# C++编程规范

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 版本 | | | **V1.0** | | | |
| 作者 | | | **王秀松** | | | |
| 时间 | | | **2015-04-13** | | | |
| 文件名称 | | | **C++编程规范** | | | |
| 文档简述 | | | **服务器端编程规范** | | | |
| **版本** | **时间** | | **说明** | **作者** |
| **V 1.0** | **2015-04-13** | | **文档创建。** | **王秀松** |

# 目录

[C++编程规范 1](#_Toc248638913)

[目录 2](#_Toc248638915)

[1 设计风格 3](#_Toc248638916)

[2 编程风格 3](#_Toc248638917)

[3 函数与操作符 4](#_Toc248638918)

[4 类的设计与继承 4](#_Toc248638919)

[5 构造、析构、与复制 5](#_Toc248638920)

[6 命名空间与模块 5](#_Toc248638921)

[7 模板与泛型 6](#_Toc248638922)

[8 错误处理与异常 6](#_Toc248638923)

[9 STL:容器 7](#_Toc248638924)

[10 STL:算法 7](#_Toc248638925)

[11 类型安全 7](#_Toc248638926)

**编码风格:**

1. **C++头文件以.h为后缀,实现文件以.cpp为后缀。**
2. **一般情况下，C++头文件中只允许定义一个主要类结构。**
3. **防止C++头文件被多次加载的宏写法如下：**

**以 项目AgentApp的AgentApp.h为例:**

|  |
| --- |
| #pragma once  #ifndef AGENT\_AGENTAPP\_H  #define AGENT\_AGENTAPP\_H  ...  #endif//AGENT\_AGENTAPP\_H |

**规则如下：**

1. **使用#ifndef 而不使用 #if !defined()的形式**
2. **宏定义全部使用大写形式**
3. **使用<工程名>\_为前缀:** **AGENT\_**
4. **使用\_H为后缀**

**4) 指针有效性判断如if(NULL == pUserInfo)**

1. **行间缩进使用TAB，不使用空格。一个TAB占用4个空格大小。**
2. **命名规则**
   1. **宏、常量、枚举值，全部使用大写。如果为词组，各组成部分使用下划线“\_”连接。**
   2. **函数命名使用动宾词组的形式，动词为小写，其后的组成部分为大写，函数命名惯例，如getIndexOfArray,find**
   3. **一组相反的操作动词要匹配，以下是常用动词对**

|  |  |
| --- | --- |
| **set** | **get** |
| **push** | **pop** |
| **add** | **remove** |
| **new** | **delete** |
| **create** | **destroy** |

* 1. **除struct中的public变量，其它成员变量名称要加m\_前缘。**
  2. **全局变量加g\_前缀**
  3. **类名，变量名，函数名，除了前缀的部分，其余要用符合其作用的英文单词，不要使用汉语拼音。**
  4. **类名的首字母要大写**
  5. **命名如果是一个词组，各组成单词的首字母大写，其余为小写。**
  6. **变量名如果仅为一个单词，要全部小写。**

1. **{}规则**

**{，}要与它们同级的代码有相同的缩进，并单独占用一行。{}内部的代码要缩进一个TAB。**

**Function 实例如下**

|  |
| --- |
| void function()  {  int a = b;  int c = a \* b; } |
| void function(int &nParam)  {  nParam = 1;  return; } |
| void function(int\* pParam)  {  If(NULL == pParam)  {  // Log  Return;  }  Int \* pAcceptParam = NULL;  pAcceptParam = pParam;  return; } |
| Int function(int nParam)  {  int a = nParam;  return a; } |
| bool function(int nParam)  {  int a = nParam;  if(a < 0)  {  return false;  }  return true; } |
| if(value > 32)  {  ...  } |
| for(int i = 0;I < 32;++i)  {  ...  } |
| Vector<int>::iterator iter = m\_Vec.begin();  Vector<int>::iterator iterEnd = m\_Vec.end();  for(; iter != iterEnd; ++iter)  {  int nElement = \*iter;  … |
| map<int>::iterator iter = m\_Map.begin();  map<int>::iterator iterEnd = m\_Map.end();  for(; iter != iterEnd; ++iter)  {  int nElement = iter->second;  …  } |

**流程控制块必须包含在{}内，即使只有一条语句。如：**

|  |
| --- |
| if (weight > 32)  {  price += 3200; }  else  {  price += weight\*100; } |

1. **Enum ,Switch,Struct,Template,Class规则**
2. **switch下的case需要缩进**
3. **case内的代码还要缩时**
4. **每个case分支都要有break或return从该分支退出，如果想让下一个分支继续处理，需写注释解释原因。**
5. **每个switch都要有default语句，如果该语句只有break，要写注释解释原因。**

**Enum 实例如下**

|  |
| --- |
| enum eIOErrorType  {  eIOSuccess = 0 ,  eIOSystemError = 200001,  eIOSelfClose = 200002,  eIOClosed = 200003,  eIOTimeOut = 200004,  eIOMax = 200005,  }; |

**Switch示例如下：**

|  |
| --- |
| switch (pPacketHead->nCmd)  {  case PROTO\_CMD\_F2W\_LOGIN\_RET:  ProcessFepLoginRet(pDataInfo,nDataLen);  break;  case PROTO\_CMD\_F2W\_HEARTBEAT\_RET:  ProcessFepHeartBeatRet(pDataInfo,nDataLen);  break;  default:  break;  } |

**Struct 实例如下**

|  |
| --- |
| typedef struct tagEmailItemPackInfo  {  \_\_INT32 nItemId;  \_\_INT32 nCount;  \_\_INT32 nBindMark;  }EmailItemPackInfo;  typedef vector<EmailItemPackInfo> EmailItemPackInfoVec;  typedef struct tagUserEmailInfo  {  wchar\_t sEmailCaption[20];  wchar\_t sEmailContent[200];  \_\_INT32 nSendDate;  EmailItemPackInfoVec vEmailItemPackInfo;  }UserEmailInfo;  typedef vector<UserEmailInfo> UserEmailInfoVec; |

**Template实例如下**

|  |
| --- |
| template <typename T>  class CEQ\_SingletonT  {  protected:  CEQ\_SingletonT(){};  ~CEQ\_SingletonT(){};  public:  static T &Instance()  {  static T instance;  return instance;  }  }; |

**Class 实例如下**

|  |
| --- |
| class CWebSocketFrameHandler:public CEQ\_SingletonT<CWebSocketFrameHandler>  {  public:  CWebSocketFrameHandler();  virtual ~CWebSocketFrameHandler();  public:  inline unsigned int& getVersion(){ return m\_nVersion;};  inline void setVersion(unsigned int nVersion){ m\_nVersion = nVersion;};  bool CheckHandleShake(const char\* data, unsigned int &version, unsigned int length, std::string &response);  private:  unsigned int m\_nVersion;  }; |

规范条例:

1. 不要拘泥于小节，之规定需要规定的事情，不要强制施加个人的喜好或者过时的做法。Don't sweat the small stuff. (Or: Know what not to standardize.) Say only what needs saying: Don't enforce personal tastes or obsolete practices
2. 在高警告级别干净利落的进行编译，使用编译器的最高警告级别。理解所有的警告。通过修改代码而不是降低警告基本来排除警告。Compile cleanly at high warning levels。Take warnings to heart: Use your compiler's highest warning level. Require clean (warning-free) builds. Understand all warnings. Eliminate warnings by changing your code, not by reducing the warning level.
3. 使用自动构建系统，一次按键就解决问题：使用完全自动化的构建系统，无需用户干预即可构建整个项目。Use an automated build system。Push the (singular) button: Use a fully automatic ("one-action") build system that builds the whole project without user intervention.
4. 使用版本控制系统，永远不要让文件长时间的登出。在新的单元测试通过之后，应该频繁的登入。确保登入的代码不会影响构建的成功。Use a version control system. The palest of ink is better than the best memory (Chinese proverb): Use a version control system (VCS). Never keep files checked out for long periods. Check in frequently after your updated unit tests pass. Ensure that checked-in code does not break the build.
5. 在代码审查上投入更多的关注有助于提高质量。两处自己的代码，阅读别人的代码。互相学习，彼此都会受益。Invest in code reviews。Re-view code: More eyes will help make more quality. Show your code, and read others'. You'll all learn and benefit

# 设计风格

1. 一个实体应该只有一个紧凑的职责，一次只解决一个问题：只给一个实体（变量、类、函数、命名空间、模块和库）赋予一个定义良好的职责。随着实体变大，其职责范围自然也会扩大，但是职责不应该发散。Give one entity one cohesive responsibility. Focus on one thing at a time: Prefer to give each entity (variable, class, function, namespace, module, library) one well-defined responsibility. As an entity grows, its scope of responsibility naturally increases, but its responsibility should not diverge.
2. 正确优于速度。简单优于复杂。清晰优于技巧。安全优于不安全。Correctness, simplicity, and clarity come first. KISS (Keep It Simple Software): Correct is better than fast. Simple is better than complex. Clear is better than cute. Safe is better than insecure
3. 编程中应知道何时和如何考虑可伸缩性。密切关注渐进复杂性，最好不差于线性关系，集中精力改善算法的O（N）复杂性，而不是进行小型的优化，比如节省一个多余的加法运算。Know when and how to code for scalability. Beware of explosive data growth: Without optimizing prematurely, keep an eye on asymptotic complexity. Algorithms that work on user data should take a predictable, and preferably no worse than linear, time with the amount of data processed. When optimization is provably necessary and important, and especially if it's because data volumes are growing, focus on improving big-Oh complexity rather than on micro-optimizations like saving that one extra addition.
4. 不要进行不成熟的优化，优化第一原则就是：不要优化。优化的第二原则是海水不要优化。再三测试，而后优化。Don't optimize prematurely. Spur not a willing horse (Latin proverb): Premature optimization is as addictive as it is unproductive. The first rule of optimization is: Don't do it. The second rule of optimization (for experts only) is: Don't do it yet. Measure twice, optimize once.
5. 不要进行不成熟的劣化，放松自己轻松编程。Don't pessimize prematurely. Easy on yourself, easy on the code: All other things being equal, notably code complexity and readability, certain efficient design patterns and coding idioms should just flow naturally from your fingertips and are no harder to write than the pessimized alternatives. This is not premature optimization; it is avoiding gratuitous pessimization.
6. 尽量减少全局数据和共享数据，共享会导致冲突，避免共享数据，尤其是全局数据。共享数据会增加耦合度，从而降低可维护性，通常还会降低性能。Minimize global and shared data. Sharing causes contention: Avoid shared data, especially global data. Shared data increases coupling, which reduces maintainability and often performance.
7. 隐藏信息，不要泄密：不要公开提供抽象的实体的内部信息。Hide information。Don't expose internal information from an entity that provides an abstraction.
8. 懂的何时和如何进行并发性编程，如果应用程序使用多个线程或者进程，应该知道如何尽量减少共享对象， 以及如何安全地共享必须共享的对象。Know when and how to code for concurrency。If your application uses multiple threads or processes, know how to minimize sharing objects where possible (see Item 10) and share the right ones safely
9. 确保资源为对象所拥有。使用显示的“资源获取即初始化”，和智能指针。分配原始资源的时候，应该立即将其传递给属主对象。永远不要再一条语句中分配一个以上的资源。Ensure resources are owned by objects. Use explicit RAII and smart pointers。Don't saw by hand when you have power tools: C++'s "resource acquisition is initialization" (RAII) idiom is the power tool for correct resource handling. RAII allows the compiler to provide strong and automated guarantees that in other languages require fragile hand-coded idioms. When allocating a raw resource, immediately pass it to an owning object. Never allocate more than one resource in a single statement.

# 编程风格

1. 宁要编译时和连接时错误，也不要运行时候错误。能够再编译时作的事情，就不要推迟到运行时。Prefer compile- and link-time errors to run-time errors. Don't put off 'til run time what you can do at build time: Prefer to write code that uses the compiler to check for invariants during compilation, instead of checking them at run time. Run-time checks are control- and data-dependent, which means you'll seldom know whether they are exhaustive. In contrast, compile-time checking is not control- or data-dependent and typically offers higher degrees of confidence.
2. 积极使用const。Use const proactively。const is your friend: Immutable values are easier to understand, track, and reason about, so prefer constants over variables wherever it is sensible and make const your default choice when you define a value: It's safe, it's checked at compile time (see Item 14), and it's integrated with C++'s type system. Don't cast away const except to call a const-incorrect function.
3. 避免使用宏。Avoid macros. Macros are the bluntest instrument of C and C++'s abstraction facilities, ravenous wolves in functions' clothing, hard to tame, marching to their own beat all over your scopes. Avoid them.
4. 避免使用“魔数”避免在代码中使用42和3.14159这样的文字常量，可以用符号名称和表达式替换他们。Avoid magic numbers.Programming isn't magic, so don't incant it: Avoid spelling literal constants like 42 or 3.14159 in code. They are not self-explanatory and complicate maintenance by adding a hard-to-detect form of duplication. Use symbolic names and expressions instead, such as width \* aspectRatio.
5. 尽可能局部地声明变量，避免作用域膨胀，尽可能少地处理状态，变量的生存周期也是越短越好。Declare variables as locally as possible. Avoid scope bloat, as with requirements so too with variables): Variables introduce state, and you should have to deal with as little state as possible, with lifetimes as short as possible.
6. 总是初始化变量。Always initialize variables. Start with a clean slate: Uninitialized variables are a common source of bugs in C and C++ programs. Avoid such bugs by being disciplined about cleaning memory before you use it; initialize variables upon definition.
7. 避免函数过长，避免嵌套过深。Avoid long functions. Avoid deep nesting Short is better than long, flat is better than deep: Excessively long functions and nested code blocks are often caused by failing to give one function one cohesive responsibility (see Item 5), and both are usually solved by better refactoring
8. 避免跨编译单位的初始化依赖。因为其初始化顺序是未定义的。Avoid initialization dependencies across compilation units. Namespace-level objects in different compilation units should never depend on each other for initialization, because their initialization order is undefined. Doing otherwise causes headaches ranging from mysterious crashes when you make small changes in your project to severe non-portability even to new releases of the same compiler.
9. 尽量减少定义性依赖。避免循环依赖。如果forword declaration能够实现，那就不要#include定义。Minimize definitional dependencies. Avoid cyclic dependencies . Don't be over-dependent: Don't #include a definition when a forward declaration will do
10. 应该确保所编写的每个头文件都能独自进行编译，维持需要包含其内容所依赖的所有头文件。Make header files self-sufficient. Behave responsibly: Ensure that each header you write is compilable standalone, by having it include any headers its contents depend upon.
11. 总是编写内部#inculde保护符，绝不要编写外部#include保护符。为头文件添加保护，以避免在多次包含时重新定义。Always write internal #include guards. Never write external #include guards。Wear head(er) protection: Prevent unintended multiple inclusions by using #include guards with unique names for all of your header files.

# 函数与操作符

1. 正确地选择通过值、（智能）指针或者引用传递参数。在函数中始终用const限制只输入参数的指针和引用。Take parameters appropriately by value, (smart) pointer, or reference。Take parameters appropriately by value, (smart) pointer, or reference。
2. 保持重载操作符的自然语义，如果做到这点很困难，那么你可能已经无用了操作符重载。Preserve natural semantics for overloaded operators。Programmers hate surprises: Overload operators only for good reason, and preserve natural semantics; if that's difficult, you might be misusing operator overloading.
3. 优先使用算术操作符和复制操作符的标准形式使a@=b和a=a@具有相同的含义，实现这一目标就是用@=来定义@。Prefer the canonical forms of arithmetic and assignment operators. If you a+b, also a+=b: When defining binary arithmetic operators, provide their assignment versions as well, and write to minimize duplication and maximize efficiency.
4. 优先使用++和—的标准形式。优先调用++c前缀形式。Prefer the canonical form of ++ and --. Prefer calling the prefix forms。If you ++c, also c++: The increment and decrement operators are tricky because each has pre- and postfix forms, with slightly different semantics. Define operator++ and operator-- such that they mimic the behavior of their built-in counterparts. Prefer to call the prefix versions if you don't need the original value.
5. 考虑重载以避免隐含类型转换。Consider overloading to avoid implicit type conversions
6. 避免重载&&、||或，（逗号），内置的这些符号得到了编译器的特殊照顾，如果重载它们，它们就会变成普通函数，具有完全不同的语义，肯定会引入微妙的错误和缺陷。Avoid overloading &&, ||, or , (comma). Wisdom means knowing when to refrain: The built-in &&, || , and, (comma) enjoy special treatment from the compiler. If you overload them, they become ordinary functions with very different semantics (you will violate Items 26 and 31), and this is a sure way to introduce subtle bugs and fragilities. Don't overload these operators naïvely.
7. 不要编写依赖于函数参数求值顺序的代码。Don't write code that depends on the order of evaluation of function arguments

# 类的设计与继承

1. 弄清所要编写的是哪种类。Be clear what kind of class you're writing。Know thyself: There are different kinds of classes. Know which kind you are writing.
2. 用小类代替巨类。小类更易于编写，更易于保证正确、测试和使用，小类更有可能适用于各种不同的情况。不要用大杂烩式的类，它们要实现的概念既多又复杂。Prefer minimal classes to monolithic classes. Divide and conquer: Small classes are easier to write, get right, test, and use. They are also more likely to be usable in a variety of situations. Prefer such small classes that embody simple concepts instead of kitchen-sink classes that try to implement many and/or complex concepts
3. 用组合代替继承，继承会增加紧密的耦合是一种不良现象。Prefer composition to inheritance. Avoid inheritance taxes: Inheritance is the second-tightest coupling relationship in C++, second only to friendship. Tight coupling is undesirable and should be avoided where possible. Therefore, prefer composition to inheritance unless you know that the latter truly benefits your design.
4. 避免从并非要设计成基类的类中继承. Avoid inheriting from classes that were not designed to be base classes。Classes meant to be used standalone obey a different blueprint than base classes (see Item 32). Using a standalone class as a base is a serious design error and should be avoided. To add behavior, prefer to add nonmember functions instead of member functions (see Item 44). To add state, prefer composition instead of inheritance (see Item 34). Avoid inheriting from concrete base classes.
5. 优先提供抽象借口。有助于我们集中精力保证抽象的正确性，不至于收到现实或者状态管理细节的干扰。Prefer providing abstract interfaces. Love abstract art: Abstract interfaces help you focus on getting an abstraction right without muddling it with implementation or state management details. Prefer to design hierarchies that implement abstract interfaces that model abstract concepts.
6. 公用继承即可替换性。继承不是为了重用，而是为了被重用。Public inheritance is substitutability. Inherit, not to reuse, but to be reused. Know what: Public inheritance allows a pointer or reference to the base class to actually refer to an object of some derived class, without destroying code correctness and without needing to change existing code.
7. 实施安全的改写，改写一个虚拟函数时，应该保持可替换下，说的更具体一些就是要保持基类中函数的前后条件。不要改变虚拟函数的默认参数，应该显示地将改写函数重新声明为virtual，谨防在虚拟类中隐藏重载函数。Practice safe overriding。When overriding a virtual function, preserve substitutability; in particular, observe the function's pre- and post-conditions in the base class. Don't change default arguments of virtual functions. Prefer explicitly redeclaring overrides as virtual. Beware of hiding overloads in the base class.
8. 在基类中进行修改代价高昂，请将公用函数设为非虚拟。应该将虚拟函数设为私有的，或者派生类需要调用基类版本，则设为保护的。此建议不适用于析构函数。Consider making virtual functions nonpublic, and public functions nonvirtual. In base classes with a high cost of change (particularly ones in libraries and frameworks): Prefer to make public functions nonvirtual. Prefer to make virtual functions private, or protected if derived classes need to be able to call the base versions.
9. 要避免提供隐式转换，应该依赖的是显式转换（explicit构造函数和命名转换函数）Avoid providing implicit conversions. Not all change is progress: Implicit conversions can often do more damage than good. Think twice before providing implicit conversions to and from the types you define, and prefer to rely on explicit conversions (explicit constructors and named conversion functions).
10. 将数据成员设为私有的，struct除外。Make data members private, except in behaviorless aggregates (C-style structs) They're none of your caller's business: Keep data members private. Only in the case of simple C-style struct types that aggregate a bunch of values but don't pretend to encapsulate or provide behavior, make all data members public. Avoid mixes of public and nonpublic data, which almost always signal a muddled design.
11. 不要公开内部数据，避免返回类所管理的内部数据的句柄，这样累的客户就不会不受控制地修改对象自己拥有的状态。Don't give away your internals. Don't volunteer too much: Avoid returning handles to internal data managed by your class, so clients won't uncontrollably modify state that your object thinks it owns.
12. 明智地使用Pimpl，通过其惯用法使私有成员真正不可见，从而实现编译器防火墙，并提高信息隐藏度。Pimpl judiciously。Overcome the language's separation anxiety: C++ makes private members inaccessible, but not invisible. Where the benefits warrant it, consider making private members truly invisible using the Pimpl idiom to implement compiler firewalls and increase information hiding.
13. 尽可能将函数指定为非成员非友元函数。Prefer writing nonmember nonfriend functions。Avoid membership fees: Where possible, prefer making functions nonmember nonfriends
14. 总是一起提供new和delete进行分配操作和释放操作。Always provide new and delete together。They're a package deal: Every class-specific overload void\* operator new(parms) must be accompanied by a corresponding overload void operator delete(void\*, parms), where parms is a list of extra parameter types (of which the first is always std::size\_t). The same goes for the array forms new[] and delete[].
15. 如果提供类专门的new，应该提供所有的标准形式，普通plain，就地in-place和不抛出nothrow的重载。If you provide any class-specific new, provide all of the standard forms (plain, in-place, and nothrow)。Don't hide good news: If a class defines any overload of operator new, it should provide overloads of all three of plain, in-place, and non-throwing operator new. If you don't, they'll be hidden and unavailable to users of your class.

# 构造、析构、与复制

1. 成员变量初始化的顺序要与类定义中声明的顺序始终保持一致。Define and initialize member variables in the same order。Agree with your compiler: Member variables are always initialized in the order they are declared in the class definition; the order in which you write them in the constructor initialization list is ignored. Make sure the constructor code doesn't confusingly specify a different order.
2. 在构造函数中用初始化代替赋值。Prefer initialization to assignment in constructors. Set once, use everywhere: In constructors, using initialization instead of assignment to set member variables prevents needless run-time work and takes the same amount of typing.
3. 避免在构造函数和析构函数中调用虚拟函数。Avoid calling virtual functions in constructors and destructors。Virtual functions only "virtually" always behave virtually: Inside constructors and destructors, they don't. Worse, any direct or indirect call to an unimplemented pure virtual function from a constructor or destructor results in undefined behavior. If your design wants virtual dispatch into a derived class from a base class constructor or destructor, you need other techniques such as post-constructors.
4. 如果允许通过指向基类Base的指针执行删除才操作，则将基类析构函数设为公用且虚拟的，否则应该是保护且非虚拟的。Make base class destructors public and virtual, or protected and nonvirtual。To delete, or not to delete; that is the question: If deletion through a pointer to a base Base should be allowed, then Base's destructor must be public and virtual. Otherwise, it should be protected and nonvirtual.
5. 析构函数、释放和交换绝对不能失败。一切尝试都必须成功。Destructors, deallocation, and swap never fail. Everything they attempt shall succeed: Never allow an error to be reported from a destructor, a resource deallocation function (e.g., operator delete), or a swap function. Specifically, types whose destructors may throw an exception are flatly forbidden from use with the C++ standard library.
6. 如果定义了复制构造函数、复制赋值操作符或者析构函数中的任何一个，那么可能也需要定义另一个或者另外两个。Copy and destroy consistently. What you create, also clean up: If you define any of the copy constructor, copy assignment operator, or destructor, you might need to define one or both of the others.
7. 使用编译器生成的复制构造函数和赋值操作符；编写自己的版本；如果应该允许复制的话，显式地禁用前两者。Explicitly enable or disable copying. Copy consciously: Knowingly choose among using the compiler-generated copy constructor and assignment operator, writing your own versions, or explicitly disabling both if copying should not be allowed.
8. 在基类中，如果客户需要进行多态（完整的、深度的）复制的话，那么请考虑禁止复制构造函数和复制赋值操作符，而改为提供虚拟的Clone成员函数。）防止子类转换到父类时候损失部分自己的函数。Avoid slicing. Consider Clone instead of copying in base classes. Sliced bread is good; sliced objects aren't: Object slicing is automatic, invisible, and likely to bring wonderful polymorphic designs to a screeching halt. In base classes, consider disabling the copy constructor and copy assignment operator, and instead supplying a virtual Clone member function if clients need to make polymorphic (complete, deep) copies
9. Prefer the canonical form of assignment。When implementing operator=, prefer the canonical formnonvirtual and with a specific signature.
10. 只要可行，就提供不会失败的swap（而且要正确地提供）Whenever it makes sense, provide a no-fail swap (and provide it correctly)。swap is both a lightweight and a workhorse: Consider providing a swap function to efficiently and infallibly swap the internals of this object with another's. Such a function can be handy for implementing a number of idioms, from smoothly moving objects around to implementing assignment easily to providing a guaranteed commit function that enables strongly error-safe calling code.

# 命名空间与模块

1. 将类型及其非成员函数接口至于同一命名空间中。如果要将非成员函数（特别是操作符和辅助函数）设计成类x的接口的一部分，那么就必须在与x相同的命名空间中定义他们，以便正确调用。Keep a type and its nonmember function interface in the same namespace Nonmembers are functions too: Nonmember functions that are designed to be part of the interface of a class X (notably operators and helper functions) must be defined in the same namespace as the X in order to be called correctly.
2. 应该将类型和函数分别至于不同的命名空间中，除非有意想让它们一起工作。Keep types and functions in separate namespaces unless they're specifically intended to work together。Help prevent name lookup accidents: Isolate types from unintentional argument-dependent lookup (ADL, also known as Koenig lookup), and encourage intentional ADL, by putting them in their own namespaces (along with their directly related nonmember functions; see Item 57). Avoid putting a type into the same namespace as a templated function or operator.
3. 不要再头文件中或者#include之前编写命名空间using。头文件无法知道以后其他头文件会出现什么样的#inculde。Don't write namespace usings in a header file or before an #include. Namespace usings are for your convenience, not for you to inflict on others: Never write a using declaration or a using directive before an #include directive
4. 要避免在不同的模块中分配和释放内存，在一个模块中分配内存，而在另一个模块中释放它，会在这两个模块之间产生微妙的远距离依赖，使程序变得脆弱。Avoid allocating and deallocating memory in different modules
5. 不要在头文件中定义具有链接的实体，重复使用头文件将会导致这些实体链接时候错误或者内存分配的浪费。请将所有具有链接的实体放入实现文件。Don't define entities with linkage in a header file
6. 不要允许异常跨越模块边界传播。除非能够控制用来构建两段代码的编译器和编译选项；否则模块可能无法支持可兼容地实现一场传播。Don't allow exceptions to propagate across module boundaries
7. 在模块的接口中使用具有良好可移植性的类型。不要让类型出现在模块的外部接口中，除非能够确保所有的客户代码都能正确地理解该类型。应该使用客户代码能够理解的最高层抽象类型。Use sufficiently portable types in a module's interface

# 模板与泛型

1. 理智地结合静态多态性和动态多态性。理解它们的优缺点，善用它们的长处，结合两者以获得两方面的优势。Blend static and dynamic polymorphism judiciously
2. 有意地进行显式自定义，显式强过隐式，在编写模板时，应够有意地、正确地提供自定义点，并清晰地记入文档。Customize intentionally and explicitly
3. 不要特化函数模板。Don't specialize function templates
4. 不要无意地编写不通用的代码。依赖抽象而非细节，使用最通用、最抽象的方法来实现一个功能。Don't write unintentionally nongeneric code

# 错误处理与异常

1. 广泛地使用断言记录内部假设和不变式。Assert liberally to document internal assumptions and invariants。
2. Establish a rational error handling policy, and follow it strictly
3. 区别错误与非错误。违法约定就是错误，函数是一个工作单元。因此失败应该视为错误，或根据其对函数的影响而定。在函数f中，当且仅当是把违反了f的一个前调解，或者阻碍了f满足其他调用代码的任何前条件、实现f自己的任何后条件或者重新建立f有责任维持的不变式时，失败才是一个错误。
4. 设计和编写错误安全代码，In each function, give the strongest safety guarantee that won't penalize callers who don't need it. Always give at least the basic guarantee.
5. 优先使用异常报告错误。应该使用异常而不是错误码来报告错误。但不能使用异常时，对于错误以及布什错误的情况，可以使用错误状态码来报告异常。当不可能从错误中恢复或者不需要恢复时，可以使用其他方法，比如正常中指或者非正常终止。Prefer to use exceptions to report errors. When harmed, take exception: Prefer using exceptions over error codes to report errors. Use status codes (e.g., return codes, errno) for errors when exceptions cannot be used (see Item 62), and for conditions that are not errors. Use other methods, such as graceful or ungraceful termination, when recovery is impossible or not required.
6. 通过值抛出，通过引用捕获。学会正确catch：通过值而（非指针）抛出异常，通过引用捕获异常。这是与异常语义配合最佳的组合。当重新抛出相同异常时，应该优先使用throw；，避免使用throw e；Throw by value, catch by reference。Learn to catch properly: Throw exceptions by value (not pointer) and catch them by reference (usually to const). This is the combination that meshes best with exception semantics. When rethrowing the same exception, prefer just throw; to throw e;.
7. 正确地报告、处理和转换错误。什么时候说什么话：在检查出并确认是错误时报告错误。在能正确处理错误的最近一层处理或者转换每个错误。在能够正确处理错误的最近一层处理或者转换每个错误。Report, handle, and translate errors appropriately。Know when to say when: Report errors at the point they are detected and identified as errors. Handle or translate each error at the nearest level that can do it correctly
8. Avoid exception specifications。

# STL:容器

1. 默认时使用vector。否则选择其他合适的容器，如果有充分的理由使用某个特定容器类型，那就用好了。Use vector by default. Otherwise, choose an appropriate container
2. 用vector和string代替数组。不要使用c语言风格的数组指针运算和内存管理原语操作实现数组抽象。使用vector或者string不仅更轻松，而且还有助于编写更安全、伸缩性更好的软件。Use vector and string instead of arrays
3. 使用vector（和string：：c\_str）与非C++API交换数据），他俩是与非c++API通信的通道。Use vector (and string::c\_str) to exchange data with non-C++ APIs
4. 在容器中只存储值和智能指针。Store only values and smart pointers in containers
5. 用push\_back代替其他扩展序列的方式。如果不需要操心插入位置，就应该使用push\_back在序列中添加元素，其他方法可能极慢而且不简明。Prefer push\_back to other ways of expanding a sequence
6. 多用范围操作，少用单元素操作调用范围操作通常更易于编写，也更易于阅读，而且比显式循环的效率更高。Prefer range operations to single-element operations
7. 使用公认的惯用法真正地压缩容量，应该使用swap。真正地删除元素。应该使用erase-remaove法。Use the accepted idioms to really shrink capacity and really erase elements

# STL:算法

1. 安全第一，即使只在其中一个编译器平台上可用，即使只能在发行前的测试中使用，也仍然要使用带检查的stl实现Use a checked STL implementation
2. 用算法调用代替手工编写的循环，可用使表达力更强、维护性更好，更不易出错，而且同样高效。Prefer algorithm calls to handwritten loops
3. 使用正确的stl查找算法。Use the right STL search algorithm
4. 使用正确的stl排序算法。选择能够实现所需而开销最低的算法。Use the right STL sort algorithm
5. 使谓词成为纯函数，如果函数的结果只取觉于其参数。则该函数就是一个纯函数。Make predicates pure functions
6. 算法和比较器的参数应多用函数对象少用函数。正确编写函数对象。类型安全Prefer function objects over functions as algorithm and comparer arguments
7. Write function objects correctly。Be cheap, be adaptable: Design function objects to be values that are cheap to copy. Where possible, make them adaptable by inheriting from unary\_- or binary\_function
8. 避免使用类型分支，多使用多态。Write function objects correctly
9. 依赖类型，而非其表示方式。不要对对象在内存中的准确表示方法做任何假设。相反，应该让类型决定如何在内存中读写其对象。Avoid type switching; prefer polymorphism
10. Rely on types, not on representations
11. 避免对指针使用static\_cast。安全替代方法很多包括使用dynamic\_cast。Avoid using reinterpret\_cast
12. 避免强制转换cosnt。Avoid using static\_cast on pointers
13. 不要使用c风格的强制转换。Avoid casting away const
14. 不要对非POD进行memcpy操作或者memcmp操作。除非有什么对象的布局就是原始内存。Don't use C-style casts
15. 不要在union中写入一个成员而读取另一个成员方式获取强制转换，这笔reinterpret\_cast更阴险，也更难预测。Don't memcpy or memcmp non-PODs
16. 不要使用可变长参数 (…)。省略会导致崩溃：省略号是来自c语言的危险遗产。要避免使用可变长参数，应改用高级的c++结构和库。Don't use unions to reinterpret representation
17. 不要使用失效对象。不要使用不安全函数。Don't use invalid objects. Don't use unsafe functions
18. 不要多态地处理数组。数组的可调整性很差，多态地处理数组是绝对的类型错误，而且编译器有可能不会做出任何提示。不要掉入这一陷阱。Don't treat arrays polymorphically