ITEM1: Iterators

```
int main(){

    vector<int> e;
    copy(istream_iterator<Date>(cin),
            istream_iterator<Date>(),
            back_inserter(e));
    vector<Date>::iterator first=
                find(e.begin(),e.end(),"01/01/95");
    vector<Date>::iterator last=
                find(e.begin(),e.end(),"12/31/95");
    *last="12/30/95";
    //上一行可能是不合法的   last 可能是 e.end()，是一个不可提领的 iterator
    copy(first,end,ostream_iterator<Date>(cout,"\n"));
    //[first,last)可能是无效的范围，last 可能在 first 之前
    e.insert(--e.end(),TodayDate() );
    //上一行--e.end()可能是不合法的，c++不允许修改内建型别的暂时物件。
    //e.insert(e.end()-1,TodayDate() );
    //上一行还有可能是错的 因为如果 e 为空的话无论是  --e.end()还是  e.end()-1 都是错的
    copy( first,end,ostream_iterator<Date>(cout,"\n"));
    //first,last 可能是无效迭代器
    //还有可能是  vector 的可能成长  导致迭代器全部失效

}

//设计准则
//使用 iterator 时，务必注意以下 4 点
1.有效的数值:这个迭代器可提领吗?如果写成*e.end()绝对错误
2.有效的寿命:这个迭代器被使用时还有效吗?或者因为某些操作已经无效
3.有效的范围:一对迭代器是否组成一个有效的范围？是否 first 真的在 last 之后?是否两者
   指向同一个迭代器?
4.不合法的操作行为:程序式的代码是否试图修改内建型别的占时物件。想--e.end()这样
   吗？
```

ITEM2-3:不区分大小写的 string

```
//写一个不区分大小写的 string 型别

ci_string s( "AbCdE" );
// case insensitive
//
assert( s == "abcde" );
assert( s == "ABCDE" );
// still case-preserving, of course
//
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );

//具体实现

 //区分大小写的 string
 template<class charT,
        class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
 class basic_string;
     //其中 char_traits 提供了如果下的字符串比较函数
     1.eq()              相等
     2.ne()          不等
     3.lt()          小于
     4.compare()  比较字符序列
     5.find()       搜索字符序列
//整合代码
struct ci_char_traits : public char_traits<char>
                // just inherit all the other functions
                //    that we don't need to replace
{
  static bool eq( char c1, char c2 )
  {
        return toupper(c1) == toupper(c2);
  }
  static bool lt( char c1, char c2 )
  {
      return toupper(c1) <    toupper(c2);
  }
  static int compare( const char* s1,const char* s2,size_t n )
  {
      return memicmp( s1, s2, n );
  }
```

```
            // if available on your platform,
            //    otherwise you can roll your own
  static const char* find( const char* s, int n, char a )
    {
  while( n-- > 0 && toupper(*s) != toupper(a) )
  {
       ++s;
   }
    return n >= 0 ? s : 0;
 }
};

//And finally, the key that brings it all together:

 typedef basic_string<char, ci_char_traits> ci_string;

  //出现的问题
  1.//下面的代码无法通过编译
    ci_string s="abc";
    cout<<s<<endl;
    //解决办法
     cout<<s.c_str()<<endl;
  2.//下面的代码无法通过编译  ci_string  和 string 使用  + += =
    string a="aaa";
    ci_string b="bbb";
    string c=a+b;
    //解决办法是定义这些操作符或
    string c=a+b.c_str();
  3.这样从 char_traits<char>继承出 ci_char_traits 是安全的
```

ITEM4-5:具有最大可复用性的通过 Container

//为下面的定长的 vector class 实现拷贝构造操作和拷贝赋值操作，提供最大的可用性

//不能修改代码的其他部分，目的不是加入模版库。

```
template<typename T, size_t size>
class fixed_vector
{
public:
   typedef    T*    iterator;
   typedef const T* const_iterator;
   iterator      begin()      { return v_; }
   iterator      end()         { return v_+size; }
   const_iterator begin() const { return v_; }
   const_iterator end()      const { return v_+size; }

private:
   T v_[size];
};
//解决方案
template<typename T, size_t size>
class fixed_vector
{
public:
   typedef    T*       iterator;
   typedef const T* const_iterator;
   fixed_vector() { }

   template<typename O, size_t osize>
   fixed_vector( const fixed_vector<O,osize>& other )
   {
       copy( other.begin(),
             other.begin()+min(size,osize),
             begin() );
   }

   template<typename O, size_t osize>
   fixed_vector<T,size>&
   operator=( const fixed_vector<O,osize>& other )
   {
      copy( other.begin(),
             other.begin()+min(size,osize),
             begin() );
      return *this;
   }
```

```
    iterator       begin()       { return v_; }
    iterator       end()          { return v_+size; }
    const_iterator begin() const { return v_; }
    const_iterator end()   const { return v_+size; }

private:
    T v_[size];
};

//
struct X
{
    template<typename T>
    X( const T& );      // NOT copy constructor, T can't be X
                        //不会隐藏原来拷贝构造声明

    template<typename T>
    operator=( const T& );
                        // NOT copy assignment, T can't be X
                        //不会隐藏原来拷贝赋值声明
};
//因此在我们的解答中，并没有隐藏原来的拷贝构造和拷贝赋值
//我们手写的代码只是增加了代码的可适应性
fixed_vector<char,4> v;
fixed_vector<int,4>  w;
fixed_vector<int,4>  w2(w);
        // calls implicit copy constructor
fixed_vector<int,4>  w3(v);
        // calls templated conversion constructor
w = w2; // calls implicit copy assignment operator
w = v;  // calls templated assignment operator

//我们写的代码的用途
1.支持可变类型(包括继承在内)
    fixed_vector<char,4> v;
      fixed_vector<int,4>  w(v);  // templated construction
      w = v;                      // templated assignment

      class B                 { /*...*/ };
      class D : public B { /*...*/ };

      fixed_vector<D*,4> x;
      fixed_vector<B*,4> y(x);    // templated construction
      y = x;                      // templated assignment
```

2.支持可变的大小

```
fixed_vector<char,6> v;
    fixed_vector<int,4>    w(v);    // initializes using 4 values
    w = v;                                  // assigns using 4 values
    class B                  { /*...*/ };
    class D : public B { /*...*/ };
    fixed_vector<D*,16> x;
    fixed_vector<B*,42> y(x);      // initializes using 16 values
    y = x;                                  // assigns using 16 values
```

//另外的解决方案　更具有实用性

1.Alternative: The Standard Library Approach

```
    //I happen to like the syntax and usability of the preceding functions,
    //but there are still some nifty things they won‘t let you do. Consider
    //another approach that follows the style of the standard library.
```

//Copying

```
    template<class RAIter>
    fixed_vector( RAIter first, RAIter last )
    {
       copy( first,
             first+min(size,(size_t)last-first),
             begin() );
    }

    //Now when copying, instead of writing:

    fixed_vector<char,6> v;
    fixed_vector<int,4>    w(v);    // initialize using 4 values

    //we need to write:

    fixed_vector<char,6> v;
    fixed_vector<int,4>    w(v.begin(), v.end());
                                            // initialize using 4 values
```

2.Assignment

```
//Note that we can't templatize assignment to take an iterator range,
//because operator=() may take only one parameter. Instead, we can
//provide a named function:

template<class Iter>
fixed_vector<T,size>&
assign( Iter first, Iter last )
{
   copy( first,
         first+min(size,(size_t)last-first),
         begin() );
   return *this;
}

//Now when assigning, instead of writing:

w = v;                                  // assign using 4 values

//we need to write:

w.assign(v.begin(), v.end());
                                        // assign using 4 value

//assign 不是必须的，我们也可以用
copy( v.begin(), v.begin()+4, w.begin() ); //比 assign 要好

//一旦你定义任何形式的构造函数，编译器就不会在为你定义构造函数
//所以在第一种解法时，我们仍然要显示的定义拷贝构造函数



//两种解法中，第一个的容易使用，第二个的可用性更好
```

ITEM6:临时对象

```
//找出下面临时对象使用的地方
string FindAddr( list<Employee> emps, string name ) //两个参数都会产生临时对象
      {
        for( list<Employee>::iterator i = emps.begin();
             i != emps.end();
             i++ )//临时对象产生
        {
          if( *i == name )//临时对象产生
          {
            return i->addr;
          }
        }
        return "";//临时对象产生
      }

    //注意以下几点
    1.请使用 const&，而不是传值拷贝
    2.请使用++i  避免使用 i++
    3.时刻注意因为参数转换操作而产生的隐藏的临时对象，
      一个避免他好的方法是使用显示的构造函数
    4.绝对绝对不要返回绝不对象的引用

  //对于上面代码的优化
  string FindAddr( const list<Employee>& emps,
                   const string& name )
{
  list<Employee>::const_iterator end( emps.end() );
  for( list<Employee>::const_iterator i = emps.begin();
       i != end;
       ++i )
  {
    if( i->name == name )
    {
      return i->addr;
    }
  }
  return "";
}
```

ITEM7:使用标准库

```
    //对于修改后的版本若然有一些问题
string FindAddr( const list<Employee>& emps,
                 const string&    name )
{
   list<Employee>::const_iterator end( emps.end() );
   for( list<Employee>::const_iterator i = emps.begin();
        i != end;
        ++i )
   {
      if( i->name == name )
      {
         return i->addr;
      }
   }
   return "";
}

   //应该用标准库
   //Solution

        //With no other changes, simply using the standard find()
        //algorithm could have avoided two temporaries, as well as
        // the emps.end() recomputation inefficiency from the original
        //code. For the best effect to reduce temporaries, provide an
        // operator==() taking an Employee& and a name string&.

        string FindAddr( list<Employee> emps, string name )
        {
           list<Employee>::iterator i(
              find( emps.begin(), emps.end(), name )
              );
           if( i != emps.end() )
           {
              return i->addr;
           }
           return "";
        }
        //在进行一些修改

        string FindAddr( const list<Employee>& emps,
                         const string& name )
```

```
{
    list<Employee>::const_iterator i(
        find( emps.begin(), emps.end(), name )
        );
    if( i != emps.end() )
    {
        return i->addr;
    }
    return "";
}
```

ITEM8-17:异常的安全性

```
//实现如下异常--中立的 Container。要求：stack 对象的状态必须保持一致性
//即有内部异常抛出，stack 对象必须是可析构的：T 的异常可以传到调用者那里
template <class T>
class Stack{
    public:
        Stack();
        ~Stack()
        Stack(const Stack&);
        Stack& operator=(const Stack&);
        unsigned Count();//返回栈内的元素数目
        void Push(const T&);//入栈
        T pop();//出栈
    private:
        T* v_;//指向一个用于 vsize_ T 对象的足够大的空间
        unsigned vsize_;//v_区域的大小
        unsigned vused_;//v_区域实际使用

};

//以下是实现

template<class T>
Stack<T>::Stack()
  : v_(new T[10]),   // default allocation   //有内部异常抛出时，T 可以处理
    vsize_(10),
    vused_(0)            // nothing used yet
{
}

template<class T>
Stack<T>::~Stack()
{
  delete[] v_;          // this can't throw
}


template<class T>
T* NewCopy( const T*   src,
            size_t   srcsize,
            size_t   destsize )
{
  assert( destsize >= srcsize );
  T* dest = new T[destsize];
```

```
  try
  {
    copy( src, src+srcsize, dest );
  }
  catch(...)
  {
    delete[] dest; // this can't throw
    throw;         // rethrow original exception
  }
  return dest;
}

template<class T>
Stack<T>::Stack( const Stack<T>& other )
  : v_(NewCopy( other.v_,
                other.vsize_,
                other.vsize_ )),
    vsize_(other.vsize_),
    vused_(other.vused_)
{
}

template<class T>
Stack<T>&
Stack<T>::operator=( const Stack<T>& other )
{
  if( this != &other )
  {
    T* v_new = NewCopy( other.v_,
                        other.vsize_,
                        other.vsize_ );
    delete[] v_;    // this can't throw
    v_ = v_new;     // take ownership
    vsize_ = other.vsize_;
    vused_ = other.vused_;
  }
  return *this;     // safe, no copy involved
}

template<class T>
size_t Stack<T>::Count() const
{
  return vused_;    // safe, builtins don't throw
}
```

```
template<class T>
void Stack<T>::Push( const T& t )
{
   if( vused_ == vsize_ )    // grow if necessary
   {                                    // by some grow factor
      size_t vsize_new = vsize_*2+1;
      T* v_new = NewCopy( v_, vsize_, vsize_new );
      delete[] v_;    // this can't throw
      v_ = v_new;      // take ownership
      vsize_ = vsize_new;
   }
   v_[vused_] = t;
   ++vused_;
}

template<class T>
void Stack<T>::Pop( T& result )
{
   if( vused_ == 0)
   {
      throw "pop from empty stack";
   }
   else
   {
      result = v_[vused_-1];
      --vused_;
   }
}



      template<class T>
      const T& Stack<T>::Top() const
      {
         if( vused_ == 0)
         {
            throw "empty stack";
         }
         else
         {
            return v_[vused_-1];
         }
      }
```

```
    //to provide Top for const Stack objects, and:

    template<class T>
    bool Stack<T>::Empty() const
    {
       return( vused_ == 0 );
    }

  //try catch 可能会引起代码的速度变慢
  //NewCopy 使用无 try catch 的写法
template<class T>
T* NewCopy( const T* src,
                size_t      srcsize,
                size_t      destsize )
{
  destsize=max(srcsize,destsize);//basic parm check
  strucy Janitor{
     Janitor(T* p):pa(p){}
     ~Janitor()
     {
          if(uncaught_exception() delete[] pa;
          T* pa;
     }
     T* dest=new T[destsize];
     //if we go here ,the allocation/ctors were okay
     Janitor j(dest);
     copy(src,src+srcsize,destsize);
     //if we got here ,the copy was okay...otherwise ,j
     //was destroyed during stack unwinding and will handle
     //the cleanup of dest to avoid leaking memory
  };
}

//不发生异常的情况下  带 try catch 的语句的速度比上面的写法要快
```

ITEM18:代码的复杂性

//How many execution paths could there be in the following code?

```
String EvaluateSalaryAndReturnName( Employee e )
{
   if( e.Title() == "CEO" || e.Salary() > 100000 )
   {
      cout << e.First() << " " << e.Last() << " is overpaid" << endl;
   }
   return e.First() + " " + e.Last();
}
```

ITEM19:代码复杂性

```
    //考虑下列函数是异常安全的还是一场中立的
String EvaluateSalaryAndReturnName( Employee e )
{
   if( e.Title() == "CEO" || e.Salary() > 100000 )
   {
      cout << e.First() << " " << e.Last() << " is overpaid" << endl;
   }
   return e.First() + " " + e.Last();
}

  //basic guarantee 是保证可销毁且没有泄漏
  //strong guarantee 是保证可销毁且没有泄漏外还保证 commit-to-rollback

  //流可能产生不可恢复的作用，即运算符<<输出了 string 一部分后抛出异常，
  //被输出的那个部分是不能反输出的，当然流的错误状态也可能被设置。
  //本题我们暂时不考虑这个。


  //以上的代码满足 basic guarantee
  //第一次修改
  // Attempt #1: An improvement?
      //
      String EvaluateSalaryAndReturnName( Employee e )
      {
         String result = e.First() + " " + e.Last();
         if( e.Title() == "CEO" || e.Salary() > 100000 )
         {
            String message = result + " is overpaid\n";
            cout << message;
         }

         return result;
      }

   //考虑下列代码
   String theName;
   theName=EvaluateSalaryAndReturnName(bob);
   //函数采用的 by value 的方式返回，因此会有 copy assignment operator 和
   //拷贝构造函数被唤起如果有一个失败，那么副作用就会产生
   //为了避免拷贝构造，我们做一次啊修改
   // Attempt #2: Better now?
            //
```

```
        void EvaluateSalaryAndReturnName( Employee e,
                                          String&    r )
        {
           String result = e.First() + " " + e.Last();
           if( e.Title() == "CEO" || e.Salary() > 100000 )
           {
              String message = result + " is overpaid\n";
              cout << message;
           }

           r = result;
        }
//由于 r 的赋值可能失败，因此做第三次尝试
// Attempt #3: Correct (finally!).
    //
    auto_ptr<String>
    EvaluateSalaryAndReturnName( Employee e )
    {
       auto_ptr<String> result
            = new String( e.First() + " " + e.Last() );

       if( e.Title() == "CEO" || e.Salary() > 100000 )
       {
          String message = (*result) + " is overpaid\n";
          cout << message;
       }

       return result;    // rely on transfer of ownership;
                         // this can't throw
    }
    //只要打印结果不抛出异常 ，我们就能把函数结果安全的返回给调用者

  //终点
  1.要对异常安全性提供保证，经常你要放弃一部分性能为代价
  2.如果一个函数有多重副作用,那么无法成为强异常安全,解决办法就是拆分成几个函数
  3.并不是所有的函数都需要异常安全。本题中第一次尝试已经足够好用，并不需要
    第三次那样损失一部分性能。
```

ITEM20:class 技术

//修改下列代码

```
class Complex
 {
 public:
    Complex( double real, double imaginary = 0 )
       : _real(real), _imaginary(imaginary)
    {
    }
    void operator+ ( Complex other )
    {
       _real = _real + other._real;
       _imaginary = _imaginary + other._imaginary;
    }
    void operator<<( ostream os )
    {
       os << "(" << _real << "," << _imaginary << ")";
    }
    Complex operator++()
    {
       ++_real;
       return *this;
    }
    Complex operator++( int )
    {
       Complex temp = *this;
       ++_real;
       return temp;
    }
 private:
    double _real, _imaginary;
 };
```

//设计准则

1.尽量重复运用程序式的代码--特别是标准库的--而不要老想自己撰写代码，那样不但快，而且安全

2.小心隐式转换带来的隐式暂时物件。避免这东西的一个好办法就是尽可能的让构造函数成为 explicit

3.尽量的以 by const& (而非 by value)的方式来传值

4.尽量的写 a op= b 而不要写成 a=a op b 。这样不但比较清楚，而且有效率

5.如果你提供了某个运算子的标准版(如 operator+)，请同时为他提供一份 assign 版本。并以后者实现前者

6.使用一下准则来决定一个运算子应该是 member function 或应该是 non-member functions

6.1 一元运算子应该是 members

6.2 = () [] -> 必须是 members

6.3 assignment 版本的运算子 += -= /= *= 应该是 members

6.4 其他所有的元算子应该是 nonmember

7.总是在 operator<< 和 operator>>函数式传回 stream reference

8.为了一致性，请总是以前置式累加算子为本，实作出后置式的累加算子。

//修改后的代码

```
class Complex
{
public:
   explicit Complex( double real, double imaginary = 0 )
      : real_(real), imaginary_(imaginary)
   {
   }

   Complex& operator+=( const Complex& other )
   {
      real_ += other.real_;
      imaginary_ += other.imaginary_;
      return *this;
   }

   Complex& operator++()
   {
      ++real_;
      return *this;
   }

   const Complex operator++( int )
   {
      Complex temp( *this );
      ++*this;
      return temp;
   }

   ostream& Print( ostream& os ) const
   {
      return os << "(" << real_ << "," << imaginary_ << ")";
   }

   private:
      double real_, imaginary_;
   };
```

```
const Complex operator+( const Complex& lhs, const Complex& rhs )
{
   Complex ret( lhs );
   ret += rhs;
   return ret;
}

ostream& operator<<( ostream& os, const Complex& c )
{
   return c.Print(os);
}
```

ITEM21:改写虚拟函式

```
#include <iostream>
#include <complex>
using namespace std;
class Base
{
public:
   virtual void f( int );
   virtual void f( double );
   virtual void g( int i = 10 );
};
void Base::f( int )
{
   cout << "Base::f(int)" << endl;
}
void Base::f( double )
{
   cout << "Base::f(double)" << endl;
}
void Base::g( int i )
{
   cout << i << endl;
}
class Derived: public Base
{
public:
   void f( complex<double> );
   void g( int i = 20 );
};
void Derived::f( complex<double> )
{
   cout << "Derived::f(complex)" << endl;
}
void Derived::g( int i )
{
   cout << "Derived::g() " << i << endl;
}
void main()
{
   Base       b;
   Derived d;
   Base*      pb = new Derived;
   b.f(1.0);
   d.f(1.0);
```

```
    pb->f(1.0);
    b.g();
    d.g();
    pb->g();
    delete pb;
}
//void main 不是标准的写法 ， 不具有很大的移植性 最好的写成
    //int main()
    //int main(int argc,char* argv[])
//delete pb;是不安全的 因为没有虚拟析构函数
//设计准则
1.让 base class 的析构函数成为 virtual
2.当你要提供一个函数式，其名称与继承而来函数式同名时，如果你不想遮掩继承而来
  的函数式请用 using declaration 来突围
3.绝不要在改变虚拟式的过程中变改预设参数


/*
 * 重载: 一个函数式 f() ,在同一个 scope 中提供另外一个同名函数式。
 *          当 f()被呼叫，编译器根据实际情况调用
 * 重写: 一个虚拟函数式 f(),在 derived class 提供一个名字相同的函数式，参数也相同
 * 遮掩: 一个函数式 f() ， 意思是在外围 scope 的内层 scope 提供一个同名的函数。
 */

//其中 f()属于重载，重载的程序是根据静态类型(Base)决定的,而不是根据静态类型(derived)
```

ITEM22:classes 之间的关系

```
//下面的代码有需要改进的吗？
class BasicProtocol /* : possible base classes */
{
public:
   BasicProtocol();
   virtual ~BasicProtocol();//不是设计用来被其他类继承的，virtual 没有必要
   bool BasicMsgA( /*...*/ );
   bool BasicMsgB( /*...*/ );
   bool BasicMsgC( /*...*/ );
};

class Protocol1 : public BasicProtocol
{
public:
   Protocol1();
   ~Protocol1();
   bool DoMsg1( /*...*/ );
   bool DoMsg2( /*...*/ );
   bool DoMsg3( /*...*/ );
   bool DoMsg4( /*...*/ );
};

class Protocol2 : public BasicProtocol
{
public:
   Protocol2();
   ~Protocol2();
   bool DoMsg1( /*...*/ );
   bool DoMsg2( /*...*/ );
   bool DoMsg3( /*...*/ );
   bool DoMsg4( /*...*/ );
   bool DoMsg5( /*...*/ );
};
//每一个 DoMsg...() member function  都呼叫
//BasicProtocol::Basic...()执行共同的工作，
//然后 DoMsg...()另外执行传送工作。

//常见错误: 绝不要使用 public inheritance ,除非你要塑膜出真正的 IS-A  和 LIKE-A  关系。
//  所有被改写的 member functions 不能要求更多也不能承诺更少。
```

//设计准则:绝不要以 public inheritance 复用 base class 内的程式码，public
// inheritance 是为了被复用--被那些以多型方式运用 object 的程式码复用

1.BasicProtocol 没有提供任何虚拟函数函数式，这就意味着他不希望以多型的凡是被使用这对 public inheritance 是一个强烈的反对暗示

2.BasicProtocol 没有任何 protected members，这就意味着没有继承接口，这对任何继承型式 (public ,private)都是一个强烈的反对暗示

3.BasicProtocol 封装了共同的工作，但是他不像 derived class 那样执行自己的传送工作。这就意味着 BasicProtocol 物件运行起来并非像一个 LIKE-A 衍生的 protocol 物件，也非可视为 IS-A 衍生的 protocol 物件。public inheritance 应该只能用来塑膜唯一一件事情，一个真正的 IS-A 接口关系。

4.所有衍生的 classes 只是用 BasicProtocol 的 public 接口。这就意味着他并没有因为自己是 derived class 而受益，他们的工作可以轻易以一个型别为 BasicProtocol 的辅助物件完成

//设计准则：当我们希望塑膜出 is implemented in terms of 的关系时，
//请选择 membership/aggregation
//而不要使用 inheritance。只有在绝对必要下才使用 private inheritance
//也就是说当你需要存取 protected members 或是需要改写虚拟函数式时，
//绝对不要为了复用程序式的代码
//使用 public inheritance

ITEM23:classes 之间的关系

```
//资料库应用软件往往需要对某个已知的表格内的每一笔记录或
//是一群圈起来的记录做某些动作
//，他们常常会执行一些唯读动作，将整个表格走一遍，放进快取装置中，
//以便获取需要处理的记录
//然后进入实际动作做些处理。

//程式员不希望一再重复这种常见的逻辑，于是试着提供一个泛型
//抽象类别提供一个可复用的框架
//想法是抽象类别将重复动作封装起来，收集需要操作各个操作行，
//其次在执行必要动作。
//衍生类别负责提供特定的动作



//------------------------------------------------
// File gta.h
//------------------------------------------------
class GenericTableAlgorithm
{
public:
   GenericTableAlgorithm( const string& table );
   virtual ~GenericTableAlgorithm();

   // Process() returns true if and only if successful.
   // It does all the work: a) physically reads
   // the table's records, calling Filter() on each
   // to determine whether it should be included
   // in the rows to be processed; and b) when the
   // list of rows to operate upon is complete, calls
   // ProcessRow() for each such row.
   //
   bool Process();

private:
   // Filter() returns true if and only if the row should be
   // included in the ones to be processed. The
   // default action is to return true (to include
   // every row).
   //
   virtual bool Filter( const Record& );

   // ProcessRow() is called once per record that
   // was included for processing. This is where
   // the concrete class does its specialized work.
```

```
    // (Note: This means every row to be processed
    // will be read twice, but assume that that is
    // necessary and not an efficiency consideration.)
    //
    virtual bool ProcessRow( const PrimaryKey& ) =0;

    struct GenericTableAlgorithmImpl* pimpl_; // MYOB
};

//For example, the client code to derive a concrete worker
//class and use it in a mainline looks something like this:

class MyAlgorithm : public GenericTableAlgorithm
{
    // ... override Filter() and ProcessRow() to
    //         implement a specific operation ...
};
int main()
{
    MyAlgorithm a( "Customer" );
    a.Process();
}

//这是个好的设计  使用了哪个设计模式？为什么他在此处有用？

    //这是 Template Method pattern。他之所以有用，因为我们只需要遵循相同的步骤，
  //就可以将某个常见解法一般化。
    //只有细节部分不同，此部分可有衍生版本提供。

    //设计准则：尽量避免使用 public 虚拟函数式,最好以 template method pattern 取代之
    //设计准则：了解什么是 design pattern  并运用之。

  //2.再不改变基础设计的基础上，试评论此一设计的执行方式,
  //你有什么不同的做法？pimpl_member 的目的？

      //这个设计以 bool 做为传回码，显然没有其他方法记录失败。
      //试需求而定，这或许是好的，
      //但是有些东西需要注意。程序中 pimpl_member 非常巧妙的将
      //实作细节隐藏于一个不透明指标之后
      //pimpl_指向的结构将内含 private member functions 和 members variable
      //使得他们的任何改变不至于使得 client 端有必要重新设计。

//3. 对这个做法的改善
```

//设计准则：尽量形成内聚。总是尽力让每一段代码---每一个模组，
//每一个类别，每一个函数式--由单一而明确的任务

```
//---------------------------------------------------
// File gta.h
//---------------------------------------------------
// Responsibility #1: Providing a public interface
// that encapsulates common functionality as a
// template method. This has nothing to do with
// inheritance relationships, and can be nicely
// isolated to stand on its own in a better-focused
// class. The target audience is external users of
// GenericTableAlgorithm.
//
class GTAClient;

class GenericTableAlgorithm
{
public:
  // Constructor now takes a concrete implementation
  // object.
  //
  GenericTableAlgorithm( const string& table,
                         GTAClient& worker );

  // Since we've separated away the inheritance
  // relationships, the destructor doesn't need to be
  // virtual.
  //
  ~GenericTableAlgorithm();

  bool Process(); // unchanged

private:
  struct GenericTableAlgorithmImpl* pimpl_; // MYOB
};
//---------------------------------------------------
// File gtaclient.h
//---------------------------------------------------
// Responsibility #2: Providing an abstract interface
// for extensibility. This is an implementation
// detail of GenericTableAlgorithm that has nothing
// to do with its external clients, and can be nicely
// separated out into a better-focused abstract
```

```
// protocol class. The target audience is writers of
// concrete "implementation detail" classes which
// work with (and extend) GenericTableAlgorithm.
//
class GTAClient
{
public:
   virtual ~GTAClient() =0;
   virtual bool Filter( const Record& );
   virtual bool ProcessRow( const PrimaryKey& ) =0;
};

//-------------------------------------------------
// File gtaclient.cpp
//-------------------------------------------------
bool GTAClient::Filter( const Record& )
{
   return true;
}


class MyWorker : public GTAClient
{
   // ... override Filter() and ProcessRow() to
   //         implement a specific operation ...
};

int main()
{
   GenericTableAlgorithm a( "Customer", MyWorker() );
   a.Process();
}
```

//注意以下重要的三点

1.如果 GenericTableAlgorithm 的公共公开界面改变了，在原来的设计中，
所有具象的 worker classes 必须重新编译因为他们衍生自 GenericTableAlgorithm。
在新版本中， GenericTableAlgorithm 公开界面的任何改变都被巧妙的隔离,
一点也不会影响具象的 worker classes

2.如果 GenericTableAlgorithm 的可扩充协定改变了，
原来的版本中， GenericTableAlgorithm 的所有外部 client 都必须重新编译,
因为再次 class 定义区可见到衍生界面。但是在最新的版本中，
GenericTableAlgorithm 的可扩充协议界面的改变被隐藏了，任何
变化不会影响外部使用者。

3.任何具象的 worker classes 如今可以在其他任何演算法中被使用--只要
该演算法能够使用 Filter()和 ProcessRow()介面来进行--而不是

被 GenericTableAlgorithm 使用而已。

//可以将 GenericTableAlgorithm 改写为 一个泛式，不必是个 classes

```
bool GenericTableAlgorithm(
                        const string& table,
                        GTAClient& method)
{
   // ... original contents of Process() go here ...
}

int main()
{
   GenericTableAlgorithm( "Customer", MyWorker() );
}

//下面的模版并不好，请拒绝因为个人利益而写出一些险招秘技
template<typename GTACworker>
bool GenericTableAlgorithm( const string& table )
{
   // ... original contents of Process() go here ...
}

int main()
{
   GenericTableAlgorithm<MyWorker>( "Customer" );
}
```

ITEM24:滥用/使用继承

//元素继承的耦合性是最强的，设计时耦合性越弱越好

//一下 template 提供了 list 的管理功能，包括特定的 list 位置上的处理元素的能力

```
// Example 1
//
template <class T>
class MyList
{
public:
   bool      Insert( const T&, size_t index );
   T         Access( size_t index ) const;
   size_t Size() const;
private:
   T*        buf_;
   size_t bufsize_;
};
```

//考虑以下代码，其中呈现以 MyList 为基础，实作出一个 MySet class 的不同做法，所有的重要元素以呈现

```
// Example 1(a)
      //
      template <class T>
      class MySet1 : private MyList<T>
      {
      public:
         bool      Add( const T& ); // calls Insert()
         T         Get( size_t index ) const;
                                          // calls Access()
         using MyList<T>::Size;
         //...
      };

      // Example 1(b)
      //
      template <class T>
      class MySet2
      {
      public:
         bool      Add( const T& ); // calls impl_.Insert()
         T         Get( size_t index ) const;
                                          // calls impl_.Access()
         size_t Size() const;       // calls impl_.Size();
         //...
      private:
```

```
    MyList<T> impl_;
};
```

//MySet1 MySet2 不具有实际意义的差别 完成的功能一样

//设计准则：尽量的采用 aggregation(又名 composition,layering,HAS-A,delegation)来取代
//inheritance。当我们准备塑膜出 IS-IMPLEMENTED-IN-TERMS-OF 关系时，采用 aggregation
//而不要采用 inheritance
//有些事情 inheritance 做到而 containment 做不到

1. 当我们需要改写虚拟式时。
2. 当我们需要处理 protected members--一般而言是 protected member functions，
   有时候是 protected ctors
3. 当我们需要在一个 base subobject 之前先构建 used object，
   或是再一个 base subobject 之后摧毁 used object。我们必须用 inheritance
4. 当我们需要(1)分享某个共同的 virtual base classes 或(2)改写某个
   virtual base classes 的建构程序时，我们必须用 inheritance
5. 当我们从 empty base class 的最佳化获得实质利益时。有时候，
   据以实作的那个 class 并没有任何的数据成员，也就是说他只是函数的组合。
   这种情况下以 inheritance 取代 containment，可因 empty base class
   最佳化之故而获得空间优势。
   简单的说，编译器允许一个 empty base subobject 占用零空间：
   虽然一个 empty member object 必须占用非零空间，即使不含有任何资料

```
class B { /* ... functions only, no data ... */ };
        // Containment: incurs some space overhead
        //
        class D
        {
           B b_; // b_ must occupy at least one byte,
        };          // even though B is an empty class
        // Inheritance: can incur zero space overhead
        //
        class D : private B
        {           // the B base subobject need not
        };          // occupy any space at all
```

6. public inheritance 总是应该塑膜 IS-A 的关系。nonpublic inheritance
   可以表现一个收到束缚的 IS-A 关系。

//假设有个 class DErived：public Base，从外部看，DErived 物件并不是一个 Base，所以他无
//法以多型的方式当作一个 Base，因为 private inheritance 带来存取上的束缚，
//然而在 Derived 自己 的 members functions 及 friends 内，一个 DErived object 可以以多型的
//方式被拿来当作一个 base，这是应为 members functions 和 friend 有足够的存取权限
//如果不采用 private inheritance 而改用 protected inheritance，那么 IS-A 的关系对更深的

//DErived classes 会更明显些，意味着 subclasses 可以使用多型。

//分析一下 MyList MySet1 MySet2

1.MyList 没有 protected members 。所以我们不需要继承来存取他们
2.MyList 没有虚拟函数式，不需要继承来改写他们
3.MySet 没有其他潜在的 base classes，所以 MyList
物件不需要在另外一个 basesubject 之前构建或之后摧毁他。
4.MyList 没有任何的 virtual base classes 啊 hiMySet
可能需要共享的，或其 construction 可能需要改写的。
5.MyList 不是空的，所以 empty base classes 动机不适用
6.MySet 不是一个 MyList，甚至在 MySet 的 members functions 和 friend 内也不是。

//综述:MySet 不应该继承 MyList，在 containment 同样有效率的情况
//下使用 inheritance，只会导入无偿的偶尔及非必要的相依性

//有时候 inheritance 还是必要的
// Example 2: Sometimes you need to inherit
//
```
class Base
{
public:
   virtual int Func1();
protected:
   bool Func2();
private:
   bool Func3(); // uses Func1
};
```
//我们需要改写 Func2,或存取一个 protected member Func2,就必须采用 inheritance

//下面的代码不是很好，不建议采用
// Example 2(a)
//
```
class Derived : private Base // necessary?
{
public:
   int Func1();
   // ... more functions, some of which use
   //         Base::Func2(), some of which don't ...
};
```
//对上面的代码的改进
// Example 2(b)

```
//
class DerivedImpl : private Base
{
public:
   int Func1();
   // ... functions that use Func2 ...
};
class Derived
{
   // ... functions that don't use Func2 ...
private:
   DerivedImpl impl_;
};

//以上代码良好的隔离并封装对 Base 的依存性。DErived 只依存 Base 的
//public 介面以及 DerivedImpl 的 public 介面。


//Containment 的优点
   1.他允许我们拥有多个 used classes 实体，这对 inheritance 而言并不实用，甚至不可能
   2.他令 used classes 成为一个 data members，这带来更多弹性。

   //对 MySet2  的重写
   // Example 1(c): Generic containment
   //
   template <class T, class Impl = MyList<T> >
   class MySet3
   {
   public:
      bool     Add( const T& ); // calls impl_.Insert()
      T        Get( size_t index ) const;
                                          // calls impl_.Access()
      size_t Size() const;       // calls impl_.Size();
      // ...
   private:
      Impl impl_;
   };

   //遵循一下规则，避免常犯的错误
   1.如果 nonpublic inheritance 可用，绝对不要考虑 public inheritance
   2.如果 class 之间的相互关系可以用多种方式体现，请使用其中最弱的一种关系

   //设计准则:总是确定 public inheritance 用以塑模出 IS-A 和 WORKS-LIKE-A 的关系，
   //并符合 Liskov Substitution Principle.
```

//所有被改写的 member functions 都必须不要求更多，也不承诺更少

//设计准则:绝对不要为了重复使用 base class 的代码使用 public inheritance。
//public inheritance 的目的是为了被既有的代码以多型方式重复运用 base objects.

//结论：

1.明智的运用 inheritance
2.如果你能够单纯的以 containment/delegation 表现某个 class 的间关系，
你应该那么做。
3.如果你需要 inheritance 但不想塑模出 IS-A 的关系，请使用 nonpublic inheritance

ITEM26-28:编译期的安全性

//大多数程序员包含的头文件都比实际应用的多，这将严重影响程序建立的时间。

```
#include "a.h" //class A
#include "b.h" //class B
#include "c.h" //class C
#include "d.h" //class D
//注意只有 A 和 C 有虚拟函数
#include<iostream>
#include<ostream>
#include<sstream>
#include<list>
#include<string>

class X : public A{

    public:
        X (const &);
        D Function1(int char*);
        D Function1(int ,C);
        B& Function2(B);
        void Function3(std::wostringstream&);
        std::ostream& print(std::ostream&)const;
    private:
        std::string name_;
        std::list<C> clist_;
        D                   d_;

};
std::ostream& print(std::ostream& os,const X& x){

    return x.print(os);
}

class Y : private B{
public;
    C Function4(A);
private:
    std::list<std::wostringstream*> alist_;
};
```

//我们可以直接去掉的头文件有

1.iostream 程序虽然用到流，但是没有用到流的实际东西

2.ostream 和 sstream,程序的参数和返回类型被前置声明是可以的,只需要 iosfwd 就可以

//我们不能去掉的头文件
1. a.h
2. b.h
3. c.h
4. d.h

//我们可以通过 X 和 Y 使用 pimp_的方法去掉 d.h   list string   c.h

//主要的事项:即使 ostream 没有被定义，内联的 operator<<也
//可能仍然保持其内联性，并使用 ostream 参数。
//这是因为你只在调用成员函数的时候才真正的需要相应的定义，
//当你想接收一个对象并将其当成在
//函数调用式的参数而不做任何其他额外的事情时，你并不需要该函数的定义

//B 是 Y 的 private 基类，而且 B 没有虚函数，因此 b.h 是可以去掉的。
//有一个主要的原因我们想使用 private 继承
//那就是重载虚拟函数。这里应该让 Y 拥有一个 B 的成员。
//要去掉 b.h 我们应该让 Y 的这个类型为 B 的成员存在于 Y 中隐藏的 pimp_部分

//X 和 Y 两个类之间没有任何关系，因此我们至少可以把 X 和 Y 的定义分别放
//到两个不同的头文件中去
//为了不影响有的代码的使用，我们还应该把现有的头文件作为一个存根，
//让其包含 x.h 和 y.h
//如此我们至少可以让 y.h 不用#include "a.h" 因为现在它只把 A
//用作函数的参数类型，不需要 A 的定义。

//改进后的代码

```
//x.h
#include "a.h"
#include<iosfwd>

class C;
class D;

class X : public A{

    public:
        X (const &);
        D Function1(int char*);
        D Function1(int ,C);
        B& Function2(B);
        void Function3(std::wostringstream&);
```

```
        std::ostream& print(std::ostream&)const;
    private:
        class XImpl* pimpl_;

};
inline std::ostream& print(std::ostream& os,const X& x){
    //这里不需要 ostream 的定义
    return x.print(os);
}

//y.h 没有 #include
class A ;
class C;
class Y{
    public:
        C Function4(A);
    private:
        C YImpl* pimp_;
};

//got007.h 做为存根包含两个#include 又通过 x.h 附带另外两个#include
#include "x.h"
#include "y.h"

//got007.cpp

struct XImpl{//声明的时候用 class 这里可以用 struct

    std::string name_;
    std::list<C> clist_;
    D            d_;
};
struct YImpl{//声明的时候用 class 这里可以用 struct
    std::list<std::wostringstream*> alist_;
    B            b_;
};
```

//到现在 X 使用者只需要#include 包含 a.h 和 iosfwd.Y 的使用者只需要包含 a.h 和 iosfwd，即使后来更新代码包含 y.h 并去掉 got007.h,也不用多加#include

```
//英文版的有点不同哦
//    x.h: original header
//
#include <iostream>
#include <ostream>
#include <list>
// None of A, B, C, D or E are templates.
// Only A and C have virtual functions.
#include "a.h"    // class A
#include "b.h"    // class B
#include "c.h"    // class C
#include "d.h"    // class D
#include "e.h"    // class E
class X : public A, private B
{
public:
        X( const C& );
   B    f( int, char* );
   C    f( int, C );
   C& g( B );
   E    h( E );
   virtual std::ostream& print( std::ostream& ) const;
private:
   std::list<C> clist_;
   D                    d_;
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
   return x.print(os);
}

//1.Remove iostream.
//2.Replace ostream with iosfwd.
//3.Replace e.h with a forward declaration.

//根据以上改变后代码
//    x.h: sans gratuitous headers
//
#include <iosfwd>
#include <list>
// None of A, B, C or D are templates.
// Only A and C have virtual functions.
#include "a.h"    // class A
#include "b.h"    // class B
```

```
#include "c.h"    // class C
#include "d.h"    // class D
class E;
class X : public A, private B
{
public:
        X( const C& );
    B   f( int, char* );
    C   f( int, C );
    C& g( B );
    E   h( E );
    virtual std::ostream& print( std::ostream& ) const;
private:
    std::list<C> clist_;
    D            d_;
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
    return x.print(os);
}

    //We had to leave a.h and b.h. We couldn't get rid of these because X
    //inherits from both A and B, and you always have to have full definitions
    // for base classes so that the compiler can determine X's object size,
    //virtual functions, and other fundamentals. (Can you anticipate how to
    // remove one of these? Think about it: Which one can you remove, and why/how?
    //The answer will come shortly.)

//We had to leave list, c.h, and d.h. We couldn't get rid of these right
//away because a list<C> and a D appear as private data members of X.
// Although C appears as neither a base class nor a member, it is being
// used to instantiate the list member, and most current compilers require
// that when you instantiate list<C>, you are able to see the definition of C.
//(The standard doesn't require a definition here, though, so even if the
//compiler you are currently using has this restriction, you can expect the
//restriction to go away over time.)


//使用 pimp_之后
//    x.h: after converting to use a Pimpl
    //
    #include <iosfwd>
    #include "a.h"    // class A (has virtual functions)
    #include "b.h"    // class B (has no virtual functions)
```

```
class C;
class E;
class X : public A, private B
{
public:
        X( const C& );
   B    f( int, char* );
   C    f( int, C );
   C& g( B );
   E    h( E );
   virtual std::ostream& print( std::ostream& ) const;
private:
   struct XImpl;
   XImpl* pimpl_;
      // opaque pointer to forward-declared class
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
   return x.print(os);
}
//    Implementation file x.cpp
//
struct X::XImpl
{
   std::list<C> clist_;
   D                    d_;
};


//    x.h: after removing unnecessary inheritance
//
#include <iosfwd>
#include "a.h"    // class A
class B;
class C;
class E;
class X : public A
{
public:
        X( const C& );
   B    f( int, char* );
   C    f( int, C );
   C& g( B );
   E    h( E );
```

```
    virtual std::ostream& print( std::ostream& ) const;
private:
    struct XImpl;
    XImpl* pimpl_; // this now quietly includes a B
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
    return x.print(os);
}
```

//编码标准

1.封装和隔离

2.在声明类的时候，应该避免暴露出私有成员

3.应该使用一个形如 struct XxxxImpl* impl_ 的不透明的指针来存储私有成员(包括状态变量和成员函数)

```
    class Map{
        private:
            struct MapImpl* impl_;
    };
```

ITEM29:编译防火墙

```
//在 C++中，如果定义中的任何部分被改变了，那么这个类所有的使用者
//的代码必须重新编译。为了降低这种依赖性，使用的一种常见技术就是
//利用一个不透明的指针来隐藏一部分实现细节
class X
{
public:
   /* ... public members ... */
protected:
   /* ... protected members? ... */
private:
   /* ... private members? ... */
   struct XImpl;
   XImpl* pimpl_;              // opaque pointer to
                                  // forward-declared class
};

//1.哪些部分应该放入 Ximpl? 有常见的四种原则，他们是
    1) 将全部的私有成员(不包括函数)放入 Ximpl
     //这是个不错的开端，因为现在我们得以对任何只用作数据成员的类进行前置声明
     //(而不是使用#include 语句包含类真正声明--这会使客户端代码形成依赖性)。
    2) 将全部的私有成员(包括函数)放入 Ximpl
     //C++中客户端成员不应该关心这些部分就意味着私有，而私有的东西最好藏起来
     //警告:
     //1.即使虚函数是私有的，也不能将虚拟成员函数隐藏在 pimpl 类中。
     //   如果虚函数复写基类中的同名函数，那么虚函数就应出现在真正的派生类中
     //如果虚函数不是继承而来，为了让后来的类能给复写他，也应该出现在可见类中
     //2.如果 pimpl 中的函数需要使用其他函数，其可能需要一个指向可见对象的反向
     //指针--这增加了一层间接性。这个反向指针通常约定成 self_
    3) 将全部的私有成员和保护成员放入 Xipml
        //这种做法是错误的，保护成员绝对不能放入 pimpl 中
    4) 使用 Ximpl 完全成为原来的 X，将 X 编写为一个完全由简单的前置函数
       组成的公共接口
     //只有在有限的情况下用，好处是可以避免使用反向指针，
     //坏处是使得可见类对于继承完成无用，无论是基类还是派生类

//2.Ximpl 需要一个指向 X 对象的反向指针。
```

ITEM30:FAST PIMPL 技术

```
//标准的 malloc 和 new 调用开销很大
// Attempt #1
//
// file y.h
#include "x.h"
class Y
{
   /*...*/
   X x_;
};

// file y.cpp
Y::Y() {}
```

```
//这个 class Y 的申明需要 class X 的申明已经可见。为了避免这个条件
// Attempt #2
//
// file y.h
class X;
class Y
{
   /*...*/
   X* px_;
};

// file y.cpp
#include "x.h"
Y::Y() : px_( new X ) {}
Y::~Y() { delete px_; px_ = 0; }

//这很好的隐藏了 X 的，但是当 Y 被广泛使用时，动态内存的开销降低了性能。
//下面是一个看似很好的解决方案，其实是糟糕的
// Attempt #3
//
// file y.h
class Y
{
   /*...*/
   static const size_t sizeofx = /*some value*/;
   char x_[sizeofx];
};
```

```
// file y.cpp
#include "x.h"
Y::Y()
{
   assert( sizeofx >= sizeof(X) );
   new (&x_[0]) X;
}
Y::~Y()
{
   (reinterpret_cast<X*>(&x_[0]))->~X();
}


//FAST PIMPL 技术
// file y.h
class Y
{
   /*...*/
   struct YImpl;
   YImpl* pimpl_;
};

// file y.cpp
#include "y.h"
struct Y::YImpl
{
   /*...private stuff here...*/
   static void* operator new( size_t )      { /*...*/ }
   static void    operator delete( void* ) { /*...*/ }
};
X::X() : pimpl_( new XImpl ) {}
X::~X() { delete pimpl_; pimpl_ = 0; }

//小心
//这个虽然好，但也不要乱用 FAST PIMPL。你得到了最佳的内存分配速度。
//但是别忘记代价:
//维护这些链表通常导致空间效能下降，因为比通常情况造成更多的碎片。

//当你证明确实需要他时，在使用。
```

ITEM31:名字搜索

//在下面的代码中，调用的是哪个函数？为什么？

```
namespace A
{
   struct X;
   struct Y;
   void f( int );
   void g( X );
}

namespace B
{
   void f( int i )
   {
  //这个 f()调用的是它自己，并且是无穷递归，在 namespace A  也有一个 f(int)函数
  //如果 B 写了 using namespace A 或  using A::f,那么 A:f 将是寻找 f(int)的候选函数
  //那么 f(i)的调用将存在二义性，由于 B 没有引入，所以不存在二义性
      f( i );      // which f()?
}
   void g( A::X x )
   {
      //这个调用存在二义性，程序员必须使用命名空间确定使用哪个函数
      //规则:如果你给函数提供一个 class 型的实参(此处时 A::X 的类型 x)
      //那么在名称收索时，编译器将认为包含实参类型的命名空间中的同名函数是
      //候选函数
      g( x );      // which g()?
   }
   void h( A::Y y )
   {
      //没有其他的 h(A::Y)函数，所以在这 f()调用的是他自己，也是无穷递归
      //虽然 B::h()原型使用 namespace A  的一个类型，但是不影响收索结果
      //namespace A 没有符合 h(A::Y)名称的函数
      h( y );      // which h()?
   }
}
```

ITEM32-34:接口原则

```
//*** Example 1 (a)
class X { /*...*/ };
/*...*/
void f( const X& );

//*** Example 1 (b)
class X
{
   /*...*/
public:
   void f() const;
};

//接口原则：对于一个类 X，包括自由函数，只要同时满足
//1.提到 X，并且
// 2 .与 X 同期提供(如 Example 1 (a))
//那么就是 X 的逻辑组成部分，因为他们形成了 X 的接口
//那么根据定义所有的成员函数都是 X 的组成部分，因为
// 1 .每个成员函数都必须提到 X（非 static 成员函数有隐含的 X* cont  或 const X* const）
//2.每个成员函数都和 X 同期提供


    //*** Example 1 (c)
    class X { /*...*/ };
    /*...*/
    ostream& operator<<( ostream&, const X& );

    //如果 operator<<与 X 同期提供，那么他就是 X 的逻辑组成部分

    //class 的定义:一个 class 描述了一组数据及操作这些数据的函数

    //考虑下面的例子
    /*** Example 2 (a) ***/
    struct _iobuf { /*...data goes here...*/ };
    typedef struct _iobuf FILE;
    FILE* fopen ( const char* filename,
                  const char* mode );
    int     fclose( FILE* stream );
    int     fseek ( FILE* stream,
                    long    offset,
                    int     origin );
    long    ftell ( FILE* stream );
           /* etc. */
```

```
//考虑把下面的例子用 class 重写
//*** Example 2 (b)
class FILE
{
public:
   FILE( const char* filename,
         const char* mode );
~FILE();
   int    fseek( long offset, int origin );
   long ftell();
        /* etc. */
private:
   /*...data goes here...*/
};
//当然我们可以将函数实现一部分为成员函数，一部分为非成员函数
//*** Example 2 (c)
class FILE
{
public:
   FILE( const char* filename,
         const char* mode );
~FILE();
   long ftell();
        /* etc. */
private:
   /*...data goes here...*/
};
int fseek( FILE* stream,
           long    offset,
           int     origin );

//是不是成员函数不是问题的关键，只要他提及 FILE 并且与 FILE 同期提供，他
//就是 FILE 的组成部分


//介绍 Koenig Lookup
//看下面的例子
//*** Example 3 (a)
namespace NS
{
   class T { };
   void f(T);
}
NS::T parm;
```

```
int main()
{
   f(parm);       // OK: calls NS::f
}
//如果你传给函数一个 class 类型的实参，为了查找这个函数名，
//编译器被要求不仅要搜索如局部作用域这样的常规空间，还要搜索
//包含实参类型的命名空间。

//*** Example 3 (b)
#include <iostream>
#include <string>    // this header declares the function
                     //      std::operator<< for strings
int main()
{
   std::string hello = "Hello, world";
   std::cout << hello;      // OK: calls std::operator<<
}
//如果没有 Koenig Lookup  编译器无法找到 operator<<.
//我们必须写 std::operator<<(std::cout,hello);或  using namespace std;


//看下面的例子
//*** Example 4 (a)
namespace NS      // this part is
{                 // typically from
class T { }; // some header
}                 // file T.h
void f( NS::T );
int main()
{
   NS::T parm;
   f(parm);       // OK: calls global f
}
//
//*** Example 4 (b)
namespace NS      // typically from
{                 // some header T.h
   class T { };
   void f( T ); // <-- new function
}
void f( NS::T );
int main()
{
   NS::T parm;
```

```
   f(parm);        // ambiguous: NS::f
}                  //      or global f?
//在命名空间增加一个函数破坏了命名空间外的代码

//*** The Myers Example: "Before"
namespace A
{
   class X { };
}
namespace B
{
   void f( A::X );
   void g( A::X parm )
   {
      f(parm);     // OK: calls B::f
   }
}//这很好，但是当下面的代码出现时
//*** The Myers Example: "After"
namespace A
{
   class X { };
   void f( X ); // <-- new function
}
namespace B
{
   void f( A::X );
   void g( A::X parm )
   {
      f(parm);     // ambiguous: A::f or B::f?
   }
}

//如果代码提供了一个提及 X 的函数，而他与 X 所处的命名空间提供某
//个函数签名重合时，调用就会有二义性，看下面的代码
namespace A
{
   class X { };
   ostream& operator<<( ostream&, const X& );
}
namespace B
{
   ostream& operator<<( ostream&, const A::X& );
   void g( A::X parm )
   {
```

```
        cout << parm; // ambiguous: A::operator<< or
    }                         //      B::operator<<?
}


//解决方法
//*** NOT the Myers Example
namespace A
{
    class X { };
    void f( X );
}
class B           // <-- class, not namespace
{
    void f( A::X );
    void g( A::X parm )
    {
        f(parm); // OK: B::f, not ambiguous
    }
};


//看下面两组代码
//*** Example 5 (a) -- nonvirtual streaming
class X
{
    /*...ostream is never mentioned here...*/
};
ostream& operator<<( ostream& o, const X& x )
{
    /* code to output an X to a stream */
    return o;
}
//*** Example 5 (b) -- virtual streaming
class X
{
    /*...*/
public:
    virtual ostream& print( ostream& ) const;
};
ostream& X::print( ostream& o ) const
{
    /* code to output an X to a stream */
    return o;
```

```
}
ostream& operator<<( ostream& o, const X& x )
{
   return x.print( o );
}
//选择 a 的好处是，依赖性低。没有虚函数调用开销
//选择 b 的好处是，可以正确的打印出派生类

//但是从接口原则来看 我们应该选择 b，两种情况都有依赖性

//一些有趣甚至诧异的结果
//通常，如果 A 和 B 都是 class，并且 f(A,B)是一个自由函数
//1.如果 A 与 f 同期提供，那么 f 是 A 的组成部分，并且 A 将依赖 B
//2.如果 B 与 f 同期提供，那么 f 是 B 的组成部分，并且 B 将依赖 A
//3.如果 A，B，f 都是同期提供的，那么 f 同时是 A 和 B 的组成部分，并且
      //A 与 B 是循环依赖。如果一个库提供了两个 class 及同时涉及二者的操作
      //那么这哥操作必须规定被同时使用。


//如果 A 与 B 都是 class，并且 A::g(B)是 A 的成员函数
//1.因为 A::g(B)的存在，A 总是依赖 B
//2.如果 A 与 B 是同期提供的，那么 A::g(B)也是同期提供的。因为 A::g(B)
   //同时满足提及 B 和与 B 同期提供，那么 A::g(B)是 B 的组成部分，
   //而 A::g(B)使用了一个
   //A*参数，所以 B 依赖 A，因为 A 也依赖 B，意味着 A,B 循环依赖。


//*** Example 6 (a)
//---file a.h---
namespace N { class B; }// forward decl
namespace N { class A; }// forward decl
class N::A { public: void g(B); };
//---file b.h---
namespace N { class B { /*...*/ }; }
//A 的用户包含了 a.h，于是 A 和 B 是同期提供的并且是循环依赖的。
//B 用户包含了 b.h 于是 A 和 B 不是同期提供的

//总结
1.接口原则：对于 class X ,所有函数，包括自由函数，只要同时满足
    a)提及 X
    b)与 X 同期提供
       那么他就是 X 的逻辑组成部分，因为他是 X 的接口一部分。
```

2.非成员函数和成员函数都是 class 的逻辑组成部分，
只不过成员数比非成员函数有更强的关联关系
3. 接口原则中，对同期提供最有用的解释是:出现在相同的头文件或命名空间中。

ITEM35:内存管理(一)

//下面总结出 C++程序的主要内存区域

//1.常量数据区

//主要存取字符串等在编译期间就能确定的值类对象不能存在这个区域

//在程序的整个生命周期内可用

//2.栈区

//栈区主要存取自动变量。一般来说栈区的分配操作比动态存取区或自由存取区快的多

//3.自由存取区

//C++两个动态内存区域之一，使用 new 和 delete 予以分配和释放

//4.堆区

//另一块动态内存区域，使用 malloc，free 以及一些相关变量来进行分配和回收。

//5.全局/静态区

//全局的或静态的变量和对象所占用的内存区域在程序启动的时候才被分配，

//而且可能直到程序开始执行的时候才初始化

ITEM36:内存管理(二)

//下面的程序有一些类使用他们自己的内存管理方案，找出错误

```
class B
 {
 //为什么 B 的 delete 还有第二个参数？
 //为什么 D 的 delete 没有第二个参数？
 //这两种都可以正常的回收，因个人喜好不同
 //但是没有提供相应的 new  和 new[]是非常危险的
 public:
    virtual ~B();
    void operator delete    ( void*, size_t ) throw();
    void operator delete[]( void*, size_t ) throw();
    void f( void*, size_t ) throw();
 };
 class D : public B
 {
 public:
    void operator delete    ( void* ) throw();
    void operator delete[]( void* ) throw();
 };

void f()
{
    //下面哥哥语句，到底哪一个 delete 被调用了？为什么？
    //调用时的参数是什么
        D* pd1 = new D;
        delete pd1;//D::operator delete(void*).
        B* pb1 = new D;
        delete pb1;//D::operator delete(void*).
        D* pd2 = new D[10];
        delete[] pd2;//D::operator delete[](void*).
        B* pb2 = new D[10];
        delete[] pb2;//不可预料的，C++语言要求，传递给 delete 的指针的静态类型必
                     //须和动态类型一样。


}
//下面两个赋值语句合法吗
B b;
typedef void (B::*PMF)(void*, size_t);
    PMF p1 = &B::f;
    PMF p2 = &B::operator delete;
    //第一个赋值语句合法
```

```
//第二个不合法，因为 void operator delete( void*, size_t ) throw()
//并不是 B 的成员函数。因为 new 和 delete 是静态的

//小技巧
//new 和 delete 总是静态的，即使不显示的声明为 static，总是把他们
//声明为 static 是个好的习惯，可以帮助理解程序
class X
{
public:
   void* operator new( size_t s, int )
                              throw( bad_alloc )
   //会产生内存泄漏，因为没有响应的 placement delete 存在。
   {
      return ::operator new( s );
   }
};
class SharedMemory
{
public:
   static void* Allocate( size_t s )
   {
      return OsSpecificSharedMemAllocation( s );
   }
      static void    Deallocate( void* p, int i = 0 )
      {
         OsSpecificSharedMemDeallocation( p, i );
      }
   };

   class Y
   {
   public:
      void* operator new( size_t s,
                                    SharedMemory& m ) throw( bad_alloc )
      {
         return m.Allocate( s );
      }
      //若没有 operator delete 也会造成内存泄漏，
      //若在构造函数抛出异常就不会释放内存
    /* void    operator delete( void* p,
                                       SharedMemory& m,
                                       int i ) throw()
      {
          m.Deallocate( p, i );
```

```
    }*/
};
//如下面的构造函数抛出异常
SharedMemory shared;
    ...
    new (shared) Y; // if Y::Y() throws, memory is leaked


    void operator delete( void* p,
                          SharedMemory& m,
                          int i ) throw()
      {
        //这里的 delete 毫无用处，因为他不会被调用
        m.Deallocate( p, i );
      }
    };
    void operator delete( void* p,
                          std::nothrow_t& ) throw()
    {
//这是一个严重的错误，因为他将要删除哪些缺省::operator new 分配出来的内存
//而不是 shareMemory::Allocate()分配出来的内存。
      SharedMemory::Deallocate( p );
    }

    void operator delete( void* p,
                          std::nothrow_t& ) throw()
    {
      //这也是一样，只是更为微妙。这里 delete 只会在 new(nothrow) T
      //失败的时候才会被调用
      //因为 T 的构造函数会带来一个异常来终止，
      //并企图回收那些不是被 SharedMemory::Allocate()分配的内存
     SharedMemory::Deallocate( p );
    }
```

ITEM37:auto_ptr

```
//下面代码哪些是好的？哪些是安全的？哪些是合法的额？哪些是非法的？
auto_ptr<T> source()
{
   return auto_ptr<T>( new T(1) );
}
void sink( auto_ptr<T> pt ) { }
void f()
{
   auto_ptr<T> a( source() );
   sink( source() );
   sink( auto_ptr<T>( new T(1) ) );
   vector< auto_ptr<T> > v;
   v.push_back( auto_ptr<T>( new T(3) ) );
   v.push_back( auto_ptr<T>( new T(4) ) );
   v.push_back( auto_ptr<T>( new T(1) ) );
   v.push_back( a );
   v.push_back( auto_ptr<T>( new T(2) ) );
   sort( v.begin(), v.end() );
   cout << a->Value();
}
class C
{
public:       /*...*/
protected: /*...*/
private:      /*...*/
   auto_ptr<CImpl> pimpl_;
};

//auto_ptr 的最初动机是:如果下面的语句遇到异常没有执行 delete 语句时候也安全
// Example 1(a): Original code
//
void f()
{
   T* pt( new T );
   /*...more code...*/
   delete pt;
}
//修改后的代码
// Example 1(b): Safe code, with auto_ptr
//
void f()
{
   auto_ptr<T> pt( new T );
```

```
   /*...more code...*/
} // cool: pt's destructor is called as it goes out
   // of scope, and the object is deleted automatically

//一些例子
//Finally, using an auto_ptr is just about as easy as
//using a builtin pointer, and to "take back" the resource
//and assume manual ownership again, we just call release().

// Example 2: Using an auto_ptr
//
void g()
{
   T* pt1 = new T;
   // right now, we own the allocated object
   // pass ownership to an auto_ptr
   auto_ptr<T> pt2( pt1 );
   // use the auto_ptr the same way
   // we'd use a simple pointer
   *pt2 = 12;              // same as "*pt1 = 12;"
   pt2->SomeFunc(); // same as "pt1->SomeFunc();"
   // use get() to see the pointer value
   assert( pt1 == pt2.get() );
   // use release() to take back ownership
   T* pt3 = pt2.release();
   // delete the object ourselves, since now
   // no auto_ptr owns it any more
   delete pt3;
} // pt2 doesn't own any pointer, and so won't
   // try to delete it... OK, no double delete

//Finally, we can use auto_ptr's reset() function to reset
//the auto_ptr to own a different object. If the auto_ptr
//already owns an object, though, it first deletes the already-owned
//object, so calling reset() is much the same as destroying
//the auto_ptr and creating a new one that owns the new object.

// Example 3: Using reset()
//
void h()
{
   auto_ptr<T> pt( new T(1) );
   pt.reset( new T(2) );
      // deletes the first T that was
```

```
        // allocated with "new T(1)"
    } // finally, pt goes out of scope and
      // the second T is also deleted


// Example 4(b): A safer Pimpl, using auto_ptr
        //
        // file c.h
        //
        class C
        {
        public:
          C();
          ~C();
          /*...*/
        private:
          struct CImpl; // forward declaration
          auto_ptr<CImpl> pimpl_;
          C& operator = ( const C& );
          C( const C& );
        };
        // file c.cpp
        //
        struct C::CImpl { /*...*/ };
        C::C() : pimpl_( new CImpl ) { }
        C::~C() {}

        // Example 5: Transferring ownership from
        //                  one auto_ptr to another
        //
        void f()
        {
          auto_ptr<T> pt1( new T );
          auto_ptr<T> pt2;
          pt1->DoSomething(); // OK
          pt2 = pt1;    // now pt2 owns the pointer,
                        // and pt1 does not
          pt2->DoSomething(); // OK
        } // as we go out of scope, pt2's destructor
          // deletes the pointer, but pt1's does nothing


        // Example 6: Never try to do work through
        //                  a non-owning auto_ptr
```

```
//
void f()
{
   auto_ptr<T> pt1( new T );
   auto_ptr<T> pt2;
   pt2 = pt1;    // now pt2 owns the pointer, and
                 // pt1 does not
   pt1->DoSomething();
                 // error: following a null pointer
}

//下面分析开头部分提出的问题

auto_ptr<T> source()
{
   return auto_ptr<T>( new T(1) );//安全合法的做法
}
void sink( auto_ptr<T> pt ) { }//安全合法的做法

void f()
{
   auto_ptr<T> a( source() );//安全合法的做法
   sink( source() );//安全合法的做法
   sink( auto_ptr<T>( new T(1) ) );//安全合法的做法
   vector< auto_ptr<T> > v;//将 auto_ptr 放入容器是不安全的且不合法，
                           //对于 auto_ptr 拷贝是不对等的
   v.push_back( auto_ptr<T>( new T(3) ) );
   v.push_back( auto_ptr<T>( new T(4) ) );
   v.push_back( auto_ptr<T>( new T(1) ) );
   v.push_back( a );
   v.push_back( auto_ptr<T>( new T(2) ) );
   sort( v.begin(), v.end() );//不安全不合法
   cout << a->Value();//不安全不合法
}

class C
{
public:       /*...*/
protected: /*...*/
private:      /*...*/
   auto_ptr<CImpl> pimpl_;//安全合法的做法
};

//精修后的 auto_ptr 也使得拷贝 const auto_ptr 非法
```

```
        const auto_ptr<T> pt1( new T );
// making pt1 const guarantees that pt1 can
// never be copied to another auto_ptr, and
// so is guaranteed to never lose ownership
        auto_ptr<T> pt2( pt1 ); // illegal
        auto_ptr<T> pt3;
        pt3 = pt1;                      // illegal
        pt1.release();                  // illegal
        pt1.reset( new T );             // illegal
```

ITEM38:对象等同问题

```
//防止自我赋值，下面的代码是充分必要的吗？
T& T::operator=( const T& other )
{
   if( this != &other )       // the test in question
   {
      // ...
   }
   return *this;
}
//从技术上看，它既不是必要的也不是充分的。在实践当中，他工作的颇好但是
//也有可能在 C++标准中被修改
```

ITEM39:自动转换

//标准的 string 不具有向 const char* 进行自动转换的能力。它应该有吗？

//把 string 做为 C 风格的 const char*来进行访问，经常是很有用的。
//实际上，string 也有一个 c_str()
//专门完成这个任务---该函数返回 const char*。下面代码体现两者区别

```
string s1("hello"), s2("world");
strcmp( s1, s2 );                    // 1 (error)  不存在自动转换
strcmp( s1.c_str(), s2.c_str() ); // 2 (ok)
```

//标准的 string 不具有向 const char* 进行自动转换的能力。它不应该有
//隐式转换是不安全的

a) 他会影响重载的解析
b) 他会使错误的代码不声不响的通过编译

//请看下面的例子

```
string s1, s2, s3;
   s1 = s2 - s3;      // oops, probably meant "+"  可能是+，-是无意义的
                      //如果存在隐式转换就不会提示该问题。
```

ITEM40:对象的生存周期

```
//评述下面的程序段，#2 处代码是安全或合法的吗？
void f()
  {
    T t(1);
    T& rt = t;
    //--- #1: do something with t or rt ---
    t.~T();
    new (&t) T(2);
    //--- #2: do something with t or rt ---
  }// t is destroyed again

//是的 #2 处的代码是安全且合法的，但是
  a)  函数做为一个整体，他是不安全得，而且
  b)  这样是一个坏习惯

//C++标准草案明确规定，允许这种代码出现。现场的析构和重构造不会使 rt
//这个引用失效。(当然，你不能在 t.~T()与 placement new 之间使用 t 或 rt,因为在那段时期
  //不存在任何对象。我们还假设 T::operator&()没有被重载，即没有用来做返回对象之
  //地址以外的其他事情)。因此只考虑这段代码，#2 是安全的，但是 f()做为一个整体
  //可能不是异常安全的


  //为什么函数是不安全的
  //因为如果在 T(2)处有异常，那么由于 t 是自动变量，所以 t 在末尾时候仍然调用 T::~T()
  //t 被销毁了 2 次，所以不安全的


  //在实际中，这种技术根本不会被使用，考虑下面的代码
  // Can you spot the subtle problem?
      //
      void T::DestroyAndReconstruct( int i )
      {
        this->~T();
        new (this) T(i);
      }
      //这种技术基本上来说不算是安全的，看下列代码
      class U : public T { /* ... */ };
      void f()
      {
        /*AAA*/ t(1);
        /*BBB*/& rt = t;
        //--- #1: do something with t or rt ---
        t.DestroyAndReconstruct(2);
```

```
    //--- #2: do something with t or rt ---
}// t is destroyed again

//如果/*AAA*/ 是 T ，那么#2 处代码仍然可行，即使/*BBB*/不是 T
//(/*BBB*/可能是 B 的基类)
//如果/*AAA*/ 是 U(而不是 T) ，那么无论/*BBB*/是什么，都毫无悬念。
//总之，最好不要用这种写法。
```

ITEM41:对象的生存周期

```
//评述下面的惯用法
T& T::operator=( const T& other )
{
   if( this != &other )
   {
      this->~T();
      new (this) T(other);
   }
   return *this;
}
//1.代码试图达到什么样的合法目的？
//这个惯用法是经常被推荐使用的，是 C++草案的例子，但是这个例子是有害无益的
//这个惯用法以拷贝构造来实现拷贝赋值，该方法保证拷贝构造与拷贝赋值是相同的操作
//防止不必要的重复。如果虚拟基类有数据成员这个方法还是挺有用的，其实虚拟基类不
//该有数据成员。既然是虚拟基类，说明该类是用于继承而设计的，这意味着我们不能用
//这种用法

//上面的代码包含一个可以修正的缺陷和以及若干个无法修正的缺陷
   //#1  它会切割对象
   //如果 T 是一个带有虚拟析构函数的基类，那么 this->~T();
   //就会出现问题这一句就执行了错误的操作。如果是对一个
   //派生类对象执行这个调用操作，这一句的执行将会销毁派生类对象并用一个 T
   //对象代替。而这种结果几乎肯定破坏性的影响后续任何试图使用这个对象的代码。
      //看下面的代码
         Derived&
         Derived::operator=( const Derived& other )
         {
            Base::operator=( other );
            // ...now assign Derived members here...
            return *this;
         }
         //这样我们得到
         class U : T { /* ... */ };
         U& U::operator=( const U& other )
         {
            T::operator=( other );
            // ...now assign U members here...
            //       ...oops, but we're not a U any more!
            return *this;    // likewise oops
         }
      //对 T::operator=()的调用一声不响的对其后的代码(包括 U
      //成员的赋值操作和返回语句)
      //产生了破坏性的影响。为了改正这个问题，可以改为
```

```
    //this->T::~T();做为替代。但是这仍然是不安全的。
     //#2 他不是异常安全的
        //new 调用拷贝构造可抛出异常
     //#3 他使赋值操作变得低效
     //#4 改变了正常对象的声明周期
     //#5 他可以对派生类对象产生破坏性的影响
     //#6 依赖于不可靠的指针比较操作

//2.加入修正了所有的缺陷，这种惯用法安全吗？
//对你的回答作出解释，如果不安全，如何达到目的
//我们以拷贝赋值来实现拷贝构造
T::T(const T& other){

     /*T:: */ operator=(other);

}
T& T::operator=(const T& other){

     //真正的工作从这里开始
     //大概可以在异常安全的状态下完成。但现在
     //其可以抛出异常，却不像原来那样产生什么不良影响
     return *this;
}
//为了代码美观  你可以这样做,和上面的没有区别
T::T(const T& other){
     do_copy(other);
}

T& T::operator=(const T& other){
     do_copy(other);
     return *this;
}
T& T::do_copy(const T& other){
     //真正的工作从这里开始
     //大概可以在异常安全的状态下完成。但现在
     //其可以抛出异常，却不像原来那样产生什么不良影响
}
```

ITEM42:变量的初始化

```
//下面的四条语句有何区别
T t; //t 被缺省的构造函数 T::T()初始化
T t();//函数声明，返回类型是 T
T t(u);//直接初始化，t 通过构造函数 T::T(u)初始化
T t = u;//拷贝初始化


//学习指导　建议总是使用 T t(u)的形式，一来只要可以用 T t=u 的地方
//都可以用 T t(u)的形式，而来还有其他优点，支持多个参数
```

ITEM43:正确使用 const

```
//下面持续的 const 使用是否正确
class Polygon
{
public:
   Polygon() : area_(-1) {}
   void    AddPoint( const Point pt )
   { //1.因为 Point 对象采用传值方式，所以声明为 const 几乎没有油水
      InvalidateArea();
      points_.push_back(pt);
   }
   Point GetPoint( const int i )
  { //2.因为 Point 对象采用传值方式，所以声明为 const 几乎没有油水，反而容器误解
   //3.这个成员函数应该是 const，因为不改变对象的状态
   //4.如果该函数的返回类型不是一个内建类型的话，
        //通常应该将其返回类型也声明为 const 这样做有利于该函数的调用者，
        //因为它是的编译器能够在函数调用者企图修改临时变量
        //时产生一个错误，从而达到保护目的。如果真的想修改临时变量，应该让 GetPoint
        //返回一个引用。
            return points_[i];
  }
   int     GetNumPoints()
   { //5.函数本身应该声明为 const
            return points_.size();
   }
   double GetArea()
   {//6.尽管这个函数修改了对象的内部状态，但是还是应该声明
    //为 const,因为被修改对象的可见状态没发生变化。
    //这意味着 area_应该被声明为 mutable.记住一定让函数声明为 const
     if( area_ < 0 ) // if not yet calculated and cached
     {
        CalcArea();         // calculate now
     }
     return area_;
   }
private:
   void InvalidateArea()
   { //7.这个函数应该声明为 const
      area_ = -1;
   }
   void CalcArea()
   {//8.这个成员函数绝对应该被声明为 const,不管怎么说，
    //它至少会被另外一个 const 成员函数 GetArea()调用
     area_ = 0;
```

```
    vector<Point>::iterator i;
        //9.既然 iterator 不会改变 points_集的状态，所以这里应该是一个 const_iterator
    for( i = points_.begin(); i != points_.end(); ++i )
      area_ += /* some work */;
  }
  vector<Point> points_;
  double        area_;
};
Polygon operator+( Polygon& lhs, Polygon& rhs )
{//10.参数应该是 const 引用传递
    //11.返回值应该是 const
  Polygon ret = lhs;
  int last = rhs.GetNumPoints();
  //12.既然 last 也从不会改变，那么应该为 const int 类型
  for( int i = 0; i < last; ++i ) // concatenate
  {
    ret.AddPoint( rhs.GetPoint(i) );
  }
  return ret;
}

void f( const Polygon& poly )
{//13.这里的 const 毫无作用，因为无论如何一个引用是不可能被改变，
//使其指向另一个对象
  const_cast<Polygon&>(poly).AddPoint( Point(0,0) );
}

void g( Polygon& const rPoly )
{ //13.这里的 const 毫无作用，因为无论如何一个引用是不可能被改变，
 //使其指向另一个对象
    rPoly.AddPoint( Point(1,1) );
}

void h( Polygon* const pPoly )
{ //14.这里的 const 没有用，对指针使用传值方式毫无意义
    pPoly->AddPoint( Point(2,2) );
}

int main()
{
  Polygon poly;
  const Polygon cpoly;
  f(poly);
  f(cpoly);
```

```
    g(poly);
    h(&poly);
}
```

//学习指导：在函数内对非内建的类型采用值返回方法时，最好让函数返回一个 const 值
//修改后的代码 不要考虑代码风格的好坏

```
class Polygon
{
public:
    Polygon() : area_(-1) {}

    void    AddPoint( Point pt )
    {
        InvalidateArea();
        points_.push_back(pt);
    }

    const Point GetPoint( int i ) const    { return points_[i]; }

    int    GetNumPoints() const { return points_.size(); }

    double GetArea() const
    {
        if( area_ < 0 ) // if not yet calculated and cached
            CalcArea();         // calculate now
        return area_;
    }

private:
    void InvalidateArea() const { area_ = -1; }

    void CalcArea() const
    {
        area_ = 0;
        vector<Point>::const_iterator i;
        for( i = points_.begin(); i != points_.end(); ++i )
            area_ += /* some work */;
    }

    vector<Point>    points_;
    mutable double area_;
};

const Polygon operator+( const Polygon& lhs,
```

```
                                    const Polygon& rhs )
{
   Polygon ret = lhs;
   const int last = rhs.GetNumPoints();
   for( int i = 0; i < last; ++i ) // concatenate
      ret.AddPoint( rhs.GetPoint(i) );
   return ret;
}

void f( Polygon& poly )
{
   poly.AddPoint( Point(0,0) );
}

void g( Polygon& rPoly ) { rPoly.AddPoint( Point(1,1) ); }
void h( Polygon* pPoly ) { pPoly->AddPoint( Point(2,2) ); }

int main()
{
   Polygon poly;
   f(poly);
   g(poly);
   h(&poly);
}
```

ITEM44:类型转换

```
class    A { public: virtual ~A(); /*...*/ };
A::~A() { }

class B : private virtual A    { /*...*/ };

class C : public    A                { /*...*/ };

class D : public B, public C { /*...*/ };

A a1; B b1; C c1; D d1;
const A    a2;
const A& ra1 = a1;
const A& ra2 = a2;
char c;


//C++风格的转型
//具体的见 More Effective C++ Rule 2
//1.const_cast
//2.dynamic_cast//不能与 C 语言相对应
//3.reinterpret_cast
//4.static_cast


//下列每一条语句，用新风格写，哪条不用新风格是不正确的
        void f()
        {
           A* pa; B* pb; C* pc;
           pa = (A*)&ra1;//pa = const_cast<A*>(&ra1);
           pa = (A*)&a2;//这一句无法使用新转型
           pb = (B*)&c1;// pb = reinterpret_cast<B*>(&c1);
           pc = (C*)&d1;//这句在 C 中是错误的，而 C++不需要转型 pc = &d1;

        }

//评判下列每一条 C++转型编写风格的正确性

//首先注意，我们并不知道本条款中给出的类时候拥有虚拟函数，如果涉及转型的类
//不具有虚拟函数，那么对 dynamic_cast 的转型是错误的
 void g()
 {
    unsigned char* puc = static_cast<unsigned char*>(&c);
    signed char* psc = static_cast<signed char*>(&c);
```

```
  //这两条语句应使用 reinterpret_cast ， 原因是:char .signed char 以及
  //unsigned char 是三个互不相同，区别开来的类型。尽管他们之间存在隐式转换
  //但是他们之间是互不联系的，当然指针也是互不联系的

  void* pv = static_cast<void*>(&b1);
  B* pb1 = static_cast<B*>(pv);
 //这两句都不错，但是第一句的转型是不必要的，
 //本来就有一个对象指针到 void*隐式转换
  B* pb2 = static_cast<B*>(&b1);
//这句也不错，但是转型是不必要的，其引数已经是一个 B*
  A* pa1 = const_cast<A*>(&ra1);
  //这一句的是合法的，但是使用转型去掉常量性是潜在不良风格的体现。
 //我们通常可以使用 mutable 关键字完成
  A* pa2 = const_cast<A*>(&ra2);
//错误，如果该指针被用来对对象施行写操作，会产生未定义行为。
//因为 a2 是 const object，会放在只读存储区
  B* pb3 = dynamic_cast<B*>(&c1);
//错误，这一句将 pb3 设置为 NULL，因为 c1 根本不是一个 B 对象。
  A* pa3 = dynamic_cast<A*>(&b1);
//错误，因为 b1 不是一个 A(B 是以 private 派生与 A 的)
  B* pb4 = static_cast<B*>(&d1);
//不错但是没有必要
  D* pd = static_cast<D*>(pb4);
//不错。如果你认为这里需要一个 dynamic_cast，那就不对了。
//因为当目标已知的时候，向下转型可以是静态的。
     pa1 = dynamic_cast<A*>(pb2);
//错误，你不能使用 dynamic_cast 将一个指向 B 对象的指针转换为
//指向 A 对象的指针，因为 B
//是以 private 派生于 A
  pa1 = dynamic_cast<A*>(pb4);
  //正确

  C* pc1 = dynamic_cast<C*>(pb4);
  //正确的
  C& rc1 = dynamic_cast<C&>(*pb2);
  //错误，pb2 并不是真的就是一个 C，dynamic_cast 可以在指针转型失败时，
  //返回 null,但是没有 null reference
  //这种说法，因此当一个引用类型失败时便无法返回 null reference。
  //除了抛出异常，没有别的办法
  }
```

ITEM45:BOOL

```
//对 bool 类型的几个实现方式，但是都是由缺陷的

//1.typedef
// Option 1: typedef
//
typedef int bool;
const bool true  = 1;
const bool false = 0;
//方法不错，但是不能重载
// file f.h
void f( int  ); // ok
void f( bool ); // ok, redeclares the same function
// file f.cpp
void f( int  ) { /*...*/ }  // ok
void f( bool ) { /*...*/ }  // error, redefinition
//2.#define
// Option 2: #define
//这种方法有害，与上面有相同的问题，而且破坏了 #define
#define bool  int
#define true  1
#define false 0
//3.enum
// Option 3: enum
//
enum bool { false, true };
//允许重载，但是条件表达式不允许自动类型转换
bool b;
b = ( i == j );//不能正常工作
//4.class
class bool
{
public:
  bool();
  bool( int );                  // to enable conversions from
  bool& operator=( int ); //   conditional expressions
  //operator int();     // questionable!
  //operator void*(); // questionable!
private:
  unsigned char b_;
};
const bool true ( 1 );
const bool false( 0 );
//除了转换操作有问题外就可以工作，这些问题原因
```

```
//1.由于自动类型转换，重载解决了 bool 的冲突
//2.如果使用自动类型转换，bool 类型的数据会影响到函数重载，
//就像其他任何一个类型的隐式构造函数和自动类型转换一样
//3.不能转换与 int 或 void*相似的类型，bool 在条件中不能做测试
//4.如果不使用自动转换类型，那么 bool 类型将无法在使用在条件判断中
bool b;
/*...*/
if( b ) // error without an automatic conversion to
{           // something like int or void*
    /*...*/
}
```

ITEM46:转呼叫函数

```
//转呼叫函数对于将任务传递给其他函数或对象时很有用，
//尤其当他们被设计得很高效时
//评论下面的转呼叫函数，
// file f.cpp
//
#include "f.h"
/*...*/
bool f( X x )
{
   return g( x );

}
//有两个主题改进可使得这个函数高效，第一个总是采用，第二个要权衡
//1.传参时使用 const 的引用代替传值
//2.函数内联(避免函数内联。除非你真的有这个必要)
```

ITEM47:控制流

```
//从下面的代码中找出控制流的毛病
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std;

//    The following lines come from other header files.
//全局变量的出现应该早引起我们的警觉，使我们特别留意那些企图在它被初始化之前就使用它的代码
//在哥翻译单元之间的那些全局变量(包括类的静态变量)的初始化顺序未定义
char* itoa( int value, char* workArea, int radix );
extern int fileIdCounter;
//
//  使类的不变量检查自动化的辅助函数
//
template<class T>
inline void AAssert( T& p )
{
   static int localFileId = ++fileIdCounter;
   //这里出问题了，如果编译器恰好在初始化任何 AAssert<T>::localFileIdCounter
   //进行了初始化，那么还算好，
   //否则的话哲理的数值将会是 fileIdCounter 在初始化前其所占用的内存区中的内容
   if( !p.Invariant() )
   {
      cerr << "Invariant failed: file " << localFileId
           << ", " << typeid(p).name()
           << " at " << static_cast<void*>(&p) << endl;
      assert( false );
   }
}

template<class T>
class AInvariant
{
public:
   AInvariant( T& p ) : p_(p) { AAssert( p_ ); }
   ~AInvariant()                    { AAssert( p_ ); }
private:
   T& p_;
};
#define AINVARIANT_GUARD AInvariant<AIType> invariantChecker( *this )
//使用这些辅助函数是个很有意思的主意，这样可以使一个类在函数
```

```
//调用的前后自动进行不变量的检查。
  //只要简单的对其一个 AItype 的 typedef,然后再把 AINVARIANT_GUARD
  //做为成员函数的第一条语句就可以了
  //本质上这样不安全是不好的。

  //然而在下面的代码中，这种做法就很不幸的变得一点都不有趣。
  //主要原因是 Ainvariant 隐藏了对 assert()的
  //调用，而当持续在 non-debug 模式下被建立的时候，编译器会自动的删除掉 assert()。
  //------------------------------------------------------------
  template<class T>
  class Array : private ArrayBase, public Container
  {
    typedef Array AIType;
  public:
    Array( size_t startingSize = 10 )
    : Container( startingSize ),
      ArrayBase( Container::GetType() ),
          //这里构造函数的初始化表有两个潜在错误。
          //第一个错误或许不必称其为错误，但让其
          //留在代码里面又会形成一个扑簌迷离的障眼法。我们分两点说这个错误:
          //1.如果 GetType()是一个静态成员函数，或者是一个既不使用
          //this 指针又不受构造操作副作用(静态的使用计数)影响的成员函数，
          //那么这里只是不良的编码风格而已，仍然能正确运行
          //2.否则的话，我们就有麻烦了，非 virtual 的基类会被从左到右按
          //照他们被声明的顺序初始化因此这里，ArrayBase 会在 Container 之前初始化。
          //不幸的是，这意味着企图使用一个尚未初始化的 Container subobject 值成员


      used_(0),
      size_(startingSize),
      buffer_(new T[size_])
      //变量是以类定义的顺序初始化
    {
      AINVARIANT_GUARD;
      //效率方面的小问题，在这里 Invariant()函数没有必要被调用 2 次，
      //这是个小问题不会引起大的麻烦。
    }

    void Resize( size_t newSize )
    {//这是个严重的控制流方面的问题。
    //代码不是异常安全的。如果对 new[]的调用导致抛出一个异常的话，
    //那么不但当前的对象会处在一个
    //无效的状态，而且原来的 buffer 还会出现内存泄漏的情况。
    //因为所有指向他的指针都丢失了从而导致不能将其删除。
```

```
    AINVARIANT_GUARD;
    T* oldBuffer = buffer_;
    buffer_ = new T[newSize];
    copy( oldBuffer, oldBuffer+min(size_,newSize), buffer_ );
    delete[] oldBuffer;
    size_ = newSize;
}

string PrintSizes()
{
//其中 itoa()原型函数使用 buf 做为存放结果的地方。这段代码也有控制流方面的问题。
//我们无法估计最后那个返回语句中对表达式的求职顺序，
//因为对函数参数的操作顺序是没有明确规定的，其完全取决于特定的实现方案。
//最后那个返回语句所表现出来的问题的严重性在下面的这个语句得到更好的展示
    //return
            //operator+(
               //operator+(
                 // operator+( string("size = "),
                                // itoa(size_,buf,10) ) ,
                 // ", used = " ) ,
                //itoa(used_,buf,10) );

    //这里我们假设 size_的值为 10，used_的值为 5.如果最外面的 operator+()的第一个
    //参数先被求值的话那么结果将会是正确的 size=10,used=5,因为第一个 itoa()函数
    //存放在 buf 里面的结果会在第二个 itoa()函数复用 buf 之前就读出来使用。但如果
    //最外面的 operator+()的第二个参数先被求值的话，那么结果回事错误的
    //size=10,used=10,因为外层的那个 itoa()函数先被求值，但其结果会在被使用之前
    //就又被内层那个 itoa()函数毁掉了
    AINVARIANT_GUARD;
    char buf[30];
    return string("size = ") + itoa(size_,buf,10) +
              ", used = " + itoa(used_,buf,10);
}

bool Invariant()
{
    //对 Resize()的调用存在两个问题
    //1.这种情况，程序压根就不会正常工作，因为如果条件判断为真，那么 Resize()
    //会被调用，这又会导致立即再次调用 Invarient()，接着条件判断仍然会为真，然
    //后在调用 Resize()，这又会导致立即再次被调用 Invarient()，接着...你明白了。这
    //是个无法终止的递归调用
    //2.如果 AAssert()编写者处于效率方面考虑的错误报告的代码删除掉并取而代之以
     //assert(p->Invarient())那又会如何？其结果只会是这里的代码变得更可悲，因为在
     //assert()调用中加入了会产生副作用的代码
```

```
    //这意味着程序在 debug mode 和 release mode 两种不同的编译模式下产生的可执
    //行代码在执行时会有不同行为。即使没有上面第一点中说明的问题，这也是很不
    //好的，这意味着着 Array 对象会依据建立模式的不同而 Resize()不同的次数，使软
    //件测试人员过上地狱般的生活

    //绝不要在对 assert()的调用中加入有副作用的代码，
    //并且总是确认递归肯定互终止
    if( used_ > 0.9*size_ ) Resize( 2*size_ );
    return used_ <= size_;
  }
private:
  T*      buffer_;
  size_t used_, size_;
};

int f( int& x, int y = x ) { return x += y; }
//那第二个设置缺省值的参数无论如何都不算是一个合法的 C++用法，
//再一个理想的编译器下应该编译不通过。说这个用法不好还是因为编译器可以采用任
//意的顺序对函数参数求值，y 可能赶在 x 之前先被初始化。
int g( int& x )                 { return x /= 2; }

int main( int, char*[] )
{
  int i = 42;
  cout << "f(" << i << ") = " << f(i) << ", "
       << "g(" << i << ") = " << g(i) << endl;
  //这里还是对参数求值的顺序问题。由于没有确定 f(i)和 g(i)被求值的先后顺序，
  //因此显示出来的结果可能是错误的。不同的编译器可能会有不同的结果。
  Array<char> a(20);
  cout << a.PrintSizes() << endl;
}
```