

Inheritance and Exceptions

Short Quiz

VF1

Which of the following is true about virtual functions in C++.

1. Virtual functions are functions that can be overridden in derived class with the same signature.
2. Virtual functions enable run-time polymorphism in a inheritance hierarchy.
3. If a function is 'virtual' in the base class, the most-derived class's implementation of the function is called according to the actual type of the object referred to, regardless of the declared type of the pointer or reference. In non-virtual functions, the functions are called according to the type of reference or pointer.
4. All of the above.



ALL OF THE ABOVE

VF

```
class Base
{
public:
    virtual void show() { cout<<" In Base \n"; }
};
```

```
class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};
```

```
int main(void)
{
    Base *bp, b;
    Derived d;
    bp = &d;
    bp->show();
    bp = &b;
    bp->show();
    return 0;
}
```

In Derived
In Base

Initially base pointer points to a derived class object. Later it points to a base class object.

VF

```
class Base
{
public:
    virtual void show() { cout<<" In Base \n"; }
};
```

```
class Derived: public Base
{
public:
    void show() { cout<<"In Derived \n"; }
};
```

```
int main(void)
{
    Base *bp = new Derived;
    bp->show();

    Base &br = *bp;
    br.show();

    return 0;
}
```

In Derived
In Derived

Since show() is virtual in base class, it is called according to the type of object being referred or pointed, rather than the type of pointer or reference.

Where will the compiler complain?

```
1 class Base {  
2 public:  
3   virtual void show() = 0;  
4 };
```

```
5 int main(void) {  
6   Base* bp;  
7   Base b;  
8   return 0;  
9 }
```

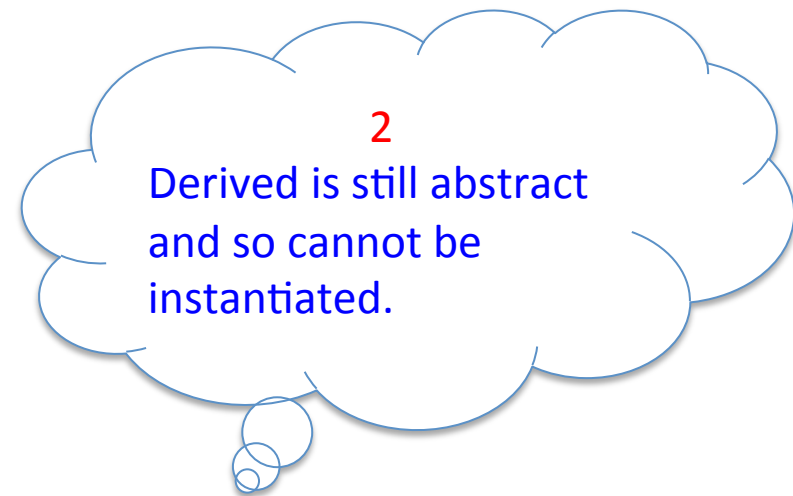
Line 7.

Note that there is no error in line "Base *bp;". We can have pointers or references of abstract classes.

```
class Base
{
public:
    virtual void show() = 0;
};

class Derived : public Base { };

int main(void)
{
    Derived q;
    return 0;
}
```



1. Compiler Error: there cannot be an empty derived class
2. Compiler Error: Derived is abstract
3. No compiler Error

```
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;
}
```

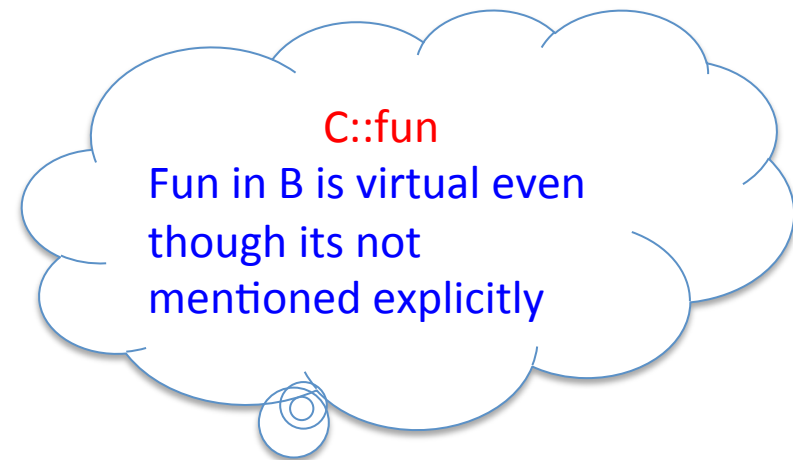


```
class A
{
public:
    virtual void fun() { cout << "A::fun() "; }
};
```

```
class B: public A
{
public:
    void fun() { cout << "B::fun() "; }
};
```

```
class C: public B
{
public:
    void fun() { cout << "C::fun() "; }
};
```

```
int main()
{
    B *bp = new C;
    bp->fun();
    return 0;
}
```



Exception handling

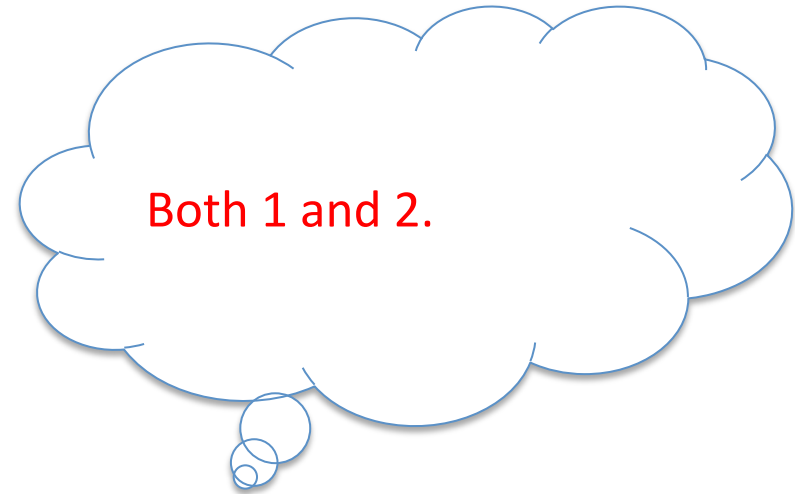
```
int main()
{
    int x = -1;
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "After catch \n";
    return 0;
}
```



What should be put in a try block?

1. Statements that might cause exceptions
2. Statements that should be skipped in case of an exception

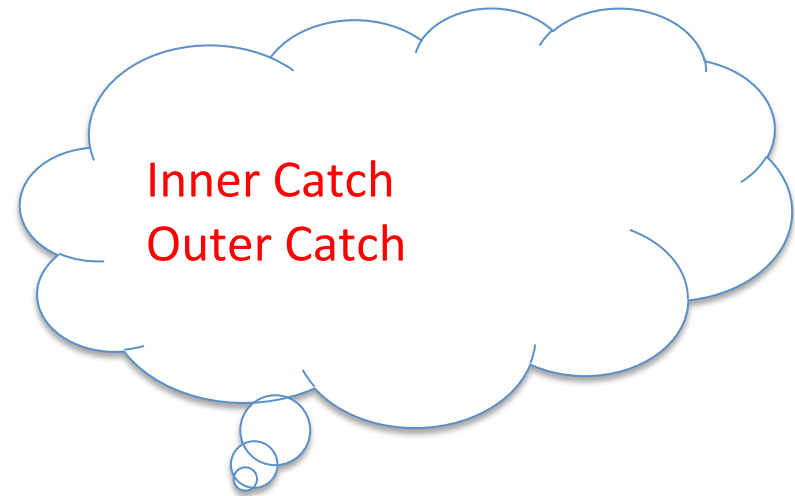


```
class Base {};  
class Derived: public Base {};  
int main()  
{  
    Derived d;  
    try {  
        throw d;  
    }  
    catch(Base b) {  
        cout<<"Caught Base Exception";  
    }  
    catch(Derived d) {  
        cout<<"Caught Derived Exception";  
    }  
    return 0;  
}
```



Caught Base Exception
First exception caught.

```
int main()
{
    try
    {
        try
        {
            throw 20;
        }
        catch (int n)
        {
            cout << "Inner Catch\n";
            throw;
        }
    }
    catch (int x)
    {
        cout << "Outer Catch\n";
    }
    return 0;
}
```



```
class Test {  
public:  
    Test() { cout << "Constructing Test " << endl; }  
    ~Test() { cout << "Destroying Test " << endl; }  
};
```

```
int main() {  
    try {  
        Test t1;  
        throw 10;  
    } catch(int i) {  
        cout << "Caught " << i << endl;  
    }  
}
```

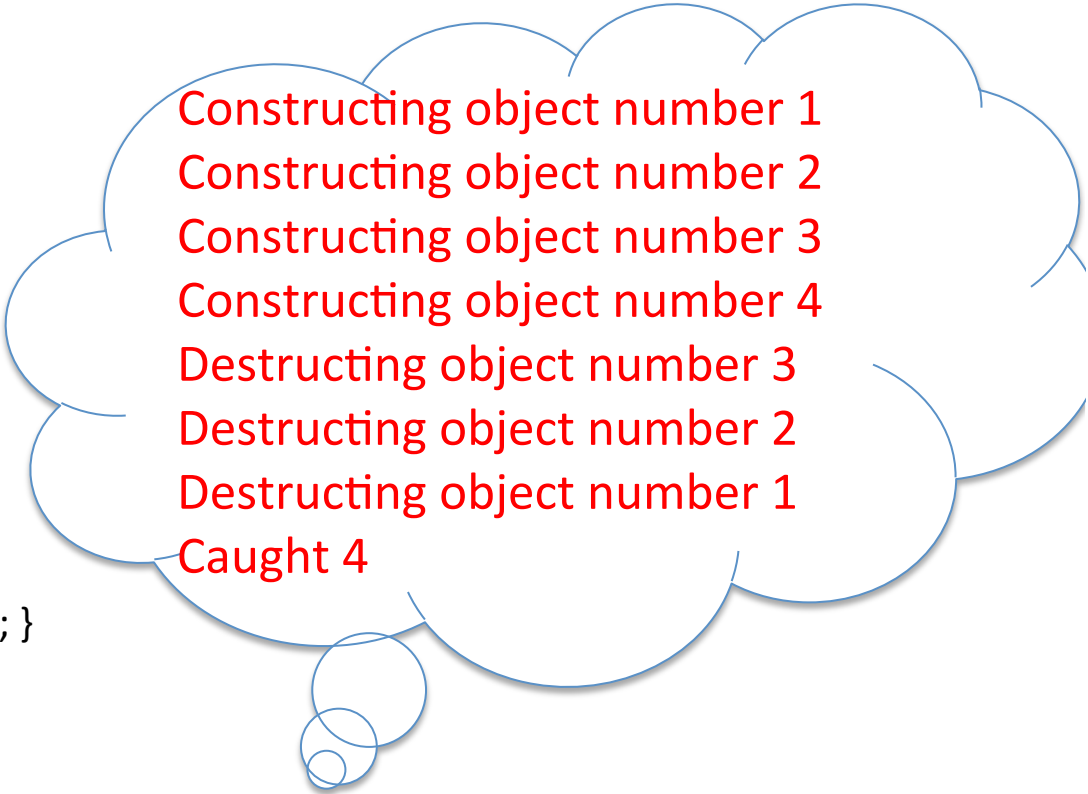


Constructing Test
Destroying Test
Caught 10

```
class Test {
    static int count;
    int id;
public:
    Test() {
        count++;
        id = count;
        cout << "Constructing " << id << endl;
        if(id == 4)
            throw 4;
    }
    ~Test() { cout << "Destroying " << id << endl; }
};
```

```
int Test::count = 0;
```

```
int main() {
    try {
        Test array[5];
    } catch(int i) {
        cout << "Caught " << i << endl;
    }
}
```



Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4

The destructors are called in reverse order of constructors.

Also, after the try block, the destructors are called only for completely constructed objects.