目 录

第1章 概述

1.1 目的

执行该规范的目的在于保证编码的规范性，确保代码质量，以及提高软件代码的可读性、可修改性，避免在开发和调试过程中出现不必要的麻烦。此外，还可以统一代码风格，便于研发人员之间的技术交流。

1.2 适用项目

本编程规范适用于本公司所有采用C++作为编程语言的软件项目。

1.3 阅读对象

本规范的阅读对象主要为：

(1) 项目经理

(2) 软件开发人员

(3) 软件测试人员

1.4 格式说明

规则的格式如下：

【规则<编号>】<分类> <规则描述>

对括号里的内容介绍如下：

(1) <编号> 每一个规则都会有一个唯一的编号，由两部分组成：章节号.序号，如 2.1.3 表示 2.1 节中的第 3 条规则。

(2) <分类> 格式为：强制/推荐，强制性规则是要求公司程序员都需要认真遵守的编码约定；推荐为一些重要性不高的，或很难在所有项目中适用的规则，程序员可选择性采用。

(3) <规则描述> 描述的是规则的具体内容。

比如，【规则 7.7.4】（强制） 在开发与安全性相关的软件时，不准使用动态变量和动态对象，不允许使用动态堆内存分配（dynamic heap memory allocation）的方式。

1.5 内容简介

本规范共有 7 个章节，这些章节可以分为两部分：

(1) 程序版式、命名约定等与代码风格相关的主观性内容。这一部分内容主要介绍了代码的格式化约定和统一的命名注释风格。

(2) 头文件的使用、类的设计等与代码键壮性、容错性相关的内容。这一部分的内容旨在提高代码的健壮性与容错性，使代码适用于与安全相关的软件项目。

第2章 程序版式

2.1 长行拆分

【规则 2.1.1】（强制） 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。

【说明】 文字长度不超过 10 个单词最利于阅读；行代码超过 10 个单词或 110 个字符可认为是长表达式。

【举例】

类型	良好格式

运算符前置表达式换行对齐	if ((very_longer_variable1 >= very_longer_variable2)
	  && (very_longer_variable3 >= very_longer_variable4)
	  && (very_longer_variable5 >= very_longer_variable6))
	{
	  DoSomething();
	}

分句换行对齐	for (very_longer_initialization;
	  very_longer_condition;
	  very_longer_updata)
	{
	  DoSomething();
	}

函数参数换行对齐	bool retval = DoSomethingThatRequiresALongFunctonName(
	        very_long_argument1,
	        argument2,
	        argument3);

2.2 缩进和对齐

【规则 2.2.1】（强制） 用缩进体现代码结构，只使用空格，不使用 Tab。缩进的空格数不作要求，但在同一文件应该一致。

【举例】

良好格式	不良格式

void Foo(int32_t x)
{
…//program code
}
	void Foo(int32_t x) {
	…//program code
	}

//如果出现嵌套的{ }，则使用缩进对齐。
{
…
{
…
}
…
}
	{
	{
	…
	}
	}

2.3 空格的使用

【规则 2.3.1】（强制） 关键字之后要留空格，否则无法辨析关键字。像“if” 、“for” 、“while” 、“catch”等关键之后应留一个空格再跟左括号“(” ，以突出关键字。

【举例】

良好格式	不良格式

if (…) //关键字后加空格
{
…
}
 	
catch (…) //关键字后加空格
{
…
}
	if(…) //关键字后无空格
	{
	…
	}
	
	catch(…) //关键字后无空格
	{
	…
	}

【规则 2.3.2】（强制） 逗号“，” 之后要留空格，如 Foo(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 2.3.3】（强制） 函数名后不要留空格，紧跟左括号“(”，以与关建字区别。

【举例】

良好格式	不良格式

Foo(…) //自定义函数名后无空格
{
…
}
	Foo (…) //函数名后有空格
	{
	…
	}

【规则 2.3.4】（推荐） 一元操作符如 “!” 、“~” 、“++” 、“—” 、“&”（地址运算符）等与操作数之间不留空格；像“[]” 、句点“.”或箭头“->”这类操作符前后不加空格。

【规则 2.3.5】（推荐） 赋值操作符、比较操作符、算术操作符、逻辑操作符、位操作符，如“=” 、“+=” 、 “>=” 、“<=” 、“+” 、“*” 、“%” 、“&&” 、“|” 、“<<” 、 “^” 等二元操作符的前后应当加空格。

【例外】 如对于表达式比较长的 for、do、while、switch 语句和 if 语句，为了紧凑起见可以适当地去掉一些空格。

```
for (i=0; i<10; i++)          //例外情况一

if ((a<=b) && (c<=d))         //例外情况二
```

【说明】 上面有些规则被列为推荐就是告诉大家这些规则并不是强制性的要死守，可以变通应用，目的在于便于阅读，提高代码的可读性。

【举例】

```
if (condition1 && condition2 && condition3)
        a = (x + y) * z;
```

2.4 空行的使用

【规则 2.4.1】（强制）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 2.4.2】（推荐）在一个函数体内，逻辑上密切相关的语句之间不加空行，其它地方应加空行分隔。

2.5 指针和引用表达式

【规则 2.5.1】（强制） 在访问指针和引用表达式时，句点“.”或箭头“->”前后不要有空格；指针、地址操作符后（“*” 、“&” ）后不要有空格。

【举例】

良好格式	不良格式

x = *p;
p = &x;
x = r.y;
x = r->y;
	x = * p;
	p = & x;
	x = r .y;
	x = r -> y;

【规则 2.5.2】（强制） 在声明指针或参数时，为便于理解，可以将修饰符 “*”和 “&” 紧靠数据类型；也可以将“*”或“&”紧靠变量名，但不准在“*”和“&”两边都留空格。

【举例】

良好格式	可接受格式	不良格式

//修饰符紧靠数据类型
char_t* name;
int* x;
int32_t y;//为避免 y 被误解
//为指针，这里必须分行写。
int32_t* Foo(void* p);
const char_t& str;
	char_t *name;
	const char_t &str;
		//*前后都有空格
		char_t * c;
		
		//&前后都有空格
		const char_t & str;

2.6 函数声明与定义

【规则 2.6.1】（强制）对函数参数排序时，将所有输入参数置于输出参数之前。既是输入也是输出的参数放在输入参数和输出参数之间。

【说明】 C++函数参数分为输入参数和输出参数，有时输入参数也会输出。输入参数一般传值或常数引用（const references），输出参数一般为非常数指针（non-const pointers）。

【举例】

```
int32_t Foo(InType& in, InOutType* in_out, OutType* out);
```

【规则 2.6.2】（推荐） 返回类型和函数名占据同一行；一般情况下，建议将参数放在同一行；如果一行不能放下所有参数，可以分行并对齐以提高可读性。

【注意】

(1) 返回值总是和函数名在同一行；

(2) 左圆括号“(”总是和函数名在同一行，中间没有空格；

(3) 圆括号“(” 、“)”与参数间没有空格；

(4) 大括号“{” 、“}”总是单独占一行；

(5) 函数声明和实现处的所有形参名称必须保持一致；

(6) 所有形参应尽可能对齐；

(7) 在如果函数为 const 的，关键字 const 应与最后一个参数位于同一行。

【举例】

类型	良好格式

参数不多时	ReturnType ClassName::FunctionName(Type par_name)
	{
	  DoSomething();
	}

如果一行文本较多，容不下所有参数。	ReturnType ClassName::LongFunctionName(Type par1,
	  Type par2,
	  Type par3)
	{
	  DoSomething();
	}

甚至连一个参数也放不下。	ReturnType ClassName::ReallyLongFunctionName(
	  Type par_name1,
	  Type par_name2,

	Type par_name3)
	{
	DoSomething();
	}

函数为 const 的，关键字 const 应与最后一个参数位于同一行。	ReturnType ClassName::ReallyLongFunctionName(
	Type par_name1,
	Type par_name2) const
	{
	…
	}

2.7 函数返回值

【规则 2.7.1】（强制） *return* 表达式中不要使用圆括号。

【举例】

良好格式	不良格式
return x;	return(x);

2.8 if 条件语句

【规则 2.8.1】（强制）*if(condition)* 语句后应该紧跟大括号“{}”的复合语句。

【说明】 条件语句如果不加大括号，程序员对代码行进行修改或增加时容易出错。

【举例】

良好格式	不良格式

if (condition)
{
x = 0;
}
	if (condition)
	x = 0; //要加大括号
	
	if (condition); //错误,要有语句

【规则 2.8.2】（强制） 所有的 *if…else if* 条件语句应该以 *else* 语句终止。

【举例】

良好格式	不良格式
if (condition) {	if (condition) {

DoThis();
}
else if (x < 5)
{
DoThat();
}
else
{
 	
}//以 else 语句结束
	DoThis();
	}
	else if (x < 5)
	{
	DoThat();
	}//不允许，没有 else 语句结束。

【规则 2.8.3】（推荐） 不在圆括号和条件表达式之间加空格，关键字 *else* 另起一行。

【说明】 对基本条件语句有两种可以接受的格式，一种是在圆括号和条件之间有空格，另一种是没有空格。后一种没有空格的格式较为更为常见。但需要注意的是，在同一代码文件中应该采用统一的格式。

【举例】

良好格式	可接受格式	不良格式

//condition 与括号之间
//没有空格。
if (condition)
{
…
}
else //else 另起一行
{
…
}
	//condition 与括号之
	//间都有空格。
	if (condition)
	{
	…
	}
		//一边留空格，一边没
		//留空格
		if (condition)
		{
		…
		}

2.9 switch 开关选择语句

【规则 2.9.1】（强制） *switch* 语句中总是包含一个 *default* 语句。

【说明】 强制程序员加 default 语句是为了让程序员一定要考虑到特殊情况，以增加代码的键壮性。注意 case 块中的语句不要丢了 break。

【举例】

良好格式	不良格式

switch (var)
{
case 0:
case 1:
{
…
break;
}
case 2:
{
…
break;
}
default: //default 是必须要加的
{
break; //注意不要丢了 break
}
}
	switch (var)
	{
	case 0:
	{
	…
	break;
	}
	case 1:
	{
	…
	break;
	}
	//没有 default
	}

2.10 循环语句

【规则 2.10.1】（强制）空循环体应使用“{}”或 *continue*，而不是一个简单的分号。

【举例】

良好格式	不良格式

//好的格式，“)”和“{”之间留
//一个空格，“{}”表示空循环体
for (int32_t i= 0; i<k; ++i)
{
}
	//不好的格式，分号容易被忽视。
	for (int32_t i=0; i<k; ++i);

//好的格式，continue 表明没有执
//行语句。
while (condition)
{
  continue;
}
	//不好的格式，看起来像 do/while
	while (condition)；

2.11 变量初始化

【规则 2.11.1】(强制)变量初始化时，基本类型必须使用“=”，而不使用“()”。

【举例】

良好格式	不良格式
int32_t x = 3;	int32_t x(3); //看起来像声明函数

2.12 预处理指令

【规则 2.12.1】（强制） 预处理指令不要缩进，从行首开始，即使位于缩进代码块中，预处理指令也应从行首开始。

【举例】

良好格式	不良格式

//#if 指令从行首开始
if (…)
{
#if DISASTER_PENDING
  DropEverything();
#endif
  BackToNormal();
}
	//预处理不应缩进
	if (…)
	{
	  #if DISASTER_PENDING
	  DropEverything();
	  #endif
	  BackToNormal();
	}

2.13 类格式

类是 C++中最重要也是使用频率最高的新特性之一。类的版式好坏将极大地影响代码质量。

类的版式主要有两种方式：

(1) 将 private 类型的数据写在前面，而将 public 类型的函数写在后面。采用这种版式的程序员主张类的设计“以数据为中心”，重点关注类的内部结构。

(2) 将 public 类型函数写在前面，而将 private 类型的数据写在后面，采用这种版式的程序员主张类的设计“以行为为中心”，重点关注的是类应该提供什么样的接口和服务。

【规则 2.13.1】（强制） 采用“以行为为中心”的书写方式，即首先考虑类应该提供什么样的函数，所以应当将公有的定义和成员放在类声明的最前面，保护的放在中间，而私有的放在最后。

【说明】 这样做不仅让自己在设计类时思路清晰，而且方便别人阅读。

【规则 2.13.2】（强制） 访问说明符（*public*、 *protected*、*private*）应该独占一行，并与类声明中的 *class* 关键字左对齐。

【规则 2.13.3】（强制） 基类直接跟在类名称之后，子类与父类尽量保持同行（有多个父类时，一行写不下可分行并对齐），访问说明符（*public*、*protected*、*private*）不可省略。

【规则 2.13.4】（强制）在类中使用特定声明次序，次序如下 *public*、*protected*、*private*。

【说明】 每一块中，声明次序一般如下：

(1) typedef 和 enum；

(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。

【规则 2.13.5】（推荐）.cpp 文件中函数的定义应和声明次序一致。

【说明】

(1) 基类名尽量与子类名放在同一行；

(2) 访问说明符独占一行，不缩进且与 class 对齐；

(3) 这些访问说明符后不空行；

(4) public 放在最前面，然后是 protected 和 private；

(5) 对于比较复杂的类，如果有多组功能不同的成员、分组注释，分组注释也要与 class 关键字对齐。

【举例】类成员的声明版式：

```
class CXXXpublic OtherClass
{
public:
//////////////////////////////类型定义
    typedef vector<string> VSTR;

public:
/////////////// /////////////构造、析构、初始化
    CXXX();
    ~CXXX();

public:
////////////////////// ////公用方法

// [[ 功能组 1
    void Function1(void) const;
    long Function2(IN int32n);
// ]] 功能组 1

// [[ 功能组 2
    void Function3(void) const;
    bool Function4(OUT int& n);
// ]] 功能组 2

private:
////////////////////// 私有属性
    int32_t private_field;
private:
////////////////////// 禁用的方法
    CXXX& operator=( const CXXX& rhs);
};
```

2.14 构造函数的初始化列表

【规则 2.14.1】（强制） 应当尽可能通过构造函数的初始化列表来初始化成员和基类。

【说明】

```
class ClassA
{
public:
   ClassA()
   {
        stringX=“hello”;
        stringY=“world”;
   }
private:
   SringType stringX,stringY;
};
```

构造函数会在系统内部生成初始化代码，如：

```
ClassA: stringX(), stringY()
{
        stringX=“hello”;
        stringY=“world”;
}
```

也就是说，并非显式初始化的对象将使用其默认的构造函数自动初始化。在初始化列表中显式初始化成员变量，代码表达意图更明确，代码通常会更小、更快。

【举例】

```
//良好的格式
ClassA: stringX(“hello”), stringY(“world”){}
```

【规则 2.14.2】（强制） 构造函数初始化列表放在同一行，或分行排列并注意缩进和对齐。

【举例】

```
// 初始化列表放一行。
// 说明：以开头的变量是类成员变量，命名约定中会介绍到。
MyClass::MyClass(int32_t var): varA(var), varB(var+1)
{
   …
}
```

```
MyClass::MyClass(int32_t var)
      : varA(var),          // 初始化列表多行，对齐。
        varB(var+1)
{
    …
}
```

【规则 2.14.3】（强制） 初始化列表的书写顺序应当与对象的构造顺序一致，即先按照声明顺序写基类初始化，再按照声明顺序写成员初始化。

【说明】 如果一个成员 "a" 需要使用另一个成员 "b" 来初始化，则 "b" 必须在 "a" 之前声明，否则将会产生运行时错误（有些编译器会给出警告）。

【举例】

```
class ClassX : public ClassBaseA, public ClassBaseB
{
   ...
   ClassY iA;   //定义时，iA 在 iB 之前，
   ClassZ iB;   //所以初始时 iA 必须在 iB 之前声明。
};

ClassX::ClassX(int32_t nA, int32_t nB, bool bC)
      : ClassBaseA(nA), ClassBaseB(nB), iA(bC), iB(iA)
          //初始化须序是先基类，后成员，分别按照声明顺序书写。
{
      ...
};
```

第3章 命名约定

3.1 命名的整体原则

在整个项目中一套定义良好、使用统一的命名规范将大大提升源代码的可读性和软件的可维护性。通过统一的命名风格可以直接确定命名实体是类型、变量、函数、常量、宏等，而无需查找实体声明。

【规则 3.1.1】（推荐）在编写一个子模块或派生类的时候，要遵循其整体模块或基类的命名风格，保持命名风格在整个模块中的同一性。

【规则 3.1.2】（推荐）标识符采用英文单词或其组合，应当直观且可以拼读，可望文知意，用词应当准确。

【规则 3.1.3】（推荐）在保持一个标识符意思明确的同时，应当尽量缩短其长度。

【举例】

```
HttpServerLogs.h      //相比 logs.h，表达的意思会更明确。
```

【规则 3.1.4】（推荐）不要出现仅靠大小写区分的相似的标识符，避免过于相似。

【举例】

```
例如：“i”与“I”；“fun”与“Fun”；“o”与“0”；“I”和“l” ；“l”和“1”；“S”和“5”；“n”和“h”；“B”和“8”；“z”和“2”等等。
再如：int32 idio, idi0;
      int32 idin, idlh;
```

【规则 3.1.5】（推荐）程序中不要出现名字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，但容易使人误解。

【规则 3.1.6】（推荐）用正确的反义词组命名具有互斥意义的标识符。

【举例】

```
"nMinValue" 和 "nMaxValue"；
"GetName()" 和 "SetName()" ；
...
```

【规则 3.1.7】（推荐）尽量避免名字中出现数字编号，除非逻辑上的确需要编号。

【举例】

```
//程序员为了方便或者节约时间而导致产生无意义的名字。
int32_t     Value1;
int32_t     Value2;
```

3.2 文件命名

【规则 3.2.1】（强制） 头文件和实现文件分别用.h 和.cpp 作为文件后缀名。

【规则 3.2.2】（强制） 文件名采用英文单词组合，英文字母不能全部大写，不能采用汉语拼音。

【举例】

良好格式	不良格式
//可以接受的文件名 trainClass.cpp TrainClass.cpp	//不可接受的文件名 TRAINCLASS.cpp huoche.cpp //火车的拼音

【规则 3.2.3】（强制） 不要使用已存在于系统 *include* 文件夹下的文件名，如 math.h。

【规则 3.2.4】（强制） 定义类时，文件名一般成对出现，如 TrainClass.h 和 TrainClass.cpp，对应类 TrainClass。

3.3 命名空间

C++命名空间是“类”概念的一种退化，它的引入为标识符名称提供了更好的层次结构，使标识符看起来更加直观简洁，同时大大降低了名字冲突的可能性。

【规则 3.3.1】（强制）*namespace* 以模块名命名，并且全部大写，如 AUTORUN。

【规则 3.3.2】（推荐）*namespace* 的命名基于项目名称和目录结构，如 PROJECT_AUTORUN。

3.4 类的命名

【规则 3.4.1】（强制） 类型命名由单词组合而成，其中每个单词以大写字母开头，不包含下划线，如 MyDateTimeClass、MyWeekClass。

【规则 3.4.2】（强制） 所有类型命名（类、结构体、枚举、*typedef*）使用相同约定。

【例外】 重新定义一个基本类型时，可以用小写字母作为类型名的首字母，如：

```
typedef    char    char_t;
```

【说明】因为它们本质上都属于类型，使用相同的命名规则更便于程序员记忆。

【举例】

```
// classes and structs
class UrlTable
{
   …
};

class UrlTableTester
{
   …
};

struct UrlTableProperties
{
   …
};

// enums
enum MyWeek
{
   ….
};
```

类中的成员命名见 3.9 节变量的命名。

【规则 3.4.3】（强制） 接口是一种特殊的类，接口的命名和类相似，只不过要加上单词 *Interface* 为后缀，以便于区别。

【说明】 C++虽然没有单独给出定义接口的机制，但可以**采取变通的方法创建接口**，详见 6.4 节——接口。

【举例】

ReadFileInterface, CalculateAreaInterface 等。

3.5 结构体的命名

结构体的命名和类的命名相同，见 3.4 节类的命名。

类型为结构体的变量和普通变量的命名方式一样，见 3.9 节变量的命名。

结构体的数据成员的命名和普通变量一样，详见 3.9 节变量的命名。

3.6 枚举命名

【规则 3.6.1】（强制）*enum* 结构中的成员命名全部大写。

【说明】 枚举结构中的成员和宏定义的常量相似，所以它们的命名也设置为相同的格式。

【举例】

```
enum Weekdays{MONDAY, TUESDAY, …};
```

枚举名称和类名的命名规则相同见 3.4 节类的命名。

3.7 函数命名

【规则 3.7.1】（推荐） 函数的名字采用英文大小写混合且第一个字母大写，没有下划线。

【举例】

良好格式	不良格式

AddFunction();
DeleteFunction();
	//不要大小写混合又加下划线
	Delete_Function();

【规则 3.7.2】（强制）存取函数成对出现，要与存取的变量名匹配。

【举例】

```
class Student
{
public:
    int32_t    GetRadius();//get 与 set 成对出现且函数名与变量名有关联
    void       SetRadius(int32_t radius);
private:
    int32_t    radius;
};
```

【规则 3.7.3】（推荐） 函数名最好采用动宾形式或动词形式。

【举例】

```
GetName();
SearchPath();
```

3.8 常量命名

【规则 3.8.1】（强制） 常量全部大写，单词间以下划线分隔。所有编译时常量，无论是局部的、全局的还是类中的，全部大写以与其他变量保持一点区别。

【举例】

```
const    int32_t    MAX_LENGTH    =    365;
```

3.9 变量命名

【规则 3.9.1】（强制） 变量命名使用 camelCase 命名规则，即第一个单词小写，从第二个单词开始首字母大写。

【举例】

```
struct    MyStudent    studentOfCollege;
Node    nodeInLink;
```

【规则 3.9.2】（强制） 类的成员变量命名也采用 camelCase 命名规则，如果函数中的变量与类的成员变量重名，则一定要在类的成员变量前加关键字 *this*（如this->member)以和其它变量相区分。

【举例】

```
void Object::SetValue(int32_t width, int32_t    height)
{
    this->width = width;
    this->height = height;
}
```

第4章 注释风格

4.1 注释的整体原则

注释虽然写起来很痛苦，但对保证代码可读性至为重要，下面的规则描述了应该注释什么，注释在哪里。请记住注释的确很重要，最好的文档就是代码本身，类型和变量命名意义明确要比通过注释解释模糊的命名好得多。

【规则 4.1.1】（强制） 注释使用“//”或“/**/”都可以，相同项目中注释风格确保统一就行。对于一块代码且注释较长时用“/**/”形式，对于单行代码的注释用“//”。

【举例】

```
void Function()
{
    int32_t flag; //作为指针是否指到堆栈底部的标志
    ……
}
```

```
/**
 * @brief  介绍函数的功能…
 */
void Function()
{
    int32_t flag;
    ……
}
```

【规则 4.1.2】（强制） 对文件、类的头部和函数头部进行注释时统一采用 Doxygen 工具能识别的格式。

Doxygen 是一个程序文档生成工具，可将程序中的特定的批注转换成为说明文件。如果你以前没有接触和使用过 Doxygen 也没关系，该文档中会简单介绍它所能识别的注释格式，不要求所有的注释都符合它的规则，**只要求在文件、类和函数的头部**进行注释时遵守它的规则。详见 6.2、6.3、6.4 节。

Doxygen 可处理的注释格式

(1) 对一段代码进行注释：

```
/**
 *…这是对下面代码的注释…
 */
```

(2) 对单行代码进行注释：

```
///…这是对该行代码的注释…（3 条斜杠才是 Doxygen）
```

在注释中加一些 Doxygen 支持的指令，主要作用是控制输出文档的排版格式，使用这些指令时需要在前面加上“\”或者“@”符号，常用指令有：

常用指令	含义
@file	档案的批注说明。
@author	作者的信息
@brief	用于 class 或 function 的简易说明，例： @brief 本函数负责打印错误信息串。
@param	主要用于函数说明中，后面接参数的名字，然后再接关于该参数的说明。
@return	描述该函数的返回值情况，例： @return 本函数返回执行结果，若成功则返回 TRUE，否则返回 FLASE。
@retval	描述返回值类型，例： @retval NULL 空字符串。 @retval !NULL 非空字符串。
@note	注解
@brief	注释文字
@exception	可能产生的异常描述，例： @exception 本函数执行可能会产生超出范围的异常。
@date	日期
@version	软件版本

【规则 4.1.3】（强制） 当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。

【举例】

```
while(flag)
```

```
{
        for(i=1;i<100;i++)
        {
            ……
            for(j=1;j<1000;j++)
            {
                  ……
            }//<应注释说明>
        }//<应注释说明>
}//<应注释说明>
```

【规则 4.1.4】（强制） 不得使用“//”或“/* */”将一段程序代码注释掉。

【举例】

```
//不正确
/ * int32_t i
    i = InitValue();
  */
// for (int32_t i = 0; i < 10; i++);
```

【规则 4.1.5】（强制） 注释的位置应与被描述的代码相邻；对于单行代码的注释可以放在该代码的上方或右方，不可放在下方。

【规则 4.1.6】（推荐）注释是对代码的“提示” ，而不是文档。程序中的注释不可喧宾夺主，注释太多会让人眼花缭乱。

【规则 4.1.7】（推荐）如果代码本来就清楚的，则不必加注释。否则多些一举，令人厌烦。

【规则 4.1.8】（推荐）边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。

【规则 4.1.9】（推荐）注释应当准确、易懂、防止注释有二义性。错误的注释不但无益反而有害。

【规则 4.1.10】（推荐）尽量避免在注释中使用不常用的缩写。

【规则 4.1.11】（推荐）代码中注释模块的数量与语句数量之比应大于 0.2。

【说明】 代码中注释所占的比例是代码可读性的一种体现，因此提高注释所占的比例有助于提高代码的可读性。

【举例】

```
/* 1 block */
/* 1 block */

/* OK - function 'comm' contains 20 statements
                and 10 blocks of comments - COMF = 0.5
                2 block */
int32_t comm1(int32_t s) /* 3 block */
{
    int32_t i;    /* 4 block */
    if (s > 0)
        i = 3;
    else
        i = 4;
    if (s > 0)
        i = 3;
    else
        i = 4;
    if (s > 0)
        i = 3;      /* 5 block */
                    /* 6 block */
    else            /* 7 block */
        i = 4;
    if (s > 0)
     /* 8 block */
     /* 8 block */
     /* 8 block */ i = 3;
    else
        i = 4;
    if (s > 0)
        i = 3;
    /* 9 block */
    else
        i = 4;
    if (s > 0){
        i = 3;
        i = 5;
    }
    else
        i = 4;
    return i;
```

```
}/* 10 block */
```

4.2 文件头注释

【规则 4.2.1】（推荐） 在每一个文件开头加入版权公告，然后是文件内容的描述。

（1）版权公告等信息

每一个文件包含以下项，依次是：

1） 体现公司的版权声明，如：Copyright (C), 2008-2011, Zhejiang Insigma Group Co.,Ltd. ；

2） 作者，列出创建该文件的作者，或对该文件有主要贡献的作者，可以列出多位。

（2）文件内容

每一个文件的版权和作者信息之后，都要对文件内容进行注释说明，说明顺序如下：

1） 文件名；

2） 文件内容的简单说明，通常 .h 文件要对所声明的类的功能和用法作简单说明，.cpp 文件包含了更多的算法讨论或实现细节；

3） 历史记录，包含日期、作者和变更记录等信息，如 2011-02-17 张三：新建文件。

【举例】

```
/**
 * @copyright 2008-2011, Zhejiang Insigma Group Co.,Ltd.
 * @author      <xxx>, [yyy], [zzz] ...（作者和逗号分割的修改者列表）
 */

/**
 * @file     文件名
 * @brief      <描述文件的功能等相关信息>
 *
 *  版本历史：
 * @date 2011-02-17     author1：新建文件
 *          2011-03-01     author2：<描述本次对文件进行修改的相关记
```

```
 *    录……>
 *          2011-04-01     author3：<描述本次对文件进行修改的相关记
 *    录……>
 *
 */
```

4.3 类注释

【规则 4.3.1】（推荐） 每个类的定义要附着描述类的功能和用法的注释。

【说明】 注释放在类声明的上方；如果类有任何同步前提，在注释里说明；如果该类的实例可被多线程访问，使用时务必加以文档说明。参考格式如下：

```
/**
 * @brief          类的简介
 * @exception      属于该类的异常类（如果有的话）
 * @note           使用该类时需要注意的问题（如果有的话）
 * @autor          [作者 1], [作者 2], ...
 */
class CXXX
{
   ...
};
```

对于功能明显的简单类（接口小于 10 个），也可以使用简单的注释头：

```
/**
 * @brief  简单介绍类
 */
class CXXX
{
   ...
};
```

4.4 函数注释

【规则 4.4.1】（推荐） 函数声明处注释描述函数功能，函数定义处描述函数实现。

（1）函数声明处

注释在函数声明之前，描述函数功能及用法，注释使用描述式而非指令式，注释只是为了描述函数而不是告诉函数做什么。如采用“实现了开门操作”而不

用“打开车门” 。通常函数声明处不描述函数如何实现，那是函数定义部分的事情。函数声明处注释的内容列出如下：

1） 输入和输出；

2） 如果函数分配了空间，需要由调用者释放；

3） 参数是否可以为 NULL；

4） 是否存在函数使用的性能隐忧；

5） 其他一些函数使用需注意的事项。

（2）函数定义处

每个函数定义时要以注释说明函数功能和实现要点，如实现的简要步骤、实现的理由，为什么前半部分要加锁而后半部分不需要等。函数注释的参考格式：

```
class MyClass
{
public:
/**
 * @brief     功能 ：  <函数实现功能>
 * @param [in]   输入参数 1 :   参数说明
 * @param [out]  输出参数 2 :   参数说明
 ...
 * @return          描述该函数的返回值情况
 * @exception     可能抛出的异常及其说明（如果有的话）
 * @note            注意事项（如果有的话 ）
*/
void Function();
};


///@brief 实现要点，实现的简要步骤
void MyClass::Function()
{
   …
}
```

对于返回值、参数意义都很明确的简单函数，可使用简单的函数头说明：

```
class MyClass
{
public:

///@brief   简要说明函数实现功能
void Function();
```

```
};
```

4.5 变量注释

通常变量名本身足以很好的说明变量的用途，特定情况下要额外注释说明。

【规则 4.5.1】（强制） 每个类的成员变量需注释说明用途。

【举例】

如果变量可以接受 NULL 或-1 等警戒值，须说明：

```
Private:
  //记录打开的数据表的数目，-1 表明不清楚打开的数据表的数目。
  int32_t numTotalEntries;
```

【规则 4.5.2】（强制） 所有常量应注释说明含义及用途。

【举例】

```
const int32_t NUTEST_CASES = 8; //在注册过程中进行测试案例个数
```

4.6 实现注释（Implementation Comments）

【规则 4.6.1】（推荐） 对于实现代码中巧妙的、有趣的、晦涩的、重要的地方加以注释。

【举例】

注释类型	良好格式

精彩的或复杂的代码前要加注释。	// Divide result by two, taking into account that x
	// contains the carry from the add.
	for (int32_t i = 0; i<result ->size(); i++)
	{
	x = (x << 8) + (*result)[i];
	(*result)[i] = x >> 1;
	x &= 1;
	}

比较隐晦的地方要在行尾加入行注释。	If (mmap_budget >= data_size
	& !MapData(mmap_chunk_bytes,mlock))
	return; // Error already logged.

向函数传入布尔值或整数时要注释说明含义。	bool success = Calculate(value,
	10, //Default base value.
	flase, //Not the first time
	//we call it.
	NULL); //No callback.

【规则 4.6.2】（推荐）不要用自然语言翻译代码作为注释。

【举例】下面的注释不好：

```
// Now go throuth the b array and make sure that if I occurs,
// the next element is i+1.
```

4.7 TODO 注释

【规则 4.7.1】（推荐）对那些临时的、短期的解决方案，未写完的或已经够好但并不完美的代码使用 TODO 注释。

【说明】 TODO 注释是为了标记一些未完成或完成的不尽如人意的地方，通过搜索就可以知道还有哪些活要干。

这样的注释以 TODO 开头，后面括号里加上你的大名，邮箱等，目的是可以根据统一的 TODO 格式进行查找。

【举例】

```
//TODO(lixiaochang@zdwxgd.com): Use a “*” for concatenation
//operator.
```

```
//TODO(Bill): Remove this code when all clients can handle XML
//responses.
```

第5章 头文件

下面的规则将引导你规避使用头文件时的各种麻烦。

5.1 #include 指令

【规则 5.1.1】（强制） #include 指令后只能紧跟<filename>或 " filename " 形式的语句。

【说明】这是 ISO/IEC 14882:2003 [错误!未找到引用源。] 唯一允许的#inlude 指令格式。

【举列】

```
#include   “filename.h”      //符合规则
#include   <filename.h>     //符合规则
#include   another.h         //不符合规则
```

5.2 #define 保护

【规则 5.2.1】（推荐） 所有头文件都应该使用*#define* 防止头文件被多重包含，命名格式为<PROJECT>_<PATH>_<FILE>_H_。

【说明】为保证唯一性，头文件的命名应基于其所在项目源代码树的全路径，不要在受保护的前后放置代码或者注释。

【举列】

例如，项目 foo 中的头文件 foo/src/bar/train.h 按如下标准方式保护：

良好格式	不良格式
#ifndef FOO_BAR_TRAIN_H_ #define FOO_BAR_ TRAIN _H_ …//文件内容 #endif //FOO_BAR_TRAIN _H_	#ifndef FOO_BAR_ TRAIN _H_ #include “foo.h” //不应有代码 #define FOO_BAR_ TRAIN _H_ …//文件内容 #endif //FOO_BAR_ TRAIN _H_

5.3 头文件依赖

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。如果头文件（IncludeOtherHead.h）包含了其他的头文件（BeIncluded.h），这些被包含头文件（BeIncluded.h）的任何改变也将导致包含了头文件的代码重新编译。因此，应尽量少的包含头文件，特别是那些被包含在其他头文件的（BeIncluded.h）。

【规则 5.3.1】（推荐） 使用前置声明（forward declarations）以尽量减少 .h 文件中*#include* 的数量，能依赖声明的就不要依赖定义。

【说明】使用前置声明也可以显著减少需要包含的头文件数量。举例说明，如果头文件中用到类File，但不需要访问File的定义，则头文件中只需要前置声明class File，而无需用 #include “file/base/file.h”。

在头文件中如何才能做到使用类 File 而无需访问它的定义呢？

(1) 将数据成员类型声明为 File*或 File&；

有时，使用指针成员替代对象成员会更有意义，但这样做的代价是降低代码可读性及执行效率。如果仅仅为了少包含头文件，还是不要这样替代好。

(2) 参数、返回类型为 File 的函数在头文件中只需要声明而不用去定义实现；

(3) 由于静态数据成员的定义在类定义之外，所以静态数据成员的类型可以被声明为 File；

(4) 在头文件中，如果定义的类是 File 的子类，或者含有类型为 File 的非静态数据成员，则必须为之包含头文件。

【举例】

```
//head.h 文件

class FileClass; //前置声明类 FileClass

class MyClass
{
public:
    FileClass GetFileClass(); //允许

private:
```

```
    FileClass*    fileClass;     //允许
}
```

5.4 内联函数

当函数体比较小的时候，特别是对于存取函数以及一些比较短的关键执行函数，声明为内联该函数可以令目标代码更加高效。

【规则 5.4.1】（强制） 不要内联超过 20 行的函数。

【规则 5.4.2】（强制） 不应内联那些包含循环或 *switch* 语句的函数。

【规则 5.4.3】（推荐）对于将析构函数定义为内联函数应慎重对待，析构函数往往比其表面看起来要长，因为可能会有一些隐式成员和基类的析构函数被调用。

【说明】 滥用内联将导致程序变慢，内联有可能使目标代码量或增或减，这取决于被内联的函数的大小。内联较短小的存取函数通常会减少代码量，但内联一个很大的函数将显著增加代码量。在现代处理上，由于更好的利用指令缓存，小巧的代码往往执行更快。

5.5 包含文件的名称及次序

【规则 5.5.1】（推荐） 将包含头文件的次序标准化可增强可读性、避免隐藏依赖（hidden dependencies:主要是指包含的文件编译），次序如下：C 库、C++库、其他库的.h 文件、项目内的.h 文件。其中项目内的头文件应按照项目源代码目录树结构排列，相同目录下的头文件按字母排序。

【说明】 假如 dir/foo.cpp 的主要作用是执行或测试 dir2/foo2.h 的功能，foo.cpp 中包含头文件的次序如下：

(1) dir2/foo2.h (优先位置，其他文件次序如下)；

(2) C 系统文件；

(3) C++系统文件；

(4) 其他库头文件；

(5) 本项目内的头文件。

【举例】

zdwx_project/src/foo/internal/fooserver.cpp 的包含次序如下：

```
#include “foo/public/fooserver.h”    //优先位置

#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”
```

5.6 只声明而不定义

【规则 5.6.1】（强制）头文件应该只用于声明对象、函数、类、类模板、宏等等，而不能包含对函数或对象的定义。

【说明】 因为头文件是会被其他许多文件所包含，如果含有定义语句，那么就会违背标识符的唯一性原则。

【举例】

```
// 头文件 a.h
void Foo1();
void Foo2()                    //错误
{
}
int32_t a;                     //错误
```

【规则 5.6.2】（强制） 具有外部链接属性的对象或函数应该在头文件中进行声明，如果想将对象或函数的外部属性去除，可以将它们置于不具名的命名空间或声明为 static；main 函数里的对象和函数可以作为该规则的例外情况。

【说明】 这样做的目的是降低对象和函数的可见性，也被认为是软件开发中的一种良好做法。

【举例】

```
//////////////////头文件 header.h
extern int32_t a1;
extern void Foo3();
//////////////////文件 File1.cpp
```

```
#include “header.h”
int32_t a1 = 0;                          //正确，在头文件中声明了。
int32_t a2 = 0;                          //不合规则
static int32_t a3 = 0;                   //正确，是静态的
namespace                                //放在不具名的命名空间，都正确。
{
    int32_t a4 = 0;
    void Foo1()
    {
    }
}
static void Foo2()
{
}                                        //正确，是静态的

void Foo3()
{
}                                        //正确，在头文件中声明了。
```

```
//续前表
void Foo4()
{
}                                        //不合规则

void main()
{
}                                //正确，main 函数里的对象和函数是例外情况。
```

第6章 类

6.1 结构体和类

C++中，关键字 struct 和 class 几乎含义等同；一个结构体（struct）仅仅是一个成员属性默认为 public 的类（class）[错误!未找到引用源。]。struct 常被用于在仅包含数据的对象上，可能包括常量，但没有存取数据之外的函数功能。如果需要更多的函数功能，class 更适合。

【规则 6.1.1】（强制） 仅当某一实体内只有数据时使用 *struct*，其它一概使用 *class*。

6.2 构造函数

【规则 6.2.1】（强制） 如果类中定义了成员变量，没有提供其他的构造函数，则需要显式地定义一个默认构造函数（没有参数）。

【说明】 如果没有提供其他构造函数，又没有定义默认构造函数，编译器将自动生成一个不带参数的构造函数，编译器生成的构造函数并不会对对象进行初始化。因此显式定义默认构造函数能更好地初始化对象，帮助程序员更好地了解类的功能。

【规则 6.2.2】（推荐） 仅在代码中需要拷贝一个类的对象时才定义和使用拷贝构造函数；不需要拷贝时应使用 DISALLOW_COPY_AND_ASSIGN。

【说明】 大量的类并不需要可拷贝的功能，也不需要一个拷贝构造函数或赋值操作，不幸的是，不主动声明它们，编译器会自动生成，而且是 public 的。C++中对象的隐式拷贝是导致很多性能问题和 bugs 的根源，拷贝构造函数降低了代码的可读性，相比按引用传递，跟踪按值传递的对象更加困难。

可以考虑在类的 private 中添加空的拷贝构造函数和赋值函数，只有声明，没有定义。由于这些空拷贝构造函数声明为 private，当其他代码试图使用它们时，

编译器将报错。为了方便可以使用宏 DISALLOW_COPY_AND_ASSIGN,用法见下例。

【举例】

```
// 禁止使用拷贝构造函数和赋值操作的宏
// 声明为 private
#define DISALLOW_COPY_AND_ASSIGN(TypeName)    \
   TypeName(const TypeName&);                 \
   void operator=(const TypeName&);

class Foo
{
public:
     Foo();
     ~Foo();

private:
     DISALLOW_COPY_AND_ASSIGN(Foo);
};
```

【规则 6.2.3】（推荐）构造函数中只进行那些没有实际意义的初始化，可能的话，使用 Init()方法集中初始化有意义的数据或增加一个成员标记用于指示对象是否已经初始化成功。

【说明】 简单初始化对于程序执行没有实际的逻辑意义，因为成员变量的“有意义”的值大多不在构造函数中确定。什么是有意义的初始化呢？如一个设备必须有一个 ID 号，给 ID 属性赋空值是无实际逻辑意义的初始化，赋数字才是有意义的初始化。

构造函数初始化的优点是使程序员无需担心类是否被初始化，但在构造函数中执行操作也会引起一些问题：

(1) 不易报告错误，不能使用异常；

(2) 操作失败会引起不确定状态；

(3) 构造函数内调用虚函数，调用不会派发到子类实现中，即使当前没有子类化实现，将来仍是隐患。

6.3 组合与继承

继承是C++中第二紧密的耦合关系，仅次于友元关系。紧密的耦合是一种不良现象，应该尽量避免。

C++实践中，继承主要用于两种场合实现继承（implementation inheritance）——子类继承父类的实现代码；接口继承（interface inheritance）——子类仅继承父类的方法名称。

继承提供了大量的功能，包括重用代码和改写虚函数。但其缺点也不容忽视对于实现继承，由于实现子类的代码在是在父类的代码上进行延展，要理解其实现变得更加困难；子类不能重写父类的非虚函数，当然也不能修改其实现。

【规则 6.3.1】（强制） 继承都采用 *public* 方式，如果想私有继承的话，应该采取包含基类实例作为成员的方式来替代。

【规则 6.3.2】（强制）当重定义派生的虚函数时，在派生类明确声明其为 *virtual*。

【说明】 虽然不加 virtual 系统也会默认为虚函数，但阅读者需要检索类的所有祖先以确定该函数是否为虚函数。

【规则 6.3.3】（推荐） 如果用组合就能表示类的关系，那么应该使用组合代替继承，除非知道后者确实对设计有好处。

【说明】 所谓组合就是指在一个类型中嵌入另一个类型的成员变量，与继承相比，组合有如下重要优点：

(1) 奇异现象减少。子类从一个父类继承后，与父类在同一名字空间中的函数或函数模板的名字查找变得非常复杂，难以调用。

(2) 更广的适用性。有些类一开始并不是想设计成基类，但大多数类都能充当一个成员的角色。

(3) 更健壮、更安全。继承的较强耦合性命名编写没有错误的安全代码更加困难。

(4) 复杂性和脆弱性降低。继承会带来额外的复杂性情况，如名字隐蔽。

因此，使用组合通常比使用继承更适宜，努力做到只在“is-a”的情况下才用继承。

6.4 接口

【规则 6.4.1】（推荐） 接口类型以 *Interface* 为后缀，满足下列条件才可定义为接口：

(1) 只有纯虚函数（=0）和静态函数；

(2) 没有非静态数据成员；

(3) 没有定义任何构造函数。如果有，也不含参数，并且为 protected；

(4) 如果有父类，也只能继承满足上述条件并以 Interface 为后缀的类。

【说明】 接口类不能被直接实例化，因为它声明了纯虚函数。为确保接口类的所有实现可被正确销毁，必须为它声明一个虚析构函数（作为第 1 条的例外，析构函数不能是纯虚函数）。

【举例】

```
class DoorInterface
{
public:
    virtual void Open() = 0;     //声明为纯虚函数
    virtual void Close() = 0;

protected:
    virtual ~DoorInterface();    //虚析构函数
};
```

6.5 多重继承

真正需要用到多重继承的时候非常少，多重实现继承（implementation inheritance）表面看上去是不错，但通常可以找到更加明确、清晰的、不同解决方案，详见下面的规则。

【规则 6.5.1】（推荐）只有当所有超类除第一个外都是纯接口时才能使用多重继承。为确保它们是纯接口，这些类必须以 *Interface* 为后缀。

【举例】

```
class MultipleClass: public BaseClass,
                     public BaseAInterface,
```

```
                    public BaseBInterface
{
    …
};
```

第7章 健壮性与容错性

本章内容参考了部分 MISRA C++ 2008 规范[错误!未找到引用源。]，旨在提高软件的安全性和健壮性。

7.1 编译

【规则 7.1.1】（推荐）高度重视警告，理解所有警告，应该要求构建是干净利落的（没有警告）。

【说明】 编译器是你的朋友，如果它对某个构造发出警告，一般表明代码中存有潜在的问题[错误!未找到引用源。]。独立编译每个头文件，并确保没有产生错误或警告。

头文件应该自给自足以确保所编写的每个头文件都能够独自进行编译，为此需要包含其内容所依赖的所有头文件。规避头文件依赖请参考第 5 章头文件。

7.2 扇入和扇出

【规则 7.2.1】（推荐） 扇入和扇出控制在 2-7 之间。

【说明】 按照结构化设计方法，一个应用程序是由多个功能相对独立的模块所组成。

扇入：A 模块的扇入指的是直接调用了 A 模块的上级模块的个数。扇入大表示模块的复用程序高。

扇出：是指该模块直接调用的下级模块的个数。扇出大表示模块的复杂度高，需要控制和协调过多的下级模块；但扇出过小（例如总是 1）也不好。扇出过大一般是因为缺乏中间层次，应该适当增加中间层次的模块。扇出太小时可以把下级模块进一步分解成若干个子功能模块，或者合并到它的上级模块中去。

设计良好的软件结构，通常顶层扇出比较大，中间扇出小，底层模块扇入大。

7.3 圈复杂度

【规则 7.3.1】（推荐） 圈复杂度控制在 10 以内。

【说明】 “圈复杂度”用来衡量一个模块判定结构的复杂程度，数量上表现为独立执行路径条数，即合理的预防错误所需测试的最少路径条数，圈复杂度大说明程序代码可能质量低且难于测试和维护，根据经验，程序的可能错误和高的圈复杂度有着很大关系” 。

7.4 无效代码

【规则 7.4.1】（强制） 程序中不得含有不被执行的代码。

【举例】

```
switch (para)
{
    local = para; //该代码永远不会被执行
    case 1:
    {
        break;
    }
    default:
    {
        break;
    }
}
```

【规则 7.4.2】（强制） 程序中不得含有无效执行路径。

【举例】

```
//u16a 是一个 16 位无符号整数
if (u16a < 0U)   //不符合该规则，u16a 是无符号整数，永远为 true
{
    //总被执行
}
else
{
    //永远不会执行到
}
```

【规则 7.4.3】（强制） 程序中不得含有不被使用的变量。

【说明】 没有使用的变量是代码中的噪声（noise）,这些不使用的变量名会污染它的命名空间。

【规则 7.4.4】（强制） 程序中不得含有不被使用的类型声明。

【举例】

```
int32_t UnusedType(viod)
{
    typedef signed char LocalType; //不符合规则，没有使用这个定义
                                   //的类型。
    return 67;
}
```

【规则 7.4.5】（强制）调用返回值非空（void）的函数时，程序应保证该返回值总会被使用。

【说明】 C++允许单纯调用返回值非空的函数，而不使用其值，但这为系统的不稳定带来了隐患。

【举例】

```
int32_t Foo(int32_t para)
{
    return para;
}
void    Discarded(int32_t para)
{
    Foo(para);   //不符合规则，返回值丢失未用。
}
```

【规则 7.4.6】（强制） 程序中不得含有“死代码”（dead code）。

【说明】 代码中的一些执行语句，去除它们之后也不会影响到程序的输出，这些语句称为“死代码”。

【举例】

```
int32_t HasDeadCode(int32_t para)
{
    int32_t local = 99;
    para = para + local;
    local = para;              // dead code
    if (0 == local)            // dead code
        local++;               // dead code
    return para;
}
```

7.5 函数

【规则 7.5.1】（强制） 函数应该具有外部副作用（external side effects）。

【说明】 如果函数即不返回值，也不产生外部副作用，那它只是仅仅消耗时间而无助于产生输出结果，这很可能与设计者的期望相背。

下面是一些函数外部副作用的例子：

(1) 读写一个文件、输入输出流等；

(2) 改变非局部变量的值；

(3) 改变引用参数的值；

(4) 引发一个异常；

(5) 使用一个 volatile 对象。

【举例】

```
void Pointless(void)   //不符合规则，函数没有产生任何外部副作用
{
    int32_t local;
    local = 0;
}
```

【规则 7.5.2】（强制） 每一个定义的函数应该至少被调用一次。

【举例】

```
void Foo1() {}   // 符合规则
```

```
void Foo2() {}   // 不符合规则，从未被调用。
void Foo3();     // 符合规则,函数声明。
void main()
{
    Foo1();
}
```

【规则 7.5.3】（强制） 函数的声明、定义和调用都不采用缺省参数。

【说明】 C++语言可以给函数中的形式参数指定缺省值，但使用缺省值给程序员和修改者增加了不必要的记忆负担，其实这也是C++特性臃肿的一种表现，限制这些不必要的特性能使代码简化，避免可能导致的各种问题，还可以让程序员明确每一个参数的作用，不必担心调用的函数会由于参数的多少而产生不必要的麻烦。

【举例】

```
//不良的格式，采用了缺省参数
float32_t Distance(float32_t x, float32_t y=0);
void main()。
{
    float32_t x = 4.5, y = 2.7;
    cout<<Distance(x)<<endl;   // 不良的格式，采用了缺省参数。
    cout<<Distance(x, y)<<endl;
}

float32_t Distance(float32_t x, float32_t y)
{
    return x – y;
}
```

【规则 7.5.4】（强制） 对于非虚函数（non-virtual function），不得存在未使用的函数参数。

【例外】 该函数包含不具名参数（unnamed parameter）且函数作为回调使用，可以存在未使用的不具名参数。

【说明】 未使用的函数参数往往是由于设计的改变而产生，会导致函数参数列表的不匹配。

【举例】

```
int32_t CallbackA(int32_t a, int32_t b) //违反规则，有参数没使用。
{
   return a;
}


//作为例外情况，不具名参数可以不使用。
int32_t CallbackB(int32_t a, int32_t)
{
    return a;
}
```

【规则 7.5.5】（强制） 函数代码行数（不包含单行注释）应控制在 100 行之内。

【说明】 控制函数的代码行数可以提高代码可读性，优化程序结构，对于行数过多的函数可以进行分解。

【例外】 有些函数由于采用防御式编程方法，需对输入数据进行检测，当检测语句占用行数较多时，允许函数代码行数超过 100 行。

【规则 7.5.6】（推荐） 只给一个实体（类、函数、名字空间、模块和库）赋予一个定义良好的紧凑的职责。

【说明】 具有多个不同职责的实体通常都是难于设计和实现的。“多个职责”经常意味着“多重性格”——可能的行为和状态的各种组合方式。应该选择目的单一的函数，小而且目的单一的类，以及边界清晰的紧凑模块。

尽量设计具有单入口和单出口的函数，使函数具有单出口 IEC 61508 **错误!未找到引用源。** 的要求。设计单入口单出口的函数不仅使程序具有良好的结构征，同时容易保证程序的正确性。设计具有单出口的函数

7.6 命名空间

使用命名空间可以避免名字冲突，它能将命名分割在不同的命名空间内，举列来说，两个不同项目的全局作用域都有一个类 Foo，编译或运行时会造成冲突，

每个项目将代码置于不同命名空间中，PROJECT1::Foo 和 PROJECT2::Foo 作为不同符号自然不会冲突。

（1）不具名命名空间

【规则 7.6.1】（强制） .cpp 文件中，使用不具名命名空间，以避免运行时的命名冲突；而.h 文件中不能使用不具名的命名空间。

【说明】 .h 文件中若使用不具名的命名空间，会导致违背唯一定义原则（One Definition Rule）。

【举例】

```
//.cpp 文件中
namespace
{
    enum PositionType{UNUSED, EOF, ERROR}; //经常使用的符号
    bool AtEof()
    {
       …
       return position == EOF;        //使用本命名空间内的符号 EOF
    }
} //namespace(记住要注释，更清晰)
```

```
//.h 文件中
namespace       //不符合规则, .h 文件中不得用不具名命名空间。
{
    class MyClass
    {
       …
    };
}
```

（2）具名命名空间

【规则 7.6.2】（推荐）最好不要使用 *using* 指示符，以保证命名空间下的所有名称都可以正常使用；如果要用，也只在. cpp 文件、. h 文件的函数、方法或类中。

【举例】

```
//允许在.cpp 文件,.h 文件的函数、方法或类的内部使用
using   :: FOO::BAR;

//在.cpp 文件、.h 文件的函数、方法或类的内部，还可以使用命名空
//间别名。
```

```
namespace FBZ    =    :: FOO::BAR;
```

【规则 7.6.3】（推荐）应将非成员函数置于某个命名空间中，如果单纯为了封闭若干个不共享任何静态数据的静态成员函数而创建类，不如使用命名空间。

【说明】 有时，不把函数限定在类的实体中是有益的，甚至需要这么做，要么作为静态成员，要么作为非成员函数，非成员函数不应依赖于外部变量，并尽量置于某个命名空间中。

【举例】

```
namespace MYNAMESPACE
{
    static void Foo()    //将非成员函数放在命名空间内
    {
        …
    }
}
```

7.7 变量

（1）局部变量

【规则 7.7.1】（强制） 在声明变量时将其初始化。

【说明】 C++允许在函数的任何位置声明变量，应在尽可能小的作用域中声明变量，这使得代码易于阅读，易于定位变量类型和初始值。同时，使用初始化代替声明+赋值方式。

如果定义的变量是一个对象，每次进入作用域都要调用其构造函数，每次退出作用域都要调用其析构函数。在循环语句中，将变量放到循环作用域外面声明要高效的多。

【举例】

声明时初始化：

```
int32_t i;
i = 30;                          //不正确——初始化和声明分离
int32_t i = 0;                   //正确——初始化时声明
```

定义变量放在循环体外：

良好的格式	不良的格式

//高效实现方式
TrainClass train;
for (i=0; i<100000; ++i)
{
//构造函数和析构函数只调
//用 1 次
train.DoSomething(i);
}
	//低效实现方式
	for (i=0; i<100000; ++i)
	{
	TrainClass train; // 构造函数和析
	//构函数分别调用 100000 次!
	train.DoSomething(i);
	}

【规则 7.7.2】（推荐） 将函数变量尽可能置于最小作用域内，离第一次使用越近越好。

（2）全局变量

【规则 7.7.3】（强制） *class* 类型的全局变量是被禁止的，如果你一定要使用，请使用单件模式（singleton pattern）；内建类型的全局变量是允许的；多线程代码中非常数全局变量也是被禁止的。

【说明】 虽然允许在全局作用域中使用全局变量，使用时务必三思。大多数全局变量应该是类的静态数据成员，或者当其只在.cpp 文件中使用时，将其定义到不具名命名空间中，或者使用 static 以限制变量的作用域。静态成员变量视作全局变量，所以也不能是 class 类型的。

（3）不使用动态变量

【规则 7.7.4】（强制） 在开发与安全性相关的软件时，不准使用动态变量和动态对象；不允许使用动态堆内存分配（dynamic heap memory allocation）的方式。

【说明】 使用动态内存分配有可能会导致存储溢出运行时错误（out-of-storage run-time failure）,内建的 new 和 delete 操作符，还有 calloc,malloc,realllloc 和 free 等函数都是使用动态堆内存，动态堆内存分配还有可能导致内存溢出、数据不一致、内存耗尽等意想不到的一些问题。

【举例】

```
void Foo()
{
   int32_t* i = new int32_t; //不符合规则
   delete i;
}
```

7.8 标识符的唯一性

（1）相同命名空间内

【规则 7.8.1】（强制） 声明在某内部作用域的一个标识符不应屏蔽了外部作用域的标识符。

【说明】 如果在某块内声明了一个标识符，同时采用了外部已存在的名字，则在该块内，外部的标识符会被隐藏，这会造成标识符的混乱。

【举例】

```
int32_t i;       //外部变量 i
…
{
    int32_t i; //这是块内的变量 i,隐藏了外部的变量 i。
    i = 3;     //不符合规则，到底是给哪个 i 赋值,造成混乱。
}
```

【规则 7.8.2】（强制） 在同一作用域内,定义的类型名（*typedef* name）应该是唯一的标识符。

【举例】

```
void   f1()
{
  //定义了一个类型，名为 TYPE，不是唯一的标识符。
    typedef int32_t TYPE;
}

void   f2()
{
    const float32_t TYPE = 3.14;   //不合规则，与类型同名。
}
```

【规则 7.8.3】（强制） 在同一命名空间内，一个类（*class*）、联合体（*union*）或枚举类型的名字应该是唯一的标识符。

【举例】

```
void    f1()
{
    class TYPE
    {
    }; //定义了一个类，名为 TYPE，不是唯一的标识符。
}

void    f2()
{
    const float32_t TYPE = 3.14;    //不合规则，与类型同名。
}
```

（2）不同命名空间内

【规则 7.8.4】（强制） 不考虑作用域，对于非成员对象或函数、具有静态存储期的对象或函数来说，它们的标识名（identifier name）不能被重用。

【举例】

```
//下面的代码，编译器能理解，也能编译，但这些重命名的存在会让
//程序员错误地将具有相同名字而不相关的变量相联系。
namespace    NAME1()
{
   static int32_t global = 0;
}//namespace NAME1
namespace    NAME2
{
   void    fn()
   {
      int32_t global;    //不合规则，与 static 变量同名。
   }
}//namespace NAME2
```

【规则 7.8.5】（强制） 在不同的编译单元都要求所有声明的变量、对象或函数具有可兼容的类型，保证对象或函数的类型相兼容的最佳途径是声明唯一的标识符。

【举例】

```
////////////////////////////////////////  文件  FileA.cpp
extern int32_t a;
extern int32_t b[5];
```

```
extern char_t c;
int32 Foo1();
int32 Foo2(int32_t para);
//////////////////////////////////////////文件 FileB.cpp
extern float32_t a;              // 不合规则，类型不兼容。
extern int32_t b[5];             // 正确
int32_t c;                       // 不合规则，类型不兼容。

char_t Foo1();                   // 不合规则，类型不兼容。
char_t Foo2(char_t para);        // 正确，“签名”不同，是不同的函数。
```

7.9 宏

【规则 7.9.1】（强制）用 *const* 或 *enum* 定义易于理解的常量，而不使用宏；采用 inline 避免函数调用的开销，而不使用宏定义函数；使用 *template* 指定函数系列和类型系列,而不使用宏定义类型。

【例外】 #define 保护，使用宏 DISALLOW_COPY_AND_ASSIGN 创建空的拷贝构造函数（见 6.2 节）。

【说明】 宏调用看上去很像符号或者函数调用，但实际上并非如此。它会根据其使用的环境令人惊奇地展开为各种东西。宏需要进行文本替换，因此编写远距离也正确的宏接近于一种魔法，而精通这种魔法既无意义又无趣味。C++为#define 的主要应用提供了如下的替代方式**错误!未找到引用源。**：

(1) const 用于常量；

(2) inline 用于开子程序(open subroutines)；

(3) template 用于以类型为参数的函数；

(4) template 用于参数化类型；

(5) namespace 用于更一般的命名问题。

【规则 7.9.2】（强制） 在声明一个数组时，需要显式地给出数组大小，或者通过初始化明确数组的大小。

【说明】 尽管编译器在定义一个没有指定大小的数组时会报错，但当声明一个外部的数组时，不指定大小是可以编译通过的。如果都显式地给出数组大小，程序将会更安全。

【举例】

```
int32_t array1[];                 //不符合规则
extern int32_t array2[];          //不符合规则，虽然编译能通过
```

```
int32_t array3[10];               //正确
extern int32_t array4[10];        //正确
int32_t array5[]= {2, 5, 98};     //正确，通过初始化明确数组大小。
```

7.10 类型与表达式

（1）重新定义数值类型

【规则 7.10.1】（强制） 在开发与安全性相关的软件时，不应该使用基本的数值类型，如 *int*、*short*、*long*、*float*、*double*、*long double* 等，而应该用 *typedef* 重新定义，定义的类型名应该能表达出数据的大小和符号特征。

【说明】这条规则的目的是让程序员可以区分和明了不同数值类型的存储大小；该规则仅在开发对安全性有较高需求的软件系统时使用。

下面给出 POSIX 标准在 32 位机器上重新定义的类型名：

```
typedef               char        char_t;
typedef signed        char        int8_t;
typedef signed        short       int16_t;
typedef signed        int32       int32_t;
typedef signed        long        int64_t;
typedef unsigned char             uint8_t;
typedef unsigned short            uint16_t;
typedef unsigned int32            uint32_t;
typedef unsigned long             uint64_t;
typedef               float       float32_t;
typedef               double      float64_t;
typedef long          double      float128_t;
```

（2）表达式

【规则 7.10.2】（强制） 用好含有 *bool* 类型的表达式，这些表达式中不可以出现下列运算符除外的内建操作符（built-in operators），这些例外的运算符有赋值运算符“=”，逻辑运算符（“&&”、“||”、“!”），等式运算符“==”和“!=”，一元操作的“&”运算符，以及条件运算符（ ?: ）。

【说明】 逻辑运算符和位运算符很容易混淆，一些没有意义的 bool 值的运算要避免，制定这条规则可以让你避免犯一些不必要的错误。

【举例】

```
bool      b1 = true;
bool      b2 = false;
int32_t   a;

if (b1 & b2)                    //不符合规则
if (b1 < b2）                   //不符合规则
if (~b1)                        //不符合规则
if (b1 ^ b2)                    //不符合规则
```

```
if (b1 == false)                //符合规则
if (b1 == b2)                   //符合规则
if (b1 != b2)                   //符合规则
if (b1 && b2)                   //符合规则
if (!b1)                        //符合规则
a= b1 ? 3 : 7;                  //符合规则
```

【规则 7.10.3】（强制） 用好含有 *enum* 类型的表达式，这些表达式中不可出现下列运算符除外的内建操作符（built-in operators），这些例外的运算符有下标操作符“[]”，赋值运算符“=”，等式运算符“==”和“!=”，一元操作的“&”运算符，关系运算符（“<”、“<=”、“>”、“>=”）。

【举例】

```
enum MyColour{RED,   GREEN,   BLACK} colour;
…
if (colour == RED)              //符合规则
if (colour < BLACK )            //符合规则
if (RED && GREEN)               //不符合规则
colour = RED + GREEN;           //不符合规则
```

【规则 7.10.4】（强制） 除了含有“&&”、“||”、“？”和逗号的表达式，表达式的值应该不依赖于它的求值顺序。

【举例】

良好格式，不依赖求值顺序	不良格式，依赖求值顺序
i++; x = b[i] + i;	x = b[i] + i++;

x = fun(i,i+1); i++;	x = func(i++, i);
p->task(p); p++;	p->task(p++);

【规则 7.10.5】（强制） 对于“++”和“—”操作符，优先使用它们的标准形式；优先使用前缀形式。

【说明】 标准形式是：

```
T& T::  operator++()          //前缀形式
{
    …执行递增                  //完成任务
    return *this;              //总是返回 *this
}

T T::  operator++(int32_t)    //后缀形式
{
      T old(*this);            //保存旧值
      ++*this;                 //调用前缀版本
      return old;              //返回旧值
}
```

后缀形式返回的是原值，面前缀形式返回的是新值，前缀形式在效率上会略高一些，因为前缀形式少创建了一个对象。

【举例】

良好格式	不良格式

//使用前缀形式
for(i = 1; i<100; ++i)
	//使用后缀形式
	for(i = 1; i<100; i++)

【规则 7.10.6】（强制） *if* 条件中的表达式和循环语句中的循环控制表达式应该是返回 *bool* 类型的值，条件运算符（ ?: ）的第一个操作数应该为 *bool* 类型。

【举例】

```
int32_t x, y, z;
bool boolA, boolB;
if ( x )                                //不符合规则
if ( x > 0 )
if (x && ( boolA <= boolB) )            //不符合规则
for (int32_t x=10; x; --x)              //不符合规则
int32_t b = x ? 4 : 5;                  //不符合规则

b = (x >= 0) ? 4 : 5;                   //正确的表达式
while (true)                            //正确的表达式
for(int32_t x=10; x>0; --x)             //正确的表达式
```

【规则 7.10.7】（强制） 避免两个相近的浮点数进行比较运算。

【举例】

```
float32_t f1 = 3.141593;
float32_t f2 = 3.1415926;
if (f1 <= f2)                           //不符合规则
```

【规则 7.10.8】（强制） 不使用“魔数”。

【说明】 要避免在代码中使用诸如 42 和 3.1415926 这样的文字常量，而应该用符号名称和表达式替换它们，如 width * aspectRatio.

【举例】

良好格式	不良格式

int32_t CalculateArea(int32_t r)
{
  const float32_t PI = 3.1415926;
  return PI * r * r;
 	
}
	//使用了“魔数”
	int32_t CalculateArea(int32_t r)
	{
	  return 3.1415926 * r * r;
	}

（3）类型转换

【规则 7.10.9】（强制）不能在有符号和无符号的整型数值之间进行隐式的转换。

【举例】

```
typedef unsigned    char    uint8_t;
typedef signed      char    int8_t;
```

```
void    f()
{
    int8_t      sign_val = 4;
    uint8_t     uns_val = 6;
    s_val = uns_val;                                    //不正确，隐式转换
    uns_val = uns_val + sign_val;                       //不正确，隐式转换
    uns_val = static_cast< uint8_t >( sign_val ) + uns_val;   //正确
}
```

7.11 指针

【规则 7.11.1】（强制） 不要使用三级及以上的间接指针。

【说明】 使用多于二级的间接指针会让程序员难以理解代码的表现性能，因此应该避免使用。

【举例】

```
typedef int32_t*    INTPTR;
void Foo(int32_t*         par1,                 //符合规则
                  int32_t**       par2,         //符合规则
                  int32_t***      par3,         //不符合规则
                  INTPTR*         par4,         //符合规则
                  INTPTR**        PAR5,         //不符合规则
                  int32_t*        par6[],       //符合规则
                  int32_t**       par7[])       //不符合规则
```

【规则 7.11.2】（强制） 只有当两个指针指向同一数组中的元素时，才可在指针之间进行减法操作。

【举例】

```
int32_t a1[10];
int32_t a2[10];
int32_t* p1 = &a1[1];
int32_t* p2 = &a2[10];
int32_t diff;
diff = p1 - a1;                    //符合规则
diff = p2 – a2;                    //符合规则
diff = p1 – p2;                    //不符合规则
```

7.12 代码安全

编写安全的软件非常重要，不良的编码会造成软件漏洞，给软件带来安全隐患；因此编码时需遵循但不限于以下规则：

【规则 7.12.1】（强制） 不得使用输入输出库〈cstdio〉中的函数。

【说明】 cstdio 库中包含了一些文件读写和 I/O 方法如 fgetpos、fopen 、ftell、gets、perror、remove、rename，以及具有转换功能的函数 atof、atoi、atol 等，这些方法包含不确定的举为，给系统安全带来威胁。更为详细的说明可以参考 ISO/IEC 9899:1990 错误!未找到引用源。。

【举例】

```
#include <cstdio>                    //不符合规则
void fn()
{
        char    array[5];
        gets(array);                 //会导致存储溢出（buffer overflow）
}
```

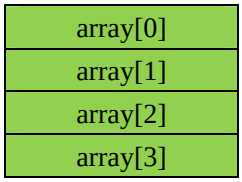
【规则 7.12.2】（强制） 不得使用库<cstring>中的一些对边界不作限制的函数，如 strcpy、strcmp、strcat、strchr、strspn、strspn 等。

【说明】 这些函数不会检查操作的数组是否会超越边界，因此对数组的读写有可能会溢出数组的边界，覆盖内存中与该数组邻近的存储区域。

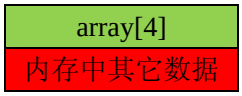
【举例】

```
#include <cstring>                   //不符合规则
void Foo(const char_t* pChar)
{
        char_t    array[5];
        strcpy (array,pChar);        //会导致存储溢出（buffer overflow）
}
```

下面用一个图示说明：



调用函数 get(array)，如果输入的字符数大于数组的大小，将会覆盖红色区域中的其它数据，破坏系统数据。



【规则 7.12.3】（强制） 不使用 goto 语句。

【说明】 从理论上讲，goto 语句是没有必要的，实际上不用它也能很容易地写出代码。所有带有 goto 语句的程序代码都可以改写成不包含 goto 语句的程序[错误! 未找到引用源。]。

【规则 7.12.4】（强制） 不使用递归函数。

7.13 防御式编程

防御式编程是提高软件质量技术的有益辅助手段。防御式编程的主要思想是：子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误数据。这种思想是将可能出现的错误造成的影响控制在有限的范围内。

防御式编程是一种主动预防问题的编码风格,作为一种编程实践,防御式编程是由很多小目标融合而成的,例如上文已提到的编写可读性强的代码、正确的命名规则、以及运用设计模式。为明确防御式编程的概念，本规范仅将对函数输入值进行检测的手段称作为防御式编程，具体如下：

防御式编程：检查来自于外部资源的所有数据值，如来源于网络的数据、其他进程的数据和来源于文件的数据等，检查的目的是保证数据值在一个允许的范围内，一旦发现非法输入，应根据情况进行处理。

【规则 7.13.1】（强制） 对于一个类中的公共方法（public function），如果有输入参数，必须对所有输入参数进行检测。

【举例】

```
float32_t GetArea( const float32_t radius)
{
        const float32_t PI = 3.1415926;
        float32_t area = 0;
        if (radius > 0)     //对输入值进行判断，不然面积为负数。
        {
            area = PI*radius*radius;
```

```
        }
        else
        {
            area=0;
        }
        return area;
}
```

【规则 7.13.2】（推荐） 对于非公共函数,如 Fun(InputType p)，如果该函数含有多个调用相同输入参数的子函数，如 FunA(InputType p)、FunB(InputType p)，则只需在所有子函数前对输入参数执行一次检测，而不要求在每个子函数中对同一输入参数都进行检测。

【举例】

```
void Fun( const float32_t sameInput)
{
    float32_t a, b;

    if (sameInput<0)     //只在子函数之前对输入参数进行检测。
    {
        //执行相应的处理
    }
    else
    {
        //执行相应的处理
    }
    ….
    a = FunA(sameInput);
    b = FunB(sameInput);
    …..
}

float32_t FunA( const float32_t sameInput)
{
    return sameInput+1;     //可以不采用防御式编程。
}

float32_t FunB( const float32_t sameInput)
{
    return sameInput+1;     //可以不采用防御式编程。
}
```