# 7、C++，C#， 类，内存等

**题目（一）：**我们可以用static修饰一个类的成员函数，也可以用const修饰类的成员函数（写在函数的最后表示不能修改成员变量，不是指写在前面表示返回值为常量）。请问：能不能同时用static和const修饰类的成员函数？

**分析：**答案是不可以。C++编译器在实现const的成员函数的时候为了确保该函数不能修改类的实例的状态，会在函数中添加一个隐式的参数const this\*。但当一个成员为static的时候，该函数是没有this指针的。也就是说此时static的用法和static是冲突的。

我们也可以这样理解：两者的语意是矛盾的。static的作用是表示该函数只作用在类型的静态变量上，与类的实例没有关系；而const的作用是确保函数不能修改类的实例的状态，与类型的静态变量没有关系。因此不能同时用它们。

**题目（二）：**运行下面的代码，输出是什么？

class A

{

};

class B

{

public:

        B() {}

        ~B() {}

};

class C

{

public:

        C() {}

        virtual ~C() {}

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

        printf("%d, %d, %d\n", sizeof(A), sizeof(B), sizeof(C));

        return 0;

}

**分析：**答案是1, 1, 4。class A是一个空类型，它的实例不包含任何信息，本来求sizeof应该是0。但当我们声明该类型的实例的时候，它必须在内存中占有一定的空间，否则无法使用这些实例。至于占用多少内存，由编译器决定。Visual Studio 2008中每个空类型的实例占用一个byte的空间。

class B在class A的基础上添加了构造函数和析构函数。由于构造函数和析构函数的调用与类型的实例无关（调用它们只需要知道函数地址即可），在它的实例中不需要增加任何信息。所以sizeof(B)和sizeof(A)一样，在Visual Studio 2008中都是1。

class C在class B的基础上把析构函数标注为虚拟函数。C++的编译器一旦发现一个类型中有虚拟函数，就会为该类型生成虚函数表，并在该类型的每一个实例中添加一个指向虚函数表的指针。在32位的机器上，一个指针占4个字节的空间，因此sizeof(C)是4。

**题目（三）：**运行下面中的代码，得到的结果是什么？

class A

{

private:

        int m\_value;

public:

        A(int value)

        {

                m\_value = value;

        }

        void Print1()

        {

                printf("hello world");

        }

        void Print2()

        {

                printf("%d", m\_value);

        }

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

        A\* pA = NULL;

        pA->Print1();

        pA->Print2();

        return 0;

}

**分析：**答案是Print1调用正常，打印出hello world，但运行至Print2时，程序崩溃。调用Print1时，并不需要pA的地址，因为Print1的函数地址是固定的。编译器会给Print1传入一个this指针，该指针为NULL，但在Print1中该this指针并没有用到。只要程序运行时没有访问不该访问的内存就不会出错，因此运行正常。在运行print2时，需要this指针才能得到m\_value的值。由于此时this指针为NULL，因此程序崩溃了。

**题目（四）：**运行下面中的代码，得到的结果是什么？

class A

{

private:

        int m\_value;

public:

        A(int value)

        {

                m\_value = value;

        }

        void Print1()

        {

                printf("hello world");

        }

        virtual void Print2()

        {

                printf("hello world");

        }

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

        A\* pA = NULL;

        pA->Print1();

        pA->Print2();

        return 0;

}

**分析：**答案是Print1调用正常，打印出hello world，但运行至Print2时，程序崩溃。Print1的调用情况和上面的题目一样，不在赘述。由于Print2是虚函数。C++调用虚函数的时候，要根据实例（即this指针指向的实例）中虚函数表指针得到虚函数表，再从虚函数表中找到函数的地址。由于这一步需要访问实例的地址（即this指针），而此时this指针为空指针，因此导致内存访问出错。

**题目（五）：**静态成员函数能不能同时也是虚函数？

**分析：**答案是不能。调用静态成员函数不要实例。但调用虚函数需要从一个实例中指向虚函数表的指针以得到函数的地址，因此调用虚函数需要一个实例。两者相互矛盾。

博主何海涛对本博客文章享有版权。网络转载请注明出处<http://zhedahht.blog.163.com/>。整理出版物请和作者联系。

**题目（六）：**运行下列C++代码，输出什么？

struct Point3D

{

        int x;

        int y;

        int z;

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

        Point3D\* pPoint = NULL;

        int offset = (int)(&(pPoint)->z);

        printf("%d", offset);

        return 0;

}

**答案：**输出8。由于在pPoint->z的前面加上了取地址符号，运行到此时的时候，会在pPoint的指针地址上加z在类型Point3D中的偏移量8。由于pPoint的地址是0，因此最终offset的值是8。

&(pPoint->z)的语意是求pPoint中变量z的地址（pPoint的地址0加z的偏移量8），并不需要访问pPoint指向的内存。只要不访问非法的内存，程序就不会出错。

**题目（七）：**运行下列C++代码，输出什么？

class A

{

public:

        A()

        {

                Print();

        }

        virtual void Print()

        {

                printf("A is constructed.\n");

        }

};

class B: public A

{

public:

        B()

        {

                Print();

        }

        virtual void Print()

        {

                printf("B is constructed.\n");

        }

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

        A\* pA = new B();

        delete pA;

        return 0;

}

**答案：**先后打印出两行:A is constructed. B is constructed. 调用B的构造函数时，先会调用B的基类及A的构造函数。然后在A的构造函数里调用Print。由于此时实例的类型B的部分还没有构造好，本质上它只是A的一个实例，他的虚函数表指针指向的是类型A的虚函数表。因此此时调用的Print是A::Print，而不是B::Print。接着调用类型B的构造函数，并调用Print。此时已经开始构造B，因此此时调用的Print是B::Print。

同样是调用虚拟函数Print，我们发现在类型A的构造函数中，调用的是A::Print，在B的构造函数中，调用的是B::Print。因此虚函数在构造函数中，已经失去了虚函数的动态绑定特性。

**题目（八）：**运行下列C#代码，输出是什么？

namespace ChangesOnString

{

    class Program

    {

        static void Main(string[] args)

        {

            String str = "hello";

            str.ToUpper();

            str.Insert(0, " WORLD");

            Console.WriteLine(str);

        }

    }

}

**答案：**输出是hello。由于在.NET中，String有一个非常特殊的性质：String的实例的状态不能被改变。如果String的成员函数会修改实例的状态，将会返回一个新的String实例。改动只会出现在返回值中，而不会修改原来的实例。所以本题中输出仍然是原来的字符串值hello。

如果试图改变String的内容，改变之后的值可以通过返回值拿到。用StringBuilder是更好的选择，特别是要连续多次修改的时候。如果用String连续多次修改，每一次修改都会产生一个临时对象，开销太大。

**题目（九）：**在C++和C#中，struct和class有什么不同？

**答案：**在C++中，如果没有标明函数或者变量是的访问权限级别，在struct中，是public的；而在class中，是private的。

                在C#中，如果没有标明函数或者变量的访问权限级别，struct和class中都是private的。struct和class的区别是：struct定义值类型，其实例在栈上分配内存；class定义引用类型，其实例在堆上分配内存。

**题目（十）：**运行下图中的C#代码，输出是什么？

namespace StaticConstructor

{

    class A

    {

        public A(string text)

        {

            Console.WriteLine(text);

        }

    }

    class B

    {

        static A a1 = new A("a1");

        A a2 = new A("a2");

        static B()

        {

            a1 = new A("a3");

        }

        public B()

        {

            a2 = new A("a4");

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            B b = new B();

        }

    }

}

**答案：**打印出四行，分别是a1、a3、a2、a4。

在调用类型B的代码之前先执行B的静态构造函数。静态函数先初始化类型的静态变量，再执行静态函数内的语句。因此先打印a1再打印a3。接下来执行B b = new B()，即调用B的普通构造函数。构造函数先初始化成员变量，在执行函数体内的语句，因此先后打印出a2、a4。

**题目（11）**：运行下图中的C#代码，输出是什么？

namespace StringValueOrReference

{

    class Program

    {

        internal static void ValueOrReference(Type type)

        {

            String result = "The type " + type.Name;

            if (type.IsValueType)

                Console.WriteLine(result + " is a value type.");

            else

                Console.WriteLine(result + " is a reference type.");

        }

        internal static void ModifyString(String text)

        {

            text = "world";

        }

        static void Main(string[] args)

        {

            String text = "hello";

            ValueOrReference(text.GetType());

            ModifyString(text);

            Console.WriteLine(text);

        }

    }

}

**答案**：输出两行。第一行是The type String is reference type. 第二行是hello。类型String的定义是public sealed class String {...}，既然是class，那么String就是引用类型。

在方法ModifyString里，对text赋值一个新的字符串，此时改变的不是原来text的内容，而是把text指向一个新的字符串"world"。由于参数text没有加ref或者out，出了方法之后，text还是指向原来的字符串，因此输出仍然是"hello".

**题目（12）**：运行下图中的C++代码，输出是什么？

#include <iostream>

class A

{

private:

        int n1;

        int n2;

public:

        A(): n2(0), n1(n2 + 2)

        {

        }

        void Print()

        {

                std::cout << "n1: " << n1 << ", n2: " << n2 << std::endl;

        }

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

        A a;

        a.Print();

        return 0;

}

**答案**：输出n1是一个随机的数字，n2为0。在C++中，成员变量的初始化顺序与变量在类型中的申明顺序相同，而与它们在构造函数的初始化列表中的顺序无关。因此在这道题中，会首先初始化n1，而初始n1的参数n2还没有初始化，是一个随机值，因此n1就是一个随机值。初始化n2时，根据参数0对其初始化，故n2=0。

**题目（13）**：编译运行下图中的C++代码，结果是什么？（A）编译错误；（B）编译成功，运行时程序崩溃；（C）编译运行正常，输出10。请选择正确答案并分析原因。

#include <iostream>

class A

{

private:

        int value;

public:

        A(int n)

        {

                value = n;

        }

        A(A other)

        {

                value = other.value;

        }

        void Print()

        {

                std::cout << value << std::endl;

        }

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

        A a = 10;

        A b = a;

        b.Print();

        return 0;

}

**答案**：编译错误。在复制构造函数中传入的参数是A的一个实例。由于是传值，把形参拷贝到实参会调用复制构造函数。因此如果允许复制构造函数传值，那么会形成永无休止的递归并造成栈溢出。因此C++的标准不允许复制构造函数传值参数，而必须是传引用或者常量引用。在Visual Studio和GCC中，都将编译出错。

**题目（14）**：运行下图中的C++代码，输出是什么？

int SizeOf(char pString[])

{

        return sizeof(pString);

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

        char\* pString1 = "google";

        int size1 = sizeof(pString1);

        int size2 = sizeof(\*pString1);

        char pString2[100] = "google";

        int size3 = sizeof(pString2);

        int size4 = SizeOf(pString2);

        printf("%d, %d, %d, %d", size1, size2, size3, size4);

        return 0;

}

**答案**：4, 1, 100, 4。pString1是一个指针。在32位机器上，任意指针都占4个字节的空间。\*pString1是字符串pString1的第一个字符。一个字符占一个字节。pString2是一个数组，sizeof(pString2)是求数组的大小。这个数组包含100个字符，因此大小是100个字节。而在函数SizeOf中，虽然传入的参数是一个字符数组，当数组作为函数的参数进行传递时，数组就自动退化为同类型的指针。因此size4也是一个指针的大小，为4.

**题目（15）**：运行下图中代码，输出的结果是什么？这段代码有什么问题？

#include <iostream>

class A

{

public:

        A()

        {

                std::cout << "A is created." << std::endl;

        }

        ~A()

        {

                std::cout << "A is deleted." << std::endl;

        }

};

class B : public A

{

public:

        B()

        {

                std::cout << "B is created." << std::endl;

        }

        ~B()

        {

                std::cout << "B is deleted." << std::endl;

        }

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

        A\* pA = new B();

        delete pA;

        return 0;

}

**答案**：输出三行，分别是：A is created. B is created. A is deleted。用new创建B时，回调用B的构造函数。在调用B的构造函数的时候，会先调用A的构造函数。因此先输出A is created. B is created.

接下来运行delete语句时，会调用析构函数。由于pA被声明成类型A的指针，同时基类A的析构函数没有标上virtual，因此只有A的析构函数被调用到，而不会调用B的析构函数。

由于pA实际上是指向一个B的实例的指针，但在析构的时候只调用了基类A的析构函数，却没有调用B的析构函数。这就是一个问题。如果在类型B中创建了一些资源，比如文件句柄、内存等，在这种情况下都得不到释放，从而导致资源泄漏。

**问题（16）**：运行如下的C++代码，输出是什么？

class A

{

public:

    virtual void Fun(int number = 10)

    {

        std::cout << "A::Fun with number " << number;

    }

};

class B: public A

{

public:

    virtual void Fun(int number = 20)

    {

        std::cout << "B::Fun with number " << number;

    }

};

int main()

{

    B b;

    A &a = b;

    a.Fun();

}

**答案**：输出B::Fun with number 10。由于a是一个指向B实例的引用，因此在运行的时候会调用B::Fun。但缺省参数是在编译期决定的。在编译的时候，编译器只知道a是一个类型a的引用，具体指向什么类型在编译期是不能确定的，因此会按照A::Fun的声明把缺省参数number设为10。

            这一题的关键在于理解确定缺省参数的值是在编译的时候，但确定引用、指针的虚函数调用哪个类型的函数是在运行的时候。

**问题（17）**：运行如下的C代码，输出是什么？

char\* GetString1()

{

    char p[] = "Hello World";

    return p;

}

char\* GetString2()

{

    char \*p = "Hello World";

    return p;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

    printf("GetString1 returns: %s. \n", GetString1());

    printf("GetString2 returns: %s. \n", GetString2());

    return 0;

}

**答案**：输出两行，第一行GetString1 returns: 后面跟的是一串随机的内容，而第二行GetString2 returns: Hello World. 两个函数的区别在于GetString1中是一个数组，而GetString2中是一个指针。

当运行到GetString1时，p是一个数组，会开辟一块内存，并拷贝"Hello World"初始化该数组。接着返回数组的首地址并退出该函数。由于p是GetString1内的一个局部变量，当运行到这个函数外面的时候，这个数组的内存会被释放掉。因此在\_tmain函数里再去访问这个数组的内容时，结果是随机的。

当运行到GetString2时，p是一个指针，它指向的是字符串常量区的一个常量字符串。该常量字符串是一个全局的，并不会因为退出函数GetString2而被释放掉。因此在\_tmain中仍然根据GetString2返回的地址得到字符串"Hello World"。

**问题（18）**：运行下图中C#代码，输出的结果是什么？

namespace StaticVariableInAppDomain

{

    [Serializable]

    internal class A : MarshalByRefObject

    {

        public static int Number;

        public void SetNumber(int value)

        {

            Number = value;

        }

    }

    [Serializable]

    internal class B

    {

        public static int Number;

        public void SetNumber(int value)

        {

            Number = value;

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            String assamblyName = Assembly.GetEntryAssembly().FullName;

            AppDomain domain = AppDomain.CreateDomain("NewDomain");

            A.Number = 10;

            String nameOfA = typeof(A).FullName;

            A a = domain.CreateInstanceAndUnwrap(assamblyName, nameOfA) as A;

            a.SetNumber(20);

            Console.WriteLine("Number in class A is {0}", A.Number);

            B.Number = 10;

            String nameOfB = typeof(B).FullName;

            B b = domain.CreateInstanceAndUnwrap(assamblyName, nameOfB) as B;

            b.SetNumber(20);

            Console.WriteLine("Number in class B is {0}", B.Number);

        }

    }

}

**答案**：输出两行，第一行是Number in class A is 10，而第二行是Number in class B is 20。上述C#代码先创建一个命名为NewDomain的应用程序域，并在该域中利用反射机制创建类型A的一个实例和类型B的一个实例。我们注意到类型A是继承自MarshalByRefObject，而B不是。虽然这两个类型的结构一样，但由于基类不同而导致在跨越应用程序域的边界时表现出的行为将大不相同。

      由于A继承MarshalByRefObject，那么a实际上只是在缺省的域中的一个代理，它指向位于NewDomain域中的A的一个实例。当a.SetNumber时，是在NewDomain域中调用该方法，它将修改NewDomain域中静态变量A.Number的值并设为20。由于静态变量在每个应用程序域中都有一份独立的拷贝，修改NewDomain域中的静态变量A.Number对缺省域中的静态变量A.NewDomain没有任何影响。由于Console.WriteLine是在缺省的应用程序域中输出A.Number，因此输出仍然是10。

    B只从Object继承而来的类型，它的实例穿越应用程序域的边界时，将会完整地拷贝实例。在上述代码中，我们尽管试图在NewDomani域中生成B的实例，但会把实例b拷贝到缺省的域。此时，调用b.SetNumber也是在缺省的域上进行，它将修改缺省的域上的A.Number并设为20。因此这一次输出的是20。

**问题（19）**：运行下图中C代码，输出的结果是什么？

int \_tmain(int argc, \_TCHAR\* argv[])

{

    char str1[] = "hello world";

    char str2[] = "hello world";

    char\* str3 = "hello world";

    char\* str4 = "hello world";

    if(str1 == str2)

        printf("str1 and str2 are same.\n");

    else

        printf("str1 and str2 are not same.\n");

    if(str3 == str4)

        printf("str3 and str4 are same.\n");

    else

        printf("str3 and str4 are not same.\n");

    return 0;

}

**答案**：输出两行。第一行是str1 and str2 are not same，第二行是str3 and str4 are same。

str1和str2是两个字符串数组。我们会为它们分配两个长度为12个字节的空间，并把"hello world"的内容分别拷贝到数组中去。这是两个初始地址不同的数组，因此比较str1和str2的值，会不相同。str3和str4是两个指针，我们无需为它们分配内存以存储字符串的内容，而只需要把它们指向"hello world“在内存中的地址就可以了。由于"hello world”是常量字符串，它在内存中只有一个拷贝，因此str3和str4指向的是同一个地址。因此比较str3和str4的值，会是相同的。

**问题（20）**：运行下图中C#代码，输出的结果是什么？并请比较这两个类型各有什么特点，有哪些区别。

namespace Singleton

{

    public sealed class Singleton1

    {

        private Singleton1()

        {

            Console.WriteLine("Singleton1 constructed");

        }

        public static void Print()

        {

            Console.WriteLine("Singleton1 Print");

        }

        private static Singleton1 instance = new Singleton1();

        public static Singleton1 Instance

        {

            get

            {

                return instance;

            }

        }

    }

    public sealed class Singleton2

    {

        Singleton2()

        {

            Console.WriteLine("Singleton2 constructed");

        }

        public static void Print()

        {

            Console.WriteLine("Singleton2 Print");

        }

        public static Singleton2 Instance

        {

            get

            {

                return Nested.instance;

            }

        }

        class Nested

        {

            static Nested() { }

            internal static readonly Singleton2 instance = new Singleton2();

        }

    }

    class Program

    {

        static void Main(string[] args)

        {

            Singleton1.Print();

            Singleton2.Print();

        }

    }

}

**答案**: 输出三行：第一行“Singleton1 constructed”，第二行“Singleton1 Print”，第三行“Singleton2 Print”。

当我们调用Singleton1.Print时，.NET运行时会自动调用Singleton1的静态构造函数，并初始化它的静态变量。此时会创建一个Singleton1的实例，因此会调用它的构造函数。Singleton2的实例是在Nested的静态构造函数里初始化的。只有当类型Nested被使用时，才回触发.NET运行时调用它的静态构造函数。我们注意到我们只在Sington2.Instance里面用到了Nested。而在我们的代码中，只调用了Singleton2.Print。因此不会创建Singleton2的实例，也不会调用它的构造函数。

这两个类型其实都是单例模式（Singleton）的实现。第二个实现Singleton2只在真的需要时，才会创建实例，而第一个实现Singleton1则不然。第二个实现在空间效率上更好。

**问题（21）**：C#是一门托管语言，那么是不是说明只要用C#，就能保证不会出现内存泄露和其他资源泄漏？如果不是，在哪些情况下可能会出现泄漏？

**答案**：C#不能保证没有资源泄漏。比如如下几种情况可能会造成资源泄漏：（1） 调用Native code，比如用P/Invoke或者调用COM；（2） 读写文件时的，没有及时close stream, 或者ADO.NET连数据库时，没有及时关闭连接，也算资源泄漏？（3）注册事件后没有remove，导致publisher和subscriber的强依 赖，垃圾回收可能会被推迟；（4）.NET还定义了一些方法直接申请非托管内存，比如Marshal.AllocHGlobal和Marshal.AllocCoTaskMem。通过这种方式得到的内存，如果没有及时释放，也会造成内存泄露。

**问题（22）**：下面的两段C#有哪些不同？

static void CatchException1()

{

    try

    {

        Function();

    }

    catch

    {

        throw;

    }

}

static void CatchException2()

{

    try

    {

        Function();

    }

    catch (Exception e)

    {

        throw e;

    }

}

**答案**：两个函数的catch都是重新抛出截获的exception，但抛出的exception的call stack是不一样的。对于第一种方法，exception的call stack是从最开始的抛出地点开始的。对于第二种方法，exception的call stack是从CatchException2开始的，最初抛出的地方相关的信息被隐藏了。

**问题（23）**：运行下图中的C++代码，打印出的结果是什么？

bool Fun1(char\* str)

{

    printf("%s\n", str);

    return false;

}

bool Fun2(char\* str)

{

    printf("%s\n", str);

    return true;

}

int \_tmain(int argc, \_TCHAR\* argv[])

{

    bool res1, res2;

    res1 = (Fun1("a") && Fun2("b")) || (Fun1("c") || Fun2("d"));

    res2 = (Fun1("a") && Fun2("b")) && (Fun1("c") || Fun2("d"));

    return res1 || res2;

}

**答案**：打印出4行，分别是a、c、d、a。

在C/C++中，与、或运算是从左到右的顺序执行的。在计算rest1时，先计算Fun1(“a”) && Func2(“b”)。首先Func1(“a”)打印出内容为a的一行。由于Fun1(“a”)返回的是false, 无论Func2(“b”)的返回值是true还是false，Fun1(“a”) && Func2(“b”)的结果都是false。由于Func2(“b”)的结果无关重要，因此Func2(“b”)会略去而不做计算。接下来计算Fun1(“c”) || Func2(“d”)，分别打印出内容c和d的两行。

                在计算rest2时，首先Func1(“a”)打印出内容为a的一行。由于Func1(“a”)返回false，和前面一样的道理，Func2(“b”)会略去不做计算。由于Fun1(“a”) && Func2(“b”)的结果是false，不管Fun1(“c”) && Func2(“d”)的结果是什么，整个表达式得到的结果都是false，因此Fun1(“c”) && Func2(“d”)都将被忽略。

**问题（24）**：运行下面的C#代码，打印出来的结果是什么？

struct Person

{

    public string Name;

    public override string ToString()

    {

        return Name;

    }

}

class Program

{

    static void Main(string[] args)

    {

        ArrayList array = new ArrayList();

        Person jim = new Person() {Name = "Jim"};

        array.Add(jim);

        Person first = (Person)array[0];

        first.Name = "Peter";

        Console.WriteLine(array[0].ToString());

    }

}

**答案**：Person的定义是一个struct，因此是一个值类型。在运行到语句Person first = (Person)array[0]的时候，first是array[0]的一个拷贝，first和array[0]不是一个实例。因此修改first对array[0]没有影响。

**问题（25）**：运行下面的C++代码，打印的结果是什么？

class Base

{

public:

    void print() { doPrint();}

private:

    virtual void doPrint() {cout << "Base::doPrint" << endl;}

};

class Derived : public Base

{

private:

    virtual void doPrint() {cout << "Derived::doPrint" << endl;}

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

    Base b;

    b.print();

    Derived d;

    d.print();

       return 0;

}

**答案**：输出两行，分别是Base::doPrint和Derived::doPrint。在print中调用doPrint时，doPrint()的写法和this->doPrint()是等价的，因此将根据实际的类型调用对应的doPrint。所以结果是分别调用的是Base::doPrint和Derived::doPrint2。如果感兴趣，可以查看一下汇编代码，就能看出来调用doPrint是从虚函数表中得到函数地址的。

**程序员面试题精选**(15)－含有指针成员的类的拷贝

题目：下面是一个数组类的声明与实现。请分析这个类有什么问题，并针对存在的问题提出几种解决方案。

template<typename T> class Array

{

public:

Array(unsigned arraySize):data(0), size(arraySize)

{

if(size > 0)

data = new T[size];

}

~Array()

{

if(data) delete[] data;

}

void setValue(unsigned index, const T& value)

{

if(index < size)

data[index] = value;

}

T getValue(unsigned index) const

{

if(index < size)

return data[index];

else

return T();

}

private:

T\* data;

unsigned size;

};

分析：我们注意在类的内部封装了用来存储数组数据的指针。软件存在的大部分问题通常都可以归结指针的不正确处理。

这个类只提供了一个构造函数，而没有定义构造拷贝函数和重载拷贝运算符函数。当这个类的用户按照下面的方式声明并实例化该类的一个实例

Array A(10);

Array B(A);

或者按照下面的方式把该类的一个实例赋值给另外一个实例

Array A(10);

Array B(10);

B=A;

编译器将调用其自动生成的构造拷贝函数或者拷贝运算符的重载函数。在编译器生成的缺省的构造拷贝函数和拷贝运算符的重载函数，对指针实行的是按位拷贝，仅仅只是拷贝指针的地址，而不会拷贝指针的内容。因此在执行完前面的代码之后，A.data和B.data指向的同一地址。当A或者B中任意一个结束其生命周期调用析构函数时，会删除data。由于他们的data指向的是同一个地方，两个实例的data都被删除了。但另外一个实例并不知道它的data已经被删除了，当企图再次用它的data的时候，程序就会不可避免地崩溃。

由于问题出现的根源是调用了编译器生成的缺省构造拷贝函数和拷贝运算符的重载函数。一个最简单的办法就是禁止使用这两个函数。于是我们可以把这两个函数声明为私有函数，如果类的用户企图调用这两个函数，将不能通过编译。实现的代码如下：

private:

Array(const Array& copy);

const Array& operator = (const Array& copy);

最初的代码存在问题是因为不同实例的data指向的同一地址，删除一个实例的data会把另外一个实例的data也同时删除。因此我们还可以让构造拷贝函数或者拷贝运算符的重载函数拷贝的不只是地址，而是数据。由于我们重新存储了一份数据，这样一个实例删除的时候，对另外一个实例没有影响。这种思路我们称之为深度拷贝。实现的代码如下：

public:

Array(const Array& copy):data(0), size(copy.size)

{

if(size > 0)

{

data = new T[size];

for(int i = 0; i < size; ++ i)

setValue(i, copy.getValue(i));

}

}

const Array& operator = (const Array& copy)

{

if(this == ?)

return \*this;

if(data != NULL)

{

delete []data;

data = NULL;

}

size = copy.size;

if(size > 0)

{

data = new T[size];

for(int i = 0; i < size; ++ i)

setValue(i, copy.getValue(i));

}

}

为了防止有多个指针指向的数据被多次删除，我们还可以保存究竟有多少个指针指向该数据。只有当没有任何指针指向该数据的时候才可以被删除。这种思路通常被称之为引用计数技术。在构造函数中，引用计数初始化为1；每当把这个实例赋值给其他实例或者以参数传给其他实例的构造拷贝函数的时候，引用计数加1，因为这意味着又多了一个实例指向它的data；每次需要调用析构函数或者需要把data赋值为其他数据的时候，引用计数要减1，因为这意味着指向它的data的指针少了一个。当引用计数减少到0的时候，data已经没有任何实例指向它了，这个时候就可以安全地删除。实现的代码如下：

public:

Array(unsigned arraySize)

:data(0), size(arraySize), count(new unsigned int)

{

\*count = 1;

if(size > 0)

data = new T[size];

}

Array(const Array& copy)

: size(copy.size), data(copy.data), count(copy.count)

{

++ (\*count);

}

~Array()

{

Release();

}

const Array& operator = (const Array& copy)

{

if(data == copy.data)

return \*this;

Release();

data = copy.data;

size = copy.size;

count = copy.count;

++(\*count);

}

private:

void Release()

{

--(\*count);

if(\*count == 0)

{

if(data)

{

delete []data;

data = NULL;

}

delete count;

count = 0;

}

}

unsigned int \*count;

}

**程序员面试题精选**100题(30)-异常安全的赋值运算符重载函数

题目：类CMyString的声明如下：

class CMyString

{

public:

CMyString(char\* pData = NULL);

CMyString(const CMyString& str);

~CMyString(void);

CMyString& operator = (const CMyString& str);

private:

char\* m\_pData;

};

请实现其赋值运算符的重载函数，要求异常安全，即当对一个对象进行赋值时发生异常，对象的状态不能改变。

分析：首先我们来看一般C++教科书上给出的赋值运算符的重载函数：

CMyString& CMyString::operator =(const CMyString &str)

{

if(this == &str)

return \*this;

delete []m\_pData;

m\_pData = NULL;

m\_pData = new char[strlen(str.m\_pData) + 1];

strcpy(m\_pData, str.m\_pData);

return \*this;

}

我们知道，在分配内存时有可能发生异常。当执行语句new char[strlen(str.m\_pData) + 1]发生异常时，程序将从该赋值运算符的重载函数退出不再执行。注意到这个时候语句delete []m\_pData已经执行了。也就是说赋值操作没有完成，但原来对象的状态已经改变。也就是说不满足题目的异常安全的要求。

为了满足异常安全这个要求，一个简单的办法是掉换new、delete的顺序。先把内存new出来用一个临时指针保存起来，只有这个语句正常执行完成之后再执行delete。这样就能够保证异常安全了。

下面给出的是一个更加优雅的实现方案：

CMyString& CMyString::operator =(const CMyString &str)

{

if(this != &str)

{

CMyString strTemp(str);

char\* pTemp = strTemp.m\_pData;

strTemp.m\_pData = m\_pData;

m\_pData = pTemp;

}

return \*this;

}

该方案通过调用构造拷贝函数创建一个临时对象来分配内存。此时即使发生异常，对原来对象的状态没有影响。交换临时对象和需要赋值的对象的字符串指针之后，由于临时对象的生命周期结束，自动调用其析构函数释放需赋值对象的原来的字符串空间。整个函数不需要显式用到new、delete，内存的分配和释放都自动完成，因此代码显得比较优雅。

**程序员面试题精选**100题(32)-不能被继承的类

题目：用C++ 设计一个不能被继承的类。

分析：这是Adobe 公司2007 年校园招聘的最新笔试题。这道题除了考察应聘者的C++ 基本功底外，还能考察反应能力，是一道很好的题目。

在Java 中定义了关键字final ，被final 修饰的类不能被继承。但在C++ 中没有final 这个关键字，要实现这个要求还是需要花费一些精力。

首先想到的是在C++ 中，子类的构造函数会自动调用父类的构造函数。同样，子类的析构函数也会自动调用父类的析构函数。要想一个类不能被继承，我们只要把它的构造函数和析构函数都定义为私有函数。那么当一个类试图从它那继承的时候，必然会由于试图调用构造函数、析构函数而导致编译错误。

可是这个类的构造函数和析构函数都是私有函数了，我们怎样才能得到该类的实例呢？这难不倒我们，我们可以通过定义静态来创建和释放类的实例。基于这个思路，我们可以写出如下的代码：

///////////////////////////////////////////////////////////////////////

// Define a class which can't be derived from

///////////////////////////////////////////////////////////////////////

class FinalClass1

{

public :

static FinalClass1\* GetInstance()

{

return new FinalClass1;

}

static void DeleteInstance( FinalClass1\* pInstance)

{

delete pInstance;

pInstance = 0;

}

private :

FinalClass1() {}

~FinalClass1() {}

};

这个类是不能被继承，但在总觉得它和一般的类有些不一样，使用起来也有点不方便。比如，我们只能得到位于堆上的实例，而得不到位于栈上实例。

能不能实现一个和一般类除了不能被继承之外其他用法都一样的类呢？办法总是有的，不过需要一些技巧。请看如下代码：

///////////////////////////////////////////////////////////////////////

// Define a class which can't be derived from

///////////////////////////////////////////////////////////////////////

template <typename T> class MakeFinal

{

friend T;

private :

MakeFinal() {}

~MakeFinal() {}

};

class FinalClass2 : virtual public MakeFinal<FinalClass2>

{

public :

FinalClass2() {}

~FinalClass2() {}

};

这个类使用起来和一般的类没有区别，可以在栈上、也可以在堆上创建实例。尽管类 MakeFinal <FinalClass2> 的构造函数和析构函数都是私有的，但由于类 FinalClass2 是它的友元函数，因此在 FinalClass2 中调用 MakeFinal <FinalClass2> 的构造函数和析构函数都不会造成编译错误。

但当我们试图从 FinalClass2 继承一个类并创建它的实例时，却不同通过编译。

class Try : public FinalClass2

{

public :

Try() {}

~Try() {}

};

Try temp;

由于类 FinalClass2 是从类 MakeFinal <FinalClass2> 虚继承过来的，在调用 Try 的构造函数的时候，会直接跳过 FinalClass2 而直接调用 MakeFinal <FinalClass2> 的构造函数。非常遗憾的是， Try 不是 MakeFinal <FinalClass2> 的友元，因此不能调用其私有的构造函数。

基于上面的分析，试图从 FinalClass2 继承的类，一旦实例化，都会导致编译错误，因此是 FinalClass2 不能被继承。这就满足了我们设计要求。

**程序员面试题精选**（41）：编译器对内存填充长度之误解

看了《C++ 对像模型》的人，往往会误以为编译器填充是按照计算机字长填充的，如下：

class A

{

double a;

char b;

};

sizeof(A) == ?

不了解填充的人会以为是9,看了c++对象模型的（像我）往往会以为是12，昨晚看《程序员面试宝典》一道类似题，开始以为答案给错了。。今天一试才知道，原来我错了。。上题答案（在编译器默认情况下）是 16，VC6.0、MinGW、VS.net均如此。。

《程序员面试宝典》上如是说：CPU的优化原则大致是这样的：对于n字节的元素（n=2、4、8……）它的首地址能被n整除，才能获得最好的性能。设计编译器的时候可以遵循这个原则。也就是说，默认情况下，编译器往往以最大的变量的长度为填充长度，而不是按字节长度。当然也可以通过 #pragma pack(n) 指定编译器的填充长度。这时候应该不是cpu的效率最高的情况了。

另外有个网友讨论说道如果一个类中含有另一个类对象，是否按照包含类的长度填充呢？试验了一下，不是这样，而是按照语言中的基本类型的最大长度填充。没想到，面试题中也会考到这么bt的题目，长见识了。

**程序员面试题精选** 打印内存小技巧收藏

在学习C++的时候，由于编译器背着我们干了太多的事儿，所以看看那些高级数据结构在汇编级别是怎么样的，在内存中是如何的，对编写高效代码很有帮助。

下边是一个小函数，帮你打印内存中的内容。如果使用微软的编译器，各个内存对象之间可能会有byte guard，即编译器会在分配的数据对象之间插入空白bytes，便于检测破坏邻接对象，所以和教科书上的连续分配内存有些差异，注意一下就行了

以下是实现加测试代码：

#include < cstdio >

struct Test{

int a;

char b;

};

void ShowBytes( void \* s, int n)

{

unsigned char \* start = (unsigned char \* )s;

printf( " [OFFSET] ADDRESS: VALUE\n\n " );

for ( int i = 0 ; i < n; i ++ )

{

printf( " [%.4d] %.8X: %.2X\n " , i, start + i, \* (start + i));

if ((i + 1 ) % 4 == 0 )

{

printf( " ----------------------\n " );

}

} // for

}

int main()

{

Test \*t;

Test a={12,'A'};

t=&a;

ShowBytes(t,8);

getchar();

return 0;

}

**程序员面试题精选100题(45)－Singleton(C/C++/C#)**

题目：设计一个类，我们只能生成该类的一个实例。

分析：只能生成一个实例的类是实现了Singleton模式的类型。

由于设计模式在面向对象程序设计中起着举足轻重的作用，在面试过程中很多公司都喜欢问一些与设计模式相关的问题。在常用的模式中，Singleton是唯一一个能够用短短几十行代码完整实现的模式。因此，写一个Singleton的类型是一个很常见的面试题。

事实上，要让一个类型是能创建一个实例不是一件很难的事情。我们可以把该类型的构造函数设为private，这样该类型的用户就不能创建该类型的实例了。然后我们在给类型中创建一个静态实例。当用户需要该类型的实例时，我们就返回这个实例。基于这个思路，我们可以用C#写出如下代码：

    // We can only get an instance of the class Singleton1. The instance

    // is created when class Singleton1 is referenced at the first time

    public sealed class Singleton1

    {

        private Singleton1()

        {

        }

        private static Singleton1 instance = new Singleton1();

        public static Singleton1 Instance

        {

            get

            {

                return instance;

            }

        }

    }

由于类Singleton1 的实例是一个静态变量，因此它会在该类型的第一次引用的时候被创建，而不是第一次在调用Singleton1.get\_Instance的时候被创建。如果我们此时并不需要该实例，那么我们就过早地初始化该实例，无论在内存空间还是CPU时间上都是一种浪费。

我们可以把上面的代码稍作改动，就能实现在第一次调用Singleton\_getInstance时候才会创建类型的唯一实例：

    // We can only get an instance of the class Singleton2.

    // The instance is created when we need it explicitly.

    public sealed class Singleton2

    {

        private Singleton2()

        {

        }

        private static Singleton2 instance = null;

        public static Singleton2 Instance

        {

            get

            {

                if (instance == null)

                    instance = new Singleton2();

                return instance;

            }

        }

    }

我们在单线程环境下只能得到类型Singleton2的一个实例，但在多线程环境下情况就可能不同了。设想如果两个线程同时运行到语句if (instance == null)，而此时该实例的确没有创建，那么两个线程都会创建一个实例。此时，类型Singleton2就不在满足模式Singleton的要求了。

为了保证在多线程环境下我们还是只能得到类型的一个实例，我们应该在判断实例是否已经创建，以及在实例还没有创建的时候创建一个实例的语句上加一个同步锁。我们把Singleton2稍作修改就得到了如下代码：

    // We can only get an instance of the class Singleton3,

    // even when there are multiple threads which are trying

    // to get an instance concurrently.

    public sealed class Singleton3

    {

        private Singleton3()

        {

        }

        private static readonly object syncObj = new object();

        private static Singleton3 instance = null;

        public static Singleton3 Instance

        {

            get

            {

                lock (syncObj)

                {

                    if (instance == null)

                        instance = new Singleton3();

                }

                return instance;

            }

        }

    }

说明一下，由于C/C++没有为线程同步提供直接的支持。为了让代码显得简洁，而不是让大量的代码在实现同步锁而偏离了实现Singleton的主题，本文的代码用C#实现。

我们还是假设有两个线程同时想创建一个实例。由于在一个时刻只能有一个线程能得到同步锁。当第一个线程加上锁时，第二个线程只能在等待。当第一个线程发现实例还没有创建时，它创建出一个实例。接着第一个线程释放同步锁。此时第二个线程可以加上同步锁，并运行接下来的代码。由于此时实例已经被第一个线程创建出来了，第二个线程就不会重复创建实例了。于是保证了我们只能得到一个实例。

但是类型Singleton3还不是完美。由于我们每次调用Singleton3.get\_Instance的时候，都会试图加上一个同步锁。由于加锁是一个非常耗时的操作，在没有必要的时候我们应该尽量避免这样的操作。

实际上，我们只是在实例还没有创建之前需要加锁操作，以保证只有一个线程创建出实例。而当实例已经创建之后，我们已经不需要再做加锁操作了。于是，我们可以把上述代码再作进一步的改进：

    // We can only get an instance of the class Singleton4,

    // even when there are multiple threads which are trying

    // to get an instance concurrently. When the instance has

    // been created, we don't need the lock any more.

    public sealed class Singleton4

    {

        private Singleton4()

        {

        }

        private static object syncObj = new object();

        private static Singleton4 instance = null;

        public static Singleton4 Instance

        {

            get

            {

                if (instance == null)

                {

                    lock (syncObj)

                    {

                        if (instance == null)

                            instance = new Singleton4();

                    }

                }

                return instance;

            }

        }

    }

 我们只需要在最开始调用Singleton4\_getInstance（可能来自一个线程，也可能来自多个线程）的时候需要加锁。当实例已经创建之后，我们就不再需要作加锁操作，从而在后续调用Singleton4\_getInstance时性能得到提升。