



BASICS

1. Arrays
2. Structure
3. Pointer
4. Reference
5. Parameter Passing
6. Classes
7. Constructor
8. Templates

ARRAYS

Collection of similar datatypes

STRUCTURES

Collection of different data types

Struct rectangle

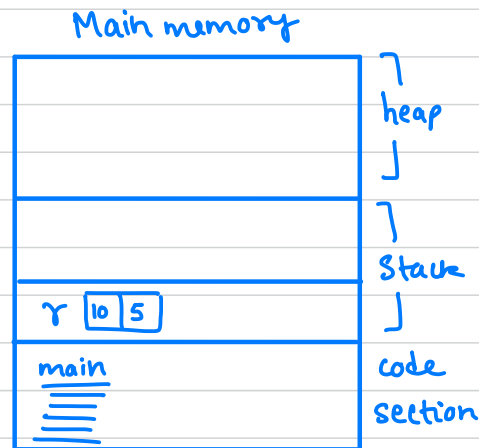
```
{  
    int length;    — 2  
    int breadth;   — 2  
};
```

4 bytes

but till here, it does not occupy any space in memory

```
int main()  
{
```

```
    struct rectangle r = {10, 5} → here it occupies memory  
    r.length = 10;  
    r.breadth = 5;  
}
```



Struct card

```
{  
    int face;  
    int shape;  
    int colour;  
};
```

```
int main()  
{
```

```
    struct card deck[52] = {....}  
    printf("%d", deck[0].face);  
    printf("%d", deck[0].shape);  
}
```

POINTERS

↳ address variables

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

```
int a=10;  
int *p; → declaration  
p=&a; → initialization  
printf(" %d", a);  
printf(" %d", *p); → dereferencing
```

USES OF POINTER

1. Accessing heap
2. Accessing resources like keyboard or mouse.
3. Parameter passing.

X

X

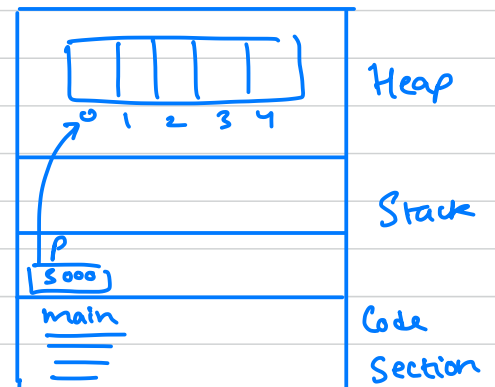
HOW TO USE POINTER FOR ALLOCATING HEAP

#include <stdlib.h> → header file for malloc()

```
int main()  
{  
    int *p;  
    p = (int *) malloc (5 * sizeof(int));  
}
```

↳ for returning
as malloc function
returns void pointer
(type casting)

↓
creating space for 5
integer values in heap.



X

X

REFERENCE IN C++

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



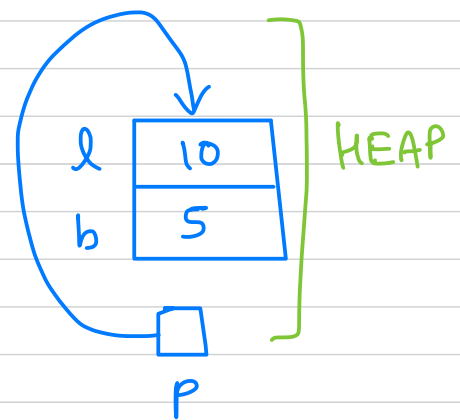
POINTER TO STRUCTURE

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

int main()
{
    struct rectangle r = {10, 5};
    struct rectangle *p = &r;

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

doesn't occupy
4 bytes of
memory
(2 bytes)



```
int main()
{
    struct rectangle *p;

    p = (struct rectangle *) malloc (sizeof (struct rectangle))

    p → length = 10;
    p → breadth = 5;
}
```

FUNCTIONS

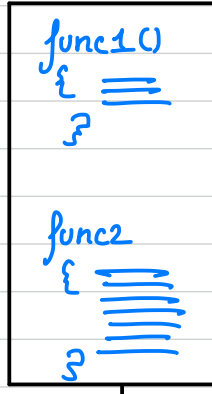
What are functions → Performs a specific task

Parameters of passing functions

- Pass by value
 - Pass by address
 - Pass by reference
-] only in C] In C++



MONOLITHIC
PROGRAMMING



MODULAR
PROGRAMMING

FUNCTION EXAMPLE

prototype

```

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

```

Formal
Parameter

```

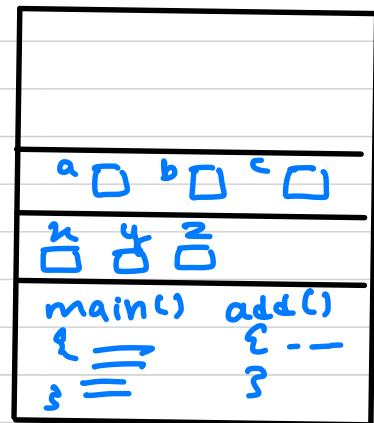
int main()
{
  int x, y, z;
  x = 10;
  y = 5;
  z = add(x, y);
  printf("%d", z);
}

```

Actual
Parameter

15

add
main

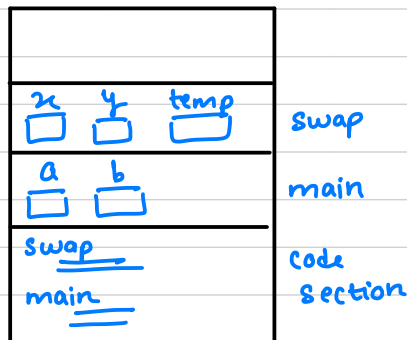


CALL BY VALUE

```
Void swap( Int x, int y)
{
    int temp;
    x = y;
    y = temp;
}
```

```
Int main()
{
    int a, b;
    a = 10;
    b = 20;
    swap(a, b);
    printf(" %d %d", a, b);
}
```

10 20

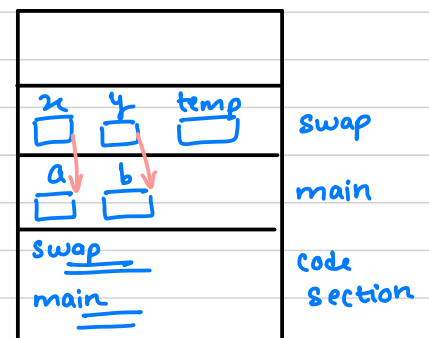
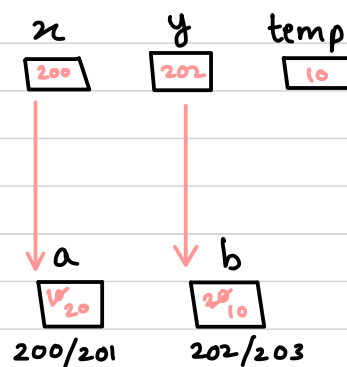


CALL BY ADDRESS

```
Void swap( Int* x, int* y)
{
    int temp;
    *x = *y;
    *y = temp;
}
```

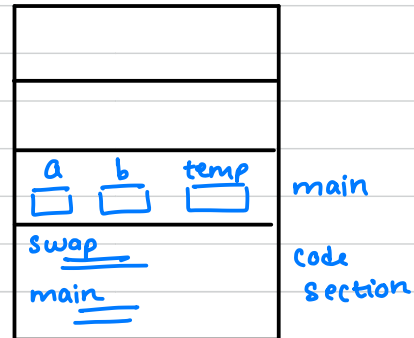
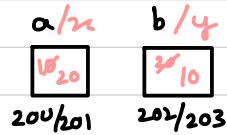
```
Int main()
{
    int a, b;
    a = 10;
    b = 20;
    swap(&a, &b);
    printf(" %d %d", a, b);
}
```

20 10



CALL BY REFERENCE (Only in c++)

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



```
int main()
{
    int a, b;
    a = 10;
    b = 20;
    swap(a, b);
    printf("%d %d", a, b);
}
```

20 10

temp
10

Only 2 bytes of memory is used as in reference, only another name is given to the pre-existing variable

ARRAY AS PARAMETER

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

→ or `int *A`

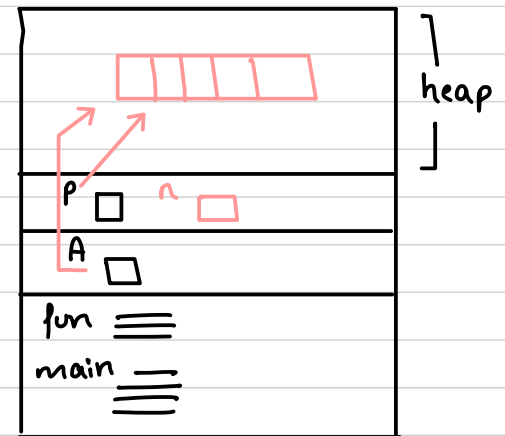
as ARRAYS can only be passed as call by address.

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

```

    ↗ or int * fun(int n)
int [] fun(int n)
{
    ↗ it returns an array
    int * p;
    p = (int *) malloc (n * sizeof (int));
    return p;
}

```



```

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

```

STRUCTURE AS PARAMETER

CALL BY VALUE

```

int area (struct rectangle r1)
{
    r1.length++;
    return r1.length * r1.bredth;
}

```

```

struct rectangle
{
    int length;
    int bredth;
};

```

```

int main()
{
    struct rectangle r = {10, 5};
    printf ("%d", area(r));
}

```

CALL BY REFERENCE

```

int area(struct rectangle *r1)

```

CALL BY ADDRESS

```

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

```

```

int main()
{
    struct rectangle r = {10, 5};
    changel (&r, 20);
}

```


PASSING ARRAYS AS CALL BY VALUE USING STRUCTURES

```
Void fun(struct test t1)
{
    t1.A[0] = 10;
}
```

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

2	4	6	8	10
5				

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

changes made in the array in the function will not be reflected as it is call by value

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 r)
```

```
{
```

```
    return r.length * r.breadth;
```

```
}
```

```
void change1 (struct rectangle *r, int l)
```

```
{
```

```
    r → length = l ;
```

```
}
```

```
int main()
```

```
{
```

```
    struct rectangle r;
```

```
    initialize (&r, 10, 5);
```

```
    area (r);
```

```
    change1 (&r, 20);
```

```
}
```

because values need to be changed, so call by address.

	r
l	10
b	5

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.area();
    r.changelength (20);
}
```

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

    public:
        rectangle()
        {
            length = breadth = 1;
        }
        rectangle (int l, int b);
        int area();
        int perimeter();
}
```

Constructors (overloaded)

Facilitators (They do operations on data members)

Accessor

```
int getlength
{
    return length;
}
```

Mutator

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

Destructor → `~ 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

template < class T >

class arithmetic

{

private:

~~T~~ int a;

~~T~~ int b;

public:

arithmetic (~~T~~ a, ~~T~~ b)

~~T~~ int add();

~~T~~ int sub();

};

→ for using various datatypes.
Using the same class for
different datatypes.

template < class T > → because effect of previous template

arithmetic < T > :: arithmetic (~~T~~ a, ~~T~~ b) has ended

{

this->a = a;

this->b = b;

}

template < class T >

~~T~~ int arithmetic :: add()

{

~~T~~ int c;

c = a + b;

return c;

}

template < class T >

int arithmetic :: sub()

{

~~T~~ int c;

c = a - b;

return c;

}

int main()

{

arithmetic < int > ar(10, 5);

cout << ar.add();

arithmetic < float > ar1(1.5, 1.2);

cout << ar1.add();

}