning a new address to array name will cause error

```
int a[] = { 1, 2, 3 };
printf ("%d", a++);
```



1000 = 1000 + 1

Passing array as an argument ⇒

(*) int a[] = {1,2,3} is an array and a represent the address of the 1st element (like pointer behave)
So we just pass add(a, len)
base address

Ex

```
int add (int b[], int len)
    {int sum=0, i;
for (i=0; i<len; i++)
    sum += b[i];
    return sum;
}
int main()
{
    int a[] = {1,2,3};
    int len = sizeof(a)/sizeof(a[0]);
printf("%d", add(a, len));
    return 0;
}
```

2D array printing using pointer:

```
for (p=&a[0][0]; p<=&a[row-1][col-1]; p++)
    printf("%d", *p);
```

(1) Consider char a[1000][100]

Assuming Main memory byte addressable and array is stored starting from 0. address of a[40][50] is

(a) 4040 (b) 4050 (c) 5040 (d) 5050

Number of cols

$$\Rightarrow \&a[i][j] = BaseAddress + [(i - lb_1) \times Nc + (j - lb_2)] \times c$$

byte

Here $BA = 0$, $lb_1 = 0$, $c = 1$

$$\&a[40][50] = 0 + (40 \times 100 + 50) \times 1$$

$$= 4050$$

(2) O/P of following C code. base address 2000

It n int require 4 byte

```
#include <stdio.h>
int main()
{
    unsigned int x[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
    printf("%u, %u, %u", x+3, *(x+2), *(x+2)+3);
}
```

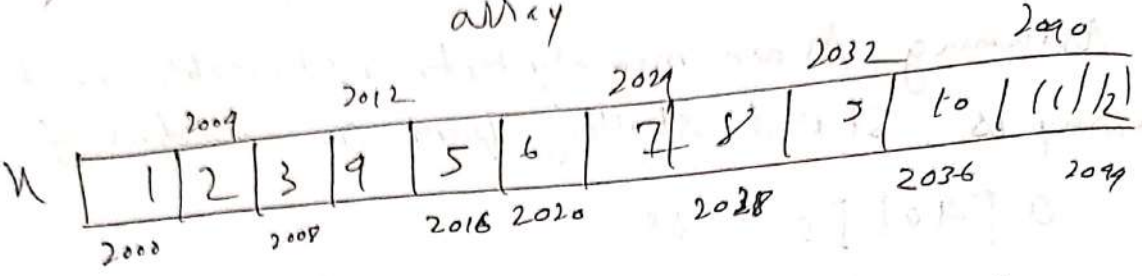
a) 2036, 2036, 2036

(b) 2012, 9, 2209

(c) 2036, 10, 10

(d) 2012, 9, 6

⇒ x = 2000 pointer to the first 1D



x+3 = Pointer to the 4th 1D array

= 9 elements to pass

= 2000 + 9×4 = 2036

*(x+3) = value first element of 4th 1D array

= 10

*(x+2) = value of first element of 3rd 1D arr.

= 7

7+3 - 3rd element after 7 - 10

(c) ✓

Shortcut after getting 2036 we know

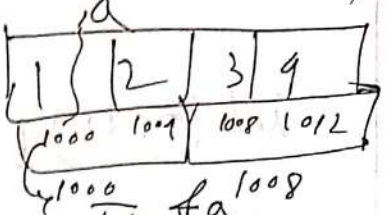
* means value So a b ✗ c ✓

d ✗

Pointer Pointing to entire array

```
int (*p)[5];
```

pointer to the whole array of 5 element



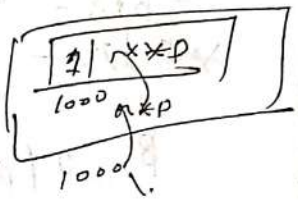

```
int (*p)[5] = &a;
```

a means first element address (&a)

&a means whole array address

Printing value

```
int (*p)[5] = &a;
printf("%d", **p);
```



Problem

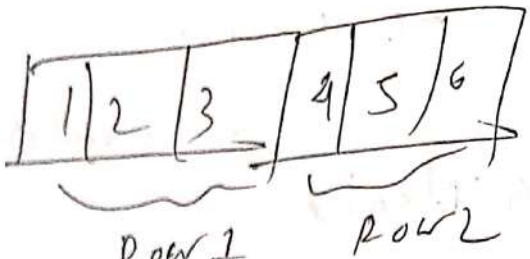
```
#include <stdio.h>
int main()
{
    int a[][3] = {1,2,3,4,5,6};
    int (*ptr)[3] = a;
    printf("%d %d", (*ptr)[1], (*ptr)[2]);
    ++ptr;
    printf("%d %d", (*ptr)[1], (*ptr)[2]);
    return 0
}
```

Legal as we know column

(a) 2 3 5 6 (c) 4 5 0 0

(b) 2 3 4 5 (d) none

⇒ int a[][3] is Legal as we specify column name.



int (*ptr)[3] — pointer to a 1D array of 3 elements, = a — address of 1st 1D array

(*ptr)[1] = *(*ptr + 1) gives 2nd element

ptr — point to 1D array 1st

*ptr — pointer to 1st element of 1D arr.

o/p — 2 3 5 6

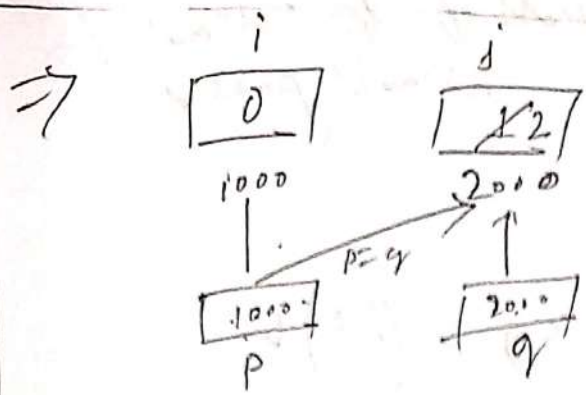
++ptr — ptr to the 2nd 1d array

---

(5) o/p —

```
void f (int *p, int *q)
{
  p = q;
  *p = 2;
}
int i = 0, j = 1;
int main()
{
  f(&i, &j);
  printf("%d %d", i, j);
  return 0;
}
```

(a) 2 2
(b) 2 1
(c) 0 1
(d) 0 2



→ (d) 0, 2

(b) O/P –

```
#include <stdio.h>
int f(int *a, int n)
{
  if(n<=0) return 0;
  else if(*a%2==0) return *a + f(a+1,n-1);
  else return *a - f(a+1,n-1);
}
int main()
{
  int a[] = {12,7,13,4,11,6};
  printf("%d", f(a,6));
  getchar();
  return 0;
}
```

(a) → (b) 5

(c) 15 (d) 13

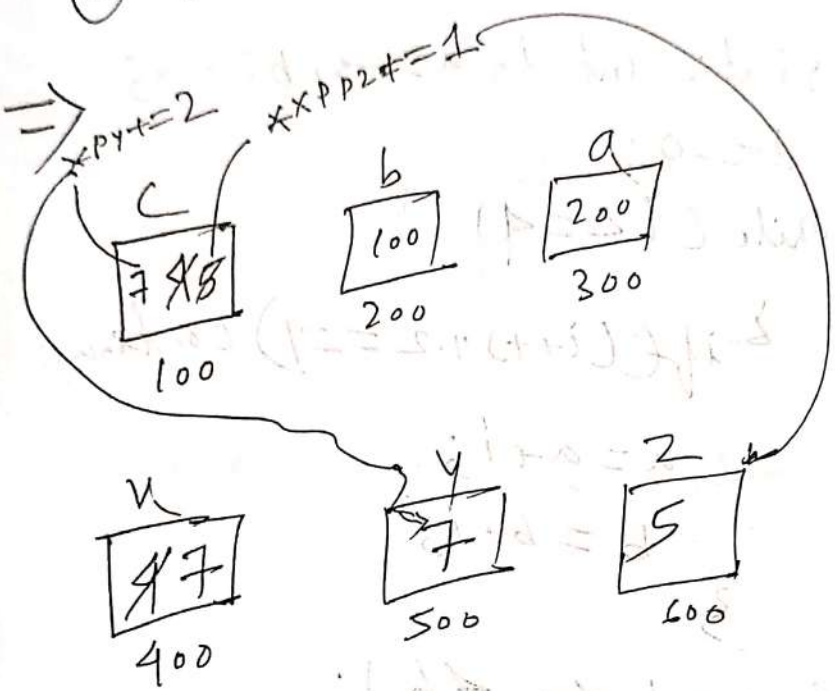
f(a,b) means address of first element of array is passed

f(a, b)
|
12 + f(7, 5) = 15 ✓
|
7 - f(13, 4)
|
13 - f(4, 3)
|
4 + f(1, 2)
|
1 - f(6, 1)
|
6 + 0

(7) int f(int x, int *py, int **ppz)
{ int y, z;
**ppz += 1;
z = **ppz;
*py += 2;
y = *py
x += 3;
return x+y+z;
}

```
Void main
{
    int c, *b, **a;
    c=4, b=&c   a=&b;
    printf("%d", f(c,b,a));
}
```

(a) 18 (b) 19 (c) 21 (d) 22



7+7+5=19

(8)
```
void swap( int *x, int *y)
{ static int *temp;
  temp = x;
  x = y;
  y = temp;
}
void printab()
{ static int i, a=-3, b=-6;
  i=0;
  while (i<=4)
  { if((i++)%2==1) continue;
    a = a+i;
    b = b+i;
  }
  swap(&a, &b);
  printf("a=%d, b=%d", a, b);
}
main()
{ printab();
  printab();
}
```

(a) a=0 b=3
a=0 b=3

(b) a=3, b=0
a=12 b=9

(c) a=3, b=6
a=3 b=6

(d) a=6 b=3
a=15 b=12

i = 0 if false but as post increment i = 1

a = -2

b = -5

i = 1 if true i = 2 continue (escaping next ins)

i = 2 if false i = 3

a = 1

b = -2

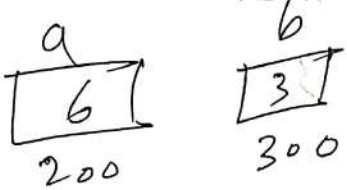
i = 3 if true i = 4 continue

i = 4 if false i = 5

a = 6

b = 3

In Swap



no use as no affect on a & b value

→ (d) ✓

(2)
```
#include <stdio.h>
int main()
{
    int i,j;
    char a[2][3] = {{'a','b','c'},
{'d','e','f'}};
    char b[3][2];
    char *p = *b;
    for(i=0; i<2; i++)
        {for(j=0; j<3; j++)
            {
            *(p+2*j+i) = a[i][j];
            }
        }
}
```

what should be the contents of array b

(a) a b
c d
e f

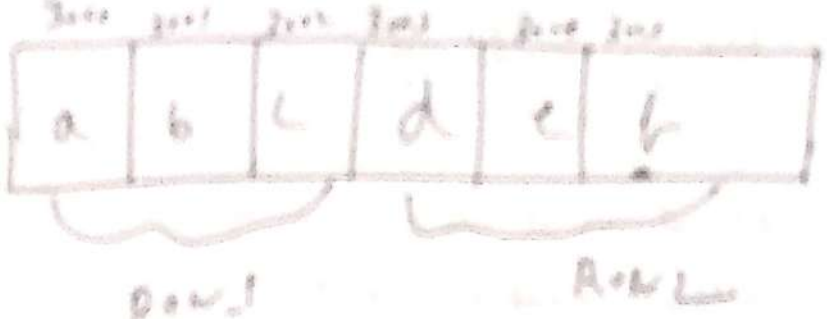
(b) a d
b e
c f

(c) a c
b d
e f

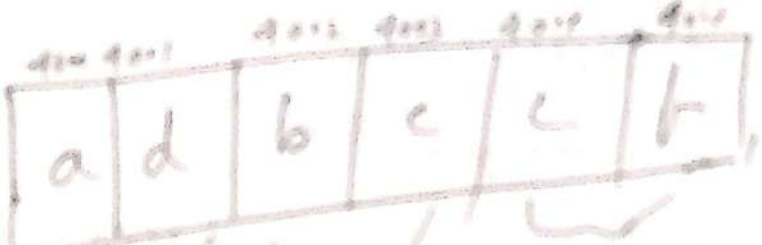
(d) a e
c d
b f

(66)

a



b



→ ① ✓

String handling

String literal is a sequence of characters enclosed within double quotes.

For ex "Hello Everyone"
"Saranghae Anisha"

Placeholders : %s is placeholder for String literal.

```
int main()
{ printf("%s", "Saranghae");
  return 0;
}
// print saranghae on the screen
```

Splicing :

```
printf("%s", "you have to dream before your dream
Can come true --- Apj Abdul kalam");
```

// error as a big string.

Instead we will use

```
printf("%s", "you have to dream before your dream \
Can come true --- Apj Abdul kalam");
```

Because of this no error with the next line → next sentence

In C is called splicing
This method is not preferable.

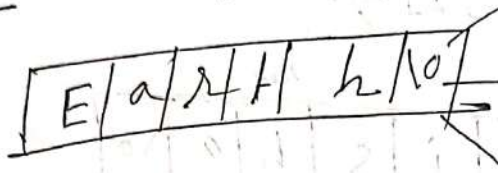
Instead what we prefer is

printf("%s", "you have to dream before your dreams can come true" "-- Apj Abdul Kalam");

Storing String literal:

String literals are stored as an array of character

"Earth"



automatically added

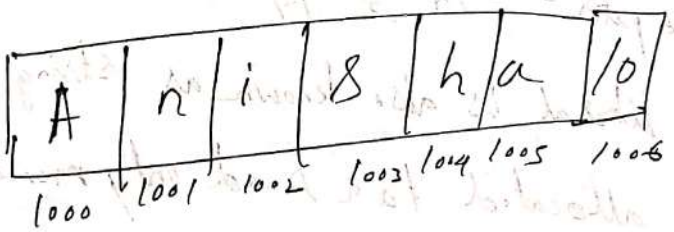
null character indicate end of string

- Total 6 byte read only memory allocated

\0 (null) is not same as 0

① first argument of printf or scanf function is always a string literal.

② Actually we dont pass the whole string instead we are passing a pointer to the first character of string literal.



printf("Anisha") ⟹ passing pointer to letter A

Assigning string literal to a pointer:

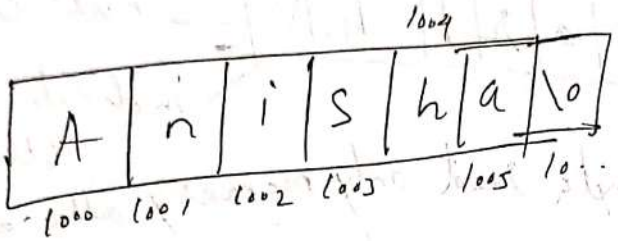
```
char *ptr = "Anisha";
```

ptr points the first character address

"Anisha"[0] → A

"Anisha"[1] → n

"Anisha"[2] → i



pointer to "A"[2] = *(pointer to "A"+2)

= *(1002) = i

(*) String literal can not be modified. It caused undefined behaviour.

```
char *ptr = "Anisha"; //Segmentation fault.
*ptr = 'n'; // not allowed
```

because- String literal is also known as String Constant They have allocated to a read only memory so we cant alter them.

but *ptr is allocated read/write memory

```
char *ptr = "Hello";
ptr = "Anisha";
printf("%s", ptr); // Anisha
```

Difference between string literal & character constant

"H" ≠ 'H'

"H" → | H | \0 |
1000 1001
ptr

'H' → Represented by an Integer (ASCII code - 72)

printf("Arijit"); ✓

printf('H'); ✗

String Variable :

A string variable is a one dimentional array of characters that is capable of holding a string at a time

For - char S[6];

note - Always make the array of 1 character longer than the string.

if we want to store "Anisha" then S[7] as last one will be reserved for null character

Initialize : char S[7] = "Anisha";

S → | A | n | i | s | h | a | \0 |

Although it seems like an string literal but it is not

When a string is assigned to a character array then this character array is treated like other type of array. We can modify it characters

char S[7] = "Anisha"; — Preferred

char S[7] = {'A','n','i','s','h','a','\0'};

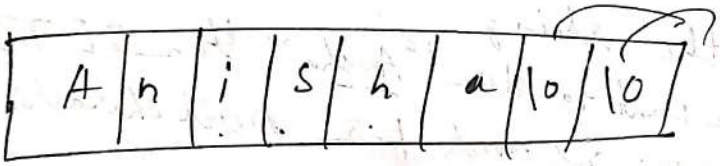
both same

⊛ We cant modify a string literal but we can modify a char array.

✗ char *ptr = "Hello";
*ptr = 'M'; // error Segmentation fault

✓ char S[7] = "Anisha";
S[0] = 'S'; // S[7] will be "Snisha"

(✓) char S[8] = "Anisha"; // Short length initializer
Compiler automatically add null character



(✗) char S[4] = "Anisha"; // Long length initializer

printf("%.5s", S);

O/p — Anis 2

(*) char S[6] = "Anisha";

A	n	i	S	h	a

— no \0 at end

it will print Anisha

(*) but if we want to copy S[6] to d[6] then will give undefined behaviour.

Omiting the length —

char S[] = "Anisha";

Automatically compiler sets aside 7 characters for S that would be enough to store the string "Anisha" with null character.

Writing String :

(1) char *ptr = "Anisha";

printf("%s", ptr);

if we want to print only 3 char then %.3s

printf("%.3s"); // Ani

printf("%.10s") // ---- Anisha

printf("%.3s"); // — also - Anisha (but don't know)

(2) puts() function declared in <stdio.h> library and is used to write string to o/p screen

Also puts automatically write a newline char after writing string

```
char *s = "Anisha";
puts(s);
puts(s);
```

} automatically new line

o/p - Anisha
Anisha

```
char s[7] = "Arijit";
puts(s);
// Arijit
```

Reading String :=>

(1) Using scanf(), we can read a string into string variable or character variable.

```
char a[10];
printf("Enter the string\n");
scanf("%s", a);
printf("%s", a);
```

Dont need & a as is itself a pointer to the first element

=> you are my love (Enter by user)

you (o/p)

The full sentence is not stored as Scanf() does not store Whitespaces char in string var.

It only reads the characters other than White spaces. It store the character array untill Whitespace is encountered.

You are my love.
↑ Whitespace

② Using gets() :

In order to read an entire line of input then we use gets()

```
char a[5];
printf("Enter the string");
gets(a);
puts(a);
```

It may be crash for big line as max capacity set to 5 characters.

Disadvantages : both function have no way to detect when the character array is full, Hence they may cause buffer overflows undefined behaviors occured

Although in scanf we can limit

scanf("%.9s", a) only 9 characters

but gets() have no way.

gets() will try to write the character beyond its allocated memory. It may cause over-write problem.

Getchar() =>

We want our function to be

(i) Reading string even after Whitespace.

(ii) Reading stop after newline character

(iii) must discard extra character

(iv) must return number of characters stored.

```
#include <stdio.h>
int input (char str[], int n)
  { int ch, i=0;
    while ((ch=getchar()) != '\n')
      if (i<n)
        str[i++] = ch;
    str[i] = '\0';
    return i;
  }
int main()
  { char str[100];
    int n = input(str, 5);
    printf("%d %s", n, str);
    return 0;
  }
```

getchar( ) function is used to read one character at a time from the user input.

It returns an integer equivalent to the ASCII code of character.

Putchar( )

int putchar(int ch)

putchar accept integer value (character ASCII) and returns an integer representating the character written on the screen

```
int main()
{
    int ch;
    for (ch='A'; ch<='Z'; ch++)
    {
        putchar(ch);
    }
    return 0;
}
```

O/P- A B C D E . . Z

True / false:

(a) printf("%c", '\n');
✓ %c is placeholder for character

(b) printf("%c", "\n");
%c → char, "\n" → string
✗

(c) putchar('\n');
✓ char

(d) putchar("\n");
string ✗

(e) puts('\n');
~~char~~ char ✗

(f) puts("\n");
✓

(g) printf("%s", '\n');
✗

(h) printf("%s", "\n");
✓

C String library ⇒

<string.h> library contain all the required function for performing string operation.

#include <stdio.h>

(i) **strcpy** : string copy function

Prototype → char* strcpy (char* destination, const char* source);

source is not modified

strcpy is used to copy a string pointed by source (+\0 char) to the destination (char array)

Ex- char s1[10] = "Hello";
char s2[20] = ;

printf ("%s\n", strcpy(str2, s1));

returns the copied string

printf ("%s", str2);

O/P - Hello
Hello

(x) strcpy (str3, strcpy (str2, str1));

✓ possible

str2

⊛ strcpy does not check whether source will fit or not in the destination

if length of destination < length of source then it will give undefined behaviour

To avoid this we call strncpy function

Prototype - strncpy (dest, source, sizeof(dest));

```
char s1[6] = "Hello";
char s2[4];
strncpy(s2, s1, sizeof(s2));
printf(s2);  // Hell
strcpy(s2, s1); // undefined behaviour
```

Strncpy will not give the destination without null character as source

=> destination

str2[sizeof(str2)-1] = '\0';

Manual add

(2) strlen :o=> String length

Prototype — size_t strlen (const char* str);
↑ unsigned int type at least 16 bits

strlen() function is used to determine length of the given string.
We should pass the pointer to the first char of string whose length want to determine

- It doesn't count null char or \n
- char *str = "Anisha";
  printf ("%d", strlen(str)); // 6
  return 0;
- char str[] = "Anisha";
  printf ("%d", strlen(str)); // 6
- char str[100] = "Anisha";
  printf ("%d", strlen(str)); // 6

↳ Why because it calculate length of the string. str has null character except A, n, i, s, h, a.

(3) Strcat( ):

Prototype - char* strcat(char *str1, const char *str2);

Strcat function append the content of str2 at the end of str1 & store in str1

It return pointer to the resulting string str1.

```
char S1[100], S2[200];
strcpy(S1, "Saranghae_");
strcpy(S2, "Anisha");
strcat(S1, S2);
printf("%s", S1);
```

⇒ Saranghae_Anisha

(*) Caution

If size of S1 is not enough to accomodate S2 then it caused Undefined behaviar

Strncat: Strncat is a safer version of Strcat

It appends the limited number of character specified by the third argument

- It automatically add null character at end

```
strncat (s1, s2, int)
                 |
        sizeof(s1) - strlen(s1) - 1
        actually the vacant block
```

```
char str1[5], str2[100];
strcpy (str1, "He");
strcpy (str2, "llo");
strncat (str1, str2, sizeof(str1) - strlen(str1) - 1);
printf ("%s", str1); // Hell
```

Strcmp :=>

Prototype: `int (const char* s1, const char* s2);`

compare string

Returns - Less than 0 $s1 < s2$

Greater than 0 $s1 > s2$

0 if $s1 == s2$

Upper case - 65 - 90 Digit - 48 - 57

Lower case - 97 - 122 Spaces - 32

When the first i characters in S1 & S2 are same and (i+1) character of S1 < of S2

```
char *S1 = "abcd";
char *S2 = "abce";
if (strcmp(S1,S2) < 0)
    printf("S1 is less than S2");
else
    printf("S1 is equal or greater than S2");
return 0;
```

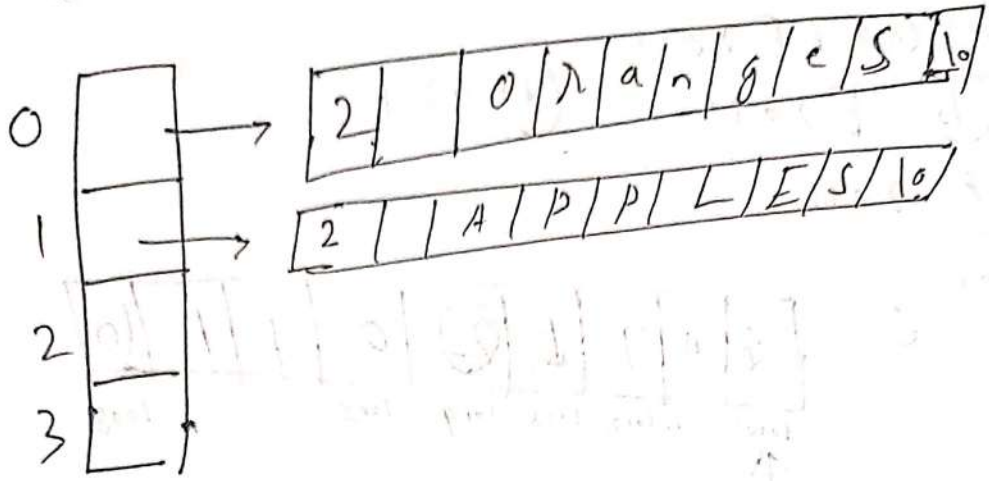
Two dimention array of String:

```
char fruits[][12] = {"2 oranges",
"2 Apples", "3 bananas", "1 pineapple"};
```

A lot of space is wasted.

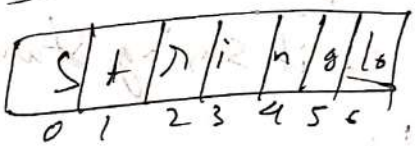
Instead of we use pointer of array

```
char *fruit[] = { "2 oranges", "2 apples",
"3 bananas", "1 pineapple" };
```



(1)
```
char P[20];
char *S = "string";
int length = strlen(S);
int i;
for (i=0; i<length; i++)
    P[i] = S[length-i];
printf("%s", P);
```

(a) gnirts
(b) String
(c) gnirt
(d) No o/p



⇒ So length = 6

P[0] = S[6-0] = S[6]

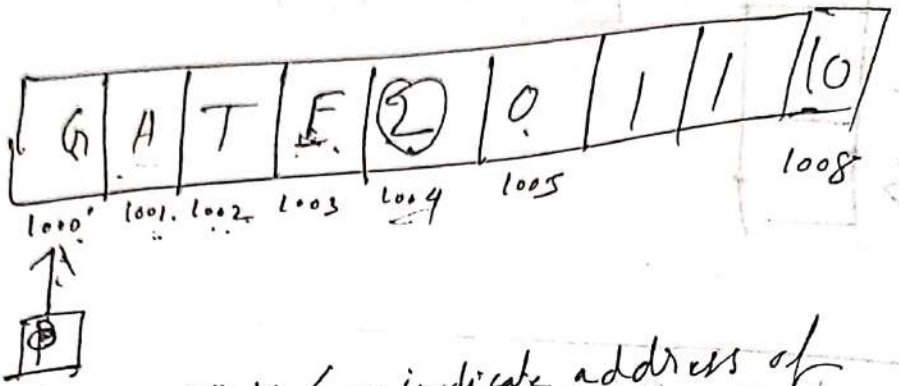
P[0] = \0

(X)(X) We know printf prints untill it gets \0 nullvalue
So no gop will be produced.

② char c[] = "GATE2011";
char *p = c;
printf("%s", p + p[3] - p[1]);

ⓐ GATE2011
ⓑ E2011 ⓓ 021
ⓒ 2011

⇒ c



p → c - indicate address of first char.

*(p+3) Go to index 3 and access the value

p + p[3] - p[1]

= 1000 + E - A — Character Integer only ASCII code. 69 - 64

= 1000 + 4

= 1004.

printf("%s", 1004);

So it saying start printing untill it gets \0.

2011 ✓

(9) The output is if input is given "ABCD_EFGH" is

```
Void foo (char *a)
{
  if (*a && *a != ' ')
  {
    foo(a+1);
    putchar(*a);
  }
}
```

(a) ABCD EFGH
(b) ABCD
(c) HGFE DCBA
(d) DCBA

⇒ a contain the address of A

*a — means value of the address pointed by a
only false when nullcharacter is available

&& — Returns true when both true
else false

*a != ' ' — true when value is space

1st iteration — (A && A != space (true value / true)) — true

```
{ foo(a+1);
  - - -
}
```

now foo is getting address of B

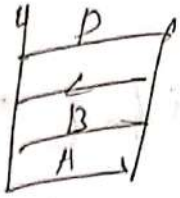
Same happen

if will be false when — space occured

So recursion is complete

now putchar will execute

as in stack we print from last — first



D C B A

(d) ✓

---

(5) Void fun1 (char *s1, char *s2)

```
{ char *tmp;
  tmp = s1;
  s1 = s2;
  s2 = tmp;
}
```

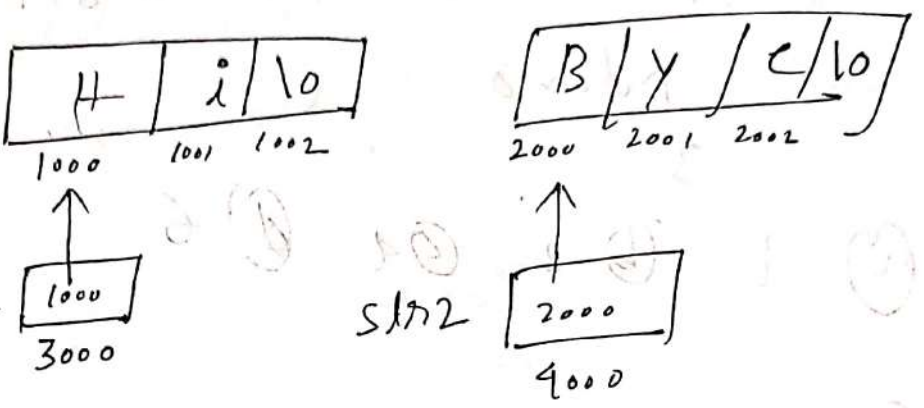
Void fun2 (char **s1, char **s2)

```
{ char *tmp;
  tmp = *s1;
  *s1 = *s2;
  *s2 = tmp;
}
int main ()
{ char *str1 = "Hi", *str2 = "bye";
fun1 (str1, str2);
printf ("%s %s", str1, str2);
```

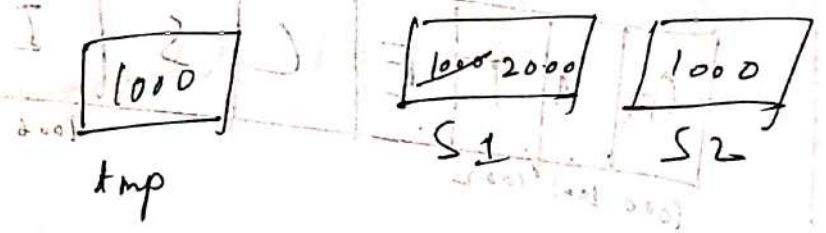
```
fun2 (&str1, &str2);
printf ("%s %s", str1, str2);
return 0;
}
```

(a) Hi bye bye hi (c) Bye hi Hi bye
(b) Hi bye Hi bye (d) Bye hi Bye hi

⇒

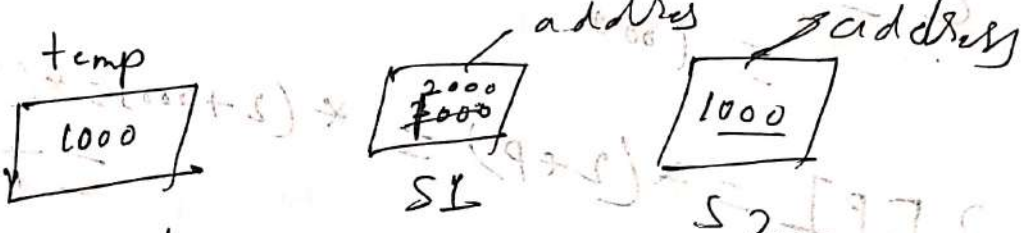


fun1 (1000, 2000) — value



first Printf — Hi bye

2nd printf: fun2 (3000, 4000) — address

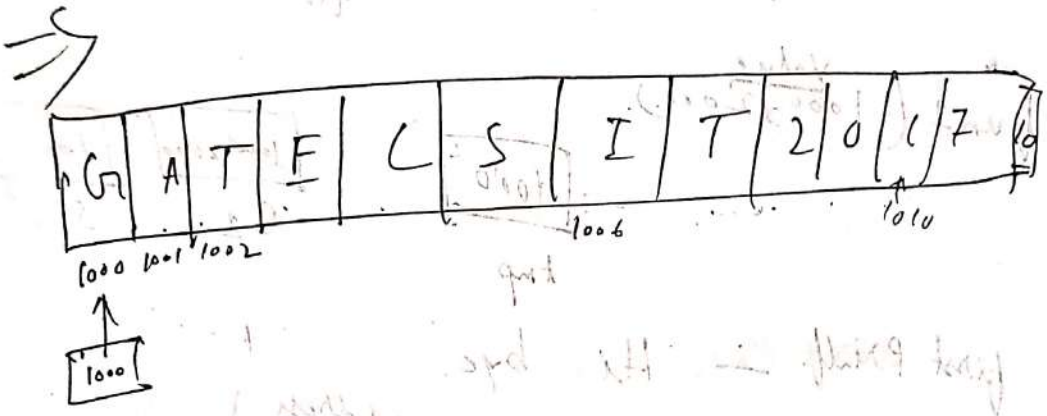


So, string is swapped

→ (a)

(6)
```
int main()
{
    char *c = "GATECSIT2017";
    char *p = c;
    printf("%d", (int)strlen(c+2[p] - 6[p] - 1));
    return 0
}
```

(a) 1 (b) 2 (c) 4 (d) 6



c = c point to address of 1st element
= 1000

2[p] = *(2+p) = *(2+1000) = *(1002)
= T

as p[2] = *(p+2)

6[p] = *(6+p) = *(1006) = I

∴ 1000 + T - I - 1
= 1000 + 11 - 1
= 1010

strlen(1010) → address → strlen starts to count until null char encounter

strlen(str) → it point to address of 1st char

17 10 → 2 char → (2) (6)

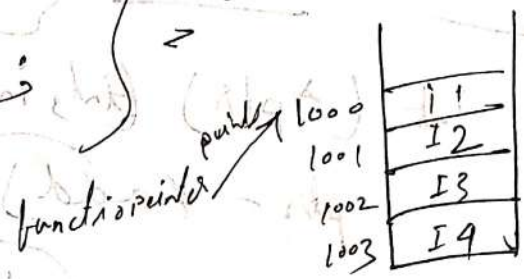
We know strlen return size_t so for explicit type cast we are using (int)

Function Pointer

Function pointer are like special pointer that have the capability to point to a function.

```
int fun(int a, int b)
{ return a+b;
}
```

→ Set of instruction



Declaring pointer to array:

✗ `int *ptr[10];` // wrong as [] is higher precedence than *

`int (*ptr)[10];` // ✓ () is higher than []

ptr is pointer that point array of 10 integers

Declaring pointer to a function:

```
int (* ptr) (int, int);
```

function containing two int arg and return int

Assigning address of a functi, to a function pointer →

```
int add( int a, int b)
{ return a+b;
}
```

```
int main()
{ return type   Argument type
int (*ptr)(int, int)
      = &add;
```

Assigning address

Call the function ←

Same

```
int (*ptr) (int, int);
   ptr = &add;
result = *ptr (10, 20)
printf ("%d", result);
```

add = &add
ptr = *ptr
As function name Point address

11/30

Application of function pointer:

If we want to call a particular function in runtime we use function pointer.

earlier in a calculator program we use switch case but we can use function pointer also

```
#include <stdio.h>
#define ops 2
float sum(float a, float b) { return a+b; }
float sub(float a, float b) { return a-b; }
float mult(float a, float b) { return a*b; }

int main()
{
    float (*ptr[ops])(float, float)
            = { sum, sub, mult };
    int choice
    float a, b;
    printf("enter choice\n");
    scanf("%d", &choice);
    printf("enter 2 number");
    scanf("%f %f", &a, &b);
    printf("%f", ptr[choice](a,b));
    return 0;
}
```

Structure =>

Suppose you have a garage & want to store all information about car which are available in garage

Car 1 Car 2 . . . Car n

Engine — DD 150 Engine — DD180
Fuel — Petrol Fuel — Diesel
Fuel Capacity — 37 Fuel capacity — 32
Seating — 5 Seating — 6

So in order to store all this we have to creat diff variable. Suppose you have more than 100 car then time & space will be wasted

car1
char *car1Engine = "DD 150";
char *car1fuel = "Diesel";
int fuelcapacity = 32;

car2
char *car2Engine = "DD 180";

This . . . . is so time consuming

(*) We also can't use array as it is not same type.

There comes struct to rescue.

A structure is a user defined data type that can be used to group elements of different types into single one.

Declaring :-

```
struct
{
    char *engine;
    char *fuel_type;
    int  *tank_capacity;
    int   seat_capacity;
} car1, car2;
```

global scope — Every function can access it

Different datatype declaration

car1, car2 → object/variable

Ex-

```
struct
{
    char *engine;
} car1, car2;

int main()
{
    car1.engine = "DD150";
    car2.engine = "DD180";
    printf("%s\n", car1.engine);
    printf("%s\n", car2.engine);
}
```

With the help of . operator we can access member of structure

// DD150
DD180

Structure tag :

Structure tag is used to identify a particular kind of structure.

```
Struct structurename
{
    Different data type
    declarati..
};
```

Later we can create Variable

```
struct structurename manager;
struct structure_name emp2;
```

Using typedef - gives freedom to create own type / renaming type.

```
typedef existing_data_type new_data_type;
```

For ex -

```
typedef int Arijit;
int main()
{
    Arijit var=100;
    printf("%d", var);
}
```

Instead of int we can use Arijit

//100

- In Structure also we can do it

typedef struct car { → add car name.

} car; → new type name

struct car c1; ✓

↓ same

Car c1; ✓

Initialization & accessing Structure members:

```
Struct ab
{ int p = 23;
  int q = 24;
};
```

✗ Wrong Way

```
Struct ab
{ int p;
  int q;
};
```

```
int main()
{
    struct ab x = {23, 24};
}
```

✓

We can access member of structure with

- operator

X.p

X.q

Designated initialization:

Designated initialization allows structure members to be initialized in any order

```
struct abc
{
    int x;
    int y;
    int z;
};
```

int x,y,z; — Same

```
int main()
{
    struct abc a={10,20,30};
```

Same order

```
int main()
{
    struct abc a={.z=30,.y=20,.x=10};
```

Designated initialized

• varname = value

Array of Structure:

Instead of declaring multiple variable we can also declare an array of structure in which each element of array will represent a struct variable

```
struct structure_name variable[2];
```

variable[0], variable[1] = same

Accessing member of Structure pointer

```
Struct abc
{
  int x;
  int y;
};
```

ptr is a pointer to some variable of type struct abc

ptr contain address of a

```
int main()
{
  Struct abc a = {0,1};
  struct abc *ptr = &a;
  printf("%d %d", ptr->x,
          ptr->y);
}
```

0 1

ptr -> x is equivalent to (*ptr).x

= a.x

Structure Padding

When an object of some structure type is declared then some contiguous block of memory will be allocated to structure members

~~the sizeof Struct -> Size~~

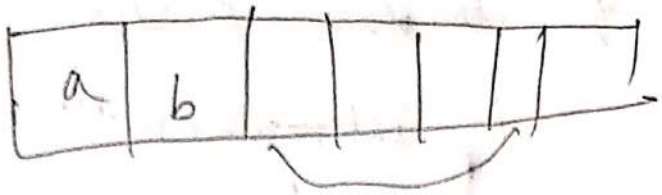
```
Struct abc{
  char a; // 1 byte
  char b; // 1 byte
  int c; // 4 byte
};
```

(*) Processor doesn't read 1 byte at a time it read 1 word at a time

If we have 32 bit Processor then it means it can access 4 bytes at a time which means word size is 4 byte

If we have 64 bit " " " " " "

8 bytes at a time " " " " is 8 byte

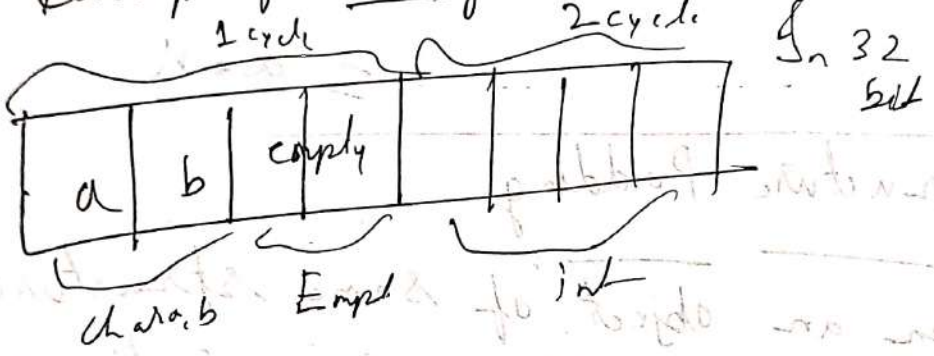


In 32 bit architecture char a, char b, 2 bytes of in can be accessed in 1 CPU cycle

So 2 cycle is required to access all. (char, char, int)

Is an Unnecessary wastage of CPU cycle

So Concept of Padding came into picture

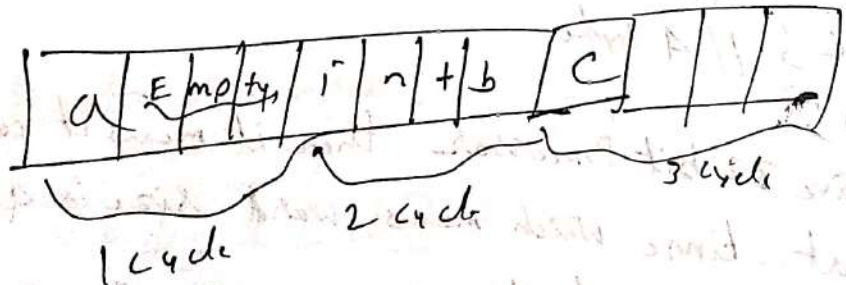


Total no of bytes = 2+4+2(empty)
= 8 byte (In 32)

If we change order of variable.

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

total = 12 byte



2021

Structure Packing =>

Because of structure paddings, size of the structure becomes more than the size of the actual structure. Due to this memory is wasted

```
struct abc
{
  char a;
  int b;
  char c;
}var;
int main()
{
  printf("%d", sizeof(var));
}
```

op is - 12 as structure padding occurred

but it should be 6

There come concept of packing

We can avaid the wastage of memory by simply writing #pragma pack(1)

Special purpose directive used on or off certain feature.

If we write #pragma pack(1)

then o/p will 6 no padding will be applied

but CPU cycle is wasted

In Packing Memory is wasted

Problem

(1)
```
#include <stdio.h>
struct point
{int x, y, z;
};
int main()
{
struct point p1={.y=0,.z=1,.x=2};
printf("%d,%d,%d", p1.x, p1.y, p1.z);
return 0
}
```

(a) 2 0 1
(b) compile error
(c) 0 1 2
(d) 2 1 0

⇒ Here we are using designated initializer.

So answer 2 0 1 (a) ✓

---

(2)
```
#include <stdio.h>
struct ournode{
    char x, y, z;
};
int main()
{
struct ournode p={'1','0','a'+2};
struct ournode *q=&p;
printf("%c,%c", *((char*)q+1), *((char*)q+2));
}
```

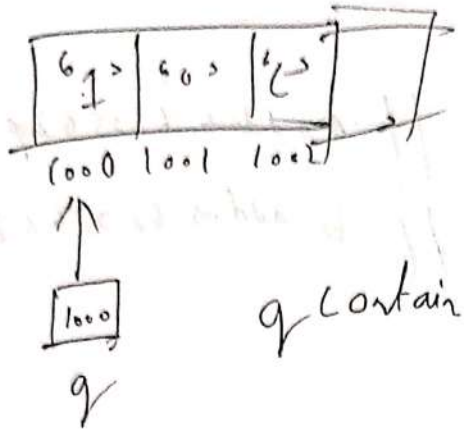
o/p — (a) 0, c
(b) 0, a+2
(c) '0', 'a+2'
(d) '0', 'c'

'a' + 2 → C

%c means Printing char 1 not '1'

x = '1'

y = '0'



a byte allocated

q contain address of whole structure

*((char *) q+1) — Particular character

q+1 — *(1001) → '0'

— *(1002) → 'C'

0, C → (a) ✓

---

(3) Struct node
{ int i;
float f;
};
struct node *s[10];

define s to be

(a) an array, each element is pointer to structure of type node

(b) A structure of 2 field

(c) A structure of 3 field

(d) Array, each element is a structure of type node

⇒ s is not a structure
b, c ✗

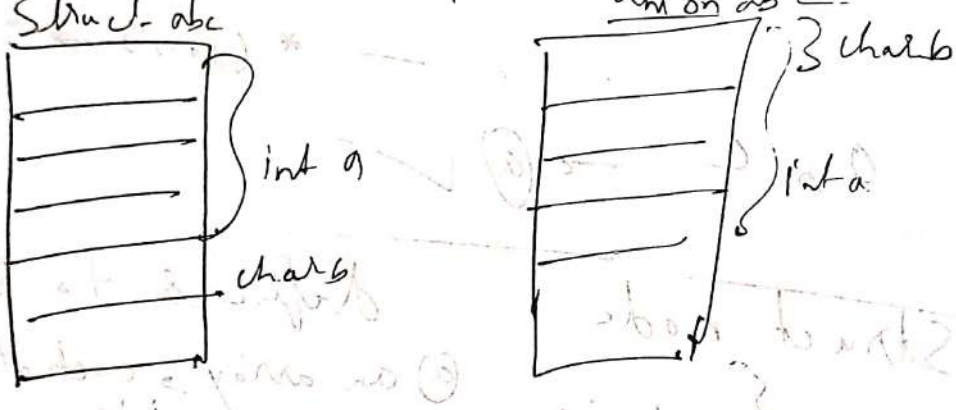
each element not a structure d ✗

f is a Pointer ✓

(a) ✓

Unions :- ⇒ Union is a user defined data type but unlike structures, union member share same memory location

```
Struct abc
    { int a;   // address 6205629
      char b;  // b address 6205628
    };
Union abc
    { int a;   // both address
      char b;  //  6205616
    };
```



(*) Union member share same memory location. If we change it affect others

```
union abc
    { int a;
      char b;
    } var;

int main()
    { var.a = 65;
      printf("a = %d \n b = %c", var.a, var.b);
    }
```

a = 65

b = A

(*) Size of union is taken according to the size of largest member of the union.

```
Union abc
{
  int a;     // 4 byte
  char b;    // 1 byt.
  double c;  // 8 byte
  float d;   // 4 byt.
};
```

Sizeof (union abc) = 8 byte (highest)

(*) We can also access member of union through pointers using arrow

```
Var. a = 90;
union abc *p = &var;
printf ("%d %d %c", p->a, p->b);
```

90 Z

(1)

```
struct {
  short s[5];
  union {
    float y;
    long z;
  } u;
} t;
```

Assume
short — 2
long — 8
float — 4 byte

The memory required for var t ignoring allignment consideration

(a) 22 (b) 14 (c) 18 (d) 10

short ex 5 array — 10 byte

union → longest type is long — 8 byte

→ (L) 18 byte

Application of Union ⇒

A store sell 2 item

(a) books (b) shirt.

property	Color
Title	Size
Author	design
number of Pages	price
price	

So they made a structure

```
Struct store
{
    double price; // 8
common
    char *title; // 8
book  char *auther; // 8
      int num_Pages; // 4
      int color; // 4
shirt int size; // 4
      char *design; // 8
};
Struct store book; // — 44 byte memory
```

There is so much wastage of memory like we dont want color, size, design in book

So what we do

```
#pragma pack(1) // padding off with packing
struct store
{
    double price;
    union
    {
        struct
        {
            char *title;
            char *author;
            int num_pages;
        } book;
        struct
        {
            char *color;
            int size;
            char *design;
        } shirt;
    } item;
};
int main()
{
    structure store S;
    S.item.book.title = "The Alchemist";
```

Creating an array with mixed type data

```
typedef union{
    int a;
    char b;
    double c;
} data;

int main()
{
    data arr[10];
    arr[0].a = 10;
    arr[1].b = 'a';
    arr[2].c = 10.178;
    .
    .
    .
}
```

If we use struct arr it will consume more memory as we know at a time in a array element there could be only one value.

Enumeration =>

It is a user defined type which is used to assign names to integral constant because name are easier to handle

* We can also do it by ~~typedef~~ #define but it is in global scope but enum can be used locally also. and enum automatically initialized

// enum bool { False, true }
0 1

```
int main()
{
    enum bool var;
    var = true;
    printf("%d", var); // 1
    return
}
```

(1) If we do not assign value to enum names then compile will assign value starting to 0 and incremented 1 gradually.

enum a { a, b, c, d }
0 1 2 3

(2) Also if enum a { a=3, b=2, c, d, e=0 }
3 4

c value will be b value + 1

(3) enum a { a=0, b=0, c=0 }; ✓

enum a { a=0, b=2.5, c=4 } ✗ only integer
no float value

(X) All enum constant must be unique in their scope

enum a {x=2, y=3}

enum b {x=3; y=3} → X

redefination not allowed

---

(1) Struct

```
{
  double a;
  union
  {
    char b[4];
    double c;
    int d;
  } e;
  char f[4];
} s;
```

how much space allocate for s (no padding)

⇒ double – 8

char – 1×4=4 } – 20

Union – double – 8

Preprocessor Directive =>

Preprocess is a program that process the source code before it passes through the compiler

Sourcecode → Preprocess → Expanded Sourcode

(*) It operate under the control of preprocess directives which began with the symbol #.

types —
- Macro substitution
- file inclusion &
- Compiler control

(i) Macro substitution

It replaces every occurance of identifier by a predefined string

#define identifier string

Ex - #define PI 3.1415

Whether it will encounter PI it just replace PI to 3.1415

(ii) File inclusion: External file containing function & macro defination

#include <stdio.h>

(iii) Compiler Control :

C preprocessors offer a feature known as conditional which can be used to switch on or off of particular line

#if #else #endif

Ex- #include <stdio.h>
#define LINE 1

int main()
{
#if

Suppose in a big program we does not want to compile or run a segment of program what we do we commented those lines if there is a lot of line it takes a lot of time to comment or Delete the comment.

```
#define VER1
int main()
{
    #ifdef VER1
    printf("This is demo");
    #endif
}
```

If VER1 is defined then it will not compile if not define it will compile.

Command line argument ⇒ It is

possible to pass some value from the command line to your C Program when they are executed. These values are called Command line argument

We have to pass 2 Parameter pointer to array of string

```
int main (int argc, char *argv[])
```

argc – arg counter
argv – arg value

argv[0] is path of exe file

So if no command line argument given then argc = 1 (as path is an argument)

File handling in C ⇒

We need to create file pointer

Prototype – FILE * <identity>

ex – FILE* ptr;

fopen() – used to open a file in different modes (Read, write, Append)

Prototype – fopen ("file path", "mode");

- On success it opens a file return address of file
- If it is failed it returns null pointer
- Return type is file*

(*) All file related function belongs <stdio.h>

```
main()
{
    FILE* fp;
```


```
fp = fopen("D://Test//test.txt", "r");
if(fp == NULL)
    printf("file not present");
else
    printf("file open Read point");
```

Reading char from file -

fgetc() takes one argument

fgetc(FILE* stream);

On success read return Ascii value of char

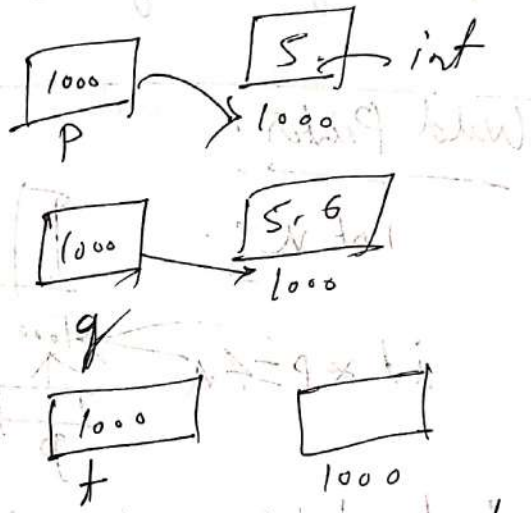
if fail it return → -1 (eof) end of file

```
FILE* fp = fopen("g://textfile", "r");
int ch;
while ((ch = fgetc(fp)) != EOF)
{
    printf("%c", ch);
}
```

(X) FILE is a ~~~~ datatype — Structure

Void Pointer :=>

```
int *p
float *q
```



Same `void *t` —

Generic pointer it can point to the any type value address

```
int x
fp = &x
```

```
float y
t = &y
```

we can directly write *t to represent value of *fp

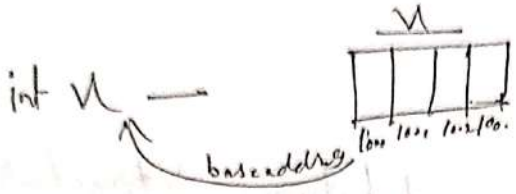
we have to typecast

```
(int*) *t
(float*) *t
```

Null Pointer :=>

Macro (already defined)

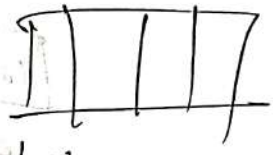
```
int *p=NULL;
```



Such a location it points as if it is pointing nothing (empty location Point)

Wild Pointer :

```
int x;
```



```
int *p=&x
```

but if we only write

illegal memory error

```
int *p;
*p=q;
```

garbage address

Wild pointer — Uninitialized Pointer (Garbage location Point)

Dangling Pointer : A pointer pointing to a memory that has been deleted (freed) is called dangling pointer

```
int *fun()                    int main()
    { int n = 5;                  { int *p = fun();
      return &n;                  printf("%d", *p);
    }                             }
```

A garbage address

if n is static then p does not become dangling

goes out of block :

```
void main
    { int *ptr;
      = :
        { int ch;
          ptr = &ch;
        }
      // Here ptr is dangling pointer
    }
```