

\* **Purpose** : Classwork.

\* **Date** : 31/12/2025

\* **Author** : Vikas Srivastava

\* **ID** : 55984

\* **Batch ID** : 25SUB4505

---

1. **Code** : Write a program to demonstrate the use of function pointers in C by creating a simple calculator, where different arithmetic operations are performed by passing function pointers to a common calculation function.

```
calcPtrFour.cpp X
calcPtrFour.cpp > calc(FPTR, int, int)
1  #include <stdio.h>
2  int Add(int,int);
3  int Sub(int,int);
4  int Divi(int,int);
5  int Mult(int,int);
6  int Modi(int,int);
7  typedef int (*FPTR)(int,int);
8  int calc(FPTR, int, int);
9  int main(){
10     FPTR arr[] = {Add, Sub, Mult, Divi, Modi, Add, Add, Sub, Mult, NULL};
11
12     for (int cnt = 0; arr[cnt] != NULL; cnt++)
13         printf("calculating... %d\n", calc(arr[cnt], 100, 20));
14 }
15 int Add(int x,int y){
16     return x+y;
17 }
18 int Sub(int x,int y){
19     return x-y;
20 }
21 int Divi(int x,int y){
22     return x/y;
23 }
24 int Mult(int x,int y){
25     return x*y;
26 }
27 int Modi(int x,int y){
28     return x%y;
29 }
30 int calc(FPTR fPtr, int x, int y){
31     return fPtr(x, y);
32 }
```

### **Output :**

```
PS C:\Users\VIKAS SRIVASTAWA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ -lcPtrFour.cpp
PS C:\Users\VIKAS SRIVASTAWA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
calculating... 120
calculating... 80
calculating... 2000
calculating... 5
calculating... 0
calculating... 120
calculating... 120
calculating... 80
calculating... 2000
```

2. **Code :** Write a program to demonstrate how function pointers can be passed as arguments to another function in C, allowing the same calculation function to perform different arithmetic operations dynamically.

```
C calcPtrOne.c X
C calcPtrOne.c > ...
1  #include <stdio.h>
2
3  int Add(int,int);
4  int Sub(int,int);
5  int Divi(int,int);
6  int Mult(int,int);
7  int Modi(int,int);
8
9  int calc(int (*)(int,int), int, int);
10
11 int main(){
12     printf("Adding %d\n", calc(Add, 100, 20));
13     printf("Subtracting %d\n", calc(Sub, 100, 20));
14     printf("Dividing %d\n", calc(Divi, 100, 20));
15     printf("Multiplying %d\n", calc(Mult, 100, 20));
16     printf("Modulus %d\n", calc(Modi, 100, 3));
17 }
18
19 int Add(int x,int y){
20     return x+y;
21 }
22 int Sub(int x,int y){
23     return x-y;
24 }
25 int Divi(int x,int y){
26     return x/y;
27 }
28 int Mult(int x,int y){
29     return x*y;
30 }
31 int Modi(int x,int y){
32     return x%y;
33 }
34 int calc(int (*fPtr)(int,int), int x, int y){
35     return fPtr(x, y);
36 }
```

### Output :

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\calcPtrOne.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Adding 120
Subtracting 80
Dividing 5
Multiplying 2000
Modulus 1
```

3. **Code :** Write a program to demonstrate the use of function pointers in C++ with the using keyword (type alias), where arithmetic functions are passed to a common calculator function to perform different operations dynamically.

```
calcPtrThree.cpp X
calcPtrThree.cpp > ...
1  #include <cstdlib>
2  using namespace std;
3  int Add(int,int);
4  int Sub(int,int);
5  int Divi(int,int);
6  int Mult(int,int);
7  int Modi(int,int);
8  using FPTR = int (*)(int,int);
9  int calc(FPTR, int, int);
10 int main(){
11     printf("Adding %d\n", calc(Add, 100, 20));
12     printf("Subtracting %d\n", calc(Sub, 100, 20));
13     printf("Dividing %d\n", calc(Divi, 100, 20));
14     printf("Multiplying %d\n", calc(Mult, 100, 20));
15     printf("Modulus %d\n", calc(Modi, 100, 3));
16 }
17 int Add(int x,int y){
18     return x+y;
19 }
20 int Sub(int x,int y){
21     return x-y;
22 }
23 int Divi(int x,int y){
24     return x/y;
25 }
26 int Mult(int x,int y){
27     return x*y;
28 }
29 int Modi(int x,int y){
30     return x%y;
31 }
32 int calc(FPTR fPtr, int x, int y){
33     return fPtr(x, y);
34 }
```

### Output :

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ -IcalcPtrThree.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Adding 120
Subtracting 80
Dividing 5
Multiplying 2000
Modulus 1
```

4. **Code :** Write a program to demonstrate the use of function pointers in C, where different arithmetic operations (addition, subtraction, multiplication, division, modulus) are passed as arguments to a common calculation function to perform operations dynamically.

```
calcPtrThree.cpp  calcPtrTwo.c
C calcPtrTwo.c > Add(int,int)
1 #include <stdio.h>
2
3 int Add(int,int);
4 int Sub(int,int);
5 int Divi(int,int);
6 int Mult(int,int);
7 int Modi(int,int);
8 typedef int (*FPTR)(int,int);
9 int calc(FPTR, int, int);
10 int main(){
11     printf("Adding %d\n", calc(Add, 100, 20));
12     printf("Subtracting %d\n", calc(Sub, 100, 20));
13     printf("Dividing %d\n", calc(Divi, 100, 20));
14     printf("Multiplying %d\n", calc(Mult, 100, 20));
15     printf("Modulus %d\n", calc(Modi, 100, 3));
16 }
17
18 int Add(int x,int y){
19     return x+y;
20 }
21 int Sub(int x,int y){
22     return x-y;
23 }
24 int Divi(int x,int y){
25     return x/y;
26 }
27 int Mult(int x,int y){
28     return x*y;
29 }
30 int Modi(int x,int y){
31     return x%y;
32 }
33 int calc(FPTR fPtr, int x, int y){
34     return fPtr(x, y);
35 }
```

### Output :

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc -IcalcPtrTwo.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Adding 120
Subtracting 80
Dividing 5
Multiplying 2000
Modulus 1
```

5. **Code :** Write a program to demonstrate multiple inheritance in C++, where a derived class inherits from two base classes containing functions with the same names, and the scope resolution operator (::) is used to explicitly specify which base class function to invoke.

```
multipleOne.cpp X
multipleOne.cpp > ...
1 #include <iostream>
2 using namespace std;
3
4 class BaseOne{
5 public:
6     void funOne(){cout<<"BaseOne::funOne()"<<endl; }
7     void funTwo(){cout<<"BaseOne::funTwo()"<<endl; }
8 };
9
10 class BaseTwo{
11 public:
12     void funOne(){cout<<"BaseTwo::funOne()"<<endl; }
13     void funTwo(){cout<<"BaseTwo::funTwo()"<<endl; }
14 };
15
16 class Derived: public BaseOne, public BaseTwo{
17
18 };
19
20 int main(){
21     Derived dobj;
22     dobj.BaseOne::funOne();
23     dobj.BaseTwo::funTwo();
24 }
```

### Output :

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ multipleOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
BaseOne::funOne()
BaseTwo::funTwo()
```

6. **Code** : Write a program to implement a custom MyString class in C++ that manages strings using dynamic memory allocation, demonstrating deep copy through a copy constructor, copy assignment operator, overloaded assignment for C-strings, destructor, and stream insertion (<<) operator overloading to safely handle memory and display string details.

```
myStringOne.cpp X
myStringOne.cpp >...
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class MyString{
6     char *str;
7     int len;
8 public:
9     MyString(const char *str=" "):len(strlen(str)+1){ //default constructor --> (1)
10         str = new char[len + 1];
11         strcpy(str, str);
12     }
13     ~MyString(){
14         if (len)
15             delete []str;
16         str = nullptr;
17         len=0;
18     }
19
20     MyString& operator=(const char *st){ //assignment operator --> (3)
21         if (len) //handling memory leakage
22             delete []str;
23         len = strlen(st) + 1;
24         str = new char[len + 1];
25         strcpy(str, st);
26         return *this;
27     }
28
29     MyString& operator=(const MyString& rhs){ //assignment operator --> (3)
30
31
32         if (this != &rhs) //handling self reference
33         {
34             if (len) //previous data deleted before assigning new value
35                 delete []str;
36             //handling dangling pointer
37             len = rhs.len;
38             str = new char[len + 1];
39             strcpy(str, rhs.str);
40         }
41         return *this;
42     }
43
44     MyString(const MyString& rhs):len(rhs.len){ //Copy constructor --> (2)
45         str = new char[len + 1];
46         strcpy(str, rhs.str);
47     }
48
49     friend ostream& operator <<(ostream &, const MyString&);
50 };
51
52 int main(){
53     MyString one = "One string here is to initialize";
54     MyString two;
55     two = "New string assigned here with new value"; //assigning a C string
56     MyString three;
57     three = two; //assigning an object of same class --> copy assignment done
58
59     cout<<"One: "<<one<<endl;
60     cout<<"Two: "<<two<<endl;
61     cout<<"Three: "<<three<<endl;
62 }
63
64 ostream& operator <<(ostream &out, const MyString& rhs){
65     out<<"len: "<<rhs.len<<"\t\tStr: "<<rhs.str;
66     return out;
67 }
```

### Output :

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_Cpp\Day_8\Classwork> g++ -VmyStringOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_Cpp\Day_8\Classwork> .\a.exe
One: len: 33          Str: One string here is to initialize
Two: len: 40          Str: New string assigned here with new value
Three: len: 40        Str: New string assigned here with new value
```

7. **Code :** Write a program to demonstrate assignment operator overloading in C++, where a class overloads the assignment operator to handle assignment from a primitive data type as well as assignment from another object of the same class, ensuring proper value copying and object behavior.

```

operatorAssignOne.cpp
1  #include <iostream>
2  using namespace std;
3  class Test{
4  |   int data;
5  |   public:
6  |       Test(int x=0): data(x){}
7  |       Test& operator=(int);
8  |       Test& operator=(const Test&);
9  |       friend ostream& operator<<(ostream &, const Test&);
10 |   };
11 |   int main(){
12 |       Test obj;
13 |       obj = 100;
14 |       Test objOne;
15 |       objOne = obj;
16 |       cout<<"Obj: "<<obj<<"\t\tObjOne: "<<objOne<<endl;
17 |   }
18 |   Test& Test::operator=(int arg){
19 |       cout<<"Test& operator=(int arg)"<<endl;
20 |       data=arg;
21 |       return *this;
22 |   }
23 |   Test& Test::operator=(const Test& rhs){
24 |       cout<<"Test& Test::operator=(const Test& rhs)"<<endl;
25 |       data = rhs.data;
26 |       return *this;
27 |   }
28 |   ostream& operator<<(ostream &out, const Test& arg){
29 |       out<<"data: "<<arg.data;
30 |       return out;
31 |   }

```

**Output :**

```

PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\operatorAssignOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Test& operator=(int arg)
Test& Test::operator=(const Test& rhs)
Obj: data: 100      ObjOne: data: 100

```

8. **Code :** Write a program to demonstrate function overriding without virtual functions in C++, where a base class pointer points to both base and derived class objects, but the base class function is called in both cases due to static (compile-time) binding.

```

polymorpOne.cpp X
1  #include <iostream>
2  using namespace std;
3
4  class Base{
5  |   public:
6  |       void disp(){cout<<"Base::disp()"<<endl; }
7  |   };
8  class Derived:public Base{
9  |   public:
10 |       void disp(){cout<<"Derived::disp()"<<endl; }
11 |   };
12
13 |   int main(){
14 |       Base *bPtr, bObj;
15 |       Derived dObj;
16
17 |       bPtr = &bObj; //storing base class object
18 |       bPtr->disp(); //--> (1) calls base class function
19 |
20 |       bPtr = &dObj; //storing derived class object
21 |       bPtr->disp(); //--> (2) calls base class function
22 |   }

```

**Output :**

```

PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\polymorpOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Base::disp()
Base::disp()

```

9. **Code :** Write a program to demonstrate runtime polymorphism in C++ using virtual functions, where a base class pointer or reference calls the overridden functions of the derived class based on the actual object type at runtime, achieving dynamic binding.

```
polymorpThree.cpp X
1  #include <iostream>
2  using namespace std;
3
4  class Base{
5  public:
6      virtual void funOne(){cout<<"Base::funOne()"<<endl; }
7      virtual void funTwo(){cout<<"Base::funTwo()"<<endl; }
8      virtual void funThree(){cout<<"Base::funThree()"<<endl; }
9  };
10
11  class Derived:public Base{
12  public:
13      void funOne(){cout<<"Derived::funOne()"<<endl; }
14      void funTwo(){cout<<"Derived::funTwo()"<<endl; }
15      void funThree(){cout<<"Derived::funThree()"<<endl; }
16  };
17
18  void demoVirtFun(Base *bPtr){//polymorphism using Base class Pointer
19      cout<<"Using Base class Pointer variable"<<endl;
20      bPtr->funOne();
21      bPtr->funTwo();
22      bPtr->funThree();
23      cout<<"-----\n";
24  }
25
26  void demoVirtFun(Base &bPtr){//polymorphism using Base class reference variable
27      cout<<"Using Base class Reference variable"<<endl;
28      bPtr.funOne();
29      bPtr.funTwo();
30      bPtr.funThree();
31      cout<<"-----\n";
32  }
33
34  int main(){
35      Base bobj;
36      Derived dobj;
37
38      demoVirtFun(&bobj);
39      demoVirtFun(&dobj);
40  }
```

### Output :

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\polymorpThree.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Using Base class Pointer variable
Base::funOne()
Base::funTwo()
Base::funThree()
-----
Using Base class Pointer variable
Derived::funOne()
Derived::funTwo()
Derived::funThree()
-----
```

10. **Code :** Write a program to demonstrate function overriding using virtual functions in C++, where a base class pointer calls the appropriate disp() function based on the actual object type at runtime, illustrating dynamic binding and runtime polymorphism.

```
polymorpTwo.cpp X
1  #include <iostream>
2  using namespace std;
3
4  class Base{
5  public:
6      virtual void disp(){cout<<"Base::disp()"<<endl; }
7  };
8  class Derived:public Base{
9  public:
10     void disp(){cout<<"Derived::disp()"<<endl; }
11 };
12
13 int main(){
14     Base *bPtr, bobj;
15     Derived dobj;
16
17     bPtr = &bobj; //storing base class object
18     bPtr->disp(); //--> (1) calls base class function
19
20     bPtr = &dobj; //storing derived class object
21     bPtr->disp(); //--> (2) calls derived class function
22 }
```

### Output :

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork\ g++ .\polymorpTwo.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork\ .\a.exe
Base::disp()
Derived::disp()
```

11. **Code :** Write a program to demonstrate the use of function pointers with void return type in C, where functions are passed as arguments to another function and invoked dynamically through the function pointer.

```
C ptr2FunFour.c X
1  #include <stdio.h>
2
3  void fun();
4  void funOne();
5
6  typedef void (*FPTR)();
7
8  void funCaller(FPTR);
9
10 int main(){
11     funCaller(fun);
12     funCaller(&funOne);
13 }
14
15 void fun(){
16     printf("fun() called\n");
17 }
18
19 void funOne(){
20     printf("funOne() called\n");
21 }
22
23 void funCaller(FPTR ptr){
24     ptr();
25 }
```

### Output :

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork\ gcc .\ptr2FunFour.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork\ .\a.exe
fun() called
funOne() called
```

12. **Code :** Write a program to demonstrate function pointers in C++ using the using keyword to define a function pointer type, and invoke different functions dynamically through a common caller function.

```
C++ ptr2FunFour.cpp X
1  #include <cstdio>
2  using namespace std;
3
4  void fun();
5  void funOne();
6
7  using FPTR = void (*)();
8
9  void funCaller(FPTR);
10
11 int main(){
12     funCaller(fun);
13     funCaller(&funOne);
14 }
15
16 void fun(){
17     printf("fun() called\n");
18 }
19
20 void funOne(){
21     printf("funOne() called\n");
22 }
23
24 void funCaller(FPTR ptr){
25     ptr();
26 }
```

### **Output :**

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\ptr2FunFour.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
fun() called
funOne() called
```

- 13. Code :** Write a program to demonstrate basic function pointer usage in C, including declaring a function pointer, assigning it the address of a function, and calling the function indirectly using the pointer.

```
C ptr2FunOne.c X
1  #include <stdio.h>
2
3  void fun();
4
5  int main(){
6      void (*funPtr()); //declaration of a pointer to function taking no args return nothing
7
8      funPtr = &fun; //funPtr = fun
9
10     funPtr(); //calling fun() using pointer
11 }
12
13 void fun(){
14     printf("fun() called\n");
15 }
```

### **Output :**

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\ptr2FunOne.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
fun() called
```

- 14. Code :** Write a program to demonstrate how function pointers can be passed directly as function parameters in C and invoked inside another function, showing both direct and dereferenced function pointer calls.

```
C ptr2FunThree.c X
1  #include <stdio.h>
2
3  void fun();
4  void funOne();
5
6  void funCaller(void (*)( ));
7
8  int main(){
9      funCaller(fun);
10     funCaller(&funOne);
11 }
12
13 void fun(){
14     printf("fun() called\n");
15 }
16
17 void funOne(){
18     printf("funOne() called\n");
19 }
20
21 void funCaller(void (*fPtr)()){
22     //fPtr();
23     (*fPtr)(); //also valid
24 }
```

### **Output :**

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\ptr2FunThree.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
fun() called
funOne() called
```



- 15. Code :** Write a program to demonstrate how a function can be passed as an argument to another function in C using function pointers, allowing the called function to execute the passed function dynamically.

```
C ptr2FunTwo.c X
1  #include <stdio.h>
2
3  void fun();
4
5  void funCaller(void (*)( ));
6
7  int main(){
8      funCaller(fun); //funCaller(&fun);
9  }
10
11 void fun(){
12     printf("fun() called\n");
13 }
14
15 void funCaller(void (*fPtr)()){
16     fPtr();
17 }
```

**Output :**

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc -xptr2FunTwo.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
fun() called
```

- 16. Code :** Write a program to demonstrate an abstract class in C++ using a pure virtual function, where the base class cannot be instantiated and the derived class provides the function implementation to achieve runtime polymorphism.

```
pureVirtualOne.cpp X
1  #include <iostream>
2  using namespace std;
3
4  class Base{
5  public:
6      virtual void disp()=0; //definition is missing
7  };
8  class Derived:public Base{
9  public:
10     void disp(){cout<<"Derived::disp()"<<endl; }
11 };
12
13 int main(){
14     Base *bPtr; //object of Base class cannot be created
15     Derived dobj;
16
17     bPtr = &dobj; //storing derived class object
18     bPtr->disp(); //--> (2) calls derived class function
19 }
```

**Output :**

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ -xpureVirtualOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Derived::disp()
```

17. **Code :** Write a program to demonstrate advanced function pointer usage in C, where the address of the printf function is typecast to an integer type and back to a function pointer, and then invoked to print output, illustrating type casting and indirect function calls (for learning purposes only).

```
C typeCastingOne.c X
1  #include <stdio.h>
2
3  int main(){
4      long myInt = (long) printf ;
5
6      ((int (*)( ))myInt)("Hello World!...\n");
7  }
```

**Output :**

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\typeCastingOne.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Hello world!...
```

18. **Code :** Write a program to demonstrate hybrid inheritance in C++, where a derived class inherits from multiple base classes that share a common ancestor, and the scope resolution operator (::) is used to resolve ambiguity when calling base class member functions.

```
C++ virtualInherOne.cpp X
1  #include <iostream>
2  using namespace std;
3
4  class Base{
5  public:
6      void funOne(){cout <<"Base::funOne()"<<endl; }
7      void funTwo(){cout <<"Base::funTwo()"<<endl; }
8  };
9
10 class BaseOne:public Base{ };
11
12 class BaseTwo:public Base{ };
13
14 class Derived: public BaseOne, public BaseTwo{};
15
16 int main(){
17     Derived d;
18     d.BaseOne::funOne();
19     d.BaseTwo::funTwo();
20 }
```

**Output :**

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\virtualInherOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Base::funOne()
Base::funTwo()
```

19. **Code :** Write a program to demonstrate virtual inheritance in C++, where multiple derived classes inherit from a common base class virtually to avoid duplicate copies of base class members (diamond problem), allowing the final derived class to access base class functions without ambiguity.

```
virtualInherTwo.cpp X
1  #include <iostream>
2  using namespace std;
3
4  class Base{
5  public:
6      void funOne(){cout <<"Base::funOne()"<<endl; }
7      void funTwo(){cout <<"Base::funTwo()"<<endl; }
8  };
9
10 class BaseOne:virtual public Base{ };
11
12 class BaseTwo:public virtual Base{ };
13
14 class Derived: public BaseOne, public BaseTwo{};
15
16 int main(){
17     Derived d;
18     d.funOne();
19     d.funTwo();
20 }
```

**Output :**

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ virtualInherTwo.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Base::funOne()
Base::funTwo()
```

20. **Code :** Write a program to demonstrate runtime polymorphism using virtual functions in C++ and explain the internal working of virtual function calls through the virtual table (vTable), showing how function calls are dynamically resolved at runtime based on the actual object type.

```
vtableOne.cpp X
1  #include <iostream>
2  using namespace std;
3  class Base{
4  public:
5      virtual void funOne(){cout<<"Base::funOne()"<<endl; }
6      virtual void funTwo(){cout<<"Base::funTwo()"<<endl; }
7      virtual void funThree(){cout<<"Base::funThree()"<<endl; }
8  };
9  class Derived:public Base{
10 public:
11     void funOne(){cout<<"Derived::funOne()"<<endl; }
12     void funTwo(){cout<<"Derived::funTwo()"<<endl; }
13     void funThree(){cout<<"Derived::funThree()"<<endl; }
14 };
15 using FPTR = void (*)( );
16 void demoFun(Base *bPtr){ //Raw function call through pointers //Internal working
17     long *vPtr = (long *) (bPtr);
18     FPTR *vTable = ((FPTR *) *vPtr);
19     vTable[0]();
20     vTable[1]();
21     vTable[2]();
22     cout<<"*****\n";
23 }
24 void demoVirtFun(Base *bPtr){//normal function call achieving polymorphism
25     bPtr->funOne();
26     bPtr->funTwo();
27     bPtr->funThree();
28     cout<<"-----\n";
29 }
30 int main(){
31     Base bobj;
32     Derived dobj;
33     demoVirtFun(&bobj);
34     demoVirtFun(&dobj);
35 }
```

## Output :

```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ -o vtableOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Base::funOne()
Base::funTwo()
Base::funThree()
-----
Derived::funOne()
Derived::funTwo()
Derived::funThree()
Base::funTwo()
Base::funThree()
-----
Derived::funOne()
Derived::funTwo()
Derived::funThree()
-----
Derived::funOne()
Derived::funTwo()
Derived::funThree()
Derived::funOne()
Derived::funTwo()
Derived::funThree()
Derived::funTwo()
Derived::funThree()
Derived::funTwo()
Derived::funThree()
```