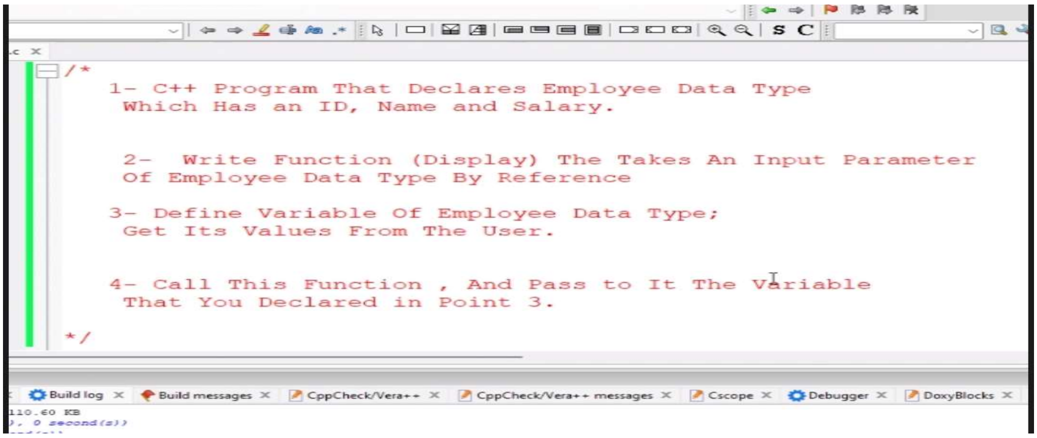
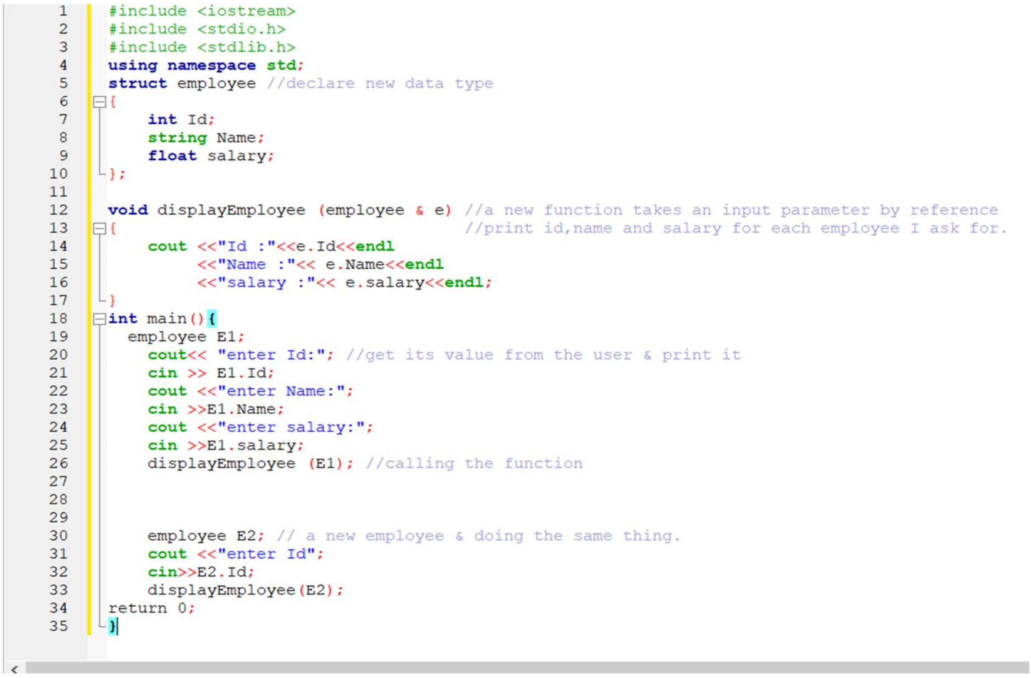
Day 1



```
/*
   1- C++ Program That Declares Employee Data Type
    Which Has an ID, Name and Salary.


    2-  Write Function (Display) The Takes An Input Parameter
    Of Employee Data Type By Reference

   3- Define Variable Of Employee Data Type;
    Get Its Values From The User.


   4- Call This Function , And Pass to It The Variable
    That You Declared in Point 3.

*/
```

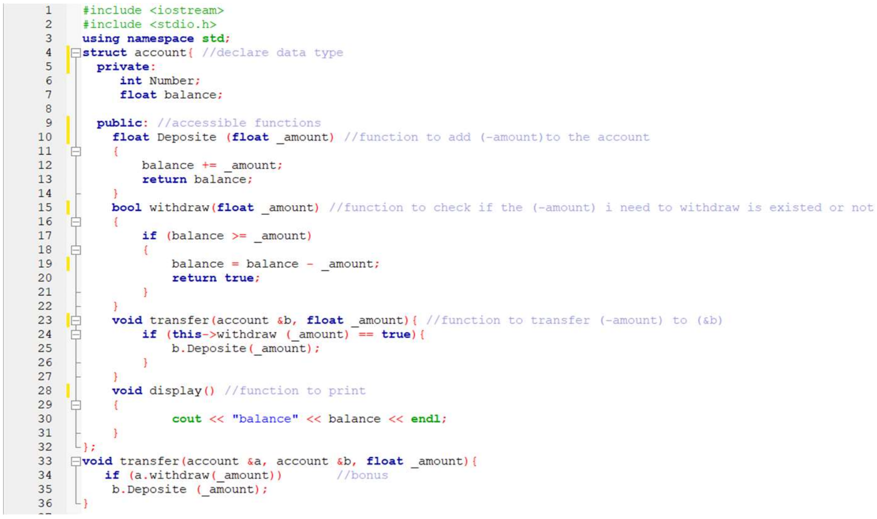
```
110.60 KB
), 0 second(s))
```



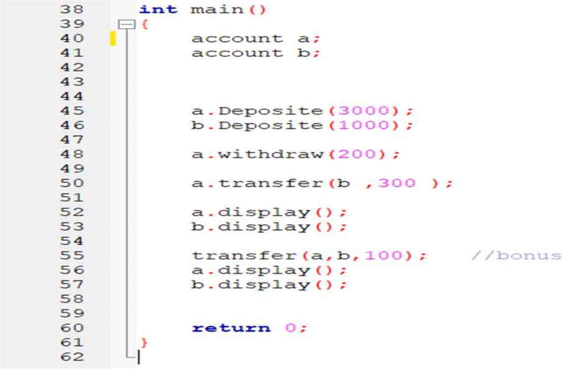
```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
using namespace std;
struct employee //declare new data type
{
    int Id;
    string Name;
    float salary;
};

void displayEmployee (employee & e) //a new function takes an input parameter by reference
{                                    //print id,name and salary for each employee I ask for.
    cout <<"Id :"<<e.Id<<endl
         <<"Name :"<< e.Name<<endl
         <<"salary :"<< e.salary<<endl;
}
int main(){
  employee E1;
    cout<< "enter Id:"; //get its value from the user & print it
    cin >> E1.Id;
    cout <<"enter Name:";
    cin >>E1.Name;
    cout <<"enter salary:";
    cin >>E1.salary;
    displayEmployee (E1); //calling the function



    employee E2; // a new employee & doing the same thing.
    cout <<"enter Id";
    cin>>E2.Id;
    displayEmployee(E2);
return 0;
}
```

Day 2

- **Create data type that Encapsulation account which has (number-balance)**
- **Support your data type with the following:**
  - **Deposit**
  - **Withdraw**
  - **transfer**
- **In main test your data type, by creating two variables, and some amount from one to another.**
- **BONUS: try to make the transfer function as Stand Alone Function, with test case.**

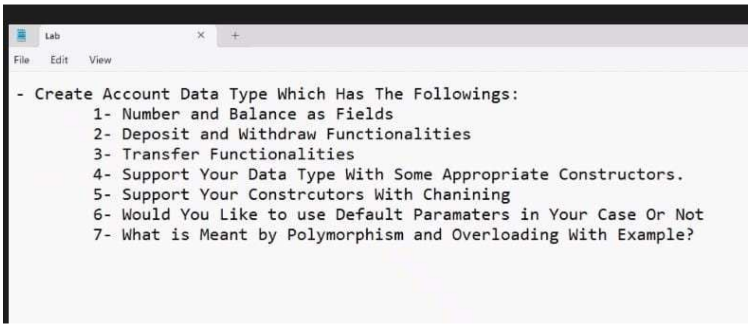


```
#include <iostream>
#include <stdio.h>
using namespace std;
struct account{ //declare data type
  private:
     int Number;
     float balance;

  public: //accessible functions
    float Deposite (float _amount) //function to add (-amount)to the account
    {
        balance += _amount;
        return balance;
    }
    bool withdraw(float _amount) //function to check if the (-amount) i need to withdraw is existed or not
    {
        if (balance >= _amount)
        {
            balance = balance - _amount;
            return true;
        }
    }
    void transfer(account &b, float _amount){ //function to transfer (-amount) to (&b)
        if (this->withdraw (_amount) == true){
            b.Deposite(_amount);
        }
    }
    void display() //function to print
    {
            cout << "balance" << balance << endl;
    }
};
void transfer(account &a, account &b, float _amount){
   if (a.withdraw(_amount))       //bonus
    b.Deposite (_amount);
}
```

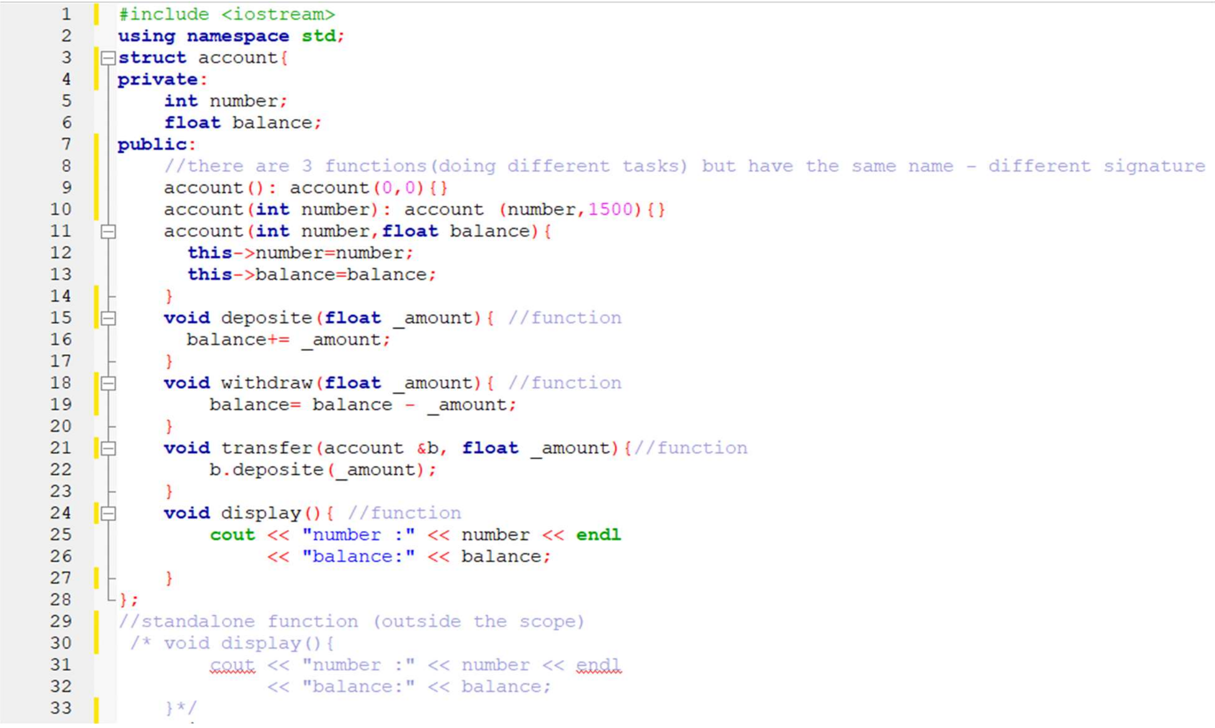


```
int main()
{
    account a;
    account b;


    a.Deposite(3000);
    b.Deposite(1000);

    a.withdraw(200);

    a.transfer(b ,300 );

    a.display();
    b.display();

    transfer(a,b,100);    //bonus
    a.display();
    b.display();

    return 0;
}
```

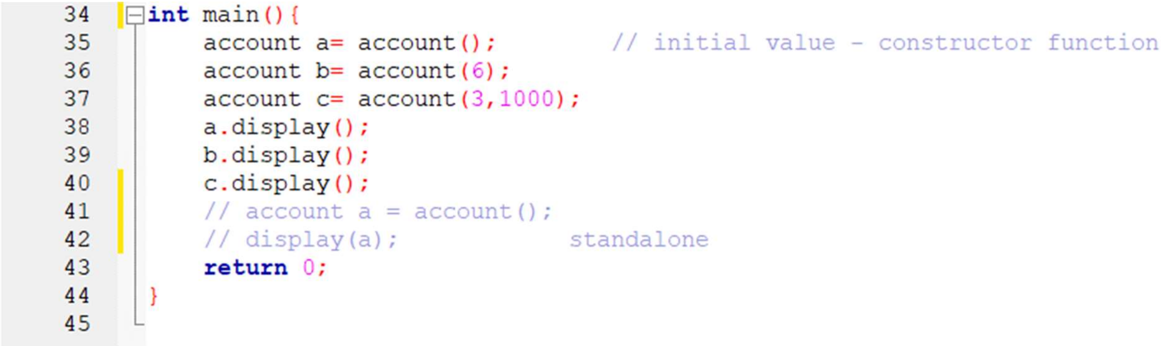
Day 3



```
- Create Account Data Type Which Has The Followings:
        1- Number and Balance as Fields
        2- Deposit and Withdraw Functionalities
        3- Transfer Functionalities
        4- Support Your Data Type With Some Appropriate Constructors.
        5- Support Your Constrcutors With Chanining
        6- Would You Like to use Default Paramaters in Your Case Or Not
        7- What is Meant by Polymorphism and Overloading With Example?
```

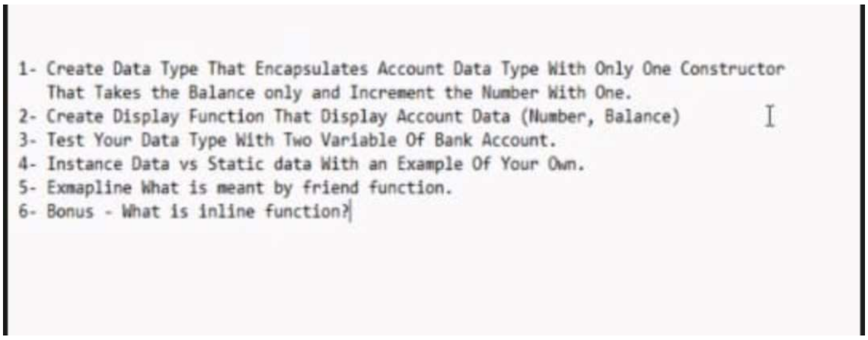


```cpp
#include <iostream>
using namespace std;
struct account{
private:
    int number;
    float balance;
public:
    //there are 3 functions(doing different tasks) but have the same name - different signature
    account(): account(0,0){}
    account(int number): account (number,1500){}
    account(int number,float balance){
      this->number=number;
      this->balance=balance;
    }
    void deposite(float _amount){ //function
      balance+= _amount;
    }
    void withdraw(float _amount){ //function
        balance= balance - _amount;
    }
    void transfer(account &b, float _amount){//function
        b.deposite(_amount);
    }
    void display(){ //function
        cout << "number :" << number << endl
             << "balance:" << balance;
    }
};
//standalone function (outside the scope)
 /* void display(){
        cout << "number :" << number << endl
             << "balance:" << balance;
    }*/
```

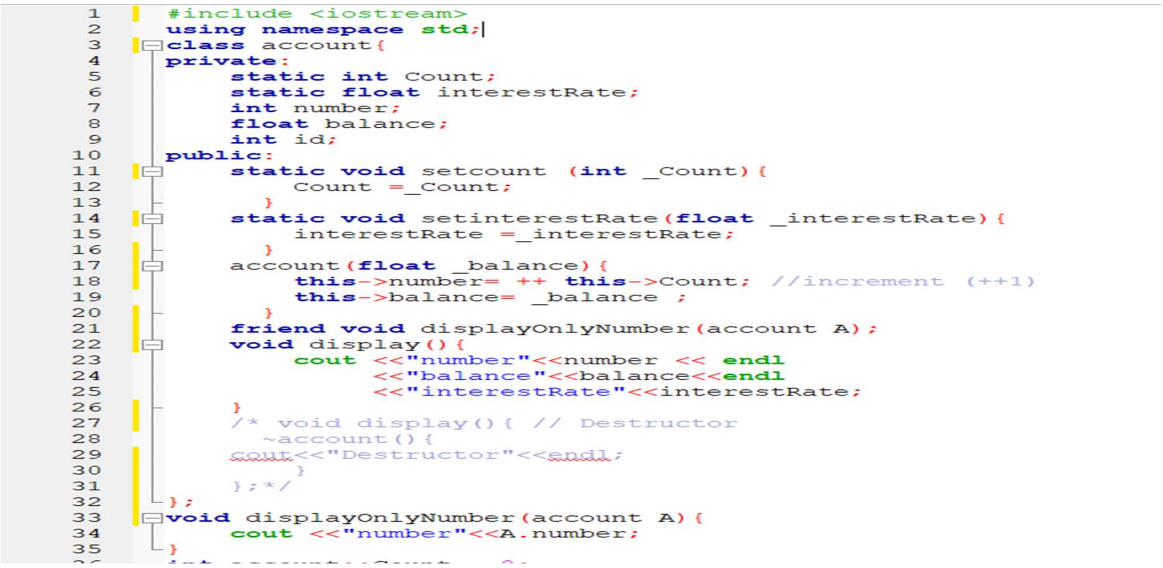


```cpp
int main(){
    account a= account();         // initial value - constructor function
    account b= account(6);
    account c= account(3,1000);
    a.display();
    b.display();
    c.display();
    // account a = account();
    // display(a);              standalone
    return 0;
}
```

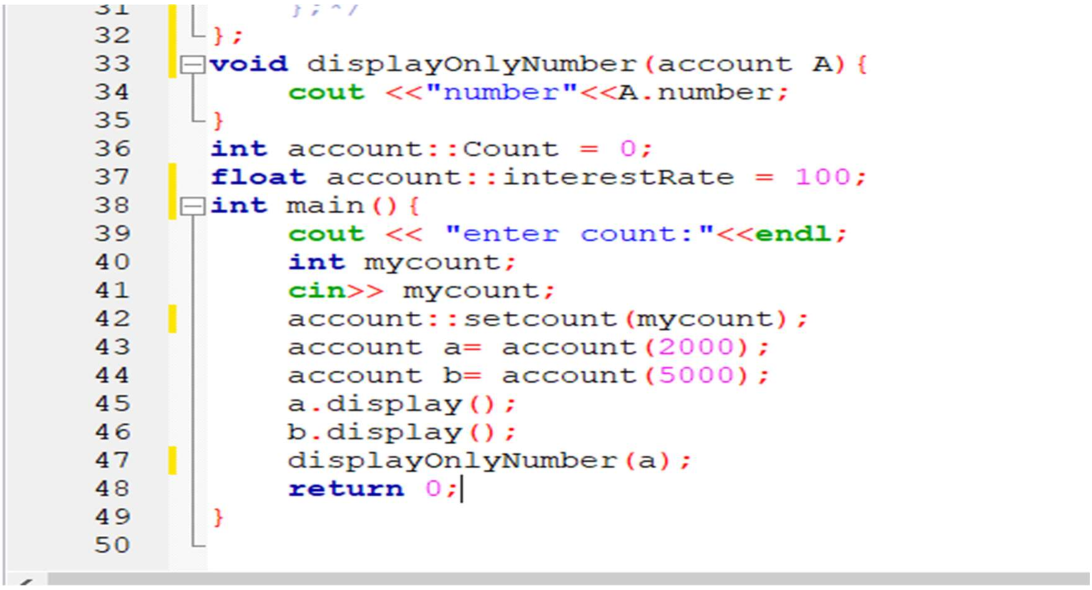
Day4



1- Create Data Type That Encapsulates Account Data Type With Only One Constructor
   That Takes the Balance only and Increment the Number With One.
2- Create Display Function That Display Account Data (Number, Balance)
3- Test Your Data Type With Two Variable Of Bank Account.
4- Instance Data vs Static data With an Example Of Your Own.
5- Exmapline What is meant by friend function.
6- Bonus - What is inline function?

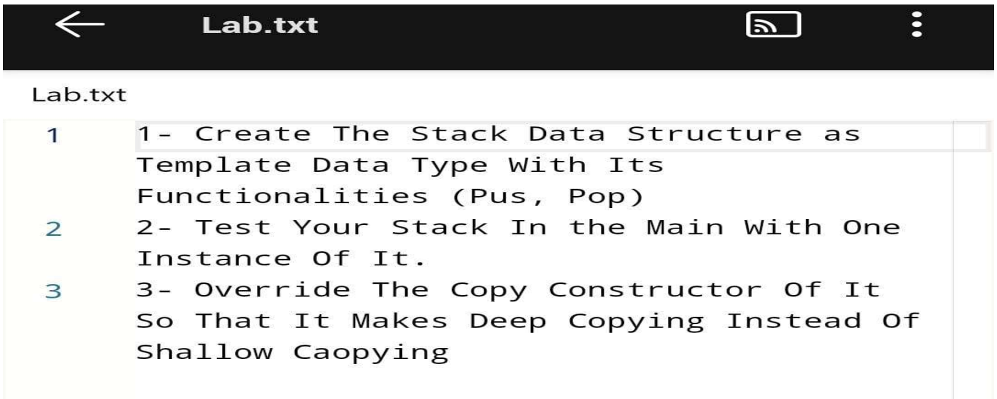


```
 #include <iostream>
using namespace std;
class account{
private:
        static int Count;
        static float interestRate;
        int number;
        float balance;
        int id;
public:
        static void setcount  (int _Count){
            Count =_Count;
          }
        static void setinterestRate(float _interestRate){
            interestRate =_interestRate;
          }
        account(float _balance){
            this->number= ++ this->Count; //increment (++1)
            this->balance= _balance ;
          }
        friend void displayOnlyNumber(account A);
        void display(){
            cout <<"number"<<number << endl
                 <<"balance"<<balance<<endl
                 <<"interestRate"<<interestRate;
        }
        /* void display(){ // Destructor
          ~account(){
        cout<<"Destructor"<<endl;
            }
        };*/
};
void displayOnlyNumber(account A){
        cout <<"number"<<A.number;
}
```



```
        };*/
};
void displayOnlyNumber(account A){
        cout <<"number"<<A.number;
}
int account::Count = 0;
float account::interestRate = 100;
int main(){
        cout << "enter count:"<<endl;
        int mycount;
        cin>> mycount;
        account::setcount(mycount);
        account a= account(2000);
        account b= account(5000);
        a.display();
        b.display();
        displayOnlyNumber(a);
        return 0;
}
```

Day 5 😐

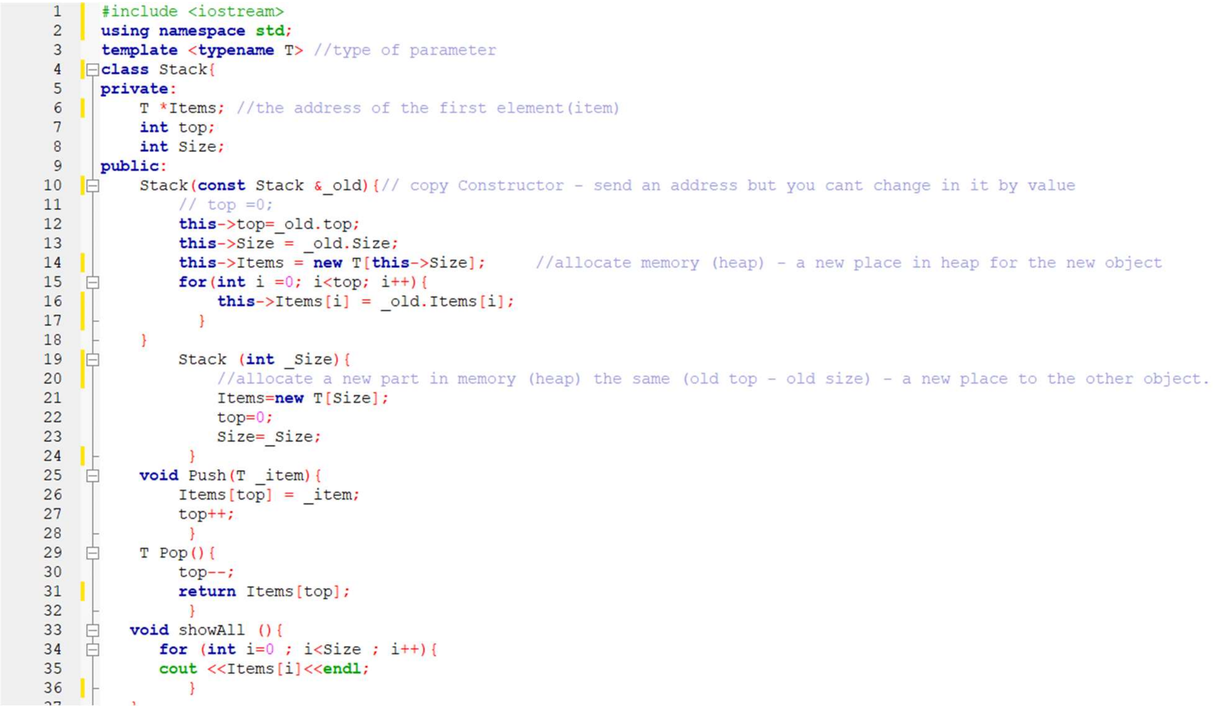


Lab.txt

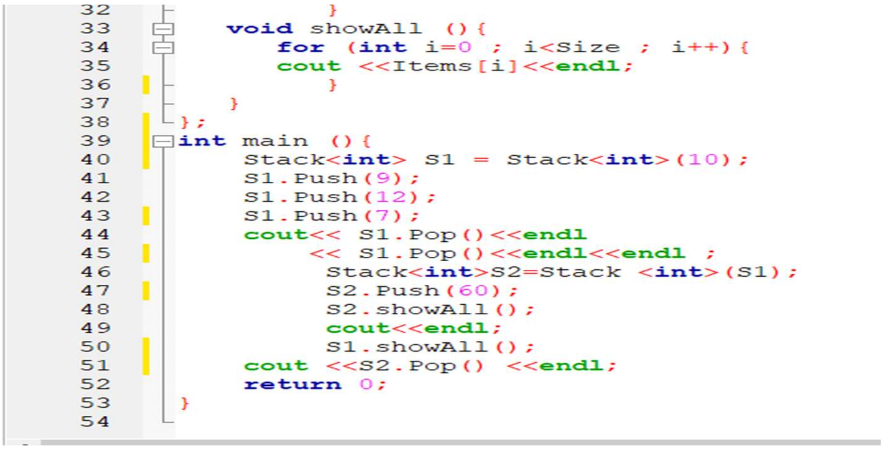
1- Create The Stack Data Structure as Template Data Type With Its Functionalities (Pus, Pop)

2- Test Your Stack In the Main With One Instance Of It.

3- Override The Copy Constructor Of It So That It Makes Deep Copying Instead Of Shallow Caopying



```cpp
#include <iostream>
using namespace std;
template <typename T> //type of parameter
class Stack{
private:
    T *Items; //the address of the first element(item)
    int top;
    int Size;
public:
    Stack(const Stack &_old){// copy Constructor - send an address but you cant change in it by value
        // top =0;
        this->top=_old.top;
        this->Size = _old.Size;
        this->Items = new T[this->Size];     //allocate memory (heap) - a new place in heap for the new object
        for(int i =0; i<top; i++){
            this->Items[i] = _old.Items[i];
          }
    }
        Stack (int _Size){
            //allocate a new part in memory (heap) the same (old top - old size) - a new place to the other object.
            Items=new T[Size];
            top=0;
            Size=_Size;
         }
    void Push(T _item){
        Items[top] = _item;
        top++;
         }
    T Pop(){
        top--;
        return Items[top];
         }
   void showAll (){
      for (int i=0 ; i<Size ; i++){
      cout <<Items[i]<<endl;
         }
```



```cpp
             }
    void showAll  (){
       for  (int  i=0  ;  i<Size  ;  i++){
       cout  <<Items[i]<<endl;
             }
    }
};
int main  (){
      Stack<int>  S1  =  Stack<int>(10);
      S1.Push(9);
      S1.Push(12);
      S1.Push(7);
      cout<<  S1.Pop()<<endl
          <<  S1.Pop()<<endl<<endl  ;
             Stack<int>S2=Stack <int>(S1);
             S2.Push(60);
             S2.showAll();
             cout<<endl;
             S1.showAll();
      cout  <<S2.Pop()  <<endl;
      return  0;
}
```