

# PPL Assignment 5

Name: Himani Verma

Admission No: U19CS075

---

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 { };
```

## Answer:

Qualification is used in derived class while inheriting base class. It is used to set visibility mode of the base class in derived class. Qualification can be private, protected or public. Both private and protected inheritance provides access to protected and public members of inherited class but not private members. Public inheritance allows only public members access.

**Member that can be accessed from within the MI class without qualification is:** Function bar can be accessed in class VMI without qualification. Call to function bar is unambiguous because there is only one bar method in direct base class Derived1 and its scope is public by default.

**Members that require qualification are:** Variables ival and cval require qualifications. Variable cval is present in both base classes Derived1 and Derived2. So directly using them is ambiguous. Also ival is present in Derived2 and Base classes so qualification has to be specified. Function foo also requires qualification. Functions foo are ambiguous as they are in both base classes Derived1 and Derived2 and a char can be type casted to int as well.

```
// direct access from VMI
bar(0); // Derived1::bar

// require qualification
ival = 0; // Derived2::ival
```

```

Base::ival = 2;    // Base::ival

Derived1::foo(0); // Derived1::foo
Derived2::foo(0); // Derived2::foo

Derived1::cval = '\0'; // Derived1::cval
Derived2::cval = '\0'; // Derived2::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 { ... };

```

- In what order are constructors and destructors run on a Final object?
- A Final object has how many Base parts? How many Class parts?
- Which of the following assignments is a compile-time error?

Base \*pb;

Class \*pc;

MI \*pmi;

D2 \*pd2;

(a) pb = new Class;

(b) pc = new Final;

(c) pmi = pb;

(d) pd2 = pmi;

**Answer:**

(a) For Constructors:

```

#include <iostream>
using namespace std;
class Class {
public:
    Class() { cout << "Class()" << endl; }
};
class Base : public Class {
public:
    Base() { cout << "Base()" << endl; }
};
class D1 : public Base {
public:
    D1() { cout << "D1()" << endl; }
};
class D2 : public Base {
public:
    D2() { cout << "D2()" << endl; }
};

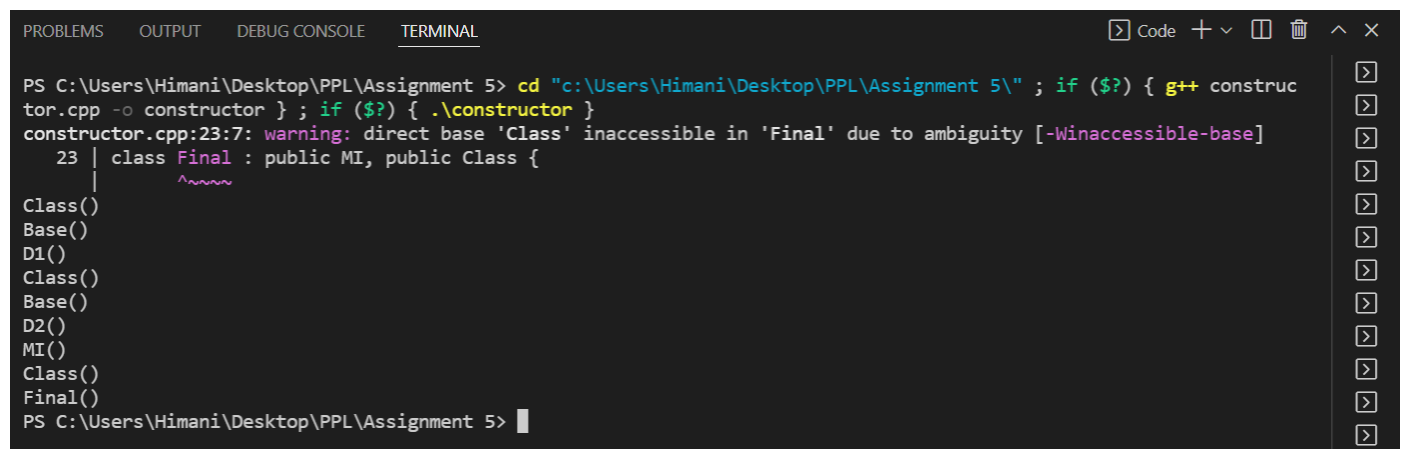
```

```

class MI : public D1, public D2 {
public:
    MI() { cout << "MI()" << endl; }
};
class Final : public MI, public Class {
public:
    Final() { cout << "Final()" << endl; }
};
int main()
{
    Final fi;
    return 0;
}

```

## Output:



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Himani\Desktop\PPL\Assignment 5> cd "c:\Users\Himani\Desktop\PPL\Assignment 5\" ; if ($?) { g++ construc
tor.cpp -o constructor } ; if ($?) { .\constructor }
constructor.cpp:23:7: warning: direct base 'Class' inaccessible in 'Final' due to ambiguity [-Winaccessible-base]
   23 | class Final : public MI, public Class {
      |         ~~~~~
Class()
Base()
D1()
Class()
Base()
D2()
MI()
Class()
Final()
PS C:\Users\Himani\Desktop\PPL\Assignment 5>

```

## For Destructors:

```

#include <iostream>
using namespace std;

class Class {
public:
    ~Class() { cout << "Class()" << endl; }
};

class Base : public Class {
public:
    ~Base() { cout << "Base()" << endl; }
};

class D1 : public Base {
public:
    ~D1() { cout << "D1()" << endl; }
};

class D2 : public Base {
public:
    ~D2() { cout << "D2()" << endl; }
};

class MI : public D1, public D2 {
public:

```

```

        ~MI() { cout << "MI()" << endl; }
};

class Final : public MI, public Class {
public:
    ~Final() { cout << "Final()" << endl; }
};

int main()
{
    Final fi;
    return 0;
}

```

## Output:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\Himani\Desktop\PPL\Assignment 5> cd "c:\Users\Himani\Desktop\PPL\Assignment 5\" ; if ($?) { g++ destruct
or.cpp -o destructor } ; if ($?) { .\destructor }
destructor.cpp:28:7: warning: direct base 'Class' inaccessible in 'Final' due to ambiguity [-Winaccessible-base]
   28 | class Final : public MI, public Class {
      |         ~~~~~
Final()
Class()
MI()
D2()
Base()
Class()
D1()
Base()
Class()
PS C:\Users\Himani\Desktop\PPL\Assignment 5>

```

(b) **Final will have one Base** part since the derived classes D1 and D2 have already defined the base class as virtual. If the base class was not declared virtual it would have resulted in as many subparts of Class and Base in the derived path, but since it is defined virtual subsequently it will pass its own subpart and have one copy in the derived class.

**Final only has two Class parts**, because Base inherits from Class, and Final has more than one indirect base classes that inherit from Base. If Base is not a virtual class, of course Final would have as many subparts of Class as the derivation path. But since Base is a virtual class, which would have only one copy in derived class. As a result, its own subparts will only have one copy in derived classes. Plus, there is a direct base class Class, so Final has two Class subparts in total.

(c)

(a) **pb = new Class;**

A pointer of a class can point to its derived classes but not to its base classes. So it will show the following **error**: *no implicit conversion from base to derived class*.

(b) **pc = new Final;**

Since final consists of two subparts of class Class, the pointer "pc" doesn't know which subpart to point to, hence it will show error.

(c) **pmi = pb;**

A pointer of a class can point to its derived classes but not to its base classes. So it will show the following **error**: *no implicit conversion from base to derived class*.

(d) `pd2 = pmi;`

A pointer of a class can point to its derived classes but not to its base classes. So this one will not show any error as no ambiguity is found.

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

```
class base {
public:
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename; } -----1
private:
    string basename;
};

class derived : public base {
public:
    void print(ostream &os) { print(os); os << " " << i; } -----2
private:
    int i;
};
```

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

**Answer:**

In the above code print function (2) refers to rewriting the print function (1) of the base class as print function of the base class is declared as a virtual function.

The derived virtual function print try to call the base version in its function body, but without the scope operator, the call will be resolved at run time as a call to the derived version itself, resulting in an infinite recursion.

To fix this problem, we need to prefix the scope operator to the call like-  
`void print(ostream& os) override{ base::print(os); os << "" << i;}`

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

`base bobj;`

`base *bp1 = &bobj;`

`base &br1 = bobj;`

`derived dobj;`

base \*bp2 = &dobj; base &br2 = dobj;  
(a) bobj.print(); (b) dobj.print(); (c) bp1->name();  
(d) bp2->name(); (e) br1.print(); (f) br2.print();

**Answer:**

(a) bobj.print();	<b>base::print()</b>
(b) dobj.print();	<b>derived::print()</b>
(c) bp1->name();	<b>base::name()</b>
(d) bp2->name();	<b>derived::name()</b>
(e) br1.print();	<b>base::name()</b>
(f) br2.print();	<b>derived::name()</b>