Question 1

#include<iostream>

using namespace std;

class Base {};

class Derived: public Base {};

int main()

{

    Base \*bp = new Derived;

    Derived \*dp = new Base;

}

A Base class pointer/reference can point/refer to a derived class object, but the other way is not possible.

Question 2

#include<iostream>

using namespace std;

class P {

public:

   void print()  { cout <<" Inside P"; }

};

class Q : public P {

public:

   void print() { cout <<" Inside Q"; }

};

class R: public Q { };

int main(void)

{

  R r;

  r.print();

  return 0;

}

Inside Q

The print function is not present in class R. So it is looked up in the inheritance hierarchy. print() is present in both classes P and Q, which of them should be called? The idea is, if there is multilevel inheritance, then function is linearly searched up in the inheritance hierarchy until a matching function is found.

Question 3

Assume that an integer takes 4 bytes and there is no alignment in following classes, predict the output.

|  |
| --- |
| #include<iostream>  using namespace std;    class base {      int arr[10];  };    class b1: public base { };    class b2: public base { };    class derived: public b1, public b2 {};    int main(void)  {    cout << sizeof(derived);    return 0;  } |

80

Since b1 and b2 both inherit from class base, two copies of class base are there in class derived. This kind of inheritance without virtual causes wastage of space and ambiguities. virtual base classes are used to save space and avoid ambiguities in such cases. For example, following program prints 48. 8 extra bytes are for bookkeeping information stored by the compiler (See this for details)

#include<iostream>

using namespace std;

class base {

int arr[10];

};

class b1: virtual public base { };

class b2: virtual public base { };

class derived: public b1, public b2 {};

int main(void)

{

cout << sizeof(derived);

return 0;

}

Question 4

#include<iostream>

using namespace std;

class Base

{

public:

    void show()

    {

        cout<<" In Base ";

    }

};

class Derived: public Base

{

public:

    int x;

    void show()

    {

        cout<<"In Derived ";

    }

    Derived()

    {

        x = 10;

    }

};

int main(void)

{

    Base \*bp, b;

    Derived d;

    bp = &d;

    bp->show();

    cout << bp->x;

    return 0;

}

Compiler Error in line " cout << bp->x"

**A base class pointer can point to a derived class object, but we can only access base class member or virtual functions using the base class pointer.**

Question 5

#include<iostream>

using namespace std;

class Base

{

public:

    int fun()  { cout << "Base::fun() called"; }

    int fun(int i)  { cout << "Base::fun(int i) called"; }

};

class Derived: public Base

{

public:

    int fun() {  cout << "Derived::fun() called"; }

};

int main()

{

    Derived d;

    d.fun(5);

    return 0;

}

Compiler Error

If a derived class writes its own method, then all functions of base class with same name become hidden, even if signaures of base class functions are different. In the above question, when fun() is rewritten in Derived, it hides both fun() and fun(int) of base class.

|  |
| --- |
| Question 6 |

|  |
| --- |
| #include<iostream>  using namespace std;    class Base {  public:      int fun()          {    cout << "Base::fun() called";     }      int fun(int i)     {   cout << "Base::fun(int i) called";  }  };    class Derived: public Base  {  public:      int fun()   {     cout << "Derived::fun() called";   }  };    int main()  {      Derived d;      d.Base::fun(5);      return 0;  } |

Base::fun(int i) called

We can access base class functions using scope resolution operator even if they are made hidden by a derived class function.

|  |
| --- |
| Question 10 |

Output of following program?

|  |
| --- |
| #include <iostream>  #include<string>  using namespace std;    class Base  {  public:      virtual string print() const      {          return "This is Base class";      }  };    class Derived : public Base  {  public:      virtual string print() const      {          return "This is Derived class";      }  };    void describe(Base p)  {      cout << p.print() << endl;  }    int main()  {      Base b;      Derived d;      describe(b);      describe(d);      return 0;  } |

This is Base class

This is Base class

Note that an object of Derived is passed in describe(d), but print of Base is called. **The describe function accepts a parameter of Base type**. This is a typical example of **object slicing**, when we assign an object of derived class to an object of base type, the derived class object is sliced off and all the data members inherited from base class are copied. Object slicing should be avoided as there may be surprising results like above. As a side note, object slicing is not possible in Java. In Java, every non-primitive variable is actually a reference.

|  |
| --- |
| Question 11 |

|  |
| --- |
| #include<iostream>  using namespace std;    class Base  {  public :      int x, y;  public:      Base(int i, int j){ x = i; y = j; }  };    class Derived : public Base  {  public:      Derived(int i, int j):x(i), y(j) {}  //Derived(int i, int j):Base(i,j) {}      void print() {cout << x <<" "<< y; }  };    int main(void)  {      Derived q(10, 10);      q.print();      return 0;  } |
|  |
|  |

Compiler Error

The base class members cannot be directly assigned using [initializer list](http://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/). We should call the base class constructor in order to initialize base class members. Following is error free program and prints "10 10"

#include

using namespace std;

class Base {

public : int x, y;

public: Base(int i, int j){ x = i; y = j; }

};

class Derived : public Base {

public:

Derived(int i, int j):Base(i, j) {} ;

void print() {cout << x <<" "<< y; }

};

int main(void) {

Derived q(10, 10);

q.print();

return 0;

}

Question 12

#include<iostream>

using namespace std;

class Base

{

protected:

    int a;

public:

    Base() {a = 0;}

};

class Derived1:  public Base

{

public:

    int c;

};

class Derived2:  public Base

{

public:

    int c;

};

class DerivedDerived: public Derived1, public Derived2

{

public:

    void show()  {   cout << a;  }

};

int main(void)

{

    DerivedDerived d;

    d.show();

    return 0;

}

Compiler Error in Line "cout << a;"

This is a typical example of [diamond problem of multiple inheritance](http://www.geeksforgeeks.org/multiple-inheritance-in-c/). Here the base class member 'a' is inherited through both Derived1 and Derived2. So there are two copies of 'a' in DerivedDerived which makes the statement "cout << a;" ambiguous. The solution in C++ is to use virtual base classes.

#include<iostream>

using namespace std;

class Base

{

protected:

    int a;

public:

    Base() {a = 0;}

};

class Derived1: virtual public Base

{

public:

    int c;

};

class Derived2: virtual public Base

{

public:

    int c;

};

class DerivedDerived: public Derived1, public Derived2

{

public:

    void show()  {   cout << a;  }

};

int main(void)

{

    DerivedDerived d;

    d.show();

    return 0;

}

|  |
| --- |
| Question 14 |

Consider the below C++ program.

|  |
| --- |
| #include<iostream>  using namespace std;  class A  {  public:       A(){ cout <<"1";}       A(const A &obj){ cout <<"2";}  };    class B: virtual A  {  public:      B(){cout <<"3";}      B(const B & obj){cout<<"4";}  };    class C: virtual A  {  public:     C(){cout<<"5";}     C(const C & obj){cout <<"6";}  };    class D:B,C  {  public:      D(){cout<<"7";}      D(const D & obj){cout <<"8";}  };    int main()  {     D d1;     D d(d1);  } |

Which of the below is not printed?

**2**

**4**

**6**

**All of the above**

Output will be 13571358 as 1357 (for D d1) and as 1358 (for D d(d1))......reason is that ......during inheritance we need to explicitly call copy constructor of base class otherwise only default constructor of base class is called. One more thing, as we are using virtual before base class, there will be only one copy of base class in multiple inheritance. And without virtual output will be......13157....&...13158 as (1315713158) respectively for each derived class object.

Question

Which of the following is true about pure virtual functions?

1) Their implementation is not provided in a class where they are declared.

2) If a class has a pure virtual function, then the class becomes abstract class and an instance of this class cannot be created.

Pure virtual functions (when we set = 0) can also have a function body.

What is the use to provide a function body for pure virtual functions, if they are not going get called at all?

Your assumption that pure virtual function cannot be called is absolutely incorrect. When a function is declared pure virtual, it simply means that this function cannot get called dynamically, through a virtual dispatch mechanism. Yet, this very same function can easily be called statically, non-virtually, directly (without virtual dispatch).

In C++ language a non-virtual call to a virtual function is performed when a qualified name of the function is used in the call, i.e. when the function name specified in the call has the <class name>::<function name> form.

For example

struct S

{

virtual void foo() = 0;

};

void S::foo()

{

// body for pure virtual function `S::foo`

}

struct D : S

{

void foo()

{

S::foo();

// Non-virtual call to `S::foo` from derived class

this->S::foo();

// Alternative syntax to perform the same non-virtual call

// to `S::foo` from derived class

}

};

int main()

{

D d;

d.S::foo();

// Another non-virtual call to `S::foo`

}

#include<iostream>

using namespace std;

class A

{

public:

virtual void Foo() = 0;

};

void A::Foo(){

cout<<"I am pure virtual function in class A."<<endl;

}

class B: public A

{

public:

void Foo(){

cout<<"I am implementation of pure virtual function Foo() in derived class B."<<endl;

A::Foo();

this->A::Foo();

}

};

int main()

{

B b;

b.Foo();

cout<<"Now calling from Main"<< endl;

b.A::Foo();

}

|  |
| --- |
| Question 5 |

|  |  |
| --- | --- |
| #include<iostream>  using namespace std;    class Base  {  public:      virtual void show() = 0;  };    int main(void)  {      Base b;      Base \*bp;      return 0;  } | |
| A | There are compiler errors in lines "Base b;" and "Base bp;" | |
| B | There is compiler error in line "Base b;" | |
| C | There is compiler error in line "Base bp;" | |
| D | No compiler Error | |

Explanation:

Since Base has a pure virtual function, it becomes an abstract class and an instance of it cannot be created. So there is an error in line "Base b". Note that there is no error in line "Base \*bp;". We can have pointers or references of abstract classes.

|  |
| --- |
| Question 6 |

Predict the output of following program.

|  |  |
| --- | --- |
| #include<iostream>  using namespace std;  class Base  {  public:      virtual void show() = 0;  };    class Derived : public Base { };    int main(void)  {      Derived q;      return 0;  } | |
| A | Compiler Error: there cannot be an empty derived class | |
| B | Compiler Error: Derived is abstract | |
| C | No compiler Error | |

Explanation:

If we don't override the pure virtual function in derived class, then derived class also becomes abstract class.

|  |
| --- |
| Question 9 |

Can a destructor be virtual? Will the following program compile?

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  class Base {  public:    virtual ~Base() {}  };  int main() {     return 0;  } | |
| A | Yes |
| B | No |

Explanation:

A destructor can be virtual. We may want to call appropriate destructor when a base class pointer points to a derived class object and we delete the object. If destructor is not virtual, then only the base class destructor may be called. For example, consider the following program.

// Not good code as destructor is not virtual

#include<iostream>

using namespace std;

class Base {

public:

Base() { cout << "Constructor: Base" << endl; }

~Base() { cout << "Destructor : Base" << endl; }

};

class Derived: public Base {

public:

Derived() { cout << "Constructor: Derived" << endl; }

~Derived() { cout << "Destructor : Derived" << endl; }

};

int main() {

Base \*Var = new Derived();

delete Var;

return 0;

}

Output on GCC:

Constructor: Base

Constructor: Derived

Destructor : Base

|  |
| --- |
| Question 10 |

|  |  |
| --- | --- |
| #include<iostream>  using namespace std;  class Base  {  public:      Base()    { cout<<"Constructor: Base"<<endl; }      virtual ~Base()   { cout<<"Destructor : Base"<<endl; }  };  class Derived: public Base {  public:      Derived()   { cout<<"Constructor: Derived"<<endl; }      ~Derived()  { cout<<"Destructor : Derived"<<endl; }  };  int main()  {      Base \*Var = new Derived();      delete Var;      return 0;  } | |
| A | Constructor: Base  Constructor: Derived  Destructor : Derived  Destructor : Base |
| B | Constructor: Base  Constructor: Derived  Destructor : Base |
| C | Constructor: Base  Constructor: Derived  Destructor : Derived |
| D | Constructor: Derived  Destructor : Derived |

Explanation:

Since the destructor is vitrual, the derived class destructor is called which in turn calls base class destructor.