C - PROGRAMMING

Data Types	Size	Range (2's comp. form)
int	2B/4B	-2^{15} to $2^{15}-1$
float	4B	[same form]
double	8B	[same form]
char	1B	-2^{8-1} to $2^{8-1}-1$

* Size of int is undefined [depends on word size of m/c] (Now → 32b compiler) → (4B) → int.

eg → Pg2 [Float]

* Char : integral data type. (imp)
  Range = −128 to 127

unsigned char : range = 0 to 255.

eg → Pg2 [Rev copy] (char x = 65)

* ASCII Codes.

A=65	a=97	0=48
B=66	b=98	1=49
C=67	c=99	2=50
⋮	⋮	⋮
Z=90	z=122	9=57

(imp)

[65, 97, 48].

Little/Big [illegible]

* int a = 10; (int → 2B [16 bit compiler])

text → 0 0 0 0 0 0 0 0 | 0 0 0 0 1 0 1 0
Byte 2 | Byte 1

in memory (imp)

0	10
10	0

Byte 1 Byte 2

Big endian [MSB → LSB]
Little endian [LSB → MSB] (mostly used).

eg → Pg 5 [Imp]

eg → Pg 6 [Applicn of Little endian] (imp)

eg → Pg 6 (char *p)

char *p : p is a char pointer, which stores address of char.
printf("%c", *p) : dereferences value pointed by p, but read 1B. (imp)

eg → Pg 4 [Rev copy] (Cyclic Behaviour)

char Range : (−128, 127) [signed]

eg → Pg 8 [130] (signed char range)

* Operators

1) Data Type of exp. is the largest data type used in expression.
2) x = a + b + c * d; (imp) [depends on comp.]
[which operation will be executed/ which function call worked first → depends on order of execution.

∴ 3 things →
- Precedence
- Associativity
- order of execution (dep. on compiler)

* C was initially implemented in both 16 or 32 bit compiler. [int → 2B/4B]

* Pre/Post increment operator

b = a++ : first store a to b, then inc. a.
b = ++a : first inc a, then store in b

eg → Pg 11 [with a=5, b=4]

∴ Shouldn't use inc/dec operator on same variable, more than 1 time.
∴ b = a++ + ++a ; X. [depends on compiler]
printf("%d %d %d", a++, ++a, a++); X (compiler)

int x=5, y=2, z=7;
fun(x, y, z); → into stack [int z] (imp)
R → L

void fun(int a, int b, int c)
L → R

eg → Pg 7 (Rev copy) (concept)

→ abnormal ... of program.

[Rev Copy]

eg → Pg 8 [void fun(int a)]
[Stack overflow]

* Conditional operator → { <, <=, >, >=, ==, !=, &&, || }

[if returns any value : true, 0 : false]

imp

eg → Pg 15

Cond^n [Test mul]

* Short circuit

a=2, b=3, c=4

if (a>b && a>c) → no need to check 2nd cond^n, if 1st cond is false ∴ false.
F

[In case of ||, no need to check 2nd cond^n, if 1st cond^n is true]

eg → Pg 9 (Rev Copy) [int A[a]];

* Bitwise operator

{ &, |, ~, ^, <<, >> }
AND OR NOT EXOR L-S R-S

- performs on integer datatype (int/char)

eg → Pg 10 (Rev Copy) [a=10, b=3]

imp

Left Shift : a<<n → a * 2^n
Right Shift : a>>n → a / 2^n

used in DSA [Test] (ME)

eg → Pg 19

* Switch Case

eg → Pg 20 [Break not present]
case 1·1 X

eg → Pg 21 [Scope & extent of var]

* Program running Process
[Most OS use dynamic linking/binding]

→ Prog is saved in HD as add.c & compiler converts add.c into add.exe [which is a m/c code].

→ CPU executes .exe file, which is brought to MM. [loaded] (* LTS)

→ loader writes add of functn in place of functn call in Code Section for CPU. Function Binding → compile time

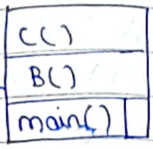
imp

→ Memory of all var: x, y, z; is allocated during run time; & all m/c instructions related to it, is clearly mentioned in m/c code. — data binding → run time

imp

eg → Pg 12 [diag] [Rev Copy]

```
C()          main()
{            {B();
}            }
B()
{ ==
  C();
}
```



when C completes its exec, AR deleted, control moves to the next line of funct call C, in the calling funct B. (imp)

* Recursion

- funct^n calling itself.

Ascend → Tail
Descend → Head (imp)

eg → Pg 13 & 14 (ascend/descend) [Rev Copy]
Head rec. Tail rec.

eg → Pg 28 [diag of AR & stack in recurs]

* Loop V/S recursion

- loop is space effective & faster than recursion.

(imp) Every recursion can be converted into loop & vice-versa [Mathematical formula]

Space complexity : depth of stack

* Tail recursion ↔ similar to loop (imp) [Ascend]

* Types of recursion

(1) Head & tail recursion.
(2) tree recursion
(3) Indirect recursion.
(4) nested recursion → fun(fun(n-1))

```
f(int n)
{
  f(n-1);
  f(n-1);
}
```

eg → Pg 16-17 (Tree Recursion) [Rev Copy] → Total 15 funct calls. (time)

(imp) Time depends on no: of funct calls
Space depends on ht of tree [depth of stack]

* Storage Classes

1. [Auto, Static, Extern, Register]
   Auto → local V.
   Static → Global V.

(1) → Extern :- loader creates it in the code Section, during loading time (b4 exec) [Text Section] (imp)
- globally accessed by all functions.
- lifetime : till prog ends.

eg → Pg 36

Funct binding : loading time
Data binding : run time → mem allocated

→ Stack Memory allocation → Dynamic (AR)
Code Section Mem allac → Static (GV/SV)
happens at loading time.
(imp)

Only memory is dynamically allocated in Stack, but if type & size, needs to be dynamic, go for Heap.

eg → Pg 18 (Rev Copy) [Attributes of Var]

[Value, : run time Address]

② Static

Scope: only that part. funct, where declared can access it.
Extent: throughout, till prog ends
memory inside code Section. (imp)

Class	Scope	Extent
auto	local	local
extern	global	global
Static	local	global
Register	local	local

→ Fastest.

All funct look into their own AR first, if var name cant be found, look into the CS, for global/Static V. (imp)

text

HEAP MEMORY

- like a resource (ext to prog),
- can be accessed indirectly - Pointer.

```
ptr = (int*) malloc (10 * sizeof(int))
ptr = new int[10]
```

[20B]

C++

→ array created in Heap.

eg → Pg 44 [Diag on Heap]

Stack → only mem. dynamically allocated
Heap → fully dynamic (imp)
→ type/size.

POINTER

→ access resources using Pointer only:
[Heap, Socket, file].

```
int x = 10;
int *p; // int Pointer (hold add of int).
p = &x;
```

x: 10 (200/201) p: 200 (250/251)

(imp)

pointer = unsigned int

eg → Pg 49 [Pointer Type Casting]

Size of Pointer = Size of int (Compiler)
→ unsigned int.

* Range of address accessed by Pointer.
if Pointer = 2B = 16 bit
Add. Range = 0 to $2^{16} - 1$ ⇒ 64 KB
∴ [Unsigned].

(imp)

⇒ Data Type of Pointer used during dereferencing time.

eg → Pg 23 [Rev Copy] → if char pointer will dereference 1B only.

* POINTER TO ARRAY

- int A[5] = {2, 4, 7, 10, 12}
  200 202 04 06 08 10.
- int *p;
- p = A ✓

p = A + 1 → 210

- p = &A ✗ : add of array object.

p can be used as array name (imp)

eg → Pg 24 (Rev Copy)

200 02 04 08 10

Negative Indexing:

int A[5] = {2, 4, 7, 10, 12}.
int *p = &A[2]; ← 204 = p
printf("%d", p[-1]); ⇒ 4
printf("%d", *(p-1)); ⇒ 4.

* POINTER ARITHMATIC

→ depends on data type.

p++ ⇒ if p = char, p moves by 1B
p++ ⇒ if p = float, p moves by 4B

① p++ // 202 (imp)
② p--
③ p+3 // 206 [assuming int].
④ p-3
⑤ int d = q - p; // how many item away from each other.

∴ (q - p) / size of dt

eg → Pg 54 int A[5]

imp → precedence

* POINTER TO POINTER [Double P]

① int A[3][4] = { {2, 4, 6, 8}, {1, 3, 5, 7}, {1, 2, 3, 4} }

- All in Stack

eg → Pg 26-27 [All the pfs] (Rev Copy) (imp)

② int *A[3]; A[1] = new int[4];

- Array of int Pointers in Stack, all other array in Heap

eg → Pg 55 [All Qs]

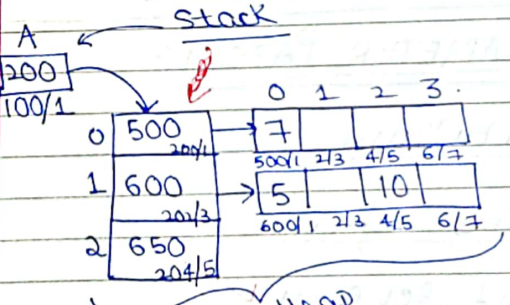
[holds address of pointer]

③ int **A;
A = new int*[3];
A[0] = new int[4];

(fully dynamic array)

(imp)

Only a Pointer in Stack, rest all in Heap. (A)



eg → Pg 55 [All Qs]

* POINTER TO A FUNCTION

Tool

```
void display()
{ printf("Hello");
}
```

```
void main()
{
void (*fp)();
fp = display;
(*fp)(); //func call
}
```

Vimp

eg → Pg 59 (max/min)

int: default agg

eg → Pg 60 [Gate Q]

R → L

eg → Pg 61 [All Points - declaration]

* REFERENCE IN C++

```
int a = 10;
int &r = a;
```

a/r → 10

→ Reference is an alias/nickname of a.
→ won't occupy any extra space.

r++ → a = 11

* PARAMETER PASSING

1. Call by Value
2. Call by Address

eg → Pg 31 (Rev Copy)

A function can access AR of another function indirectly, using Pointers

imp

3. Call by reference → inline functions

eg → Pg 31 [Rev Copy]

[can be used for smaller functions]

* PROBLEMS

eg → Pg 67 (Call) (25) [Extern] (discard)

eg → Pg 69 (Call) / Pg 71 (Call) [if Pointer, leave —]

eg → Pg 73 (Call) [int &x, int c] [if ref. leave]

eg → Pg 75 (Call) [Recursion tracing] : Sum → Local V.

eg → Pg 34 [Rev Copy] [Dynamic Prog]

eg → Pg 79 (Call) / 80 (Call) [Nested recur] → identify pattern (return)

eg → Pg 81 [Vimp]

* Dynamic DS like linked list, trees must use pointers & Heap Space.
* Aliasing: multiple variables having the same memory location. [references]

* Dangling Pointer : If a pointer points to a memory location which is deleted (freed). (imp)

* char *p = "GATE2020";
String literal, goes to Code Section.

DATA STRUCTURES

1) ARRAYS

int A[10] = {5,2,4}; → 10 blocks contiguously allocated in Stack, & 3 will be int.

int *A = new int[10]; // Heap

[1] 1D ARRAY

	0	1	2	3	4
A	2	8	6	9	3
	200/1	2/3	4/5	6/7	8/9

[start address] $L_0 = 200$

Address of A[i] = $L_0 + (i * W)$ (imp)

L_0 = Base Address

i = index

W = Size of data type

[Compiler uses this to access any loc^n in Array]

If array starts from 1.

Add(A[i]) = $L_0 + ((i-1) * W)$

→ [Compiler prefers indexing from 0]
(to avoid more computation)

[2] 2D ARRAY

int A[3][4];
[Will in stack].

	0	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2	a_{20}	a_{21}	a_{22}	a_{23}

[Logical]

In MM

- stored like a 1D Array in MM using row major & column Major.

A : m x n

Add(A[i][j]) = $L_0 + [i * n + j] * w$ (Row Major) (imp)

Add(A[i][j]) = $L_0 + [j * m + i] * W$ (Column Major)

→ Both are equally efficient → (Gate)

eg → Pg

var A[1...10][1...15]; (imp?)

Add(A[i][j]) = $L_0 + [(i-1)15 + (j-1)] w$

eg → Pg 40 [Rev copy] (Vimp)

[Arrays always passed by Addresses]

(imp)

[1st can be left, rest written]

eg → Pg 90 [Sending 2D Arrays]

5/3 4D ARRAYS

int A[D1][D2][D3][D4];

→ L-R

Add of (A[i1][i2][i3][i4]) =

Row Major

$L_0 + [i_1 * D_2 * D_3 * D_4 + i_2 * D_3 * D_4 + i_3 * D_4 + i_4] * W$

← R-L

Add of (A[i1][i2][i3][i4]) =

Column Major

$L_0 + [i_4 * D_3 * D_2 * D_1 + i_3 * D_2 * D_1 + i_2 * D_1 + i_1] * W$

(imp)

⇒ nD Array (Accessing) → $n(n+1) = O(n^2)$ (mul² operns)

eg → Pg 92 [Compiler Q] (imp)

* HORNER'S RULE

- Horner attempts to reduce the no. of multiplications by arranging & taking common.

Normally, n degree polyn. → n^2 mul.

(imp) Horner, n degree polyn → n Mult.

- If highest degree term (n) has coefficients as 1, you need 1 multiplication less. [n-1 mul]

eg → Pg 95 $(x^5 + 2x^3 + x^2 + 1)$ → 4 mult^n

eg → Pg 96 $(2x^7 + 4x^5 + 2x^3 + 5x)$ (Imp) [temporary var.]

* Operations on Array:

1. Insert: [O(1) to O(n)] (last to first)
   Append: O(1)
   Replace/Set: O(1).
2. delete: [O(1) to O(n)] [last to first]
3. Search → Linear Search: O(n)
   → Binary Search: O(log n)

eg → Pg 45 [2 Qs] [Rev Copy]

→ In Sorted Array, whether any element coming half / more than half of array? → O(log n)

* Other operations on Array:

1. Union of 2 Arrays:

A → n, B → n.

[First A is copied, & then checked with B] → $O(n^2+n) = O(n^2)$

ii) Sorted → $O(m+n)$ (Merge) (imp)

2. Intersection
3. difference

] $O(n^2)$

* STRUCTURE & UNION

```
struct rectangle
{ int length;
  int breadth;
};
main()
{ struct rectangle r;
  r.length = 10;
  r.breadth = 5;
}
```

r → [a | b] [4B] — Now mem. allocated.

· `struct rectangle *p = &r;` → struct Pointer. (imp)

`p → length = 20;`

or,

`(*p).length = 20;`

* Used for dereferencing → * (Any pointer) → [If struct type pointer].

· `struct rectangle *p;` (imp)

C++ : `p = new rectangle;`

C : `p = (struct rectangle *) malloc (size of (struct rectangle))`

malloc returns void Pointer (needs to be type casted)

(imp) eg → Pg 48 (Rev copy) (++p → a)

* typedef struct Test (imp)

```
{
} TEST;
main()
{ TEST x;
}
```

→ instead of `struct Test x;`

Structure variable.

* struct [Anonymous Structure]

```
{ int a, b;
} x;
x.a = 10;
x.b = 10;
```

eg → Pg 104 or Pg 49 [Rev Copy]

PREV YEAR QS [INSIGHTS]

- In the 'call by name'; unevaluated expressions are passed to the funct^n & exp are evaluated in this funct^n. (1.4)
→ C doesn't support this, but uses 'thunk' to implement the same.

free(n);
n → value = 5; (1.24) (imp)
→ [n is freed, n is now pointing to an invalid memory → Dangling Pointer]

* Goal of Structured Programming: be able to infer flow of control from program text. (1.37)

* ADT is a data type for which only the operations defined on it can be used, but none else. [1.49] eg: Stack → push, pop.

* Q. 1.42 [Imp Q → Structure] (imp) (Q) (call by Value)

* An unrestricted use of "goto", makes it difficult to verify programs. [1.11]

* When a function is called before its declaration, it leads to compilation errors. [1.52] (Error)

(Must do)

* Q. 1.58 (Imp Struct Q) (Vimp) (Q) → [like Q 1.42] GO

* Reference count is used by Garbage collector to clear the memory whose reference count becomes 0.
- Linking Loader performs Address Relocation.
Location counter used by Assembler to give address to each instruction. (imp) (Must) (Vimp) ME

* 2D Array Good Q → Q 1.60 (Q)

* Any Pointer can cause segmentation fault. It is a common fault of pointer due to illegal virtual memory access.

* Heap Allocation is required for languages that supports dynamic data structures like LL, trees.

* Call by Reference [Good QS] (Q) → 1.71, 1.28 → Gate (ME)

* Local V inside Block QS [Imp] (Q) → Q 1.78 (Gate) (imp)

* Arbitrary go to, recursion or repeat loop, may lead to infinite calls in the Programming Language.

* Good Qs on Global & Static Variable → Q1.113

Q → Good Q on Ascend/descend of recursive function [descending time] (imp)
[don't write the value of static V/ any variable, after call of recursive funct^n, while tracing tree] (imp)

Q → good Q on unsigned Integer → Q.1.124
[If we comp. with unsigned int, then read as default int]

Q → Good Q on
Structure & → Q1.58
Pointers → Q1.42
→ Q1.133

Q Good Q on Memory Leak & Dangling Pointer → Q1.131.
→ Last memory in Help.

* y divides x by repeated subtraction
∴ x = (y × q) + r
if, (q == 0) && (x == r) && (y > 0). (imp)

y → divisor

* Stack overflow causes abnormal termination of the program. [It is totally different from an infinite loop].

* Precedence using Brackets :
Q 1.56. TEST-2019

* int m(int n) (imp)
{ if (n>3)
{ r = 50;
return (r + m(n-1));
}
return (r - m(n-1));
}
m(8)

m(8)
|
r + m(7)
|
r + m(6)
|
r + m(5)

call the function, don't execute next Statement.

* unsigned char : 1B → [0 to 255]
unsigned Short → [0 to $2^{16}-1$]
(2B) ⇒ [0 to 65535]

* If a number is preceded by 0 [eg: 011, 010, 012]; C interprets it as octal number & convert into decimal format (if %d). (imp)