

# Data Structures & Algorithms with C, C++

CS Academy

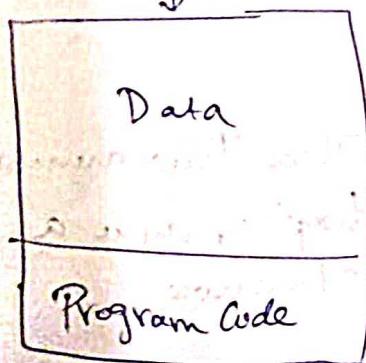
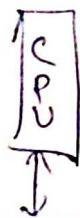
Subramanyam

Physical DS

- Arrays
- Matrices
- linked lists

logical DS

- Stack
- Queues
- Trees
- Graph
- Hashing
- Recursion
- Sorting



memory

## C & C++ Concepts

### Arrays

```
int main() {  
    int a[5]; // Declaration of array  
    int b[5] = {2, 4, 8, 7, 10}; // Initialize array  
    // Declaration // Initialization
```

```

int i;
for {i=0; i<5; i++)

```

printf("%d\n", B[i]);

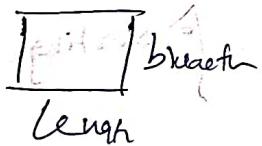
Stack  
main  
Code section

Stack is above  
Heap is below:

Just for understanding  
purposes → This layout

Structure:

→ User defined data types



Struct Rectangle

int length;  
int breadth;

}

These two members  
together define a  
structure.

Size of Structure → Total amount of memory  
consumed by all its members

int main()

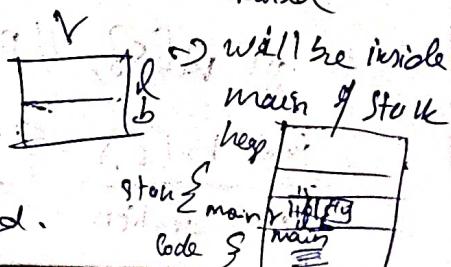
Struct Rectangle r; → Declaration of structure

Struct Rectangle r = {10, 15}; // A memory space will  
be created

r.length = 15;

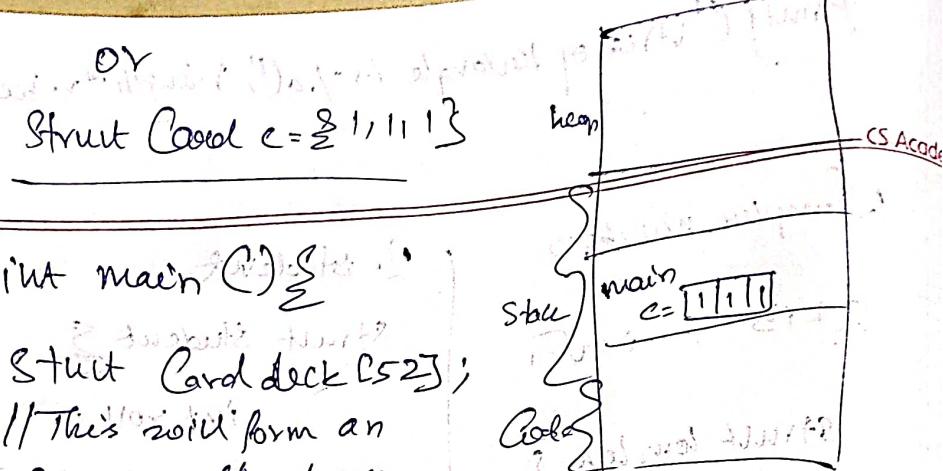
r.breadth = 10;

// To read or write the  
member of a struct,  
dot operator is used.





Struct Const  $c = \{1, 1, 1\}$



```
int main()
```

Declaration Stmt Card deck C52J;  
// This will form an  
array of structures  
(C52 such structures)  
(C 312 bytes if size of

## II Declaration + initialization

→ Strict Carol Deck  $\{S_{5,2}\} = \{S_{11,1}, S_{10,2}, S_{11,3}, S_{11,12}\}$

Printf("!!-!cl!!", deckLogFile);

```
printf ("%d", decklo} - shape);
```

## Pointers:

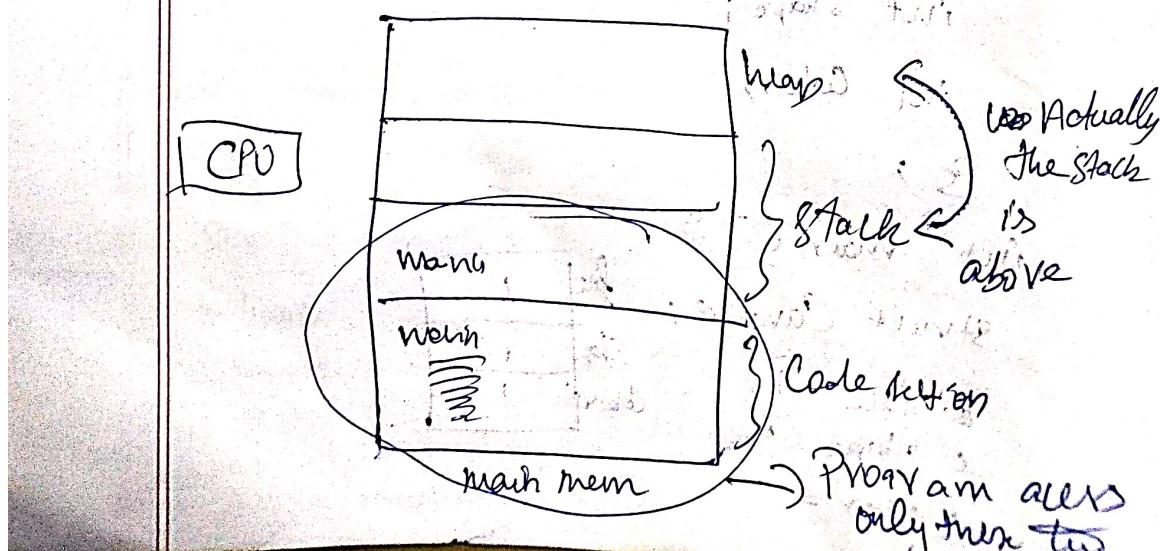
QUESTION

11) Pointers have the address to the Data or not The Data itself !!

"They store the address of the data!"

Normal variables  $\rightarrow$  DATA variables

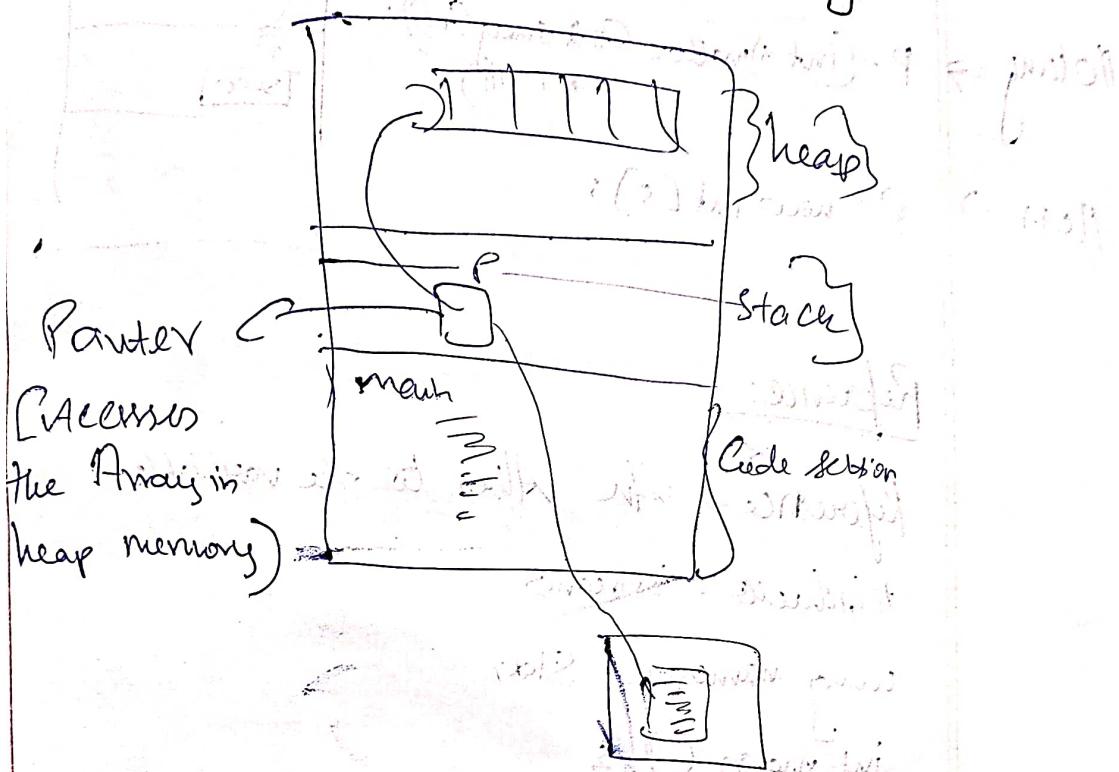
Pointer Variables → Address Variables



o Program would not automatically access heap.

Heap memory  $\rightarrow$  external to the program

One of the reasons of Using Pointers:  
↳ Accessing heap memory



Major Use of Pointers:

1. Accessing heap [With pointers, we access the external files]

2. Parameter Passing

3. Accessing Resources

int main()

int a=10; // Data variable

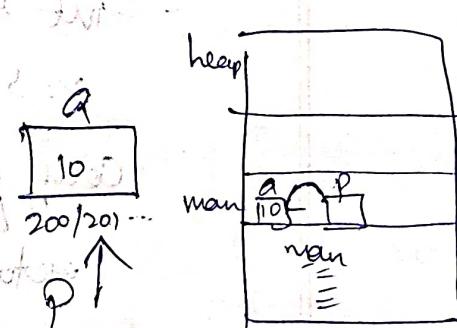
int \*p; // Address variable

P = &a; // And gives the address

printf("%d\n", a); -10

printf("%d\n", \*p);

\*Dereferencing



200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

210

200

`malloc()` → Creating memory in heap

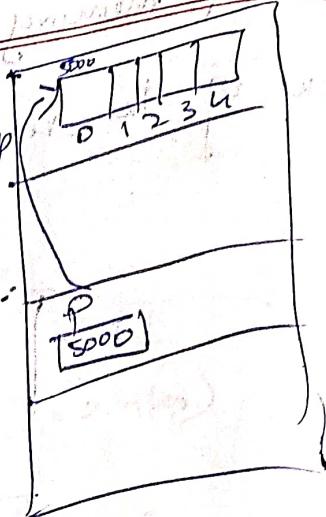
#include <stdio.h>

#include <stdlib.h>

int main()

{  
    int \*P;  
    P = (int \*)malloc(5 \* sizeof(int));

// clang → P = (int \*)malloc(5 \* sizeof(int));  
// C++ → P = new int(5);



### Reference:

Reference → like Alias to the variable

#include <iostream>

using namespace std;

int main()

{  
    int a = 10;

    int &r = a; // Reference must be initialized

cout << a << endl << r << endl;

// Same output 10

int b = 25;

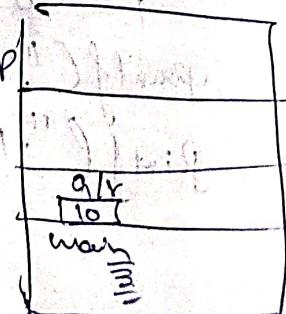
r = b;

cout << a << endl << r << endl;

// Both values of 25

return 0;

r is a reference to a



Pointer to a Structure:

Struct Rectangle

int length;

int breadth;

{

int main()

Struct Rectangle

\*p;

printf("%d %d", p->length, p->breadth);

C \*p; length = 20; // Accessing using pointer

P->length = 20; // Accessing using pointer in C

Creating Pointer to Structure Dynamically:

Struct Rectangle

int length;

int breadth;

{

Struct Rectangle \*p;

P = (Struct Rectangle \*)malloc(sizeof(StructRectangle)),

typecast → size since malloc fun returns void pointer

P → length = 10;

P → breadth = 5;

## Functions

### - Parameter Passing

- Pass by Value
- Pass by Address
- Pass by Reference (Only in C++)

function → Part of a program where

the code defines a specific task.

[Set of instructions for specific task]

Grouping Data → Structure

Grouping Instructions → Functions

Functions → Modules/Procedures

Having thousands of lines of code inside  
main() is a bad practice

```
int main() {
    func1();
    func2();
    func3();
}
```

Modular Programming

Modular/  
Procedural  
Programming

## Function

### Prototype / Header signature

Reusability:  
Using a function to perform a task again & again by just calling the function func().

CS Academy

```
int add (int a, int b) {
```

    int c;  
    c = a + b;

    return c;

}

int main () {

    x = 10;

    y = 5;

    z = add (x, y);

    printf ("%d\n", z);

    return 0;

Activation record  
add function

main

a	b	c
10		15
x	y	z
10	5	15

add

main

Variables of one function cannot be accessed by the other function with proper name.

- The activation record will be DELETED

When the program / function ends / terminates

# Parameter Passing

CS Academy

• Pass by value or Call by Value

" Pass by Value is used when we don't want to modify actual parameters

We use Pass by value of the function

by returning some results.

Like void swap(int a, int b)

int temp;

10 temp = x;

20 x = y;

10 y = temp;

The function returns no value.

Only the formal parameters are

changed - a & b remains unchanged.

" Returns an output result: Greatest number in array, finding max of two, adding ...

Call by Address:

" The addresses of the actual parameters

are passed to formal parameters.

Formal parameters must be

Pointers.

Any changes done inside function will modify actual parameters.

"One function can access the variables of another using pointers." CS Academy

Void Swap (int \*x, int \*y);

Means interchanging the values between

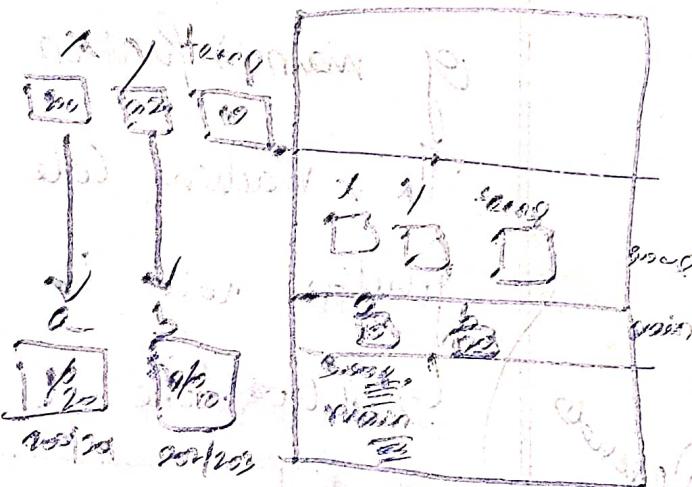
int temp;

temp = \*x;

\*x = \*y;

\*y = temp;

int main () {  
int a, b;  
a = 10;  
b = 20;  
Swap(&a, &b);  
printf("%d %d", a, b);  
return 0;}



int a, b;

a=10;

b=20;

Swap(a, b);

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

return 0;

3 part of a parameter how it

call by Reference: and about the

Only in C and C++

When formal Parameters are changed, the

Actual parameters are modified.

In In call by reference, taking the Swap function example, the Swap function becomes a part of the main function.

The Variables (actual parameters) of the main function & the Variables (formal parameters) of the Swap function is recorded Created under the activation record of main() function.

The machine code of the Swap function will be passed to main() More like a monolithic program.

Depends on Compiler (not mandatory)

The machine Code  $\rightarrow$  no linkage  
Source Code  $\rightarrow$  Modular Procedural (and linking jointly)

NOTE: Call by reference should not be used frequently or for heavy functions.

It should be used carefully.

For one or two small functions,

It can be used.

Call by reference is slow.

Call by reference is slow.

Call by reference is slow.

Void swap (int &x, int &y);

int temp;

temp = x;

x = y;

y = temp;

machine  
Code  
(parted  
into  
main())

int main () {

int a, b;

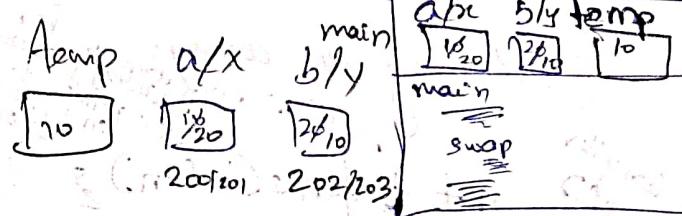
a = 10;

b = 20;

Swap (a, b);

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

return 0;



Array as Parameter:

Arrays can be passed only by Address.

(Both in C or C++)

Pointer to an Array  
 $A[i] \rightarrow$  specific to Array

void func (int A[], int n) {

    int i;  
    for (i = 0; i < n; i++)

        printf ("%d\n", A[i]);

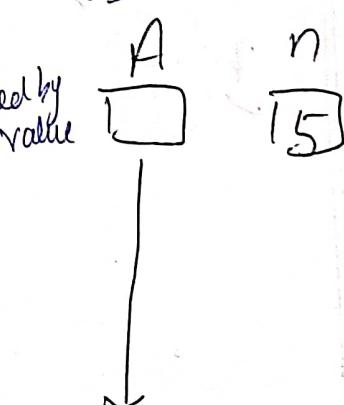
}

int main () {

    int A[5] = {2, 4, 6, 8, 10};

    func (A, 5);

}



2	4	6	8	10
0	1	2	3	4

void func (int n, int d) {  
 // Array values/elements can be  
 // changed as well

CS Acade

A[0] = 25;

Returning an array,  
→ C → only pointer for  
→ \* → pointer to an element  
or array of elements

int \* func (int n) {

int \* p;

p = (int \*) malloc (n \* sizeof (int));

return (p); // returns address of array

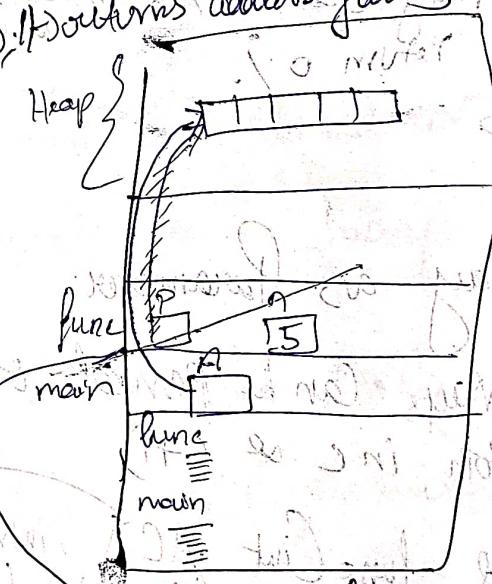
Some compiler  
won't allow  
returning  
function  
returning an  
array

int main () {

int \* A;

A = func (5);

return 0;



Once the function ends,  
the activation record  
is deleted.

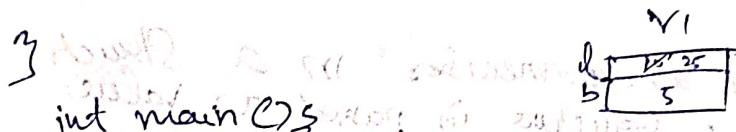
Structure as parameter:

Call by value; Call by reference

& call by reference (only in C++)

Call by Value:

int area (struct Rectangle r1) {  
 r1.length = 25; // This won't affect the actual parameter  
 return r1.length \* r1.breadth; }

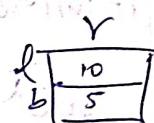


int main () {

struct rectangle r = {10, 5};

area(r);

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

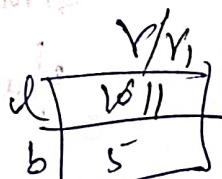


Call by reference:

int area (struct Rectangle \*r) {

r->length ++;

return r->length \* r->breadth;



int main () {

=

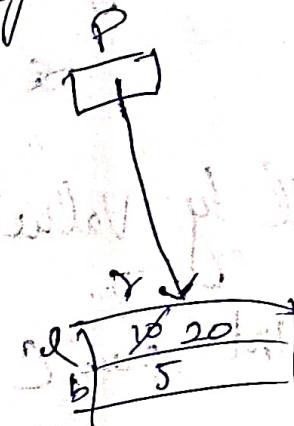
}

Call by address:

void ChangLength (Struct Rectangle \*P, int);

P → Length = d;

3



int main();

struct Rectangle r = {10, 20};

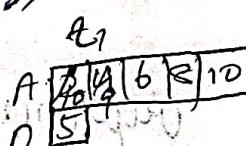
ChangLength (&r, 10);

Array as datamember in a Struct  
When a structure is passed as value,  
even if it is an array as Datamem, Struct Test S  
it can be passed by value to functions.

int A (S);

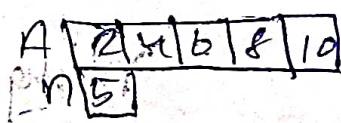
Void func (Struct Test t1) {  
    int n;

t1.A[0] = 10  
t1.A[1] = 9;



(10. expects 10)  $\rightarrow$  This <sup>value</sup> be changed.

3



int main();

Struct Test t = {2, 3, 4, 5, 6, 7, 8, 9, 10};

func (t);

Structures & functions:

Struct Rectangle {

int length;

int breadth;

};

void initialize C Struct Rectangle \*P, (int l, int b) {

P → length = l;

P → breadth = b;

}

int area C Struct Rectangle r) {

return r.length \* r.breadth;

}

void changelength (Struct Rectangle \*P, int l) {

P → length = l;

}

int main () {

Struct Rectangle r;

initialize (&r, 10, 15);

area (r); // 150 units

changelength (&r, 20);

area (r); // 100 units

return 0;

}

Classes & Constructors

Converting  $C \rightarrow C++$

Class Rectangle:

=  
=  
=

→ (esp. first) I am all

- all based on type

;

If just class is written, all

members become private.

2) Struct → All members are public.

Struct Rectangle

int width;

int breadth;

void initialize (int l, int b);

;

= (2 int) definition

= new struct (1 base)

= (as 1 base) definition

= new struct (1 base)

y;

## Class & Constructor - continued

:: Scope resolution operator

↳ Use Cases:

CS Academy

- 1.) To access a global variable, when there is a local var with same name.
- 2.) To define a function outside of class
- 3.) To access a class's static variables.
- 4.) In case of multiple inheritance
- 5.) For namespace
- 6.) Refer to a class inside of another class

#include <iostream>

using namespace std;

class Rectangle {

private:

int length;

int breadth;

public:

Overloaded Constructors [ Rectangle() { length = breadth = 1; } ] → Default Constructor

[ Rectangle(int l; int b) { } ] → Parameterized Constructors

Facilitators [

int area();

int perimeter();

Accessor,

gettr function [ int getLength() { return length; } ]

mutator/Puttr

function [ void setLength(int l) { length = l; } ]

Destructor [

~

Rectangle(); ] → Destructor : To destroy anything

Dynamic memory allocation inside constructor  
→ no arguments  
→ release that memory  
Destroy

Rectangle :: Rectangle (int l, int b) {

length = l;  
breadth = b;

CS Academy

int Rectangle::area() {

return length \* breadth;

int Rectangle::perimeter() {

return 2 \* (length + breadth);

Rectangle::~Rectangle()

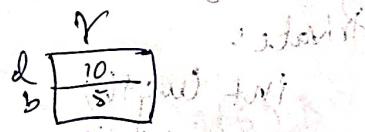
} // If de-allocation (like dynamic memory)

is required, destructor is called  
with class name or scope resolution

operator delete is defined.

int main() {

Rectangle r(10, 5);



cout << r.area();

cout << r.perimeter();

r.setLength(20);

cout << r.getLength();

return 0;

} // Life and death of Object is done here

Once the main() ends, the Destructor

will be automatically called, the object

will be destroyed.

## Template Classes:

Generic Class  $\Rightarrow$  Suitable for any data type, CS Academy  
One data type at a time.  
[Use the same class for multiple data types]

```
#include <iostream>
using namespace std;
```

Template class T  $\Rightarrow$

Class Arithmetic  $\{\}$

Private:

T int a;

T int b;

Public:

Arithmetic(T a, T b);

T add();

T sub();

};

Template class T  $\Rightarrow$  :: Arithmetic(a, b)

Arithmetic <T> :: :: Arithmetic(a, b)  $\Rightarrow$  class with methods

{ this  $\rightarrow$  a = a;

    this  $\rightarrow$  b = b;

}

Template class T  $\Rightarrow$

Arithmetic <T> :: add()

T C;

C = a + b;

return C;

}

Int week 3

CS Acad.

Arithmetic & int a(2015)

cout < a; add c endl;

=  
==  
==

variables declared

as separate files

< T & D

structures can

be

defined

in

## INTRODUCTION:

### Data Structures:

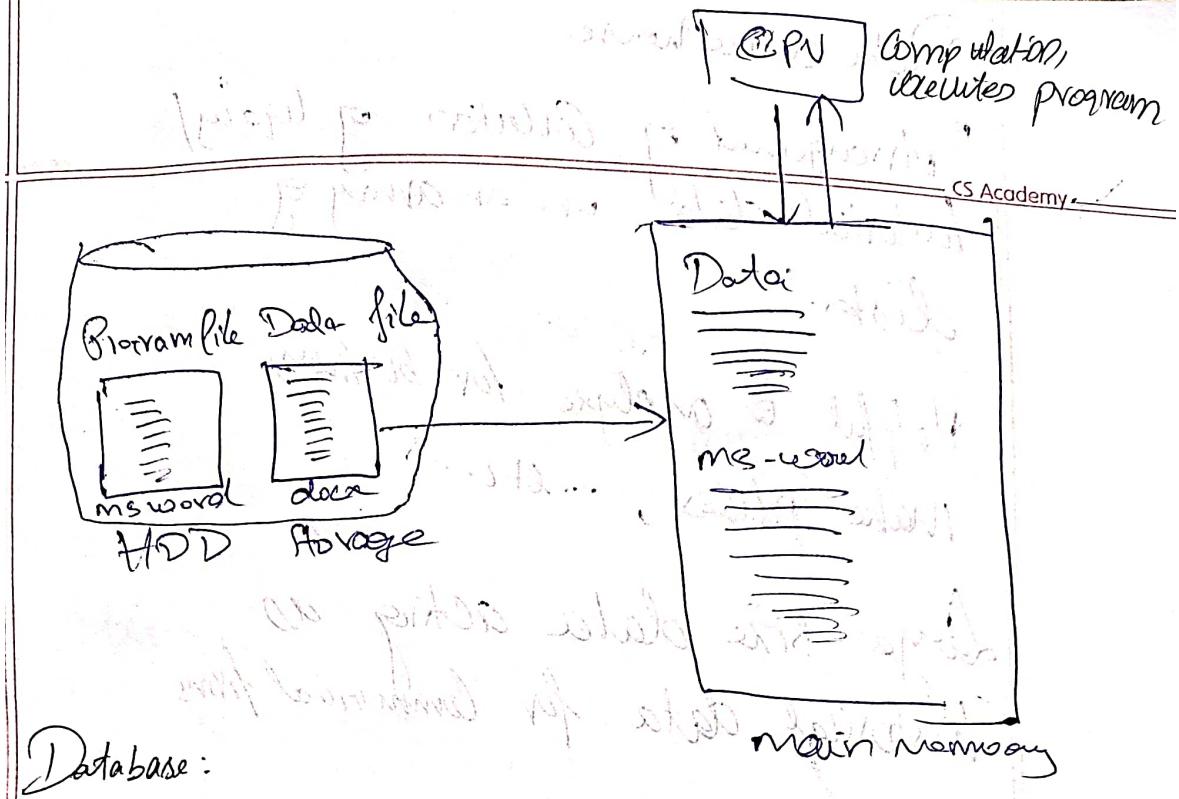
"Arrangement of collection of data items  
so that they can be utilized efficiently  
& the operations on the data can be  
done efficiently."

All about efficient arrangement of  
operations on the data.

During the program execution, how will  
the program manage data inside main  
memory & perform efficient operation on  
them.

Efficient organization of the data in  
main memory so the operations  
can be carried out efficiently.

C Foster

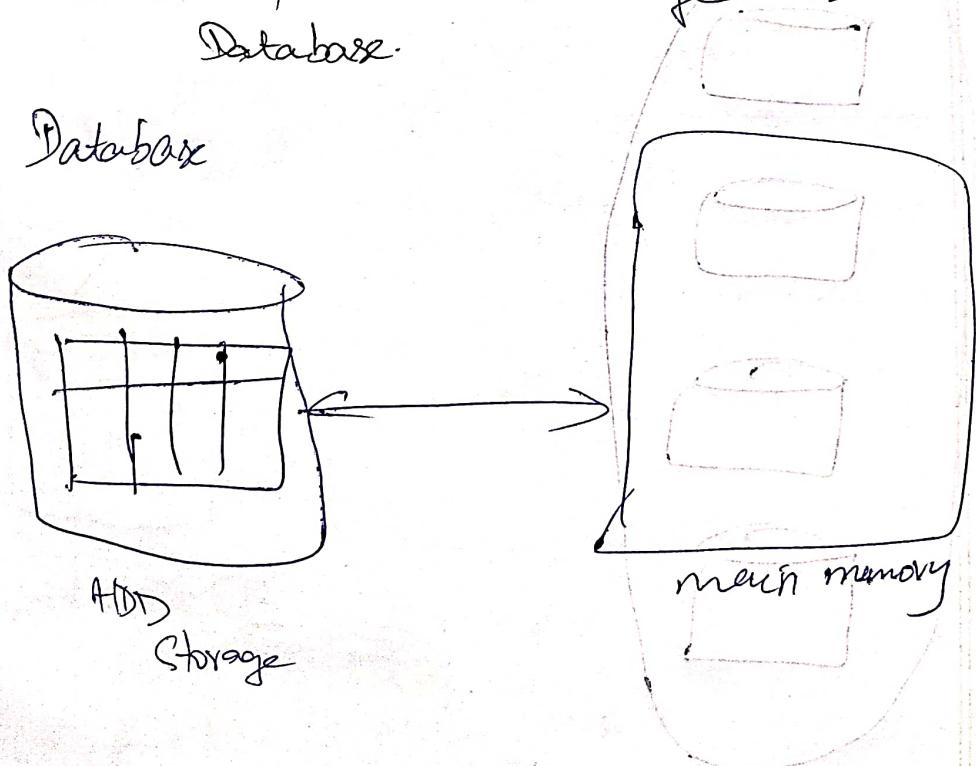


Database:

Arranging the data in some model, like relational model in the permanent storage so that it can be retrieved or accessed by applications easily.

↳ This type of arrangement of data in permanent storage is called Database.

Database



## Data Warehouse

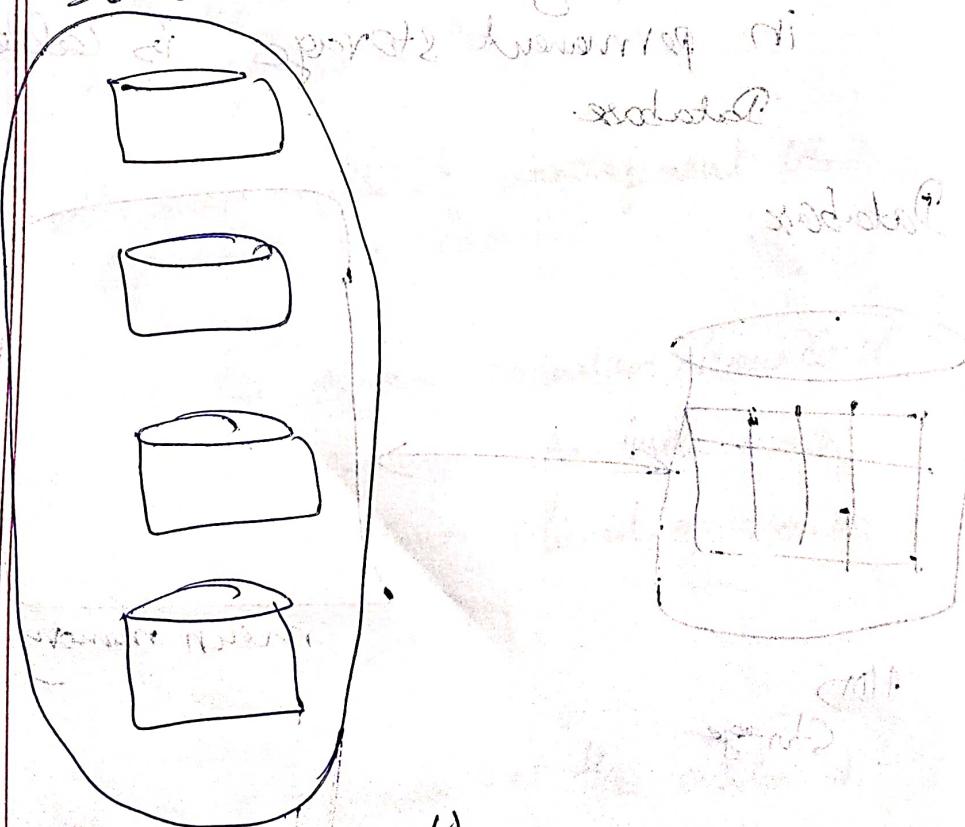
"Arrangement of Collection of legacy / historical data in an array of disks."

Helpful to analyse for business, make policies, etc.

Large size data acting as historical data for commercial firms.

Algorithms written for analysing these data are called Data mining algorithms.

## Data warehousing and big data



Big data: (Internet)

Storing & analyzing very large size of data

Stack vs Heap memory :-

## 1) About main memory

2.) How a program uses main memory

## 3.) Static Allocation

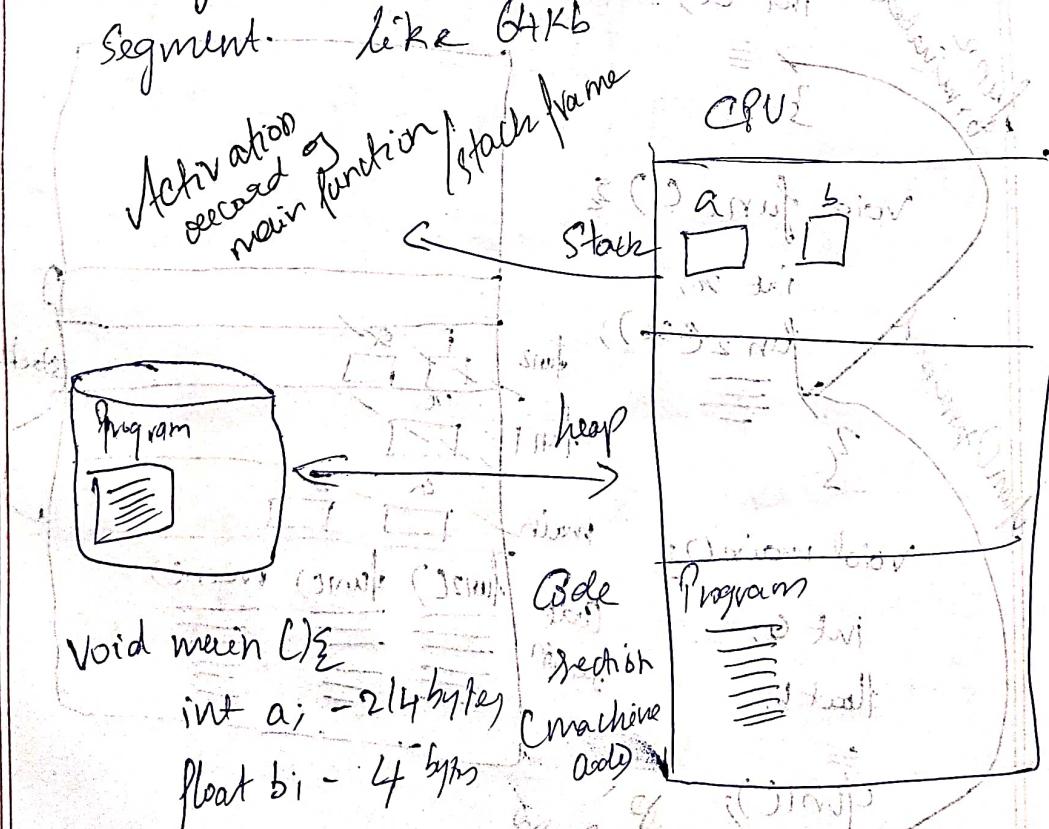
## 4) Dynamic Allocation

## Static vs Dynamic Allocation

Address → single dimension / Number

Memory will be divided into a manageable

Segment like 64KB



The portion of the memory that is given to function- Activation record of that function.

The size of the memory occupied by a function was decided at Compile time.

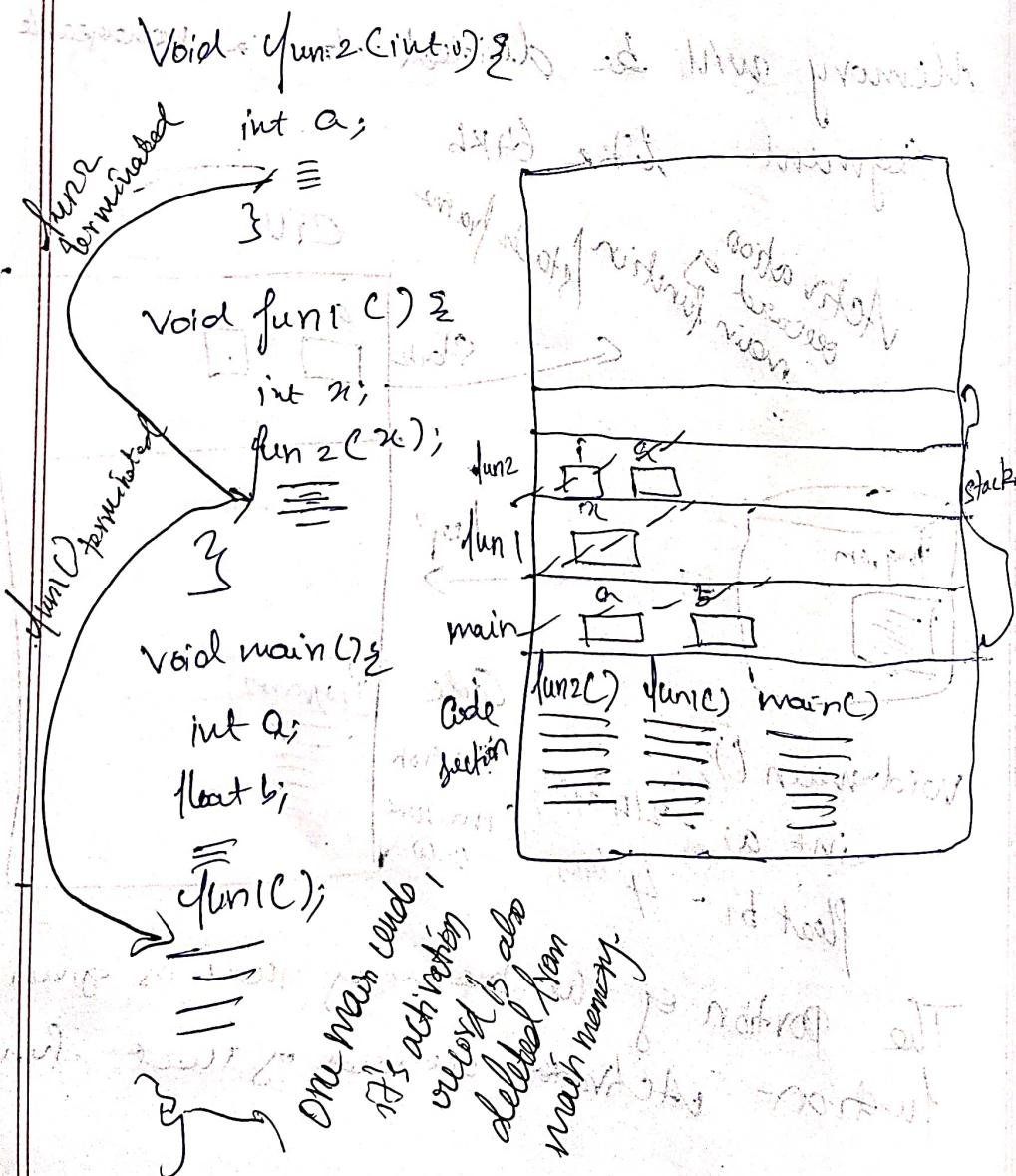
CSACOM

How many bytes of memory is planned by this function was decided at compile time.

Size of the memory is static  
Decided at compile time  
before run time

Memory allocation for sequence of function

Calls:



The mechanism by which the functions activation records are created on top of another or deleted, is called Stack.

It behaves like a stack.

HEAP:

Large pile of memory, organized but unorganized, like a folder.

Heap should be treated as a resource.

Heap  $\Rightarrow$  Dynamic memory allocation.

Heap memory can only be accessed by pointers

C++

int \*p;

$p = \text{new } p[5];$

C-lang  $p = (\text{int } *)\text{malloc}(24 * 5)$

Heap memory should always be freed after utilizing it.  $\Rightarrow$  Good Practice.

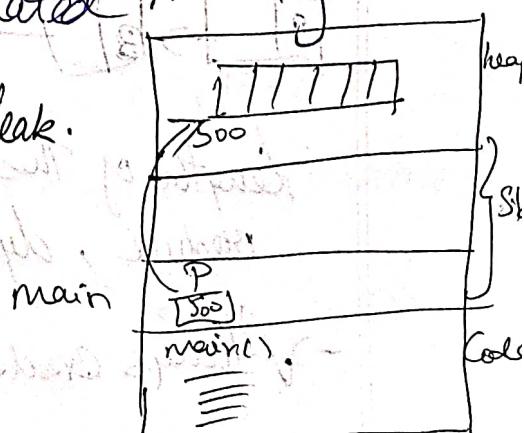
If failed to free allocated memory

L) memory leak.

`delete[] p;`

`P=NULL`

`if (P != NULL) free(P)`



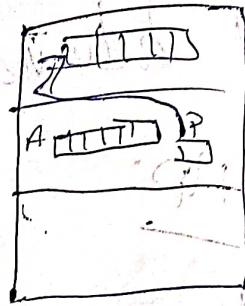
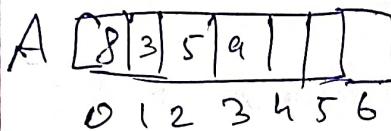
## Types of Data Structures:

1) Physical Data Structures

2) Logical Data Structures

### Physical:

1) Physical Array



The size of an array is static.

Cannot be changed, fixed.

→ Can be created inside Stack

→ In C/C++ heap with pointers

→ If we know the length/size

of the list → we can go for arrays.

2) Linked Lists

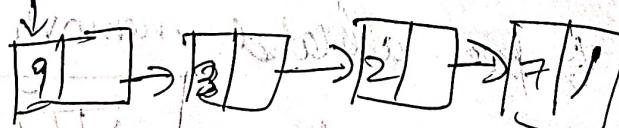
→ Dynamic Data Structure

→ Collection of nodes

where each node contains data

& is linked with other node

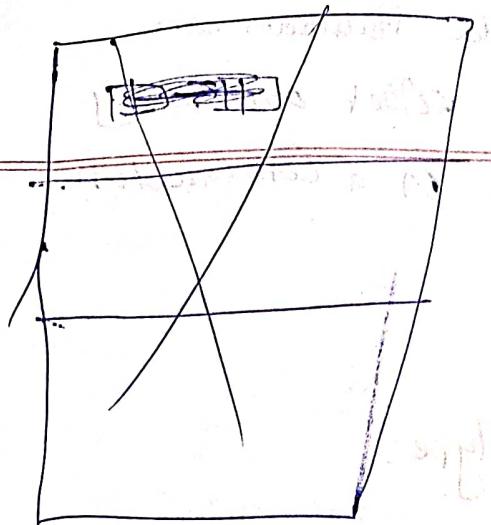
Head



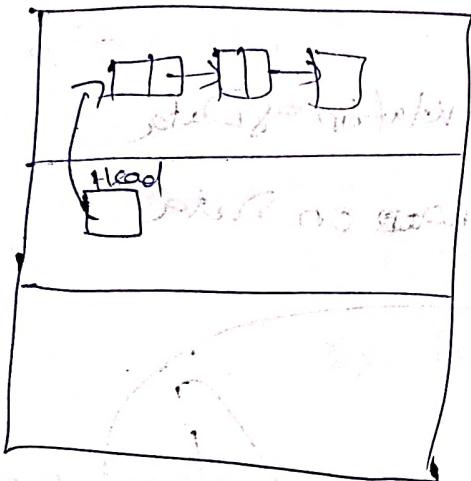
length of this list can grow or

reduce, dynamically.

→ Always created inside Heap.



CS Academy



Stack  
Heap

Stack

Logical Data Structures: (Used in algorithms)

Stack → last In first Out

- 1.) Stack ] linear
- 2.) Queues ]
- 3.) Trees ] Non-linear
- 4.) Graphs ]
- 5.) Hash tables ] Tabular

Queues → FIFO  
Graphs → Hierarchy, Collection of nodes

Physical DS → How we store the Data in main memory.

Logical DS: How we want to utilize those values? How efficiently can insertion & deletion be performed.

What discipline to follow? → logical DS

These logical Data Structures are implemented using either an array or linked list or a combination.

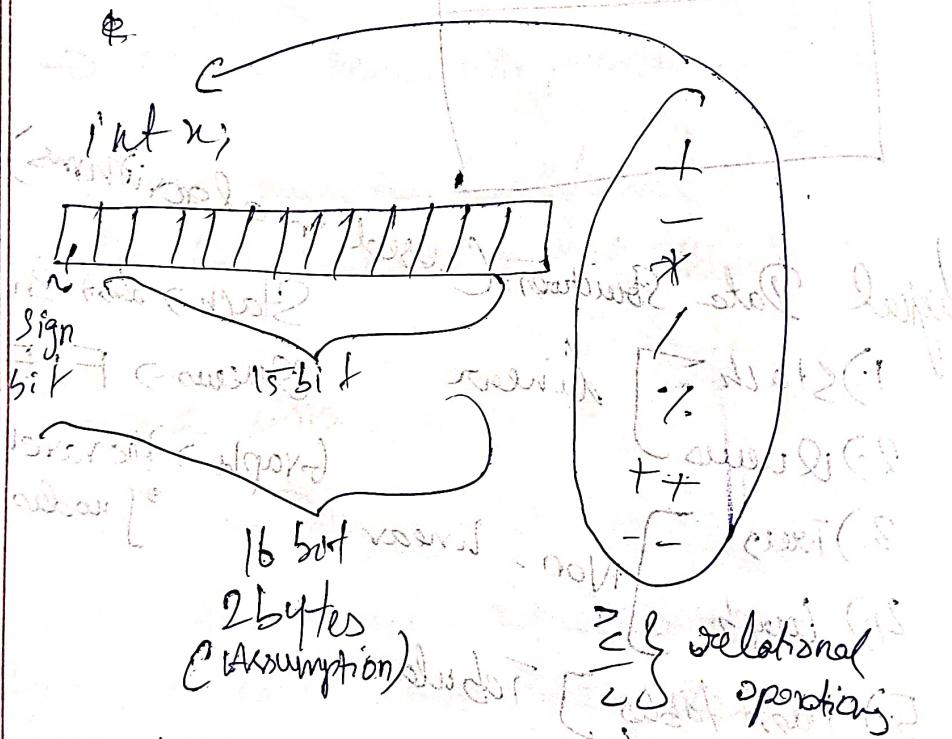
ADT:

Abstract Data type:

Data type:

1) ~~representation~~ = Data

2) Operations on Data



The ~~other~~ internal details show the operations are hidden (are performed by Abstract in binary form in main memory)

list  $\rightarrow$  8, 3, 9, 11, 6, 10, 11, 2, 3, 4, 5, 6  
0 1 2 3 4 5 6

CS Academy

How can the list be represented.

Data:

- 1) Space for storing elements
- 2) Capacity (Max capacity)
- 3) Size of list (Number of elements)

To represent them

1) Array

2) Linked List

Operations:

add (x)

remove ()

search (key)

When a class is written, the object is created in the class & we can use it -

We need not worry how things work  
working intervals  $\rightarrow$  Abstract.

We will represent Logical DS as ~~DS~~ A.

List  $\rightarrow$  8, 13, 9, 4, 6, 10, 12, 15  $\rightarrow$   
0, 1, 2, 3, 4, 5, 6, 7  
↑ (odd)

CS Acc

- add(~~Element~~ Element) / append(Element)
- add(Element, element) / insert(index, elem)
- remove(index) / list ~~remove~~ <sup>speed</sup>, re-arrange
- set(index, element) / replace(index, elem)
- get(index) / index 5  $\rightarrow$  10
- search(key) / contains(key)

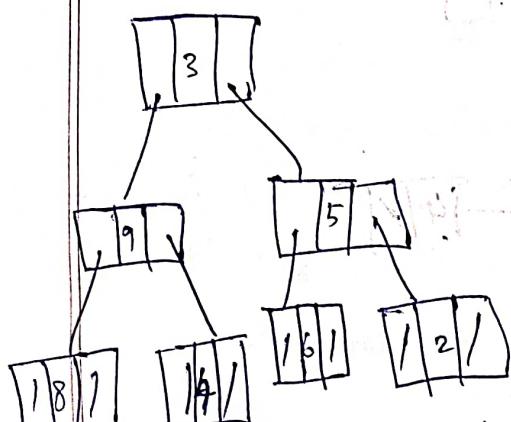
whether key element is there  
in the list

list sorted()

- reverse()
- combine()
- merge()
- split()

## Time & Space Complexity

CS Academy



$$O(\log n)$$

2) Searching thru all nodes -  $O(n)$

$O()$  → order( $\downarrow$ )  $\rightarrow$   $\downarrow$   $\rightarrow$   $\downarrow$

```
for(i=0 ; i<n ; i++) {
```

for(j=0; j<n; j++) {

三

$\hookrightarrow O(n^2)$  nested for loop over n elements

## Matrices

The diagram consists of two vertical columns of six small black dots each. The left column is enclosed in a rectangular frame with brackets at the top and bottom. The right column is also enclosed in a similar rectangular frame with brackets. The columns are positioned side-by-side.

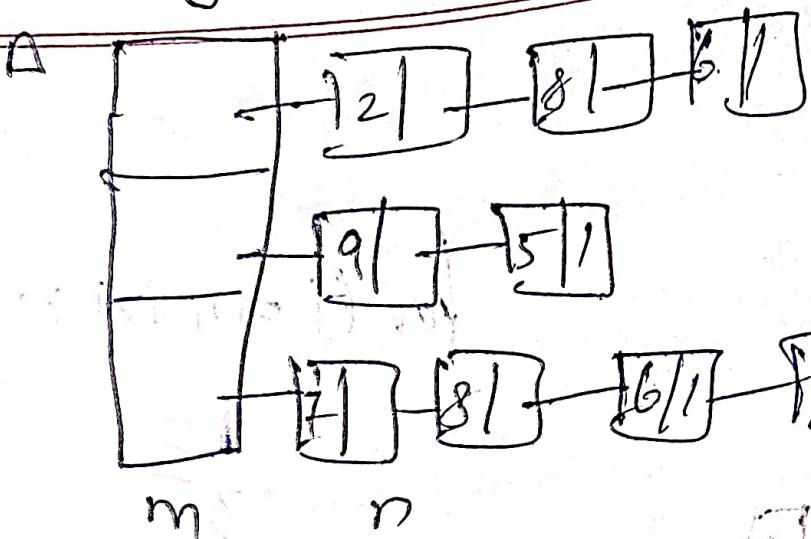
$\sigma_{xm - k \times k}$

$\rightarrow$  If all elements  
( $\exists$ )

$\hookrightarrow O(n^2)$

Array of linked lists:

CS Academy



$O(m+n)$

Space Complexity -

space consumed in main memory

$O(1)$ ,  $n$  elements

$\cup O(n)$  elements

↳ Does not mean bytes.

Refers to, The space is dependent  
on what ?

## Recursion:

CS Academy

1.) What is Recursion?

2.) Example of Recursion

3.) Tracing Recursion

4.) Stack used in Recursion.

5.) Time Complexity

6.) Recurrence Relation

int func (int n) {  
 if (n == 0)  
 return 1;  
 else  
 return func(n-1) \* 2;

int main() {  
 int result = func(5);  
 cout << result;  
}

func(2) \* 2; → The (\*) multiplication by 2 will  
done once the function was a  
reached with some value.

6) Type func (param) {

if (C (Case Condition)) {

func (param)

A function calling  
itself → recursion.

3) {

return (not possible)

Recursive functions are traced in the form of a Tree.

Void sum (int n) {

if (n > 0) {

printf("%d", n);

sum (n-1);

}

}

int main() {

int x=3;

func (3);

} return 0;

func (3)

func (2)

func (1)

Condition  
False

func (0)

Tracing Process  
Recursive function.

O/P: 3 2 1 \*

"Printing done at Calling time!"

Void fun2 (int n) {  
    if (n > 0) {

        fun2(n-1);

        printf("%d\n", n);

    }

}

int main () {

    int n = 3;

    fun2 (n);

    return 0;

}

        fun2(2)

        fun2(1)

        fun2(0)

O/p: 1 2 3

Printing done at returning Time.

1st Case:

Rooms with bulbs

10101010

O/p: 1 2 3

1. Switch on bulb

2. Go to next room

2nd Case

10101010

O/p: 3 2 1

1. Go to next room

2. Switch on bulb

Recovery has two phases:

→ Calling Phase

→ Returning Phase

Void fun (int n) {

if (n > 0) {

Ascending ↙

1. Calling

2. fun(n-1)\*2;

Descending ↙

3. Returning

3

loops → Repeating Statement

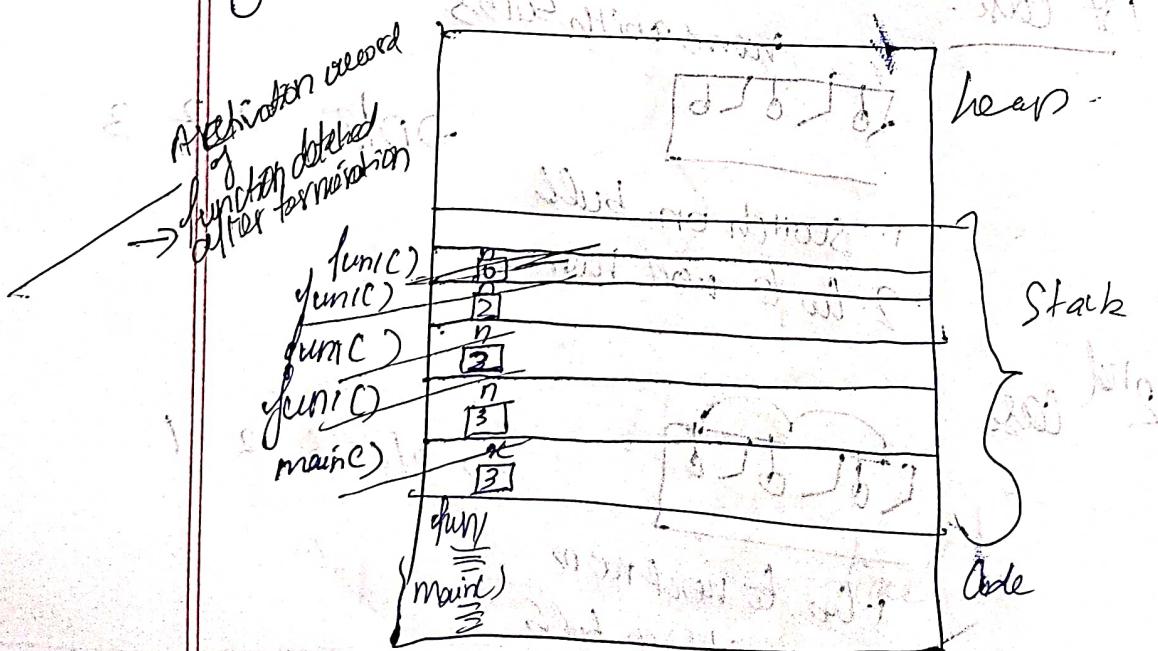
↳ Only Ascending Phase

Recursion =

↳ Ascending & Descending phases

How Recursion uses Stack ?:

Taking Previous Recursion Code:



For  $n$  elements/value/...  
in stack.

$\hookrightarrow n+1$  calls /function calls /activation record  
in stack.

$\hookrightarrow O(n)$ , Polynomial degree - 1

$\hookrightarrow$  Space Complexity.

Total AR  $\rightarrow n+1$

(Activation record)

Void func(int n) {

if ( $n > 0$ ) {

1. func(n-1)

2. printf("%d\n", n)

}

}

Void main()

int x = 3;

func(x);

}

func(3)

func(2)

3

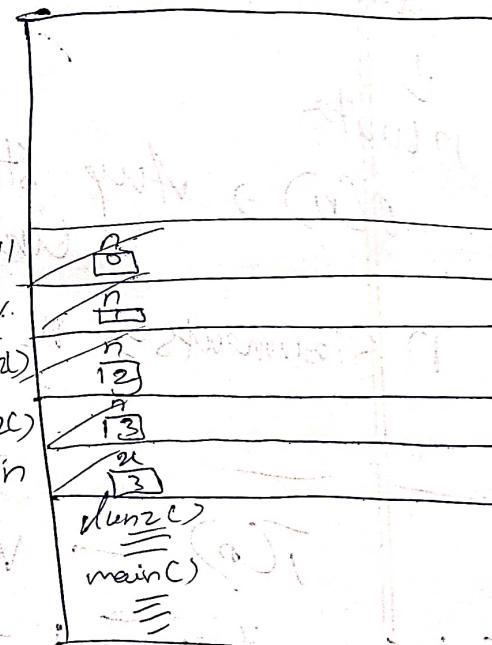
func(1)

2

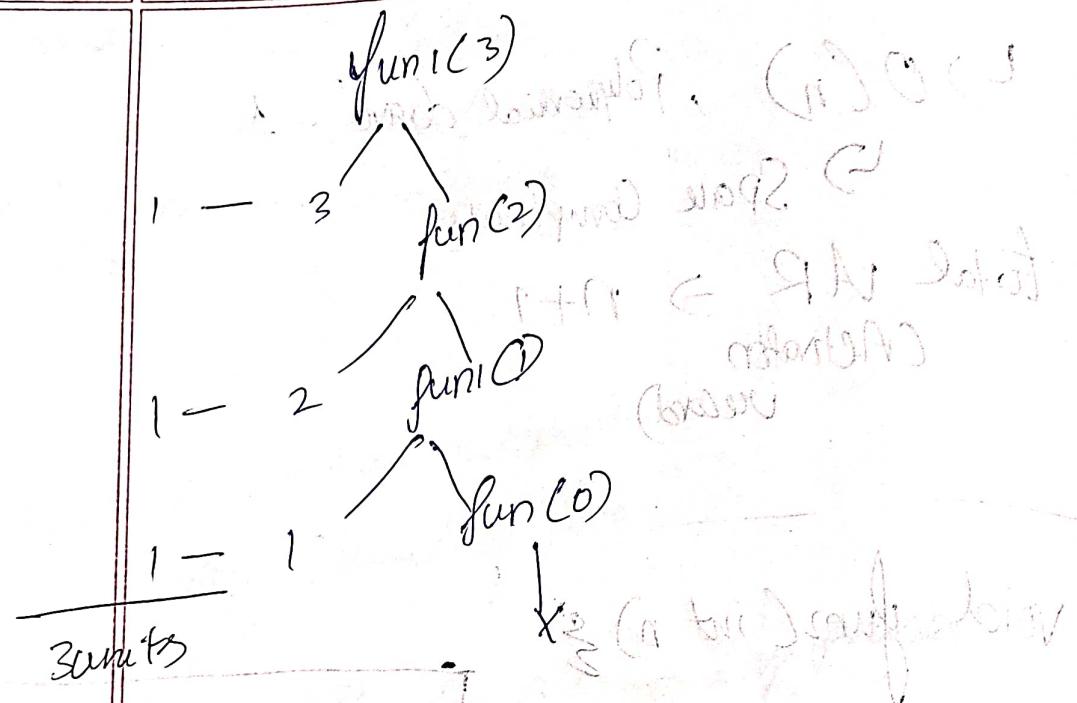
func(0)

1

x



Recurrence  
Recursion Relation  
Time Complexity



$\downarrow$   
 $n$  units

$O(n) \Rightarrow$  Any statement taken one unit of time

$n$  statements  $\Rightarrow n$  units of time

$T(n) \rightarrow$  void `fun1` (int  $n$ ) { when  $n <= 1$

$T(n) = 1 + \text{if } n > 0$   
 $T(n-1) + 2, n > 0$

$T(n-1) = T(n-2) + 1 + 2;$

$\therefore T(n) = T(n-1) + 2;$

lets say  $T(n) = T(n-1) + 1;$

$T(n) = T(n-2) + 1 + 1;$

$T(n) = T(n-3) + 1 + 2;$

$T(n) = T(n-4) + 1 + 3;$

$$T(n) = T(n-k) + K;$$

$$\text{If } T(n-k) = 0$$

$$\Rightarrow n=k$$

$$\therefore T(n) = T(n-n) + n \quad \text{Time taken}$$
  
$$= T(0) + n$$

$$T(n) = 1 + n$$

$\rightarrow O(n) \rightarrow$  Time taken

Printing for n times or no. of calls  
 $O(n+1)$

### Static Variables in Recursion:

Static Variables are created in the stack section of code section called Static Variables.

$\rightarrow$  They won't copy every time the definition gets called. Just a single copy

~~int fun1(int n)~~  
~~if (n==0)~~  
~~return 1~~  
~~else~~  
~~int fun2(int n)~~  
~~Static int x=0;~~  
~~x=x+1~~  
~~if (n>0)~~  
~~return x+fun2(n-1)~~

P.T.O

int fun (int n) {

if ( $n > 0$ ) {

n++;

return fun (n-1) + x;

}

}

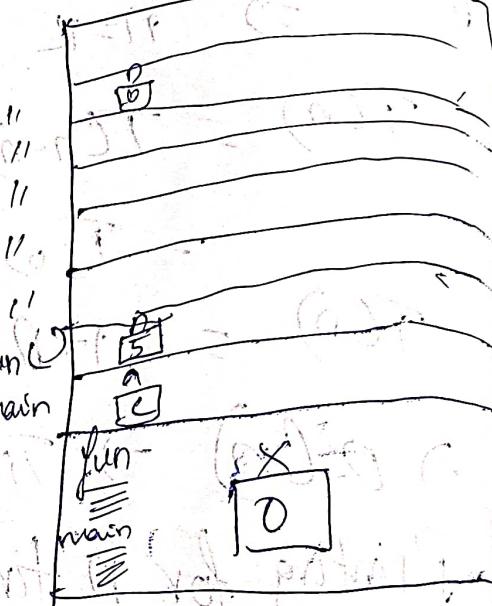
int main () {

int a = 5;

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

return 0;

}



→ should not be included  
as tracing

| 0 | 1 | 2 | 3 | 4 | 5 |

fun(5) = 25

$$\text{fun}(4) + \underline{\underline{5}} = 25$$

$$\text{fun}(3) + \underline{\underline{5}} = 20$$

$$\text{fun}(2) + \underline{\underline{5}} = 15$$

$$\text{fun}(1) + \underline{\underline{5}} = 10$$

$$\text{fun}(0) + \underline{\underline{5}} = 5 ; x=5$$

| 0 |

# Recursion:

↳  $\Sigma$

CS Academy

- 1.) Tail Recursion
- 2.) Head Recursion
- 3.) Tree Recursion
- 4.) Indirect Recursion
- 5.) Nested Recursion

## Tail Recursion:

In a recursive function, where a function calls itself, that call is called Recursive call.  
If the recursive call statement is the last statement of the function it is called Tail Recursion.

fun (n){  
    if(n>0){  
        all operations performed at calling time. No operations at returning time.  
        fun(n-1); → Recursive call  
    }  
}

When operations perform at return time?

fun (n-1)+n;  
    ↳ Return time  
    ↳ NOT A TAIL RECURSION

Tail Recursion can be written as a loop.

Void fun (int n) {

while (n > 0) {

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

n =;

}

}

Time → same for both.

$\Theta(n)$

Space Complexity?

Tail Recursion:  $\Theta(1)$

While Loop:

$\Theta(1) \rightarrow \text{Constant}$

loop is efficient than Tail recursion.

Not the case for all recursion.

## Head Recursion:

All operations done in returning time.

Void fun(Cint n){

if (condition) {

Jan En-i

Void fun (int n){

if  $(n > 0) \in$

· funcn-1;

return  
time

2

3

## Free Recursion

Two or more recursive cells inside a recursive function.

Vid fun Cint

if  $(n > 0)$  is

$$\overline{\lim}_{n \rightarrow \infty} (c_n - 1);$$

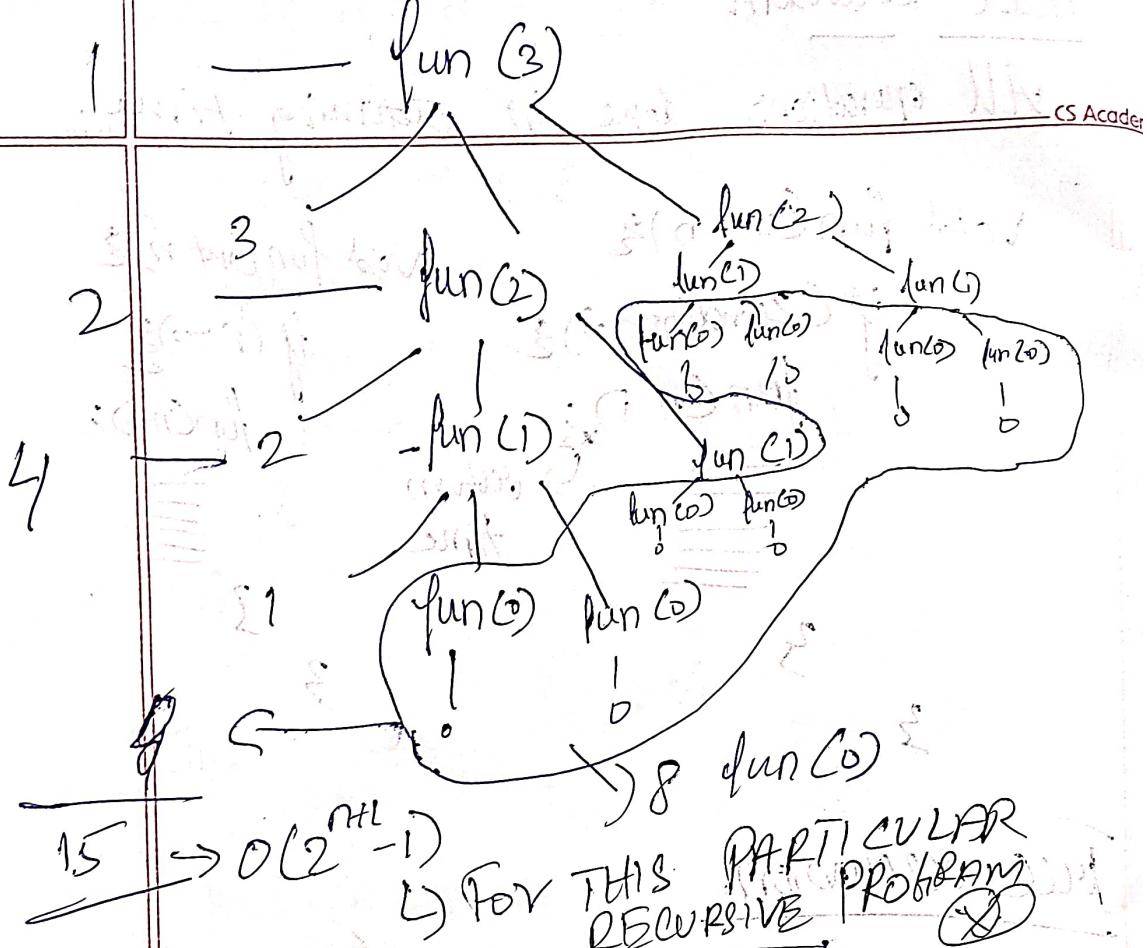
Fun(n-1)

void fun (int n){  
if (n>0){

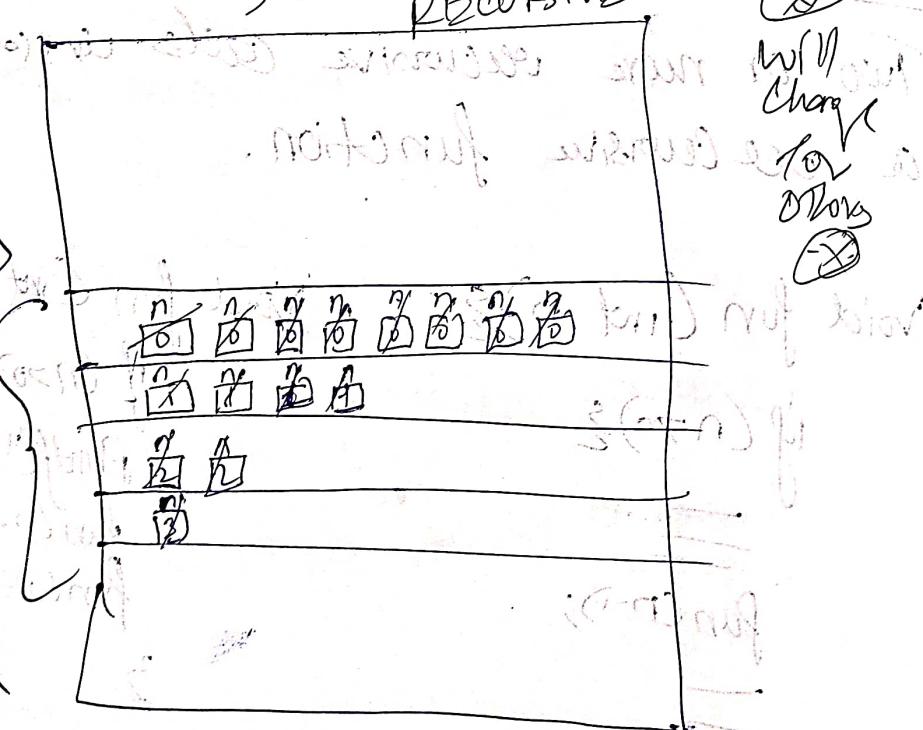
Printf("%d\n", n);  
fun (n->);  
fun (n->);

int main () {

```
int n=3;  
visitf("1.0n!", fun(x));  
return 0;
```



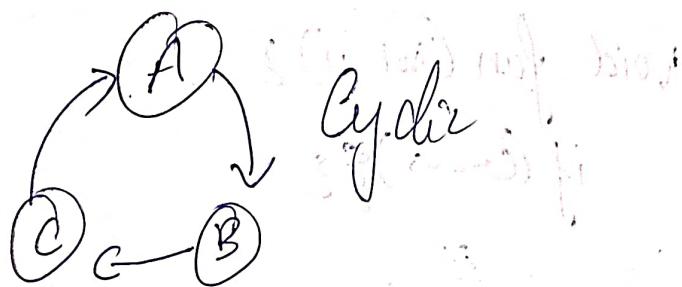
Recursion Tree  
 $O(n)$   
 For  $n$   
 Space =  $n^2$



$$\begin{aligned}
 T(\text{fun}(n)) &= 1 + 2 + 4 + 8 \\
 &= 2^0 + 2^1 + 2^2 + 2^3 \\
 \Rightarrow 2^{n+1} &= 15
 \end{aligned}$$

## Indirect Recursion:

More than one function calling each other  
in a circular fashion.



void AC(int n){

if (n >= 0) {

funB(n-1);

}

Void funA (int n){

if (n > 0) {

Printf ("%.d\\n", n);  
funB(n-1);

}

} void B (int n){

if (n >= 0) {

funA(n/2);

3 / \ n-1;

Void ofunB (int n){

if (n > 1) {

Printf ("%.d\\n", n);

3 / \ n-1;

All activation records deleted

3 / \ n-1;

funC(20)

20

funB(19)

19

funA(9);

9

funB(8);

8

funA(4);

4

funB(3);

3

funA(1);

funB(0);

1 X

## Nested Recursion:

Recursive function will call parameter  
as a recursive call.

void fun (int n) {

if (n == 0) {

=

fun(fun(n-1));

↳ Unless this recursive  
call's result is obtained  
this recursive call cannot  
be made.

↳ A recursive call is taking  
its parameter as recursive call.

↳ This recursion is called  
recursion inside recursion.

↳ Nested Recursion

int fun (int n) {

if (n > 100) {

return n - 10;

}

else {

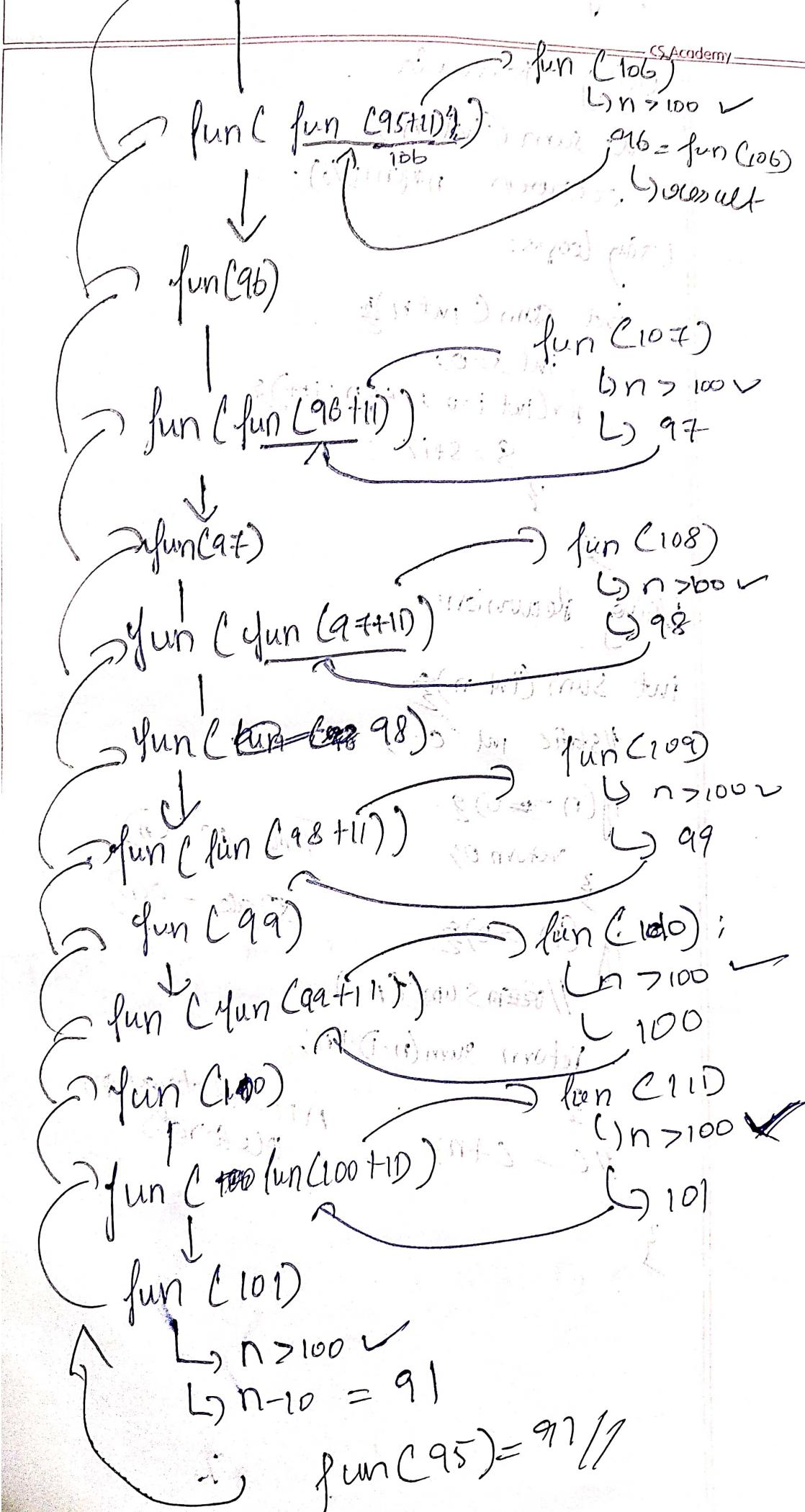
return fun (fun(n+1));

}

}

fun(95)

$$\text{fun}(95) = 91//$$



## Sum of first n numbers using Recursion:

CS Acade

Using formula -

int sum (int n) {  
 return n\*(n+1)/2;

Using loops:

int sum (int n) {

int s = 0;  
 for (int i=0; i<n; i++) {  
 s = s+i;

Using Recursion:

int sum (int n) {

// static int c=0;

if (n == 0) {  
 return 0;

if (n > 0) {

// sum(n-1)

return sum(n-1)+n;

// c = c+n;

if (c == 0) {

## Factorial of a Number:

$$5! = \underline{5 \times 4 \times 3 \times 2 \times 1} = 120 \quad \text{CSA}$$

int ~~fact~~ fact(n) {

if (n < 0) {

return 0;

if (n > 0) {

return fact(n-1) \* n;

if (n == 0) {

return 1;

return fact(n-1) \* n;

Power Using Recursion:

$$\text{exp} = (m)^n$$

$$= m \times m \times \dots \times m$$

*n times*

$$= \underbrace{m \times m \times \dots \times m}_{\text{exp } (n-1) \times m}$$

int pow(int m, int n) {  
 if (n == 0) {  
 return 1;  
 } else {  
 return pow(m, n - 1) \* m;  
 }
}

CS Academy

pow(2, 9)

pow(2, 8) \* 2

pow(2, 7) \* 2^1

pow(2, 6) \* 2^1

pow(2, 5) \* 2

pow(2, 4) \* 2

pow(2, 3) \* 2

pow(2, 2) \* 2

pow(2, 1) \* 2 = 4

pow(2, 0) \* 2 = 2

Another way:

$$2^8 = (2^2)^4$$

$$= (2 \times 2)^4$$

$$2^9 = (2^2)^4 \times 2$$

$$= (m \times m)^n \times m$$

~~$$\text{POW}(2, 9) = 2^9$$~~

$$2 + \text{POW}(2^2, 4) = 2^{2+2}$$

$$\text{POW}(2^4, 2) = 2^8$$

$$\text{POW}(2^8, 1) = 2^8$$

$$2^8 \times (2^{16}, 0) = 2^{8+16}$$

↓

6 multiplications  
less than  
than one

## Taylor's series:

$$1 + 2 + 3 + 4 + 5 + 6 + \dots$$

CS Academy

Can use the previous functions, factorial or power function to create Taylor series.

Q

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

double

int & (int x, int n) {

Static double p=1;

Static double f=1;

double v;  $\left( \frac{f \cdot x^n}{n!} \right)$   $\rightarrow$  multiplication

if (n==0) {  $\frac{x^0}{0!} = 1$  } else {  $\frac{x^n}{n!}$  }  $\rightarrow$  O(n^2)

return 1;

$v = e(x, n-1);$

$p = p * x^{(n)}$

$f = f * n;$

$\rightarrow$  O(n^2)

return v + (p/f);

Time is  
Quadratic

3

int main() {

Printf("%d", f(2, 2));

return 0;

3

Using Recur functions:

double taylor(int x, int n) {

Static int y=0;

Static double c=1.0;

If (y < n) {

$y = y + 1;$

$c = \frac{c}{y} (double) p(x, y) / (double) func(x, y)) + c;$

return c;

```
    return c;
```

```
}
```

```
int main () {
```

```
    printf ("1 fm", log (2.7));
```

```
    return 0;
```

```
}
```

Taylor's Series with Horner's rule:

$$P(x) = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots n\text{ terms}$$

$$= 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^n}{n!}$$

$$= 1 + \frac{x}{1} \left[ 1 + \frac{x}{2} + \frac{x^2}{2+3} + \frac{x^3}{2+5+4} \right]$$

$$= 1 + \frac{x}{1} \left[ 1 + \frac{x}{2} \left[ 1 + \frac{x}{3} + \frac{x^2}{3+4} \right] \right]$$

$$= 1 + \frac{x}{1} \left[ 1 + \frac{x}{2} \left[ 1 + \frac{x}{3} \left[ 1 + \frac{x}{4} \right] \right] \right]$$

↳ Time  $\rightarrow O(n)$   
↳ linear

Fibonacci Series:

~~for loop~~

$$\text{fib}(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ \text{fib}(n-2) + \text{fib}(n-1) & n>1 \end{cases}$$

For Loop: ~~Don't print without answer~~

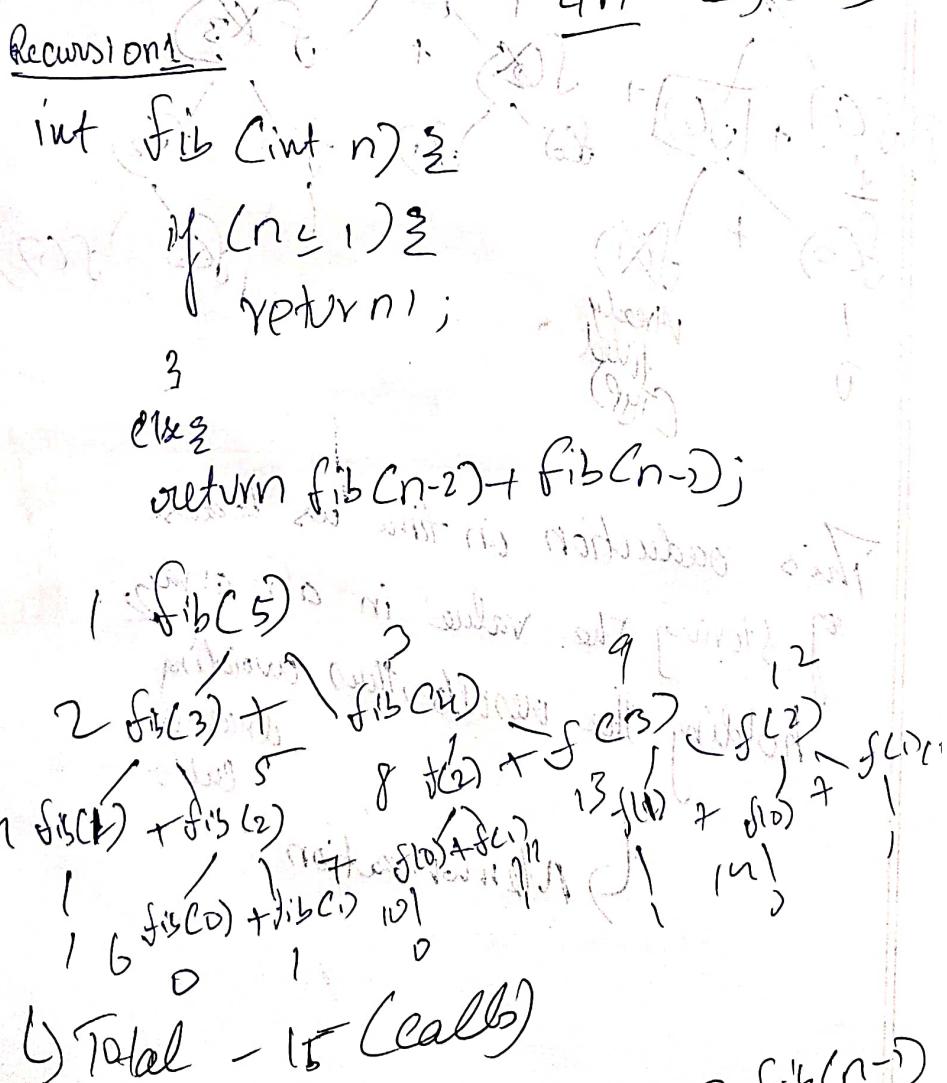
---

int fib (int n) {  
 int t0=0, t1=1, s, i; — X  
 if (n<=0) return n; — X  
 for (i=2; i<=n; i++) — ~~more~~  
 s=t0+t1 — ~~n-1~~  
 t0=t1; — ~~n-1~~  
 t1=s; — ~~n-1~~  
 }  
 return s; — ~~more~~

Recursion:

---

int fib (int n) {  
 if (n<=1) —  
 return n;  
 else —  
 return fib(n-1)+fib(n-2);



(Total - 15 calls)

Assum.  $\boxed{fib(n-2) + fib(n-1)} = 2fib(n-1)$

Calling 2 times for a needed value  $\boxed{T = O(2^n)}$

~~Reduced value after 2 times~~

A recursive function calling itself multiple times for the same values.

memorization

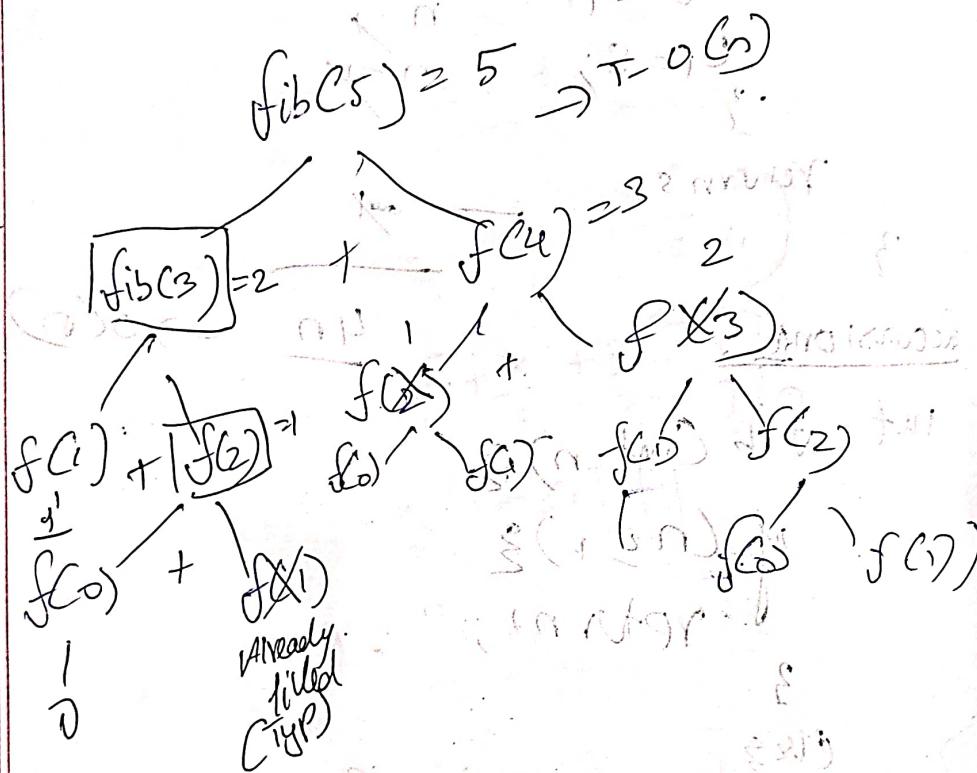
↓  
↓  
↓

Global  
OR  
Static  
Array

Initialize  
with  
→

F	10	11	12	13	14	15	16
	0	1	2	3	4	5	6

Excessive recursion



This reduction in time is because

of storing the value in an array,

holding the results, thus avoiding excessive calls.

↳ Memorization

Storing the function calls, so that it  
can be utilized again

CS Academy

## L) Memoization.

int F[10];

int fib(int n){

if (n == 0){

F[n] = n;  
return n;

else {

if (F[n-2] == -1){

F[n-2] = fib(n-2);

}

if (F[n-1] == -1){

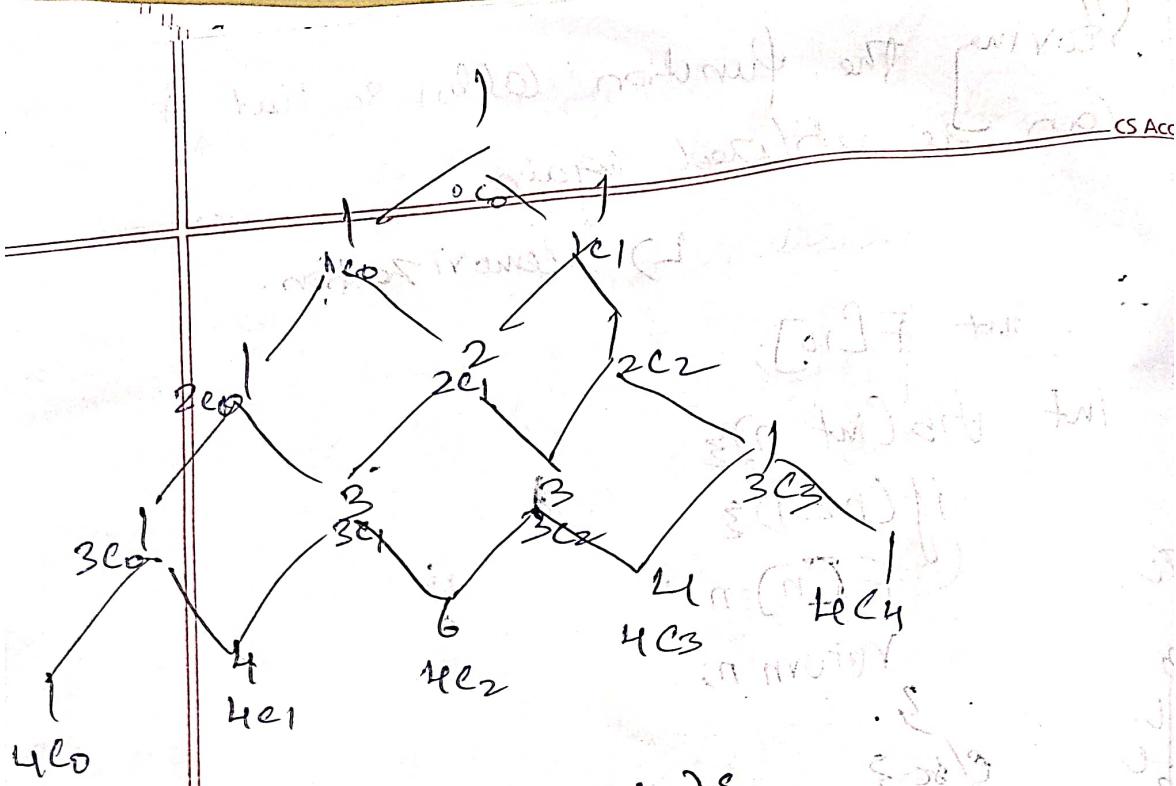
F[n-1] = fib(n-1);

}

return fib(n-2) + fib(n-1);

ner

$$ner = \frac{n!}{r!(n-r)!}$$



`int C (int n, int r) {`

`if (r == 0 || n == r)`

`return 1;`

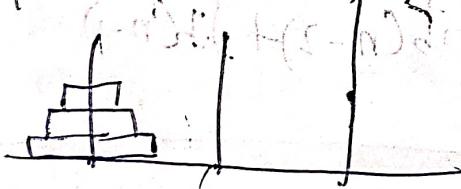
`}`

`else`

`return C(n-1, r-1) + C(n-1, r);`

`};`

## Tower of Hanoi

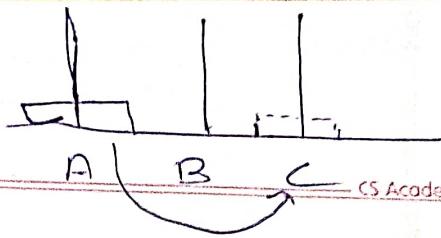


$$T(n) = 2T(n-1) + 1$$

$$T(n) = 2^n - 1$$

Tow(1, A, B, C)

- 1) Move Disk from A to C using B



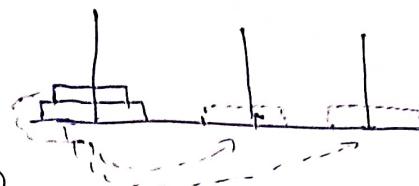
CS Academy

Tow(2, A, B, C)

1) Tow(1, A, C, B)

2) ~~Move Disk~~

From A to C using B



3) Tow(1, B, A, C)

int ~~your~~ C(int n){

int ~~your~~  $\pi = 1$ ;

int ~~your~~  $tow(n, int a, int b, int c)$  {

if ( $Cn == 0$ ) {

return 0;

}

else {

$tow(Cn-1, a, c, b);$

printf("Move disk from %d to %d\n",  $a, c$ );

$tow(Cn-1, b, a, c);$

}

}

Quez - Q4: ~~also suggested not~~

int fun(int n) {  
 if (n == 1) return 1;  
 int x = 1, y;

if (n == 1) {  
 return x;

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

x = x + fun(k) + fun(n - k);

}

return x;

g

fun(5)

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

$$+ \text{fun}(3) + \text{fun}(2)$$

$$+ \text{fun}(4) + \text{fun}(1)$$

n	1	2	3	4	5
fun(n)	1	2	5	15	51

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

$$+ \text{fun}(3) + \text{fun}(1)$$

$$= 1 + 1 \times 5 + 2 + 2 + 5 \times 1$$

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

$$= 15$$

$$\text{fun}(5) = n + \text{fun}(1) + \text{fun}(4) + \text{fun}(2) + \text{fun}(3)$$

$$+ \text{fun}(3) + \text{fun}(2)$$

$$+ \text{fun}(4) + \text{fun}(1)$$

$$= 1 + 1 \times 15 + 2 \times 5 + 5 \times 2$$

$$+ 15 \times 1$$

$$= 1 + 15 + 10 + 10 + 15$$

$$= \underline{\underline{51}}$$

$f(5)$ 

$$(2 \times 5)^{10} + f(3) \times f(2) + f(4) \times f(3)$$

$$5^{10} + f(2) \times f(3) + f(4) \times f(3)$$

$$5^{10}$$

 $f(4)$  $f(3)$  $f(2)$  $f(1)$  $f(0)$  $f(-1)$  $f(-2)$  $f(-3)$  $f(-4)$  $f(-5)$  $f(-6)$  $f(-7)$  $f(-8)$  $f(-9)$  $f(-10)$  $f(-11)$  $f(-12)$  $f(-13)$  $f(-14)$  $f(-15)$  $f(-16)$  $f(-17)$  $f(-18)$  $f(-19)$  $f(-20)$  $f(-21)$  $f(-22)$  $f(-23)$  $f(-24)$  $f(-25)$  $f(-26)$  $f(-27)$  $f(-28)$  $f(-29)$  $f(-30)$  $f(-31)$  $f(-32)$  $f(-33)$  $f(-34)$  $f(-35)$  $f(-36)$  $f(-37)$  $f(-38)$  $f(-39)$  $f(-40)$  $f(-41)$  $f(-42)$  $f(-43)$  $f(-44)$  $f(-45)$  $f(-46)$  $f(-47)$  $f(-48)$  $f(-49)$  $f(-50)$  $f(-51)$  $f(-52)$  $f(-53)$  $f(-54)$  $f(-55)$  $f(-56)$  $f(-57)$  $f(-58)$  $f(-59)$  $f(-60)$  $f(-61)$  $f(-62)$  $f(-63)$  $f(-64)$  $f(-65)$  $f(-66)$  $f(-67)$  $f(-68)$  $f(-69)$  $f(-70)$  $f(-71)$  $f(-72)$  $f(-73)$  $f(-74)$  $f(-75)$  $f(-76)$  $f(-77)$  $f(-78)$  $f(-79)$  $f(-80)$  $f(-81)$  $f(-82)$  $f(-83)$  $f(-84)$  $f(-85)$  $f(-86)$  $f(-87)$  $f(-88)$  $f(-89)$  $f(-90)$  $f(-91)$  $f(-92)$  $f(-93)$  $f(-94)$  $f(-95)$  $f(-96)$  $f(-97)$  $f(-98)$  $f(-99)$  $f(-100)$  $f(-101)$  $f(-102)$  $f(-103)$  $f(-104)$  $f(-105)$  $f(-106)$  $f(-107)$  $f(-108)$  $f(-109)$  $f(-110)$  $f(-111)$  $f(-112)$  $f(-113)$  $f(-114)$  $f(-115)$  $f(-116)$  $f(-117)$  $f(-118)$  $f(-119)$  $f(-120)$  $f(-121)$  $f(-122)$  $f(-123)$  $f(-124)$  $f(-125)$  $f(-126)$  $f(-127)$  $f(-128)$  $f(-129)$  $f(-130)$  $f(-131)$  $f(-132)$  $f(-133)$  $f(-134)$  $f(-135)$  $f(-136)$  $f(-137)$  $f(-138)$  $f(-139)$  $f(-140)$  $f(-141)$  $f(-142)$  $f(-143)$  $f(-144)$  $f(-145)$  $f(-146)$  $f(-147)$  $f(-148)$  $f(-149)$  $f(-150)$  $f(-151)$  $f(-152)$  $f(-153)$  $f(-154)$  $f(-155)$  $f(-156)$  $f(-157)$  $f(-158)$  $f(-159)$  $f(-160)$  $f(-161)$  $f(-162)$  $f(-163)$  $f(-164)$  $f(-165)$  $f(-166)$  $f(-167)$  $f(-168)$  $f(-169)$  $f(-170)$  $f(-171)$  $f(-172)$  $f(-173)$  $f(-174)$  $f(-175)$  $f(-176)$  $f(-177)$  $f(-178)$  $f(-179)$  $f(-180)$  $f(-181)$  $f(-182)$  $f(-183)$  $f(-184)$  $f(-185)$  $f(-186)$  $f(-187)$  $f(-188)$  $f(-189)$  $f(-190)$  $f(-191)$  $f(-192)$  $f(-193)$  $f(-194)$  $f(-195)$  $f(-196)$  $f(-197)$  $f(-198)$  $f(-199)$  $f(-200)$  $f(-201)$  $f(-202)$  $f(-203)$  $f(-204)$  $f(-205)$  $f(-206)$  $f(-207)$  $f(-208)$  $f(-209)$  $f(-210)$  $f(-211)$  $f(-212)$  $f(-213)$  $f(-214)$  $f(-215)$  $f(-216)$  $f(-217)$  $f(-218)$  $f(-219)$  $f(-220)$  $f(-221)$  $f(-222)$  $f(-223)$  $f(-224)$  $f(-225)$  $f(-226)$  $f(-227)$  $f(-228)$  $f(-229)$  $f(-230)$  $f(-231)$  $f(-232)$  $f(-233)$  $f(-234)$  $f(-235)$  $f(-236)$  $f(-237)$  $f(-238)$  $f(-239)$  $f(-240)$  $f(-241)$  $f(-242)$  $f(-243)$  $f(-244)$  $f(-245)$  $f(-246)$  $f(-247)$  $f(-248)$  $f(-249)$  $f(-250)$  $f(-251)$  $f(-252)$  $f(-253)$  $f(-254)$  $f(-255)$  $f(-256)$  $f(-257)$  $f(-258)$  $f(-259)$  $f(-260)$  $f(-261)$  $f(-262)$  $f(-263)$  $f(-264)$  $f(-265)$  $f(-266)$  $f(-267)$  $f(-268)$  $f(-269)$  $f(-270)$  $f(-271)$  $f(-272)$  $f(-273)$  $f(-274)$  $f(-275)$  $f(-276)$  $f(-277)$  $f(-278)$  $f(-279)$  $f(-280)$  $f(-281)$  $f(-282)$  $f(-283)$  $f(-284)$  $f(-285)$  $f(-286)$  $f(-287)$  $f(-288)$  $f(-289)$  $f(-290)$  $f(-291)$  $f(-292)$  $f(-293)$  $f(-294)$  $f(-295)$  $f(-296)$  $f(-297)$  $f(-298)$  $f(-299)$  $f(-300)$  $f(-301)$  $f(-302)$  $f(-303)$  $f(-304)$  $f(-305)$  $f(-306)$  $f(-307)$  $f(-308)$  $f(-309)$  $f(-310)$ 

&lt;math

## Arrays:

CS Academy

- 1) What is an Array?
- 2) Declaring & Initializing
- 3) Accessing Array

### Array

Scalar  $\rightarrow$  `int n = 10;`

10,
100/101

Array  $\rightarrow$  Collection of elements of same type (Data type)

$\hookrightarrow$  Storage list of values

`int A[5];`



Vector

(1-Dimension)

A	1	15		
0	1	2	3	4

`A[2]=15;`

$\hookrightarrow$  memory

$\hookrightarrow$  contiguous  
- Side-by-side

$\hookrightarrow$  memory

$\hookrightarrow$  contiguous  
- Side-by-side

### Initializing:

(Declaration)

0	1	2	3	4
0	1	2	3	4

(Declaration + Initialization)

`int A[5] = {2, 1, 4, 6, 8, 10};`

Garbage

0	1	2	3	4
2	1	4	6	8

`int A[5] = {2, 1, 4, 6, 8};`

Garbage

0	1	2	3	4
2	1	4	6	8

`int A[] = {2, 1, 4, 6, 8, 10, 12};`

Garbage

0	1	2	3	4
2	1	4	6	8

int A[5] = {2, 5, 4, 9, 18};

Printf("%d", &A[0]);  
printf("%d", \*(&A));  
printf("%d", \*(A+1));  
printf("%d", A[1]);

Printing address  
size of array?

## Static vs Dynamic Array

Static Array - size of the array is static  
Dynamic Array - Size of the array is Dynamic

Valid main C :-

int A[5]; → size memory of variable  
allocated in AR  
of main in stack.

↳ Static - Because size of the array  
decided at compile time,

but the memory will be  
allocated during run time -

only size of the array  
decided at run time -

④ class

In C++, we can create an array of any size at run time. → created in Stack only.

CS Academy

```
int n; // user input n to size  
cin >> n; // taken off stack  
int A[n]; // created on stack
```

Creating Array in Heap:

↳ Size & type decided at run time.

↳ We need pointer for it.

```
void main () {  
    int A[5]; // A[0]=5  
    int *p; // p[0]=5;
```

```
C++ p = new int [5];
```

```
C p = (int*) malloc (5 * sizeof(int));
```

malloc only allocates raw memory (block of memory)

↳ To allocate it as an integer use typecast it.

If there is no use of heap memory, it must be deleted. Or else Memory leak problem.

```
--> C++ delete [] p;
```

```
C free (p);
```

# How to create Array Size?

Size of an array cannot be given because the memory of the array is contiguous. [The next memory location may not be free]

We can create a bigger array and transfer the elements from the smaller array.

```
int *P = new int[3];
```

```
int *Q = new int[7];
```

```
for(i=0; i<3; i++)
```

```
    Q[i] = P[i];
```

```
}
```

```
delete []P;
```

P = Q;

Q = null