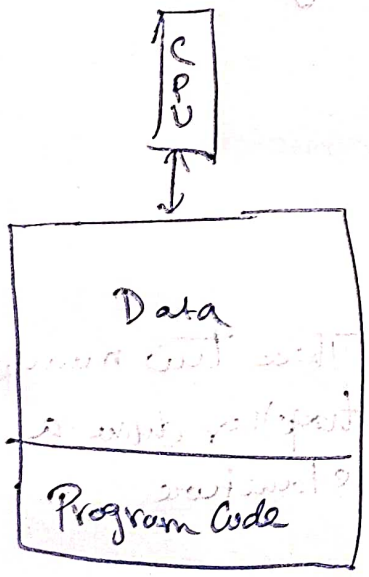
2.

Data Structures & Algorithms with C, C++ CS Academy

Physical DS
- Arrays
- Matrices
- linked lists

logical DS
- Stack
- Queue
- Trees
- Graph
- Hashing
- Recursion
- Sorting



C & C++ Concepts

Arrays

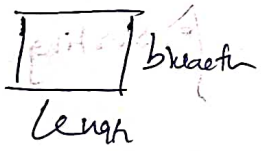
```
int main() {
    int A[5]; // Declaration of Array
    int b[5] = {2, 4, 8, 7, 10}; // Initialize array
    // Declaration   // Initialization
```

```
int i;
for(i=0; i<5; i++){
    printf("%d\n", B[i]);
}
```

Stack is above,
Heap is below:
Just for understanding purposes → This layout.

Structure:

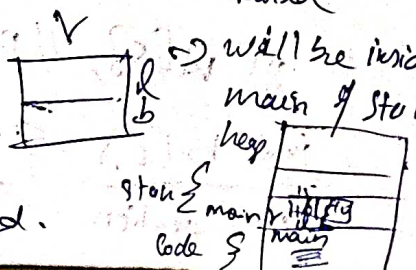
→ User defined data types




```
Struct Rectangle{
    int length;
    int breadth;
};
```
→ These two members together define a structure.

Size of Structure → Total amount of memory consumed by all it's members

```
int main(){
    Struct Rectangle r;  → // Declaration of structure
    // A memory space will be created
    Struct Rectangle r={10,5};  // Declaration + initialization
    r.length = 15;
    r.breadth = 10;
    // To read or write the member of a Structure, dot operator is used.
```



printf("Area of Rectangle is %d", r.length * r.breadth);

CS Academy

1. Complex numbers

a+ib , $i=\sqrt{-1}$

```
struct complex {
    int real;
    int imag;
};
```

2. Student

```
struct Student {
    int roll;
    Char name[25]; // character space of 25 bytes in memory
    Char dept[10];
    Char address[50];
};
```

Each character takes 1 Byte.

```
Struct student s;
s.roll = 10;
```

Playing Cards

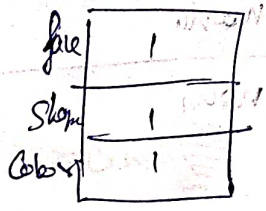
Face value, Shape - Hearts, Color - Red } Properties

Face → 1, 2 .... 10, J, Q, K
11 12 13

Shapes :- Hearts, Diamond, Spades, clubs
0 or 1 1 2 3

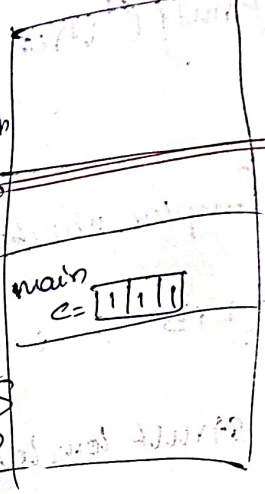
Color → 0, 1
black red

```
Struct Card {
    int face;
    int shape;
    int Color;
};
int main() {
    Struct Card c;
    c.face = 1;
    c.shape = 1;
    c.color = 1
```



OR

Struct Card c = {1,1,1}



int main() {

Declaration Struct Card deck[52];

// This will form an array of structures (52 such structures) (312 bytes if size of (int) = 2 bytes)

// Declaration + initialization

Struct Card deck[52] = {{1,1,1}, {10,2,0}, {11,3,0}, ... {11,1,1}} (51)

Printf("%d", deck[0].face);

printf("%d", deck[0].shape);

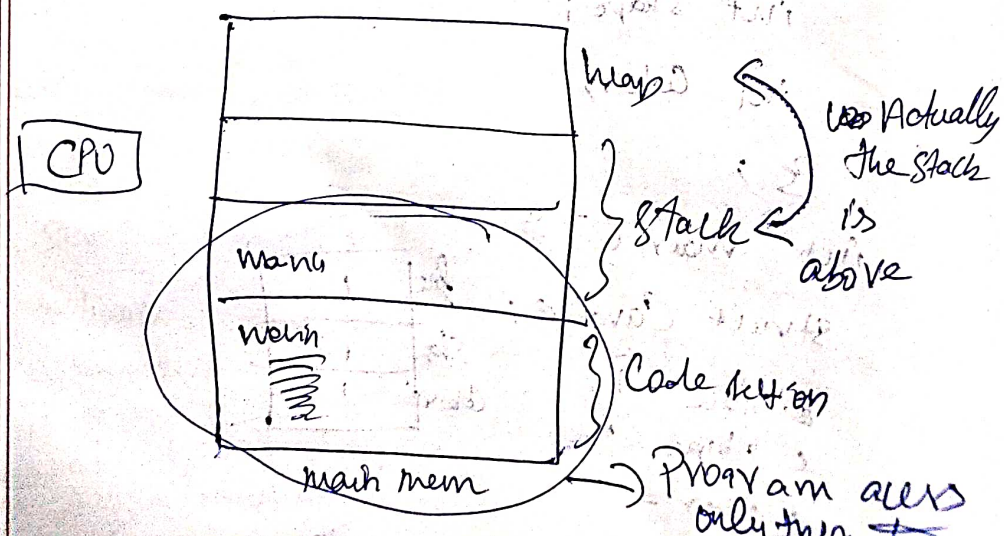
Pointers:

// Pointers are the address of the Data & not the Data itself //

"They store the address of the data"

Normal variables → DATA variables

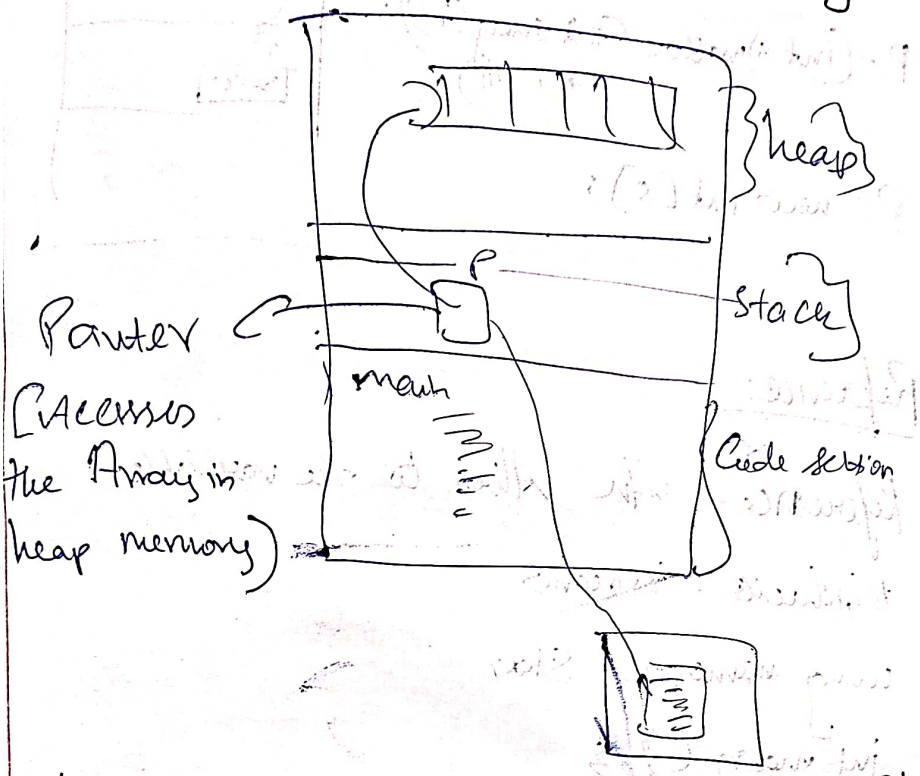
Pointer variables → Address variables



- Program will not automatically access heap.

Heap memory → external to the program

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

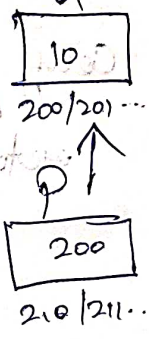


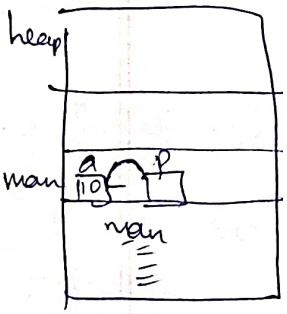
[With pointers, we access the external files]

Major uses of pointers:

1. Accessing heap
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", *p);
                ↑ Dereferencing
```





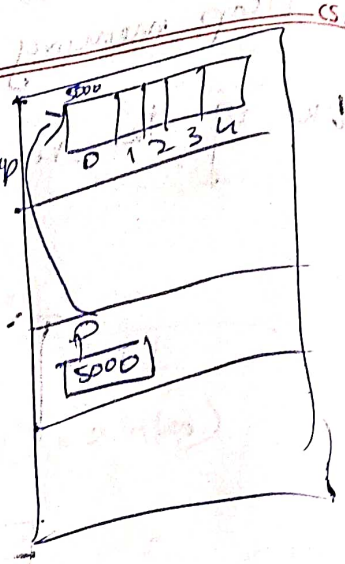
* → Also for * pointer declaration

malloc() → creating memory in heap

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p;
```

//Clang => `p = (int*)malloc(5 * sizeof(int));`
`5 * (2/4)`

//C++ → `p = new int[5];`

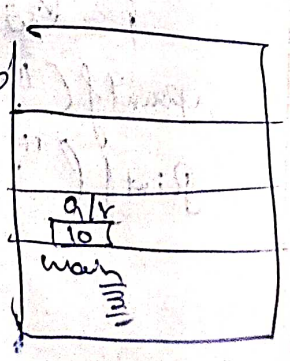


Reference:

Reference → with Alias to the variable

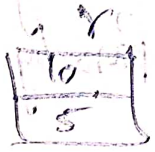
```
#include <iostream>
using namespace std;
int main() {
    int a = 10;
    int &r = a; // Reference must be Initialized
    cout << a << endl << r << endl;
    // Same output 10
    int b = 25;
    r = b;
    cout << a << endl << r << endl;
    // Both values of 25
    return 0;
}
```

r is a reference to a



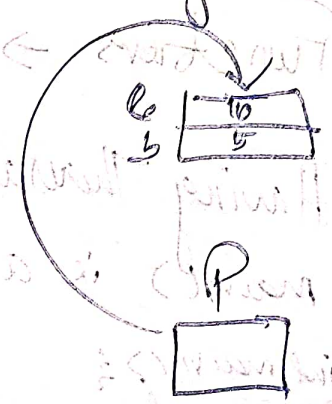
Pointer to a structure:

```
Struct Rectangle
{
    int length;   - 4
    int breadth;  - 4
};                - 8
```




```
int main()
{
    Struct Rectangle r = {10,5};
    Struct Rectangle *p = &r;
    r.length = 10; //Accessing using r structure
    (*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;
};
int main()
{
    Struct Rectangle *p;
    p = (Struct Rectangle *)malloc(sizeof(StructRectangle));
```

typecast -> 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++)

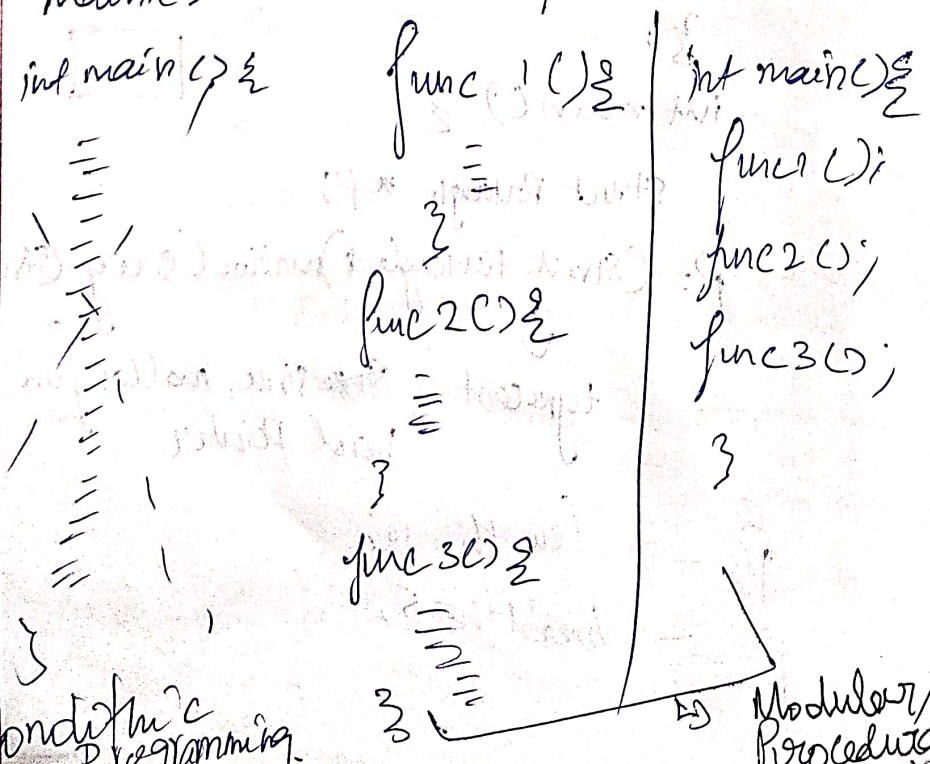
Function → Part of a program where the code functions a specific task.
[Set of instructions for specific task]

Grouping Data ⇒ Structures

Grouping Instructions → Functions

Functions ⇒ modules/Procedures

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

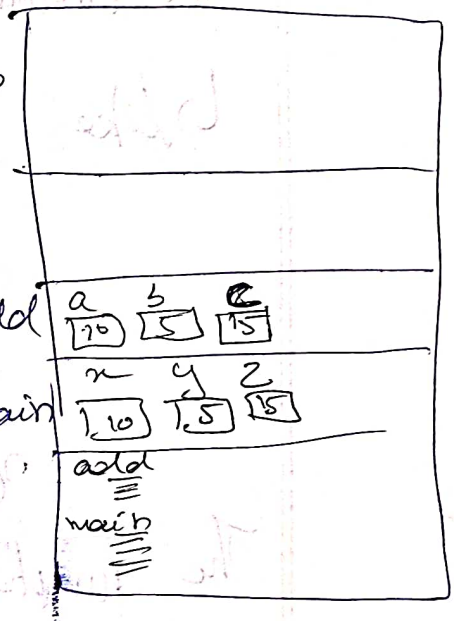


Re-usability:
Using a function to perform a task again and again by just calling the function fun() ...

Prototype / Header / Signature function

```
int add(int a, int b){
    int c;
    c = a + b;
    return c;
}
int main(){
    int x, y, z;
    x = 10;
    y = 5;
    z = add(x, y);
    printf("%d\n", z);
    return 0;
}
```

10, 5 → Formal Parameters



Variables of one function cannot be accessed by the other.

- 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 if the function is returning some results."

a=10 b=20

↓ ↓

↳ Like void swap(int x, int y){

int temp;
temp = x;
x = y;
y = temp;

}

The function returns no value.
Only the formal parameters x, y is changed - a & b remains unchanged.

" Returns an output/result : Greatest number in arrays, finding max of two, adding ..., Palindrome...)

Call by Address:

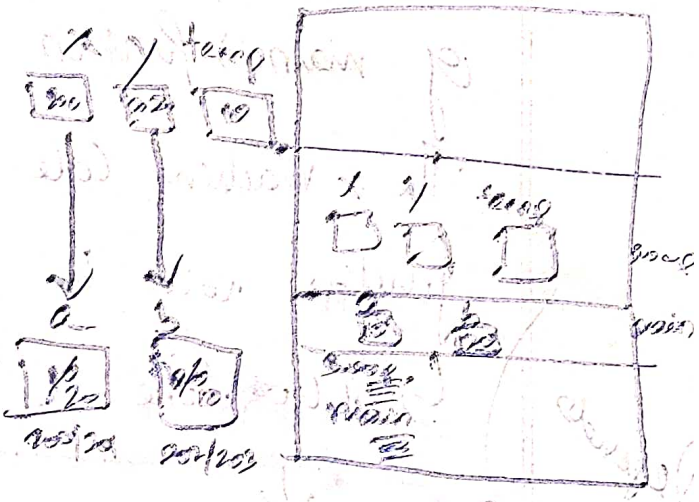
" The address of the actual parameters are passed to formal parameters & formal parameters must be Pointers.

Any changes done inside function will modify actual parameters.

" One function can access the variable of another using pointers. "

CS Academy

```
void swap(int *x, int *y){
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
int main(){
    int a, b;
    a = 10;
    b = 20;
    swap(&a, &b);
    printf("old %d %d", a, b);
    return 0;
}
```



swap(&a, &b); → the address of a & b

Call by Reference:

Only in C++

" When formal parameters are changed, the actual parameters are modified. "

→ 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 (formals parameters) of the swap function is created under the activation record of main() function.

∴ The machine code of the swap function will be pasted to main()
↳ Above like a monolithic program.

(Depends on compiler (not mandatory))

The machine code → non modular

Source code → Modular / Procedural

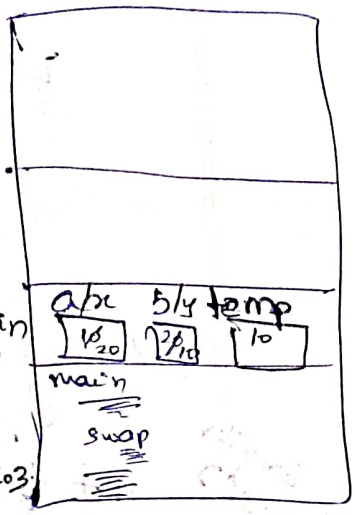
NOTE: Call by reference should not be used frequently or for heavy functions.

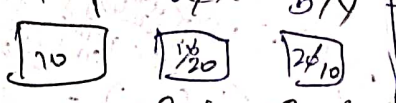
It should be used carefully

For one or two small functions, it can be used.

```
Void swap (int &x, int &y)        // reference to actual parameter / Alias
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
int main ()
{
    int a, b;
    a = 10;
    b = 20;
    swap (a, b);
    printf ("%d %d \n", a, b);
    return 0;
}
```

machine code (pasted into main())





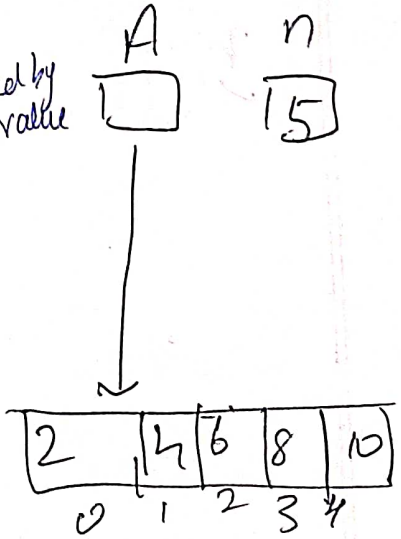
Array as Parameter:

Arrays can be passed only by Address.
(Both in C or C++)

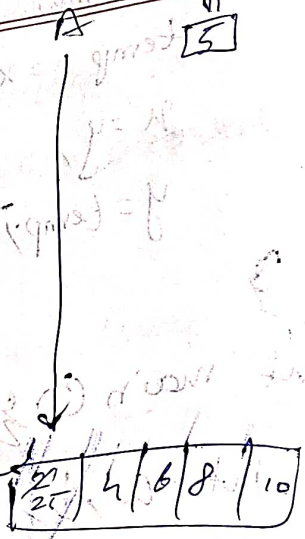
Pointer to an Array
A[] -> specific to Array

```
void func (int A[], int n)
{
    (or *A -> pointer variable to integer array)
    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);
}
```

Called by value



```
void func(int A[], int n){
// Array values/elements can be
Changed as well
A[0] = 25;
}
```



Returning an array:

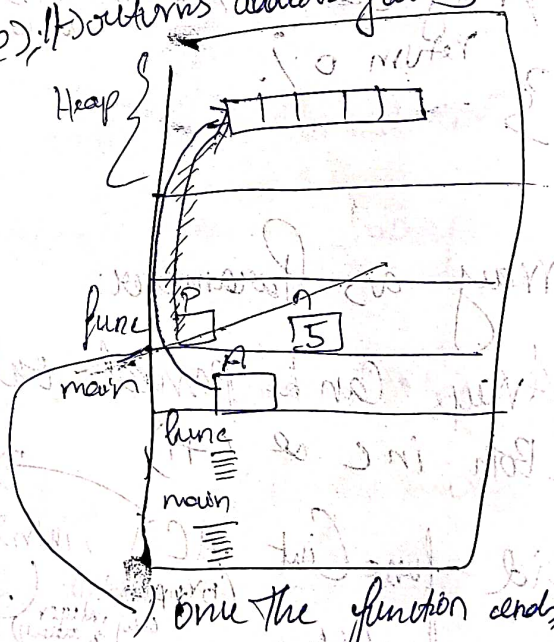
-> [] -> only pointer for array of elements

* -> Pointer to an element or array of elements

Some compiler won't allow this *

returning function returning an array

```
int *func(int n){
int *p;
p = (int *) malloc (n * sizeof(int));
return(p); // returns address of array
}
int main(){
int *A;
A = func(5);
≡≡≡
return 0;
}
```



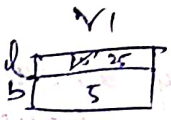
Once the function ends, the activation record is deleted.

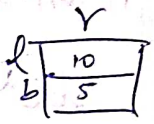
Structure as parameter:

- 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};
    ...
    printf("%d\n", area(r));
}
```

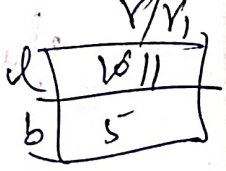




Call by reference:

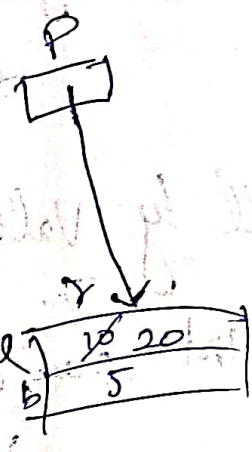
reference to r

```
int area (struct Rectangle &r1){
    r1.length ++;
    return r1.length * r1.breadth;
}
int main(){
    ...
}
```



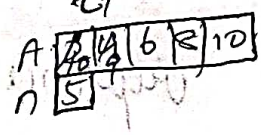
Call by address:

```
void changelength (struct Rectangle *p, int l){
    p->length = l;
}
int main(){
    struct Rectangle r = {10,5};
    changelength(&r, 20);
```

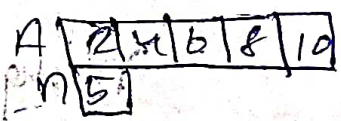


Array as datamember in a Struct:
Where a structure is passed as value, even if it as an array as Datamember, it can be passed by value to functions.

```
struct Test {
    int A[5];
    int n;
};
void func(struct Test t1){
    t1.A[0] = 10
    t1.A[1] = 9;
}
```



→ This could be changed.




```
int main(){
    struct Test t = {{2,4,6,8,10},10};
    func(t);
```

Structures & functions:

```
struct Rectangle {
    int length;
    int breadth;
};
```

```
void initialize (struct Rectangle *p, int l, int b) {
    p -> length = l;
    p -> breadth = b;
}

int area (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); // 50 units
    changeLength (&r, 20);
    area (r); // 100 units
    return 0;
}
```

Classes & Constructors

Converting C → C++

```
class Rectangle {
    =
    =
    =
    =
    =
    =
    =
    =
};
```

1) If just class is written, all members become private.

2) Struct → All members are public.

```
Struct Rectangle {
    int length;
    int breadth;
    void initialize (int l, int b);
};
```

```
    =
    =
    =
    =
    =
    =
};
```

Class & Constructor - continued

:: -> Scope resolution operator

↳ Use Cases:

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:
        Rectangle() { length = breadth = 1; } -> Default Constructor
        Rectangle (int l, int b) { }; -> Parameterized Constructors
        int area();
        int perimeter()
        int getLength() { return length; }
        void setLength (int l) { length = l; }
        ~Rectangle(); -> // Destructor : To destroy anything
};                      no arguments   // Dynamic memory allocation inside Constructor - release/destroy that memory
```

Overloaded Constructors

Facilitators

Accessor/getter function

mutator/setter function

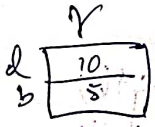
Destructor

```
Rectangle::Rectangle(int l, int b){
    length = l;
    breadth = b;
```



```
int Rectangle::area(){
    return length * breadth;
}
int Rectangle::perimeter(){
    return 2*(length + breadth);
}
Rectangle::~Rectangle(){
} // Dy de-allocation (like dynamic memory)
```

is released, destructor is called with class name & scope resolution operator.




```
int main(){
    Rectangle r(10,5);
    cout << r.area();
    cout << r.perimeter();
    r.setLength(20);
    cout << r.getLength();
    return 0;
}
```

Once the main() ends, the Destructor will be automatically called, the object will be destroyed.

Template classes:

Generic class => Suitable for any data type, one data type at a time.

[Use the same class for multiple data types]

```
#include <iostream>
using namespace std;
template<class T>
class Arithmetic {
  private:
  T a;
  T b;
  public:
  Arithmetic(T a, T b);
  T add();
  T sub();
};
template<class T>
Arithmetic<T>::Arithmetic(T a, T b)
{
  this->a=a;
  this->b=b;
}
template<class T>
T Arithmetic<T>::add()
{
  T c;
  c=a+b;
  return c;
}
```

Any data type

```
int main() {
    Arithmetic a1(10,15);
    cout << a1.add() << endl;
    ...
}
```

INTRODUCTION:

Data Structures:

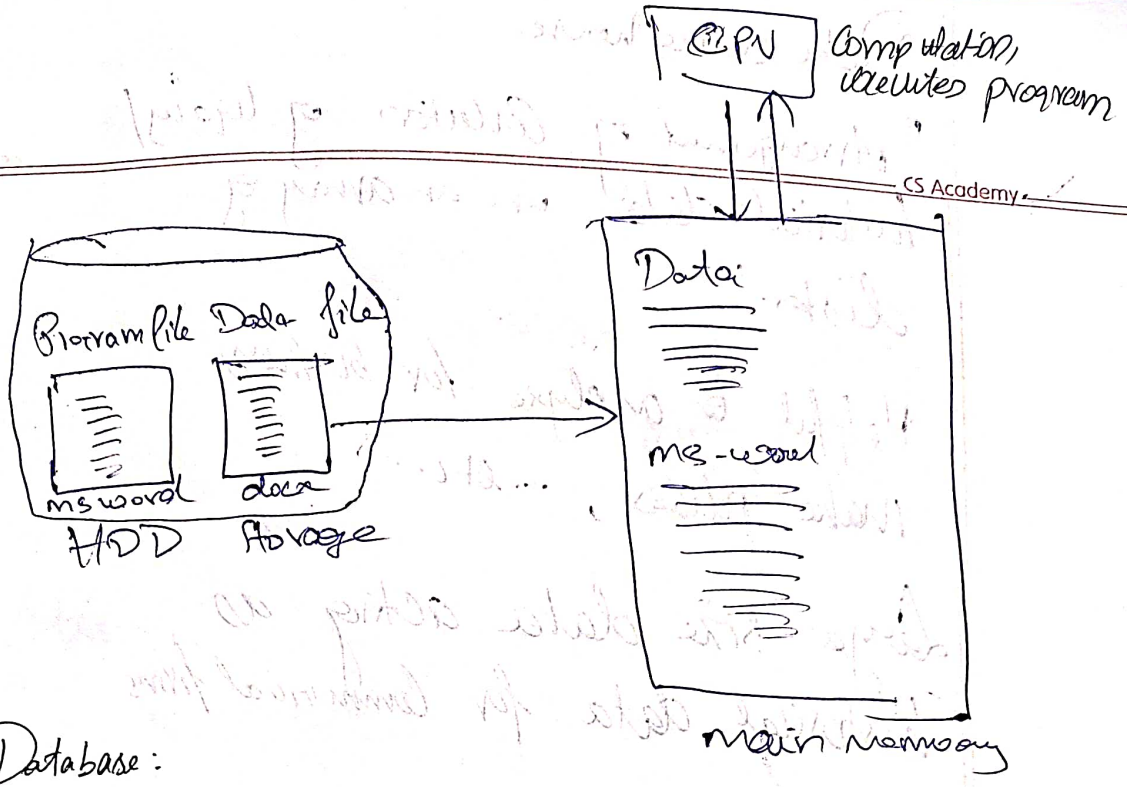
"Arrangement of Collection of data items so that they can be utilized efficiently & the operations on the data can be done efficiently."

All about efficient arrangement & operation 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.

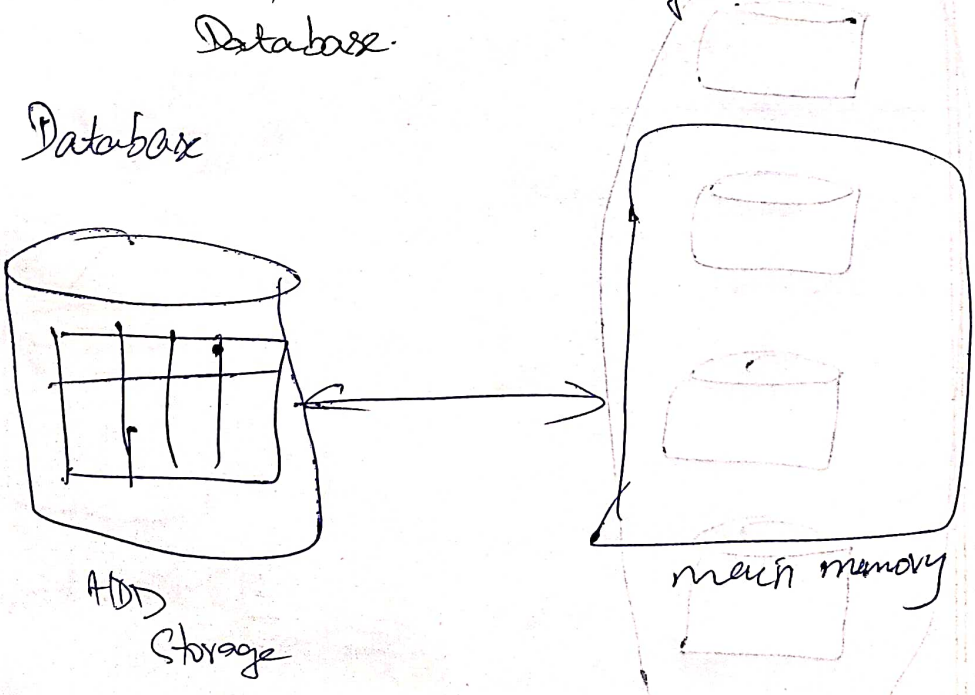
C Faster



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.

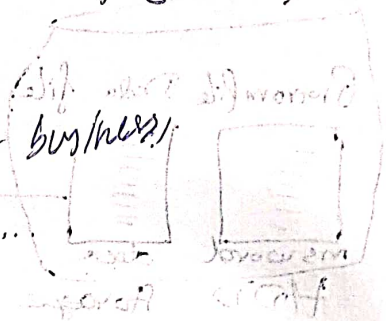
↳ This type of arrangement of data in permanent storage is called Database.



Data warehouse.

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

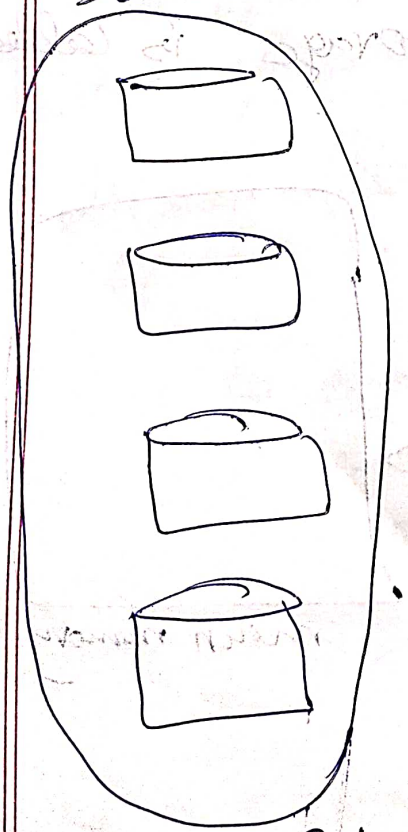
Helpful to analyse for business, make policies, ...etc...

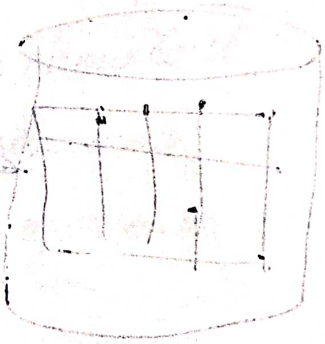


Large size data acting as historical data for Commercial firms.

Algorithms written for analyzing these data are called Data mining Algorithms.

Data warehouse





Big data: (Internet)
Storing & analyzing very large size of data

Stack vs Heap memory :-

1.) About main memory

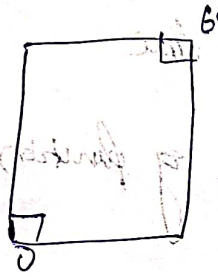
2.) How a program uses main memory

3.) Static Allocation

4.) Dynamic Allocation

Static vs Dynamic memory Allocation

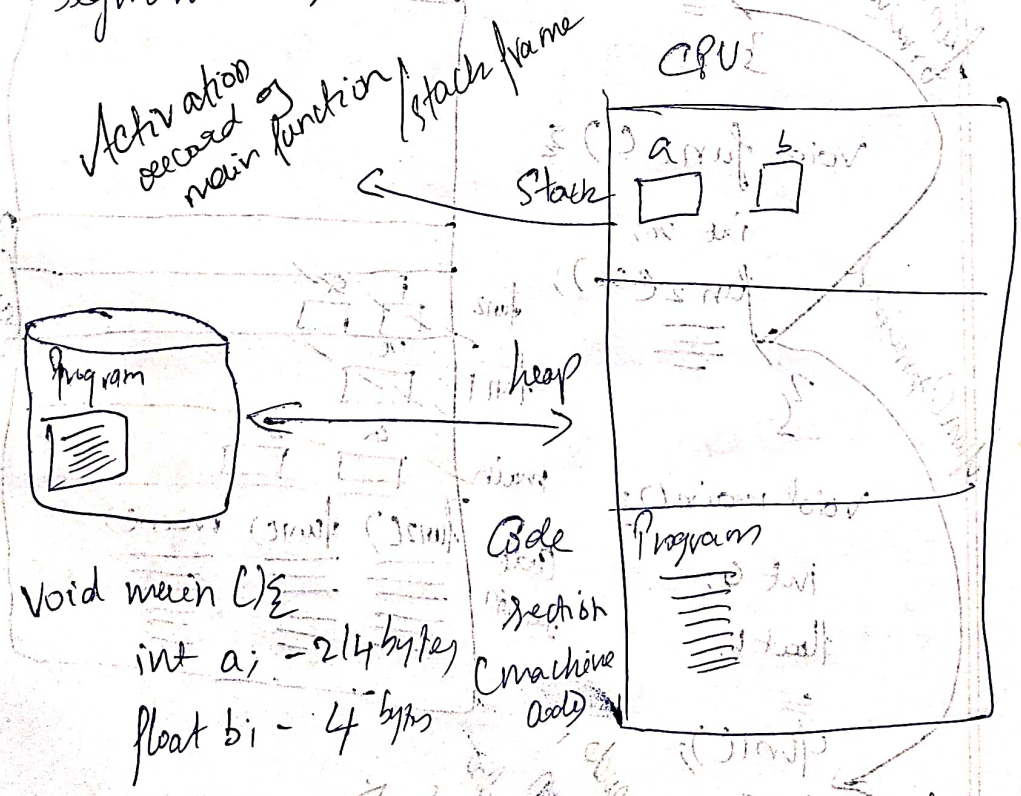
Address :- single dimension / number



64 X 1024 = 64 KB

0 - 65535

Memory will be divided into a manageable segment. like 64KB



Void main ()
{
int a; - 2/4 bytes
float b; - 4 bytes

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

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

How many bytes of memory is reserved by this function was decided at Compile time

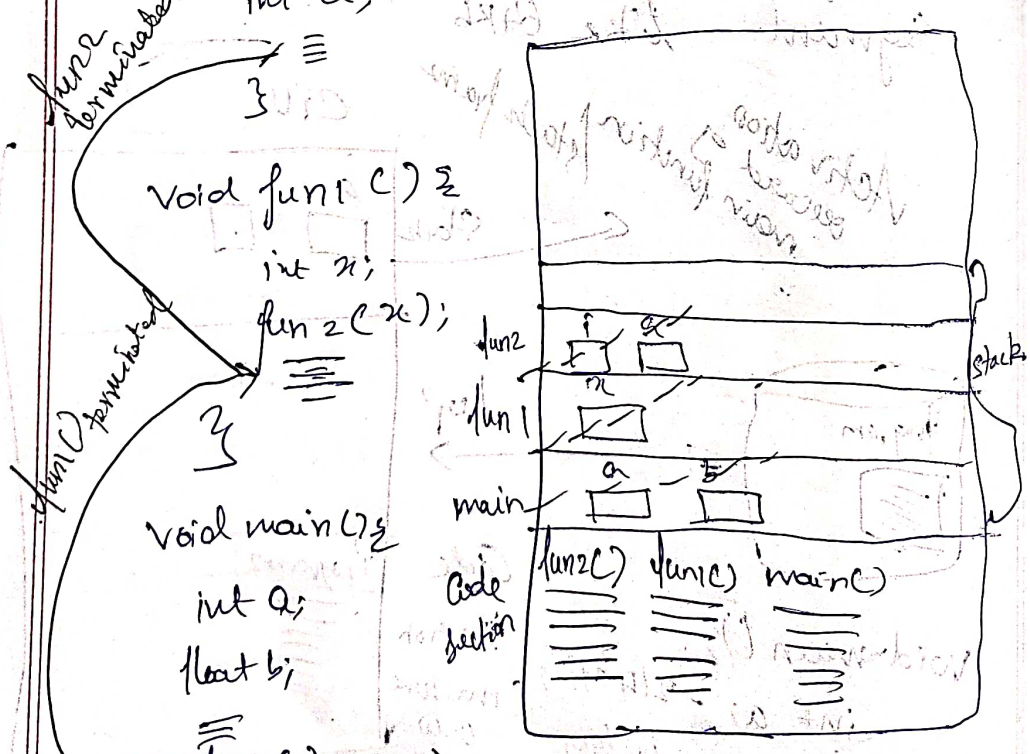
Size of the memory - Static
↳ Decided at Compile time
/ before run time

Memory allocation for sequence of function calls:

```
Void fun2(int x){
    int a;
    ≡
}
Void fun1(){
    int x;
    fun2(x);
    ≡
}
Void main(){
    int a;
    float b;
    ≡
    fun1();
    ≡
}
```

fun2 terminated

fun1() terminated



One main code, its activation record is also deleted from main memory.

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 / unorganized, like a tower.

Heap should be treated as a resource.

Heap ⇒ Dynamic memory allocation.

Heap memory can only be accessed by pointers

```
int *p;
```

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

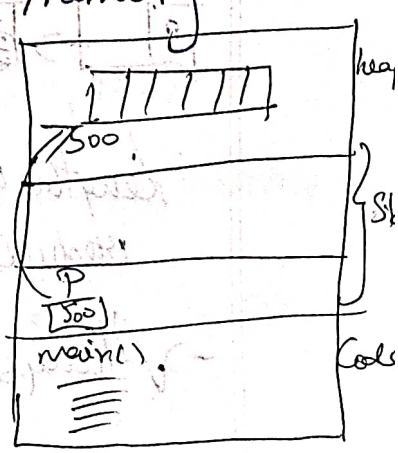
C-lang `p = (int *) malloc(20*4);` (20*4 = 2*(5))

Heap memory should always be freed after utilizing it ⇒ Good Practice.

If failed to free allocated memory

1) memory leak.

```
delete[] p;
p = NULL
//or, free(p)
```

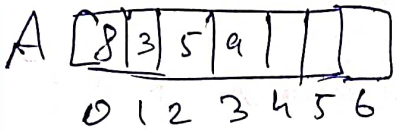


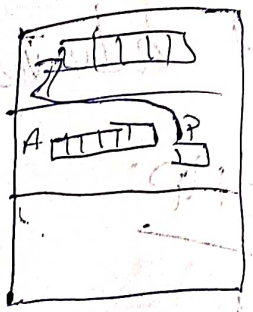
Types of Data Structures:

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

→ " " " heap with pointers.

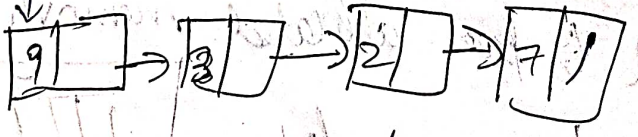
→ If we know the length/size of the list → we can go for arrays.

2) Linked lists

→ Dynamic Data structure

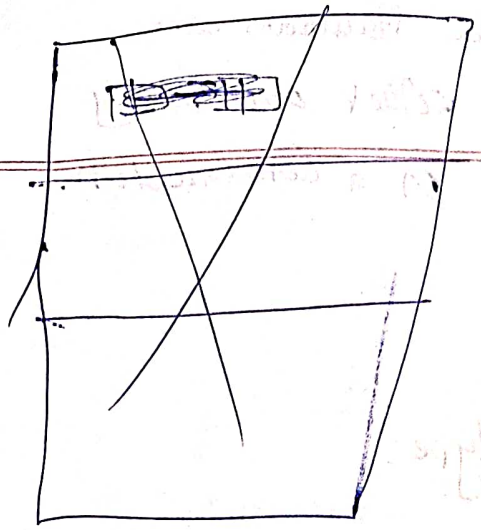
→ Collection of nodes ~~linked with~~ where each node contain data & is linked with other node

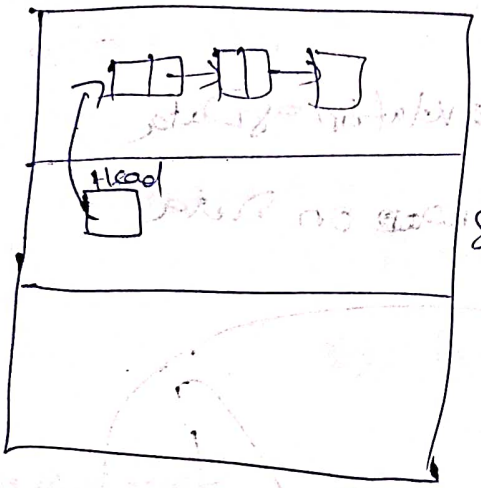
Head



Length of this list can grow or reduce, dynamically.

→ Always created inside Heap





Logical Data Structures: (used in algorithms)

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

Stack → last In First Out
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..

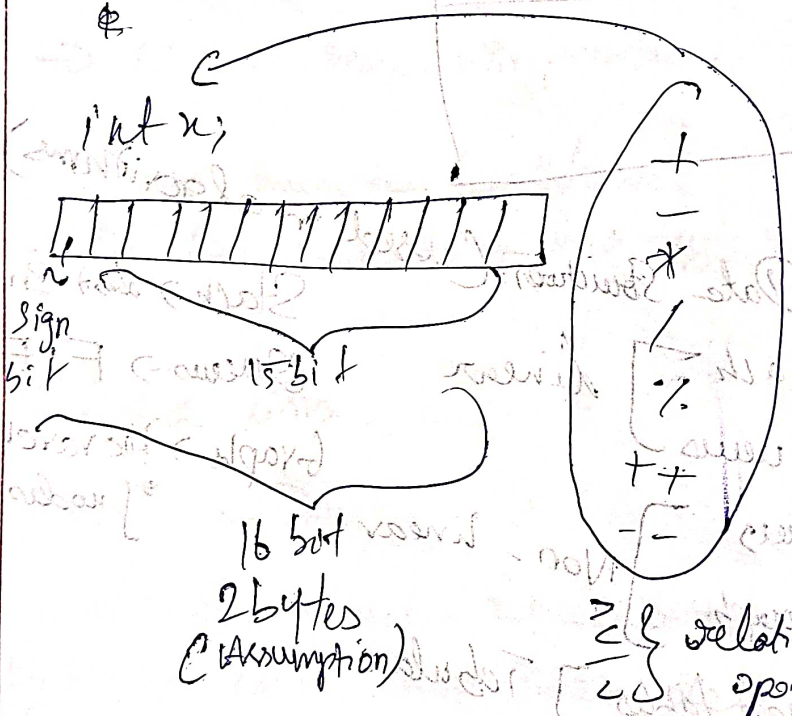
ADT:

Abstract Data type:

Data type:

1) Representation of Data

2) Operations on Data



The internal details are hidden (Abstract)

How the operations are performed in binary form in main memory

List → 8, 3, 9, 4, 6, 10, 12
0 1 2 3 4 5 6

How can the list be represented.

Data:
1.) Space for storing elements
2.) Capacity (max capacity)
3.) Size of a 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 are working internally → Abstract.
We will represent Logical DS as ADT.

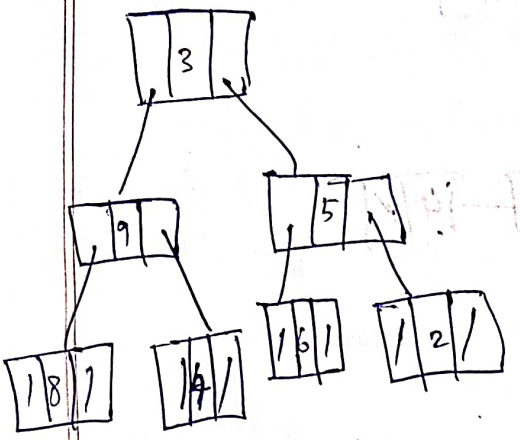
List ⇒ 8, 13, 9, 4, 6, 10, 12, 15 → 8
0, 1, 2, 3, 4, 5, 6 (add)

- add(Element) // append(elem)
- add(index, element) // insert(index, elem)
- remove(index) // list ~~adds~~ re-arranged
- set(index, element) // replace(index, elem)
- get(index) // index 5 → 10
- search(key) // contains(key)

Whether key element is there in the list

- sorted()
- reverse()
- compile()
- merge()
- split()

Time & Space Complexity:



$O(\log n)$

for (i = n; i > 1; i = i/2) {

≡

3

↳ $\log_2 n$

↳ i = n;
while (i > 1) {
i = i/2;
}
}

1) Searching thru all nodes, $O(n)$

$O() \rightarrow$ order ()

---

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

≡

}
}

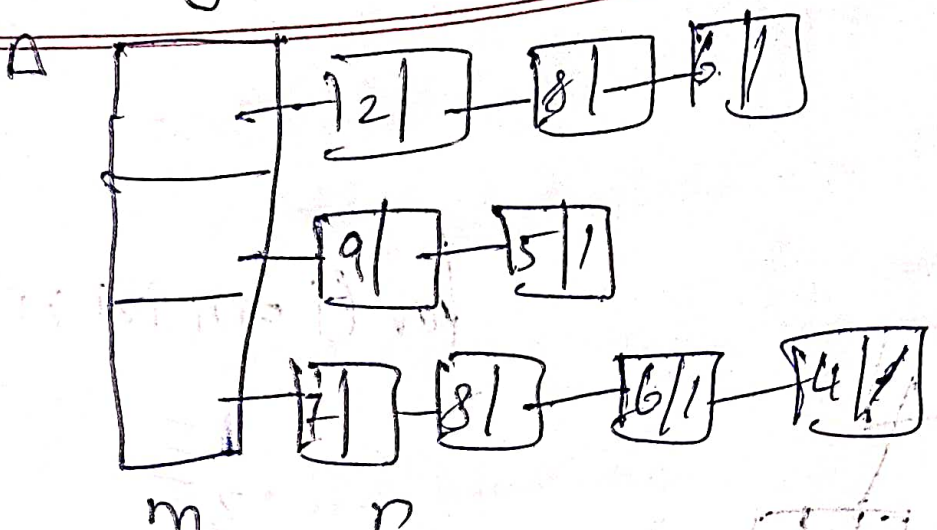
↳ $O(n^2)$ nested for loop over n element

Matrices $\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$ → Of all elements

$n \times m$ - $k \times k$

↳ $O(n^2)$

Array of linked lists:



$O(m+n)$

Space Complexity:-

Space Assumed in main memory

O( ), n elements?

↳ O(n) elements

↳ Does not mean bytes.

Refers to, The space is dependent on What?

Recursion:

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){
    ===
}

int main(){
    ===
    func(2)*2; => The (*2) multiplication by 2 will
    ===          done once the function has a
}                returned with some value.
```

---

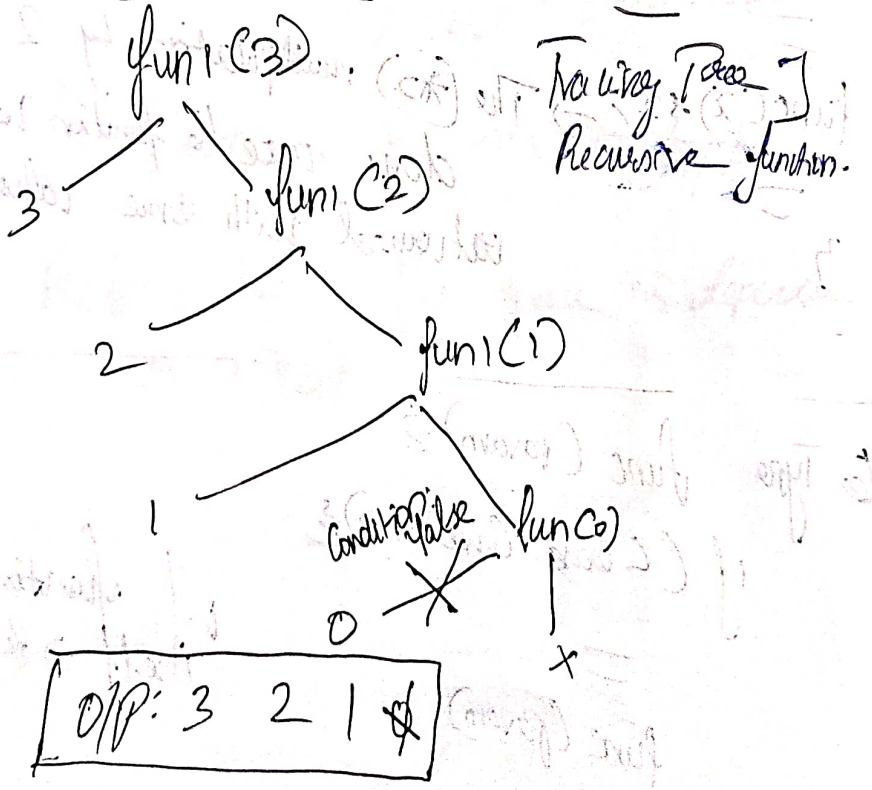
6 Type

```
func(param){
    if(<base condition>){
        ===
        func(param)
        ===
    }
}
```

A function calling itself -> Recursion.

Recursive functions are traced in the form of a Tree.

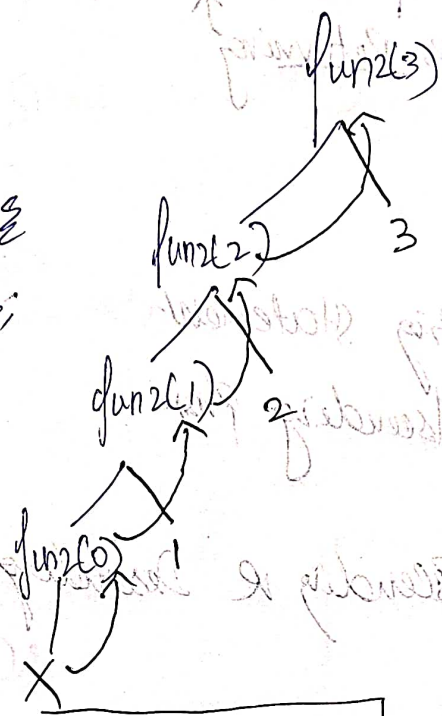
```
void fun(int n){
    if (n>0){
        printf("%d",n);
        fun(n-1);
    }
}
int main(){
    int x=3;
    fun(3);
    return 0;
}
```



Tracing Tree
Recursive function.

"Printing done at Calling time".

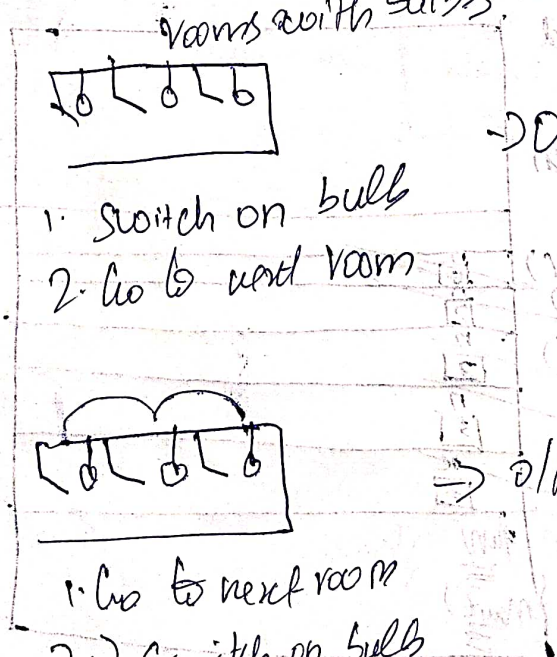
```
void fun2(int n){
    if(n>0){
        fun2(n-1);
        printf("%d\n", n);
    }
}

int main(){
    int x=3;
    fun2(x);
    return 0;
}
```



O/P: 1 2 3

"Printing done at returning time."

1st Case:



→ O/P: 1 2 3

2nd case

⇒ O/P: 3 2 1

Recursion has two phases:
- ⇒ Calling Phase
- ⇒ Returning Phase.

```
Void fun (int n) {
    if (n > 0) {
        1. Calling
        2. fun(n-1) * 2;
        3. Returning
    }
}
```

Ascending ← 1. Calling

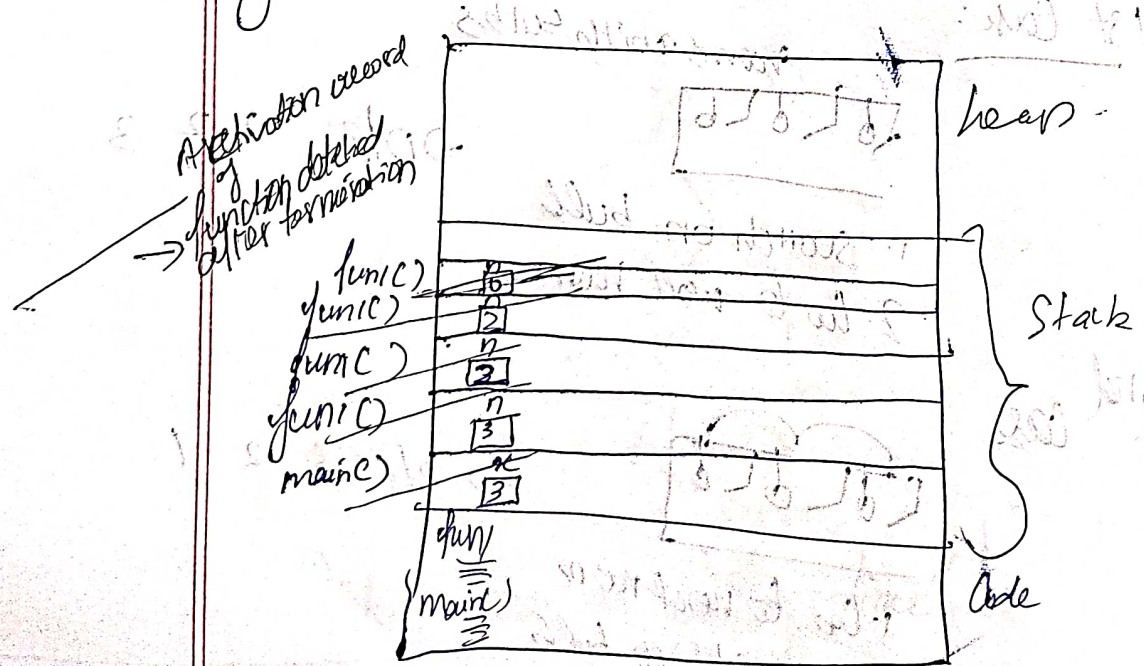
Descending ← 3. Returning

Loops → Repeating statement
↳ only Ascending phase

Recursion:
↳ Ascending & Descending phases

How Recursion uses Stack ?:

Taking Previous Recursion code:



For n elements/value/data..

↳ n+1 calls / function calls / activation record in Stack.

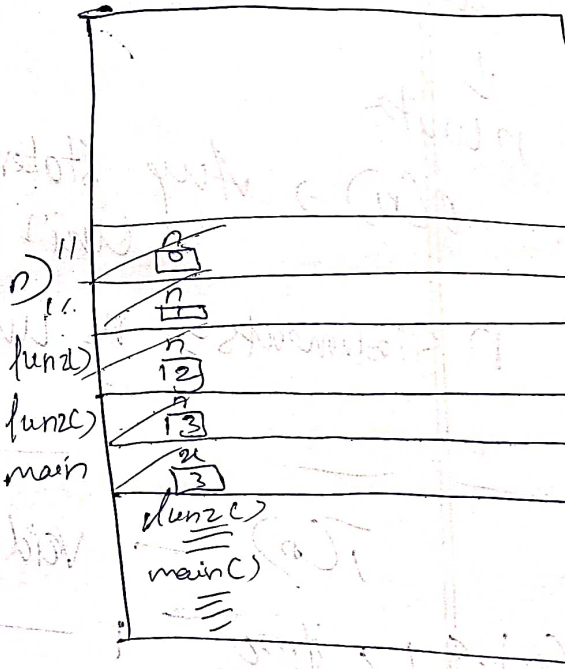
↳ O(n), Polynomial degree - 1

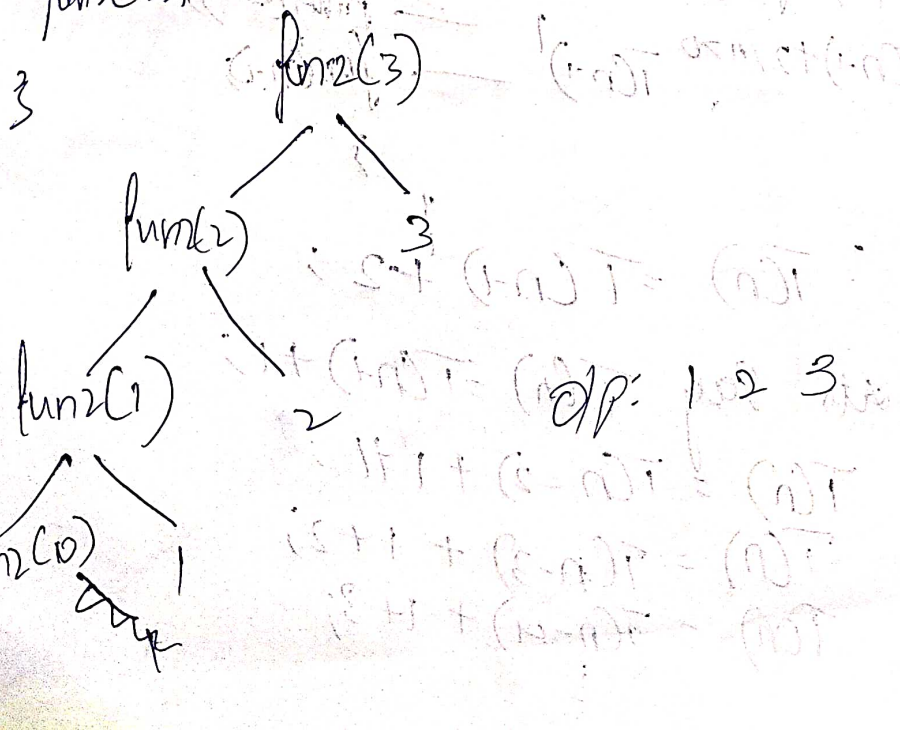
↳ Space Complexity.

Total AR ⇒ n+1

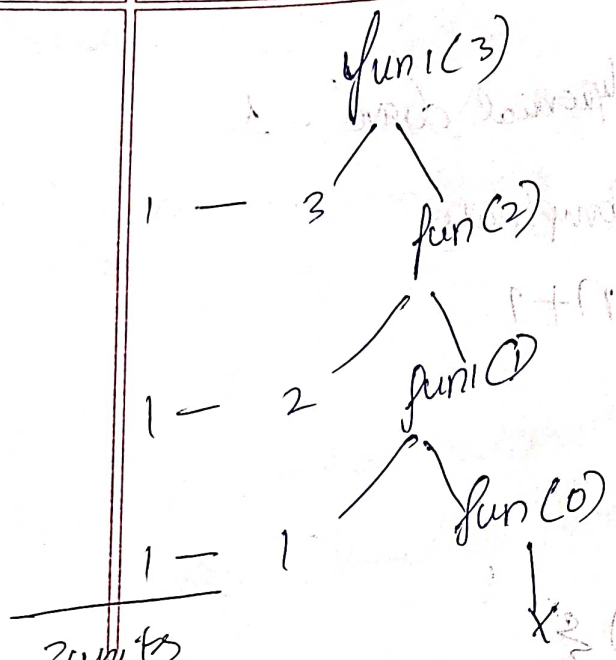
(Activation record)

```
void fun2(int n){
    if(n>0){
    1:) fun2(n-1)
    2:) printf("%d\n", n);
    }
}
void main(){
    int x = 3;
    fun2(x);
}
```





Recurrence
Recursion Relation
↳ Time Complexity of Recursion



3 units

↓

n units

O(n) ⇒ Any statement take one unit of time

n statements ⇒ n units of time

T(n) — void fun1(int n){

1 — if(n>0){

T(n-1) — fun1(n-1);

}

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

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

Let's say $T(n) = T(n-1) + 1;$

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

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

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

⋮

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

If $T(n-k) = 0$

$\Rightarrow n = k$

$\therefore T(n) = T(n-n) + n$

$= T(0) + n$

$T(n) = 1 + n$

$\Rightarrow O(n)$ → Time taken

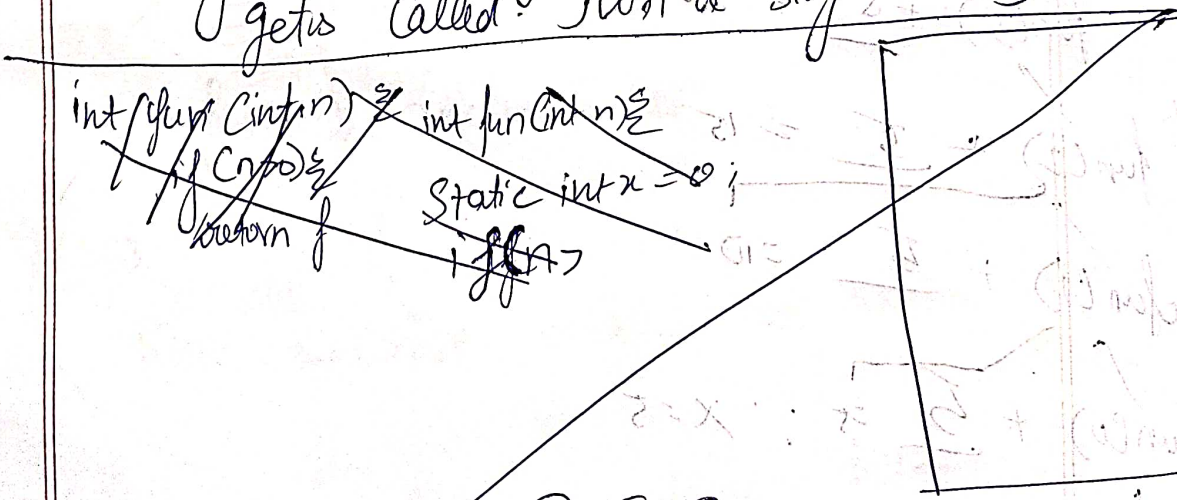
- Printing for n times or no. of calls $\Rightarrow O(n+1)$

Static Variables in Recursion:

Static Variables are created in the subsection of code section called Static variables or Global variables.

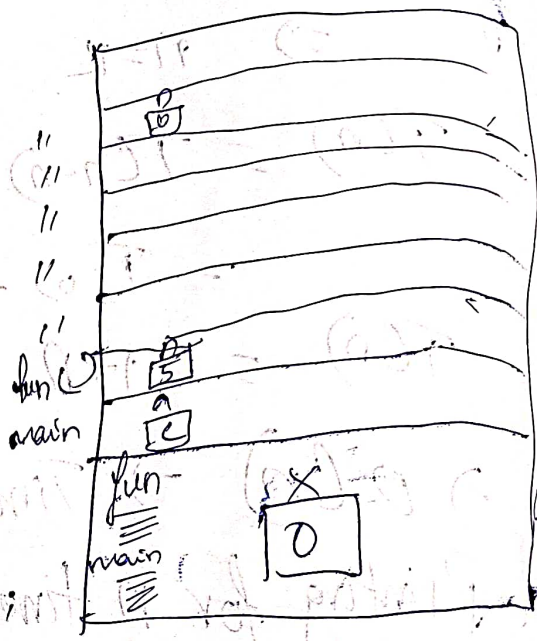
→ They won't copy every time the function gets called. Just a single copy

```
int fun(int n){
    static int x = 0;
    if(n>0){
        x++;
        return fun(n-1)+x;
    }
}
```



P.T.O

```
int fun (int n) {
    if (n > 0) {
        x++;
        return fun (n-1) + x;
    }
}
int main () {
    int a = 5;
    printf ("%d\n", fun (a));
    return 0
}
```



→ should not be included as tracing

x
0 1 2 3 5

fun (5) = 25

fun(4) + 5 = 25

fun(3) + 5 = 20

fun(2) + 5 = 15

fun(1) + 5 = 10

fun(0) + 5 ∴ x = 5

0

Recursion:
↳ 5E

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){
    fun(n-1); -> Recursive call
  }
}
```

→ All operations performed at calling time. No operations at returning time.

When will 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.
↳ O(n)

Space Complexity ?

Tail Recursion:
O(n)

while loop:
O(1) -> constant

Loop is efficient than Tail Recursion.
↳ Not the case for all recursions.

Head Recursion:

All operations done in returning time.


```
Void fun(int n){
    if (condition){
        fun(n-1);
        ==========
        ==========
    }
}
```

↳ return time

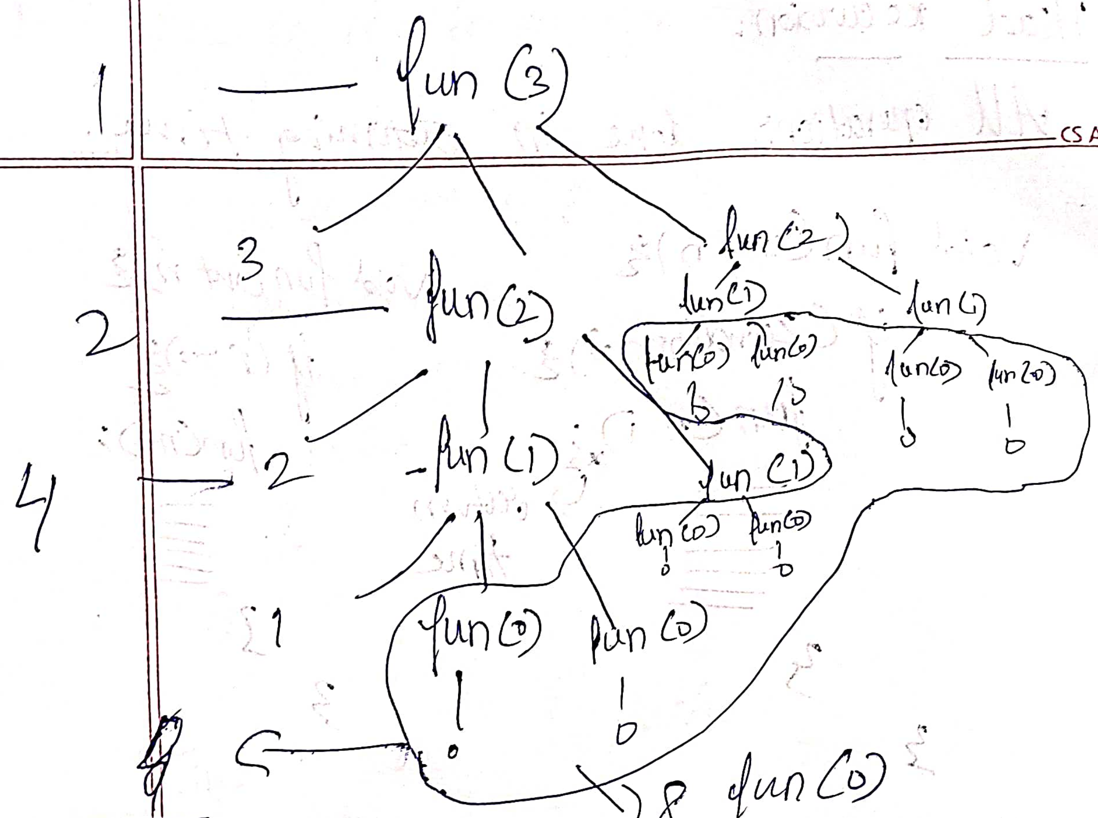
```
Void fun(int n){
    if (n>0){
        fun(n-1);
        ==========
    }
}
```

Tree recursion:

Two or more recursive calls inside a recursive function.

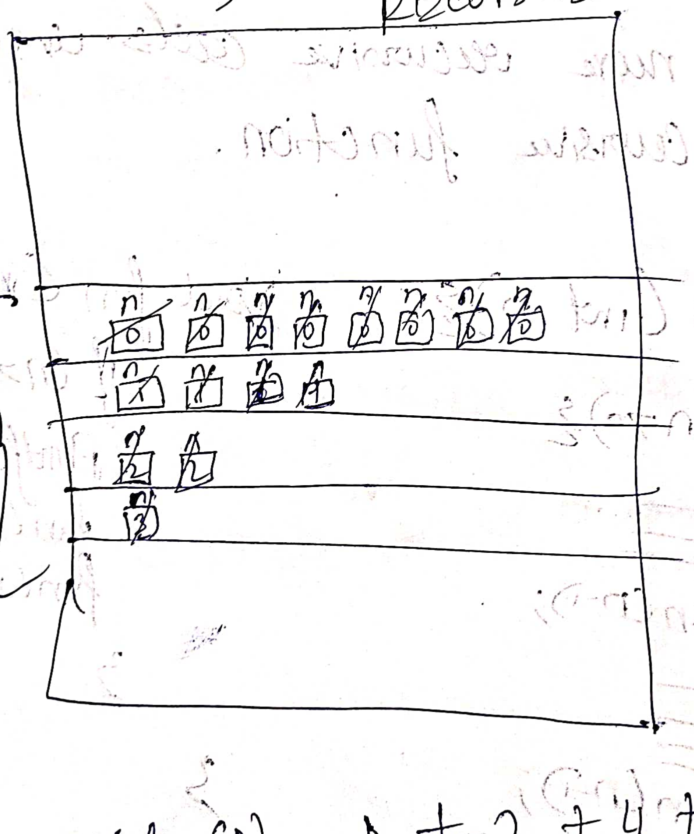
```
Void fun(int n){
    if (n>0){
        ==========
        fun(n-1);
        ==========
        fun(n-1);
        ==========
    }
}
```

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



15 → $O(2^{n+1}-1)$

→ For this particular recursive program will change for other

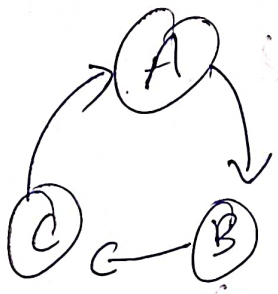


Space Complexity
→ 4 levels
$O(n)$
For n space = n+1

$T(fun(3)) = 1 + 2 + 4 + 8$
$= 2^0 + 2^1 + 2^2 + 2^3$
$= 15$
$\Rightarrow 2^{n+1} - 1$

Indirect Recursion:

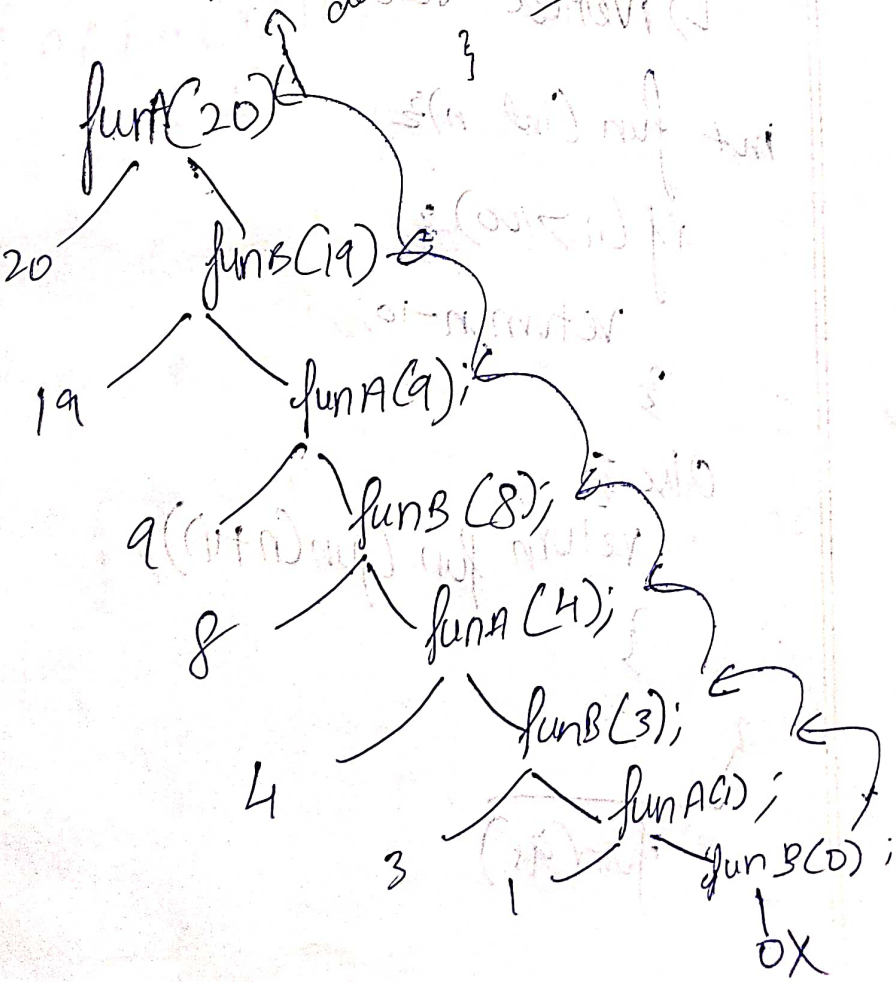
More than one function calling each other in a circular fashion.



Cyclic

```
void A(int n){
    if (){
        B(n-1);
    }
}
void B(int n){
    if (){
        A(n-1);
    }
}
```

```
void funA(int n){
    if (n>0){
        printf("%d\n", n);
        funB(n-1);
    }
}
void funB(int n){
    if (n>1){
        Printf("%d\n", n);
        funA(n/2);
    }
}
```



Nested Recursion:

Recursive function will call parameter as a recursive function call.

```
void fun(int n){
    if(n>0){
        fun(fun(n-1));
    }
}
```

≡

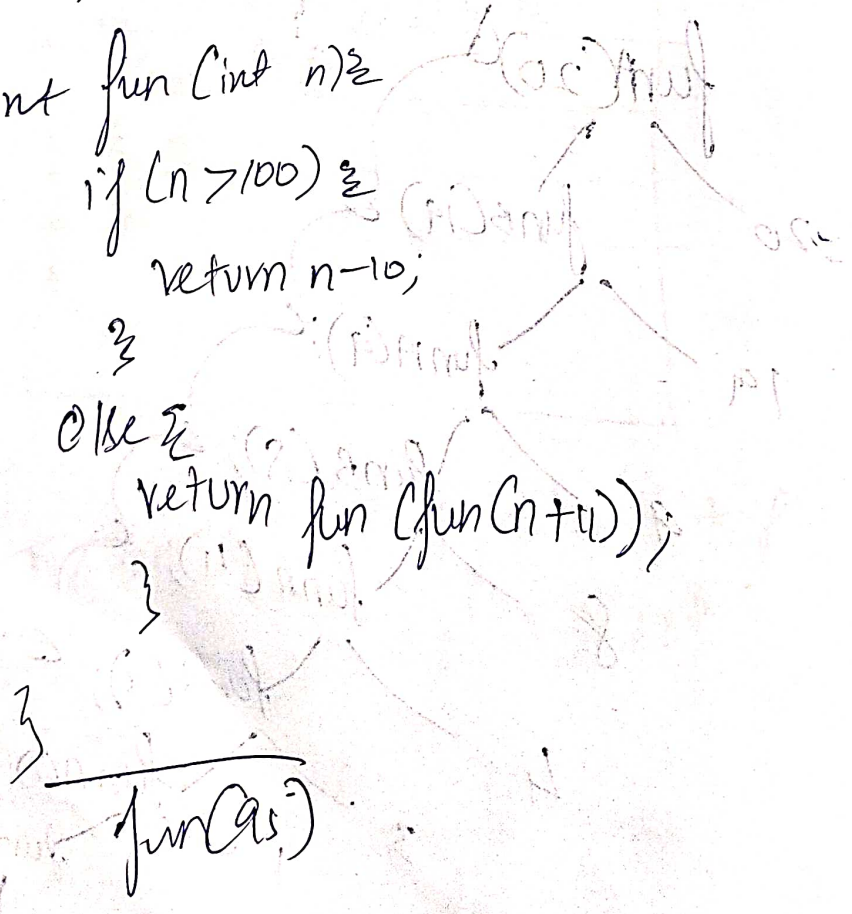
- Unless this recursive call's result is obtained this recursive call cannot be made.

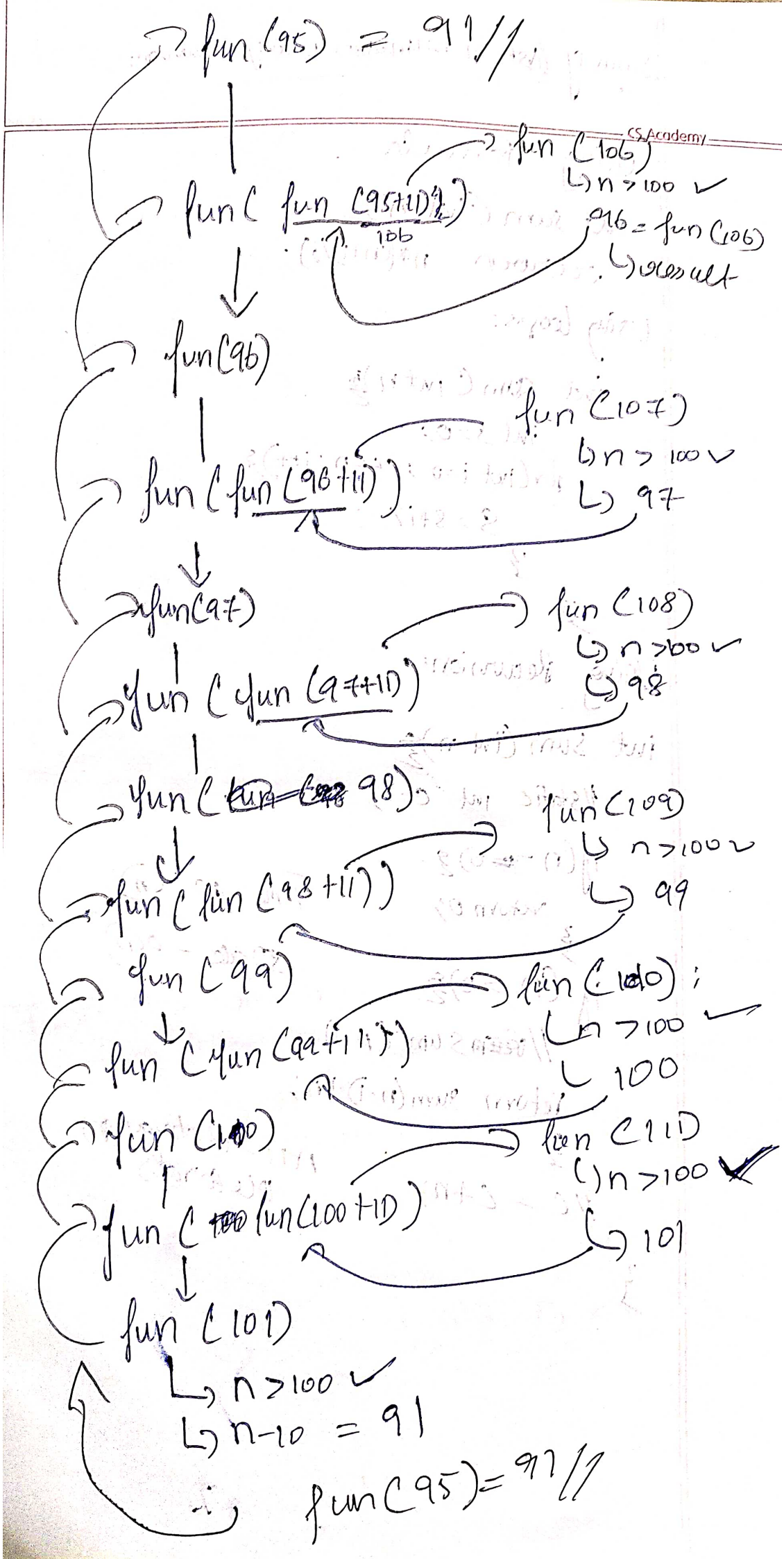
- A recursive call is taking it's parameter as recursive call.
- It's recursion inside recursion.

↳ Nested Recursion

```
int fun(int n){
    if(n>100){
        return n-10;
    }
    else{
        return fun(fun(n+11));
    }
}
```

fun(95)




fun(95) = 91 //
CS Academy
fun(106)
n > 100 ✓
fun(fun(95+11))
106
96 = fun(106)
result
fun(96)
fun(107)
n > 100 ✓
fun(fun(96+11))
97
fun(97)
fun(108)
n > 100 ✓
fun(fun(97+11))
98
fun(98)
fun(109)
n > 100 ✓
fun(fun(98+11))
99
fun(99)
fun(110);
n > 100 ✓
fun(fun(99+11))
100
fun(100)
fun(111)
n > 100 ✓
fun(fun(100+11))
101
fun(101)
n > 100 ✓
n - 10 = 91
∴ fun(95) = 91 //

Sum of first n numbers using Recursion:

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){
    //return sum(n-1)
    return sum(n-1)+n;
  }
  //c=c+n;
}
```

Time - O(n)

Space - O(n)

n+1 - activation records

Factorial of a Number:

CSA

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

```
int fact(int 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:

$exp = (m)^n$

$= \underbrace{m \times m \times \dots m}_{n \text{ times}}$

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

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

pow(2,9)
|
pow(2,8) x 2
|
pow(2,7) *2
|
pow(2,6) x2
|
pow(2,5) x2
|
pow(2,4) x2
|
pow(2,3) x2
|
pow(2,2) x2 = 4
pow(2,1) x2 = 2
|
pow(2,0) = 1

Faster way:

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

$= (2*2)^4$

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

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

pow(2,9) = 2^9
|
2 * pow(2^2, 4) = 2^{8} = 2*2*2*2
|
pow(2^4, 2) = 2^8
|
pow(2^8, 1) = 2^8
|
2^8 * (2^{16}, 0) = 2^8 * 1
|
1

multiplication denser than pow one

Taylor's series:

1 + 2 + 3 + 4 + 5 + 6 +

CS Academy

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

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

```
double ~~int~~ e(int x, int n){
    static double p=1;
    static double f=1;
    double r;
    if(n==0){
        return 1;
    }
    r = e(x,n-1);
    p = p * x;
    f = f * n;
    return r+(p/f);
}
int main(){
    printf("%lf\n", e(2,2));
    return 0;
}
```

0 0 2 4 6 8

$2[1+2+3+4]$

$2\left[\frac{n(n+1)}{2}\right]$

$= n(n+1)$

$O(n^2)$ → multiplication

$\Rightarrow O(n^2)$.

Time is Quadratic

Using Power functions:

```
double taylor(int x, int n){
    static int y=0;
    static double c=1.0;
    if(y<n){
        y = y+1;
        c = (double)p(x,y)/(double)fune(x,y))+c;
        tay(x,n);}
```

```
    return c;
}
int main() {
    printf("%f\n", log(2.3));
    return 0;
}
```

Taylor's series with Horner's rule:

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

$$1 + \frac{x}{1} \cdot \frac{+x^2}{1\times 2} + \frac{+x^3}{1\times 2\times 3} + \frac{x^4}{1\times 2\times 3\times 4}$$

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

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

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

↳ Time → O(n)
↳ linear

Fibbonaci series:

~~for loop~~

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

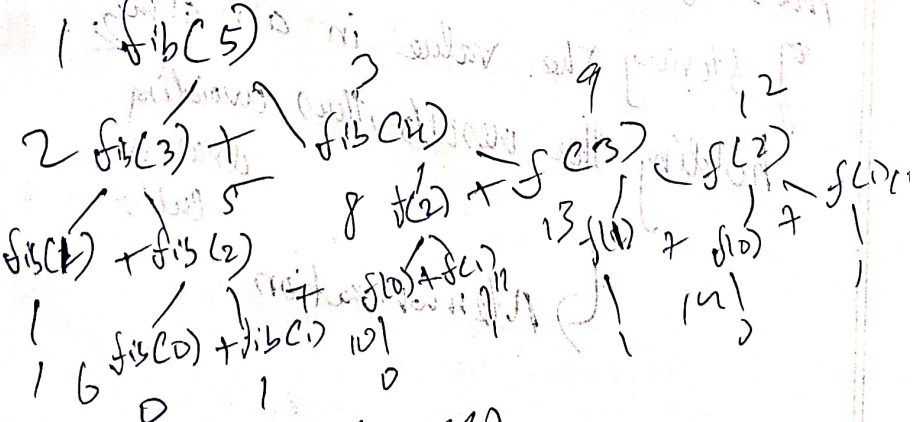
For Loop:

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

4n → O(n)

Recursion:

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



1) Total - 15 (calls)

Assum. 2 fib(n-2) + fib(n-1) = 2 fib(n-1)

calling 2 times for a reduced value of n → T - O(2^n)

~~Reduced value of n for each time~~

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

↳ Excessive recursion.

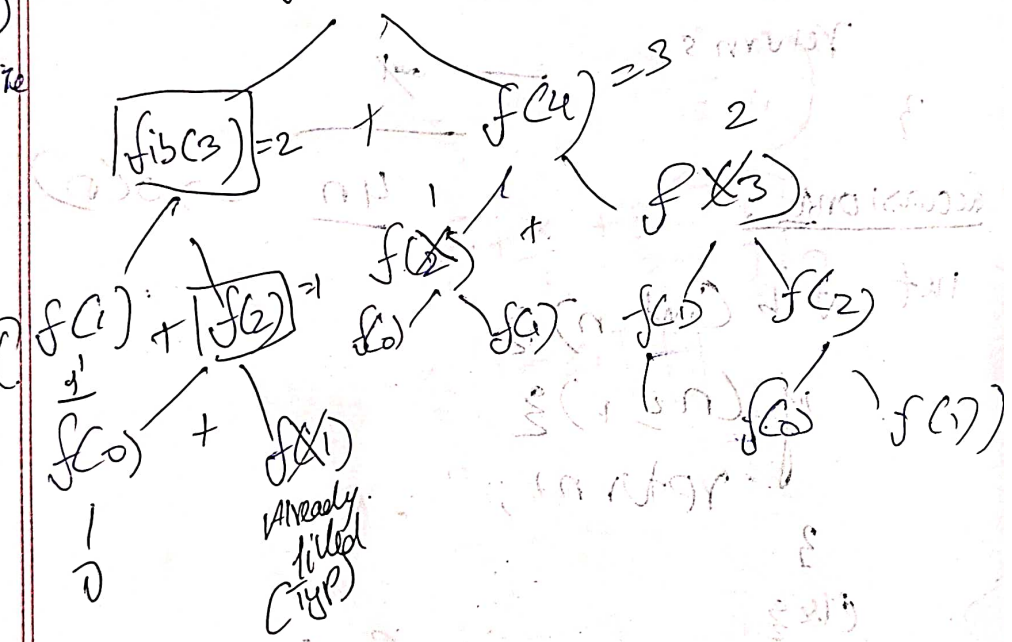
Memorization

F

-6	-11	-1	-2	-3	-5	-1
0	1	2	3	4	5	6

Global or Static Array

Initialize with -1



This reduction in time is because of storing the value in an array, holding the results, thus avoiding excessive calls.

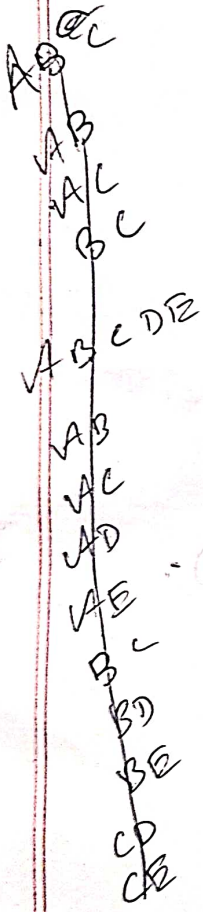
↳ Memorization

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

CS Academy

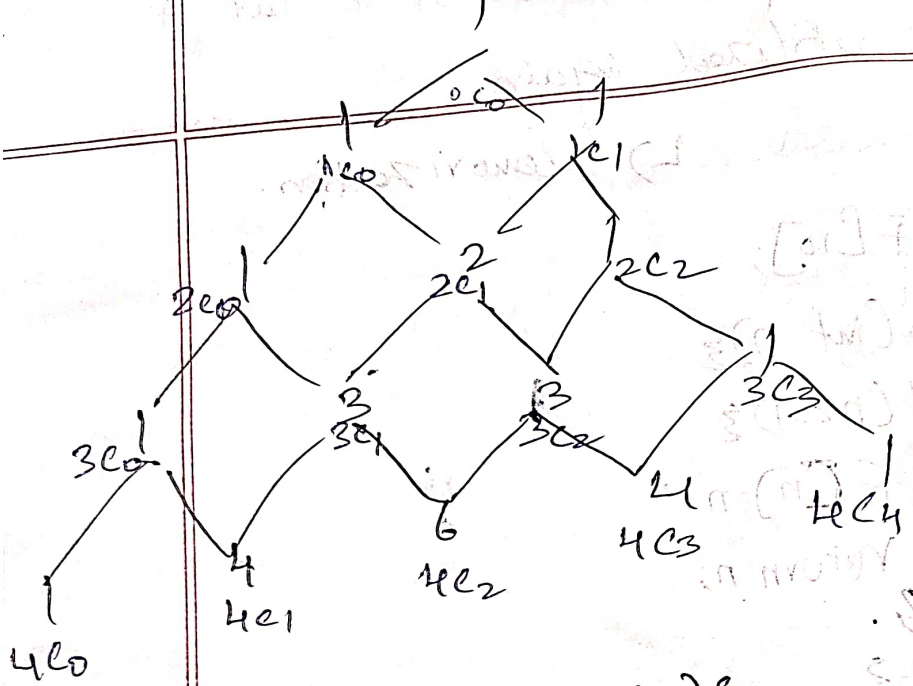
2) Memorization.

```
int F[10];
int fib(int n){
    if(n<=1){
        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)
```



nCr

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

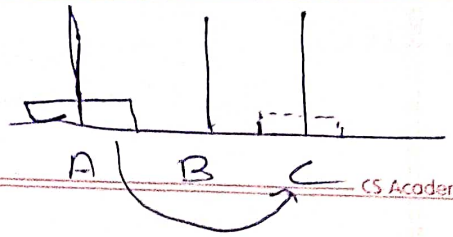



```
int C (int n, int r){
    if (r==0 || n==r){
        return 1;
    }
    else {
        return C(n-1, r-1) + C(n-1, r);
    }
```

Tower of Hanoi

TOW(1, A, B, c)

1) Move Disk from A to C using B

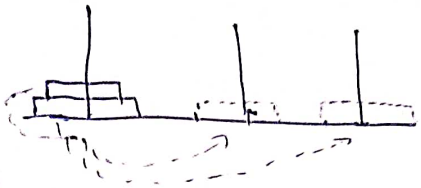


CS Academy

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 yuh(int n){

int x = 1;

int toH(int n, int a, int b, int c){

if(n==0){

return 0;

}

else{

toH(n-1, a, c, b);

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

toH(n-1, b, a, c);

}

}

Quiz- Q4:

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

fun(5)

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

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

1

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

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

$= 1 + 5 + 4 + 5$

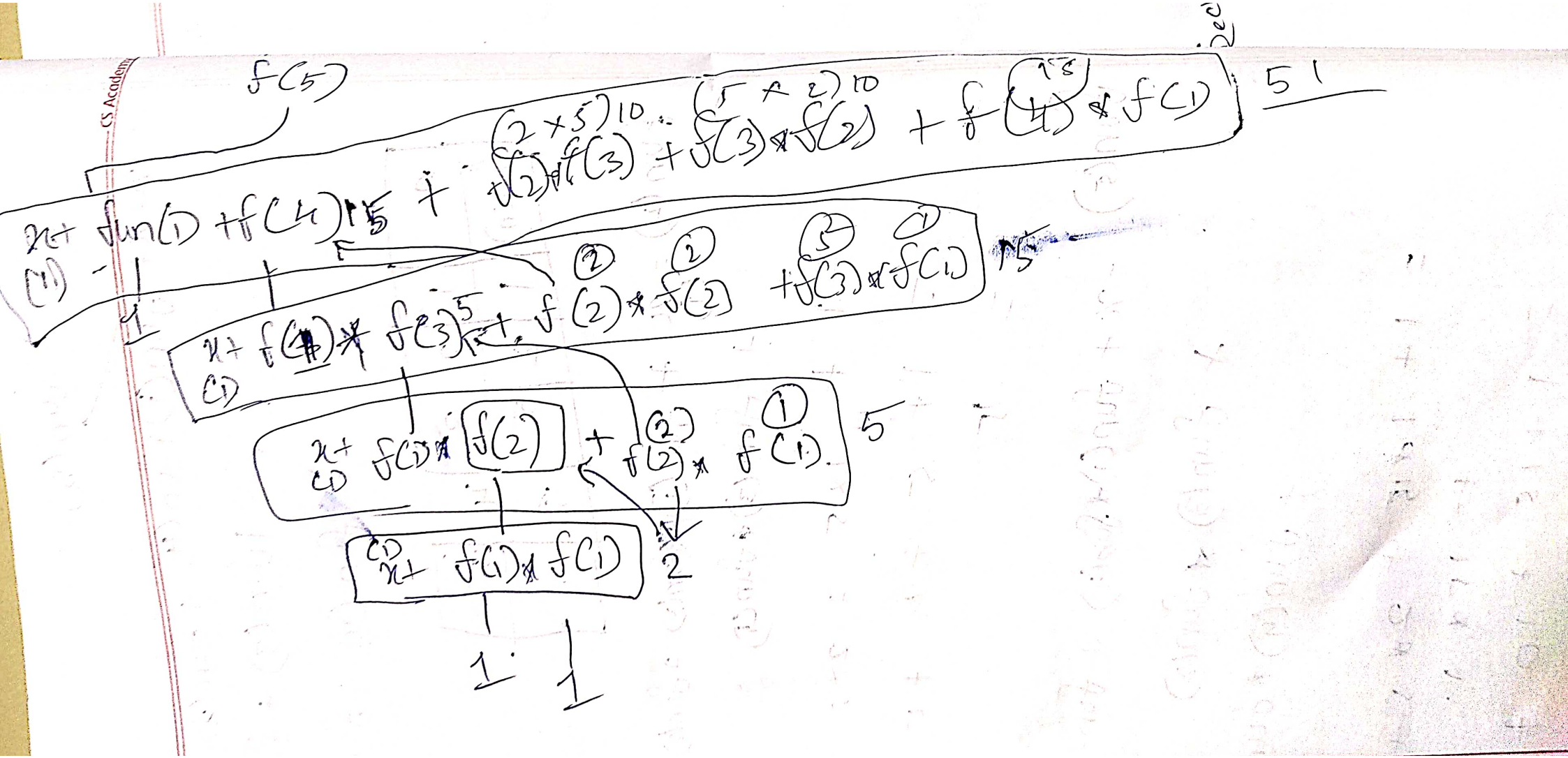
$= 15$

$fun(5) = x + fun(1) * fun(4) + fun(2) * fun(3)$
$+ fun(3) * fun(2)$
$+ fun(4) * fun(1)$

$= 1 + 1 * 15 + 2 * 5 + 5 * 2$
$+ 15 * 1$

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

$= \underline{51}$ //

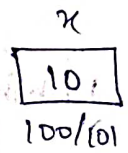

f(5)
return f(1) + f(4) * 5
(1)
return f(1) * f(3) * 5 + f(2) * f(2) + f(3) * f(1)
(2) (2) (1) (1)
return f(1) * f(2) + f(2) * f(1)
(1) (2) (1)
return f(1) * f(1)
(1)
1
1
5
2
5
5
1

Arrays:

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

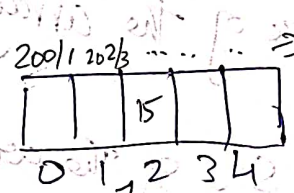
Array

Scalar ⇒ int x = 10;



Array ⇒ collection of elements of same type [Data type]
↳ Store list of values

int A[5];



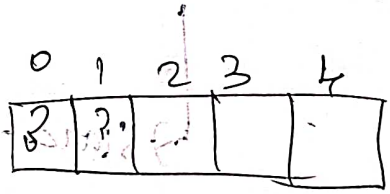
⇒ 5*2 bytes
↳ Total 10 bytes memory

↓
Vector
(1-Dimension)

A[2] = 15;

↳ memory contiguous - Side-by-side

Initializing:

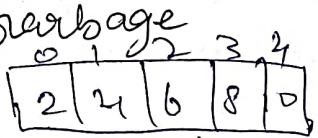


(Declaration) int A[5];

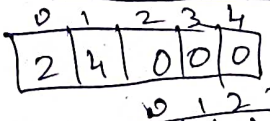
Garbage

(Declaration + Initialization)

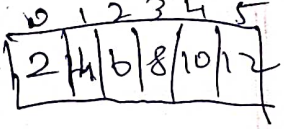
int A[5] = {2,4,6,8,10}



int A[5] = {2,4}



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



int A[5] = {2, 5, 4, 9, 8};

Printf("%d", A[i]); (i = 0, 1, ... n)

Printf("%d", Z[A]);

Printf("%d", *(A+2));

Printf("%u \n", &A[i]);

↑ Printing addresses

Static vs Dynamic Arrays

Static Array - Size of the array is static

Dynamic Array - Size of the array is Dynamic

void main() {

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

↳ Static - Because size of the array decided at compile time, but the memory can be allocated during runtime.

↓

only size of the array decided at runtime.

C language

In C++, we can create an array of any size at own time. -> created in Stack only.

CS Academy

```
int n;
cin >> n;
int A[n];
```

Creating Array in Heap:

1) Size & type decided at own time.

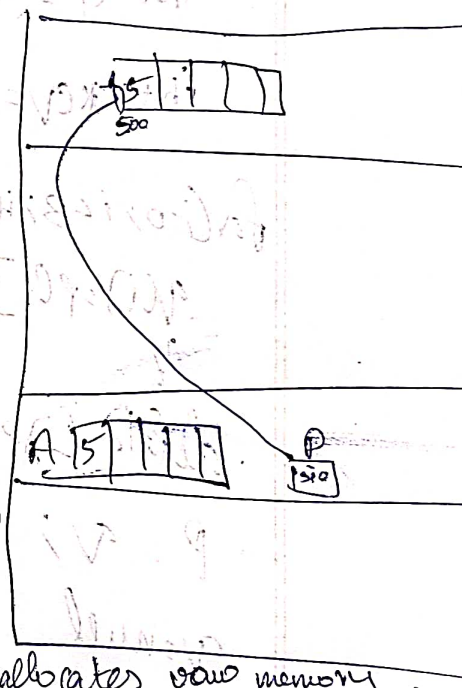
2) We need pointer

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

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

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

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 grown 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
```