**VYSYA COLLEGE, SALEM-103**

**CLASS: I BCA** **PROGRAMS – 6 TO 10**

**SUBJECT: PRACTICAL: C++ PROGRAMMING** **SUBJECT CODE: 22UCAP02**

## EX. NO: 6 - TO DEMONSTRATE CONSTRUCTOR AND DESTRUCTOR

**AIM**:

To write a C++ program to demonstrate constructor and destructor using class.

**PROCEDURE**:

**Step 1 :** Start the program.

**Step 2 :** Include Header File.

**Step 3 :** Define Class **MyClass** with a constructor and a destructor.

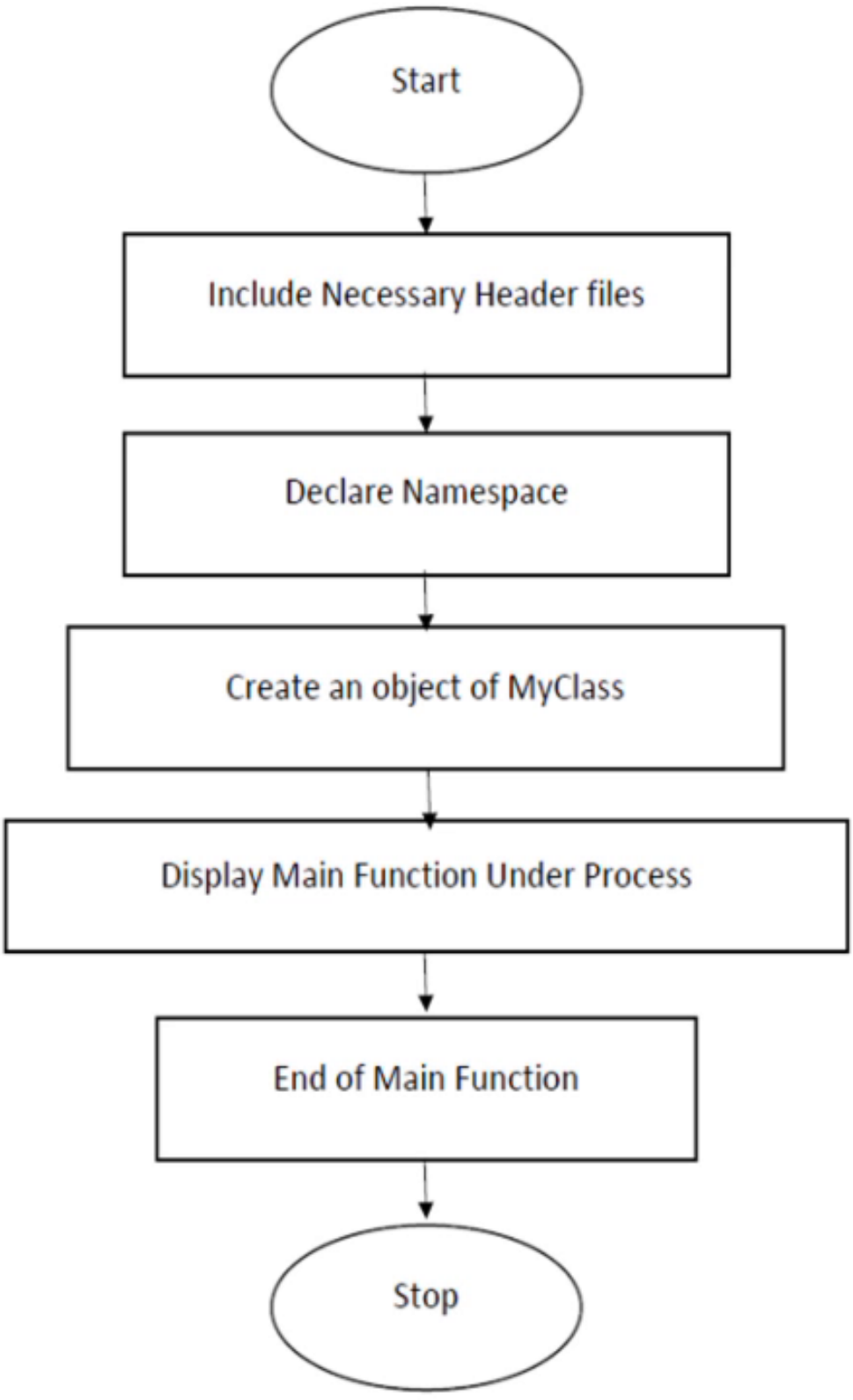
**Step 4 :** define main function, an object **myObject** of type **MyClass** is created.

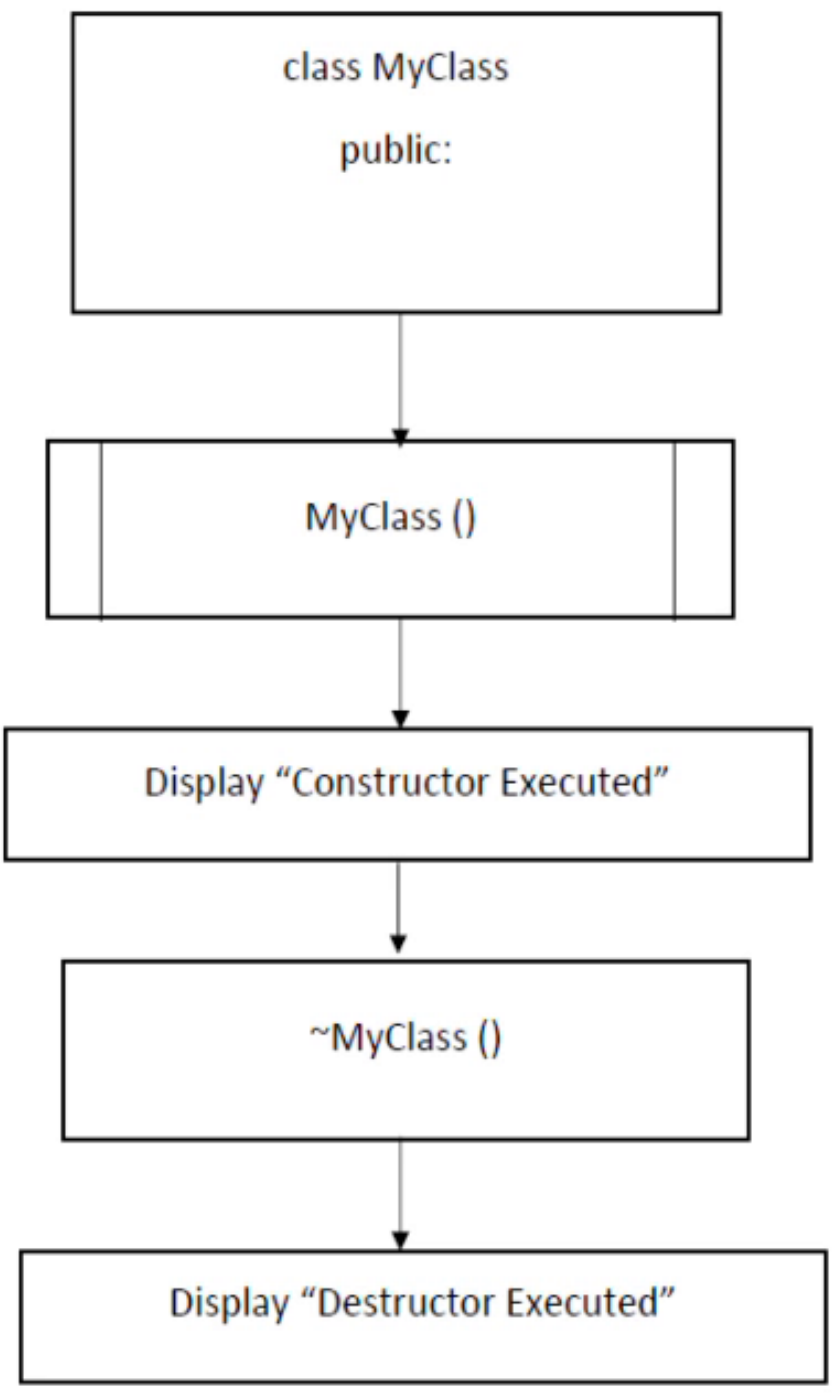
**Step 5 :** The **Constructor** is executed, printing "Constructor Executed."

**Step 6:** The main Function automatically call **Destructor** to Execute, when the program **myObject** goes out of scope.

**Step 7 :** Stop the program

**FLOWCHART**


Start
Include Necessary Header files
Declare Namespace
Create an object of MyClass
Display Main Function Under Process
End of Main Function
Stop


class MyClass
public:
MyClass ()
Display “Constructor Executed”
~MyClass ()
Display “Destructor Executed”

**SOURCE CODE:**

```
#include<iostream>
using namespace std;
class MyClass
{
        public:
        // Constructor
                MyClass()
                {
                        cout<<"Constructor Executed"<<endl;
                }
         // Destructor
                ~MyClass()
                {
                        cout<<"Destructor Executed"<<endl;
                }
};

int main()
{
   // Creating an object of MyClass
MyClass myObject;
   // The object goes out of scope at the end of the block
   // Destructor will be called automatically
cout<<"Main Function under Process"<<endl;

return 0;
}
```

**OUTPUT:**

Constructor Executed

Main Function under Process

Destructor Executed

**RESULT:**

Thus, the demonstrate constructor and destructor using class was executed successfully.

## EX. NO: 7 – TO DEMONSTRATE UNARY OPERATOR OVERLOADING

**AIM**: To write a C++ program to demonstrate unary operator overloading.

**PROCEDURE**:

**Step 1 :** Start the program.

**Step 2 :** Include Header File.

**Step 3 :** Define Class Number with a private data member value.

**Step 4 :** Define main Function .

**Step 5 :** Create an object num1 of type Number with an initial value of 5.

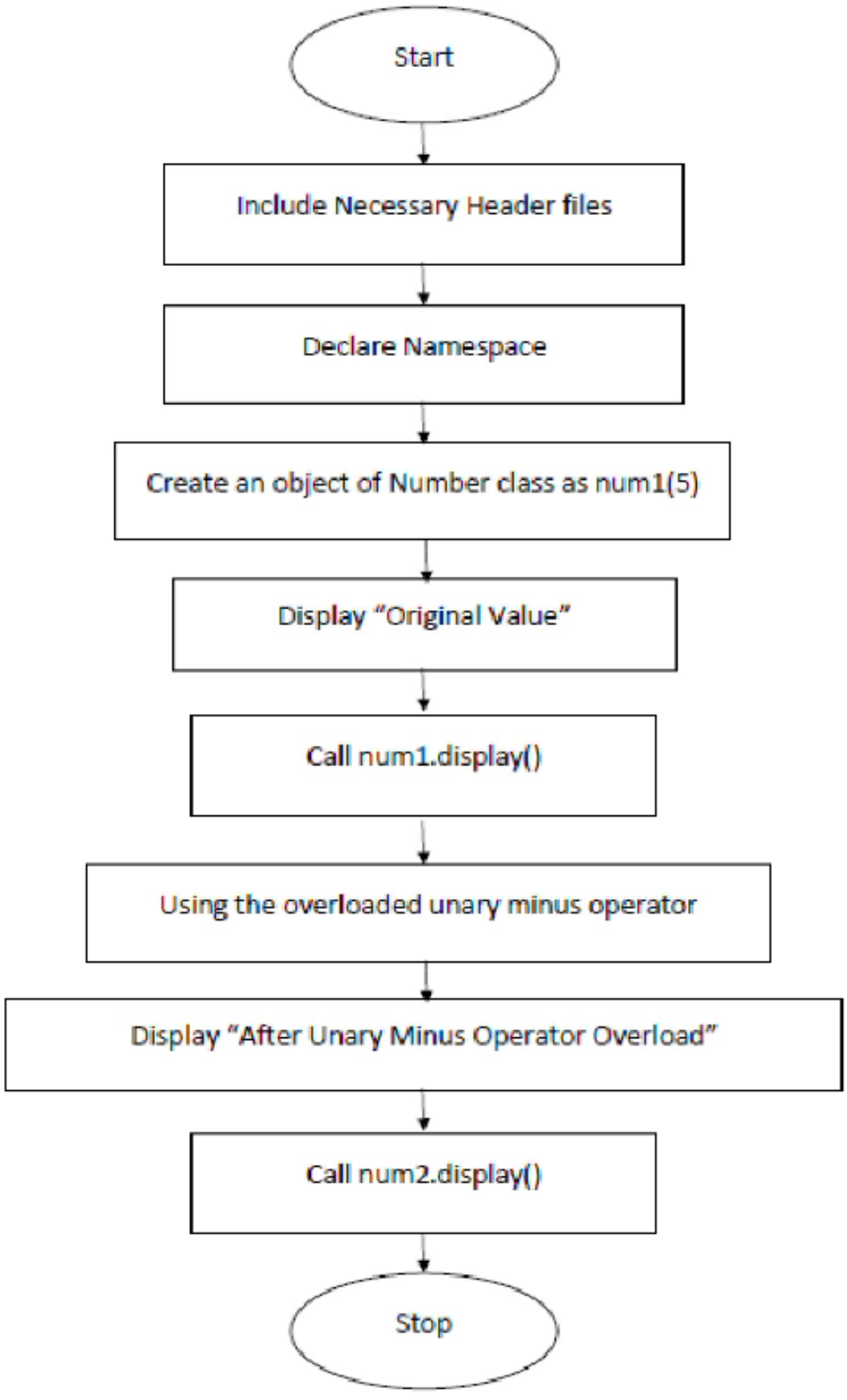
**Step 6 :** Execute the program.

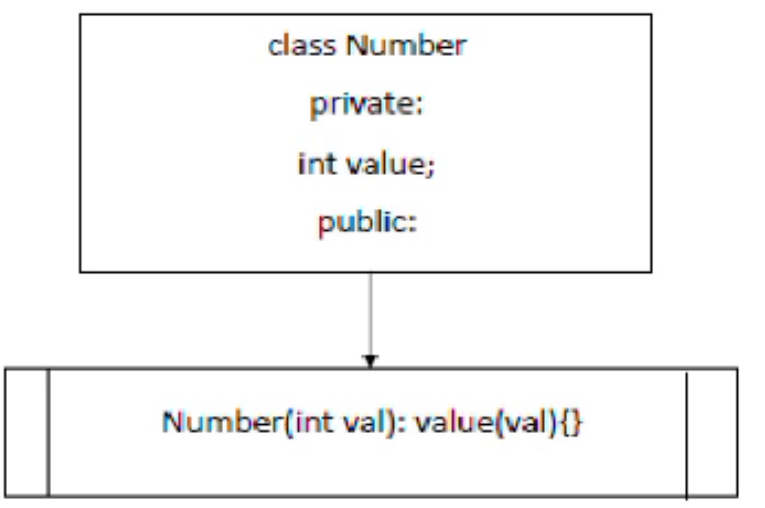
**Step 7 :** The original value of num1 is displayed using the display function.

**Step 8 :** The unary minus operator is overloaded, and a new Number object num2 is created by negating num1.

**Step 9 :** Stop the process.

**FLOWCHART**


Start
Include Necessary Header files
Declare Namespace
Create an object of Number class as num1(5)
Display "Original Value"
Call num1.display()
Using the overloaded unary minus operator
Display "After Unary Minus Operator Overload"
Call num2.display()
Stop




**SOURCE CODE:**

```
#include<iostream>
using namespace std;
class Number
{
private:
int value;
```

```
public:
   // Constructor
Number(int val) : value(val)
 {
  }

   // Overloading unary minus (-) operator
   Number operator-()
  {
        Number result(-value);
        return result;
   }

   // Function to display the value
void display()
{
        cout<< value <<endl;
   }
};

int main()
{
   // Creating an object of Number class
   Number num1(5);
   cout<<"Original value:" ;
   num1.display();

      // Using the overloaded unary minus operator
      Number num2 = -num1;

   cout<<"After Unary Minus Operator Overload:" ;
   num2.display();
   return 0;
   }
```

**OUTPUT:**

Original Value : 5
After Unary Minus Operator Overload: -5

**RESULT:**

Thus, the demonstration of unary operator overloading program was executed successfully.

## EX. NO: 8 - TO DEMONSTRATE BINARY OPERATOR OVERLOADING

**AIM**: To write a C++ program to demonstrate binary operator overloading.

**PROCEDURE**:

**Step 1 :** Start the program

**Step 2 :** Include Header File:

**Step 3 :** Define Class Complex to represent complex numbers

**Step 4 :** The constructor initializes the real and imaginary parts of the complex number.

**Step 5 :** The class overloads the binary addition (+) operator, allowing the addition of two complex numbers.

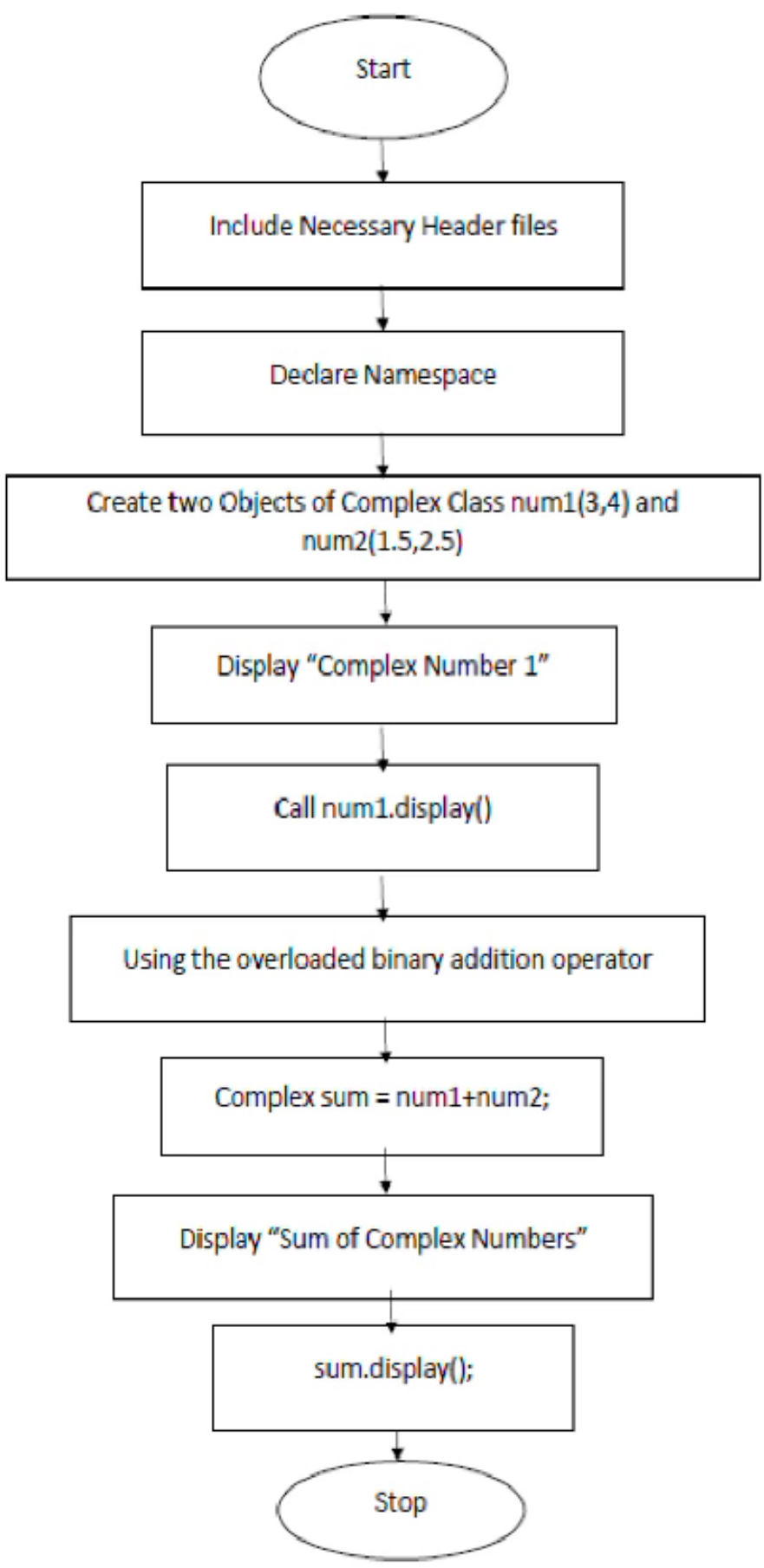
**Step 6 :** Define main Function with two objects num1 and num2 of type Complex are created with different real and imaginary parts.

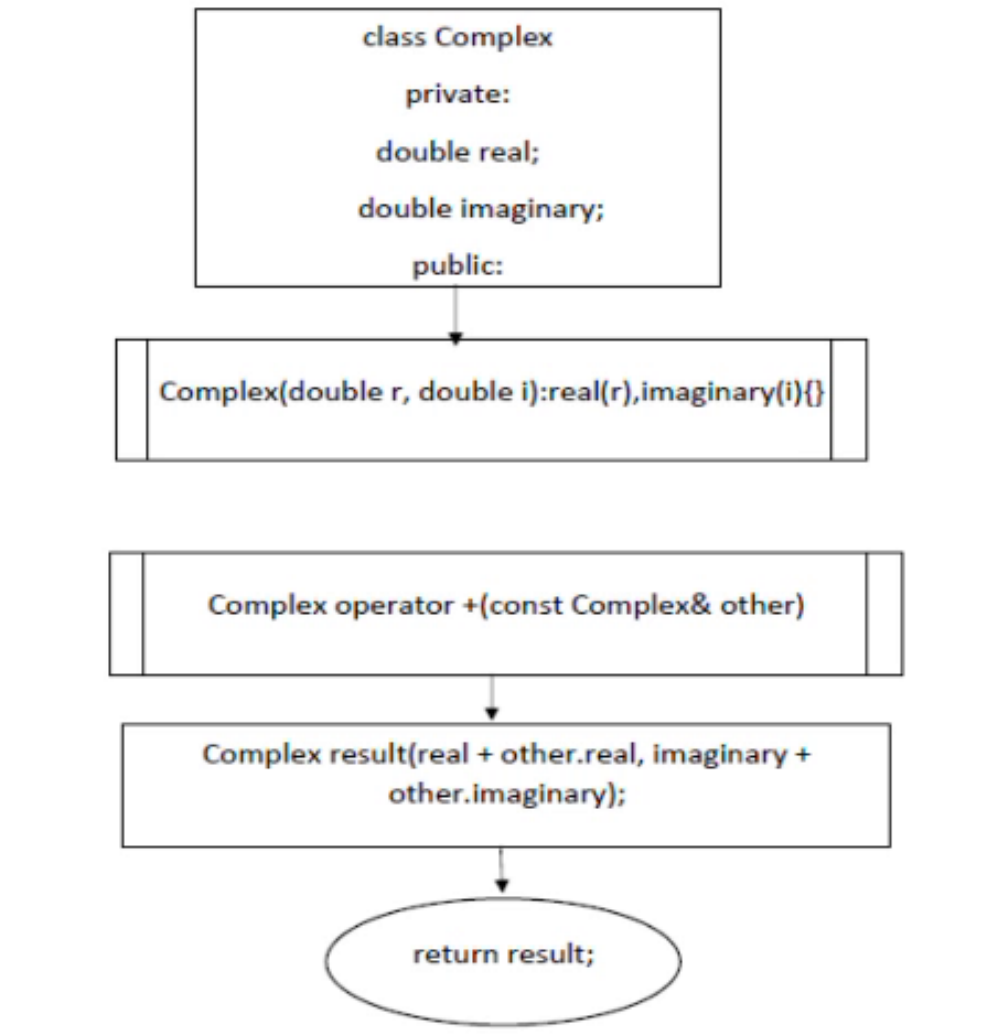
**Step 7 :** The original values of both complex numbers are displayed using the display function.

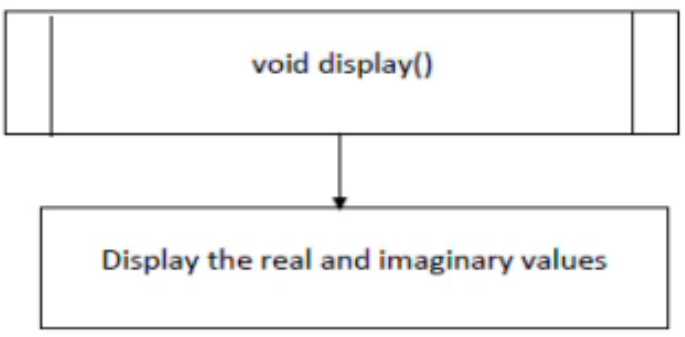
**Step 8 :** The binary addition operator is overloaded, and a new Complex object sum is created by adding num1 and num2.

**Step 9 :** Stop the program

**FLOWCHART**


Start
Include Necessary Header files
Declare Namespace
Create two Objects of Complex Class num1(3,4) and num2(1.5,2.5)
Display "Complex Number 1"
Call num1.display()
Using the overloaded binary addition operator
Complex sum = num1+num2;
Display "Sum of Complex Numbers"
sum.display();
Stop




**SOURCE CODE:**

```
#include <iostream>
using namespace std;
class Complex
{
private:
double real;
double imaginary;
public:
   // Constructor
Complex(double r, double i) : real(r), imaginary(i) {}
   // Overloading binary addition (+) operator
   Complex operator + (const Complex& other)
```

```
{
      Complex result(real + other.real, imaginary + other.imaginary);
return result;
   }
   // Function to display the complex number
void display()
{
cout<<"Real: "<< real <<" + Imaginary: "<< imaginary <<"i"<<std::endl;
   }
};

int main()
{
   // Creating two objects of Complex class
   Complex num1(3, 4);
   Complex num2(1.5, 2.5);
cout<<"Complex Number 1:"<<endl;
num1.display();

cout<<"Complex Number 2:"<<endl;
num2.display();

// Using the overloaded binary addition operator
 Complex sum = num1 + num2;
 cout<<"Sum of Complex Numbers:"<<endl;
 sum.display();
 return 0;
 }
```

**OUTPUT:**

**Complex Number 1:**
Real: 3 + Imaginary: 4i

**Complex Number 2:**
Real: 1.5 + Imaginary: 2.5i

**Sum of Complex Numbers:**
Real: 4.5 + Imaginary: 6.5i

**RESULT:**

Thus, the demonstration of binary operator overloading program was executed successfully.

## EX. NO: 9 - TO DEMONSTRATE DIFFERENT TYPES OF INHERITANCE

### 1) SINGLE INHERITANCE:

**AIM:** To write a C++ program to demonstrate Single Inheritance

**PROCEDURE:**

**Step 1:** Start the process

**Step 2:** Create a class "base" and assign the value for the data member x using the getdata() member function.

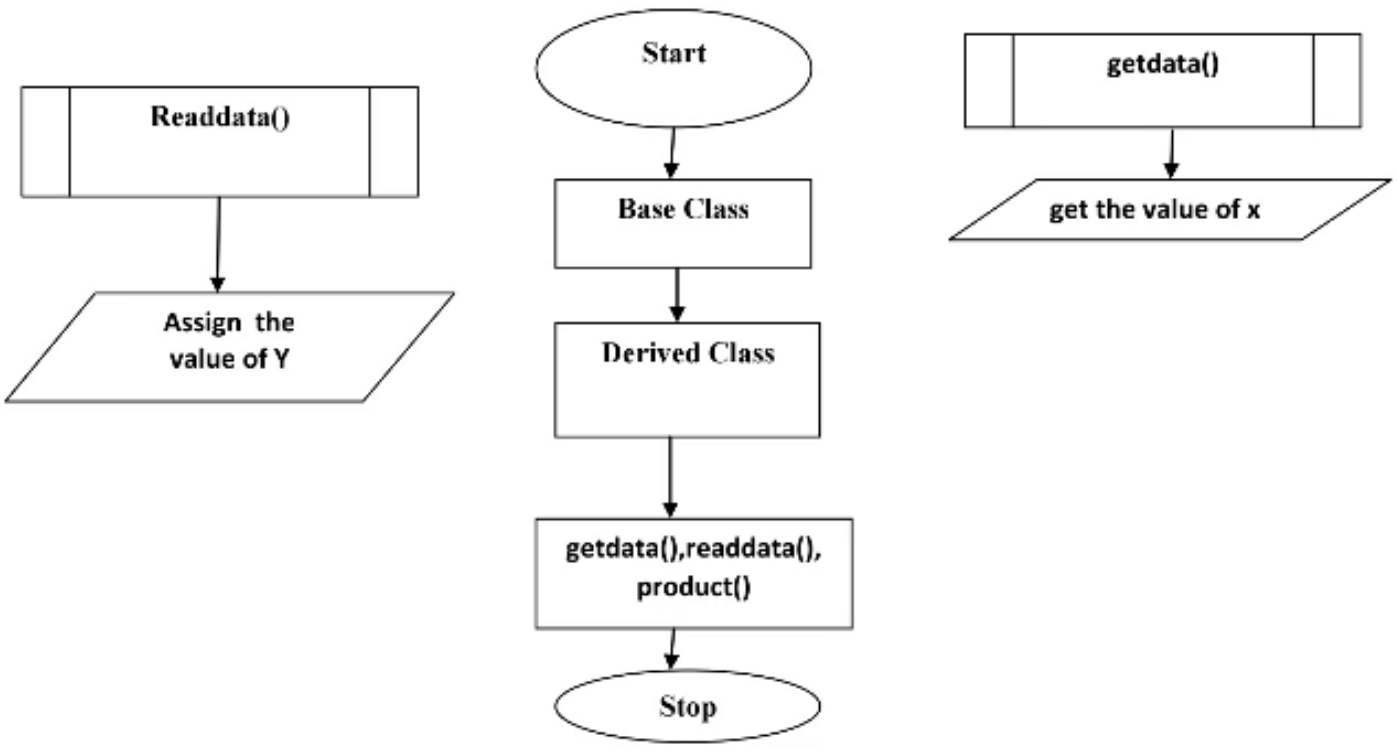
**Step 3:** Create a class "derived" which inherits the class "base" in public mode and initialize the value for y using the readdata().

**Step 4:** Multiply x and y in product() member function.

**Step 5:** In the main function create a object "a" for derived class and then call the member functions.

**Step 6:** Stop the process.

**FLOWCHART:**



**SOURCE CODE;**

```
#include<iostream>
using namespace std;
class base
{
public:
int x;
void getdata()
{
cout<<"Enter the value of x=";
cin>>x;
}
};
class derive:public base
{
private:
int y;
public:
void readdata()
{
cout<<"Enter the value of y= ";
cin>>y;
}
void product()
{
cout<<"Product= "<<x*y;
}
};
int main()
{
derive a;
a.getdata();
a.readdata();
a.product();
return 0;
}
```

**OUTPUT:**

Enter the value of x = 5
Enter the value of y = 5
Product = 25

**RESULT:**

Thus, the demonstration of single inheritance was executed successfully

## 2) MULTIPLE INHERITANCE:

**AIM:**

To write a C++ program to demonstrate Multiple Inheritance

**PROCEDURE:**

**Step 1:** Start the process

**Step 2:** Create a class "M", use the protected mode to assign the value for the data member m using get_m(int) member function.

**Step 3:** Create the class "N", use the protected mode to assign the value for the data member n using get_n(int) member function

**Step 4:** Create a class "P", which inherits the class "M". class "N". Here the display(void) function is created, class "M" use the scope resolution operator for the get_m(int x), x is given as argument.

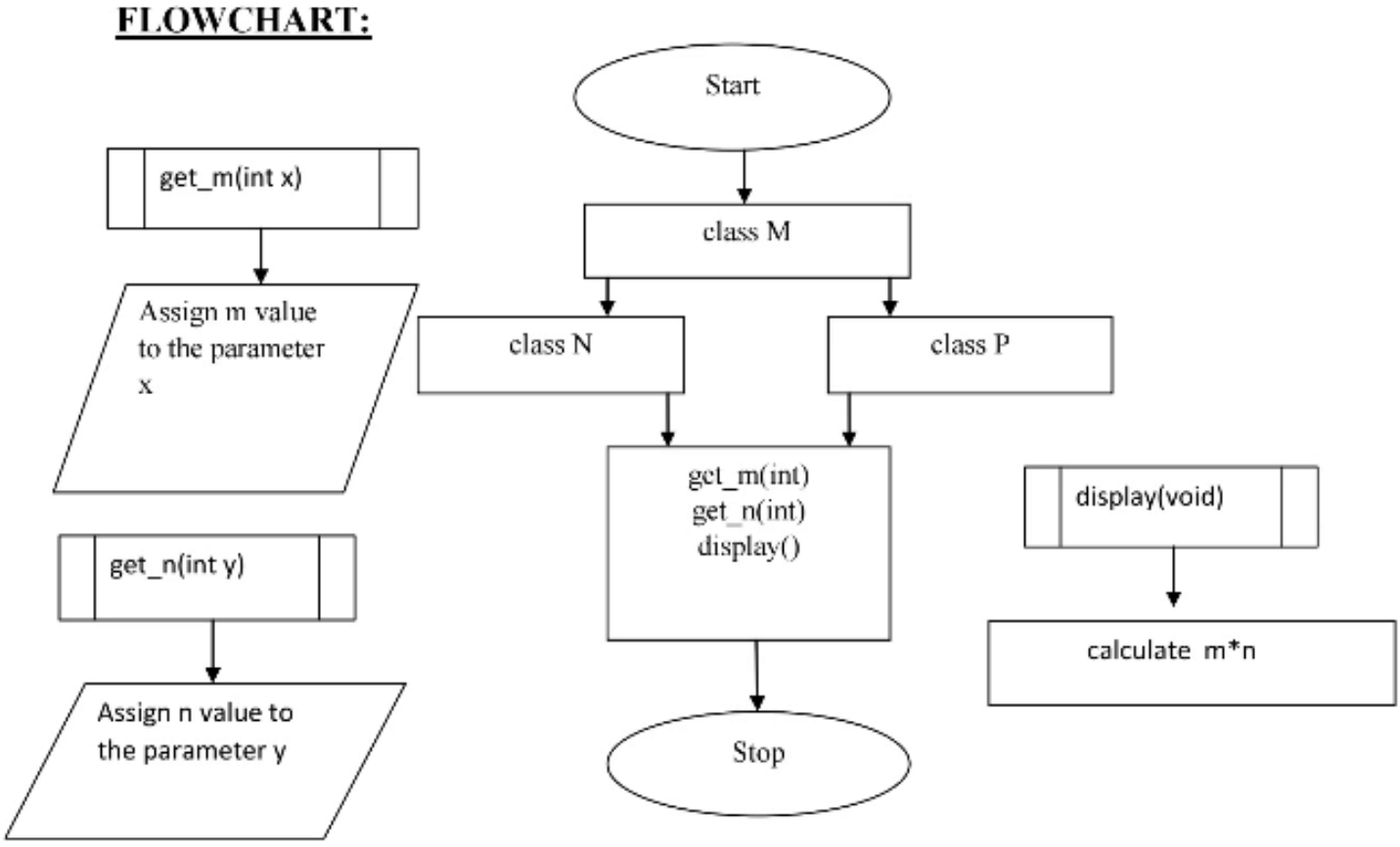
**Step 5:** And a class "N", use the scope resolution operator for the get_n(int y) member function ,y is given as argument.

**Step 6:** In the class "P" use the scope resolution operator for the display(void) member function and calculate the m*n value.

**Step 7:** In the main function create the object "p" for the class "P" to execute the get_m(),get_n(),display() member function.

**Step 8:** Stop the process.

**FLOWCHART:**




**SOURCE CODE :**

```
#include<iostream>
using namespace std;
class M
{
protected:
int m;
public:
void get_m(int);
};
class N
{
protected:
int n;
public:
void get_n(int);
};
```

```
class P:public M,public N
{
public:
void display(void);
};

void M::get_m(int x)
{
m=x;
}


void N::get_n(int y)
{
n=y;
}

void P::display(void)
{
cout<<"m= "<<m<<"\n";
cout<<"n= "<<n<<"\n";
cout<<"m*n= "<<m*n<<"\n";
}

int main()
{
P p;
p.get_m(10);
p.get_n(20);
p.display();
return 0;
}
```

**OUTPUT:**

```
M     = 10
N     = 20
m*n   = 200
```

**RESULT:**

Thus the demonstration of the multiple inheritances was successfully executed

**3) MULTI - LEVEL INHERITANCE:**

**AIM:** To write a C++ program to demonstrate multilevel inheritance

**PROCEDURE:**

**Step 1:** Start the process

**Step 2:** Create a class "student"and use protected mode to assign the roll_number variable using get_number(), put_number()member function .

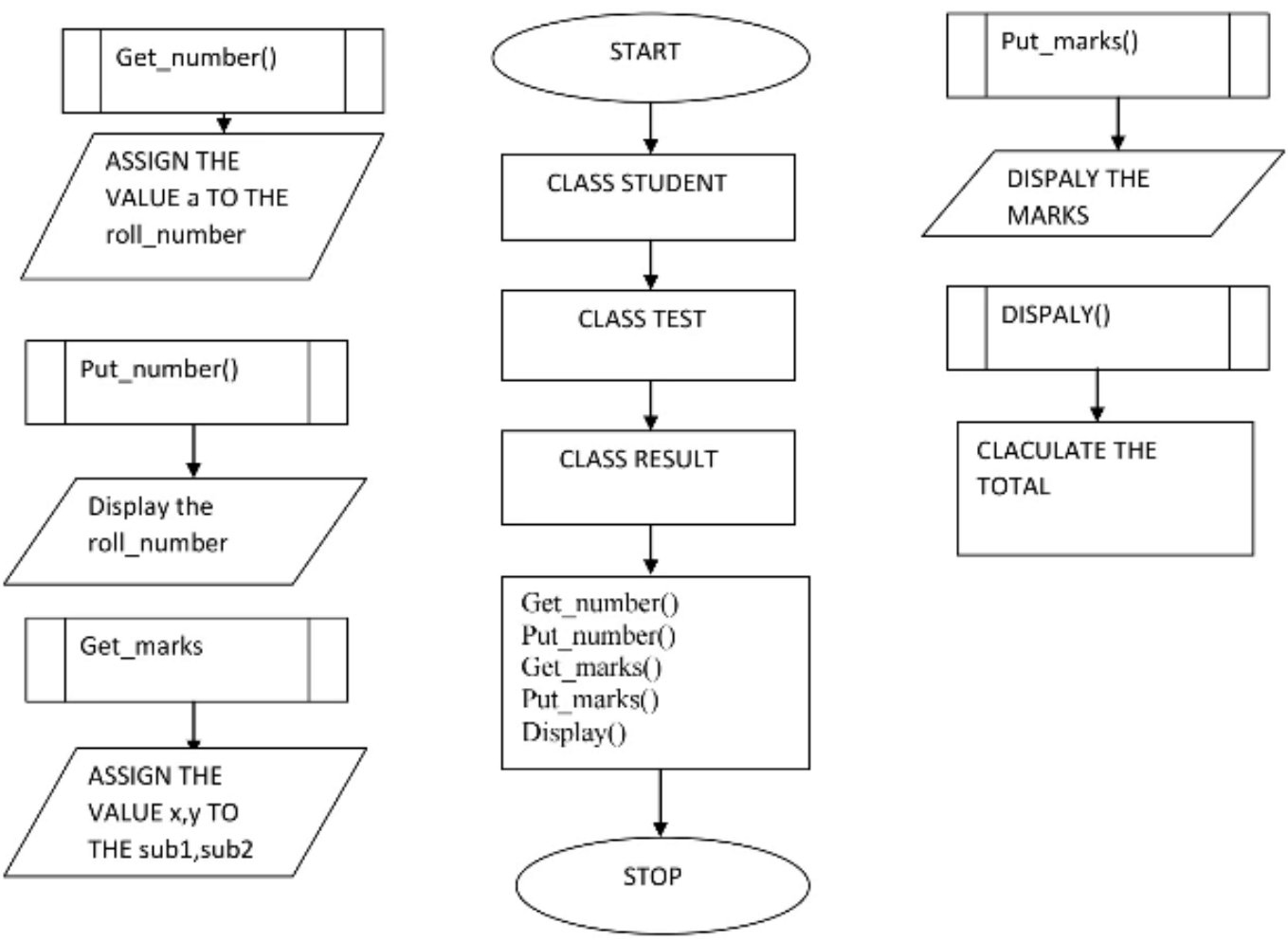
**Step 3:** Create a class "test", which inherits a " student" class use the protected mode to assign the "sub1","sub2" variable using get_marks(),put_marks() member function.

**Step 4:** Create a class "result", which inherits a "test" class to assign the total variable using display() member function .

**Step 5:** In the main function, create object "student_1"for the result class name and call the get_number(), get_marks(),display() function. Here the value is given as the parameter in the member function.

**Step 6:** Stop the process.

**FLOWCHART:**




**SOURCE CODE;**

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

class student
{
protected:
int roll_number;
public:
void get_number(int);
void put_number(void);
};

void student::get_number(int a)
{
roll_number=a;
```

```
}

void student::put_number()
{
cout<<"ROLL NUMBER: "<<roll_number<<"\n";
}

class test:public student
{
protected:
float sub1;
float sub2;
public:

void get_marks(float,float);
void put_marks(void);
};

void test::get_marks(float x,float y)
{
sub1=x;
sub2=y;
}

void test::put_marks()
{
cout<<"Marks in Sub1= "<<sub1<<"\n";
cout<<"Marks in Sub2= "<<sub2<<"\n";
}

class result:public test
{
float total;
public:

void display(void);
};


void result::display(void)
{
```

```
total=sub1+sub2;
put_number();
put_marks();
cout<<"Total = "<<total<<"\n";
}


int main()
{
result student1;

student1.get_number(3010);
student1.get_marks(60,59.88);
student1.display();

return 0;
}
```

**OUTPUT:**

```
ROLL NUMBER  :    3010
Marks in Sub1     :    60
Marks in Sub2     :    59.88
Total             :    119.88
```

**RESULT:**

Thus the demonstration of the multilevel inheritances was successfully executed

**4) HIERARHICAL INHERITANCE:**

**AIM:** To write a c++ program to demonstrate hierarchical inheritance

**PROCEDURE :**

**Step 1:** Start the process.

**Step 2:** Create a class "test", use protected mode to assign the part1, part2 variable using get_marks(), put_marks() member function.

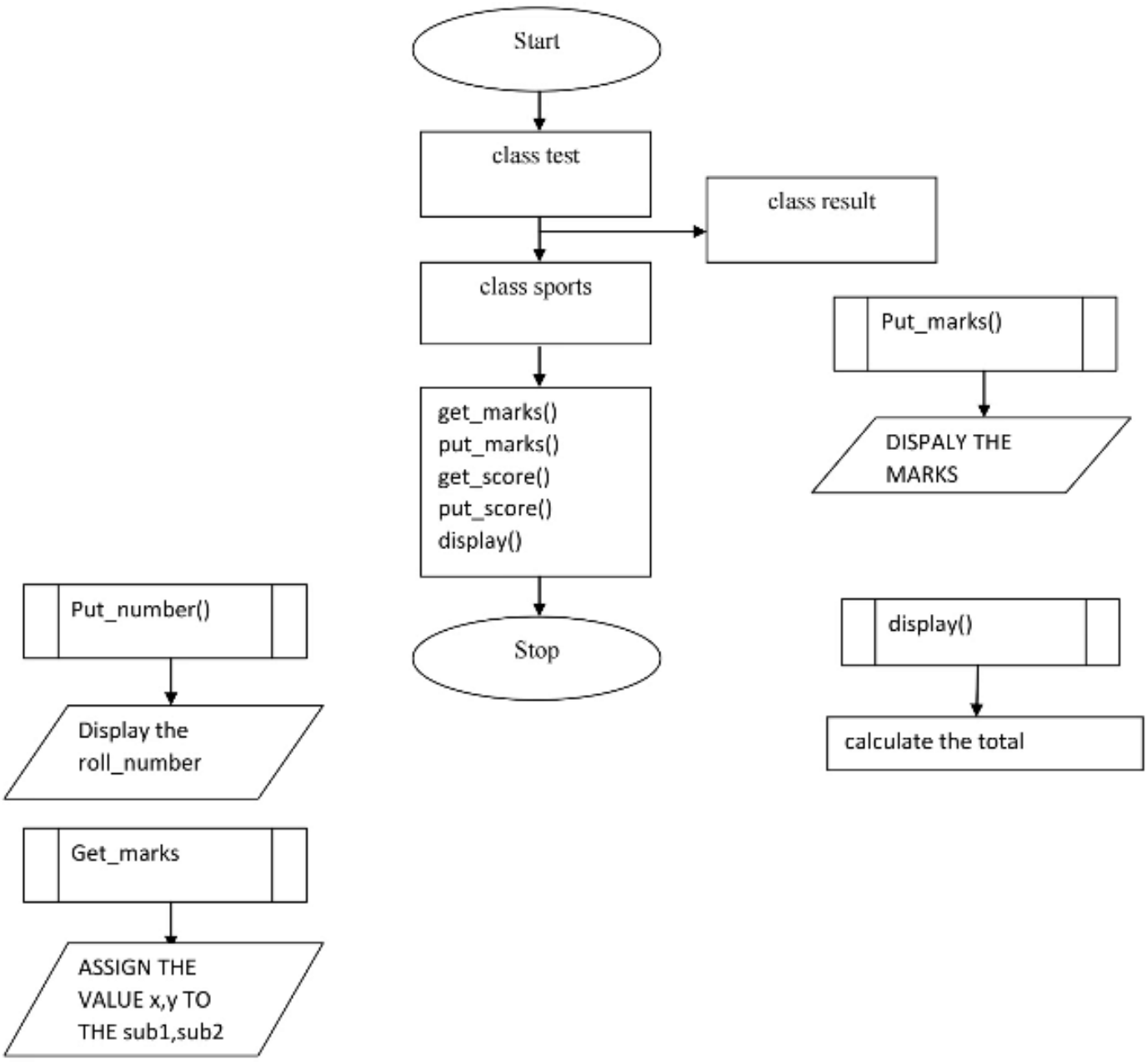
**Step 3:** Create a class "sports", use protected mode to assign the score variable using get_score(), put_score() member function.

**Step 4:** Create a class "result", which inherits the class "test", and class "sports" use the display() function to calculate the total.

**Step 5:** In the main function create the object "student_1"for the class result and execute get_marks(),get_score(),and display() member function

**Step 6:** Stop the process.

**FLOWCHART:**




**SOURCE CODE;**

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

class test
{
protected:
float part1,part2;
public:
```

```
void get_marks(float x,float y)
{
part1=x; part2=y;
}

void put_marks(void)
{
cout<<"MARKS OBTAINED: "<<"\n";
cout<<"PART1 = "<<part1<<"\n";
cout<<"PART2= "<<part2<<"\n";
}
};
class sports
{
protected:
float score;
public:
void get_score(float s)
{
score=s;
}

void put_score(void)
{
cout<<"sports weightage:"<<score<<"\n\n";
}
};
class result:public test,public sports
{
float total;
```

```
public:

void display(void);
 };

void result::display(void)
{
total=part1+part2+score;
put_marks();
put_score();
cout<<"Total Score:" <<total<<"\n";
}
int main()
{
result student_1;
student_1.get_marks(27.5,33);
student_1.get_score(6.0);
student_1.display();
return 0;
}
```

**OUTPUT:**

MARKS OBTAINED:

PART1 = 27.5

PART2= 33

Sports Weightage : 6

Total Score: 66.5

**RESULT:**

Thus the demonstration of the hierarchal inheritances was successfully executed.

**5) HYBRID INHERITANCE:**

**AIM:**

To write a C++ program to demonstrate Hybrid Inheritance

**PROCEDURE:**

**Step 1:** Start the process

**Step 2:** Create a class "student" , use the protested mode to assign the member variable roll number using get_number(),put_number() member function .

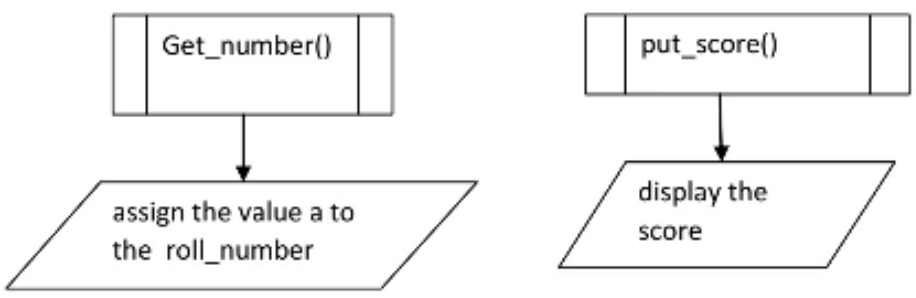
**Step 3:** Create a class "test", which inherits the "student" class, use the protected mode to assign the member variable part1,part2, using get_marks(),put_marks() member function.

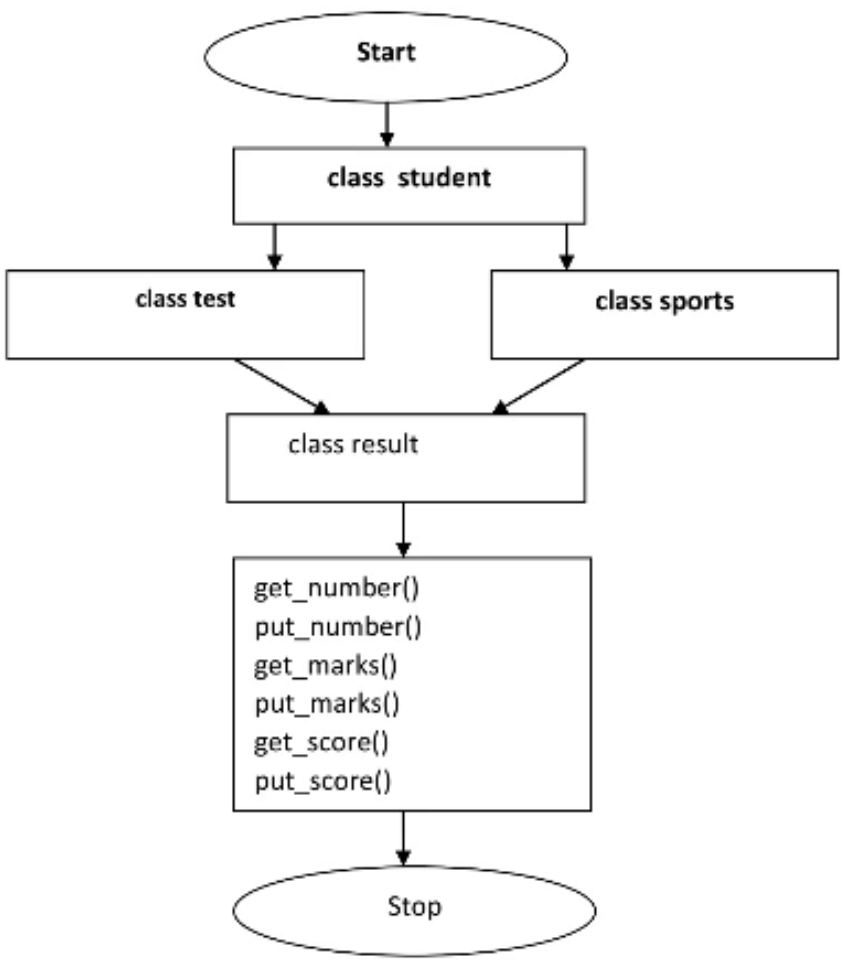
**Step 4:** Create a class "result", which inherits a class "test", and class "sports" use the display() function to calculate the total.

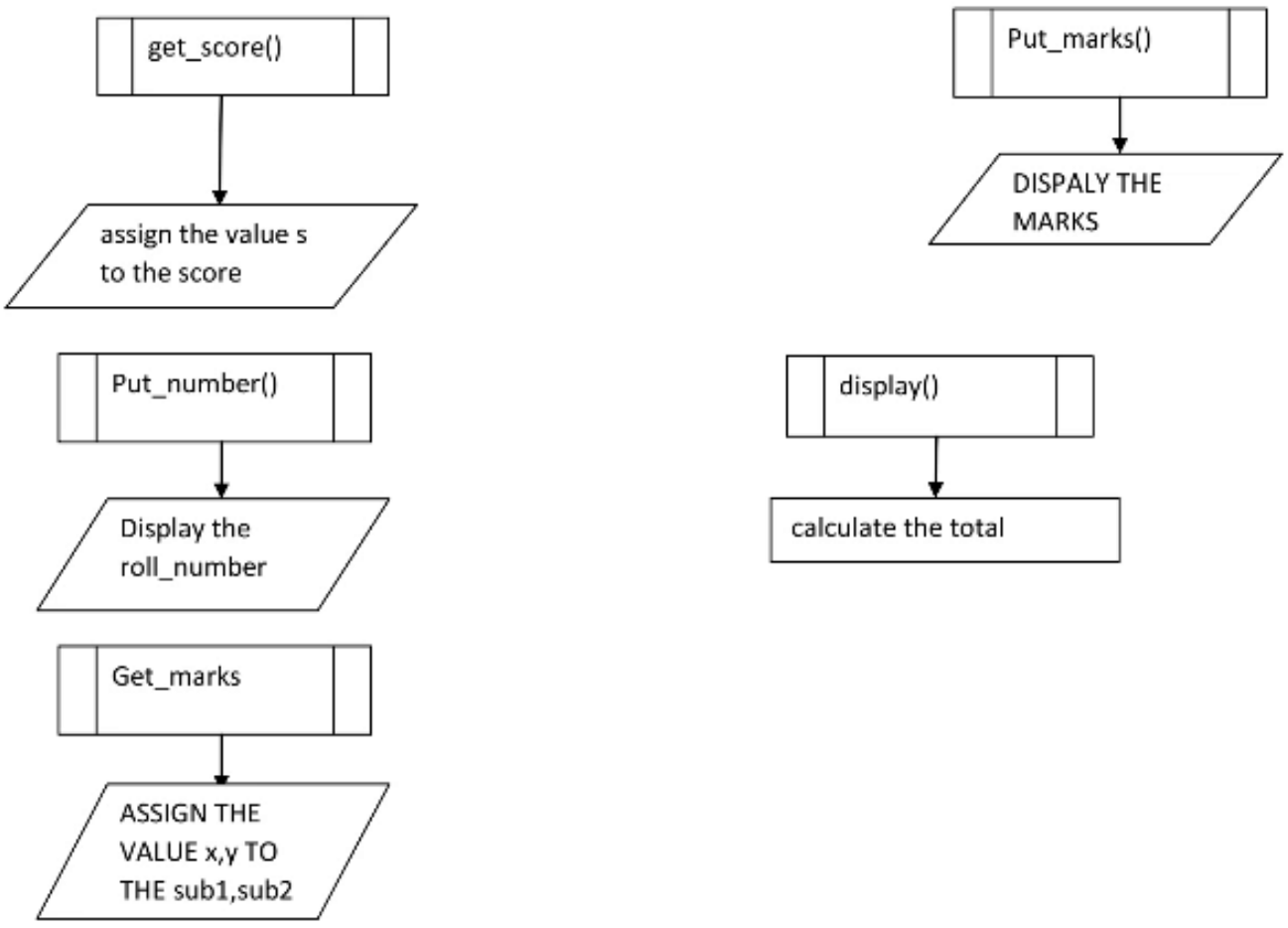
**Step 5:** In the main function create the object student_1 for a class "result" and execute get_marks(),get_score(),and display() member function

**Step 6:** Stop the process.

**FLOWCHART:**




Start
class student
class test
class sports
class result
get_number()
put_number()
get_marks()
put_marks()
get_score()
put_score()
Stop


get_score()
assign the value s to the score
Put_number()
Display the roll_number
Get_marks
ASSIGN THE VALUE x,y TO THE sub1,sub2
Put_marks()
DISPALY THE MARKS
display()
calculate the total

**SOURCE CODE;**

```
#include<iostream>

using namespace std;

class student
{
protected:
int roll_number;
public:
void get_number(int a)
{
roll_number=a;
}

void put_number(void)
{
cout<<"Roll No: "<<roll_number<<"\n";
}
};

class test:public student
{
protected:
float part1,part2;
public:

void get_marks(float x,float y)
{
part1=x;part2=y;
}

void put_marks(void)
{
cout<<"MARKS OBTAINED: "<<"\n";
cout<<"PART1 = "<<part1<<"\n";
cout<<"PART2= "<<part2<<"\n";
}
};
```

```
class sports
{
protected:
float score;
public:

void get_score(float s)
{
score=s;
}
void put_score(void)
{
cout<<"Sports Score:"<<score<<"\n\n";
}
};

class result:public test, public sports
{
float total;
public:
void display(void);
};
void result::display(void)
{
total=part1+part2+score;
put_number();
put_marks();
put_score();
cout<<"Total Score:" <<total<<"\n";
}

int main()
{
result student_1;
student_1.get_number(1234);
student_1.get_marks(27.5,33.0);
student_1.get_score(6.0);
student_1.display();
return 0;
}
```

**OUTPUT:**

Roll No : 1234
MARKS OBTAINED:
PART1 = 27.5
PART2 = 33
Sports Score:6
Total Score: 66.5

**RESULT:**
Thus the demonstration of the hybrid inheritances was successfully executed

## EX. NO: 10 - VIRTUAL FUNCTIONS

**AIM**:

To Write a C++ Program To Demonstrate Virtual Functions

**PROCEDURE**:

**Step 1 :** Start the process.

**Step 2 :** Define a class B with the following members: void display ( ) and Virtual void show ().

**Step 3 :** Define a derived class Derived from the base class B with the Following members: void display ( ) and void show().

**Step 4 :** Create the base class object and pointer variable.

**Step 5 :** Create the derived class object.

**Step 6 :** Assign base class object address to the pointer variable.

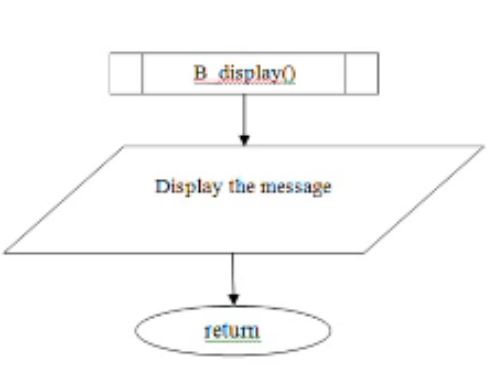
**Step 7 :** Call the functions display() and show() using the pointer variable.

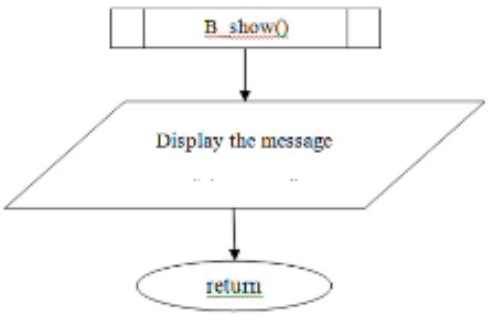
**Step 8 :** Assign derived class object address to the pointer variable.

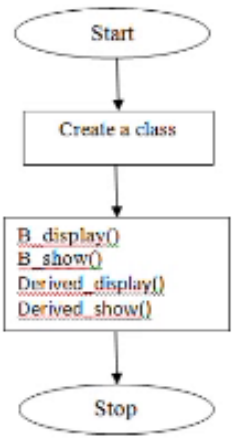
**Step 9 :** Call the functions display() and show() using the pointer variable.

**Step 10 :** Stop the process.

**FLOWCHART**



**SOURCE CODE:**

```
#include<iostream>
using namespace std;
class B
{
public:
        void display()
        {
                cout<<"\n Display Base";
        }
        virtual void show()
        {
                cout<<"\n Show Base";
        }
};
class Derived:public B
{
public:
        void display()
        {
                cout<<"\n Display Derived";
        }
        void show()
        {
                cout<<"\n Show Derived";
        }
};
int main()
{
B B1;
Derived D;
B *bptr;
cout<<"\n bPTR POINTS TO BASE\n";
bptr=&B1;
bptr->display();
bptr->show();
cout<<"\n\n bPTR POINTS TO DERIVED \n";
bptr=&D;
bptr->display();
bptr->show();
return 0;
}
```

**OUTPUT:**

bPTR POINTS TO BASE
Display Base
Show Base
bPTR POINTS TO DERIVED
Display Base
Show Derived

**RESULT:**

Thus the demonstration of the virtual function was successfully executed