DELHI TECHNOLOGIAL UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CS203: OBJECT ORIENTED PROGRAMMING CONCEPTS

LAB FILE

SUBMITTED TO:	SUBMITTED BY:
DR. R K YADAV,	KSHITIJ SINHA,
ASSISTANT PROFESSOR,	23/CS/225,
CSE DEPARTMENT	A4, G1

INDEX

Experiment 1

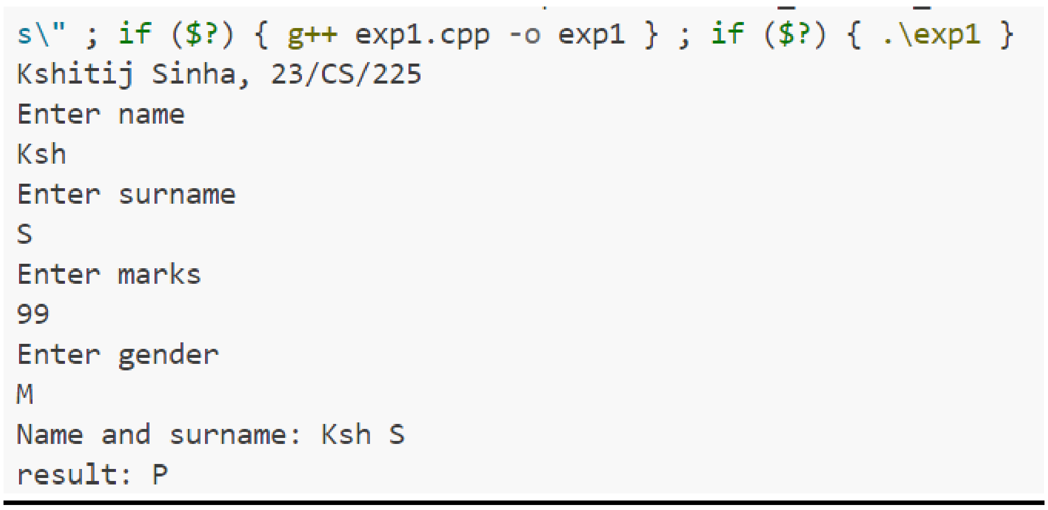
AIM: Write a C++ program to print your personal details name, surname (single char), total marks, gender (M/F), result (P/ F) by taking input from the user.

THEORY: This utilizes a class to encapsulate the data and member functions for input and output operations. The program demonstrates basic concepts of object-oriented programming, such as encapsulation and data management. It collects and displays personal details of a student, including their name, surname, total marks, gender, and result status, each of a different data-type.

CODE:

```
//Kshitij Sinha, 23/CS/225
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define endl '\n'
class info{
    public:
        string name;
        char surname;
        int marks;
        char gender;
        char result;
    info(){
        cout<<"Enter name"<<endl;
        cin>>name;
        cout<<"Enter surname"<<endl;
        cin>>surname;
        cout<<"Enter marks"<<endl;
        cin>>marks;
        cout<<"Enter gender"<<endl;
        cin>>gender;
        cout<<"Name and surname: "<<name<<" "<<surname<<endl;
        if(marks>33) result='P';
        else result='F';
        cout<<"result: "<<result<<endl;
    }
};
int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    info s1;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp1.cpp -o exp1 } ; if ($?) { .\exp1 }
Kshitij Sinha, 23/CS/225
Enter name
Ksh
Enter surname
S
Enter marks
99
Enter gender
M
Name and surname: Ksh S
result: P
```

Experiment 2

AIM: Create a class called Employee that has Empnumber and Empname as data members and member functions getdata() to input data, display() to output data. Write a main function to create an array of Employee objects. Accept and print and accept the details of at least 6 employees.

THEORY: This program defines an Employee class with data members Empnumber and Empname, encapsulating employee information. It includes member functions getdata() for inputting employee details and display() for outputting them. The main function creates an array of Employee objects, allowing the user to input and display the details of six employees, demonstrating the use of classes and arrays in C++.

CODE:

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define endl '\n'
class Employee{
    public:
    int Empnumber;
    string Empname;
    void getdata(){
        int x; string s;
        cout<<"Enter Empnumber: "<<endl;
        cin>>x;
        cout<<"Enter Empname: "<<endl;
        cin>>s;
        Empnumber=x;
        Empname=s;
    }
    void display(){
        cout<<"Enter Empnumber: "<<Empnumber<<endl;
        cout<<"Enter Empname: "<<Empname<<endl;
    }
};

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    Employee emp[6];
    for(int i=0; i<6; i++){
        emp[i].getdata();
    }
    for(int i=0; i<6; i++){
        emp[i].display();
    }
    return 0;
}
```

OUTPUT:

```
Enter Empnumber:
1
Enter Empname:
Abby
Enter Empnumber:
2
Enter Empname:
James
Enter Empnumber:
3
Enter Empname:
Jeremy
Enter Empnumber:
4
Enter Empname:
Richard
Enter Empnumber:
5
Enter Empname:
Kshitij
Enter Empnumber:
8
Enter Empname:
Charles
Enter Empnumber: 1
Enter Empname: Abby
Enter Empnumber: 2
Enter Empname: James
Enter Empnumber: 3
Enter Empname: Jeremy
Enter Empnumber: 4
Enter Empname: Richard
Enter Empnumber: 5
Enter Empname: Kshitij
Enter Empnumber: 8
Enter Empname: Charles
```

Experiment 3

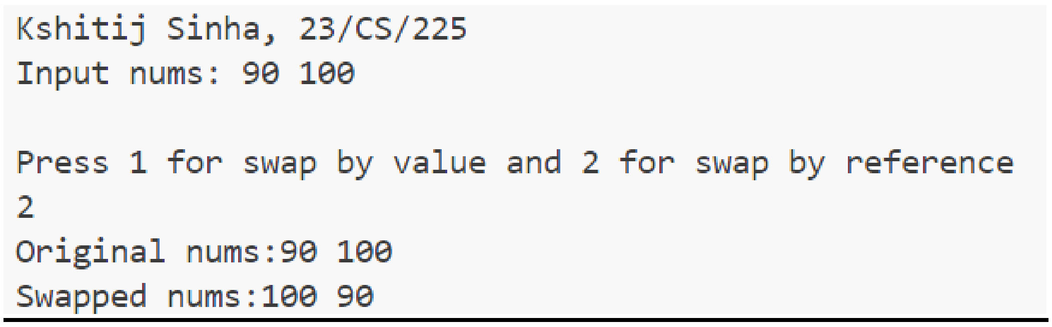
AIM: Write a C++ program to swap two numbers by both call by value, and call by reference mechanism, using two functions swap_value(), and swap_reference() respectively, by getting the choice from the user, and executing the user’s choice by switch case.

THEORY: This C++ program demonstrates the swapping of two numbers using two different mechanisms: call by value and call by reference. It defines two functions, swap_value() for swapping numbers using call by value, and swap_reference() for swapping them using call by reference. The program prompts the user to choose the swapping method and executes the corresponding function based on the user's choice using a switch case statement, illustrating the differences between the two parameter-passing techniques.

CODE:

```
// Kshitij Sinha
// 23/CS/225, A4
// Swap two numbers, first by using value and then reference. Make a switch statement for
the same
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define endl '\n'
class swapp{
    public:
    void swap_value(int a, int b){
    cout<<"Original nums:"<<a<<" "<<b<<endl;
    int c=a;
    a=b;
    b=c;
    cout<<"Swapped nums:"<<a<<" "<<b<<endl;
}
void swap_reference(int *a, int*b){
    cout<<"Original nums:"<<*a<<" "<<*b<<endl;
    int c=*a;
    *a=*b;
    *b=c;
    cout<<"Swapped nums:"<<*a<<" "<<*b<<endl;
}
};
int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    int x,y;
    cout<<"Input nums: ";
    cin>>x>>y;
    cout<<endl;
    int a;
    cout<<"Press 1 for swap by value and 2 for swap by reference"<<endl;
    cin>>a;
    swapp s1;
    switch(a){
        case 1:
            s1.swap_value(x,y);
            break;
        case 2:
            s1.swap_reference(&x, &y);
            break;
    }
}
```

OUTPUT:



```
Kshitij Sinha, 23/CS/225
Input nums: 90 100

Press 1 for swap by value and 2 for swap by reference
2
Original nums:90 100
Swapped nums:100 90
```

Experiment 4

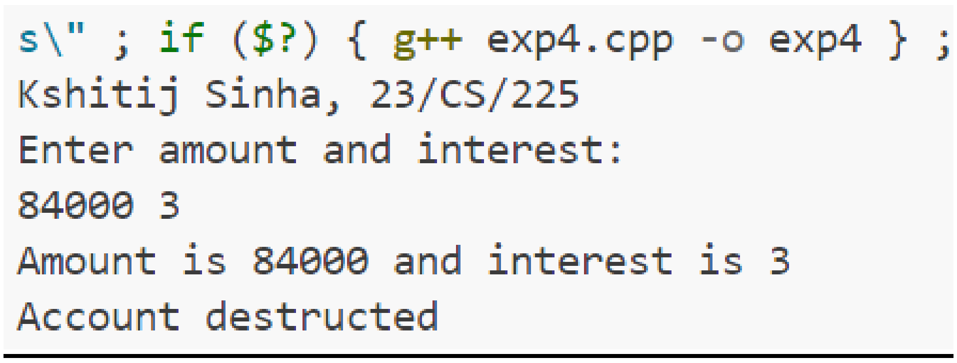
AIM: Write a C++ program to create a simple banking system in which the initial balance and the rate of interest are read from the keyboard and these values are initialized using the constructor. The destructor member function is defined in this program to destroy the class object created using constructor member function.

THEORY: This C++ program implements a simple banking system where an initial balance and a rate of interest are inputted by the user and initialized through a constructor. The constructor ensures that the object's data members are set up properly upon creation. Additionally, the program includes a destructor that is invoked when the object goes out of scope, allowing for cleanup and resource management.

CODE:

```cpp
// Kshitij Sinha
// 23/CS/225, A4
// Create a banking system using constructors and destructors and a show function

#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define endl '\n'
class bank{
    int amount=0, interest=0;
    public:
    bank(int, int);
    void show(){
        cout<<"Amount is "<<amount<<" and interest is "<<interest<<endl;
    }
    ~bank(){
        cout<<"Account destructed"<<endl;
    }
};

bank::bank(int a, int b){
    amount=a;
    interest=b;
}

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    int a,b;
    cout<<"Enter amount and interest: "<<endl;
    cin>>a>>b;
    bank s(a,b);
    s.show();
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp4.cpp -o exp4 } ;
Kshitij Sinha, 23/CS/225
Enter amount and interest:
84000 3
Amount is 84000 and interest is 3
Account destructed
```

Experiment 5

AIM: Write a program to accept five different numbers by creating a class called friendfunc1 and friendfunc2 taking 2 and 3 arguments respectively and calculate the average of these numbers by passing object of the class to friend function.

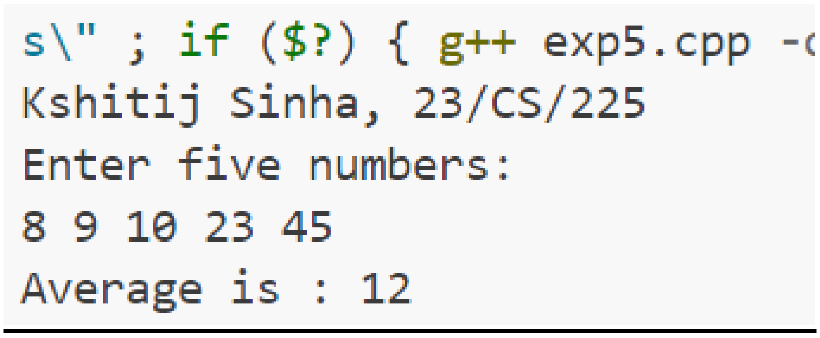
THEORY: Friend functions in C++ allow external functions to access the private and protected members of a class. In this C++ program, a single friend function is defined that can access the private members of both friendfunc1 and friendfunc2 classes. This allows the function to calculate the average of five different numbers stored within these classes, demonstrating how friend functions enable seamless interaction between multiple classes while maintaining encapsulation. By granting access to class internals, the friend function can perform necessary calculations without needing to be a member of either class, thus enhancing modularity and code clarity.

CODE:

```
// Kshitij Sinha
// 23/CS/225, A4
// Take the average of five numbers from two classes using a friend function

#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define endl '\n'

class fun2;
class fun1{
    public:
    double a,b,c;
    fun1(double ,double, double);

    void show(){
        cout<<a<<" "<<b<<" "<<c<<endl;
    }
    friend double avg(double, double,double, double, double);
};

fun1::fun1(double x, double y, double z){
    a=x, b=y, c=z;
}

class fun2{

    public:
    double d,e;
    fun2(double , double);
```

```cpp
    void show(){
        cout<<d<<" "<<e<<endl;
    }
    friend double avg(double, double, double, double, double);
};

fun2::fun2(double x, double y){
    d=x, e=y;
}


double avg(fun1 aa, fun2 bb){
    double ans=aa.a+aa.b+aa.c+bb.d+bb.e;
    ans/=5.0;
    return ans;
}

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    cout<<"Enter five numbers: "<<endl;
    double q,w,e,r,t;
    cin>>q>>w>>e>>r>>t;
    fun1 aa(q,w,e);
    fun2 bb(e,r);
    cout<<"Average is : "<<avg(aa, bb)<<endl;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp5.cpp -c
Kshitij Sinha, 23/CS/225
Enter five numbers:
8 9 10 23 45
Average is : 12
```

Experiment 6

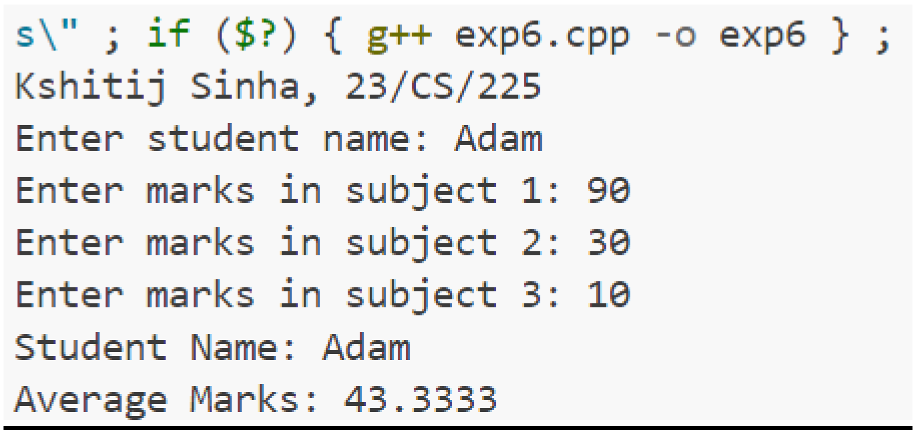
AIM: Write a program to accept the student detail such as name and 3 different marks by get_data() method and display the name and average of marks using display() method. Define a friend class for calculating the average of marks using the method mark_avg().

THEORY: This program illustrates the use of friend classes in C++, where one class is granted access to the private members of another class. It defines a student class that collects details such as the student's name and three marks through the get_data() method. A friend class is implemented to calculate the average of the marks using the mark_avg() method, enabling cleaner access to the student's private data.

CODE:

```cpp
// Kshitij Sinha
// 23/CS/225, A4
// Create a student class; ask the user for name and marks, create friend function to show average, get_data() and display() in class


#include <bits/stdc++.h>
using namespace std;
class student;
class avg_class{
    public:
    double mark_avg(student s1);
};
class student{
    string name;
    int marks1, marks2, marks3;
    public:
    void get_data(){
        string nm;
        cout<<"Enter student name: ";
        cin>>nm; name = nm;
        int mrk;
        cout<<"Enter marks in subject 1: ";
        cin>>mrk; marks1 = mrk;
        cout<<"Enter marks in subject 2: ";
        cin>>mrk; marks2 = mrk;
        cout<<"Enter marks in subject 3: ";
        cin>>mrk; marks3 = mrk;
    }
    friend class avg_class;
    void display(){
        avg_class a1;
        double avg = a1.mark_avg(*this);
        cout<<"Student Name: "<<name<<endl;
        cout<<"Average Marks: "<<avg<<endl;
    }
};
double avg_class::mark_avg(student s1){
    double avg = (s1.marks1 + s1.marks2 + s1.marks3)/3.0;
    return avg;
}
int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    student s1;
    s1.get_data();
    s1.display();
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp6.cpp -o exp6 } ;
Kshitij Sinha, 23/CS/225
Enter student name: Adam
Enter marks in subject 1: 90
Enter marks in subject 2: 30
Enter marks in subject 3: 10
Student Name: Adam
Average Marks: 43.3333
```

Experiment 7

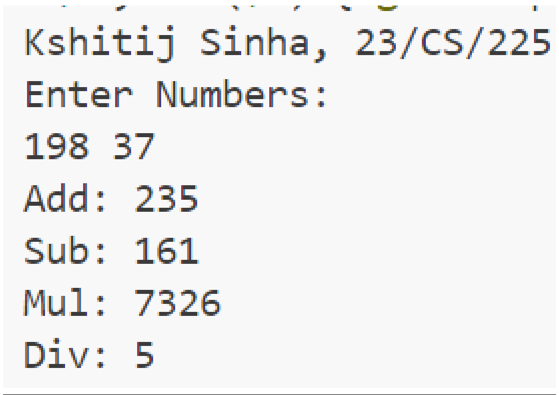
AIM: Write a C++ program to perform different arithmetic operations such as addition, subtraction, division, modulus, and multiplication using inline functions.

THEORY: They are a special type of function which tell the compiler that the function code should be inserted directly into the calling code instead of being called through the usual function call mechanism. This can enhance performance by reducing function call overhead, especially for small, frequently called functions. In the context of this program, inline functions are used to perform various arithmetic operations—addition, subtraction, multiplication, division, and modulus— allowing for efficient execution of these operations while keeping the code organized and readable.

CODE:

```cpp
// Kshitij Sinha
// 23/CS/225, A4
// Implement inline functions for addition, subtraction, multiplication, and division

#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define endl '\n'
class func{
    public:
    inline int mul(int a, int b){
        return a*b;
    }
    inline int divd(int a, int b){
        if(b==0){
            return -1;
        }
        return a/b;
    }
    inline int sub(int a, int b){
        return a-b;
    }
    inline int add(int a, int b){
        return a+b;
    }
};
int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    func s;
    cout<<"Enter Numbers: "<<endl;
    int a,b;
    cin>>a>>b;
    cout<<"Add: "<<s.add(a,b)<<endl;
    cout<<"Sub: "<<s.sub(a,b)<<endl;
    cout<<"Mul: "<<s.mul(a,b)<<endl;
    cout<<"Div: "<<s.divd(a,b)<<endl;
}
```

OUTPUT:



```
Kshitij Sinha, 23/CS/225
Enter Numbers:
198 37
Add: 235
Sub: 161
Mul: 7326
Div: 5
```

Experiment 8

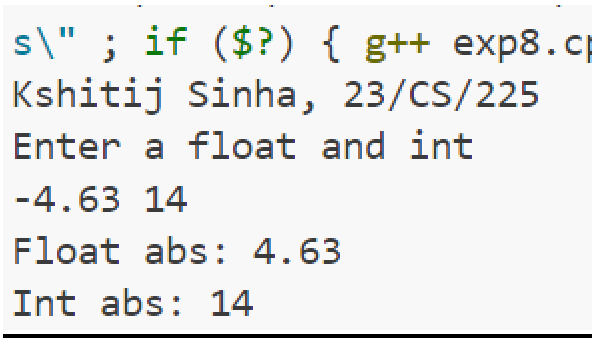
AIM: Write a C++ program to return absolute value of variable types integer and float using function overloading.

THEORY: Function overloading in C++ allows multiple functions to have the same name but differ in the number or type of their parameters. This C++ code demonstrates function overloading by implementing two versions of the absolute() function to return the absolute value for both integer and float variable types. By allowing multiple functions to share the same name but differ in their parameter types, the code enhances readability and provides a consistent interface for users.

CODE:

```cpp
// Kshitij Sinha
// 23/CS/225, A4
#include <bits/stdc++.h>
using namespace std;
float absolute(float a){
    if(a>0) return a;
    return float(a)*float(-1.0);
}

int absolute(int a){
    if(a>0) return a;
    else return a*(-1);
}

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    cout<<"Enter a float and int\n";
    float f; int i;
    cin>>f>>i;
    cout<<"Float abs: "<<absolute(f)<<endl;
    cout<<"Int abs: "<<absolute(i)<<endl;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp8.c
Kshitij Sinha, 23/CS/225
Enter a float and int
-4.63 14
Float abs: 4.63
Int abs: 14
```

Experiment 9

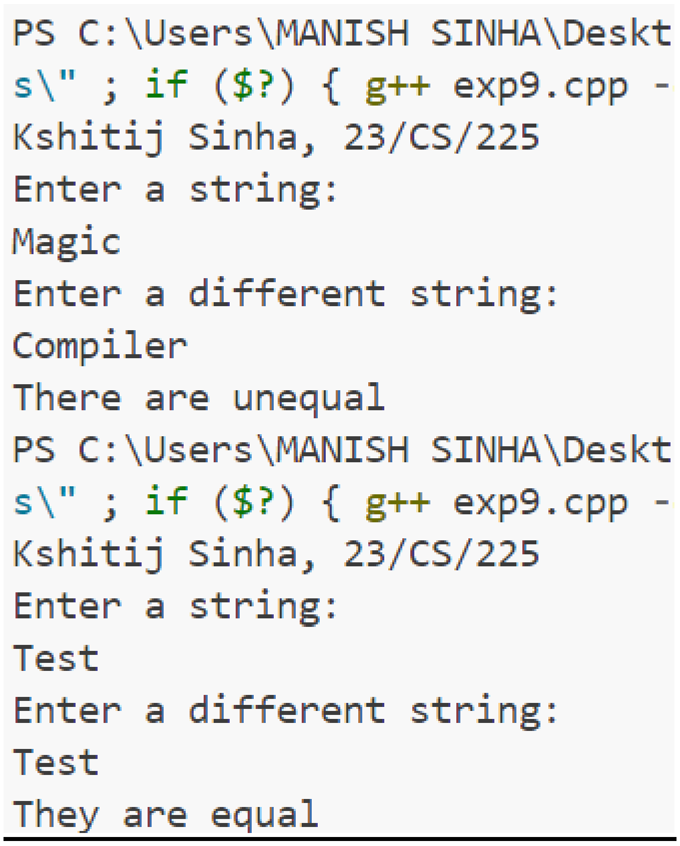
AIM: WAP to perform string operations using operator overloading in C++: (i) = String Copy, (ii) == Equality, (iii) + Concatenation.

THEORY: Operator overloading in C++ allows developers to redefine the behaviour of operators for user-defined types. In this program, operator overloading is utilized to implement string operations, including the assignment operator (=) for string copying, the equality operator (==) for comparing two strings, and the addition operator (+) for string concatenation. This approach enhances code readability and usability, allowing string objects to be manipulated in a manner similar to built-in data types.

CODE:

```
// Kshitij Sinha
// 23/CS/225, A4
#include <bits/stdc++.h>
using namespace std;
class expp{
    public:
    string s;

    expp(){
        s="";
    }
    expp(string p){
        s=p;
    }
    void operator =(string p){
        s=p;
    }
    bool operator ==(expp t1){
        if(t1.s.size()!=s.size()){
            return false;
        }
        else{
            int valid=true;
            for(int i=0; i<s.size(); i++){
                if(s[i]!=t1.s[i]){
                    valid=false;
                    break;
                }
            }
            return valid;
        }
    }
    string operator +(expp t1){
        for(int i=0; i<t1.s.size(); i++){
            s.push_back(t1.s[i]);
        }
        return s;
    }
};

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    cout<<"Enter a string: "<<endl;
    string s;
    cin>>s;
    expp e;
    e=s;
    cout<<"Enter a different string: "<<endl;
    string p;
    cin>>p;
    cout<<(e==p?"They are equal":"There are unequal")<<endl;
}
```

OUTPUT:



```
PS C:\Users\MANISH SINHA\Deskt
s\" ; if ($?) { g++ exp9.cpp -
Kshitij Sinha, 23/CS/225
Enter a string:
Magic
Enter a different string:
Compiler
There are unequal
PS C:\Users\MANISH SINHA\Deskt
s\" ; if ($?) { g++ exp9.cpp -
Kshitij Sinha, 23/CS/225
Enter a string:
Test
Enter a different string:
Test
They are equal
```

Experiment 10

AIM: Consider a class network as given below. The class master derives information from both account and admin classes which in turn derive information from the class person. Define all the four classes and write a program to create, update and display the information contained in master objects. Also demonstrate the use of different access specifiers by means of member variables and member functions.

THEORY:

Inheritance: A fundamental object-oriented programming concept that allows a class (derived class) to inherit attributes and behaviours (methods) from another class (base class), promoting code reuse and establishing a hierarchical relationship.

Access Specifiers:

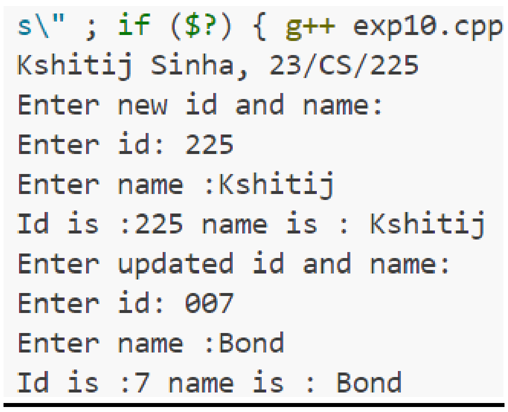
- Public: Members declared as public are accessible from any part of the program, including outside the class.
- Protected: Members declared as protected are accessible within the class itself and by derived classes, but not from outside the class hierarchy.
- Private: Members declared as private are accessible only within the class itself, restricting access from derived classes and outside code.

CODE:

```
// Kshitij Sinha
// 23/CS/225, A4
#include<bits/stdc++.h>
using namespace std;
class person{
    protected:
    int id;
    string name;
    public:
    void show(){
        cout<<"Id is :"<<id<<" name is : "<<name<<endl;
    }
};

class admin: virtual public person{
    public:
    void update(){
        int idd;
        string namee;
        cout<<"Enter updated id and name: "<<endl;
        cout<<"Enter id: ";
        cin>>idd;
        cout<<"Enter name :";
```

```cpp
        cin>>namee;
        id=idd;
        name=namee;
        show();
    }
};

class account: virtual public person{
    public:
    void create(){
        int idd;
        string namee;
        cout<<"Enter new id and name: "<<endl;
        cout<<"Enter id: ";
        cin>>idd;
        cout<<"Enter name :";
        cin>>namee;
        id=idd;
        name=namee;

        show();

    }
};

class master: public account, public admin{
    public:
    master(){
        create();
        update();
    }
};

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    master m1;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp10.cpp
Kshitij Sinha, 23/CS/225
Enter new id and name:
Enter id: 225
Enter name :Kshitij
Id is :225 name is : Kshitij
Enter updated id and name:
Enter id: 007
Enter name :Bond
Id is :7 name is : Bond
```

Experiment 11

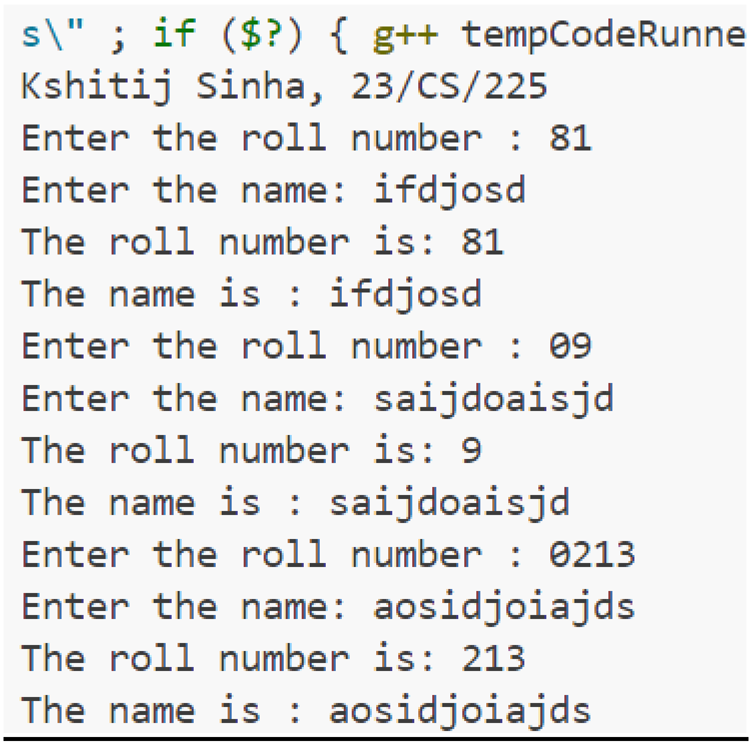
AIM: Write a C++ program to create three objects for a class named pntr_obj with data members such as roll_no and name. Create a member function set_data() for setting data values, and print() member function to print which object has invoked it using 'this' pointer.

THEORY: This C++ program defines a class named pntr_obj with data members roll_no and name. It includes a member function set_data() to initialize these values and a print() member function that utilizes the this pointer to identify the specific object invoking it. The use of the this pointer allows the program to access the calling object's members, demonstrating the concept of object identity and encapsulation.

CODE:

```cpp
// Kshitij Sinha
// 23/CS/225, A4
#include <bits/stdc++.h>
using namespace std;
class pntr_obj{
    protected:
        int roll_no;
        string name;
    public:
    void set_data(){
        int x;
        string y;
        cout<<"Enter the roll number : ";
        cin>>x;
        cout<<"Enter the name: ";
        cin>>y;
        this->roll_no=x;
        this->name=y;
    }
    void print(){
        cout<<"The roll number is: "<<this->roll_no;
        cout<<endl;
        cout<<"The name is : "<<this->name;
        cout<<endl;
    }
    pntr_obj(){
        set_data();
        print();
    }
};

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    pntr_obj p1, p2, p3;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ tempCodeRunne
Kshitij Sinha, 23/CS/225
Enter the roll number : 81
Enter the name: ifdjosd
The roll number is: 81
The name is : ifdjosd
Enter the roll number : 09
Enter the name: saijdoaisjd
The roll number is: 9
The name is : saijdoaisjd
Enter the roll number : 0213
Enter the name: aosidjoiajds
The roll number is: 213
The name is : aosidjoiajds
```

Experiment 12

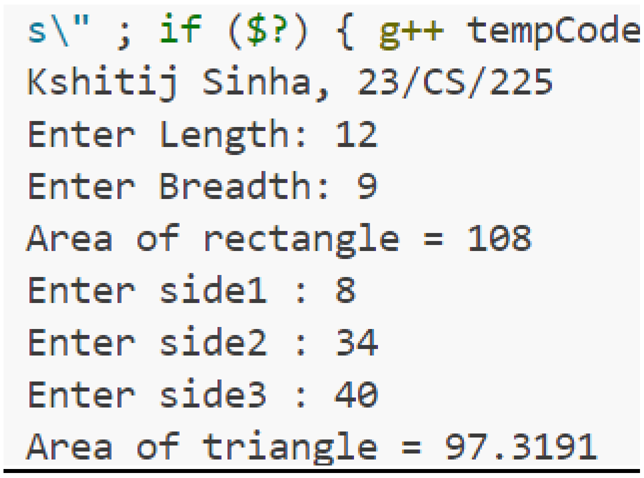
AIM: Write a C++ program to explain virtual function (polymorphism) by creating a base class c_polygon which has virtual function area(). Two classes c_rectangle and c_triangle derived from c_polygon and they have area() to calculate and return the area of rectangle and triangle respectively.

THEORY: Virtual functions are member functions in a base class that can be overridden in derived classes, enabling polymorphism in C++. Polymorphism allows for invoking derived class methods through base class pointers or references, ensuring the correct method is called at runtime. This mechanism facilitates dynamic binding, promoting code flexibility and reusability. The base class c_polygon defines a virtual function area(), which is overridden in the derived classes c_rectangle and c_triangle. Each derived class provides its specific implementation of the area() function to calculate the area of a rectangle and a triangle, respectively.

CODE:

```
# include <iostream>
# include <cmath>
using namespace std;
class c_polygon{
    public:
    virtual void area() = 0;
};
class c_rectangle : public c_polygon{
    int l, b;
    public:
    void get_dimensions(){
        cout<<"Enter Length: "; cin>>l;
        cout<<"Enter Breadth: "; cin>>b;
    }
    void area(){
        cout<<"Area of rectangle = "<<l*b<<endl;
    }
 };
class c_triangle : public c_polygon{
    int a, b, c;
    public:
    void get_dimensions(){
        cout<<"Enter side1 : "; cin>>a;
        cout<<"Enter side2 : "; cin>>b;
        cout<<"Enter side3 : "; cin>>c;
    }
    void area(){
        double s = (a + b + c)/2;
        double area = sqrtl(s*(s - a)*(s - b)*(s - c));
        cout<<"Area of triangle = "<<area<<endl;
    }
};
int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    c_polygon *ptr = NULL;
    c_rectangle rect;
    rect.get_dimensions();
    ptr = &rect;
    ptr->area();
    c_triangle tri;
    tri.get_dimensions();
    ptr = &tri;
    ptr->area();
}
```

OUTPUT:



```
s\" ; if ($?) { g++ tempCode
Kshitij Sinha, 23/CS/225
Enter Length: 12
Enter Breadth: 9
Area of rectangle = 108
Enter side1 : 8
Enter side2 : 34
Enter side3 : 40
Area of triangle = 97.3191
```

Experiment 13

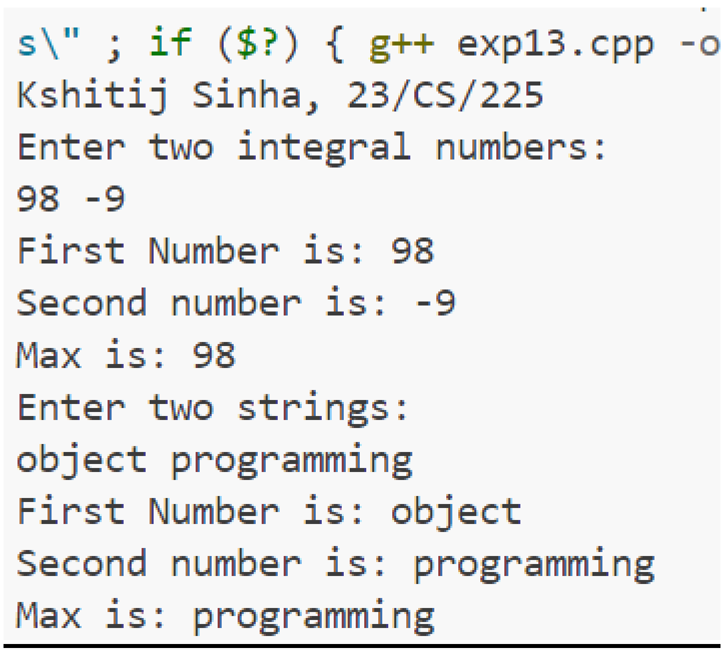
AIM: WAP to explain class template by creating a template T for a class named pair having two data members of type T which are inputted by a constructor and a member function get_max() to return the greatest of two numbers to main. Note: the value of T depends upon the data type specified during object creation.

THEORY: Templates in C++ allow the creation of generic classes and functions that can operate with any data type. This C++ program demonstrates the use of class templates by defining a template class pair that can handle two data members of a generic type T. The class constructor initializes these data members, and the member function get_max() returns the maximum of the two values. This approach allows for flexibility, as the data type of T is determined at the time of object creation, enabling the same code to work with different data types such as int, float. char, string etc.

CODE:

```
#include <bits/stdc++.h>
using namespace std;
template <typename T> class ppair{
    protected:
    T x,y;
    public:
    ppair(T a, T b){
        x=a;
        y=b;
        cout<<"First Number is: "<<a<<endl;
        cout<<"Second number is: "<<b<<endl;
    }

    T get_max(){
        return (x>y?x:y);
    }
};

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    cout<<"Enter two numbers: "<<endl;
    int a,b;
    cin>>a>>b;
    ppair p1(a,b);
    cout<<"Max is: "<<p1.get_max()<<endl;

    cout<<"Enter two strings: "<<endl;
    string c,d;
    cin>>c>>d;
    ppair p2(c,d);
    cout<<"Max is: "<<p2.get_max()<<endl;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp13.cpp -o
Kshitij Sinha, 23/CS/225
Enter two integral numbers:
98 -9
First Number is: 98
Second number is: -9
Max is: 98
Enter two strings:
object programming
First Number is: object
Second number is: programming
Max is: programming
```

Experiment 14

AIM: WAP to accept values from users, find sum, product, difference, division of two numbers, (a) using 3-5 inline functions, (b) using reference variables, (c) using macros.

THEORY: This C++ program demonstrates three different methods for performing arithmetic operations (sum, product, difference, division) on two numbers input by the user. (a) Inline functions are used to define simple operations that can be executed quickly without the overhead of a regular function call.

(b) Reference variables allow direct manipulation of arguments passed to functions, enabling changes to the original variables without copying them.

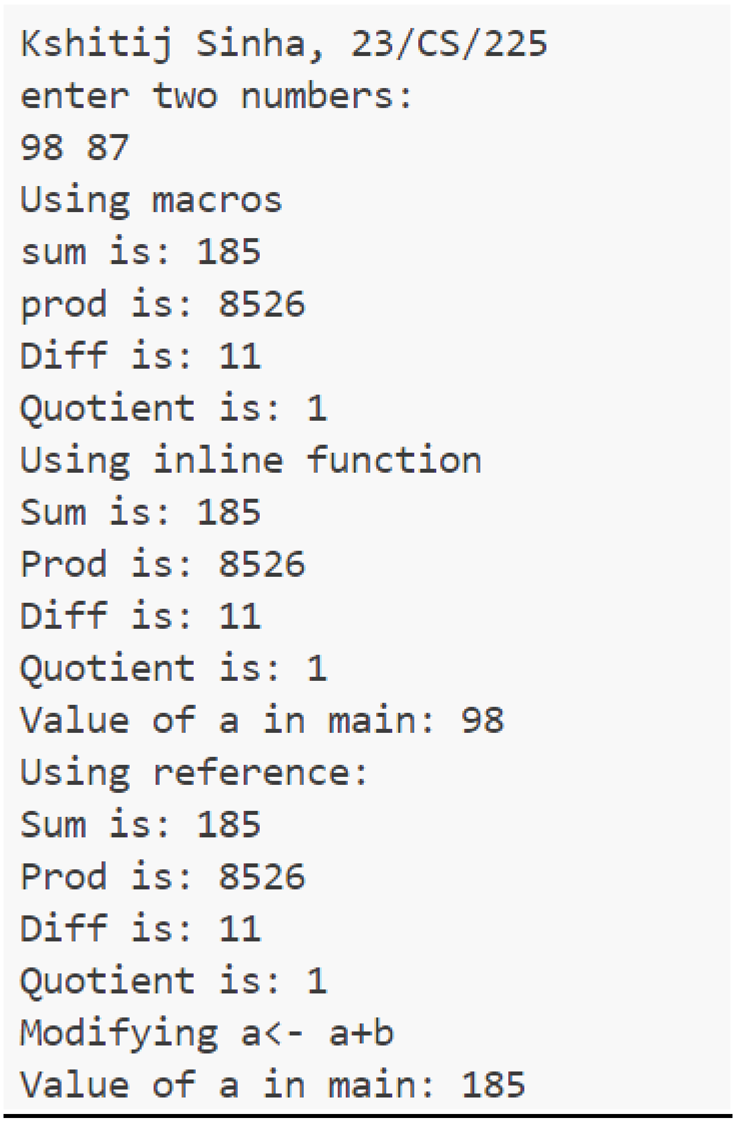
(c) Macros are preprocessor directives that define reusable code snippets, which can improve code readability but may introduce complexity in debugging due to lack of type safety

The sum has also been "assigned" to the first number, demonstrating that reference variable actually change the value of the input variable.

CODE:

```
#include <bits/stdc++.h>
using namespace std;
#define sum(a, b) cout<<"sum is: "<<a+b<<endl
#define prod(a, b) cout<<"prod is: "<<a*b<<endl
#define diff(a, b) cout<<"Diff is: "<<a-b<<endl
#define quotient(a, b) cout<<"Quotient is: "<<a/b<<endl

inline void all_inline(int a, int b){
    cout<<"Sum is: "<<a+b<<endl;
    cout<<"Prod is: "<<a*b<<endl;
    cout<<"Diff is: "<<a-b<<endl;
    cout<<"Quotient is: "<<a/b<<endl;
    a=a+b;
}

void all_reference(int &a, int &b){
    cout<<"Sum is: "<<a+b<<endl;
    cout<<"Prod is: "<<a*b<<endl;
    cout<<"Diff is: "<<a-b<<endl;
    cout<<"Quotient is: "<<a/b<<endl;
    cout<<"Modifying a<- a+b"<<endl;
    a=a+b;
}

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    cout<<"enter two numbers: "<<endl;
    int x,y;
    cin>>x>>y;
    cout<<"Using macros"<<endl;
    sum(x,y);
    prod(x,y);
    diff(x,y);
    quotient(x,y);
    cout<<"Using inline function"<<endl;
    all_inline(x,y);
    cout<<"Value of a in main: "<<x<<endl;
    cout<<"Using reference: "<<endl;
    all_reference(x,y);
    cout<<"Value of a in main: "<<x<<endl;
}
```

OUTPUT:



```
Kshitij Sinha, 23/CS/225
enter two numbers:
98 87
Using macros
sum is: 185
prod is: 8526
Diff is: 11
Quotient is: 1
Using inline function
Sum is: 185
Prod is: 8526
Diff is: 11
Quotient is: 1
Value of a in main: 98
Using reference:
Sum is: 185
Prod is: 8526
Diff is: 11
Quotient is: 1
Modifying a<- a+b
Value of a in main: 185
```

Experiment 15

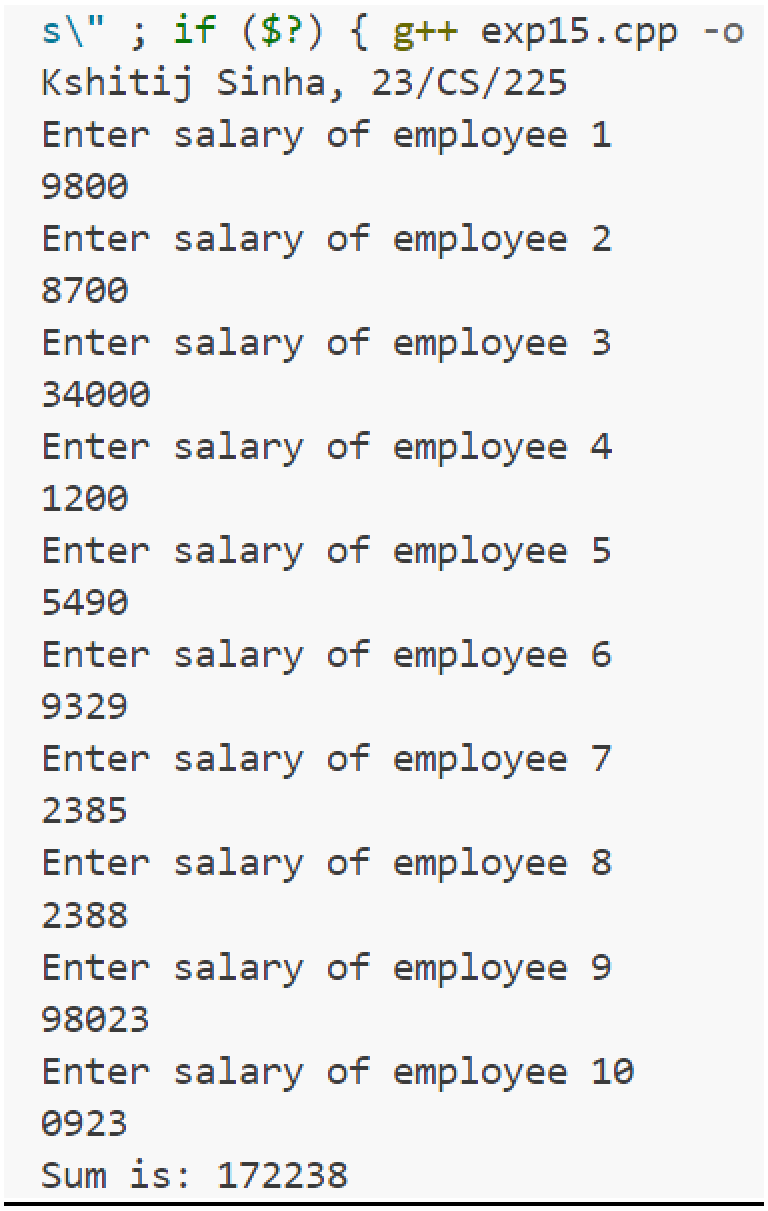
AIM: Write a program in C++ using static variable to get the sum of the salary of 10 employees.

THEORY: : In C++, a static data member is a class member that retains its value between function calls and is shared across all instances of the class. It is initialized only once and exists for the lifetime of the program. A static member function can access static variables but cannot access non-static members directly since it does not operate on a specific object instance. This program uses a static variable to accumulate the salaries of 10 employees, demonstrating how static members can maintain state and perform calculations that apply to all instances of a class.

CODE:

```
#include <bits/stdc++.h>
using namespace std;
class employee{
    static int sum;
    static int index;
    int salary;
    public:
    employee(){
        cout<<"Enter salary of employee "<<index<<endl;
        int x;
        cin>>x;
        sum+=x;
        index+=1;
    }
    int getsum(){
        return sum;
    }
};
int employee::sum=0;
int employee::index=1;

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    employee e[10];
    cout<<"Sum is: "<<e[0].getsum()<<endl;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp15.cpp -o
Kshitij Sinha, 23/CS/225
Enter salary of employee 1
9800
Enter salary of employee 2
8700
Enter salary of employee 3
34000
Enter salary of employee 4
1200
Enter salary of employee 5
5490
Enter salary of employee 6
9329
Enter salary of employee 7
2385
Enter salary of employee 8
2388
Enter salary of employee 9
98023
Enter salary of employee 10
0923
Sum is: 172238
```

Experiment 16

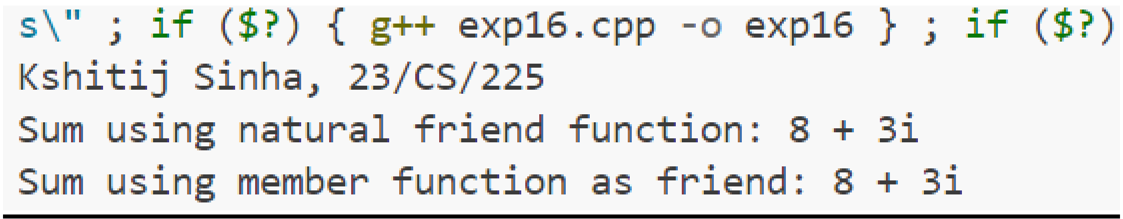
AIM: Write a program using, (a) natural function as friend, (b) member function as friend, to calculate the sum of two complex numbers by using a class complex

THEORY: In this code, in the first approach, a natural function is defined as a friend of the Complex class, allowing it to access the private data members directly to perform the addition. In the second approach, a member function of another class is declared as a friend of the Complex class, enabling it to access the complex number's private members. This program uses friend class and function to do the same.

CODE:

```cpp
#include <bits/stdc++.h>

using namespace std;

class Complex {
private:
    int real;
    int imag;

public:
    Complex(int r, int i) {
        real=r;
        imag=i;
    }

    friend Complex addComplex(const Complex& c1, const Complex& c2);

    Complex addComplexMember(const Complex& c) const{
        return Complex(real + c.real, imag + c.imag);
    }

    void display() const {
        cout << real << " + " << imag << "i" << endl;
    }
};

Complex addComplex(const Complex& c1, const Complex& c2) {
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main() {
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    Complex c1(3, 4);
    Complex c2(5, 6);

    Complex sum1 = addComplex(c1, c2);
    cout << "Sum using natural friend function: ";
    sum1.display();

    Complex sum2 = c1.addComplexMember(c2);
    cout << "Sum using member function as friend: ";
    sum2.display();

    return 0;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp16.cpp -o exp16 } ; if ($?)
Kshitij Sinha, 23/CS/225
Sum using natural friend function: 8 + 3i
Sum using member function as friend: 8 + 3i
```

Experiment 17

AIM: Write a program to find the area of different shapes with one or two arguments -type int or long or float or double or short or unsigned. Write at least 5 overloading functions for 5-6 shapes. Each function should print input, shape, output.

THEORY: In this program, by overloading functions for calculating the area of various shapes (like circles, rectangles, and triangles), we can provide multiple implementations that accept different argument types (e.g., int, float, double). This allows the same function name to be used for different shapes and data types, enhancing code readability and maintainability.

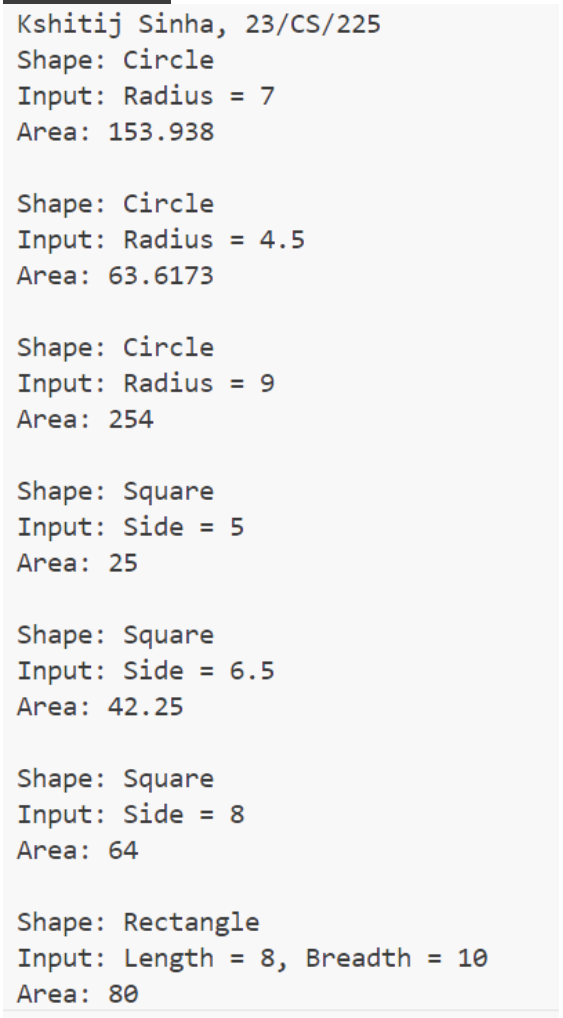
Here we have demonstrated it by calculating the area of shapes like circle, square, rectangle, and rhombus.

CODE:

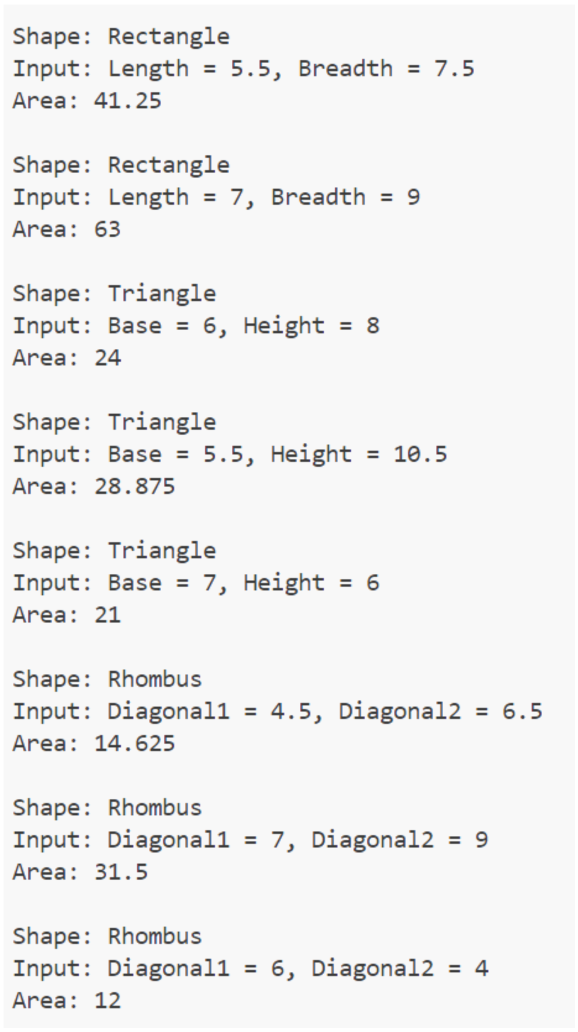
```
#include <bits/stdc++.h>
using namespace std;

void area_c(int radius) {
    double result = M_PI * radius * radius;
    cout << "Shape: Circle\n";
    cout << "Input: Radius = " << radius << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_c(float radius) {
    double result = M_PI * radius * radius;
    cout << "Shape: Circle\n";
    cout << "Input: Radius = " << radius << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_c(long long radius) {
    long long result = M_PI * radius * radius;
    cout << "Shape: Circle\n";
    cout << "Input: Radius = " << radius << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_s(int side) {
    int result = side * side;
    cout << "Shape: Square\n";
    cout << "Input: Side = " << side << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_s(float side) {
    float result = side * side;
    cout << "Shape: Square\n";
    cout << "Input: Side = " << side << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_s(long long side) {
    long long result = side * side;
    cout << "Shape: Square\n";
    cout << "Input: Side = " << side << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_r(int length, int breadth) {
    int result = length * breadth;
    cout << "Shape: Rectangle\n";
    cout << "Input: Length = " << length << ", Breadth = " << breadth << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_r(double length, double breadth) {
    double result = length * breadth;
    cout << "Shape: Rectangle\n";
    cout << "Input: Length = " << length << ", Breadth = " << breadth << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_r(long long length, long long breadth) {
    long long result = length * breadth;
    cout << "Shape: Rectangle\n";
    cout << "Input: Length = " << length << ", Breadth = " << breadth << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_t(int base, int height) {
    double result = 0.5 * base * height;
    cout << "Shape: Triangle\n";
    cout << "Input: Base = " << base << ", Height = " << height << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_t(double base, double height) {
    double result = 0.5 * base * height;
    cout << "Shape: Triangle\n";
    cout << "Input: Base = " << base << ", Height = " << height << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_t(long long base, long long height) {
    long long result = 0.5 * base * height;
    cout << "Shape: Triangle\n";
    cout << "Input: Base = " << base << ", Height = " << height << "\n";
    cout << "Area: " << result << "\n\n";
```

```
}

void area_rh(float diagonal1, float diagonal2) {
    double result = 0.5 * diagonal1 * diagonal2;
    cout << "Shape: Rhombus\n";
    cout << "Input: Diagonal1 = " << diagonal1 << ", Diagonal2 = " << diagonal2 << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_rh(int diagonal1, int diagonal2) {
    double result = 0.5 * diagonal1 * diagonal2;
    cout << "Shape: Rhombus\n";
    cout << "Input: Diagonal1 = " << diagonal1 << ", Diagonal2 = " << diagonal2 << "\n";
    cout << "Area: " << result << "\n\n";
}

void area_rh(long long diagonal1, long long diagonal2) {
    long long result = 0.5 * diagonal1 * diagonal2;
    cout << "Shape: Rhombus\n";
    cout << "Input: Diagonal1 = " << diagonal1 << ", Diagonal2 = " << diagonal2 << "\n";
    cout << "Area: " << result << "\n\n";
}

int main() {
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    area_c(7);
    area_c(4.5f);
    area_c(9LL);
    area_s(5);
    area_s(6.5f);
    area_s(8LL);
    area_r(8, 10);
    area_r(5.5, 7.5);
    area_r(7LL, 9LL);
    area_t(6, 8);
    area_t(5.5, 10.5);
    area_t(7LL, 6LL);
    area_rh(4.5f, 6.5f);
    area_rh(7, 9);
    area_rh(6LL, 4LL);

    return 0;
}
```

OUTPUT:



```
Kshitij Sinha, 23/CS/225
Shape: Circle
Input: Radius = 7
Area: 153.938

Shape: Circle
Input: Radius = 4.5
Area: 63.6173

Shape: Circle
Input: Radius = 9
Area: 254

Shape: Square
Input: Side = 5
Area: 25

Shape: Square
Input: Side = 6.5
Area: 42.25

Shape: Square
Input: Side = 8
Area: 64

Shape: Rectangle
Input: Length = 8, Breadth = 10
Area: 80
```



```
Shape: Rectangle
Input: Length = 5.5, Breadth = 7.5
Area: 41.25

Shape: Rectangle
Input: Length = 7, Breadth = 9
Area: 63

Shape: Triangle
Input: Base = 6, Height = 8
Area: 24

Shape: Triangle
Input: Base = 5.5, Height = 10.5
Area: 28.875

Shape: Triangle
Input: Base = 7, Height = 6
Area: 21

Shape: Rhombus
Input: Diagonal1 = 4.5, Diagonal2 = 6.5
Area: 14.625

Shape: Rhombus
Input: Diagonal1 = 7, Diagonal2 = 9
Area: 31.5

Shape: Rhombus
Input: Diagonal1 = 6, Diagonal2 = 4
Area: 12
```

Experiment 18

AIM:

(A)(Hierarchical Inheritance) Write a program to calculate the salary of faculty, staff, using inheritance of class employee with constructors.

(B) (Multiple Inheritance) Write a program to calculate the salary of Faculty using inheritance of class Employee and class Salary with constructors.

THEORY:

Hierarchical Inheritance involves a single base class being inherited by multiple derived classes. This structure allows for a clear organizational hierarchy and promotes code reuse, as the base class can provide common attributes and methods that all derived classes can utilize.

Multiple Inheritance occurs when a derived class inherits from more than one base class. This allows the derived class to access features from multiple parent classes, enhancing flexibility and functionality.

CODE:

A)

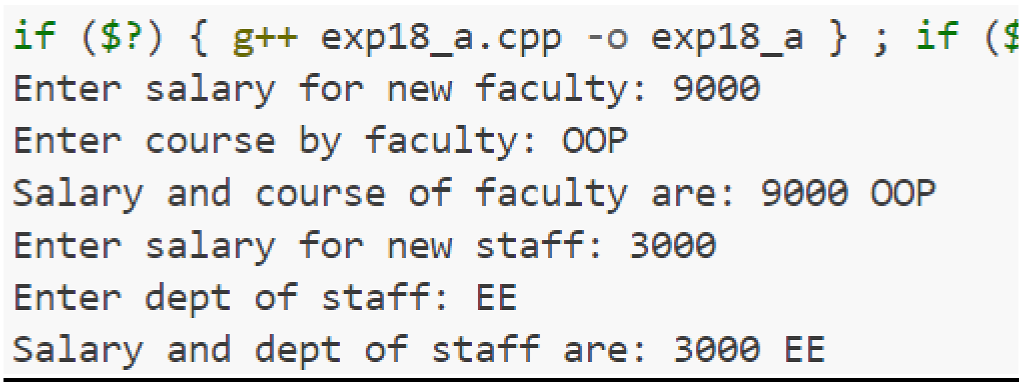
```
#include <bits/stdc++.h>
using namespace std;
class employee{
    protected:
    int salary;
};

class faculty: public employee{
    protected:
    string course;
    public:
    faculty(){
        cout<<"Enter salary for new faculty: ";
        int x;
        cin>>x;
        cout<<"Enter course by faculty: ";
        string s;
        cin>>s;
        salary=x;
        course=s;
    }

    void show(){
        cout<<"Salary and course of faculty are: ";
        cout<<salary<<" "<<course<<endl;
    }
};

class staff: public employee{
    protected:
    string dept;
    public:
    staff(){
        cout<<"Enter salary for new staff: ";
        int x;
        cin>>x;
        cout<<"Enter dept of staff: ";
        string s;
        cin>>s;
        salary=x;
        dept=s;
    }

    void show(){
        cout<<"Salary and dept of staff are: ";
        cout<<salary<<" "<<dept<<endl;
    }
};

int main(){
    faculty f1;
    f1.show();
    staff s1;
    s1.show();
}
```

B)

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


class EMPLOYEE {
protected:
    int empID;
    string empName;
    string department;

public:

    EMPLOYEE(int id, string name, string dept) : empID(id), empName(name),
department(dept) {
        cout << "Employee Constructor Called\n";
    }

    void displayEmployeeDetails() const {
        cout << "Employee ID: " << empID << "\n";
        cout << "Employee Name: " << empName << "\n";
        cout << "Department: " << department << "\n";
    }
};


class SALARY : public EMPLOYEE {
private:
    double basicPay;
    double allowances;
    double deductions;

public:

    SALARY(int id, string name, string dept, double basic, double allowance, double
deduction)
        : EMPLOYEE(id, name, dept), basicPay(basic), allowances(allowance),
deductions(deduction) {
        cout << "Salary Constructor Called\n";
    }

    double calculateSalary() const {
        return basicPay + allowances - deductions;
    }

    void displaySalaryDetails() const {
        displayEmployeeDetails();
        cout << "Basic Pay: " << basicPay << "\n";
        cout << "Allowances: " << allowances << "\n";
        cout << "Deductions: " << deductions << "\n";
        cout << "Total Salary: " << calculateSalary() << "\n";
    }
};

int main() {
    SALARY faculty(101, "Kshitij", "Computer Science", 250000, 40000, 50000);

    cout << "\nFaculty Salary Details:\n";
    faculty.displaySalaryDetails();

    return 0;
}
```

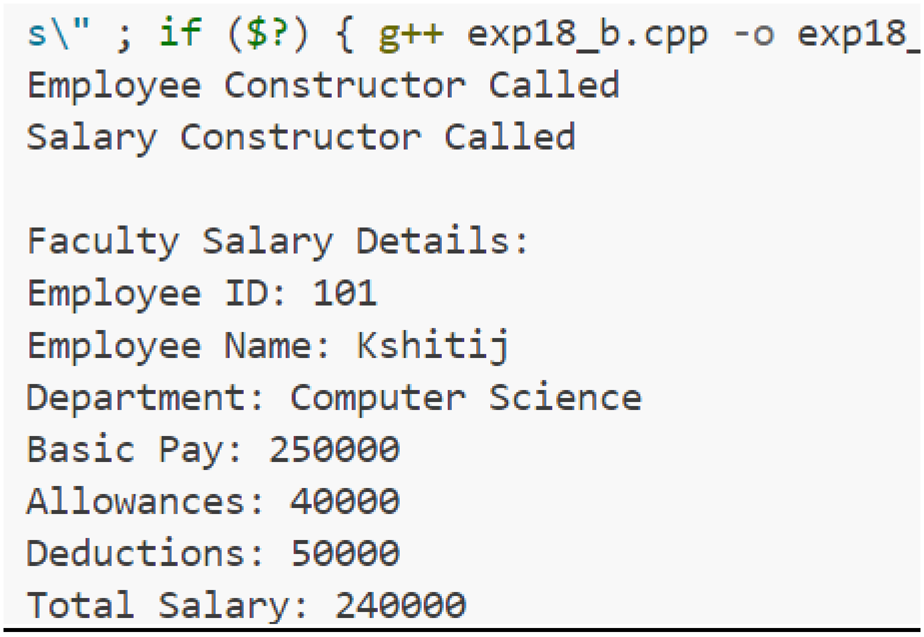
OUTPUT:

A)



```
if ($?) { g++ exp18_a.cpp -o exp18_a } ; if ($
Enter salary for new faculty: 9000
Enter course by faculty: OOP
Salary and course of faculty are: 9000 OOP
Enter salary for new staff: 3000
Enter dept of staff: EE
Salary and dept of staff are: 3000 EE
```

B)



```
s\" ; if ($?) { g++ exp18_b.cpp -o exp18_
Employee Constructor Called
Salary Constructor Called

Faculty Salary Details:
Employee ID: 101
Employee Name: Kshitij
Department: Computer Science
Basic Pay: 250000
Allowances: 40000
Deductions: 50000
Total Salary: 240000
```

Experiment 19

AIM: Write a program to find square and cube of a number using (write functions for getData, showData, showDesult) late/dynamic binding using base pointer.

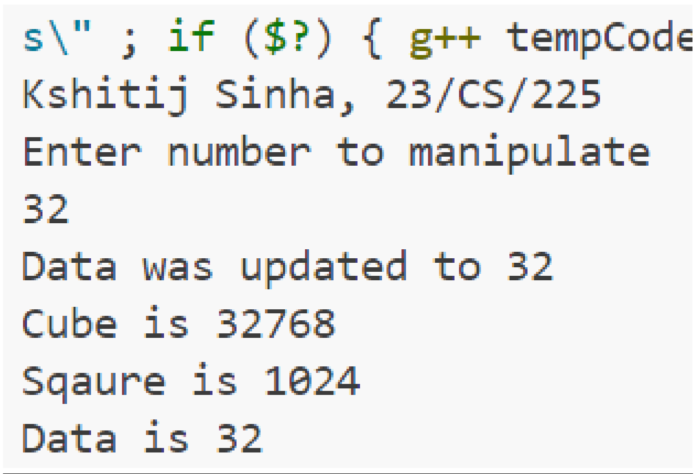
THEORY: Dynamic binding in C++ refers to the mechanism of resolving function calls at runtime based on the type of the object being pointed to, rather than the type of the pointer itself. This enables polymorphism, allowing derived classes to provide specific implementations of functions declared in a base class.

By using virtual functions, a base class pointer can invoke the appropriate derived class function, facilitating flexible and extensible code design. A pure abstract class in C++ is a class that contains at least one pure virtual function, making it impossible to instantiate and primarily serving as an interface for derived classes

CODE:

```
#include <bits/stdc++.h>
using namespace std;
class number{
    int data;
    public:
    void getdata(int x){
        data=x;
        cout<<"Data was updated to "<<data<<endl;
    }
    void showdata(){
        cout<<"Data is "<<data<<endl;
    }
    void cube(){
        cout<<"Cube is "<<data*data*data<<endl;
    }
    void square(){
        cout<<"Sqaure is "<<data*data<<endl;
    }
};

int main(){
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    cout<<"Enter number to manipulate "<<endl;
    int x;
    cin>>x;
    number* num=new number;
    num->getdata(x);
    num->cube();
    num->square();
    num->showdata();
}
```

OUTPUT:



```
s\" ; if ($?) { g++ tempCode
Kshitij Sinha, 23/CS/225
Enter number to manipulate
32
Data was updated to 32
Cube is 32768
Sqaure is 1024
Data is 32
```

Experiment 20

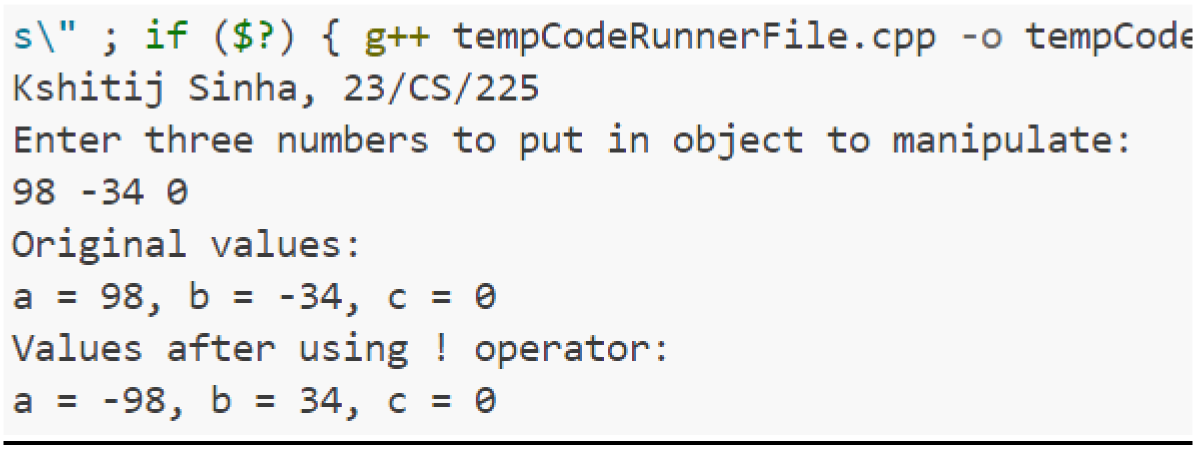
AIM: Write a program to overload ! (not) operator (unary) all data members of a class.

THEORY: By overloading operators, we can make objects of a class work intuitively with operators such as +, -, and even logical operators. This enhances code readability and usability, enabling operators to interact with class instances as if they were built-in types. In this program, the logical NOT operator ! is overloaded to negate all the integer data members of the class.

CODE:

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

class testt {
private:
    int a,b,c;

public:
    testt(int x, int y, int z) : a(x), b(y), c(z) {}
    void operator!() {
        a = -a;
        b = -b;
        c = -c;
    }

    void display() const {
        cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
    }
};

int main() {
    cout<<"Kshitij Sinha, 23/CS/225"<<endl;
    int a,b,c;
    cout<<"Enter three numbers to put in object to manipulate: "<<endl;
    cin>>a>>b>>c;
    testt obj(a, b, c);

    cout << "Original values:\n";
    obj.display();

    !obj;

    cout << "Values after using ! operator:\n";
    obj.display();

    return 0;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCode
Kshitij Sinha, 23/CS/225
Enter three numbers to put in object to manipulate:
98 -34 0
Original values:
a = 98, b = -34, c = 0
Values after using ! operator:
a = -98, b = 34, c = 0
```

Experiment 21

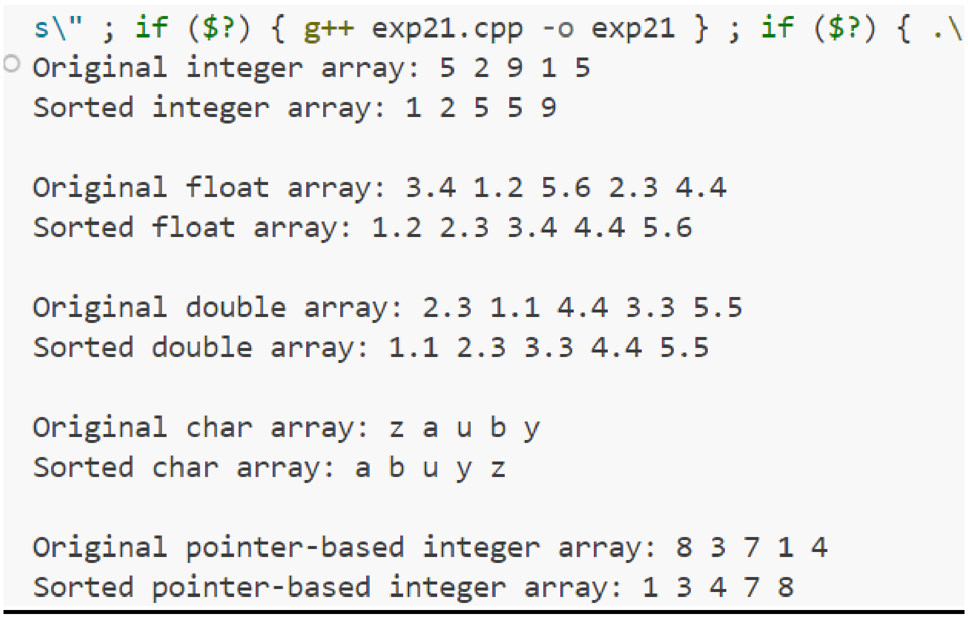
AIM: Use Template Function to sort an array call function with 5 different combinations including pointer, float, int, etc.

THEORY: : Template functions in C++ allow a single function to operate on different data types, promoting code reusability and flexibility. By defining a template, the same function can be used to handle various types. This is especially useful for generic operations like sorting. When a template function is called with a specific data type, the compiler automatically generates the appropriate version of the function for that type. In this code, we have sorted an array of integers, floats, doubles, and chars as well

CODE:

```
#include <iostream>
using namespace std;
template <typename T>
void sortt(T arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                T temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
template <typename T>
void print(T arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int intArr[] = {5, 2, 9, 1, 5};
    int intSize = sizeof(intArr) / sizeof(intArr[0]);
    cout << "Original integer array: ";
    print(intArr, intSize);
    sortt(intArr, intSize);
    cout << "Sorted integer array: ";
    print(intArr, intSize);


    float floatArr[] = {3.4f, 1.2f, 5.6f, 2.3f, 4.4f};
    int floatSize = sizeof(floatArr) / sizeof(floatArr[0]);
    cout << "\nOriginal float array: ";
    print(floatArr, floatSize);
    sortt(floatArr, floatSize);
    cout << "Sorted float array: ";
    print(floatArr, floatSize);


    double doubleArr[] = {2.3, 1.1, 4.4, 3.3, 5.5};
    int doubleSize = sizeof(doubleArr) / sizeof(doubleArr[0]);
    cout << "\nOriginal double array: ";
    print(doubleArr, doubleSize);
    sortt(doubleArr, doubleSize);
    cout << "Sorted double array: ";
    print(doubleArr, doubleSize);


    char charArr[] = {'z', 'a', 'u', 'b', 'y'};
    int charSize = sizeof(charArr) / sizeof(charArr[0]);
    cout << "\nOriginal char array: ";
    print(charArr, charSize);
    sortt(charArr, charSize);
    cout << "Sorted char array: ";
    print(charArr, charSize);

    int* ptrArr = new int[5]{8, 3, 7, 1, 4};
    int ptrSize = 5;
    cout << "\nOriginal pointer-based integer array: ";
    print(ptrArr, ptrSize);
    sortt(ptrArr, ptrSize);
    cout << "Sorted pointer-based integer array: ";
    print(ptrArr, ptrSize);

    delete[] ptrArr;
    return 0;
}
```

OUTPUT:



```
s\" ; if ($?) { g++ exp21.cpp -o exp21 } ; if ($?) { .\
Original integer array: 5 2 9 1 5
Sorted integer array: 1 2 5 5 9

Original float array: 3.4 1.2 5.6 2.3 4.4
Sorted float array: 1.2 2.3 3.4 4.4 5.6

Original double array: 2.3 1.1 4.4 3.3 5.5
Sorted double array: 1.1 2.3 3.3 4.4 5.5

Original char array: z a u b y
Sorted char array: a b u y z

Original pointer-based integer array: 8 3 7 1 4
Sorted pointer-based integer array: 1 3 4 7 8
```