前言

有感于先前Qt工程中的各源文件的代码规范并不统一，故参考了《华为C++语言编程规范》、《google C++编程指南》以及《高质量C++C 编程指南 林锐》编写了一份适用于本公司的C++语言编程规范文档，来帮助团队合理使用 C++，快速上手公司已有项目代码。

一、文件结构

每个 C++/C 程序通常分为两个文件。一个文件用于保存程序的声明（declaration），称为头文件。另一个文件用于保存程序的实现，称为定义（definition）文件。

C++/C 程序的头文件以 “.h” 为后缀，C 程序的定义文件以 “.c” 为后缀，C++ 程序的定义文件通常以 “.cpp” 为后缀（也有一些系统以 “.cc” 或 “.cxx” 为后缀）。

1.1 版权和版本的声明

版权和版本的声明位于头文件的开头，主要内容有：

1）版权信息；

2）文件名称，标识符，摘要；

3）当前版本号，作者/修改者，完成日期；

4）版本历史信息 。

```
/*
* Copyright (c) 2019,浙江智澜科技有限公司
* All rights reserved.
*
* 文件名称：fileName.h
* 摘 要：简要描述本文件的功能和用法
*
* 当前版本：1.1
* 作 者：输入作者（或修改者）名字
* 完成日期：2019 年 7 月 20 日
*
* 取代版本：1.0
* 原作者 ：输入原作者（或修改者）名字
* 完成日期：2019 年 5 月 10 日
*/
```

1.2 头文件的结构

头文件由三部分内容组成：

（1）头文件开头处的版权和版本声明。 （2）预处理块。 （3）函数和类结构声明等。

【规则 1-2-1】为了防止头文件被重复引用，应当用 `ifndef/define/endif` 结构产生预处理块。

【规则 1-2-2】用 `#include <filename.h>` 格式来引用标准库的头文件（编译器将从标准库目录开始搜索）。

【规则 1-2-3】用 `#include “filename.h”` 格式来引用非标准库的头文件（编译器将从用户的工作目录开始搜索）。

【规则 1-2-4】头文件中只存放 “声明” 而不存放 “定义” 。

在 C++ 语法中，类的成员函数可以在声明的同时被定义，并且自动成为内联函数。这虽然会带来书写上的方便，但却造成了风格不一致，弊大于利。建议将成员函数的定义与声明分开，不论该函数体有多么小。

【规则 1-2-5】不提倡使用全局变量，尽量不要在头文件中出现象 extern int value 这类声明。

```
// 版权和版本声明见示上例，此处省略。

#ifndef GRAPHICS_H // 防止 graphics.h 被重复引用
#define GRAPHICS_H

#include <math.h> // 引用标准库的头文件
…
#include “myheader.h” // 引用非标准库的头文件
…
void fun(…); // 全局函数声明
…
class Box // 类结构声明
{
    …
};

#endif
```

1.3 头文件依赖

【规则 1-3-1】使用**前置声明**尽量减少 .h 文件中 #include 的数量。

当一个头文件被包含的同时也引入了一项新的依赖 ，只要该头文件被修改，代码就要重新编译。 如果你的头文件包含了其他头文件， 这些头文件的任何改变也将导致那些包含了你的头文件的代码重新编译。因此，我们宁可尽量少包含头文件，尤其是那些包含在其他头文件中的。

使用前置声明可以显著减少需要包含的头文件数量。举例说明：头文件中用到类 File，**但不需要访问File的声明**，则头文件中只需前置声明 `class File`；无需 `#include "file/base/file.h"`。

在头文件如何做到使用类 Foo而无需访问类的定义?

1) 将数据成员类型声明为 Foo *或Foo &;

2) 参数、返回值类型为 Foo的函数只是声明（但不定义实现）；

3) 静态数据成员的类型可以被声明为 Foo，因为静态数据成员的定义在类定义之外。

1.4 包含文件的次序

【规则 1-4-1】将包含次序标准化可增强可读性，次序如下： C库、 C++库、其他库的.h、项目内的.h。

项目内头文件应按照项目源代码目录树结构排列，并且避免使用 UNIX文件路径 `.`（当前目录）和 `..`（父目录）。例如， `google-project/src/base/logging.h` 应像这样被包含：`#include "base/logging.h` 。示例如下：

```
#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>
#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/public/bar.h
```

1.5 目录结构

【规则 1-5-1】如果一个软件的头文件数目比较多（如超过十个），通常应将头文件和定义文件分别保存于不同的目录，以便于维护。

例如可将头文件保存于 include 目录，将定义文件保存于 source 目录（可以是多级目录）。

如果某些头文件是私有的，它不会被用户的程序直接引用，则没有必要公开其“声明”。为了加强信息隐藏，这些私有的头文件可以和定义文件存放于同一个目录。

二、程序的版式

版式虽然不会影响程序的功能，但会影响可读性。程序的版式追求清晰、美观，是程序风格的重要构成因素。

可以把程序的版式比喻为“书法”。好的“书法”可让人对程序一目了然，看得兴致勃勃。差的程序“书法”如螃蟹爬行，让人看得索然无味，更令维护者烦恼有加。 所以学习程序的“书法”， 很有必要。

2.1 空格还是制表位

只使用空格，每次缩进4个空格。有些人更加青睐每次缩进2个空格，也是可以的，这个纯粹看个人喜好，但如果是团队协作的话需要统一。

【规则 2-1-1】只使用空格进行缩进，不要在代码中使用 tabs，设定编辑器将tab转为空格。（有些编辑器已经这样默认设置了，例如Qt）

2.2 空行

空行起着分隔程序段落的作用。空行得体（不过多也不过少）能将使程序的布局更加清晰。空行不会浪费内存，虽然打印含有空行的程序是会多消耗一些纸张，但是值得。所以不要舍不得用空行。

【规则 2-2-1】在每个类声明之后、每个函数定义结束之后都要加空行。

【规则 2-2-2】在一个函数体内，逻揖上密切相关的语句之间不加空行，其它地方应加空行分隔。

```
while (condition)
{
    statement1;
    // 空行
    if (condition)
    {
        statement2;
    }
    // 空行
    statement3;
}
```

2.3 代码行

【规则 2-3-1】 一行代码只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便写注释。

```
// 风格良好的代码行
int width; // 宽度
int height; // 高度
int depth; // 深度

// 风格不良的代码行
int width, height, depth; // 宽度 高度 深度
```

【规则 2-3-2】 if、for、while、do 等语句自占一行，执行语句不得紧跟其后。 不论执行语句有多少都要加{}。这样可以防止书写失误。

```
// 风格良好的代码行
if (width < height)
{
    doSomething();
}

// 风格不良的代码行
if (width < height) doSomething();
```

【建议 2-3-3】尽可能在定义变量的同时初始化该变量（就近原则）

如果变量的引用处和其定义处相隔比较远，变量的初始化很容易被忘记。如 果引用了未被初始化的变量，可能会导致程序错误。本建议可以减少隐患。例如：

```
int width = 10; // 定义并初始化 width
int height = 10; // 定义并初始化 height
int depth = 10; // 定义并初始化 depth
```

2.4 代码行内的空格

【规则 2-4-1】关键字之后要留空格。像 if、for、while 等关键字之后应留一个空格再跟左括号‘(’，以突出关键字。

【规则 2-4-2】函数名之后不要留空格，紧跟左括号‘(’，以与关键字区别。

【规则 2-4-3】‘，’之后要留空格，如 `fun(x, y, z)`。如果‘;’不是一行的结束符号，其后要留空格，如 `for (initialization; condition; update)`。

【规则 2-4-4】赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如“=”、“+=”“>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<”、“^”等二元操作符的前后要加上空格。

【规则 2-4-5】一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。

【规则 2-4-6】像“[]”、“.”、“->”这类操作符前后不加空格。

【建议 2-4-7】对于表达式比较长的 for 语句和 if 语句，为了紧凑起见可以适当地去掉一些空格，如 `for (i=0; i<10; i++)` 和 `if ((a<=b) && (c<=d))`。

```
void fun(int x, int y, int z); // 良好的风格
void fun (int x,int y,int z); // 不良的风格

if ((a>=b) && (c<=d)) // 良好的风格
if(a>=b&&c<=d) // 不良的风格

for (i=0; i<10; i++) // 良好的风格
for(i=0;i<10;i++) // 不良的风格
for (i = 0; i < 10; i ++) // 过多的空格

array[5] = 0; // 不要写成 array [ 5 ] = 0;
a.Function(); // 不要写成 a . Function();
b->Function(); // 不要写成 b -> Function();
```

2.5 对齐

【规则 2-5-1】程序的分界符‘{’和‘}’应独占一行并且位于同一列，同时与引用它们的语句左对齐。

【规则 2-5-2】{}之内的代码块在‘{’右边缩进后再左对齐。

```
// 良好的风格
void function(int x)
{
    doSomething();
    other();
}

// 不良的风格
void function(int x) {
    doSomething();
    other();
}
```

2.6 长行拆分

【规则 2-6-1】代码行最大长度宜控制在 70 至 80 个字符以内。代码行不要过长，否则眼睛看不过来，也不便于打印。

【规则 2-6-2】长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

【规则 2-6-3】构造函数初始化列表过长，可以按4格缩进并排几行。

```
// 良好的风格
if (theOneThing > ONE
    && theThridThing == TWO
    && yetAnother == LAST)
{
    doSomething();
}

// 良好的风格
MyClass::MyClass(int var)
    : some_var_(var),
      some_other_var_(var + 1)
{
    doSomething();
}
```

2.7 修饰符的位置

修饰符 * 和 & 应该靠近数据类型还是该靠近变量名，是个有争议的话题。

若将修饰符 * 靠近数据类型，例如：`int* x;` 从语义上讲此写法比较直观，即 x 是 int 类型的指针。

上述写法的弊端是容易引起误解，例如：`int* x, y;` 此处 y 容易被误解为指针变量。虽然将 x 和 y 分行定义可以避免误解，但并不是人人都愿意这样做。

【规则 2-7-1】应当将修饰符 * 和 & 紧靠变量名。

```
// 良好的风格
char *name;
int *x, y; // 此处 y 不会被误解为指针
```

2.8 函数参数顺序

【规则 2-8-1】定义函数时，参数顺序为：输入参数在前，输出参数在后。

C/C++函数参数分为输入参数和输出参数两种， 有时输入参数也会输出 （值被修改时）。 输入参数一般传值或常数引用 ， 输出参数或输入/输出参数为非常数指针 。

对参数排序时，将所有输入参数置于输出参数之前。不要仅仅因为是新添加的参数，就将其置于最后，而应该依然置于输出参数之前。

三、命名规则

比较著名的命名规则当推 Microsoft 公司的“匈牙利”法，该命名规则的主要思想是“在变量和函数名中加入前缀以增进人们对程序的理解”。例如所有的字符变量均以 ch 为前缀，若是指针变量则追加前缀 p。

“匈牙利”法最大的缺点是烦琐，例如：

```
int i, j, k;
float x, y, z;
```

倘若采用“匈牙利”命名规则，则应当写成：

```
int iI, iJ, ik; // 前缀 i 表示 int 类型
float fX, fY, fZ; // 前缀 f 表示 float 类型
```

如此烦琐的程序会让绝大多数程序员无法忍受。

据考察，没有一种命名规则可以让所有的程序员赞同，程序设计教科书一般都不指定命名规则。命名规则对软件产品而言并不是“成败悠关”的事，我们不要花太多精力试图发明世界上最好的命名规则，而应当制定一种令大多数项目成员满意的命名规则，并在项目中贯彻实施。

3.1 共性规则

本节论述的共性规则是被大多数程序员采纳的，我们应当在遵循 这些共性规则的前提下，再扩充特定的规则。

【规则 3-1-1】标识符应当直观且可以拼读，可望文知意，不必进行 “解码”。

标识符最好采用英文单词或其组合，便于记忆和阅读。切忌使用汉语拼音来命名。程序中的英文单词一般不会太复杂，用词应当准确。例如不要把 CurrentValue 写成 NowValue。

【规则 3-1-2】命名规则尽量与所采用的操作系统或开发工具的风格保持一致。

例如 Windows 应用程序的标识符通常采用 “大小写” 混排的方式，如 addChild。而 Unix 应用程序的标识符通常采用 “小写加下划线” 的方式，如 add_child。别把这两类风格混在一起用。

【规则 3-1-3】程序中不要出现仅靠大小写区分的相似的标识符。

```
int x, X; // 变量 x 与 X 容易混淆

void foo(int x); // 函数 foo 与 FOO 容易混淆
void FOO(float x);
```

【规则 3-1-4】程序中不要出现标识符完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但会使人误解。

【规则 3-1-5】变量的名字应当使用“名词”或者“形容词 + 名词”。

```
float value;
float oldValue;
float newValue;
```

【规则 3-1-6】函数的名字应当使用“动词”或者“动词 + 名词”（动宾词组）。

```
drawBox(); // 普通函数
box->draw(); // 类的成员函数
```

【规则 3-1-7】用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

```
int minValue;
int maxValue;

int SetValue(…);
int GetValue(…);
```

【规则 3-1-8】尽量避免名字中出现数字编号，如 Value1,Value2 等，除非逻辑上的确需要编号。这是为了防止程序员偷懒，不肯为命名动脑筋而导致产生无意义的名字（因为用数字编号最省事）。

【规则 3-1-8】除非缩写放到项目外也非常明确，否则不要使用缩写。

```
// 良好的风格
int num_dns_connections; // Most people know what "DNS" stands for
int price_count_reader; // OK, price count. Makes sense

// 不良的风格
int wgc_connections; // Only your group knows what this stands for
int pc_reader; // Lots of things can be abbreviated "pc"
```

3.2 简单的 Windows 应用程序命名规则

作者对 “匈牙利” 命名规则做了合理的简化，下述的命名规则简单易用，比较适合于Windows 应用软件的开发。

【规则 3-2-1】文件命名使用 “小驼峰命名法”，除第一个单词之外，其他单词首字母大写。

```
lockScreenW.h
changePasswdW.cpp
```

【规则 3-2-2】无论是普通函数还是成员函数的命名，都使用 “小驼峰命名法”，除第一个单词之外，其他单词首字母大写。

```
// 普通函数
void setAge()
{
...
}

// 成员函数
class MyClass
{
public:
    void setAge(int age);
}
```

【规则 3-2-3】结构体、类型定义（typedef）、枚举等所有类型，均使用“大驼峰命名法”，所有单词首字母大写。

```
// classes and structs
class UrlTable { ...
struct UrlTableProperties { ...

// typedefs
typedef hash_map<UrlTable*, string> UrlTableMap;

// enums
enum UrlTableErrors { ...
```

【规则 3-2-4】变量和参数命名使用“小驼峰命名法”，除第一个单词之外，其他单词首字母大写。

```
bool flag;
int drawMode;

class MyClass :
{
private:
    string m_tableName;
};
```

【规则 3-2-5】无论是宏常量还是普通常量的命名，都全用大写的字母，用下划线分割单词。

```
// 宏常量
#define MAX_ARRAY_LEN 100

// 普通常量
const int MAX_ARRAY_LEN = 100;
const float PI = 3.14159;
```

【规则 3-2-6】如果不得已需要全局变量，则使全局变量加前缀 g_（表示global），即“匈牙利+小驼峰命名法”。

```
int g_howManyPeople; // 全局变量
```

【规则 3-2-7】静态变量加前缀 s_（表示 static），即 “匈牙利+小驼峰命名法”。

```
void init()
{
    static int s_initValue; // 静态变量
}
```

【规则 3-2-8】类的数据成员加前缀 m_（表示 member），即 “匈牙利+小驼峰命名法”，这样可以避免数据成员与成员函数的参数同名。

```
void Object::SetValue(int width, int height)
{
    m_width = width;
    m_height = height;
}
```

【规则 3-2-9】枚举值推荐全部大写，单词间以下划线相连。

```
enum UrlTableErrors
{
    OK = 0,
    ERROR_OUT_OF_MEMORY,
    ERROR_MALFORMED_INPUT,
};
```

四、注释

C++语言中，程序块的注释常采用 “/*...*/”，行注释一般采用 “//...”。注释通常用于：

（1）版本、版权声明；（2）函数接口说明；（3）重要的代码行或段落提示。

虽然注释有助于理解代码，但注意不可过多地使用注释。

4.1 类注释

【规则 4-1-1】每个类的定义要添加描述类的功能和用法的注释。

如果你觉得已经在文件顶部详细描述了该类，想直接简单的来上一句“完整描述见文件顶部”的话，还是多少在类中加点注释吧。

如果该类的实例可被多线程访问，使用时务必注意备注说明一下。

4.2 函数注释

函数声明处注释描述函数功能，定义处描述函数实现。

函数声明

【规则 4-2-1】函数声明处的注释，只描述函数功能及用法，而不会描述函数如何实现，因为那是定义部分的事情。

```
void setAge(int age); // 设置学生年龄
```

函数定义：

【规则 4-2-2】每个函数定义时要以注释说明函数功能和实现要点，如使用的漂亮代码、实现的简要步骤、如此实现的理由等等。

不要从 .h 文件或其他地方的函数声明处直接复制注释，简要说明函数功能是可以的，但重点要放在如何实现上。

```
/*
* 函数介绍：设置学生年龄
* 函数实现：将参数age的值赋给成员变量m_age
* 输入参数：age-传入的学生年龄值
* 返回值 ：NULL
* 注意事项：NULL
*/
void setAge(int age)
{
    m_age = age;
}
```

4.3 注释风格

关于注释风格，很多 C++ 的 coders 更喜欢行注释， C coders 或许对块注释依然情有独 钟，或者在文件头大段大段的注释时使用块注释。

【规则 4-3-1】对于行注释，注释与 `//` 留一个空格，若是注释在程序右侧，则 `//` 与程序之间留一个空格。

```
// 注释1
fun1();

fun2(); // 注释2
```

五、表达式和基本语句

表达式和语句都属于 C++/C 的短语结构语法。它们看似简单，但使用时隐患比较多。本节归纳了正确使用表达式和语句的一些规则与建议。

5.1 运算符的优先级

【规则 5-1-1】如果代码行中的运算符比较多，用括号确定表达式的操作顺序，避免使用默认的优先级。

```
word = (high << 8) | low
if ((a | b) && (a & c))
```

5.2 复合表达式

如 `a = b = c = 0` 这样的表达式称为复合表达式。允许复合表达式存在的理由是：（1）书写简洁；（2）可以提高编译效率。但要防止滥用复合表达式

【规则 5-2-1】不要编写太复杂的复合表达式。

```
i = a >= b && c < d && c + f <= g + h ; // 复合表达式过于复杂
```

【规则 5-2-2】不要有多用途的复合表达式。

```
d = (a = b + c) + r ; // 该表达式既求 a 值又求 d 值。
```

5.3 if 语句

if 语句是 C++/C 语言中最简单、最常用的语句，然而很多程序员用隐含错误的方式写 if 语句。本节以“与零值比较”为例，展开讨论。

【规则 5-3-1】不可将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较。

```
// 良好的风格
if (flag) // 表示 flag 为真
if (!flag) // 表示 flag 为假

// 不良的风格
if (flag == TRUE)
if (flag == 1 )
if (flag == FALSE)
if (flag == 0)
```

【规则 5-3-2】整型变量与零值比较。

```
// 良好的风格
if (value == 0)
if (value != 0)

// 不良的风格
if (value) // 会让人误解 value 是布尔变量
if (!value)
```

【规则 5-3-3】不可将浮点变量用“==”或“！=”与任何数字比较。

```
// 良好的风格
if ((f>=-EPSINON) && (f<=EPSINON)) // EPSINON 是允许的误差（即精度）

// 不良的风格
if (f == 0.0) // 隐含错误的比较
```

【规则 5-3-4】应当将指针变量用“==”或“！=”与 NULL 比较。

```
// 良好的风格
if (p == NULL) // p 与 NULL 显式比较，强调 p 是指针
if (p != NULL)

// 不良的风格
if (p == 0) // 容易让人误解 p 是整型变量
if (p != 0)
if (p) // 容易让人误解 p 是布尔变量
if (!p)
```

【规则 5-3-5】提倡 if 和左圆括号间保留一个空格，且不在圆括号中添加空格 。

```
// 良好的风格
if (condition) // if 和左圆括号间有个空格，且不在圆括号中添加空格
{
    ...
}
else // 关键字else另单独起一行
{
    ...
}
```

5.4 循环语句的效率

C++/C 循环语句中，for 语句使用频率最高，while 语句其次，do 语句很少用。本节重点论述循环体的效率。提高循环体效率的基本办法是降低循环体的复杂性。

【规则 5-4-1】在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少 CPU 跨切循环层的次数。

```
//  低效率：长循环在最外层
for (row=0; row<100; row++)
{
    for (col=0; col<5; col++ )
    {
        sum = sum + a[row][col];
    }
}

// 高效率：长循环在最内层
for (col=0; col<5; col++ )
{
    for (row=0; row<100; row++)
    {
        sum = sum + a[row][col];
    }
}
```

【规则 5-4-2】如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体的外面。

```
//  效率低但程序简洁
for (i=0; i<N; i++)
{
    if (condition)
    {
        doSomething();
    }
    else
    {
        doSomething();
    }
}

// 效率高但程序不简洁
if (condition)
{
    for (i=0; i<N; i++)
        doSomething();
    }
else
{
    for (i=0; i<N; i++)
        doSomething();
}
```

5.5 for 语句的循环控制变量

【规则 5-5-1】如不可在 for 循环体内修改循环变量，防止 for 循环失去控制。

【规则 5-5-2】建议 for 语句的循环控制变量的取值采用 “半开半闭区间” 写法。

```
// (a)循环变量属于半开半闭区间
for (int x=0; x<N; x++)
{
    doSomething();
}

// (b)循环变量属于闭区间
for (int x=0; x<=N-1; x++)
{
    doSomething();
}
```

示例 (a) 中的 x 值属于半开半闭区间 “0 =< x < N”，起点到终点的间隔为 N，循环次数为 N。

示例 (b) 中的 x 值属于闭区间 “0 =< x <= N-1”，起点到终点的间隔为 N-1，循环次数为N。

相比之下，示例 (a) 的写法更加直观，尽管两者的功能是相同的。

5.6 switch 语句

【规则 5-6-1】switch 语句中的 case 块可以使用大括号也可以不用， 取决于你的喜好。

【规则 5-6-2】如果有不满足 case 枚举条件的值，要总是包含一个 default（如果有输入值没有 case 去处理，编译器将报警）。如果 default 永不会执行，可以简单的使用assert。

```
switch (var)
{
    case 0:
    {
        ...
        break;
    }

    default:
    {
        assert(false);
    }
}
```

六、函数设计

函数是 C++/C 程序的基本功能单元，其重要性不言而喻。函数设计的细微缺点很容易导致该函数被错用，所以光使函数的功能正确是不够的。本节重点论述函数的接口设计和内部实现的一些规则。

6.1 函数声明与定义

返回类型和函数名在同一行，合适的话，参数也放在同一行。函数看上去像这样：

```
returnType ClassName::funName(Type parName1, Type parName2)
{
    doSomething();
    ...
}
```

如果同一行文本较多，容不下所有参数：

```
returnType ClassName::reallyLongFunctionName(Type parName1,
                                             Type parName2
                                             Type parName3)
{
    doSomething();
    ...
}
```

甚至连第一个参数都放不下：

```
returnType ClassName::reallyLongFunctionName(
    Type parName1, // 4 space indent
    Type parName2
    Type parName3)
{
    doSomething();
    ...
}
```

注意以下几点：

1）左圆括号总是和函数名在同一行；

2）函数名和左圆括号间没有空格；

3）圆括号与参数间没有空格；

4）左大括号总在最后一个参数下一行的行首（Windows风格）；

5）所有形参应尽可能对齐，且形参之间留一个空格。

注：关于UNIX/Linux风格为什么要把左大括号置于行尾，我的理解是代码看上去比较简约，想想行首除了函数体被一对大括号封在一起之外，只有右大括号的代码看上去确实也舒服；而Windows风格将左大括号置于行首的优点是匹配情况一目了然。

6.2 函数调用

函数调用尽量放在同一行，实参之间留一个空格。函数调用遵循如下形式：

```
bool retval = doSomething(argument1, argument2, argument3);
```

如果同一行放不下，可断为多行，后面每一行都和第一个实参对齐，左圆括号后和右圆括号前不要留空格：

```
bool retval = doSomething(reallyReallyReallyLongArgument1,
                          argument2, argument3)
```

如果函数参数比较多，可以出于可读性的考虑每行只放一个参数：

```
bool retval = DoSomething(argument1,
                          argument2,
                          argument3,
                          argument4);
```

6.3 函数返回值

函数返回时，return表达式中不要使用圆括号：

```
return x; //not return(x);
```

七、类

类是C++中基本的代码单元，自然被广泛使用。本节列举了在写一个类时要做什么、不要做什么。

7.1 构造函数的职责

构造函数中只进行那些没有实际意义的（对于程序执行没有实际的逻辑意义） 初始化，可能的话，使用 `init()` 方法集中初始化有意义的数据。

为避免隐式转换，需将单参数构造函数声明为 `explicit` 。

7.2 存取控制

将数据成员私有化，并提供相关存取函数，如定义变量 `foo` 及取值函数 `getFoo()`、赋值函数 `setFoo()`。存取函数的定义一般内联在头文件中。

7.3 声明次序

在类中的声明次序如下：`public:`、`protected:`、`private:`，定义次序如果那一块没有，直接忽略即可。

每一块中，声明次序一般如下：

1) typedefs 和 enums；

2) 常量；

3) 构造函数；

4) 析构函数；

5) 成员函数，含静态成员函数；

6) 数据成员，含静态数据成员。

另外 cpp 文件中函数的定义应尽可能和声明次序一致。

八、其他 C++ 特性

下面介绍一些使 C++ 代码更加健壮的技巧和使用方式。

8.1 引用参数

按引用传递的参数必须加上 const。

定义：在C语言中， 如果函数需要修改变量的值，形参必须为指针， 如 `int foo(int *pval)`。在C++中，函数还可以声明引用形参： `int foo(int &val)`。

优点：定义形参为引用避免了像 `(*pval)++` 这样丑陋的代码。

缺点：容易引起误解，因为引用在语法上是值却拥有指针的语义。

结论：

函数形参表中，所有引用必须是 const：

```
void Foo(const string &in, string *out);
```

事实上这是一个硬性约定：输入参数为值或常数引用，输出参数为指针；输入参数可以是常数指针，但不能使用非常数引用形参。

8.2 类型转换

使用 `static_cast<>()` 等 C++ 的类型转换，不要使用 `int y = (int)x` 或 `int y = int(x);`。

定义：C++ 引入了有别于 C 的不同类型的类型转换操作。

优点：C 语言的类型转换问题在于操作比较含糊：有时是在做强制转换（如 `(int)3.5`），有时是在做类型转换（如 `(int)"hello"`）。另外， C++ 的类型转换查找更容易、更醒目。

缺点：语法比较丑陋 。

结论：

使用 C++ 风格而不要使用 C 风格类型转换。

8.3 const的使用

我们强烈建议你在任何可以使用的情况下都要使用const。

定义：在声明的变量或参数前加上关键字 const 用于指明变量值不可修改，为类中的函数加上 const 限定表明该函数不会修改类成员变量的状态。

优点：人们更容易理解变量是如何使用的，编辑器可以更好地进行类型检测、更好地生成代码。人们对编写正确的代码更加自信，因为他们知道所调用的函数被限定了能或不能修改变量值。

缺点：如果你向一个函数传入 const 变量，函数原型中也必须是 const 的（否则变量需要 const_cast 类型转换），在调用库函数时这尤其是个麻烦。

结论：const 变量、数据成员、函数和参数为编译时类型检测增加了一层保障，更好的尽早发现错误。因此，我们强烈建议在任何可以使用的情况下使用 const：

1) 如果函数不会修改传入的引用或指针类型的参数，这样的参数应该为 const；

2) 尽可能将函数声明为 const，访问函数应该总是 const，其他函数如果不会修改任何数据 成员也应该是 const，不要调用非 const 函数，不要返回对数据成员的非 const 指针或引用；

3) 如果数据成员在对象构造之后不再改变，可将其定义为 const。然而，也不要对 const 过度使用，像 `const int * const * const x`；就有些过了，即便这样写精确描述了x，其实写成 `const int** x` 就可以了。

8.4 整型

C++ 内建整型中，唯一用到的是 int，如果程序中需要不同大小的变量，可以使用<stdint.h>中的精确宽度的整型，如 int16_t。

定义：C++ 没有指定整型的大小，通常人们认为 short 是16位，int 是32位，long 是32 位，long long 是64位。

优点：保持声明统一。

缺点：C++中整型大小因编译器和体系结构的不同而不同。

结论：

<stdint.h> 定义了 int16_t、uint32_t、int64_t 等整型，在需要确定大小的整型时可以 使用它们代替 short、unsigned long long 等，在 C 整型中，只使用 int。适当情况下，推荐使用标准类型如 size_t 和 ptrdiff_t。

最常使用的是，对整数来说，通常不会用到太大，如循环计数等，可以使用普通的 int。你可以认为 int 至少为32位，但不要认为它会多于32位，需要 64位整型的话，可以使用 int64_t 或 uint64_t。

不要使用uint32_t等无符号整型，除非你是在表示一个位组而不是一个数值。即使数值不会为负值也不要使用无符号类型，使用断言来保护数据。

无符号整型：

有些人，包括一些教科书作者，推荐使用无符号类型表示非负数，类型表明了数值取值形式。但是，在C语言中，这一优点被由其导致的 bugs 所淹没。看看：

```
for (unsigned int i = foo.Length()-1; i >= 0; --i) ...
```

上述代码永远不会终止！有时 gcc 会发现该 bug 并报警，但通常不会。类似的 bug 还会出现在比较有符合变量和无符号变量时，主要是 C 的类型提升机制会致使无符号类型的行为出乎你的意料。

因此，使用断言声明变量为非负数，不要使用无符号型。

8.5 预处理宏

使用宏时要谨慎，尽量以内联函数、枚举和常量代替之。

宏意味着你和编译器看到的代码是不同的， 因此可能导致异常行为， 尤其是当宏存在于全局作用域中。

值得庆幸的是， C++ 中，宏不像C中那么必要。宏内联效率关键代码可以用内联函数替代； 宏存储常量可以 const 变量替代； 宏"缩写"长变量名可以引用替代；

使用宏进行条件编译，这个……，最好不要这么做，会令测试更加痛苦（ #define防止头文件重包含当然是个例外）。

下面给出的用法模式可以避免一些使用宏的问题，供使用宏时参考：

1) 可能被多个C++文件用到的宏定义，一般都放在头文件中（.h），如果只需被一个文件所用，放在 .cpp 或 .h 里面都可以；

2) 使用前正确 #define，使用后正确 #undef。

8.6 0和NULL

整数用0，实数用0.0，指针用NULL，字符（串）用'\0'。

8.7 sizeof（ sizeof)

尽可能用 sizeof(varname) 代替 sizeof(type)。

使用 sizeof(varname) 是因为当变量类型改变时代码自动同步，有些情况下 sizeof(type) 或许有意义，还是要尽量避免，如果变量类型改变的话不能同步。

```
Struct data;
memset(&data, 0, sizeof(data)); //Good - 变量类型改变时，代码自动同步
memset(&data, 0, sizeof(Struct)) //Bad - 变量类型改变时，代码不会自动同步
```

8.8 预处理指令

预处理指令不要缩进，从行首开始。即使预处理指令位于缩进代码块中，指令也应从行首开始。

```
void main()
{
    if (condition)
    {
#if DISASTER_PENDING // Good
        dropEverything(); // 预处理指令中的程序仍然正常缩进
#endif
        backToNormal();
    }
}
```

8.9 类格式

声明属性依次序是`public:`、`protected:`、`private:`，都位于行首。除第一个关键词（一般是public）外，其他关键词前空一行：

```
class MyClass : public OtherClass
{
public:
    MyClass();
    ~MyClass() {}

    void someFunction();
    void setSomeVar(int var) { m_someVar = var; }
    int someVar() const { return m_someVar; }

protected:
    bool someInternalFunction();

private:
    int m_someVar;
    int m_someOtherVar;
}
```

8.10 命名空间格式化

命名空间内容不缩进。命名空间不添加额外缩进层次，例如：

```
namespace {
void foo() { // Correct. No extra indentation within namespace.
...
}
```

注意：命名空间的左大括号可以在`namespace`所在一行，之间留一个空格。