

(1)

## C Programming (Nesco Academy)

Why C

⇒ Assembly portable X

C is portable ✓ good for portability

② It takes less line of programs. ↓↓↓↓↓

in assembly                      in C with lib  
 (it has goto) but no goto - result = 2 + 3  
 mov a, 2  
 add a, b  
 mov result, a

Assembly code -  
 mov b, 3  
 add a, b  
 mov result, a

Features of C

i) Procedural programming - divides into small

functions : subroutines

ii) Middle level language -

difficult for robots to learn

High

Low

Abstraction degree : internal details

c++, lisp

Assembly language

if does make simpler

a) Direct access to memory through pointers

b) Bit manipulation

Writing assembly code within C code.

III Popular choice for system level apps -

Device driver, editor, games

(iv) Wide variety of built-in functions, standard libraries & header file.  
stdio.h, <conio.h>, printf

(v)

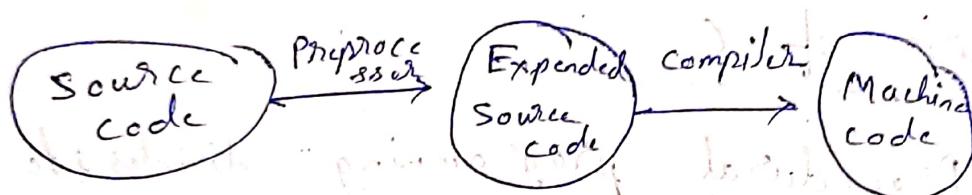
To comment in any program we use //

#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

(3)

## Syntax of function $\Rightarrow$

Return-type name-of-function (Parameter-type name of parameter, Parameter-type name-of-parameter, ...)

{ set of statements }

3. It is used to map function.

④ Parameters are input to that function.

Header files  $\Rightarrow$  contains prototype of a function

• Declaration like - `printf` etc. add to stdio.h

Standard library  $\Rightarrow$  Actual definitions of functions

i.e. libm, libstdc++ etc.

Linker  $\Rightarrow$  it maps between declaration of function & actual definition.

Why so  $\Rightarrow$  Header files consists of only declarations of function that tells your computer which function you going to use. Preprocessor combine its & produce expanded source code. Linker maps the declaration with actual definition after completion of compilation.

order principle  $\Rightarrow$  what does the work of

④ If preprocessor is ~~not~~ there it will take huge time to compile.

linker if visiting card is header file then

④ If visiting card is stl.

$\Leftrightarrow$  the person you met is stl.

Variables → class = variable holder for value  
water = value

⊗ you have to declare the variable before use.

Declaration → Announcing property of the Variable to the compiler.

Properties → size of the Variable

→ name of the variable.

Definition → allocating memory to a variable.

⊗ Most of the time declaration & definition will be done at the same time to differentiate one statement from another.

Writing into Var →  
name of the variable is broken  
into two parts.  
1. Data type which much space a variable is  
going to occupy, going to give

Initialization → Assigning Value of Variable at the

time of declaration is called initialization.

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

    printf("%d %d", var2); // variable to another.

(5)

Naming conventions of Variables  $\rightarrow$  Composed of letters & digits

1) You can't start the variable name with digit.

Var ✓ var ✓ iVar ✗

2) Beginning with underscore(\_) is valid but not recommended for styling.

\_ 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, VAS, VaR are all different.

\* usually lowercase are used to name variables.

④ \* uppercase are used for symbolic constants.

4) Most of the special characters (@, #, %, /, \*, ^) not allowed.

However, though (-) allowed.

Using them to name variables not allowed.

5) Blanks or White spaces not allowed.

int my variable X

int my-variable ✓ ( ) is your Variable

6) Don't use Keywords to name your variables.

Reserved words

like - if, else, switch etc.

7) Don't use long names.

shorter names are better.

## Surprise test

Fill in the valid name of the Variable.

int   ;

- a) int b) @go c) 362 d) Switch e) While  
 (switch keywords are lower case)

Printf :  $\Rightarrow$  Printf is a function which prints

output on the screen, such string constant

$\Rightarrow$  printf ("Anujit");

O/P  $\rightarrow$  Anujit.  $\Rightarrow$  placeholder

$\Rightarrow$  printf ("%f", Var);  $\Rightarrow$  f means decimal

$\Rightarrow$  %f is a placeholder of Var  
 O/P - 3  $\rightarrow$  it prints the value of Var

$\Rightarrow$  printf ("%d %d", Var1, Var2);

O/P - 3 2

$\Rightarrow$  For each %d you have to provide variable name. if you don't provide, it will give error.

#include <stdio.h>

int main()

{

int a=2, b=3;

int n = a+b;

printf ("I am adding %d + %d, result is %d", a, b);

printf ("The result is %d", n);

O/P - I am adding 2+3

The result is 5

(7)

## 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 programmatically →

sizeof is an unary operator & not a function

#include <stdio.h>

int main()

{ printf("%d", sizeof(int)); } → 4 bytes

← 32 bits for proof

Range → strictly lower limit of set of data strictly upper limit of set of data

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

(L - U) Range → 0 - 4

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

Range → 0 → 9 = 10 bytes → 5.32 bytes

$$= 5 \times 10^0 + 3 \times 10^1 + 2 \times 10^2$$

But machine cannot understand the number system.

Binary number system → Machine understandable number system. Also called base 2 number system.

Range → 0 → 1

$$\begin{array}{r} 2^3 \ 2^2 \ 2^1 \ 2^0 \\ \hline 1 \ 1 \ 0 \ 0 \end{array} = 2^3 \times 1 + 2^2 \times 1 - 2^1 \times 0 + 2^0 \times 0 = 8 + 4 - 0 + 0 = 12$$

Range of 9 bit data

min value = 0

$$\begin{array}{r} 2^3 \ 2^2 \ 2^1 \ 2^0 \\ \hline 0 \ 0 \ 0 \ 0 \end{array}$$

max value = 15

$$\begin{array}{r} 2^3 \ 2^2 \ 2^1 \ 2^0 \\ \hline 1 \ 1 \ 1 \ 1 \end{array}$$

$$0000 \rightarrow 1111 \text{ with } n=9$$

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

$$\therefore \text{max} = 2^9 - 1$$

$$= 15$$

Range of integers  $\Rightarrow$

1) 2 bytes  $\Rightarrow$  2 bytes = 16 bits

= Unsigned range 0 to 65535

$$P = 0 \rightarrow 2^{16} - 1 \quad (2^16 - 1)$$

Signed range  $\pm 32768$  to  $32767$

$$\text{Range} = 2^{15} \text{ or } 32,768 \quad (-(2^{15}-1) \text{ to } (2^{15}-1))$$

2) 4 bytes  $\Rightarrow$  4 bytes = 32 bits

Unsigned range 0 to 4294967295

$$\text{Signed range } -2147483648 \text{ to } 2147483647$$

3

Modifiers (Long, short, signed, unsigned)  $\Rightarrow$

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

(\*)  $\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$

Note  $\Rightarrow$  By default int var-name is signed integer variable.

Unsigned int var-name allows only positive values.

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?  $\Rightarrow$

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

```
int main ()
```

```
unsigned int var = 4294967295;
printf ("%d", var);
```

O/P - 4229267295

But if you exceed the range in

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

{ unsigned int var = 4294967296;

printf( " %c %d %s %f ", Var );

$$O(1^P) = \overline{O}$$

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

3 bit unsigned 4 bits 1111 1010 0101 0011 3 bits

range < logistic growth - C exceeding

$$\begin{array}{r} 0 \quad 0 \quad 1 \quad - \quad 1 \\ 0 \quad 0 \quad 1 \quad 0 \quad = \quad ? \end{array}$$

~~top left~~ 0 1 1 — 2  
1 0 0 — 3  
1

$$\begin{array}{r} 1 \quad 0 \quad 0 \\ 1 \quad 0 \quad 1 \end{array} \begin{array}{l} \text{---} 9 \\ \text{---} 5 \end{array}$$

1 10 — 6

$$\underline{1} \quad \underline{2} \quad \underline{3} - 7$$

$$8 \rightarrow \underline{1} \overline{0 \ 0 \ 0} \rightarrow 0$$

$$9 \rightarrow 1 \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} - 1$$

$$10 \rightarrow \overline{1} \overline{0} \rightarrow$$



(12)

Extended ASCII  $\Rightarrow$  To utilize most significant bit & to represent other language symbols, 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) In extended it requires 8 bit.

### Fundamental datatypes $\Rightarrow$

Float, double, Long Misleading to represent fractional numbers. IEEE 754 standard.

Float  $\rightarrow$  4 bytes - float  $\times$  size of these data

Double - 8 bytes - float type is totally depends from system.

Long - 12 bytes - long

(\*) Float can represent 7 digits.

(\*) Double in 16 digits

(\*) Long Double in 19 digits

(13)

Question

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

→ Before giving answer to this question you have to know that ~~it is used to print~~ string of character.

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

So first of all if Hello World! is printed then it will return 12 as total number of characters. If it will return 8 as total number of characters then it is incomplete so Ans - Hello world! 12 = 12 characters.

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

⇒ .... Hello (Ans)

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

(3)

#include <stdio.h>

int main ()

options

char c = 255;

or 265

c = c + 10;

b) Some char

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

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 ( $2^8 - 1$ )

• 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.

$$\text{• } \frac{265}{2^n} = \frac{265}{2^8} = \frac{265}{256}$$

(1) sign bit  
(2) flags  
(3) value

it will print 9

(4) Which of the following statement is ~~15~~  
correct corresponding to the definition of integer.

- (i) signed int i;      a) Only I & V are correct
- (ii) signed I;      b) Only I is correct
- (iii) unsigned I;      ~~c) All are correct~~
- (iv) long i;      d) Only IV, V, VI are correct
- (v) long int i;
- (vi) long long i;

~~→ If 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 = - 9 ; // implicitly signed negative

printf ("%u", i+j);

getchar();

as garbage

b) -3

~~c) integer value depends machine~~

d) None

→ (A) unsigned means it will only take positive value.

(B) %u is the placeholder of unsigned integer.

(C)  $i+j = (+E4) - 3$  if there was %d then output will be -3 but as here is %u we cannot represent -3 in negative format

So in order to represent -3 in 2's complement

-3 in binary

00000000    00000000    00000000    00000011

1st Complement

11111111    11111111    11111111    11111110

2nd complement → 1st complement + 1

11111111    11111111    11111111    11111101

- 999967293

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 variable 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 func(); // global Variable ; }

int var = 10; // global

int main() {

    int var = 3; // local to main

    printf("%d", var);

    return 0; } // main function

3. Global value of variable.

int fun()

{ printf("%d", var); }

3

0 11 = 3

10

variable declared A is available

variable declared var is available

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

int main()

void b() { int var = 3;

void c() { int var = 4; }

printf("%d", var); } // error because both

variables have same name

points to return value of A

→ error: redefinition of var in the function

{ definition of outer block upto }

--- content of outer block upto ---  
this point are visible to inner block

content of internal block

are not visible to outer block

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

# Variable Modifier

(15)

There are ~~two~~ 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;  
    float var;  
}

⇒ int main()  
{  
    auto int var;  
    auto float var;  
    printf("%d %f", var);  
}

benefits - (a) it don't waste its memory.

(b) it is destroyed after execution.

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

On the other hand global variable by

(\*) 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;  $\Rightarrow$  extern int var;

Declaration

definition - allocating  
memory

\* extern is shorthand 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\n", a); }

getchar();

}

Output - 200

other.c

int a = 20;

extern int a;

int b;

Compilat. error  
we can't say extern  
as it is initialized.

\* When we write extern no memory allocation  
only property is announced.

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

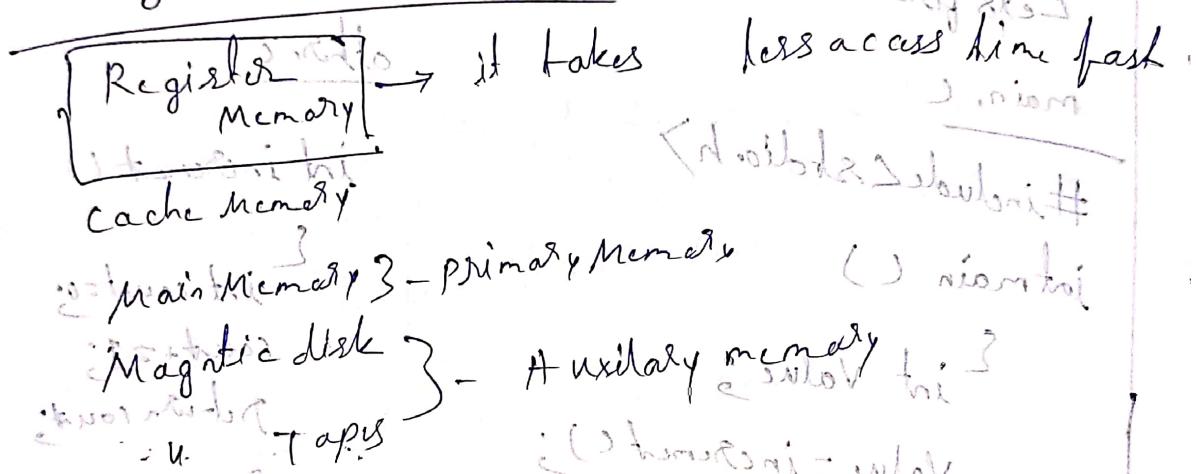
(21)

(X) certain Variables says to compiler to go outside from my scope and you will find the definition of the variable is declared.

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

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

## Register Modifier



Syntax → register data-type var-name;

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

{ register int var; }

3 advantages of using register  
1. fast access to memory  
2. smaller size of program  
3. less memory usage

But this is type of higher level language

## 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 $\Rightarrow$ Updated Value

Let's first understand a program.

main.c

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

```
int main ()
```

```
{ int
```

```
Value = 10;
```

```
Value = increment();
```

```
Value = increment();
```

```
Value = increment();
```

```
printf ("%d", Value);
```

```
return 0;
```

3

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

But you might forget a print!!

(23)

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 I get our desired output is 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 completion 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  $\Rightarrow$  Once defined & cannot be modified in the code.

Defining  $\Rightarrow$  a) using #define keyword.

Syntax #define  $\Rightarrow$  #define Name value  
Macro

Job of preprocessor (not compiler) To  
replace Name with Value

Ex - `#include <stdio.h>`

```
#define PI 3.14159  
NAME VALUE  
int main()  
{  
    printf("%f", PI);  
}
```

(\*) Please don't add Semicolon add the  
end of #define.

#define PI 3.14159 ;

(\*) Always Choose in Capital Letters  
for NAME  
because if there any Variable with Mass  
Name it will give compilation error.

`#include <stdio.h>`

`#define value 89`

main

int ~~value~~( )

{

int value=79;

printf("%d", value);

printf("%d", value);

return;

`#include <stdio.h>`

`#define value 89`

int main()

{

int 89=79;

printf("%d", value);

return;

(\*) Whatever inside double quotes <sup>(25)</sup> won't  
get replaced

(\*) We can use macros like functions

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

3

o/p - 7

(\*) We can write multiple lines using \

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

```
#define greater(x,y) q if(x>y)\n        printf("%d is greater than %d\n", x,y)\n    else\n        printf("%d is less than %d\n", x,y)
```

```
int main()
```

```
{\n    greater(6,5);}\n    printf("function output")\n}
```

3

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\n", add(2,3)); }

return 0;

}

$$\text{Ans} \rightarrow 5 * 4 + 3 = 23$$

= 23 digitum error nos 31

Ansible submitted

\* 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

(33) (27)

Constants using const  $\Rightarrow$

Syntax  $\rightarrow$  const datatype varname;

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

(X) 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);  
}
```

We can't change value through pointer though with warning message.

Output: But an error message is shown as  
"ifc3" is aligned to 0x0000000000000000

①

int main () {  
 int var = 052; // 56  
 printf ("%d", var); // 42  
 return 0;  
}

Output: 42

if var = 052, then  
printf ("%d", var); // 42  
return 0;

As there are 0 before the number it is treated as octal value  
 $052 = 5 \times 8^1 + 2 \times 8^0 = 42$

Answer will be 42

(\*) If there was 1.0  $\text{Mg}^{2+}$  then output would be

2

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

```
#define String "Hello, I am a string."
```

```
#define Nescio "Welcome!"
```

int main ( )

{  
} {  
} {  
}

Print (String, NsCo); < Garbage

3

→ As we know preprocessor replace the macros before compilation. So after preprocessing the programme will be converted.

plaintiff (§ 1.81n<sup>2</sup>) b<sup>2</sup> (Welcome, b.1.2) this

O/P will be welcome.

Scarf  $\Rightarrow$

(\*) Scarf stands for Scan formatted string.

(\*\*) It accepts characters, string and numeric data from the user using standard input-keyboard.

(\*) Scarf also uses format specification like printf -

• %d for integer

• %c for character

• %s for string

assigning the value.

(\*) int var;  
Scarf ( &var, 4Var ); - 918

Why  $\&$ ?  $\Rightarrow$  ~~down below~~ b.c  
as while scanning the input scarf needs

to store the data

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

(\*) & is also called as address-of operator

4Var  $\rightarrow$  Address of Var  
( &Var ) bsp

Example →

30

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

```
int main( )
```

```
{ int a, b; // Enter first number.  
printf("Enter first number.");  
scanf("%d %d", &a);
```

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

`scanf(" %c", &ch);`

```
printf ("addition of %d + %d is %d", a+b);
```

3 return of quite a few

DIP - Enter first number

3 4

Enter second number: 5.0

4 21 fibrous & sticky

~~address - 3~~

the kind of lobes in fl

(1) #include <stdio.h>

(X) Oxa, OX<sub>mean</sub>

The real decimal numbers

```
int main ()
```

int var = 0x43FF;

print ("%.2f", var);

return on

01P → 43FF

## Types of operator $\Rightarrow$

### 1) Arithmetic operator $\Rightarrow$

$+$   $\rightarrow$  addition,  
 $-$   $\rightarrow$  subtraction

$*$   $\rightarrow$  multiply  
 $/$   $\rightarrow$  Divide  
 $:$   $\rightarrow$  Modulous

### 2) Increment / Decrement $\Rightarrow$

$++ \rightarrow$  increment

$-- \rightarrow$  Decrement

### 3) Relational operator $\Rightarrow$

$== \rightarrow$  equal to  $!= \rightarrow$  not equal to

$\leq \rightarrow$  less than equal to  $\geq \rightarrow$  greater than equal to

$< \rightarrow$  less than  $> \rightarrow$  greater than

### ④ Logical operator $\Rightarrow$

~~And~~  $\wedge$   $\rightarrow$  And

~~Or~~  $\vee$   $\rightarrow$  Or

### ⑤ Bit wise $\rightarrow$

$\&$   $\rightarrow$  bit wise and

$\sim$   $\rightarrow$  bit wise not

$\vee$   $\rightarrow$  or

$\sim$   $\rightarrow$  complement

$\gg$   $\rightarrow$  right shift

$\ll$   $\rightarrow$  left shift

## ⑥ Assignment $\Rightarrow$

- =  $\rightarrow$  assignment
- + =  $\rightarrow$  increment then assign
- =  $\rightarrow$  decrement then assign
- \* =  $\rightarrow$  multiply then assign
- / =  $\rightarrow$  division then assign
- % =  $\rightarrow$  modulus then assign

## ⑦ Others $\Rightarrow$

Conditional  $\rightarrow ? :$

Address of  $\rightarrow \&$

start  $\rightarrow *$

size off  $\downarrow$

9

## ⑧ Arithmetic operators $\Rightarrow$

All are binary operator means two operands need to perform operation - II

## ⑨ Precedence & Associativity $\Rightarrow$

Precedence operators



Highest  $\rightarrow *, /, \%$



lowest  $\rightarrow +, -$

Associativity

$\left. \begin{array}{l} \text{left to right} \\ \text{left to right} \end{array} \right\}$

$\left. \begin{array}{l} \text{left to right} \\ \text{left to right} \end{array} \right\}$

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

For example -  $a + b + c$ , here +, +, + are same precedence therefore we use associativity means left most sign will be evaluated first.

Suppose  $a = 2, b = 3, c = 9$   
then  $a * b / c$  - here \* & / are of same precedence, so leftmost sign will be performed first then division will be performed.

Increment operator  $\Rightarrow$  It is used to increment the value of a variable by one.

Decrement operator  $\Rightarrow$  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  $a = a - 1;$

④ Both are unary operator  
 $\rightarrow$  because they are applied on single operand.

(\*) You can not use lvalue, before increment/decrement operator.  
Ex -  $(a+b)$  <sup>lvalue</sup>  
 $\rightarrow$   $a+b$  lvalue.

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

$\rightarrow$  in any assignment statement "lvalue" must have the capability to hold the data.

$\rightarrow$  lvalue must be a variable as they have the capability to store the data.

$\rightarrow$  lvalue cannot be the function like  $(a+b)$  etc.

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

$\rightarrow$  Anything which is capable of returning a constant expression or value.

$\rightarrow$  Expression like  $(a+b)$  will return some constant value.

Question  $\Rightarrow$  What is the difference between pre-increment and post-increment operator?

$\Rightarrow$  Pre  $\rightarrow$  means first increment / decrement then assign it to another variable.

Post  $\rightarrow$  means first assign it to another variable then increment / decrement

$n = ++a;$

↓  
first a will be incremented by 1

if initial a = 5

then first a = 6

then assign to n = 6



$n = a++;$

↓  
example if 5

some value a



↓  
example if 6

(1)

#include <stdio.h>

int main()

{ int a = 4, b = 3;

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

Output : 7  
Explanation : a=4, b=3  
so a+b=7

3  
 $a+b$

$\Rightarrow$  We need to know a concept of token generation.

Lexical analysis is the first phase of compilation process.

(3b)  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. keyword var operator value

int a=5;

  
int [a] [=] [5] [:]

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

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

After as  $++$  is valid operator it generates token for  $\boxed{++}$

then as  $+b$  is not valid sequence it generates different token  $\boxed{+} \boxed{b}$

so  $a = 4, b = 3$  if we then increment  $a + ++b \rightarrow a + \overset{\text{First use}}{\boxed{++}} b \rightarrow 4 + 3 = 7$

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

int main() { int a = 4, b = 3; printf("%d %d", a + ++b); }

return 0; } ~~getchar();~~

3 tokens  $\boxed{a} \boxed{+} \boxed{++} \boxed{b}$

as unary operator needs an operand then  $a + \boxed{++} \boxed{b}$

will be merged  $a + \boxed{+} \boxed{b}$

as  $++b$  is increment then  $b$  will be 8

put the value in equation  $a + 8 = 12$

output will be 12

~~1. If there is no operator then it generates tokens~~

~~2. If there is operator then it generates tokens~~

~~3. If there is operator then it generates tokens~~

③ #include <stdio.h>

int main ()

{  
    int a=4, b=3;  
    printf ("%d %d", a++, ++b);  
    return 0;  
}

a>7

b>8

c>5

d>11

→  $\boxed{a} \boxed{++} \boxed{++} \boxed{+} \boxed{b}$   
as first need an operand ( $a++$ ) will be merged  
but second ++ need also an operand  
 $(a++) ++$

as  $(a++)$  ~~is~~ should be a l-value means  
has to be a identifiable location

B:ut a++ is not a l-value

So it will produce error.

✗ 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.

✗ According to stack Last in first out the  
evaluation start from right to left  
but printf print left to right ←

135.

(33)

So 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

~~for~~ 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 12.  
So finally  $i = 8$  and printf start to  
print from left to right.  
 $\rightarrow 13, 8, 8, 8$

int  $i = 10;$   
printf("%d %d %d %d", i++, ++i, +i, +i++);  
so here all are postincrement.  
 $\rightarrow 13, 14, 14, 10$

print 13 increment to 14  
increment to 12 | Incremented to 12  
incremented to 11 | but waiting for final  
incremented to 10 | print 10

13 14, 14, 10

## Relational Operator $\Rightarrow$

- (\*) All relational operator will return either true or false.

$$4 == 5 \\ \rightarrow \text{false}$$

$$4 != 5 \\ \rightarrow \text{true}$$

## Logical operator $\Rightarrow$

- (\*)  $\&$  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 conditions are false.

- (\*) ! operator is used to complement the condition ~~under~~ under consideration.

## Concept of Short circuit in Logical operator $\Rightarrow$

Short circuit in case of  $\&$ : simply means if there is a condition of 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;
}

```

3  
 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 of  $b$  note  $\Rightarrow$  If an expression ~~is~~ returns except 0 then it will be treated as true

then it will be true &  $b$  will be 9

so  $c$  will be true &  $b$  will be 1  
 O/P will be 1  
 9

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 ("%d\n", c);
```

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

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

so  $c$  will be true.

$b$  will remain same

O/P - will be 1

## Bitwise Operator $\Rightarrow$

It does bitwise manipulation.

### \* Bitwise And (&) operator $\rightarrow$

- 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.

both bits are 1

$$7 - 0111$$

$$4 - 0100$$

$$\hline 0100$$

$$7 \& 4 = 4$$

A	B	AB
0	1	0
0	0	0
1	1	1
1	0	0

### \* Bitwise OR (|) operator $\rightarrow$

- OR is a binary operator.

- Results of OR is 0 when both bits are 0.

both bits are 0

$$7 - 0111$$

$$4 - 0100$$

$$\hline 0111$$

$$7 | 4 = 7$$

A	B	AB
0	0	0
0	1	1
1	0	1
1	1	1

### \* Bitwise Not ( $\sim$ ) operator $\rightarrow$

- Not is a unary operator.

- Results of Not is 0 when bit is 1.

bit is 1 and 1 when bit is 0

$$7 \rightarrow 0111$$

$$\hline 1000$$

$$\sim 7 = 8$$

A	$\sim A$
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 %d"),

if (x&&y) //true

printf ("Result of x&&y is %d"),

return 0;

3

$$\Rightarrow x = 1 = 00000001 \text{ (1)}$$

$$y = 2 = 00000010 \text{ (2)}$$

$$1 \& 2 = 00000000 \text{ (0)}$$

$\therefore x \& y = \text{false}$

but  $x \& y = \text{True} \neq \text{False}$

$\therefore$  True  $\neq$  False

1 & 1 = 1

0 & 1 = 0

(16/12)

(A5)

## Left shift operator

its a binary operator

First operand  $\ll$  second operand

Whose bits have  
been left shifted

Decides number of places  
to shift the bits

- ⊗ When bits are shifted left, then trailing positions are filled with zeros

ff include <stdio.h>

int main()

{

char var = 3;  
printf("%d", var  $\ll$  1);  
return 0;

3 is 0000 0011 in binary

Var  $\ll$  1

Left shift by 1 position

3 = 0000 0110

↓

at trailing position  
there will be 0's  
6 in decimal

- ⊗ Left shifting is equivalent to multiplication by 2 right operand

Var = 3

Var  $\ll$  1

O/P : 6 [  $3 \times 2^1$  ]

## Right Shift Operator $\Rightarrow$

It is also binary operator.

First operand  $\gg$  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 \n", var >> 1);

return 0;

Var >> 1

Var = 3

Var = 3 = 0110 0011

0 110 - 1

0 000 0001

Leading pair

\* Right shifting is equivalent to division by 2 Right operat

Var = 3

Var >> 1

0 110 - 1 [3 / 2 ]

= 1

Bitwise XOR  $\Rightarrow$

- 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	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$$7 \rightarrow 0111$$

$$4 \rightarrow 0100$$

$$7 \wedge 4 = 3$$

~~#include <stdio.h>~~

~~int main()~~

{

~~int a=4, b=3;~~

~~a=a ^ b;~~

~~b=a ^ b;~~

~~a=a ^ b;~~

~~printf("%d %d", a, b);~~

~~return 0;~~

$$\Rightarrow a = 4 = 0100$$

$$b = 3 = 0011$$

$$a = a \wedge b = \frac{0100}{0011} = 0111$$

$$b = a \wedge b = \frac{0111}{0011} = 0100$$

$$a = a \wedge b = \frac{0111}{0100} = 0011$$

$$\therefore a = 0011 = 3, b = 0100 = 4$$

o IP will be  $a = 3$  and  $b = 4$

Assignment Operator  $\Rightarrow$

\*) 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.

$var = 5$   
 /  
 L value      R value  
 Assignment operator.

```

(*) int main()
{
    char a = 'f';
    a ^= 5;
    printf("%c\n", a);
    return 0;
}

```

$\Rightarrow$  (\*) 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.

$$a = f = 0111$$

$$a \wedge 5 \Rightarrow a = a \wedge 5 = f \wedge 5$$

$$\begin{array}{r}
 0111 \\
 0101 \text{ (relation)} \\
 \hline
 0010 = 2
 \end{array}$$

$$: a = 2$$

$$a + 3 \rightarrow a + a + 3 = 2 + 3 = 5$$

$\therefore$  printf("%c\n", a + 3) will print 5 & return 1 as it prints only one character.

$\therefore$  printf("%c\n", 1) it will also print 1 & return 1 as it prints only one character.

OP will be 51

## Conditional Operator $\Rightarrow$

If is a ternary operator

Suppose

char result;

int marks;

if (marks > 33)

{ result = 'P'; };

else {

result = 'F'; }

char result;

int marks;

result = (marks > 33) ?

'P'; 'F';  
↓

easy

3

Hard

True False

result = (marks > 33) ? 'P' : 'F';

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

→ If its value is true then result is 'P' else result is 'F'

Expression 1 ? Expression 2 : Expression 3  
false

```

X #include <stdio.h>
int main ()
{
    int var1 = 75;
    int var2 = 56;
    int num;
    num = sizeof(var1) ? (var2 > 23 ? ((var1 == 75) ? 'A', 0) : 0);
    printf ("%d\n", 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  
Var will be defined.

As C supports auto type casting so A will be stored in memory and then

will be 65. As in print placeholder is %d then it will print the ASCII value of char 'A' which is 65.

Ans - 65

## Comma Operator $\Rightarrow$ 32

(\*) Comma operator can be used as a separator

`int a=3, b, c;`  $\rightarrow$  `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 rejects them.

So here 8 will be printed and assigned to a  
O/p will be 8

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

$\Rightarrow$  first it will evaluate the operands  
so `printf ("%s\n", "Hello")` will be printed and rejected. 5 will be assigned to var

$\therefore$  O/p will be Hello

(53)

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

int a;

int 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 separator. Comma operator behaving like separator.

a) function call definition

b) Variable declaration

c) enum declaration

↳ int a = 3, 4, 8 is equivalent to

int a = 3; int 4; int 8;

order because we can't start name with digit.

(X)

```
int a;  
a = (3,418);  
printf ("%d", a);
```

→ As brackets has the highest precedence  
than any other operator.  
∴ 8 will be assigned to a  
o O/P will be 8

(X)

```
#include <stdio.h>  
int main()  
{  
    int var;  
    int num;  
    num = (var = 15, var + 35);  
    printf ("%d", num);  
    return 0;
```

→ 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 operators

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.

$$\underline{\text{Ex-}} \quad 2 + 3 * 5$$

$$(2+3)*5 = 25$$

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

$$E_x = 10/2 * 5$$

$$\text{Left to Right} - (10/2) \times 5 = 25 \checkmark$$

$$\text{right} + \text{left} \rightarrow 101(2^{*5}) = 1 \times$$

Category	operators	Associativity
parentheses/backet	( ) [ ] → . ++ --	Left to a right
unary	! ~ ++ -- + - * , & , size of ()	Right to left
Multiplication call v/c	* / %	L to R
Additive	+ -	L to R
Bitwise shift	<< >>	L to R

Relational	$L <= > >=$	L to R
Equality	$= = !=$	Left to Right
Bitwise And	&	L to R
Bitwise Xor	$\Delta$	L to R
Bitwise Or		L to R
Logical And	And	L to R
Logical Or	Or	L to R
Conditional	?:	Right to Left
Assignment	$= += *= /=$ $% = \wedge =$	R to L
Comma	,	Right to Left

Ex

```

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

int fun1()
{
    printf("A");
    return 1;
}

int fun2()
{
    printf("M");
    return 1;
}

```

3

3

Here + has more precedence. So first + will be 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.

It is not defined whether  $\text{first}()$  will be called first or whether  $\text{first}()$  will be called. Behaviour is undefined and output is compiler dependent.

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

Output shown above is because of the implicit brackets formed by the return value.

$\Rightarrow$  As relational has more precedence than equality.

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

$\text{True} \geq \text{True}$

O/P will be True.

Problem →

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

→ sizeof() operator will return 4 as int can hold 4 bytes.

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  $\bar{a} = 1;$   
 int  $\bar{b} = 1;$   
 int  $c = ++a \text{ || } b++;$   
 int  $d = b-- \text{ || } --a;$   
 $\text{printf}("d \cdot d \cdot d \cdot d", d, b, a);$

$a > 1111$

$b > 0100$

$c > 1001$

$d > 1101$

$\Rightarrow$  at first in  $c = ++a \text{ || } b++;$

$++a$  means pre increment so  $a$  is incremented by 1 and  $a$  becomes 2

as we all know any value except 0 is true

and then  $c = \text{True} \text{ || } b++;$

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.

$$\boxed{c = 1}$$

in the last  $d = b-- \text{ || } --a$   
 in the last  $d = \text{True} \text{ || } \text{true}$

$b--$  means post decrement  $b$  will point to then decremented to 0.

$$\boxed{b = 0}$$

so  $--a$  means first  $a$  is decremented by 1

as previously  $a$  was 2 so  $a$  will be now 1 it will be printed.

$$\boxed{OP = 1101}$$

$$\boxed{a = 1}$$

$$d = T \text{ || } T = 1$$

## Rapid Fire

(1) sizeof operator returns size in?

⇒ a) bits

    ↳ Bytes

    ↳ Kilobytes

    ↳ Megabytes

    → (b) bytes

(2) Which of the following is the correct initialization of variables?

    a) int a, b, c;

    b) int a, int b, int c;

    c) int a, b, c;

    d) int a, b, c;

    → d) uses ; as separator

(3) What does printf function returning

    a) size of integer

    b) size of character

    c) number of characters printed on the screen.

    d) size of variable

    → e) number of char printed

(61)

ASCII decimal range of characters

①

A 102 is?

a) ~~65 - 20~~

b) ~~97 - 122~~

c) ~~100 - 127~~

d) ~~1 - 28~~

a) ~~65 - 20~~

a - 2

97 - 112

⇒

Size of integer?

a) 32 bytes

b) 8 bytes

c) 16 bytes

Depends from machine to machine

Depends from machine to machine

⇒

d) Depends from machine to machine

Consider the following variable declarations

and definitions in C?

i)  $\text{int } 3 = 1;$

ii)  $\text{int var-3 = 2;}$

iii)  $\text{int - = 3;}$

Which of the following is correct

a) Both (i) is valid

b) Only (ii) is valid

Both (i) and (ii) valid

⇒ (i) Both (i) and (ii) is valid

- 7) Consider the following options
- a) int var;  
extern int var;  
which is correct
- b) Both statements only declare var not define.  
c) Both statements declare and define them.  
d) Statement 1 declares & statement 2 defines  
a variable.
- e) Statement 1 declares ~~var~~ and defines but  
statement 2 just declares.

→ 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 Var as it does  
not give the compilation error.  
But as C can not handle scope resolution.

(63)  
So var on the right doesn't 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.

① #include <stdio.h>

int main()

{

    int var = 10;

    3

    3 printf("%d\n", var);

3

3 (garbage value) -> 10

a) 10  
garbage value.

b) Compilation error -> long 2 errors b)

~~so as var is inside the inner block~~

→  
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\n", ~var);  
    return 0;

    a>10 b>-10 c>-11  
    d>-5

O/P -

$$\sim a \equiv -a - 1$$

$$\equiv -10 - 1 = -11$$

$\text{var} = 10(00000000_8 1010)$  9 bytes

1st complement of var - ( $\sim \text{var}$ )

1111 1111 1111 0101 (1st complement)

-1d means signed integer

→ MSB is 1 means negative so we have  
to convert to  $2^8$  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 - elseif ⇒

if ( is clicked)

{ Application page will be opened;

elseif ( is clicked)

{ Google chrome is opened;

3

## FAQs

(\*) Is it necessary to put the else part?

→ 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  $\Rightarrow$  ... ?

Switch is a great replacement to long else construct.

int n = 2;

switch (n) → checks if  $n = 2$

2

case 1: printf ("n is 1");

break;

case 2: printf ("n is 2");

break;

case 3: printf ("n is 3");

break;

default: printf ("default");

break;

3

→ If none of the cases are satisfied then default will be executed.

~~(\*) Suppose  $n=1$  condition is satisfied and there is no break after printf then subsequent expression will also get evaluated until we reach the next break.~~

~~(\*) In this case  $n=2$  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 be evaluated after all the cases.~~

~~(\*) The cases will be evaluated from Top to bottom.~~

~~(\*) Default is optional.~~

Facts -

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

~~case 1 : > if will give duplicate case value~~

~~case 1 :~~

~~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 X~~

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

~~Case 3+3.5;  
Case 3+4\*5.1~~

~~allowed ✓~~

~~case 3.14:  
case 1.1~~

~~not allowed X~~

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

```

int u=2; y=2, z=23;
switch (u)
{
    case y:
    case z:
        not allowed X
}

```

```

#define y 2
#define z 23
int u=2;
switch (u)
{
    case y:
    case z:
        allowed ✓
}

```

## Loops

Suppose you are a CEO of a MNC. You have thousands of employees. 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 employeeCode = 1000;
```

```
while (employeeCode != 0)
```

```
{
```

Send the above notice.

```
employeeCode --;
```

3

## While loop

Syntax - While (expression)

{      }

- - statements;

- - ;

If the expression is true then the statements will be executed until the expression is false.

Ex - int i=3;

while (@i>0) IP- 321

```
{
    printf("%d", i);
}
```

For loop  $\rightarrow$

Syntax -  $\text{for} (\text{initialization}; \text{condition}; \text{increment/decrement})$   
{ statements; }

3

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

3

Explanation  $\rightarrow$  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 happens. The same  
process repeats until the condition is evaluated to be false.  
If it is false the loop is terminated.

Do While  $\Rightarrow$  Syntax  $\Rightarrow$  do

{ statements;  
} while (condition);

When should i prefer do while over while? Important

$\Rightarrow$  Suppose you are asked to enter an integer until  
he/she enters a value  $= 0$

int n;  
printf ("Enter an integer in %d");

scanf ("%d", &n);

while ( $n \neq 0$ )

{ printf ("Enter an integer in %d");

scanf ("%d", &n);

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

(70)

using do while  $\Rightarrow$

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 at least once then we will use do while.

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

Break  $\Rightarrow$

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:  $\Rightarrow$

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.

71

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

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

3  
So as  $i = 1024$  is initialized before so we simply keep the initialization step as blank then

we see  $i$  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  $i = \frac{1024}{2} = 512$

$$i = 256 \quad 1024 = \underline{100.0000.0000}$$

$$i = 128$$

then  $i = 64$   
 $i = 32$   
 $i = 16$   
 $i = 8$   
 $i = 4$   
 $i = 2$   
 $i = 1$   
 $i = 0$

So answer will be 11

(72) 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);
    }
}
```

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

- ④ At first i is initialized to 0.
- ④ Case 0 is evaluated to be true i will be incremented by 5
- ④ as after incrementation there is no break statement so the subsequent statements ( $i=2$ ,  $i=5$ ,  $i=4$ ) will also get evaluated though other case is not satisfied.  
i will be 16 after all execution.

(\*) print will print 16 and with increment i to 17

(73)

(\*) Again control goes to switch statement as none of the case is evaluated to be true so default will be evaluated. i will be 21

(\*) print will print the 21. Then again will increment i to 22

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

Output - 16 21

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

int i = -5;

while (i <= 5)

{ if (i >= 0)

break; } // break part

else

if (i = 0 + 1) { // i = 1, 2, 3, 4, 5  
continue; } // loop

print ("Nesco");

→ as  $-5 <= 5$  is true then control will go to while loop.

(\*) as  $-5 >= 0$  is false then it will go to else part.

(\*) as  $-5 >= 0$  is false then it will go to else part increment

(+4) 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,  $0 >= 6$  is false  
then break encountered & will terminate the loop

(\*) Whenever encounter printf all  
So O/P is 0 times.

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

Assuming sizeof(unsigned int) = 4

#include <stdio.h>  
int main ()

{ unsigned int i = 500; }

while ( i++ == 0 ); } ; } compiler error

printf ("%d\n");

return 0;

→ (\*) 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  $\square$  after the while loop it means while loop has no body and the very next statement  $\text{printf}$  is not a part of it.

\* So a time will come 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  $\text{printf}$  will print the value 1.

O/P will be 1

~~(\*)~~ What is the output of the following programme (76)

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

```
{ int n = 3;
```

```
if (n == 2) { n = 0; }
```

```
if (n == 3) n += 3;
```

```
else n += 2;
```

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

```
return 0;
```

~~(\*)~~ at first n is defined as 3

~~(\*)~~ if ( $n \geq 2$ ) there is  $\boxed{2}$  after that it means the next statements  $n = 0$ ; is not part of it.

~~(\*)~~ n will be again initialized to 0

~~(\*)~~  $0 == 3$  is false so ~~n += 2~~ n + will not execute

~~(\*)~~ else part will be execute and n will

be  $0 + 2 = 2$

~~-~~  $n = 2$  (O/P)

What is the output of the following?

(\*) program  
int main ()

{  
    int i=0;  
    for (printf("one\n"); i<3; printf("one\n"),  
                i++)

{  
    printf("Hello\n");

3  
given i:

3  
In the very first line of the main function

$\Rightarrow$  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 consequence

it will be returned. Hence checking part there

next in the condition checking part there

(\*) and operator.  $i < 3$  is true

is logical

so loop continues

## Functions

(102)

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

Syntax  $\Rightarrow$  In function-type function-name (set of input).

$\oplus$  It is not mandatory for giving input.  
It can also work without any input.

Benefits  $\Rightarrow$  Reusability  $\rightarrow$  once the function is defined it can be reused over and over again.

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

Ex  $\Rightarrow$  Scanf function

Example  $\Rightarrow$  ~~#include <stdio.h>~~  
int areaofRect (int length, int breadth)

{  
    int area;  
    area = length \* breadth;  
    return area;  
}

int main ()

int area = areaOfRect(i, j);

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

i = 50, j = 10;

area = areaOfRect(i, j)

printf("%d", area);

3

Declaration  $\Rightarrow$  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  $\rightarrow$  It is not necessary to put the name of the parameters in function prototype, must

int fun(int a, int b);

not necessary

(104)

Ex - #include <stdio.h>

```
char fun(); // declaration
```

```
int main()
```

```
{
```

```
    char c = fun();
```

```
    printf ("Character is: %c", c);
```

3

```
char func(); // definition
```

```
{
```

```
    return 'a';
```

D/I/P - character is a

Is it always necessary to declare the function before using it?  $\Rightarrow$

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

If you want to use any function but there is no definition of that function before that. Then you must declare the function prototype. As compiler can't read the code from top to bottom, it

Will produce implicit declaration of function. (15)

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

Function Definition  $\Rightarrow$  Function

definition 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; } // This is called body of function

How function works  $\Rightarrow$

#include <stdio.h>

int add (int, int); — (1)

int main ()

{ int m = 20, n = 30, sum;  
    sum = add (m, n); — (2)  
    printf ("%d", sum); } // sum is l.d. of sum

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

01P - sum is 50

Q) As there is no definition of the function before calling it, so we declared the prototype of the function.

④ There is no need to mention names of the parameters.

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

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 definition of the function.

Argument is the actual value of the parameters that gets passed to the function.

Note - Parameter is called as Formal parameter.

Argument is called as Actual Parameter.

$\text{sum} = \text{add}(m, n);$

variables in calling function  
arguments

$\text{int sum(int a, int b)}$

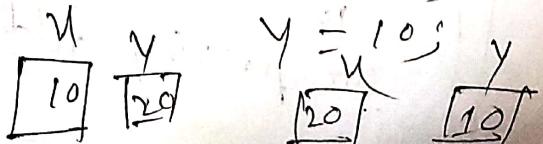
parameters

Call By Value  $\Rightarrow$  Here values of actual parameters will be copied to formal parameters.

These two different parameters store values in different location.

$\text{int } x = 10 \rightarrow y = 20;$

$\text{fun}(x, y);$



$\text{int fun(int x, int y)}$

$$x = 20;$$

$$y = 10;$$

$$y$$

(\*) Variable  $x$  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.

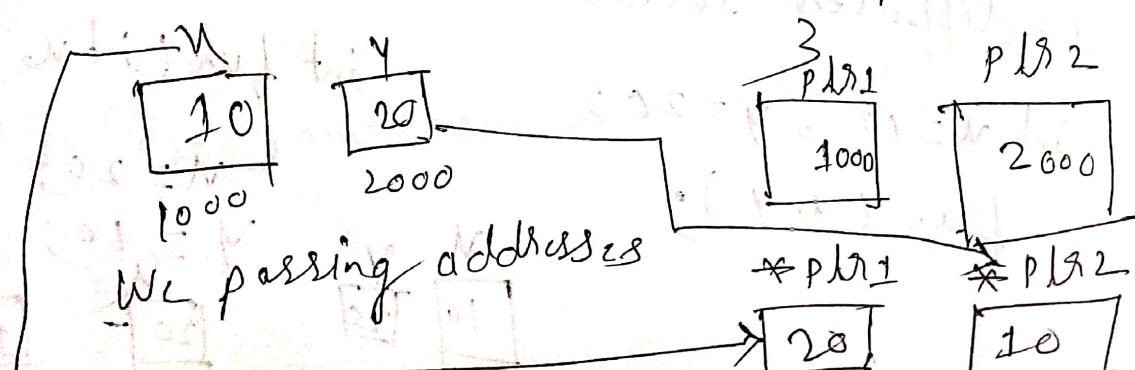
In short we just pass the value not the variable itself.

Call by Reference  $\rightarrow$  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 n=10, y=20;
int fun(int *ptr1, int *ptr2);
fun(&n, &y);
```

address of operator



(Q) As normal variable can not hold the address  
so we need special type of ~~operator~~ variable  
called pointer  
pointer are those variables which can hold  
addresses.

\*x means the value of the ~~pointer~~  
\* is used to access the value which is  
called dereference operator.

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

Problems

1. int fun (int num)  
{  
 int count = 0;  
 while (num != 0)  
 {  
 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 be executed

(10)

Count will be 1 and then right shift assignment operator

$$\text{num} \gg= 1 \rightarrow \text{num} = \text{num} \gg 1$$

and we know  $\gg$  in right shift the number is changed to  $\text{num} = \text{num}/2$  right operand

as here right operand is 1 then

$$\text{num} = 435/2^1 = 217$$

Count 1

$$\text{num} = 217/2^1 = 108$$

Count 2

$$\text{num} = 108/2^1 = 54$$

Count 3

$$\text{num} = 54/2^1 = 27$$

Count 4

$$\text{num} = 27/2^1 = 13$$

Count 5

$$\text{num} = 13/2^1 = 6$$

Count 6

$$\text{num} = 6/2^1 = 3$$

Count 7

$$\text{num} = 3/2^1 = 1$$

Count 8

$$\text{num} = 1/2^1 = 0$$

Count 9

$$\text{num} = 0/2^1 = 0$$

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

int fun (int num)

2

int count = 2;

while (num)

{

count ++;

num  $\gg= 2$ ;

return (count);

The value returned by fun(435) is -

Count = 2

num = 935

(11)

Count = 3

num =  $935/2 = 108$

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.

Q) The output of the following C program

is -

void f1(int a, int b)  
int matn()

{ int c;

c=0; a=b; b=c;

int a=9, b=5,

c=6;

3  
void f2 (int \*a, int \*b)

f1(a,b);

f2(&b,&c);

{ int c;

c=\*a; \*a=\*&b; \*b=c; c=a-b

3  
cout << c;

3  
The calling of f1 function is useless

⇒ The calling of f1 function is useless  
as it is ~~call by value~~ 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.

(112)

(b) 'c' is a local variable in  $f_2$  function which holds value of pointer a means value of b [ $c = 5$ ] (local to  $f_2$ )

(c) Value of pointer b = value of var c ( $\text{main}$ ) 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  $f_2$ ).  $c = 5$

(e) a is remain same as its address is not passed.  $a = 9$

$$\therefore C - a - b = 5 - 9 - 6$$

( $C - a - b$  = address of b - address of a - address of b)

~~(a)~~ int fun()

static int num = 16;

return num;

int main()

{  
    fun();  
    fun();  
    fun();  
}

printf("%d", num);  
return 0;

What is the O/P of the C programme?

a) infinite loop

b) 13 10 7 4 1

c) 14 11 8 5 2

d) 15 12 8 5 2

Before solving the question you should have knowledge of for loop

② 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 happens. 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 increment or decrement happens and fun() will be called and repeat same step until the condition is evaluated to be false.

The answer will be (C) 19 11 8 5 1

## Static Function $\Rightarrow$

### Basics

- (\*) In C, functions are global by default. This means if we want to access the function outside from the file where it's 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 in front of the function.
- (\*) Static means we can't access the function outside the file.
- (\*) Reuse of the same function in another file is possible. (You can declare same function in another file.)

```

main.c
=====
#include <stdio.h>
#include <math.h>

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

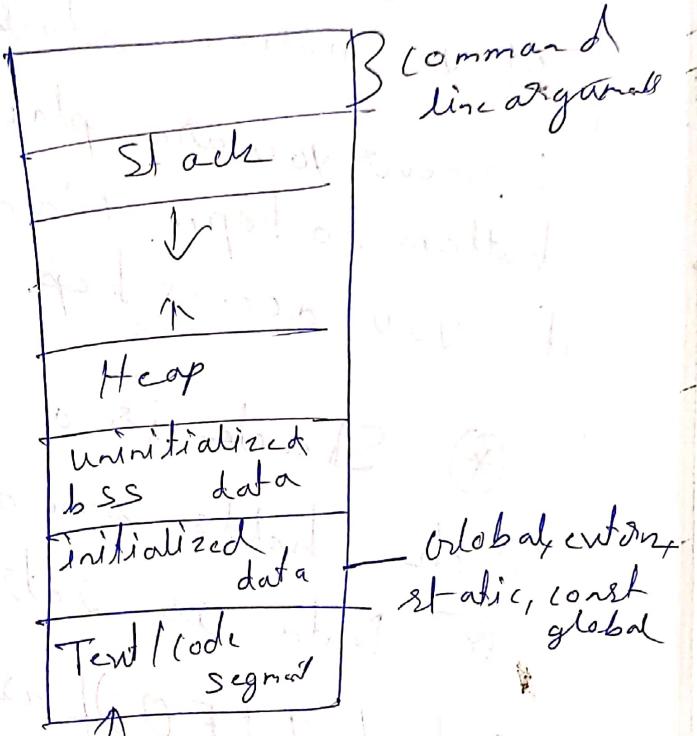
int main()
{
    int a, b, sum;
    a = 3;
    b = 4;
    sum = fun(a, b);
    printf("%d", sum);
}
  
```

If you put static in front of fun  
`static int fun (int a, int b)` it will  
 give you error - undefined reference.

## Static and Dynamic Scoping ↗

### Memory Layout →

- memory segment
- Text / code segment
- Data segment
  - initialized
    - Read only
    - Read write
  - uninitialized
    - (bss - Block started by symbol)
- stack
- 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

① Reinitialization is not valid.

② After initialization assigning value is not allowed in global scope as

the compiler implicitly adds it before  
In global scope. (116) Variable

static int j = 27;

i = 45;

→ not allowed

static int i; j;

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 (array) which holds some data.

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

\* Two operations - Push - Placing element  
Pop - Retrieving element

main()

{

fun1();

3

fun1() {

fun2();

3

return

fun2() {

fun3();

3

return();

|        |
|--------|
| fun3() |
| fun2() |
| fun1() |
| main() |

under execution

Stack (call stack)

Whenever it 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

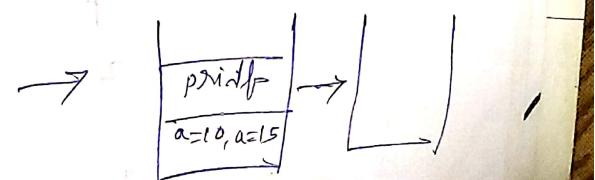
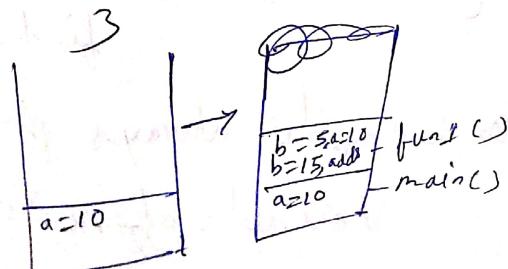
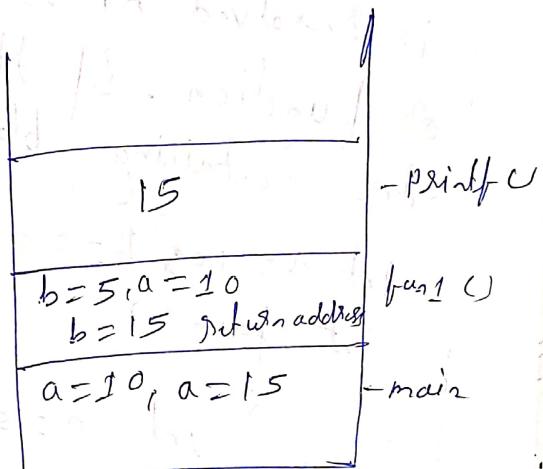
1. Locals of the callee

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  $\Rightarrow$  Scoping is necessary if you want to reuse variable names.

④ 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.

Variables we can use  
int func()  
int func()

$\text{int } a = 10^\circ$ ;  $\text{slope } 3$

Static Scoping  $\Rightarrow$  Definition of a variable  
is resolved by searching its containing block  
or function. If that fails then searching the  
outer containing block and so on,

$$\text{int } a = 10^\circ, b = 20^\circ;$$

int func()

{ int a=5;

$\sum \text{int } c_i$

$$c = b/a;$$

printf C-style

\* As there is no a def in definition block

④ In outer block weight

④ Again in

other block w.  
find h

(\*) Compiler always prefers

## Variables of nearest block

```

int fun1 (int); // Prototype
int fun2 (int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1 (a);
    printf ("%d,%d,%d", a);
}

```

3  
int fun1 (int b)

$$\{ \quad b = b + 10 \}$$

$b = fun2 (b)$

get  $b'$

3  
int fun2 (int b)

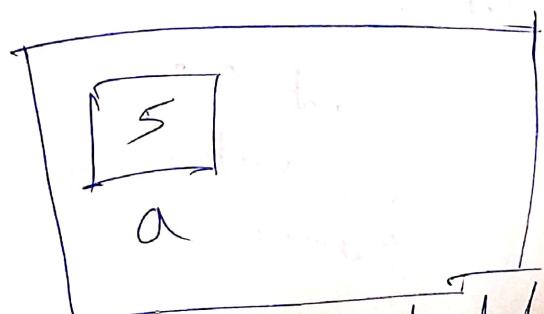
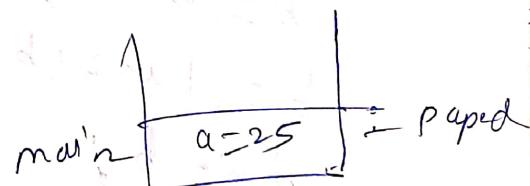
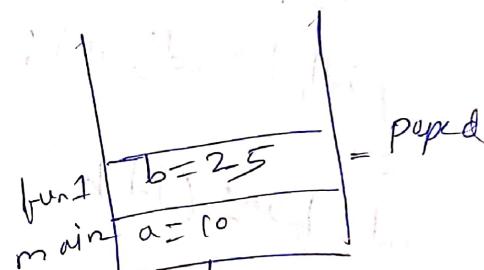
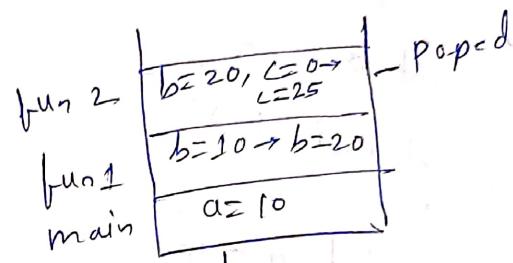
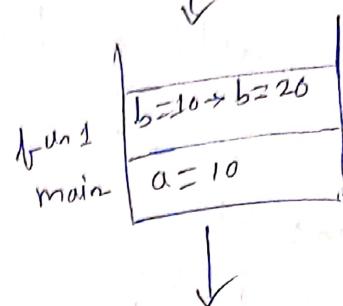
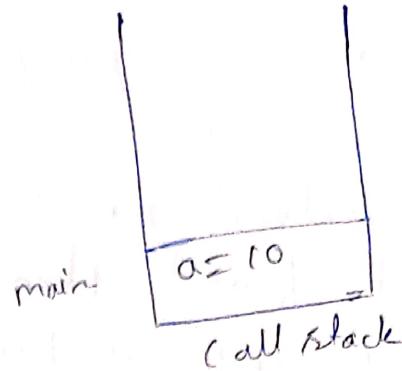
$\{ \quad int c;$

$$c = a + b;$$

get  $c'$

3

O/P - 25



(B)

Dynamic Scoping  $\rightarrow$  Definition of a variable  
 $\rightarrow$  is containing block and  
 is resolved by searching its calling function  
 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);
}

```

```

3
int fun1 (int b)
{
  b = b + 10;
  b = fun2(b);
  return b;
}

```

```

3
int fun2 (int b)

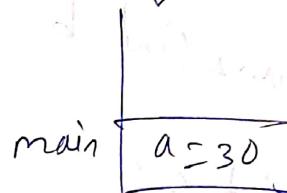
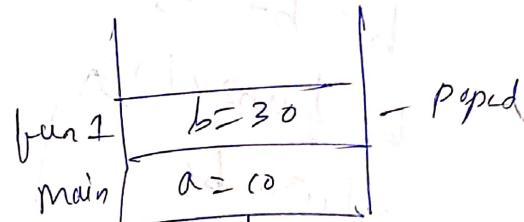
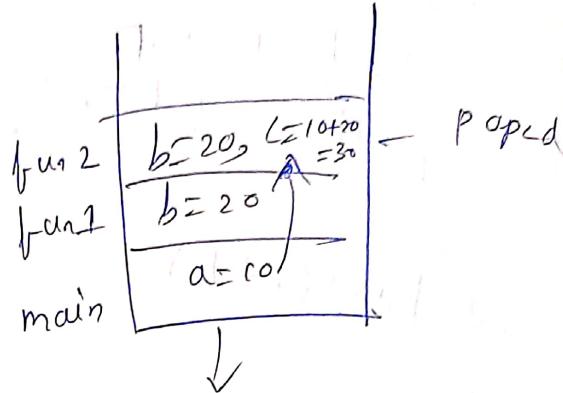
```

```

{
  int c;
  c = a + b;
  return c;
}

```

3



0 (p -> 3) 0

# Homework Problem

12.0

int a, b;

void print()

{ printf("%d %d", a, b); }

int fun1()

{ int a, c;

a = 0; b = 1; c = 2;

return c;

⇒ with static Scoping

a) 2 9

b) 3 1

c) 2 5

d) 3 9

⇒ with dynamic Scoping

a) 2 9

b) 3 1

c) 2 5

d) 3 9

int main()

{

a = fun1();

fun2();

3

⇒ Note - If we don't declare the variable in the function but assign it means we are doing this with global variables.

(21) 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 funcy

\* Global variable b was last set in funcy to be 1.

o (P- 3 1

2) caller function of print is funcy  
in funcy 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, LISP uses dynamicalled scoping

Perl is a programming language which supports both static as well as dynamic scoping.

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

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

X under static scoping  
Y under dynamic scoping  
a)  $x=10, y=10$   
b)  $x=20, y=10$   
~~c)  $x=10, y=20$~~   
d)  $y=20, y=20$

2) What will be the O/P of the following pseudocode when parameters are passed by reference, and dynamic Scoping is assumed? (12)

$a = 3$

void n(x)

{  
    n = x \* a;  
    print(n);  
}

void m(y)

{  
    a = 15;  
    a = y - a;  
    n(a);  
    print(a);  
}

$a = 4 - 3 = 1$

$n(1) \rightarrow 200$

print(a);

3

So here address of a is passed to m function.

$y = 4a$

$$a = 3 - 1 = 2$$

$n(200) \rightarrow n = 4a$

$$n = 2 * 2 = 4$$

print(n)  $\rightarrow 4$

O/P - 1 4

void m(a())

{  
    a[1] = 100;  
    m(a());  
}

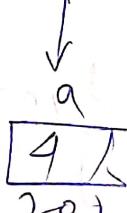
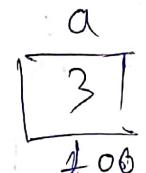
3

$a[6, 2]$

$b[6, 6]$

c) 14, 2

d) 9, 9



Array

## Recursion

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

Ex- int fun()

[

fun();

3

int fun(int n)

{ if ( $n=1$ ) return  
    return 1;

else, return  $1 + \text{fun}(n-1)$

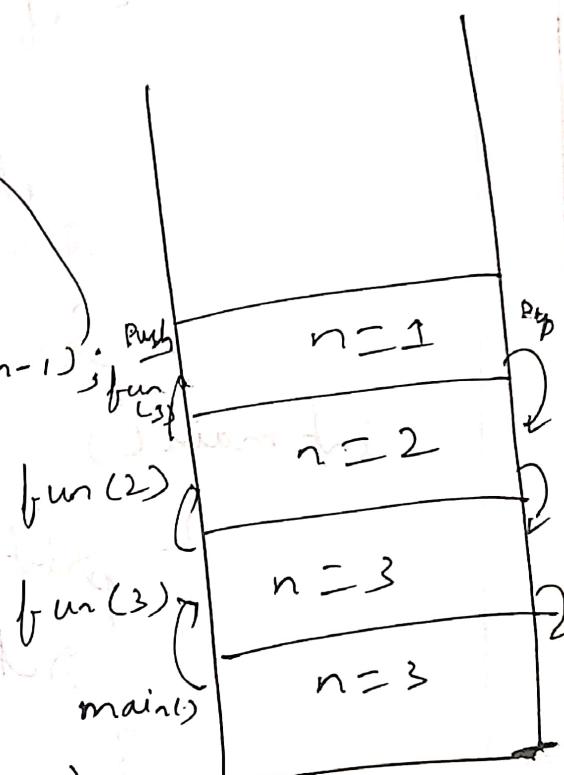
3  
int main()

{ int n=3;

printf("%d", fun(n));

return 0;

3



Stack

olp-3

(125)

$\text{fun}(3) \rightarrow \text{return } 1 + \underbrace{\text{fun}(2)}$   
 ↓      ↓  
 $\text{return } 1 + \text{return } 1 + \underbrace{\text{return } \text{fun}(1)}$   
 ↓  
 $\text{return } 1 + \text{return } 1 + \text{return } 1$   
 ↓  
 3

H.W | #include <stdio.h>

int fun(int n)

{ if (n==0){ return 1;  
      3  
    }

else return 7 + fun(n-2);

3

int main()

{ printf("%d", fun(4));

return 0;

3

(a) 9

(b) 7 (d) 12

(c) 15

$\Rightarrow \text{fun}(4) \rightarrow 7 + \text{fun}(2) \rightarrow 7 + 7 + \text{fun}(0)$   
 → 15

- (120)
- 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

$$7 \quad 5! = 5 \times 4 \times 3 \times 2 \times 1$$

~~int fact (int~~      base case

~~fact(1)=1~~

~~fact(2)=2\*1=2\*fact(1)~~

~~fact(3)=3\*2\*1=3\*fact(2)~~

~~fact(4)=4\*3\*2\*1=4\*fact(3)~~

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

#include <stdio.h>

int fact (int n)

{ if (n==1) } - base  
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 calls the same function again.

Structure:      fun()  
                      {  
                       - . . .  
                       }  
  
                      fun()  
                      {  
                       - . . .  
                       }  
                       }

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

The diagram illustrates the execution flow between two functions, `func()` and `func2()`. It shows two separate stack frames. The left frame, associated with `func()`, contains a local variable `a` with the value `10`. The right frame, associated with `func2()`, contains a local variable `b` with the value `20`. Arrows indicate the flow of control from the end of `func()` back to the caller, and then into `func2()`.

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(~~int~~)

{ if ( $\alpha \leq 10$ )

```
{ printf("%d\n", n+1); }
```

9 + + ;

Conclusions

void even (~~int~~)

{ if ( $n \leq 10$ ) }

```
printf ("%d,%-1j");
```

$$3 \quad \text{odd}(s; \frac{3}{2} n)$$

```

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

```

Tail Recursive  $\Rightarrow$  A recursive function is said to be tail recursive if the recursive call is the last thing done by the function.  
There is no need to keep record of the previous state.

```

void fun(int n)
{
    if (n == 0) // return back to previous
        return; // function
    else printf("%d", n);
    return fun(n - 1); // last thing
}

```

```

int main()
{
    fun(3);
    return 0;
}

```

3

(129)

IV) Non fail Recursion  $\Rightarrow$  If Recursive call is 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; }  
0 1 2

Advantage of Recursion  $\Rightarrow$

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

(129)

(iv) Non tail Recursive  $\Rightarrow$  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);

3

int main()

{ fun(3);      O/P - 1 2  
    return 0;

Advantage of Recursion  $\Rightarrow$

(i) Every recursive program can be modeled into an iterative program but recursive programs 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  $\Rightarrow$

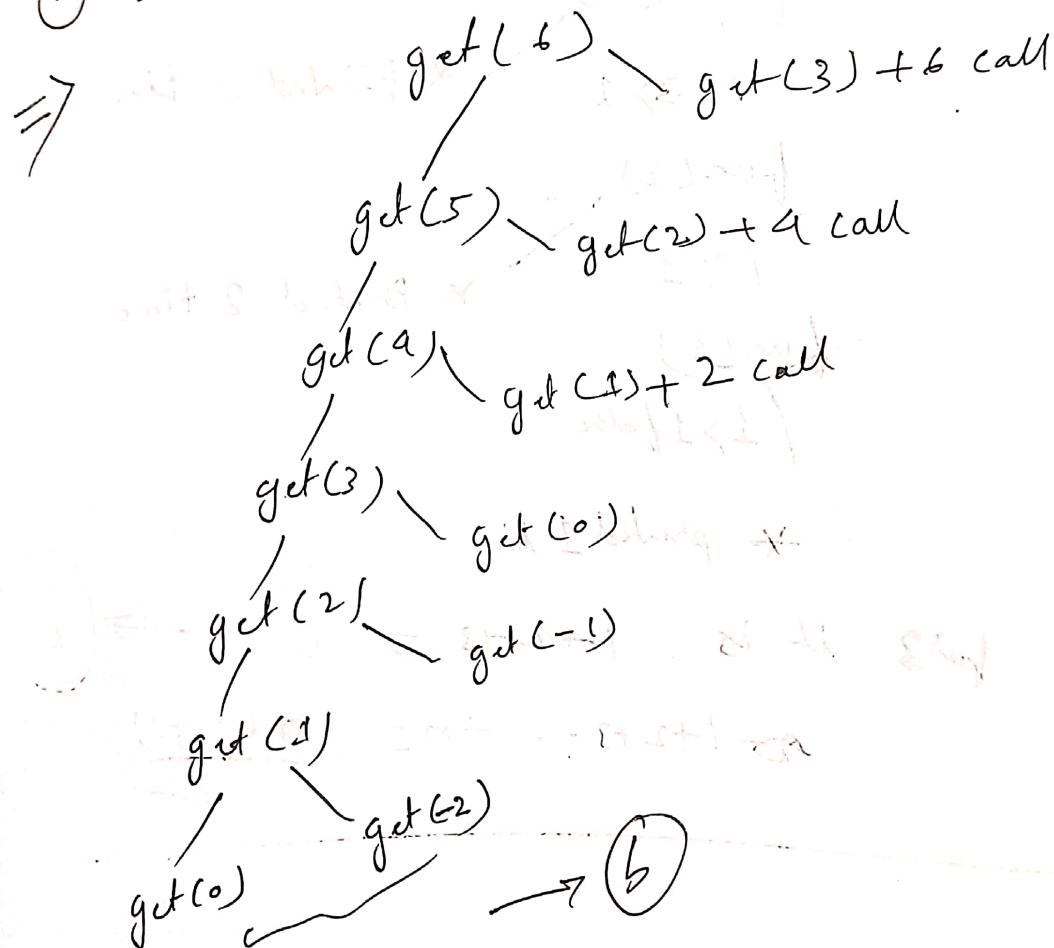
① void get (int n)  
 { if ( $n < 1$ ) return;

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



(131)

(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)  $\frac{n(n+1)}{2}$
- (c)  $n^*$
- (d)  $n_{\text{non}}$

3

 $\Rightarrow$ For  $n = 3$ 

fun(3)

3 &gt; 1

\* Printed 3 time

fun(2)

1 &gt; 1

\* Printed 2 time

fun(1)

1 &gt; 1 false

\* printed 1 time

For 3 it is  $1 + 2 + 3 = 6 \rightarrow$ 

$$\text{as } 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

(b)

(3) int f(int j) {

{ static int i=50;

int k;

if (j==5) {

{ printf("%s", "Something"); }

k=f(i); }

return 0; }

}

else return 0;

}

Which of the following is true?

- (a) The function returns 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  $j=50$ .

$\Rightarrow$  if  $j = 1, 2$  or anything except 50 then  
if part will not executed (b) X

if  $j = 50$  then something will printed (a) X

if  $j = 50 + n$  then function n (c) X

(d) V

(4) int fun(int n)

{ int n=1, k;

if (n == 1) return n;

for (k=1; k < n; ++k)

n = n + fun(k) \* fun(n-k)

return n;

The \$ return value of fun(5) is

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

$\Rightarrow$  1st iteration  $k=1$

$$n = 1 + \cancel{\text{fun}(1)} * \cancel{\text{fun}(4)}$$

$$\text{fun}(1) = 1$$

$$\text{fun}(2) = 1 + \cancel{\text{fun}(2)} \\ \cancel{* \text{fun}(3)}$$

$$= 2$$

$$n = n + \cancel{\text{fun}(2)} * \cancel{\text{fun}(3)}$$

$$\text{fun}(3) = 1 + \cancel{\text{fun}(4)} \\ \cancel{* \text{fun}(2)} \\ + \cancel{\text{fun}(2)} * \cancel{\text{fun}(3)}$$

$$= 5$$

$$n = n + \text{fun}(3) * \cancel{\text{fun}(2)}$$

$$k=3$$

$$n = n + \cancel{\text{fun}(4)} * \cancel{\text{fun}(1)}$$

$$\therefore n = 1 + \text{fun}(1) * \cancel{\text{fun}(4)} + \text{fun}(2) * \cancel{\text{fun}(3)} \\ + \cancel{\text{fun}(3)} * \cancel{\text{fun}(2)} + \cancel{\text{fun}(4)} * \cancel{\text{fun}(1)}$$

$$= 1 + 2 * [ \text{fun}(4) * \cancel{\text{fun}(3)} + \text{fun}(2) + \text{fun}(3) ]$$

$$\begin{aligned}
 \text{fun}(a) &= 1 + \text{fun}(3) * \text{fun}(3) + \text{fun}(2) * \text{fun}(2) \\
 &\quad + \text{fun}(3) * \text{fun}(1) \\
 &= 1 + 5 + 9 + 5 = 15
 \end{aligned}$$

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

$$= 51$$

(5) `Void count (int n)`

{ static int d = 1;

`printf ("%d", n);` if digit

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

d++

if (n > 1) count (n-1);

else printf ("%1.d", d);

return 0;

int main ()

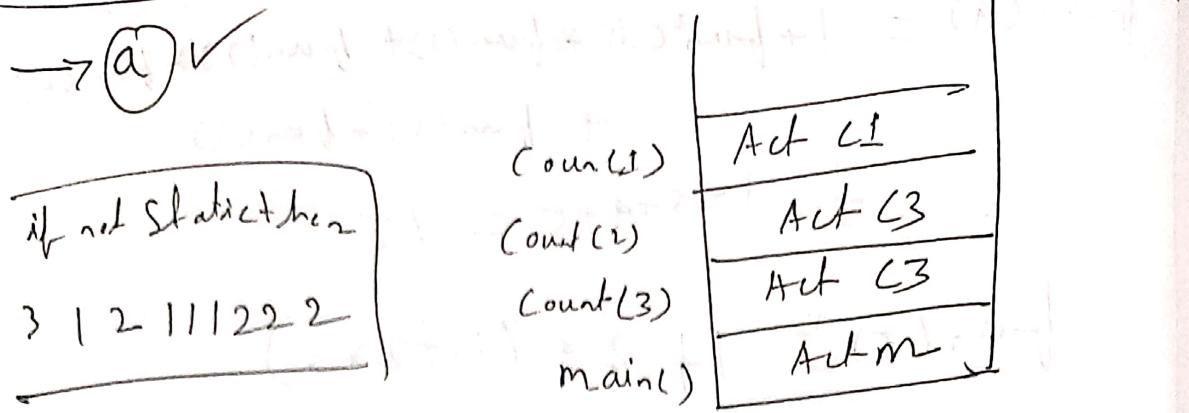
global { count(3); }

(a) 3 1 2 2 1 3 4 4 9

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

(c) 3 1 2 2 1 3 9

(d) 3 1 2 1 1 1 2



After printing 3 1 2 2 1 3 q we ~~pop~~ again come back to count(2) (calling function) and the next statement (Print d) will be executed.

Count(2), count(3) deactivated.

Rapid Fire Question  $\Rightarrow$  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

A declaration - tells compiler

function name

return type

Parameters

③ Which keyword is used to come out of loop only for that iteration?

- (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 supports dynamic Escaping it

- (a) yes (b) no

⑥ In C Parameter 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 parameters of any type

- (c) Function can be called with any number of integer parameters

- (d) Function can be called with one integer

(3P)

(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 until stack overflow happens
- d) Hi will be printed once because main() inside main() is ignored.
- g) In a context of "break", "continue" statement which is best of 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("Hm%"),

i--;

(a) Hm%

will be printed 10 times

(b) n

n n in 9 Y

(c) Compile error

(d) none.

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

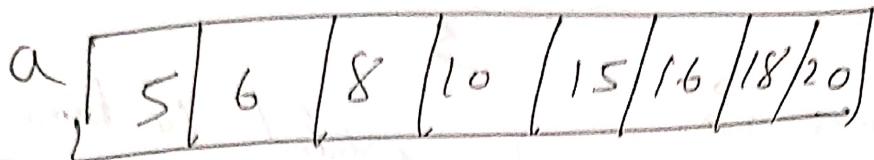
(\*) In C parameters are always passed by value. Pass by reference is simulated in C by explicitly passing pointer value.

(\*) In C only static Scoping supports

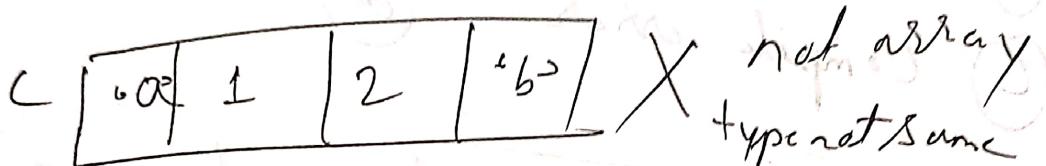
(\*) In C it is not mandatory to declare a function before use. void add(int, int); definition is mandatory. ~~function prints all - either if~~

(139)

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



a is an array of integer type and has 8 values



→ One dimensional array

Syntax for declaring

data-type name-of-array [number of elements]  
 For ex- int arr[5]; // declare + definatn.



definition — allocating memory

Memory = size of (int) allocated

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

int arr[5]; ✓      int arr[-5]; ✗

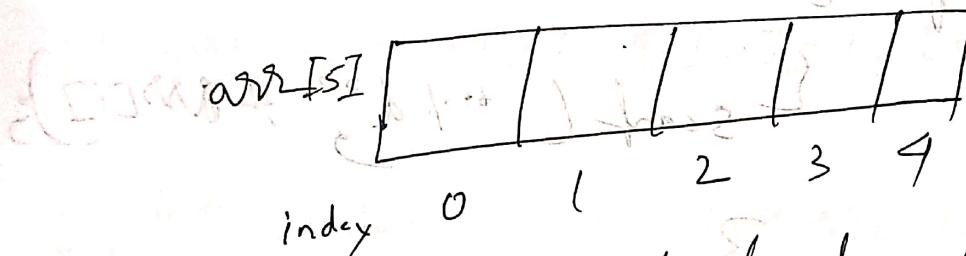
int arr[5+5]; ✓      int arr[a=2/3]; ✓  
int arr[5\*3]; ✓

~~(\*)~~ Specifying the length of an array using Macro is considered to be an excellent practice  
~~#define N 10~~ 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])



arr[0] - value of first element

index range :  $0 \rightarrow n-1$

Example = Dot notation

More about ~~length~~ length of array

0 is last index

length = no. of elements

Initializing 1D array  $\Rightarrow$

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

(ii) ~~int arr[5] = {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];~~ Value access 0 to 4

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

~~{~~  
~~scanf("%d", &arr[i]);~~

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

$\Rightarrow \text{int arr[10]} = \{1, 2, 6, 3\}$

~~the remaining field will be initialized as 0~~

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

(192)

(X)

int arr[10] = { 3; } ;  $\Rightarrow$  illegal

you must have specified at least one element

it is also illegal that you can not add more number of array than it ~~are~~ initialized.int arr[2] = { 1, 2, 3 }  $\Rightarrow$  XDesignated initializing  $\Rightarrow$ 

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

we want 1 in position 0

2 in 1 5 rest are 0

arr[0] = 3 arr[1] = 1 arr[2] = 2 arr[3] = 0 arr[4] = 0 arr[5] = 3 arr[6] = 0 arr[7] = 0 arr[8] = 0 arr[9] = 0

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

index value ~~for all others are 0~~  $\Rightarrow$  [0] = 1, [5] = 2, [6] = 3

Designator

arr[0] = 1  $\Rightarrow$  2nd slot is explicitly

Advantages: no need to bother about order

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

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

(143)

What if I won't 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] = 50, [20] = 4, [1] = 3  
[45] = 78 };

length will be 50

(\*) you can mix both traditional &  
designated

int a[ ] = { 1, 7, 15, [5] = 50, 6, [8]  
= 9 };

if there is any clash then designator  
initializer will win

int a[ ] = { 1, 7, 15, [2] = 50 };

at index 2 there is 4 also 50 so will  
be considered & based on designation A  
equivalent

↳ s = int a[ ] = { 1, 7, 50 };

↳ a[2] = 50

4

Counting number of elements.

We know `sizeof()` operator returns how many bytes it occupies in Memory + total

: number of elements =  $\frac{\text{sizeof}(\text{name of array})}{\text{sizeof}(\text{name of array}[0])}$  \ 1 element

~~int a[4] = { 1, 2, 3, 4 }~~

~~printf( " %d ",  $\frac{\text{sizeof}(a)}{\text{sizeof}(a[0])}$  );~~

$\Rightarrow 4$

Multidimensional array  $\Rightarrow$

~~It is an array of array~~

Declaration is - data-type `name of array [size1][size2] .. [sizenn]`

For -

~~int a[3][4]; // two dimensional array~~

Total number of elements :

$$a[10][20] \Rightarrow 10 \times 20 \\ = 200$$

~~size of a[10][20] = 200 \times 4 = 800 bytes~~

Dependent  
on machine

= 800 bytes

Constant array  $\Rightarrow$

Either one dimensional or multidimensional array can be made constant by starting the declaration with const.

Ex:  $\text{const int arr[10] = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}}$

$\downarrow$   
We can't change any value

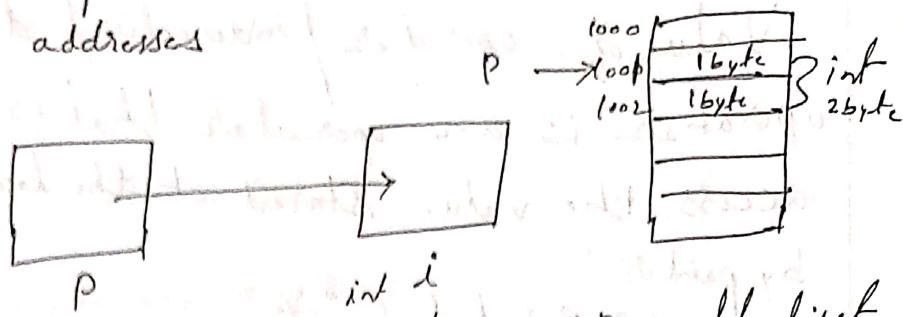
$a[1] = 15; // \text{error}$

### Advantage

- if values are valuable then const is used.

## Defination:=> Pointer

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



If  $p$  points the memory location where the first byte is stored.

$$p \rightarrow 1001$$

Declaring & initializing =>

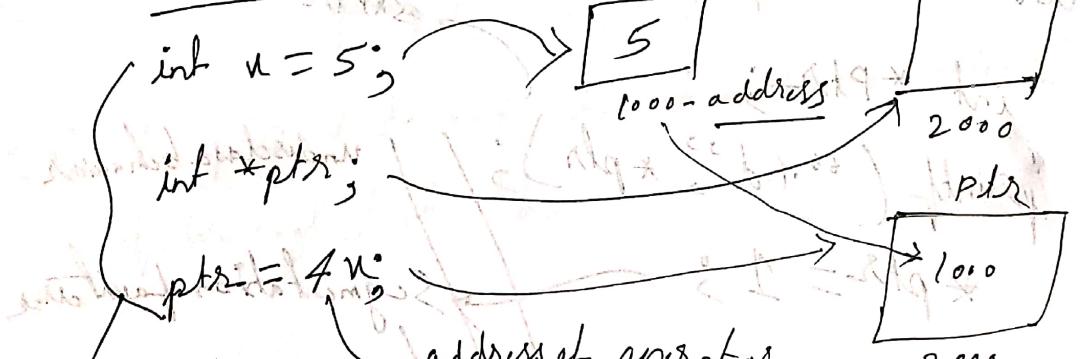
General syntax → `datatype *pointername;`  
 (Refers to type of value that the pointer will point to)

e.g. `int *ptr` →  $\text{ptr}$  points to integer  
`char *ptr` →  $\text{ptr}$  points 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



equivalent

`int n=5, *ptr=4 n;`

$\boxed{\text{ptr - address}}$   
 $\boxed{\text{*ptr - value of the address}}$

## Dereference of operator (Value of operator $\Rightarrow$ )

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 n = 5;
```

```
int *ptr;
```

```
ptr = &n;
```

```
printf("%d", *ptr); // 5
```

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

```
int n = 10;
```

```
int *ptr;
```

```
ptr = &n;
```

```
*ptr = 9;
```

- Never apply dereferencing operator to the uninitialized pointer.

```
int *ptr;
```

```
printf("%d", *ptr);
```

```
*ptr = 1;
```

as \*ptr is not initialized

undesirable behavior

Segmentation fault

trying to read illegal memory location.

## Pointer assignment to another Pointer:

(1)

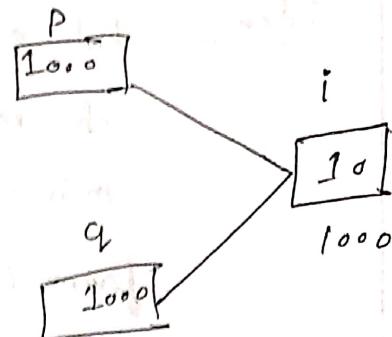
int i = 10;

int \*P, \*q; // declaring pointer

P = &amp;i; // initialize pointer P

q = P; // q is assigned address of variable i

printf("%d %d", \*P, \*q); // 10 10



(2)

int i = 10, j = 20;

int \*P, \*q; // declaring

P = &amp;i; // initializing

q = &amp;j;

\* q = \*P; // 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 = &amp;i;

q = P;

\*q = 5;

printf("%d", \*P);

As q is not declared then it will give error  
 if we neglect it then value of P will be 5

## (Q1)

### Finding Largest & Smallest Elements in an array

Idea -  $\text{int } a[7] = \{ 23, 45, 6, 28 \};$

$\text{int min, max;}$   
 $\text{min} = \text{max} = a[0];$        $\text{len} = \frac{\text{Sizeof}(a)}{\text{Sizeof}(a[0])}$

$\text{for } i = 1 \text{ to } 3$

$\text{if } a[i] < \text{min} \text{ then}$

$\text{min} = a[i]$

$\text{if } a[i] > \text{max} \text{ then}$

$\text{max} = a[i]$

#### 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)
```

~~$\text{xmin} = a[i]$~~

```
if (a[i] > *max)
```

~~$\text{xmax} = a[i]$~~

```
int main()
```

```
{ int a[7] = { 23, 45, 6, 28 }; }
```

```
int len, min, max;
```

```
len = sizeof(a) / sizeof(a[0]);
```

```
minmax(a, len, &min, &max);
```

```
return 0; printf("%d = min, %d = Max", 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\n", *mid); // 3 (C1P)
    return 0;
}

> as we are returning pointer/address
int *findmid (int a[], int n)
{
    return &a[n/2];
}

```

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

```

int *fun()
{
    int i=10;
    return &i; // function returns address
            // of local variable
}

```

After returning local  
var is destroyed.

```
int main()
```

```

{
    int *p = fun();
    printf("%d\n", *p);
    return 0;
}

```

(1)  ~~$\otimes$~~   $\otimes$  int  $*p = 4i;$   
it is not an dereferencing operator it is just saying compiler that P is a pointer

$*p = 4i;$  // not valid as  $*p$  is a value  
of an variable where  $4i$  is address, we can't  
assign address to some integer variable.  
 $*p$  has different meaning in different context.

(2) What is  ${}^0(p)$ ?

void fun (const int \*p)

{  
 $*p = 0;$  }

Should same

int main ()

{  
const int i = 10;

fun (&i); } // suffix to

return 0; }

$\Rightarrow$  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`

`0x7fffd5b2a08c`

3

(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) `x.i`
- (e) `*i`

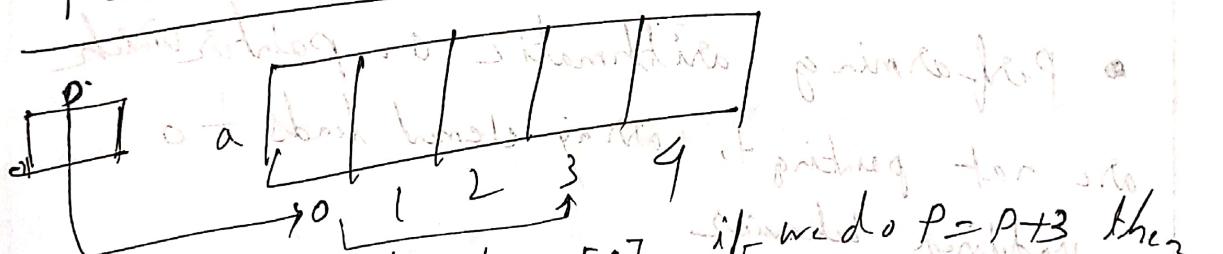
(a) - (e) ✓

⇒ `int i=10;` } - given

`int *p=&i;` aliasing not happening now

`*p = 10, &i = p → *i = *p = i is also ok`

Pointer Addition



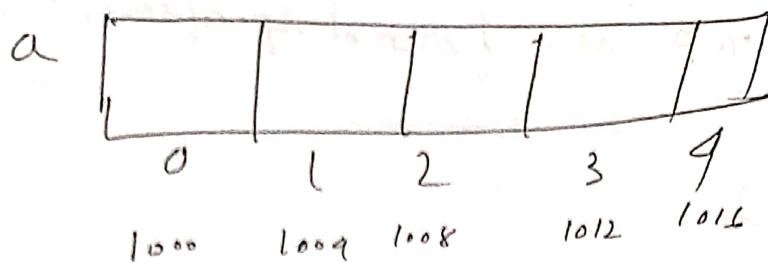
Pointer `P` is pointing to `a[0]` if we do `P=P+3` then

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  $\rightarrow P \rightarrow a[i]$

$$P = P - j \equiv fa[i \oplus i - j]$$

$P = P - j$  means j shift towards left from initial address

Subtraction between two pointers

$$*P - *q = \frac{4P - 4q}{\text{size of the datatype}}$$

- Performing arithmetic on pointer which are not pointing to array element leads to undefined behaviour

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]; as post increment  
first value is

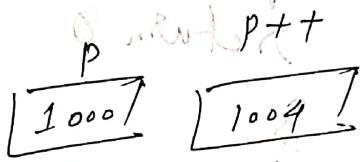
printf("%d\n", \*(p++)); printed  
then incremented

printf("%d\n", \*p); p → a[1]

return 0;

3

⇒ 5 16



$$31 = \text{FLD} = 9 * 5$$

### Comparing Pointer

- Only possible when both pointer point to same array
- Use relational operators & equality operator
- Output
  - True → p > q
  - False → p < q

int main()

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

int \*p = &a[3], \*q = &a[5];

printf("%d\n", p < q); // q is larger address

printf("%d\n", p > q); → p - 1  
→ q - 0

(133)

H/w o/p of the following program

```
int main()
```

$\{ \text{int} a[] = \{ 5, 16, 7, 8, 9, 45, 32, 23, 10 \} \}$

int \*p = fa[1], \*q = fa[5];

printf ("%d", \*(P+3)); (a) 45 + a ()

printf ("%.d", \*(q-3)); (b) 954711

printf("1.d", q-p); ⑥ 447410

printf ("%.d\n", p->a); ④ 457410

print( "d<sup>2</sup> \* p2 \* q) : ) - phisq

printf ( " %s \n", name ) ;

return 0

3 300-1

$$\Rightarrow x_p = 4 \cdot (-1) = 16$$

$$x(p+3) = 4a[4] = 45.$$

2

$$*(q-3) = 4a[2] = 7$$

$$q - p = \frac{1020 - 1009}{9} = 4$$

$P < q - \text{loop} < 1.20 + \text{true} - 1$

$$*P < *q = 16 \angle 32 \quad \text{true} - \underline{1}$$

## Sum of array using pointer.

int main()

$\sum \text{int } a[5] = \{ 11, 22, 33, 44, 55 \};$

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]; }

|      |      |      |      |      |   |
|------|------|------|------|------|---|
| 10   | .    | .    | .    | .    | . |
| 1000 | 1004 | 1008 | 1012 | 1016 |   |

x = &a[0];

printf ("%d", a[0]);

(ans 10) but return 0's

[if  $*(\text{a}+1) = 10$  then  $a[1] = 10$   
 if  $*(\text{a}+i) = n$  then  $a[i] = n$ ]

Caution - Assigning new address to array name will cause error

int a[3] = { 1, 2, 3 };

printf ("%d", a++);

|      |      |      |
|------|------|------|
| 1    | 2    | 3    |
| 1000 | 1004 | 1008 |

$1000 = 1000 + 1$

## (13)

### Passing array as an argument, $\Rightarrow$

(\*)  $\text{int } a[3] = \{1, 2, 3\}$  is an array, and  
it represents the address of the 1st element  
(like pointer behavior)  
So we just pass  $\text{add}(a, len)$   
Ex:  $\text{base address}$

```
int add (int b[], int len)
{
    int sum = 0;
    for (int i=0; i < len; i++) {
        sum += b[i];
    }
    return sum;
}
```

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

return 0;

$3 \rightarrow 3 + (1+2) * 1$

$3 + (1+2) * 1 = 6$

original

base address

array elements

2D array printing using pointer.

```
for ( P = &a[0][0] ; PL = &a[nrow][ncol] ; P++ )
    printf ("%d", *P);
```

1.

Consider char a[100][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 bytes add at which = 4040

$$\Rightarrow a[i][j] = \text{BaseAddress} + [(i - 1)b_1] \times NC$$

$$+ [j - 1b_2] \times C \text{ byte}$$

$$\text{Here } BA = 0 \quad b_1 = 0 \quad C = 1$$

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

$$\text{Ans} 4050$$

$$\text{Ans} 4050 \quad \text{number of bytes} = (n+m) *$$

2) O/P of following C code :- base address 2000  
 If n int requires 4 bytes then - 84 F

#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,%u", \*X, \*(X+1), \*(X+2), \*(X+3));

a) 2036, 2036, 2036

b) 2012, 9, 2209

c) 2036, 10, 10

d) 2012, 9, 6

$\Rightarrow n = 2000$  pointer to the first 1D array

|     | 2009 | 2012 | 2015 | 2018 | 2020 | 2021 | 2028 | 2032 | 2036 | 2039 | 2040 |
|-----|------|------|------|------|------|------|------|------|------|------|------|
| $n$ | 1    | 2    | 3    | 9    | 5    | 6    | 7    | 8    | 1    | 3    | 10   |

$n+3$  = pointer to the 9th 1D array

$$\begin{aligned} &= 9 \text{ elements to pass} \\ &= 2000 + 9 \times 9 = 2036 \end{aligned}$$

$*(\mathbf{n+3})$  = value of first element of 9th 1D array

$$= 10$$

$*(\mathbf{n+2})$  = value of first element of 3rd 1D array,

$$= 7$$

$\mathbf{n+3} - 3$ rd element after 7 = 10

① ✓

Short cut after getting 2036 we know

$*\mathbf{m}$  means value so a bit ✓

so answer is 10

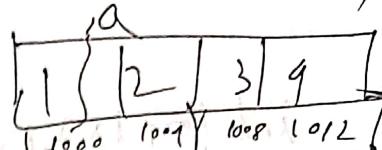
(160)

## Painter Painting to entire array

int (\*P)[5];

of 5 element

pointer to the whole array



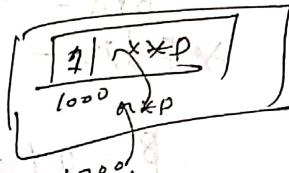
int (\*P)[5] = &a;

means first element address ( $\&a$ )

&a means Whole array address

Printing Value

int (\*P)[5] = &a;



printf ("%d", \*P);

Problem #include <stdio.h>

legal int main()

as we know column

int a[3][3] = {{1,2,3},{4,5,6},{7,8,9}};



int (\*ptr)[3] = a;

printf ("%d %d %d", \*ptr, (\*ptr+1), (\*ptr+2));

+ P[1];

printf ("%d %d %d", (\*ptr)[1], (\*ptr)[2]);

return 0;

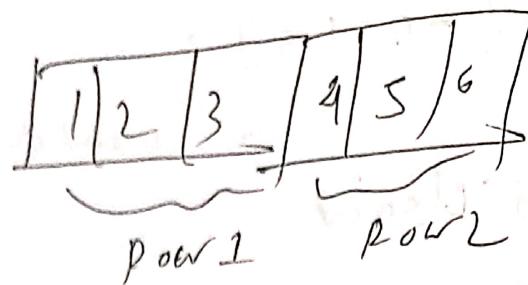
(A) 2 3 5 6

(B) 2 3 9 5

(C) 9 5 0 0

(D) none

⇒ int a[2][3] is Legal as we specify  
Column name.



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

$\ast$  [ptr][1] =  $\ast$  [(\*ptr) + 1]  $\Rightarrow$  gives  
2nd  
ptr — point to 1D array  
 $\ast$  ptr — pointer to 1st element of 1D arr. element

0 1 2 3 4 5 6

$\ast$  ptr — ptr to the 2nd 1D array

⑤ 0 (p →

void f (int \*p, int \*q)

(a) 2 2

(b) 2 1

int i = 0, j = 1; (c) 0 1

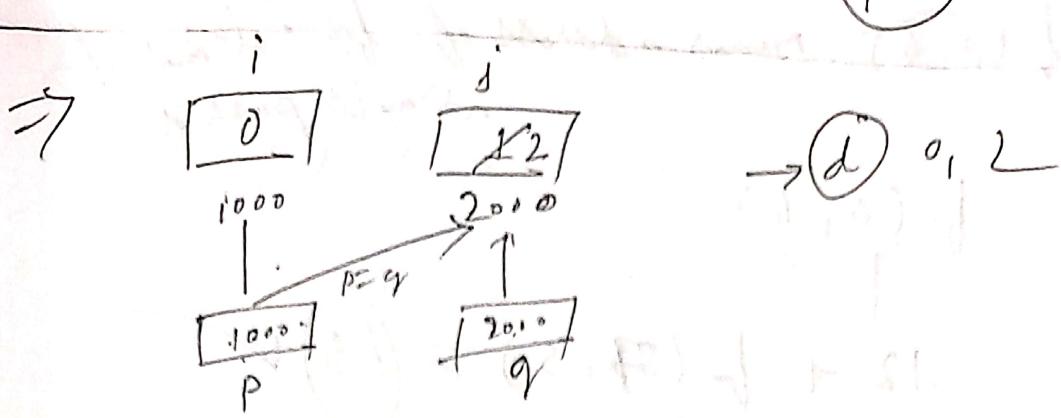
int main () (d) 0 2

{ f (i, j)

printf ("%d %d %d", i, j);

return 0;

3



(b) 0/1 -

#include &lt;stdio.h&gt;

int f(int \*a, int n)

{ if ( $n \leq 0$ ) return 0;else if ( $*a[1] == 0$ ) return \*a + f(a[n-1]);

else return \*a - f(a[n-1]);

int main()

{ int arr[] = {12, 7, 13, 4, 11, 6};

printf("%d\n", f(arr, 6));

getchar();

return 0;

(a) 3  
(b) 5(c) 15  
(d) 19

25 + 8 + 4 = 37

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

$f(a, b)$

$$12 + f(7, 5) = 15 \checkmark$$

(1

3 7 -  $f(13, 4)$

(1 ~~address, value~~

4  $13 + f(9, 3)$

(1 ~~value~~  $f(14, 2)$

5  $11 + f(16, 1)$

6 + 0 ~~return~~

⑦ ~~int \*f(int n, int \*px, int \*pp2)~~

{ ~~int y, z;~~  $b + 1$  } ~~return~~

~~$*pp2 + 1$~~   $\rightarrow$  ~~and also~~

~~$z = **pp2$~~   $\rightarrow$  ~~so what~~

~~$*py + 2$~~

~~$y = *py$~~

~~$n + 3$~~

~~return  $n + y + z$~~

(16)

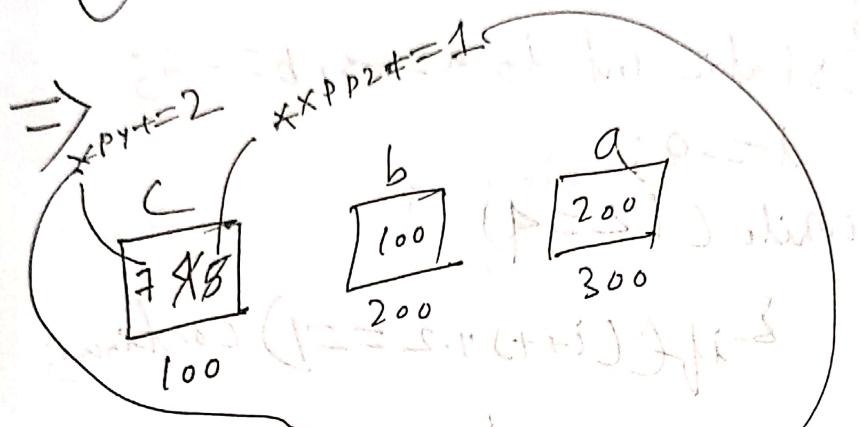
Vuid main yekhane kya hoga?

{ int c = b \* a;

c = 4, b = 4, a = 4 b  $\Rightarrow$

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

(a) 18 (b) 16 (c) 21 (d) 22



$$47 + 7 + 5 = 59$$

Ques. Kuch kaise kare?  $\rightarrow$  10

$$x = 2, y = 0$$

$x = 2, y = 0$  (a) 10

$x = 2, y = 0$  (b) 10

$x = 2, y = 0$  (c) 10

$x = 2, y = 0$

$x = 2, y = 0$  (d)

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 + 1) % 2 == 1 ) continue;

a = a + i;

b = b + i;

30

02

30

sop

Swap ( &a, &b );

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

3

(a) a = 0 b = 3

main()

a = 0 b = 3

{ printab();

(b) a = 3, b = 0

printab();

a = 12 b = 9

(c) a = 3, b = 6

3

a = 3 b = 6

(d) a = 6 b = 3

a = 15 b = 12

166

$i = 0$  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  $i = 3$

$$a = \frac{1}{2}$$

$$\bar{b} = -2$$

i-3

if true       $i = 4$  continue  
~~(loop until i > 10)~~

$i = 1$

$$a = b$$

$$6 = 3$$

Swap

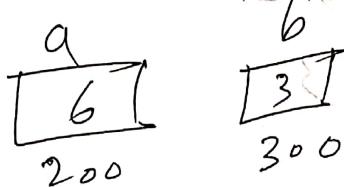


photo Shutter off and back no use as

+ cmp



~~no use~~  
~~this affect~~  
On exp b value

On each val

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];



3. What should be the contents of array

- (a) a b (b) a d (c) a ~~c~~ a e  
c d b e c b d c  
e f c f d f b f

16

a

| row 1 | row 2 | row 3 | row 4 | row 5 |
|-------|-------|-------|-------|-------|
| a     | b     | c     | d     | e f   |

row 1 → row 2 → row 3 → row 4 → row 5



b

| row 1 | row 2 | row 3 | row 4 | row 5 |
|-------|-------|-------|-------|-------|
| a     | d     | b     | c     | e f   |

row 1 → row 2 → row 3 → row 4 → row 5

P → R → S → T → U

row 1 → row 2 → row 3 → row 4 → row 5

row 1 → row 2 → row 3 → row 4 → row 5

P → R → S → T → U

row 1 → row 2 → row 3 → row 4 → row 5

row 1 → row 2 → row 3 → row 4 → row 5

row 1 → row 2 → row 3 → row 4 → row 5

## String handling

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

For ex. "Hello Everyone"

"Saranghae Aishaa"

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");

// work as a big

String.

Instead we will use

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

Because of this no new with next line

In C is called splicing

This Method is not preferable.

Instead what we prefer is

`printf("I.S", "you have to dream before your dreams  
can come true" - Apj Abdul Kalam")`

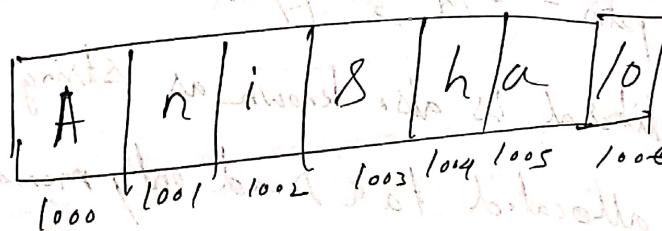
### String String literal.

String literals are stored as an array of character "Earth" automatically added

• Total 6 byte need only memory allocated

\0 (null) is not the same as 0

④ first argument of printf or scanf function is always a string literal.  
Actually we don't pass the whole string instead we are passing a pointer to the first character of string literal.



printf("Anisha")  $\Rightarrow$  passing pointer to letter

(17)

## Assigning string literal to a pointer

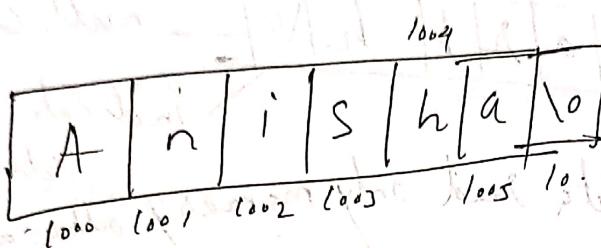
char \*ptr = "Anisha"

ptr points to the first character address

"Anisha" [0] → A

"Anisha" [1] → n

"Anisha" [2] → i



pointer  $^{\text{char}} A[2] = *(\text{pointer to } ^{\text{char}} A + 2)$

$= * (1002) = i$

④ String literal can not be modified.

It caused undefined behaviour.

char \*ptr = "Anisha"; // Segmentation fault

$*\text{ptr} = 'M' ; // \text{not allowed}$

because String literal is also known as String Constant

They have allocated to a read only memory so we cant alter them.

but  $\text{ptr}$  is allocated read/write memory

char \*ptr = "Hello";

$\text{ptr} = "Anisha"$   
 $\text{printf}("%s", \text{ptr}); // Anisha$

(72)

## Difference between string literal & character constant

$'H' \neq 'H'$



Represented by an Integer

(ASCII code - 72)

`printf("Anish");` ✓

`printf("6A");` X

## String Variable

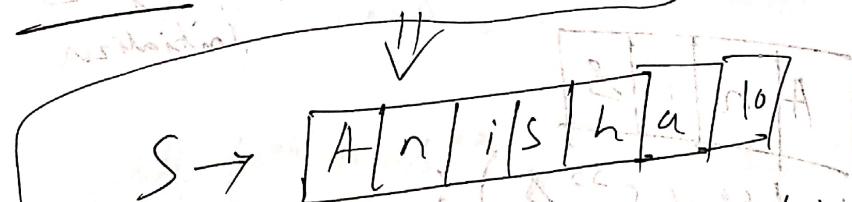
A string variable is a one dimensional array of characters that is capable of holding a string at a time.

For - `char S[7];`

Note - Always make the array 1 character longer than the string.

If we want to store "Anisha" in `S[7]` as last one will be reserved for null character.

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



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 its characters.

char s[7] = "Anisha"; — Preferred

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

both same

- ④ We can't modify a string literal but  
We can modify a char array.

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

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

- ✗ char s[8] = "Anisha"; // short length  
Compiler automatically add null initializer character

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | n | i | s | h | a | l | o | l |
|---|---|---|---|---|---|---|---|---|

- ✗ char s[9] = "Anisha"; // long length initializer

|   |   |   |   |
|---|---|---|---|
| A | n | i | s |
|---|---|---|---|

printf("%s\n", s);

O/p - Anis 2

(\*)  $\text{char } S[6] = "Anisha"$

|   |   |   |   |   |   |  |
|---|---|---|---|---|---|--|
| A | n | i | s | h | a |  |
|---|---|---|---|---|---|--|

- no  $\backslash 0$  at end

it will print Anisha

(\*) but if we want to copy  $S[6]$  to  $G[7]$  then will give undefined behavior.

Omitting the length -

$\text{char } S[7] = "Anisha"$

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

Writing String

(1)  $\text{char } * \text{ptr} = "Anisha"$ , prints

$\text{printf}(" \% s", \text{ptr})$

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

$\text{printf}("% .3s")$ , /A/3 Anisha

$\text{printf}("% .10s")$ , --- Anisha

$\text{printf}("% .13s")$ , 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 writes newline char after writing string

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

`puts(s); } // automatically newline`

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

O/P - Anisha  
Anisha  
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);`

=> You are my love  
(Entered by user)

You (O/P)

The full sentence is not stored as `scanf()` does not store Whitespace char in string var.

It only reads the character other than White spaces. ~~then~~ It stores the character array until whitespace is encountered.

You are my love.  
  ↓  
  whitespace point of failure {3}

② Using `gets()`

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

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

It may crash for big lines as max capacity set to 5 character.

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

Although in `scanf()` we can limit to 9 characters  
`scanf("%9s", arr);` 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) Readin 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;
        else
            break;
    str[i] = '\0';
    return i;
}

int main()
{
    char str[100];
    int n = input(str, 5);
    printf ("%d\n", n, str);
}
```

`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 representing 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 F G H I J K L M N O P Q R S T U V W X Z

Output of program will be

True / false:

- (a) `printf("%c", 'n');` is character
- ✓ 'c' is placeholder for character
- (b) `printf("%c", "n");` reads string
- (c) ~~putchar('n');~~ with box
- ✓ prints char at start afterwards off
- (d) `putchar("n");` (using the string)
- ✓ string X: do this
- (e) `puts("n");`
- ✓ char X
- (f) `puts("n");`
- ✓ do what
- (g) `printf("%s", "n");`
- X
- (h) `printf("%s", "n");`
- ✓

## C String library ⇒

<string.h> library contains 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 (+1 char) to the destination (char array)

Ex - char str1[10] = "Hello";  
char str2[20] = ;

printf ("%s\n", strcpy(str2, str1));  
returns the copied string

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

O/P - Hello  
Hello

(x) strcpy (str3, strcpy (str2, str1));  
✓ possibl. str2

(\*) strcpy does not check whether source will fit or not in the destination

if length of destination  $\leq$  length of source then it will give undefined behavior

To avoid this we call strncpy function

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

char s1[6] = "Hello";

char s2[9];

strncpy(s2, s1, sizeof(s2));

printf(s2); // Hello

strncpy(s2, s1); // undefined behavior

strncpy will not give the destination without null character as source

$\Rightarrow$  destination

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

Manually add

## (2) String $\Rightarrow$ 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 In

- `char *str = "Anisha";` and  
`printf("%d", strlen(str));` is // 6
- `char str[10] = "Anisha";`  
`printf("%d", strlen(str));` is // 6
- `char str[100] = "Anisha";`  
`printf("%d", strlen(str));` is // 6

Why because it calculate length of the string.  
 str has null character except Anisha.

Revised handout

### (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\n", s1);`

$\Rightarrow$  Saranghae-Anisha

(X) Cautio,

if size of s1 is not enough to accomodate s2 then it caused undefined behavior.

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[10];

strcpy (str1, "He");

strcpy (str2, "Mo");

strncat (str2, str1, 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

(185)

When the first character in  $s_1$  &  $s_2$  are same and  $(i+1)$  character of  $s_1$  is less than  $i^{\text{th}}$  character of  $s_2$

`char *s1 = "abcd";`

`char *s2 = "abcc";`

`if (strcmp(s1, s2) < 0)`

`printf("%s is less than %s");`

`else`

`: printf("%s is equal or greater  
than %s");`

`return 0;`

Two dimension array of string:

`char fruits[2][12] = {"2 oranges",`

`"2 Apples", "3 bananas", "1 pineapple"}.`

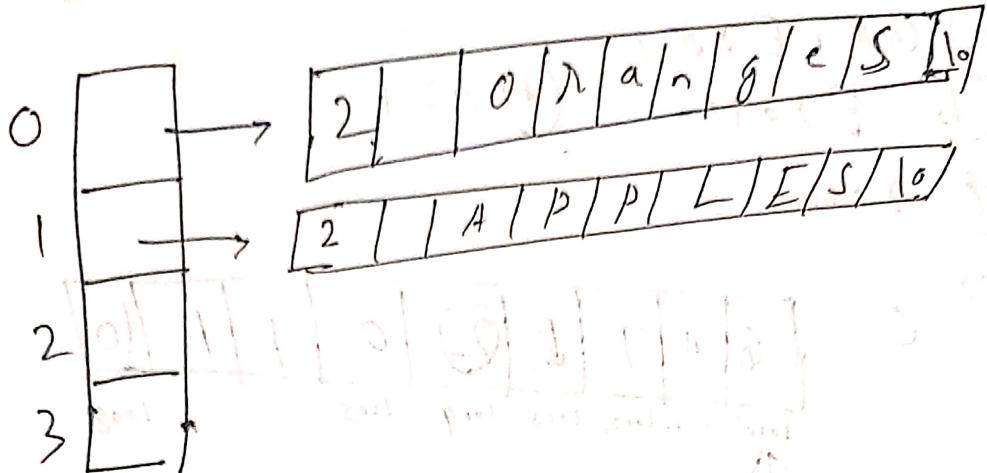
|   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | o | r | a | n | g | c | s | b | t | o | l |
| 2 | A | p | P | l | c | s | l | o | l | o | l |
| 3 | B | a | n | a | n | a | g | l | o | n | o |
| 1 | P | i | n | c | a | p | p | l | c | l | o |

A lot of space is wasted.

(186)

Instead of we use pointer of array

`char *fruit[] = { "2 oranges", "2 apples",  
"3 bananas", "1 pineapple" };`



①

`char p[20];`

(a) works

`char *s = "string";`

(b) String

`int length = strlen(s);`

(c) work

`int i;`

(d) No op

`for (i=0; i < length; i++)``p[i] = s[length - i];``printf("%s", p);`

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| S | T | R | I | N | G | L |
| 0 | 1 | 2 | 3 | 4 | 5 | c |

`So length = 6`

$$\Rightarrow P[0] = S[6-0] = S[6]$$

$$P[0] = \text{\\0}$$



(\*) We know `printf` prints until it gets '\0' null value.  
So no pop will be produced.

②

char c[7] = "GATE2011";

char \*p = c;

printf("%s", p + PE3 - PC17);

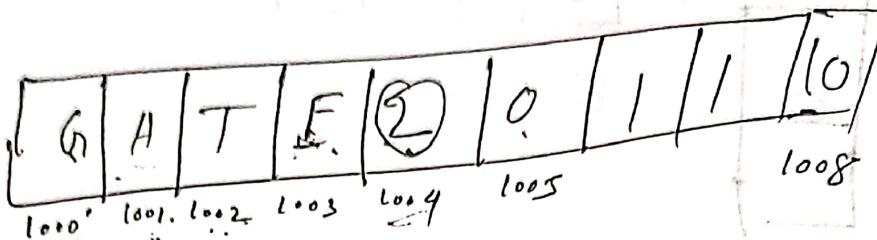
a) GATE2011

b) E2011

c) 021

d) 2011

? c



$p \rightarrow$  C - indicate address of first char.

$p + PE3 - PC17$   $\rightarrow$  Go to index 3 and

access the character value.

$= 1000 + E = A$   $\rightarrow$  Integrally ASCII cod.

$= 1000 + 65$   $\rightarrow$   $65 - 69$   $\rightarrow$   $69$

$= 1009$   $\rightarrow$   $1009 - 1000 = 9$

printf("%s", 1009);

So it saying start printing until it gets

\0.

$2011 \rightarrow 1002 - 1000 = 2$

$01 = 5019$

Q) The output is if input is given

"ABCD\_EFGH" is

Void foo (char \*a)

{  
if (\*a & \*a+1 == 'D')

{  
foo(a+1);

putchar(\*a);

(a) ABCD EFGH

(b) ABCD

(c) HGFEDCB

(d) DCBA

$\Rightarrow$  a contain the address of A

\*a - means value of the address pointed  
only false when null character is available

ff - returns true when both true  
else false

\*a != ' ' - true when value is space

1st iteration -  $(A \neq A \text{ is Space})$  - true

{  
foo(a+1);

now foo is getting address of B

Same happen

if will be false when - space occurred

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 () {  
    3

{  
    char \*str1 = "Hi", \*str2 = "Bye";

    fun1(str1, str2);

    printf (" %s %s ", str1, str2);

(190)

fun2 (str1, str2);

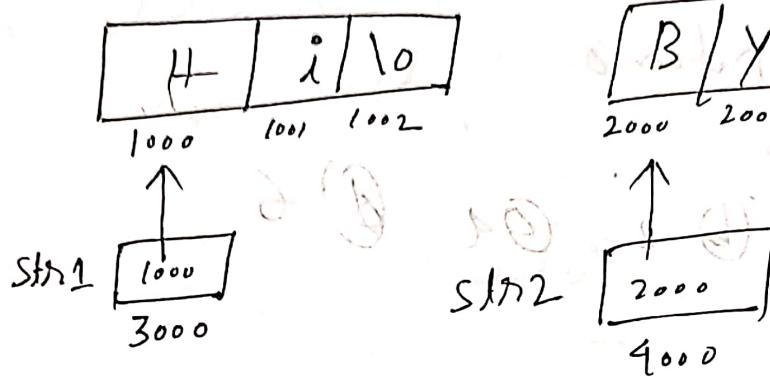
printf ("%.1s %.1s", str1, str2);

return 0;

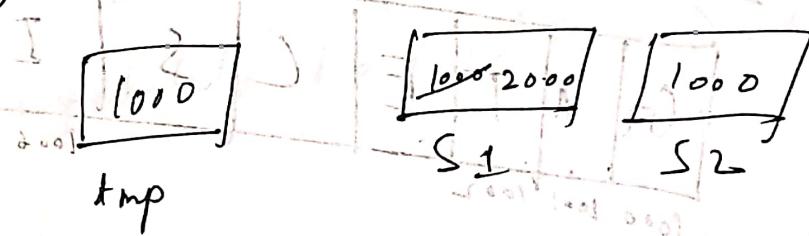
3) ① Hi bye ; bye hi ② Bye hi Hi bye

③ Hi bye ; hi bye ④ Bye hi Bye hi

⇒

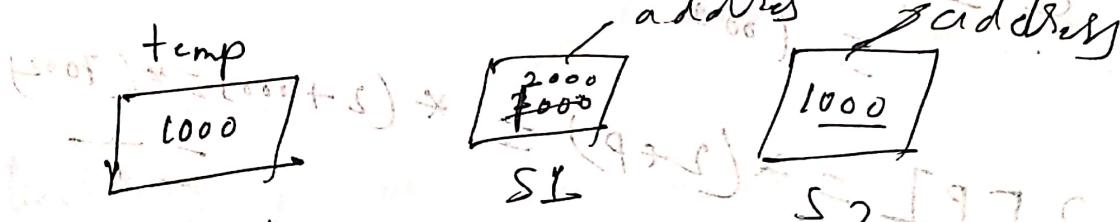


fun1 (1000, 2000)



first printf — Hi bye.

2nd printf : fun2 (3000, 4000)



So, string is swapped.



(101)

⑥ int main()

{ char \*c = "GATECSIT2017";

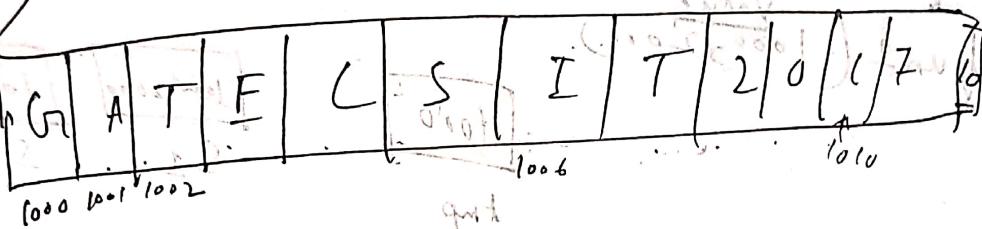
char \*p = c;

printf("%d", \*(int \*)system(c + p) - 6[P] - 1);

Return 0

3

① 1 ② 2 ③ 4 ④ 6



P = ~~char \*P; P = &c;~~  
~~c = &c; printf("Address of 1st element~~

$$2[P] = *(2 + P) = *(2 + 1000) = *(1002)$$

$$\text{as } P[2] = *(P + 2)$$

$$6[P] = *(6 + P) = *(1006) = 10^2$$

$$1000 + 7 - 1 = 1006$$

$$= 1000 + 11 - 1$$

$$= 1010$$

(192)

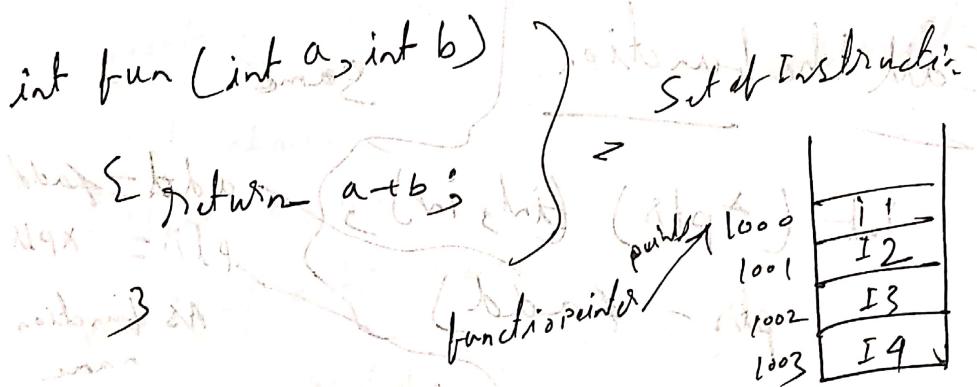
strtol(1010) → strtol starts to count until null char encounter

strtol(str) it point to address of 1st char  
→ 1 7 10 → ② ⑥  
    2 char

We know strtol return size\_t so far  
+ explicit type cast we are using (int)

## Function Pointer

Function pointer are like special pointer that have capability to point to a function.



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 function to a function pointer

`int add(int a, int b);`

{ return type } Argument  
`int (*ptr)(int, int)`

= `add`

Assigning address

Call the function

`int (*ptr) (int, int);`

`ptr = add`

`add = add`

`ptr = &add`

As function

name  
Point address

`result = *ptr(10, 20)`

`printf("%d", result)`

1/30

## Application of function pointer.

(124)

If we want to call a particular function  
In runtime we use function pointer.

earlier in calculator program we use switch  
case but we can use function pointer also.

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

```
#define ops 3
```

```
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; }
```

```
float divide (float a, float b) { return a/b; }
```

```
int main()
```

```
{ float (*ptr[ops]) (float, float)
```

```
ptr = {sum, sub, mult};
```

```
int choice;
```

```
float a, b;
```

```
printf("enter choice\n");
```

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

```
printf("enter 2 numbers\n");
```

```
scanf("%f %f", &a, &b);
```

```
scanf("%f", &choice);
```

```
printf("%f", ptr[choice](a, b));
```

```
return 0;
```

```
}
```

## Structure $\Rightarrow$

(155) 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 100 Engine - DD 180

Fuel - Petrol Fuel - Diesel

Fuel Capacity - 37 Fuel capacity - 32

Seating n - 5 Seating n - 6

So in order to store all this we have to create diff variable. Suppose you have more than 100 car then time & space will be wasted.

car1

char \*car1Engine = "DD 100";

char \*car1fuel = "Diesel";

int fuelcapacity = 32;

car2

char \*car2Engine = "DD 180";

This is so time consuming process.

(\*) 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;
}
```

global Scope → Every function can access

→ car1, car2 → object / variable

Different datatype declarations,

Ex :-

```
Struct {
    char *engine;
}
3 car1, car2;
```

int main () {
 // input part
 car1.engine = "DD 150";
 car2.engine = "DD 180";
}

With the help of . operator we can access member of structure.

car1.engine = "DD 150"

car2.engine = "DD 180"

printf ("%s\n", car1.engine);

printf ("%s\n", car2.engine);

// Output : DD 150      DD 180

## Structure tag

(137)

Structure tag is used to identify a particular kind of structure.

struct structurename

{

~~variables~~ Different data type declarations

}

Later we can create Variable

struct ~~as~~ structurename manager;

struct :: structurename comp;

Using type def - gives freedom to create own type / renaming type.

typedef existingdatatype newdatatype;

For ex - typedef int Arijit;

int main()

Instead

int

we can

use

Arijit

Arijit var = 100;

printf("%d", Val);

1100

(158)

In Structure also we can do it  
~~old name~~  
 typedef struct car { } new type name

3 car;   
 new type name

struct car c1;   
 Use same

Car c1 ✓

Initialization & accessing Struct members

Struct abc

{ int p = 23 ; }

int q = 24 ;

};

Struct abc { algorithm priors main ()  
 { int p ; int q ; }

Struct abc , n = { 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 main() Same order
{
    int x;          {
    int y;          } struct abc a={10, 20, 30}
    int z;          { Same
                    } x=10, y=20, z=30
}
3:
int main()
{
```

Designated  
initialized

• Varname = Value

struct abc a={.z=30, .y=20,  
.x=10}

## 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

{[0], {[1]} same}

Variables

[2]:

## Accessing members of Structure Pointers

(200)

struct abc

{

int x;

int y;

};

int main()

{

struct abc a = {0, 15};

struct abc \*ptr = &a;

printf("%d %d", ptr->x, ptr->y);

ptr -> y; (X)

ptr is a pointer to some  
variable of type struct abc

ptr contains address of a

3 (address, value)

1 (and c1 is 316)

ptr->x is equivalent to (\*ptr).x

## Structure Padding

When an object of some structure type is declared then some contiguous block of memory will be allocated to structure members.

struct abc

the size of struct → size

(\*) Processor doesn't read 1 byte at a time

char a; // 1 byte

byte at a time

char b; // 1 byte

it reads 1 word at a

int c; // 4 bytes

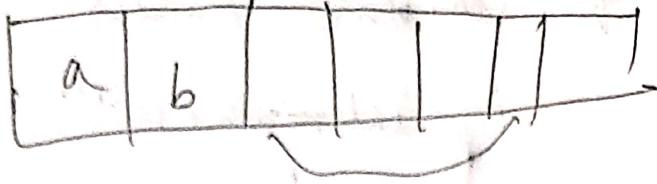
time

3 vars

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 then n n n n n n n n

8 bytes at a time n n n n n n n n is 8 byte

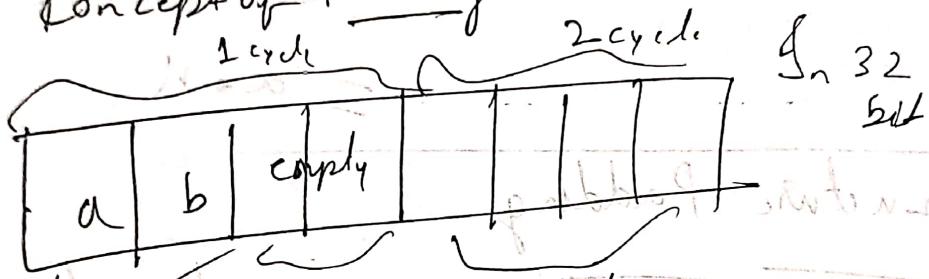


In 32 bit architecture. Char a, char b,  
2 bytes of int can be accessed in 1 CPV  
cycle.

So 2 cycle is required to access all.  
(char, char, int)

Its an unnecessary wastage of CPV cycle

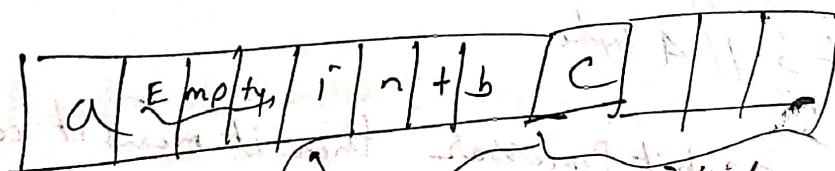
So concept of Padding came into picture



$$\begin{aligned} \text{Total no. of bytes} &= 2 + 4 + 2 (\text{empty}) \\ &= 8 \text{ bytes (In 32 bits)} \end{aligned}$$

If we change order of variable.

Want to fit char a; Total = 12 bytes  
int b; Total = 4 bytes  
char c; Total = 4 bytes



1 cycle 2 cycles

## Structure Packing $\Rightarrow$

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  $\Rightarrow$  Op is - 12 as structure padding occurred

{ char a; but it should be 6

int b;  $\Rightarrow$  4

char c;  $\Rightarrow$  1

3 vars

int main()  $\Rightarrow$  4

printf(  $\Rightarrow$  size of (val) ) ;

There come concept of packing below code  
We can avoid the wastage of memory by simply writing #pragma pack(1)

Special purpose directive used on or off certain features.

If we write #pragma pack(1)

then o/p will be 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; } . struct point p1 = { .y = 0, .z = 6,  
 .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 initialization.

So answer 2 0 1 (c) ✓

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%c", \*(char\*q), \*(char\*q+1), \*(char\*q+2));

ans - (a) 0, <

(b) 0, a+2

(c) 6, a+2

(d) 6, <

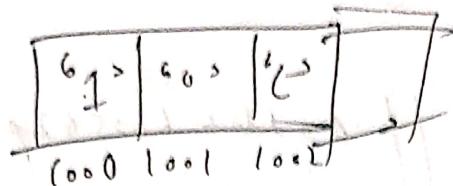
209

$${}^6 \text{a}^2 + 2 \rightarrow C$$

$$W = {}^6 I^3$$

$$Y = {}^6 O^3$$

1. c means prints  
char  ${}^1 I^3$  not  ${}^6 I^3$



a byte alloc

q contain address of whole structure

$*((\text{char}^*)q + 1)$  — Particular character

$$q + 1 - * (1001) \rightarrow {}^6 O^3$$

$$- * (1002) \rightarrow {}^6 C$$

$$O, C \rightarrow @$$

Define S to be

(3) Struct node

{  
int i;  
float f;  
};

(a) an array, each element

is pointer to structure of type node

(b) A structure of

Struct node  $S[10]$ ; 2 field

(c) A structure of

3 field

(d) May, each element

is a structure of type node

$\Rightarrow$  S is not a structure  
but a

each element not a structure of type node

T is a pointer

(a)

Unions  $\Rightarrow$  Union is a user defined data type but unlike structures, union member share same memory location

Struct abc

```
{ int a; // address 6205624
  char b; // b address 6205628
}
```

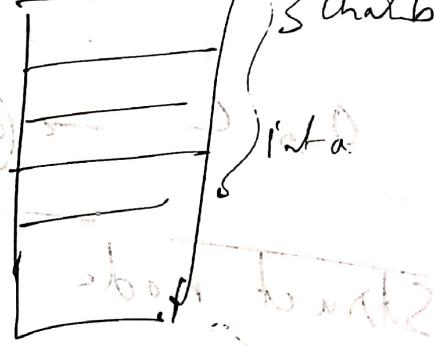
Union abc

```
{ int a; // both address
  char b; // 6205616 }
```

Struct abc



Union abc



\* Union member share same memory location. If we change it affect others

union abc

```
{ int a;
  char b; }
```

3 var's

int main()

```
{ var.a=65;
```

a = 65

b = A

```
printf("a=%d b=%c", var.a, var.b);
```

(\*) Size of union is taken according to the size of largest member of the union.

Union abc

|           |                    |                        |
|-----------|--------------------|------------------------|
| int a;    | 119 byte           | size of (union<br>abc) |
| char b;   | 111 byte           |                        |
| double c; | 118 byte           |                        |
| float d;  | 114 byte (Highest) |                        |

3;

(\*) We can also access member of union through pointers using ~~arr[4]~~<sup>3-512</sup>

Var. a = 90;

union abc  $\Rightarrow$  p = 4 Var;

printf("%d.%d.%d.%d", p->a, p->b,

(1)

struct {

short s[5];

Assume

short - 2

union { char d; long - 8 }

float e; float - 4 byte

long f; long - 8

float g; float - 4 byte

float h; float - 4 byte

alignment const derivation

(a) 22 (b) 19 (c) 18 (d) 10

20+

short or s array - 10 byte

union → longest type is long → 128 byte

→ ① [8 byte] + 10 byte

## Application of Union $\Rightarrow$

A store sell 2 item

(a) books (b) shirt,

Property

Title

Author

number of Pages

price

Color

Size

design

So they made a structure

struct store

{ double price; // 8 byte

char \* title; // 8

char \* author; // 8

int num\_Pages; // 4

int color; // 4

shirt { int size; // 4 byte  
char \* design; // 8 byte }  
3;

struct store book; // 94 byte  
memory

There is so much wastage of memory  
like we don't 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 numPages;

} book;

struct

{

char \*color;

int size;

char \*design;

short;

} item;

} ;

int main()

{ struct 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 = 0.178;
}
  
```

If we use struct or it will consume

more memory

as we know at a time in a array element  
there could be only one value.

## Enumeration $\Rightarrow$

It is a user defined type ~~which is~~ which is used to assign names to integral constant because name are easier to handle

We can also do it by ~~typedef~~ ~~int~~ ~~define~~ but it is ~~in~~ on global scope but enum can be used locally also. and enum automatically initialized

// enum bool {False, true}

int main()

```
{
    enum bool Var;
    Var = true;
    printf("%d,%d", Var);
}
```

return 0;

If we do not assign value to enum names then compiler will assign value starting to 0 and incremented 1 gradually.

enum { a, b, c, d };

a=0, b=1, c=2, d=3

c value will be b value + 1

enum { a=0, b=0, c=0 };

enum { a=0, b=2.5, c=9 }  $\rightarrow$  no float value  $\times$  only integer

(271) (X) All enum constant must be unique  
in their scope

enum a  $\{x=2, y=3\}$  → X  
enum b  $\{x=3, y=3\}$  /  
redefinition not allowed

① Struct

{ double d; } local union  
union

{ char b[4]; }

double c;

char d[4]; } e;

char f[4]; }

how much space to allocate for  
S (no padding)

$\Rightarrow$  double - 8

char -  $2 \times 4 = 8$  - 20

Union - double - 8

→ total buffer size = 20 bytes

→ 10 bytes for class definition

## Preprocessor Directive $\Rightarrow$

Preprocess is a program that process the Source code before it passes through the compiler

Sourcecode Preprocess  $\rightarrow$  Expanded Sourcecode

- (\*) It operate under the control of preprocessor directives which began with the symbol

#. Macro Substitution

types file inclusion

Compiler Control

### (i) Macro Substitution

It replaces every occurrence 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 definition

`#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 LINE == 1  
 printf("Hello world");  
 #else  
 printf("Hello world 2");  
 #endif  
}

Suppose in a big program we does not want to compile or run a segment of program

What we do we commented those lines of program  
If there is a lot of line it takes a lot of time to comment or Delete the comment

#define VER1  
int main()  
{  
 #if defined(VER1)  
 printf("This is demo program");  
 #endif  
}

If VER1 is defined then it will not compile if not define it will compile.

(24)

Command line argument  $\Rightarrow$  It is possible to pass sum value from the command line to your C Program when they are executed. These values are called Command line argument.

We have to pass Parameter ~~points to array of strings~~ int main (int argc, char \*argv[])

argc - arg counter  
argv - arg value

argv[0] is path of exec file  
So if no command line argument given then  
argc = 1 (as path is an argument)

File handling in C  $\Rightarrow$  File

We need to create file pointer

prototype - file \* fopen (file name, mode);

example - file \* ptr;

fopen - used for open a file in different modes (Read, Write, Append)

prototype - fopen ("filepath", "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 functions belong to stdio.h

`<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 = fget(fp)) != EOF)

{

printf("%c", ch);

}

(\*) FILE is also ~~datatype~~ structure

which has its own pointer

Void Pointer :-

int \* p

float \* q

Some void \* t

Generic pointer it can point to the any type

Value & address

int n

of = 4n

float y

We can not directly write \*t to represent value of  
\*t

We have to typecast

(~~int~~\*) \* (int\*) t

\* (float\*) t

Null Pointer  $\Rightarrow$

int \*P=NULL; Macro (already defined)

int n =

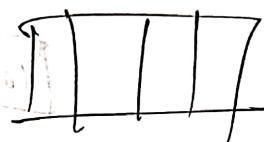
base address



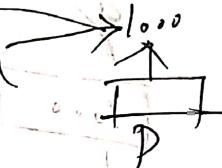
Such a location it points as if it is pointing nothing (empty location Point)

Wild Pointer

int n =



int \*P=4n



but if we only write  
illegal memory over  
\*P=q;  $\Rightarrow$  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 ~~void~~\*fun()

{ int n = 5;

return &n;

}

A garbage address

if ~~n~~ n is static then it doesn't become dangling

void <sup>goes out of block</sup> main

{ int \*ptr;

= ;

{ int ch;

ptr = &ch;

}

// Here ptr is dangling pointer

3