附录B 编 程 准 则

这个附录[1]收集了C++编程的一些建议，它们是我在教学和实践过程中收集而成的，当然还有：

从朋友那儿得到的忠告，包括 Dan Saks（与Tom Plum合写了“ C++ Programming Guidelines ”，Plum Hall 1991）、Scott Meyers(“ Effective C++ ”一书的作者， Addison-Wesley,1992)和Rob Murray（“ C++ strategies & Tactics ”的作者， Addision-Wesley,1993），有许多条目都是从这本书上摘录下来的。

1. 不要用C++主动重写我们已有的C代码，除非我们需要对它的功能做较大的调整，（也就是说，不破不立）。用C++重新编译是很有价值的，因为这可以发现隐藏的错误。把一段运行得很好的C代码用C++重写可能是在浪费时间，除非 C++的版本以类的形式提供许多重用的机会。

2. 要区别类的创建者和类的使用者（客户程序员）。类的使用者才是“ 顾客 ”，他们并不需要或许也不想知道类的内部是怎样运作的。类的创建者必须是设计类和编写类的专家，以使得被创建的类可以被最没有经验的程序员使用，而且在应用程序中工作良好。库只是在透明的情况下才会容易使用。

3. 当我们创建一个类时，要尽可能用有意义的名字来命名类。我们的目标应该是使用户接口要领简单。可以用函数重载和缺省参数来创建一个清楚、易用的接口。

4. 数据隐藏允许我们（类的创建者）将来在不破坏用户代码（代码使用了该类）的情况下随心所欲地修改代码。为实现这一点，应把对象的成员尽可能定义为 private，而只让接口部分为public，而且总是使用函数而不是数据。只有在迫不得已时才让数据为 public。如果类的使用者不需要调用某个函数，就让这个函数成为 private。如果类的一部分要让派生类可见，就定义成protected，并提供一个函数接口而不是直接暴露数据，这样，实现部分的改变将对派生类产生最小的影响。

5. 不要陷入分析瘫痪之中。有些东西只有在编程时才能学到并使各种系统正常。 C++有内建的防火墙，让它们为我们服务。在类或一组类中的错误不会破坏整个系统的完整性。

6. 我们的分析和设计至少要在系统中创建类、它们的公共接口、它们与其他类的关系、特殊的基类。如果我们的方法产生的东西比这些更多，就应当问问自己，是不是所有的成分在程序的整个生命期中都是有价值的，如果不是，将会增加我们对它们的维护开销。开发小组的人都认为不应该维护对他们的产品没有用的东西。许多设计方法并不大奏效，这是事实。

7. 记住软件工程的基本原则：所有的问题都可以通过引进一个额外的间接层来简化(Andrew Koenig向我解释了这一点)。这是抽象方法的基础，而抽象是面向对象编程的首要特征。

8. 使类尽可能地原子化。也就是每个类有一个单一、清楚的目的。如果我们的类或我们设计的系统过于复杂，就应当将所有复杂的类分解成多个简单的类。

9. 从设计的角度，寻找并区分那些变化和不变的成分。也就是在系统中寻找那些修改时不

[1] 增加这个附录是Andrew Binstock建议的，他是《Unix Review》的主编，兼撰稿人。

需要重新设计的成分，把它们封装到一个类中。

10. 注意不同点。两个语义上不同的对象可能有同样的操作或反应，自然就会试着把一个作为另一个的子类以便利用继承性的好处。这就叫差异，但并没有充分的理由来强制这种并不存在的父子关系。一个好的解决办法是产生一个共同的父类：它包含两个子类——这可能要多占一点空间，但我们可以从继承中获益，并且可能对这种自然语言的解有一个重要发现。

11. 注意在继承过程中的限制。最清晰的设计是向被继承者加入新的功能，而如果在继承过程删除了原有功能，而不是加入新功能，那这个设计就值得怀疑了。但这也不是绝对的，如果我们正在与一个老的类库打交道，对已有的类在子类中进行限制可能更有效，而不必重建一套类层次来使我们的新类适应新的应用。

12. 不要用子类去扩展基类的功能。如果一个类接口部分很关键的话，应当把它放在基类中，而不是在继承时加入。如果我们正在用继承来添加成员函数，我们可能应该重新考虑我们的设计。

13. 一个类一开始时接口部分应尽可能小而精。在类使用过程中，我们会发现需要扩展类的接口。然而一个类一旦投入使用，我们要想减少接口部分，就会影响那些使用了该类的代码，但如果我们需要增加函数则不会有影响，一切正常，只需重新编译一下即可。但即使用新的成员函数取代了原来的功能，也不要去改正原有接口（如果我们愿意的话，可以在低层将两个函数合并）。如果我们需要对一个已有的函数增加参数，我们可以让原来的参数保持不变，把所有新参数作为缺省参数，这样不会妨碍对该函数已有的调用。

14. 大声朗读我们的类，确保它们是合理的。读基类时用“ is-a ”，读成员对象时用“ has-a ”。

15. 在决定是用继承还是用组合时，问问自己是不是需要向上映射到基类。如果不需要，就用组合（成员对象）而不用继承。这样可以减少多重继承的可能。如果我们选择继承，用户会认为他们被假设向上映射。

16. 有时我们为了访问基类中的 protected成员而采用继承。这可能导致一个可察觉的对多重继承的需求。如果我们不需要向上映射，首先导出一个新类来完成保护成员的访问，然后把这个新类作为一个成员对象，放在需要用到它的所有对象中去。

17. 一个典型的基类仅仅是它的派生类的一个接口。当我们创建一个基类时，缺省情况下让成员函数都成为纯虚函数。析构函数也可以是纯虚函数（强制派生类对它重新定义），但记住要给析构函数一个函数体，因为继承关系中所有的析构函数总是被调用。

18. 当我们在类中放一个虚函数时，让这个类的所有函数都成为虚函数，并在类中定义一个虚析构函数。当我们要求高效时再把 virtual关键词去掉，这种方法防止了接口的行为出格。

19. 用数据成员表示值的变化，用虚函数表示行为的变化。如果我们发现一个类中有 几个状态变量和几个成员函数，而成员函数在这些变量的作用下改变行为，我们可能要重新设计它，用子类和虚函数来区分这种不同的作用。

20. 如果我们必须做一些不可移植的事，对这种服务做一个抽象并将它定位在一个类的内部，这个额外的间接层可防止这种不可移植性影响我们的整个程序。

21. 尽量不用多重继承。这可帮我们摆脱困境，尤其是修复我们无法控制的类的接口。除非我们是一个经验相当丰富的程序员，否则不要在系统中设计多重继承。

22. 不要用私有继承。虽然 C++中可以有私有继承，而且似乎在某些场合下很有用，但它和运行时类型识别一起使用时，常常引起语义的模棱两可。我们可以用一个私有成员对象来代替私有继承。

23. 运算符重载仅仅是“ 语法糖 ”：另一种函数调用方法。如果重载一个运算符不会使类的

接口更清楚、更易于使用，就不要重载它。一个类只创建一个自动类型转换运算符，一般情况下，重载运算符应遵循第11章介绍的原则和格式。

24. 首先保证程序能运行，然后再考虑优化。特别是，不要急于写内联函数、使一些函数为非虚函数或者紧缩代码以提高效率。这些在我们开始构建系统时都不用考虑。我们开始的目标应该是证明设计的正确性，除非设计要求一定的效率。

25. 不要让编译器来为我们产生构造函数、析构函数或“ = ”运算符。这些是训练我们的机会。类的设计者应该明确地说出类应该做什么，并完全控制这个类。如果我们不想要拷贝构造函数或“ = ”运算符，就把它们声明为私有的。记住，只要我们产生了任何构造函数，就防止了缺省构造函数被生成。

26. 如果我们的类中包含指针，我们必须产生拷贝构造函数、“ = ”运算符和析构函数，以使类运行正常。

27. 为了减少大项目开发过程中的重复编译，应使用第3章介绍的类句柄/Cheshire cat技术，只有需要提高运行效率时才把它去掉。

28. 避免用预处理器。可以用常量来代替值，用内联函数代替宏。

29. 保持范围尽可能的小，这样我们的对象的可见性和生命期也就尽可能的小。这就减少了错用对象和隐藏难以发现的错误的可能性。比方说，假设我们有一个包容器和一段扫描这个包容器的代码，如果我们拷贝这些代码来用一个新的包容器，我们可能无意间用原有的包容器的大小作为新包容器的边界。如果原来的这个包容器超过了这个范围，就会引起一个编译错误。

30. 避免使用全局变量。尽可能把数据放在类中。全局函数存在的可能性要比全局变量大，虽然我们后来发现一个全局函数作为一个类的静态成员更合适。

31. 如果我们需要声明一个来自库中的类或函数，应该用包含一个头文件的方法。比如，如果我们想创建一个函数来写到 ostream中，不要用一个不完全类型指定的方法自己来声明ostream，如

```
class ostream；
```

这样做会使我们的代码变得很脆弱（比如说 ostream实际上可能是一个typedef）。我们可以用头文件的形式，例如：

```
#include <iostream.h>
```

当创建我们自己的类时，如果一个库很大，应提供给用户一个头文件的简写形式，文件中包含有不完全的类型说明（这就是类型名声明），这是对于只需要用到指针的情况（它可以提高编译速度）。

32. 当选择重载运算符的返回值类型时候，要一起考虑串连表达式：当定义运算符“ = ”时应记住x=x。对左值返回一个拷贝或一个引用（返回 *this），所以它可以用在串连表达式（A=B=C）中。

33. 当写一个函数时，我们的第一选择是用const引用来传递参数。只要我们不需要修改正在被传递进入的对象，这种方式是最好的。因为它有着传值方式的简单，但不需要费时的构造和析构来产生局部对象，而这在传值方式时是不可避免的。通常我们在设计和构建我们的系统时不用注意效率问题，但养成这种习惯仍是件好事。

34. 当心临时变量。当调整完成时，要注意临时创建的对象，尤其是用运算符重载时。如果我们的构造函数和析构函数很复杂，创建和销毁临时对象就很费时。当从一个函数返回一个值时，总是应在return语句

```
return foo(i,j);
```

中调用构造函数来“就地”产生一个对象。这优于

```
foo  x(i,j);
return  x;
```

前一个返回语句避免了拷贝构造函数和析构函数的调用。

35. 当产生构造函数时，要考虑到异常，在最好的情况下，构造函数不需要引起一个异常，另一种较好的情况：类将只从健壮的类被组合和继承，所以当一个异常产生时它会自动清除它自己。如果我们必须使用一个裸指针，我们应该负责捕获自己的异常，然后在我们的构造函数中释放所有异常出现之前指针指向的资源。如果一个构造函数无法避免失败，最好的方法是抛出一个异常。

36. 在我们的构造函数中只做一些最必要的事情，这不仅使构造函数的调用有较低的时间花费（这中间有许多可能不受我们控制），而且我们的构造函数更少地抛出异常和引起的问题。

37. 析构函数的作用是释放在对象的整个生命期内分配的所有资源，而不仅仅是在创建期间。

38. 使用异常层次，从标准C++异常层次中继承并嵌套，作为能抛出这个异常的类中的一个公共类。捕获异常的人然后可以确定异常的类型。如果我们加上一个新的派生异常，客户代码还是通过基类来捕获这个异常。

39. 用值来抛出异常，用引用来捕获异常。让异常处理机制处理内存管理。如果我们抛出一个指向在堆上产生的异常的指针，则异常处理器必须知道怎样破坏这个异常，这不是一种好的搭配。如果我们用值来捕获异常，我们需要额外的构造和析构，更糟的是，我们的异常对象的派生部分可能在以值向上映射时被切片。

40. 除非确有必要，否则不要写自己的类模板。先查看一个标准模板库，然后查问创建特殊工具的开发商。当我们熟悉了这些产品后，我们就可大大提高我们的生产效率。

41. 当创建模板时，留心那些不带类型的代码并把它们放在非模板的基类中，以防不必要的代码膨胀。用继承或组合，我们可以产生自己的模板，模板中包含的大量代码都是类型有关的，因此也是必要的。

42. 不要用STDIO.H中的函数，例如printf()。学会用输入输出流来代替，它们是安全类型和可扩展类型，而且功能也更强。我们在这上面花费的时间肯定不会白费（参看第6章）。一般情况下都要尽可能用C++中的库而不要用C库。

43. 不要用C的内部数据类型，虽然C++为了向后兼容仍然支持它们，但它们不像C++的类那样强壮，所以这会增加我们查找错误的时间。

44. 无论何时，如果我们用一个内部数据类型作为一个全局或自动变量，在我们可以初始化它们之前不要定义它们。每一行定义一个变量，并同时对它初始化。当定义指针时，把‘*’紧靠在类型的名字一边。如果我们每个变量占一行，我们就可以很安全地定义它们。这种风格也使读者更易于理解。

45. 保证初始化出现在我们的代码的所有方面。在构造函数初始化表达式表中完成所有成员的初始化，甚至包括内部数据类型（用伪构造函数调用）。用任何簿记技术来保证没有未初始化的对象在我们的系统中运行。在初始化子对象时用构造函数初始化表达式表常常更有效，否则调用缺省构造函数，并且我们最终调用其他成员函数，也可能是运算符“=”，为了得到我们想要的初始化。

46. 不要用“foo a=b;”的形式来定义一个对象。这是常常引起混乱的原因。因为它调用构

造函数来代替运算符“ = ”。为了清楚起见，可以用“ foo a(b); ”来代替。这个语句结果是一样的，但不会引起混乱。

47. 使用C++中新的类型映射。一个映射践踏了正常的类型系统，它往往是潜在的错误点。通过把C中一个映射负责一切的情况改成多种表达清楚的映射，任何人来调试和维护这些代码时，都可以很容易地发现这些最容易发生逻辑错误的地方。

48. 为了使一个程序更强壮，每个组件都必须是很强壮的。在我们创建的类中运用 C++中提供的所有工具：隐藏实现、异常、常量更正、类型检查等等。用这些方法我们可以在构造系统时安全地转移到下一个抽象层次。

49. 建立常量更正。这允许编译器指出一些非常细微且难以发现的错误。这项工作需要经过一定的训练，而且必须在类中协调使用，但这是值得的。

50. 充分利用编译器的错误检查功能，用完全警告方式编译我们的全部代码，修改我们的代码，直到消除所有的警告为止。在我们的代码中宁可犯编译错误也不要犯运行错误（比如不要用变参数列表，这会使所有类型检查无效）。用assert()来调试，但要用异常来处理运行时错误。

51. 宁可犯编译错误也不要犯运行错误。处理错误的代码离出错点越近越好。尽量就地处理错误而不要抛出异常。用最近的异常处理器处理所有的异常，这里它有足够的信息处理它们。在当前层次上处理我们能解决的异常，如果解决不了，重新抛出这个异常。

52. 如果我们用异常说明，用set_unexpected() 函数安装我们自己的unexpected()函数。我们的unexpected()应该记录这个错误并重新抛出当前的异常。这样的话，如果一个已存在的函数被重复定义并且开始引起异常时，它不会不引起整个程序中止。

53. 建立一个用户定义的terminate()函数（指出一个程序员的错误）来记录引起异常的错误，然后释放系统资源，并退出程序。

54. 如果一个析构函数调用了任何函数，这些函数都可能抛出异常。一个析构函数不能抛出异常（这会导致terminate()调用，它指出一个程序设计错误）。所以任何调用了其他函数的析构函数都应该捕获和管理它自己的异常。

55. 不要自己创建私有数据成员名字“ 分解 ”，除非我们有了许多已有的全局值，否则 让类和命名空间来为我们做这些事。

56. 如果我们打算在for循环结束之后使用一个循环变量，要在 for控制表达式之前定义这个变量，这样，当 for控制表达式中定义的变量的生命期被限制在 for循环之内时，我们的程序依然正确。

57. 注意重载，一个函数不应该用某一参数的值来决定执行哪段代码，如果遇到这种情况，应该产生两个或多个重载函数来代替。

58. 把我们的指针隐藏在包容器类中。只有当我们要对它们执行一个立即可以完成的操作时才把它们带出来。指针已经成为出错的一大来源，当用 new运算符时，应试着把结果指针放到一个包容器中。让包容器拥有它的指针，这样它就会负责清除它们。如果我们必须有一个游离状态的指针，记住初始化它，最好是指向一个对象的地址，必要时让它等于 0。当我们删除它时把它置0，以防意外的多次删除。

59. 不要重载全局new和delete，我们可以在一个类跟随类的基础上去重载它们。重载全局new和delete会影响整个客户程序员的项目，有些事只能由项目的创建者来控制。当为类重载new和delete时，不要假定我们知道对象的大小，有些人可能是从我们的类中继承的。用提供的参数，如果我们做任何特殊的事，要考虑到它可能对继承者产生的影响。

60. 不要自我重复。如果一段代码在派生类的许多函数中重复出现，就把这段代码放在基类的一个单一的函数中然后在派生类中调用它。这样我们不仅节省了代码空间，也使将来的修改容易传播。这甚至适用于纯虚函数（见第 14章）。我们可以用内联函数来提高效率。有时这种相同代码的发现会为我们的接口添加有用的功能。

61. 防止对象切片。实际上用值向上映射到一个对象毫无意义。为了防止这一点，在我们的基类中放入一些纯虚函数。

62. 有时简单的集中会很管用。一个航空公司的“旅客舒适系统”由一系列相互无关的因素组成：座位、空调、电视等等，而我们需要在一架飞机上创建许多这样的东西。我们要创建私有成员并建立一个全部的接口吗？不，在这种情况下组件本身也是公开接口的一部分，所以我们应该创建公共成员对象。这些对象有它们自己的私有实现，所以也是很安全的。