- ***Purpose* :** Classwork.

- ***Date* :** 31/12/2025

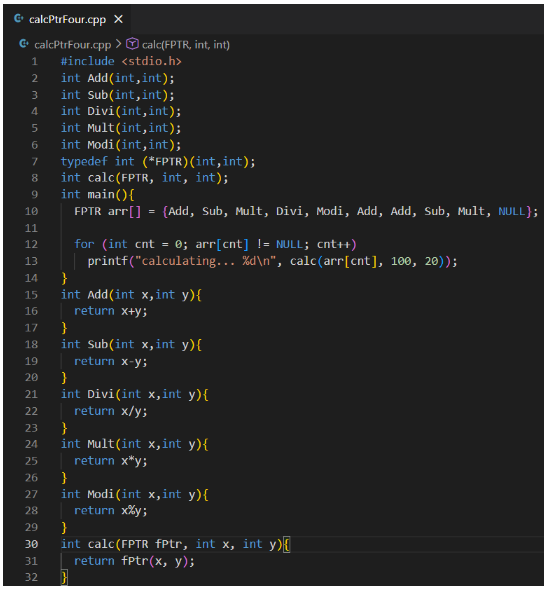
- ***Author* :** Vikas Srivastava

- ***ID* :** 55984

- ***Batch ID* :** 25SUB4505

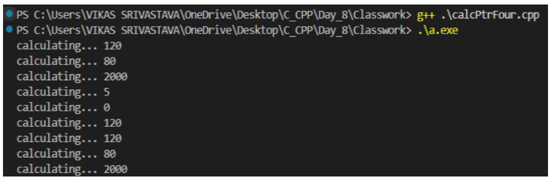
---

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.



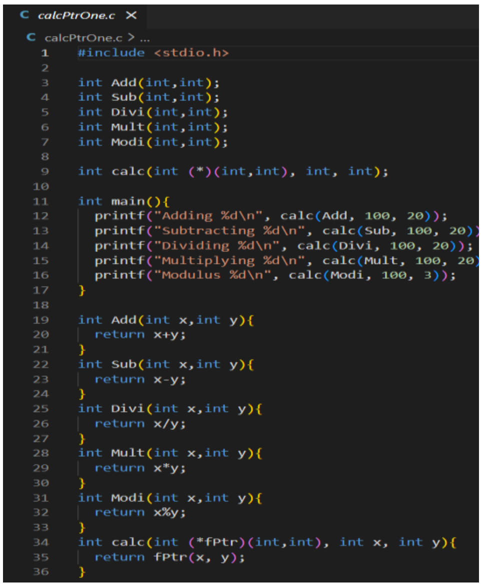
```c
#include <stdio.h>
int Add(int,int);
int Sub(int,int);
int Divi(int,int);
int Mult(int,int);
int Modi(int,int);
typedef int (*FPTR)(int,int);
int calc(FPTR, int, int);
int main(){
  FPTR arr[] = {Add, Sub, Mult, Divi, Modi, Add, Add, Sub, Mult, NULL};

  for (int cnt = 0; arr[cnt] != NULL; cnt++)
    printf("calculating... %d\n", calc(arr[cnt], 100, 20));
}
int Add(int x,int y){
  return x+y;
}
int Sub(int x,int y){
  return x-y;
}
int Divi(int x,int y){
  return x/y;
}
int Mult(int x,int y){
  return x*y;
}
int Modi(int x,int y){
  return x%y;
}
int calc(FPTR fPtr, int x, int y){
  return fPtr(x, y);
}
```

Output :



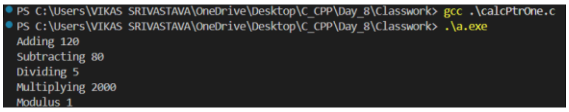
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\calcPtrFour.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
calculating... 120
calculating... 80
calculating... 2000
calculating... 5
calculating... 0
calculating... 120
calculating... 120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



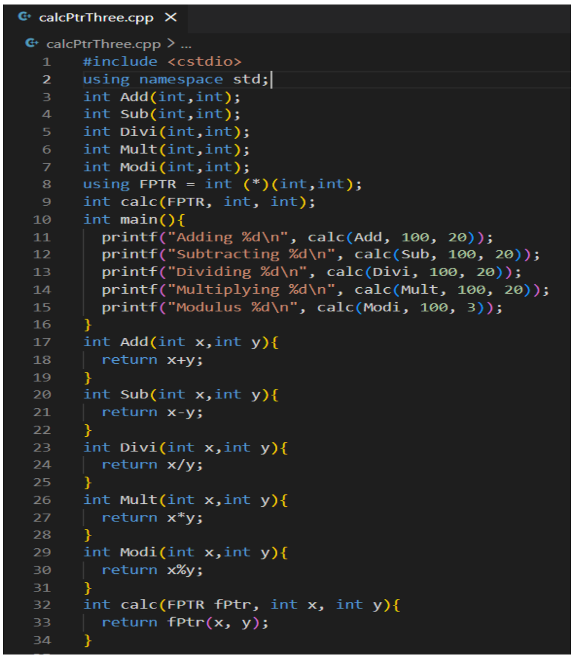
```c
#include <stdio.h>

int Add(int,int);
int Sub(int,int);
int Divi(int,int);
int Mult(int,int);
int Modi(int,int);

int calc(int (*)(int,int), int, int);

int main(){
  printf("Adding %d\n", calc(Add, 100, 20));
  printf("Subtracting %d\n", calc(Sub, 100, 20))
  printf("Dividing %d\n", calc(Divi, 100, 20));
  printf("Multiplying %d\n", calc(Mult, 100, 20)
  printf("Modulus %d\n", calc(Modi, 100, 3));
}

int Add(int x,int y){
  return x+y;
}
int Sub(int x,int y){
  return x-y;
}
int Divi(int x,int y){
  return x/y;
}
int Mult(int x,int y){
  return x*y;
}
int Modi(int x,int y){
  return x%y;
}
int calc(int (*fPtr)(int,int), int x, int y){
  return fPtr(x, y);
}
```

Output :



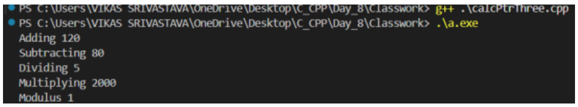
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\calcPtrOne.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Adding 120
Subtracting 80
Dividing 5
Multiplying 2000
Modulus 1
```

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.



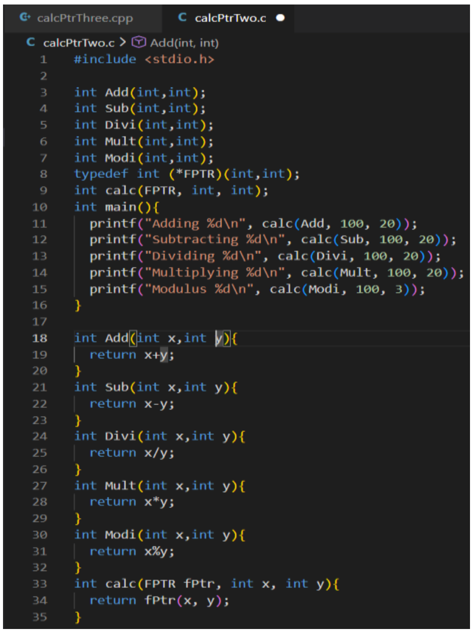
```cpp
#include <cstdio>
using namespace std;
int Add(int,int);
int Sub(int,int);
int Divi(int,int);
int Mult(int,int);
int Modi(int,int);
using FPTR = int (*)(int,int);
int calc(FPTR, int, int);
int main(){
  printf("Adding %d\n", calc(Add, 100, 20));
  printf("Subtracting %d\n", calc(Sub, 100, 20));
  printf("Dividing %d\n", calc(Divi, 100, 20));
  printf("Multiplying %d\n", calc(Mult, 100, 20));
  printf("Modulus %d\n", calc(Modi, 100, 3));
}
int Add(int x,int y){
  return x+y;
}
int Sub(int x,int y){
  return x-y;
}
int Divi(int x,int y){
  return x/y;
}
int Mult(int x,int y){
  return x*y;
}
int Modi(int x,int y){
  return x%y;
}
int calc(FPTR fPtr, int x, int y){
  return fPtr(x, y);
}
```

Output :



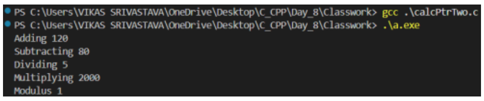
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\calcPtrThree.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Adding 120
Subtracting 80
Dividing 5
Multiplying 2000
Modulus 1
```

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.



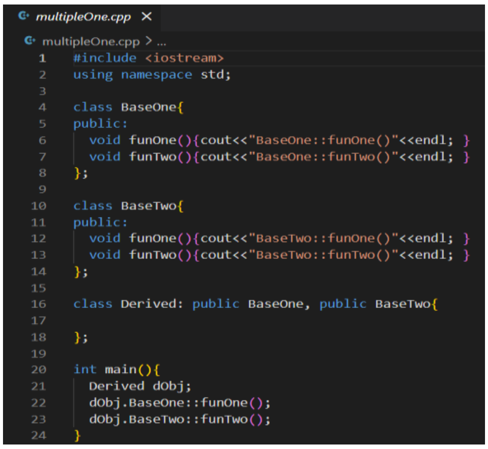
```c
#include <stdio.h>

int Add(int,int);
int Sub(int,int);
int Divi(int,int);
int Mult(int,int);
int Modi(int,int);
typedef int (*FPTR)(int,int);
int calc(FPTR, int, int);
int main(){
  printf("Adding %d\n", calc(Add, 100, 20));
  printf("Subtracting %d\n", calc(Sub, 100, 20));
  printf("Dividing %d\n", calc(Divi, 100, 20));
  printf("Multiplying %d\n", calc(Mult, 100, 20));
  printf("Modulus %d\n", calc(Modi, 100, 3));
}

int Add(int x,int y){
  return x+y;
}
int Sub(int x,int y){
  return x-y;
}
int Divi(int x,int y){
  return x/y;
}
int Mult(int x,int y){
  return x*y;
}
int Modi(int x,int y){
  return x%y;
}
int calc(FPTR fPtr, int x, int y){
  return fPtr(x, y);
}
```

Output :



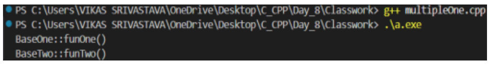
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\calcPtrTwo.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Adding 120
Subtracting 80
Dividing 5
Multiplying 2000
Modulus 1
```

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.



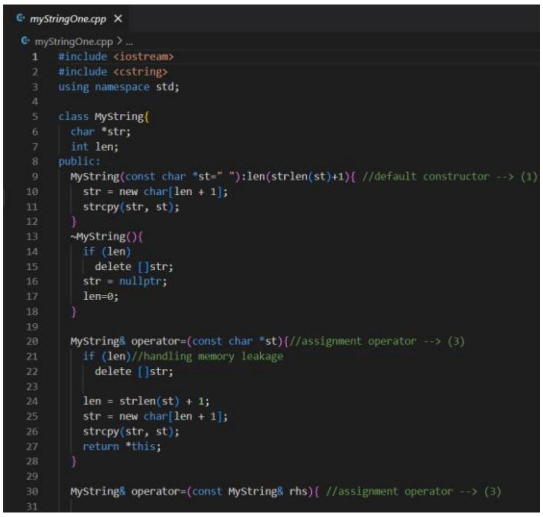
```cpp
#include <iostream>
using namespace std;

class BaseOne{
public:
  void funOne(){cout<<"BaseOne::funOne()"<<endl; }
  void funTwo(){cout<<"BaseOne::funTwo()"<<endl; }
};

class BaseTwo{
public:
  void funOne(){cout<<"BaseTwo::funOne()"<<endl; }
  void funTwo(){cout<<"BaseTwo::funTwo()"<<endl; }
};

class Derived: public BaseOne, public BaseTwo{

};

int main(){
  Derived dObj;
  dObj.BaseOne::funOne();
  dObj.BaseTwo::funTwo();
}
```

Output :

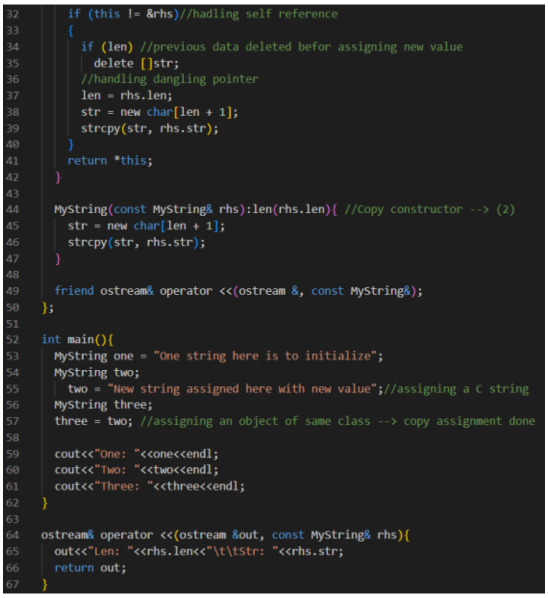


```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ multipleOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
BaseOne::funOne()
BaseTwo::funTwo()
```

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.

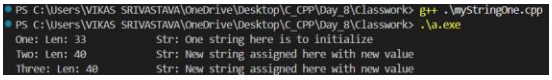


```cpp
#include <iostream>
#include <cstring>
using namespace std;

class MyString{
  char *str;
  int len;
public:
  MyString(const char *st=" "):len(strlen(st)+1){ //default constructor --> (1)
    str = new char[len + 1];
    strcpy(str, st);
  }
  ~MyString(){
    if (len)
      delete []str;
    str = nullptr;
    len=0;
  }

  MyString& operator=(const char *st){//assignment operator --> (3)
    if (len)//handling memory leakage
      delete []str;

    len = strlen(st) + 1;
    str = new char[len + 1];
    strcpy(str, st);
    return *this;
  }

  MyString& operator=(const MyString& rhs){ //assignment operator --> (3)

```



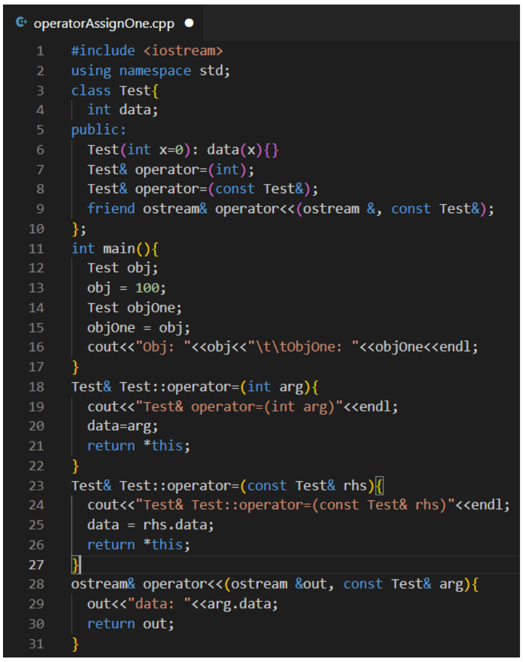
```cpp
    if (this != &rhs)//hadling self reference
    {
      if (len) //previous data deleted befor assigning new value
        delete []str;
      //handling dangling pointer
      len = rhs.len;
      str = new char[len + 1];
      strcpy(str, rhs.str);
    }
    return *this;
  }

  MyString(const MyString& rhs):len(rhs.len){ //Copy constructor --> (2)
    str = new char[len + 1];
    strcpy(str, rhs.str);
  }

  friend ostream& operator <<(ostream &, const MyString&);
};

int main(){
  MyString one = "One string here is to initialize";
  MyString two;
    two = "New string assigned here with new value";//assigning a C string
  MyString three;
  three = two; //assigning an object of same class --> copy assignment done

  cout<<"One: "<<one<<endl;
  cout<<"Two: "<<two<<endl;
  cout<<"Three: "<<three<<endl;
}

ostream& operator <<(ostream &out, const MyString& rhs){
  out<<"Len: "<<rhs.len<<"\t\tStr: "<<rhs.str;
  return out;
}
```

Output :



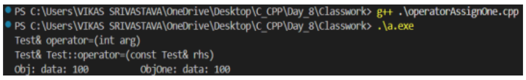
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\myStringOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
One: Len: 33            Str: One string here is to initialize
Two: Len: 40            Str: New string assigned here with new value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



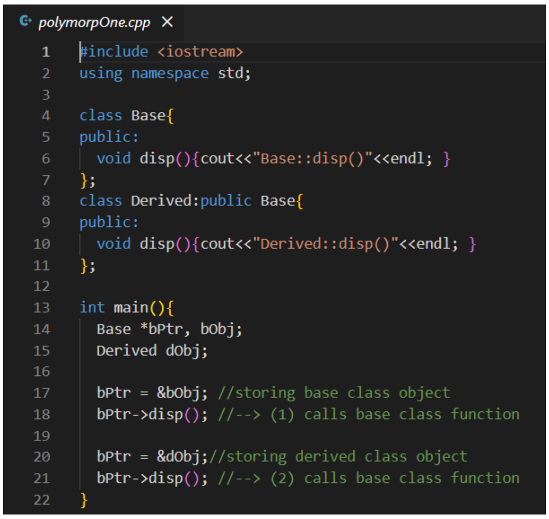
```cpp
#include <iostream>
using namespace std;
class Test{
    int data;
public:
    Test(int x=0): data(x){}
    Test& operator=(int);
    Test& operator=(const Test&);
    friend ostream& operator<<(ostream &, const Test&);
};
int main(){
    Test obj;
    obj = 100;
    Test objOne;
    objOne = obj;
    cout<<"Obj: "<<obj<<"\t\tObjOne: "<<objOne<<endl;
}
Test& Test::operator=(int arg){
    cout<<"Test& operator=(int arg)"<<endl;
    data=arg;
    return *this;
}
Test& Test::operator=(const Test& rhs){
    cout<<"Test& Test::operator=(const Test& rhs)"<<endl;
    data = rhs.data;
    return *this;
}
ostream& operator<<(ostream &out, const Test& arg){
    out<<"data: "<<arg.data;
    return out;
}
```

Output :



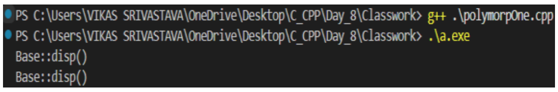
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\operatorAssignOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Test& operator=(int arg)
Test& Test::operator=(const Test& rhs)
Obj: data: 100          ObjOne: data: 100
```

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.



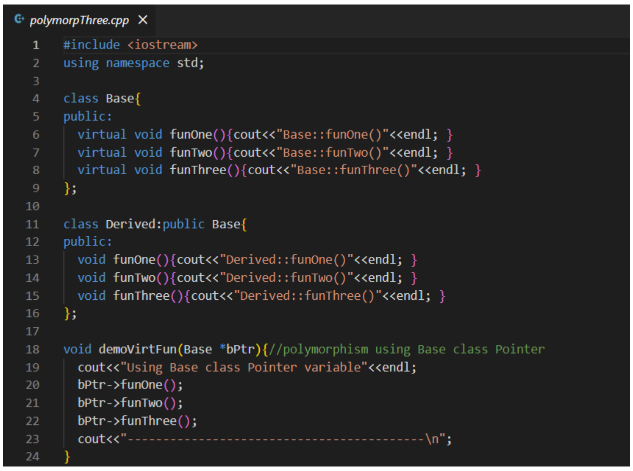
```cpp
#include <iostream>
using namespace std;

class Base{
public:
    void disp(){cout<<"Base::disp()"<<endl; }
};
class Derived:public Base{
public:
    void disp(){cout<<"Derived::disp()"<<endl; }
};

int main(){
    Base *bPtr, bObj;
    Derived dObj;

    bPtr = &bObj; //storing base class object
    bPtr->disp(); //--> (1) calls base class function

    bPtr = &dObj;//storing derived class object
    bPtr->disp(); //--> (2) calls base class function
}
```

Output :

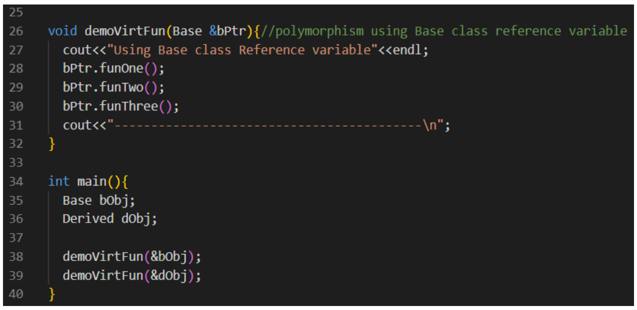


```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\polymorpOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Base::disp()
Base::disp()
```

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.

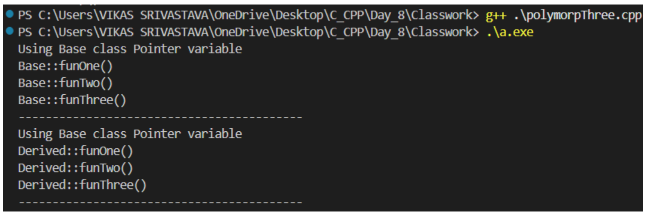


```
polymorpThree.cpp
#include <iostream>
using namespace std;

class Base{
public:
  virtual void funOne(){cout<<"Base::funOne()"<<endl; }
  virtual void funTwo(){cout<<"Base::funTwo()"<<endl; }
  virtual void funThree(){cout<<"Base::funThree()"<<endl; }
};

class Derived:public Base{
public:
  void funOne(){cout<<"Derived::funOne()"<<endl; }
  void funTwo(){cout<<"Derived::funTwo()"<<endl; }
  void funThree(){cout<<"Derived::funThree()"<<endl; }
};

void demoVirtFun(Base *bPtr){//polymorphism using Base class Pointer
  cout<<"Using Base class Pointer variable"<<endl;
  bPtr->funOne();
  bPtr->funTwo();
  bPtr->funThree();
  cout<<"----------------------------------------\n";
}
```



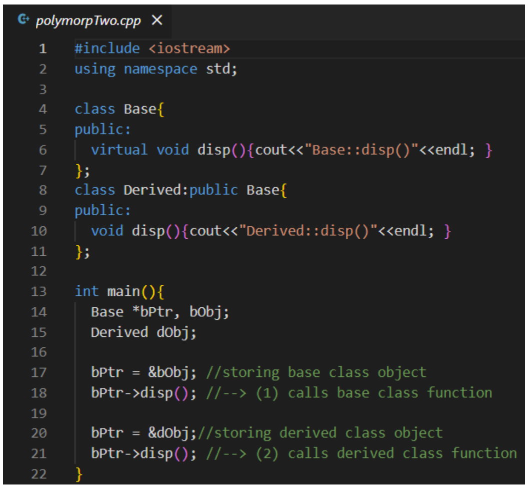
```

void demoVirtFun(Base &bPtr){//polymorphism using Base class reference variable
  cout<<"Using Base class Reference variable"<<endl;
  bPtr.funOne();
  bPtr.funTwo();
  bPtr.funThree();
  cout<<"----------------------------------------\n";
}

int main(){
  Base bObj;
  Derived dObj;

  demoVirtFun(&bObj);
  demoVirtFun(&dObj);
}
```

Output :



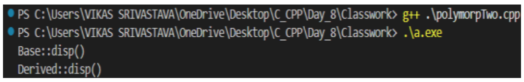
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\polymorpThree.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Using Base class Pointer variable
Base::funOne()
Base::funTwo()
Base::funThree()
----------------------------------------
Using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



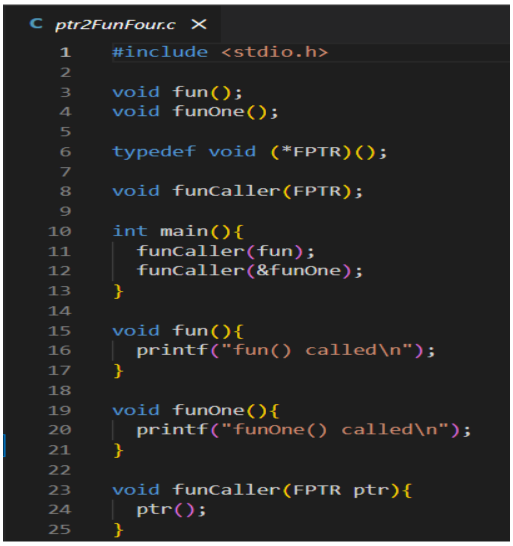
```
polymorpTwo.cpp
#include <iostream>
using namespace std;

class Base{
public:
  virtual void disp(){cout<<"Base::disp()"<<endl; }
};
class Derived:public Base{
public:
  void disp(){cout<<"Derived::disp()"<<endl; }
};

int main(){
  Base *bPtr, bObj;
  Derived dObj;

  bPtr = &bObj; //storing base class object
  bPtr->disp(); //--> (1) calls base class function

  bPtr = &dObj;//storing derived class object
  bPtr->disp(); //--> (2) calls derived class function
}
```

Output :



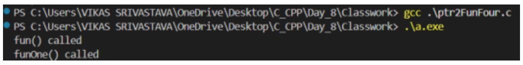
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\polymorpTwo.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Base::disp()
Derived::disp()
```

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.



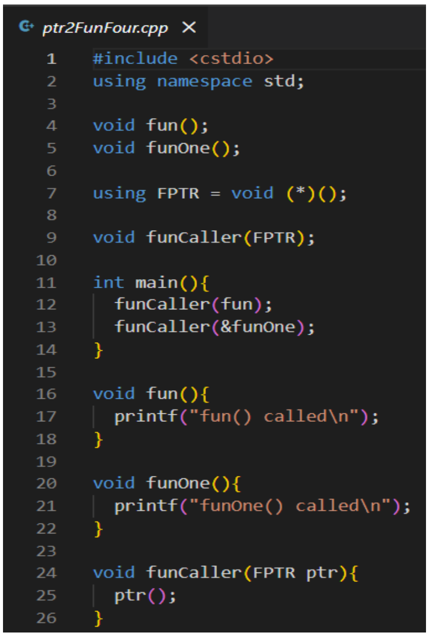
```c
#include <stdio.h>

void fun();
void funOne();

typedef void (*FPTR)();

void funCaller(FPTR);

int main(){
  funCaller(fun);
  funCaller(&funOne);
}

void fun(){
  printf("fun() called\n");
}

void funOne(){
  printf("funOne() called\n");
}

void funCaller(FPTR ptr){
  ptr();
}
```

Output :



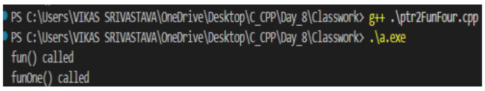
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\ptr2FunFour.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
fun() called
funOne() called
```

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.



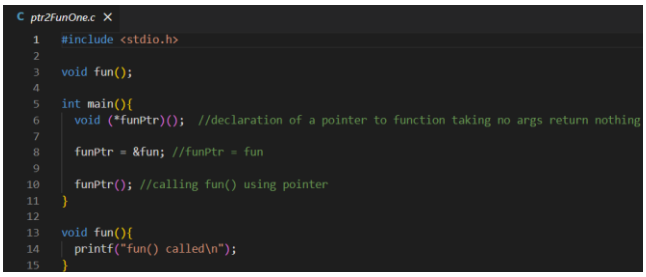
```cpp
#include <cstdio>
using namespace std;

void fun();
void funOne();

using FPTR = void (*)();

void funCaller(FPTR);

int main(){
  funCaller(fun);
  funCaller(&funOne);
}

void fun(){
  printf("fun() called\n");
}

void funOne(){
  printf("funOne() called\n");
}

void funCaller(FPTR ptr){
  ptr();
}
```

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
```

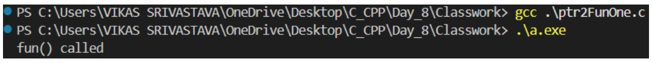
13. ***Code :*** Write a program to demonstrate basic function pointer usage in C, including declaring a function pointer, assigning it the address of a function, and calling the function indirectly using the pointer.



ptr2FunOne.c

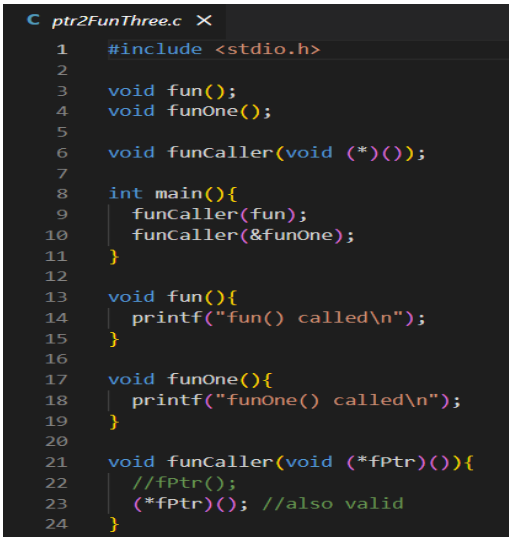
```c
#include <stdio.h>

void fun();

int main(){
    void (*funPtr)();  //declaration of a pointer to function taking no args return nothing

    funPtr = &fun; //funPtr = fun

    funPtr(); //calling fun() using pointer
}

void fun(){
    printf("fun() called\n");
}
```

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
```

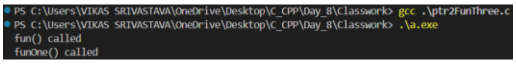
14. ***Code :*** Write a program to demonstrate how function pointers can be passed directly as function parameters in C and invoked inside another function, showing both direct and dereferenced function pointer calls.



ptr2FunThree.c

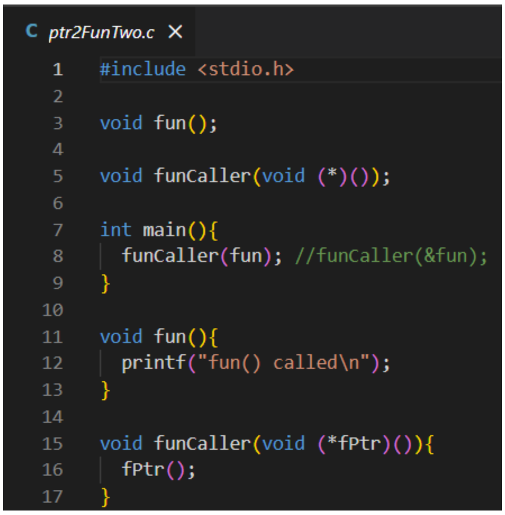
```c
#include <stdio.h>

void fun();
void funOne();

void funCaller(void (*)());

int main(){
    funCaller(fun);
    funCaller(&funOne);
}

void fun(){
    printf("fun() called\n");
}

void funOne(){
    printf("funOne() called\n");
}

void funCaller(void (*fPtr)()){
    //fPtr();
    (*fPtr)(); //also valid
}
```

Output :



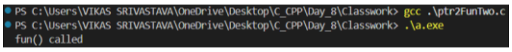
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\ptr2FunThree.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
fun() called
funOne() called
```

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.



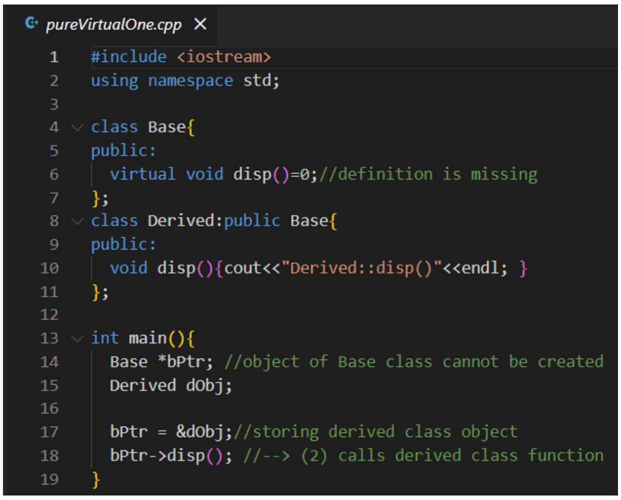
```c
#include <stdio.h>

void fun();

void funCaller(void (*)());

int main(){
    funCaller(fun); //funCaller(&fun);
}

void fun(){
    printf("fun() called\n");
}

void funCaller(void (*fPtr)()){
    fPtr();
}
```

Output :



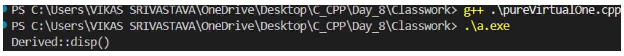
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\ptr2FunTwo.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
fun() called
```

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.



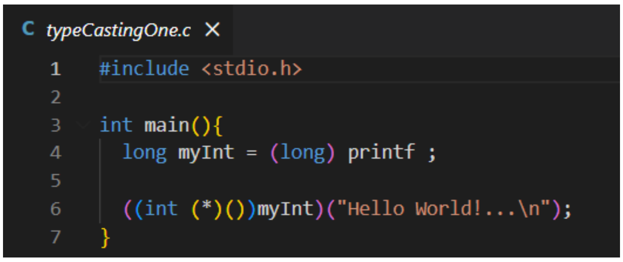
```cpp
#include <iostream>
using namespace std;

class Base{
public:
    virtual void disp()=0;//definition is missing
};
class Derived:public Base{
public:
    void disp(){cout<<"Derived::disp()"<<endl; }
};

int main(){
    Base *bPtr; //object of Base class cannot be created
    Derived dObj;

    bPtr = &dObj;//storing derived class object
    bPtr->disp(); //--> (2) calls derived class function
}
```

Output :



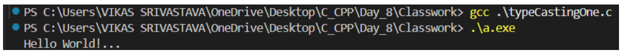
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\pureVirtualOne.cpp
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Derived::disp()
```

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).



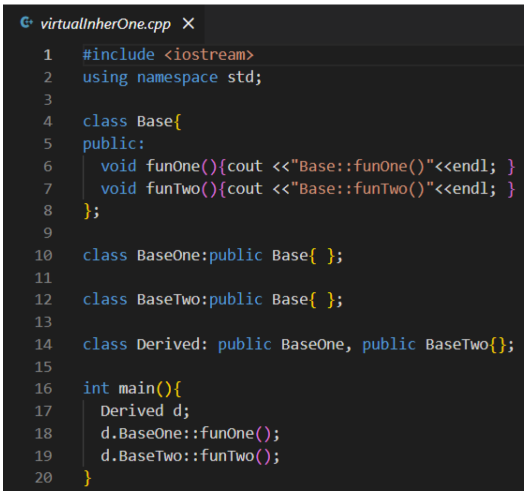
```c
#include <stdio.h>

int main(){
  long myInt = (long) printf ;

  ((int (*)())myInt)("Hello World!...\n");
}
```

Output :



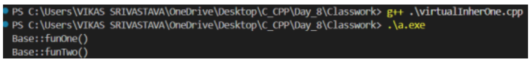
```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> gcc .\typeCastingOne.c
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> .\a.exe
Hello World!...
```

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.



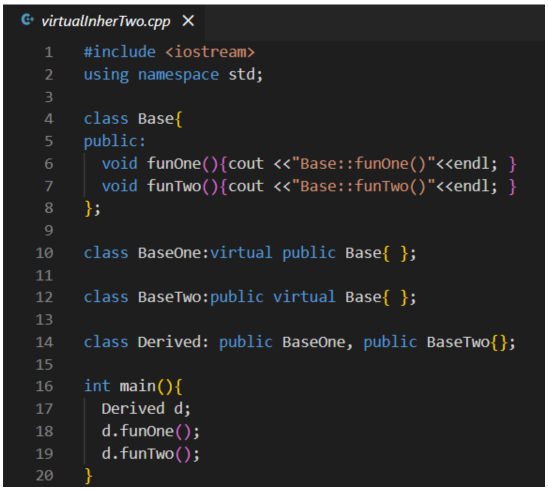
```cpp
#include <iostream>
using namespace std;

class Base{
public:
  void funOne(){cout <<"Base::funOne()"<<endl; }
  void funTwo(){cout <<"Base::funTwo()"<<endl; }
};

class BaseOne:public Base{ };

class BaseTwo:public Base{ };

class Derived: public BaseOne, public BaseTwo{};

int main(){
  Derived d;
  d.BaseOne::funOne();
  d.BaseTwo::funTwo();
}
```

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()
```

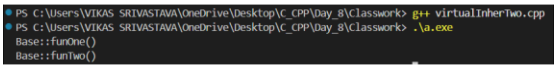
19. ***Code :*** Write a program to demonstrate virtual inheritance in C++, where multiple derived classes inherit from a common base class virtually to avoid duplicate copies of base class members (diamond problem), allowing the final derived class to access base class functions without ambiguity.



virtualInherTwo.cpp

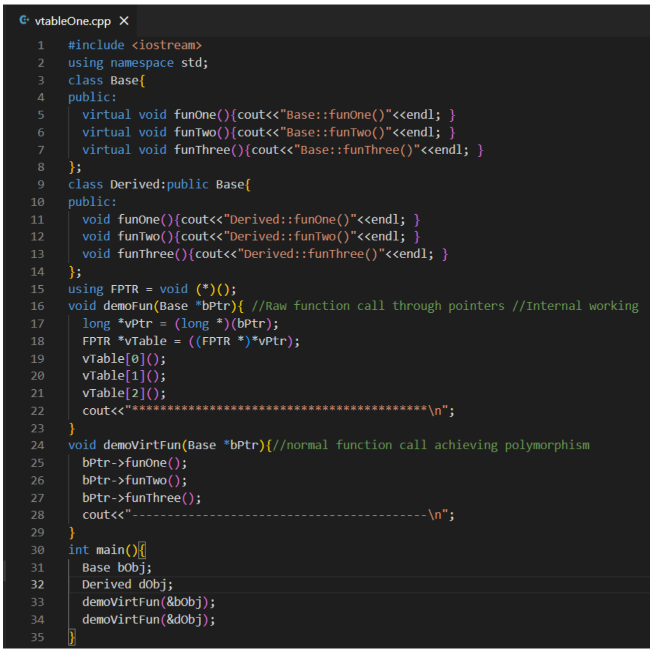
```cpp
#include <iostream>
using namespace std;

class Base{
public:
  void funOne(){cout <<"Base::funOne()"<<endl; }
  void funTwo(){cout <<"Base::funTwo()"<<endl; }
};

class BaseOne:virtual public Base{ };

class BaseTwo:public virtual Base{ };

class Derived: public BaseOne, public BaseTwo{};

int main(){
  Derived d;
  d.funOne();
  d.funTwo();
}
```

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()
```

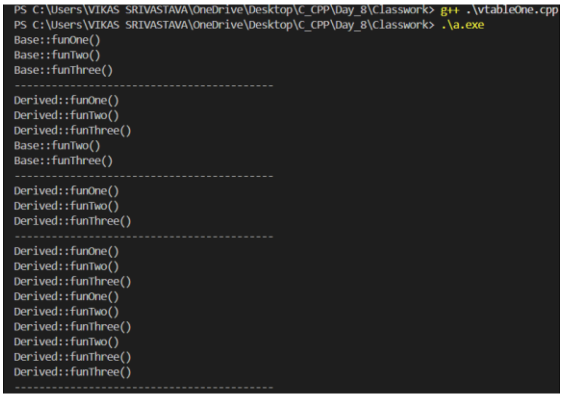
20. ***Code :*** Write a program to demonstrate runtime polymorphism using virtual functions in C++ and explain the internal working of virtual function calls through the virtual table (vTable), showing how function calls are dynamically resolved at runtime based on the actual object type.



vtableOne.cpp

```cpp
#include <iostream>
using namespace std;
class Base{
public:
  virtual void funOne(){cout<<"Base::funOne()"<<endl; }
  virtual void funTwo(){cout<<"Base::funTwo()"<<endl; }
  virtual void funThree(){cout<<"Base::funThree()"<<endl; }
};
class Derived:public Base{
public:
  void funOne(){cout<<"Derived::funOne()"<<endl; }
  void funTwo(){cout<<"Derived::funTwo()"<<endl; }
  void funThree(){cout<<"Derived::funThree()"<<endl; }
};
using FPTR = void (*)();
void demoFun(Base *bPtr){ //Raw function call through pointers //Internal working
  long *vPtr = (long *)(bPtr);
  FPTR *vTable = ((FPTR *)*vPtr);
  vTable[0]();
  vTable[1]();
  vTable[2]();
  cout<<"******************************************\n";
}
void demoVirtFun(Base *bPtr){//normal function call achieving polymorphism
  bPtr->funOne();
  bPtr->funTwo();
  bPtr->funThree();
  cout<<"------------------------------------------\n";
}
int main(){
  Base bObj;
  Derived dObj;
  demoVirtFun(&bObj);
  demoVirtFun(&dObj);
}
```

Output :



```
PS C:\Users\VIKAS SRIVASTAVA\OneDrive\Desktop\C_CPP\Day_8\Classwork> g++ .\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::funThree()
------------------------------------------
```