Exceptional C++ 笔记

By Ken 2010-11-21

（一） 泛型程序设计与 C++标准库

- ✧ **原则：**永远不要 de-reference 无效的迭代器引用。使用迭代器时，注意以下四个主要问题：
  1) 有效值：迭代器可以解除引用吗？例如，写成“*e.end()”的代码永远是编程错误。
  2) 有效生存期：使用迭代器时，它仍然是有效的吗？或者这样思考：自从我们获得它依赖，是否有一些操作已经使它无效了？（向 vector/deque/list 等容器中插入或移除元素时，以前获得的迭代器均有可能失效。）
  3) 有效范围：一对迭代器是一个有效的范围吗？first 确实在 last 之前（或相等）吗？两个迭代器确实都指向同一个容器的元素吗？
  4) 非法的内置类型操作：例如，是否有代码企图去修改内置类型的临时对象，正如上述的“--e.end()”。（编译器通常会捕获这种错误。对于类类型的迭代器，为了语法的方便，库的作者通常会选择允许这种操作。）
- ✧ **LSP、SRP、OCP、DIP 和 ISP 并称面向对象的五大特征。在设计 OO 程序时，永远牢记并遵循这五大法则。**
- ✧ **注意：由于模板构造函数永远不是拷贝构造函数，因此这种模板的存在不能禁止拷贝构造函数的隐式声明。模板构造函数和其他构造函数（包括拷贝构造函数）都参与重载解析。而且如果拷贝构造函数比其他构造函数提供了更好的匹配，那么可能用它来拷贝一个对象。**
- ✧ **原则：**永远不要使异常安全性成为时候才采取的措施。异常安全性影响着一个类的设计。它永远不是一个实现细节。
- ✧ **在创建自己的类时，使用成员函数模板会有很好的效果（考虑使用成员模板函数）。**
- ✧ **原则：**选择以 const&方式而不是传值的方式来传递对象。对于不会改变的值应预先计算，而不是多此一举地重新创建对象。

  for(/*…*/; i != e.end();/*…*/) ← save e.end() in a temporary var.
- ✧ **原则：**选择前置递增，只有打算使用初始值时才使用后置递增。
- ✧ **原则：**为了保持一致性，应该始终用前置递增来实现后置递增，否则你的用户将获得令人惊讶的（而且常常是令人不愉快的）结果。
- ✧ **原则：**注意由隐式转换创建的隐藏临时对象。避免这个问题的一个好办法是尽可能地创建显示的（explicit）构造函数和避免编写转换操作符。
- ✧ **原则：**重用代码（尤其是标准库的代码），而不是手工编写自己的代码，这样更快，更容易也更安全。

（二） 异常安全性问题与技术

- ✧ **原则：**如果一个函数不打算去处理（或转换或者故意吸收）异常，那么它应当允许异常向上传递给能够处理该异常的调用者。
- ✧ **原则：**总是应该设法这样构建代码，即使在异常存在的情况下，资源仍然能够被正确的释放，数据也能处于一致状态。

- **原则：**优先选择内聚性（cohesion）。设法使每块代码（每个模块、每个类、每个函数）有单一的、定义良好的职责。[更改（mutator fuction）不应当以传值方式返回 T 对象（比如 list.pop()函数，其实是在试图让一个函数同时做两件事情。）]
- 常见错误：“异常不安全”和“拙劣的设计”通常是紧密相关的。如果一块代码不是异常安全的，那一般没有太大问题，可以被简单地纠正。但是如果一块代码由于内在的设计问题而不能被编写成异常安全的，那几乎总是拙劣设计的表现。例 1：带有两个不同职责的函数是难于编写成异常安全的。例 2：必须检测自赋值的拷贝复制操作符不可能是异常安全的。
- 一个异常安全的 Stack 实现。

```
#include <cstddef>
namespace STACK
{
template<typename T1, typename T2>
void construct( T1* p, const T2& value )
{
   new (p) T1( value );
}

template<typename T>
void destroy( T* p )
{
   p->~T();
}

template<typename FwdIter>
void destroy( FwdIter first, FwdIter last )
{
   while( first != last )
   {
      destroy( first );
      ++first;
   }
}

template<typename T>
class StackImpl
{
public:
   StackImpl( size_t size=0 )
      : v_( static_cast<T*>( size==0 ? 0 : operator new(sizeof(T)*size) ) ),
      vsize_( size ),
      vused_( 0 )
   {
   }

   ~StackImpl()
   {
      destroy( v_, v_ + vused_ );
      operator delete( v_ );
   }

   void Swap( StackImpl& other ) throw()
   {
      swap( v_, other.v_ );
      swap( vsize_, other.vsize_ );
      swap( vused_, other.vused_ );
   }

   T* v_;
   size_t vsize_;
   size_t vused_;

private:
   StackImpl( const StackImpl& );
```

```
    StackImpl& operator=( const StackImpl& );
};

template<typename T>
void swap( T& a, T& b )
{
    T temp( a );
    a = b;
    b = temp;
}

template< typename T>
class Stack
{
public:
    Stack( size_t size=0 )
        : impl_(size)
    {
    }

    ~Stack()
    {
    }

    Stack( const Stack& other )
        : impl_(other.impl_.vused_)
    {
        while( impl_.vused_ < other.impl_.vused_ )
        {
            construct( impl_.v_ + impl_.vused_, other.impl_.v_[impl_.vused_]);
            ++impl_.vused_;
        }
    }

    Stack& operator=( const Stack& other )
    {
        Stack temp( other );
        impl_.Swap( temp.impl_ );
        return *this;
    }

    size_t Count( void ) const
    {
        return impl_.vused_;
    }

    void Push( const T& t )
    {
        if( impl_.vused_ == impl_.vsize_ )
        {
            Stack temp( impl_.vsize_*2+1 );
            while( temp.Count() < impl_.vused_ )
            {
                temp.Push( impl_.v_[temp.Count()] );
            }
            temp.Push( t );
            impl_.Swap( temp.impl_ );
        }
        else
        else
        {
            construct( impl_.v_ + impl_.vused_, t );
            ++impl_.vused_;
        }
    }

    T& Top( void ) const
    {
        if( impl_.vused_ == 0 )
```

```
        {
            throw "empty stack";
        }
        return impl_.v_[ impl_.vused_ - 1 ];
    }

    void Pop( void )
    {
        if( impl_.vused_ == 0 )
        {
            throw "pop from emtpy stack";
        }
        else
        {
            --impl_.vused_;
            destroy( impl_.vused_ );
        }
    }

  private:
    StackImpl<T> impl_;
};
} //namespace STACK
```

- 在绝对必要的时候才使用继承来代替包含，如一下情形：
  - 需要访问这个类的保护数据成员。
  - 需要重载虚函数。
  - 这个对象需要在其他基本子对象之前被构造。
- 在编写一个模板化类的时候，尤其是类似于泛型容器这样可能被广泛使用的东西，应该总是自问一个至关重要的问题：我们的类具有怎样的可重用性？或者以另外一种方式提出：我们给这个类的用户强加了哪些约束，这些约束是否过度地限制了类的用户所能做的合理的操作？
- Stack 应当为他的成员函数提供异常规范吗？
  - 不，因为我们 Stack 的创建者并不知道 T 的最够的信息，而且即使我们知道足够的信息，我们大概也不愿意去这么做。在原则上，对任何泛型容器都是一样的。
  - 即使像 Stack<T>::Count()这样的简单的函数，也不要为他提供 throw()。有两个很好的理由：（a）万一将来你想要将内部的实现改为能够抛出异常的形式，那么编写 throw()就会限制你这样做。放松异常规范总是要冒破坏已存在的客户程序的危险，因此你的类对变化也就具有更大的内在抵抗力，也就更脆弱。[对虚函数编写 throw()也会使类具有更小的可扩展性，因为它大大地限制了那些想要从你的类派生的人。它可能是说的通的，但做这样的决定要仔细地考虑。]；（b）不管是否抛出异常，异常规范都会产生性能上的开销。

✧ **原则：**遵守规范的异常安全性原则：（1）永远不要允许异常从析构函数、overloaded 操作符 operator delete()或 operator delete[]()中抛出；编写每个析构函数或内存释放函数，好像有“throw()“这样的异常规范一样。（2）在每个函数中，将所有可能抛出异常的代码以及能够安全地处理此类工作的代码单独放置到一边。只有确信实际的工作已经成功时，才使用无异常抛出的操作来修改（和清除）程序的状态。（3）使用“初始化就是获得资源”这一习惯法来隔离资源所有权和资源管理（auto_ptr、shared_ptr）。

- **在析构函数中抛出异常以及他们有害的原因。**

```
template<typename FwdIter>
void destroy( FwdIter first, FwdIter last )
```

```
{
   while( first != last )
   {
      destroy( first );
      ++first;
   }
}
```

➔（2）

```
template<typename FwdIter>
void destroy( FwdIter first, FwdIter last )
{
   while( first != last )
   {
      try
      {
         destroy( first );
      }
      catch(…)
      {
         /*what to do ?*/
      }
      ++first;
   }
}
```

如果 destroy 抛出异常，我们有可能写出（2）这样的代码。注释/*what to do*/是最棘手之处。实际上只有三种选择：catch 代码块重新抛出这个异常；catch 代码块通过抛出其它异常来转换这个异常；catch 代码块不抛出任何异常，并且继续这个循环。

（1）如果 catch 代码块重新抛出这个异常，那么 destroy 函数就能够很好地符合异常中立的要求，因为它确实允许任何 T 异常正常地传递出去。然而，它仍然不符合即使异常发生也没有任何资源泄漏的安全性要求。由于 destroy 函数没有办法用信号标志还有多少个对象没有被成功地摧毁，因此那些对象永远不能被正确地销毁，由此与它们相关联的资源将 不可避免地会泄漏。无疑，这不是好办法。

（2）如果 catch 块代码块通过抛出其他异常来转换这个异常，很明显，这既不符合异常中立要求也不符合异常安全性要求。

（3）如果 catch 代码块不抛出或者不重新抛出任何异常，那么 destroy 函数就能很好地符合“即使异常抛出也没有任何资源泄漏”的安全性要求。然而，很明显它不符合允许 T 异常传递出去的中立要求，因为异常被吸收和忽略了。

（*）曾经有人建议这个函数应当捕获这个异常，且“保存”而继续销毁其他对象然后在结束时重新抛出这个异常。那并不是一个什么好的解决方案——例如，它不能正确地处理由多个 T 析构函数抛出多个异常。（即使可以保存所有的异常到最后，最终也只能抛出其中一个来结束程序，其他异常都被悄无声息地吸收了。）你可能正在考虑其他的可选方案，但是相信我，所有都归结为在某处编写类似的代码，因为有一组对象并且它们都需要被销毁。

（4）如果析构函数可以抛出异常，那么 new[]和 delete[]操作符都不可能被编写成异常和异常中立的。

（5）异常用来报告构造失败（包括数组和数组-new[]构造失败）是很有用的。

（二） 类的设计与继承

- ✧ 选择编写“a op=b;”而不是“a=a op b;”（这里的 op 代表任意操作符）。这样更清楚，并且通常更有效率。因为 a op=b 是直接对其左边的对象进行操作，并且只返回引用，而不是一个临时对象。而 a=a op b 必须返回一个临时对象。
- ✧ **原则：**如果提供一个操作符的一个版本（例如，operator+）,那么应该始终提供相同操作符的复制版本（例如，operator+=）,并且根据后者来实现前者。同样，始终保持 op 和 op=之间的天然关系。

✧ **原则：**操作符是成员函数还是非成员函数，应按照如下原则：

- =，()，[]和->必须是成员函数。new/new[]，delete/delete[]永远是 static 成员函数。
- >>和<<始终是非成员函数（如果必要可以是友元函数）。
- 对于其他操作符：
  如果函数必须是虚拟的，那么为该类添加一个虚函数来提供此虚行为，并且该操作符函数根据此虚成员函数来实现，那么使它成为一个成员函数；如果它需要向它的最左边的参数进行类型转换，那么使它成为一个非成员函数（如果必要可以是友元函数）；如果它可以只使用类的公共接口来实现，那么使它成为非成员函数和非友元函数；否则使它成为成员函数。

✧ **原则：**始终从 operator>>和 operator<<返回流引用。

```
class Complex   ← Initial edition
{
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;
};

class Complex ← Corrected edition
{
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++( void )
```

```
    {
        ++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 temp( lhs );
    ret += rhs;
    return ret;
}

ostream& operator<<( ostream& os, const Complex& c )
{
    return c.Print( os ); ← we can implement virtual behavior by this way.
}
```

- ✧ **原则：**将基类的析构函数编写成虚析构函数（除非你能肯定无论何时，都没有人试图通过指向基类的指针来删除派生类对象）。
- ✧ **原则：**当提供一个与继承函数（inherited function）具有相同名字的函数时，如果不想屏蔽继承函数，那么应使用 using 声明来确保继承函数进入作用域（确保继承函数不被隐藏）。
- ✧ **原则：**永远不要改变重载继承函数的默认参数。
- ✧ 重载解析是由静态类型而不是动态类型来决定的。和重载函数一样，默认参数是从对象的静态类型取得。

```
class Base{
public:
    virtual ~Base();
    virtual void g( int i = 10 ){ cout << “Base::” << i << endl; }
};
class Derived: public Base
{
public:
   virtual void g( int i = 20 ){ cout << “Derived::” << i << endl; }
};
Base* pd = new D;
pd->g();    // Derived::10, coz pd’s static type is Base, pd’s dynamic type is Derived
Base b;
Dervied d;
b.g(); // Base::10, b’s static type is Base
d.g(); // Derived::20, d’s static type is Derived
```

- ✧ 常见错误：永远不要使用公有继承，除非是要建模真正的 Liskov IS-A 和 WORKS-LIKE-A 关系，所有被继承的成员函数必须不再有更多的要求，也不能有更少的承诺。
- ✧ **原则：**永远不要为了重用（基类的）代码而进行共有继承。公有继承的目的是（被多态性地使用基类对象的代码）重用。
- ✧ **原则：**当建模“照此实现”关系时，始终选择成员关系/聚集，而不要选择继承。只有当继承是绝对必要时才使用私有继承——也就是说，当需要访问保护成员或者需要重载虚函数的时候。永远不要为了代码重用而使用公有继承。
- ✧ **原则：**避免使用公有虚函数；相反，选择使用 Template Method 模式。
- ✧ 原则：对于被广泛使用的类，优先选择使用编译器防火墙用法（Pimpl 习惯用法）来隐藏实现细节。使用声明为“struct XxxxImpl* pimpl_;”的不透明指针（指向已声明但未定义的类）来存储私有成员（包括状态变量和成员函数）。例如，“class Map{ private: struct MapImpl* pimpl_;};”。

（三） 编译器防火墙和 Pimpl 习惯用法

- ✧ C++最强大的地方在于，它支持两种功能强大的抽象方法：面向对象编程和泛型编程。两者本质上都是用于帮助管理依存性从而管理复杂性的工具。可以讲，所有常见的面向对象/泛型概念——包括封装性、多态性、类型独立，以及所有的设计模式，都是用来“描述在一个软件系统中，如果通过管理代码的相互依存性来管理复杂性”的方式而已。
- ✧ **原则：**永远不要包含（#include）不必要的头文件（这样做会严重地降低 build 时间，尤其是在一个常用头文件中包含太多其他头文件时）。
- ✧ **原则：**当流的前置声明足够时，优先选用#include<iosfwd>。
- ✧ **原则：**当前置声明足够时，永远不要包含（#include）头文件。
- ✧ 问题：在 C++中，当类中的任何东西改变时，这个类的所有用户代码都必须被重新编译。为了降低这些依存性，一种常用的技术是使用非透明指针来隐藏一些实现细节。

```
class X
{
public:
  /*…公有成员…*/
protected:
  /*…保护成员…*/
private:
  /*…私有成员…*/
  struct XImpl* pimpl_;   // 非透明指针，指向已前置声明的类
};
```

1) 什么东西应该放入 XImpl 中？有四种方式。

   1. 将所有私有数据成员（但不是成员函数）放入 XImpl 中。

   （分数：6/10）：将所有私有数据成员放入 XImpl 中。这是一个好的开端，因为现在我们可以前置声明任何只作为数据成员的类（而不是包含这个类的实际声明，这将使客户代码依赖这个类）。尽管如此，我们通常可以做的更好。

   2. 将所有私有成员放入 XImpl 中。

   （分数：10/10）：将所有私有成员放入 XImpl 中。现在（几乎）是 Sutter 的惯用法。毕竟，在 C++中，短语“客户代码不应当也不会担心这些部分”就相当

于“私有”，而私有总是意味着隐藏。

有一些警告，这是上面使用“几乎”的理由：

- 在 Pimpl 中不能隐藏虚函数，即使这些虚函数是私有的。如果虚函数改写了从基类继承的虚函数，那么它必须出现在实际的派生类中。如果虚函数没有被继承，那么为了可以被更进一步派生的类改写，它仍然必须出现在可视类中。

  无论如何，虚函数总应该是 private 的，除非派生类需要调用基类的虚函数，这时候虚函数应该是 protected 的。
- 如果 Pimpl 中的函数需要使用可视函数，那么它们可能要求有一个指向可视对象的“回指指针”，这将增加间接性（为了方便，这样的回指指针在 PeerDirect 公司通常命名为 self_）。
- 通常一个最好的折中办法是使用选择 2，另外只将那些需要被私有函数调用的非私有函数放入 XImpl 中。

3. 将所有的私有成员和保护成员放入 XImpl 中。

（分数：0/10）：采用这个额外的步骤来包含保护成员确实是一个错误。保护成员永远不应该放入 Pimpl 中，因为将它们放入只会削弱他们。毕竟，保护成员是专门为了被派生类看见和使用而存在的，因此如果派生类不恩那个看见并使用他们，那么它们几乎是没有用的。

4. 使 XImpl 完全成为与原来 X 相似的类，而且将 X 编写为只由简单的前置函数组成的公有接口（句柄/实体习惯用法的一种变体）。

（分数：10/10 在某些特定的情况下）：这在一些特定情况下是有用的而且可以避免使用回指指针，因为在 Pimpl 类中的所有服务都是可用的。这种方法的主要缺陷是，它使可视类不再具有意义，无论是作为基类还是派生类。

2) XImpl 要求有一个指向 X 对象的回指指针吗？

有些时候需要。毕竟，我们做的是：为了隐藏其中的一部分，我们（有点人工地）将每个对象分成了两个部分。如果 Pimpl 内部经常必须调用可视类内的函数（调用可视类的共有函数或者虚函数），那么将这个问题影响降至最小的一种办法是对于这些相关的函数明智地使用选择 4——也就是说，实现选择 2，然后将被私有函数使用的任何非私有函数放入 Pimpl 中。

✧ 一般来说，处理特定类分配性能问题的一种正确的方法是为这个类提供一个类特定的 operator new() 操作符，并且使用固定大小的分配器，因为固定大小的分配器比通用分配器效率更高。

✧ 原则：避免使用内联（inline）或者琐细的优化，除非性能分析证明这是必要的。

（四） 名字查找、名字空间和接口规则

✧ 在如下的代码中，将调用哪些函数？为什么？分析其中涵义。

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

namespace B
```

```
{
  void f( int i )
  {
    f( i ); //  （1）调用哪个 f()?
  }

  void g( A::X x )
  {
    g( x ); //  （2）调用哪个 g()?
  }

  void h( A::Y y )
  {
    h( y ); //  （3）调用哪个 h()?
  }
}
  void h( A::Y y )
  {
    h( y ); //  调用哪个 h()?
  }
}
```

（1） 这个调用将使用无限递归的方式调用其自身。理由是，惟一可见的 f()是 B::f()自身。原型为 f(int)的还有两外一个函数，也就是名字空间 A 中的函数 f(int)。如果 B 已经写了“using namespace A;”或者“using A::f”，那么当查找 f(int)时，A::f(int)将作为一个候选者，而且 f(i)调用是 A::f(int)还是 B::f(int)将模棱两可。然而，由于名字空间 B 没有将 A::f(int)引入其作用域内，只有 B::f(int)可以考虑，因此无疑是调用 B::f(int)。

（2） 这个调用 A::g(X)还是 B::g(X)是模棱两可的。程序员必须使用适当的名字空间限制这个调用，才能调用到想要的 g()函数。这里涉及到 Koening 查找规则：如果提供了一个类类型的函数参数（这里是 X、A::X 类型），那么查找正确的函数名时，编译器将考虑包含参数类型的名字空间内匹配的函数名。

（3） h()将以无限递归的方式调用其自身。虽然 B::h()的原型中提到名字空间 A 内的类型，但是这并不影响名字查找规则，因为在名字空间 A 内没有函数与 h(A::Y)的名字和原型相配。

（4） 简单地说，名字空间 B 内的代码的意义受完全分离的名字空间 A 内声明的函数的影响，即使 B 只是简单地提及名字空间 A 内的一个类型，而且没有使用任何“using”指令。

✧ 接口规则：对于类 X，所有函数，包括自由函数，只要符合如下条件，逻辑上都是 X 的一部分，因为他们构成了 X 接口的一部分。

- ■ “提及”X。
- ■ 与 X“一起提供”

✧ 虽然接口规则规定，成员函数和非成员函数逻辑上都可以是类的“一部分”，但是它并没有声称成员函数和非成员函数是等价的。例如，成员函数自动地拥有对类内部完全的访问权限，而非成员函数只有在被声明为友元函数时才拥有这样的访问权

限。同样地，对于名字查找（包括 Koeing）查找，C++语言有意规定，成员函数被认为比非成员函数与类具有更强的关系。如下面的例子，当 namespace B 换成 class B 时，f 的调用没有任何二义性。当编译器发现一个名为 f()的成员函数，它将不会试图使用 Koening 查找规则来发现自由函数。

```
namespace A
{
   class X{};
   void f( X );
}
class B
{
void f( A::X );
void g( A::X param )
{
   f( param ); // OK: B::f, 无二义性
}
};
```

✧ 在接口规则中，“一起提供”的一种有用的诠释方式是“出现在相同的头文件和/或名字空间中。”如果函数和类出现在同一头文件中，那么根据依存性，它是类的“一部分”。如果函数和类出现在相同的名字空间中，那个从对象使用和名字查找的意义上来说，它是类的“一部分”。

✧ 一般来说，如果 A 和 B 是类，f(A, B)是一个自由函数，那么，

- 如果 A 和 f 被一起提供，那么 f 是 A 的一部分，因此 A 依赖于 B。
- 如果 B 和 f 被一起提供，那么 f 是 B 的一部分，因此 B 依赖于 A。
- 如果 A、B 和 f 被一起提供，那么 f 同时是 A 和 B 的一部分，因此 A 和 B 相互依赖。

✧ 如果 A 和 B 是类，A::g(B) 是 A 的一个成语函数，那么，

- 由于 A::g(B)存在，很明显，A 始终依赖于 B。迄今为止，没有任何疑问。
- 如果 A 和 B 被一起提供，那么 A::g(B)和 B 当然是被一起提供的。那么由于 A::g(B)“提及”B 并与 B“一起提供”，那么根据接口规则，A::g(B)是 B 的一部分是理所当然的。两外由于 A::g(B)使用一个（隐式的）A*参数，因此 B 依赖于 A。由于 A 也依赖于 B，因此这意味着 A 和 B 相互依赖。这似乎可以看作是“一个类的成员函数同时也是另外一个类的一部分”这种情况的扩展，但是只有在 A 和 B 也被一起提供时才是正确的。

✧ 名字隐藏，请看下面的例子：

```
struct B
{
   int f( int );
   int f( double );
   int g( int );
};

struct D: public B
{
```

```
private:
   int g( std::string, bool );
};
D d;
d.(i);   //正确，表示 B::f( int )
d.g(i); //  错误：g 有两个参数
```

d. g(i)通不过编译的原因：当我们在派生类D中声明一个名为g的函数时，它将自动地隐藏所有直接和间接基类中具有相同名字的所有函数。To see what's really going on, let's look in a little more detail at what the compiler does when it encounters the function call d.g(i). First, it looks in the immediate scope, in this case the scope of class D, and makes a list of all functions it can find that are named g (regardless of whether they're accessible or even take the right number of parameters). Only if it doesn't find any at all does it then continue "outward" into the next enclosing scope and repeat. In this case, the scope of the base class B. Until it eventually either runs out of scopes without having found a function with the right name or else finds a scope that contains at least one candidate function. If a scope is found that has one or more candidate functions, the compiler then stops searching and works with the candidates that it's found, performing overload resolution and then applying access rules.

[There are very good reasons why the language must work this way.For example, one might think that if none of the functions found in an inner scope were usable, then it could be okay to let the compiler start searching further enclosing scopes. That would, however, produce surprising results in some cases (consider the case in which there's a function that would be an exact match in an outer scope, but there's a function in an inner scope that's a close match, requiring only a few parameter conversions). Or, one might think that the compiler should just make a list of all functions with the required name in all scopes and then perform overload resolution across scopes. But, alas, that too has its pitfalls (consider that a member function ought to be preferred over a global function, rather than result in a possible ambiguity).]

✧ 下面这个例子能正确编译吗？尽可能完整地回答。

```
namespace N{ class C{}; }
int operator+( int i, N::C )
{
   return i+1;
}

#include <numeric>
//#include <vector>
int main()
{
   N::C a(10);
   std::accumulate(a.begin(), a.end(), 0);
}
```

这个例子能否编译完全取决于标准头文件 numeric 的版本是否：a)声明了一个 operator+()（任何 operator+()，无论是否适、是否可访问）；或者 b)包含任何其它做

同样事情的标准头文件。与标准 C 不同，标准 C++没有指定哪些标准头文件相互包含关系。因此当包含 numeric 头文件时，你可能同时包含定义了几个 operator+()函数的头文件，也可能没有。当前流行的一些编译器，有些能够编译这个例子代码（g++可以），但如果增加一行#include<vector>就都不能编译了。

这个例子正确的解答应该是将我们的 operator+()函数放到它真正属于而且它应当一开始就被放入的地方：namespace N。

- ✧ 如果一个操作，甚至是自由函数（尤其是操作符）提及一个类，而且打算构成类的接口的一部分，那么请确保它与这个类一起提供——也就是说要将它放入与这个类相同的名字空间中。无论如何，确保类和接口在一起，当其他人试图使用你的类时，这是避免复杂的名字查找问题的一个简单的办法。
- ✧ **原则：**正确地使用名字空间。如果将一个类放入一个名字空间内，那么一定将所有父的辅助函数和操作符也放入相同的名字空间内。如果不这么做，你将在代码中发现令人惊奇的结果。

（五） 内存管理

- ✧ **C++的内存区域**

内存区域	特性和对象生存期
常量数据 （Const Data）	常量数据区存储字符串字面值和其他在编译时就知道值的数据。这个区域不保存类类型对象。 该区域内的所有数据在这个程序的整个生存期内都是可用的，而且所有数据都是只读的。如果试图修改这些数据，那么结果将难以预料。这甚至在某种程度上是由于编译器对底层存储格式任意最优化的结果。例如，一个特定的编译器的可选优化可能选择将字符串字面值存储在重叠对象内。
栈 （Stack）	栈存储自动变量。对象在定义点被立即构造，并且在同一作用域结束点被立即销毁，因此程序员没有机会直接操作已经分配但未初始化的栈空间（除非蓄意地使用显式的析构函数和定位 new 来干预）。 栈内存分配通常要比动态存储空间分配（堆和自由存储）快得多，因为每次栈内存分配只涉及到一个栈指针的自增操作，而不是更复杂的管理。
自由存储 （Free Store）	自由存储是两种动态内存区域之一，它有 new/delete 来分配/释放。对象生存期可以小于存储空间的分配时间，也就是说自由存储对象被分配内存，而不用立即被初始化；自由存储区域对象被销毁时，不用立即释放内存空间。在存储空间被分配但出于对象生存期之外的这段时间里，这块存储空间可以通过一个 void*指针来访问和操作，但是任何原始对象的非静态成员或成员函数都不能被访问，不能获得它们的地址，或者进行其他方式的操作。
堆 （Heap）	堆是另外一种动态内存区域，它由 malloc/free 函数及其变体来分配/释放。注意当一个特定的编译器根据 malloc/free 函数来实现默认的全局操作符 new 和 delete 时，堆和自由存储是不同的，在一个区域内分配的内存不可能在另外一个区域内被安全地释放。 通过定位 new 构造操作和显式的析构操作，从堆中分配的内存可以被用于类类型的对象。如果这样用了，关于自由存储对象生存期的注意事项在这里也同样适用。
全局或静态 （Global/Static）	全局或静态变量和对象在程序启动时就被分配了存储空间，但直到程序执行时才可以被初始化。例如，在一个函数内的静态变量只有在程序执

	行第一次经过它的定义点时才被初始化。 跨越多个编译单元的全局变量的初始化顺序是不确定的，在管理全局对象（包括类静态成员变量）之间的依存性时需要特别注意。正如通常一样，未初始化的原始对象存储空间可以通过一个 void*指针来访问和操作，但是非静态成员或成员函数不能在这个对象的实际生存期之外被使用或引用。

- ✧ **原则：**优先使用自由存储（new/delete）；避免使用堆（malloc/free）。
- ✧ **原则：**如果提供了类的专有的 new（或者 new[]）和类专有的 delete（或 delete[]）中的任一个时，一定同时提供两者。
- ✧ **原则：**必须一直要明确地声明操作符 new()和操作符 delete()为静态成员函数。它们永远都不是非静态的成员函数。
- ✧ **原则：**永远不要多态地处理数组（例如，语言要求传递给 delete[]操作符的类型必须同它的动态类型保持一致。B* pb = new D[10]; delete[] pb;会导致未定义行为。）
- ✧ **原则：**使用 vector<>或者 deque<>，不要使用数组。
- ✧ **原则：**永远不要写忽略返回值的代码。
- ✧ **原则：**永远不要把 auto_ptr 放入标准容器。
- ✧ **警告：**使用一般指针成员变量而不是一个 auto_ptr 成员时，必须为这些类提供自己的析构函数、拷贝构造函数和拷贝赋值操作符（即使你通过将其标志为 private 并且不定义它们，而使它们无效），因为默认的将会进行错误操作。
- ✧ const auto_ptr 永远不会失去所有权。

```
const auto_ptr<T> pt1( new T ); //  使 pt1 为 const，能保证永远不会复制
                                //到另一个，因而可以保证不失去所有权
auto_ptr<T> pt2( pt1 ); //  非法
auto_ptr<T> pt3 ;
pt3 = pt1 ;                 //非法
pt1.release() ;             //非法
pt1.reset( new T ) ;        //非法
```

（六） 缺陷、陷阱和错误习惯用法

- ✧

```
T& T::operator=(    const T& other )
{
  if( this != &other) //  要讨论的测试句
  {
    //…
  }
}
```

为了防止自身赋值，“this != &other”测试是一种常用的编程实践。但是此判断对于这一目的是否必要且充分呢？简短的答案：In most cases, it's bad only if, without the test, self-assignment is not handled correctly。尽管这种检测不能防止所有可能的误用，但是实际上即使只用于优化目的也是很好的。过去一些人曾认为，多重继承影响了这个问题的正确性。这种说法不正确，只能混淆视听。

如果 T::operator=()采用 create-a-temporary-and-swap 手法实现的，那么它

将是强异常安全并且不需要进行这种自赋值检测。

- ✧ **原则：**不要编写要靠检测自身赋值才能保证正常运行的拷贝赋值操作符。用 create-a-temporary-and-swap 手法来实现的拷贝赋值操作符是强异常安全和自赋值安全的。

  这个原则对也不对。事实上，不进行自身赋值检测可能引起两种潜在的性能损失。

  - ■ 如果能够对自身赋值进行检查，那么就能够完全地优化赋值操作。
  - ■ 异常安全的代码效率会降低。

  实践中，如果经常需要进行自身赋值（在大多数程序里很少发生），并且对于应用程序来说，如果通过删除自身赋值中不必要的工作能极大地提高性能（在大多数程序中更少见），那就检测自身赋值好了。
- ✧ **原则：**使用自身赋值检测作为优化手段以去除不必要的工作，总是可以的。
- ✧ **原则：**通常情况下，避免编写自动转换的代码，无论是通过转换操作符还是单参数的非显式构造函数，都是好主意。主要原因在于，隐式转换通常不安全：
  - ■ 隐式转换有可能干扰重载解析
  - ■ 隐式转换有可能是“错误”的代码通过编译。
- ✧ **原则：**不要编写转换操作符。不要写非显式的构造函数。
- ✧ **以下来自 PeerDirect 公司的编码标准：**
  - ■ 以 T& T::operator=(const T&)形式申明拷贝赋值操作。
  - ■ 不要返回 const T&。有些时候这样做也许很好，因为它阻止了“(a=b)=c”这种用法。但这也意味着不能可移植地把 T 对象放入标准库的容器中，因为标准容器要求赋值操作必须返回普通的 T&。

- ✧ **原则：**优先考虑提供不抛出异常的 Swap()函数，并且使用如下的拷贝构造函数来实现拷贝赋值：

```
T& T::operator=( const T& other )
{
  T temp( other );
  Swap( temp );
  return *this;
}
```

- ✧ **原则：**不要使用“先一个显示的析构函数，再紧跟 new 语句”这种技巧来完成拷贝构造函数实现拷贝赋值的目的，即使这个技巧每隔三个月都会突然出现在新闻组中，请不要这样编写：

```
T& T::operator=( const T& other)
{
  if( this != &other )
  {
    this->~T();              // evil
    new(this) T(other);    // evil
  }
  return *this;
```

}

（七） 其他主题

- **分析 T t(u) 和 T t = u; 的区别：**
  - T t(u);是直接初始化。它调用 T::T(u)从 u 直接初始化变量 t。
  - T t = u;是拷贝初始化。可能调用了其它函数后，它总是使用 T 的拷贝构造函数初始化变量 t。如果 u 是 T 类型，这同写“T t(u)”是一样的意思，仅仅调用了拷贝构造函数；如果 u 是其他类型变量，它相当于“T t(T(u))”——也就是说，u 开始被转换成一个临时的 T 对象，然后 t 是通过它的拷贝构造函数构造的。注意，在这种情况下编译器允许优化“多余的”拷贝，然后把它直接转变成直接初始化形式[也就是使它同“T t(u)”相同]。如果编译器这样做了，拷贝构造函数必须仍然是可以访问的。但是如果拷贝构造函数有副作用，那么你有可能得不到所期望的值，因为拷贝构造函数副作用可能发生也可能不发生，这依赖于编译器是否执行优化。
- **原则：**在可能用到的地方宁可使用“T t(u);”形式也不使用“T t = u”形式；前者适用于后者适用的任何地方，并且还有更多的优点——例如，它可以带多个参数。
- **原则：**不要申明 const 的按值传递的函数参数。
- **原则：**当非内置返回类型使用按值返回时，最好返回一个 const 值。
- **记住：**Remember that the correct use of mutable is a key part of const-correctness. If your class contains a member that could change even for const objects and operations, make that member mutable. That way, you will be able to write your class's const member functions easily and correctly, and users of your class will be able to correctly create and use const and non-const objects of your class's type.
- **原则：不要转换掉 const。使用 mutable 代替。**
- **解析下面的类型转换问题：**

```
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;
```

1. A* pa = (A*)&a2;
   这可能表达成一个新风格转换。最接近的候选为 const_cast，但因为 a2 是一个 const 对象，结果是未定义的。
2. unsigned char* puc = static_cast<unsigned char*>(&c);
   signed char* psc = static_cast<signed char*>(&c);
   错误。我们必须对以上两种情况使用 reinterpret_cast。这样做开始有可能让你觉得惊讶，但是原因是 char、signed char 和 unsigned char 是三种截然不同的类型。即使在他们之间有隐式的转换，他们

也是毫无联系的，所以它们的指针也毫无联系。

3. A* pa2 = const_cast<A*>(&ra2);

错误。如果指针被用来改写对象。这样将会产生未定义的状态，因为 a2 真的是一个 const 对象。要明白问题的原因，可以这样想，编译器知道 a2 是作为一个 const 对象被创建的，并利用这个信息把它存储在只读的内存中，这是一种优化。对于这样的对象转换掉 const 显然很危险。

- ✧ **原则：**不要使用全局或者静态变量。如果你必须使用静态或全局对象，一定要注意初始化顺序。
- ✧ **原则：**在构造函数的初始化列表中的基类应该按照其在类声明中出现的顺序列出。
- ✧ **原则：**在构造函数的初始化列表中的数据成员应该按照其在类定义中的出现顺序列出。
- ✧ **原则：**永远不要写出依赖于函数参数的计算顺序的代码。