ITEM1:流

//1.std::cin 和 std::cout 的类型是?

std::basic_istream<char,std::char_traits<char> >

std::basic_ostream<char,std::char_traits<char> >

//2.写一个 ECHO 程序 ，让他简单的相应输入，并通过以下方式等效的调用

ECHO<infile> outfile;

ECHO infile outfile;

//最简单的解决方案

```
// Example 1-1: A one-statement wonder
//
#include <fstream>
#include <iostream>

int main( int argc, char* argv[] )
{
   using namespace std;

   (argc > 2? ofstream(argv[2], ios::out | ios::binary): cout)<<
   (argc > 1? ifstream(argv[1], ios::in | ios::binary): cin).rdbuf();
}
```

//以上代码太过精简 可读性不高。而设计准则是尽量的体改可读性，

//避免撰写精简的代码。

//实现方法

```
#include <fstream>
#include <iostream>

 int main( int argc, char* argv[] )
 {
    using namespace std;

    fstream in, out;
    if( argc > 1 ) in.open ( argv[1], ios::in    | ios::binary );
    if( argc > 2 ) out.open( argv[2], ios::out | ios::binary );

    Process( in.is_open()    ? in    : cin,
                  out.is_open() ? out : cout );
 }
```

//有效的工程设计准则——尽量提高可扩充性

//避免写出的代码只能解决当前的问题，几乎任何时候，若能写出可

//扩充的方案，按将是更加的选择

//有效的工程设计准则——尽量提高封装性，将关系分离

//如果可能，一段代码－－函数或类－－应该只知道或负责一件事情

```
//Process 的实现
 //方法 A   模版
 // Example 1-2(a): A templatized Process()
      //
      template<typename In, typename Out>
      void Process( In& in, Out& out )
      {
         // ... do something more sophisticated,
         //         or just plain "out << in.rdbuf();"...
      }
//方法 B   虚函数
//缺点是要求输入和输出流是 basic_istream<char>和 basic_ostream<char>的派生类
      // Example 1-2(b): First attempt, sort of okay
      //
      void Process( basic_istream<char>& in,
                    basic_ostream<char>& out )
      {
         // ... do something more sophisticated,
         //         or just plain "out << in.rdbuf();"...
      }
//方法 B 的加强版   及采用方法 B 同时采用模版
//让编译器推导合适的参数
// Example 1-2(c): Better solution
//
template<typename C, typename T>
void Process( basic_istream<C,T>& in,
                  basic_ostream<C,T>& out )
{
   // ... do something more sophisticated,
   //         or just plain "out << in.rdbuf();"...
}
```

ITEM2:Prediacte,之一:remove()删除了什么

//1.removed()算法完成什么功能

//remove()并没有将对象从容器中删除，remove()执行完成后，容器的大小不变。

//简单的说 remove()只是用未删除的对象填补已删除的对象留下的缺口，每一个被

//删除的对象在尾部还说有相应的死亡对象。remove()返回指向第一个死亡的对象的

//迭代器。如果没有对象被删除，remove()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 ？ ？ ？

//2. 写一段代码用来删除 std::vector<int>中值为 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() );
```

//3.为删除容器中的第 n 个元素，写一段代码

```
// Example 2-3(a): Solving the problems
//
// Precondition:
//    - n must not exceed the size of the range
//
template<typename FwdIter>
FwdIter remove_nth( FwdIter first, FwdIter last, size_t n )
{
   // Check precondition. Incurs overhead in debug mode only.
   assert( distance( first, last ) >= n );

   // The real work.
   advance( first, n );
   if( first != last )
   {
      FwdIter dest = first;
      return copy( ++first, last, dest );
   }
   return last;
}
```

```
// Example 2-3(b)
//
// 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_;
};
// Example invocation
... remove_if( v.begin(), v.end(), FlagNth(3) )
```

//Example 2-3(a)的优点：
1.他是正确的
2.利用了 iterator traits 特性，特别是迭代器类别
   因而在随机访问的迭代器身上表现更加
//Example 2-3(b)的缺点：
 将在本短系列的第二部分详细分析

ITEM3:状态带来的问题

```
//1.什么是 predicate

// Example 3-1(a): Using a unary predicate //  一元 predicate
    //
    if( pred( *first ) )
    {
       /* ... */
    }
    // Example 3-1(b): Using a binary predicate//  二元 predicate
    //
    if( bpred( *first1, *first2 ) )
    {
       /* ... */
    }
    //例子
    // Example 3-1(c): A sample find_if()
        //
        template<typename Iter, typename Pred> inline
        Iter find_if( Iter first, Iter last, Pred pred )
        {
           while( first != last && !pred(*first) )
           {
              ++first;
           }
           return first;
        }
        //通过函数指针来使用 find_if
        // Example 3-1(d):
        // Using find_if() with a function pointer.
        //
        bool GreaterThanFive( int i )
        {
           return i > 5;
        }

        bool IsAnyElementGreaterThanFive( vector<int>& v )
        {
           return find_if( v.begin(), v.end(), GreaterThanFive )
                      != v.end();
        }
        //通过函数对象使用 find_if
        // Example 3-1(e):
        // Using find_if() with a function object.
```

```
//
class GreaterThanFive
   : public std::unary_function<int, bool>
{
public:
   bool operator()( int i ) const
   {
      return i > 5;
   }
};
bool IsAnyElementGreaterThanFive( vector<int>& v )
{
   return find_if( v.begin(), v.end(), GreaterThanFive() )
               != v.end();
}

//2.状态性 predicate 何时可用？
// Example 3-2(a):
// Using find_if() with a more general function object.
//
class GreaterThan
   : public std::unary_function<int, bool>
{
public:
   GreaterThan( int value ) : value_( value ) { }
   bool operator()( int i ) const
   {
      return i > value_;
   }
private:
   const int value_;
};

bool IsAnyElementGreaterThanFive( vector<int>& v )
{
   return find_if( v.begin(), v.end(), GreaterThan(5) )
               != v.end();
}
//比上面的例子更具有通用性
// Example 3-2(b):
// Using find_if() with a fully general function object.
//
template<typename T>
class GreaterThan
```

```
    : public std::unary_function<T, bool>
{
public:
    GreaterThan( T value ) : value_( value ) { }

    bool operator()( const T& t ) const
    {
        return t > value_;
    }

private:
    const T value_;
};

bool IsAnyElementGreaterThanFive( vector<int>& v )
{
    return find_if( v.begin(), v.end(), GreaterThan<int>(5) )
                != v.end();
}

//写一个函数对象，当他使用 n 次后返回 true
// Example 3-2(c)
// (From Item 2, Example 2-3(b))
//
// 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_;
};
//状态性 predicate 和非状态性 predicate 的区别:
//对于状态性 predicate，拷贝和他本身不是等同的,
//而非状态性 predicate，拷贝和他本身是相同的。
```

//3.为了让状态性的 predicate 正常工作，随算法的要求

1.算法绝不能对 predicate 做赋值
2.算法必须“以某个已知顺序”，将 predicate 作用到区间上的元素
//状态性 predicate 一般不能和标准协同工作，建议少用
//一个不完整方案，拷贝之间共享状态
// Example 3-3(a): A (partial) solution
// that shares state between copies.
//

```
class FlagNthImpl
{
public:
   FlagNthImpl( size_t nn ) : i(0), n(nn) { }
   size_t            i;
   const size_t n;
};

class FlagNth
{
public:
   FlagNth( size_t n )
      : pimpl_( new FlagNthImpl( n ) )
   {
   }

   template<typename T>
   bool operator()( const T& )
   {
      return ++(pimpl_->i) == pimpl_->n;
   }

private:
   CountedPtr<FlagNthImpl> pimpl_;
};
```

//CountPtr 的实现

```
template<typename T>
class CountedPtr
{
private:
   class Impl
   {
   public:
      Impl( T* pp ) : p( pp ), refs( 1 ) { }
```

```
        ~Impl() { delete p; }

        T*          p;
        size_t refs;
    };
    Impl* impl_;
public:
    explicit CountedPtr( T* p )
        : impl_( new Impl( p ) ) { }

    ~CountedPtr() { Decrement(); }

    CountedPtr( const CountedPtr& other )
        : impl_( other.impl_ )
    {
        Increment();
    }

    CountedPtr& operator=( const CountedPtr& other )
    {
        if( impl_ != other.impl_ )
        {
            Decrement();
            impl_ = other.impl_;
            Increment();
        }
        return *this;
    }

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

    T& operator*() const
    {
        return *(impl_->p);
    }

private:
    void Decrement()
    {
        if( --(impl_->refs) == 0 )
        {
```

```
            delete impl_;
        }
    }

    void Increment()
    {
        ++(impl_->refs);
    }
};
```

ITEM4:使用继承还是 traits

//1.什么是 traits 类

```
//in the standard itself std::char_traits<T> gives information about
//the character-like type T, particularly how to compare and manipulate
// such T objects. This information is used in such templates as std::basic_string
// and std:: basic_ostream to allow them to work with character types that are
// not necessarily char or wchar_t, including working with user-defined types
//for which you provide a suitable specialization of std::char_traits. Similarly,
// std::iterator_traits provides information about iterators that other templates,
// particularly algorithms and containers, can put to good use. Even std::numeric_limits
//gets into the traits act, providing information about the capabilities and behavior of
// various kinds of numeric types as they're implemented on your particular platform
//and compiler.
```

//2.示范如何检测好使用模版类成员。

```
//You want to write a class template C that can be instantiated only on types that
//have a member function named Clone() that takes no parameters and returns
//a pointer to the same kind of object.


// Example 4-2
//
// T must provide T* T::Clone() const
template<typename T>
class C
{
   // ...
};

//方法 1
// Example 4-2(a): Initial attempt,
// sort of requires Clone()
//
// T must provide /*...*/ T::Clone( /*...*/ )
template<typename T>
class C
{
public:
   void SomeFunc( const T* t )
   {
      // ...
      t->Clone();
      // ...
```

```
    }
};
//方法 2
// Example 4-2(b): Revised attempt, requires Clone()
//
// T must provide /*...*/ T::Clone( /*...*/ )
template<typename T>
class C
{
public:
    ~C()
    {
        // ...
        const T t; // kind of wasteful, plus also requires
                           // that T have a default constructor
        t.Clone();
        // ...
    }
};

//方法 3
// Example 4-2(c): Better, requires
// exactly T* T::Clone() const
//
// T must provide T* T::Clone() const
template<typename T>
class C
{
public:
    // in C's destructor (easier than putting it
    // in every C constructor):
    ~C()
    {
        T* (T::*test)() const = &T::Clone;
        test; // suppress warnings about unused variables
        // this unused variable is likely to be optimized
        // away entirely

        // ...
    }

    // ...
};
//方法 4
```

```
// 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;
      test; // suppress warnings about unused variables
      // ...
      return true;
   }

public:
   // in C's destructor (easier than putting it
   // in every C constructor):
  ~C()
   {
      assert( ValidateRequirements() );
   }

   // ...
};
```

方法 5

```
// 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; }
};
```

```
template<typename T>
class C : HasClone<T>
{
   // ...
};
//3.某个程序员想写一个模版，模版要求:他在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拥有的 Cloneable 基类

   // Example 4-3(a): An IsDerivedFrom1 value helper
      //
      // Advantages: Can be used for compile-time value test
      // Drawbacks:    Pretty complex
      //
      template<typename D, typename B>
      class IsDerivedFrom1
      {
         class No { };
         class Yes { No no[2]; };

         static Yes Test( B* ); // declared, but not defined
         static No Test( ... ); // declared, but not defined

      public:
         enum { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };
      };
      //
      // Example 4-3(a), continued: Using IsDerivedFrom1
      // helper to enforce derivation from Cloneable
      //
      template<typename T>
      class X
      {
         bool ValidateRequirements() const
         {
            // typedef needed because otherwise the , will be
            // interpreted as delimiting macro parameters to assert
            typedef IsDerivedFrom1<T, Cloneable> Y;

            // a runtime check, but one that can be turned
            // into a compile-time check without much work
            assert( Y::Is );

            return true;
```

```
    }

public:
    // in X's destructor (easier than putting it
    // in every X constructor):
    ~X()
    {
    assert( ValidateRequirements() );
    }

    // ...
};
//解法 2
// Example 4-3(b): An IsDerivedFrom2 constraints base class
//
// Advantages: Compile-time evaluation
//                        Simpler to use directly
// Drawbacks:    Not directly usable for compile-time value test
//
template<typename D, typename B>
class IsDerivedFrom2
{
    static void Constraints(D* p)
    {
        B* pb = p;
        pb = p; // suppress warnings about unused variables
    }

protected:
    IsDerivedFrom2() { void(*p)(D*) = Constraints; }
};
// Force it to fail in the case where B is void
template<typename D>
class IsDerivedFrom2<D, void>
{
    IsDerivedFrom2() { char* p = (int*)0; /* error */ }
};

//Now the check is much simpler:

// Example 4-3(b), continued: Using IsDerivedFrom2
// constraints base to enforce derivation from Cloneable
//
template<typename T>
```

```
class X : IsDerivedFrom2<T,Cloneable>
{
   // ...
};

//上面 2 种解法的综合
// 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; }
};

//选择不同实现版本
// 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_ ...
};

//4. 对 3 中的解法使用 traits

// 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();
   }
};
```

ITEM5:typename

//1.什么事 typename 有什么用处

//如果一个名称被使用在模版声明或定义中并且依赖于模版参数，
//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，或这个名称使用 typename 修饰

//2.下列代码有错吗

```
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 ...
};
```

//编译器不能解释出 instantiated_type()
//依赖性的名称只有在实例化时候可见
//应该向下面一样写

```
// Example 5-2(a): Somewhat horrid
      //
      template<typename A, typename B>
      class X : public X_base<B>
      {
      public:
         bool operator()(
            const typename X_base<B>::instantiated_type& i
            ) const
         {
            return i != typename X_base<B>::instantiated_type();
         }

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

```
//或这样
// 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 ...
};


//一个玩笑
#include <iostream>

class Rose {};

class A { public: typedef Rose rose; };

template<typename T>
class B : public T { public: typedef typename T::rose foo; };

template<typename T>
void smell( T ) { std::cout << "awful"; }

void smell( Rose ) { std::cout << "sweet"; }

int main()
{
   smell( A::rose() );
   smell( B<A>::foo() );    // :-)
}

//
#include <iostream>

class Rose {};
```

```
class A { public: typedef Rose rose; };

template<typename T>
class B : public T { public: typedef    T foo; };

template<typename T>
void smell( T ) { std::cout << "awful"; }

void smell( Rose ) { std::cout << "sweet"; }

int main()
{
   smell( A::rose() );
   smell( B<A>::foo() );    // :-)
}
```

ITEM6:容器，指针和不是容器的容器

```
//1. 下面代码
vector<char> v;
// ... populate v ...

char* p = &v[0];

// ... do something with *p ...

//我们总是说能使用迭代器的地方尽量不使用指针，然而有时候使用指针的地方
//不一定能使用迭代器

//使用迭代器的缺陷
1.能使用指针的地方不一定能使用迭代器
2.如果迭代器是一个对象而不是一个普通指针，使迭代器招致空间和性能的开销

//假设有一个 map<Name,PhoneNumber> 他在程序启动时载入 此后只用于查询，
//也就是给出一个名字我们很任意查询出电弧号码。我们要进行反向查询怎们办？
//一个简单的方法就是在构造一个数据结构 map<PhoneNumber*,Name*,Deref>
//这样避免了双倍的开销，因为第二个拥有的只是指针

//但是如果用迭代器是无法办到的

//2.下面的代码合法吗？安全吗？ 好吗？

vector<char> c;
//....填充 v....
char* p=&v[0];
//....使用*p.....


//这段代码完全合法
//这段代码是安全的 只要我们知道指针什么时候失效
//这段代码是有意义的，拥有一个指向容器内部的指针是绝对有意义的


//对代码的改进
vector<char> c;
//....填充 v....
vector<char>::iterator i=c.begin();
//....使用*i.....

//我们总是说能使用迭代器的地方尽量不使用指针，然而有时候使用指针的地方
//不一定能使用迭代器
```

//使用迭代器的缺陷
1.能使用指针的地方不一定能使用迭代器
2.如果迭代器是一个对象而不是一个普通指针，使迭代器招致空间和性能的开销

//假设有一个 map<Name,PhoneNumber> 他在程序启动时载入 此后只用于查询，
//也就是给出一个名字我们很任意查询出电弧号码。我们要进行反向查询怎们办？
//一个简单的方法就是在构造一个数据结构 map<PhoneNumber*,Name*,Deref>
//这样避免了双倍的开销，因为第二个拥有的只是指针

//3. 下面代码合法吗

```
template<typename T>
void f( T& t )
{
   typename T::value_type* p1 = &t[0];
   typename T::value_type* p2 = &*t.begin();
   // ... do something with *p1 and *p2 ...
}
```

//合法

1. To make the expression &t[0] valid, T::operator[]()
   must exist and must return something that understands
   operator&(), which in turn must return a valid T::value_type*
   (or something that can be meaningfully converted to a valid
   T::value_type*).

   In particular, this is true of containers that meet the standard's
   container and sequence requirements and implement the optional
   operator[](), because that operator must return a reference to
   the contained object. By definition, you can then take the
   contained object's address.

2. To make the expression &*t.begin() valid, T::begin() must exist,
   and it must return something that understands operator*(),
   which in turn must return something that understands operator&,
   which in turn must return a valid T::value_type* (or something
   that can be meaningfully converted to a valid T::value_type*).

//现在的问题是:对标准库中的支持 operator[]()的任何容器来说，上面的
//代码都是合法的，如果去掉&t[0]那行语句，对标准库的每个容器也是成立的
//唯独 std::vector<bool>除外

//下面的代码除 bool 外都成立

```
// Example 6-3: Also works for every T except bool
//
template<typename T>
void g( vector<T>& v )
{
   T* p = &v.front();
   // ... do something with *p ...
}
```

//vector<bool>被特化，所以他不是一个容器，不符合标准库的条件

//谨慎多早的优化

1.不要太早优化

2.除非确实必要，否则不要使用优化

3.即使那样，除非知道想要优化什么，哪里需要优化，否则不要优化

//vector<bool>是特化版本 具体见 c++ STL 或 MSDN

ITEM7:使用 vevtor 和 deque

```
//vector 用作数组:一个启发式的例子

//c 风格的数组
// Example 7-1(a): A function that operates
    // on a C-style array
    //
    int FindCustomer(
      const char* szName,         // name to search for
      Customer*     pCustomers, // pointer to beginning
                                        //    of a Customer array
      size_t          nLength )     //    with nLength entries
    {
      // performs a (possibly optimized) search and
      // returns the index at which the specified
      // name was found
    }

    //To use the above function, we might create,
    // populate, and pass in an array as follows:

    // Example 7-1(b): Using an array
    //
    Customer c[100];
      //-- populate the contents of c somehow —-
    int i = FindCustomer( "Fred Jones", &c[0], 100 );

//vector 的使用
vector<Customer> c;
    //-- populate the contents of c somehow —-
    int i = FindCustomer( "Fred Jones", &c[0], c.size() );

//尽量使用 vector  不要使用数组


//vector 是那种应该在默认情况下使用的序列...当大多数插入删除操作发生
//在序列的开头和结尾时，应该使用 deque

//缩小 vector 的空间

vector<C> c(10000);
//c.size()=10000 c.capacity()>=10000
c.erase(c.begin()+10,c.end());
////c.size()=10
```

```
c.reserve(10);
//没有改变 vector 容器的大小


//将 vector 容器缩小的方法

//
vector<C> c(10000);
//c.capacity()>=10000
c.erase(c.begin()+10,c.end());
//删除前 10 个元素外的所有元素
vector<C>(c).swap(c);
//缩小至合适的大小
//现在 c.capacity==c.size()  或略大于他

//通常 vector 或 deque 会保留而外的空间，以备将来增长的需要，
//防止增加元素频繁的分配有可能完全清除一个 vector 或 deque 吗？

//可以
// Example 7-3: The right way to clear out a vector.
    //
    vector<Customer> c( 10000 );
    // ...now c.capacity() >= 10000...

    // the following line makes
    // c be truly empty
    vector<Customer>().swap( c );

    // ...now c.capacity() == 0, unless the
    // implementation happens to enforce a
    // minimum size even for empty vectors
```

ITEM8:使用 set 和 map

//What's wrong with the following code? How would you correct it?

```
map<int,string>::iterator i = m.find( 13 );
if( i != m.end() )
{
   const_cast<int&>( i->first ) = 9999999;
}
```

//To what extent are the problems fixed by writing the following instead?

```
map<int,string>::iterator i = m.find( 13 );
if( i != m.end() )
{
   string s = i->second;
   m.erase( i );
   m.insert( make_pair( 9999999, s ) );
}
```

//可以通过 set::iterator 修改 set 的内容吗？

//关联式容器的一个基本准则
//一旦一个键值被插入容器。那么无论键值被什么方式修改，
//他在容器的相对位置不会改变。

//关联式容器的编程规范
1.规定 const 就是 const
//对于 map 和 multimap，键是不允许修改的，map<key,value>::iterator 指向的是
//pair<const key,value>,也就是说你只能改变值部分，
2.规定总是以先删除再重复插入的方式对键值进行修改

//To what extent are the problems fixed by writing the following instead?

```
map<int,string>::iterator i = m.find( 13 );
if( i != m.end() )
{
   string s = i->second;
   m.erase( i );
   m.insert( make_pair( 9999999, s ) );
}
```

//遗憾的是，这个解决方案并不是很好
1.假设 key 类型具有某种外部可见状态，这个状态值可以被其他代码得到
--例如一个指针指向缓冲区，系统的其他部分无需操作 key 就可以修改

这一缓冲区。在假如，那个外部可见的状态参与了 compare 的比较操作那么，即使对 key 对象一无所知，即使对使用这个关联式容器的代码一无所知，如果对外部可见的状态值进行修改，还是可以改变键值的相对顺序

2.假设 key 的类型为 String:compare 类型将这个 key 解释为文件名，在做比较时，比较的对象是文件的内容。这种情况下，即使键友元没有被修改，但是文件被另一个进程修改，建的相对次序还是可以被修改

//可以通过 set::iterator 修改 set 的内容吗？

//set::iterator 和 set::const_iterator 是常量迭代器，想要修改 set
//的对象，必须使用 const_cast

ITEM9:等同的代码吗

```
//1.Describe what the following code does:

// Example 9-1
//
f( a++ );

//Be as complete as you can about all possibilities

//解答
//若 f 是下面的一种
    #define f(x) x                              // once
        #define f(x) (x,x,x,x,x,x,x,x,x)        // 9 times
        #define f(x)                                    // not at all
  //设计准则  避免使用宏，宏往往使得代码更难以理解，从而难以维护

//若是函数

    //这种情况下，首先，a++会被赋值，结果被传递给函数做参数。通常
    //后置对象会以临时对象的形式返回 a  的旧值，所以 f 获取参数的方式
    //要么通过传值，要么通过 const 引用，但不会以传递非 const 引用，
    //因为非 const 引用不能绑定于临时对象
  //若是对象

    //f 是一个函数对象，类似于仿函数，f 的 operator()()可以以传值或 const
    //引用获取参数
  //若是类型名称

    //先对 a++求值，赋给 f 的临时对象

//若 a 是宏
  //这种情况下 a 可以表示任何东西
//若 a 是对象(有可能是内建类型)
  //这种情况下必须定义合适的后置递增操作符
  // Canonical form of postincrement:
        T T::operator++(int)
        {
          T old( *this ); // remember our original value

          ++*this;              // always implement postincrement
                                //    in terms of preincrement

          return old;         // return our original value
        }
```

```
        //设计准则:始终为重载的运算符保持正确的语义
//2.下面两段代码的区别

        // Example 9-2(a)
        //
        f( a++ );

        //(1) a++  递增 a 并返回 a 的旧值
        //(2)f()   将 a 的旧值传递给 f(),执行 f()
        // Example 9-2(b)
        //
        f( a );
        a++;
        //(1) f()  将 a 的旧值传递给 f(),然后执行 f()
        //(2)a++  递增 a 并返回 a 的旧值,旧值随后被忽略


        //这连个的主要后果
       //(1) f()产生异常的情况下，第一个代码保证 a++和他所有的副作用都成功执行结束，
       //第二个代码保证 a++没有执行，他的副作用一个也没发生
       //(2)  即使没有异常产生，如果 f()和 a.operator++(int)有可见的副作用，那么他们的
       //执行顺序很重要。

        //3.假设 f()是一个函数，他通过传值的方式获得参数，a 是一个类对象，
        //提供了正常语义的 operator++(int)
        //那么代码 1 和代码 2 的真正的区别？
        //对于完全正规的 c++，第二段代码不合法，第一段代码合法

         //如果将将 f()换成 list::erase()
         //第一种形式合法
         // Example 9-3(a)
            //
            // l is a list<int>
            // i is a valid non-end iterator into l
            //
            l.erase( i++ ); // OK, incrementing a valid iterator

            //第二段代码不合法:

            // Example 9-3(b)
            //
            // l is a list<int>
            // i is a valid non-end iterator into l
```

```
//
l.erase( i );
i++;                    // error, i is not a valid iterator
```

ITEM10:模版特殊化与重载

//1.模版特殊化的例子

//显示特殊化

//通用模版

```
template<typename T> void sort( Array<T>& v ) { /*...*/ };
```

//对 char* 特殊模版

```
template<> void sort<char*>( Array<char*>& );
```

//编译器会选出合适的模版

```
Array<int>      ai;
Array<char*> apc;

sort( ai );            // calls sort<int>
sort( apc );           // calls specialized sort<char*>
```

//2.什么是部分特殊化？给出一个例子

//The first template is the primary class template:

```
template<typename T1, typename T2, int I>
class A                    { };                    // #1
```

//We can specialize this for the case when T2 is a T1*:

```
template<typename T, int I>
class A<T, T*, I>      { };                    // #2
```

//Or for the case when T1 is any pointer:

```
template<typename T1, typename T2, int I>
class A<T1*, T2, I> { };                    // #3
```

//Or for the case when T1 is int and T2 is any pointer and I is 5:

```
template<typename T>
class A<int, T*, 5> { };                    // #4
```

//Or for the case when T2 is any pointer:

```
template<typename T1, typename T2, int I>
class A<T1, T2*, I> { };                    // #5
```

//编译器会选择合适的模版

```
A<int, int, 1>      a1;    // uses #1

A<int, int*, 1>     a2;    // uses #2, T is int,
                           // I is 1

A<int, char*, 5> a3;    // uses #4, T is char

A<int, char*, 1> a4;    // uses #5, T1 is int,
                           // T2 is char,
                           // I is 1

A<int*, int*, 2> a5;    // ambiguous:
                           // matches #3 and #5
//3.C++允许对函数模版进行重载
template<typename T1, typename T2>
void g( T1, T2 );                                  // 1
template<typename T> void g( T );                  // 2
template<typename T> void g( T, T );               // 3
template<typename T> void g( T* );                 // 4
template<typename T> void g( T*, T );              // 5
template<typename T> void g( T, T* );              // 6
template<typename T> void g( int, T* ); // 7
template<> void g<int>( int );                     // 8
void g( int, double );                             // 9
void g( int );                                     // 10

//将问题简化

//接受 2 个参数
template<typename T1, typename T2>
void g( T1, T2 );                                  // 1
template<typename T> void g( T, T );               // 3
template<typename T> void g( T*, T );              // 5
template<typename T> void g( T, T* );              // 6
template<typename T> void g( int, T* ); // 7
void g( int, double );                             // 9
//接受一个参数
template<typename T> void g( T );                  // 2
template<typename T> void g( T* );                 // 4
template<> void g<int>( int );                     // 8
void g( int );                                     // 10

//一下每条语句调用上面的哪个函数
int    i;
```

```
double                  d;
float                   f;
complex<double> c;

g( i );                 // a----------10
g<int>( i );            // b----------8
g( i, i );              // c----------3
g( c );                 // d----------2
g( i, f );              // e----------9
g( i, d );              // f----------9
g( c, &c );             // g----------6
g( i, &d );             // h----------7
g( &d, d );             // i----------5
g( &d );                // j----------4
g( d, &i );             // k----------1
g( &i, &i );            // l----------3
```

ITEM12:内联

//1.inline 有什么作用

//将一个函数说明为 inline 意味着告诉编译器:编译器可以将
//这个函数代码拷贝直接在每一个使用这个函数的地方，编译
//器可以这么做也可以不这么做。

//2.将函数内联会提高效率吗？

//不一定

//效率值指的是什么

1.程序体积

//内联不一定使代码体积变大，也可能使其减少

2.内存占用

//内联除了对代码体积有可能有影响外，基本对内存使用没影响

3.执行时间

//如果不是被声明为内联的函数平凡调用，对代码的执行时间没有影响

4.开发速度和编译时间

//普通函数修改时，调用者无需重新编译，只需要重新连接
//而内联函数必须重新编译，会增大编译时间。会在调试期影响开发速度

//3.何时决定使用内联函数？如何确定？

//除非你确定应该使用内联函数，否则尽量不要使用它
//空函数应该设计为内联，只要该函数一直保持为空

//所有被内联代码会增加耦合性，只要增加耦合性，内联总是会带来成本，
//绝对不要为某个东西先支付成本，除非你知道他带来好处--也就是回报大于支出

//在性能分析证明确实必要之前，避免内联或详细优化

ITEM13:缓式优化之一:一个普通的旧式 String

//分析下面简式的 String 类

```
namespace Original
{
   class String
   {
   public:
      String();                          // start off empty
     ~String();                          // free the buffer
      String( const String& ); // take a full copy
      void Append( char );         // append one character

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

   private:
      char*       buf_;                 // allocated buffer
      size_t      len_;                 // length of buffer
      size_t      used_;                // # chars actually used
   };
   }
```

//对上面的实现

```
namespace Original {

   String::String() : buf_(0), len_(0), used_(0) { }
   String::~String() { delete[] buf_; }
   String::String( const String& other )
               : buf_(new char[other.len_]),
                 len_(other.len_),
               used_(other.used_)
  {
      copy( other.buf_, other.buf_ + used_, buf_ );
  }
  void String::Reserve( size_t n )
  {
         if( len_ < n )
         {
            size_t newlen = max( len_ * 1.5, n );
            char*    newbuf = new char[ newlen ];
            copy( buf_, buf_+used_, newbuf );
            delete[] buf_;    // now all the real work is
            buf_ = newbuf;    //    done, so take ownership
            len_ = newlen;
           }
  }
```

```
    void String::Append( char c )
    {
       Reserve( used_+1 );
       buf_[used_++] = c;
    }


}
```

//缓冲区增长策略
1.精确增长
2.固定增量增长
3.指数增长

增长策略	Allocations	Char Copies	Wasted Space
精确增长	O(N)高常数	O(N)高常数	无
固定增量增长	O(N)	O(N)	O(1)
指数增长	O(logN)	O(1)	O(N)

ITEM14:缓式优化之二:引入缓式优化

```
//有时候，在得到一个字符串对象拷贝后，用户可能不会在使用中做任何修改，然后
//又丢弃了。对于一个 String，我们每次都做了分配新缓冲区(开销很昂贵)，可是如
//果所有用户只是从新字符串中读取数据然后摧毁。那么我所做的其实没有必要。我
//们可以让两个字符串在底层共享一个缓冲区,暂时避免拷贝操作。只是在确实知道需
//要拷贝的时候，也就是当其中一个对象试图修改这个字符串的时候，我才进行拷贝
//如果用户永远不修改这个拷贝，我们就永远不做额外的工作。

//下面是一个缓式拷贝的实现。对底层字符串实体实施引用计数
namespace Optimized
{
  class StringBuf
  {
  public:
    StringBuf();                    // start off empty
   ~StringBuf();                    // delete the buffer
    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

  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

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

  private:
    StringBuf* data_;
```

```
    };
}
//实现  Optimized::StringBuf and Optimized::String，
//你可能需要辅助函数 String::AboutToModify()

    namespace Optimized
    {

        StringBuf::StringBuf()
              : buf(0), len(0), used(0), refs(1) { }
          StringBuf::~StringBuf() { delete[] buf; }
    void StringBuf::Reserve( size_t n )
    {
        if( len < n )
        {
            size_t newlen = max( len * 1.5, n );
            char*    newbuf = new char[ newlen ];
            copy( buf, buf+used, newbuf );

            delete[] buf;      // now all the real work is
            buf = newbuf;      //    done, so take ownership
            len = newlen;
        }
    }
    String::String() : data_(new StringBuf) { }
    String::~String()
    {
        if( --data_->refs < 1 )    // last one out ...
        {
            delete data_;    // ... turns off the lights
        }
    }
    String::String( const String& other )
        : data_(other.data_)
    {
        ++data_->refs;
    }
    void String::AboutToModify( size_t n )
    {
        if( data_->refs > 1 )
        {
            auto_ptr<StringBuf> newdata( new StringBuf );
            newdata->Reserve( max( data_->len, n ) );
            copy( data_->buf, data_->buf+data_->used, newdata->buf );
```

```
        newdata->used = data_->used;

        --data_->refs;                          // now all the real work is
        data_ = newdata.release(); //    done, so take ownership
    }
    else
    {
        data_->Reserve( n );
    }
}
void String::Append( char c ) {
        AboutToModify( data_->used+1 );
        data_->buf[used++">data_->used++] = c;
    }

}
```

ITEM15:缓式优化之三:迭代器与引用

```
//运用 copy-on-write 技术的 Optimized::String 类，增加了 Length(),operator[]()两个函数
namespace Optimized
{
  class StringBuf
  {
  public:
    StringBuf();                    // start off empty
   ~StringBuf();                    // delete the buffer
    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

  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 );
                                         // lazy copy, ensure len>=n
      StringBuf* data_;
    };
```

```
    }

  //This allows code such as the following:

  if( s.Length() > 0 )
  {
    cout << s[0];
    s[0] = 'a';
  }
```

//实现 Optimized::String 新成员，增加函数后，其他的什么成员需要修改吗？

//实现 Optimized::String 新成员

```
namespace Optimized
  {

    size_t String::Length() const
    {
      return data_->used;
    }
   }
   //对于 operator[]()比较麻烦，下面是比较幼稚的做法
   // BAD: Na?ve attempt #1 at operator[]
       //
       char& String::operator[]( size_t n )
       {
         return data_->buf[n];
       }

       const char String::operator[]( size_t n ) const
       {
         return data_->buf[n];
       }
       //如果用下面的测试
       // Example 15-1: Why attempt #1 doesn't work
       //
       void f( const Optimized::String& s )
       {
         Optimized::String s2( s ); // take a copy of the string
         s2[0] = 'x';                    // oops: also modifies s!
       }

       //对于 operator[]()比较麻烦，下面是比较幼稚的做法
```

```
// BAD: Inadequate attempt #2 at operator[]
//
char& String::operator[]( size_t n )
{
  AboutToModify( data_->len );
  return data_->buf[n];
}

const char String::operator[]( size_t n ) const
{
  // no need to check sharing status this time
  return data_->buf[n];
}
//如果用下面的测试
// Example 15-2: Why attempt #2 doesn't work either
//
void f( Optimized::String& s )
{
  char& rc = s[0];    // take a reference to the first char
  Optimized::String s2( s ); // take a copy of the string
  rc = 'x';                  // oops: also modifies s2!
}
//我们要做的就是不共享 String，我们可以将 String 标记为永远不共享，
//也可以标记为暂时不共享。如果 operator[]返回一个指向字符串内部的引用，
//那么下一次修改后
//我们必须让那个引用无效。例子:
// Example 15-3: Why references are
// invalidated by mutating operations
//
void f( Optimized::String& s )
{
  char& rc = s[0];
  s.Append( 'i' );
  rc = 'x';     // 错误:如果 s 执行了重新分配
}                    // 缓冲区可能被移动

//增加了新成员后，有些成员是需要修改的
//对于 operator[]正确的方案
//为了方便，增加一个新静态成员
//并对 AboutToModify()做适当修改
//因为我们需要在不止一个函数中复制 StringBuf
//我们还需要将这一逻辑放进一个单独函数中
//不管怎样 Stringbuf 现在得有一个自己的拷贝构造函数
const size_t String::Unshareable =numeric_limits<size_t>::max();
```

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

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];
}

//如果设置了不可共享的状态值，我们还要让 String 的拷贝构造函数
//来使用他
String::String( const String& other )
{
   // If possible, use copy-on-write.
   // Otherwise, take a deep copy immediately.
   //
```

```
            if( other.data_->refs != Unshareable )
            {
               data_ = other.data_;
               ++data_->refs;
            }
            else
            {
               data_ = new StringBuf( *other.data_ );
            }
         }
         //String 的析构函数也小小改动
         String::~String()
         {
            if( data_->refs == Unshareable || --data_->refs < 1 )
            {
               delete data_;
            }
         }
         //String 的其他函数还是像最初写的那样
         String::String() : data_(new StringBuf) { }

      void String::Append( char c )
      {
         AboutToModify( data_->used+1 );
         data_->buf[data_->used++] = c;
      }

   }
   //代码整合:对 StringBuf::Reserve(); 做小小的修改，他会对新缓冲区大小
   //进行上舍入计算，使他的值增大至下一个 4 的倍数，从而保证内存缓冲
   //区的大小总是 4 字节的倍数，这是为效率
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

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()
    {
      if( data_->refs == Unshareable || --data_->refs < 1 )
      {
        delete data_;
      }
    }

    String::String( const String& other )
    {
      // If possible, use copy-on-write.
      // Otherwise, take a deep copy immediately.
      //
      if( other.data_->refs != Unshareable )
      {
        data_ = other.data_;
        ++data_->refs;
```

```
    }
    else
    {
        data_ = new StringBuf( *other.data_ );
    }
}

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

        void String::Append( char c )
        {
            AboutToModify( data_->used+1 );
            data_->buf[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];
        }

    }
```

ITEM16:缓式优化之四:多线程环境

```
    //Optimized::String 不是现成安全的
 void String::AboutToModify(
    size_t n,
    bool      markUnshareable /* = false */
    )
  {
    if( data_->refs > 1 && data_->refs != Unshareable )
  {//如果两个线程中，线程 1 对 data_->refs > 1 求值而线程 2 正在更改 data_->refs >1
    //的值，就会出问题
    /* ... etc. ... */
    }
  }
  //针对下面两种情况，分别演示如何使 Optimized::String 变得线程安全
    (a) 假设具备获取(get) 设置(set) 和比较(compare)的原子操作
    (b) 假设没有以上原子操作

    //首先解决 b，在其他工作开始前，需要在 Optimized::StringBuf 中
    //增加一个成员对象，其名称为 m
     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& );
```

```
    };
//必定会同时操作两个 StringBuf 对象的函数只有一个，即拷贝构造函数，
//String 只会在两个地方调用 StringBuf 的拷贝构造函数
//(String 自身的拷贝构造函数和 AbouttoModify()) ，注意，String 只需对引用计数
//访问进行串行优化，因为根据定义，没有 String 会对共享的 StringBuf 进行任何操作

//缺省的构造函数不需要加锁
String::String() : data_(new StringBuf) { }
    //析构函数只需对 refs 计数值的查询和更新操作加锁
    String::~String()
    {
      bool bDelete = false;
      data_->m.Lock(); //---------------------------
      if( data_->refs == Unshareable || --data_->refs < 1 )
      {
        bDelete = true;
      }
      data_->m.Unlock(); //-------------------------
      if( bDelete )
      {
        delete data_;
      }
    }
    //对于 String 的拷贝构造函数，我们可以假设，在这个操作期间，其他的 String
    //的数据缓冲区不会被修改或移动，因为，对可见对象的访问进行串行优化是调用
    //者的责任，但是对引用计数本身访问，我们还是得串行化处理
    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_ );
      }
    }

//再来看看 AboutToModify，这里深拷贝操作实际上都被加锁。严格来说
```

```
//只需要在两个地方加锁，即查看 refs 值以及最后更新 refs 值的地方。
//但是我们还是对整个操作加锁。
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;
        }
        //其他函数不需要修改
        void String::Append( char c )
        {
              AboutToModify( data_->used+1 );
              data_->buf[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];
           }
```

```
        }
        //对于问题 a 的解决
        //这里是一个线程安全的 String 实现，它使用了三个函数，
        //IntAtomicGet(), and IntAtomicDecrement() and IntAtomicIncrement()
        //这三个函数可以安全的返回新值，我们本质是做与前面相同的事情，
        //但是这里，我们只使用原子操作来串行访问
namespace Optimized{
   String::String() : data_(new StringBuf) { }
   String::~String()
       {
           if( IntAtomicGet( data_->refs ) == Unshareable ||
                IntAtomicDecrement( data_->refs ) < 1 )
         {
           delete data_;
         }
       }


      String::String( const String& other )
      {
          if( IntAtomicGet( other.data_->refs ) != Unshareable )
           {
            data_ = other.data_;
              IntAtomicIncrement( data_->refs );
           }
          else
          {
             data_ = new StringBuf( *other.data_ );
          }
      }
      void String::AboutToModify(
            size_t n,
            bool      markUnshareable /* = false */
        )
        {
           int refs = IntAtomicGet( data_->refs );
           if( refs > 1 && refs != Unshareable )
           {
              StringBuf* newdata = new StringBuf( *data_, n );
              if( IntAtomicDecrement( data_->refs ) < 1 )
              {                              // just in case two threads
                   delete newdata;    //    are trying this at once
              }
              else
```

```
      {                                // now all the real work is
        data_ = newdata; //    done, so take ownership
      }
    }
    else
    {
      data_->Reserve( n );
    }
    data_->refs = markUnshareable ? Unshareable : 1;
  }

  void String::Append( char c )
  {
    AboutToModify( data_->used+1 );
    data_->buf[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];
  }
}
//在性能上的影响
//没有原子操作，copy-on-write 往往会招致重大的性能损失，
//即使有了原子整数操作，COW 也会使一个普通的 String 操作
//耗时增长 50%，甚至在单线程也是如此。
```

ITEM17:构造函数失败之一:对象的生命期

```
//看下面这个类
// Example 17-1
//
class C : private A
{
   B b_;
};
//在 C 的构造函数中，如何捕捉从基类镀锡 i 昂或成员对象构造函数
//中抛出的异常


//这是 function try block 的用武之地
// Example 17-1(a): Constructor function try block
//
C::C()
try
   : A ( /*...*/ )      // optional initialization list
   , b_( /*...*/ )
{
}
catch( ... )
{
   // We get here if either A::A() or B::B() throws.

   // If A::A() succeeds and then B::B() throws, the
   // language guarantees that A::~A() will be called
   // to destroy the already-created A base subobject
   // before control reaches this catch block.
}

//下面代码
// Example 17-2
//
{
   Parrot p;
}
//这个对象的生命周期何时开始？
//当他的构造函数成功执行完毕并正常返回之时，也就是说，当控制抵达
//构造函数整体末尾之时或是完成更早一个 returne 语句时

//何时结束？
//当他的析构函数开始执行之时，也就是说抵达析构函数开始处
```

//在对象的声明周期外，对象处于什么状态？
//在生命周期开始之前和结束之后，对象的状态完全一样，没有对象存在

//最后，如果他的构造函数抛出一个异常，那将意味着什么？
//意味着构造失败，对象没有存在过，他的生命周期没有开始过

//对于 C++的构造函数，总结如下:只会是二者之一

(A) 构造函数正常返回，对象真实存在
(B) 构造函数抛出异常后退出，对象从未存在。

ITEM18:构造函数失败之二:吸收异常

```
//在下面代码中，如果 A 或 B 的构造函数抛出异常，C 的构造函数有可能
//吸收这个异常并完全不发出异常吗？
// Example 18-1
    //
    class C : private A
    {
      B b_;
    };
    //如果不考虑对象的生命周期，我们可以写出下面的代码
    // Example 18-1(a): Absorbing exceptions?
    //
    C::C()
    try
      : A ( /*...*/ )      // optional initialization-list
      , b_( /*...*/ )
    {
    }
    catch( ... )
    {
      // ?
    }
    //这个 try block 如何退出？注意以下几点
    //1.这个处理程序不能简单的以 return  返回，因为这个不合法
    //2.如果处理程序写上 throw，那么无论 A::A()或 B::B()最初抛出
    //什么异常，他都会重新抛出
    //3.如果处理程序抛出其他某个异常，最终抛出的也就是那个异常，
    //而不是基类或成员子对象的构造函数最初抛出的异常
    //4.如果处理程序没有以抛出异常的方式退出，那么在控制抵达构
    //造函数或析构函数的 catch block 的末尾时，最初的异常会被自动的重新抛出

    //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。


//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
        成员子对象的构造函数抛出的异常，此外没有其他用途。
法则 2: 析构函数的 function try block 没有什么实际用处，
        因为析构函数不应该产生异常
法则 3: 其他所有的 function try block 都没有用处。
法则 4: 获取未管理资源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。
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
抛出虽有异常，这个异常必须留有余地。
法则 7: 使用 Pimpl 手法保存类内的可选部分。如果一个成员对象的构
造函数可以抛出异常，但是你不需要这个成员也可以照常运作，
那么你可以将这个成员用指针保存，并通过 null 判断是否得到这个对象。
法则 8: 尽量以使用获得资源才是初始化的技术类管理资源

```
//
// Example 18-1(c): Auto object
      //
      {
        X x;//一旦异常发生，无论什么原因，程序都不会在继续进行。
        g( x ); // do something else
      }

//为了能够狗在 C 的构造函数上安全的放上一个空 throw 规范，
//A 和 B 必须满足的最小条件？
//如果你想给一个构造函数加上空 throw 规范，我们就得承认，
//所有的基类和成员子对象绝对不会抛出异常。
//如果基类和成员子对象抛出异常，那么程序会 terminate()
```

ITEM19:未捕获的异常

```
//std::uncaught_exception()完成什么功能

//标准 uncaught_exception()函数提供一种方法，
//让你知道当前是否有一个异常正处于活动状态。

//抛出异常的析构函数带来的问题
// Example 19-1: The problem
//
class X {
public:
   ~X() { throw 1; }
};

void f() {
   X x;
   throw 2;
} // calls X::~X (which throws), then calls terminate()
//如果一个异常已经处于活动状态，如果此时析构函数抛出异常，
//程序会终止，这不是件好事


//请看下面的代码
// Example 19-2: The wrong solution
//
T::~T()
{
   if( !std::uncaught_exception() )
   {
      // ... code that could throw ...
   }
   else
   {
      // ... code that won't throw ...
   }
}
//这个错误的方案为什么不正确？
//上面的代码不会想你想象的那样做，这是因为即使是在可以安全抛出异常的时候
//他也会采用那条不抛出异常的执行路径。

// Example 19-2(a): Why the wrong solution is wrong
//
U::~U()
{
```

```
    try
    {
        T t;
        // do work
    }
    catch( ... )
    {
        // clean up
    }
}
//如果在异常传播期间，由于堆栈展开使得 U 对象被摧毁，那么 T::~T()将不会
//使用那条会抛出异常的执行路径，即使此时可以安全的抛出异常，因为 T::~T()
//不知道在这种情况下，他已经受到了外部 U::~U()代码块的保护

//另一种不同形式的错误方案
// Example 19-3: Variant, another wrong solution
//
Transaction::~Transaction()
{
    if( uncaught_exception() )
    {
      RollBack();
    }
    else
    {
      // ...
    }
}
//同样，在堆栈展开期间，如果一个被调用的析构函数用到了 Transaction，
//这段代码不会做正确的事情。
// Example 19-3(a): Why the variant wrong solution
   // is still wrong
   //
   U::~U() {
      try {
         Transaction t( /*...*/ );
         // do work
      } catch( ... ) {
         // clean up
      }
   }
//总之，Example 19-2 不会像你想象的那样工作。
//正确的方案
// Example 19-4: The right solution
```

```
//
T::~T() /* throw() */
{
   // ... code that won't throw ...
}
//另一个正确的方案
// 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，写析构函数时，
//就像他已经有了一个 throw()异常规范一样

//设计准则：如果析构函数调用了一个可能会抛出异常的函数，
//一定要让这个调用包装在 try/catch block 中，以防止异常逃出析构函数


//关于 uncaught_exception() ，建议最好不要使用
```

ITEM20:未管理指针存在问题之一:参数求值

```
    //1.下面语句中，你能说出他们对函数 f,g,h 和表达式 expr1,expr2 的求值顺序吗
    // Example 20-1(a)
    //下面代码可以先对 expr1 求值，也可以后对 expr1 求值，
    //也可以 expr1,expr2 交叉求值
    f( expr1, expr2 );

    // Example 20-1(b)
    //expr1 必须在 g()被调用前求值，expr2 必须在 h()被调用前求值
    //h(),g()必须在 f()被调用前执行完毕
    //expr1,expr2 可以交叉运行，任何函数调用不可交叉运行
    f( g( expr1 ), h( expr2 ) );

  //c++的规则
  //(1)在函数调用之前，对函数所有参数的求值全部完成，如果参数是表达式，
  //那么表达式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不会开始求值或继续求值，
  //直至被调用函数执行结束，函数不会交叉运行
  //(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，除非另有其他规则限制

  //2.下面代码有异常安全隐患吗？
    // Example 20-2
    //

    // In some header file:
    void f( T1*, T2* );

    // In some implementation file:
    f( new T1, new T2 );

  //编译器的执行顺序若如下
  //(1)为 T1 分配内存
  //(2)构造 T1
  //(3)为 T2 分配内存
  //(4)构造 T2
  //(5)调用 f()

  //若异常发生在第三步或第四步，那么 T1 将不会销毁，造成内存泄漏

   //编译器的执行顺序若如下
  //(1)为 T1 分配内存
  //(2)为 T2 分配内存
  //(3)构造 T1
```

//(4)构造 T2
//(5)调用 f()

//如果异常在第三步，那么 T1 分配对象会被释放，T2 分配的将不会释放
//如果异常在第四步，那么 T1 分配对象已经完成，造成内存泄漏

ITEM21:未管理指针存在问题之二:使用 auto_ptr

```
//1.下面代码异常安全还存在吗？较上个主题的代码改进在哪？
// Example 21-1
//
// In some header file:
void f( auto_ptr<T1>, auto_ptr<T2> );

// In some implementation file:
f( auto_ptr<T1>( new T1 ), auto_ptr<T2>( new T2 ) );
//多了 auto_ptr 之后的，编译器的执行步骤如下(一)
//(1)为 T1 分配内存
//(2)构造 T1
//(3)为 T2 分配内存
//(4)构造 T2
//(5)构造 auto_ptr<T1>
//(6)构造 auto_ptr<T2>
//(7)调用 f()
//上面的代码在第三步或第四步任何一步出现问题，上个主题的异常仍然发生
//多了 auto_ptr 之后的，编译器的执行步骤如下(二)
//(1)为 T1 分配内存
//(2)为 T2 分配内存
//(3)构造 T1
//(4)构造 T2
//(5)构造 auto_ptr<T1>
//(6)构造 auto_ptr<T2>
//(7)调用 f()
//上面的代码在第三步或第四步任何一步出现问题，上个主题的异常仍然发生

//2.演示如何写一个 auto_ptr_new 工具，来解决(1)中问题，并可以如下调用
// Example 21-2
    //

    // In some header file:
    void f( auto_ptr<T1>, auto_ptr<T2> );

    // In some implementation file:
    f( auto_ptr_new<T1>(), auto_ptr_new<T2>() );

    //最简单的方案，提供下面一个模版
    // Example 21-2(a): Partial solution
    //
    template<typename T>
    auto_ptr<T> auto_ptr_new()
    {//只对有缺省构造函数有效
```

```
    return auto_ptr<T>( new T );
}
//这解决了一场安全问题，他产生的代码不会有资源泄漏问题，
//因为我没问你只有两个函数，而我们知道，函数不会交叉运行

//请看下面的运算次序,无论哪步出现异常，不会有内存泄漏
//(1)  调用 auto_ptr<T1>()
//(2) 调用 auto_ptr<T1>()

//解决没有缺省函数的问题，时期具有通用性
// 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 ) );
}

//一个更好的解决方案，将两个临时的 auot_ptr 对象分别放进各自的命名变量中
// Example 21-1(b): A simpler 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 );
}
```

//设计原则: 在各自独立的程序语句中执行每一个显示的资源
//分配(new)，并将通过(new)分配的资源立即交给管理者对象(auto_ptr)

ITEM22:异常安全与类的设计之一:拷贝赋值

```
//异常安全的三个级别
//1.基本保证
//2.强烈保证
//3.不抛出异常的保证

//标准的 auto_ptr 的一个重要特性是，auto_ptr 操作绝不会抛出异常

//具有强烈常安全性的拷贝赋值的规范是什么？

//一个是提供一个不抛出异常的 swap()
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
}

//3.请看下面这个类
// Example 22-1: The Cargill Widget Example
//
class Widget
{
public:
   Widget& operator=( const Widget& ); // ???
   // ...
private:
   T1 t1_;
   T2 t2_;
};
//假设 T1，T2 的某一个操作会抛出异常。如果不改变类的结构，
//有可能写出具有异常安全的 Widget::operator=(const Widget&)吗？

//如果不改变结构，我们无法写出具有异常安全的 Widget::operator=(const Widget&)

//4.说明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
//并可以很容易的使得那个类的拷贝赋值具有强烈的异常安全训过，
//其他场合，我们看到过这种技术？
```

```
//尽管不改变 Widget 的结构我们无法写出具有异常安全的 Widget::operator=()
//但是通过下面转换技术，我们可以实现一个具有近乎强烈异常安全的赋值。即
//通过指针而不是值来拥有成员对象，最好使用 Pimpl 转换手法，将一切隐藏在
//单个指针之后

// Example 22-2: The general solution to
// Cargill's Widget Example
//
class Widget
{
public:
   Widget();   //用新的 WidgetImpl 初始化  pimpl_

   ~Widget(); //这个析构函数必须提供，因为隐式生成的版本会
         //   导致使用上的问题
         //   (see Items 30 and 31)
   Widget& operator=( const Widget& );

   // ...

   private:
      class WidgetImpl;
      auto_ptr<WidgetImpl> pimpl_;
      // ... 提供可以正常工作的拷贝
      //        构造函数或者进制它  ...
};

// 然后一般是一个单独的实现文件中
// implementation file:
//
class Widget::WidgetImpl
{
public:
   // ...
   T1 t1_;
   T2 t2_;
};
//注意，如果使用 auto_ptr 成员你必须
//(1) 要么，你必须将 WidgetImpl 的定义提供给 Widget，要么，如果你很想隐藏
//WidgetImpl，你就必须为 Widget 写出自己的析构函数，即使这个析构函数很简单
//(2)对于 Widget，还应该提供自己的拷贝构造函数和赋值函数，一般来说你不希望
//类的成员具有"拥有权转移"语义。如果你有另外一个智能指针，你可以考虑
//代替 auto_ptr,但是上述原则依然重要现在我们可以很容易实现一个不抛出异
//常的 swap，那么我们很容器写出一个近似满足强烈异常安全保证的拷贝赋值函数
```

```
void Widget::Swap( Widget& other ) /* throw() */
 {
    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
 }

//得到的结论
//结论 1:  异常安全影响类的设计
//结论 2:  总能让你的代码具有强烈的异常安全性
//结论 3:  明智的使用指针
    //结论 3: (1)指针是你的敌人，它带来的种种问题是 auto_ptr 要消除的问题
    //结论 3: (2)指针是你的朋友，使用指针不会带来异常
```

ITEM23:异常安全与类的设计之二:继承

//1.Is-Implemented-In-Terms-Of 的含义是什么？

//如果 T 在他的实现中以某种形式使用了另一种类型 U，就称 T
//Is-Implemented-In-Terms-Of U。以某种形式使用这一措辞当然
//留有很大余地，他表示的范围很广，如 T 可以是 U 的一个适配
//器代理或包装类;或者 T 仅仅只是在他的实现细节中偶尔用到 U。

//T IIITO U 通常意味着:要么，T 有一个 U 例如:

```
// Example 23-1(a): "T IIITO U" using Has-A
//
class T
{
   // ...
private:
   U* u_;    // or by value or by reference
};
```

//要么 T 非共有派生于 U，例如

```
// Example 23-1(b): "T IIITO U" using derivation
//
class T : private U
{
   // ...
};
```

//2.在 C++中，Is-Implemented-In-Terms-Of 可以通过非公有继承或包容/委托来表达。
//具体来说，在写一个类 T 的时候，如果它要用类 U 来实现。两个主要的选择是:
//让 T 从 U 私有继承，或者让 T 包含一个 U 成员对象。

//继承容易被使用过度，一条有效的工程设计原则:将耦合性降至最低，
//如果一种关系可以用多种有效方式表达请使用关系最弱的那一个，
//既然继承近似乎是 C++中可以表达的最强烈的关系，仅次于友元，
//那么，只有在没有更弱的关系时，我们才使用它。

//最小耦合性原则无疑会直接影响到代码的健壮性，编译时间，以及其他可见后果。
//有趣的是，为了实现 IIITO ,在继承和委托之间所做的选择还会有异常安全性上的牵连
//

//对异常安全的影响
//如果想用 Has-A 来表达 IIITO 关系，我们应该怎样写 T::operator=()。

```
// Example 23-2(a): "T IIITO U" using Has-A
//
```

```
class T
{
   // ...
   private:
      U* u_;
};
//近似强烈安全性的 T::operator=()
T& T::operator=( const T& other )
{
    U* temp = new U( *other.u_ );      // do all the work
                                                 //    off to the side

    delete u_;           // then "commit" the work using
    u_ = temp;            //    nonthrowing operations only
    return *this;
}


//一旦 U 和 T 之间的关系涉及到任何方式的继承，问题会有什么变化
// Example 23-2(b): "T IIITO U" using derivation
   //
   class T : private U
   {
      // ...
   };
   T& T::operator=( const T& other )
   {
      U::operator=( other );    // ???如果可以在开始修改目标的情况下抛出异常
      //我们无法写出具有强烈安全性的 T::operator=(),除非 U 通过其他某个
      //函数提供了合适功能
      return *this;
   }
   //换句话说，如果 T 为他的成员函数 T::operator=()提供异常安全保证的能力必然
   //依赖与 U 的安全和保证。在表达 T 和 U 之间的关系时，上述代码使用了最大可
  //能的紧密关系从而导致最大可能的高耦合性。


//总结:
//松散的耦合性促进程序的正确性(包括异常安全性)，紧密的耦合性降低程序的最大可能
//正确性(包括异常安全)
```

ITEM24:为什么要使用多继承

//1.什么是多继承(MI)?在 C++中引入多继承带来了哪些额外的可能性和复杂性?

//多继承代码示例

```
class Derived : public Base1, private Base2
{
   //...
};
```

//在 C++中引入多继承所带来的可能性是:一个类的同一个基类可能会不只一次的作为
//它的基础类出现。这时我们必须引入虚拟继承。

//设计准则:尽量避免多继承，但不是不用。

//2.多继承到底有必要吗？为什么
//多继承在以下三个方面还是有必要的
//(1) 结合使用程序模块或程序库
//(2) interface 类
//(3) 易于(多态)使用

//有时候单纯的从两个不同的基类继承并没有必要，相反，
//我们要让每一个继承都有不同的理由

ITEM25:模拟多继承

```
//看下面的例子
class A
{
public:
   virtual ~A();
   string Name();
private:
   virtual string DoName();
};

class B1 : virtual public A
{
   string DoName();
};

class B2 : virtual public A
{
   string DoName();
};

A::~A() {}
string A::Name(){      { return DoName(); }
string A::DoName()    { return "A"; }
string B1::DoName() { return "B1"; }
string B2::DoName() {       return "B2"; }

class D : public B1, public B2
{
     string DoName() { return "D";    }
};

//不运用 MI，写一个与上面等价的类 D，即，在不使用多继承的情况下，演示你能找到
//拐弯抹角解决问题的方法，让 D 得到相同的效果和可用性，同时又要避免对调用者代
//码的句法多做修改，能做到吗？

//首先考虑下面程序的情况，然后开始设计
void f1( A&    x ) { cout << "f1:" << x.Name() << endl; }
void f2( B1& x ) { cout << "f2:" << x.Name() << endl; }
void f3( B2& x ) { cout << "f3:" << x.Name() << endl; }

void g1( A      x ) { cout << "g1:" << x.Name() << endl; }
void g2( B1    x ) { cout << "g2:" << x.Name() << endl; }
void g3( B2    x ) { cout << "g3:" << x.Name() << endl; }
```

```
int main()
{
   D      d;
   B1* pb1 = &d;      // D* -> B* conversion
   B2* pb2 = &d;
   B1& rb1 = d;       // D& -> B& conversion
   B2& rb2 = d;

   f1( d );           // polymorphism
   f2( d );
   f3( d );

   g1( d );           // slicing
   g2( d );
   g3( d );

                      // dynamic_cast/RTTI
   cout << ( (dynamic_cast<D*>(pb1) != 0) ? "ok " : "bad " );
   cout << ( (dynamic_cast<D*>(pb2) != 0) ? "ok " : "bad " );

   try
   {
      dynamic_cast<D&>(rb1);
      cout << "ok ";
   }
   catch(...)
   {
      cout << "bad ";
   }

   try
   {
      dynamic_cast<D&>(rb2);
      cout << "ok ";
   }
   catch(...)
   {
      cout << "bad ";
   }
}
```

```
//下面的方案和我们的要求非常接近
class D : public B1
  {
  public:
    class D2 : public B2
    {
    public:
      void      Set ( D* d ) { d_ = d; }
    private:
      string DoName();
      D* d_;
    } d2_;

    D() { d2_.Set( this ); }

    D( const D& other ) : B1( other ), d2_( other.d2_ )
                                  { d2_.Set( this ); }

    D& operator=( const D& other )
     {
          B1::operator=( other );
          d2_ = other.d2_;
          return *this;
      }

    operator B2&()    { return d2_; }

    B2& AsB2()    { return d2_; }

  private:
    string DoName()    { return "D"; }
  };

  string D::D2::DoName(){ return d_->DoName(); }

//上面的代码的不足之处
//(1) 由于提供 operator B2&(),较之指针，引用就得特殊对待
//(2) 将 D 做为 B2 使用时，调用者代码必须显示的调用 D::AsB2()
//意味着，在测试程序中，必须将 B2 *pb2=&d;修改为 B2 *pb2=&d.AsB2();
//(3) 不能通过 dynamic_cast 将 D 转换为 B2*。

//你不是经常使用多继承，一旦你使用多继承，那么说明你真的需要多继承
```

ITEM26:多继承与连体双婴问题

```
//请看下面的两个类
class BaseA
{
   virtual int ReadBuf( const char* );
   // ...
};

class BaseB
{
   virtual int ReadBuf( const char* );
   // ...
};
//BaseA 和 BaseB 有一个共同点---他们显然都想被用作基类，但是除此之外他们
//毫不相干，他们的 ReadBuf()函数用来做不同的事情，而且这两个类还来自不同
//程序供应商

//示范如何写一个 Derived 类，这个类从 BaseA 和 BaseB 共有继承，
//而且还要对两个 ReadBuf()进行改写，让他们做不同的事情

//下面的写法不正确
// Example 26-1: Attempt #1, doesn't work
      //
      class Derived : public BaseA, public BaseB
      {
         // ...

         int ReadBuf( const char* );
               // overrides both BaseA::ReadBuf()
               // and BaseB::ReadBuf()
      };
      // Example 26-1(a): Counterexample,
      // why attempt #1 doesn't work
      //
      Derived d;
      BaseA*    pba = d;
      BaseB*    pbb = d;

      pba->ReadBuf( "sample buffer" );
            // calls Derived::ReadBuf

      pbb->ReadBuf( "sample buffer" );
            // calls Derived::ReadBuf
```

```
//正确的写法
// 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
};
//现在一切正常了
// Example 26-2(a): Why attempt #2 works
//
```

```
Derived d;
BaseA*    pba = d;
BaseB*    pbb = d;

pba->ReadBuf( "sample buffer" );
      // calls Derived::BaseAReadBuf

pbb->ReadBuf( "sample buffer" );
      // calls Derived::BaseBReadBuf
```

ITEM27:(非)纯虚函数

```
//什么是纯虚函数？给出一个例子
// Example 27-1
//
class AbstractClass
{
   // declare a pure virtual function:
   // this class is now abstract
   virtual void f(int) = 0;
};

class StillAbstract : public AbstractClass
{
   // does not override f(int),
   // so this class is still abstract
};

class Concrete : public StillAbstract
{
public:
   // finally overrides f(int),
   // so this class is concrete
   void f(int) { /*...*/ }
};

AbstractClass a;       // error, abstract class
StillAbstract b;       // error, abstract class
Concrete      c;       // ok, concrete class

//在声明一个纯虚函数后，你为什么还会为他提供定义(函数体)？
//尽可能给出这样做的理由和场合

//纯虚析构函数
  /*
   * 所有的基类的析构函数要么应该是虚拟共有成员，要么应该是非虚拟保护成员。
   * 简单的说，这是因为，首先，要记住，你应该总是避免从实体类派生，因而假设接力
   *不是实体类那么，他就不会出于实例化自身的目的去提供一个一个共有析构函数，这
   *样就只剩下 两个选择:
   * (1) 要么，你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
   *          拟共有成员。
   * (2)要么，你不需要这一功能，这种情况下，析构函数应该是非虚拟公有成员--之所以
   *          是保护成员，为了防止滥用。
   * 如果一个类应该是抽象类，但是他没有其他任何纯虚函数，而有一个共有析构函数，
   *那么将这个析构函数生命为纯虚函数.
```

```
*/
// 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的调用基类的析构函数，
        //所以析构函数还是得定义，即使为空
        // Example 27-2(a), continued
        //
        // file b.cpp
        //

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

        //如果不提供这个定义，你还是可以让其他类从 B 派生，
        //但那些派生类不能实例化，从而没有什么特别用处

        //设计准则:基类的析构函数要么是虚拟共有成员要么是非虚拟保护成员

//2.明确的使用缺省行为
//如果派生类没有改写某个普通的虚函数，他就会默认的继承基类中的行为
//如果想提供一个默认行为但是又不想让派生类这么无声无息的继承，你可
//以声明一个并且依然提供缺省行为实现，这样派生类的设计者如果想要使
//用它，就必须主动对他进行调用.
// 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
//有时候我们需要想派生类提供部分行为，同时这个派生类还必须
//保持完整，这是一种很有价值的应用，其设计思想是:在派生类中
//将基类实现做为派生类实现的一部分来执行

// 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 ...
   }
};
```

ITEM28:受控的多态

```
//看下面的代码
class Base
{
public:
   virtual void VirtFunc();
   // ...
};

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

void SomeFunc( const Base& );
//另外还有两个函数 f1()和 f2()，我们的目标是，允许 f1()在接受 Base 对象的地方
//多态使用 Derived 对象，但防止其他所有函数包括(f2())这样做
void f1()
{
   Derived d;
   SomeFunc( d ); // works, OK
}
void f2()
{
   Derived d;
   SomeFunc( d ); // we want to prevent this
}
//演示如何达到这一效果

//我们之所以能够在接受 Base 对象的地方多态的使用 Derived 对象，原因在于 Derived
//从 Base 共有继承。相反如果 Derived 从 Base 私有继承，那么几乎没有代码可以多态的
//将 Derived 做为 Base 使用。之所以说几乎，原因在于:如果源码可以访问 Derived 的私
//有成员，他还是可以访问 Derived 的私有基类，因而可以动态的将 Derived 代替为 Base
//使用，正常来说只有 Derived 的成员函数具有访问权。然后通过 C++的友元特性，我们
//可以将类似的访问权扩充到其他外部代码中。


//想要得到题目中的效果
class Derived : private Base
      {
      public:
         void VirtFunc();
```

```
    // ...
    friend void f1();
};
```

ITEM29:使用 auto_ptr

```
//1.使用 auto_ptr 的常见错误，下面代码存在什么问题？
template<typename T>
void f( size_t n ) {
   auto_ptr<T> p1( new T );
   auto_ptr<T> p2( new T[n] );

   // ... more processing ...
}
//p2 的问题在于 auot_ptr 只是用来包含单个对象，所以对于自己拥有的指针。auto_ptr
//总是会调用 delete 而不是 delete[] ，所以使用普通的 delete，p1 会被正确的清除，p2
//则不会。

//如果使用错误的 delete，产生的实际后果取决于你的编译器。最好结果是内存泄漏，
//更常见的是内存被破坏，然后程序崩溃。下面的代码你可以体验一下效果!!!!
#include <iostream>
#include <memory>
#include <string>
using namespace std;
int c = 0;

class X {
public:
   X() : s( "1234567890" ) { ++c; }
  ~X()                                { --c; }
      string s;
};

template<typename T>
void f( size_t n )
{
   {
      auto_ptr<T> p1( new T );
      auto_ptr<T> p2( new T[n] );
   }
   cout << c << " ";      // report # of X objects
}                                  // that currently exist

int main()
{
   while( true )
   {
      f<X>(100);
   }
```

```
}

//零长度的数组是完全合法的  new T[0]  与 new T[n]有一样的行为特征
//auto_ptr 不能拥有零长度数组


//2.如何解决这一问题？尽可能提供更多的方案，包括运用 Adapter 模式，替换
//有问题结构，替换 auto_ptr 等

//方案(1)  打造自己的 auto_ptr
//          方案(1)(a)    通过从 auto_ptr 派生
            /*
             * 优点几乎没有
             * 缺点举不胜举
             */
//          方案(1)(b)    通过复制 auto_ptr 的代码
            /*
             * 去除程序库中的 auot_ptr 代码。将其中的 delete 改为 delete[]
             * 有点: 易于实现。没有空间或时间上的显著开销
             * 缺点: 难以维护
             */


//方案 2:运用 Adapter 模式

    //以下是设计思路  我们不这么写
    auto_ptr<T> p2( new T[n] );
    //而是这么写
    auto_ptr< ArrDelAdapter<T> >
    p2( new ArrDelAdapter<T>(new T[n]) );
  //其中，ArrDelAdapter 具有一个参数为 T 指针的构造函数，
  //在析构函数中，调用 delete[]
  template<typename T>
        class ArrDelAdapter {
        public:
          ArrDelAdapter( T* p ) : p_(p) { }
         ~ArrDelAdapter() { delete[] p_; }
          // operators like "->" "T*" and other helpers
        private:
          T* p_;
        };
  //既然只有一个 ArrDelAdapter<T>对象，~auto_ptr()中的单个对象形式的 delete 就不
  //会有问题因为~ArrDelAdapter<T>对数组正确调用 delete[]
```

```
        //有点:易于实现
        //缺点:可读性性差，难以使用，带来空间上的开销

//方案 3: 用手工编写的异常处理逻辑取代 auto_ptr
      //我们手工的为 p2 数组剥掉 auto_ptr 这层外衣，手工编写自己的异常处理逻辑
      //我们不这么写
        auto_ptr<T> p2( new T[n] );
        //
        // ... more processing ...
        //
        //我们这么写
        T* p2( new T[n] );
        try {
          //
          // ... more processing
          //
        }
        delete[] p2;
        //有点:易于使用，没有空间上的开销
        //缺点:难以实现，缺乏健壮性，可读性差

        //方案 4:用 vector 数组替代
        //我们不这么写
        auto_ptr<T> p2( new T[n] );
        //我们这么写
        vector<T> p2( n );
        //有点: 易于实现，可读性好，程序健壮性提高，没有空间和时间上的开销
        //缺点: 语法改变，使用性改变

        //设计准则:尽量使用 vector，不要使用内建的指针
```

ITEM30:智能指针成员之一:auto_ptr 存在的问题

```
//请看下面的类
// Example 30-1
//
class X1
{
   // ...
private:
   Y* y_;
};
//如果 X1 对象拥有所指向的 Y 对象，那么 X1 的设计者为什么不能
//使用编译器自动的生成析构函数，拷贝构造函数和拷贝赋值函数

//简单的说，如果 X1 的拥有权指向的 Y，编译器生成上述函数不会提供正确的功能
//首先，如果 X1 拥有 Y，X1 就得有某个函数，可能是构造函数，来创建 Y 对象，
//还得有另外一个函数删除它，可能是 X1::X1()
// Example 30-1(a): Ownership semantics.
//
{
   X1 a; // 分配新的 Y 对象，并指向他

   // ...
}// 当 a 走出生存空间并被摧毁
   // 时，他会删除指向的 Y

   //那么，如果使用以成员为单位的缺省拷贝构造函数，就会导致多个 X1 对象指向
   //同一个 Y 对象，将会带来奇怪的后果。如修改一个 X1 对象会同时修改另一个对象
   //的状态
   // Example 30-1(b): Sharing, and double delete.
      //
      {
         X1 a;     // 分配一个新的 Y 对象并指向他

         X1 b( a ); // b 和 a 指向同一个 Y 对象

         // 操作 a 和 b 会修改同一个 Y 对象
         // the same Y object ...

      }// 当 b 走出生存空间，会删除指向 Y 的对象...a 也会这样做

      //如果使用以成员为单位的缺省拷贝赋值，也会导致多个 X1
      //对象指向同一个 Y 对象，从而，同样状态共享和两次 delete 问题
      //更有甚者，当一些对象永远没有被删除时，他还会造成资源泄漏
      // Example 30-1(c): Sharing, double delete, plus leak.
```

```
//
{
  X1 a;   // 分配一个新 Y 对象指向它

  X1 b;   // 分配一个新 Y 对象指向它

  b = a; // b 现在和 a 一样，指向同一个 Y 对象
        // 但是没有任何对象指向 b 创建的 Y 对象

  // 操作 a 和 b 会修改同一个 Y 对象


} // 当 b 走出生存空间，会删除指向 Y 的对象...a 也会这样做

  //b 分配的 Y 对象永远没有删除

//2.下面的做法有什么优点和缺点
// Example 30-2
//
class X2
{
  // ...
private:
  auto_ptr<Y> y_;
};
//这会带来一定的好处但没有完全解决问题，自动生成的拷贝构造函数和
//拷贝赋值函数，还是会做错误的事情，只不过早做不同的错事

//首先，如果 X2 拥有用户自定义的构造函数，这些构造函数具有异常安全性会更
//容易，因为构造函数抛出异常，auto_ptr 会自动的清楚工作。但是在 auto_ptr
//对象获得 Y 对象的拥有权之前，X2 的设计者还是得分配自己的 Y 对象并通过一个
//普通的指针保存

//第二，现在，编译器自动生成的析构函数的确会做正确的事情。当 X2 对象走出
//生存空间并被摧毁时，auto_ptr<Y>的析构函数会删除它拥有的 Y 对象，自动执行
//清除工作但是有一个限制，如果你依赖自动生成的析构函数，那么在使用了 X2 的
//每一个编译单元中，这个析构函数都得提供定义。这意味着，使用了 X2 的任何
//人，Y 的定义都必须可见。
// Example 30-2(a): Y must be defined.
//
{
  X2 a; //  分配新的 Y 对象指向他

  // ...
```

```
}// 当 a 走出生存空间被摧毁时，删除指向的 Y
 //只有 Y 存在完整的定义，这才会发生

//第三，至于自动生成的拷贝构造函数，不在有两次 delete 问题
//但是带来了另外一个问题--数据偷窃。被创建的 X2 对象会偷走被拷贝的 X2
//对象中的 Y 对象，包括 Y 的所有信息
// Example 30-2(b): Grand theft pointer.
//
{
  X2 a; // 分配新的 Y 对象指向他

  X2 b( a ); // b 偷走了 a 中的 Y 对象，是 a 中的 y_成员是一个 null auto_ptr


  // if a attempts to use its y_ member, it won't
  // work; if you're lucky, the problem will manifest
  // as an immediate crash, otherwise it will likely
  // manifest as a difficult-to-diagnose intermittent
  // failure
}
```

ITEM31:智能指针成员之二:设计 valuePtr

```
//写一个合适的 ValuePtr 模版，他可以向下面这样使用
// Example 31-1
//
class X
{
   // ...
private:
   ValuePtr<Y> y_;
};
//我们将考虑三种情况，在所有三种情况中，构造函数的好处还是得保证:消除工作自动
//运行，X::X()的设计者只需要做少量的工作就可以保证异常安全性，避免构造函数失败
//造成的内存泄漏，同样在所有的三种情况中，对于析构函数的限制依然存在，要么 Y
//的完整定义必须和 X 如影随形，要么必须显示的提供 X 的析构函数，即使函数体空

//分别适应以下三种特定场合
//(a) 不允许对 ValuePtr 进行拷贝和赋值，这是在没什么好处
// Example 31-2(a): Simple case: ValuePtr without
// copying or assignment.
//
template<typename T>
class ValuePtr
{
public:
   explicit ValuePtr( T* p = 0 ) : p_( p ) { }

   ~ValuePtr() { delete p_; }
//当然还得有什么途径来访问指针，所以就像 auto_ptr 那样，提供下面的代码
T& operator*() const { return *p_; }

T* operator->() const { return p_; }
//不需要 reset 和 release 两个函数
void Swap( ValuePtr& other ) { swap( p_, other.p_ ); }
//将构造函数声明为 explicit 是一种好的做法，他避免隐式转换，
//对于这种转换，ValuePtr 的用户是绝不需要的
private:
   T* p_;

   // no copying
   ValuePtr( const ValuePtr& );
   ValuePtr& operator=( const ValuePtr& );
};
```

```
//(b)允许对 ValuePtr 进行拷贝和赋值，且具有这样的语义，在创建其 Y 对象的拷贝时
//使用的是 Y 的拷贝构造函数

//下面代码能满足你的需要，但是不具有通用性，但为拷贝构造和拷贝赋值提供了定义
// Example 31-2(b): ValuePtr with copying and
// assignment, take 1.
//
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_ ); }

   //--- new code begin ------------------------------
   ValuePtr( const ValuePtr& other )
      : p_( other.p_ ? new T( *other.p_ ) : 0 ) { }


   ValuePtr& operator=( const ValuePtr& other )
   {
      ValuePtr temp( other );
      Swap( temp );
      return *this;
   }
   //--- new code end --------------------------------

private:
   T* p_;
};
//我们允许将来可以在不同类型的 ValuePtr 之间进行拷贝和赋值，这会对上面的代码带
//来哪些影响？也就是说我们希望 ，如果 X 可以转换为 Y，就可以将 ValuePtr<X>拷贝
//和赋值给 ValuePtr<Y>
//答案是这种影响可以做到最小。我们可以复制出拷贝构造函数和拷贝赋值函数的
//模版化版本，这只需要在他们前面加上 template<typename U>,并且取一个类型为
//ValuePtr<U>&的参数
```

```
// Example 31-2(c): ValuePtr with copying and
// assignment, take 2.
//
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_( other.p_ ? new T( *other.p_ ) : 0 ) { }

   ValuePtr& operator=( const ValuePtr& other )
   {
      ValuePtr temp( other );
      Swap( temp );
      return *this;
   }

   //--- new code begin ------------------------------
   template<typename U>
   ValuePtr( const ValuePtr<U>& other )
      : p_( other.p_ ? new T( *other.p_ ) : 0 ) { }

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

private:
   template<typename U> friend class ValuePtr;
   //--- new code end --------------------------------
```

```
T* p_;
};
//这里我们还得写出非模版形式的拷贝和赋值函数，这样做是为了禁止自动生成的版本。

//这里还有一个问题，无论模版形式或非模版形式的拷贝和赋值函数，在我们执行切割
//操作的情况下，源对象 other 还是可以拥有一个指向派生类型的指针，下面是例子
class A {};
class B : public A {};
class C : public B {};

ValuePtr<A> a1( new B );
ValuePtr<B> b1( new C );

// 调用拷贝构造函数 切割
ValuePtr<A> a2( a1 );

// 调用模版化构造函数 切割
ValuePtr<A> a3( b1 );

// 调用拷贝赋值函数 切割
a2 = a1;

// 调用模版化赋值函数 切割
a3 = b1;

//31-2(b)和 31-2(c) 都是正确的 看你权衡，避免设计过度


//3.允许对 ValuePtr 进行拷贝和赋值，且具有这样的语义:如果 Y 提供一个 virtual Y::Clone()
//函数，在创建 Y 对象的拷贝时，将使用这个函数，否则，如果 Y 没有提供这样一个函
//数，将使用 Y 的拷贝构造函数
// Example 31-2(d): 允许拷贝赋值的 ValuePtr，对 Example 31-2(c)
// 做了少量的修改
//
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_ ) ) { } // changed

    ValuePtr& operator=( const ValuePtr& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }

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

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

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

    template<typename U> friend class ValuePtr;

    T* p_;
};


// Example 31-2(e): 允许拷贝和赋值 ValuePtr
//  提供基于 traits 的完整定制功能
//
//
```

```
//--- new code begin --------------------------------
template<typename T>
class VPTraits
{
static T* Clone( const T* p ) { return new T( *p ); }
};
//--- new code end ----------------------------------

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_;
};

//应用实例
// Example 31-3: Sample usage of ValuePtr.
//
class X
{
public:
   X() : y_( new Y(/*...*/) ) { }

   ~X() { }

   X( const X& other ) : y_( new Y(*(other.y_) ) ) { }

   void Swap( X& other ) { y_.Swap( other.y_ ); }

   X& operator=( const X& other )
   {
      X temp( other );
      Swap( temp );
      return *this;
   }

private:
   ValuePtr<Y> y_;
};
//设计准则:一般情况下尽量提高设计的通用性，但应该避免设计过度
```

ITEM32:递归声明

//1.什么是函数指针？如何使用他？

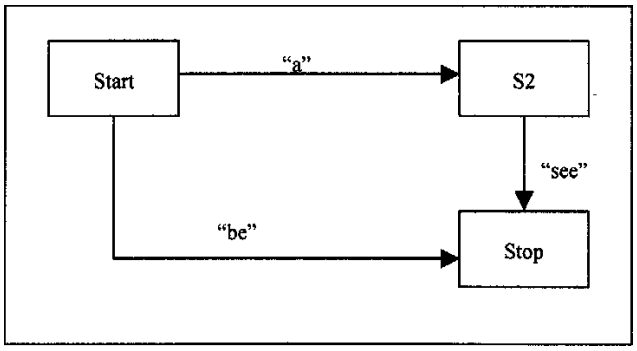
//函数指针是可以让你动态的指向具有某种原型的函数，例如:

```
// Example 32-1
//

// Create a typedef called FPDoubleInt for a function
// signature that takes a double and returns an int.
//
typedef int (*FPDoubleInt)( double );

// Use it.
//
int f( double ) { /* ... */ }
int g( double ) { /* ... */ }
int h( double ) { /* ... */ }

FPDoubleInt fp;
fp = f;
fp( 1.1 );        // calls f()
fp = g;
fp( 2.2 );        // calls g()
fp = h;
fp( 3.14 );       // calls h()
```

//2.假设可以写出这么一个函数，他能返回指向自身的指针。这个函数同样可以返
//回一个指针，指向任何一个和他具有相同原型的函数，这种功能有什么用？

//为了实现一部状态机，有时候将每一个状态写成一个函数就够了
//所有状态函数有相同的原型，他们都返回一个指针，指向下一个将要调用得
//函数

//此处应该画出状态机图



```
//下面是一个极度简化的代码片段，演示上面的设计思想
// Example 32-2
//
StatePtr Start( const string& input );
StatePtr S2     ( const string& input );
StatePtr Stop ( const string& input );
StatePtr Error( const string& input ); // error state
StatePtr Start( const string& input )
{
   if( input == "a" )
   {
      return S2;
   }
   else if( input == "be" )
   {
      return Stop;
   }
   else
   {
      return Error;
   }
}
//3.有可能写出一个函数 f()，使其返回自身的指针吗？在下面这种很自然的使用下
//他应该很有用
// Example 32-3
//
// FuncPtr is a typedef for a pointer to a
// function with the same signature as f()
//
FuncPtr p = f();       // executes f()
(*p)();                    // executes f()
//如果有可能，请演示如何实现，如果不可能，说明为什么？

//是的，有可能实现，但是方法不是那么显而易见

// Example 32-3(a): 天真的尝试 不正确
//
typedef FuncPtr (*FuncPtr)(); // error

// Example 32-3(b): 非标准且不具有移植性
//这并不是一个解决方案，不满足问题要求，而且很危险
//
typedef void* (*FuncPtr)();
```

```
void* f() { return (void*)f; }    // cast to void*
FuncPtr p = (FuncPtr)(f());          // cast from void*
p();

// Example 32-3(c): 符合标准且具有可移植性
//  但是是有害的
//
typedef void (*VoidFuncPtr)();
typedef VoidFuncPtr (*FuncPtr)();

VoidFuncPtr f() { return (VoidFuncPtr)f; }
                                        // cast to VoidFuncPtr
FuncPtr p = (FuncPtr)f(); // cast from VoidFuncPtr
p();

//下面是一个正确可移植性的方案
// 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_;
};

//可以很自然的声明，定义和使用 f()

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

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

ITEM33:模拟嵌套函数

```
//1.什么是嵌套类?他有什么用处?
//嵌套类是一个包含在另一个类中的类，例如
// Example 33-1: Nested class
//
class OuterClass
{
/*...public, protected, or private...*/:
   class NestedClass
   {
      // ...
   };

   // ...
};
//在组织代码和控制访问访问权限和从属关系方面，嵌套类有很多用处。
//和类中的其它部分一样，嵌套类也遵守访问规则。上面 NestedClass
//被声明为 public，那么任何外部代码可以称之为 OuterClass::NestedClass，
//通常嵌套类是私有实现细节。因而会被声明为 private。如果 NestedClass
//为 private，那么只有 OuterClass 的成员和友元可以使用 NestedClass

//什么是局部类？有什么用处？
//局部类是一个定义在一个函数范围内的类--任何函数，无论是成员函数
//还是自由函数，例如
// Example 33-2: Local class
//
int f()
{
   class LocalClass
   {
      // ...
   };

   // ...
};
//只有 f()的内部代码知道 LocalClass 并能够使用它，当 LocalClass
//是 f()的内部实现细节，因而永远不应该公开给其他代码的时候，他就显示出价值

//在使用非局部类大多数的地方都可以使用局部类，但是局部类或未命名的类不能作为
//模版参数使用。

template<class T>
class X { /* ... */ };
void f()
```

```
{
   struct S { /* ... */ };
   X<S> x3;    // error: local type used as
                       //    template-argument
      X<S*> x4; // error: pointer to local type
                       //    used as template-argument
   }

       //3.C++不支持嵌套函数，也就说我们不能写出下面的代码
      // Example 33-3
      //
      int f( int i )
      {
         int j = i*2;

         int g( int k )    // not valid C++
         {
            return j+k;
         }

         j += 4;

         return g( 3 );
      }
      //要解决上面的难题，大多数人会从下面入手
      // Example 33-3(a): Na?ve "local function object"
      // approach (doesn't work)
      //
      int f( int i )
      {
         int j = i*2;

         class g_
         {
         public:
            int operator()( int k )
            {
               return j+k;      // error: j isn't accessible
            }
         } g;

         j += 4;

         return g( 3 );
```

```
}
//上面代码的缺陷是，局部对象不能访问外围函数的变量

//进一步尝试，将函数中所有变量的指针或引用提供给这个局部类
// Example 33-3(b): Na?ve "local function object plus
// references to variables" approach (complex,
// fragile)
//
int f( int i )
{
   int j = i*2;

   class g_
   {
   public:
      g_( int& j ) : j_( j ) { }

      int operator()( int k )
      {
         return j_+k;    // access j via a reference
      }

   private:
      int& j_;
   } g( j );

   j += 4;

   return g( 3 );
}
//这个方案可行，但只是勉强可行，这个方案脆弱，难以扩充
//准确的说，我们只能将它看作一种非常手段。例如要增加一个变量
//我们将要做四个改动
//(1) 增加这个变量
//(2) 为 g_增加一个相应的私有引用成员
//(3) 为 g_增加一个相应的构造函数参数
//(4) 为 g::g_()增加一个相应的构造函数参数

//下面是一份略有改善的方案
// Example 33-3(c): A better solution
//
int f( int i )
{
   class g_
```

```
    {
    public:
        int j;

        int operator()( int k )
        {
            return j+k;
        }
    } g;

    g.j = i*2;
    g.j += 4;
    return g( 3 );
}

//快接近正确的方案
// Example 33-3(d): Nearly there!
//
int f( int i )
{
    // Define a local class that wraps all
    // local data and functions.
    //
    class Local_
    {
    public:
        int j;

        // All local functions go here:
        //
        int g( int k )
        {
            return j+k;
        }
        void x() { /* ... */ }
        void y() { /* ... */ }
        void z() { /* ... */ }
    } local;

    local.j = i*2;
    local.j += 4;

    local.x();
    local.y();
```

```
    local.z();

    return local.g( 3 );
}

//一个完整的极具有扩充性的方案
// Example 33-3(e): A complete and nicely
// extensible solution
//
class f
{
    int    retval; // f's "return value"
    int    j;
    int    g( int k ) { return j + k; };
    void x() { /* ... */ }
    void y() { /* ... */ }
    void z() { /* ... */ }

public:
    f( int i )      // original function, now a constructor
        : j( i*2 )
    {
        j += 4;
        x();
        y();
        z();
        retval = g( 3 );
    }
    operator int() const // returning the result
    {
        return retval;
    }
};

//注意这个方案可以很容易扩充为成员函数。假设 f()不是自由函数
//而是成员函数，我们想在 f()中写一个嵌套函数 g(),如下
// Example 33-4:  这不是合法的 C++代码，但他演示
//                了我们的需要，一个局部函数存在
//                于一个成员函数之中
//
class C
{
    int data_;
```

```
public:
   int f( int i )
   {
      // a hypothetical nested function
      int g( int i ) { return data_ + i; }

      return g( data_ + i*2 );
   }
};

//要表达这样的关系，我们可以将 f()变成一个类。向 33-3(e)那样
//只不过，33-3(e)的那个类实在全局空间，而现在他是一个嵌套类
//并需要通过辅助函数来访问
// Example 33-4(a):完整且极具扩充性
//                  的解决方案，
//                  现在用于成员函数
class C
{
   int data_;
   friend class C_f;
public:
   int f( int i );
};

class C_f
{
   C*    self;
   int retval;
   int g( int i ) { return self->data_ + i; }

public:
   C_f( C* c, int i ) : self( c )
   {
      retval = g( self->data_ + i*2 );
   }

   operator int() const { return retval; }
};

int C::f( int i ) { return C_f( this, i ); }

//设计原则：力求清晰，避免复杂设计，避免招致困惑
```

ITEM34:预处理宏

```
//我们使用宏的理由

//1.守护头文件

//为了防止头文件多次包含，这是一种技巧
#ifndef MYPROG_X_H
#define MYPROG_X_H
// ... the rest of the header file x.h goes here...
#endif

//2.使用预处理特性
//在诊断代码中，插入行号或编译时间这类信息通常很有用。要做到这一点，
//一个简单的方法是使用预定义标准宏，如__FILE__, __LINE__, __DATE__，和__TIME__.
//基于相同原因，以及其他原因，使用 stringizing(字符串化)和 token-pasting(标记合并)预
//处理运算符也很有用。

//3.在编译时期选择代码
  //A  调试代码
    //在编译你的系统时，有时候你想使用某些额外代码，但有时候你又不想这样做
     void f()
      {//这段实际上是两段不同的代码
            #ifdef MY_DEBUG
               cerr << "some trace logging" << endl;
            #endif
               // ... the rest of f() goes here...
       }
            //用条件表达式代替这个#define 会更好
     void f()
     {
        if( MY_DEBUG )
        {
           cerr << "some trace logging" << endl;
        }
        // ... the rest of f() goes here...
     }
     //B  特定平台代码
     //通常在处理针对特定平台的代码时，最好运用 factory 模式，采用这种方法，
     //代码的组合会更合理运行时期会更具有灵活性。但有时，由于存在的差异太少，
     //你很难构造一个合理的 factory，这时候预处理是一种切换可选代码的方法

     //C  不同的数据表示方式
     //一个常见的例子是:对于一个模块所定义的一组错误代码,外部用户看到的应该是
     //一个简单的 enum,并带有解释。但在模块内部，他们应该被存在一个 map 中，便
```

```
//于查找
   // For outsiders
   //
   enum Error
   {
      ERR_OK = 0,                    // No error
      ERR_INVALID_PARAM = 1, // <description>
      ...
   };

   // For the module's internal use
   //
   map<Error,const char*> lookup;
   lookup.insert( make_pair( ERR_OK,
                                        (const char*)"No error" ) );
   lookup.insert( make_pair( ERR_INVALID_PARAM,
                                        (const char*)"<description>" ) );
   ...
   //我们想同时拥有两种表达方式，但不希望将实际信息定义两次，有了宏这一魔法
   //我们就可以像下面这样，简单的写一个错误列表，编译时期创建相应的数据结构

   ERR_ENTRY( ERR_OK,                    0, "No error" ),
   ERR_ENTRY( ERR_INVALID_PARAM, 1, "<description>" ),
   ...


   //设计准则:处理以下情况外，避免使用预处理宏
      //1.守护头文件
      //2.条件编译，以获取可移植性，或在.cpp 文件中进行测试
      //3.用#pragma 禁止无伤大雅的警告，但是这种#pragma 总得
      //包含一个为了获得可移植性而提供的条件编译之中，防止编
      //译器不认识他们而发出警告
```

ITEM35:宏定义

```
//1.示范如何写一个简单的预处理宏 max(),这个宏有两个参数，并通过普通的<元算
//他比较出其中的较大值，在写这样一个宏，一般会有哪写易犯错误。

//1.不要忘记为参数加上括号
// Example 35-1(a): Paren pitfall #1: arguments
//
#define max(a,b) a < b ? b : a
//我们写
max( i += 3, j )
//展开后
i += 3 < j ? j : i += 3
//上面代码的实际运算顺序为
i += ((3 < j) ? j : i += 3)

//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)

//我们写
k = max( i, j ) + 42;
//展开后
k = (i) < (j) ? (j) : (i) + 42;
//上面代码的实际运算顺序为
k = (((i) < (j)) ? (j) : ((i) + 42));

//3.当心多余参数运算
// Example 35-1(c): Multiple argument evaluation
//
#define max(a,b) ((a) < (b) ? (b) : (a))
//我们写
max( ++i, j )
//实际为
((++i) < (j) ? (j) : (++i))

//我们写
max( f(), pi )
//实际为
((f()) < (pi) ? (pi) : (f()))

//4.名字冲突
// Example 35-1(d): Name tromping
//
```

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

#include <algorithm> // oops!
//问题在于头文件中有下面代码的东西时
template<typename T> const T&
max(const T& a, const T& b);
//于是宏就将源代码替换的一团糟
template<typename T> const T&
((const T& a) < (const T& b) ? (const T& b) : (const T& a));
//5.宏不能递归
//6.宏没有地址
//7.宏有碍测试
```

ITEM36:初始化

```
//1.直接初始化和拷贝初始化有何不同？

//直接初始化
U u;
T t1(u); // calls T::T( U& ) or similar
//拷贝初始化
T t2 = t1;    // same type: calls T::T( T& ) or similar
T t3 = u;     // different type: calls T::T( T(u) )
              //    or T::T( u.operator T() ) or similar

//设计准则:变量初始化尽量采用 T t(u),不要采用 T t=u;

//2.下列例子中哪些使用直接初始化，哪些使用拷贝初始化？
class T : public S
  {
  public:
    T() : S(1),                  // 基类初始化
          x(2) {}                // 成员初始化
    X x;
  };

  T f( T t )                     // 传递函数参数
  {
    return t;                    // 返回值
  }

  S s;
  T t;
  S& r = t;

  reinterpret_cast<S&>(t);   // 执行 reinterpret_cast
  static_cast<S>(t);         // 执行 static_cast
  dynamic_cast<T&>(r);// 执行 dynamic_cast
  const_cast<const T&>(t);   // 执行 const_cast

  try
  {
    throw T();                   // 抛出异常
  }
  catch( T t )                   // 处理异常
  {
  }
```

```
f( T(s) );                    // functional-notation type conversion
S a[3] = { 1, 2, 3 };         // brace-enclosed initializers
S* p = new S(4);              // new expression


//上面问题的答案
class T : public S
{
public:
   T() : S(1),                // base initialization
         x(2) {}              // member initialization
   X x;
};

//基类和成员初始化采用直接初始化

T f( T t )                    // passing a function argument
{
   return t;                  // returning a value
}

//值得传递和返回都是采用拷贝初始化

S s;
T t;
S& r = t;

reinterpret_cast<S&>(t);   // performing a reinterpret_cast
dynamic_cast<T&>(r);       // performing a dynamic_cast
const_cast<const T&>(t);   // performing a const_cast

//圈套: 这些地方完全没有涉及到新对象初始化，只是创建了引用。

static_cast<S>(t);            // performing a static_cast

//A static_cast uses direct initialization.

try
{
   throw T();                 // throwing an exception
}
catch( T t )                  // handling an exception
{
}
```

//异常对象的抛出和捕获使用拷贝初始化
f(T(s));　　//函数形式的类型转换，直接初始化

S a[3] = { 1, 2, 3 };　　//大括号的初始化语句，拷贝初始化

//Brace-enclosed initializers use copy initialization.

S* p = new S(4);　　// new 表达式 直接初始化

ITEM37:前置声明

//1.前置声明是非常有用的工具。但是在这个例子中，它没有像程序员预计的那样工作，做标记的

//那两行代码为什么是错误的？

```
// file f.h
//
class ostream;    // error
class string;     // error
string f( const ostream& );
```

//错误在于，你不能以这种方式前置声明 ostream 和 string,
//因为他们不是类，他们是模版 typedef

//2.不包含任何其他文件，为上面的 ostream 和 string 写出正确的前置声明

//答案是不可能写出的。实际情况是:不存在某种标准且具有可移植性的方法，
//可以做到不包含另一个文件却能前置声明 ostream,根本没有一种标准且具有
//可移植性的方法可以前置声明 string

//我们不能对 namespace std 写出我们自己的声明

//你能做到最好的只能像下面这样

```
#include <iosfwd>
#include <string>
```

//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，
//在不需要流的完整定义时尽量只包含#include<iosfwd>

ITEM38:typedef

//1.为什么使用 typedef,尽可能的列举使用场合和理由。

```
/*
 * 1.便于打字，名字越短，打字越容易。
 * 2.typedef 便于代码阅读
 *     int ( *t(int) )( int* );
 *     用 type 的简化后为:
 *     typedef int (*Func)( int* );
 *     Func t( int );
 *     这是一个函数声明，此函数的名称为 t，输入参数为 int,返回值为一个函数
 *     指针，指向一个输入参数为 int，返回值为 int 的函数
 * 3.便于交流，typedef 有利于表达程序的意图
 *     int x;
 *     int y;
 *     y = x * 3;      // might be okay -- who knows?
 *
 *      比较下面的代码
 *
 *      typedef int Inches;
 *      typedef int Dollars;
 *      Inches    x;
 *      Dollars y;
 *      y = x * 3;      // hmmm...?
 * 4.可移植性，对于那些和平台相关的名称或不具有可移植性的名称使用 typedef
 *      #if defined USING_COMPILER_A
 *          typedef __int32 Int32;
 *          typedef __int64 Int64;
 *      #elif defined USING_COMPILER_B
 *          typedef int           Int32;
 *          typedef long long Int64;
 *      #endif
 * /
```

//2.在用到了标准容器的代码中，使用 typedef 为什么是个好主意？

5.灵活性，较之整个代码修改名称，仅在一处修改 typedef 更容易

```
/*
 *     void f( vector<Customer>& custs )
 *     {
 *          vector<Customer>::iterator i = custs.begin();
 *           ...
 *      }
 *     与下面代码相比较
 *      typedef vector<Customer> Customers;
```

```
*       typedef Customers::iterator CustIter;
*
*        ...
*
*        void f( Customers& custs )
*        {
*            CustIter i = custs.begin();
*            ...
*         }
*/
```

ITEM39:名字空间。之一:using 声明和 using 指令

```
//什么是 using 声明和 using 命令?如何使用他们?给出例子。另外
//他们中的哪一个和顺序相关?

//using 声明为实际声明在另一个名字空间中的名称创建一个本地同义词。
//为了重载和名称解析的需要，using 声明的运作方式和其他声明一样
// Example 39-1(a)
//
namespace A
{
   int f( int );
   int i;
}

using A::f;

int f( int );

int main()
{
   f( 1 );        //  歧义: A::f() or ::f()?

   int i;
   using A::i; //  错误，重复声明  (像是
                  //    写了两次 int i 一样)
}
//只有已经出现过声明的名称，using 声明才能将其引入
// Example 39-1(b)
//
namespace A
{
   class X {};
   int Y();
   int f( double );
}
using A::X;
using A::Y; //  函数  A::Y(),  非类  A::Y
using A::f; //    A::f(double),  非  A::f(int)

namespace A
{
   class Y {};
   int f( int );
```

```
}

int main()
{
   X x;          // 正确, X 是 A::X 同义词

   Y y;          // 错误, A::Y 不可见 因为
                 //   using 声明出现在他想要
                 //   的实际声明之前

   f( 1 );     // 这里偷偷使用了一个隐式转换，
                 //   他调用的是 A::f(double),而不是
                 //   A::f(int), 因为在 using 声明之前
                 //   只有头一个 A::f() 声明出现过
                  //   seen at the point of the using-
                  //   declaration for the name A::f
}

//有了 using 指令，另一个名字空间中的所有名称都可以使用于 using 指令所在的空间，
//和 using 声明不同，using 指令会将声明在 using 指令之前，之后所有的名称都引入进
//来。当然他在使用某个名称之前，那个名称必须出现过才行。
// Example 39-1(c)
//
namespace A
{
   class X {};
   int f( double );
}
void f()
{
   X x;          // OK, X is a synonym for A::X
   Y y;          // error, no Y has been seen yet
   f( 1 );      // OK, calls A::f(double) with parameter promotion
}

using namespace A;

namespace A
{
   class Y {};
   int f( int );
}

int main()
```

```
{
   X x;          // OK, X is a synonym for A::X
   Y y;          // OK, Y is a synonym for A::Y
   f( 1 );      // OK, calls A::f(int)
}
```

ITEM40:名字空间。之二:迁徙到名字空间

```
//假设你在开发一个数百万行代码的项目，其中的.h 和.cpp 文件有上千个。
//这时，项目小组要将编译器升级到最新版本，这个版本的编译器支持名字空间，标准
//库的所有构件也都放在名字空间 std 中。不幸的是这种顺应标准的做法也有副作用
//会导致现有代码不能通过编译。而你没有足够的时间去仔细分析每个文件
//该如何解决这个问题呢？

//安全高效的迁徙到名字空间
//如今标准库在于名字空间 std 中，所以在这个项目中，不带修饰
//的使用 std::名称将无法编译成功

//下列代码过去可能通过编译
// Example 40-1: This used to work
//
#include <iostream.h>    // 标准出台前的头文件

int main()
{
   cout << "hello, world" << endl;
}
//现在，要么你选择写出哪些名称存在于 std 中
// Example 40-2(a): Option A, 明确指明一切
//
#include <iostream>

int main()
{
   std::cout << "hello, world" << std::endl;
}
//要么使用 using 将需要的 std 名称引入到当前空间
// Example 40-2(b): Option B, write using-declarations
//
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   cout << "hello, world" << endl;
}
//要么简单的使用一个 using  指令，将所有 std 名称整个引入到当前空间
// Example 40-2(c): Option C, write using-directives
```

```
//
#include <iostream>

int main()
{
   using namespace std; // or this can go at file scope
   cout << "hello, world" << endl;
}
//要么，综合以上方法


//好的长期方案的设计准则，至少应该遵循一下规则
//规则 1:绝对不要在头文件中使用 using 指令，using 指令会污染名字空间，
//因为它可能引入大量的名称，其中许多名称是不必要的，不必要名称一
//旦出现，名字冲突就可能加大。
//规则 2:绝对不要在头文件中使用 using 声明。
//规则 3: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
//规则 4:在使用 C 头文件时，采用新风格的#include<cheader> 而不采用旧风格的
#include<header.h>

//一个启发的例子
// Example 40-3(a): 没有名字空间的原始代码
//

//--- file x.h ---
//
#include "y.h"    // defines Y
#include <deque.h>
#include <iosfwd.h>

ostream& operator<<( ostream&, const Y& );
Y operator+( const Y&, int );
int f( const deque<int>& );

//--- file x.cpp ---
//
#include "x.h"
#include "z.h"    // defines Z
#include <ostream.h>

ostream& operator<<( ostream& o, const Y& y )
{
   // ... uses Z in the implementation ...
   return o;
```

```
}

Y operator+( const Y& y, int i )
{
   // ... uses another operator+() in the implementation...
   return result;
}

int f( const deque<int>& d )
{
   // ...
}

//一个好的长期方案
// Example 40-3(b): A good long-term solution
//

//--- file x.h ---
//
#include "y.h"    // defines Y
#include <deque>
#include <iosfwd>

std::ostream& operator<<( std::ostream&, const Y& );
Y operator+( const Y&, int i );
int f( const std::deque<int>& );

//--- file x.cpp ---
//
#include "x.h"
#include "z.h"    // defines Z
#include <ostream>
using std::deque;      // "using" appears AFTER all #includes
using std::ostream;
using std::operator+;
// or, "using namespace std;" if that suits you
ostream& operator<<( ostream& o, const Y& y )
{
   // ... uses Z in the implementation ...
   return o;
}

Y operator+( const Y& y, int i )
{
```

```
   // ... uses another operator+() in the implementation...
   return result;
}

int f( const deque<int>& d )
{
   // ...
}

//一个不那么好的长期方案
// Example 40-3(c): Bad long-term solution
// (or, Why to never write using-declarations
// in headers, even within a namespace)
//

//--- file x.h ---
//
#include "y.h"    // defines MyProject::Y and adds
                  //    using-declarations/directives
                  //    in namespace MyProject
#include <deque>
#include <iosfwd>
namespace MyProject
{
   using std::deque;
   using std::ostream;
   // or, "using namespace std;"

ostream& operator<<( ostream&, const Y& );
Y operator+( const Y&, int );
int f( const deque<int>& );
}

//--- file x.cpp ---
//
#include "x.h"
#include "z.h"    // defines MyProject::Z and adds
                  //    using-declarations/directives
                  //    in namespace MyProject
   // error: potential future name ambiguities in
   //             z.h's declarations, depending on what
   //             using-declarations exist in headers
   //             that happen to be #included before z.h
   //             in any given module (in this case,
```

```
   //             x.h or y.h may cause potential changes
   //             in meaning)
#include <ostream>

namespace MyProject
{
   using std::operator+;

   ostream& operator<<( ostream& o, const Y& y )
   {
      // ... uses Z in the implementation ...
      return o;
   }

   Y operator+( const Y& y, int i )
   {
      // ... uses another operator+() in the implementation...
      return result;
   }

   int f( const deque<int>& d )
   {
      // ...
   }
}
//一个有效的短期方案
//移植步骤 1:在每个头文件中，为所有所需要之处添加 std::修饰符
//移植步骤 2:创建一个 myproject_last.h 的新头文件，使之包含 using
//             namespace std 指令。然后，在每一个实现文件中，在所有其他#include
//             指令之后包含(#include)myproject_last.h
//使用上面步骤后

// Example 40-3(d): Good short-term solution,
   // applying our two-step migration
   //

   //--- file x.h ---
   //
   #include "y.h"    // defines Y
   #include <deque>
   #include <iosfwd>

   std::ostream& operator<<( std::ostream&, const Y& );
      Y operator+( const Y& y, int i );
```

```
int f( const std::deque<int>& );

//--- file x.cpp ---
//
#include "x.h"
#include "z.h"    // defines Z
#include <ostream>
#include "myproject_last.h"
                          // AFTER all other #includes
ostream& operator<<( ostream& o, const Y& y )
{
   // ... uses Z in the implementation ...
   return o;
}

Y operator+( const Y& y, int i )
{
   // ... uses another operator+() in the implementation...
   return result;
}

int f( const deque<int>& d )
{
   // ...
}

//--- common file myproject_last.h ---
//
using namespace std;
//这不会累及长期方案，因为他做任何事情都不需要长期方案去撤销
//同时，和完整的长期方案相比，他更简单，需要的代码修改量更少。实际上
//想让代码在支持名字空间的编译器上工作，并且不至于以后还得回头去撤销
//已做的工作，这种方法需要的工作量最少

//最后，很可能以后某个适当的时候，你暂时不再为工程期限所迫，
//你就可以实施简单的移植策略过渡到 40-3(b)所介绍的长期策略。
//简单遵循以下步骤

//1.在 myproject_last.h 中，注释掉 using 指令
//2.重新编译工程，看看哪些地方不能通过编译，然后在每个实现文件中
//   增加正确的 using 指令或 using 声明
//3.这一步你可以不做，在每一个头文件或实现文件中，将包含 C 头文件
//   代码修改为新的<cheader>形状
```