Exceptional C++ Style 笔记

By Ken 2010-12-25

（一） 泛型程序设计与 C++标准库

✧ 下面的代码中，注释 A 和注释 B 所示的两行代码有何区别？

```
// Example 1-1: [] vs. at
//
void f(vector<int>& v) {
  v[0];         // A
  v.at(0);    // B
}
```

在实例 1-1 中，如果 v 非空，A 行跟 B 行就没有任何区别；如果 v 为空，B 行一定会抛出一个 std::out_of_range 异常，至于 A 行的行为，标准未加任何说明。

有两种途径可以访问 vector 内的元素。

(1) 其一，使用 vector<T> ::at。该成员函数会进行下标越界检查以确保当前 vector 中的确包含了需要的元素。试图在一个目前只包含 10 个元素的 vector 中访问第 100 个元素是毫无意义的，这样做会导致抛出一个 std ::out_of_range 异常。

(2) 其二，我们也可以使用 vector<T>::operator[]，C++98 标准说 vector<T>::operator 可以、但不一定要进行下标越界检查。实际上，标准对 operator[]是否需要进行下标越界检查只字未提，不过标准同样也没有说它是否应该带有异常规格说明。因此，标准库实现方也可以自由选择是否为 operator[]加上下标越界检查功能。如果使用 operator[]访问一个不在 vector 中的元素，你可就得自己承担后果了。

既然下标越界检测帮助我们避免了许多常见问题，那么为什么标准不要求 operator[]实施下标越界检查呢？简短的答案是效率。总是强制下标越界检查会增加所有程序的性能开销（虽然不大），即使有些程序根本不会越界访问。况且我们还有另一个理由要求 operator[]具有高效性：设计 vector 是用来替代内置数组的，因此其效率应该与内置数组一样，内置数组在下标索引时是不进行越界检查的。如果你需要下标越界检查，可以使用 at。

✧ 考虑如下的代码：

```
// Example 1-2: Some fun with vectors
//
vector<int> v;

v.reserve(2);
assert(v.capacity() == 2); //(1)
v[0] = 1;    // (2)
v[1] = 2;    // (3)
for(vector<int>::iterator i = v.begin(); i < v.end(); i++) { //(4)
  cout << *i << endl;
}
```

(1),这里的断言存在两个问题，一个是实质性的，另一个则是风格上的。

首先实质性问题是，这里的断言可能会失败。为什么？因为上一行代码中对 reserve 的调用将保证 vector 的容量至少为 2，然而它也可能大于 2.事实上这种可能性是很大的，因为 vector 的大小必须呈指数速度上升，因而 vector 的典型实现可能会选择总是按指数边界来增大其内部缓冲区，即使是通过 reserve 来申请特定大小的时候。因此上面代码中的断言条件表达式应该使用>=，而不是==。

其次，风格上的问题是，改断言（即使是改正后的版本）是多余的。为什么？因为标准已经保证了这里所有断言的内容。所以再将它明确地写出来只会带来不必要的混乱。

(2)(3), 大小（size，跟 resize 对应）跟容量（capacity，与 reserve 相对应）之间有着很大的区别：

(1) size 告诉你容器中目前实际有多少个元素，而对应地，resize 则会在容器的尾部添加或删除一些元素，来调整容器当中实际的内容，使容器到达指定大小。这两个函数对 list、vector 和 deque 都适用，但对其它容器并不适用。

(2) capacity 则告诉你最少添加多少个元素才会导致容器重分配内存，而 reserve 在必要的时候总是会使容器的内部缓冲区扩充至一个更大的容量，以确保至少能满足你所指出的空间大小。这两个函数仅对 vector 使用。

本例中我们使用的是 v.reserve(2)，因此我们知道 v.capacity() >= 2，这没问题，但值得注意的是，我们实际上并没有向 v 当中添加任何元素，因而 v 仍然是空的！v.reserve(2)只是确保 v 当中有空间能够放得下两个或更多的元素而已。

(4), 这段 for 循环代码什么都不会打印，因为 vector 现在根本就是空的！这段代码还存在一些风格上的问题：

(3) **尽量做到 const 准确性。**以上的循环当中，迭代器并没有用来修改 vector 中的元素，因此应当改用 const_iterator。

(4) **尽量使用!=而不是<来比较两个迭代器。**确实，由于 vector<int>::iterator 恰巧是一个随机访问迭代器（当然，并不是 int*），因此在这种特定情况下将它跟 v.end()比较是没有任何问题的。但问题是<只对随机访问迭代器有效，而!=对于任何迭代器都是有效的，因此我们应该使用!=比较迭代器作为日常惯例，除非某些情况下去而是需要<（注意，使用!=还有一个好处就是便于将来（如果需要）更改容器类型）。

(5) **尽量使用前缀形式的--和++。**让自己习惯于写++i 而不是 i++，除非真的需要用到 i 原来的值。例如，如果既要访问 i 所指的元素，又要将 i 向后递增一位的话，后缀形式 v[i++]就比较适用了。

(6) **避免无谓的重复求值。**本例中 v.end()所返回的值在整个循环的过程中是不会改变的，因此应当避免在每次判断循环条件时都调用一次 v.end()，或许我们应当在循环之前预先将 v.end()求出来。

(7) **尽量使用\n 而不是 endl。**使用 endl 会迫使输出流刷新其内部缓冲区。如果该流的确有内部缓冲区，而且确实不需要每次都刷新它的话，可以在整个循环结束之后写一行刷新语句，这样程序会执行得快很多。

(8) **尽量使用标准库中的 copy()和 for_each()，而不是自己手写循环，因为利用标准库的设施，你的代码可以变得更为干净简洁。**而且这样前面所有的问题都消除了。

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
```

- ✧ **准则：**记住 size/resize 以及 capacity/reserve 之间的区别。
- ✧ **准则：**确保 const 正确性。特别是不对容器内的元素做任何改动的时候，记得使用

const_iterator。尽量使用!=而不是<来比较迭代器。养成默认情况下使用前缀形式--和++的习惯，除非你的确需要用到原来的值。实施复用：尽量复用已有的算法，特别是标准库算法（例如 for_each），而不是手写循环。

✧ **格式化准则小结：**

永远不要使用 sprintf	默认情况下，当效率不是关键时	如果效率确实低到成问题的话（这必须是根据性能分析结果得出的结论，而不是臆测）
如果想要做的只是将数据转化为其字符串表示的话	boost::lexical_cast	std::strstream 或 snprintf
较为简单的格式化任务，或者需要宽字符支持，或者需要让格式化代码能够用于模板当中	std::stringstream 或 std::strstream	std::strstream 或 snprintf
较为复杂的格式化任务，而且不需要宽字符串支持，也不需要将负责格式化任务的代码置于模板当中	snprintf	snprintf

✧ 运行时多态的两个主要缺点：

(1) 首先，这些类型必须位于同一个继承自某个公共基类的类层次当中；

(2) 其次，当虚函数在一个密集的循环当中被调用的时候，你可能会注意到它会带来一些运行时开销，因为通常每次对虚函数的调用都要通过一个额外的间接层——虚函数指针，同时，编译器负责根据你的意思将虚函数调用分发到派生类当中对应的函数。

✧ 什么是 std::mem_fun？你什么时候会去使用它？给出一个例子。

标准库里面的 mem_fun 是一个适配器（adapter）类，它那能够将成员函数适配为所谓的函数子（functor），从而可被标准库算法以及其它正常情况下只使用自由函数的代码所使用。

例如，假设有如下的代码：

```
class Employee {
public:
 int DoStandardRaise() { /*…*/ }
 //…
};

int GiveStandardRaise(Employee& e) {
 return e.DoStandardRaise();
}

std::vector<Employee> emps;
```

下面这种用法可能是我们司空见惯了的：

```
std::for_each(emps.begin(), emps.end(), &GiveStandardRaise);
```

但是，如果 GiveStandardRaise()并不存在，或者由于某些原因需要直接去调用 Employee 中对应的成员函数的话，该怎么办呢？答案是可以这么写：

```
std::vector<Employee> emps;
std::for_each(emps.begin(), emps.end(),
```

std::mem_fun_ref(&Employee::DoStandardRaise));

mem_fun_ref 末尾的_ref 是由一些历史性的古怪癖好造成的。在写这类代码的时候，如果你的容器是老式的，其中包含的是对象的话，你就应当记住使用 mem_fun_ref，因为 for_each 操纵的将会是容器中对象的引用。而倘若容器中包含的是指向对象的指针，你就应该使用 mem_fun：

```
std::vector<Employee*> emp_ptrs;
std::for_each(emp_ptrs.begin(), emp_ptrs.end(),
        std::mem_fun(&Employee::DoStandardRaise));
```

你可能注意到了，我在示例中展示的成员函数皆是无参的。对于这些接受一个参数的成员函数，可以利用 std::bind…()辅助函数，使用原则跟 mem_fun 一样。然而遗憾的是，这种做法不适用于那些接受两个或多个参数的函数。

使用 mem_fun，但不要将它用在标准库自己身上。（标准库函数实现可以有一个或多个额外的默认参数）。使用 mem_fun 必须知道函数的确切签名。

✧ 在如下的函数中存在着一个不易觉察的陷阱，是关于代码泛型性的。找出它，并给出修正它的最佳办法。

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

template <class FwdIter>
void destroy(FwdIter first, FwdIter last) {
 while(first != last) {
  destroy(first);   ➔ destroy(&*first)
  ++first;
 }
}
```

两个参数的 destroy(FwdIter, FwdIter)是被模板化了的，因而可以接受任何泛型迭代器，它会依次将区间当中的每个迭代器传递给单参的 destroy(T*)，而后者要求 FwdIter 必须为纯粹的指针！这毫无必要地丧失了一些原本对 FwdIter 进行模板化带来的泛型性（**泛型丧失**）。这也同样意味着，某些代码如果试图调用 destroy(FwdIter, FwdIter)，而给出的迭代器又不是指针类型的话，则可能导致相当晦涩的错误信息。

✧ 下面这段代码的语义是什么？其中模板形参 T 的要求是什么？又可能将这些要求当中的任何一条消除掉吗？如果可能的话，请说明如何做，并讨论这么做是好还是坏，给出相应的理由。

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

swap()只不过通过拷贝构造函数和赋值操作符来将两个值进行交换而已，因而它要求 T 具有一个拷贝构造函数和一个赋值操作符。还有就是这个 swap 不是异常安全的。有两个办法可以消除“T 必须具有一个赋值操作符”这一限制。

(1) 第一个办法即特化或重载 swap()，假设我们有一个类 MyClass，该类遵循了一个常用手法，即提供一个不抛出异常的 Swap()成员函数。这么一来我们就可以针

对 MyClass 来特化标准的 swap()函数了。如下：

```
// Example 6-2(c): Overloading swap.
    //
    class MyClass {
    public:
      void Swap(MyClass&) /* throw() */ ;
      // ...
    };

    //(1)
    namespace std {
      template<> swap<MyClass>(MyClass& a, MyClass& b) { // throw()
        a.Swap(b);
      }
    }

    //(2) NOTE: Not in namespace std.
    swap(MyClass& a, MyClass& b) /* throw() */ {
      a.Swap(b);
    }
```

这就是说，如果创建的新类型提供了类似于 swap 的操作，那么通常最好也为你的类特化一下 std::swap()（或者在其它名字空间中提供自己的 swap()承载也行）。你的 swap()通常要比一般化的 std::swap()更高效，后者使用蛮力，而你的 swap()则可以用巧劲，而且你的 swap()往往能够改善 swap()本身的异常安全性。

(2) 第二个办法是“先析构在重建”。其核心理念是通过 T 的拷贝构造函数而非赋值操作符来实现交换，这种做法的前提是 T 必须确实是具有拷贝构造函数。这种做法不是异常安全的。（非常规手段，不推荐使用）

```
// Example 6-2(d): swap without assignment.
    //
    template <class T>
    void swap(T& a, T& b) {
      if(&a != &b) {        // note: this check is now necessary!
        T temp(a);

        destroy(&a);
        construct(&a, b);

        destroy(&b);
        construct(&b, temp);
      }
    }
```

✧ **准则：**如果你的类型（跟只知道蛮干的 std::swap()相比）有一个更好的途径可以用于交换其对象的话，请考虑为它特化 std::swap()。

- **准则：**记住，函数不能偏特化，只能重装。写一个看似函数模板偏特化的函数模板实际上是在写一个单独的主函数模板。
- **函数模板/普通函数调用顺序：**

  (1) 非模板函数是 C++中的一等公民。如果一个普通的非模板函数跟一个函数模板在重载解析的参数匹配中表现一样好的话，编译器会选择普通函数。

  (2) 如果编译器发现没有合适的“一等公民”可选的话，那么**主函数模板**为 C++中的二等公民就会被纳入考虑。具体选择哪个**主函数模板**则取决于哪个的参数类型匹配得最好，如果这样还不能选出惟一的准函数模板的话，编译器则会根据下面一组相当晦涩的规则（模板偏序规则）来确定哪个**主函数模板**是“最特化”的。

  - 如果显而易见存在着一个“最特化”的**主函数模板**的话，该**主函数模板**就会被选中，如果这个被选中**主函数模板**碰巧又针对所使用的模板实参（列表）做了特化的话，该特化版本就会被编译器选中，否则编译器将使用正确的类型实例化的主模板。

  (3) 否则，如果有两个或多个“最特化”的**主函数模板**相互之间不能分出孰优孰劣的话，调用就是二义性的，因为编译器不能确定它们中哪个是跟好的匹配。程序员只能以某种方式对调用进行一些限制，明确指出想要调用的是哪个函数。

  (4) 否则，如果没有任何**主函数模板**可以匹配调用的话，调用就是错误的，这时候程序员就有必要去纠正他们的代码。
- 在如下的代码中，最后一行代码调用的是 f()的哪个版本？为什么？(Dimov / Abrahams)

```
template<class T>    //a
void f(T);

template<>           //b
void f<int*>(int*);

template<class T>    //c
void f(T*);

// …

int *p;
f(p);    // which of the f's is called here?
```

  最后一行调用的是(b)!重载决议无视特化的存在，只在主函数模板之间进行决议。理解这个例子的关键是：模板特化并不参与重载决议。
- **准则：**记住，函数模板特化并不参与重载决议。只有在某个主模板被重载决议选中的前提下，其特化版本才有可能被使用。而且，编译器在选择主模板的时候并不关心它是否有某个特化版本。（理由：标准委员会认为，如果仅仅由于你碰巧为某个特定的模板写了一个特化就导致编译器选择了不同的模板，人们就会感到惊讶。）
- **准则：**教训#1：如果你想要将一个主模板特化，同时又希望该特化版本能够参与重载决议（或者希望确保当它能够完全匹配用户调用的时候能够被编译器选用）的话，只需要将其写出普通函数即可。**推论：如果你确实提供了某个函数模板的重载，那么你应当避免为它提供特化。**
- **准则：**教训#2：如果你正在写一个可能需要被特化的主函数模板的话，请尽量将它

写成一个孤立的、永远不该被特化或重载的函数模板，并将其具体实现全部放入一个包含了一个静态函数的类模板当中。这么一来任何人都可以对后者进行特化（全特化或偏特化），而同时又不会影响到主函数模板的重载决议。如下所示：

```
// Example 7-2(c): Illustrating Moral #2
//
template<class T>
struct FImpl;

template<class T>
void f(T t) { FImpl<T>::f(t); }          // clients, don't touch this!

template<class T>
struct FImpl {
  static void f(T t);                    // clients, go ahead and specialize this
};
```

✧ **在声明一个友元时，你有四种选择，将它们归结如下（#2 和#4 情形只匹配非模板函数，因此想将模板特化声明为类的友元，我们有两个选择，一是制造出情形#1，二是制造出情形#3）：**

1) 如果友元的名字看起来像是一个模板特化，即显示给出了模板实参的话（例如 Name<SomeType>），**则该友元就是指那个模板的特化体**。
2) 否则如果该友元的名字由一个类或名字空间所限定（例如 Some::Name），并且这个类或名字空间当中的确包含一个名字与友元声明中的名字相匹配的**非模板函数**的话，则这个**非模板函数**就是所声明的友元。
3) 否则如果该友元的名字有一个类或名字空间所限定（例如 Some::Name），并且这个类或名字空间中的确包含一个名字与友元声明中的名字相匹配的函数模板的话（推导出相应的模板参数），则该友元就是那个**函数模板**的特化体。
4) 否则友元声明中的名字肯定是非限定名，声明（或重复声明）了一个普通（非模板）函数。

✧ 假设我们有一个函数模板，它会对其所操作的对象做一些“私事”。例如，考虑[boost]中的 boost::checked_delete()函数模板，她负责将接受的对象销毁，在它做的其它事中有一件事就是调用该对象的析构函数。

```
namespace boost {
template<typename T> void checked_delete(T* x) {

   // … other stuff …

   delete x;
 }
}
```

现在，假设想要将这个函数模板用到一个类上，而这个类的析构函数有碰巧是私有的。

```
class Test {
 ~Test() { }                    // private!
};
```

```
Test* t = new Test;
boost::checked_delete(t);    // error: Test's destructor is private,
                                        // so checked_delete can't call it.
```

解决的方案很简单，只需要将 checked_delete 声明为 Test 的友元即可（除此之外惟一的选择是放弃 Test 的析构函数的私有性，即将它声明为公用的）。实际上，把 checked_delete 声明为友元虽然合法，但不是所有的编译器都支持这种（友元模板）声明。

(8-1，很多编译器都能不能编译通过。大多数不能正确处理 8-1 的编译器的主要问题，都出在将一个位于另一名字空间中的函数模板的特化体声明为友元的时候)

```
// Example 8-1 again
//
namespace boost {
  template<typename T> void checked_delete(T* x) {

    // … other stuff …

    delete x;
  }
}

class Test {
  ~Test() { }
  friend void boost::checked_delete(Test* x);                    // the original code
};

int main() {
  boost::checked_delete(new Test);
}
```

(8-2，绝大部分编译器都能编译通过，gcc3.4 编译不能通过，但 4.+没问题)

```
// Example 8-2: The other way to declare friendship
//
namespace boost {
  template<typename T> void checked_delete(T* x) {
    // … other stuff …
    delete x;
  }
}

class Test {
  ~Test() { }
  friend void boost::checked_delete<>(Test* x);              // the alternative
```

```
};

int main() {
  boost::checked_delete(new Test);
}
```

Or, equivalently, we could have spelled out:

```
friend void boost::checked_delete<Test>(Test* x);        // equivalent
```

Aside: It's the Namespace That's Confusing Them(8-3，基本所有的编译器都能编译通过)

```
// Example 8-3: If only checked_delete weren't in a namespace…
//
template<typename T> void checked_delete(T* x) { // no longer in boost::
  // … other stuff …
  delete x;
}

class Test {
  friend void checked_delete<Test>(Test* x);            // no longer need "boost:"
};
int main() {
  checked_delete(new Test);
}

//(8-4)  不是办法的办法
namespace boost {
  template<typename T> void checked_delete(T* x) {

    // … other stuff …

    delete x;
  }
}

using boost::checked_delete;
// or "using namespace boost;"

class Test {
  ~Test() { }

friend void checked_delete<>(Test* x);      // legal?
```

};

8-4 代码也许并非合法的 C++代码，标准并没有明确指出这种做法到底合不合法。几乎所有的编译器都拒绝这种做法。问题在于一致性，因为 using 关键字的作用是让程序员更容易地使用名字，例如在调用函数的时候，或者在声明变量或参数时给出类型名的时候。而声明就完全是另外一回事了，因为你必须在主模板所在的名字空间中才能声明它的特化版本（不能通过 using 从而在另一个名字空间中声明其特化版本），因此在声明友元的时候你也应该恪守一致行，既然想将一个模板的特化体声明为友元，那么就应该指出它的主模板所在的名字空间（而不仅仅是通过一个 using）。

小结：

想要将一个函数模板特化声明为友元，应该选择 8-2 的做法。

```
// From Example 8-2: add <> or <Test>
friend void boost::checked_delete<>(Test* x);                    // or "<Test>"
```

- **准则：（在代码中）明确地表达出你的意图。如果你所指的确实一个模板，然而你的表达式可能存在歧义的话，可以在模板名字的后面加上模板实参列表（这个列表可能是空的，即<>）。避开语言的冷僻特性，包括那些虽然可以说是合法但容易使程序员甚至编译器迷惑的地方。**
- **准则：如果你想编写可移植的代码，那就别用 export。**
- **准则：（就目前而言）避免使用 export。**
- **准则：如果你打算有选择地针对某些模板使用 export 的话：**

不要以为 export 可以让你不用发布源代码（或其等价形式）。那是个误会，而且这一点永远也不会改变。

不要期望 export 会给代码的构建过程带来令人惊讶的速度提升。虽说最初的经验并不能说明什么，但你的构建时间反而可能会变得更长。

确信你手头的工具和环境可以应付新的构建要求和依赖性（例如，如果你的 export 实现是让连接器去改变它所接收到的.obj/.o 文件的话，那么得确保你所有的工具能够理解这一点）。

如果你的导出模板使用了任何来自匿名名字空间中的函数或对象，或者使用了任何文件范围内的 static 函数或对象的话：

(1) 必须认识到，这些函数/对象的行为就好像它们是被声明为 extern 似的，如果是个函数的话，它就有可能参与到来自任意数目的源文件中的其它匿名名字空间中的任意数目的函数重载决议当中去。

(2) 因此你应当总是将这些函数的名字加上额外的修饰（丑化，例如加上类似“XXX_”这样的前缀）以防它们无意中遭受语义改变。（这个结果令人遗憾，因为匿名名字空间和文件范围的 static 修饰的目的就是为了防止这种情况的发生，然而只要你使用了 export，就很容易失去这种保护，因而你不得不重新将它们的名字修饰一番）。

认识到上面这些并非全部，你或许还会遇到一些其它的问题，而这些问题很可能超出了我们通常的模板使用中积累的经验和认识。正如 Spicer 指出的：“很难单凭一些简单的准则就能够使我们免于麻烦。”要深刻认识到 export 某种程度上仍然还是一个实验性的特性，而且 C++界目前尚没有机会来学习如何去使用 export，因此目前我们尚未能够列出一些让你可以安全使用 export 的良好原则。在不久的将来这种状况或与会有所改观。

（二） 异常安全问题及相关技术

- ✧ **准则：**尽量通过析构函数来进行异常环境下的自动清理工作，而不是通过 try/catch。
- ✧ **什么时候应当使用 try/catch？什么时候不应当使用它们？给出一个能够作为良好的编码标准的答案。**

(1) 为你的程序或子系统确立一个整体上的错误报告/处理策略并始终遵循它。特别地，该策略应当照顾到如下几个基本的方面（通常就我们遇到的实际情况而言，远远不止要考虑这些方面）：

- 错误报告。对哪些错误需要被报告以及如何报告它们给出一个明确的定义，尽量使用异常而不是其它错误报告机制（理解并参见（2））。通常，对你面临的每种情况都默认选用最具可读性和可维护性的方案。例如，对于那些不能返回值的函数（如构造函数和操作符）来说，异常时最有效的方案。同样，对于错误抛出端与处理端相隔遥远的情况来说，异常也是最理想的方案。
- 错误传播。明确定义异常不应当穿越哪些边界，通常就是指模块或 API 边界。
- 错误处理。规定在任何可能的情况下使用对资源具有所有权的对象及其析构函数来执行清理任务。

(2) 只在那些侦测到错误但自己却无法处理的地方抛出异常。（很明显，能够就地将错误处理掉的代码当然不需要报告该错误了！）。

(3) 在设置 try/catch 的地方必须有足够的知识和上下文去处理/转换特定异常，或强制实施错误策略中指出的异常边界。Sutter 发现编写 try/catch 有三个主要理由：

- 为了处理错误。这是最简单的情况：错误发生了，我们知道应该对此做些什么，于是就做了。程序继续运行（原先的异常已经被安全地处理掉了）。再次申明，尽可能地在析构函数中完成你的动作，实在不行才去使用 try/catch。
- 为了转换异常。这意味着捕获一个汇报了较低层错误的异常，然后抛出另外一个在执行转换的代码自身的（较高层面的）语义上下文中重新表达的异常。还有一种做法是将原先的异常转换为另一种表示形式，如错误代码。
- 为了在子系统边界或其它运行时防火墙边界上进行 catch(…)。这通常也会涉及到异常/错误的转换，通常是转换为错误代码或其它非异常形式的表示方式。例如，当栈展开（stack unwind）到达了某个 C API 的时候，就只剩下了两个选择：一是从该 API 函数中立即返回一个错误代码，而是设置一个错误状态以便事后调用方可以通过 GetLastError() API 函数去查阅。

- ✧ **准则：**为应用程序或子系统确立一个整体的错误报告/处理策略，并始终遵循它。并为错误报告、错误传播以及错误处理制定策略。
  在那些侦测到错误而自身又无法对其进行处理的地方抛出异常。
  在那些具有足够知识和上下文信息去处理/转换错误或强制实施边界条件的地方编写 try/catch（例如，在子系统边界上或其它运行时防火墙边界上进行 catch(…)）。
- ✧ **总是考虑下面的技术来使得保证异常安全变得容易。**

(1) 使用“资源获得即初始化”(RAII)惯用法来管理资源的所有权。使用对资源具有所有权的对象，如 Lock 类和 shared_ptr 类，通常是一个好主意。

(2) 使用“先在一旁将所有事情做完，然后再通过不抛出异常的操作来提交整个任务”的手法，避免在不确定所有操作能否都成功的情况下贸然去改变对象的内部状

态。

(3) 尽量使用“单一职守的类（或函数）”。像 Stack::Pop() 和 EvaluateSalaryAndReturnName()这样的做多件事情的函数是很难实现强异常安全保证的。只需简单地遵从单一职守原则，许多异常安全问题都会变得更简单，甚至无需多加思考就迎刃而解了。

✧ **准则：**函数应当总是支持它所能支持的最强的异常安全保证，但前提是不能给那些并不需要该保证的调用者带来额外开销。

✧ **准则：**永远不要允许析构函数、释放操作（deallocation）以及 swap()函数抛出任何异常，因为否则的话，就没法安全可靠地进行资源清理了。

✧ **违反异常规格会导致什么样的后果？为什么？讨论该 C++特性的基本理念。**

异常规格的理念在于使用运行期检查来确保只有某些特定的异常能够从某个函数当中逃逸出来（其它情况下则不会有任何异常从中逃出来）。例如，如下的函数的异常规格确保了 f 只会允许 A 或 B 类型的异常从它的函数体内逃出来：

int f() throw (A, B);

如果函数体内抛出了某个不在该函数的异常规格列表里面的异常，unexpected()函数就会被调用。可以通过调用标准的 set_unexpected 函数，来注册你自己的用于处理非预期异常（unexpected-exception）的函数。你自己的 handler 处理函数必须为无参且返回 void 的函数。例如：

void MyUnexpectedHandler(){…}

std::set_unexpected(& MyUnexpectedHandler);

这个用于处理非预期异常的函数究竟能够做些什么？首先，无论如何有一件事他不能做，即它不能通过通常的函数返回方式来返回。而它可以做的事情有如下两件：

(1) 它可以决定将当前异常转换为某个被异常规格列表所允许的异常，具体做法是抛出它自己的、满足那个引发它被调用的异常规格列表要求的异常。于是栈开解就会从原来离开（去调用函数）的地方继续下去。

(2) 它可以调用 terminate，该函数会终止程序。（terminate 函数本身也可以被替换掉，但替换它的任何函数同样还是必须终止程序。）

✧ **异常规格并非函数的类型的一部分。。。。。**

。。。。。有些时候它们又的确是。

```
// Example 13-3(a): You can't write an exception specification in a typedef.
//
void f() throw(A,B);
typedef void (*PF)() throw(A,B);        // syntax error
PF pf = f;                              // can't get here because of the error
```

代码中的那个 typedef 中的异常规格是非法的。C++不允许这样写，因此，C++并不允许异常规格成为函数的类型的一部分。。。。。至少在 typedef 中不能。但在其它情况下，异常规格却又确实是函数的类型的一部分，譬如说，如果你把上面代码中的 typedef 关键字去掉，而其它部分保留原样的话：

```
// Example 13-3(b): But you can if you omit the typedef!
//
void f() throw(A,B);
void (*pf)() throw(A,B);                // ok
pf = f;                                 // ok
```

对于函数指针赋值而言，只要赋值目标（及函数指针）的异常规格不比赋值源（即

函数）异常规格更严格，赋值就可以成立。

```
// Example 13-3(c): Also kosher, low-carb, and fat-free.
//
void f() throw(A,B);
void (*pf)() throw(A,B,C);                  // ok
pf = f;                                     // ok, pf's type is less restrictive
```

此外，如果你想重写虚函数的话，异常规格也会影响虚函数的函数类型（派生类的虚函数必须拥有比其基类的对应虚函数更为严格的异常规格，这个限制很明显，一个派生类对象可以被当成基类对象来使用，所以派生类对象的虚函数也理所当然应该满足其对应的基类虚函数所具有的限制（这里的限制是异常规格）。站在使用指向基类的指针来操纵派生类对象的用户的角度来说，他们会假定所调用的 f 不管动态分派到哪个派生类对象上去，都满足静态决议出的 f（即基类的 f）所限定的异常规格，即至多只能抛出 A 和 B，这样他们的代码中就只需要捕获 A 和 B 异常。而本例中的 D::f 却可能抛出任何东西，如果语言允许这一点的话，D::f 抛出的其它异常对于把它当成 B::f 的调用方代码来说就是漏网之鱼了！其后果可想而知。）：

```
// Example 13-3(d): Exception specifications matter for virtual functions.
//
class C {
 virtual void f() throw(A,B);               // same exception specification
};
class D : C {
 void f();                                  // error, now the ES matters
};
```

所以对于当前 C++中的异常规格而言，其首要问题在于它们确实是个“浅陋的类型系统”，它们的游戏规则跟 C++类型系统的其它部分不同。

✧ **异常规格到底真正能够做些什么[sic]：**

(1) 强制（Enforce）函数在运行期只能抛出其异常规格列表里列出的异常（也可能不抛出任何异常）。

(2) 编译器必须检查是否的确只有列出的异常被抛出了，这一点是编译器能够（或阻止了）。

为了弄清编译器究竟必须做什么，考虑如下的例子，它为我们的示例函数 Hunc 提供了一个函数体：

```
    // Example 13-4(a)
int Hunc() throw(A,B) {
 return Junc();
}
```

从功能上说，编译器必须生成类似如下的代码，它的运行期开销跟你自己手写这些 try-catch 相比通常是一样的（虽然省去了许多键盘敲击，因为是编译器自动帮你生产的）：

```
// Example 13-4(b): A compiler's massaged version of Example 13-4(a)
//
int Hunc()
try {
 return Junc();
```

```
}
catch(A) {
 throw;
}
catch(B) {
 throw;
}
catch(…) {
 std::unexpected();   // won't return! but might throw an A or a B if you're lucky
}
```

在这里我么可以把在异常规格背后发生的事实看得更清楚，并不是假设只有某些特定的（异常规格列表中指出的）异常会抛出从而让编译器据此进行优化，而是恰恰相反：编译器必须得在运行期进行更多的工作以确保只有这些特定的（异常规格列表中指出的）异常会从函数中逃逸出来。这会导致性能损失。（此外，有些编译器会自动拒绝将那些具有异常规格的函数进行内联。有些编译器根本不能很好地基于异常相关的信息进行优化，对于这些编译器而言，即使函数体内的代码清清楚楚地表明不可能抛出任何异常，它们也会在里面生成 try/catch 块。）

✧ **准则：**原则 1：永远不要为函数加上异常规格。

原则 2：除非你想声明的是空异常规格列表（即 throw()），但若果是我的话，我甚至连这种做法也会避免。

（三） 类的设计、继承与多态

✧ 当创建一个类类型的 C++对象时，其各个“部件”分别是按什么顺序被初始化的？请尽量给出明确而完整的答案。

- 首先，最上层派生类的构造函数负责调用虚基类子对象的构造函数。所有虚基类子对象会按照深度优先、从左到右的顺序进行初始化。
- 其次，直接基类子对象按照它们在类定义中声明的顺序被一一构造起来。
- 再其次，（非静态）成员子对象按照它们在类定义体中声明的顺序被一一构造起来。
- 最后，最上层派生类的构造函数体被执行。

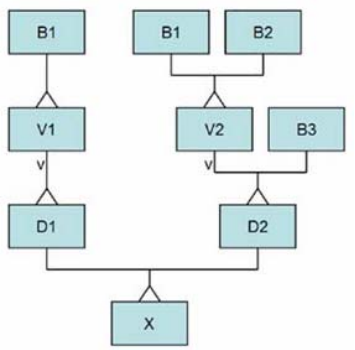
```
class B1 { };
class V1 : public B1 { };
class D1 : virtual public V1 { };

class B2 { };
class B3 { };
class V2 : public B1, public B2 { };
class D2 : public B3, virtual public V2 { };

class M1 { };
class M2 { };

class X : public D1, public D2 {
 M1 m1_;
 M2 m2_;
```

};



首先，构造虚基类子对象：

构造 V1：B1::B1() V1::V1()

构造 V2：B1::B1() B2::B2() V2::V2()

其次，构造非虚基类子对象：

构造 D1：D1::D1()

构造 D2：B3::B3() D2::D2()

再次，构造所有成员：M1::M1() M2::M2()

最后，构造 X 本身：X::X()

✧ **准则：**避免过度使用继承。

✧ 考虑如下的头文件，如何想让调用代码都能够直接访问该类的 private_成员。

```
// File x.h
//
class X {
public:
  X() : private_(1) { /*...*/ }

  template<class T>
  void f(const T& t) { /*...*/ }

  int Value() { return private_; }

  // …

private:
  int private_;
};
```

(1) 伪造者（非法）

```
// Example 15-1: Lies and forgery
//
class X {
  // instead of including x.h, manually (and illegally) duplicates X's
  // definition, and adds a line such as:
  friend ::Hijack(X&);
};
```

```
void Hijack(X& x) {
  x.private_ = 2;                                  // evil laughter here
}
```

(2) 偷窃者（非法）

```
// Example 15-2: Evil macro magic
//
#define private public                          // illegal
#include "x.h"

void Hijack(X& x) {
  x.private_ = 2;                                  // evil laughter here
}
```

(3) 骗子（非法）

```
// Example 15-3: Nasty attempt to simulate the object layout.
//
class BaitAndSwitch {                          // hopefully has the same data layout as X
public:                                          // so we can pass him off as one
  int notSoPrivate;
};

void f(X& x) {
  (reinterpret_cast<BaitAndSwitch&>(x)).notSoPrivate = 2;
}                                                // evil laughter here
```

(4) 语言律师（合法）

```
// Example 15-4: The legal weasel
//
namespace {
  struct Y {};
}
template<>
void X::f(const Y&) {
  private_ = 2;                                  // evil laughter here
}

void Test() {
  X x;
  cout << x.Value() << endl;                     // prints 1
  x.f(Y());
  cout << x.Value() << endl;                     // prints 2
}
```

✧ **准则：**永远不要对语言搞破坏。例如，永远不要企图通过复制类定义再添加友元声明，或提供成员模板函数特化等途径来破坏封装。

下面的 C++程序会正确编译运行吗？如果不能，为什么？如果能，其输出结果是是

什么？

```
// Twice(x) returns 2*x
//
class Calc {
public:
 double    Twice(double d);
private:
 int    Twice(int i);
 std::complex<float>    Twice(std::complex<float> c);
};

int main() {
 Calc c;
 return c.Twice(21);
}
```

简单的答案是：不行！有些程序员认为上面的做法是合理的，理由是对于 c.Twice(21)这个调用来说，惟一可访问的 Twice()重载版本是参数类型为 double 的，而 21 则正好可以转型为 double，因此当然应该调用它。但事实并非如此，理由很简单：重载决议发生在可访问性检查之前。

编译器在对 c.Twice(21)这个调用进行决议时，主要做了三件事，依次如下：

(1) **名字查找。**在做其它任何事情之前，编译器首先寻找一个至少包含一个名为 Twice 的实体“作用域”，并将其中的候选实体列表。本例中，编译器的名字查找首先是从 Calc 的作用域中开始的，编译器会看到 Calc 中是否至少存在一个名为 Twice 的成员，如果没有，就会继续依次在其基类和外围名字空间中查找，直到找到一个至少具有一个候选函数的作用域。不过在本例中，编译器查找的第一个作用域就具有三个候选 Twice。

(2) **重载决议。**接下来编译器进行重载决议，这一步目的是在候选的重载函数中选出惟一的最佳匹配。本例中的调用参数是 21，其类型是 int，同时编译器看到，Twice()的三个可用的重载版本分别接收 double，int 和 std::complex<float>类型的参数。因此很显然，接受 int 为参数的 Twice()是最佳匹配，因此 Twice(int)被重载决议选中。

(3) **可访问性检查。**最后编译器会进行可访问性检查，以确定被选出的函数到底是否可被调用。本例中该检查失败了。

有趣的是：甚至二义性匹配也比可访问性匹配更优先。

```
// Example 16-4(a): Introducing ambiguity
//
#include <complex>

class Calc {
public:
 double    Twice(double d);
private:
 unsigned    Twice(unsigned i);
 std::complex<float>    Twice(std::complex<float> c);
};
```

```
int main() {
  Calc c;
  return c.Twice(21);                    // error, Twice is ambiguous
}
```

在这个例子中，永远也通不过上面的“决议三部曲”中的第二步，即重载决议。在这一步，编译器会发现找不到一个惟一的最佳匹配，因为实参类型 int 既可以转型为 unsigned 又可以转型为 double，因而根据语言规则，这两个 Twice()被认为是同样好的匹配，正因为如此，编译器无法在它们之间决出优劣，因此给出二义性调用的错误。

甚至更为有趣的是，即使一个不可能的匹配也比一个可访问的匹配更优先。

```
// Example 16-4(b): Introducing plain old name hiding
//
#include <string>

int Twice(int i);                        // now a global function

class Calc {
private:
  std::string Twice(std::string s);
public:
  int Test() {
    return Twice(21);                    // error, Twice(string) is unviable
  }
};

int main() {
  return Calc().Test();
}
```

这里同样无法通过第二步：重载决议无法在候选列表（只包含一个 Calc::Twice(string)）中找到一个可行的匹配，因为实参类型 int 无法转换为 string。因此编译器仍然执行不到可访问性检查。记住：在进行名字查找时，编译器只要找到一个至少包含一个给定名字的作用域便不再继续查找了，即使这些所有查找到的候选函数可能都是无法调用或不可访问的。外围作用域中的其它潜在匹配则被隐藏了，永远也不会被纳入考虑。

✧ 对于私有成员：

(1) 私有成员对于任何能够看到其所属类定义的代码来说都是“可见”的（注意和“可访问”的区别）。这意味着它会参与名字查找和重载决议，因而可能会使调用变得无效或具有二义性，即使它本身可能永远不被调用。

(2) 具有对某个成员的访问权的代码可以通过泄漏该成员的指针（摆脱了名字束缚）的方式将其访问权授予其它任何代码。

```
// Example 16-5: Granting access
//
class Calc;
```

```
typedef int (Calc::*PMember)(int);

class Calc {
public:
  PMember CoughItUp() { return &Calc::Twice; }

private:
  int Twice(int i);
};
int main() {
  Calc c;
  PMember p = c.CoughItUp();        // yields access to Twice(int)
  return (c.*p)(21);                // ok
}
```

(3) Private 成员的名字只对其所属类的其它成员或友元来说是可访问的，而这里的其它成员也包括成员模板的任何显式特化（不管它的某个给定的显式特化是否为意料之中的）。

- ✧ **准则：**总是将所有数据成员放在私有区段。惟一的例外是 C 风格的 struct，后者的意图并不在于封装什么东西，因而其所有成员都可以是公用的（一个功能完整的类至少有自己的接口、行为和不变式）。
- ✧ **准则：**接口是最需要第一时间做对的事情。其它东西都可以在后期进行修正。如果你一开始就没有把接口做对的话，那么以后你可能就永远没有机会去改正它。
- ✧ **准则：**尽量让接口成为非虚函数（NVI Pattern）。
- ✧ **准则：**尽量将虚函数置为私有的。
- ✧ **准则：**只有当派生类需要调用基类中实现的虚函数的时候，我们才需要将后者设为保护的。
- ✧ **准则：**基类的析构函数要么应当为公用虚函数，要么应当为保护的非虚函数。
- ✧ **准则：**不要继承自具体类（让继承树中的非叶子节点成为抽象的）。
- ✧ **准则：**永远不要让异常从析构函数中跑出来。在编写任何析构函数的时候总是就当它具有一个不抛出任何异常的异常规格声明那样。永远不要为函数编写异常规格声明。或许你可以编写一个空的，不过如果我是你的话，我连这个也会避免。
- ✧ **准则：**避免将赋值操作符设为虚函数。
- ✧ **准则：**为了阻止编译器为派生类隐式生成默认构造函数、复制构造函数或者复制赋值操作符，最简单的、最佳的是将基类中相应的函数“藏”在非公用区段（或干脆让它消失，例如默认构造函数）。

（四） 内存及资源管理

- ✧ 在 C++标准库以及使用了 C++标准库实现的典型环境中存在有几种不同层次的内存管理？它们之间的关系有什么特点？它们之间是如何交互？它们是如何各司其职的？

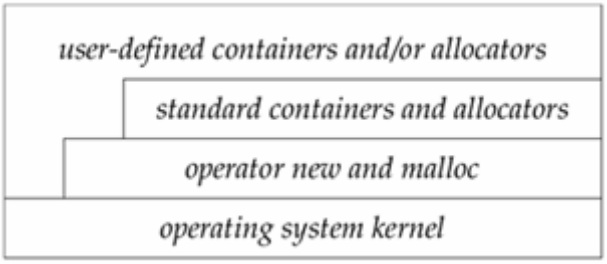
(1) 首先是操作系统内核提供了最为基础的内存分配服务。这个底层的分配策略及其特性可能根据操作系统平台的不同而又很大差异，而且这一层内存管理是可能受到硬件相关的考虑所影响的。

(2) 编译器的默认运行库也会建立起它自己的内存分配服务，譬如 C++中 operation new 和 C 中的 malloc，这一层是建立在操作系统的本地分配服务基础上的，可能

只是后者的一层薄薄的外衣，继承了其特性；也有可能通过先从后者那里“买进”大块的内存（chunk）然后再切割转手分配出去的方式来覆盖操作系统的本地内存分配服务，至于具体如何切割“买来的”大块内存就是由这一层运行时库自由选择了。

(3) 而标准容器和标准分配器进而又利用（也许会覆盖）运行时库提供的服务来实现它们自己的策略和优化。

(4) 最后，用户**自定义的容器或用户自定义的分配器**可能会进一步复用上面提到的任何一层服务（例如，如果可移植性并不那么重要的话，它们可能会想要直接访问本地分配服务）并按照它们自己的意愿来进行定制。



✧ C++标准保证所有通过 operator new 或 malloc 分配的内存对于你想要存储的任何类型的对象来说都是满足对齐条件的，这就意味着 operator new/malloc 必须遵从本地平台对于可能的数据类型的最为严格的对齐要求（例如，在典型的 32 位机器上，最大的类型为 double，通常其大小为 8 个字节。因此 operator new/malloc 分配的内存的起始地址就是 8 的倍数。另外，一个结构的对齐要求就是该结构中最大的成员的对齐要求（译者注））。

✧ 各种不同的标准库容器所包含的每个元素的基本额外开销。

Container	Typical housekeeping overhead per contained object
`vector`	No overhead per `T`.
`deque`	Nearly no overhead per `T`, typically just a fraction of a bit.
`list`	Two pointers per `T`.
`set, multiset`	Three pointers per `T`.
`map, multimap`	Three pointers per `pair<const Key, T>`.

✧ 标准 C++中提供了哪三种形式的 new?

(1) // The standard-provided overloads of operator new
// (there are also corresponding ones for array new[]):
void* ::operator new(std::size_t size) throw(std::bad_alloc);
// usual plain old boring new
// usage: new T

(2) void* ::operator new(std::size_t size, const std::nothrow_t&) throw();
// nothrow new ? usage: new (std::nothrow) T

(3) void* ::operator new(std::size_t size, void* ptr) throw();
// in-place (or "put-it-there") new ? usage: new (ptr) T

三种 new 的相互比较：

Standard version of new	Additional parameter	Performs allocation	Can fail [p7]	Throws	Replaceable
Plain	None	Yes	Yes (throws)	std::bad_alloc	Yes
Nothrow	std::nothrow_t	Yes	Yes (returns null)	No	Yes
In-Place	void*	No	No	No	No

✧ **准则：**如果你为某个类提供了任何类相关的 new，那么记得同样为它提供类相关的简单的（无额外参数的）new[名字隐藏，隐藏全局的 new]。

```
// Preferred implementation of class-specific plain new.
void* C::operator new(std::size_t s) throw(std::bad_alloc) {
 return ::operator new(s);
}
```

✧ **准则：**如果你为某个类提供了任何类相关的 new，那么记得同样为它提供类相关的定位 new。

```
// Preferred implementation of class-specific in-place new.
void* C::operator new(std::size_t s, void* p) throw() {
 return ::operator new(s, p);
}
```

✧ **准则：**如果你为某个类提供了任何类相关的 new，那么考虑是否同样提供类相关的 nothrow new（因为有些客户代码可能的确想调用它）；否则就会被其它的类 new 相关 new 隐藏了。

```
// "Option A" implementation of class-specific nothrow new. Favors consistency
// with global nothrow new. Should have the same effect as Option B.
void* C::operator new(std::size_t s, const std::nothrow_t& n) throw() {
 return ::operator new(s, n);
}

// "Option B" implementation of class-specific nothrow new. Favors consistency
// with the corresponding class-specific plain new. Should have the same effect
// as Option A.
void* C::operator new(std::size_t s, const std::nothrow_t&) throw() {
 try {
  return C::operator new(s);
 }
 catch(...) {
  return 0;
 }
```

✧ **准则：**教训#1：避免使用 nothrow new。

✧ **准则：**教训#2：无论如何，检查 new 是否失败通常都没有多大意义。

（五） 优化和效率

✧ **准则：**避免按 const 值传递对象。尽量按 const 引用传递，除非它们是像 int 这样的复制开销很低的对象。

- ✧ **准则：**避免写 inline 或试图进行其它优化，除非性能测试显示出由此必要。
- ✧ **准则：**记住，内联可能发生在任何时候（编码期，编译期，连接期，应用程序安装时，运行时，其它时候）。
- ✧ **准则：**绝对不要用 auto 关键字。它跟空白没什么区别。
- ✧ **准则：**不要使用 register（除非你明确地知道你的编译器会特殊对待带有 register 的代码）。对大多数编译器而言它跟空白没有什么区别。

（六） 陷阱、缺陷和谜题

- ✧ **C++标准：**There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion (_expr.type.conv_) as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a (. In those cases the statement is a declaration. ?[C++03] §6.8（如果一段代码能够被解释成声明，那么就是声明。）
- ✧ **准则：**优先使用具名变量作为构造函数的参数。这样可以避免与声明语句产生二义性。况且这样做也能使你的代码更清晰，更易于维护。
- ✧ **准则：**记住，浮点运算时神秘且深奥难懂的。在使用浮点数时请提高警惕，避免以来于浮点转换操作。人们对算术的所有认识在面对浮点运算时几乎都或多或少存在错误。
- ✧ **准则：**避免使用宏。永远，永远，永远不要写一个名字为常见词或缩写的宏，连想都别想。
- ✧ **准则：**尽量使用名字空间来封装名字。

（七） 风格案例研究

- ✧ **准则：**一般来说，优先考虑编写清晰、正确的代码。
- ✧ **准则：**不要依赖于隐式 int；这不是符合标准的可移植的 C++。特别地，“void main()”或“main()”从来就不是标准 C++写法，虽然仍然有很多编译器将它们作为扩展加以支持。（可移植的标准 main 必须定义为 int main()或 int main(int, char*[])）。
- ✧ **准则：**避免不必要的类型硬编码，从而扩展泛型组件的可复用性。
- ✧ **准则：**尽量使用!=而非<来比较迭代器。
- ✧ **准则：**尽量选择使用前置递增而非后置递增，除非确实需要使用旧的值。
- ✧ **索引表：**

```
template<class T, class U>
struct ComparePair1stDeref{
   bool operator()(const std::pair<T, U>& lhs, const std::pair<T, U>& rhs) const
   {
      return (*lhs).first < (*rhs).first;
   }
};
template<class IterIn, class IterOut>
void sort_idxtbl(IterIn first, IterIn last, IterOut out)
{
   std::vector<std::pair<IterIn, int> > s(last - first);
   for(int i = 0; i < s.size(); ++i)
   {
      s[i] = std::make_pair(first + i, i);
   }
```

```
    std::sort(s.begin(), s.end(), ComparePair1stDeref<IterIn, int>());
    for(int j = 0; j < s.size(); ++j, ++out)
    {
      *out = s[j].second;
    }
  }
```

- ✧ **准则：**尽量使用 explicit 构造函数，除非真的想使该构造函数能用于类型转换。
- ✧ **准则：**为惯用的函数对象提供 operator(),而不是具名的 execute()函数。
- ✧ **准则：**将非类型的模板参数换成普通的函数参数通常是一个好主意，除非它们确实应该是模板参数。（非类型的模板参数某种程度上是不常用的，因为它意味着在编译期就严格将类型确定下来，这几乎肯定不会带来太多的好处。）
- ✧ **准则：**尽量让你的对象与容器相兼容。特别是，一个对象要想能够被放入进标准容器中，就必须是可赋值的。
- ✧ **准则：**考虑为类（模板）加上多态能力，这样类模板的不同实例就可以被替换使用了，不过前提是这一点对类模板来说的确有意义。如果是这样的话，做法是为类模板提供一个基类，这样类模板的任何实例就都会继承该基类了。
- ✧ **准则：**不要限制模板，避免对特定的类型（或不那么一般的类型）进行硬编码。
- ✧ **泛型回调：**

```
class CallbackBase{
public:
  virtual void operator()(void) const {};
  virtual ~CallbackBase() = 0;
};
CallbackBase::~CallbackBae(){}
template<typename T>
{
public:
  typedef void (T::*F)();
  Callback(T& t, F f): t_(&t), f_(f) {}
  Void operator()()const { t_->*f_() ;}
private:
  T* t_;
  F f_;
};
template<typename T>
Callback<T> make_callback(T& t, void (T::*f)())
{
  return Callback<T>(t, f);
}
```

- ✧ 什么是 union，他们存在的目的是什么？
  - union 允许多个对象（不管是类还是内建类型）占用内存中的同一块空间，同一时刻只有其中一个类型处于活跃状态。
- ✧ 哪些类型的对象不可以作为 union 的成员？为什么有这一限制？
  - C++标准，具有非平凡（non-trivial）构造函数、非平凡复制构造函数、非平凡

析构函数或非平凡复制赋值操作符的类的对象和数组不可以作为 union 的成员。简单地说，一个类类型要想能被放入 union，就必须满足：

- 只可以有编译器生成的构造函数、析构函数以及复制赋值操作符。
- 无虚函数或虚基类。
- 其所有基类和非静态成员（或数组）都满足以上两点要求。

✧ **准则：**遵守“大三法则”：如果一个类需要自定义的复制构造函数、复制赋值操作符或析构函数中的任一个，那么它往往同时需要这三者。

✧ **准则：**（只）写有用的注释。注释不应该表达代码已经表达过的意思，而应该解释代码，以及解释为什么要编写这段代码。

✧ **准则：**避免使用**类型 switch**；尽量寻求类型安全性[google switch C++ Dewhurst]。

✧ **准则：**永远不要使用保留名字，即前导下划线开头或含有双下划线的名字。它们是语言为编译器和标准库实现保留的。

✧ **准则：**如果要表示一个 variant 类型，就目前而言，应当尽量使用 boost::any（或某些同样简单的东西）。