条款 10.STL 中的 Queue 将弹出元素分为两步——top,pop；目的之一是为了避免削弱函数的异常安全性

A.条款 14.很多实现中都将类分成接口和具体实现，比如 Stack 和 Stack_impl，主要是为了进行异常的分别单独处理，将异常处理放在 impl 中。同时在此例中还有个作用是，将内存管理和对象的构造/析构这两个方面分离开

B.条款 23.如果里面的 impl 声明为指针，则是由于 impl 指向的对象存放具体实现的私有成员函数和私有数据成员，为了方便对这些成员更改时，客户代码（使用这个类的代码）不需要重新编译。

条款 16.引入异常的初衷之一就是为了使构造函数和析构函数能够报告失败（因为这些函数不能返回一个值）（注：为什么构造函数没有返回值，一种说法是为了避免语法混乱，如 C c = C()，如果有返回值此例不成立，同时也可以说明返回值是对象本身）

条款 19.

```
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;
}
```

在某些设计中不用返回值返回某个值，而是通过参数传递一个引用来获得这个值，其中一个目的是在通过返回值返回一个对象值时，一般会带来拷贝构造和拷贝赋值两个操作，有可能会出错，但是这个已经出了函数管理的范围，现在是由函数来保证正确的获得这个值（注：一个更好的解法，到达 commit-or-rollback 的语义，利用智能指针）

解决这个问题的方法之一是：返回一个指针，并且该指针指向一个动态分配的 String 对象。但最好的解决方案是更进一步，即将指针包含在 auto_ptr 中并返回。

```
// 方法 3：正确(最终的解答！)
//
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;
```

条款 20：应该尽量使用+=而不是使用+，因为后者是用前者实现的，同时用+会产生一个临时变量。Operator+不应该申明为成员函数。

如果 operator+像上面的代码那样被定义为成员函数，那么这个运算符并不会像用户期待的那样工作，即使你确实实现了从其他类型到这个类的隐式转换。例如，当需要将 Complex 对象与数值进行相加时，你可以写“a=b+1.0”，却不能写“a=1.0+b”，因为成员函数 operator+的左参数要求是一个 Complex 对象（而不是一个 double 类型的常量）。

条款 27.类中使用 pimpl 的好处，隐藏实现细节，降低编译依赖

现在，我们来讨论 Pimpl 的优点。

C++可以使我们很容易将类的私有部分封装起来，从而防止未授权的访问。但不幸的是，由于头文件这种方法是从 C 语言中继承而来的，因此，如果要将私有部分的依赖性也封装起来，就需要多做一些工作。“但是，”你说，“封装的要点就在于客户代码无须去了解或者关心类的私有部分的实现细节，对不对？”是的，在 C++中，客户代码确实无须了解或者关心如何对类的私有部分进行访问（因为除非类的友元，否则将不允许访问类的私有部分）。然而，我们在类的头文件中可以看到类的私有部分，因此客户代码不得不依赖在这些私有部分中使用的所有类型。

如何才能更好地将客户代码与类的私有实现细节分离开来？其中一种很好的方式就是使用句柄/本体（handle/body）（Coplien92）惯用法（我把这个惯用法也叫做 Pimpl 惯用法[1]，这个名字特意取自于 pimpl_指针）的一种特殊形式——编译器防火墙（Lakos96，Meyers98，Meyers99，Murray93）。

Pimpl 只是一个不透明的指针（指向一个进行了前置声明但又没有定义的辅助类），

用来隐藏类的私有成员。也就是说，现在我们不是这样写：

```
// file x.h
class X
{
  // 公有成员和保护成员
private:
  // 私有成员；当私有成员发生改变时，
  // 所有的客户代码都必须重新编译
};
```

而是写成：

```
// file x.h
class X
{
  // 公有成员和保护成员
private:
  struct XImpl;
  XImpl* pimpl_;
    // 指向前置声明类的指针
};

// file x.cpp
struct X::XImpl
{
  // 私有成员；完全被隐藏起来，当私有成员发生改变时，
  // 客户代码不需要重新编译
};
```

在每个 X 对象中对包含的 XImpl 对象都是动态分配的。如果将对象看做一个物理内存块，那么现在这种做法相当于从根本上削减了对象中的大部分内存块，并且只用“另外一小块内存”来代替——也就是这个不透明的指针，Pimpl。

这个惯用法的主要优点就在于它打破了编译期的依赖性。

- 在私有部分使用的类型定义，只有在类的实现中才会需要，而在客户代码中是不需要的，这样就可以消减额外的#include 指令并提高编译速度。
- 类的实现可以被修改，也就是说，可以自由地增加或者删除类的私有成员，而客户代码无须重新编译。

条款 32.

```
//*** 示例 3(b)
#include <iostream>
#include <string>       // 在这个头文件中声明了
                        // 用于字符串的 std::operator<<
int main()
{
  std::string hello = " Hello, world";
  std::cout << hello;   // 正确:调用的是 std::operator<<
}
```

对于上述的 std::cout<<hello 如何找到 string 类中重载的<<，与 Koenig 查找规则有关，而不需要指明<<所在的命名空间 std，函数参数类型所在的命名空间也会参与检索，上面 string 所在的 std 也会被查找。

条款 34.

子类中同名的函数会隐藏父类的所有同名函数（即所有重载的所有函数）

查找过程，先找函数，在判断权限是否可访问

过编译的，这会使大多数新 C++程序员感到奇怪。简单地说，当我们在派生类 D 中声明一个名为 g 的函数时，这个函数将会自动地隐藏所有在直接基类和间接基类中名字相同的函数。虽然“很明显”的是，D::g 不可能是程序员想要调用的函数（因为不仅 D::g 的函数原型不正确，而且这个函数是私有的，根本就无法被访问），但这并不是主要的原因，这里的主要原因是函数 B::g 被隐藏了，并且在名字查找时不会被考虑到。

让我们来仔细分析一下，看看到底发生了什么，当编译器遇到函数调用 d.g(i)时，它会去做哪些工作。首先，它将在最内层的作用域中进行查找（在本例中是类 D 的作用域），并且列出所有能找到且名字为 g 的函数（无论这些函数是否是可以访问的，甚至不管参数的个数是否正确）。只有在没有找到任何合适的函数时，编译器才会继续“往外”进入下一个封闭的作用域中重复查找过程——在本例中，这个作用域就是基类 B 的作用域——最后的结果是，编译器在找遍了所有的作用域之后，没有发现拥有正确名字的函数，或者在某个作用域中，至少找到了一个候选函数。如果编译器在作用域中找到了一个或多个候选函数，将停止查找，并对找到的候选函数进行分析，包括执行重载解析，应用访问权限规则等处理步骤。

查找理由：

至于为什么编译器必须按照这种方式工作，理由很充分[10]。以极端的情况为例，这种方式是很直观的：如果仅从参数类型的角度来考虑，编译器将优先选择参数类型近似正确匹配的成员函数，而不是参数类型完全匹配的全局函数。

条款 43

Const 若只是修饰函数值传递的参数，那么它不具有重载的功能

1. Point 对象是通过传值方式传递给函数的，因此用 const 声明这个对象并不会带来好处。事实上，无论在参数前面是否使用了 const，对于编译器来说都是没有区别

的。例如：

```
int f( int );
int f( const int );  // 这将再次声明 f(int)，
                     // 而并不是重载，因此只能有一个函数
int g( int& );
int g( const int& ); // 这与 g(int&) 是不同的
                     // 函数 g 被重载了
```