

📑 题目列表 (/course/46/contest/566/home)

📊 提交状态 (/course/46/contest/566/submissions)

🏆 排行榜 (/course/46/contest/566/ranklist/normal)

## 选择题

刷新 ↺

### 选择题

1. 【单选】关于以下代码，说法正确的是

```
class Base {
    int x;
public:
    virtual void f1(){}
    virtual void f2() final {}
    void g(int){}
    void g(float){}
    void g(){}
};

class Derive: public Base {
    int y;
public:
    void f1(){}
    void g(){}
};
```

- A. Derive::g() 是 Base::g(int) 和 Base::g(float) 的重载函数
- B. Derive::g() 重写隐藏 Base::g(int) 和 Base::g(float)
- C. Derive::f1() 重写覆盖 Base::f1() ; Derive::g() 重写覆盖 Base::g()
- D. Base::f2() 由于被 final 关键字修饰而不能被重写覆盖，但 Derive 类中仍然可以定义成员函数 void f2()，并重写隐藏 Base::f2()

2. 【单选】关于以下代码，说法正确的是

```

#include <iostream>
using namespace std;

class Instrument{
public:
    int price;
    void tune() { cout << "Instrument::tune()" << endl; }
    virtual void play() { cout << "Instrument::play()" << endl; }
};

class Violin : public Instrument {
public:
    void tune() { cout << "Violin::tune()" << endl; } // (1)
};

void tune(Instrument *p){
    p->tune();
}

int main(){
    Violin obj;
    tune(&obj); // (2)
    return 0;
}

```

- A. (1) 处是重写隐藏，在 (1) 处加上 `override` 关键字变成 `void tune() override {}` 后可以实现重写覆盖
- B. `Instrument` 类的 `tune` 函数继承给了 `Violin` 类，(1) 处又一次定义了 `tune` 函数，这是函数重载
- C. (2) 处函数调用将输出 `Instrument::tune()`
- D. 一个 `Instrument` 对象包含一个 `int` 成员，因此占用 4 个字节

3. 【单选】以下代码的输出是

```

#include <iostream>

struct A {
    A() { foo(); }
    virtual ~A() { foo(); }
    virtual void foo() { std::cout << "1"; }
    void bar() { foo(); }
};

struct B : public A {
    virtual void foo() { std::cout << "2"; }
};

int main() {
    B b;
    b.bar();
}

```

- A. 111

B. 121

C. 122

D. 编译错误

4. 【多选】下列说法错误的有

```
#include <iostream>
using namespace std;
class Base
{
public:
    virtual void foo() {}
    virtual void foo(int) {}
    void bar() {}
};

class Derived1 : public Base
{
public:
    void foo() {}
    void foo(int) {}
};

class Derived2 : public Base
{
public:
    void foo(float) {} // (a)
};

int main()
{
    Derived1 d1;
    Derived2 d2;
    // d1.bar(); // (1)
    d2.foo(3.0); // (2)

    Base *pb1 = &d1;
    Base *pb2 = &d2;
    pb1->foo(3); // (3)
    pb2->foo(3.0); // (4)

    Base b1 = d1;
    b1.foo(); // (5)

    Derived2 *pd2 = &d2;
    pd2->foo(3.0); // (6)
    return 0;
}
```

A. (1)处代码存在编译错误，不能将其取消注释

B. (2)(6)处调用的是 `Derived2::foo(float)`

C. (3)处调用的是 `Derived1::foo(int)`

D. (4)处调用的是 `Base::foo(int)`，若想让(4)处调用变成 `Derived2::foo(float)`，则可以把(a)行改为 `virtual void foo(float) {}`

E. (5)处调用的是 `Base::foo()`

5. 【多选】关于以下代码，说法正确的有

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

class Base {
    int* x;
public:
    Base(){x = new int[10];}
    Base(const Base& other){fn();} // (1)
    virtual void fn(){}
    virtual void g1(){}
    virtual Base& g2(){}
    ~Base(){delete [] x;}
};

class Derive: public Base {
    int* y;
public:
    Derive(){y = new int[10];}
    Derive(const Derive& other):Base(other){fn();}
    void fn(){}
    void g1() const {}
    Derive& g2(){}
    ~Derive(){delete [] y;}
};

void fn(){
    Base* b = new Derive();
    delete b; // (*)
}

int main(){
    fn();
    return 0;
}
```

A. (1)处调用的是 `Base::fn()`，这是因为虚机制在拷贝构造函数中不起作用

B. `Derive::g1()` 能重写覆盖 `Base::g1()`

C. `Derive::g2()` 能重写覆盖 `Base::g2()`

D. 由于 `Base` 类析构函数不是虚函数，(\*)处析构 `b` 指向的 `Derive` 类对象时不会调用 `Derive` 类的析构函数，故存在内存泄漏问题

6. 【多选】关于以下代码，说法正确的是

```

#include <iostream>
using namespace std;

template<typename T>
T add(T a, T b){
    T c = a + b;
    return c;
}

template<typename T>
T subtract(T a, T b){
    T c = a - b;
    return c;
}

int main(){
    int a = add(1.2, 1.3); // (1)
    int b = subtract(add(1, 2), add(2.1, 3.2)); //(2)
    int c = add(1, (int)add(1.2, 2.3)); // (3)
    int d = add<int>(1, 1.2); // (4)
    return 0;
}

```

- A. (1) 处将出现编译错误，因为 add 函数在(1)处的返回值类型为浮点数
- B. (2) 处将出现编译错误，因为编译时无法确定 subtract 中 T 的具体类型
- C. (3) 处调用了模板函数 add 的两个实例化版本，T 分别为整数和浮点数
- D. (4) 处的返回值为2，第二个参数1.2被强制类型转换成了 int 类型

7. 【单选】关于以下代码，说法正确的是

```

#include <iostream>
using namespace std;

// (1)
template <typename T0>
class A
{
    T0 value;

public:
    void set(const T0 &v)
    {
        value = v;
    }
    T0 get();
};

// (2)
template <typename T1>
T1 A<T1>::get() { return value; }

template <typename T0, typename T1>
T1 sum(T0 a, T0 b)
{
    return T1(a.get() + b.get());
}

int main()
{
    A<double> a;
    a.set(4.3);
    cout << a.get() << endl;
    cout << sum<A<double>, int>(a, a) << endl;
    // (3)
    return 0;
}

```

- A. (2)处 A::get() 的定义采用的类型名 T1 与(1)处类A采用的类型名 T0 不同，因此编译错误
- B. 程序的执行结果为 4.3\n8\n
- C. 根据函数模板 sum 的定义，实参类型需要定义成员函数 get，且 sum 的返回值类型必定与参数类型不同
- D. 如果(3)处添加代码 sum(9, 2); 程序依然能正常编译运行

8. 【多选】关于抽象类，以下说法正确的有

- A. 抽象类必定存在一个成员函数没有函数体
- B. 如果在一个抽象类的派生类中将该抽象基类的所有纯虚函数都重写覆盖，则该派生类不再是抽象类
- C. 当尝试将派生类对象向上转换为抽象类对象时，会产生编译错误，这是因为抽象类不允许定义对象
- D. 抽象类不允许有数据成员

## 语言和编译选项

#	名称	编译器	额外参数	代码长度限制
0	answer	cp		1048576 B

## 递交历史

#	状态	时间

## 递交答案

选择文件

未选择任何文件

添加

提交