DSA with Abdul Bari

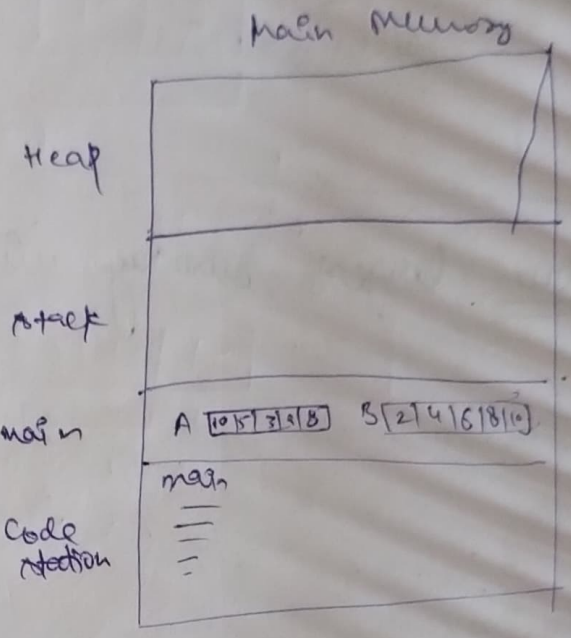
#1: Essential of C++ / C

=> Arrays

```
int A[5];

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

    int i;
    for(i=0; i<5; i++){
        cout<<(B[i]);
    }
```

declaration & initialization

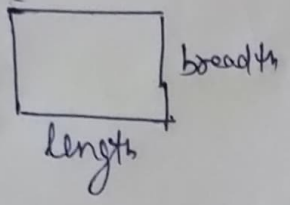
scanning



```
int main(){
int A[6] = {2,4,6,8,10,12}

    for(int x : A){
        cout<<x<<endl;
    }

    return 0;
}
```

⇒ Structure:

It is used to define a Custom data type that can group multiple variables of different types under one name.



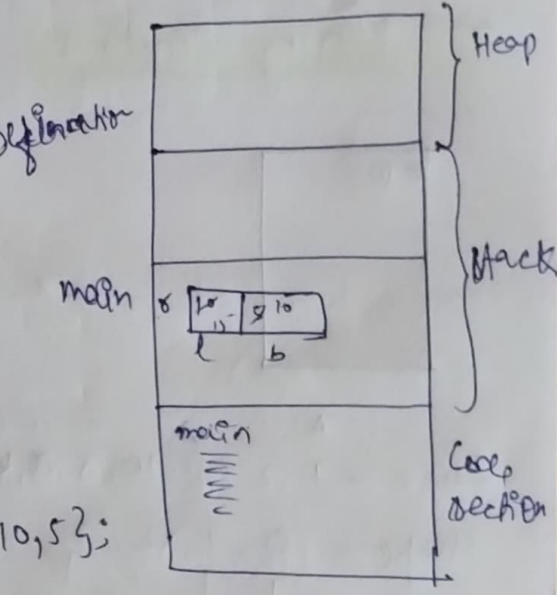

```
struct Rectangle
{
    int length;
    int breadth;
}
```
Defination

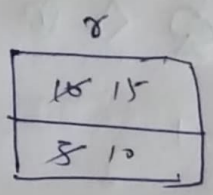
```
int main(){
```

decl. → `struct Rectangle r;`

decl & init → `struct Rectangle r = {10,5};`

accessing member
```
r.length = 15;
r.breadth = 10;
```






```
printf("Area of Rect is :%d", r.length * r.breadth);
cout << "Area" << r.length * r.breadth << endl;
```

Eg: 1. Complex No.

$\sqrt{-1}$ a + ib

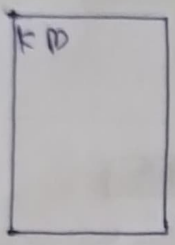
```
struct Complex
{
    int real; -2
    int imag; -2
}
```
4 bytes

2. Student

```
struct Student
{ int roll; -2
  char name[25]; -25
  char dept[10]; -10
  char address[50]; -50
}
```
75 bytes

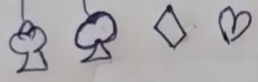
```
Struct student s;
s.roll = 10;
s.name = "John"
:
```

★ Playing Cards :



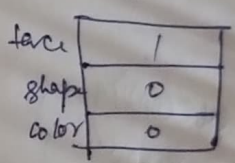
face - 1, 2 ... 10, J, Q, K
A 11 12 13

shape - 0, 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 = 0;
  c.color = 0;
```




```
Struct Card c = {1, 0, 0}
```

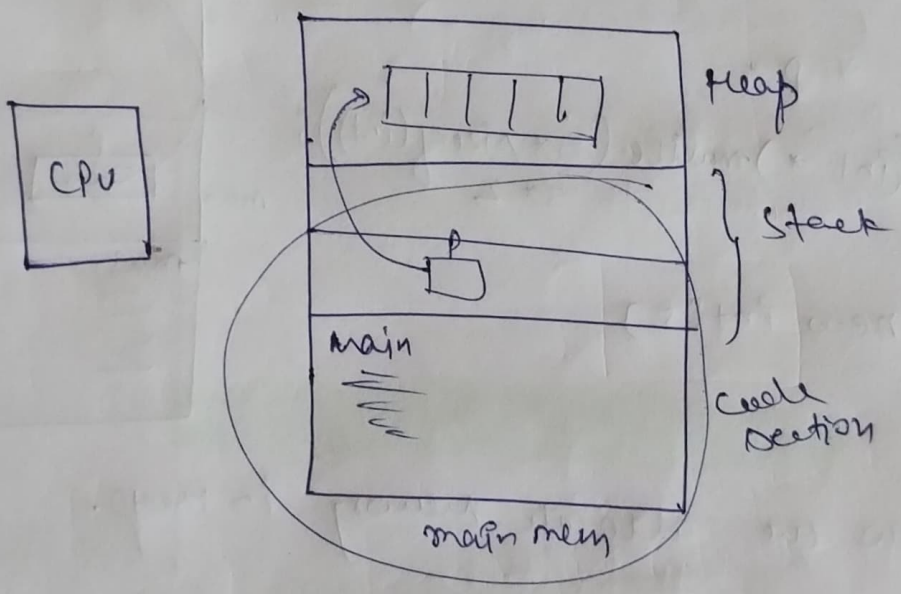
```
int main()
{ Struct Card deck[52] = {{1,0,10}, {2,0,0} ...
                          {1,1,0}, {2,1,0}}
printf("%d", deck[0].face);
printf("%d", deck[0].shape);
printf("%d", deck[0].color);
}
```

Pointers

1. Why pointers
2. Declaration
3. Initialization
4. Dereferencing
5. Dynamic Allocation

Pointers - pointer is a address variable that meant to storing the address of Data.

* Why pointers needed ?



In Normal, programme can only access the stack and code section (not Heap). So accessing Heap (outer resources) we need pointers.

1. Accessing heap
2. " Resources
3. Parameter passing.

Syntax :

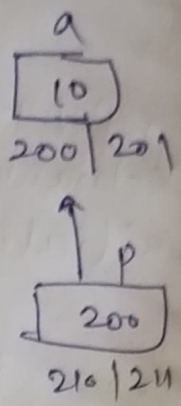
data variable → int a = 10;

Address variable → int *p; ← Declaration

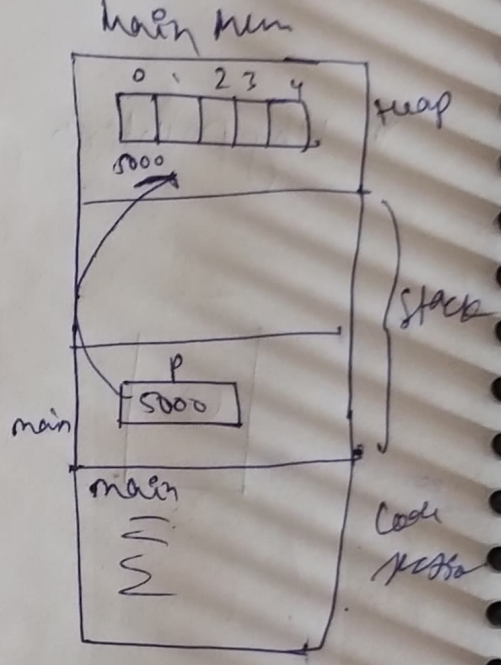
Initialization → p = &a;

printf("%d", a);

dereferencing → printf("%d", *p);




```
#include<stdlib.h>
int main()
{   int *p;
C lang  p=(int *)malloc(5*sizeof(int));
                        5*2
C++ ~   p = new int[5]
}
```



This how we allocate memory in heap.

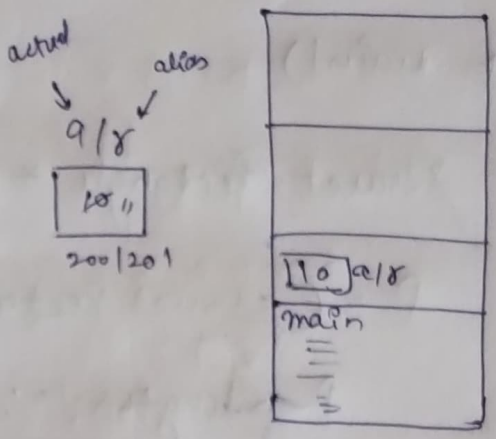
* Pointer with array :-

```
int main()
{   int A[5] = {2,4,6,8,10};
    int *p;
    p = A;   // without '&' sign for Array
    for(int i=0; i<5; i++)
      cout<<p[i]<<endl;
    return 0;
}
```

Reference in C++

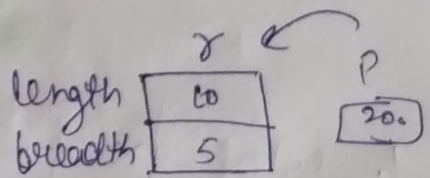
Reference - when a variable is declared as a reference, it becomes an alternative name for an existing variable.

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



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;
   r.length = 15;
   (*p).length = 20;
   p → length = 20;
}
```



* Pointer to a structure using heap memory:

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

In C++, "struct" keyword is not necessary in Ent main.

++ functions

1. what are functions.
2. Parameter Passing
   1. pass by value
   2. pass by Address
   3. Pass by Reference.

function - grouping set of instructions to perform a specific task.

- They are modules or procedures

Monolithic Prog^m

```
int main()
{
    ____
    ____
    ____
    ____
    ____
    ____
    ____
    ____
    ____
    ____
    ____
    ____
}
```

Modular Prog. / Procedural prog

```
fun1(){
    ____
}
fun2(){
    ____
}
fun3(){
    ____
}
int main(){
    fun1();
    fun2();
    fun3();
}
```

Prototype → `int add (int a, int b)` → formal parameter.

Definition →

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

```
int main()
{
    int x, y, z;
    x = 10;
    y = 5;
    z = add(x, y);
    printf("sum is %d", z);
}
```

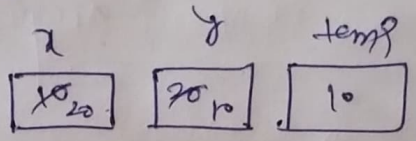
fun. call → `add(x, y)` → Actual Parameter

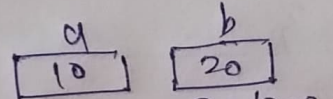
#parameter passing

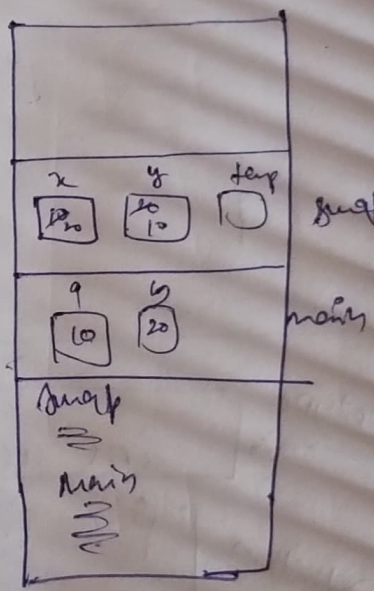
① Pass by value / Call by value
② Pass by address
③ Pass by reference

* Pass by value / Call by value :

```
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 << endl;
}
```



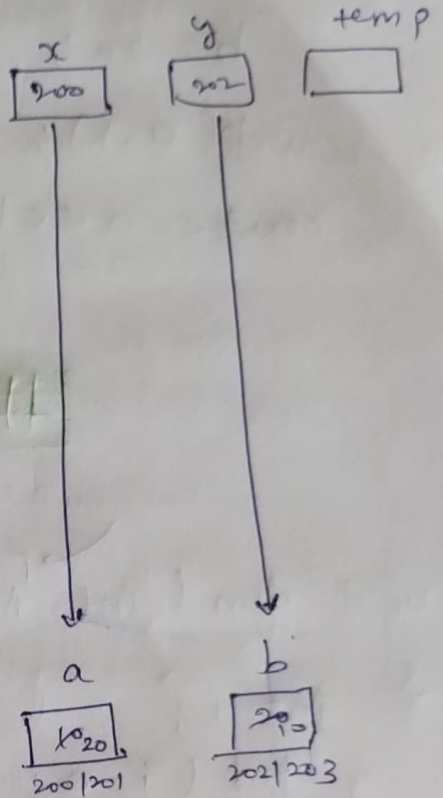




In pass by value changes doesn't effect the Actual value (i.e. a, b).

Call by Address -

```
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 << endl;
}                  20    10
```

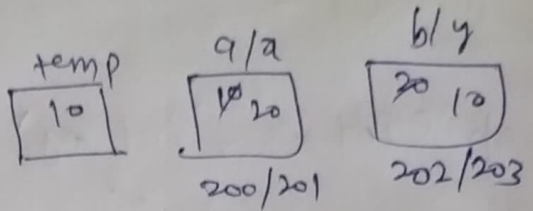


⇒ Call by Address used for changing Actual parameter.

* Call by Reference: (called only in CPP)

```
void swap (int &x, &y)
{  int temp;
     temp = x;
     x = y;
     y = temp;
}
```

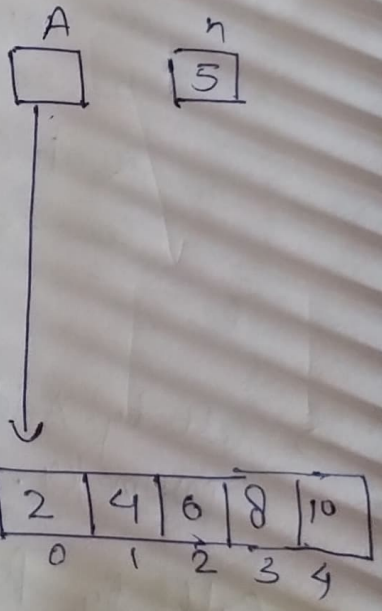
```
int main()
{   int a,b;
    a = 10;
    b = 20;
   swap(a,b);
   cout<<a<<b<<endl;
          20  10
```



#Array as Parameter

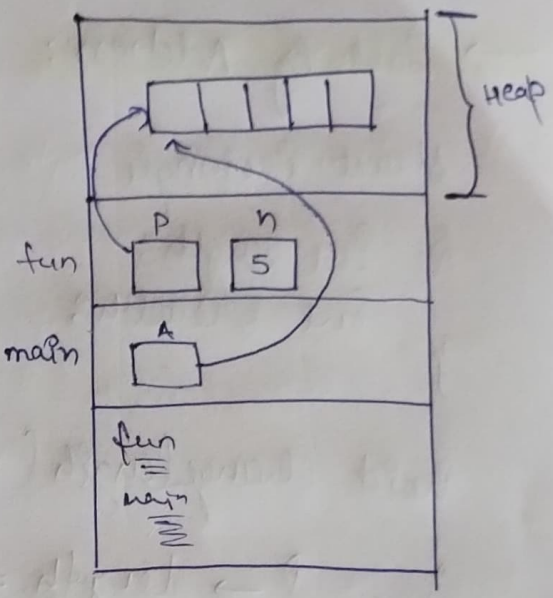
Call by Address → Pointer to Array

```
void fun ( int A[] , int n)
{ int i;
   for (i=0; i<n; i++){
         cout << A[i] << endl;
   }     }

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



* function returning an array:

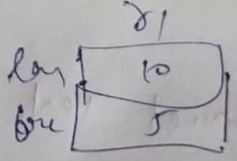
```
int [] fun (int n)
{
    int *p;
    p = new p;
    return (p);
}

int main ()
{
    int *A;
    A = fun(5);
    =
}
```



structure as a parameter

→ Call by value:

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



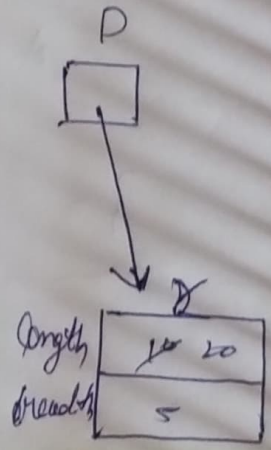
→ Call by Reference

Small change only and Rest of all same.

Change the formal parameter → struct Rectangle &r1

→ Call by Address:

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

void changelength(struct Rectangle *p, int l)
{   p -> length = l;
}

int main()
{   struct Rectangle r = {10,5};
    changelength(&r, 20);
}
```



→ Structure as para. (if it contain an Array)

```
struct Test
{   int A[5];
    int n;
};
```

```
void fun( struct Test t1)
{
    t1.A[0] = 10;
    t1.t1 = 9;
}

int main()
{
    struct Test t = { {2,4,6,8,10}, 5};
    fun(t);
}
```

t1

2 / 10	4 / 9	6	8	10

Structures and functions

```
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)
{
    return r.length * r.breadth;
}
```

r

length	10 / 20
breadth	5

r

10	l
5	b

```
void ChangeLength (struct Rectangle *r, int l)
{
    r -> length = l;
}
```

It is also pointing structure r

```
int main()
{
    struct Rectangle r;
    initialize (&r, 10, 5);
    area(r);
    ChangeLength (&r, 20);
}
```

Note: This is the HL-Programming in C-programming. later on it is converted into OOPs programming.

Converting C prog^m to CPP class

* Class and Constructor:

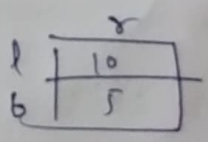
```
class Rectangle
{
  private:
      int length;
      int breadth;

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

```
int area()
{ return length * breadth ;
}

void changelength( int l)
{   length l;
 }

int main()
{
    Rectangle R(10,5);
        r. initialize(10,5);
        r. area();
        r. changeLength(20);
  }
```

Class and Constructor

```
#include <iostream>
using namespace std;
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();
};
```

```
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()
{
}

int main()
{
    Rectangle r(10,5);

    cout<< r.area();
    cout<< r.perimeter();
    r.setLength(20);
    cout<< r.getLength();
}
```



Template Class

```
class Arithametic
{
    private:
        int a;
        int b;
    public:
        Arithametic(int a, int b);
        int add();
        int sub();
};

Arithametic :: Arithametic (int a, int b)
{
    this -> a = a;
    this -> b = b;
}

int Arithametic :: add()
{
    int c;
    c = a+b;
    return c;
}

int Arithmatic :: sub()
{
    int c;
    c = a-b;
    return c;
}
```

The above code is only for "int" datatype, here generic class or template class comes in picture.

```
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;
}

template<class T>
int Arithmetic <T> :: sub()
{  T c;
     c = a-b;
     return c;
}
```

```
int main()
{
    Arithmetic <int> ar(10,5);
    cout<< ar.add();
    Arithmetic <float> ar(1.5, 1.2);
    cout << ar.add();
}
```