

More Exceptional C++ 笔记

By Ken 2010-11-29

(一) 泛型程序设计与 C++ 标准库

- ✧ **原则：** 尽量提高可读性，避免撰写过度精简的代码（即，简洁但难以理解和代码）。避免晦涩。
- ✧ 均衡的判断力是有经验的程序员的一个特征。尤其是，在“编写专用代码只解决当前问题”（短视，难以扩充）和“编写一个宏大的通用框架去解决本来应该很简单的问题”（追求过度设计）之间，有经验的程序员懂得如何去获得最佳的平衡。
- ✧ **原则：** 尽量提高封装性，将关系分离。只要有可能，一段代码——函数或类——应该只知道并且只负责一件事。
- ✧ 编写代码时，考虑使用两种多态技术：编译时多态（模板），运行时多态（虚函数）。有些时候选择模板具有更好的关系分离。
- ✧ `std::remove()` 算法完成了什么功能？请明确回答。

- 从物理意义上来说，标准算法 `remove()` 没有将对象从容器中删除；`remove()` 执行完成后，容器的大小不变。更确切地说，`remove()` 只是简单地使用“为删除”对象来填补被删除对象留下的缺口，每一个被删除对象在尾部还是会留下一个相应的“死亡”对象；或者，如果没有对象被删除，`remove()` 将返回 `end()` 迭代器。

For example, consider a `vector<int> v` that contains the following nine elements:



1 2 3 1 2 3 1 2 3

Say that you used the following code to try to remove all 3's from the container:

// Example 2-1

```
remove( v.begin(), v.end(), 3 ); // subtly wrong
```

What would happen? The answer is something like this:

1	2	1	2	1	2	?	?	?
								
unremoved						"dead"		
						objects		
						↑		
						iterator returned by		
						remove() points to		
						the third-last object		
						(because three were		
						removed)		

三个对象必须得被删除，剩余的对象会被拷贝，以填充它们留下的缺口。容器尾部的对象可能还是保持原始值（1 2 3），也可能不是——所以不要相信这一假设。在次注意，容器的大小没变。

`remove()` 为什么要以这样的方式工作。其根本原因在于：`remove()` 不是作用在容器身上，而是作用在迭代器区间（range）上，“从任何一种容器中删除迭代

器所指元素”这样的迭代器操作是不存在的。要想这样做，我们必须真正、直接地得到容器。

- ✧ 写一段代码，用来删除 `std::vector<int>` 中值等于 3 的所有元素。

```
// Example 2-2: Removing 3's from a vector<int> v
v.erase( remove( v.begin(), v.end(), 3 ), v.end() );
```

- ✧ 用 `advance(first, n)` 来步进迭代器。它可以分辨迭代器类别 (iterator categories)；针对随机访问迭代器，它能自动地进行优化。具体地讲，针对随机访问迭代器，它只需要消耗常数级时间，而对于其它迭代器，它需要消耗线性级时间。

- ✧ 什么是 predicate？

- Predicate（谓词）是一个函数指针或函数对象（一个提供了函数调用运算符 `operator()` 对象）；针对一个“关于对象的提问”，它做出“是”或“否”的回答。

- 注意，predicate 只能通过“被解引用的迭代器”来使用 `const` 函数。

- ✧ 有些人尝试写一些所谓的“状态性 (stateful)” predicate，这些 predicate 在功能上更进一步：它们在使用时会发生变化——也就是说，在使用一个 predicate 时，其使用结果依赖于前一次使用的历史情况。这种状态性 predicate 是不能可移植和 STL 算法一起使用（非状态性 predicate 函数对象携带的内部值在对象构造时固定并且在该对象的生命周期内是不可变的，也即所有该对象的拷贝都是等价的；而状态性 predicate 对象的拷贝和它自己是不等价的。）

- ✧ 要让状态性 predicate 对算法真正有用，在如何使用 predicate 方面，算法一般得保证两点：

- 算法绝对不能对 predicate 做复制（即，自始至终只能使用同一给定对象）。

- 算法必须“以某个已知的顺序”将 predicate 作用到区间里面的元素上。（通常是第一个元素到最后一个元素）。

可惜的是，C++ 标准要求标准算法提供以上两个保证。

一个状态性 predicate 的例子。没有办法让 `FlagNth` 这样的 predicate 和 C++ 标准算法一起可靠的工作。

```
// Method 2: Write a function object which returns
```

```
// true the nth time it's applied, and use
```

```
// that as a predicate for remove_if.
```

```
class FlagNth
```

```
{
```

```
public:
```

```
    FlagNth( size_t n ) : current_(0), n_(n) { }
```

```
    template<typename T>
```

```
    bool operator()( const T& ) { return ++current_ == n_; }
```

```
private:
```

```
    size_t current_;
```

```
    const size_t n_;
```

```
};
```

- ✧ 什么是 traits 类？

- C++ 定义，一个 traits 类是：一个封装了一组类型 (types) 与函数 (functions) 的类，以使模板类与模板函数可以操纵实例化类型的对象。

- 基本概念是：traits 类是模板实例，用于携带 traits 模板被实例化时类型的额外信息——特别是，可以被其它模板使用的信息。它所带来的好处是：在对某个

类 C 不做任何修改的情况下，T<C>这个 traits 类能让我们记录 C 的（以上所说的那种）额外信息。

- ✧ 示范如何检测和运用模板参数的成员，请使用下面这个具有启发性的例子：假设你想写一个类模板 C，能够实例化此模板的类型必须具有一个名为 Clone()的 const 成员函数，此函数不带参数，返回值为指针，指向同种类型的对象（T* T::Clone() const）。

```
// Example 4-2(d): Alternative way of requiring
// exactly T* T::Clone() const
// T must provide T* T::Clone() const
template<typename T>
class C
{
    bool ValidateRequirements() const
    {
        T* (T::*test)() const = &T::Clone;    ← good trick
        test; // suppress warnings about unused variables
        // ...
        return true;
    }
public:
    // in C's destructor (easier than putting it
    // in every C constructor):
    ~C()
    {
        assert( ValidateRequirements() );
    }
    // ...
};
```

```
// Example 4-2(e): Using constraint inheritance
// to require exactly T* T::Clone() const
// HasClone requires that T must provide
// T* T::Clone() const
template<typename T>
class HasClone
{
public:
    static void Constraints()
    {
        T* (T::*test)() const = &T::Clone;
        test; // suppress warnings about unused variables
    }
    HasClone() { void (*p)() = Constraints; }
};
```

Now we have an elegant?dare I say "cool" way to enforce the constraint at compile time:

```
template<typename T>
class C : HasClone<T>
{
    // ...
};
```

- ✧ 在一个模板中，只有被实际使用了的成员函数才会把实例化。
- ✧ 某程序员想写一个模板，此模板可以要求（或者检测出）：它在被实例化时所使用的类型具有一个 `Clone()`成员函数。这个程序员采用的方案基于这样一个要求：提供 `Clone()`的类必须派生于某个已有 `Cloneable` 基类。

1) 解决模板要求 `T` 派生于 `Cloneable` 问题。

// Example 4-3(c): An `IsDerivedFrom` constraints base

// with testable value

```
template<typename D, typename B>
```

```
class IsDerivedFrom
```

```
{
    class No { };
    class Yes { No no[2]; };

```

```
    static Yes Test( B* ); // not defined
```

```
    static No Test( ... ); // not defined
```

```
    static void Constraints(D* p) { B* pb = p; pb = p; }
```

```
public:
```

```
    enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };

```

```
    IsDerivedFrom() { void(*p)(D*) = Constraints; }
```

```
};
```

- 2) [...]如果 `T` 派生于 `Cloneable`，此模式提供一个可供选择的实现：否则，此模板工作于缺省模式。

// Example 4-3(d): Using `IsDerivedFrom` to make use of

// derivation from `Cloneable` if available, and do

// something else otherwise.

```
template<typename T, int>
```

```
class XImpl
```

```
{
    // general case: T is not derived from Cloneable
};
```

```
template<typename T>
```

```
class XImpl<T, 1>
```

```
{
    // T is derived from Cloneable
};
```

```

template<typename T>
class X
{
    XImpl<T, IsDerivedFrom<T, Cloneable>::Is> impl_;
    // ... delegates to impl_ ...
};

```

使用 traits 来解决问题:

// Example 4-4: Using traits instead of IsDerivedFrom
// to make use of Cloneability if available, and do
// something else otherwise. Requires writing a
// specialization for each Cloneable class.

```

template<typename T>
class XTraits
{
public:
    // general case: use copy constructor
    static T* Clone( const T* p ) { return new T( *p ); }
};

```

```

template<>
class XTraits<MyCloneable>
{
public:
    // MyCloneable is derived from Cloneable, so use Clone()
    static MyCloneable* Clone( const MyCloneable* p )
    {
        return p->Clone();
    }
};

```

// ... etc. for every class derived from Cloneable

Traits 和 2) 中那个普通的 XImpl 之间的区别主要在于: 有了 traits, 当用户定义了某个新类型的时候, 如果想将这个新类型用于 X, 所要做的大部分工作都在 X 的外部——我们只用特殊化 traits 模板, 为新类型“做正确的事”就可以了。Traits 还允许使用其它 clone 方法, 而不仅仅是特别命名为 Clone() 的某个函数。Traits 方案主要的缺点是: 针对继承层次结构中的每一个类, 它都要单独的特殊化。有办法可以一次性地为整个结构中的类提供 traits, 从而不必枯燥地写出大量的特殊化 [C++编程新思维]。

✧ 什么是 typename? 它有什么用?

// Example 5-1

```

template<typename T>
void f()
{
    T::A* pa; // what does this line do?
}

```

T::A 是一个有依赖性的名称（dependent name），因为它依赖于模板参数 T。
C++标准有这样的叙述：如果一个名称被使用在模板声明或定义中并且依赖于模板参数，则这个名称不被认为是一个类型的名称，除非名称查找到了一个合适的类型名称，或者这个名称用关键字 `typename` 修饰。
下面的例子说明的问题是：为什么要使用 `typename` 来引用（refer to）有依赖性的名称，以及如何使用。

```
// Example 5-2
//
template<typename T>
class X_base
{
public:
    typedef T instantiated_type;
};

template<typename A, typename B>
class X : public X_base<B>
{
public:
    bool operator()( const instantiated_type& i ) const
    {
        return i != instantiated_type();
    }
    // ... more stuff ...
};
```

X 的问题在于，“instantiated_type”本来想引用基类 X_base继承而来的 typedef。不幸的是，在编译器解析 X<A, B>::operator()()的内联定义之时，有依赖性的名称（即，依赖于模板参数的名称，例如继承而来的 X_base::instantiated_type）是不可见的，所以编译器将会发出抱怨，因为它还不知道 instantiated_type 指的是什么。有依赖性的名称只会在后来可见，也就是模板被真正实例化的时候。要想告诉编译器这种东西是类型名称，方法是为其加上 `typename` 关键字。为了让代码的可读性更好，我们只用如下提供另外一个 typedef。（标准库中会有很多这样的例子：空基类 X_base 存在的全部理由在于提供 typedef，但是，派生类往往最终又一次对它使用了 typedef。）

```
// Example 5-2(b): Better
template<typename A, typename B>
class X : public X_base<B>
{
public:
    typedef typename X_base<B>::instantiated_type
        instantiated_type;

    bool operator()( const instantiated_type& i ) const
    {
```

```

        return i != instantiated_type();
    }

```

```

// ... more stuff ...
};

```

- ✧ 通常，当你想指向一个容器内部的对象时，一条不错的规则是：尽量使用迭代器而不是用指针。毕竟，迭代器和指针往往是在同样的情况下以同样的方式失效。迭代器存在的一个理由是，它提供了一种方式，用以“指向”一个被包含对象。如果可以选择，尽量使用迭代器来指向容器内部。
- ✧ 不可能使“被代理集合”成为一个满足标准容器或序列要求的容器。标准算法一般不适合用于被代理容器，因为，较之存在于内存中的普通容器，被代理容器具有不同的性能特征。
- ✧ 如果想完全清除一个 **vector**，使它不包含任何元素并完全没有额外的容量，代码几乎和前面相同。我们只需要将临时 **vector** 初始化为空，而不要使它成为 **c** 的拷贝。这些技术对 **deque** 也是适用。

// Example 7-2(b): The right way to shrink-to-fit a vector.

```
//
```

```
vector<Customer> c( 10000 );
```

```
// ...now c.capacity() >= 10000...
```

```
// erase all but the first 10 elements
```

```
c.erase( c.begin()+10, c.end() );
```

```
// the following line does shrink c's
```

```
// internal buffer to fit (or close)
```

```
vector<Customer>( c ).swap( c );
```

```
// ...now c.capacity() == c.size(), or
```

```
// perhaps a little more than c.size()
```

- ✧ 除非真的需要空间优化，否则请总是使用 **deque<bool>**，而不要使用 **vector<bool>**。
- ✧ 关联式容器的重要使用规则：一个键一旦被插入到关联式容器中，那么在容器中的相对位置绝对不能改变。一定要知道哪些直接或间接的行为会改变一个键的相关次序，并且避免这样的行为。这样，在使用标准关联容器时，就可以避免不必要的问题发生。

（二）优化与性能

- ✧ 内联可能带来的问题：
 - 程序体积增大。
 - 内存占用增大（由于程序的体积增大）。
 - 执行时间可能反而增大。
 - 开发速度和编译时间增大（模块耦合，调试）。
- ✧ **原则：**在性能分析证明确实必要之前，避免内联或详细优化。
- ✧ **String** 的一个实现

```

namespace Optimized{
class StringBuf

```

```

{
public:
    StringBuf();           // start off empty
    ~StringBuf();          // delete the buffer
    StringBuf( const StringBuf& other, size_t n = 0 );
                        // initialize to copy of other,
                        // and ensure len >= n

    void Reserve( size_t n ); // ensure len >= n
    char*    buf;           // allocated buffer
    size_t    len;          // length of buffer
    size_t    used;         // # chars actually used
    unsigned refs;          // reference count
    Mutex     m;            // serialize work on this object

private:
    // No copying...
    //
    StringBuf( const StringBuf& );
    StringBuf& operator=( const StringBuf& );
};

class String
{
public:
    String();           // start off empty
    ~String();          // decrement reference count
                        // (delete buffer if refs==0)
    String( const String& ); // point at same buffer and
                        // increment reference count
    void Append( char ); // append one character
    size_t Length() const; // string length
    char& operator[](size_t); // element access
    const char operator[](size_t) const;

    // ... operator=() etc. omitted ...

private:
    void AboutToModify( size_t n, bool bUnshareable = false );
                        // lazy copy, ensure len>=n
                        // and mark if unshareable
    static size_t Unshareable; // ref-count flag for "unshareable"
    StringBuf* data_;
};

```



```

StringBuf::StringBuf()
    : buf(0), len(0), used(0), refs(1) { }

StringBuf::~StringBuf() { delete[] buf; }

StringBuf::StringBuf( const StringBuf& other, size_t n )
    : buf(0), len(0), used(0), refs(1)
    {
        Reserve( max( other.len, n ) );
        copy( other.buf, other.buf+other.used, buf );
        used = other.used;
    }

void StringBuf::Reserve( size_t n )
{
    if( len < n )
    {
        // Same growth code as in Item 14, except now we round
        // the new size up to the nearest multiple of 4 bytes.
        size_t needed = max<size_t>( len*1.5, n );
        size_t newlen = needed ? 4 * ((needed-1)/4 + 1) : 0;
        char* newbuf = newlen ? new char[ newlen ] : 0;
        if( buf )
        {
            copy( buf, buf+used, newbuf );
        }

        delete[] buf;    // now all the real work is
        buf = newbuf;    // done, so take ownership
        len = newlen;
    }
}

const size_t String::Unshareable = numeric_limits<size_t>::max();

String::String() : data_(new StringBuf) { }

String::~String()
{
    bool bDelete = false;
    data_>m.Lock(); //-----
    if( data_>refs == Unshareable || --data_>refs < 1 )
    {
        bDelete = true;
    }
}

```

```

data_>m.Unlock(); //-----
if( bDelete )
{
    delete data_;
}
}

String::String( const String& other )
{
    bool bSharedIt = false;
    other.data_>m.Lock(); //-----
    if( other.data_>refs != Unshareable )
    {
        bSharedIt = true;
        data_ = other.data_;
        ++data_>refs;
    }
    other.data_>m.Unlock(); //-----
    if( !bSharedIt )
    {
        data_ = new StringBuf( *other.data_ );
    }
}

void String::AboutToModify(
    size_t n,
    bool    markUnshareable /* = false */
    )
{
    data_>m.Lock(); //-----
    if( data_>refs > 1 && data_>refs != Unshareable )
    {
        StringBuf* newdata = new StringBuf( *data_, n );
        --data_>refs;    // now all the real work is
        data_>m.Unlock(); //-----
        data_ = newdata; // done, so take ownership
    }
    else
    {
        data_>m.Unlock(); //-----
        data_>Reserve( n );
    }
    data_>refs = markUnshareable ? Unshareable : 1;
}

void String::Append( char c )

```

```

{
    AboutToModify( data_>used+1 );
    data_>buf[used++>data_>used++] = c;
}
size_t String::Length() const
{
    return data_>used;
}
char& String::operator[]( size_t n )
{
    AboutToModify( data_>len, true );
    return data_>buf[n];
}
const char String::operator[]( size_t n ) const
{
    return data_>buf[n];
}
} // Optimized

```

（三）异常安全议题及技术

- ✧ 对于 C++ 的构造函数模型，只会是下面二者之一：
 - 1) 构造函数正常返回，即，控制抵达函数体的尾部，或者执行了一个 **return** 语句。这种情况下，对象真实存在。
 - 2) 构造函数抛出异常后退出。这种情况下，对象不仅不会继续存在，而且，实际上它根本就从未作为对象存在过。
- ✧ 从构造函数中抛出异常意味着什么？
 - 意味着构造已经失败，对象从没存在过，它的生命期从没开始过。构造函数不成功，析构函数就永远不会被调用，其原因正在于此——没有东西可以摧毁。它无法死亡，因为它从来就未曾生存过。
- ✧ C++ 标准：如果处理程序没有以抛出异常的方式退出（既没有重新抛出最初的异常，也没有抛出什么新东西），那么，在控制抵达构造函数析构函数的 **catch block** 的末尾时，最初的异常会被自动地重新抛出，就像处理程序的最后一条语句是 “**throw**” 一样。这就意味着：一个构造函数或析构函数的 **function try block** 的处理程序代码 “必须” 以抛出某个异常结束，没有其他方式！只要不违反异常规范，C++ 语言不关心你抛出的是什么异常——可以是最初的那个，或者是其它某个经过转化的异常——但必须有个异常！只要基类或成员子对象的构造函数抛出任何异常，就一定会导致某个异常从它们的外层构造函数（**containing constructor**）中泄漏出来，想阻止这一点是不可能的。
- ✧ 在 C++ 中，只要任何一个基类或成员子对象构造失败，整个对象的构造必然失败。
- ✧ 如果一个类确实可以具有某种合理的 “部分” 构造失败（**construction partially failed**）状态呢？——也就是说，它真的具有某些不是绝对需要的 “可选” 成员，对象可以没有它们而苟延残喘，只是又可能缺少某些功能而已。这种情况下就可以运用 **Pimpl** 手法，在一定的安全距离内拥有对象中可能损坏的部分。“对象的可选部分” 这一观念也是 “无论何时都要尽量使用委托而不使用继承” 的另外一个原因。基类子对象永远不能成为可选部分，因为你不可能将基类子对象放进一个 **Pimpl**。

- ✧ C++标准: “在一个对象的构造函数或析构函数的 **function try block** 处理程序中, 引用对象的任何非静态成员或基类将导致不可预测的行为。”
- ✧ 有关 **Function Try Block** 的法则
 - 法则 1: 构造函数的 **function try block** 处理程序只能用于转化从基类或成员子对象的构造抛出的异常 (也可能做一些相应的记录工作, 或其它某种附带性的工作, 以响应构造失败)。此外, 没有其它用途。
 - 法则 2: 析构函数的 **function try block** 鲜有或没有实际用处, 因为析构函数绝不应该产生异常。
 - 法则 3: 其它所有 **function try block** 都没有实际用处。对一个函数来说, 其内部正常的 **try block** 不能捕捉到的东西, 其正常的 **function try block** 也不能捕捉得到。
 - 法则 4: “获取未管理资源 (**unmanaged resource**)” 的操作总是应该放在构造函数体内, 绝不要放在初始化列表中。换句话说, 要么运用 “获得资源才是初始化” 的策略 (从而完全避免未管理资源的存在), 要么在构造函数体内执行资源获取的操作。(使用 **RAII** 往往可以完全避免写 **try** 和 **catch**)
 - 法则 5: 清除 “获取未管理资源的操作” 总要放在构造函数或析构函数体内的局部 **try block** 处理程序中, 绝对不要放在构造函数或析构函数的 **function try block** 处理程序中。
 - 法则 6: 如果构造函数有异常规范, 那么, 对于基类和成员子对象可能抛出的所有异常, 这个异常规范必须留有余地。
 - 法则 7: 使用 **pimpl** 手法保存类内部的 “可选部分”。
 - 法则 8: 尽量使用 “获得资源才是初始化” 的技术来管理资源。
- ✧ 构造函数的异常必须被传播。没有其它方式可以表示构造函数失败。
- ✧ 如果为构造函数写了一个空 **throw** 规范, 但某个基类或成员子对象的构造函数真的会抛出异常, 那么将会发生些什么? 简单的回答是: “会执行 **terminate()**。会直接执行 **terminate()**, 不会经过 **try**”。详细一点地说: 函数 **unexpected()** 会被调用, 它有两个选择——一个选择是: 抛出或重新抛出异常规范所允许的一个异常 (这不可能, 因为它是空的, 不允许抛出任何异常); 另一个选择是: 调用 **terminate()**。而 **terminate()** 会立即终止程序 (可以使用 **set_unexpected()** 和 **set_terminate()** 来让自己的处理程序得以调用, 这给了我们多做一些记录或清理工作的机会, 但它们最终还是得做相同的事)。
- ✧ **std::uncaught_exception()** 完成什么功能?
 - 标准 **uncaught_exception()** 函数提供了一种方法, 让你知道 “当前是否有一个异常正处于活动状态”。要注意的重要一点是, 这和知道 “当前是否可以安全地抛出一个异常” 是两码事。
- ✧ 请看下面的代码, 这是个好办法吗?

```
T::~T()
{
    if( !std::uncaught_exception() )
    {
        // ... code that could throw ...
    }
    else
    {

```

```

// ... code that won't throw ...
}
}

```

- 1) 这个例子背后的思想很简单：只要当前可以安全地抛出异常，我们就选择那条可以抛出异常的执行路径。这个思想在两个方面。首先，这段代码不会那么做。第二，这个思想本身就是错误的。
- 2) 让 `T::~~T()` 的报错语义具有两种不同“模式”的操作，这不能不说是个糟糕的设计。因为，允许一个操作以两种不同的方式去报告同一个错误，这绝对是个拙劣的设计（具有精神分裂症般的接口）。是接口不只具有一种形态，而让它处于调用者代码无法轻易控制或理解的形态，这将带来两个重大缺陷。首先，它使得接口和语义复杂化。其次，它使得调用者处境艰难，因为调用者必须能够处理两种不同的报错形式——在太多的调用者本来就不能很好地完成出错检测的时候，情况更是如此。
- 3) 正确的方案：

// Example 19-4: The right solution

```

T::~~T() /* throw() */
{
    // ... code that won't throw ...
}

```

注意，为了表示不抛出异常，这个 `throw()` 的规范只是作为注释出现在代码中。这是（Herb Sutter）所采用的编码风格，其部分原因在于：实际上，异常规范没有像它所吹嘘的那样带来多少好处。是否真的要写一个异常规范，这只是个人喜好问题。

- 4) 如果需要，`T` 可以提供一个“用在析构之前的函数”（例如 `T::Close()`），这个函数可以抛出异常，并针对 `T` 对象及其拥有的所有资源执行终止（shutdown）操作。采用这种方法，在调用者代码需要检测严重错误的时候，它就可以调用 `T::Close()`；`T::~~T()` 可以用 `T::Close()` 加上一个 `try/catch block` 来实现。这很好地遵循了“一个函数，一种职责”的原则。最初的代码存在一个问题是，它让同一个函数既销毁对象，又要负责最后的清理及报错工作。

// Example 19-5: Alternative right solution

```

T::Close()
{
    // ... code that could throw ...
}

```

```

T::~~T() /* throw() */
{
    try
    {
        Close();
    }
    catch( ... ) { }
}

```

✧ **设计准则：**决不允许异常从析构函数抛出。写析构函数的时候，就像它已经有了一

个 `throw()` 异常规范一样（至于这个 `throw` 规范是否真的出现在代码中，这纯属个人喜好问题。）

✧ **设计准则：**如果析构函数调用了可能会抛出异常的函数，一定要将这个调用包装在 `try/catch block` 中，以防止异常逃出析构函数。

✧ `uncaught_exception()` 有其它什么好的用处吗？

■ 它没有什么既好又安全的用途。不要用它。

✧ C++ 标准：

1) 在函数调用之前，对函数所有参数的求值必须全部完成。这包括，如果函数参数是表达式，那么，表达式所产生的任何副作用也得全部完成。

2) 一旦一个函数开始执行，调用者函数中的表达式将不会开始求值或继续求值，直至被调用函数执行结束。**函数执行永远不会交叉进行。**

3) 如果函数参数是表达式，这些表达式通常可以按任何次序求值，包括交叉求值，除非另外有其它规则限制。（C++ 标准和 C 标准一样没有规定表达式求值顺序的一个原因是：这样给了编译器一定范围的自由，只有这样，编译器才有可能执行优化，否则不行）

✧ `f(new T1, new T2)` 不是异常安全的（因为 `new` 不是原子操作）。

✧ `f(auto_ptr<T1>(new T1), auto_ptr<T2>(new T2))` 不是异常安全的。

■ 一个受限的解决方案：`f(auto_ptr_new<T1>(), auto_ptr_new <T2>())`。依赖于“函数的执行永远不会交叉执行”这一标准。

// Example 21-2(a): Partial solution

```
template<typename T>
auto_ptr<T> auto_ptr_new()
{
    return auto_ptr<T>( new T );
}
```

■ 一个改进的方案。

// Example 21-2(b): Improved solution

```
template<typename T>
auto_ptr<T> auto_ptr_new()
{
    return auto_ptr<T>( new T );
}
```

```
template<typename T, typename Arg1>
auto_ptr<T> auto_ptr_new( const Arg1& arg1 )
{
    return auto_ptr<T>( new T( arg1 ) );
}
```

```
template<typename T, typename Arg1, typename Arg2>
auto_ptr<T> auto_ptr_new( const Arg1& arg1,
                        const Arg2& arg2 )
{
    return auto_ptr<T>( new T( arg1, arg2 ) );
}
```

```
}
```

```
// etc.
```

■ 一个正确的方案:

```
// Example 21-1(a): A solution
```

```
//
```

```
// In some header file:
```

```
void f( auto_ptr<T1>, auto_ptr<T2> );
```

```
// In some implementation file:
```

```
{
```

```
    auto_ptr<T1> t1( new T1 );
```

```
    auto_ptr<T2> t2( new T2 );
```

```
    f( t1, t2 );
```

```
}
```

- ✧ **设计准则:** 在各自独立的程序语句中执行每一个现实的资源分配（例如，`new`），并将（通过 `new`）分配的资源立即交给管理者对象（例如，`auto_ptr`）。
- ✧ 异常安全的规范形式：
 - **基本保证（basic guarantee）:** 如果有异常抛出，资源不会泄漏，对象保持可摧毁、可使用但不一定可预测的状态。这是异常安全可以使用的最弱级别，它使用的场合是：失败的操作已经对对象状态做了改变，但调用者代码仍然能够应付。
 - **强烈保证（strong guarantee）:** 如果有异常抛出，程序状态保持不变。这一级别总是蕴涵着“提交或回退（commit-or-rollback）”语义，这包括：如果操作失败，指向容器内部的引用或迭代器不会失效。
 - **不抛出异常的保证（nothrow guarantee）:** 在任何条件下，函数都不会产生异常。有时，除非某些函数保证不抛出异常（例如，析构函数、资源释放函数），否则不可能实现强烈保证甚至基本保证。
- ✧ 具有强烈异常安全性的“拷贝赋值”的规范形式是什么？
 - 规范形式的拷贝赋值包括两步操作。首先，提供一个不抛出异常的 `Swap()` 函数，用以交换两个对象的内容，或内部状态：

```
void T::Swap( T& other ) /* throw() */  
{  
    // ...swap the guts of *this and other...  
}
```

接着，运用“创建一个临时对象然后交换”的手法实现 `operator=()`:

```
T& T::operator=( const T& other )  
{  
    T temp( other );    // do all the work off to the side  
    Swap( temp );       // then "commit" the work using  
    return *this;       // nonthrowing operations only  
}
```
- ✧ 说明并示范一种简单的转换技术：这种转换可以应用于任何一个类，并可以很容易

地使得那个类的拷贝赋值具有（近似）强烈的异常安全。

```
// Example 22-2: The general solution to
// Cargill's Widget Example
class Widget
{
public:
    Widget(); // initializes pimpl_ with new WidgetImpl

    ~Widget(); // must be provided, because the implicit
               // version causes usage problems
               // (see Items 30 and 31)

    Widget& operator=( const Widget& );
    // ...

private:
    class WidgetImpl;      auto_ptr<WidgetImpl> pimpl_;
    // ... provide copy construction that
    // works correctly, or suppress it ...
};

// Then, typically in a separate
// implementation file:
class Widget::WidgetImpl
{
public:
    // ...

    T1 t1_;
    T2 t2_;
};

void Widget::Swap( Widget& other ) /* throw() */
{
    //注意对于 auto_ptr, std::swap()不一定能正常工作。
    auto_ptr<WidgetImpl> temp( pimpl_ );
    pimpl_ = other.pimpl_;
    other.pimpl_ = temp;
}

Widget& Widget::operator=( const Widget& other )
{
    Widget temp( other ); // do all the work off to the side
    Swap( temp );        // then "commit" the work using
    return *this;        // nonthrowing operations only
}
```

附笔：请注意，如果使用 auto_ptr 成员，那么：

- 1) 要么, 必须将 `WidgetImpl` 的定义提供给 `Widget`; 要么, 如果想隐藏 `WidgetImpl`, 就必须为 `Widget` 写一个自己的析构函数, 即使这个析构函数很简单; 如果使用编译器自动生成的析构函数那个析构函数将被定义在每个编译单元 (translate unit) 中, 因而, `WidgetImpl` 的定义必然在每个编译单元可见;
 - 2) 对于 `Widget`, 还应该提供自己的拷贝构造和赋值函数, 因为一般来说, 你不会希望类的成员具有“拥有权转移”语义。如果有另外一个智能指针, 也可以考虑用它来取代 `auto_ptr`, 但上述原则依然重要。
 - 3) 这个方案具有“近似”强烈的异常安全性。如果一个异常抛出, 它不绝对保证程序状态会完全保持不变。因为, 当创建临时 `Widget` 对象并因此创建 `pimpl_` 的 `t1_` 和 `t2_` 成员的时候, 那些成员的创建 (以及/或者析构——如果操作失败的话) 可能会有副作用, 比如修改一个全局变量, 对于这一点, 我们无法知晓, 也无法控制。
- ◇ 对于 EH (Exception Handling), Scott Meyers 总结:
- 指针是你的敌人。因为它们带来的种种问题正是 `auto_ptr` 想要消除的 (这就是说, 普通指针通常应该被管理者对象拥有, 这个管理者对象拥有所指向的资源, 并自动执行清理工作)。
 - 指针是你的朋友。因为指针上的操作不会抛出异常。
- ◇ 请记住: 如果使用了 `auto_ptr`, 你的类就必须提供自己的具有正确语义的析构函数、拷贝构造函数和拷贝赋值函数; 或者, 如果拷贝构造和赋值函数对你的类没有用, 你可以禁止掉它们。
- ◇ Is-Implemented-In-Terms-Of (IIITO) 的含义是什么?
- 如果 `T` 在它的实现中以某种形式使用了另外一种类型 `U`, 就称 “`T IIITO U`”。“以某种形式使用”这一措辞表示的范围很广。例如, `T` 可以是 `U` 的一个 `adapter`、`proxy`、`wrapper`; 或者, `T` 仅仅只是在它的实现细节中偶尔用到了 `U`。
 - 如果可以, 总应该使用委托而不是使用继承来实现 IIITO, 因为委托具有低耦合性和容易编写异常安全程序。

(四) 继承与多态

- ◇ 设计准则: 避免多继承自多个“非 protocol 类”。(protocol 类是一种抽象基类 (Abstract Base Class), 或简称 ABC, 它完全是由纯虚函数组成, 没有数据成员。)
- ◇ MI 究竟有必要吗? MI 的应用场合?
- 简单的回答是: 只要程序可以用汇编语言 (或更低级语言) 来写, 就不能说某种特性绝对必要。
 - 什么时候使用 MI 才算合适? 简而言之, 只有在每一个继承单独取出来看都合适的时候, 这样的 MI 才算合适。在现实世界中, MI 的应用大多逃不出以下三类:
 - 1) 结合使用程序模块或库程序。在实践中, 知道如何运用 MI 来结合使用供应商提供的程序库, 这是每一个 C++ 程序员必须具备的素质。无论你是否经常运用它, 你绝对应该知道它并理解它。
 - 2) Protocol 类 (interface 类)。在 C++ 中, MI 最适合、最安全的应用是定义 protocol 类——即, 完全由纯虚函数构成的类。由于这种基类没有数据成员, MI 臭名昭著的复杂性就得以完全避免。
 - 3) 易于 (多态) 使用。有了继承, 在接受基类对象的任何代码中, 我们就都可以使用派生类对象, 这是一项威力强大的功能。在某些场合, 如果同一个派生对象可以代替数种基类对象使用, 那将会很有用处, 这正是 MI 大

显身手的地方。

注意：第 3)点在很大程度上与第 1)、2)点重叠。在实施另外两点之一时，同时、并处于相同的理由运用第 3)点，往往很有用处。

- ✧ 如何分割“连体双婴”问题？请看下面两个类：

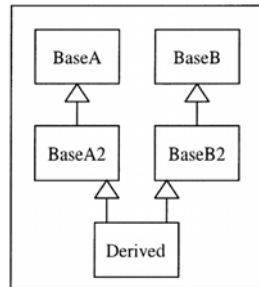
```
class BaseA
{
    virtual int ReadBuf( const char* );
    // ...
};
```

```
class BaseB
{
    virtual int ReadBuf( const char* );
    // ...
};
```

BaseA 和 **BaseB** 有一个共同点——它们显然都想被用作基类，但除此之外它们毫不相干。它们的 **ReadBuf()**函数用来做不同的事，而且这两个类还来自不同的程序库提供商。示范如何写出一个 **Derived** 类，这个类从 **BaseA** 和 **BaseB** 公有继承，而且还要对两个 **ReadBuf()**进行改写，让它们做不同的事。

这个问题的关键在于，这两个可以改写的函数具有完全相同的名称和原型。所以，解决方案的关键必然在于：至少需要改写其中一个函数的原型，而函数原型中最容易修改的部分是函数名称。

如何改变一个函数的名称？当然是通过继承！这就需要有一个中间类（**intermediate class**），这个中间类从基类派生，它声明一个新的虚函数，并改写那个继承而来的虚函数，使之调用这个新函数。继承层次结构看起来如下图所示。



代码大致如下：

// Example 26-2: Attempt #2, correct

```
class BaseA2 : public BaseA
{
public:
    virtual int BaseAReadBuf( const char* p ) = 0;
private:
    int ReadBuf( const char* p )    // override inherited
    {
        return BaseAReadBuf( p );    // to call new func
    }
};
```

```

class BaseB2 : public BaseB
{
public:
    virtual int BaseBReadBuf( const char* p ) = 0;
private:
    int ReadBuf( const char* p )    // override inherited
    {
        return BaseBReadBuf( p );    // to call new func
    }
};

```

```

class Derived : public BaseA2, public BaseB2
{
    /* ... */
public: // or "private:", depending whether other
    // code should be able to call these directly

    int BaseAReadBuf( const char* );
    // overrides BaseA::ReadBuf indirectly
    // via BaseA2::BaseAReadBuf

    int BaseBReadBuf( const char* );
    // overrides BaseB::ReadBuf indirectly
    // via BaseB2::BaseBReadBuf
};

```

BaseA2 和 BaseB2 可能还需要复制 (duplicate) BaseA 和 BaseB 的构造函数，使得 Derived 可以调用它们。但要做的修改也就那么多。(通常，比“在代码中复制构造函数”更简单的办法是：让 BaseA2 和 BaseB2 从基类虚拟继承。这样，Derived 就可以直接访问基类的构造函数。) BaseA2 和 BaseB2 是抽象基类，所以，它们不需要复制 BaseA 或 BaseB 的其它任何函数或运算符，例如赋值运算符。

✧ 为纯虚函数提供函数体究竟有没有意义？之所以这样做，一般处于三个主要原因。

1) 纯虚析构函数

[设计准则:]所有基类的析构函数要么应该是虚拟公有成员要么应该是非虚拟保护成员。简单地说，这是因为：首先，要记住，应该总是避免从实体类 (concrete class) 派生。因而，假设基类不是实体类，那么，它就不会出于“实例化自身”的目的而去提供一个公有析构函数。这样，就剩下两个选择：

1. 要么，需要“通过基类指针进行多态删除”的功能，这种情况下，析构函数必须是虚拟公有成员；
2. 要么，不需要这一功能，这种情况下，析构函数应该是非虚拟保护成员——之所以是保护成员，是为了防止被滥用。

如果一个类应该是抽象类 (想禁止将其实例化)，但它没有其它任何纯虚函数，而有一个公有析构函数，那么，请将这个析构函数声明为纯虚函数。

```
// Example 27-2(a)
// file b.h
class B
{
public: /*...other stuff...*/
    virtual ~B() = 0; // pure virtual destructor
};
```

当然，任何派生类的析构函数必须隐式地调用基类的析构函数，所以，析构函数还是得定义（即使为空）。如果不提供这个定义，还是得让其它类从 **B** 派生，但那些派生类就不能被实例化，从而没有合适吗特别的用处。

```
// Example 27-2(a), continued
// file b.cpp
```

```
B::~~B() { /* possibly empty */ }
```

2) 明确使用缺省行为

如果派生类没有改写某个普通的虚函数，它就会默认地继承基类中的行为。如果想提供一个默认行为但又不想让派生类这么“无声无息”地继承，可以声明一个纯虚函数并依然提供缺省实现；这样，派生类的设计者如果想使用它，就必须主动对它进行调用。（GOF 的 Design Pattern 中 State 模式。）

```
// Example 27-2(b)
class B
{
protected:
    virtual bool f() = 0;
};

bool B::f()
{
    return true;    // this is a good default, but
}

// shouldn't be used blindly
class D : public B
{
    bool f()
    {
        return B::f(); // if D wants the default
    }                // behaviour, it has to say so
};
```

3) 提供部分行为

有时，我们需要向派生类提供“部分行为”（partial behavior），同时，这个派生类还必须保持完整。这是一种很有价值的應用。其设计思想是：在派生类中，将基类实现作为派生类实现的“一部分”来执行。（GOF 的 Decorator 模式。）

```
// Example 27-2(c)
//
class B
```

```

{
    // ...
protected virtual bool f() = 0;
};

bool B::f()
{
    // do something general-purpose
}

class D : public B
{
    bool f()
    {
        // first, use the base class's implementation
        B::f();

        // ... now do more work ...
    }
};

```

4) 应付功能不足的编译器诊断程序

有些时候，会无意中调用了一个纯虚函数（间接地从基类的构造函数或析构函数调用）。出现这种情况，并不是所有的编译器都会确切地告诉你的问题所在。要想避免在这种调试时间上的浪费，一个办法是：为这种永远不应该被调用的纯虚函数提供定义，并在那些定义中置入一些恶性代码。这样一来，如果你无意中调用了那些函数，你就会立刻知道。例如：

```

// Example 27-2(d)

class B
{
public:
    bool f();          // possibly instrumented, calls do_f()
                      // (Non-Virtual Interface pattern)

private:
    virtual bool do_f() = 0;
};

bool B::do_f()    // this should NEVER be called
{
    if( PromptUser( "pure virtual B::f called -- "
                    "abort or ignore?" ) == Abort )
        DieDieDie();
}

```

在公用的 `DieDieDie()` 函数中，做你喜欢做的。

```

void DieDieDie() // a C way to scribble through a null
{
    // pointer... a proven crowd-pleaser
    memset( 0, 1, 1 );
}

void DieDieDie() // another C-style method
{
    abort();
}

void DieDieDie() // a C++ way to scribble through a null
    // data pointer
    *static_cast<char *>(0) = 0;
}

void DieDieDie() // a C++ way to follow a null function
    // pointer to code that doesn't exist
    static_cast<void(*)>(0)();
}

void DieDieDie() // unwind back to last "catch(...)"
{
    class LocalClass {};
    throw LocalClass();
}

void DieDieDie() // an alternative for standard::distes
{
    throw std::logic_error();
}

void DieDieDie() throw() // for standard::distes having good compilers
{
    throw 0;
}

```

✧ **受控的多态。**在 OO 模型的建立中，“IS-A”多态是一种非常有用的工具。但有时候，对于某些类，可能想限制某些代码多态地使用它。如何达到这种效果？

■ 如果 **Derived** 从 **Base** 私有继承，那么几乎没有代码可以多态地将 **Derived** 作为 **Base** 使用。之所以说“几乎”，原因在于：如果代码可以访问 **Derived** 的私有成员，它还是可以访问 **Derived** 的私有基类，因而可以动态地将 **Derived** 代替为 **Base** 使用。正常来说，只有 **Derived** 的成员函数具有这种访问权。然而，通过 C++ 的友元特性，我们可以将类似的访问权限扩充到其它外部代码中。

```

class Base
{

```

```

public:
    virtual void VirtFunc();
    // ...
};

class Derived : public Base
{
public:
    void VirtFunc();
    // ...
};

void SomeFunc( const Base& );

void f1()
{
    Derived d;
    SomeFunc( d ); // works, OK
}

void f2()
{
    Derived d;
    SomeFunc( d ); // we want to prevent this
}

```

The answer is to write:

```

class Derived : private Base
{
public:
    void VirtFunc();
    // ...
    friend void f1();
};

```

（五）内存及资源管理

- ✧ 原则：不要混用数组和 `auto_ptr`。下面示例是使用 `auto_ptr` 的一个常见错误。
 - `auto_ptr p(new T[n]); // Evil, even n equals zero`
 - 零长度数组是合法的。`new T[0]`的结果是一个指针，指向包含零个元素的数组。这个指针具有的行为特征和其他 `new T[n]`的结果一样，包括：对于数组，不可能访问到 `n` 个以上的元素。对零长度数组来说，根本就不能访问到任何元素，因为数组中没有元素。
 - C++标准：在直接 `new` 声明符中，如果表达式的值为零，分配函数将被调用，用来分配一个包含零个元素的数组。`new` 表达式返回的指针不是 `null`。
 - 如果零长度的数组什么事都不能做（除了需要记住它的地址），那又何必允许

它的存在呢？一个重要的原因是：有了它，在编写动态数组分配代码时会更容易（不需要检查参数 `n` 的值）。

✧ 如果 `T` 类型不支持拷贝和赋值，任何标准容器（包括 `vector`）就不能通过这样的 `T` 类型实例化。设计类时需要考虑是否让它支持拷贝构造和赋值。

✧ `ValuePtr`

```
// assignment and full traits-based
// customizability.
//
//--- new code begin -----
template<typename T>
class VPTraits
{
static T* Clone( const T* p ) { return new T( *p ); }
};
//--- new code end -----
```

A brief coda: Since `VPTraits` has only a single static function template, why make it a class template instead of just a function template? The main motive is for encapsulation (in particular, better name management) and extensibility. We want to avoid cluttering the global namespace with free functions. The function template could be put at namespace scope in whatever namespace `ValuePtr` itself is supplied in. But even then, this couples it more tightly to `ValuePtr` as opposed to use by other code also in that namespace. The `Clone()` function template may be the only kind of trait we need today, but what if we need new ones tomorrow? If the additional traits are functions, we'd otherwise have to continue cluttering things up with extra free functions. But what if the additional traits are typedefs or even class types? `VPTraits` gives us a nice place to encapsulate all those things.

```
template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr( T* p = 0 ) : p_( p ) { }

    ~ValuePtr() { delete p_; }

    T& operator*() const { return *p_; }

    T* operator->() const { return p_; }

    void Swap( ValuePtr& other ) { swap( p_, other.p_ ); }

    ValuePtr( const ValuePtr& other )
        : p_( CreateFrom( other.p_ ) ) { }

    ValuePtr& operator=( const ValuePtr& other )
    {
```



```

    ValuePtr temp( other );
    Swap( temp );
    return *this;
}

template<typename U>
ValuePtr( const ValuePtr<U>& other )
    : p_( CreateFrom( other.p_ ) ) { }

template<typename U>
ValuePtr& operator=( const ValuePtr<U>& other )
{
    ValuePtr temp( other );
    Swap( temp );
    return *this;
}

private:
    template<typename U>
    T* CreateFrom( const U* p ) const
    {
        ///--- new code begin -----
        return p ? VPTraits<U>::Clone( p ) : 0;
        ///--- new code end -----
    }

    template<typename U> friend class ValuePtr;

    T* p_;
};

```

（六）自由函数与宏

☆ 能否写出一个函数，使其返回指向函数自身的指针？

1) `typedef FuncPtr (*FuncPtr)();` //错误，这种递归形式的 `typedef` 是不合法的。

2) `typedef void* (*FuncPtr)();`

`void* f(){ return (void*)f; }` // 转换为 `void*`

`FuncPtr p = (FuncPtr)(f());` //从 `void*`转换回来

`p();`

这种做法不符合标准。尽管一个 `void*` 的大小足以保存任何对象指针大小，但不一定适合保存一个函数指针。在某些平台上，一个函数指针比一个对象指针要大。有一个办法避免这个问题，可以借助另外一种“函数指针类型”来回转型，而不是借助简单的 `void*`：

// Example 32-3(c): Standard and portable,

// but nonetheless still a hack

//

```
typedef void (*VoidFuncPtr)();
typedef VoidFuncPtr (*FuncPtr)();
```

```
VoidFuncPtr f() { return (VoidFuncPtr)f; } // cast to VoidFuncPtr
```

```
FuncPtr p = (FuncPtr)f(); // cast from VoidFuncPtr
p();
```

- 3) 一个正确且具可移植性的方案。其做法是：增加一层间接性——具体形式是一个代理类，这个类不但接受它想要的指针类型，同时还有一个隐式转换，可以转换到它想要的指针类型。

// Example 32-3(d): A correct solution

```
//
class FuncPtr_;
typedef FuncPtr_ (*FuncPtr)();
class FuncPtr_
{
public:
    FuncPtr_( FuncPtr p ) : p_( p ) { }
    operator FuncPtr() { return p_; }
private:
    FuncPtr p_;
};
```

Now we can declare, define, and use f() naturally:

```
FuncPtr_ f() { return f; } // natural return syntax
```

```
int main()
{
    FuncPtr p = f(); // natural usage syntax
    p();
}
```

✧ 设计准则：除了一下情况之外，避免使用预处理宏：

- 1) 守护头文件；
- 2) 条件编译，以获得可移植性，或在.cpp 文件（不是.h 文件！）中进行调试；
- 3) 用#pragma 禁止掉无伤大雅的警告；但这种#pragma 总得包含在一个“为了获得可移植性而提供的条件编译”之中，以防编译器不认识它们而发出警告。

✧ 预处理宏常见陷阱：

- 1) 宏的参数必须加上括号。

// Example 35-1(a): Paren pitfall #1: arguments

```
#define max(a,b) a < b ? b : a
```

```
//max( i += 3, j )
```

- 2) 不要忘记为整个展开式加上括号。

// Example 35-1(b): Paren pitfall #2: expansion

```
#define max(a,b) (a) < (b) ? (b) : (a)
// k = max( i, j ) + 42;
```

- 3) 当心多参数运算。

```
// Example 35-1(c): Multiple argument evaluation
```

```
#define max(a,b) ((a) < (b) ? (b) : (a))
```

```
max( ++i, j )
```

- 4) 名字冲突。

如果在其它代码中，有些变量或什么东西刚好也叫 `max`，宏会把事情搞得一团糟。如果非得写一个宏不可，那么就尽量为它想出一个不平常的、难以拼写的名字，这样才能最大可能地避免和其它名字相冲突。

- 5) 宏不能递归。

- 6) 宏没有地址。

- 7) 宏有碍调试。

- ✧ 在 C++ 标准中，条款 2.1 对编译阶段（phases of translation）做了严格的规定。预处理指令和宏的展开发生在第 4 阶段。这样一来，在一个符合标准的编译器上，宏不可能创建一下任何东西：

- 1) a trigraph (trigraphs are replaced in phase 1);
- 2) a universal character name (\uXXXX, replaced in phase 1);
- 3) an end-of-line line-splicing backslash (replaced in phase 2);
- 4) a comment (replaced in phase 3);
- 5) another macro or preprocessing directive (expanded and executed in phase 4); or
- 6) changes to a character literal (for example, 'x') or string literal (for example, "hello, world") via macro names inside the strings.

（七）杂项议题

- ✧ 设计原则：变量初始化时，尽量采用 “`T t(u)`” 形式，不要采用 “`T t = u`” 形式。
- ✧ 不包含任何其它头文件，能写出 `ostream` 和 `string` 写出正确的前置声明吗？
 - 简短的答案是：这不可能。实际情况是：不存在某种标准且具有可移植的方法可以做到不包含另一个文件却能前置申明 `ostream`；根本没有一种标准且有可移植性的方法前置声明 `string`。之所以不能对二者进行前置声明，原因在于，C++ 标准已经有了明确的规定，不能对 `namespace std` 写出自己的声明，而 `ostream` 和 `string` 就在 `namespace std` 之中。
 - C++ 标准：除非另有说明，如果一个 C++ 程序向 “`namespace std`” 或 “`namespace std` 内的某个名字空间” 增加声明或定义，其后果不可预测。C++ 标准还允许：在供应商提供的标准库实现中，标准库模板具有的模板参数可以比标准所要求的还要多（当然，必须提供合适的缺省值，以保持兼容性）。即使可以前置声明标准库模板和类，这种做法也是不具可移植性的，因为供应商可以扩充那些声明，使得不用的实现都各不相同。
- ✧ 设计准则：当前置声明可以满足需要时，绝对不要包含 (`#include`) 头文件。在不需要 (`stream`) 的完整定义时，尽量只包含 (`#include`) `<iosfwd>`
- ✧ 名字空间使用规则（其根本是避免名字污染）：
 - 绝对不要在头文件中使用 `using` 指令；
 - 绝对不要再头文件中使用名字空间 `using` 指令；
 - 在实现文件中，绝对不要在 `#include` 指令之前使用 `using` 声明或 `using` 指令。

- 在引用 C 头文件时，采用新风格的`#include<header>`，而不采用旧风格的`#include<header.h>`