C++编程风格指南

英文名：C++ Programming Style Guidelines

中文名：C++编程风格指南

原作者：*Geotechnical Software Services*

版本：4.9

时间：2011年1月

英文原文：http://geosoft.no/development/cppstyle.html

译者：Guoning-Chen

译者邮箱：cgn1874@163.com

译者前言

内容简介

众所周知，对于一名合格的程序员而言，保持良好的代码风格非常重要。好的代码风格能够让别人（也不只是别人，还可能是写完代码N天之后的自己）快速地掌握一段代码的逻辑结构，从而高效地进行维护，这对于团队协作开发一个大型项目是非常有利的。另外，良好的代码风格甚至还能够从源头上避免自己无意间造成的一些bug。

关于代码风格的书或在线文档有很多，已经出版的书籍有《重构-改善既有代码的设计》、《代码整洁之道》等等；很多大公司也有完善的编程规范，例如：Google 开源项目风格指南 (中文版)。虽然这些资料的内容都很详细、很权威，但缺点就是——**太长**（Google编程规范中文版就长达5万字），不太适合**C++初学者**。

我接触到这份指南是在刚开始学习C++的时候，由MOOC公开课——北京邮电大学《C++程序设计（面向对象进阶）》的老师推荐的。**之所以说这份指南适合初学者，主要有以下原因：**

1. **字数少**：翻译之后不到8000字。
2. **格式友好，方便阅读**：这里面其实是一条一条的建议，只有90条左右；每条建议还给出了示例和**之所以这样做的原因**（这一点对于记忆很有用！）；另外格式很清晰，方便多次回看。
3. **内容很基础，初学者也能用得到**：学习编程规范的最好方法就是**边学边用**，但是像Google编程规范中的很多内容，初学者的日常编程其实是用不到的，而这份规范中的大部分内容都很基础，也很常用；又因为它很短，用到的时候可以很快翻到相应的地方，不至于太影响编程体验。
4. **有利于培养代码规范的意识**：正如作者前两条建议所说——**“只要能够提升可读性，允许采用不同于该指南的做法。”** **“如果你个人很抵触该指南的某项建议，可以不采纳。”**作者一直在强调，重要的是这种**下意识为阅读代码的人着想的意识**，而不是某种固定、死板的做法，这一点让我受益很多。

总之，**这篇指南适合以下读者：**

1. **从一开始就想养成良好编程习惯的C++初学者**

   对你来说，这份短小精悍的指南将是一个非常好的选择。

2. **想要从现在开始纠正之前的编程习惯的C++使用者**

   相信会有很多人像我一样，因为畏惧Google编程规范之类文档的过于庞杂而不知道从何处开始，这份指南同样适合你们。万事开头难，你完全可以把阅读这份指南当做一个相对轻松的开头，等到培养一定的习惯之后再考虑学习那些更加复杂、也更加完善的编程指南。

注意事项

1. 由于水平有限，文中部分名词和句子翻译的似乎不太通顺，但未找到更好的译法，已在文中用下划线标出；
2. 部分单词未找到合适的译法，在文中**保留了英文**，同样用下划线标出；
3. 部分标记了下划线的英文单词在原文中出现多次，我推测可能是术语，在**译者附录**中给出了我自己的理解；
4. 如果你发现翻译内容有误或对下划线部分有新的见解，欢迎在**issue**中指出，同样欢迎任何其他的意见。

1 简介

该指南包含了C++开发社区中对于C++编程的常用建议。

这些建议的来源包括：一些资源中的现有标准、个人经验、本地需求以及参考资料[1]~[4]。

我们之所以提出新的指南而不是简单遵循上述内容是有一些原因的。主要是因为上述内容太宽泛，需要建立更具体的规范（尤其是命名规范）。而且该指南采用了独特的排版，在检查项目代码的过程中使用起来要比其它指南方便得多。另外令人有些困惑的是，编程建议经常将代码风格问题和编程技术问题弄混。本文档主要关注代码风格而不包含任何与C++技术相关的建议。想了解更多有关c++编程风格的内容，请参考C++编程实践指南。

尽管集成开发环境（IDE）通过权限可视化、代码着色、自动排版等功能可以提高代码可读性，但编程人员从来都不应该依赖这些特性。源代码应该总是被放到比编写代码的IDE更重要的位置，尽可能提升其独立于IDE的可读性。

1.1 建议的排版

所有建议按照主题进行了分组，同时标注了序号，便于回顾。

每条建议都采用如下排版：

序号. 对该建议的简要描述
举例
该建议的出发点、背景和其它信息

第3部分的内容很重要。编程标准和指南容易引发**“宗教战争”**，因此说明建议的使用背景很重要。

1.2 建议的重要性

该指南中的**must**、**should**和**can**都有特殊的含义。**must（必须）**代表该建议必须遵循；**should（应该）**代表强烈推荐；**can（可以）**代表一般建议。

2 整体建议

1. 只要能够提升可读性，允许采用不同于该指南的做法。
无
建议的意义在于提升代码的可读性，从而有助于阅读者理解代码，提高代码的维护性和代码质量。一份指南很难适用于所有情况，编程者应该灵活应对。

2. 如果你个人很抵触该指南的某项建议，可以不采纳。

无

撰写指南并不是为了强制个人使用某种特定的代码风格。一些有经验的编程者通常会采用类似于该指南的风格，但是有自己的风格并推荐给其他人，通常会让人们开始**思考**编程风格这件事并反思他们自己的习惯。
另一方面，通过使用编程风格指南，没有经验的新手更容易快速熟悉编程术语。

3 命名规范

3.1 通用命名规范

3. 【must】类型的名称中所有单词的首字母必须大写。

```
Line, SavingsAccount
```

C++开发社区中的通用做法。

4. 【must】变量名中的第一个单词必须小写，其余大写。

```
line, savingsAccount
```

C++开发社区中的通用做法。使得变量区别于各种类的名称，有效解决如`Line line`这种声明中潜在的命名冲突。

5. 【must】常量（包括枚举量）的名称中所有字母都必须大写，并用下划线分开。

```
MAX_ITERATIONS, COLOR_RED, PI
```

C++开发社区中的通用做法。通常应该尽可能避免使用常量。在大部分情况下，用函数代替是更好的选择：

```
int getMaxIterations() // NOT: MAX_ITERATIONS = 25
{
  return 25;
}
```

这样既容易阅读，又能保证和类的成员变量统一接口。

6. 【must】方法或函数的名称必须包含动词且首字母小写、其余大写。

```
getName(), computeTotalWidth()
```

C++开发社区中的通用做法。虽然和变量的命名方式相同，但函数本身的特征已经足以将它和变量区别开来。

7. 【should】命名空间的名称中的字母应该全部小写。

```
model::analyzer, io::iomanager, common::math::geometry
```

C++开发社区中的通用做法。

8. 【should】模板类的名称应该是一个大写英文字母。

```
template<class T> ...
template<class C, class D> ...
```

C++开发社区中的通用做法。使得模板类的名称比其它名称都要显眼。

9. 【must】当用于命名的时候，缩写词汇只有首字母可以大写 [4]

```
exportHtmlSource(); // NOT: exportHTMLSource();
openDvdPlayer();    // NOT: openDVDPlayer();
```

如果将缩写词全部大写，将会破坏上述命名规范。相应类型的变量就只能命名为 `dVD`， `hTML` 等等，可读性显然不好。另一个问题在例子中已经说明：当名称中包含多个单词的时候，（如果将缩写词全部大写）可读性会严重降低，不再容易分辨出缩写词之后的单词。

10. 【should】应该一直通过 :: 操作符来使用全局变量。

```
::mainWindow.open(), ::applicationContext.getName()
```

通常应该减少使用全局变量。考虑用单例对象来代替。

11. 【should】类的私有成员应该加下划线。

```
class SomeClass {
  private:
    int length_;
}
```

除了名称和类型之外，变量的**作用域**是最重要的特征。利用下划线来表明其作用域容易将类的成员变量和本地变量区别开来。这之所以很关键是因为类的成员变量比成员函数更重要，要特殊对待。使用下划线带来的另一个好处是可以很好地解决如何给setter和构造函数的参数命名的问题：

```
void setDepth (int depth)
{
  depth_ = depth;
}
```

将下划线作为前缀还是后缀是个问题。这两种做法都很常见，但更推荐后者，因为似乎最完整地保留了名称的可读性。

值得注意的是——区分变量的作用域很久以来都是有争议的话题。不过现在这种做法似乎正逐渐被人们所接受，在专业开发社区中正逐渐成为一种惯例。

12. 【should】通用变量的名称应该和它的类型相同。

```
void setTopic(Topic* topic) // NOT: void setTopic(Topic* value)
                            // NOT: void setTopic(Topic* aTopic)
                            // NOT: void setTopic(Topic* t)

void connect(Database* database) // NOT: void connect(Database* db)
                                 // NOT: void connect (Database* oracleDB)
```

通过减少术语和名称的数量来降低代码的复杂度。同时根据变量名就很容易推断出它所属的类。
如果因为一些原因，该惯例**并不适合**你使用的场景，这很可能说明你选择了一个不合适的类名。
非通用变量通常扮演某种**角色**，通常可以结合他们各自的角色和所属的类给他们命名：

```
Point  startingPoint, centerPoint;
Name   loginName;
```

13. 【should】所有名称都应该用英文。

```
fileName;   // NOT: filNavn
```

英语更加适合国际化开发。

14. 【should】作用域比较大的变量应该用长名字，作用域小的用段名字。

无

用于临时存储或索引的变量应该尽量短。程序员看到这样的变量应该可以假定其作用域不超过几行代码。用于整数的通用变量可以是`i`，`j`，`k`，`m`，`n`，对于字符常用`c`和`d`。

15. 【should】类方法中应该避免出现对象的名称。

```
line.getLength();   // NOT: line.getLineLength();
```

如样例所示，后者虽然在类的定义中看起来很自然，但使用时显得多余。

3.2 特殊的命名惯例

17. 【must】直接访问类的属性必须使用术语get或set。

```
employee.getName();
employee.setName(name);

matrix.getElement(2, 4);
matrix.setElement(2, 4, value);
```

C++开发社区的常用做法。在Java中这种命名惯例已经多少成为了一种标准。

18. 【can】当计算某样东西时可以使用术语compute。

```
valueSet->computeAverage();
matrix->computeInverse()
```

明确提醒读者这很可能是一个消耗时间的运算，如果重复运行，可能需要考虑缓存计算结果。一致地使用该术语可以提高程序可读性。

19. 【can】当查找某样东西时可以在类方法中使用术语find。

```
vertex.findNearestVertex();

matrix.findMinElement();
```

明确提醒读者这是一个运算量较少的查找操作。一致使用该术语可以提高程序可读性。

20. 【can】当新建对象或concept时可以使用术语initialize。

```
printer.initializeFontSet();
```

美式的 `initialize` 好于英式的 `initialise`。应该避免用 `init`。

21. 【should】代表GUI控件的变量应该用组件类型作为后缀。

```
mainWindow, propertiesDialog, widthScale, loginText,
leftScrollbar, mainForm, fileMenu, minLabel, exitButton, yesToggle etc.
```

让用户立刻明白变量的类型和对象资源，从而提高代码可读性。

22. 【should】代表一组对象的变量名应该用复数。

```
vector<Point>  points;
int            values[];
```

这样的名字让用户立刻明白变量的类型和可以用于该变量的运算，从而提高可读性。

23. 【should】表示对象数量的变量应该使用前缀n。

```
nPoints, nLines
```

这种记号是数学中表示对象数量的已有惯例。

24. 【should】代表对象编号的变量应该使用后缀No。

```
tableNo, employeeNo
```

这种记号是数学中表示对象编号的惯例。
另一种优雅的命名是给变量加上前缀i，例如：`iTable`，`iEmployee`。这样这些变量就成了named iterator。

25.【should】循环变量应该用i，j，k等命名。

```
for (int i = 0; i < nTables); i++) {
  :
}

for (vector<MyClass>::iterator i = list.begin(); i != list.end(); i++) {
  Element element = *i;
  ...
}
```

这种表示方法是数学中的惯例。
变量`j`，`k`等应该只用于循环嵌套。

26.【should】布尔变量或返回布尔变量的方法应该加上前缀is。

```
isSet, isVisible, isFinished, isFound, isOpen
```

C++开发社区中很常见，而Java中近乎强制使用这种命名。
is前缀的使用避免了常见的像`status`或`flag`这种不好的命名。`isStatus`或`isFlag`明显不合适，因此编程者不得不想出一个更有意义的名字。
在类似如下的情形中，用`has`、`can`和`should`代替`is`可能更合适：

```
bool hasLicense();
bool canEvaluate();
bool shouldSort();
```

27.【must】成对的操作必须用成对的名称。

```
get/set, add/remove, create/destroy, start/stop, insert/delete,
increment/decrement, old/new, begin/end, first/last, up/down, min/max,
next/previous, old/new, open/close, show/hide, suspend/resume, etc.
```

通过对称降低代码复杂度。

28.【should】命名时应该尽量避免用缩写。

```
computeAverage();    // NOT: compAvg();
```

这里主要考虑两种词汇。第一种是字典中的常用词汇。这些一定不能缩写，**错误**示范：

```
cmd     instead of   command
cp      instead of   copy
pt      instead of   point
comp    instead of   compute
init    instead of   initialize
```

第二种是特定领域的短语，它们的缩写广为人知。这些短语应该保持缩写，**错误**做法是：

```
HypertextMarkupLanguage   instead of   html
CentralProcessingUnit     instead of   cpu
PriceEarningRatio         instead of   pe
```

29. 【should】避免为指针特殊命名。

```
Line* line; // NOT: Line* pLine;
            // NOT: LIne* linePtr;
```

C/C++环境中很多变量都是指针，几乎不可能遵循这样的命名惯例。而且C++中的对象通常都是oblique类型，程序员应该忽略具体的实现。只有当一个对象的实际类型有特殊意义时，才应该强调他的类型。

30. 【must】布尔变量必须避免使用否定含义的名称。

```
bool isError; // NOT: isNoError
bool isFound; // NOT: isNotFound
```

当进行逻辑非运算`!`时，这样的命名会产生双重否定，例如`!isNotFound`的真实含义就不是那么一目了然。

31. 【can】枚举常量可以用一个通用类型名称作为前缀。

```
enum Color {
  COLOR_RED,
  COLOR_GREEN,
  COLOR_BLUE
};
```

这一做法提供了以下信息：在哪里能找到其定义、哪些常量属于同一个集合、这些常量代表了什么含义。
另一种代替的做法是始终通过公共类型使用这些枚举常量：`Color::RED`，`Airline::AIR_FRANCE`等。
注意公共类型的名称应该是像`enum Color {...}`的单数形式，虽然`enum Colors {...}`的复数形式在在声明时看起来不错，但使用时会显得有点傻。

32. 【should】异常类应该以exception作为后缀。

```
class AccessException
{
  :
}
```

异常类并非程序设计的主要内容，这样命名能够把它们和其他类区分开来。

33. 【should】有返回值的函数应该以它返回的内容命名，没有返回值的函数应该以其执行的操作命名。

无

提升可读性。清楚地说明该函数应该执行的内容和（尤其是）不会执行的内容。这一做法容易避免side effect。

4 文件

4.1 源文件

34.【should】C++头文件的拓展名应该是.h或.hpp，源文件的拓展名应该是.c++（推荐）、.C、.cc或.cpp。

```
MyClass.c++, MyClass.h
```

这些都是为C++标准所接受的拓展名。

35.【should】应该在头文件中对类进行声明，而在与头文件同名的源文件中定义。

```
MyClass.h, MyClass.c++
```

容易找到与指定类相关的文件。但模板类是明显的例外，必须同时在头文件中定义和声明。

36.【should】 所有定义都应该位于源文件中。

```
class MyClass
{
public:
  int getValue () {return value_;}  // NO!
  ...

private:
  int value_;
}
```

头文件负责声明一个类，而源文件负责实现它。当查看某个实现时，应该始终确保程序员能在源文件中找到它。

37.【must】每行的内容必须保持在80列之内。

无

80列是编辑器、终端模拟器、打印机和调试器的通用规格，多人共享的文件应该符合这种限制。在程序员互相传递文件时，这种做法能够避免产生意外的换行，从而提升代码可读性。

38.【must】避免使用TAB或分页符等特殊字符。

无

当跨编程人员和跨平台使用这些字符时，注定会导致编辑器、打印机、终端模拟器和调试器出错。

39. 【must】 如果一行代码太长而被迫换行，换行处必须清楚地表示出不完整性。[1]

```
totalSum = a + b + c +
           d + e;

function (param1, param2,
          param3);

setText ("Long line split"
         "into two parts.");

for (int tableNo = 0; tableNo < nTables;
     tableNo += tableStep) {
  ...
}
```

当某个语句的长度超过第38条规范所要求的80列时，往往会出现分行的语句。虽然很难提出一套如何分行的固定规范，但上述例子提供了大体的思路。
通常在逗号或运算符之后换行，同时新行应该和位于上一行的表达式开头对齐。

4.2 包含文件和include语句

40. 【must】头文件必须有“包含保护”

```
#ifndef COM_COMPANY_MODULE_CLASSNAME_H
#define COM_COMPANY_MODULE_CLASSNAME_H
  :
#endif // COM_COMPANY_MODULE_CLASSNAME_H
```

这种写法是为了避免编译错误。宏的命名与头文件在源文件树中的位置一致，避免命名冲突。

41. 【should】应该对#include语句进行排序和分组。按照头文件在系统中的层次排序，先包含底层的头文件；组和组之间用空行分隔。

```
#include <fstream>
#include <iomanip>

#include <qt/qbutton.h>
#include <qt/qtextfield.h>

#include "com/company/ui/PropertiesDialog.h"
#include "com/company/ui/MainWindow.h"
```

除了展示包含的独立头文件之外，还能清晰地表明该文件用到了哪些模块。
被包含文件的路径不要用绝对路径。应该用编译器指令指出被包含文件的根目录。

42. 【must】必须将 `#include` 语句放在文件的顶部。

无

通用的做法。避免在编译时由于 `#include` 语句在源文件中隐藏过深而造成side effect。

5 声明

5.1 类型

43. 【can】作用范围局限于某个文件内的变量类型可以在该文件内部声明。

无

强制隐藏信息。

44. 【must】类中的内容必须以 `public`、`protected`、`private` 的顺序排列。所有部分必须明确划分。不能遗漏任何有效的内容。

无

这样排序符合"most public first"的原则，如果某个程序员只是单纯为了使用这个类，当他看到 `protected` 或 `private` 部分时就可以停止阅读。

45. 【must】类型转换必须是显式的，永远不要依赖隐式类型转换。

```
floatValue = static_cast<float>(intValue); // NOT: floatValue = intValue;
```

这样做清楚地表明：代码的作者知道这里涉及了不同的变量类型，他是有意为之。

5.2 变量

46. 【should】应该在声明变量的位置同时对其初始化。

无

这样做保证了变量始终是有效的。有时侯可能会像下面这样初始化声明的变量：

```
int x, y, z;
getCenter(&x, &y, &z);
```

这种情况下，与其将其初始化为“假的”值，不如不初始化。

47. 【must】变量的名称永远不允许有双重含义。

无

保证所有concept被唯一地表示。减少由side effect产生的错误。

48. 【should】应该尽可能避免使用全局变量。

无

在C++中完全没有理由使用全局变量。global function和静态变量也一样。

49. 【should】类成员变量永远不应该被声明为public。

无

公有变量破坏了C++隐藏、封装信息的理念。应该用私有变量+成员函数来代替公有变量。一个例外是：如果某个类在本质上是一个数据结构（相当于C语言的结构体），最好令其成员变量都是公有的。
注意C++中的结构体只是为了和C语言兼容，避免使用结构体能够减少所使用的construct，提升代码可读性。请用类代替结构体。

51. 【should】C++指针和引用的操作符应该紧挨着变量类型而不是变量名。

```
float* x; // NOT: float *x;
int& y;   // NOT: int &y;
```

变量的指针和引用是针对类型的属性，而非变量名的属性。C程序员通常采用前一种写法，但在C++中，后一种写法越来越常见。

53. 【should】除了布尔变量和指针之外，不应该使用隐含的方式测试其是否为0。

```
if (nLines != 0)  // NOT: if (nLines)
if (value != 0.0) // NOT: if (value)
```

C++标准并没有规定整数0和浮点数0.0必须由二进制的0来实现。此外，显式的测试还能清楚地表明测试对象的类型。

54. 【should】变量的作用域应该越小越好。

无

把对变量的操作限制在更小的作用域内，容易控制变量所造成的（希望出现和不希望出现）的影响。

5.3 循环

55. 【must】for循环体中只能包含与循环控制有关的语句。

```
sum = 0;                          // NOT: for (i = 0, sum = 0; i < 100; i++)
for (i = 0; i < 100; i++)                   sum += value[i];
  sum += value[i];
```

提高维护性和可读性。将控制循环的语句和循环中的内容明确地区分开来。

56. 【should】循环变量应该在紧靠循环体的位置初始化。

```
isDone = false;             // NOT: bool isDone = false;
while (!isDone) {           //       :
  :                         //      while (!isDone) {
}                           //        :
                            //      }
```

无

57.【can】可以避免使用do-while循环。

无

do-while循环的可读性不如while循环和for循环，因为循环条件在循环体的下方。为了知道循环的范围，读者必须浏览整个循环体。
另外，没有必要使用do-while循环。任何do-while循环都很容易改写成while循环和for循环。减少construct的数量有助于提升可读性。

58.【should】应该避免在循环中使用`continue`和`break`

无

仅当二者的可读性要比他们的structured counterparts更好时才使用它们。

60.【should】对于死循环应该用`while(true)`。

```
while (true) {
  :
}

for (;;) {  // NO!
  :
}

while (1) { // NO!
  :
}
```

测试是否为1既无必要也无意义。`for (;;)`的可读性不太好，也不能明确表示是一个死循环。

5.4 条件表达式

61.【should】避免使用复杂的条件表达式，推荐使用临时的布尔变量[1]。

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}

// NOT:
if ((elementNo < 0) || (elementNo > maxElement)||
     elementNo == lastElement) {
  :
}
```

通过将布尔变量传入表达式，程序变得像文档一样清晰。这样的代码结构容易阅读、调试和维护。

62.【should】使用if-else语句时，正常执行的语句应该放在`if`分支中，处理异常的语句放在`else`分支[1]。

```
bool isOk = readFile (fileName);
if (isOk) {
   :
}
else {
   :
}
```

保证异常处理部分的代码不会模糊代码正常的执行顺序。这一点对于程序的可读性和性能都很重要。

63.【should】条件表达式必须独占一行。

```
if (isDone)         // NOT: if (isDone) doCleanup();
  doCleanup();
```

这一点为了调试着想。如果把二者写在同一行，将很难判断条件表达式的运行结果是`true`还是`false`。

64.【must】条件表达式中不能有可执行语句。

```
File* fileHandle = open(fileName, "w");
if (!fileHandle) {
   :
}

// NOT:
if (!(fileHandle = open(fileName, "w"))) {
   :
}
```

带有可执行语句的条件表达式的可读性很差。尤其是对于C/C++的新手。

5.5 其他

65.【should】应该避免使用“魔法数字”。对于0和1之外的数字都应该考虑将其声明为常量。

无

如果一个数字本身含义不明，将其定义为常量可以提升可读性。另一种可行的方法是定义一个返回该数字的函数。

66.【should】浮点数应该始终有小数点并且至少有一位小数。

```
double total = 0.0;     // NOT:  double total = 0;
double speed = 3.0e8;   // NOT:  double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

这种写法强调了浮点数与整数的不同。在数学中，二者是完全不同的概念。

67.【should】浮点数常量的小数点之前应该总是有数字。

```
double total = 0.5;   // NOT:  double total = .5;
```

C++中的数字和表达式体系借鉴自数学，我们应该始终遵守数学中的惯例。另外，`0.5` 的可读性要比 `.5` 好得多，完全不会和 `5` 混淆。

68.【must】必须始终指定函数返回值的类型。

```
int getValue()    // NOT: getValue()
{
   :
}
```

如果没有显式指定，C++会自动推断返回值的类型为int。程序员永远都不能依赖这种特性，

69.【should】不应该使用 `goto` 。

无

`goto` 违背了代码结构化的初衷。只有在极少数情况下（例如从多层嵌套的循环中跳出），且仅当用于代替 `goto` 的写法可读性更差时，才应该考虑使用 `goto` 语句。

70.【should】应该用 0 替换 `NULL` 。

无

`NULL` 是标准C语言库的内容，但在C++中已经不再使用。

6 布局和注释

6.1 布局

71.【should】基本缩进的大小应该是2。

```
for (i = 0; i < nElements; i++)
  a[i] = 0;
```

1格缩进太小，不足以强调代码的逻辑结构。4格以上的缩进使得多层嵌套的代码难以阅读，同时增加了被迫换行的可能性。在2、3、4中，2格缩进和4格缩进最常见，而2格缩进能够减少了被迫换行的可能性。

72.【should，must】代码块的布局应该参考下面的样例1（推荐）或样例2，禁止像样例3这样。函数体和类必须使用样例2的布局。

```
while (!done) {
  doSomething();
  done = moreToDo();
}
```

```
while (!done)
{
  doSomething();
  done = moreToDo();
}
```

```
while (!done)
  {
    doSomething();
    done = moreToDo();
  }
```

样例3引入了额外的缩进级别，导致代码结构不如样例1和样例2那么清晰。

73.【should】`class`声明应该采用如下格式：

```
class SomeClass : public BaseClass
{
  public:
    ...

  protected:
    ...

  private:
    ...
}
```

该格式符合上述的代码块通用格式。

74.【should】方法的定义应该采用以下格式：

```
void someMethod()
{
  ...
}
```

该格式符合上述的代码块通用格式。

75.【should】`if-else`之类的语句应该采用如下格式：

```
if (condition) {
  statements;
}

if (condition) {
  statements;
}
else {
   statements;
}

if (condition) {
  statements;
}
else if (condition) {
  statements;
}
else {
  statements;
}
```

该格式符合上述的代码块通用格式。但有待讨论的是，`else`语句是否应该和前面`if`或`else if`语句的右花括号在同一行，像这样：

```
if (condition) {
  statements;
} else {
  statements;
}
```

我们推荐的写法比这种写法要好，这是因为前者的`if`、`else`、`else if`等分支在文件单独成行，使得编辑它们比较容易，例如删除某个分支。

76.【should】`for`应该采用如下格式：

```
for (initialization; condition; update) {
  statements;
}
```

该格式符合上述的代码块通用格式。

77.【should】空的`for`语句应该采用如下格式：

```
for (initialization; condition; update)
  ;
```

该格式强调循环体是空的，并告诉读者是有意为之。不过事实上应该避免使用空循环。

78.【should】`while`语句应该采用如下格式：

```
while (condition) {
  statements;
}
```

该格式符合上述的代码块通用格式。

79.【should】`do-while`语句应该采用如下格式：

```
do {
  statements;
} while (condition);
```

该格式符合上述的代码块通用格式。

80.【should】`switch`语句应该采用如下格式：

```
switch (condition) {
  case ABC :
    statements;
    // Fallthrough

  case DEF :
    statements;
    break;

  case XYZ :
    statements;
    break;

  default :
    statements;
    break;
}
```

注意每个`case`分支的整体都相对于`switch`语句都有一级缩进。这样就突出了整个`switch`语句。还有注意：前有一个空格。只要某个`case`分支没有`break`语句，都应该加上`// Fallthrough`注释。漏掉`break`语句是很常见的错误，如果缺少了`break`语句，应该清楚地表明是故意为之。

81.【should】`try-catch`语句应该采用如下格式：

```
try {
  statements;
}
catch (Exception& exception) {
  statements;
}
```

该格式符合上述的代码块通用格式。第75条讨论的`if-else`语句中右花括号的问题同样适用于`try-catch`语句。

82.【can】单分支的 `if-else`、 `for` 、 `while` 语句可以不加大括号。

```
if (condition)
  statement;

while (condition)
  statement;

for (initialization; condition; update)
  statement;
```

通常推荐无论任何时候都加大括号。但是花括号经常用于组合多条语句，对于单个语句来说是多余的。反对这种做法的一个常见理由是：如果添加新的语句却忘了加括号，容易导致程序出错。但是通常情况下，写代码时永远不应该适应将来可能出现的变化。

83.【can】函数返回值可以放在紧邻函数名的上一行。

```
void
MyClass::myMethod(void)
{
  :
}
```

这样更容易定位到文件中的函数名，因为它们都在行首。

6.2 空格

84.【should】
- 传统运算符两边都应该有1个空格
- C++保留字之后应该有1个空格
- 逗号 `,` 之后应该有1个空格
- 冒号 `:` 之后应该有1个空格
- `for` 语句中的分号 `;` 之后应该有1个空格

```
a = (b + c) * d; // NOT: a=(b+c)*d

while (true)    // NOT: while(true)
{
  ...

doSomething(a, b, c, d);  // NOT: doSomething(a,b,c,d);

case 100 :  // NOT: case 100:

for (i = 0; i < 10; i++) {  // NOT: for(i=0;i<10;i++){
  ...
```

突出语句中每个独立的部分。提高可读性。虽然很难给出在C++中使用空格的完整建议，但上述例子给出了大致的思路。

85.【can】如果函数的参数列表非空，函数名之后可以加一个空格。

```
doSomething (currentFile);
```

突出函数名，从而提高可读性。如果函数列表为空，可以不加空格（例如`doSomething()`），因为这时函数名已经足够显眼。
另一种做法是在左括号之后加一个空格。采用这种做法的人通常也会在右括号之前加一个空格：`doSomething( currentFile );`。这样确实也能突出函数名，但右括号之前的空格显得太刻意，不加的话(像`doSomething( currentFile);`)语句看起来又不对称。

86.【should】同一个代码块中的逻辑单元应该用空行分隔开。

```
Matrix4x4 matrix = new Matrix4x4();

double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

matrix.setElement(1, 1,  cosAngle);
matrix.setElement(1, 2,  sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2,  cosAngle);

multiply(matrix);
```

通过在逻辑单元间插入空行来提高可读性。

87.【should】类的成员函数应该用3行空行分隔。

无

使（类成员函数之间的）空白区域大于函数体内的空白区域，从而在类中突出它们。

88.【can】声明时的变量可以左对齐。

```
AsciiFile* file;
int        nPoints;
float      x, y;
```

提高可读性。通过对齐容易从类型中区分出变量名称。

89. 【can】任何能够提高可读性的对齐都可以使用。

```
if      (a == lowValue)    compueSomething();
else if (a == mediumValue) computeSomethingElse();
else if (a == highValue)   computeSomethingElseYet();

value = (potential        * oilDensity)   / constant1 +
        (depth            * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity)   / constant3;

minPosition     = computeDistance(min,     x, y, z);
averagePosition = computeDistance(average, x, y, z);

switch (value) {
  case PHASE_OIL   : strcpy(phase, "Oil");   break;
  case PHASE_WATER : strcpy(phase, "Water"); break;
  case PHASE_GAS   : strcpy(phase, "Gas");   break;
}
```

代码中有很多位置都可以利用空格提高可读性，即使违反了上述的某些建议。其中许多情形都需要使用对齐。虽然很难给出关于代码对齐的通用建议，但上面的这些例子提供了通用的思路。

6.3 注释

90. 【should】棘手的代码不应该注释，而是直接重写！[1]

无

通常，应该通过选择合适的名称和清晰的逻辑结构使得代码的含义不言自明，从而减少注释。

91. 【should】所有注释都应该用英语。 [2]

无

在国际化环境下，英语是最适合的语言。

92. 所有注释都应该用 `//` （行注释），即使有多行。

```
// Comment spanning
// more than one line.
```

由于不支持多级C-commenting，使用行注释总是能够保证在调试时可以通过 `/**/` 注释掉整块代码。

93. 【should】注释的位置应该和被注释对象保持一致 [1]

```
while (true) {          // NOT:  while (true) {
  // Do something                 // Do something
  something();                      something();
}                                 }
```

这样做是为了避免注释破坏代码原本的逻辑结构。

94. 【should】类和方法的头文件的注释应该遵循 JavaDoc 的惯例。
无
说到类和方法的标准化文档，Java开发社区中比C/C++更成熟。这是因为标准且自动化的Javadoc已经成为开发工具的一部分，并且有助于根据注释生成高质量的超文本文档。 C++也有类似于Javadoc的工具，它们采用和Javadoc相同的标记语法，例如 Doc++ or Doxygen。

7 参考资料

[1] Code Complete, Steve McConnell - Microsoft Press

[2] Programming in C++, Rules and Recommendations, M Henricson, e. Nyquist, Ellemtel (Swedish telecom)

[3] Wildfire C++ Programming Style, Keith Gabryelski, Wildfire Communications Inc.

[4] C++ Coding Standard, Todd Hoff

[5] Doxygen documentation system

译者附录

部分单词没有找到合适的译法，在上面的译文中保留了英文形式，根据我搜集的资料尝试解释如下：

- **side effect**（11, 32, 42, 47, 54）：直译是“副作用”，但听起来并不直白。根据维基百科，我的理解是：side effect是指某个操作、函数或表达式在返回目标值之外，还修改了作用域之外的内容，例如：
  - 修改了非局部变量、静态局部变量、通过引用传递的可变参数
  - 执行I/O操作
  - 调用其他会产生side effect的函数
- **concept**（20, 47, 49）：根据维基百科，我的理解是：concept是指某个类可以执行的操作。
- **constructs**（49, 57）：第49条谈到struct结构体和第57条谈到do-while循环时，都提到了construct这个单词，第49条是说class完全可以代替struct，第57条是说完全可以用while循环和for循环代替do-while，这两条建议都在强调：减少使用construct的数量有助于提高代码可读性。根据维基百科对于**language construct**的解释，我认为可以把本文档中的construct理解为**代码的布局或结构**，当用C++来实现某种功能时，代码结构越单一、代码结构形式的种类越少，当然就越容易让别人看懂。