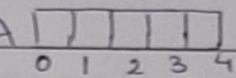
- Array
- Structure

Date 25/11/2021
Page 1

C and C++ concept

→ Arrays :→ list of same data type in consecutive memory location.

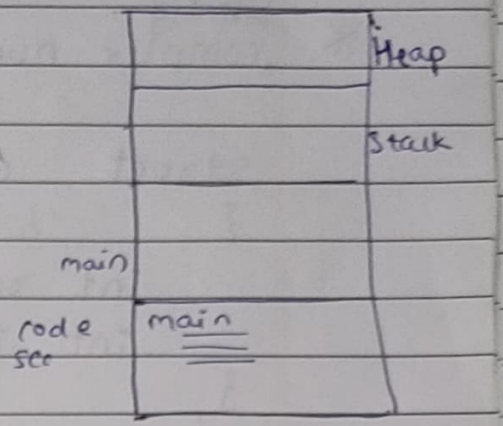
int A[5];



B[5] = { 1, 9, 8, 7, 10 }

↑ declaration of array

→ Initialization of array



→ a[5] = {0}

printf (a[2]) → 0

because a[0] was initialized

→ for(int x:A) → for each element as x in array A

{

x = x * 2

print(x)

}

→ Structure :- Collection of related data member in one name (can be same or different data type)

- Struct Rectangle → as rectangle has 2 member length & breath

{

int length; → 2 byte or 4 byte } in Struct, will have 4 or 8 byte

int breath; → 2 or 4 byte

};

int main()

{

struct Rectangle r; ← declaration

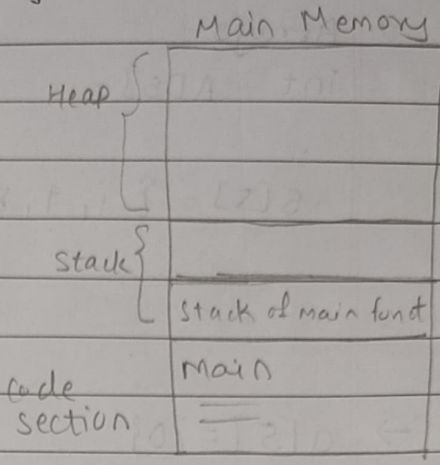
struct Rectangle r = {10, 5} ← declaration & initialization

• Structure.
• Basics of cards.

Date 25/11/2021
Page

for accessing member of structure (dot) operator is used.

r.length = 15;



* Defining complex number in structure.

```
Struct complex
{
    int real;
    int img;
};
```

* Cards

Face → Axe 1, 2, 3 ···· 10, J, K, Q

Shape → ♢, ♡, ♧ (Club), ♤ (spade)
0 1 2 3

color → Black, Red
0 1

```
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}

"lu" → for long unsigned int

"%d" → for int

Date
Page 25/11/2021

```
int main()
{ struct card deck[52];

for (i=0; i<52; i++)
{
  cout << "face";
  cin >> deck[i].face; cout << endl;
  cout << "Shape";
  cin >> deck[i].face;
  cout << endl << "color";
  cin >> deck[i].color;
}

for (i=0; i<52; i++)
{
  cout << deck[i].face << " " << deck[i].shape << " " << deck[i].color;
```

* struct rectangle
{
  int length;
  int breath;
  char x;
} r1, r2;
↳ declaring global struct variable

struct rectangle r3;

int main()
{
  printf ("lu", sizeof(r1)); → prints 12 instead of 9
  For long unsigned int

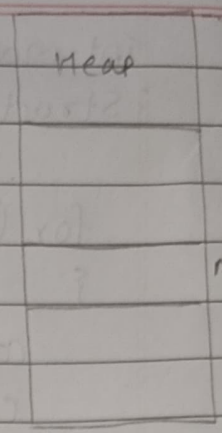
character has been alloted 4 bytes instead of one of which it uses only one. This allotment is known as padding (padding of memory is done) becz other two has been alloted 4 bytes. It becomes handy to allot 4 bytes to character.

Pointer takes 2 bytes | malloc return void pointer

Heap memory
Pointer → Address variable

Date
Page 30/11/2021

- Program (sequence of instruction in programming language that a comp need to execute) can access only code & stack section of memory. To access Heap or any external resources, pointer is required



external file or

- Uses
  1) Accessing Heap
  2) Accessing any external resources
  3) parameter passing

```
int *p; ← declaration
p = &a ; ← initializing
printf("%d", *p); ← dereferencing
```

→ To allocate Heap memory → library file → <stdlib.h>
→ function → malloc

```
int *p;
p = (int*)malloc(5*sizeof(int)); → for c
```
↑ to return int pointer ↑ array size (no of element it can hold)

```
p = new int[5]; → for c++
```

} creating array in Heap memory.

~~For delete~~
→ dealocating the memory in heap.

```
for c :- free(p);
for c++ - delete [] p;
```
↑ only for array

pointer → *r
refoence → &r

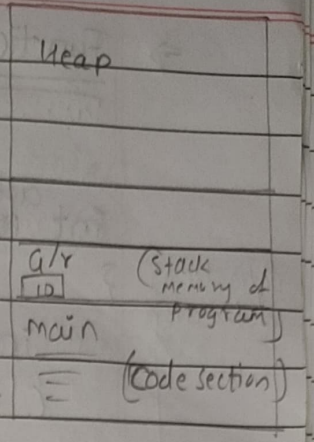
only for C++

Date
Page 1/11/2021

Refoence → use for parameter passing.

Refoence: Another name given to existing variable

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



* Reference is not like pointer & it does not consume any memory it uses the same memory of the variable it is initialized (Here r uses memory of a)

* Accessing struct member using pointer.

```
int main()
{
    struct Rectangle r = {10,5}
    struct Rectangle *p = &r
    r.length=15
    (*p).length = 20;  or  p→length=20;
    // accessing structure elements using pointer
```

Creating object variable of type of structure dynamically in heap using pointer
(↑ datatype)

→ stout rectangle *p;

for C → p = (struct rectangle*)malloc(size of(struct Rectangle));

for C++ → p = new Rectangle;

p → length = 10; or (*p).length = 10

p → breath = 5;

Monolithic Program	Modular or Procedural program
int main() { }	fun() 2 3 main() 2 3

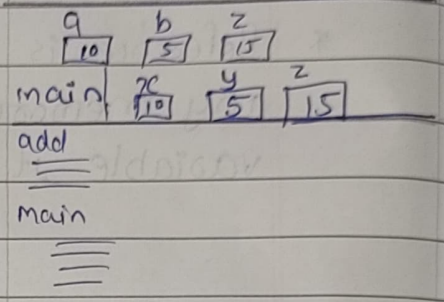
Date Page 2/12/2021

→ Functions

```
int add (int a, int b)        → Prototype or Header of function
{                               ↑ formal parameter
    int c;
    c = a+b;
    return(c);
}
```

definations of function

```
int main ()
{
    int x, y, z;
    x = 5;
    y = 10;                  ↓ Actual parameters
    z = add(x, y);
    print(z); → 15
}
```



⊗ → function variable dies as soon as controls returns from function

*p → Dereferencing.
Pointer takes 2 byte.
reference dont take any memory.

Date
Page 2/12/2021

→ Parameter passing:

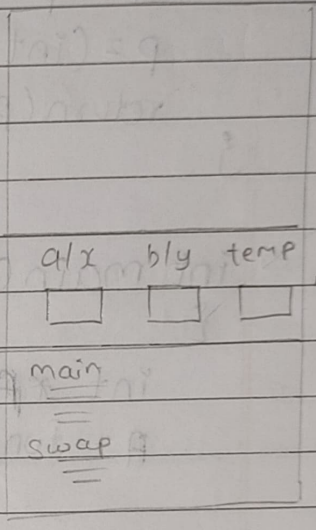
~~(suitable for returning the result)~~
call by value: formal parameter does not affect actual parameter

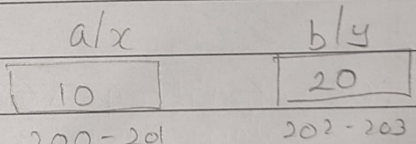
~~void (suitable for returning more than 1 variable or changing actual parameter)~~
call by address: Any change in formal parameter affect actual parameter.

C++
call by reference:- although the source code in procedural the program is monolytic while compiling
becz as soon as swap is called it becomes part of main func°

```
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);
    print (a, b)
```





pointer in latest compiler takes 8 byte
We cannot use for each loop on pointer

Date
Page 2/12/26

→ Array as parameter → can be done by call by address
pointer for an Array → A[] or *A
i.e. int funct (int A[]) or int funct (int *A[])
~~According to parameter~~.

* Returning an array from function
by creating array in heap memory in fun()

```
int * fun (int n)
{
    int *p
    p = (int *) malloc (n * sizeof (int));
    return (p);
}

int main()
{
    int *A,
    A = func(5);
```

Complete array in structure can be completely pass even by call by value.

Date
Page 2/11/21

→ Structure as Parameter (to funct)

* Call by value → Actual parameter wont be affected

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

struct Rectangle
{
   int length;
   int breath;
}

int main()
{
  struct Rectangle r,
  int area;
  area=0
  a = area(r)
  print(a);
}
```

* call by Reference
int main is same

```
into area(struct Rectangle &r1)
 {
    ____
    ____
 }
```

* variable accessing structure → p.length

pointer accessing structure → p→length

Date 2/12/21

* call by address

~~void fun(struct (Name) *r)~~

```
void fun(struct RECTANGLE *r1)
{
    r->length++;
    r->breath++;
}

int main()
{ struct Rectangle r={10,5};
  fun(&r1);
}
```

→ Creating struct in heap memory and returing it address by funct

```
struct RECTANGLE * fun()
{
    struct rectangle r*;
    r = new Rectangle(); or (struct Rectangle *)
                           malloc(sizeof(struct
                           Rectangle));
```

r = new Rectangle[3] → This will create 3 object of data type Rectangle in heap memory

Date Page 3/12/21

Class and Constructor

In C: class and constructor are not there.
But in C++ we do have them. When many functions are related mostly to some structure it can be converted to class

classes C

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

void initialize(struct Rectangle *r, l, b)
{
    r->length = l;
    r->breath = b;
}

int areaof(struct rectangle r)
{ return r.length * r.breath;
}

void changelength(struct rectangle *r, int l)
{
    r.length = l
}
```

→

```
class Rectangle
{ private:
   int length;
   int breath;
  public:                    // constructor: To initialize class variable as soon as class is delared
   Rectangle(int l, int b)
   { length = l;
     breath = b;
   }
   int area()                 // method of object Retangle
   {
   }
   int changelength(int l)
   {
   }
```

#include <iostream> → for C++
#include <stdio.h> → for C

Date Page 3/2/21

→ Initialize a variable after declaration
int i=0; instead of int i;

* Monolythic Program → program just in main function or just using one main function

→ Procedural or Modular Program → main function with other functions.
i.e main function using other functions.

→ using structure and function.

→ oop style (using class & constructor)
declaring a class → Class name (variable name)
Eg: Rectangle r;

having more than one constructor is known as constructor overloading

Date 3/2/21
Page

Class and Constructor

```
#include<iostream>
using namespace std;

class Rectangle
{
  private:
    int length;  → data member of Rectangle class
    int breath;
  public:
    Rectangle(){length=breath=1;}  → non argument constructor or default constructor
    Rectangle(int l, int b)  → parameterized constructor
    int area();
    int perimeter();
    int getlength(){return length;}   accessor or getter function
    int setlength(){length=l;}        mutator or setter function
    ~Rectangle()                      Destructor
};

Rectangle::Rectangle(int l, int b)
{
    length=l;
    breath=b;
}

int Rectange::area()   ← Return type
{
  return length*breath;
}
```

constructor overloading

facilatator

Destructor: Destroys object

~ → Destructor is called by compiler after main function.

Date
Page 3/12/21

```
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();
}
```

Now Destructor will be called and it destroys Rectangle object

:: → Scope Resolution operator

Date 3/12/2021

Template class

~~Generic call~~

Generic class → Supports all datatype (no datatype restriction)

```
template <class T>
class ~~Arithm~~ Arithametic
{ private:
    T a;
    T b;
  public:
   Arithmetic (T a, T b);
     T  add();
     T  sub();
};
```

Scope of it ends here

```
template < class T>
Arithametic <T> :: Arithmatic (T a, T b)
```

As class Arithmatic is template

```
{   original class Arithmatic elements
  this → a = a;
  this → b = b;
 }
```

same parameter

```
int main()
{
   Arithmatic <int>  ar(10,5);
   cout <<( ~~add~~ ar.add();
   Arithmatic <float> ar1(10.2, 5.1);
   cout << ar1.add();
```

As Arithmetic is template declaring T is datatype T