C++/C 编码规范（Code Convention）

1 为什么要有编码规范(Why Have Code Conventions)

编码规范对于程序员而言尤为重要，有以下几个原因：

- 一个软件的生命周期中， 80%的花费在于维护；
- 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护；
- 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码；
- 如果你将源码作为产品发布，就需要确任它是否被很好的打包并且清晰无误，一如你已构建的其它任何产品；

为了执行规范，每个软件开发人员必须一致遵守编码规范。

2 文件名(File Names)

这部分列出了常用的文件名及其后缀。

2.1 文件后缀(File Suffixes)

C/C++程序使用下列文件后缀：

文件类别	文件后缀
C源文件	.c
C++源文件	.cpp
C/C++头文件	.h

3 文件组织(File Organization)

一个 C/C++程序由.h.和.cpp（或.c）文件组成；

- 通常将函数原型说明、外部变量说明、宏定义放到一个.h 文件中（C 程序）。
- 对于 C++程序，每个类对应一个类声明文件（.h 文件）和一个相应类方法实现文件（.cpp），而且它们的文件名前缀应相同，如：MyClass.h 和 MyClass.cpp。
- 每个函数（或方法）的长度原则上不应超过 200 行，过长将导致函数变得复杂而容易出错。可采用函数分解的方法将过长函数简化。

3.1 头文件(Header Files)

- 每个 C/C++头文件的开始及结束应包括如下预处理语句，以避免头文件被重复包含。

```
#ifndef  _HEADER_FILENAME_H
#define  _HEADER_FILENAME_H
 ... Rest of Header File ...
#endif
```

- 头文件中的 include 应分组并排序，底层头文件放在最前面，中间用空行隔开，如：

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

#include <Xm/Xm.h>
#include <Xm/ToggleB.h>

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

3.2 类定义头文件

每个 C++类声明通常遵循以下形式，即首先声明方法（接口），然后声明数据成员：

```
class Class_name: public BaseClass
{
```

```
    public:
        constructor declarations
        destructor declaration
        public member function declarations
    protected:
        protected member function declarations;
    private:
        private member function declarations;

    protected:
        protected data fields
    private:
        private data fields
}
```

4 缩进排版(Indentation)

4个空格常被作为缩进排版的一个单位。缩进的确切解释并未详细指定（空格 vs.制表符）。一个制表符等于 8个空格（而非 4个）。

4.1 行长度(Line Length)

尽量避免一行的长度超过 80个字符，因为很多终端和工具不能很好处理之。

注意：用于文档中的例子应该使用更短的行长，长度一般不超过 70个字符。

4.2 换行(Wrapping Lines)

当一个表达式无法容纳在一行内时，可以依据如下一般规则断开之：

- 在一个逗号后面断开；
- 在一个操作符前面断开；
- 宁可选择较高级别(higher-level)的断开，而非较低级别(lower-level)的断开；
- 新的一行应该与上一行同一级别表达式的开头处对齐；
- 如果以上规则导致你的代码混乱或者使你的代码都堆挤在右边，那就代

之以缩进 8 个空格。

以下是断开方法调用的一些例子：

```
someMethod(longExpression1, longExpression2, longExpression3,
                    longExpression4, longExpression5);

var = someMethod1(longExpression1,
                             someMethod2(longExpression2,
                                         longExpression3));
```

以下是两个断开算术表达式的例子。前者更好，因为断开处位于括号表达式的外边，这是个较高级别的断开。

```
longName1 = longName2 * (longName3 + longName4 - longName5)
                    + 4 * longname6; //PREFFER

longName1 = longName2 * (longName3 + longName4
                                - longName5) + 4 * longname6; //AVOID
```

以下是两个缩进方法声明的例子。前者是常规情形。后者若使用常规的缩进方式将会使第二行和第三行移得很靠右，所以代之以缩进 8个空格。

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
                    Object andStillAnother) {
  ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
ReturnType MyClass:: horkingLongMethodName(int anArg,
```

```
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

if语句的换行通常使用 8个空格的规则，因为常规缩进 (4个空格)会使语句体看起来比较费劲。比如：

```
//DON  T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();       //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

这里有三种可行的方法用于处理三元运算表达式：

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;
alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

5 注释(Comments)

注释是帮助程序读者的一种手段。如果在注释中只说明代码本身已经讲明的事情，或者与代码矛盾，或是以精心编排的形式干扰读者，则它们反而帮了倒忙。好的注释应简洁地说明程序的突出特征，或是提供一种概观，帮助别人理解程序。

C++使用 /*...*/和 //来界定注释。

注释用以注释代码或者实现细节。

注释应被用来给出代码的总括，并提供代码自身没有提供的附加信息。注释应该仅包含与阅读和理解程序有关的信息。

在注释里，对设计决策中重要的或者不是显而易见的地方进行说明是可以的，但应避免提供代码中已清晰表达出来的重复信息。多余的的注释很容易过时。通常应避免那些代码更新就可能过时的注释。

注意：频繁的注释有时反映出代码的低质量。当你觉得被迫要加注释的时候，考虑一下重写代码使其更清晰。

C++程序可以有 4种注释的风格：块 (block)、单行 (single-line)、尾端 (trailing) 和行末 (end-of-line)。

5.1. 块注释(Block Comments)

块注释通常用于提供对文件，类，方法，数据结构和算法的描述。块注释被置于每个文件的开始处以及每个类声明和每个方法之前。它们也可以被用于其他

地方，比如方法内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释之首应该有一个空行，用于把块注释和代码分割开来，比如：

```
/*
 * Here is a block comment.
 */
```

5.2 单行注释(Single-Line Comments)

短注释可以显示在一行内，并与其后的代码具有一样的缩进层级。如果一个注释不能在一行内写完，就该采用块注释（参见“块注释”）。单行注释之前应该有一个空行。以下是一个 C++代码中单行注释的例子：

```
if (condition) {

  /* Handle the condition. */
```

5.3 尾端注释(Trailing Comments)

极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中，它们应该具有相同的缩进。

以下是一个 C++代码中尾端注释的例子：

```
if (a == 2) {
    return TRUE;             /* special case */
} else {
    return isPrime(a);       /* works only for odd a */
}
```

5.4 行末注释(End-Of-Line Comments)

注释界定符 "//"，可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本；然而，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;           // Explain why here.
}

//if (bar > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

5.5 函数/方法注释

对每个 C++的类方法及函数建议包含如下格式说明信息的注释，即说明其功能、参数及返回值。对于类方法注释可放在类定义中方法说明的后面，而对于普通函数注释可放在函数实现之前。该形式注释已被 Java编程规范所采用，并有相应工具自动将方法 /函数说明信息抽取成 HTML文档。

```
/**
```

```
    Computes the maximum of two integers.
    @param x an integer
    @param y an integer
    @return the larger of the two inputs
*/
int max(int x, int y)
{
    if ( x > y)
        return x ;
    else
        return y;
}
```

文件头注释

好的程序设计风格源文件头应该有一个注释块，说明该文件的用途、开发者等信息，以便于文件的维护。下面是一个文件头的注释样例：

```
/*
** FILE: filename.cpp
**
** ABSTRACT:
**   A general description of the module's role in the
**   overall software architecture,  What services it
**   provides and how it interacts with other components.
**
** DOCUMENTS:
**   A reference to the applicable design documents.
**
** AUTHOR:
**   Your name here
**
```

```
 ** CREATION DATE:
**   14/03/1998
**
** NOTES:
**   Other relevant information
   */
```

6 声明(Declarations)

6.1 每行声明变量的数量(Number Per Line)

推荐一行一个声明，因为这样有利于写注释。亦即，

```
int level;  // indentation level
int size;   // size of table
```

要优于，

```
int level, size;
```

不要将不同种类变量的声明放在同一行，例如：

```
int foo,  fooarray[];   //WRONG!
```

注意：在上面的例子中，在类型和标识符之间放了一个空格，另一种被允许的替代方式是使用制表符：

```
int             level;          // indentation level
int             size;           // size of table
Object          currentEntry;   // currently selected table entry
```

定义指针及引用变量时，指针及引用符号应放在相应变量名之前，如：

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

6.2 初始化(Initialization)

尽量在声明局部变量的同时初始化，以保证变量有一个合法值。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

6.3 布局(Placement)

只在代码块的开始处声明变量。（一个块是指任何被包含在大括号 "{"和 "}" 中间的代码。）不要在首次用到该变量时才声明之。这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。

```
void myMethod() {
    int int1 = 0;         // beginning of method block

    if (condition) {
        int int2 = 0;     // beginning of "if" block
        ...
    }
}
```

该规则的一个例外是 for循环的索引变量：

```
for (int i = 0; i < maxLoops; i++) { ... }
```

避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名：

```
int count;
...
myMethod() {
    if (condition) {
```

```
        int count = 0;      // AVOID!
        ...
    }
    ...
}
```

6.4 类和接口的声明(Class and Interface Declarations)

当编写类和接口是，应该遵守以下格式规则：

- 在方法名与其参数列表之前的左括号 "(" 间不要有空格；
- 左大括号 "{" 位于声明语句同行的末尾；
- 右大括号 "}" 另起一行，与相应的声明语句对齐，除非是一个空语句， "}" 应紧跟在 "{" 之后 ；

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

    ...
}
```

- 方法与方法之间以空行分隔；

7 语句(Statements)

7.1 简单语句(Simple Statements)

每行至多包含一条语句，例如：

```
argv++;         // Correct
argc--;         // Correct
argv++; argc--;         // AVOID!
```

7.2 复合语句(Compound Statements)

复合语句是包含在大括号中的语句序列，形如"{ 语句 }"。例如下面各段。

- 被括其中的语句应该较之复合语句缩进一个层次；
- 左大括号"{"应位于复合语句起始行的行尾；右大括号"}"应另起一行并与复合语句首行对齐；
- 大括号可以被用于所有语句，包括单个语句，只要这些语句是诸如 if-else 或 for控制结构的一部分。这样便于添加语句而无需担心由于忘了加括号而引入 bug。

7.3 返回语句(return Statements)

一个带返回值的 return语句不使用小括号"()"，除非它们以某种方式使返回值更为显见。例如：

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

7.4 if，if-else，if else-if else 语句(if, if-else, if else-if else Statements)

if-else语句应该具有如下格式：

```
if (condition) {
    statements;
}

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

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

注意：

1) if语句总是用 "{"和 "}"括起来，避免使用如下容易引起错误的格式：

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

2) 应将正常情况放到 if为真的部分，异常情况放到 else部分，如：

```
isError = readFile (fileName);
if (!isError) {
```

```
    :
  }
  else {
    :
  }
```

3）应避免在条件中出现执行表达式，如：

```
  // Bad!
if (!(fileHandle = open (fileName, "w"))) {
  :
}

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

7.5 for 语句(for Statements)

一个 for语句应该具有如下格式：

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

一个空的 for语句 (所有工作都在初始化，条件判断，更新子句中完成）应该具有如下格式：

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

当在 for语句的初始化或更新子句中使用逗号时，避免因使用三个以上变量，而导致复杂度提高。若需要，可以在 for循环之前 (为初始化子句)或 for循环末尾 (为更新子句)使用单独的语句。

应只有循环变量才放到 for循环语句的头部，以提高程序的可读性和可维护性，如：

```
sum = 0;
for (i = 0; i < 100; i++)
    sum += value[i];
```

而不要：

```
for (i = 0, sum = 0; i < 100; i++)
    sum += value[i];
```

7.6 while 语句(while Statements)

一个 while语句应该具有如下格式：

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

一个空的 while语句应该具有如下格式：

```
while (condition);
```

对于 while循环，循环变量应在循环开始前初始化，如：

```
isDone = false;
while (!isDone) {
    :
}
```

而不要：

```
bool isDone = false;
   :
 while (!isDone) {
     :
  }
```

7.7 do-while 语句(do-while Statements)

一个 do-while语句应该具有如下格式：

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

注意：应在程序中尽量减少对 do-while语句的使用，因为它的可读性不如for和 while语句。

7.8 switch 语句(switch Statements)

一个 switch语句应该具有如下格式：

```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;
```

```
default:
    statements;
    break;
}
```

每当一个 case顺着往下执行时 (因为没有 break语句)，通常应在 break语句的位置添加注释。上面的示例代码中就包含注释 /* falls through */。

7.9 try-catch 语句(try-catch Statements)

一个 try-catch语句应该具有如下格式：

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

8 空白(White Space)

8.1 空行(Blank Lines)

空行将逻辑相关的代码段分隔开，以提高可读性。

下列情况应该总是使用两个空行：

- 一个源文件的两个片段 (section)之间；

下列情况应该总是使用一个空行：

- 两个方法之间；
- 方法内的局部变量和方法的第一条语句之间；

- 块注释（参见 "5.1"）或单行注释（参见 "5.2"）之前；
- 一个方法内的两个逻辑段之间，用以提高可读性

8.2 空格(Blank Spaces)

下列情况应该使用空格：

- 一个紧跟着括号的关键字应该被空格分开，例如：

```
while (true) {
    ...
}
```

注意：空格不应该置于方法名与其左括号之间。这将有助于区分关键字和方法调用。

- 空白应该位于参数列表中逗号的后面；
- 所有的二元运算符，除了 "."，应该使用空格将之与操作数分开。一元操作符和操作数之间不因该加空格，比如：负号 ("-")、自增 ("++") 和自减 ("--")。例如：

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- for语句中的表达式应该被空格分开，例如：

```
for (expr1; expr2; expr3)
```

- 强制转换后应该跟一个空格，例如：

```
myMethod((int) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

9 命名规范(Naming Conventions)

命名规范使程序更易读，从而更易于理解。同时，它们也可以提供一些有关标识符功能的信息，以助于理解代码。

标识符类型	命名规则	例子
类 (Classes)	命名规则：类名是个一名词，采用大小写混合的方式，每个单词的首字母大写。尽量使类名简洁而富于描述。使用完整单词，避免缩写词（除非该缩写词被更广泛使用，像 URL，HTML）	class Raster; class ImageSprite;
模板 (Template)	模板名应为单个大写字母。	template< class T> ... template< class C, class D> ...

方法/函数 (Methods or Functions)	方法名或函数名是一个动词，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。	run();
		runFast();
		getBackground();

变量 (Variables)	变量名均采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。变量名不应以下划线，尽管这在语法上是允许的。变量名应简短且富于描述。变量名的选用应该易于记忆，即，能够指出其用途。尽量避免单个字符的变量名，除非是一次性的临时变量，如循环变量。临时变量通常被取名为 i，j，k，m 和 n。	char c; int i; float myWidth;
常量 (Constants)	常量应该全部大写，单词间用下划线隔开。	const int MIN_WIDTH = 4;

参考文献（**Reference**）

1) 《Java Code Convention》, from Sun Microsystems Inc.
2) 《C++ Programming Style Guidelines》, *Version 3.0, January 2002 Geotechnical Software Services Copyright © 1996 – 2002*
3) 《Coding Conventions for C++ and Java applications》， Macadamian Technologies Inc
4) 《software Testing in the Real World：Improve to Process》
5) 《The Practice of Programming》, Brian W. Kernighan, Rob Pike 1999
6) The C++ Programming Language, Bjarne Stroustrup, 0-201-88954-4, Addison-Wesley, 1997.
7) Code Complete: A Practical Handbook of Software Construction, Code Complete, Steve McConnell, Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Effective C++, Scott Meyers, 0-201-92488-9, Addison-Wesley, 1998.
8) *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, More
9) Effective C++, Scott Meyers, 0-201-63371-X, 1996, Addison-Wesley, 1998.
10) Software Testing In The Real World : Improving The Process, Edward Kit.