第二讲：C C++基础（一）

1）请简述智能指针原理，并实现一个简单的智能指针

答：智能指针是一种资源管理类，通过对原始指针进行封装，在资源管理对象进行析构时对指针指向的内存进行释放；通常使用引用计数方式进行管理。

class Object; class SmartPointer; class Counter {  friend class SmartPointer; public:  Counter()  {   ptr = NULL;   cnt = 0;  }  Counter(Object\* p)  {   ptr = p;   cnt = 1;  }  ~Counter()  {   delete ptr;  } private:  Object\* ptr;  int cnt; }; class SmartPointer { public:  SmartPointer(Object\* p)  {   ptr\_counter = new Counter(p);  }  SmartPointer(const SmartPointer &sp)  {   ptr\_counter = sp.ptr\_counter;   ++ptr\_count->cnt;  }  SmartPointer& operator=(const SmartPointer &sp)  {   ++sp.ptr\_counter->cnt;   --ptr\_counter->cnt;   if (ptr\_counter->cnt == 0)   {    delete ptr\_counter;   }   ptr\_counter = sp.ptr\_counter;  }  ~SmartPointer()  {   - -ptr\_counter->cnt;   if (ptr\_counter->cnt == 0)   {    delete ptr\_counter;   }  } private:  Counter\* ptr\_counter; };

2）如何处理循环引用问题

答：

      循环引用：两个类互相引用，导致单纯互相引用头文件，编译无法通过。

      解决方案：声明和实现之间的差别。声明只需要在使用类的前面添加class \*\*\*，而实现是包含具体成员函数和变量的类。

//"A.h"

class B;

class A {

...

private:

    B\* b;

}

//"B.h"

#include <A.h>

using A;

class B {

...

private:

    A\* a;

}

3）请实现一个单例模式的类，要求线程安全

答：单例模式：是设计模式中比较简单的一种，适合于一个类只有一个实例的情况，比如窗口管理，打印缓冲池和文件系统

单例模式特点：

a.确保一个类只有一个实例被创建

b.在不影响单例类的客户端的情况下允许将来有多个实例，经典的模式:懒汉式，饿汉式，登记式

c.提供了一个对全局对象的全局访问指针

模式：

(1).懒汉式：延迟加载，也就是说实例直到第一次用到的时候才会创建（时间来换空间），并在以后仅返回此实例

　　需要用锁来保证线程的安全性 原因：多个线程可能进入判断是否已经存在实例的if语句，从而不安全

　　使用double-check来保证线程安全，但是处理大量的数据的时候，该锁会成为严重的性能瓶颈

(a).静态成员实例的懒汉模式

class Singleton

{

private:

    static Singleton\* m\_instance;   //静态成员实例

    Singleton(){}

public:

    static Singleton\* getInstance();  //声明

};

Singleton\* Singleton::getInstance()

{

    if(NULL == m\_instance)      //第一次检查是否创建

    {

        Lock();//借用其它类来实现，如boost，加锁

        if(NULL == m\_instance)    //再次判断是否创建

        {

            m\_instance = new Singleton;    //创建

        }

        UnLock();   //解锁

    }

    return m\_instance;   //返回实例

}

(b).内部静态实例的懒汉模式

class SingletonInside

{

private:

    SingletonInside(){}

public:

    static SingletonInside\* getInstance()

    {

        Lock(); // c++0x之后不用加锁

        static SingletonInside instance;  //内部静态实例

        UnLock(); // c++0x之后不用

        return instance;

    }

};

(2).饿汉式：一开始就创建了实例，所以后仅返回此实例（空间换时间）

由静态初始化实例保证线程安全性，因为静态实例初始化在程序开始时进入主函数之前就由主线程以单线程的方式完成了初始化，所以不必担心多线程的问题，所以要求高性能时，应使用这种模式，避免繁杂的锁争夺。

class SingletonStatic

{

private:

    static const SingletonStatic \*m\_instance;  //声明实例

    SingletonStatic(){}

public:

    static SingletonStatic\* getInstance()

    {

        return instance;

    }

};

//注意这里不是进行创建对象，而是进行上述声明对象的初始化 ，所以必然在main之前

const SingletonStatic \* SingletonStatic::m\_instance =new SingletonStatic;

思考？为什么非要用 const static ？

答案：为了更好地线程安全，因为const表示的是一旦被赋值，不能修改，static用来保证单例进入main函数之前就已经被初始化了。

(3).登记式

登记式实际上是对于一组单例模式进行的维护，主要是在数量上的扩展，通过map我们把单例存进去，然后在使用的时候判断实例是否存在，如果不存在，则创建一个存入map中，再返回，如果存在直接返回，在数量上分又分为固定数量的和不固定数量的。

登记类单例类为了克服饿汉式单例类以及懒汉式单例类均不可继承的缺点设计的，只是它的子类实例化只能是懒汉式的，

缺点：a).由于子类必须允许父类以构造函数调用产生实例，这样它的构造函数必须是公开的，这样一来就等于允许以这样的方式产生实例，而不在父类的登记中; b).父类的实例必须存在才能有子类的实例，这在有些情况下是一个浪费。

public class RegSingleton {

 //登记薄，用来存放所有的实例

    private static Map regSingletonMap = new HashMap();

    //在类加载的时候添加一个实例到登记薄

    static{

     RegSingleton regSingleton = new RegSingleton();

     regSingletonMap.put(regSingleton.getClass().getName(),regSingleton);

    }

    //保护的默认构造函数

    protected RegSingleton(){

    }

    public static RegSingleton getInstance(String name){

       if(name == null){

         name = "RegSingleton";

       }

       //如果没有找到实例，那就创建一个实例，加入map并返回

       if(regSingletonMap.get(name) == null){

        regSingletonMap.put(name, (RegSingleton)Class.forName(name).newInstance());

       }

       return regSingletonMap.get(name);

    }

}

4）如何定义一个只能在堆上生成对象的类

5）如何定义一个只能在栈上生成对象的类

答：   在C++中，类的对象建立分为两种，一种是静态建立，如A a；另一种是动态建立，如A\* ptr=new A；这两种方式是有区别的。

        静态建立一个类对象，是由编译器为对象在栈空间中分配内存，是通过直接移动栈顶指针，挪出适当的空间，然后在这片内存空间上调用构造函数形成一个栈对象。使用这种方法，直接调用类的构造函数。

        动态建立类对象，是使用new运算符将对象建立在堆空间中。这个过程分为两步，第一步是执行operator new()函数，在堆空间中搜索合适的内存并进行分配；第二步是调用构造函数构造对象，初始化这片内存空间。这种方法，间接调用类的构造函数。

        那么如何限制类对象只能在堆或者栈上建立呢？下面分别进行讨论。

1、只能建立在堆上

      类对象只能建立在堆上，就是不能静态建立类对象，即不能直接调用类的构造函数。

      容易想到将构造函数设为私有。在构造函数私有之后，无法在类外部调用构造函数来构造类对象，只能使用new运算符来建立对象。然而，前面已经说过，new运算符的执行过程分为两步，C++提供new运算符的重载，其实是只允许重载operator new()函数，而operator()函数用于分配内存，无法提供构造功能。因此，这种方法不可以。

      当对象建立在栈上面时，是由编译器分配内存空间的，调用构造函数来构造栈对象。当对象使用完后，编译器会调用析构函数来释放栈对象所占的空间。编译器管理 了对象的整个生命周期。如果编译器无法调用类的析构函数，情况会是怎样的呢？比如，类的析构函数是私有的，编译器无法调用析构函数来释放内存。所以，编译 器在为类对象分配栈空间时，会先检查类的析构函数的访问性，其实不光是析构函数，只要是非静态的函数，编译器都会进行检查。如果类的析构函数是私有的，则 编译器不会在栈空间上为类对象分配内存。

        因此，将析构函数设为私有，类对象就无法建立在栈上了。代码如下：

class A

{

public:

    A(){}

    void destory(){delete this;}

private:

    ~A(){}

};

      试着使用A a;来建立对象，编译报错，提示析构函数无法访问。这样就只能使用new操作符来建立对象，构造函数是公有的，可以直接调用。类中必须提供一个destory函数，来进行内存空间的释放。类对象使用完成后，必须调用destory函数。

       上述方法的一个缺点就是，无法解决继承问题。如果A作为其它类的基类，则析构函数通常要设为virtual，然后在子类重写，以实现多态。因此析构函数不 能设为private。还好C++提供了第三种访问控制，protected。将析构函数设为protected可以有效解决这个问题，类外无法访问 protected成员，子类则可以访问。

       另一个问题是，类的使用很不方便，使用new建立对象，却使用destory函数释放对象，而不是使用delete。（使用delete会报错，因为delete对象的指针，会调用对象的析构函数，而析构函数类外不可访问）这种使用方式比较怪异。为了统一，可以将构造函数设为protected，然后提供一个public的static函数来完成构造，这样不使用new，而是使用一个函数来构造，使用一个函数来析构。代码如下，类似于单例模式：

class A

{

protected:

    A(){}

    ~A(){}

public:

    static A\* create()

    {

        return new A();

    }

    void destory()

    {

        delete this;

    }

};

  这样，调用create()函数在堆上创建类A对象，调用destory()函数释放内存。

2、只能建立在栈上

       只有使用new运算符，对象才会建立在堆上，因此，只要禁用new运算符就可以实现类对象只能建立在栈上。将operator new()设为私有即可。代码如下：

class A

{

private:

    void\* operator new(size\_t t){}     // 注意函数的第一个参数和返回值都是固定的

    void operator delete(void\* ptr){} // 重载了new就需要重载delete

public:

    A(){}

    ~A(){}

};

6）下面的结构体大小分别是多大（假设32位机器）

struct A {

char a;

char b;

char c;

};//3

struct B {

int a;

char b;

short c;

};//8

struct C {

char b;

int a;

short c;

};//12

#pragma pack(2)

struct D {

char b;

int a;

short c;

};//8

7）引用和指针有什么区别

答：a).引用一旦定义不能修改，而指针可以指向其他变量，是可以修改的（引用只能在定义时被初始化一次，之后不可变；指针可变）；

       b).引用是变量的别名，而指针是指向变量的变量（引用只是一个别名，而指针是一个实体）；

       c).引用不能多级引用，而指针可以多级指针；

       d).引用比指针更加安全，一般情况推荐使用引用；

       e).引用使用时无需解引用（\*），而指针需要；

        f).引用不能为空，而指针可以为空；

        g).“sizeof引用”得到的是所指向变量（对象）的大小，而“sizeof指针”得到的是指针本身（所指向的变量或对象的地址）的大小；

        h).指针和引用的自增（++）运算意义不一致；

         i).从内存分配上看：程序为指针变量分配内存区域，而引用不需要分配内存区域。

      相同点：都是地址的概念（指针指向一块内存，它的内容是所指内存的地址；引用是某块内存的别名）。

8） const和define有什么区别

答：const是c++中的修饰符，c++中常用来定义变量，修饰左值；define是宏定义语句，在预处理阶段直接做文本替换，不做类型检查。

      a).对于const变量，系统只给了一个相应的内存地址，而define则是给出了一个立即数，因为const变量是存放在内存静态区域中，所以在程序运行过程中const变量只是一个拷贝，而define所定义的宏变量却有多个拷贝，所以宏在程序运行过程中所消耗的内存要比const变量的大得多。define所定义的宏变量在预编译的时候进行替换，const变量在编译时确定其值。const enum变量进符号表，可在调试中查看变量的值，而define不行；

       b).const对数据进行类型检查。define无需进行类型检测。定义宏参数时需要将参数（）起来，而且调用时参数不应在宏中再做运算；

       c).除了定义变量外，const还有强大的修饰功能，define能增加程序的可读性，有些复杂的功能只需一条宏显示。

9） define和inline有什么区别

答：define是定义预编译时处理的宏，只进行简单的字符替换，无类型检测；inline内联函数对编译器提出建议，是否进行宏替换，编译器有权拒绝。

        a).define是由预处理器对宏进行替代，inline是通过编译器控制来实现的；

        b).内联函数是真正的函数，只是在需要用时，内联函数像宏一样展开，所以取消了函数的参数压栈，减少了调用的开销；

         c).define的定义很容易产生二义性，inline很好的避免了这个问题；

         f).defiine不能访问对象的私有成员，inline可以访问。

       inline具有局限性，就是函数中的执行代码不能太多了，如果内联函数的函数体过大，一般编译器会放弃内联方式，而采取普通方式调用函数。

10） malloc和new有什么区别

答：a)malloc是一个C++/C语言的标准库函数，new是C++的运算符，它们都可以用于申请动态内存 ；

       b)malloc需要自己计算开辟空间大小，new会自动计算；

       c)malloc开辟的用free释放，new开辟的用delete释放。

       d).对于非内部数据类型的对象而言，光用malloc无法满足动态对象的要求，对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数的任务强加到malloc，因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new；

        e).new可以认为是malloc加构造函数的执行，new出来的指针是直接带类型信息的，而malloc返回的都是void指针。

11） C++中static关键字作用有哪些

答：a).static总是使用变量或对象的存储形式变成静态存储，连接方式变成内部连接，对于局部变量（已经是内部连接），它仅改变其存储方式；

        b).对于全局变量（已经是静态存储），它仅改变其连接类型（连接方式：成为内部连接；存储形式：存放在静态全局存储区）。

12） C++中const关键字作用有哪些

答：a).const用于指针的两种情况的分析：

          int const \* A;//A可变，\*A不可变

          int \* const  A;//A不可变，\*A可变

      const是一个左结合的类型修饰符，它与其左侧的类型修饰符和为一个类型修饰符。

        b).const限定函数的传递值参数：

           void Fun(const int Val);

           上述写法限定函数在函数体中不可被改变。

         c).const限定函数的返回值类型：

            const int Fun1();

            const MyClass Fun2();

       上述写法限定函数的返回值不可被更新，当函数返回内部的类型时（如Fun1），已经是一个数值，当然不可以被赋值更新，所以，此时const无意义，最好去掉，以免困惑。当函数返回自定义的类型时（如Fun2），这个函数仍然包含可以被赋值的变量成员，所以，此时有意义；

          d).传递和返回地址；

          e).const限定类的成员函数：

             class ClassName{

                public:

                     int Fun() const;

                 ……

             }

         注意：采用此种const后置的形式是一种规定，亦为了不引起混淆，在此函数声明中和定义中均要使用const，因为const已经成为类型信息的一部分。

13 ） C++中包含哪几种强制类型转换？他们有什么区别和联系

答：四种

    （1）static\_cast： 在功能上基本上与C风格的类型转换一样强大，含义也一样。它有功能上的限制。例如，你不能用static\_cast像用C风格转换一样把struct转换 成int类型或者把double类型转换成指针类型。另外，static\_cast不能从表达式中去除const属性，因为另一个新的类型转换符 const\_cast有这样的功能。

      可以静态决议出类型的转换可能性，即使是在继承体系中，即使包括了多重继承和虚继承，只要可以进行静态决议就可以转换成功 。

    （2）const\_cast：用于类型转换掉表达式的const或volatile属性。通过使用const\_cast，你向人们和编译器强调你通过类型 转换想做的只是改变一些东西的constness或者volatieness属性。这个含义被编译器所约束。如果你试图使用const\_cast来完成修 改constness或者volatileness属性之外的事情，你的类型转换将被拒绝。

    （3）dynamic\_cast：它被用于安全地沿着类的继承关系向下进行类型转换。这就是说，你能用dynamic\_cast把指向基类的指针或引用转 换成指向其派生类或其兄弟类的指针或引用，而且你能知道转换是否成功。失败的转换将返回空指针（当对指针进行类型转换时）或者抛出异常（当对引用进行类型 转换时）。

    （4）reinterpret\_cast：使用这个操作符的类型转换，其转换结果几乎都是执行期定义。因此，使用reinterpret\_cast的代码很难移植。reinterpret\_casts的最普通的用途就是在函数指针类型之间进行转换。