

9.2 泛型编程

面试例题1：解释一下什么是泛型编程，泛型编程和C++及STL的关系是什么？并且，你是怎么在C++环境里进行泛型编程的？[美国某著名CPU生产公司面试题]

答案：泛型编程是一种基于发现高效算法的最抽象表示的编程方法。也就是说，以算法为起点并寻找能使其工作且有效率工作的最一般的必要条件集。令人惊讶的是，很多不同的算法都需要相同的必要条件集，并且这些必要条件有多种不同的实现方式。类似的事实在数学里也可以看到。大多数不同的定理都依赖于同一套公理，并且对于同样的公理存在多种不同的模型。泛型编程假定有某些基本的法则在支配软件组件的行为，并且基于这些法则有可能设计可互操作的模块，甚至还有可以使用此法则去指导我们的软件设计。STL就是一个泛型编程的例子。C++是我可以实现令人信服的例子的语言。

面试例题2：Below is usual way we find one element in an array: In this case we have to bear the knowledge of value type “int”, the size of array, even the existence of an array. Would you re-write it using template to eliminate all these dependencies?（我们通常求解一个数在一个数列里的方法是这样的：在这个例子中，我们得先知道int类型的知识，知道队列的大小，甚至要建立一个队列。你能否用泛型编程模拟这个队列，做到不知道上面的附加知识仍然能得出结果？）[德国某著名软件咨询企业2004年面试题]

```
const int *find1(const int* array, int n,
    int x)
{
    const int* p = array;
    for(int i = 0; i < n; i++)
    {
        if(*p == x)
```

```
        {
            return p;
        }
        ++p;
    }
    return 0;
}
```

解析：这是一个把普通函数改成泛型函数的问题，在这里公司考的是如何将普通函数转换成泛型函数。

答案：

修改代码如下：

```
template<typename T>
const T* My_find(T *array, T n, T x)
{
    const T* p = array;
    int i;
    for(i=0;i<n;++i)
    {
```

```
        if(*p == x)
            return p;
        ++p;
    }
    return 0;
};
```

或者：

```
template<typename T>
const T* My_find2(const T* s, const T* e, T x)
{
    const T* p=s;
    while(p!=e)
    {
        if(*p == x)
```

```
    {
        return p;
    }
    ++p;
}
return e;
};
```

9.3 模板

面试例题 1: Please write a program to realize the model described in the figure. You should design your program as generic as possible so that we can enhance the model in the future easily without making too much change in your program. (写一个程序来实现下图的模型中整型变量的问题。你必须尽可能地将程序设计为一类(同类),以便于我们今后在不做大量更改的情况下对它进行升级。) [德国某著名软件咨询企业 2005 年 11 月面试题]

解析: 这是一道考函数指针的面试例题,当然也可以用模板来做。

如题目所示:要求 A 与 B 之和、C 与 D 之积,以及由于 E 的情况不同而得出的结果。我们当然可以单独为 A 和 B 设置一个求和函数,为 C 和 D 设置一个求积函数。但是一旦功能改变,比如说我们不得不求 A 与 B 之差或 C/D 的情况该怎么做?第三次也许是求最小值和两数之积……如果不用函数指针,我们需要写多少个这样的 test() 函数?显然,函数指针为我们的编程提供了灵活性。

一个函数在编译时被分配给一个入口地址,这个入口地址就称为函数的指针,正如同指针是一个变量的地址一样。函数指针的用途很多,最常用的用途之一是把指针作为参数传递到其他函数。

答案: 程序源代码与解释如下:

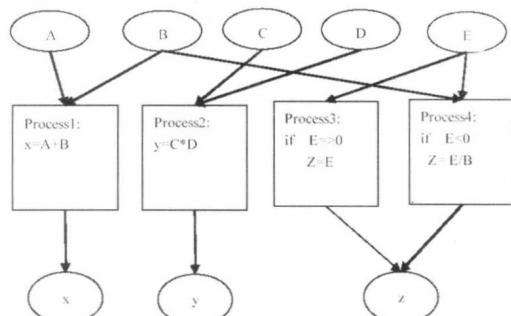
```
# include<stdio.h>

//比较函数
int jug(int x,int y)
{
    if (x>=0)
    {
        return x;
    }
    else if (y==0)
    {
        return x;
```

```
}
else
    return x/y;
}

//求和函数
int sub(int x, int y)
{
    return (x+y);
}

//求差函数
```



```

int minus(int x,int y)
{
    return(x-y);
}
//函数指针
void test(int (*p)(int,int),int a,int b)
{
    int Int1;
    Int1=(*p)(a,b);
}

```

```
printf("a=%d,a=%d %d\n",a,b,Int1);
```

```

}
int main()
{
    int a=1,b=2,c=3,d=4,e=-5;

    test(sub,a,b);           //求和
    test(minus,c,d);        //求差

    test(jug,e,b);          //判断
    return 0;
}

```

另外，有些地方必须使用函数指针才能完成给定的任务，特别是异步操作的回调和其他需要匿名回调的结构。另外，像线程的执行和事件的处理，如果缺少了函数指针的支持也是很难完成的。当然，该程序也可以用静态模板类来实现，解决方法是一致的，代码如下：

```

#include <iostream>
using namespace std;

template<class T>
//建立一个静态模板类
class Operate{
public:
    static T Add(T a, T b) {
        return a+b;
    }
    static T Mul(T a, T b) {
        return a*b;
    }
    static T Judge(T a, T b=1) {
        if(a>=0) {
            return a;
        }
    }
}

```

```

}
else {
    return a/b;
}
};

int main() {
    int A, B, C, D, E, x, y, z;
    A=1, B=2, C=3, D=4, E=5;
    x=Operate<int>::Add(A,B);
    y=Operate<int>::Mul(C,D);
    z=Operate<int>::Judge(E,B);
    cout<<x<<'\n'<<y<<'\n'<<z<<endl;

    return 0;
}

```

扩展知识（模板与容器）

数据结构本身十分重要。当程序中存在着对时间要求很高的部分时，数据结构的选择就显得更加重要。

经典的数据结构数量有限，但是我们常常重复着一些为了实现向量、链表等结构而编写的代码。这些代码都十分相似，只是为了适应不同数据的变化而在细节上有所不同。STL容器就为我们提供了这样的方便，它允许我们重复利用已有的实现构造自己的特定类型下的数据结构。通过设置一些模板类，STL容器对最常用的数据结构提供了支持。这些模板的参数允许我们指定容器中元素的数据类型，可以将许多重复而乏味的工作简化。

容器部分主要由头文件<vector>、<list>、<deque>、<set>、<map>、<stack>和<queue>组成。对于常用的一些容器和容器适配器（可以看作由其他容器实现的容器），可以通过下表总结一下它们和相应头文件的对应关系。

数据结构	描述	实现头文件
向量 (vector)	连续存储的元素	<vector>
列表 (list)	由节点组成的双向链表	<list>
双队列 (dequeue)	连续存储的指向不同元素的指针所组成的数组	<deque>
集合 (set)	由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序	<set>
多重集合 (multiset)	允许存在两个次序相等的元素的集合	<set>
栈 (stack)	后进先出的值的排列	<stack>
队列 (queue)	先进先出的值的排列	<queue>
优先队列 (priority_queue)	元素的次序是由作用于所存储的值对上的某种谓词决定的一种队列	<queue>
映射 (map)	由{键, 值}对组成的集合，以某种作用于键对上的谓词排列	<map>

面试例题 2：试用多态实现线性表（队列、串、堆栈），要求具备线性表的基本操作：插入、删除、测长等。[美国著名软件企业 GS 公司 2007 年 11 月面试题]

解析：队列、串、堆栈都可以实现 push、pop、测长等操作。现在要求用多态去实现，就要建立一个线性表的共性模板，来实现以上的功能。

答案：程序源代码与解释如下：

```
#include "iostream"
using namespace std;
template<typename t>
struct tcontainer
{
    virtual void push(const t &) = 0;
    virtual void pop() = 0;
    virtual const t& begin() = 0;
    virtual const t& end() = 0;
    virtual size_t size() = 0;
};
template<typename t>
struct tvector : public tcontainer<t>
{
    static const size_t _step = 100;
    tvector()
    {
        _size = 0;
        //初始化向量实际大小
        _cap = _step;
        //向量容量为 100
    }
}
```

```
buf = 0;
//首地址，需动态分配内存
re_capacity(_cap);
//此时 buf 为空，即要设置 buf 初始值
//配了 100 个元素的空间
}
~tvector()
{
    free(buf);
}
void re_capacity(size_t s)
//调整容量
{
    if (!buf)
        buf = (t*)malloc(sizeof(t) * s);
    else
        buf=(t*)realloc(buf,sizeof(t)*s);
}
virtual void push(const t & v)
{
    if (_size >= _cap)
```

```

        re_capacity(_cap += _step);
        buf[_size++] = v;
    }
    virtual void pop()
    {
        if (_size)
            _size--;
    }
    virtual const t& begin()
    {
        return buf[0];
    }
    virtual const t& end()
    {
        if (_size)
            return buf[_size - 1];
    }
    virtual size_t size()
    {

```

```

        return _size;
    }
    const t& operator[](size_t i)
    {
        if (i >= 0 && i < _size)
            return buf[i];
    }
private :
    size_t _size;           //实际元素个数
    size_t _cap;            //已分配的容量
    t* buf;                //首地址
};

int main()
{
    tvector<int> v;
    for (int i = 0; i < 1000; ++i)
        v.push(i);
    for (int i = 0; i < 1000; ++i)
        cout<<v[i]<<endl;
}

```

面试例题 2：1~9 的 9 个数字，每个数字只能出现一次，要求这样一个 9 位的整数：其第一位能被 1 整除，前两位能被 2 整除，前三位能被 3 整除……依此类推，前 9 位能被 9 整除。

[中国著名互联网企业 B 公司 2013 年 11 月面试题]

解析：可以采用枚举实现以上的功能。

答案：程序源代码下：

```

#include <stdio.h>
#include <vector>
using namespace std;
bool used[10];
vector <long long> v;
void dfs(int k, long long a)
{
    if (k&&a%k!=0)
        return;
    if (k==9)
    {
        v.push_back(a);
        return;
    }
    for (int i=1;i <=9;i++)

```

```

        if (!used[i])
        {
            used[i]=1;
            dfs(k+1,a*10+i);
            used[i]=0;
        }
    }
int main()
{
    dfs(0,0);
    for (int i=0;i < v.size();i++)
        printf("%lld ",v[i]);
    getchar();
}

```

第 10 章

面向对象

有这样一句话：“编程是在计算机中反映世界”，我觉得再贴切不过。面向对象（Object-Oriented）对这种说法的体现也是最优秀的。比如，我们设计的数据结构是一个学生成绩的表现，而对数据结构的操作（函数）是分离的，虽然这些操作是针对这种数据结构而产生的。为了管理大量的数据，我们不得不小心翼翼地使用它们。

作为一名软件开发人员，我们可以深刻地体会到面向对象系统设计带来的种种便利。

- 良好的可复用性：开发同类项目的次数与开发新项目的时间成反比，减少重复劳动。
- 易维护：基本上不用花太大的精力跟维护人员讲解，他们可以自己读懂源程序并修改。否则开发的系统越多，负担就越重。
- 良好的可扩充性：以前，在向一个用结构化思想设计的庞大系统中加一个功能则必须考虑兼容前面的数据结构，理顺原来的设计思路。即使客户愿意花钱修改，作为开发人员多少都有点儿恐惧。在向一个用面向对象思想设计的系统中加入新功能，不外乎是加入一些新的类，基本上不用修改原来的东西。

在面试过程中，求职者是否对面向对象的基本概念、结构和类、多态性及构造函数有清晰的认识，是否能够有效地编程实现面向对象的各种功能，是 IT 企业考查的重点内容。

10.1 面向对象的基本概念

面试例题 1：Which of the following(s) are NOT related to object-Oriented Design? (以下选项中哪个不是面向对象设计)

- A. Inheritance (继承)
- B. Liskov substitution principle (里氏代换原则)

- C. Open-close principle (开闭原则)
- D. Polymorphism (多态)
- E. Defensive programming (防御式编程)

解析：面向对象设计的三原则：封装，继承，多态。

Liskov Substitution Principle (里氏代换原则) 是继承复用的基石：子类型必须能够替换它们的基类型。如果每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，使得以 T1 定义的所有程序 P 在所有的对象 o1 都代换为 o2 时，程序 P 的行为没有变化，那么类型 T2 是类型 T1 的子类型。换言之，一个软件实体如果使用的是一个基类的话，那么一定适用于其子类，而且它根本不能察觉出基类对象和子类对象的区别。只有衍生类替换基类的同时软件实体的功能没有发生变化，基类才能真正被复用。

Open-close principle (开闭原则) 是面向对象设计的重要特性之一：软件对扩展应该是开放的，对修改应该是关闭的。通俗点说，已经设计好的代码应该是不做修改的（闭），如果需求改变，就另外自己扩展一块去（开），别破坏我原来的代码。

Defensive programming (防御式编程) 只是一种编程技巧，与面向对象设计无关。

防御式编程的主要思想是：子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误数据。这种思想是将可能出现的错误造成的影响控制在有限的范围内。简单来说，就是不管出了什么错，程序都不能崩溃。用过 Windows 98，应该都对蓝屏深有体会。所以不让程序崩溃是合理的。但是这样会让程序留下隐患，比如传进来的字符串太长了，固定大小的字符串数组装不下，需要截取一段，但问题是，截取之后的信息可能就不是用户真正想要的信息了，这样可能会在后面使用数据的时候带来问题，而且对于很多后端模块，数据都应该是合法的，存在非法数据，应该是上游模块出 bug 了。

理想情况下，每一步都做防御，发现非法的情况，就输出 log，然后解决问题。但是，如果是上游模块出了问题，上游的程序员可就不积极帮忙了，反正程序没崩溃，服务还正常。这时候也没办法，只能做些简单的容错处理，做了容错之后，错误就传递到了下一层，下一层也只能继续做容错。然后哪天来个新人，看到 code 有 bug，修一下，这下惨了，下游已经把有 bug 的输入作为标准了，你这一修，把下游弄崩溃了，然后只能回滚。最后的结果就是系统里充满了各种各样的 bug，而且没法子修。每个人想着不要在我这个模块崩溃，就算要在我这个模块崩溃，也不要在我维护的时候崩溃。这本质上是一种大公司病。各部门自扫门前雪，每个人都在想，我走了之后哪管洪水涛天。

不是说模块可以随便崩溃，想要服务稳定，更重要的不是单个模块是否崩溃，而是整个系统是否稳定，把错误控制在一定范围内。比如某些特定的输入会引起系统崩溃，在探测出

这些输入之后，就把它挡在前端。如果是数据更新引起的崩溃，就先少量更新，一旦发现更新出错，剩下的就不更新了。系统中尽量不要有单点。但这些事情，在大公司里也不是程序员能做到的，而是架构师的设计问题了。

答案：E

面试例题 2： Which of the following C++ keyword(s) is(are) related to encapsulation? (下面哪个/些关键字与封装相关?)

- A. virtual B. void C. interface D. private E. all of the above

解析：什么是封装?

从字面意思来看，封装就是把一些相关的东西打包成一“坨”。封装最广为人知的例子，就是在面向对象编程里面，把数据和针对该数据的操作，统一到一个 class 里。

很多人把封装的概念局限于类，认为只有 OO 中的 class 才算是封装。这实际上是片面的在很多不使用“类”的场合，一样能采用封装的手法：

(1) 通过文件。

比如 C 和 C++ 支持对头文件的包含 (#include)。因此，可以把一些相关的常量定义、类型定义、函数声明，统统封装到某个头文件中。

(2) 通过 namespace/package/module。

C++ 的 namespace、Java 的 package、Python 的 module，这些语法虽然称呼各不相同，但具有相同的本质。因此，也可以利用这些语法来进行封装。

那么封装有一个主要的好处，就是增加软件代码的内聚性。通过增加内聚性，进而提高可复用性和可维护性。此外还可以“信息隐藏”：把不该暴露的信息藏起来。如 private、protected 之类的关键字。这些关键字可以通过访问控制，来达到信息隐藏的目的。

本题中，interface 属于继承，virtual 属于多态，private 才是与封装相关。

答案：D

面试例题 2： C++ 中的空类默认产生哪些类成员函数? [中国某著名综合软件公司 2005 年面试题]

```
class Empty
{
    public:
};
```

解析：类的概念问题。

答案：对于一个空类，编译器默认产生 4 个成员函数：默认构造函数、析构函数、复制构造函数和赋值函数。

10.2 类和结构

面试例题 1：structure 是否可以拥有 constructor / destructor 及成员函数？如果可以，那么 structure 和 class 还有区别么？[中国台湾某著名计算机硬件公司 2005 年 11 月面试题]

答案：区别是 class 中变量默认是 private，struct 中的变量默认是 public。struct 可以有构造函数、析构函数，之间也可以继承，等等。C++ 中的 struct 其实和 class 意义一样，唯一不同的就是 struct 里面默认的访问控制是 public，class 中默认的访问控制是 private。C++ 中存在 struct 关键字的唯一意义就是为了让 C 程序员有个归属感，是为了让 C++ 编译器兼容以前用 C 开发的项目。

扩展知识

我们可以写一段 struct 继承的例子，代码如下：

```
#include <iostream>
using namespace std;
enum BREED { GOLDEN, CAIRN, DANDIE, SHETLAND, DOBERMAN, LAB };
struct Mammal
{
public:
    Mammal(): itsAge(2), itsWeight(5) {}
    ~Mammal() {}

    int GetAge() const { return itsAge; }
    void SetAge(int age) { itsAge = age; }
    int GetWeight() const { return itsWeight; }
    void SetWeight(int weight) { itsWeight = weight; }

    void Speak() const { cout << "\n Mammal sound !"; }
    void Sleep() const { cout << "\n Shhh. I'm sleeping."; }
protected:
    int itsAge;
    int itsWeight;
};

struct Dog : public Mammal
{

public:
    Dog(): itsBreed(GOLDEN) {}
    ~Dog() {};
    BREED GetBreed() const { return itsBreed; }
    void SetBreed(BREED breed) { itsBreed = breed; }
```

```

void WagTail() const {cout << "Tail wagging ... \n";}
void BegForFood() const {cout << "Begging for food... \n";}

private:
    BREED itsBreed;

};

int main()
{
    Dog fido;
    fido.Speak();
    fido.WagTail();
    cout << "Fido is " << fido.GetAge() << " years old \n";
    return 0;
}

```

面试例题 2：现有以下代码，则编译时会产生错误的是（ ）。[中国某著名计算机金融软件公司 2005 年面试题]

```

struct Test
{
    Test(int){}
    Test(){}
    void fun(){}
};

int main ()

```

```

    {
        Test a(1); //语句1
        a.fun(); //语句2
        Test b(); //语句3
        b.fun(); //语句4
        return 0;
    }

```

- A. 语句 1 B. 语句 2 C. 语句 3 D. 语句 4

解析：Test b()这个语法等同于声明了一个函数，函数名为 b，返回值为 Test，传入参数为空。但是实际上，代码作者是希望声明一个类型为 Test，变量为 b 的变量，应该写成 Test b;，但程序中这个错误在编译时是检测不出来的。出错的是语句 4 “b.fun();”，它是编译不过去的。

答案：D

10.3 成员变量

面试例题 1：哪一种成员变量可以在同一个类的实例之间共享？[中国台湾某著名计算机硬件公司 2005 年 11 月面试题]

答案：必须使用静态成员变量在一个类的所有实例间共享数据。如果想限制对静态成员变量的访问，则必须把它们声明为保护型或私有型。不允许用静态成员变量去存放某一个对象的数据。静态成员数据是在这个类的所有对象间共享的。

面试例题2：指出下面程序的错误。如果把静态成员数据设为私有，该如何访问？[中国台湾某著名计算机硬件公司2005年11月面试题]

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

class Cat
{
public:
    Cat(int age):itsAge(age)
{HowManyCats++;}
    virtual ~Cat() {HowManyCats--;}
    virtual int GetAge(){return itsAge;}
    virtual void SetAge(int age)
{itsAge = age;}
    static int HowManyCats;
private:
    int itsAge;
};

int main()
{
```

```
const int MaxCats = 5;int i;
Cat *CatHouse[MaxCats];
for(i=0;i<MaxCats;i++)
    CatHouse[i]=new Cat(i);
for(i=0;i<MaxCats;i++)
{
    cout << "THere are";
    cout << Cat::HowManyCats;
    cout <<"cats left!\n";
    cout <<"Deleting the one which is";
    cout << CatHouse[i]->GetAge();
    cout <<"years old\n";
    delete CatHouse[i];
    CatHouse[i]=0;
}
return 0;
}
```

答案：该程序错在设定了静态成员变量，却没有给静态成员变量赋初值。如果把静态成员数据设为私有，可以通过公有静态成员函数访问。

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

class Cat
{
public:
    Cat(int age):itsAge(age)
{HowManyCats++;}
    virtual ~Cat() {HowManyCats--;}
    virtual int GetAge(){return itsAge;}
    virtual void SetAge(int age)
{itsAge = age;}
    static int GetHowMany()
{return HowManyCats;}
    //static int HowManyCats ;
private:
    int itsAge;
    static int HowManyCats ;
};

int Cat::HowManyCats = 0;

void tele();
```

```
int main()
{
    const int MaxCats = 5;int i;
    Cat *CatHouse[MaxCats];
    for(i=0;i<MaxCats;i++)
    {
        CatHouse[i]=new Cat(i);
        tele();
    }
    for(i=0;i<MaxCats;i++)
    {

        delete CatHouse[i];
        tele();
    }
    return 0;
}

void tele()
{
    cout << "THere are" << Cat::GetHowMany()
    << "Cats alive!\n";
}
```

面试例题3：请问下面程序打印出的结果是什么？[中国著名杀毒软件企业J公司2008年4月面试题]

```
# include <iostream>
```

```
# include <string>
```

```

using namespace std;
class base
{
private:
    int m_i;
    int m_j;
public:
    base( int i ) : m_j(i), m_i(m_j) {}
    base() : m_j(0), m_i(m_j) {}
    int get_i() {return m_i;}
}

```

```

int get_j() {return m_j;}
};

int main(int argc, char* argv[])
{
    base obj(98);
    cout << obj.get_i() << endl
    << obj.get_j() << endl;
    return 0;
}

```

解析：本题想要得到的结果是“98,98”。但是成员变量的声明是先 m_i，然后是 m_j；初始化列表的初始化变量顺序是根据成员变量的声明顺序来执行的，因此 m_i 会被赋予一个随机值。更改一下成员变量的声明顺序可以得到预想的结果。如果要得到“98,98”的输出结果，程序需要修改如下：

```

#include <iostream>
#include <string>
using namespace std;
class base
{
private:
    int m_j;
// 修改成员变量声明顺序
    int m_i;
public:
    base() : m_j(0), m_i(m_j){}
}

```

```

base( int i ) : m_i(m_j), m_j(i){}
int get_i() {return m_i;}
int get_j() {return m_j;}
};

int main(int argc, char* argv[])
{
    base obj(98);
    cout << obj.get_i() << endl
    << obj.get_j() << endl;
    return 0;
}

```

答案：输出结果第一个为随机数，第二个是 98。

面试例题 4：这个类声明正确吗？为什么？

```

class A
{
    const int Size = 0;
};

```

解析：这道程序题存在着成员变量问题。常量必须在构造函数的初始化列表里面初始化或者将其设置成 static。

答案：

正确的程序如下：

```

class A
{ A()
    {const int Size=9;}
};

```

或者：

```

class A
{ static const int Size=9;
};

```

10.4 构造函数和析构函数

面试例题 1：MFC 类库中，CObject 类的重要性不言自明。在 CObject 的定义中，我们看到一个有趣的现象，即 CObject 的析构函数是虚拟的。为什么 MFC 的编写者认为 virtual destructors are necessary（虚拟的析构函数是必要的）？[美国某著名移动通信企业 2004 年面试题]

解析：

我们可以先构造一个类如下：

```
class CBase
{
public:
    ~CBase() { ... };
    ...
};

class CChild : public CBase
{
public:
```

```
~CChild() { ... };

...
};

main()
{
    Child c;
    ...
    return 0;
}
```

上面代码在运行时，由于在生成 CChild 对象 c 时，实际上在调用 CChild 类的构造函数之前必须首先调用其基类 CBase 的构造函数，所以当撤销 c 时，也会在调用 CChild 类析构函数之后，调用 CBase 类的析构函数（析构函数调用顺序与构造函数相反）。也就是说，无论析构函数是不是虚函数，派生类对象被撤销时，肯定会依次上调其基类的析构函数。

那么为什么 CObject 类要搞一个虚的析构函数呢？

因为多态的存在。

仍以上面的代码为例，如果 main()中有如下代码：

```
CBase * pBase;
CChild c;
pBase = &c;
```

那么在 pBase 指针被撤销时，调用的是 CBase 的析构函数还是 CChild 的呢？显然是 CBase 的（静态联编）析构函数。但如果把 CBase 类的析构函数改成 virtual 型，当 pBase 指针被撤销时，就会先调用 CChild 类构造函数，再调用 CBase 类构造函数。

答案：在这个例子里，所有对象都存在于栈框中，当离开其所处的作用域时，该对象会被自动撤销，似乎看不出什么大问题。但是试想，如果 CChild 类的构造函数在堆中分配了内存，而其析构函数又不是 virtual 型的，那么撤销 pBase 时，将不会调用 CChild::~CChild()，从而不会释放 CChild::CChild()占据的内存，造成内存泄漏。

将 CObject 的析构函数设为 virtual 型，则所有 CObject 类的派生类的析构函数都将自动变为 virtual 型，这保证了在任何情况下，不会出现由于析构函数未被调用而导致的内存泄漏。这才是 MFC 将 CObject::~CObject() 设为 virtual 型的真正原因。

面试例题 2：析构函数可以为 virtual 型，构造函数则不能。那么为什么构造函数不能为虚呢？

[美国某著名移动通信企业 2004 年面试题]

答案：虚函数采用一种虚调用的办法。虚调用是一种可以在只有部分信息的情况下工作的机制，特别允许我们调用一个只知道接口而不知道其准确对象类型的函数。但是如果要创建一个对象，你势必要知道对象的准确类型，因此构造函数不能为虚。

面试例题 3：如果虚函数是非常有效的，我们是否可以把每个函数都声明为虚函数？

答案：不行，这是因为虚函数是有代价的：由于每个虚函数的对象都必须维护一个 v 表，因此在使用虚函数的时候都会产生一个系统开销。如果仅是一个很小的类，且不想派生其他类，那么根本没必要使用虚函数。

面试例题 4：析构函数可以是内联函数吗？[英国某著名计算机图形图像公司面试题]

解析：我们可以先构造一个类，让它的析构函数是内联函数，如下所示：

```
#include <iostream>
using namespace std;
class A
{
public :
    void foo()
    {cout<<"A";}
    ~A();
};
```

```
inline A::~A()
{ cout << "inline";}

int main()
{
    A *niu=new A();
    niu->foo();
    delete niu;
    return 0;
}
```

该程序可以正确编译并得出结果。

答案：析构函数可以是内联函数。

面试例题 5：请看下面一段程序：

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class B
{
private:
    int data;
public:
```

```
B()
{
    cout<<"default constructor"
<<endl;
}
~B()
{
    cout << "destructed " <<endl;
}
```

```

B(int i):data(i)
{
    cout<<"constructed by parameter"
<<data<<endl;
}
B Play( B b)
{

```

```

    return b ;
}
int main(int argc, char* argv[])
{
    B temp=Play(5);
    return 0;
}

```

问题：

- (1) 该程序输出的结果是什么？为什么会有这样的输出？
- (2) B(int i):data(i)，这种用法的专业术语叫什么？
- (3) Play(5)，形参类型是类，而5是个常量，这样写合法吗？为什么？

[英国著名图形图像公司 A 2008 年面试题]

解析： B temp=Play(5)，理论上该有两次复制（复制）构造函数，编译器把这两次合为一次，提高效率。所以把此句改为 Play(5)，会发现结果一样。都是 2 次析构，只不过区别在于：Play(5)的第一次析构是在函数退出时，对形参的副本进行析构。第二次析构是在函数返回类对象时，再次调用复制构造函数来创建返回类对象的副本。所以还需要一次析构函数来析构这个副本。而 B temp=Play(5)中的第二次析构是析构 B temp。在 B temp=Play(5)加一句 system('pause');可以验证第二次析构是在析构 B temp，而不是析构函数返回值对象的副本，编译器把这两次合为一次，提高效率。

答案：

- (1) 应该有两个“destructed”输出：

```

constructed by parameter5
//在Play(5)处，5通过隐含的类型转换调用了B::B(int i)
destructed //Play(5)返回时，参数的析构函数被调用
destructed //temp的析构函数调用；temp的构造函数调用的是编译器生成的复制构造函数

```

- (2) 带参数的构造函数，冒号后面是成员变量初始化列表（member initialization list）。
- (3) 合法。单个参数的构造函数如果不添加 explicit 关键字，会定义一个隐含的类型转换（从参数的类型转换到自己）；添加 explicit 关键字会消除这种隐含转换。

10.5 复制构造函数和赋值函数

面试例题 1： 编写类 String 的构造函数、析构函数和赋值函数。[中国某著名综合软件公司 2005 年面试题]

答案：

已知类 String 的原型为：

```
class String
{
public:
// 普通构造函数
String(const char *str = NULL);
// 复制构造函数
String(const String &other);
// 析构函数
```

```
~ String(void);
// 赋值函数
String & operate =(const String &other);
private:
// 用于保存字符串
char *m_data;
};
```

编写 String 的上述 4 个函数。

1. String 的析构函数

为了防止内存泄漏，我们还需要定义一个析构函数。当一个 String 对象超出它的作用域时，这个析构函数将会释放它所占用的内存。代码如下：

```
String::~String(void)
{
delete [] m_data;
// 由于 m_data 是内部数据类型，也可以写成 delete m_data;
}
```

2. String 的构造函数

这个构造函数可以帮助我们根据一个字符串常量创建一个 MyString 对象。这个构造函数首先分配了足量的内存，然后把这个字符串常量复制到这块内存，代码如下：

```
String::String(const char *str)
{
if(str==NULL)
{
m_data = new char[1]; // 若能加NULL 判断则更好
*m_data = '\0';
}
else
{
int length = strlen(str);
// 若能加NULL 判断则更好
m_data = new char[length+1];
strcpy(m_data, str);
}
}
```

strlen 函数返回这个字符串常量的实际字符数（不包括 NULL 终止符），然后把这个字符串常量的所有字符赋值到我们在 String 对象创建过程中为 m_data 数据成员新分配的内存中。有了这个构造函数后，我们可以像下面这样根据一个字符串常量创建一个新的 String 对象：

```
string str("hello");
```

3. String 的复制构造函数

所有需要分配系统资源的用户定义类型都需要一个复制构造函数，这样我们可以使用这样的声明：

```
MyString s1("hello");
MyString s2=s1;
```

复制构造函数还可以帮助我们在函数调用中以传值方式传递一个 MyString 参数，并且在当一个函数以值的形式返回 MyString 对象时实现“返回时复制”。

```
String::String(const String &other) // 3 分
{
int length = strlen(other.m_data);
// 若能加NULL 判断则更好
```

```
m_data = new char[length+1];
strcpy(m_data, other.m_data);
}
```

4. String 的赋值函数

赋值函数可以实现字符串的传值活动：

```
MyString s1(hello);
MyString s2;
s1=s2;
```

代码如下：

```
String & String::operator=(const String
&other)
{
// 检查自赋值
if(this == &other)
return *this;
// 释放原有的内存资源
delete [] m_data;
```

```
// 分配新的内存资源，并复制内容
int length = strlen(other.m_data);
m_data = new char[length+1];
strcpy(m_data, other.m_data);
// 返回本对象的引用
return *this;
}
```

扩展知识

这里还有一个问题：String & String::operator=(const String &other) 的 const 是做什么用的？

Const 有两个作用。一个如果不加入 const 的话，比如：

```
MyString s3(pello);
Const MyString s4(qello);
s3=s4;
```

这样就会出现问题。因为一个 const 变量是不能随意转化成非 const 变量的。

其次是诸如：

```
MyString s7(pello);
MyString s8(pello);
MyString s9(qello);
```

```
S9=s7+s8;
```

不用 `const` 也会报错，因为用“+”赋值必须返回一个操作值已知的 `MyString` 对象，除非它是一个 `const` 对象。

面试例题 2： Which of the following is true about “Copy Constructor”? (下面关于复制构造函数的说法哪一个是正确的?) [中国某著名综合软件公司 2005 年面试题]

- A. They copy constructor into each other. (给每一个对象复制一个构造函数。)
- B. A default is provided, but simply does a member-wise copy. (有一个默认的复制构造函数。)
- C. They can't copy arrays into each other. (不能复制队列。)
- D. All of the above. (以上结果都正确。)

解析： 复制构造函数问题。

答案： B

面试例题 3： Which of the following class DOES NOT need a copy constructor? (下面所列举的类哪个不需要复制构造函数?) [中国台湾某著名杀毒软件公司 2004 年面试题]

- A. A matrix class in which the actual matrix is allocated dynamically within the constructor and is deleted within its destructor. (一个矩阵类：动态分配，对象的建立是利用构造函数，删除是利用析构函数。)
- B. A payroll class in which each object is provided with a unique ID. (一个花名册类：每一个对象对照着唯一的 ID。)
- C. A word class containing a string object and vector object of line and column location pairs. (一个 word 类，对象是字符串类和模板类。)
- D. A library class containing a list of book object. (一个图书馆类：由一系列书籍对象构成。)

解析：

按照题意，寻找一个不需要复制构造函数的类。

A 选项要定义复制构造函数。

B 选项中，不自定义复制构造函数的话，势必造成两个对象的 ID 不唯一。至于说自定义了复制构造函数之后，如何保证新对象的 ID 唯一，那是实现的问题。实现的方法多种多样，比如可以使用当前的系统 tick 数作为新 ID。当然语义上有损失，不是完全意义上的复制，但在这儿只能在保持语义和实现目的之间来一个折中。

选 C 的原因是使用默认的复制构造，`string` 子对象和 `vector` 子对象的类都是成熟的类，

都有合适的赋值操作，复制构造函数以避免“浅复制”问题。

D选项显然是定义复制构造函数。

答案：C

面试例题4： Which virtual function re-declarations of the Derived class are correct? (哪个子类的虚函数重新声明是正确的?) [中国台湾某著名杀毒软件公司 2004 年面试题]

- | | |
|---|--------------------------------|
| A. Base* Base::copy(Base*); | B. Base* Base::copy(Base*); |
| Base* Derived::copy(Derived*); | Derived* Derived::copy(Base*); |
| C. ostream& Base::print(int,ostream&=cout); | D. void Base::eval() const; |
| ostream& Derived::print(int,ostream&); | void Derived::eval(); |

解析：本题问的是哪个派生类的虚函数再声明是对的。

A 选项错误，因为虚函数的声明必须与基类中定义方式完全匹配。而子类的虚函数的形参为 Derived*，与父类的虚函数形参不同。因此，子类不是虚函数的声明。但是书上解释 A 是函数重载，这个说法是错的。

A 选项子类只是重新定义了一个具有不同形参的同名函数而已，并且这个同名函数会屏蔽父类的同名函数。因为派生类的作用域嵌套在基类的作用域中。

B 选项正确，C++ Primer 第四版 P477 页所述：“派生类中虚函数的声明必须与基类中定义方式完全匹配，但是有一个例外，返回对基类型的引用或指针的虚函数。派生类中的虚函数可以返回基类函数所返回类型的派生类的引用或指针”。因此 B 选项就是 P477 页所述的例外，基类虚函数的返回类型是 Base*，而子类虚函数的返回类型是 Derived*，且 Derived 是 Base 的派生类。所以，B 的虚函数声明是正确的。

C 选项正确，虽然基类的虚函数声明中多了一个默认实参，但是依然和子类的虚函数属于同一个函数声明。

D 选项错误，因为 D 的子类的虚函数不是一个 const 函数，和基类的虚函数声明不一致。D 选项也不是函数重载，只是子类重新定义了一个非 const 同名函数而已。

答案：B 和 C

面试例题5：以下程序存在什么问题？该如何修改？[中国著名杀毒软件企业 J 公司 2008 年 4 月面试题]

```
#include <new>
#include <iostream>
using namespace std;
class NamedStr
{
public:
    NamedStr()
    {
        static const char s_szDefaultName[] =
            "Default name";
        static const char s_szDefaultStr[] =
            "Default string";
        strcpy(m_pName, s_szDefaultName);
        strcpy(m_pData, s_szDefaultStr);
    }
}
```

```

    }
    NamedStr(const char *pName, const char
    *pData)
    {
        m_pName = new char[strlen(pName)];
        m_pData = new char[strlen(pData)];
        strcpy(m_pName, pName);
        strcpy(m_pData, pData);
    }
    ~NamedStr(){}
    void Print()
    {
        cout<< "Name: "<< m_pName<< endl;
        cout<< "String: "<< m_pData<< endl;
    }
private:
    char *m_pName;
    char *m_pData;
}

```

```

    };
    int _tmain(int argc, _TCHAR* argv[])
    {
        NamedStr *pDefNss = NULL;
        try
        {
            pDefNss = new NamedStr[10];
            NamedStr ns("Kingsoft string",
            "This is for test.");
            ns.Print();
        }
        catch (...)
        {
            cout<< "Exception!"<< endl;
        }
        delete pDefNss;
        return 0;
    }
}

```

解析：本题有如下几个错误。

- (1) 析构函数中应处理字符指针的释放。
- (2) 应该编写复制构造函数与赋值函数，这是因为类中已经包含了需要深复制的字符指针。
- (3) 这个构造函数：NamedStr(const char *pName, const char *pData) 中，存在为字符指针与内存大小不匹配的错误，应在原来的基础上增加一个字节，用来保存结束符。如 m_pName = new char[strlen(pName) + 1];，并在复制结束后手工增加结束符。另外最好使用较安全的 strncpy 代替 strcpy。
- (4) 默认构造函数 NamedStr() 中对未分配内存空间的字符指针赋值，会引起异常。
- (5) 缺少头文件 tchar.h。

答案：

修改后的程序代码如下：

```

#include <tchar.h> //加此头文件
#include <new>
#include <iostream>
using namespace std;
class NamedStr
{
public:
    NamedStr()
    {
        static const char s_szDefaultName[] =
        "Default name";
        static const char s_szDefaultStr[] =
        "Default string";
    }
}

```

```

m_pName=newchar[strlen(s_szDefaultName)+1];
m_pData=newchar[strlen(s_szDefaultStr)+1];
strcpy(m_pName, s_szDefaultName);
strcpy(m_pData, s_szDefaultStr);

}
NamedStr(const char *pName, const char
*pData)
{
    //空间要加 1，以避免越界
    m_pName = new char[strlen(pName)+1];
}

```

```

    m_pData = new char[strlen(pData)+1];
    strcpy(m_pName, pName);
    strcpy(m_pData, pData);
}

~NamedStr()
{
    //构造函数里动态分配了内存，析构函数里要
    // (delete) 释放内存
    delete[] m_pName;
    delete[] m_pData;
}
void Print()
{
    cout << "Name: " << m_pName << endl;
    cout << "String: " << m_pData << endl;
}
private:
    char *m_pName;
    char *m_pData;
};


```

```

int _tmain(int argc, _TCHAR* argv[])
{
    NamedStr *pDefNss = NULL;
    try
    {
        pDefNss = new NamedStr[10];
        NamedStr ns("Kingsoft string", "This is
for test.");
        ns.Print();
    }
    catch (...)
    {
        cout << "Exception!" << endl;
    }

    delete[] pDefNss;

    return 0;
}


```

10.6 多态的概念

面试例题 1：什么是多态？

答案：

开门，开窗户，开电视。在这里的“开”就是多态！

多态性可以简单地概括为“一个接口，多种方法”，在程序运行的过程中才决定调用的函数。多态性是面向对象编程领域的核心概念。

多态（Polymorphism），按字面的意思就是“多种形状”。多态性是允许你将父对象设置成为和它的一个或更多的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单地说就是一句话，允许将子类类型的指针赋值给父类类型的指针。多态性在 Object Pascal 和 C++ 中都是通过虚函数（Virtual Function）实现的。

扩展知识（多态的作用）

虚函数就是允许被其子类重新定义的成员函数。而子类重新定义父类虚函数的做法，称为“覆盖”（override），或者称为“重写”。这里有一个初学者经常混淆的概念。上面说了覆盖（override）和重载（overload）。覆盖是指子类重新定义父类的虚函数的

做法。而重载，是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。其实，重载的概念并不属于“面向对象编程”。重载的实现是编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数 `function func(p:integer):integer;` 和 `function func(p:string):integer;`。那么编译器做过修饰后的函数名称可能是 `int_func`, `str_func`。对于这两个函数的调用，在编译器间就已经确定了，是静态的（记住：是静态）。也就是说，它们的地址在编译期就绑定了（早绑定），因此，重载和多态无关。真正与多态相关的是“覆盖”。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态（记住：是动态）地调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的（晚绑定）。结论就是重载只是一种语言特性，与多态无关，与面向对象也无关。

引用一句 Bruce Eckel 的话：“不要犯傻，如果它不是晚绑定，它就不是多态。”

那么，多态的作用是什么呢？我们知道，封装可以隐藏实现细节，使得代码模块化；继承可以扩展已存在的代码模块（类）；它们的目的都是为了代码重用。而多态则是为了实现另一个目的——接口重用！而且现实往往是，要有效重用代码很难，而真正最具有价值的重用是接口重用，因为“接口是公司最有价值的资源。设计接口比用一堆类来实现这个接口更费时间，而且接口需要耗费更昂贵的人力和时间”。其实，继承为重用代码而存在的理由已经越来越薄弱，因为“组合”可以很好地取代继承的扩展现有代码的功能，而且“组合”的表现更好（至少可以防止“类爆炸”）。因此笔者个人认为，继承的存在很大程度上是作为“多态”的基础而非扩展现有代码的方式。

那么什么是接口重用？我们举一个简单的例子。假设我们有一个描述飞机的基类如下（Object Pascal 语言描述）：

```
type
  plane = class
  public
    procedure fly(); virtual; abstract;      //起飞纯虚函数
    procedure land(); virtual; abstract;     //着陆纯虚函数
    function modal() : string; virtual; abstract;
    //查询型号纯虚函数
  end;
```

然后，我们从 `plane` 派生出两个子类，直升机（copter）和喷气式飞机（jet）：

```
copter = class(plane)
private
  fModal : String;
```

```
public
  constructor Create();
  destructor Destroy(); override;
```

```

procedure fly(); override;
procedure land(); override;
function modal() : string;
override;
end;

jet = class(plane)
private
fModal : String;

```

```

public
constructor Create();
destructor Destroy(); override;
procedure fly(); override;
procedure land(); override;
function modal() : string;
override;
end;

```

现在，我们要完成一个飞机控制系统。有一个全局的函数 plane_fly，它负责让传递给它的飞机起飞，那么，只需要这样：

```

procedure plane_fly
(const pplane : plane);
begin

```

```

pplane.fly();
end;

```

就可以让所有传给它的飞机（plane 的子类对象）正常起飞。不管是直升机还是喷气机，甚至是现在还不存在的、以后会增加的飞碟。因为，每个子类都已经定义了自己的起飞方式。

可以看到 plane_fly 函数接受的参数是 plane 类对象引用，而实际传递给它的都是 plane 的子类对象。现在回想一下开头所描述的“多态”：多态性是允许你将父对象设置成为和一个或更多的它的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。很显然，parent = child; 就是多态的实质。因为直升机“是一种”飞机，喷气机也“是一种”飞机，因此，所有对飞机的操作都可以对它们操作。此时，飞机类就是一种接口。多态的本质就是将子类类型的指针赋值给父类类型的指针（在 OP 中是引用），只要这样的赋值发生了，多态也就产生了，因为实行了“向上映射”。

应用多态的例子非常普遍。在 Delphi 的 VCL 类库中，最典型的就是 TObject 类有一个虚拟的 Destroy 虚构函数和一个非虚拟的 Free 函数。Free 函数中是调用 Destroy 的。因此，当我们对任何对象（都是 TObject 的子类对象）调用 .Free(); 之后，都会执行 TObject.Free();，它会调用我们所使用的对象的析构函数 Destroy();。这就保证了任何类型的对象都可以正确地被析构。

多态性是面向对象最重要的特性。

面试例题 2：重载和覆盖有什么不同？

答案：虚函数总是在派生类中被改写，这种改写被称为“override”（覆盖）。

override 是指派生类重写基类的虚函数，就像我们前面在 B 类中重写了 A 类中的 foo() 函数。重写的函数必须有一致的参数表和返回值（C++ 标准允许返回值不同的情况，但是很

少有编译器支持这个特性)。Override 这个单词好像一直没有什么合适的中文词汇来对应。有人译为“覆盖”，还贴切一些。

overload 约定成俗地被翻译为“重载”，是指编写一个与已有函数同名但是参数表不同的函数。例如一个函数既可以接收整型数作为参数，也可以接收浮点数作为参数。重载不是一种面向对象的编程，而只是一种语法规则，重载与多态没有什么直接关系。

面试例题 3：which of the following one is NOT resolved at compile time?(下面哪个不能在编译时间被解析)

- | | |
|--------------------|----------------------------------|
| A. Macros | B. Inline functions |
| C. Template in C++ | D. virtual function calls in C++ |

解析：宏，内联函数，模板都可以在编译时候解析，唯独虚函数不行，它必须在运行时才能确定。

答案：D

10.7 友元

面试例题 1：写一个程序，设计一个点类 Point，求两个点之间的距离。[中国软件企业 LC 公司 2007 年 12 月面试题]

解析：本题可以使用友元。

类具有封装和信息隐藏的特性。只有类的成员函数才能访问类的私有成员，程序中的其他函数是无法访问私有成员的。非成员函数可以访问类中的公有成员，但是如果将数据成员都定义为公有的，这又破坏了隐藏的特性。另外，应该看到在某些情况下，特别是在对某些成员函数多次调用时，由于参数传递、类型检查和安全性检查等都需要时间开销，而影响程序的运行效率。

为了解决上述问题，提出一种使用友元的方案。友元是一种定义在类外部的普通函数，但它需要在类体内进行说明，为了与该类的成员函数加以区别，在说明时前面加以关键字 friend。友元不是成员函数，但是它可以访问类中的私有成员。友元的作用在于提高程序的运行效率，但是，它破坏了类的封装性和隐藏性，使得非成员函数可以访问类的私有成员。

友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类。

答案：

代码如下：

```

class Point
{
private:
    float x;
    float y;
public:
    point(float a=0.0f, float b=0.0f):x(a,b){};
    friend float distance(Point& left, Point& right)
};

//最简单的点类的写法
float distance(Point& left, Point& right){
    return ((left.x-right.x)^2+(left.y-right.y)^2)^0.5
}

```

面试例题 2：请描述模板类的友元重载，用 C++代码实现。[中国软件企业 LC 公司 2007 年 12 月面试题]

答案：

代码如下：

```

#include <iostream>
using namespace std;
template <class T>
class Test;
template <class T>
ostream& operator<<(ostream& out,
const Test<T> &obj);
template <class T>
class Test
{
private:
    int num;
public:
    Test(int n=0){num=n;}
    Test(const Test <T> &copy){num=copy.num;}
    friend ostream& operator<< <T>
    (ostream& out,const Test<T> &obj);
}

```

```

//注意在“<<”后加上“<>”表明这是个函数模板
};

template <class T>
ostream& operator<<(ostream& out,
const Test<T> &obj)
{
    out<<obj.num;
    return out;
}
int main()
{
    Test<int> t(2);
    cout<<t;
    return 0;
}

```

10.8 异常

面试例题 1：In C++, you should NOT throw exceptions from: (在 C++ 中，你不应该从以下哪个抛出异常)。[美国著名软件企业 M 公司 2013 年面试题]

- A. Constructor (构造函数)
- B. Destructor (析构函数)
- C. Virtual function (虚方法)
- D. None of the above (以上答案都不对)

解析：构造函数中抛出异常是有一定必要的，试想如下情况：

构造函数中有两次 new 操作，第一次成功了，返回了有效的内存，而第二次失败，此时因为对象构造尚未完成，析构函数是不会调用的，也就是 delete 语句没有被执行，第一次 new 出的内存就悬在那儿了（发生内存泄露），所以异常处理程序可以将其暴露出来。

```
//...
Base()
{
    int* p = new int();
    try{
        int* q = new int(); //假如失败
        //throw 2... //如果直接抛异常，构造函数失败，不执行析构函数
    }
    catch(...){
        delete p; //
        throw;
    }
}
```

构造函数中遇到异常是不会调用析构函数的，一个对象的父对象的构造函数执行完毕，不能称之为构造完成，对象构造是不可分割的，要么完全成功，要么完全失败，C++保证这一点。对于成员变量，C++遵循这样的规则，即会从异常的发生点按照成员变量的初始化的逆序释放成员。举例来说，有如下初始化列表：

```
A::A():m1(),m2(),m3(),m4(),m5()
{...}
```

假定 m3 的初始化过程中抛出异常，则会按照 m2,m1 的顺序调用这两个成员的析构函数。在 {} 之间发生的未捕捉异常，最终会导致在栈的开解时析构所有的数据成员。

处理这样的问题，使用智能指针是最好的，这是因为 auto_ptr 成员是一个对象而不是指针。换句话说，只要不使用原始的指针，那么就不必担心构造函数抛出异常而导致资源泄漏。所以在 C++ 中，资源泄漏的问题一般都用 RAII（资源获取就是初始化）的办法：把需要打开/关闭的资源用简单的对象封装起来（这种封装可以同时有多种用处，比如隐藏底层 API 细节，以利于移植）。这可以省去很多的麻烦。

如果不使用 RAII，即使当前构造函数里获取的东西在析构函数里都释放了，如果某天对类有改动，要新增加一种资源，构造函数里一般能适当地获取，但记不得要在析构函数里相应地释放呢？失误的比率很大。如果考虑到构造函数里抛出异常，就更复杂了。随着项目的不断扩大和时间的推移，这些细节不可能都记得住，而且，有可能会由别人来实施这样的改动。

从运行结果可以得出如下结论：

(1) C++中通知对象构造失败的唯一方法，就是在构造函数中抛出异常；

(2) 对象的部分构造是很常见的，异常的发生点也完全是随机的，程序员要谨慎处理这种情况；

(3) 当对象发生部分构造时，已经构造完毕的子对象将会逆序地被析构（即异常发生点前面的对象）；而还没有开始构建的子对象将不会被构造了（即异常发生点后面的对象），当然它也就没有析构过程了；还有正在构建的子对象和对象本身将停止继续构建（即出现异常的对象），并且它的析构是不会被执行的。

下面再说一下析构函数抛异常的情况。

Effective C++建议，析构函数尽可能地不要抛出异常。设想如果对象出了异常，现在异常处理模块为了维护系统对象数据的一致性，避免资源泄漏，有责任释放这个对象的资源，调用对象的析构函数，可现在假如析构过程又再出现异常，那么请问由谁来保证这个对象的资源释放呢？而且这新出现的异常又由谁来处理呢？不要忘记前面的一个异常目前都还没有处理结束，因此这就陷入了一个矛盾之中，或者说处于无限的递归嵌套之中。所以C++标准就做出了这种假设。看一下如下析构函数抛出异常的例子：

```
#include<iostream>
#include<stdlib.h>
using namespace std;

class Base
{
public:
    void fun()    { throw 1;    }
    ~Base()       { throw 2;    }
};

int main()
```

```
{
    try
    {
        Base base;
        //base.fun(); //注释1
    }
    catch (...)
    {
        //cout <<"get the catch"<<endl;
    }
}
```

运行没有问题。下面打开注释1（//base.fun()；），再试运行，结果程序会崩溃。

为什么呢？

因为SEH是一个链表，链表头地址存在FS:[0]的寄存器里面。函数base.fun先抛出异常，从FS:[0]开始向上遍历SHL节点，匹配到catch块。找到代码里的一个catch块，再去展开栈，调用base的析构函数，然而析构又抛出异常。如果系统再去从SEL链表匹配，会改变FS:[0]值，这时候程序迷失了，不知道下面该怎么办？因为它已经丢掉了上一次异常链的那个节点。

如果把异常完全封装在析构函数内部，不让异常抛出函数之外。程序还会正常运行吗？

```
#include<iostream>
#include<stdlib.h>
using namespace std;
```

```
class Base
{
```

```

public:
    void fun()    { throw 1; }
    ~Base()
    {
        try
        {
            throw 2;
        }
        catch (int e)
        {
            //do something
        }
    }
};

```

```

int main()
{
    try
    {
        Base base;
        base.fun();
    }
    catch (...)
    {
        cout <<"get the catch"<<endl;
    }
}

```

的确可以运行。因为析构抛出来的异常，在到达上一层析构节点之前已经被别的 catch 块给处理掉了。那么当回到上一层异常函数时，其 SEH 没有变，程序可以继续执行。所以“析构函数尽可能地不要抛出异常”。如果非抛不可，语言也提供了方法，就是自己的异常，自己给吃掉。但是这种方法不提倡，有错最好早点报出来。

答案：B

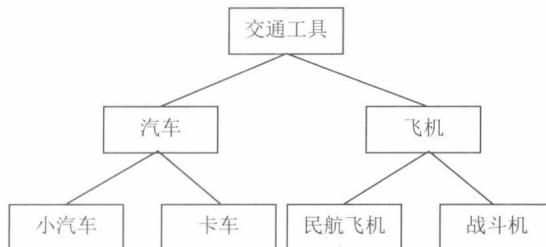
第 11 章

继承与接口

整个 C++ 程序设计全面围绕面向对象的方式进行。类的继承特性是 C++ 的一个非常重要的机制。继承特性可以使一个新类获得其父类的操作和数据结构，程序员只需在新类中增加原有类中没有的成分。

可以说这一章的内容是 C++ 面向对象程序设计的关键。

下面我们简单地说一下继承的概念，先看下图。



上图是一个抽象描述的特性继承表。

交通工具是一个基类（也称作父类）。在通常情况下所有交通工具所共同具备的特性是速度与额定载人的数量。但按照生活常规，我们继续对交通工具进行细分的时候，我们会分别想到汽车类和飞机类等。汽车类和飞机类同样具备速度和额定载人数量这样的特性，而这些特性是所有交通工具所共有的。那么当建立汽车类和飞机类的时候，我们无须再定义基类已经有的数据成员，而只需要描述汽车类和飞机类所特有的特性即可。飞机类和汽车类的特性是由其在交通工具类原有特性的基础上增加而来的，那么飞机类和汽车类就是交通工具类的派生类（也称作子类）。依此类推，层层递增，这种子类获得父类特性的概念就是继承。

一旦成功定义派生类，那么派生类就可以操作基类的所有数据成员，包括受保护型的。甚至我们可以在构造派生类对象的时候初始化它们，但我们不推荐这么做，因为类与类之间

的操作是通过接口进行沟通的，为了不破坏类的这种封装特性，即使是父类与子类的操作也应遵循这个思想。这么做的好处也是显而易见的。当基类有错的时候，只要不涉及接口，那么基类的修改就不会影响到派生类的操作。



至于为什么派生类能够对基类成员进行操作，右图可以简单地说明基类与子类在内存中的排列状态。

我们知道，类对象操作的时候在内部构造时会有一个隐性的 this 指针。由于 Car 类是 Vehicle 的派生类，那么当 Car 对象创建的时候，这个 this 指针就会覆盖到 Vehicle 类的范围，所以派生类能够对基类成员进行操作。

在面试过程中，各大企业会考量你对虚函数、纯虚函数、私有继承、多重继承等知识点的掌握程度。因此，这是本书比较难掌握的一章。

11.1 覆盖

面试例题 1：以下代码的输出结果是什么？[中国著名门户网站 W 公司 2007 年 9 月校园招聘面试题]

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

class A
{
protected:
    int m_data;

public:
    A(int data = 0)
    {m_data = data; }
    int GetData()
    {return doGetData();}
    virtual int doGetData()
    {return m_data;}
};

class B:public A
{
protected:
    int m_data;
public:
    B(int data = 1)
    {m_data = data; }
    int doGetData()
    {return m_data;}
};

};

class C:public B
{
protected:
    int m_data;
public:
    C(int data=2)
    {m_data = data; }
};

int main()
{
    C c(10);

    cout <<c.GetData() <<endl;
    cout <<c.A::GetData() <<endl;
    cout <<c.B::GetData() <<endl;
    cout <<c.C::GetData() <<endl;
    cout <<c.doGetData() <<endl;
    cout <<c.A::doGetData() <<endl;
    cout <<c.B::doGetData() <<endl;
    cout <<c.C::doGetData() <<endl;
    system( "PAUSE ");
    return 0;
}
```

解析：构造函数从最初始的基类开始构造，各个类的同名变量没有形成覆盖，都是单独的变量。理解这两个重要的C++特性后解决这个问题就比较轻松了。下面我们详解这几条输出语句。

```
cout <<c.GetData() <<endl;
```

本来是要调用C类的GetData(), C中未定义，故调用B中的，但是B中也未定义，故调用A中的GetData(), 因为A中的doGetData()是虚函数，所以调用B类中的doGetData(), 而B类的doGetData()返回B::m_data，故输出1。

```
cout <<c.A::GetData() <<endl;
```

因为A中的doGetData()是虚函数，又因为C类中未重定义该接口，所以调用B类中的doGetData(), 而B类的doGetData()返回B::m_data，故输出1。

```
cout <<c.B::GetData() <<endl;
```

肯定返回1了。

```
cout <<c.C::GetData() <<endl;
```

因为C类中未重定义GetData(), 故调用从B继承来的GetData(), 但是B类也未定义，所以调用A中的GetData(), 因为A中的doGetData()是虚函数，所以调用B类中的doGetData(), 而B类的doGetData()返回B::m_data，故输出1。

```
cout <<c.doGetData() <<endl;
```

肯定是B类的返回值1了。

```
cout <<c.A::doGetData() <<endl;
```

因为直接调用了A的doGetData(), 所以输出0。

```
cout <<c.B::doGetData() <<endl;
```

因为直接调用了B的doGetData(), 所以输出1。

```
cout <<c.C::doGetData() <<endl;
```

因为C类中未重定义该接口，所以调用B类中的doGetData(), 而B类的doGetData()返回B::m_data，故输出1。这里要注意存在一个就近调用，如果父辈存在相关接口则优先调用父辈接口，如果父辈也不存在相关接口则调用祖父辈接口。

答案：1 1 1 1 0 1 1。

面试例题2：以下代码的输出结果是什么？[德国某著名电子/通信/IT企业2005年面试题]

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

```
class A {
public:
void virtual f() {
```

```

        cout<<"A"<<endl;
    }
};

class B :public A{
public:
void virtual f() {
    cout<<"B"<<endl;
}
};

int main() {

```

```

A* pa=new A();
pa->f();
B* pb=(B*)pa;
pb->f();

delete pa,pb;
pa=new B();
pa->f();
pb=(B*)pa;
pb->f();

};

```

- A. AABA B. AABB C. AAAB D. ABBA

解析：这是一个虚函数覆盖虚函数的问题。A 类里的 f 函数是一个虚函数，虚函数是被子类同名函数所覆盖的。而 B 类里的 f 函数也是一个虚函数，它覆盖 A 类 f 函数的同时，也会被它的子类覆盖。但是在 B*pb=(B*)pa;里面，该语句的意思是转化 pa 为 B 类型并新建一个指针 pb，将 pa 复制到 pb。但这里有一点请注意，就是 pa 的指针始终没有发生变化，所以 pb 也指向 pa 的 f 函数。这里并不存在覆盖的问题。

delete pa,pb;删除了 pa 和 pb 所指向的地址，但 pa、pb 指针并没有删除，也就是我们通常说的悬浮指针。现在重新给 pa 指向新地址，所指向的位置是 B 类的，而 pa 指针类型是 A 类的，所以就产生了一个覆盖。pa->f();的值是 B。

pb=(B*)pa;转化 pa 为 B 类指针给 pb 赋值，但 pa 所指向的 f 函数是 B 类的 f 函数，所以 pb 所指向的 f 函数是 B 类的 f 函数。pb->f();的值是 B。

答案：B

11.2 私有继承

面试例题 1：Tell me the difference in public inherit and private inherit. (公有继承和私有继承的区别是什么？) [中国某著名计算机金融软件公司 2005 年面试题]

- A. No difference. (没有区别。)
- B. Private inherit will make every member from parent class into private. (私有继承使对象都可以继承，只是不能访问)
- C. Private inherit will make functions from parent class into private. (私有继承使父类中的函数转化成私有。)
- D. Private inherit make every member from parent not-accessible to sub-class. (私有继承使对象不能被派生类的子类访问。)

解析：

A 肯定错，因为子类只能继承父类的 protected 和 public，所以 B 也是错误的。

C 的叙述不全面，而且父类可能有自己的私有方法成员，所以也是错误的。

答案：D

扩展知识

一个私有的或保护的派生类不是子类，因为非公共的派生类不能做基类能做的所有的事。例如，下面的代码定义了一个私有继承基类的类：

```
#include
class Animal
{
public:
    Animal(){}
    void eat(){cout<<"eat\n"; }
};

class Giraffe: private Animal
{
public:
    Giraffe(){}
    void StrecthNeck(double)
    {cout<<"strecth neck \n"; }
};

class Cat: public Animal
```

```
{
Cat(){}
Void Meaw(){cout<<"meaw\n"; }
};

void Func(Animal& an)
{
    an.eat();
}

void main()
{
    Cat dao;
    Giraffe gir;
    Func(dao);
    Func(gir); //error
}
```

函数 Func()要用一个 Animal 类型的对象，但调用 Func(dao)实际上传递的是 Cat 类的对象。因为 Cat 是公共继承 Animal 类，所以 Cat 类中的对象可以使用 Animal 类的所有公有成员变量或函数。Animal 对象可以做的事，Cat 对象也可以做。

但是，对于 gir 对象就不一样。Giraffe 类私有继承了 Animal 类，意味着对象 gir 不能直接访问 Animal 类的成员。其实，在 gir 对象空间中，包含 Animal 类的对象，只是无法让其公开访问。

公有继承就像是三口之家的小孩，享受父母所给的温暖，享有父母的一切（public 和 protected 的成员）。其中保护的成员不能被外界所享有，但可以为小孩所拥有。只是父母还有其一点点隐私（private 成员）不能为小孩所知道。

私有继承就像是离家出走的小孩，一个人在外面漂泊。他（她）不能拥有父母的住房和财产（如 gir.eat()是非法的），在外面自然也就不能代表其父母，甚至不算是其父母的小孩。但是在她（她）的身体中，流淌着父母的血液，所以，在小孩自己的行为中又有其与父母相似的成分。

例如下面的代码中，Giraffe 继承了 Animal 类，Giraffe 的成员函数可以像 Animal 对象那样访问其 Animal 成员：

```
#include
class Animal
{
public:
    Animal(){}
    void eat(){ cout << "eat.\n"; }
};

class Giraffe : private Animal
{
public:
    Giraffe(){}
    void StretchNeck()
        {cout << "stretch neck.\n"; }
```

```
void take(){ eat(); } //ok
};

void Func(Giraffe & an)
{
    an.take();
}

void main()
{
    Giraffe gir;
    gir.StretchNeck();
    Func(gir); //ok
}
```

运行结果为：

```
stretch neck.
eat.
```

上例中，gir 对象就好比是小孩。eat()成员函数是其父母的行为，take()成员函数是小孩的行为，在该行为中，渗透着其父母的行为。但是小孩无法直接使用 eat()成员函数，因为，离家出走的他（她）无法拥有其父母的权力。保护继承与私有继承类似，继承之后的类相对于基类来说是独立的。保护继承的类对象，在公开场合同样不能使用基类的成员。代码如下：

```
#include
class Animal
{
public:
    Animal(){}
    void eat(){ cout << "eat\n"; }
};

class Giraffe: protected Animal
{
    Giraffe(){}
    void StreachNeck(double)
        {cout << "streachneck\n"; }
```

```
void take()
{
    eat(); //ok
};

void main()
{
    Giraffe gir;
    gir.eat(); //error
    gir.take(); //ok
    gir.StretchNeck();
}
```

派生类的 3 种继承方式小结如下。

公有继承（public）、私有继承（private）和保护继承（protected）是常用的 3 种继承方式。

1. 公有继承方式

基类成员对其对象的可见性与一般类及其对象的可见性相同，公有成员可见，其他

成员不可见。这里保护成员与私有成员相同。

基类成员对派生类的可见性对派生类来说，基类的公有成员和保护成员可见，基类的公有成员和保护成员作为派生类的成员时，它们都保持原有的状态；基类的私有成员不可见，基类的私有成员仍然是私有的，派生类不可访问基类中的私有成员。

基类成员对派生类对象的可见性对派生类对象来说，基类的公有成员是可见的，其他成员是不可见的。

所以，在公有继承时，派生类的对象可以访问基类中的公有成员，派生类的成员函数可以访问基类中的公有成员和保护成员。

2. 私有继承方式

基类成员对其对象的可见性与一般类及其对象的可见性相同，公有成员可见，其他成员不可见。

基类成员对派生类的可见性对派生类来说，基类的公有成员和保护成员是可见的，基类的公有成员和保护成员都作为派生类的私有成员，并且不能被这个派生类的子类所访问；基类的私有成员是不可见的，派生类不可访问基类中的私有成员。

基类成员对派生类对象的可见性对派生类对象来说，基类的所有成员都是不可见的。

所以，在私有继承时，基类的成员只能由直接派生类访问，而无法再往下继承。

3. 保护继承方式

这种继承方式与私有继承方式的情况相同。两者的区别仅在于对派生类的成员而言，基类成员对其对象的可见性与一般类及其对象的可见性相同，公有成员可见，其他成员不可见。

基类成员对派生类的可见性对派生类来说，基类的公有成员和保护成员是可见的，基类的公有成员和保护成员都作为派生类的保护成员，并且不能被这个派生类的子类的对象所访问，但可以被派生类的子类所访问；基类的私有成员是不可见的，派生类不可访问基类中的私有成员。

基类成员对派生类对象的可见性对派生类对象来说，基类的所有成员都是不可见的。

C++支持多重继承，从而大大增强了面向对象程序设计的能力。多重继承是一个类从多个基类派生而来的能力。派生类实际上获取了所有基类的特性。当一个类是两个或多个基类的派生类时，必须在派生类名和冒号之后，列出所有基类的类名，基类间用逗号隔开。派生类的构造函数必须激活所有基类的构造函数，并把相应的参数传递给它们。派生类可以是另一个类的基类，这样，相当于形成了一个继承链。当派生类的构造函数被激活时，它的所有基类的构造函数也都会被激活。在面向对象的程序

设计中，继承和多重继承一般指公共继承。在无继承的类中，protected 和 private 控制符是没有差别的。在继承中，基类的 private 对所有的外界都屏蔽（包括自己的派生类），基类的 protected 控制符对应用程序是屏蔽的，但对其派生类是可访问的。

保护继承和私有继承只是在技术上讨论时有其一席之地。

面试例题 2：请考虑标记为 A 到 J 的语句在编译时可能出现的情况。如果能够成功编译，请记为“RIGHT”，否则记为“ERROR”。[中国台湾某著名计算机硬件公司 2005 年 12 月面试题]

```
#include <iostream>
#include <stdio.h>
class Parent
{
public:
    Parent(int var = -1)
    {
        m_nPub = var;
        m_nPtd = var;
        m_nPrt = var;
    }
public:
    int m_nPub;
protected:
    int m_nPtd;
private:
    int m_nPrt;
};

class Child1:public Parent
{
public:
    int GetPub(){return m_nPub;};
    int GetPtd(){return m_nPtd;};
    int GetPrt(){return m_nPrt;};
//A
};

class Child2:protected Parent
{
public:
    int GetPub(){return m_nPub;};
    int GetPtd(){return m_nPtd;};
    int GetPrt(){return m_nPrt;};
//B
};
```

```
class Child3:private Parent
{
public:
    int GetPub(){return m_nPub;};
    int GetPtd(){return m_nPtd;};
    int GetPrt(){return m_nPrt;};
//C
};

int main()
{
    Child1 cd1;
    Child2 cd2;
    Child3 cd3;
    int nVar = 0;
//public inherited
    cd1.m_nPub = nVar;
//D
    cd1.m_nPtd = nVar;
//E
    nVar = cd1.GetPtd();
//F
//protected inherited
    cd2.m_nPub = nVar;
//G
    nVar = cd2.GetPtd();
//H
//private inherited
    cd3.m_nPub = nVar;
//I
    nVar = cd3.GetPtd();
//J
    return 0;
}
```

解析：

A、B、C 都是错误的。因为 m_nPrt 是父类 Parent 的私有变量，所以不能被子类访问。

D 正确。cd1 是公有继承，可以访问并改变父类的公有变量。

E 错误。m_nPtd 是父类 Parent 的保护变量，不可以被公有继承的 cd1 访问，更不可以被修改。虽然 m_nPtd 是父类 Parent 的保护变量，经过公有继承后，m_nPtd 在子类中依然是

Protected，而子类的对象【cd1】是不能访问自身的 protected 成员，只能访问 public 成员。

F 正确。可以通过函数访问父类的保护变量。

G 错误。cd2 是保护继承的，不可以直接修改父类的公有变量。

H 正确。可以通过函数访问父类的保护变量。

I 错误。cd3 是私有继承的，不可以直接修改父类的公有变量。

J 正确。可以通过函数访问父类的保护变量。

答案：

A、B、C、E、G、I 是“ERROR”。

D、F、H、J 是“RIGHT”。

11.3 虚函数继承和虚继承

面试例题 1： Which of the following options describe the expected answer for a class that has 5 virtual functions? (一个类有 5 个虚方法，下列说法正确的是哪项？) [英国某图形软件公司 A2009 年 10 月面试题]

A. Every object of the class holds the address of the first virtual function, and each function in turn holds the address of the next virtual function. (类中的每个对象都有第一个虚方法的地址，每一个方法都有下一个虚方法的地址。)

B. Every object of the class holds the address of a link list object that holds the addresses of the virtual functions. (类中的每个对象都有一个链表用来存虚方法地址。)

C. Every object of the class holds the addresses of the 5 virtual functions. (类中的每个对象都保存 5 个虚方法的地址。)

D. Every object of the class holds the address of a structure that holding the addresses of the 5 virtual functions. (类中的每个对象有一个结构用来保存虚方法地址。)

解析： 每个对象里有虚表指针，指向虚表，虚表里存放了虚函数的地址。虚函数表是顺序存放虚函数地址的，不需要用到链表（link list）。

答案： B

面试例题 2： 下面程序的结果是什么？

```
#include <iostream>
#include <memory.h>
#include<assert.h>
using namespace std;
```

```
class A
{
    char k[3];
public:
```

```

        virtual void aa() {};
};

class B : public virtual A
{
    char j[3];
    //加入一个变量是为了看清楚 class 中的
    //vptr 放在什么位置
public:
    virtual void bb() {};
};

class C : public virtual B
{
    char i[3];
public:

```

```

        virtual void cc() {};
};

int main(int argc,char *argv[])
{
    cout << "sizeof(A) : " << sizeof(A)
        << endl;
    cout << "sizeof(B) : " << sizeof(B)
        << endl;
    cout << "sizeof(C) : " << sizeof(C)
        << endl;
    return 0;
}

```

解析：C++ 2.0 以后全面支持虚函数继承。这个特性的引入为 C++ 增强了不少功能，也引入了不少烦恼。如果能够了解编译器是如何实现虚函数继承，它们在类的内存空间中又是如何布局的，就可以对 C++ 的了解深入不少。

(1) 对于 class A，由于有一个虚函数，那么必须有一个对应的虚函数表来记录对应的函数入口地址。每个地址需标有一个虚指针，指针的大小为 4。类中还有一个 char k[3]，每一个 char 值所占位置是 1，所以 char k[3] 所占大小是 3。做一次数据对齐后（编译器里一般以 4 的倍数为对齐单位），char k[3] 所占大小变为 4。sizeof (A) 的结果就是 char k[3] 所占大小 4 和虚指针所占大小 4，两者之和等于 8。

(2) 对于 class B，由于 class B 虚继承了 class A，同时还拥有自己的虚函数，那么 class B 中首先拥有一个 vptr_B，指向自己的虚函数表。还有 char j[3]，大小为 4。可虚继承该如何实现？首先要通过加入一个虚类指针（记 vptr_B_A）来指向其父类，然后还要包含父类的所有内容。有些复杂，不过还不难想象。sizeof (B) 的结果就是 char k[3] 所占大小 4 和虚指针 vptr_B 所占大小 4 加 sizeof (A) 所占大小 8，三者之和等于 16。

(3) 下面是 class C 了。class C 首先也得有个 vptr_C，然后是 char i[3]，然后是 sizeof (B)，所以 sizeof (C) 的结果就是 char i[3] 所占大小 4 和虚指针 vptr_C 所占大小 4 加 sizeof (B) 所占大小 16，三者之和等于 24。

答案：在 gcc 中打印上面几个类的大小，结果为 8、16、24。

扩展知识

关键字 `virtual` 告诉编译器它不应当完成早绑定，相反，它应当自动安装实现晚绑定所必需的所有机制。这意味着，如果我们对 brass 对象通过基类 instrument 地址调用 `play()`，我们将得到恰当的函数。

为了完成这件事，编译器对每个包含虚函数的类创建一个表（称为 vtable）。在 vtable 中，编译器放置特定类的虚函数地址。在每个带有虚函数的类中，编译器秘密地设置一指针，称为 vpointer（缩写为 vptr），指向这个对象的 vtable。通过基类指针做虚函数调用时（也就是做多态调用时），编译器静态地插入取得这个 vptr，并在 vtable 表中查找函数地址的代码，这样就能调用正确的函数使晚绑定发生。

为每个类设置 vtable、初始化 vptr、为虚函数调用插入代码，所有这些都是自动发生的，所以我们不必担心这些。利用虚函数，这个对象的合适的函数就能被调用，哪怕编译器还不知道这个对象的特定类型。

画图简单表示一下虚表跟踪后的结果，如右图所示。

我们已经知道有一个 vptr，不过 vptr 的位置也许在对象的开始，也许在对象的尾部。所以上面的操作的值应该是 8 或者 12（如果 vptr 在前面的话）。但实际上取回的值被加上了 1。原因是必须要区别一个不指向任何成员的指针和一个指向第一个成员的指针。这里又有点儿不好理解了，举个例子来说明一下。

想象你和你的另外两个朋友合住一个有 3 个房间的别墅，你住在第三间。如果一个你们 3 人共同的朋友来找你玩，你就给他别墅的地址就行了，不用给出你们任意一个人的房间号（不指向任何成员）。但如果你的一个私人朋友来拜访你（这个私人朋友不认识你的那两位朋友），你会给出别墅的地址和你的那个房间号。为了使这个地址有区别，你必须用第一个房间作为偏移量（offset）来表示你的房间位置。你可以对你的朋友说从进门后的第一间房子再往里面走两个房子就是我的房子。如果房子间距是 4，那么第一间到第三间的距离是 8。

如果以上函数稍加变动，不采用虚继承的方式而是直接继承，结果就将不会产生偏移，结果为 8, 12, 16。



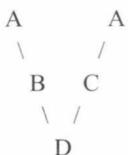
面试例题 3：什么是虚继承？它与一般的继承有什么不同？它有什么用？写出一段虚继承的 C++ 代码。[美国某著名计算机软件公司 2005 年面试题]

答案：

虚拟继承是多重继承中特有的概念。虚拟基类是为解决多重继承而出现的。请看下图：



类 D 继承自类 B 和类 C，而类 B 和类 C 都继承自类 A，因此出现如下图所示这种情况：



在类 D 中会两次出现 A。为了节省内存空间，可以将 B、C 对 A 的继承定义为虚拟继承，而 A 就成了虚拟基类。最后形成如下图所示的情况：



代码如下：

```

class A;
class B : public virtual A;      // virtual inheritance.
class C : public virtual A;
class D : public B, public C;

```

注意：虚函数继承和虚继承是完全不同的两个概念，请不要在面试中混淆。

面试例题 4：为什么虚函数效率低？

答案：因为虚函数需要一次间接的寻址，而一般的函数可以在编译时定位到函数的地址，虚函数（动态类型调用）是要根据某个指针定位到函数的地址。多增加了一个过程，效率肯定会低一些，但带来了运行时的多态。

11.4 多重继承

面试例题 1：请评价多重继承的优点和缺陷。