Notebook
Pg. No.
Date / /

Data Structure using C

Notebook
Pg. No.
Date / /

Introduction

Topics

- Array
- Matrices
- Linked list

→ (Physical Data Structures)
It define how the data is arranged in the memory

- Stack
- Queues
- Trees
- Graph
- Hashing
- Recursion
- Sortings.

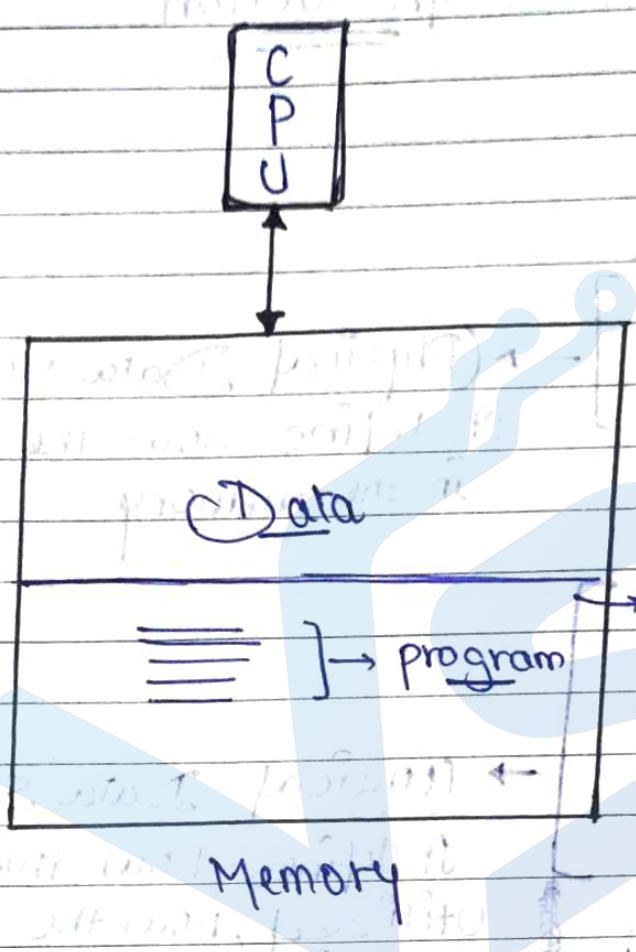
→ (Logical Data Structure)
It defines how the data can be utilized, how the arranged is and how it should be utilized.

What are Data Structures?

→ The way of organize the data in the ~~mem~~ main memory during execution time of program is called data structures.

OR

A data structure is a way to organize and store the data in a computer memory so that it can be managed and processed by the computer effectively.



C → C language does not have builtin data structures.

C++ (STL)

Java (collection)

C#

Python

Javascript (DOM)

→ These language have builtin data structure.

STL, collection class, containes

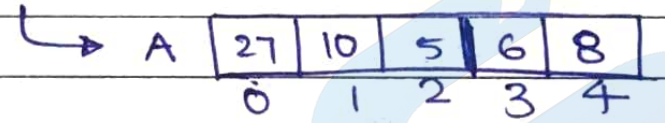
[Essential C and C++ Concepts]

Notebook
Pg. No.
Date / /

- **Arrays**

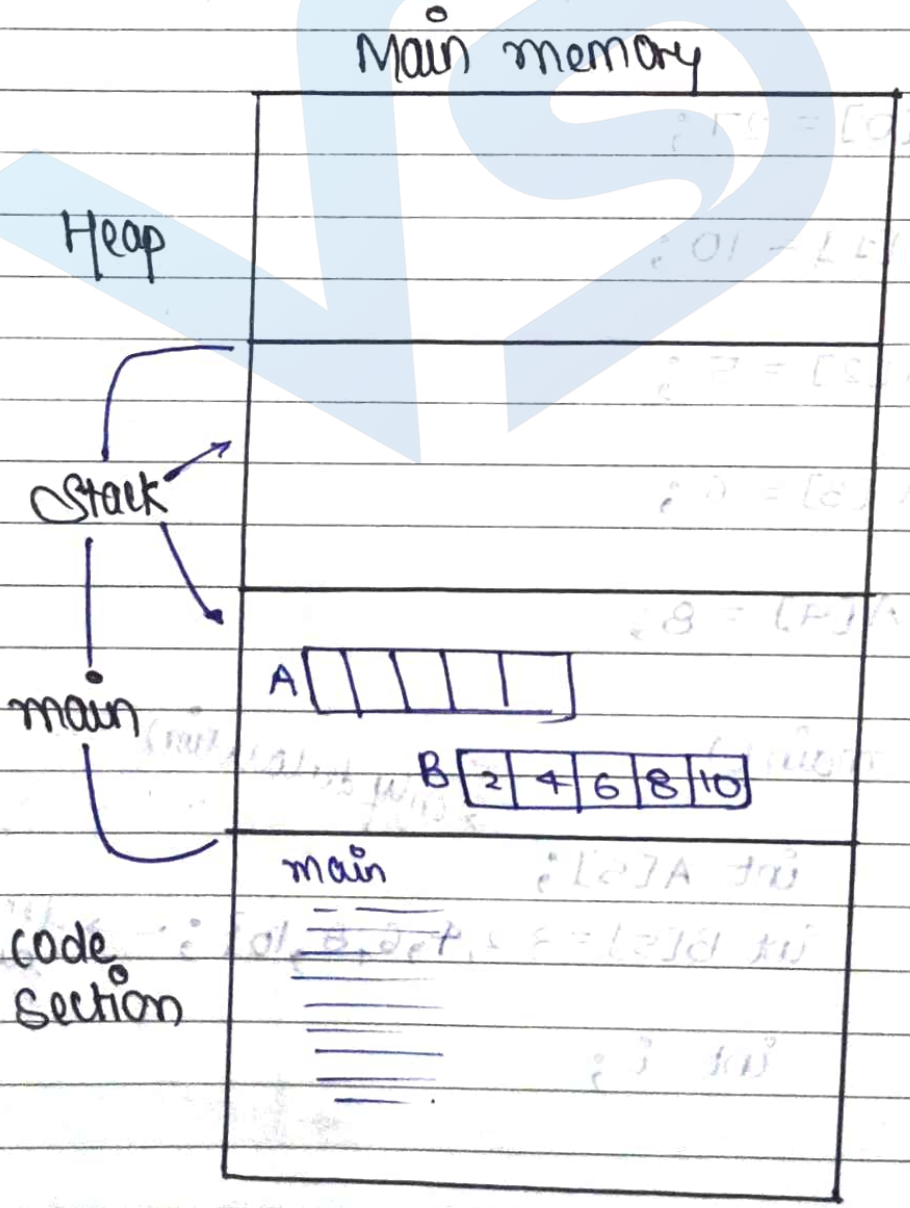
It is defined as the collection of similar data element.

```
int A[5];
```

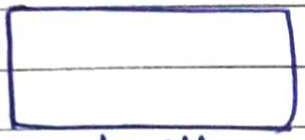



```
A[0] = 27;

A[1] = 10;

A[2] = 5;

A[3] = 6;

A[4] = 8;

int main()
{
    int A[5];        → only declaration
    int B[5] = {2,4,6,8,10};   → declaration and initialization

    int i;
```

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

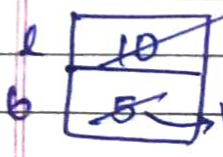


Structure




```
Struct Rectangle
{
    int length;  ——— 2
    int breadth; ——— 2
};                   4 bytes

int main()
{
    Struct Rectangle r;
```

declaration of structure



declare and initialization of structure

```
Struct Rectangle r = {10,5};
```

dot operator for Accessing number of structure

```
r.length = 15;
r.breadth = 10;
printf("Area of Rectangle is %d",
r.length * r.breadth);
}
```

Structure is a method of packing data of different types. A structure is a ~~convenient~~ convenient method of handling a group of related data

items of different data types.

Example

1. Complex No:-

a + ib

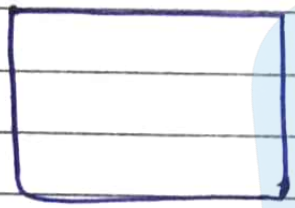
```
Struct complex
{
    int real;        ---- 2
    int img;         ---- 2
};                       4 bytes
```

2. Student

```
Struct student
{
    int roll_no;          ---- 2
    char name[25];        ---- 25
    char dept[10];        ---- 10
    char address[50];     ---- 50
    int age;              ---- 2
};                            89 bytes

int main
{
    Struct student s;
```

```
    s.roll = 10;
    s.name = "John";

    Return
};
```

3. Playing cards



face → A, 1, 2 - - - 10, J, Q, K

Shape → 0 (♣) 1 (♦) 2 (♥) 3 (♠)

color → 0 (Black) 1 (Red)

```
Struct card
{
    int face;
    int shape;
    int color;
};
```

Notebook
Pg. No.
Date / /

C

face	1
shape	0
color	0

```
int main()
{
    Struct card c;

    c.face = 1;
    c.shape = 0;
    c.color = 0;
    // Struct card c = {1,0,0};
};
```

or

```
int main()

{
    Struct card deck[52];  → Array of Structures

Struct card deck[52]={ {1,0,0}, {2,0,0}... }

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

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

int i;
 for(i=0; i<52; i++)
  {
      scanf("%d", &deck[0].face);

  }
}
```

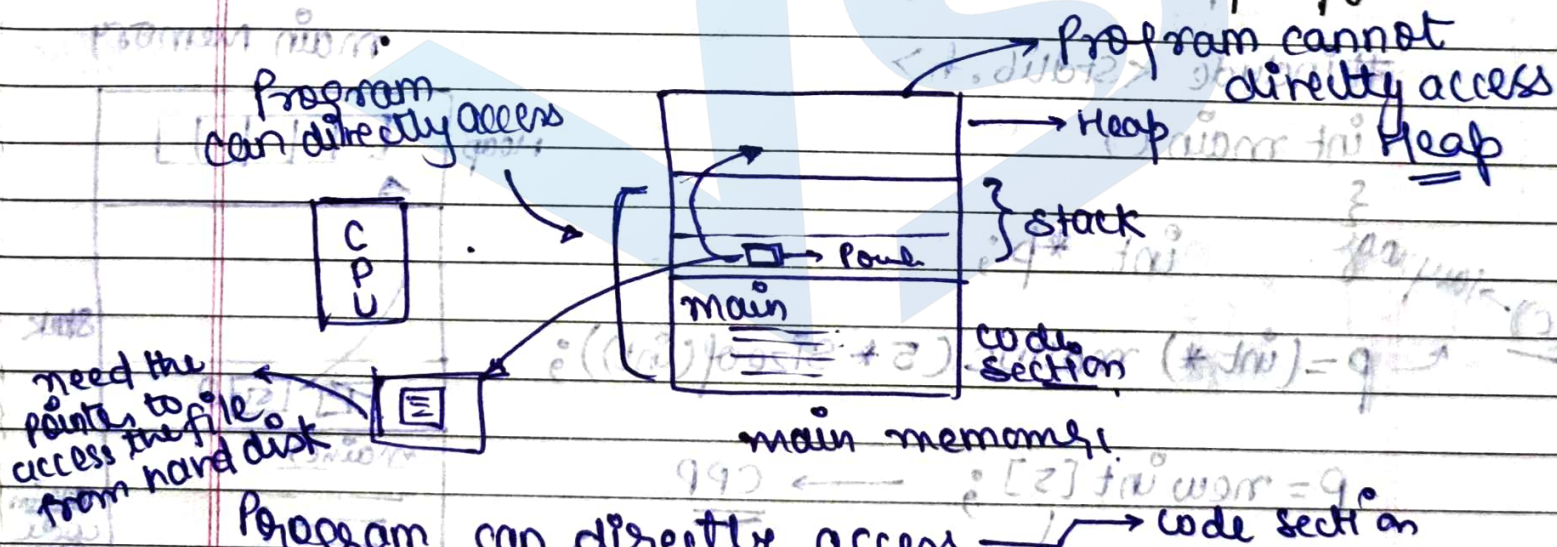
Notebook
Pg. No.
Date / /

Pointers → Address Variable

A pointer is a variable that contains the memory location of another variable.

Pointer is used to indirectly access the data.

Why we need to access the data indirectly?



Program can directly access → code section
→ stack

but not access → Heap

One of main reason to use pointer is to access the heap.

- → Accessing Heap
- → Accessing Resources
- → ~~Accessing~~ ~~Heap~~ Parameter Passing

Example :-

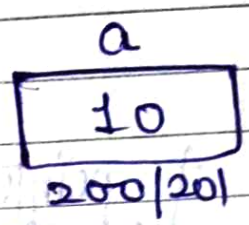
data variable ⟶ int a = 10;

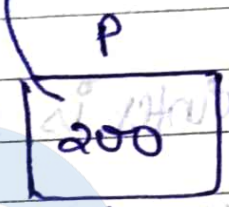
(Variable address) ⟶ int *p;

p = &a;

printf("%d", a); ⟶ 10

printf("%d", *p); ⟶ 10



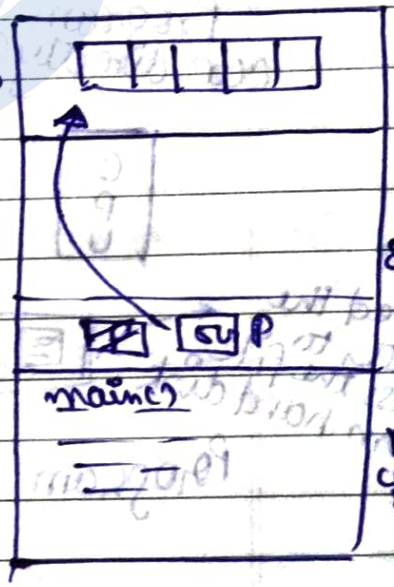



```
#include <stdlib.h>
int main()
{
    int *p;
    p = (int *) malloc(5 * sizeof(int));
    p = new int[5];
}
```

(C) language

CPP

(C++ language)



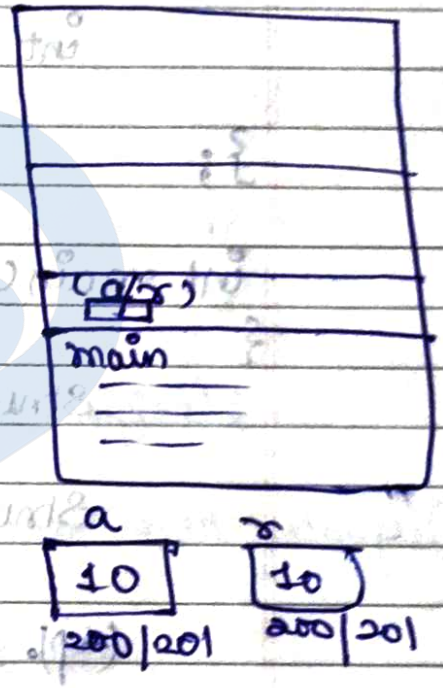
Malloc → Memory in Heap

Reference in C++

Reference is a nickname given to a variable or alias given to a variable.

```
int main()
{
    a=10;
    int &r = a;

    cout <<a;   ⟶ 10
    r++;

    cout << r;  ⟶ 11

    cout << a;  ⟶ 11

}
```

Reference



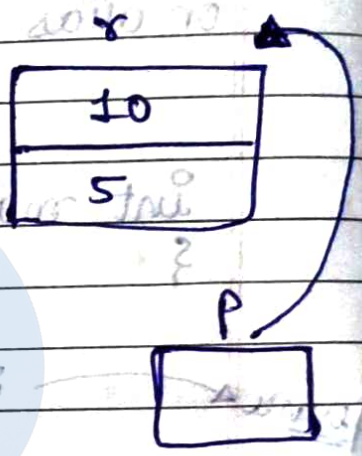
Reference is used in parametering passing

Reference is actually the another name of variable.

Notebook
Pg. No.
Date / /

Pointers to a Structure

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

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



its takes only 2 byte

for normal variable, use dot operator.

for pointer variable, use -> (arrow).

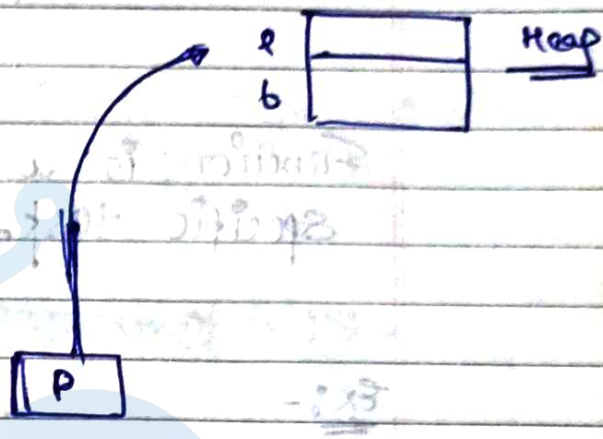
*p.length = 20;

It is wrong because dot operator has high precendence.

Notebook
Pg. No. ____
Date ____ / / ____

Dynamic

```
Struct rectangle
{
    int length;
    int breadth;

};
int main()
{
    Struct rectangle *p;

    (Struct rectangle* malloc(sizeof(Struct rectangle));

    p = (Struct rectangle *) malloc(sizeof(Struct rectangle));

    P -> length = 10

    P -> breadth = 5;

}
```



Type casting

Grouping data → Structures

Grouping instruction → function

Notebook

Date / /

Functions

Function is a piece of code which perform a specific task.

Ex:-

```
int main()
{
    ——
    ——
    ——
    ——
    ——
    ——
    ——
    ——
    ——
}
```

Monolithic programming

(Modular Programming / Procedural Progra...)

```
func1()
{
    ——
    ——
}

func2()
{
    ——
    ——
}

func3()
{
    ——
    ——
}

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

Parameter passing:-

- Pass by value
- Pass by Address
- Pass by Reference

Notebook
Pg. No.
Date / /

Example :-

```
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("sum is %d", z);
}
```

Proto-type/signature

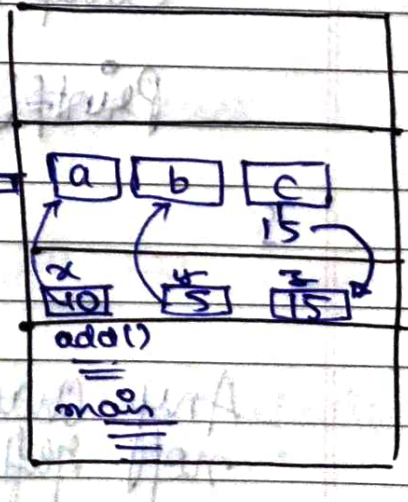
Formal Parameters

declaration of variables

actual Parameters

Main cannot access the variable of add function.

Add function cannot access the variable of main function.



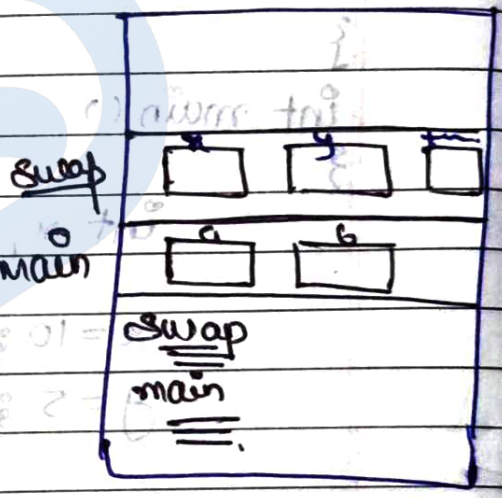
Parameter Passing

→ call by value / Pass by value

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

Actual parameters



a	b
10	20
200/201	202/203

→ 10, 20

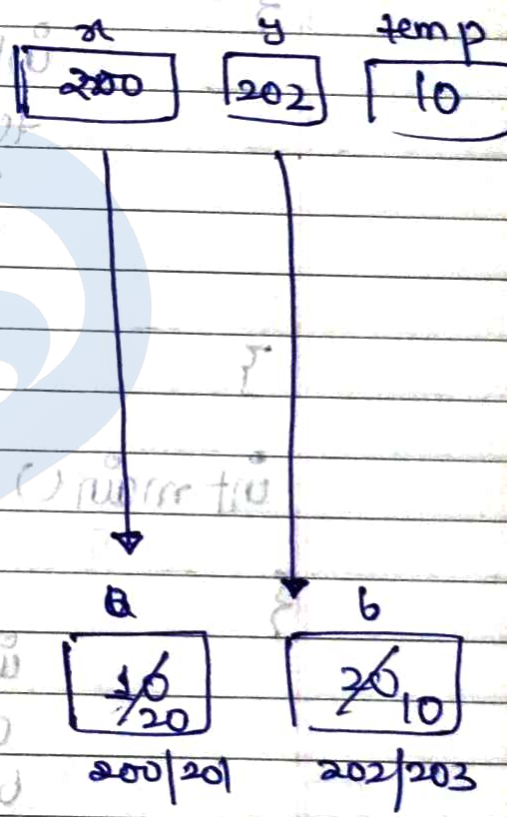
10 20

Any change done on formal parameters does not reflect in Actual parameters.

The function cannot access the variable of another function but it can access the variable using pointers. (indirectly access)

→ Call by address → It use pointers

```
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("%d %d", a, b);
}
```



→ 20 10

Any changes done on formal parameter does reflected in actual parameter.

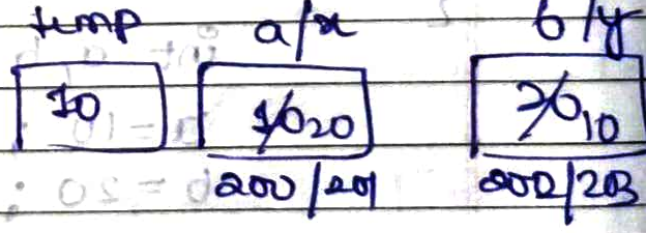
Notebook

Pg. No.

It is just like Monolithic code

→ Call by 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;
    ~~cout << printf("%d %d", a, b);~~ =

    → 20 10

}
```



Array cannot be passed by value, it always passed by address.

Array as Parameters :-

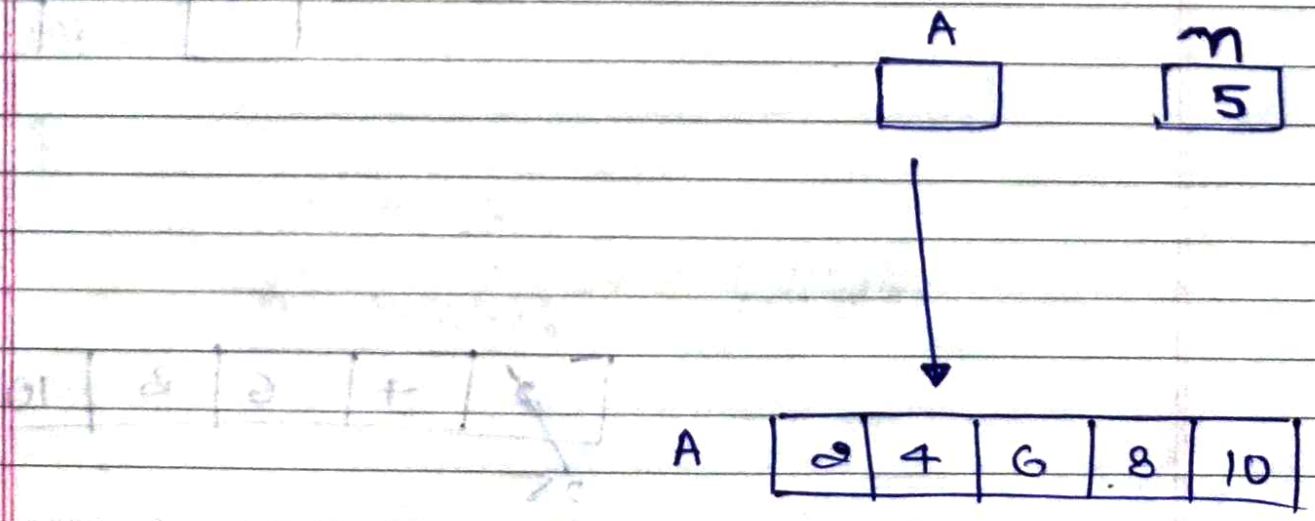
pointer to an array

Also write as *A → General method

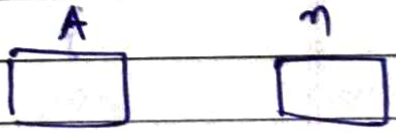
```
void fun(int A[], int n)
{
    int i;
    for(i=0; i<n; i++)
    {
        printf("%d", A[i]);
    }

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

        fun(A, 5);

    }
```

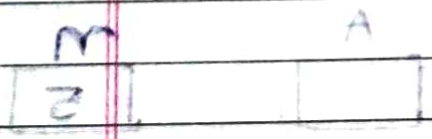
call by value

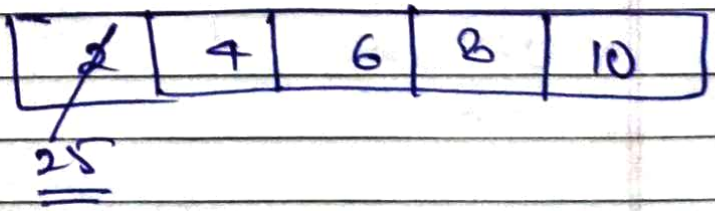
Pass by address

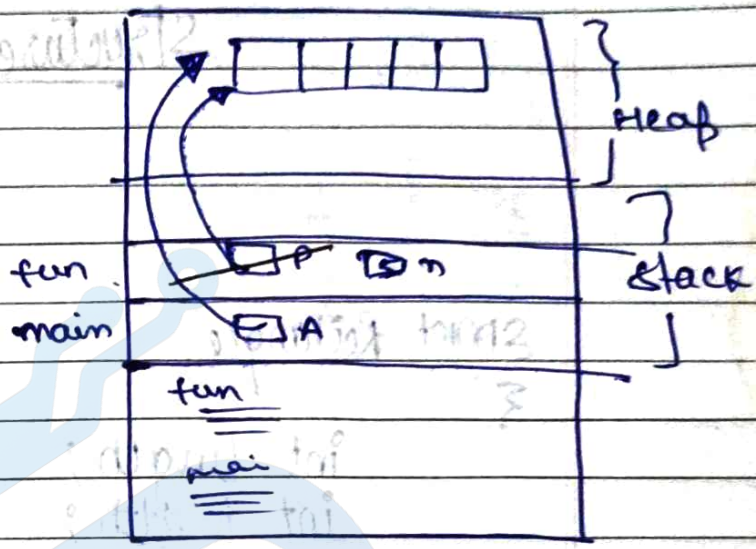


```
void fun( int A[], int n)
{
        A[0]=25;
}
int main()
{
        int  A[5] = {2,4,6,8,10};
        fun(A,5);
```







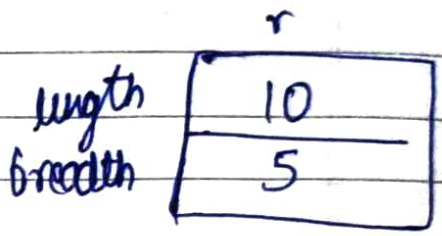



```
int [ ] fun (int n)        → call by value
{
    int *p;
    p = (int *) malloc(n*sizeof(int));
    return (p);
}

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

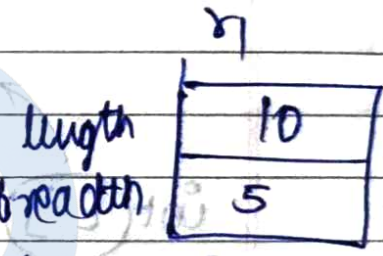
Also write (*)

If some compilers don't allow [], replace [] with *

Structure as parameter




```
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 of Rectangle : "<<area(r);
    return 0;
}
```



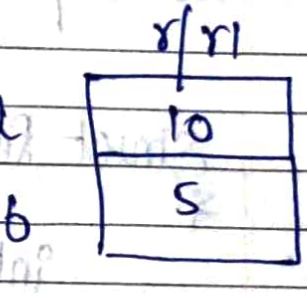
→ Seperate object is created

(It does change the actual parameter of structure)

Notebook
Pg. No.
Date / /

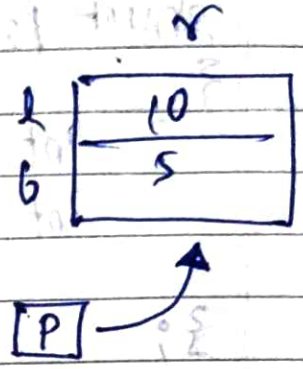
No separate object is created

Call by reference :-




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

int area(struct Rectangle &r1)
{
    r1.length++; (it change the actual parameter of structure)
    return r1.length * r1.breadth;
}

int main()
{
    Struct Rectangle r={10,5};
    Cout<<"Area of Rectangle : "<< area(r);
    return 0;
}
```

Call by address:-




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

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


}

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

We can also pass the structure by call by value, if it is containing array.


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

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

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

t1 | 2 | 4 | 6 | 8 | 10

A | 2 | 4 | 6 | 8 | 10

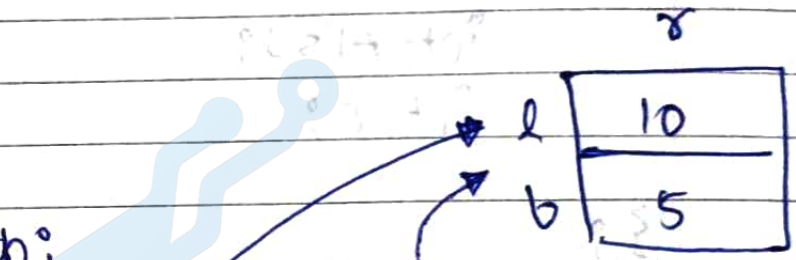
n | 5

10 9

t1 | ~~2~~ | ~~4~~ | 6 | 8 | 10

Structure 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 ( ~~int~~ struct rectangle r)
{
        return r.length * r.breadth;
}

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



Grouping of data in one place → structure

Grouping the instruction in one place → function

```
int main()
{
    struct Rectangle r;
    initialize(&r,10,5); // call by address
    area(r); // call by value
    changelength(&r,20); // call by address
    return 0;
}
```

Notebook
Pg. No.
Date / /

Class and Constructor

```
class Rectangle
{     private:
      int length;
      int breadth;
      public:
      void initialize(int l, int b)
      {
            length = l;
            breadth = b;
      }

      int area()
      {
            return length * breadth;
      }

      void changeLength(int l)
      {
            length = l;
      }
};
```

```
int main()
{
    Rectangle r;
    r.initialize(10,20);
    r.area();
    r.changelength(40);
    return 0;
}
```

Notebook
Pg. No.
Date / /

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(); // 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 :: ~Reltangle ();

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

Template Class

C++ programming supports generic function and generic classes. Generic function are template function, and class are 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 Arithametic :: sub()
{
        int c;
        c = a-b;
        return c;
}
int main()
{
        Arithametic a;
        a.add();
        a.sub();
        return 0;
}
```

Notebook
Pg. No. _____
Date ___/___/___

Template class

```
template <class T>

class Arithametic
{
    private:
      T a;
      T b;

    public:

     Arithametic (T a, T b)

    T add();
    T sub();

};

template<Class T>

Arithametic<T> :: Arithametic (T a, T b)
{

        this -> a = a;
        this -> b = b;
}
```

```
template<class T>
T Arithametic <T>:: add()
{       T c;
        c=a+b;
        return c;
}

template<class T>
T Arithametic <T>:: ~~add~~ sub()
{
        T c;
        c=a-b;
        return c;
}
int main()
{
        Arithametic <int> ar(10,5);
        cout << ar.add();
```

```
Arithametic <float> ari(10.5, 11.6);

cout << ari.add();

return 0;

}
```