15/12/21

Data Structures and Algorithms
by Abdul Bari

* Section 1: Before we start

2. Introduction

* Physical Data Structures
  - Arrays
  - Matrices
  - Linked List

* Logical Data structure
  - Stack
  - Queues
  - Trees
  - Graphs
  - Hashing
  - Heap

- Recursion
- Sortings
- Asymptotic Notations

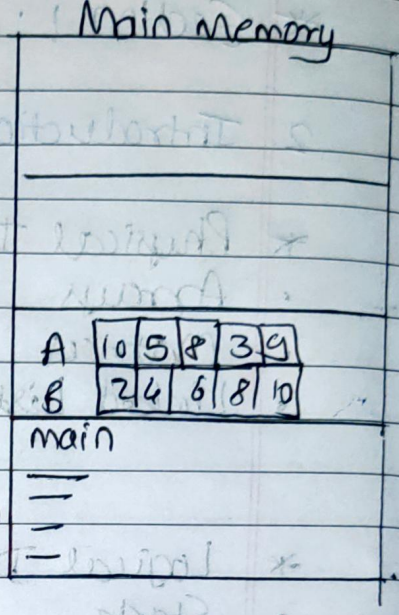
* Section 2: Essential Core C++ Concepts

3. Array Basics

```
int main()
{
  int A[5];
  int B[5] = {2,4,6,8,10};
  int i;
  for( i=0; i<5; i++)
  {
  }
  printf("%d", B[i]);
}
```

Declaration

initialization



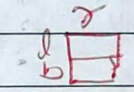
4. Practice: Array Basics

```
#include<iostream>
using namespace std;
int main()
{
    int A[10] = {2,4,6,8,10,12};
    int n;
    cout<< "Enter the no. of elements in array:"<<endl;
    cin>>n;
    int B[n];
    B[0] = 1;
```

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

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

return 0;
}
```

5. Structures

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

int main()
{
    Struct Rectangle r; <- declaration
    Struct Rectangle r = {10,5}; <- declaration & initialization
```



For accessing member of structure is used

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

* Defining complex no. in structure

$a + ib$ ← imaginary

↑ real ↑ $\sqrt{-1}$

```
struct complex {
    int real;   - 2
    int img;    - 2
};              ---
                  4
```

* Student

```
struct student {
    int roll;           - 2
    char name[25];      - 25
    char dept[10];      - 10
    char address[50];   - 50
};                      ---
                        87
```

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

* Playing cards

Face → Ace, 1, 2, ... , 10, J, K, Q
(Ace = 1; J = 11, K = 12, Q = 13)

Shape → ♠ 0, ♣ 1, ♦ 2, ♥ 3

color → Black, Red
(Black = 0, Red = 1)

```
struct Card {
    int face;      -2
    int shape;     -2
    int color;     -2
};                  6

int main() {
    struct card c;
    c.face = 1;
    c.shape = 0;
    c.color = 0;
}
```

OR

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

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

{{1,0,0}, {2,0,0}, ..
{1,1,0}, {2,1,0}, ..

52
× 6
312 bytes

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

6. Practice: Structures

```
#include <stdio.h>
#include <iostream>
using namespace std;

struct Rectangle {
    int length;
    int breadth;
    char x;
} r1, r2;
struct Rectangle r3;

int main() {
    r1 = {10, 20};
//  printf("%lu\n", sizeof(r1));
    r1.length = 15;
    r1.breadth = 25;
    cout << r1.length << endl;
    cout << r1.breadth << endl;
    return 0;
}
```

- 2
- 4
- 1
- 2
- for ease
- anyone allowed

7. Pointers

Program can access only code and stack section of memory. To access external file or Heap or any external resources pointer is required

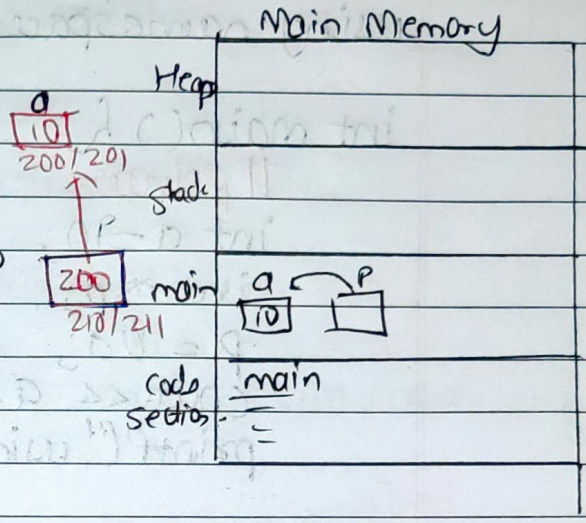
Pointer takes 2 bytes

Pointer - address variable

- Uses
  1> Accessing Heap memory
  2> Accessing any external resources
  3> Parameter passing

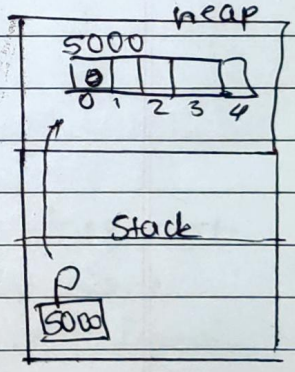
Declaration

data variable int a = 10;
Address variable int *p;
Initialization → p = &a;
printf("%d", a); → 10
Dereferencing → printf("%d", *p);



* To allocate memory in heap → #include <stdlib.h>

```
int main(){
    int *p;
    p = (int*) malloc (5 * sizeof(int));  → for C
    p = new int[5];  → for C++
```



* Deallocate memory in heap

free(p); → for C
delete [] p; → for C++
↑ only for array

8. Practice : Pointers

```
#include<stdio.h>
#include<iostream>
#include<stdlib.h>
using namespace std;

int main(){
    // pointer
    int a=10;
    int *p;
    p = &a;
    cout << a<<endl;
    printf("using pointer %d\n %d\n %d\n", *p, p, &a);
```

print value

address

```
    //pointer to arrays
    int A[5]={2,4,6,8,10};
    p= &A[0];   } Anyone
    p =A;       // p= &A invalid
    for(int i=0; i<5;i++){
        cout <<A[i] <<endl;   } Anyone
        cout <<p[i] <<endl;
    }

    // array in heap
    p=(int*) malloc(5* sizeof(int));  // using c
    p= new int[5];  // using c++
    p[0] = 10;
    p[1]= 20;
```

```
p[2] = 30;
p[3] = 40;
p[4] = 50;
for(int i=0; i<5; i++){
    cout<<p[i]<<endl;
}
// delete memory
delete [] p; // using C++
free(p);     // using C
```

For parameter passing pointer ⇒ *r

reference ⇒ &r

9. Reference in C++

Another name given to existing variable

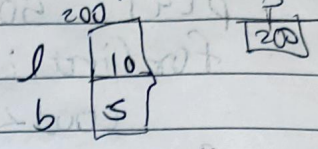
```
int main(){
    int a=10;
    int &r=a;
    cout<<a;  ⇒10
    r++;
    cout<<r;  ⇒11
    cout<<a;  ⇒11
}
```

a/r (alias) → [10]

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

110. Pointer to Structure

```
Struct Rectangle {
    int length;   -2
    int breadth;  -2
};
```




```
int main()
    Struct Rectangle r={10,5};
    struct Rectangle *p=&r;
    r.length = 15
    (*p).length=20;   } anyone
     p→length=20;
```

Creating object variable of type of structure dynamically in heap using pointer

```
    Struct Rectangle *p;
```

for c → `p=(struct Rectangle*) malloc(sizeof(struct Rectangle));`

for c++ →
```
p = new Rectangle;
p→length=10;   or (*p).length=10;
p→breadth=5;
```

```
cout << p→length << endl;
cout << p→breadth << endl;

return 0;
}
```

12. Practice : Pointers to Structure

13. Functions

Monolithic Program

```
int main()
{
=
=
=
```

Modular Prog
Procedural Prog

```
fun1()
{
=
fun2();
}
int main(){
    fun1();
    fun2();
}
```

```
int add(int a, int b){ → Prototype or Header of function
    int c;
    c = a+b;
    return c;
}
int main(){
    int x, y, z;
    x = 5;
    y = 10;
    z = add(x, y); ← Actual parameters
    printf(z); → 15
    cout << "Sum: " << z << endl;
}
```

Formal parameters

Function call

15. Parameter Passing Methods

call by reference

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

call by address

a/x 10 b/y 20

Output:

10 20 - Call by value (because of void)

20 10 - Call by address

20 10 - Call by reference

* Call by value - formal parameter does not affects actual parameters. (for returning results)

* Call by address - Any change in formal parameter affect actual parameter (for returning more than 1 variable or changing actual parameters)

*For each loop cannot be used by pointer

D D M M Y Y Y Y

For C++

* Call by reference - source code is procedural by while compiling it becomes part of monolithic

F. Array as Parameter → Pass by address only

```
void fun(int *A, n)          } Anyone
void fun(int A[], int n) {   } → Preferred
    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);
}
```

* Returning an array from function

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

19. Structure as Parameter

* Call by value ⇒ Actual parameters won't be effected.

```
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};
    printf("%d",area(r));
}
```

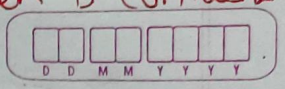
call by reference

Output
10×5 ⇒ call by value
11×5 ⇒ call by reference

* Call by address

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

Structure can be pass by value even it contains array




```
struct Test{
    int A[5];
    int n;
};
// call by value
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);
}
```

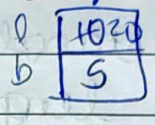
*
```
struct Rectangle{
    int length;
    int breadth;
};

void fun(struct Rectangle r){
    r.length = 10;
    cout<< "Length: " << r.length << endl <<
           "Breadth: " << r.breadth << endl;
}

void fun(struct Rectangle *p){
    p->length = 20;
    cout<< "Length: " << p->length << endl <<
           "Breadth: " << p->breadth << endl;
}
```

```
struct Rectangle *fun(){
    struct Rectangle *p;
    p = new Rectangle;
    p=(struct Rectangle*) malloc(sizeof
                    struct Rectangle));
    p->length = 30;
    p-> breadth = 40;
    return p;
}

int main(){
    struct Rectangle r = {10,5};
    fun(r);
    fun(&r);
    printf("Length: %d\nBreadth: %d\n",
                r.length, r.breadth);

    struct Rectangle *ptr = fun();
    cout << "Length: " << ptr -> length << endl <<
        "Breadth: " ptr -> breadth << endl;
    return 0;
}
```

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

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

Call by address



Call by value

Call by address

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

22 Converting a C Program to C++ class

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

    public:

    Rectangle (int l, int b){ <- (constructor- To
        length=l;              initialize class variable
        breadth = b;           as soon as class is declared
    }

    int area(){
        return length * breadth;
    }

    void changeLength (int l){
        length = l;
    }
};
int main() {
    Rectangle r(10,5);
    r.area();
    r.changeLength(20);
}
```

7. C++ 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;
      }
      int setlength{
            length = l;
      }
~Rectangle();
};
```

data member of Rectangle class

Constructor overloading

→ Default constructor

→ Parameterized constructor

← Facilitator

accessor or getter function

mutator or Setter function

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

Destructor is called after main function to destroy rectangle object

r = Object

29. Template class

Generic class → supports all datatype

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

same parameters

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