Principles of Programming Language (CS302)

Assignment - 5

**U19CS012**

1.) Given the following class hierarchy, which **inherited members** can be accessed without qualification from within the **VMI** class? Which requires qualification? Explain your reasoning.

struct Base

{

    void bar(int); *// public by default*

protected:

    int ival;

};

struct Derived1 : virtual public Base

{

    void bar(char); *// public by default*

    void foo(char);

protected:

    char cval;

};

struct Derived2 : virtual public Base

{

    void foo(int); *// public by default*

protected:

    int ival;

    char cval;

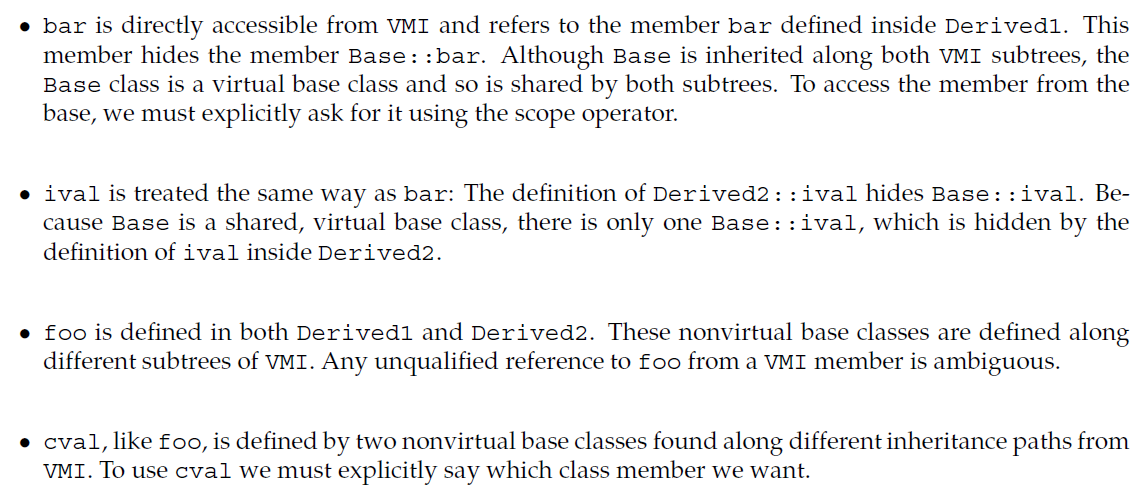
};

class VMI : public Derived1, public Derived2

{

};

VMI class object has members: bar, ival, foo, cval.



2.) Given the following class hierarchy:

class Class

{

    ...

};

class Base : public Class

{

    ...

};

class D1 : virtual public Base

{

    ...

};

class D2 : virtual public Base

{

    ...

};

class MI : public D1, public D2

{

    ...

};

class Final : public MI, public Class

{

    ...

};

(a) In what order are constructors and destructors run on a Final object?

* The following is the order in which a final item is built:

The shared virtual base class is built first, which implies the **Class** and **Base** constructors are executed in that sequence to build the shared **Class** subobject. The **D1 and D2** subobjects, as well as the **MI** subobject, are then created. Then, to indicate Final's direct inheritance from Class, a second, unshared **Class** subobject is created. Finally, the **Final** part is constructed.

**Constructors** run order: **Class 🡪Base 🡪D1 🡪 D2 🡪 MI 🡪Class 🡪 Final.**

* The **Final** object is destroyed first, followed by the (non-virtual) Class sub-object, then the **MI, D2, D1** objects, and finally the shared base class sub-object Base and its base class **Class**.

**Destructors** run order: **Final 🡪 Class 🡪 MI 🡪 D2 🡪 D1 🡪 Base 🡪 Class.**

(b) A Final object has how many Base parts? How many Class parts?

There is one (shared) **Base** subobject and two **Class** subobjects in a **Final** object. The **Class** subobjects are the one from which **Final** inherits directly and the **Class** subobject from which the shared **Base** object inherits.

(c) Which of the following assignments is a compile-time error?

Base \*pb;

Class \*pc;

MI \*pmi;

D2 \*pd2;

|  |  |
| --- | --- |
| Assignment Statement | Error/No Error |
| (a) pb = new Class; | **Error**: tries to create a pointer to a derived class from a **base class pointer**. |
| (b) pc = new Final; | **Error**: The conversion from a pointer to **Final** to a pointer to its parent class **Class** is unclear since a **Final** object has two **Class** subobjects. |
| (c) pmi = pb; | **Error**: tries to create a pointer to a derived class from a base class pointer. |
| (d) pd2 = pmi; | **No Error**: A pointer of derived class can be **cast to a pointer of base class**. |

3.) Given the following classes, explain each print function:

class base

{

public:

    string name() { *return* basename; }

    virtual void print(ostream &os) { os << basename; }

private:

    string basename;

};

class derived : public base

{

public:

    void print(ostream &os)

    {

        print(os);

        os << " " << i;

    }

private:

    int i;

};

If there is a problem in this code, how would you fix it?

* The print method in derived **calls** its base-class **print** member to print the derived object's **base::basename**. The call as written, on the other hand, is a virtual call that (repeatedly) calls the print member in the derived.
* In derived, the print function should have been written as:

void print(ostream &os)

    {

        base::print(os);

        os << " " << i;

    }

Explanation of each **print** function:

* The print function in class base prints its string member named **basename**.
* The print function in class **derived** prints **basename** and then prints the member “i” of the derived object.

4.) Given the classes from the previous problem and the following objects, determine which function is called at run time:

*#include* <bits/stdc++.h>

using namespace std;

class base

{

public:

    string name() { *return* basename; }

    virtual void print(ostream &os) { os << basename; }

private:

    string basename;

};

class derived : public base

{

public:

    void print(ostream &os) override

    {

        base::print(os);

        os << " " << i;

    }

private:

    int i;

};

int main()

{

    base bobj;

    derived dobj;

    base \*bp1 = &bobj;

    base \*bp2 = &dobj;

    base &br1 = bobj;

    base &br2 = dobj;

    bobj.print(cout); *// base::print*

    dobj.print(cout); *// derived::print*

    bp1->name(); *// base::name*

    bp2->name(); *// base::name*

    br1.print(cout); *// base::print*

    br2.print(cout); *// derived::print*

*return* 0;

}

|  |  |
| --- | --- |
| Statement | Function Called |
| bobj.print(cout); | base::print |
| dobj.print(cout); | derived::print |
| bp1->name(); | base::name |
| bp2->name(); | base::name |
| br1.print(cout); | base::print |
| br2.print(cout); | derived::print |

(a) bobj.print();

* Since, it is an **object**. Therefore, it will be resolved at Compile time.

(b) dobj.print();

* dobj.print() calls **derived::dobj.print** is an object. So, the call is resolved at compile time.

(c) bp1->name();

* bp1->name() calls **base::name**. The name function is **non-virtual** so this call is resolved at compile time.
* Which **name** function is called is **determined** based on the type of the object, reference or pointer through which the call is made.
* In this case, **bp1** is a pointer to base, which means that the name function defined in class **base** is called.

(d) bp2->name();

* bp2->name() calls **base::name**. The name function is **non-virtual** so this call is resolved at compile time and is based on the **type of the object**, reference or pointer through which the function is called.
* The function is called through **bp2**, which is a pointer to base. The fact that the pointer points to a derived object is **irrelevant**.

(e) br1.print();

* br1.print() calls **base::print**. Because print is virtual and this call is made through a reference, the decision as to which version of print to call is made at runtime and is based on the type of the object to which the reference refers.
* In this case, we know that **br1** refers to a base object and so the call is resolved to **base::print**.

(f) br2.print();

* br2.print() calls **derived::print**. Again, print is virtual and the call is made through a reference and so the call is resolved at runtime. In this case, we know that **br2** refers to a derived object and so the call is resolved to **derived::print.**

**SUBMITTED BY**: U19CS012

BHAGYA VINOD RANA