**QB/DQAT**

**南京大全自动化股份有限公司企业标准**

QB/DQAT 49.008—2015

C++编码规范(草稿)

2014 - 03 -25 发布 2014 - 04 -01 实施

南京大全自动化科技有限公司 发 布

**目录**

[前 言 3](#_Toc450308750)

[C++编码规范 4](#_Toc450308751)

[1 目的 4](#_Toc450308752)

[2 范围 4](#_Toc450308753)

[3 职责 4](#_Toc450308754)

[4 基本原则 4](#_Toc450308755)

[4.1 规范制定原则 4](#_Toc450308756)

[4.2 执行准则 4](#_Toc450308757)

[5 规范说明 5](#_Toc450308758)

[5.1 文件结构 5](#_Toc450308759)

[5.2 排版规则 7](#_Toc450308760)

[5.3 代码注释 10](#_Toc450308761)

[5.4 命名规则 14](#_Toc450308762)

[5.5 表达式和基本语句 16](#_Toc450308763)

[5.6 函数设计 20](#_Toc450308764)

[5.7 变量和结构 22](#_Toc450308765)

[5.8 宏 23](#_Toc450308766)

[5.9 内存管理 26](#_Toc450308767)

[5.10 命名空间 27](#_Toc450308768)

[5.10 类的设计与继承 27](#_Toc450308769)

[5.11 构造、析构与复制 29](#_Toc450308770)

[5.12 重载与内联 29](#_Toc450308771)

[5.13 类型与安全 31](#_Toc450308772)

[5.14 异常与错误处理 35](#_Toc450308773)

[5.15 程序可读性 37](#_Toc450308774)

[5.16 程序效率 37](#_Toc450308775)

[5.17 质量保证 38](#_Toc450308776)

[5.18 QT 最佳实践 41](#_Toc450308777)

[6 相关指导书及其他文件 42](#_Toc450308778)

# 前 言

为了保证公司软件产品的质量，提高程序的可维护性和可测试性，特制定本规范。 本规范明确了公司软件产品C++编码规范和标准，适用于公司所有使用 C++语言开发的软件产品。本规范中的举例，只是针对某一条规范的具体说明，有可能不符合其他规范，例如变量定义的举例，对变量的定义做了说明，但在其它举例中，并没有使用规定的定义方法，请使用者注意。

本规范由研发部编制。

本规范于2016年01月首次发布。

# C++编码规范

## 1 目的

提高公司 C++程序的质量，培养好的编程风格，增强代码的可读性与可维护性，作为大的软件项目中多人合作开发的基础。

## 2 范围

本规范适用于公司所开发的后台软件、主站系统平台及应用软件的 C++编码。

## 3 职责

a) 编码人员：遵照本编码规范进行 C++代码编写；

b) 测试人员：执行代码静态测试，确认实际代码与编码规范相符。

## 4 基本原则

### 4.1 规范制定原则

a) 方便代码的交流和维护；

b) 只规定需要规定的事情，不强制施加个人喜好或者过时的做法；

c) 不影响编码的效率，不与大众习惯冲突；

d) 使代码更加美观、阅读更为方便；

e) 坚持正确、简单和清晰第一的原则；

f) 使代码的逻辑更清晰、更易于理解。

### 4.2 执行准则

a) 在公司使用 C++编程开发后台软件、主站系统平台及应用软件时，必须遵循本规范；

b) 本规范中标有【规范】字样的内容要求必须遵守；标有【建议】字样的属于建议性内容。

## 5 规范说明

### 5.1 文件结构

5.1.1 源程序文件内容

完整的程序文件由若干部分内容构成，各部分内容及顺序如下：

5.1.1.1 文件头部注释

说明该文件模块的功能和内容（函数、外部数据说明等）。应列出版本号，生成日期，作者，主要函数及其功能，修改日志等。

5.1.1.2 各种定义及类型定义顺序：

“constant” macro

“function” macro

typedefs

enums

5.1.1.3 全局（外部）数据说明顺序：

externs

non-static globals

static globals

如果一组定义仅应用于某一特定的全局数据块，则该定义应紧跟在此数据说明之后，或嵌

入到结构说明之中。

5.1.1.4 函数模块

功能类似的函数应尽量放在一起，每一函数之前应有函数头部注释，主要提供函数的接口说明，内容包括函数基本功能描述、出入口参数、调用关系，必要时也应包括实现算法。函数体中，根据需要可有代码块注释，对某个代码块的功能及临时变量进行说明。

5.1.1.5 头文件内容

头文件中一般允许放下列内容：

a) 宏定义

b) 各种数据结构说明

c) 类型定义说明

d) 外部函数说明

e) 全局变量说明

5.1.2 【规范 1.1】

头文件（\*.h 文件）的开始代码部分，一定要加上 ifndef/define/endif 等预编译判断条件，防止头文件被重复包含。为保证唯一性，建议命名格式为：<PROJECT>\_<PATH>\_<FILE>\_H。

5.1.3 【规范 1.2】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）；用#include“filename.h”格式来引用自定义/非标准库的头文件（编译器将从用户的工作目录或者指定的路径开始搜索）。

5.1.4 【规范 1.3】头文件名不应与标准函数库名相同。

5.1.5 【规范 1.4】头文件中只应包括多个文件都需要的内容。对于功能不同的内容应放在不同的头文件中。

5.1.6 【规范 1.5】禁止在#include中使用绝对路径，尽量采用相对路径。公共头文件的路径应该在编译器的选项中设置。

5.1.7 【规范 1.6】头文件包含语句应放在源程序文件一开始的地方，禁止前面有可执行代码。

5.1.8 【建议 1.1】头文件保存在 include 目录，程序文件保存在 source 或者 src 目录（可以根据需要设置为多级目录），资源文件保存在res目录，执行文件保存在bin 目录，LIB 库文件保存在 lib 目录，如果有配置文件可以保存在config 目录。

5.1.9 【建议 1.2】对于某个程序文件所私有的头文件，没有必要公开声明。为了加强信息隐藏，这些私有的头文件可以存放于定义文件的子目录下，如“private/name\_p.h”。

5.1.10【建议 1.3】工程目录的各级子目录可以根据某种标准细分为各类子目录，每个子目录都应该包含一个 readme 文件，readme 文件应该列举目录中包含的子文件及其主要作用说明。

5.1.11 【建议 1.4】如果有需要，可以增加其他目录如 tmp 等等，但建议基本结构不变。

5.1.12 【建议 1.5】在头文件中只存放声明而不存放定义。

5.1.13 【建议 1.6】小组工程命名编码代号如 XX、D6K 等，统一使用它做代码命名前缀。

5.1.14 【建议 1.7】使用前置声明（forward declarations）尽量减少.h文件中#include的数量。能依赖声明的，就不要依赖定义。

5.1.15 【建议 1.8】头文件包含次序如下：项目内的.h 、C库、C++库、其他库的.h。

### 5.2 排版规则

5.2.1【规范 2.1】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。 相对独立的程序块之间，变量说明之后必须加空行。

5.2.2【规范 2.2】变量和运算符间要留有空隙，用一个空格符隔开，便于阅读。特殊的单目运算符 等除外，如“&”、“.”、“->”、“()”及作为指针运算符使用时的“\*”。函数定义和调用中若出 现多个参数，前一参数后应紧跟逗号运算符，并用一个空格符与后面参数隔开。

示例1：

int i = 0; /\*正确\*/

int i=0; /\*错误\*/

a = b \* c; /\*正确\*/

a=b\*c; /\*错误\*/

if ((a == b) && (a != c)) /\*正确\*/

if((a == b)&&(a != c)) /\*错误\*/

*func*(a, b, c, d); /\*正确\*/

*func*(a,b,c,d); /\*错误\*/

for (i = 0; i < 10; i++) /\*正确\*/

for (i=0;i<10;i++) /\*错误\*/

pPointer->a = pPointer->b \* pPointer->c /\*正确\*/

pPointer -> a = pPointer -> b \* pPointer -> c /\*错误\*/

int \*pX = &i, y; /\*正确,此处y不会被误解为指针\*/

int\* pX = & i, y; /\*错误\*/

5.2.3【规范 2.3】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，并且方便于写注释。

示例2：如下例子不符合规范：

Rect.length = 0; rect.width = 0;

应如下书写：

Rect.length = 0;

Rect.width = 0;

5.2.4【规范 2.4】if、for、while、do 等语句自占一行，除只有一条语句外，执行语句不得紧跟其后，并加{ }。这样可以防止书写失误。但要求 do…while()结构语句的 while 子句与｝同行且紧跟其 后，以区别于 while()结构语句。

示例3：如下例子不符合规范：

if(pUserCR == NULL)return；

do{}

while(pUserCR == NULL)

应如下书写：

if (pUserCR == NULL)

{

return;

}

do

{

} while (pUserCR == NULL);

5.2.5 【规范 2.5】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。

5.2.6 【规范 2.6】{ }之内的代码块在“{”右边数格处左对齐。如果出现嵌套的{ }，则使用缩进对齐。

5.2.7 【规范 2.7】使用 4 字符长度的缩进风格，最好使用 tab 键缩进代码。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。

5.2.8 【规范 2.8】代码行最大长度宜控制在 80 个字符以内。

5.2.9 【规范 2.9】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,拆分出的新行 要进行适当的缩进，使排版整齐，语句可读。

示例4：

If ((very\_longer\_variable1 >= very\_longer\_variable2)

&& (very\_longer\_variable3 <= very\_longer\_variable4)

&& (very\_longer\_variable5 <= very\_longer\_variable6))

{

… … …

}

5.2.10 【规范 2.10】应当将修饰符“\*”和“&”紧靠变量名。

5.2.11 【规范 2.11】关键字之后要留空格。像 const、case 等关键字之后至少要留一个空格，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。

5.2.12 【规范 2.12】函数名之后不要留空格，紧跟左括号“(”，以与关键字区别。

5.2.13 【规范 2.13】“(”向后紧跟，“)”、“,”、“;”向前紧跟，紧跟处不留空格。

5.2.14 【规范 2.14】“,”之后要留空格，如 Function(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)。

5.2.15 【规范 3.15】 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 后应当加空格。

5.2.16 【规范 2.16】一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。

5.2.17 【规范 2.17】像“［］”、“.”、“->”这类操作符前后不加空格。

5.2.18 【规范 2.18】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))。

示例5：

void Func1(int x, int y, int z); /\*良好的风格\*/

void Func1 (int x,int y,int z); /\*不良的风格\*/

if (year >= 2000) /\*良好的风格\*/

if(year>=2000) /\*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\*/

x = a < b ? a : b; /\*良好的风格\*/

x=a<b?a:b; /\*不良的风格\*/

int \*x = &y; /\*良好的风格\*/

int \* x = & y; /\*不良的风格\*/

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();\*/

5.2.19 【建议 2.1】尽可能在定义变量的同时初始化该变量（就近原则）。

5.2.20 【建议 2.2】右花括号“}”后建议加一个注释以便于方便的找到与之相应的“{”。

示例6：

while (1)

{

if (valid)

{

}/\*if valid\*/

else

{

}/\*not valid\*/

}/\*end forever\*/

### 5.3 代码注释

5.3.1 软件开发人员所编制的所有源代码应有良好注释。

5.3.2 【规范 3.1】注释应当准确、易懂，防止注释有二义性，注释率达到 20%以上。

5.3.3 【规范 3.2】对于主要的宏、常量、结构，在定义时需要加注释。

5.3.4 【规范 3.3】全局变量要有详细的注释，包括对其功能、取值范围、存取时注意事项等。

5.3.5 【规范 3.4】重要循环和条件语句一般要写注释，注释放在这段代码之前，同时要保持相同的 层次感。典型的算法要编写注释，说明算法的原理、使用的公式、必要时列出参考书籍。

5.3.6 【规范 3.5】头文件的注释，采用如下格式：

示例7：

/\*! @file filename.h

<PRE>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

模块名 :

文件名 : filename.h

文件实现功能 : 功能概述

作者 : 张三

版本 : V1.00

--------------------------------------------------------------------------------

备注 : <其它说明>

--------------------------------------------------------------------------------

修改记录 :

日 期 版本 修改人 修改内容

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

</PRE>

\* @brief 功能概述

\* @author 张三

\* @version 1.0

\* @date 2014.09.11

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef INC\_FILENAME //预编译，防止头文件被重复引用

#define INC\_FILENAME

#include "rtu\_type.h" //自定义的非标准库本地头文件

#include "xx\_global.h" //自定义的非标准库工程通用头文件

#include <memory> //标准库的头文件

#include <QFont>

#include <QGraphicsItem>

#define PI 3.1415926 //宏定义（如不得已而需要的话）

…

struct StructName //结构体声明（或typedef形式声明）

{

…

}

//FORWARD REFERENCES //引用原型声明等（函数需功能概述）

…

class XX

{

public: //类声明，注意顺序，先函数再数据

protected:

private:

};

…

//INLINE METHODS //内嵌函数

void FunctionName(… …);

…

//EXTERNAL REFERENCES //外部引用说明

…

#endif // INC\_FILENAME

/\*\* @}\*/

5.3.7 【规范 3.6】程序文件的注释，采用如下格式：

示例8：

5.3.8 【规范 3.7】函数的注释，采用如下格式：

示例9：

/\*! \fn int FunctionName(int argl, int arg2)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* \brief FunctionName

\*\* \details 功能描述

\*\* \param argl 输入参数

\*\* \param arg2 输入参数

\*\* \return int 返回参数

\*\* \author 张三

\*\* \date 2011年11月11日

\*\* \note 其他备注

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int FunctionName(int argl, int arg2)

{

… …

}

5.3.9 【规范 3.8】尽量避免在注释中使用缩写，特别是不常用缩写。

5.3.10 【建议 3.1】使用良好一致的注释风格，大段注释使用如下格式。除首末行外每行顶头用“\*” 标注。

示例10：

/\*

\* This is the correct format for a multiline comment

\* in a section of code

\*

\*/

5.3.11 【建议 4.2】使用固定的注释提示。

示例11：

TODO: to be done

BUG: 表明BUG的存在

LIMITATION: 程序的限制

BUGFIX: 修复BUG记录

DEBUG: 表明所加代码是为了寻找BUG

TRICKY: 表明紧随的代码比较敏感，修改要谨慎

WARNING: 警示

COMPILER: 表明是为了避免编译器的BUG

KLUDGE: 表明这段代码写的并不明智

5.3.12 【建议 3.3】在一段庞大的代码体（比如函数）结尾处放置一个识别注释。

示例12：

void foo()

{

…

A very very long function boby

…

}/\*foo\*/

5.3.13 【建议 3.4】在一个文件中宏、结构、变量、函数等声明放在一起，并且在之前加上注释。

示例13：

/\*

\*以下为宏定义

\*/

#define … …/\*注释\*/

/\*

\*以下为结构体定义

\*/

typedef struct/\*注释\*/

{

… …/\*注释\*/

}StructName;

/\*

\*以下为全局变量声明

\*/

… …

### 5.4 命名规则

5.4.1 【规范 4.1】采用大写字母惯例（骆驼式），组成标识符的各个单词以大写字母打头，然后联接起来，同时有一个匈牙利式的前缀。

5.4.2 【规范 4.2】程序中不要出现仅靠大小写区分的相似的标识符。

示例14：

int x, X; /\*变量x与X容易混淆\*/

5.4.3 【规范 4.3】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。

5.4.4 【规范 4.4】标识符应当直观且可以拼读，可望文知意，不要使用意义不明确的缩写。

示例15：

缩写的三种方法：

1. 抽取元音字母

task->tsk

user->usr

1. 取单词意义明显的头部

maximum->max

minimum->min

register->reg

惯例，实际上是先用（2）后用（1）进行处理而得

temporary->tmp

remove->rm

5.4.5 【规范 4.5】标识符最好采用英文单词或其组合，切忌使用汉语拼音命名，命名用词应当准确，例如不要把 CurrentValue 写成 NowValue。

5.4.6 【规范 4.6】命名长度不得超过 30 个字符。

5.4.7 【规范 4.7】用全部的大写字母加下划线来命名预处理的宏。

5.4.8 【规范 4.8】对于超过两个字符长度的缩写，使用 Pasca 或 Camel 大小写。例如，使用 HtmlButton 或 htmlButton；对于仅有两个字符的缩写应大写，如 System.IO，而不是 System.Io。

5.4.9 【规范 4.9】类，联合体，枚举型，非成员对象或具有静态存储期的函数的名称应该是一个唯一的标识符，不能被重用，以免导致错误和混淆。

示例16：

struct stag{int a;};

void stag(void){} //违规

【修复】

struct stag{int a;};

void foo(void){} //OK

5.4.10 【规范 4.10】避免内部或外部变量名字与 C++保留字冲突。

5.4.11 【规范 4.11】局部变量和类、父类、父结构体应该有不同的名字。

5.4.12 【建议 4.1】避免容易被主观解释的难懂的名称，这样的名称会导致多义性。

5.4.13 【建议 4.2】可以在变量名的末尾或开头加合适的限定符（Avg、Sum、Min、Max、Index）。

5.4.14 【建议 4.3】布尔变量名应该包含Is，这意味着 Yes/No 或 True/False 值，例如 fileIsFound。

5.4.15 【建议 4.4】对于仅出现在几个代码行中的生存期很短的变量，也要使用有意义的名称。仅对于短循环索引使用单字母变量名，如 i 或 j。 可能的情况下，尽量不要使用原义数字或原义字符串， 如 For i = 1 To 7，而是使用命名常数，如 For i = 1 To NUM\_DAYS\_IN\_WEEK，便于维护和理解。

5.4.16 【建议 4.5】除非必要，不要用数字或较奇怪的字符来定义标识符。

5.4.17 【建议 4.6】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。

下面是一些在软件中常用的反义词组：

add / delete begin / end create / destroy insert / remove first / last get / release increment / decrement put / get

add / delete lock / unlock open / close min / max old / new start / stop next / previous source / target show / hide send / receive source / destination

cut / paste up / down

示例17：

int nMinSum;

int nMaxSum;

int AddUser(char \*user\_name);

int DeleteUser(char \*user\_name);

5.4.18 【规范 4.6】类型的命名，所有类型命名——类、结极体、类型定义（typedef）、枚举——每个单词以大写字母开头，不包含下划线。

5.4.19 【规范 4.7】变量的命名 ：

全局变量以g\_ 做为前缀。

枚举的命名：枚举值应该全部大写，单词间以下划线相连

普通函数命名采用大小写混合，函数名以大写开头，每个单词以大写开头，使用下划线。

5.4.20 【规范 4.8】文件名要全部小写，可以包含下划线(\_)。

通常，尽量让文件名更加明确，http\_server\_logs.h就比httpserverlogs.h要好，定义类时文件名一般成对出现，如foo\_bar.h和foo\_bar.cc，对应类CFooBar。

### 5.5 表达式和基本语句

5.5.1 【规范 5.1】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，应当用括号确定表达式的操作顺序。

5.5.2 【规范 5.2】不要编写太复杂的复合表达式。

5.5.3 【规范 5.3】在混合精度的表达式中应该使用同一精度的数据类型进行计算。

5.5.4 【规范 5.4】不可将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较。

示例18：

int x=1;

if (x == 0) //写法正确

if (!x) //写法错误

bool b=true;

if (b == 0) //写法错误

if (!b) //写法正确

5.5.5 【规范 5.5】应当将整型变量用“==”或“！=”直接与 0 比较。

5.5.6 【规范 5.6】不可将浮点变量用“==”或“！=”与任何数字比较。

5.5.7 【规范 5.7】应当将指针变量用“==”或“！=”与 NULL 比较。

5.5.8 【规范 5.8】不可在 for 循环体内修改循环变量，防止循环失去控制。

5.5.9 【规范 5.9】每个 case 语句的结尾不要忘了加 break，否则将导致多个分支重叠（除非有意使 多个分支重叠）。Switch 语句要有 Case 配合。

5.5.10【规范 5.10】不要忘记最后那个 default 分支。即使程序真的不需要 default 处理，也应该 保留语句 default : break。

5.5.11 【规范 5.11】循环、分支层次一般不要超过五层。

5.5.12 【规范 5.12】for 循环的增量表达式除了改变循环参数外，不应该执行其他任何动作。

5.5.13 【规范 5.13】避免用等号运算符 (“==”、“!=”) 和 TRUE 宏/枚举(enum)常量做比较。

5.5.14 说明：因为 true 有很多可能值（任何非零值均视为 true），但宏 TRUE 或者枚举类型常量往 往被定义为一个固定值，例如为 1。

5.5.15 【规范 5.14】运算符应只适用于无符号的基本类型操作数。

注：按位运算 (“~”，“<<”，“<<=”，“>>”，“>>=”，“&”，“&=”，“^”，“^=”，“|”，“|=”) 作用于整数或枚举常量通常是没有意义的。另外，如果在一个负数上实施右移，将得到一个未知的执行结果。

5.5.16 【规范 5.15】无条件的 throw 或 break 语句应终止一切非空的 switch 子句。

注：在每一个switch语句应该添加一个break语句，或者如果switch语句是一个复合语句，则在复合语句的最后声 明处添加break语句。此规则可以防止不可预料的程序行为。

5.5.17 【规范 5.16】所有的循环必须有一个固定的上限或下限，防止死循环。

5.5.18 【规范 5.17】逻辑运算符“&&”、“||”和“!”的操作数应该是基本表达式，表达式结果是 有效的布尔值；布尔表达式不应使用除 (“&&”，“||”，“!”，“=”，“==”，“!=”，“?:”) 以外的操作符。

5.5.19 【规范 5.18】普通的 char 和 wchar\_t 类型的表达式不得用于除了赋值运算符“=”、等号运 算符“==”和“!=”、一元运算符“&”以外的内置运算符的操作数。

示例19：

void foo()

{

char ch = ’t’; //OK

if ((ch >= ’a’) && ( ch >= ’z’)) //违规

{

//…

}

}

【修复】

void foo()

{

char ch = 't'; //OK

if (ch == 't') //OK

{

//…

}

}

5.5.20 【规范 5.19】if 语句的条件和循环语句的条件应该是布尔类型。

5.5.21 【规范 5.20】for 循环应该包含一个没有浮动类型的单一的循环计数器。

注：for循环没有准确的循环计数器只是一个while循环。如果这是期望的行为，则 while 循环更为合适。循环计 数器应该被“--”，“++”，“-=n”或“+=n”修改；这里的n在整个循环期间保持不变，此规则有助于确保 确定性循环终止。

示例20：

void foo()

{

int x;

for(x = 0; x < 10; x = x + 1){} //违规

}

【修复】

void foo()

{

int x;

for (x = 0; x < 10; x++){} //OK

}

5.5.22 【规范 5.21】循环计数器以外的循环控制变量不得在判定条件或运算表达式中被修改，在循环体语句中被修改的循环计数器以外的循环控制变量应该是布尔类型的。

注：循环控制变量要么是循环计数器，要么是用于提前终止循环的标识。如果这些在判定条件或表达式中不被修改， 那么代码更容易理解。

5.5.23 【规范 5.22】对指针判断不能使用类似于 if(test)或 if(!test) 的逻辑表达式。

示例21：

void foo()

{

int \*ptr;

if (ptr){} //违规

if (!ptr){} //违规

}

【修复】

void foo()

{

int \*ptr;

if (ptr == NULL){} //OK

}

5.5.24 【规范 5.23】禁止逻辑判别的表达式不是逻辑表达式，此规则可以区分整型和逻辑值。

5.5.25 【规范 5.24】switch 语句的表达式禁止是逻辑表达式，包括等号运算符(“==”，“!=”)、 逻辑运算符(“!”，“&&”，“||”)、关系运算符(“<”，“>”，“<=”，“>=”)。

5.5.26 【规范 5.25】循环体必须用大括号括起来。

5.5.27 【规范 5.26】逻辑表达式必须使用圆括号连接（只包含逻辑与、逻辑或的表达式除外）。

5.5.28 【规范 5.27】禁止条件判别成立时相应分支无可执行语句。

5.5.29 【规范 5.28】禁止在逻辑表达式中使用赋值操作符；禁止对布尔表达式的操作数使用赋值运 算符，如果需要对布尔表达式的操作数赋值，则必须单独在操作数外面赋值，这样做可避免混淆“=” 和“==”。

5.5.30 【规范 5.29】赋值运算符必须对所有成员进行赋值，包括基类型中的成员。

5.5.31 【建议 5.1】避免使用 register 关键字声明变量。

5.5.32 【建议 5.2】尽量使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串。

### 5.6 函数设计

5.6.1 【规范 6.1】函数的规模尽量控制在 200 行之内（不包括空行和注释）。

5.6.2 【规范 6.2】一个函数只完成一项功能。

5.6.3 【规范 6.3】如果参数是指针，且仅作输入用，则应在类型前加 const，以防止该指针在函数体 内被意外修改。

5.6.4 【规范 6.4】不要省略返回值的类型。如果没有返回值，应该声明为 void 类型。

5.6.5 【规范 6.5】检查通过其它途径进入函数的变量的有效性，如全局变量、句柄等。

5.6.6 【规范 6.6】函数参数尽量使用 C++的 const 机制保证数据安全性；尽量不要使用类型和数目不确定的参数，如 C 语言的 printf 函数是采用不确定类型和参数数目的典型代表，提倡使用 C++的 I/O 机制，因其是类型安全的。

5.6.7 【规范 6.7】在 C++中，指针的使用应慎重，如果的确只需要借用一下对象的别名，提倡使用 C++的引用机制，不使用指针。

5.6.8 【规范 6.8】不要引用未经赋值的指针，因为这样常常会引起系统的崩溃。

5.6.9 【规范 6.9】return 语句不可返回指向“栈内存”的指针或引用，因为该内存在函数体结束时被自动销毁。

5.6.10 【规范 6.10】一个函数中不能声明另外一个函数。

5.6.11 【规范 6.11】标识符 main 除了全局的 main 函数以外不得用于其它函数。

5.6.12 【规范 6.12】函数不得使用省略符号定义，不要使用有可变数量参数的函数，这条规则可以 防止很多潜在的问题。

5.6.13 【规范 6.13】main 必须定义为 int main(void) 或 int main(int,char\*[]) 的形式。此规则有助于错误的确定和参数的输入。

5.6.14 【规范 6.14】禁止将参数指针赋值给过程指针。以防导致不可预料的结果。

5.6.15 【规范 6.15】禁止将过程声明为指针类型。不要使用非常量指针指向函数。

5.6.16 【规范 6.16】避免以非调用方式使用函数。在表达式中使用一个过程标识符有可能无法执行函数的调用或函数地址的分配。

5.6.17 【规范 6.17】禁止 void 类型的函数 return 语句带有返回值。对于返回值类型为 void 的函数， 返回语句不能有表达式。

5.6.18 【规范 6.18】有返回值的函数中 return 必须带有返回值。缺了这个返回值，将会导致未知的行为。

5.6.19 【规范 6.19】不要将引用类型的参数作为函数的返回值返回。引用类型的参数可能是一个临时对象，当函数结束的时候，它将被销毁。

5.6.20 【规范 6.20】当数组作为参数传入一个函数时不要声明数组的大小。

5.6.21 【建议 6.1】尽量避免函数的参数过多，参数个数最好不超过 5 个。如果参数太多，使用时容易将相同数据类型的参数顺序用错。

5.6.22 【建议 6.2】用于出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。

5.6.23 【建议 6.3】不要将正常值和错误标识混在一起返回。正常值用输出参数获得，而错误标识用 return 语句返回。

5.6.24 【建议 6.4】使用灵活的、动态分配的数据，不要使用固定大小的数组。

5.6.25 【建议 6.5】在多任务操作系统的环境下编程，要注意函数可重入性。

5.6.26 【建议 6.6】定义函数时，参数顺序为：输入参数在前，输出参数在后。

### 5.7 变量和结构

5.7.1 【规范 7.1】合理设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。

5.7.2 【规范 7.2】常量用全大写字母、下划线分割单词的方式表达。

示例22：

常量命名

#define MAX\_LENGTH 100 (不推荐)

const int MAX\_WIDTH = 1024 (推荐)

5.7.3 【规范 7.3】如果引用所指向的对象地址能够改变，不要初始化该引用。若引用指向的对象能被释放，则该引用能通过一个指针删除，结果导致该引用指向 NULL，防止非法访问已经被释放的变量。

示例23：

void foo()

{

int \*ptr = 0;

int &rptr = \*ptr; //违规

}

【修复】

void foo()

{

int ptr = 0;

int &rptr = ptr; //OK

}

5.7.4 【规范 7.4】避免在文件中使用命名空间外的全局变量、全局函数和类。

5.7.5 【规范 7.5】项目中不得包含未使用的变量。

5.7.6 【规范 7.6】声明在内部范围的标识符不得掩盖声明在外部范围的标识符。

5.7.7 【规范 7.7】禁止枚举类型中的元素名与已有的变量名同名，可以避免代码死循环。

5.7.8 【规范 7.8】禁止结构体声明不完整。

5.7.9 【规范 7.9】循环变量必须是局部声明的，应该定义在最小的范围内。

5.7.10 【规范 7.10】将函数的局部变量尽可能置于最小作用域内，在声明变量时将其初始化，应使用初始化代替声明+赋值的方式。如果局部变量是对象，则不要在循环体内定义，如：

// 低效的实现

for (int i = 0; i < 1000000; ++i)

{

Foo f; // 构造函数和析构函数分别调用 1000000 次！

f.DoSomething(i);

}

5.7.11 【建议 7.1】使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。使用标准的数据类型，有利于程序的移植。

5.7.12 【建议 7.2】尽量减少没必要的数据类型默认转换与强制转换。

### 5.8 宏

5.8.1 【规范 8.1】用宏定义表达式时，要使用完备的括号。

示例24：

如下定义的宏都存在风险

#define RECTANGLE\_AREA(a, b) a \* b

#define RECTANGLE\_AREA(a, b) (a \* b)

#define RECTANGLE\_AREA(a, b) (a) \* (b)

正确的定义应为：

#define RECTANGLE\_AREA(a, b) ((a) \* (b))

5.8.2 【规范 8.2】将宏所定义的多条表达式放在大括号中。

示例25：

下面的语句只有宏的第一条表达式被执行。为了说明问题，for语句的书写稍不符规范。

#define INTI\_RECT\_VALUE(a,b)

a=0;

b=0;

for (index = 0; index < RECT\_TOTAL\_NUM; index++)

INTI\_RECT\_VALUE(rect.a, rect.b);

正确的用法应为：

#define INTI\_RECT\_VALUE(a,b)

{

a=0;

b=0;

}

for (index = 0; index < RECT\_TOTAL\_NUM; index++)

{

INTI\_RECT\_VALUE(rect[index].a, rect[index].b);

}

5.8.3 【规范 8.3】使用宏时，不允许参数发生变化。

示例26：

#define SQUARE(a) ((a)\*(a))

int a = 5;

int b;

b = SQUARE(a++); /\*结果： a=7,即执行了两次增1。\*/

正确的用法是：

b = SQUARE(a);

a++; /\*结果：a=6,即只执行了一次增1。\*/

5.8.4 【规范 8.4】不得在#if 或#endif 预处理中使用未定义的宏标识符，除了作为 defined 运算符的操作数。

示例27：

#if X // X在这里还没有被定义

#endif

【修复】

#define X

#if X //OK

#endif

5.8.5 【规范 8.5】defined 预处理运算符只能使用两个标准形式：defined ( identifier )和 defined

identifier。

示例28：

#if defined X > Y //违规

#endif

#define DEFINED defined

#if DEFINED(X) //违规

#endif

【修复】

#if defined X //OK

#endif

#if define(X) //OK

#endif

5.8.6 【规范 8.6】不得覆盖标准库宏和对象的名称。

5.8.7 【规范 8.7】不要在宏定义中使用指针解引用。

示例29：

#define MAC1(a, b) \*a + b //违规

#define MAC2(a, b) (a + \*b) //违规

5.8.8 【规范 8.8】不能重定义原始类型。

示例30：

#define T30 char //违规

#define T31 unsigned char //违规

【修复】

#define T30 char\_def //OK

#define T31 unsigned\_char\_def //OK

5.8.9 【规范 8.9】"#pragma once" 预处理指令不能使用在源代码文件中。

### 5.9 内存管理

5.9.1 【规范 9.1】只引用属于自己的存储空间，防止引用已经释放的内存空间。过程/函数中分配的内存，在过程/函数退出之前要释放。

5.9.2 【规范 9.2】指针变量一定要初始化。

5.9.3 【规范 9.3】 分配内存前，先判断指针是否非空，如果为非空则应先释放指针，然后将指针置 为 NULL。

5.9.4 【规范 9.4】避免数组或指针的下标越界，特别要当心发生“多 1”或“少 1”操作。

5.9.5 【规范 9.5】任何时候使用指针时，均先判断它是否为空，为空则应相应处理。

5.9.6 【规范 9.6】由于函数参数的评估顺序尚未定义，禁止在函数参数表中进行资源分配。

5.9.7 【规范 9.7】禁止对含有构造函数的对象调用 malloc/realloc，确保所有的成员变量都经过初始化。

5.9.8 【规范 9.8】禁止返回函数范围内的 new 运算符初始化的本地指针。

5.9.9 【规范 9.9】为动态分配内存的类定义赋值运算符、声明复制构造函数。

5.9.10 【规范 9.10】在析构函数中对指针成员调用 delete，避免发生内存泄漏。

5.9.11 【规范 9.11】在调用 fopen 函数的附近提供文件打开错误的错误处理机制，预防在出现 IO 错误的情况下访问文件（FILE）指针导致程序崩溃。

5.9.12 【规范 9.12】在析构函数中调用 fclose()函数来关闭用于打开文件的指针成员，预防资源泄漏。

5.9.13 【规范 9.13】避免传递自动变量的地址给主调函数。

5.9.14 【规范 9.14】不要在一条语句中分配多个资源。

5.9.15 【规范 9.15】不要使用 memcpy 或 memcmp 函数来拷贝或比较结构化的对象。

5.9.16 【规范 9.16】 如果一个类定义了 new/new[] 或 delete/delete[] 操作符的任何重载，就应 该提供 plain, in-place, 和 non-throwing 所有三种标准形式的 new/new[] 或 delete/delete[] 操 作符重载。此避免隐藏操作符 new 的其他重载。

5.9.17 【规范 9.17】释放的内存在任何情况下都不能访问。析构函数不能人工调用。

5.9.18 【规范 9.18】检查 new 的返回值，当内存没分配，预防出现空指针。

5.9.19 【规范 9.19】拷贝和析构必须一致，避免产生内存管理问题。

5.9.20 【规范 9.20】不要在指针类型上使 sizeof 来指定由“malloc”，“calloc”或“realloc” 函数分配的内存大小，避免不正确的内存分配。

5.9.21 【建议 9.1】尽量避免在大循环内部使用 new、delete，这种操作会降低执行效率。

### 5.10 命名空间

5.10.1.1 【规范 10.1】在头文件中使用不具名的空间容易违背C++的唯一定义原则（One Definition Rule（ODR））。在.h文件中避免使用不具名命名空间。

5.10.1.2 【规范 10.2】使用using 命名空间需谨慎，为防止污染命名空间，建议只在源文件及头文件的函数、方法或者类中可以使用using 命名空间。

### 5.10 类的设计与继承

5.10.1.1 【规范 10.1】用组合代替继承。

5.10.1.2 【规范 10.2】避免从并非要设计成基类的类中继承。

5.10.1.3 【规范 10.3】要避免提供隐式转换。

5.10.1.4 【规范 10.4】将数据成员设为私有的，无行为的聚集（C 语言形式的 struct）除外。

5.10.1.5 【规范 10.5】总是一起提供 new 和 delete。

5.10.1.6 【规范 10.6】如果提供类专门的 new，应该提供所有标准形式。

5.10.1.7 【规范 10.7】类不能多次继承其它类，除非是虚拟继承。

5.10.1.8 【规范 10.8】对未抽象接口的类进行多重继承。本规则仅允许一个直接非接口的父类。

5.10.1.9 【规范 10.9】禁止改变虚拟函数的默认参数，防止对默认传递的值的错误解释。

5.10.1.10 【规范 10.10】所有用作基类并且含有虚拟函数的类必须定义相应的虚拟析构函数。

5.10.1.11 【规范 10.11】指向抽象类的指针不能被强制转换为指向继承于该类的子类的指针。

5.10.1.12 【规范 10.12】在同一层次结构中，基类不能同时具有虚拟及非虚拟的属性。

5.10.1.13 【规范 10.13】不要从多于一个基类中派生出相同名字的函数。

5.10.1.14 【规范 10.14】禁止把指向派生类对象的指针强制转化为指向虚基类对象的指针。

5.10.1.15 【规范 10.15】指向虚拟基类的指针只能通过 dynamic\_cast 的方式转换为指向其衍生类的指针。

5.10.1.16 【规范 10.16】类与结构的选择时，当只有数据时使用struct。类与结构体的成员变量使用不同的命名规则。

5.10.1.17 【规范 10.17】在类中使用特定的声明次序：public在private之前，成员函数在数据成员（变量）前。

定义次序如下：public:、protected:、private:,如果那一块没有，直接忽略即可。在每一块中，声明次序如下：

|  |  |  |
| --- | --- | --- |
| 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 |  |
|  |  |  |

### 5.11 构造、析构与复制

5.11.1 【规范 11.1】以同样的顺序定义和初始化成员变量。

5.11.2 【规范 11.2】在构造函数中用初始化代替赋值。

5.11.3 【规范 11.3】避免在构造函数和析构函数中调用虚拟函数。

5.11.4 【规范 11.4】将基类析构函数设为公用且虚拟的，或者保护且非虚拟的。

5.11.5 【规范 11.5】使用赋值的标准形式。

5.11.6 【规范 11.6】构造函数允许使用 explicit 进行显示转换。

注：带参数的构造函数能够通过编译器执行显示转换。这样的访问即便在最好的情况下也可能导致创建和销毁临时 对象，影响运行，同时最糟糕的情况是无意义的代码被编译。在后一种情况下，可能在编译时发现的问题都将被推到运行时。

5.11.7 【规范 11.7】类的析构函数不得带有异常退出，可能会导致内存泄漏和不当的对象析构。

5.11.8 【规范 11.8】至少声明一个构造函数以防止编译器提供一个公共的构造函数。

注：遵从该规范，将使类进行显示的初始化，从而预防编译器默认的初始化导致不当，尤其是指针成员。此规则增 加代码的可读性。

5.11.9 【规范 11.9】禁止在构造函数、析构函数、成员函数中直接访问全局变量。

5.11.10 【规范 11.10】复制构造函数应复制所有的数据成员以及基类型。

### 5.12 重载与内联

5.12.1 【规范 13.1】运算符既可以定义为全局函数，也可以定义为成员函数，应按照以下规则。

示例31：

|  |  |
| --- | --- |
| 运算符 | 规则 |
| 所有的一元运算符 | 建议重载为成员函数 |
| = () [] -> | 只能重载为成员函数 |
| += -= /= \*= &= |= ~= %= >>= <<= | 建议重载为成员函数 |
| 所有其它运算符 | 建议重载为全局函数 |

5.12.2 【规范 12.2】不要改变 C++内部数据类型（如 int,float 等）的运算符。

5.12.3 【规范 12.3】不能重载“.”，因为它在类中对任何成员都有意义，已经成为标准用法。

5.12.4 【规范 12.4】不能重载目前 C++运算符集合中没有的符号，如“#”，“@”，“$”等。原因 有两点，一是难以理解，二是难以确定优先级。

5.12.5 【规范 12.5】对已经存在的运算符进行重载时，不能改变优先级规则，否则将引起混乱。

5.12.6 【规范 12.6】如果程序同时重载了“+=”和“+”，则应该使“a += b”和“a = a + b”完 成同样的工作。其他类似的操作符重载也一样。

5.12.7 【规范 12.7】关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用。

示例32：

inline void Foo(int x, int y); //inline 仅与函数声明放在一起

void Foo(int x, int y)

{

… …

}

而如下风格的函数Foo则成为内联函数：

void Foo(int x, int y);

inline void Foo(int x, int y); //inline与函数定义体放在一起

{

… …

}

5.12.8 【规范 12.8】慎用内联，在以下情况下不宜使用内联。

a.如果函数体内的代码比较长，使用内联将导致内存消耗代价较高，不宜使用内联。

b.如果函数体内出现循环或者switch语句，那么执行函数体内代码的时间要比函数调用的开销大，也不宜使用内联。

5.12.9 【规范 12.9】避免在指针和数字类型上使用重载。

5.12.10 【规范 12.10】不能定义带有局部静态数据的内联非成员函数。

注：调用此类函数可能会在代码中创建多个局部静态对象。

5.12.11 【规范 12.11】避免重载“&&”、“||”或者“,”。

注：编译器会对内建的“&&”、“||”或者“,”特殊处理，如果用户重载它们，他们会变成具有特殊语义的普通 功能，这种情况下肯定会引入隐蔽的缺陷和脆弱性的方式。避免重载这些操作符将避免改变编译器读取表达式 语义的方式，防止造成不可预知的程序行为。

5.12.12 【规范 12.12】在源文件中不要定义内联函数

注：内联函数应该只在头文件中被定义，或者是“.i”文件。静态内联函数以及内联函数定义在匿名namespace中 是允许的。

5.12.13 【规范 12.13】一元“&”运算符不得被重载。

注：在完整类型包含用户声明的操作符“&”时，取不完整类型对象的地址将导致不确定的行为。

### 5.13 类型与安全

5.13.0 【规范 13.0】使用C++风格的类型转换，不要使用C风格类型转换。

5.13.1 【规范 13.1】避免使用 reinterpret\_cast。

5.13.2 【规范 13.2】避免对指针使用 static\_cast。

5.13.3 【规范 13.3】避免强制转换 const，强制转换 const 有时会导致未定义的行为，即使合法，也 是不良编程风格的主要表现。

5.13.4 【规范 13.4】不要对非 POD 进行 memcpy 操作或者 memcmp 操作。

5.13.5 【规范 13.5】不要使用可变长参数（…）。

5.13.6 【规范 13.6】不要多态地处理数组，数组的可调整性很差，多态地处理数组是绝对的类型错 误，而且编译器可能不会做出任何提示。

5.13.7 【规范 13.7】不要释放空指针。

5.13.8 【规范 13.8】提供正确的字符串终止符，不要将终止符‘\0’写成字符串“\0”。

5.13.9 【规范 13.9】如果类中含有非静态指针并且没有拷贝构造函数，应使用引用传递类对象。

5.13.10 【规范 13.10】操作符不能将引用作为返回值，应该返回对象，以防止发生内存泄漏。

注：操作符包括：a&b, a^b, a|b, ~a, a!=b, a<=b, a<b, a==b, a>=b, a>b, a!b, a&&b, a||b,+a, -a, a%b, a\*b, a+b, a-b, a/b。

5.13.11 【规范 13.11】构造函数的定义中不能包含特定的缺省参数，以免与隐式声明的拷贝构造函 数混淆。

5.13.12 【规范 13.12】不要在 a(b)，a[b] 和类型转换语句内部使用赋值或递增、递减表达式。

5.13.13 【规范 13.13】不允许在三元运算符中使用混合类型。

5.13.14 【规范 13.14】赋值操作符两边的操作数类型要匹配。

5.13.15 【规范 13.15】不能将两个整数相除之后的结果赋给浮点类型，此规则可以避免精度损失。

5.13.16 【规范 13.16】memcpy/strncpy/memmove 的第三个参数不应该依赖于第二个参数。

5.13.17 【规范 13.17】printf 函数中格式化字符串的数量与对应的参数个数必须相同，并且标签要 与对应的参数类型匹配。

5.13.18 【规范 13.18】不要定义包含成员函数的结构体。

注：因为类支持多实例和封装，所以成员函数应该被包含在类中，而不是在结构体中。默认的针对一个类的成员和 基类的访问级别为私有的，但结构体则经常声明为公共类型。

示例33：

struct A

{

public:

int foo(); //违规

};

【修复】

class A

{

public:

int foo(); //OK

}

5.13.19 【规范 13.19】当两个操作符作用相反的时候（例如“==”和“!=”），两个都应该定义。

注：当类的对象通过相应操作符与另一个类进行运算时，需要实现其针对该类相对应的操作符。该规范可提高代码 可读性和代码可维护性，并防止误解重载操作符的含义。

示例34：

class C

{

public:

bool operator <= (C&c); //违规

};

【修复】

class C

{

public:

bool operator <= (C&c); //OK

bool operator >= (C&c); //OK

};

5.13.20 【规范 13.20】转换操作符 operator->、operator()、operator[]应该是 const 类型的。

5.13.21 【规范 13.21】从不是重载运算符函数得到的非 void 类型的返回值应该始终被使用。

注：在C++中调用一个函数而不使用其返回值，可能是个错误。

5.13.22 【规范 13.22】NULL 不得用作一个整数值，字面量“0”不得用作空指针常量。

示例35：

#include <cstddef>

void f1(int);

void foo()

{

f1(NULL); //违规

}

【修复】

void f1(int);

void foo()

{

f1(0); //OK

}

5.13.23 【规范 13.23】const 成员函数不得返回指向类数据的非 const 指针或引用。

注：当一个对象声明为const类的类型时，只有const成员函数可以调用该对象。const成员函数的期望是对象的状 态不会在调用函数的时候被修改。然而，从一个const函数返回一个指向类数据的非const指针或引用，则会允 许修改对象的状态。

5.13.24 【规范 13.24】避免指针用于算数运算，防止类型的混淆。

5.13.25 【规范 13.25】禁止将 const 转换成非 const。

注：编译器会把常量数据放置到ROM或是在RAM页中写保护，将const转换成非const会引起内存错误。

5.13.26 【规范 13.26】禁止使用不合适的循环变量类型，清楚 for 语句的三个表达式的用途。

5.13.27 【规范 13.27】禁止对指针变量使用强制类型转换赋值，整数类型不能强制转换成指针，允 许整型常量“0”强制转换成指针类型。

5.13.28 【规范 13.28】结构体变量初始化的类型必须一致，提高代码的可读性和可维护性。

5.13.29 【规范 13.29】禁止指针的指针超过两级。

注1：此规则禁止显式的指针值的计算，任何显式计算过的指针值都有无意识访问非法内存地址的可能性。指针可 能指向超出数组或者结构体的范围，或者甚至指向一个任意的地址，此规则用来提高代码的安全性。

5.13.30 【规范 13.30】禁止位操作符带有布尔型的操作数。这种情况通常是用错了操作符，例如把 “&&”误写为“&”或把“||”误写成“|”。

5.13.31 【规范 13.31】禁止给无符号变量赋负值。对无符号整型或者无符号长整型表达式进行一元 减法运算是没有意义的。

5.13.32 【规范 13.32】有符号类型的位长度必须大于等于两位。

5.13.33 【规范 13.33】强制类型转换不能把一个指向函数的指针转换成其他指针类型，包括一个函 数类型的指针。此规则可以阻止不确定的行为。

### 5.14 异常与错误处理

5.14.1 【规范 14.1】广泛地使用断言记录内部假设和不变式。

5.14.2 【规范 14.2】建立合理地错误处理策略，并严格遵守，优先使用异常报告错误。

5.14.3 【规范 14.3】设计和编写错误安全代码。

5.14.4 【规范 14.4】通过值（而非指针）抛出异常，通过引用（通常是 const 引用）捕获异常。

5.14.5 【规范 14.5】正确地报告、处理和转换错误。

5.14.6 【规范 14.6】throw 语句的赋值表达式不得自身引起一个异常抛出。

注：如果当构造异常对象，或者当初始化此异常对象的赋值表达式求值的时候抛出一个异常，那么这个异常优先于 将要抛出的异常。

5.14.7 【规范 14.7】应该只能在程序启动之后和终止之前抛出异常。

注：在程序开始执行main函数体之前的启动阶段，构造和初始化静态对象；在main函数返回之后的终止阶段，静态 对象被销毁。如果在这两个阶段抛出一个异常，那么它会导致程序按照一个设定执行的方式终止。

5.14.8 【规范 14.8】异常对象不应该有指针类型。

注：如果抛出指针类型的异常对象，并且该指针指向了一个动态创建的对象，那么可能哪一个函数负责销毁它的以 及在什么时候销毁都是未知的。

5.14.9 【规范 14.9】不得使用 goto 或 switch 开关语句将控制传递到 try 或 catch 模块中。

注：使用goto或switch开关语句将控制传递到try或catch块中的程序是不规范的，并非所有的编译器都会报错。

5.14.10 【规范 14.10】针对代码中每个显式抛出的异常，应在可能导致该异常的所有调用路径上存 在处理程序。

5.14.11 【规范 1411】类(class)类型的异常应该始终按照引用方式捕获。

注：如果一个类类型的异常按照值方式捕获，会发生切断（派生关系）。也就是说，如果是一个派生类的异常对象 并且作为基类捕获，那么只有此基类的函数（包括虚函数）可以被调用。此外，任何在派生类中的其他成员数 据不能被访问。

5.14.12 【规范 14.12】构造函数或析构函数中，执行 try 功能模块的处理程序不得引用来自此类或 其基类的非静态成员。

注2：在类的构造函数或析构函数中的 try 功能模块的处理程序中（例如 catch 部分）访问一个类或基类的非静态 成员是不确定的。例如，如果在创建对象的过程中抛出一个内存分配异常，那么当处理程序试图去访问其成 员时该对象不存在。相反，在析构函数中，该对象可能在异常被处理之前已经成功销毁，因此不会对处理程 序可用。相比之下，静态成员的生命期大于对象本身的生命期，所以静态成员能够保证在处理程序访问它时 存在。

5.14.13 【规范 14.13】如果单独的 try-catch 语句或 try 功能模块中提供多个处理程序给派生类和 部分基类，这些处理程序应该遵循从最底层的派生类到基类的顺序。

注：如果基类的处理程序在派生类的处理程序之前被找到，那么就会使用基类的处理程序，派生类的处理程序是不可达的代码并且不会被执行。

5.14.14 【规范 14.14】异常对象不得有指针类型。如果抛出指针类型的异常对象，并且该指针指向 了一个动态创建的对象，那么可能会不清楚哪一个函数是负责销毁它的，同时也不清楚在什么时候。该 规则防止内存管理问题。

### 5.15 程序可读性

5.15.1.1 【规范 15.1】避免使用不易理解的数字，用有意义的标识来代替。涉及物理状态或含有物 理意义的常量，不应直接使用数字，必须使用有意义的枚举或宏来代替。

5.15.1.2 【规范 15.2】字面量后缀应使用大写字母。

5.15.1.3 【建议 15.1】源程序中关系较为紧密的代码应尽可能相邻，便于程序阅读和查找。

### 5.16 程序效率

5.16.1 【规范 16.1】尽可能声明局部化的变量，以改善代码的可读性，减少不必要的内存分配。

5.16.2 【规范 16.2】尽量使用“a @= b”形式，而不使用“a = a @ b”，其中“@”可以是“+”, “-”,“\*”,“/”,“%”,“&”,“|”,“^”,“<<”,“>>”，以提高代码可读性和效率。

5.16.3 【规范 16.3】使用引用而非数值传递对象，以提高代码效率。

5.16.4 【规范 16.4】含有递归、循环或者虚函数调用的函数不应定义成内联函数。有利于减少代码 占用，提高运行速度。

5.16.5 【规范 16.5】避免使用内联构造函数和析构函数。

5.16.6 【规范 16.6】每个定义的函数应至少被调用过一次。

5.16.7 【规范 16.7】保持一个类型和它的非成员函数接口在同一个命名空间，改善代码的效率和可读性。

5.16.8 【规范 16.8】在函数的执行体中不应定义类、结构体、联合体(class/struct/union)。

5.16.9 说明：如果在函数中定义一个结构体或类，那它相当于一个局部类型，会导致不能被编译器重复使用并且会导致同类型定义的不相容。

5.16.10 【建议 16.1】仔细分析有关算法，并进行优化。

5.16.11 【建议 16.2】对模块中的函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构，提高程序效率。

5.16.12 【建议 16.3】对于迭代器和其他模板对象使用前缀形式（++i）的自增、自减运算符。后置的自增自减需要对表达式的值i进行一次拷贝，如果i是迭代器或其他非数值类型，拷贝的代价比较大。

### 5.17 质量保证

5.17.1 【规范 17.1】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。

示例36：

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];

}

}

5.17.2 【规范 17.2】少用、慎用 goto 语句，而不是禁用。

5.17.3 【规范 17.3】命名中若使用特殊约定或缩写，应有注释说明。应在源文件的开始之处，对文件中所使用的缩写或约定，特别是特殊的缩写，进行必要的注释说明。

5.17.4 【规范 17.4】在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。

5.17.5 【规范 17.5】除了编译开关/头文件等特殊应用，应避免使用 \_EXAMPLE\_TEST\_ 之类以下划线开始和结尾的定义。

5.17.6 【规范 17.6】系统运行之初，要初始化有关变量及运行环境，防止未经初始化的变量被引用。

5.17.7 【规范 17.7】编程时，要防止差 1 错误。

注：此类错误一般是由于把“<=”误写成“<”或“>=”误写成“>”等造成的，由此引起的后果，很多情况下是很 严重的，所以编程时，一定要在这些地方小心。当编完程序后，应对这些操作符进行彻底检查。

5.17.8 【规范 17.8】要时刻注意易混淆的操作符。当编完程序后，应从头至尾检查一遍这些操作符，以防止拼写错误。

注：形式相近的操作符最容易引起误用，如C中的“=”与“==”、“|”与“||”、“&”与“&&”等，若拼写错了， 编译器不一定能够检查出来。

5.17.9 【规范 17.9】有可能的话，if 语句尽量加上 else 分支，对没有 else 分支的语句要小心对待； switch 语句必须有 default 分支。

5.17.10 【规范 17.10】避免在表达式中用赋值语句。

5.17.11 【规范 17.11】避免对浮点类型做等于或不等于判断。

5.17.12 【规范 17.12】尽量避免强制类型转换。

5.17.13 【规范 17.13】如果不得不做类型转换，尽量用显式方式。

5.17.14 【规范 17.14】 不能在结构体中使用硬编码的数值作为偏移量。在 32 位环境下，有 4 字节 的偏移量 (4\*8 比特=32 比特)，但是当环境转换成为 64 位时偏移量将会增长到 8 字节(8\*8\*比特=64 比特)。

5.17.15 【规范 17.15】算法中不应假设由访问标识符分隔的非静态数据成员的分配顺序。这条规则 是为了防止应用程序被假设关于被标志符分隔的非静态成员的顺序是有序的。被标志符分隔的非静态成员的顺序不确定。

5.17.16 【规范 17.16】避免使用标准库中 string 类中的 data()函数，可以使用 c\_str()代替。

注：String::data()返回一个没有终止符的字符串，使用不当可能会导致缓冲区溢出；而 c\_str()函数可以返回一个包含终止符的字符串。

5.17.17 【规范 17.17】避免使用 vfork()函数，可使用 fork()替代。

注：vfork()和fork()都用于创建一个新进程，通过vfork()创建的子进程会阻塞父进程，可能导致死锁，也可能在子进程中修改全局数据，导致父进程不能继续。

5.17.18 【规范 17.18】不要使用 mbstowcs()函数，防止缓冲区溢出。

5.17.19 【规范 17.19】不要使用返回当前目录或窗口目录的函数，如 GetTempPath。

注：首先，如果临时目录的路径大于分配给存储这些信息的缓冲区，可能存在缓冲区溢出。其次， 可能返回一个不安全路径。此外，并不能保证返回的路径是有效的或可用的。

5.17.20 【规范 17.20】不要在源代码中使用 setuid，应使用 seteuid。

5.17.21 【规范 17.21】不能在程序代码中使用 getpw 和 cuserid 函数，使用 getpwuid 代替。

注：函数getpw是不安全的，可能会产生溢出问题。

5.17.22 【规范 17.22】避免使用已经废弃的 C 函数 usleep，使用 sleep 或 nanosleep 函数代替。

5.17.23 【规范 17.23】不要使用用户自定义的转换函数。

注：使用用户自定义的转换函数即便在最好的情况下也可能导致创建和销毁临时对象，影响运行，更有可能导致无 意义的代码被编译。

5.17.24 【规范 17.24】对变量进行移位运算必须保证不会产生溢出。移位运算符的右边操作数应当 介于零和一个小于左边操作数宽度位的之间（包含) 。

5.17.25 【建议 17.1】除非为了满足特殊需求，避免使用嵌入式汇编。

5.17.26 说明：程序中嵌入式汇编，一般都对可移植性有较大的影响。

5.17.27 【建议 17.2】精心地构造、划分子模块，并按“接口”部分及“内核”部分合理地组织子模 块，以提高“内核”部分的可移植性和可重用性。

注：对不同产品中的某个功能相同的模块，若能做到其内核部分完全或基本一致，那么无论对产品的测试、维护， 还是对以后产品的升级都会有很大帮助。

5.17.28 【建议 17.3】对较关键的算法最好使用其它算法来确认。

5.17.29 【建议 17.4】时刻注意表达式是否会上溢、下溢。

示例37：

如下程序将造成变量下溢。

unsigned char size;

while (size-- >= 0) /\*将出现下溢\*/

{

…/\*program code\*/

}

当size等于0时，再减1不会小于0，而是0xFF,故程序是一个死循环。应如下修改。

char size;/\*从unsigned char 改为char\*/

while (size-- >= 0)

{…/\*program code\*/}

5.17.30 【建议 17.5】使用变量时要注意其边界值的情况。

### 5.18 QT 最佳实践

5.18.1 【规范 18.1】每一个 QObject 子类只能包含一个 Q\_OBJECT 宏。

注：每一个QObject子类只能包含一个Q\_OBJECT宏，否则Qt元类型系统不会被使用。

5.18.2 【规范 18.2】SIGNAL 和 SLOT 宏里的方法必须存在。

注：SIGNAL和SLOT宏里的方法必须存在(参数必须匹配)并且这些方法需要标以Q\_SLOTS或Q\_SIGNALS。QT信号和槽机 制是非常强大的，因为其提供在signal和slots之间的运行时的动态连接。这支持灵活脚本和动态用户接口开 发，但同时也带来相应的代价：在编译阶段，当signal和slot连接无效时，编译器不会提示你。在运行时，你 会得到一个警告，但这太迟了。该规则在代码分析阶段强制检测signal和slot连接的有效性。

5.18.3 【规范 18.3】不能将 QThreads 的优先级设置为 Idle 或者 Critical，只能设置成 High/Low/Med。

注：Idle和Critica优先级会引起死锁。某些操作系统将不允许设置线程的优先级，这样的结果是平台特定的。

示例38：

#include<QtGui/QtGui>

void MyFunction(QThread\* qt1, QThread\* qt2)

{

qt1->start(QThread::IdlePriority); //违规

qt2->setPriority(QThread::IdlePriority); //违规

}

【修复】

#include<QtGui/QtGui>

void MyFunction(QThread\* qt1, QThread\* qt2)

{

qt1->start(QThread::NormalPriority); //OK

qt2->setPriority(QThread::InheritPriority); //OK

}

5.18.4 【规范 18.4】不要通过访问 QThread::terminate()强迫终止线程。

注：该函数危险，不鼓励使用它。线程是否立即终止依赖于操作系统的调度策略。线程可能在代码执行的任何路径 上终止也可能在修改数据时终止。线程没有任何机会在其之后清除，解锁互斥体等。

5.18.5 【规范 18.5】如果在某函数中使用互斥锁，应由同一个函数进行释放。

5.18.6 【规范 18.6】针对 QMutexLocker 不要使用 new。

注：QMutexLocker是唯一方便的用于锁和解锁互斥体的类。将其放在栈上，代码中的异常将释放可能发生错误的互 斥体。

5.18.7 【规范 18.7】不要直接设置某个在 Qt 内核中设置 Widget 属性。

注：Qt使用widget属性改变信号状态。修改由Qt内核设置的属性可能导致不同的影响，最坏的情况可能导致应用程 序崩溃。

## 6 相关指导书及其他文件

——《高质量C++编程指南》；

——《C&C++编程规范》；

D6K2编码规范

1. 变量

1.1 命名

类成员变量：int32 m\_nCount; string m\_strName; CTest\* m\_pTest;

结构成员变量：int32 Count; string Name; Point \*pTest;

局部变量：int32 nCount; string strName; CTest\* pTest;

全局变量：int32 g\_nCount; string g\_strName; CTest\* g\_pTest;

1.2 初始化

任何变量在使用前必须初始化

1.3 使用

指针在使用前，必须判断是否为NULL

1. 函数

2.1 命名

int32 ThisIsAFunction(int32 nCount,string strName,CTest\* pTest);

函数命名采用大小写字母结合的形式，但专有名词不受限制。

2.2 代码格式

函数体明确分三部分，初始化区，核心逻辑区，返回值区

2.3 复杂度

函数主体不允许前后出现多个循环

函数代码长度不得超过200行

函数过于复杂则拆分为多个函数

1. 文件

3.1 头文件

定义：一个头文件只允许定义一个外部使用的类，该类内部使用的结构或类可在同一头文件定义

包含：在.h文件中，尽可能少的包含其他头文件，尽可能使用类的前置声明。顺序为：系统，库，自定义

3.3 实现文件

包含：尽可能在实现文件中包含所需的头文件。顺序为：系统，库，自定义

格式：实现文件中的函数顺序，与头文件保持一致

3.4 文件名

文件名一般用单词命名，全小写，具有自描述性；多个词组中间用“\_”隔开。

1. 类与结构

4.1 命名

类：class CTest，结构 struct TEST

4.2 代码格式

类头文件定义分类区，辅助区和多个功能区；

类区：存放构造和析构函数，以及类级成员变量声明(如：类的父指针)和常量定义，类的初始化和清理接口和类的核心功能函数

辅助区：存放类的辅助功能函数

功能区：类的子功能按区存放（子功能的成员函数，成员变量和常量定义）

4.3 复杂度

类的子功能过多或过于复杂，则需拆分子类实现

1. 命名空间

最好不要在全局使用using指示符，以保证命名空间下的所有名称都可以正常使用。

// 禁止 —— 污染名字空间

using namespace std;

在 .cpp 文件, .h 文件的**函数**, **方法**或**类**中, 可以使用 *using* 关键字.

// 允许: .cpp 文件中

// .h 文件的话, 必须在函数, 方法或类的内部使用

using ::foo::bar;

在 .cpp 文件, .h 文件的函数, 方法或类中, 允许使用名字空间别名.

// 允许: .cpp 文件中

// .h 文件的话, 必须在函数, 方法或类的内部使用

namespace fbz = ::foo::bar::baz;

1. 注释

标准注释

1. 工程

SOLUTION视图中，根据各代码文件的用途，建立相应的目录存放

1. 系统

8．1 通用系统变量命名

模拟量:AIN

开关量:DIN

遥调:AOUT

遥控:DOUT

8．2 通用数据变量类型

UINT32 uint32

INT32 int32

参见datatypes.h