Date __/__/____

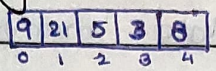
Saathi

Cpp Concepts

1. Arrays: Collection of similar data elements

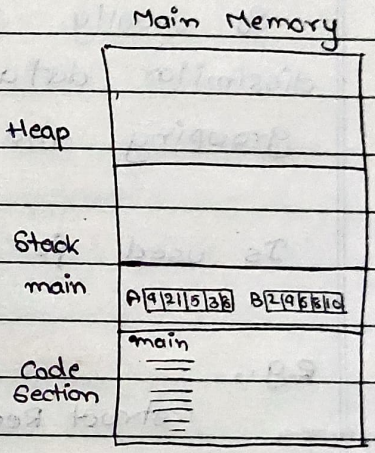
Declaring an array

```
int A[5];
A[0] = 9 ;
A[1] = 21 ;
```



Program

```
int main(){
    int A[5];
    int B = {2,9,6,8,10};
    int i ;
    for (i=0 ; i<5 ; i++){
        printf("%d", B[i])
    }
```



Note: • When you are initialising an array you can skip its size.

e.g. `int arr[] = {1, 2, 3};`

• If you have initialise an array of some size & initialise few elements, then the rest of the elements will by default become zero.

for each loop for accessing the elements

```
int main(){
    int A[6] = {2,4,6,8,10};
    for (int x : A){
        cout << x << endl; }
    return 0; }
```

Output : 2 4 6 8 10 0 0

Saathi

Date __/__/____

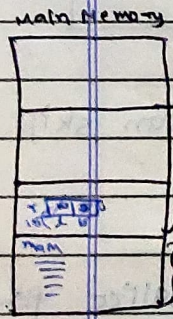
Note : • You cannot initialize variable size array, you need to take it from loop or initialize sparetely.

2. **Structures** : A collection of data members that are related data members under one name; & those data members may be of similar type, maybe of dissimilar type.

So usually it is defined as a collection of dissimilar data items under one name; i.e., grouping the data items

Is used for defining user-defined data types

e.g.,




```
struct Rectangle {
    int length;      — 2
    int breadth;     — 2
};                   4 byte

int main() {
    struct Rectangle r;
    r.length = 15;
    r.breadth = 10;
    printf("Area of Rec. %d", r.length * r.breadth);
}
```

defination ~~declaration~~ doesn't occupy space

declaration occupy space

declaration + initialization

Page No. 2

Date __/__/____

Saathi

Some More Examples of Structure

1) Complex No. $a + ib$ (a: real no., b: imaginary) $i = \sqrt{-1}$

```
struct Complex {
    int real ;      - 2
    int img ;       - 2
};                  4 bytes of Memory
```

2) Student

```
struct Student {
    int roll ;              - 2
    char name[25] ;         - 25
    char dept[10] ;         - 1×10 = 10
    char address[50] ;      - 1×50 = 50
};                          77 bytes of Memory
struct Student s ;   <- declaring variable
s.roll = 10 ;
s.name = Nandini ;
```

3) Playing Card

face - A(1), 2(2), 3 ... J(11), Q(12), K(13)

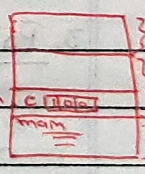
shape - ♣(0), ♠(1), ♢(2), ♡(3)

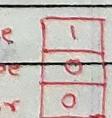
color - 0 (black), 1 (Red)

```
struct Card {
    int face ;  - 2
    int shape ; - 2
    int color ; - 2 } ;
                  6 bytes
int main() {
    struct card c ;
    c.face = 1 ;
    c.shape = 0 ;
    c.color = 0 ;
```

otherwise directly ->

```
struct card c = {1, 0, 0} ;
```





→ pt.0 →

Note: • Defination doesn't consume memory until or unless you declare any variable of that structure.

Saathi

globally { • can also declare var. outside main function directly
• can also declare var. along with deffination of struct.

struct Rec { } r1;

```
int main() {
    struct card deck[52];
    struct card deck[52] =
    {{1,0,0},{2,0,0}...{1,1,0},{2,1,0}...};
    printf("%d", deck[0].face);
    :
}
```

will form array of structure so, here it will form 52 structures.

52 × 6 (one structure) = 312 bytes of Memory (52 structures)

Output : Ace

→ Practice → Rectangle.

for character also it will allocate 4 bytes but use only one bite.

not one (for making its accessibility easy)

• for different datatypes, it changes like for char, it takes instead of taking 1 byte, it will take the nearest bigger size i.e., int. (4 byte). & this adjustment in memory is scareless padding.

3. Pointers : is a address variable that is meant for studying storing address of data, not data itself ;

normal variables are data variables, but pointer variables are address variables. & Pointers are used for indirectly accessing the data.

So, the question is, Why do you need to access data indirectly ?

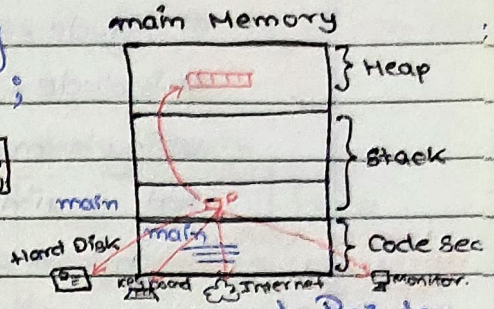
Why actually we need Pointers ?

Note : When you declare var., it will be inside Stack

Saathi

Date __/__/____

So, the program can directly access Stack & Code Section; program will not automatically access heap. So heap memory is external to the program. So, for accessing Heap Memory we need Pointer. Pointers are used for accessing resources which are outside the program.



- Pointers are useful for
  - Accessing Heap
  - Accessing Resources
  - Parameters Passing

→ How to declare, initialize & use Pointers

e.g., int a = 10; // data var.

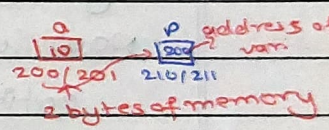
declaration → int *p; // address var.

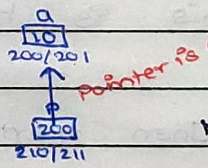
assignment/initialization → p = &a;

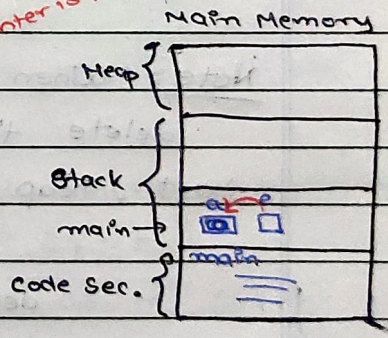
pointer is also var. so it will occupy memory

printf("%d", a);

dereferencing → printf("%d", *p);







So, for dereferencing we use *
for declaring we use *
for initializing directly we can write p

→ How to access Heap Memory using Pointers (Dynamic Memory Allocation)

```
int main() {
```

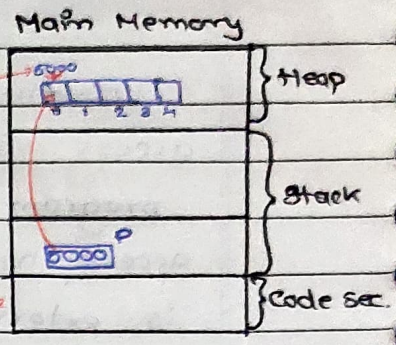
In C++ → new int *p

In C → when you write malloc() → function take the size, then only you get the memory in heap.

P.T.O →

Saathi

Date ___/___/_____




```
#include <stdio.h>
#include <iostream>
using namespace std;
int main(){
```

*: malloc function returns void; we need to typecast it.

```
int *p;
```

how much(5 integer copies) to get compiler decide the size.

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

5 * 2 = 10 bytes of memory

```
Cpp lang → p = new int[5];
```

this line will allocate heap memory

Remember: Whenever we say malloc then only it will allocate in heap.

So the p will pointing ~~at~~ in the address 5000 So, it is just like something is there in heap, and a pointer is holding it. It is just holding one edge of that array i.e., 5000. Now using that pointer we can access the entire array.

So, this is how heap memory is created.

Note: • When memory is not required you should delete the memory; if the program is very short, heap memory is automatically deleted.

```
delete []p; → deallocate memory in cpp
free(p);    → deallocate memory in c
```

• Whatever type of pointer is, it is independent; size of a pointer is independent of its datatype.

Note :
For making a variable reference, use &

Saathi

Date __/__/____

not a part of C Language

4. Reference : is a nickname / alias given to a variable

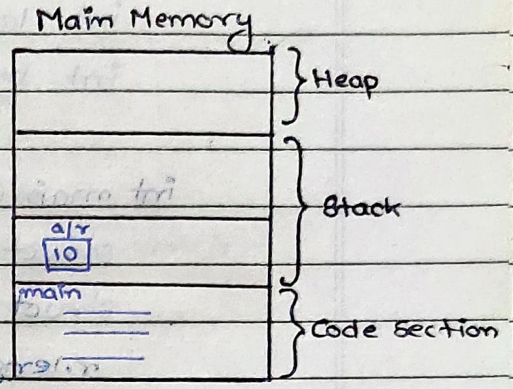
```
int main(){
    int a = 10;
    int &r = a;
    cout << a ;  — 10
    r++;
    cout << r;  — 11
    cout << a ;  — 11
}
```

actual name → a/r ← alias

10

200/201 ↑ 2 bytes of memory

declaration & is assigned with a, so, this is initialized with a.



→ Why you need another name to the same var.?

- This is useful in parameter passing.
- This is very useful feature of C++ ; for writing small functions, we use references instead of using pointers.

Reference doesn't consume memory (It uses same memory of a.)

Note :

If &a → reference variable

If a → variable

If *a → pointer which can store address

→ What does it mean when you have a reference that is referring to a ? [int &r = a;]

This means, this is itself is old as r. So you call this value with name a as well as r. a/r 10

So, now we have 2 names for the same value inside memory. So it means, if that var. a is occupying 2 bytes of memory, let's say 200/201 is address & value 10., then value of r is also same as a.

Reference must be initialized (kiska reference h)

&r = a

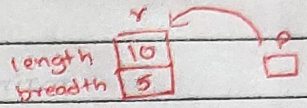
reference of a (nickname of a)

Date __/__/____

Saathi

5. Pointer to a Structure

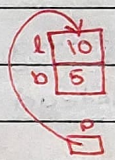
```
struct Rectangle {
    int length;      - 2
    int breadth; };  - 2
                       4 bytes
```




```
int main() {
    struct Rectangle r = {10,5};
    struct Rectangle *p = &r; //pointer
    r.length = 15;
    (*p).breadth = 20; // pointer pointing to a structure
```

or ⇒ `p->breadth = 20;`

Create the object dynamically in the heap using pointer




```
int main() {
    struct Rectangle *p;
    p = (struct Rectangle *) malloc(sizeof(struct Rectangle));
    p->length = 10;
    p->breadth = 5;
}
```

6. Functions : is a piece of code that performs a specific task. It is a group of instructions that perform a specific task.

Note: Grouping data is structure.
Grouping instruction is functions.

Functions are cld as modules or procedures.

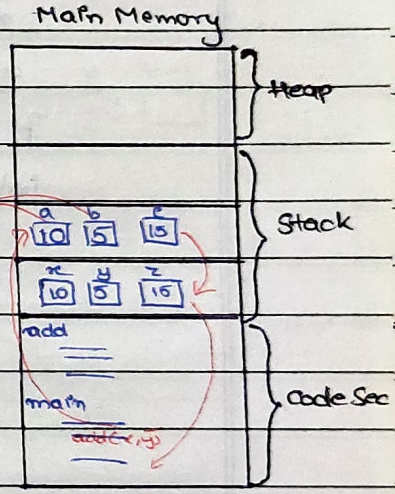
Saathi

You can break the program into smaller tasks & you can focus on smaller tasks and finish them & make them perfect.

→ Increased productivity.
→ reuseability
This Modular & Procedural Programming.

e.g.,

```
prototype/signature
declaration of func" → int add(int a, int b) {
                          int c;          ← formal parameters
defination or             c = a+b;
elaboration of func"      return(c); }
Starting point of a program → int main() {
                          int x, y, z;
                          x = 10;
                          y = 5;
                          z = add(x, y);   ← Actual parameters
func" call →              printf("Sum is %d", z);
                          }
```



Output: 15

Result of add(x,y) is copied in z

- Formal parameters are temporary, when function is called these are used & there are destroyed when function ends.
- Values of actual parameters are copied in formal parameters.
- For reducing the workload from main function, we use functions.

Date ___/___/_____

7. Parameter Passing Methods

1. Pass by value
2. Pass by address } In C
3. Pass by reference } In Cpp

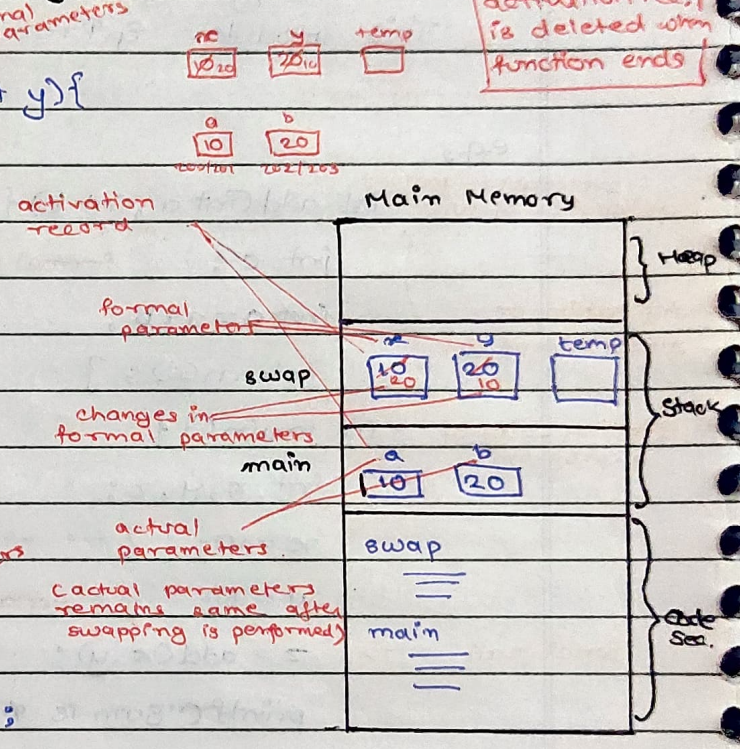
1. → Pass by value / Call by - Value

doesn't return anything

formal parameters

```
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);
    cout << a << b;
}
```

actual parameters



a → When you should use Pass by Value.

- When you don't have to modify actual parameters.
- When function is returning some results.

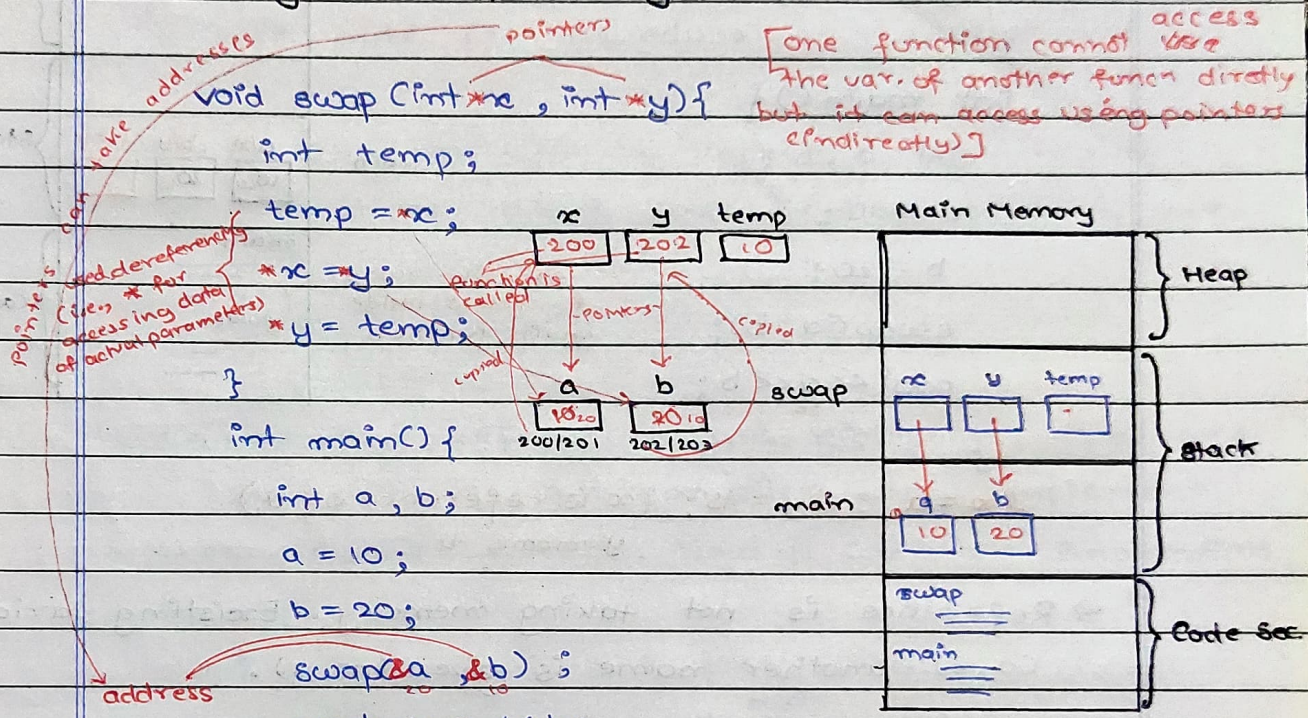
[So, swap function should not be returning / work done using Pass by Value. Adding 2 no.s is suitable (as it is returning value) for Pass by Value].

Date __/__/____

Saathi

Remember :- In Pass by Value, actual parameters will not be modified if any changes are done to formal parameters.

2 → Pass by address / Call by Address



⇒ *x = &a ; *y = &b (pointers pointing to an address)

Remember :- Call by Address is suitable mechanism for modifying the actual parameters.

★ Call by Address is more useful.

3 → Pass by Reference / Call by Reference

[Reference are supported only in Cpp; not part of C].

So, here we have to learn 2 things:

- How to write Call by Reference ?
- How it works ?

Saathi

Date __/__/____

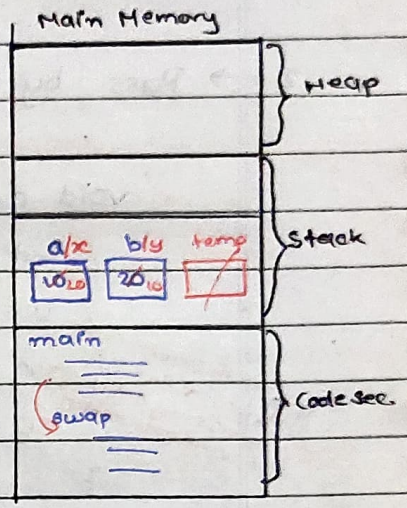
reference

```
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);
    cout<<a<<b;
}
```

alias

temp | a/x | b/y

10 | 10/20 (200/201) | 20/10 (202/203)



Its like swap function is inside main function.

20 10

&x = a ; &y = b ; (x is reference of a)

y means b

⇒ Reference is not taking memory. Existing variable (a, b), another name is given (x, y).

Remember:- In Call by Reference, when these formal parameters (x & y) are manipulated, the actual parameters are modified.

Quick Imp. Question:- As "One function cannot access the variables of another function directly, it can access indirectly"; but how it is possible that it is accessing directly in Call by Reference?

⇒ If you see inside the Main Memory, swap fun° has become the part of Main funcⁿ. So when the main funcⁿ running variables are a & b & when the swap fun° starts these are dd as x & y & also temp

variable is created inside a same code activation record of Main function. & once the swap func^n ends, it has become the a/x=20, b/y=10, So it is more like the monolithic program. So the Machine code is Monolithic, Though the source code is procedural or modular.

Note :- Don't use Call by Reference for heavy functions which are having loops & are having complex logic.
Use it carefully.

Conceptually, references are nicknames. They are not pointers. But compiler may implement them as a pointer. How a compiler implements we can't say. But

But there are two possibilities:

1. A compiler can convert swap fn function as an inline function & copy that code in main fn.
2. It can make the formal parameters as pointers.

Note: You can have different type of arguments.

```
int main(int a, int *b, int &c){..}
```

[We don't use the parameter passing mechanism name with a func^n name, we give it to the parameter].

Note : In A[] -> if we don't mention the size, initialized then size will be dependent on the no. of elements

Saathi

Date ___/___/____

8. Array as Parameter

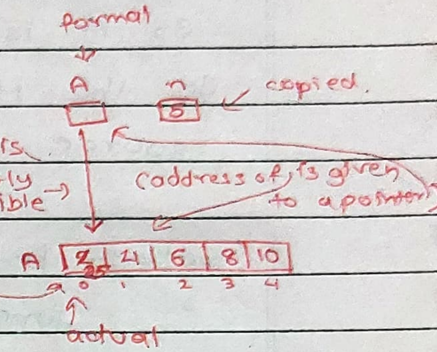
```
void fun(int A[], int n){
    int i;
    for(i=0; i<n; i++){
        printf("%d", A[i]);}
}

int main(){
    int A[5]={2,4,6,8,10};
    fun(A,5);
}
```

(parameters pointer to an array -> can also be written as * (* can be the array, [] specifically for array)) (call by value)

printing all the elements of array

Like A[0] = 25

array / size of array



[usually when we pass array as a parameter to a func, we also pass size of that array]

Arrays are passed by address [Cannot be passed by value]
(Giving a [] means pointer to an array)
[If we make changes to an array (formal para) it will make changes to actual parameters]

→ Returning array as a parameter.

```
int [] fun(int n){
    int *p;
    p=(int *)malloc(n*sizeof(int));
    return p;
}
int main(){
    int *A;
    A = fun(5);
    ===
}
```

returning an array / pass by value

can also written * can point on one element of array or element

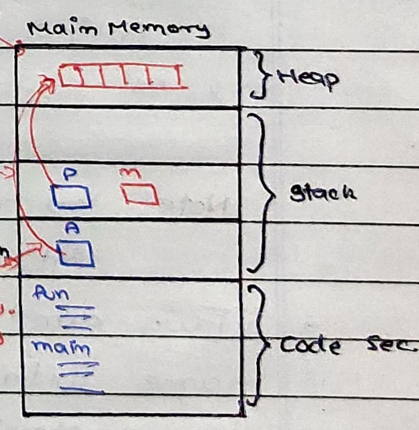
having pointer

allocating an array of size of type int

address of array will be present in p

activation record

A will get address of array. so, A will be pointing to an array.



Once the func^n ends, its activation record is deleted so, p is gone, but A is accessing that array present in heap.

{[] for returning is not supported in every compiler so use *. }

[You should know very well that which func^n is used should use which method for parameter passing]

Date ___ / ___ / _____

Saathi

9. Structure as Parameter

1) Call by Value
2) Call by Address
3) Call by Reference

1) Call by Value

```
struct Rectangle{
    int length;
    int breadth;
};
int area(struct Rectangle r1){
    r1.length++;
    return r1.length*r1.breadth;
}
int main(){
    struct Rectangle r={10,5};
    cout<<area(r);
}
```

//we used .(dot) operator for accessing the member

Note: If we make any changes in formal parameters (like r1.length++), formal parameter will change but the actual parameters will not change i.e., call by value

call by value → formal r1 ← new variable is created & values are copied here

r1: 10 (11), 5

(It means separate object will be created by call by value)

This is feature of structure

r ← actual

10 l
5 b

2) Call by Reference

```
struct Rectangle {
    int length;
    int breadth;
};
int area( struct Rectangle &r1){
    r1.length++;
    return r1.length*r1.breadth;
}
int main(){
    struct Rectangle r={10,5};
    cout << area(r);
}
```

(area will not be separate part of machine code, it will be pasted in a main func^n).

separate var. is not created.

If we do length++ it will modified

alias → r/r1 ← actual parame

10 (11) l
5 b

[So though for calculating area we don't need call by Ref; we need call by Val, becaz it just supposed to read the values].

Saathi

3) Call by Address

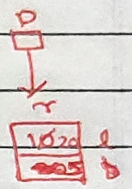
[When you want some func^n to modify actual parameters, then it must be cld by address or by reference].

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

pointer

address

// we used -> for accessing the members



→ Structure can be passed by call by value even if it is having an array. [as array is always passed by address]

```
struct Test {
    int A[5];
    int n;
};
void fun(struct Test t1) {
    t1.A[0] = 10;
    t1.A[1] = 9;
}
int main() {
    struct Test t = {{2, 4, 6, 8, 10}, 5};
    fun(t);
}
```

[when a structure is passed by value, even if it is having array, an array will be created separetely in the member & it will be filled.]

[So, imagine all these values are copied by compiler automatically just done by the compiler. so this is time consuming]

call by value [This is the reason arrays are passed by reference]

t1

A | 2 | 4 | 6 | 8 | 10 |
n | 5 |

values are copied

t

A | 2 | 4 | 6 | 8 | 10 |
n | 5 |

[But if an array is inside structure, they support it]

Date __/__/____

Saathi

// Returning address of a structure

→ Creating a variable of type structure on heap inside a func^n & return its address.

```
struct Rectangle {
    int length;
    int breadth;
};
struct Rectangle *func(){
    struct Rectangle *p;
    p = new Rectangle;
    // p = (struct Rectangle *)malloc(sizeof(struct Rectangle));
    p->length = 15;
    p->breadth = 7;
    return p;
}
int main(){
    struct Rectangle *ptr = func();
    cout<<"length"<<ptr->length<<endl;
    cout<<"breadth"<<ptr->breadth<<endl;
    return 0;
}
```

not taking any parameters

Return a pointer of type rectangle

It will create an object of type rec. on heap

Cpp Syntax =>

new means in Heap

creating

C. Syntax →

// for pointer we use →

Saathi

Date __/__/____

10. Structures & Functions (writing Functions upon a structure)

C-Lang

```
struct Rectangle {
    int length;
    int breadth;
};
void initialize(struct Rectangle *r, int l, int b){
    r->length = l;
    r->breadth = b;
}
int area(struct Rectangle r){
    return r.length * r.breadth;
}
void changeLength(struct Rectangle *r, int l){
    r->length = l;
}
int main(){
    struct Rectangle r;
    initialize(&r, 10, 5);
    int rec = area(r);
    changeLength(&r, 20);
}
```

all three func^ns : initialize, area, changelength is for a structure.

r
length 10 20
breadth 5

need one more var. for area.

(taking rectangle by reference)

(Function was taking Rectangle as parameter)

→ How it leads to Object Oriented & how it is highest?

All these functions are related to that structure. So, in C programming, this is the highest level of programming, where we define a structure & we write all the functions related to that structure. Becaz grouping of data at one place is a structure; grouping of instructions

defining Class → class className { ← User-defined name
Keyword
Access specifier; // can be private, public or protected.
Data Members; // Variables to be used.
Member Functions(){} // Methods to access data members.
}; // Class name ends with a semicolon.

Saathi

Date ___/___/_____

Declaring object → ClassName Object Name;

for performing a task is a function.

11. Class & Constructor

Converting C program to Cpp Program

Cpp-Lang → class Rectangle { [class contains data members as well as functions (Rectangle, area, changeLength)]

private:
int length;
int breadth; } data members.

[We declare these data members as Private; & Functions as Public (because who is accessing these data members, so many func's are there, why you need that data to be public? so hide it) (Everything should be accessible using these functions.)]

public:
void initialize(int l, int b){
r → length = l;
r → breadth = b;
}

X replace it with directly access

Constructor → Rectangle (int l, int b){
directly access → r → length = l;
r → breadth = b;
}

[Constructors are special class members which are called by the compiler everytime an object of that class is instantiated. Constructors have the same name as the class & may be defined inside or outside the class definition.]

3 Types of constructors:
1. Default 2. Parameterized 3. Copy

[Destructors is another special member function i.e. cld by the compiler when the scope of the object ends.]

int area(){
return length * breadth;
}
void changeLength(int l){
length = l;
}};

r
length 10 20
breadth 5

int main(){
Rectangle r(10,5); ← Declaration as well as Initialization
(constructor)
For this we need Func → Rectangle(int, int)
object (instance of a class)
X r.initialize(10,5); directly initialize
r.area();
r.changeLength(20);
} member func's of object

[When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data & functions defined in the class, you need to create objects.]

(now initialize func itself is a part of rectangle)

Saathi

Date __/__/____

12. Practice Monolithic Program

(For monolithic, everything will be inside main func)
(whenever you declare a var., you should initialize them ⇒ its a good practice).

```
int main(){
    int length = 0, breadth = 0;
    printf("Enter Length & Breadth:");
    cin >> length >> breadth;
    int area = length * breadth;
    int peri = 2 * (length + breadth);
    printf("Area: %d, Peri: %d", area, peri);
    return 0;
}
```

13. Practice Modular Program

(Interaction in main func, Processing in funcns)

```
int area(int length, int breadth){
    return length * breadth;
}
int ~~prod~~(){
    int l=10; b=15;
    int a = area(l, b);
    cout << ~~area~~;
    return 0;
}
```

// In CPP writing struct in parameter is not mandatory
int area(struct Rectangle r); & int area(Rectangle r); both are valid.

Saathi

Date __ / __ / ____

14. Practice Structure and Functions

(same Program as 10. Structure & Functions ; P-13)

15. Practice Object-Oriented Program

(same Program as 11. Class & Constructor)

// instead of writing class as keyword you can write struct also in cpp

class R{}; or struct R{}; both are valid.

but class is private & struct in public
- class → cannot access members as by default private → make it public.
- struct → can access members

16. Class & Constructor

```
class Rectangle {
  private:
    int length;
    int breadth;
  public:
    Rectangle(){length = breadth = 1;}
    Rectangle(int l, int b);
    int area();
    int perimeter();
    int getLength(){return length;}
    void setLength(int l){length = l;}
    ~Rectangle();
};
```

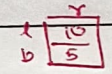
- data members
- default constructor
- constructor → (doesn't have return type) (but it may or may not take parameters)
- parameterized constructor. (header of a function)
- Methods
- constructor → facilitators → (perform some operation on data members of an object)
- accessor / mutator
- destructor →
- Type of functions

[If you have to destroy anything and if you have any dynamic memory or location inside the constructor, then you can release that memory inside a destructor.]

P.T.O. →

scope resolution

```
Rectangle :: Rectangle(int l, int b){
    length = l;
    breadth = b;
}
int Rectangle :: area(){
    return length * breadth;
}
int Rectangle :: Perimeter(){
    2 * (length + breadth);
}
Rectangle :: ~Rectangle(){

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



17. Template Class

template → a simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types.

generic functions are template func^n's, & generic class are template classes

generic (enables the programmer to write a general algorithm which will work with all data types).

If you have functions outside the class, you should have declaration of those functions inside the class.

Saathi

Date __/__/____

[Note/Suggestion: First complete a program then convert it into a template]

Cpp Program

```
class Arithmatic{
private:
    int a;
    int b;
public:
    Arithmatic(int a, int b);
    int add();
    int sub();
};
Arithmatic::Arithmatic(int a, int b){
    this->a=a;
    this->b=b;
}
int Arithmatic::add(){
    int c;
    c=a+b;
    return c;
}
Arithmatic::sub(){
    int c;
    c=a-b;
    return c;
}
int main(){
    Arithmatic ar(10,5);
    cout<<ar.add;
    cout<<ar.sub;
}
```

constructor for initializing a & b

this refers to the members of a class; this is a pointer to a current object

object

Arithmatic ar(10,5), ar2(20,10); (can write)

Template

```
template<class T>   T is parameter of type template
class Arithmatic{
private:
    T a;
    T b;
public:
    Arithmatic(T a, T b);
    T add();
    T sub();
};
template<class T>
Arithmatic<T>::Arithmatic(T a, T b)   class arithmatic template
{ this->a=a;
  this->b=b;
}
template<class T>
T Arithmatic<T>::add(){
    T c;
    c=a+b;
    return c; }
template<class T>
T Arithmatic<T>::sub(){
    T c;
    c=a-b;
    return c; }
int main(){
    Arithmatic<int> ar(10,5);   T replace with int
    cout<<ar.add();
    Arithmatic<float> ar(10.5,5.5);   replaces with float
    cout<<ar.sub();
}
```