C++程序设计语言（特别版）

第1章　忠告

这里是一组在你学习C++的过程中或许应该考虑的“规则”。随着你变得更加熟练，你将能把它转化为某种更适合你的那类应用系统或者你自己的程序设计风格的东西。它们有意被写得很简单，因此都缺乏细节。请不要太拘泥于它们的字面意义。要写出一个好程序需要智慧、品味和耐性。你不会第一次就能把它搞好的。试验！

[1] 在编程序时，你是在为你针对某个问题的解决方案中的思想建立起一种具体表示。让程序的结构尽可能地直接反映这些思想：

[a] 如果你能把“它”看成一个独立的概念，就把它做成一个类。

[b] 如果你能把“它”看成一个独立的实体，就把它做成某个类的一个对象。

[c] 如果两个类有共同的界面，将此界面做成一个抽象类。

[d] 如果两个类的实现有某些显著的共同东西，将这些共性做成一个基类。

[e] 如果一个类是一种对象的容器，将它做成一个模板。

[f] 如果一个函数实现对某容器的一个算法，将它实现为对一族容器可用的模板函数。

[g] 如果一组类、模板等互相之间有逻辑联系，将它们放进一个名字空间里。

[2] 在你定义一个并不是实现某个像矩阵或复数这样的数学对象的类时，或者定义一个低层的类型如链接表的时候：

[a] 不要使用全局数据（使用成员）。

[b] 不要使用全局函数。

[c] 不要使用公用数据成员。

[d] 不要使用友元，除非为了避免 [a] 或 [c]。

[e] 不要在一个类里面放“类型域”；采用虚函数。

[f] 不要使用在线函数，除非作为效果显著的优化。

更特殊或更详尽的实用规则可以在每章最后的“忠告”一节里找到。请记住，这些忠告只是粗略的实用规则，而不是万古不变的定律。它们只应使用在“合理的地方”。从来就没有任何东西能够替代智慧、经验、常识和好的鉴赏力。

我发现具有“绝不要做这个”形式的规则不大有帮助。因此，大部分忠告被写成应该做什么的建议，而否定性的建议也倾向于不采用绝对禁止的短语。据我所知，没有任何一种主要的C++特征没有被良好地使用过。在有关“忠告”的节里不包括解释，相反，每条忠告都引用了本书中某些适当的章节。在给出否定性的忠告时，对应章节里通常都提供了有关其他替代方式的建议。

第2章　忠告

[1] 不用害怕，一切都会随着时间的推移而逐渐明朗起来； 2.1节。
[2] 你并不需要在知道了C++的所有细节之后才能写出好的C++程序；1.7节。
[3] 请特别关注程序设计技术，而不是各种语言特征； 2.1节。

第3章　忠告

[1] 不要像重新发明车轮那样企图做每件事；去使用库。
[2] 不要相信奇迹；要理解你的库能做什么，它们如何做，它们做时需要多大的代价。
[3] 当你遇到一个选择时，应该优先选择标准库而不是其他的库。
[4] 不要认为标准库对于任何事情都是最理想的。
[5] 切记 *#include*你所用到的功能的头文件； 3.3节。
[6] 记住，标准库的功能定义在名字空间 *std*之中；3.3节。
[7] 请用*string*，而不是*char**；3.5节、3.6节。
[8] 如果怀疑，就用一个检查区间范围的向量（例如 *Vec*）；3.7.2节。
[9] *vector*<*T*>、*list*<*T*> 和*map*<*key, value*> 都比*T*[ ] 好；3.7.1节、3.7.3节、3.7.4节。
[10] 如要向一个容器中添加一个元素，用*push_back*( ) 或*back_inserter*( )；3.7.3节、3.8节。
[11] 采用对*vector*的*push_back*( )，而不是对数组的*realloc*( )；3.8节。
[12] 在*main*( ) 中捕捉公共的异常；3.7.2节。

第4章　忠告

[1] 保持较小的作用域；4.9.4节。
[2] 不要在一个作用域和它外围的作用域里采用同样的名字； 4.9.4节。
[3] 在一个声明中（只）声明一个名字；4.9.2节。
[4] 让常用的和局部的名字比较短，让不常用的和全局的名字比较长； 4.9.3节。
[5] 避免看起来类似的名字；4.9.3节。
[6] 维持某种统一的命名风格；4.9.3节。
[7] 仔细选择名字，反映其意义而不是反映实现方式； 4.9.3节；
[8] 如果所用的内部类型表示某种可能变化的值，请用 *typedef*为它定义一个有意义的名字；4.9.7节。
[9] 用*typedef*为类型定义同义词，用枚举或类去定义新类型； 4.9.7节。
[10] 切记每个声明中都必须描述一个类型（没有“隐式的 *int*”）；4.9.1节。
[11] 避免有关字符数值的不必要假设；4.3.1节、C.6.2.1节。
[12] 避免有关整数大小的不必要假设；4.6节。
[13] 避免有关浮点类型表示范围的不必要假设； 4.6节。
[14] 优先使用普通的*int*而不是*short int*或者*long int*；4.6节。
[15] 优先使用*double*而不是*float*或者*long double*；4.5节。

[16] 优先使用普通的*char*而不是*signed char*或者*unsigned char*；C.3.4节。
[17] 避免做出有关对象大小的不必要假设；4.6节。
[18] 避免无符号算术；4.4节。
[19] 应该带着疑问去看待从*signed*到*unsigned*，或者从*unsigned*到*signed*的转换；C.6.2.6节。
[20] 应该带着疑问去看待从浮点到整数的转换；C.6.2.6节。
[21] 应该带着疑问去看待向较小类型的转换，如将 *int*转换到*char*；C.6.2.6节。

第5章　忠告

[1] 避免非平凡的指针算术；5.3节。
[2] 当心，不要超出数组的界线去写；5.3.1节。
[3] 尽量使用*0*而不是*NULL*；5.1.1节。
[4] 尽量使用*vector*和*valarray*而不是内部（C风格）的数组；5.3.1节。
[5] 尽量使用*string*而不是以0结尾的*char*数组；5.3节。
[6] 尽量少用普通的引用参数；5.5节。
[7] 避免*void**，除了在某些低级代码里；5.6节。
[8] 避免在代码中使用非平凡的文字量（“神秘的数”）。相反，应该定义和使用各种符号常量。4.8节、5.4节。

第6章　忠告

[1] 应尽可能使用标准库，而不是其他的库和“手工打造的代码”；6.1.8节。
[2] 避免过于复杂的表达式；6.2.3节。
[3] 如果对运算符的优先级有疑问，加括号；6.2.3节。
[4] 避免显式类型转换（强制）；6.2.7节。
[5] 若必须做显式类型转换，提倡使用特殊强制运算符，而不是C风格的强制；6.2.7节。
[6] 只对定义良好的构造使用 *T*(*e*) 记法；6.2..8节。
[7] 避免带有无定义求值顺序的表达式；6.2.2节。
[8] 避免*goto*；6.3.4节。
[9] 避免do语句；6.3.3节。
[10] 在你已经有了去初始化某个变量的值之前，不要去声明它；6.3.1节、6.3.2.1节、6.3.3.1节。
[11] 使注释简洁、清晰、有意义；6.4节。
[12] 保持一致的缩进编排风格；6.4节。
[13] 倾向于去定义一个成员函数*operator new*( )（15.6节）去取代全局的*operator new*( )；6.2.6.2节。
[14] 在读输入的时候，总应考虑病态形式的输入；6.1.3节。

第7章　忠告

[1] 质疑那些非*const*的引用参数；如果你想要一个函数去修改其参数，请使用指针或者返回值；5.5节。

[2] 当你需要尽可能减少参数复制时，应该使用 *const*引用参数；5.5节。
[3] 广泛而一致地使用*const*；7.2节。
[4] 避免宏；7.8节。
[5] 避免不确定数目的参数；7.6节。
[6] 不要返回局部变量的指针或者引用；7.3节。
[7] 当一些函数对不同的类型执行概念上相同的工作时，请使用重载；7.4节。
[8] 在各种整数上重载时，通过提供函数去消除常见的歧义性；7.4.3节。
[9] 在考虑使用指向函数的指针时，请考虑虚函数（2.5.5节）或模板（2..7.2节）是不是一种更好的选择；7.7节。
[10] 如果你必须使用宏，请使用带有许多大写字母的丑陋的名字；7.8节。

第8章　忠告

[1] 用名字空间表示逻辑结构；8.2节。
[2] 将每个非局部的名字放入某个名字空间里，除了 *main*( ) 之外；8.2节。
[3] 名字空间的设计应该让你能很方便地使用它，而又不会意外地访问了其他的无关名字空间；8.2.4节。
[4] 避免对名字空间使用很短的名字；8.2.7节。
[5] 如果需要，通过名字空间别名去缓和长名字空间名的影响；8.2.7节。
[6] 避免给你的名字空间的用户添加太大的记法负担；8.2.2节、8.2.3节。
[7] 在定义名字空间的成员时使用 *namespace*::*member*的形式；8.2.8节。
[8] 只在转换时，或者在局部作用域里，才用 *using namespace*；8.2.9节。
[9] 利用异常去松弛“错误”处理代码和正常处理代码之间的联系；8.3.3节。
[10] 采用用户定义类型作为异常，不用内部类型；8.3.2节。
[11] 当局部控制结构足以应付问题时，不要用异常；8.3.3.1节。

第9章　忠告

[1] 利用头文件去表示界面和强调逻辑结构；9.1节、9.3.2节。
[2] 用 #*include*将头文件包含到实现有关功能的源文件里；9.3.1节。
[3] 不要在不同编译单位里定义具有同样名字，意义类似但又不同的全局实体；9.2节。
[4] 避免在头文件里定义非*inline*函数；9.2.1节。
[5] 只在全局作用域或名字空间里使用 #*include*；9.2..1节。
[6] 只用 #*include*包含完整的定义；9.2.1节。
[7] 使用包含保护符；9.3.3节。
[8] 用 #*include* 将C头文件包含到名字空间里，以避免全局名字；8.2.9.1节、9..2.2节。
[9] 将头文件做成自给自足的；9.2.3节。
[10] 区分用户界面和实现界面；9.3.2节。

[11] **区分一般用户界面和专家用户界面；** 9.3.2**节。**

[12] **在有意向用于非** C++ **程序组成部分的代码中，应避免需要运行时初始化的非局部对象；** 9.4.1**节。**

第10章　忠告

[1] **用类表示概念；** 10.1**节。**

[2] **只将*public*数据（*struct*）用在它实际上仅仅是数据，而且对于这些数据成员并不存在不变式的地方；** 10.2.8**节。**

[3] **一个具体类型属于最简单的类。如果适用的话，就应该尽可能使用具体类型，而不要采用更复杂的类，也不要用简单的数据结构；** 10.3**节。**

[4] **只将那些需要直接访问类的表示的函数作为成员函数；** 10.3.2**节。**

[5] **采用名字空间，使类与其协助函数之间的关系更明确；** 10.3.2**节。**

[6] **将那些不修改对象值的成员函数做成*const*成员函数；** 10.2..6**节。**

[7] **将那些需要访问类的表示，但无须针对特定对象调用的成员函数做成*static*成员函数；** 10.2.4**节。**

[8] **通过构造函数建立起类的不变式；** 10.3.1**节。**

[9] **如果构造函数申请某种资源，析构函数就应该释放这一资源；** 10.4.1**节。**

[10] **如果在一个类里有指针成员，它就需要有复制操作（包括复制构造函数和复制赋值）；** 10.4.4.1**节。**

[11] **如果在一个类里有引用成员，它就可能需要有复制操作（复制构造函数和复制赋值）；** 10.4.6.3**节。**

[12] **如果一个类需要复制操作或析构函数，它多半还需要有构造函数、析构函数、复制赋值和复制构造函数；** 10.4.4.1**节。**

[13] **在复制赋值里需要检查自我赋值；** 10.4.4.1**节。**

[14] **在写复制构造函数时，请小心地复制每个需要复制的元素（当心默认的初始式）；** 10.4.4.1**节。**

[15] **在向某个类中添加新成员时，一定要仔细检查，看是否存在需要更新的用户定义构造函数，以使它能够初始化新成员；** 10.4.6.3**节。**

[16] **在类声明中需要定义整型常量时，请使用枚举；** 10.4.6.1**节。**

[17] **在构造全局的和名字空间的对象时，应避免顺序依赖性；** 10.4.9**节。**

[18] **用第一次开关去缓和顺序依赖性问题；** 10.4.9**节。**

[19] **请记住，临时对象将在建立它们的那个完整表达式结束时销毁；** 10.4.10**节。**

第11章　忠告

[1] **定义运算符主要是为了模仿习惯使用方式；** 11.1**节。**

[2] **对于大型运算对象，请使用*const*引用参数类型；** 11.6**节。**

[3] **对于大型的结果，请考虑优化返回方式；** 11.6**节。**

[4] 如果默认复制操作对一个类很合适，最好是直接用它； 11.3.4节。
[5] 如果默认复制操作对一个类不合适，重新定义它，或者禁止它； 11.2.2节。
[6] 对于需要访问表示的操作，优先考虑作为成员函数而不是作为非成员函数； 11.5.2节。
[7] 对于不访问表示的操作，优先考虑作为非成员函数而不是作为成员函数； 11.5.2节。
[8] 用名字空间将协助函数与“它们的”类关联起来； 11.2.4节。
[9] 对于对称的运算符采用非成员函数； 11.3.2节。
[10] 用 ( ) 作为多维数组的下标； 11.9节。
[11] 将只有一个“大小参数”的构造函数做成 *explicit*；11.7.1节。
[12] 对于非特殊的使用，最好是用标准 *string*（第20章）而不是你自己的练习； 11.12节。
[13] 要注意引进隐式转换的问题； 11.4节。
[14] 用成员函数表达那些需要左值作为其左运算对象的运算符； 11.3.5节。

第12章　忠告

[1] 避免类型域； 12.2.5节。
[2] 用指针和引用避免切割问题； 12.2.3节。
[3] 用抽象类将设计的中心集中到提供清晰的界面方面； 12.3节。
[4] 用抽象类使界面最小化； 12.4.2节。
[5] 用抽象类从界面中排除实现细节； 12.4.2节。
[6] 用虚函数使新的实现能够添加进来，又不会影响用户代码； 12.4.1节。
[7] 用抽象类去尽可能减少用户代码的重新编译； 12.4.2节。
[8] 用抽象类使不同的实现能够共存； 12.4.3节。
[9] 一个有虚函数的类应该有一个虚析构函数； 12.4.2节。
[10] 抽象类通常不需要构造函数； 12.4.2节。
[11] 让不同概念的表示也不相同； 12.4.1.1节。

第13章　忠告

[1] 用模板描述需要使用到许多参数类型上去的算法； 13.3节。
[2] 用模板表述容器； 13.2节。
[3] 为指针的容器提供专门化，以减小代码规模； 13.5节。
[4] 总是在专门化之前声明模板的一般形式； 13.5节。
[5] 在专门化的使用之前先声明它； 13.5节。
[6] 尽量减少模板定义对于实例化环境的依赖性； 13.2.5节、C..13.8节。
[7] 定义你所声明的每一个专门化； 13.5节。
[8] 考虑一个模板是否需要有针对 C风格字符串和数组的专门化； 13.5.2节。
[9] 用表述策略的对象进行参数化； 13.4节。
[10] 用专门化和重载为同一概念的针对不同类型的实现提供统一界面； 13.5节。

[11] 为简单情况提供简单界面，用重载和默认参数去表述不常见的情况； 13.5节、13.4节。
[12] 在修改为通用模板之前，在具体实例上排除程序错误； 13.2.1节。
[13] 如果模板定义需要在其他编译单位里访问，请记住写 *export*；13.7节。
[14] 对大模板和带有非平凡环境依赖性的模板，应采用分开编译的方式； 13.7节。
[15] 用模板表示转换，但要非常小心地定义这些转换； 13.6.3..1节。
[16] 如果需要，用*constraint*( ) 成员函数给模板的实参增加限制； 13.9[16]，C.13..10节。
[17] 通过显式实例化减少编译和连接时间； C.13.10节。
[18] 如果运行时的效率非常重要，那么最好用模板而不是派生类； 13.6.1节。
[19] 如果增加各种变形而又不重新编译是很重要的，最好用派生类而不是模板； 13.6.1节。
[20] 如果无法定义公共的基类，最好用模板而不是派生类； 13.6.1节。
[21] 当有兼容性约束的内部类型和结构非常重要时，最好用模板而不是派生类； 13.6.1节。

第14章 忠告

[1] 用异常做错误处理；14.1节、14.5节、14.9节。
[2] 当更局部的控制机构足以应付时，不要使用异常； 14.1节。
[3] 采用“资源申请即初始化”技术去管理资源； 14.4节。
[4] 并不是每个程序都要求具有异常时的安全性； 14.4.3节。
[5] 采用“资源申请即初始化”技术和异常处理器去维持不变式； 14.3.2节。
[6] 尽量少用try块，用“资源申请即初始化”技术，而不是显式的处理器代码； 14.4节。
[7] 并不是每个函数都需要处理每个可能的错误； 14.9节。
[8] 在构造函数里通过抛出异常指明出现失败； 14.4.6节。
[9] 在从赋值中抛出异常之前，使操作对象处于合法状态； 14.4.6.2节。
[10] 避免从析构函数里抛出异常；14.4.7节。
[11] 让*main*( ) 捕捉并报告所有的异常；14.7节。
[12] 使正常处理代码和错误处理代码相互分离； 14.4.5节、14.5节。
[13] 在构造函数里抛出异常之前，应保证释放在此构造函数里申请的所有资源； 14.4节。
[14] 使资源管理具有层次性；14.9节。
[15] 对于主要界面使用异常描述；14.9节。
[16] 当心通过*new*分配的内存在发生异常时没有释放，并由此而导致存储的流失； 14.4.1节、14.4.2节、14.4.4节。
[17] 如果一函数可能抛出某个异常，就应假定它一定会抛出这个异常； 14.6节。
[18] 不要假定所有异常都是由*exception*类派生出来的；14.10节。
[19] 库不应该单方面终止程序。相反，应该抛出异常，让调用者去做决定； 14.1节。
[20] 库不应该生成面向最终用户的错误信息。相反，它应该抛出异常，让调用者去做决定；14.1节。
[21] 在设计的前期开发出一种错误处理策略； 14.9节。

第15章　忠告

[1] 利用常规的多重继承表述特征的合并；15.2节、15.2.5节。
[2] 利用多重继承完成实现细节与界面的分离；15.2.5节。
[3] 用*virtual*基类表达在类层次结构里对某些类（不是全部类）共同的东西；15.2.5节。
[4] 避免显式的类型转换（强制）；15.4.5节。
[5] 在不可避免地需要漫游类层次结构的地方，使用 *dymamic_cast*；15.4.1节。
[6] 尽量用*dynamic_cast*而不是*typeid*；15.4.4节。
[7] 尽量用*private*而不是*protected*；15.3.1.1节。
[8] 不要声明*protected*数据成员；15.3.1.1节。
[9] 如果某个类定义了*operator delete*( )，它也应该有虚析构函数；15.6节；
[10] 在构造和析构期间不要调用虚函数；15.4.3节。
[11] 尽量少用为解析成员名而写的显式限定词，最好是在覆盖函数里用它；15.2.1节。

第16章　忠告

[1] 利用标准库功能，以维持可移植性；16.1节。
[2] 决不要另行定义标准库的功能；16.1.2节。
[3] 决不要认为标准库比什么都好。
[4] 在定义一种新功能时，应考虑它是否能够纳入标准库所提供的框架中；16.3节。
[5] 记住标准库功能都定义在名字空间*std*里；16.1.2节。
[6] 通过包含标准库头文件声明其功能，不要自己另行显示声明；16.1.2节。
[7] 利用后续抽象的优点；16.2.1节。
[8] 避免肥大的界面；16.2.2节。
[9] 与自己写按照反向顺序的显式循环相比，最好是写利用反向迭代器的算法；16.3.2节。
[10] 用*base*( ) 从*reverse_iterator*抽取出*iterator*；16.3.2节。
[11] 通过引用传递容器；16.3.4节。
[12] 用迭代器类型，如*list*<*char*>::*iterator*，而不要采用索引容器元素的指针；16.3.1节。
[13] 在不需要修改容器元素时，使用 *const*迭代器；16.3.1节。
[14] 如果希望检查访问范围，请（直接或间接）使用 *at*( )；16.3.3节。
[15] 多用容器和*push_back*( ) 或*resize*( )，少用数组和*realloc*( )；16.3.5节。
[16] *vector*改变大小之后，不要使用指向其中的迭代器；16.3.8节。
[17] 利用*reserve*( ) 避免使迭代器非法；16.3.8节。
[18] 在需要的时候，*reserve*( ) 可以使执行情况更容易预期；16.3.8节。

第17章　忠告

[1] 如果需要用容器，首先考虑用 *vector*；17.1节。
[2] 了解你经常使用的每个操作的代价（复杂性，大 O度量）；17.1.2节。

[3] 容器的界面、实现和表示是不同的概念，不要混淆； 17.1..3节。
[4] 你可以依据多种不同准则去排序和搜索； 17.1.4.1节。
[5] 不要用C风格的字符串作为关键码，除非你提供了一种适当的比较准则； 17.1.4.1节。
[6] 你可以定义这样的比较准则，使等价的但是不相同的关键码值映射到同一个关键码； 17.1.4.1节。
[7] 在插入和删除元素时，最好是使用序列末端的操作（*back*操作）；17.1.4.1节。
[8] 当你需要在容器的前端或中间做许多插入和删除时，请用 *list*；17.2.2节。
[9] 当你主要通过关键码访问元素时，请用 *map*或*multimap*；17.4.1节。
[10] 尽量用最小的操作集合，以取得最大的灵活性； 17.1.1节。
[11] 如果要保持元素的顺序性，选用 *map*而不是*hash_map*；17.6.1节。
[12] 如果查找速度极其重要，选 *hash_map*而不是*map*；17.6.1节。
[13] 如果无法对元素定义小于操作时，选 *hash_map*而不是*map*；17.6.1节。
[14] 当你需要检查某个关键码是否在关联容器里的时候，用 *find*( )；17.4.1.6节。
[15] 用*equal_range*( ) 在关联容器里找出所有具有给定关键码的所有元素； 17.4..1.6节。
[16] 当具有同样关键码的多个值需要保持顺序时，用 *multimap*；17.4.2节。
[17] 当关键码本身就是你需要保存的值时，用 *set*或*multiset*；17.4.3节。

第18章　忠告

[1] 多用算法，少用循环； 18.5节。
[2] 在写循环时，考虑是否能将它表述为一个通用的算法； 18.2节。
[3] 常规性地重温算法集合，看看是不是能将新应用变得更明晰； 18.2节。
[4] 保证一对迭代器参数确实表述了一个序列； 18.3.1节。
[5] 设计时应该让使用最频繁的操作是简单而安全的； 18.3节、18.3.1节。
[6] 把测试表述成能够作为谓词使用的形式； 18.4.2节。
[7] 切记谓词是函数和对象，不是类型； 18.4.2节。
[8] 你可以用约束器从二元谓词做出一元谓词； 18.4.4.1节。
[9] 利用*mem_fun*( ) 和*mem_fun_ref*( ) 将算法应用于容器； 18.4.4.2节。
[10] 当你需要将一个参数约束到一个函数上时，用 *ptr_fun*( )；18.4.4.3节。
[11] 切记*strcmp*( )用*0*表示“ 相等 ”，与 == 不同；18.4.4.4节。
[12] 仅在没有更特殊的算法时，才使用 *for_each*( ) 和*tranform*( )；18.5.1节。
[13] 利用谓词，以便能以各种比较准则和相等准则使用算法； 18.4.2.1节、18.6.3.1节。
[14] 利用谓词和其他函数对象，以使标准算法能用于表示范围广泛的意义； 18.4.2节。
[15] 运算符 < 和 == 在指针上的默认意义很少适用于标准算法； 18.6.3.1节。
[16] 算法并不直接为它们的参数序列增加或减少元素； 18.6节。
[17] 应保证用于同一个序列的小于和相等谓词相互匹配； 18.6..3.1节。
[18] 有时排好序的序列用起来更有效且幽雅； 18.7节。

[19] 仅为兼容性而使用*qsort*( ) 和*bsearch*( )；18.11节。

第19章　忠告

[1] 在写一个算法时，设法确定需要用哪种迭代器才能提供可接受的效率，并（只）使用这种迭代器所支持的操作符去表述算法；19.2.1节。

[2] 当给定的迭代器参数提供了多于算法所需的最小支持时，请通过重载为该算法提供效率更高的实现；19.2.3节。

[3] 利用*iterator_traits*为不同迭代器类别描述适当的算法；19.2.2节。

[4] 记住在*istream_iterator*和*ostream_iterator*的访问之间使用 ++；19.2.6节。

[5] 用插入器避免容器溢出；19.2.4节。

[6] 在排错时使用额外的检查，后面只在必须时才删除这些检查；19.3.1节。

[7] 多用 ++*p*，少用*p*++；19.3节。

[8] 使用未初始化的存储去改善那些扩展数据结构的算法的性能；19.4.4节。

[9] 使用临时缓冲区去改善需要临时数据结构的算法的性能；19.4.4节。

[10] 在写自己的分配器之前三思；19.4节。

[11] 避免*malloc*( )、*free*( )、*realloc*( ) 等；19.4.6节。

[12] 你可以通过为*rebind*所用的技术去模拟对模板的*typedef*；19.4.1节。

第20章　忠告

[1] 尽量使用*string*操作，少用C风格字符串函数；20.4.1节。

[2] 用*string*作为变量或者成员，不作为基类；20.3节、25.2.1节。

[3] 你可以将*string*作为参数值或者返回值，让系统去关心存储管理问题；20.3.6节。

[4] 当你希望做范围检查时，请用 *at*( ) 而不是迭代器或者 [ ]；20.3.2节、20.3.5节。

[5] 当你希望优化速度时，请用迭代器或 [ ] 而不是*at*( )；20.3.2节、20.3.5节。

[6] 直接或者间接地使用*substr*( ) 去读子串，用*replace*( ) 去写子串；20.3.12节、20.3.13节。

[7] 用*find*( ) 操作在*string*里确定值的位置（而不是写一个显式的循环）；20.3.11节。

[8] 在你需要高效率地添加字符时，请在 *string*的后面附加；20.3.9节。

[9] 在没有极端时间要求情况下用 *string*作为字符输入的目标；20.3.15节。

[10] 用*string*::*npos*表示“ *string*的剩余部分 ”；20.3.5节。

[11] 如果必要，就采用低级操作去实现极度频繁使用的 *string*（而不是到处用低级数据结构）；20.3.10节。

[12] 如果你使用*string*，请在某些地方捕捉*length_error*和*out_of_range*异常；20.3.5节。

[13] 小心，不要将带值*0*的*char** 传递给字符串函数；20.3.7节。

[14] 只是到必须做的时候，（再）用*c_str*( ) 产生*string*的C风格表示；20.3.7节。

[15] 当你需要知道字符的类别时，用 *isalpha*( )、*isdigit*( ) 等函数，不要自己去写对字符值的检测；20.4..1节。

第21章　忠告

[1] 在为用户定义类型的值定义 << 和 >> 时，应该采用意义清晰的正文表示形式； 21.2.3节、21.3.5节。
[2] 在打印包含低优先级运算符的表达式时需要使用括号； 21.2节。
[3] 在添加新的 << 和 >> 运算符时，你不必修改*istream*或*ostream*；21.2.3节。
[4] 你可以定义函数，使其能基于第二个（或更后面的）参数，具有像 *virtual*函数那样行为； 21.2.3.1节。
[5] 切记，按默认约定 >> 跳过所有空格；21.3.2节。
[6] 使用低级输入函数（如*get*( ) 和*read*( )）主要是为了实现高级输入函数；21.3.4节。
[7] 在使用*get*( )、*getline*( ) 和*read*( ) 时留心其终止准则；21.3.4节。
[8] 在控制I/O时，尽量采用操控符，少用状态标志；21.3.3节、21.4节、21.4.6节。
[9] （只）用异常去捕捉罕见的I/O错误；21.3.6节。
[10] 联结用于交互式I/O的流；21.3.7节。
[11] 使用哨位将许多函数的入口和出口代码集中到一个地方； 21.3.8节。
[12] 在无参数操控符最后不要写括号； 21.4.6.2节。
[13] 使用标准操控符时应记住写 *#include* <*iomanip*>；21.4.6.2节。
[14] 你可以通过定义一个简单函数对象得到三元运算符的效果（和效率）； 21.4.6.3节。
[15] 切记，*width* 描述只应用于随后的一个I/O操作；21.4.4节。
[16] 切记*precision*描述只对随后所有的浮点数输出操作有效； 21.4.3节。
[17] 用字符串流做内存里的格式化； 21.5.3节。
[18] 你可以描述一个文件流的模式； 21.5.1节。
[19] 在扩充I/O系统时，应该清楚地区分格式化（ *iostream*）和缓冲（*streambuf*）；21.1节、21.6节。
[20] 将传输值的非标准方式实现为流缓冲区； 21.6.4节。
[21] 将格式化值的非标准方式实现为流操作； 21.2.3节、21.3.5节。
[22] 你可以利用一对函数隔离和封装起对用户定义代码的调用； 21.6.4节。
[23] 你可以在读入之前用*in_avail*( ) 去确定输入操作是否会被阻塞； 21.6.4节。
[24] 划分清楚需要高效的简单操作和实现某种策略的操作（将前者做成 *inline*，将后者做成*virtual*）；21.6.4节。
[25] 用*locale*将“文化差异”局部化；21.7节。
[26] 用*sync_with_stdio*(x) 去混合C风格和C++ 风格的I/O，或者离解C风格和C++ 风格的I/O； 21.8节。
[27] 当心C风格I/O的类型错误；21.8节。

第22章　忠告

[1] 数值问题常常很微妙。如果你对数值问题的数学方面不是100%有把握，请去找专家或者做试验；22.1节。

[2] 用*numeric_limits*去确定内部类型的性质；22.2节。
[3] 为用户定义的标量类型描述*numeric_limits*；22.2节。
[4] 如果运行时效率比对于操作和元素的灵活性更重要的话，那么请用 *valarray*去做数值计算；22.4节。
[5] 用切割表述在数组的一部分上的操作，而不是用循环；22.4.6节。
[6] 利用组合器，通过清除临时量和更好的算法来获得效率；22.4.7节。
[7] 用*std*::*complex*做复数算术；22.5节。
[8] 你可以把使用*complex*类的老代码通过一个*typedef*转为用*std*::*complex*模板；22.5节。
[9] 在写循环从一个表出发计算某个值之前，先考虑一下 *accumulate*( )、*inner_product*( )、*partial_sum*( ) 和*adjacent_difference*( )；22.6节。
[10] 最好是用具有特定分布的随机数类，少直接用 *rand*( )；22.7节。
[11] 注意使你的随机数充分随机；22.7节。

第23章 忠告

[1] 知道你试图达到什么目的；23.3节。
[2] 心中牢记软件开发是一项人的活动；23.2节、23.5.3节。
[3] 用类比来证明是有意的欺骗；23.2节。
[4] 保持一个特定的实实在在的目标；23.4节。
[5] 不要试图用技术方式去解决社会问题；23.4节。
[6] 在设计和对待人员方面都应该有长期考虑；23.4.1节、23.5.3节。
[7] 对于什么程序在编码之前先行设计是有意义的，在程序规模上并没有下限；23.2节。
[8] 设计过程应鼓励反馈；23.4节。
[9] 不要将做事情都当做取得了进展；23.3节、23.4节。
[10] 不要推广到超出了所需要的、你已有直接经验的和已经测试过的东西；23.4.1节、23.4.2节。
[11] 将概念表述为类；23.4.2节、23.4.3.1节。
[12] 系统里也存在一些不应该用类表述的性质；23.4.3.1节。
[13] 将概念间的层次关系用类层次结构表示；23.4.3.1节
[14] 主动到应用和实现中去寻找概念间的共性，将由此得到的一般性概念表示为基类；23.4.3.1节、23.4.3.5节。
[15] 在其他领域中的分类方式未必适合作为应用中的继承模型的分类方式；23.4.3.1节。
[16] 基于行为和不变式设计类层次结构；23.4.3.1节、23.4.3.5节、23.4.3.7.1节。
[17] 考虑用例；23.4.3.1节。
[18] 考虑使用CRC卡片；23.4.3.1节。
[19] 用现存系统作为模型、灵感的源泉和出发点；23.4.3.6节。
[20] 意识到视觉图形工程的重要性；23.4.3.1节。
[21] 在原型成为负担时就抛弃它；23.4.4节。

[22] 为变化而设计，将注意力集中到灵活性、可扩展性、可移植性和重用； 23.4.2节。
[23] 将注意力集中到组件设计；23.4.3节。
[24] 让每个界面代表在一个抽象层次中的一个概念； 23.4.3.1节。
[25] 面向变化进行设计，以求得稳定性；23.4.2节。
[26] 通过将广泛频繁使用的界面做得最小、最一般和抽象来使设计稳定；23.4.3.2、23.4.3.5节。
[27] 保持尽可能小，不为“特殊需要”增加新特征；23.4.3.2节。
[28] 总考虑类的其他表示方式。如果不可能有其他方式，这个类可能就没有代表某个清晰的概念；23.4.3.4节。
[29] 反复评审、精化设计和实现；23.4节、23.4.3节。
[30] 采用那些能用于调试，用于分析问题、设计和实现的最好工具；23.3节、23.4.1节、23.4.4节。
[31] 尽早、尽可能频繁地进行试验、分析和测试；23.4.4节、23.4.5节。
[32] 不要忘记效率；23.4.7节。
[33] 保持某种适合项目规模的规范性水平；23.5.2节。
[34] 保证有人负责项目的整体设计；23.5.2节。
[35] 为可重用组件做文档、推介和提供支持；23.5.1节。
[36] 将目标与细节一起写进文档里；23.4.6节。
[37] 将为新开发者提供的教学材料作为文档的一部分；23.4.6节。
[38] 鼓励设计、库和类的重用，并给予回报；23.5.1节。

第24章　忠告

[1] 应该向数据抽象和面向对象程序设计的方向发展；24.2节。
[2] （仅仅）根据需要去使用C++的特征和技术；24.2节。
[3] 设计应与编程风格相互匹配；24.2.1节。
[4] 将类/概念作为设计中最基本的关注点，而不是功能/处理；24.2.1节。
[5] 用类表示概念；24.2.1节、24.3节。
[6] 用继承（仅仅）表示概念间的层次结构关系；24.2.2节、24.5.2节、24.3.2节。
[7] 利用应用层静态类型的方式给出有关界面的更强的保证；24.2.2节。
[8] 使用程序生成器和直接界面操作工具去完成定义良好的工作；24.2.3节。
[9] 不要去使用那些与任何通用程序设计语言之间都没有清晰界面的程序生成器或者直接界面操作工具；24.2.4节。
[10] 保持不同层次的抽象相互分离；24.3.1节。
[11] 关注组件设计；24.4节。
[12] 保证虚函数有定义良好的意义，每个覆盖函数都实现预期行为；24.3.4节、24.3.2.1节。
[13] 公用界面继承表示的是“是一个”关系；24.3.4节。
[14] 成员表示的是“有一个”关系；24.3.4节。
[15] 在表示简单包容时最好用直接成员，不用指向单独分配的对象的指针；24.3.3节、24.3.4节。
[16] 设法保证使用依赖关系为易理解的，尽可能不出现循环，而且最小；24.3.5节。

[17] **对于所有的类，定义好不变式；**24.3.7.1**节。**
[18] **显式地将前条件、后条件和其他断言表述为断言（可能使用** ***Assert*****( )）；**24.3.5**节。**
[19] **定义的界面应该只暴露出尽可能少的信息；** 24.4**节。**
[20] **尽可能减少一个界面对其他界面的依赖性；** 24.4.2**节。**
[21] **保持界面为强类型的；**24.4.2**节。**
[22] **利用应用层的类型来表述界面；** 24.4.2**节。**
[23] **将界面表述得使请求可以传递给远程的服务器；** 24.4.2**节。**
[24] **避免肥大的界面；**24.4.3**节。**
[25] **尽可能地使用*private*数据和成员函数；**24.4.2**节。**
[26] **用*protected*/*public*区分开派生类的设计者与一般用户间的不同需要；** 24.4.2**节。**
[27] **使用模板去做通用型程序设计；**24.4.1**节。**
[28] **使用模板去做算法策略的参数化；**24.4.1**节。**
[29] **如果需要在编译时做类型解析，请使用模板；** 24.4.1**节。**
[30] **如果需要在运行时做类型解析，使用类层次结构；** 24.4.1**节。**

第25章　忠告

[1] **应该对一个类的使用方式做出有意识的决策（作为设计师或者作为用户）；** 25.1**节。**
[2] **应注意到涉及不同种类的类之间的权衡问题；** 25.1**节。**
[3] **用具体类型去表示简单的独立概念；** 25.2**节。**
[4] **用具体类型去表示那些最佳效率极其关键的概念；** 25.2**节。**
[5] **不要从具体类派生；**25.2**节。**
[6] **用抽象类去表示那些对象的表示可能变化的界面；** 25.3**节。**
[7] **用抽象类去表示那些可能出现多种对象表示共存情况的界面；** 25.3**节。**
[8] **用抽象类去表示现存类型的新界面；** 25.3**节。**
[9] **当类似概念共享许多实现细节时，应该使用结点类；** 25.4**节。**
[10] **用结点类去逐步扩充一个实现；** 25.4**节。**
[11] **用运行时类型识别从对象获取界面；** 25.4.1**节。**
[12] **用类去表示具有与之关联的状态信息的动作；** 25.5**节。**
[13] **用类去表示需要存储、传递或者延迟执行的动作；** 25.5**节。**
[14] **利用界面类去为某种新的用法而调整一个类（不修改这个类）；** 25.6**节。**
[15] **利用界面类增加检查，** 25.6**节。**
[16] **利用句柄去避免直接使用指针和引用；** 25.7**节。**
[17] **利用句柄去管理共享的表示；** 25.7**节。**
[18] **在那些能预先定义控制结构的应用领域中使用应用框架；** 25.8**节。**

附录B　忠告

[1] **要学习**C++**，应该使用你可以得到的标准**C++**的最新的和完全的实现；**B.3**节。**

[2] C和C++的公共子集并不是学习C++时最好的开始子集；1.6节、B.3节。

[3] 对于产品代码，请记住并不是每个C++实现都是完全的最新的。在产品代码中使用某个新特征之前应先做试验，写一个小程序，测试你计划使用的实现与标准的相符情况和性能。例如，参见8.5[6~7]、16.5[10]和B.5[7]。

[4] 避免被贬斥的特征，例如全局的*static*；还应避免C风格的强制；6.2.7节、B.2.3节。

[5] “隐含的*int*”已禁止，因此请明确描述每个函数、变量、*const*等的类型；B.2.2节。

[6] 在将C程序转为C++程序时，首先保证函数声明（原型）和标准头文件的一致使用；B.2.2节。

[7] 在将C程序转为C++程序时，对以C++关键字为名的变量重新命名；B.2.2节。

[8] 在将C程序转为C++程序时，将*malloc*( )的结果强制到适当类型，或者将*malloc*( )的所有使用都改为*new*；B.2.2节。

[9] 在将*malloc*( )和*free*( )转为*new*和*delete*时，请考虑用*vector*、*push_back*( )和*reserve*( )而不是*realloc*( )；3.8节、16.3.5节。

[10] 在将C程序转为C++程序时，记住这里没有从*int*到枚举的隐式转换；如果需要，请用显式转换；4.8节。

[11] 在名字空间*std*里定义的功能都定义在无后缀的头文件里（例如，*std*::*cout*声明在<*iostream*>里）。早些的实现将标准库功能定义在全局空间里，声明在带 *.h*后缀的头文件里（例如，*std*::*cout*声明在<*iostream.h*>里）；9.2.2节、B.3.1节。

[12] 如果老的代码检测 *new*的结果是否为 *0*，那么必须将它修改为捕捉 *bad_alloc*或者使用*new*(*nothrow*)；B.3.4节。

[13] 如果你用的实现不支持默认模板参数，请显式提供参数；用 *typedef*可以避免重复写模板参数（类似于*string*的*typedef*使你无须写*basic_string*<*char*, *char_traits*<*char*>, *allocator*<*char*> >）；B.3.5节。

[14] 用<*string*>得到*std*::*string*（<*string.h*>里保存的是C风格的串函数）；9.2.2节、B.3.1节。

[15] 对每个标准C头文件<*X.h*>，它将名字放入全局名字空间；与之对应的头文件 <*cX*>将名字放入名字空间*std*；B.3.1节。

[16] 许多系统有一个“ *String.h*”头文件里定义了一个串类型。注意，这个串类型与标准库的*string*不同。

[17] 尽可能使用标准库功能，而不是非标准的功能；20.1节、B.3节、C.2节。

[18] 在声明C函数时用*extern* "*C*"；9.2.4节

附录C 忠告

[1] 应集中关注软件开发而不是技术细节；C.1节。

[2] 坚持标准并不能保证可移植性；C.2节。

[3] 避免无定义行为（包括专有的扩充）；C.2节。

[4] 将那些实现定义的行为局部化；C.2节。

[5] 在没有{、}、[、]、|或! 的系统里用关键字和二联符表示程序，在没有 \ 的地方用三联符；C.3.1节。

[6] **为了方便通信，用**ASCII**字符去表示程序；**C.3.3**节。**

[7] **采用符号转义字符比用数值表示字符更好些；** C.3.2**节。**

[8] **不要依赖于*char*的有符号或者无符号性质；**C.3.4**节。**

[9] **如果对整数文字量的类型感到有疑问，请使用后缀；** C.4**节。**

[10] **避免破坏值的隐式转换；**C.6**节。**

[11] **用*vector*比数组好；**C.7**节。**

[12] **避免*union*；**C.8.2**节。**

[13] **用位域表示外部确定的布局；**C.8.1**节。**

[14] **注意不同存储管理风格间的权衡；**C.9**节。**

[15] **不要污染全局名字空间；**C.10.1**节。**

[16] **在需要作用域（模块）而不是类型的地方，用 *namespace*比*class*更合适；**C.10.3**节。**

[17] **记住*static*类成员需要定义；**C.13.1**节。**

[18] **用*typename*消除对模板参数中类型成员的歧义性；** C.13.5**节。**

[19] **在需要用模板参数显式限定之处，用 *template*消除模板类成员的歧义性；**C.13.6**节。**

[20] **写模板定义时，应尽可能减少对实例化环境的依赖性；** C.13.8**节。**

[21] **如果模板实例化花的时间过长，请考虑显式实例化；** C.13..10**节。**

[22] **如果需要编译顺序的显式可预见性，请考虑显式实例化；** C.13.10**节。**

附录D　忠告

[1] **应预期每个直接与人打交道的非平凡程序或者系统都会用在多个国家；** D.1**节。**

[2] **不要假定每个人使用的都是你所用的字符集；** D.4.1**节。**

[3] **最好是用*locale*而不是写实质性代码去做对文化敏感的** I/O**；**D.1**节。**

[4] **避免将现场名字字符串嵌入到程序正文里；** D.2.1**节。**

[5] **尽可能减少全局格式信息的使用；**D.2.3**节、**D.4.4.7**节。**

[6] **最好是用与现场有关的字符串比较和排序；** D.2.4**节、**D.4.1**节。**

[7] **保存*facet*的不变性；**D.2.2**节、**D.3**节。**

[8] **应保证改变现场的情况只出现在程序里的几个地方；** D.2.3**节。**

[9] **利用现场去管理刻面的生存期；**D.3**节。**

[10] **在写对现场敏感的** I/O**函数时，记住去处理用户（通过覆盖）提供的函数所抛出的异常；** D.4.2.2**节。**

[11] **用简单的*Money*类型保存货币值；**D.4.3**节。**

[12] **要做对现场敏感的**I/O**，最好是用简单的用户定义类型保存所需的值（而不是从内部类型的值强制转换）；**D.4.3**节。**

[13] **在你对涉及到的所有因素有了很好的看法之前，不要相信计时结果；** D.4.4.1**节。**

[14] **当心*time_t*的取值范围；**D.4.4.1**节、**D.4.4.5**节。**

[15] **使用能接受多种输入格式的日期输入例程；** D.4.4.5**节。**

[16] **最好采用那些明显表明了所用现场的字符分类函数；** D.4.5**节、**D.4.5.1**节。**

附录E　忠告

[1] **弄清楚你想要什么级别的异常时安全性；** E.2**节。**
[2] **异常时安全性应该是整体容错策略的一部分；** E.2**节。**
[3] **为所有的类提供基本保证，也就是说，维持一个不变式，而且不流失资源；** E.2**节、**E.3.2**节、**E.4**节。**
[4] **在可能和可以负担之处提供强保证，使操作或者成功，或者保持所有操作对象不变；** E.2**节、**E.3**节。**
[5] **不要从析构函数里抛出异常；**E.2**节、**E.3.2**节、**E.4**节。**
[6] **不要从一个遍历合法序列的迭代器里抛出异常；**E.4.1**节、**E.4.4**节。**
[7] **异常时安全性涉及到仔细检查各个操作；**E.3**节。**
[8] **将模板设计为对异常透明的；**E.3.1**节。**
[9] **更应该用申请资源的构造函数方式，不要采用 *init*( ) 函数；**E.3.5**节。**
[10] **为类定义一个不变式，使什么是合法状态变得非常清晰；** E.2**节、**E.6**节。**
[11] **确保总将对象放在合法状态中，也不要怕抛出异常；**E.3.2**节、**E.6**节。**
[12] **保持不变式简单；**E.3.5**节。**
[13] **在抛出异常之前，让所有操作对象都处于合法状态；** E.2**节、**E.6**节。**
[14] **避免资源流失；**E.2**节、**E.3.1**节、**E.6**节。**
[15] **直接表示资源；**E.3.2**节、**E.6**节。**
[16] **记住*swap*( ) 有时可以成为复制元素的替代方式；** E.3.3**节。**
[17] **在可能时依靠操作的顺序，而不是显式地使用** try**块；**E.3.4**节。**
[18] **在替代物已经安全生成之前不销毁“老”信息；** E.3.3**节、**E.6**节。**
[19] **依靠“资源申请即初始化”技术；**E.3**节、**E.3.2**节、**E.6**节。**
[20] **确保关联容器的比较操作能够复制；** E.3.3**节。**
[21] **标明关键性数据结构，并为它们定义能够提供强保证的操作；** E.6**节。**