

1. Inline functions are avoided when _____

- A. function contains static variables
- B. function have recursive calls
- C. function have loops
- D. all of the mentioned

答案：D

- 内联函数：通过将函数体直接嵌入到调用点，而不是通过函数调用来执行，从而减少函数调用的开销。
- 优点：
 - 减少函数调用的开销（如栈操作、寄存器保存等）。
 - 提高程序运行效率。
- 缺点：
 - 可能增加代码大小（由于函数体被复制到每个调用点）。
 - 不适用于复杂或大型函数，因为会导致代码膨胀。

A. Function contains static variables

- 静态变量（static variables）：在函数内部声明为 `static` 的变量具有持久性，其值在函数调用之间保持不变。
- 问题：当函数被内联时，每次调用该函数时都会生成一个新的内联副本。如果函数中含有静态变量，那么每个内联副本都会有自己的静态变量实例，这会导致行为不一致，不符合预期。
- 结论：含有静态变量的函数不适合内联。

B. Function have recursive calls

- 递归调用（recursive calls）：函数在自身内部调用自己。
- 问题：内联函数在编译时会被展开为代码片段，而递归调用需要动态地管理调用栈。如果一个函数是递归的，将其内联会导致无限展开，无法正确处理递归逻辑。
- 结论：含有递归调用的函数不适合内联。

C. Function have loops

- 循环（loops）：函数中包含 `for`、`while` 或 `do-while` 等循环结构。
- 问题：虽然循环本身不会直接阻止函数内联，但循环可能会导致函数变得较大或复杂。如果循环体内有大量代码，内联后会导致代码膨胀，降低性能。
- 结论：含有循环的函数可能不适合内联，尤其是当循环体较大或复杂时。

3. Pick out the correct statement.

- A. A friend function must be a member of another class
- B. A friend function cannot be a member of another class
- C. A friend function may or may not be a member of another class
- D. None of the mentioned

答案：C

- 友元函数：是一种特殊的函数，它可以访问类的私有（private）和保护（protected）成员，即使它不是该类的成员函数。
- 定义方式：
 - 在类的内部声明为 `friend`，例如：

```
cpp
1 class MyClass {
2     int data;
3 public:
4     friend void myFriendFunction(MyClass& obj);
5 };
```

- 友元函数可以是独立的全局函数，也可以是另一个类的成员函数。
- 特点：
 - 友元函数不是类的成员函数，因此没有隐式的 `this` 指针。
 - 它可以通过类的私有成员直接操作对象。

4. When a copy constructor is called?

- A. When an object of the class is returned by value
- B. When an object of the class is passed by value to a function
- C. When an object is constructed based on another object of the same class
- D. All of the mentioned

答案：D

1. 当一个对象被另一个对象初始化时：例如，通过另一个同类型的对象来创建新对象。

```
cpp
1 MyClass obj1; // 创建对象 obj1
2 MyClass obj2(obj1); // 调用拷贝构造函数，用 obj1 初始化 obj2
```

2. 当一个对象作为值传递给函数时：将对象作为参数传递给函数时，会创建该对象的一个副本，此时调用拷贝构造函数。

```
cpp
1 void func(MyClass obj) { /* ... */ } // 按值传递对象时调用拷贝构造函数
2 MyClass obj1;
3 func(obj1); // 调用拷贝构造函数
```

3. 当一个对象作为返回值返回时：从函数返回一个对象时，需要创建该对象的副本，此时调用拷贝构造函数。

```
cpp
1 MyClass func() {
2     MyClass obj;
3     return obj; // 返回对象时调用拷贝构造函数
4 }
```

```
Mammal *M = new Mammal();
Male m;
Female f;
*M = m;
M->Define();
M = &m;
M->Define();
return 0;
```

(4) `*M = m;`

- 这里使用了赋值操作符将 `Male` 对象 `m` 赋值给 `Mammal` 对象 `*M`。
- 关键点：`Mammal` 类中没有显式定义赋值操作符，因此会使用默认的赋值操作符。默认的赋值操作符只会浅拷贝基类部分的数据，不会涉及派生类的部分。
- 因此，`*M` 只是被赋值为一个 `Mammal` 对象，其行为仍然遵循 `Mammal` 类的定义。

(6) `M = &m;`

- 将指针 `M` 重新指向 `Male` 对象 `m`。
- 此时，`M` 的实际类型是 `Male`，但它的声明类型仍然是 `Mammal*`。由于 `Define()` 是虚函数，调用时会发生动态绑定 (runtime polymorphism)。

(7) `M->Define();`

- 此时，`M` 指向的是一个 `Male` 对象，虽然它的声明类型是 `Mammal*`，但由于 `Define()` 是虚函数，调用时会根据实际对象的类型 (即 `Male`) 来决定调用哪个版本的 `Define()`。

11. The static data member _____

A. Can be accessed directly

B. Can be accessed with any public class name

C. Can only be accessed with dot operator

D. Can be accessed using class name if not using static member function

答案：D

- 静态数据成员：
 - 是类的一个共享成员，属于整个类而不是某个特定的对象。
 - 在内存中只有一份实例，所有该类的对象共享同一个静态数据成员。
 - 静态数据成员可以是任何类型，包括基本类型、类类型等。
 - 静态数据成员必须在类外部进行定义和初始化。
- 访问方式：
 - 静态数据成员可以通过以下两种方式访问：
 1. 通过类名直接访问：`ClassName::static_member_name`
 2. 通过对象访问：`object.static_member_name` 或 `this->static_member_name` (在类的成员函数中)
 - 如果不使用静态成员函数，可以直接通过类名访问静态数据成员。

D. Can be accessed using class name if not using static member function

- 含义：如果不使用静态成员函数，静态数据成员可以通过类名访问。
- 分析：这是正确的。静态数据成员可以通过类名直接访问，例如：

```
cpp
1  ClassName::static_member_name;
```

这种访问方式不需要依赖对象或静态成员函数。

静态成员函数是属于整个类 而不是类的某个对象的函数。它可以直接访问类中的其他静态成员（包括静态数据成员和其他静态成员函数），但不能直接访问非静态成员变量或非静态成员函数。

✅ 语法示例

```
cpp
1 class MyClass {
2 private:
3     static int count; // 静态数据成员
4     int value;        // 普通数据成员
5
6 public:
7     MyClass() { ++count; }
8
9     static void showCount() { // 静态成员函数
10         cout << "Count: " << count << endl;
11         // 下面这行会报错：无法在静态函数中访问非静态成员
12         // cout << value; // 错误!
13     }
14 };
```

特性	说明
属于类，不属于对象	可以通过类名直接调用，如 <code>MyClass::showCount();</code>
没有 this 指针	因为不绑定到任何具体对象
只能访问静态成员	不能访问普通成员变量或普通成员函数
可以被重载	和普通函数一样，支持重载
不能是虚函数	因为虚函数机制依赖于对象（this 指针）

1. 通过类名访问（推荐）：

```
cpp
1 MyClass::showCount();
```

2. 通过对象访问：

```
cpp
1 MyClass obj;
2 obj.showCount();
```

❌ 为什么不能访问非静态成员？

因为静态成员函数没有 `this` 指针，它不知道应该访问哪个对象的数据成员。也就是说，它不知道要操作的是哪一个对象的 `value`、`name` 等属性。

13. Which among the following is true for default arguments?

- A. They are only allowed in the return type of the function declaration.
- B. They are only allowed in the parameter list of the function declaration.
- C. They are only allowed with the class name definition.
- D. They are only allowed with the integer type values.

答案: B

Void func(int a =10, int b =5)

15. Pick the correct statement about references in C++

- A. References stores the address of variables
- B. References and variables both have the same address
- C. References use dereferencing operator(*) to access the value of variable its referencing
- D. References were also available in C

答案: B

- 分析：这是正确的。引用是变量的别名，因此引用和被引用的变量共享相同的内存地址。例如：

```
cpp
1 int x = 10;
2 int& ref = x; // ref 是 x 的引用
```

```
Base* b = new Derived(); // (1)
Derived* d = dynamic_cast<Derived*>(b); // (2)
```

(1) `Base* b = new Derived();`

- 作用：创建一个 `Derived` 类的对象，并将其地址赋值给基类指针 `b`。
- 多态性：由于 `Derived` 是从 `Base` 派生的，因此可以将 `Derived` 的对象指针赋值给 `Base` 类型的指针。这是多态的基础，允许通过基类指针操作派生类对象。
- 内存分配：使用 `new` 动态分配了一个 `Derived` 对象。

(2) `Derived* d = dynamic_cast<Derived*>(b);`

- 作用：使用 `dynamic_cast` 将基类指针 `b` 转换为派生类指针 `d`。
- 动态类型转换 (`dynamic_cast`)：
 - `dynamic_cast` 是 C++ 中的一种运行时类型转换机制，用于在继承层次中进行向上或向下转换。
 - 如果转换成功，返回指向派生类的指针；如果转换失败（例如，`b` 不是指向 `Derived` 类型的对象），则返回 `nullptr`。
- 场景：这里 `b` 确实指向一个 `Derived` 对象，因此转换会成功。

✓ static_cast

- 在编译时 完成转换。
- 不进行运行时类型检查，因此不安全（如果转换错误，可能导致未定义行为）。
- 更快，但需要程序员确保转换是合法的。

示例：

cpp

```
1 double d = 3.14;
2 int i = static_cast<int>(d); // 正确: double -> int
3
4 Base* b = new Derived();
5 Derived* d = static_cast<Derived*>(b); // 即使 b 实际指向的是 Derived, 也允许
```

✓ dynamic_cast

- 在运行时 完成转换。
- 只用于具有虚函数表（即有多态）的类体系中。
- 如果转换失败，返回 `nullptr`（对指针）或抛出异常（对引用）。
- 更安全，但性能略低。

5. Which of the following operator cannot be used to overload when that function is declared as friend function?

A. `[]`

B. `==`

C. `-=`

✓ `[]`

1. 大多数运算符可以重载：

- C++ 允许重载大多数运算符，包括算术运算符（如 `+`、`-`）、关系运算符（如 `<`、`>`）、逻辑运算符（如 `&&`、`||`）等。
- 但有些运算符不能被重载，例如成员访问运算符 `.` 和作用域解析运算符 `::`。

2. 运算符重载的方式：

- 可以作为类的成员函数重载。
- 也可以作为非成员函数（通常声明为友元函数）重载。

3. 友元函数的作用：

- 友元函数可以访问类的私有成员。
- 当运算符涉及多个操作数且其中一个不是该类的对象时，通常需要将运算符重载为友元函数。

A. []

- 含义：下标运算符 []。
- 分析：
 - 下标运算符 [] 是一种特殊的运算符，用于数组或容器的索引访问。
 - 它必须被重载为类的成员函数，而不能作为友元函数重载。
 - 原因是 [] 的语法要求第一个操作数必须是对象本身，因此它只能作为成员函数重载。
- 结论：不能被重载为友元函数。

6. If a class have default constructor defined in private access, and one parameter constructor in protected mode, how will it be possible to create instance of object?

A. Directly create the object in the subclass

☒ B. Define a constructor in public access with different signature

C. Directly create the object in main() function

D. Not possible

- public : 公有，任何地方都可以访问。
- private : 私有，只能在本类内部访问，派生类不可见。
- protected : 受保护，只能在本类和派生类中访问，外部不可见。

C. Directly create the object in main() function

- 含义：直接在 `main()` 函数中创建对象。
- 分析：
 - 如果类的默认构造函数是私有的，那么无法直接在 `main()` 函数中使用默认构造函数创建对象。
 - 如果类的带参构造函数是受保护的，那么也无法直接在 `main()` 函数中调用它。
- 示例：

```
cpp
1 class MyClass {
2     private:
3         MyClass(); // 私有默认构造函数
4     protected:
5         MyClass(int x); // 受保护的带参构造函数
6 };
7
8 int main() {
9     MyClass obj; // 错误：无法访问私有默认构造函数
10    return 0;
11 }
```

- 这种方式不可行。
- 结论：错误。

7. Which programming paradigm below is not well supported in C++?

- | | |
|--------------------------------|--------------|
| A. Object-oriented programming | OOP |
| B. Procedural programming | 过程式C语言 |
| C. Declarative programming | 声明式SQL不关心如何做 |
| D. Generic programming | 泛型 |


```

class WeirdString {
public:
    WeirdString() = default;

    // 前置 ++ 运算符重载
    WeirdString& operator++() {
        s += "Hey";
        return *this;
    }

    // 后置 ++ 运算符重载 (注意 int 参数是标记)
    WeirdString operator++(int) {
        WeirdString old = *this; // 保存旧值
        s += "Ho";                // 修改当前对象
        return old;               // 返回旧值
    }

    // += 运算符重载
    WeirdString& operator+=(const WeirdString& rhs) {
        s += ("Ha" + rhs.str()); // 拼接字符串
        return *this;
    }

    string str() const { return s; } // 获取内部字符串

private:
    string s;
};

```

```

cpp
1 float b = 10.0f;
2 cout << func(b); // 输出 using namespace B20

```

虽然返回值是 `float` 类型 (即 `20.0f`)，但当你使用 `cout<<` 输出浮点数时，默认情况下 C++ 不会打印小数点和尾随的 `.0`。

类别	运算符
✗ 不可重载	<code>.</code> , <code>*</code> , <code>::</code> , <code>?:</code> , <code>sizeof</code> , <code>typeid</code> , <code>alignof</code> , <code>noexcept</code>
✓ 可重载	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>==</code> , <code>!=</code> , <code>[]</code> , <code>()</code> , <code>=</code> , <code>new</code> , <code>delete</code>

这是一个基类指针指向派生类对象的典型操作。

cpp

```
1 C2* pC2 = new C2(); // 创建了一个 C2 对象，pC2 是指向它的指针
2 C1* pC1 = pC2;      // 把 pC2 赋值给一个 C1* 类型的指针
```

✅ 这个赋值不会发生以下行为：

- ❌ 不会调用拷贝构造函数
- ❌ 不会创建新对象
- ❌ 不会复制整个对象的数据

✅ 它只是做了：

- 将派生类指针隐式转换为基类指针
- 即：把 `C2*` 类型的指针赋值给 `C1*` 类型的指针
- 这是 C++ 中面向对象编程中多态的基础



cpp

```
1 C1 a = *pC2; // 这会调用拷贝构造函数，因为这是对象构造
2 C1* pC1 = pC2; // 这只是指针赋值，不涉及对象拷贝
```

所以这一行：

cpp

```
1 C1* pC1 = pC2;
```

“👉 不会调用任何构造函数或拷贝构造函数。”

1-1 对单目运算符重载为友元函数时，可以说明一个形参。而重载为成员函数时，不能显式说明形参。

☒ T ☐ F

+obj; // 调用 operator+
-obj; // 调用 operator-

在实现时，不需要显式声明参数，直接通过 `this` 指针访问对象：

```
cpp
1 void MyClass::operator+() {
2     // 对 this->value 进行操作
3 }
```

因此，成员函数形式不需要显式声明形参。

<code>void operator+();</code>	✅ 合法	成员函数形式的单目运算符，无参数
<code>void operator+(MyClass& a);</code>	❌ 不合法	成员函数不能有参数，会被当作双目运算符
<code>friend void operator+(MyClass& a);</code>	✅ 合法	友元函数形式的单目运算符，需要一个参数

1-4 因为静态成员函数不能是虚函数，所以它们不能实现多态。

☒ T ☐ F

1. 静态成员函数：

- 静态成员函数属于类本身，而不是类的某个具体对象。
- 它们没有隐含的 `this` 指针，因此无法访问非静态成员变量或非静态成员函数。
- 静态成员函数在调用时不需要实例化对象，可以直接通过类名调用。

2. 虚函数与多态：

- 虚函数是实现运行时多态的关键机制。
- 多态的核心在于通过基类指针或引用调用派生类的函数，从而实现动态绑定（Dynamic Binding）。
- 虚函数必须是非静态的，因为静态成员函数没有 `this` 指针，无法区分不同的对象实例。

3. 静态成员函数是否可以是非静态函数：

- 静态成员函数不能被声明为虚函数，因为虚函数依赖于对象的动态类型，而静态成员函数不属于任何特定的对象实例。
- 静态成员函数的行为是固定的，不会根据对象的动态类型发生变化，因此无法实现多态。

2-4 关于动态绑定的下列描述中，（ ）是错误的。

- ☐ A. 动态绑定是以虚函数为基础的
- ☐ B. 动态绑定在运行时确定所调用的函数代码
- ☐ C. 动态绑定调用函数操作是通过指向对象的指针或对象引用来实现的
- ☒ D. 动态绑定是在编译时确定操作函数的

- 动态绑定（Dynamic Binding）是面向对象编程中的一种机制，用于在运行时根据实际对象的类型来决定调用哪个函数。
- 动态绑定的核心依赖于 虚函数，它是实现多态的关键。

2. 动态绑定的特点

- 动态绑定在运行时确定所调用的函数代码：根据实际对象的类型，而不是指针或引用的声明类型，来决定调用哪个函数。
- 动态绑定通过指向对象的指针或对象引用来实现：通过基类指针或引用调用虚函数时，会根据实际对象的类型动态绑定到相应的派生类实现。
- 动态绑定是在运行时确定操作函数的：与静态绑定（编译时绑定）不同，动态绑定的决策是在程序运行时做出的。

2-7 在下面类声明中，关于生成对象不正确的是（ ）。

```
class point
{ public:
  int x;
  int y;
  point(int a,int b) {x=a;y=b;}
};
```

- ☐ A. point p(10,2);
- ☐ B. point *p=new point(1,2);
- ☒ C. point *p=new point[2];
- ☐ D. point *p[2]={new point(1,2), new point(3,4)};

C. `point *p = new point[2];`

- 这是动态分配数组的方式，试图创建一个包含两个 `point` 对象的数组。
- 然而，类 `point` 的构造函数是带参构造函数 `point(int a, int b)`，而不是默认构造函数。
- 在使用 `new point[2]` 时，编译器会尝试调用默认构造函数（即无参构造函数），但类中并未提供默认构造函数。
- 因此，这段代码会导致编译错误，因为无法为数组中的每个元素调用合适的构造函数。
- 语法不正确。

D. `point *p[2] = {new point(1, 2), new point(3, 4)};`

- 这里定义了一个数组 `p`，其中包含两个 `point` 指针。
- 数组的每个元素分别被初始化为 `new point(1, 2)` 和 `new point(3, 4)`，这两个表达式都调用了类的构造函数 `point(int a, int b)`。
- 动态分配的对象会被正确初始化，语法也是正确的。
- 语法正确，且符合类的定义。

```


#include <iostream>
using namespace std;
class CAT
{
    public:
        CAT();
        CAT(const CAT&);
        ~CAT();
        int GetAge() const { return *itsAge; }
        void SetAge(int age){ *itsAge=age; }
    protected:
        int* itsAge;
};
CAT::CAT()
{
    itsAge=new int;
    *itsAge =5;
}
CAT::CAT(const CAT& c)
{
    itsAge = new int (5分);
    *itsAge= *(c.itsAge) (5分);
}

```

4. Destructors can not be overloaded. (T/F)

析构函数没有参数，因此无法通过改变参数列表来实现重载。

5. Among the following statements about new and malloc , which one is correct?

- A. new returns a void * pointer, which needs to be typecasted to the appropriate type.
- B. Both new and malloc call an appropriate constructor for object allocation.
-  C. None of the other options is correct.
- D. new and malloc are both operators.

1. 选项 A: new returns a void * pointer, which needs to be typecasted to the appropriate type.

• 错误：

- new 返回的是一个指向特定类型的指针，而不是 void *。
- 例如，new int 返回的是一个 int*，new MyClass 返回的是一个 MyClass*。
- 因此，new 返回的指针已经是类型化的，不需要显式类型转换。

11. In a C++ program, objects communicate each other by

- A. inheritance
- B. encapsulation 封装
- C. function overloading
- D. calling member functions

12. Among the following statements about C++, which is correct?

- A. All the static members of a class, both variables and functions included, need to be defined outside the class.
- B. Objects of polymorphic class type which has non-virtual C'tor might cause undefined behavior.
- C. The following C++ code segment leads some error when compiling since `another.x` is private.

```
class A {  
    int x;  
public:  
    A(): x(0) {}  
    int fun(A &another) {another.x += this->x;}  
};
```

D. The `const` in the following C++ code segment imposes restriction on `this`.

```
class A {  
public:  
    int x;  
    A(): x(0) {}  
    int fun(int i) const {return x + i;}  
};
```


选项 A: All the static members of a class, both variables and functions included, need to be defined outside the class.

分析:

- 静态成员 (`static` 成员) 包括静态变量和静态函数。
- 静态变量：
 - 如果在类内初始化为常量值 (如 `static int x = 10;`), 则无需在类外定义。
 - 如果没有在类内初始化, 则必须在类外定义。
- 静态函数：
 - 静态函数的声明可以在类内, 但实现必须在类外。

选项 B: Objects of polymorphic class type which has non-virtual C'tor might cause undefined behavior.

分析:

- 多态性 (Polymorphism) 是通过虚函数实现的, 通常涉及基类指针或引用指向派生类对象。
- 构造函数是否为虚函数：
 - 构造函数不能是虚函数, 因为构造函数在对象创建时调用, 而虚函数机制依赖于运行时类型信息 (RTTI), 这在构造过程中尚未完全建立。
 - 即使基类的构造函数不是虚函数, 也不会导致未定义行为。构造函数的作用是初始化对象, 而不是支持多态行为。
- 析构函数是否为虚函数：
 - 如果一个类具有多态性 (即包含虚函数), 那么它的析构函数应该声明为虚函数, 否则可能会导致未定义行为。

因此, 选项 B 的描述不准确, 因为它混淆了构造函数和析构函数的概念。

1. 虚函数机制依赖对象已经存在

- 虚函数通过虚函数表 (vtable) 来实现。
- 这个虚函数表是在对象构造完成后才建立的。
- 所以在构造函数执行时, 对象还没有完全构造完成, 虚函数表还未初始化, 无法支持虚函数调用。

“类比: 你不能让一个“正在出生”的婴儿自己选择将来要做什么职业 😊”

2. 构造函数本身没有 this 指针指向完整的对象

- 在构造函数执行期间, `this` 指针指向的对象还在逐步构建中。
- 如果构造函数是虚函数, 就可能试图访问派生类中的成员, 而这些成员尚未构造, 会导致未定义行为。

3. 虚函数用于运行时多态, 而构造过程是静态的

- 构造函数的调用顺序是确定的 (从基类到派生类), 不涉及运行时多态。
- 虚函数的作用是根据对象的实际类型动态绑定函数, 而构造过程中类型是明确的、静态的。

4. For the code below:

```
class A {  
    static int i;  
    //...  
};
```

Which statement is NOT true?

- A. All objects of class A reserve a space for i
- B. i is a member variable of class A
- C. All objects of class A share the space of i
- D. i is allocated in global data space

```

#include <iostream>
using namespace std;
class A {
public:
    int x;
    A(): x(6) {}
    int fun() {
        return x;
    }
};
class B: public A {
public:
    int fun() {
        return A::fun() + x;
    }
};
class C: public A {
public:
    int fun() {
        return A::fun() + x;
    }
};
class D: public B, public C {
public:
    int fun() {
        return B::fun() + C::fun();
    }
};
int main()
{
    D d;
    cout << d.fun();
}

```

✅ 类 D

```

cpp
1 class D : public B, public C {
2 public:
3     int fun() { return B::fun() + C::fun(); }
4 };

```

- D 同时继承了两个 A 子对象：
 - 一个来自 B
 - 一个来自 C
- 因此，D 中存在两个独立的 x 成员变量，分别属于 B 和 C 的基类部分。

由于没有使用 虚继承 (virtual inheritance)，类 D 中会有两个独立的 A 子对象，这可能导致歧义和数据冗余。例如：

```

cpp
1 d.x; // 编译错误！因为有两个 `x`（一个来自 B，一个来自 C）

```

```

int main()
{
    const MyClass obj1(10); // obj1 是 const 对象
    MyClass obj2(20);       // obj2 是非 const 对象

    obj1.Print(); // 调用 const 版本
    obj2.Print(); // 调用非 const 版本

    return 0;
}

```

```

#include <iostream>

struct X {
    X() {
        std::cout << "X::X()" << std::endl;
    }
    ~X() {
        std::cout << "X::~X()" << std::endl;
    }
};

struct Y : public X {
    Y() {
        std::cout << "Y::Y()" << std::endl;
    }
    ~Y() {
        std::cout << "Y::~Y()" << std::endl;
    }
};

struct Parent {
    Parent() {
        std::cout << "Parent::Parent()" << std::endl;
    }
    ~Parent() {
        std::cout << "Parent::~Parent()" << std::endl;
    }
    X x;
};

struct Child : public Parent {
    Child() {
        std::cout << "Child::Child()" << std::endl;
    }
    ~Child() {
        std::cout << "Child::~Child()" << std::endl;
    }
    Y y;
};

int main() {
    Child c;
}

```

```

X::X()
Parent::Parent()
X::X()
Y::Y()
Child::Child()
Child::~Child()
Y::~Y()
X::~X()
Parent::~Parent()
X::~X()

```

4. Which of the following sentences is not correct

(D)

- a. Derived class object can be assigned to Base class object.
- b. Base class object can be assigned to Derived class object.
- c. Derived class object point can be assigned to Base class object point.
- d. Base class object point can be assigned to Derived class object point.

- 派生类对象的指针 (`Derived*`) 可以赋值给基类对象的指针 (`Base*`), 这是向上转型 (Upcasting) 的一种形式。
- 向上转型是合法的, 因为派生类对象总是可以被视为基类对象。

例如:

```
cpp
1 class Base {};
2 class Derived : public Base {};
3
4 int main() {
5     Derived d;
6     Base* basePtr = &d; // 正确: 派生类对象指针可以赋值给基类对象指针
7     return 0;
8 }
```

- 基类对象的指针 (`Base*`) 不能直接赋值给派生类对象的指针 (`Derived*`), 因为这涉及到向下转型 (Downcasting), 而向下转型需要显式转换 (如 `static_cast` 或 `dynamic_cast`)。
- 如果直接赋值, 编译器会报错。

例如:

```
cpp
1 class Base {};
2 class Derived : public Base {};
3
4 int main() {
5     Base b;
6     Derived* derivedPtr = &b; // 错误: 基类对象指针不能直接赋值给派生类对象指针
7     return 0;
8 }
```