| **上海脑虎科技有限公司** | | 起草人 |  |
| --- | --- | --- | --- |
| **C++语言编程规范** | | 审核人 |  |
| 文件编号 | NH-RD-WG-003 | 批准人 |  |
| 版本号 | A/0 | 颁发部门 |  |
| 生效日期 | 2024.10.17 | 分发部门 |  |

# 目的

本规范目的规范C++语言编码工作，从目录、结构、函数、变量、注释各个方面对C++语言编程作出了具体指导。

# 适用范围

适用于本公司及其子公司内使用C++语言编码的部门和产品。

# 术语和定义

无。

# 规范

## 目录组织结构

### 目录和文件命名规则

1. 目录名和文件名应使用英文字母命名, 禁止使用中文或者全部使用数字。
2. 目录名和文件名中不应包含空格。
3. 目录名称应使用小写英文字母进行命名。

### 应用程序的文件组织形式

应用程序是可以独立运行并且功能完成的软件产品，其目录组织形式如下：

产品名称

模块名称

├─common

├─模块

├─thirdparty

├─doc

├─test

├─output

├─debug

├─release

├─config

└─log

目录说明:

|  |  |
| --- | --- |
| **目录名称** | **目录描述** |
| common | 用于编译项目公共的功能函数。  例如：枚举、全局结构体、加密算法、字符转换等功能类 |
| 模块 | 根据模块的区分分别建不同的功能文件夹。  模块保持独立，尽可能减少对其他模块的依赖。  模块内部采用三层架构继续划分子模块。 |
| thirdparty | 当前项目所依赖的所有第三方软件，可以按类建立子目录。 |
| doc | 项目所涉及文档的文件夹。 |
| test | 单元测试代码，可以按照测试用例来建立子目录。 |
| output/debug  output/release | 编译好的二进制库文件，如\*.lib文件，\*.dll文件等。  可以按编译器、编译选项、平台分设子目录。如在VC中，可以分为debug, release两个子目录。 |

### 接口文件组织规则

接口文件主要是指C/C++代码的头文件，内联函数定义文件，模板实现文件等。从模块角度来分，可以分为外部接口文件和内部接口文件。外部接口文件供模块外部调用着使用，内部接口文件只在模块内部使用。

作为接口的文件，必须放在模块目录中；内部实现细节的接口文件放在自己模块目录（项目目录或者子目录）下。

对于一个模块的接口文件，可能会划分更细的子模块，这种情况下要求上层目录的接口不能访问底层目录的接口。如：

├─模块

├─module1

├─module2

这个结构中，module2是module1的子模块，在这种情况下，module2可以引用module1的接口文件， 但是反过来module1引用module2的接口文件是不允许的。这样做的主要目的是避免模块之间的循环依赖。

## 文件结构

文件统一使用UTF-8编码格式存储，防止出现中文编码问题。

### 文件名命名规则

1. 每一个公共类的定义必须放在单独的头文件中，禁止多个公用类放在同一个文件内定义。
2. 接口文件/实现文件的文件名，以这个类的类名（不包括类前缀，但区分大小写）命名。比如一个类的类名为CMyWnd，那么对应的文件名（不包括后缀）应该命名为MyWnd。
3. 类的定义放在.h文件中。所有实现文件，不放在.h文件中。
4. 类的实现文件放在.cpp文件中，C代码的实现文件放在.c文件中。

文件类型以及描述

|  |  |
| --- | --- |
| 扩展名（统一使用小写字母） | 描述 |
| .h | 接口定义文件 |
| .cpp | 类的实现文件 |
| .c | C 代码的实现文件 |
| .inl | inline函数，template实现函数的文件 |
| .asm | 汇编实现文件 |

*注：对于Windows环境，资源文件等其他类型文件的扩展名使用开发环境的默认规则。*

### 文件头注释

每一个文件，包括头文件，实现文件等，要求有头文件注释。头文件注释格式如下：

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

\* 版权所有（C）2022-2023，上海脑虎科技有限公司，保留所有权利。

\* 作者 : <作者姓名><邮件地址>

\* 修改记录:

\* <修改者姓名><邮件地址> <修改内容>

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

* 原则上由作者进行维护，本人修改无需修改记录，他人修改时需填写修改记录
* 产品开发过程中，不需要修改版本号，也不用填写修改记录
* 修改记录一直保留在文件中

### 头文件

#### 文件组成

头文件的扩展名是.h，一般由以下几部分组成：

|  |  |
| --- | --- |
| 文件头注释 | 每一个头文件，都应该以一个标准的文件头注释开始。 |
| 预处理块 | 为了防止头文件被重复引用，应当用#ifndef/#define/#endif结构生成预处理块。 |
| 函数和类/结构的声明等 | 声明类或者函数的接口等 |
| 需要包含的内联函数定义文件（可选） | 如果类有内联函数定义或者模板实现定义，可以做成单独的文件，然后在头文件中#include进来。 |

#### 预处理块

头文件的预处理块采用如下的方式定义：

#ifndef \_\_CFileName\_h

#define \_\_CFileName\_h 1

...

#endif //\_\_CFileName\_h

#### 编码规则

1. 头文件的文件名与类名一致

头文件的文件名应与定义的类一致，这样对于在整个工程中阅读和查找代码，可以带来很多便利。比如类名为CMyWnd，那么对应的头文件名应该为MyWnd.h。

1. 一个头文件中只定义一个公用类

在一个头文件中，尽量只定义一个公用类，这样可以使头文件的结构清晰，有利于提高代码可读性。对于该公用类独立使用的struct, enum,类型定义等可以放在这一个头文件中。

### 内联函数定义/模板实现文件

在内联函数较多的情况下，为了避免头文件过长、版面混乱，可以将所有的内联函数定义移到一个单独的文件中去，这样的文件称为一个内联函数定义文件。

对于模板的实现代码，也可以像内联函数那样将代码放在一个单独的文件中，这样的文件称为模板实现文件。

内联函数定义和模板实现可以放在同一个文件中，将这个文件命名为“filename.inl”，其中“filename”与相应的头文件相同，扩展名为.inl，然后再用#include指令将它包含到类声明的后面。

#### 文件组成

内联函数定义/模板实现文件由以下几部分组成：

|  |  |
| --- | --- |
| 文件头注释 | 每一个文件，都应该以一个标准的文件头注释开始。 |
| 内联函数定义 | 内联函数的实现代码 |
| 模板实现 | 模板的函数实现代码 |

内联函数定义和模板实现内容可以同时有，也可以只有其中一种类型。

#### 编码规则

与头文件不同，内联函数定义/模板实现文件通常不需要定义预处理块，这是因为它通常被包含在与其相应的头文件预处理块内。

内联函数定义/模板实现的代码，应该按照头文件中对应函数的声明顺序编写实现代码，这样有利于提高代码可读性。

### 实现文件

实现文件包含所有数据和函数的实现代码。对于C++文件，扩展名使用.cpp，对于C文件，扩展名使用.c 。

#### 文件组成

|  |  |
| --- | --- |
| 文件头注释 | 每一个实现文件，都应该以一个标准的文件头注释开始。 |
| 对应头文件的引用 | 引用声明了此文件实现的类、函数及数据的头文件 |
| 对一些仅用于实现的头文件的引用 | 将仅与实现相关的接口包含在实现文件里（而不是头文件中）是一个非常好的编程习惯。这样可以有效地屏蔽不应该暴露的实现细节，将实现改变对其它模块的影响降低到最少 。 |
| 实现代码 | 数据和函数等的实现代码 |

#### 编码规则

包含头文件的时候，应把此实现文件对应的头文件放在最前面，这样可以避免实现文件中包含的其他文件内容对头文件产生影响。

实现文件中的代码，应该按照头文件中对应函数的声明顺序编写实现代码，这样有利于提高代码可读性。

### 包含头文件规则

1. 引用文件的格式
   * + 包含头文件的文件名要求与实际文件名完全一致，区分大小写。
     + 用#include <FileName.h>格式来引用标准库和系统库的头文件（编译器将从标准库目录开始搜索）。
     + 用#include "FileName.h"格式来引用当前工程中的头文件（编译器将从该文件所在目录开始搜索）。
2. 包含头文件的顺序

当包含多个文件时，对于所包含的头文件顺序依次是：

* + - 当前工程头文件
    - 其他模块头文件
    - 系统库头文件（平台相关，编译器相关的库文件）
    - 标准库头文件（标准C++库文件）

1. 包含最少的头文件

对于一个头文件，应该尽可能少的去包含其他头文件，这样可以减少头文件之间的依赖关系，当工程比较大的时候，也可以提高编译速度。

如果头文件中不需要声明类的对象，只是使用指针或者引用类型，可以做类的向前声明，而不需要包含对应的头文件。

## 命名规则

在整个项目中统一使用的命名规范，可以提升源代码的可读性和软件的可维护性。

### 标识符命名的原则

* 同一性

在编写一个子模块或派生类的时候，要遵循其基类或整体模块的命名风格，保持命名风格在整个模块中的同一性。

* 标识符组成

识符采用英文单词或其组合，命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。

禁止使用汉语拼音来命名标识符。

* 最小化长度 && 最大化信息量原则

在保持一个标识符意思明确的同时，应当尽量缩短其长度。

* 避免过于相似

不要出现仅靠大小写区分的相似的标识符，例如“i”与“I”，“function”与“Function”。

* 避免在不同级别的作用域中重名

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。

* 避免名字中出现数字编号

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。

* 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 / remove begin / end create / destroy

insert / delete 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

### 类的命名

#### 命名规则

* 类的名称都要以大写字母“C”开头，后跟一个或多个单词。为便于界定，每个单词的首字母要大写。注：每个项目可能含有不同的类名前缀，可能会根据项目不同而进行变更。
* 类的命名推荐用"名词"或"形容词＋名词"的形式，例如："CAnalyzer"， "CFastVector"等形式。
* 纯虚抽象接口类以大写字母“I”开头。

#### 异常类

如果一个类是用来抛出异常的，那么这个类的类名应该以Exception结尾，如处理文件异常的类可以命名为：CFileException。

#### MFC子类

对于MFC的部分子类，比如CWnd、CView等基类的子类，命名时使用“C”+ 用途 +“后缀”的方式，如：CGridView，CStatusWnd等。

### 结构的命名

结构体的命名规则，与类名的命名规则一致。

### 函数的命名

#### 命名规则

* 函数的名称由一个或多个单词组成。为便于界定，每个单词的首字母要大写。例如：SetFileName，CreateService等。
* 函数名应当使用“动词”或者“动词＋名词”（动宾词组）的形式。例如：GetName(), SetValue(), Erase(), Reserve() 等。

#### Factory 函数

Factory Method用Create开始的函数名，例如：X\* CreateX() ;

#### Converter函数

Converter 函数（一个对象向别的对象转换）用To开始的函数名，例如：

std::string ToString() const ;

#### Accessor函数

Accessor 函数用于对类的成员变量赋值或者取值，这些函数一般用Get、Set开始的函数名，例如：X GetX() const, void SetX( X value )。

#### 返回bool的函数

返回bool类型的方法，采用容易明白什么是真的名字。推荐以下的组合：

* Is + 形容词

例如：bool IsEmpty() const ;

* Can + 动词

例如：bool CanSet() const ;

* Has + 过去分词

例如：bool HasChanged() const ;

* 及物动词 + 名词

例如：bool ContainsValue() const ;

对bool类型的变量也有同样的规则。

#### 回调和事件处理函数

回调和事件处理函数的函数名以单词“On”开头。例如：OnTimer()，OnExit()。

### 变量

#### 变量的命名

变量名由<**一个或多个单词>**组成，首字母小写，变量名中的其他单词的首字母要大写。

变量名推荐使用“名词”或者“形容词＋名词”的命名方式。例如：code，maxWidth等。

#### 变量的类型

不建议使用auto类型，除非类型名特别冗长，可以用来简化代码，提升可读性。

auto类型只能用在局部变量里用。

### 常量

常量名由**<全大写字母>**组成，单词间通过下划线来界定，如：

DELIMITER MAX\_BUFFER

禁止使用Magic数字，常数都应该有含义清晰的名字。

### 枚举、联合、typedef

枚举、联合、typedef语句生成的类型名，命名规则与普通类名保持一致。

### 宏、枚举值

宏和枚举值由全大写字母组成，单词间通过下划线来界定，如：ERROR\_UNKNOWN, OP\_STOP 。

## 代码风格与版式

### 缩进和对齐

#### 代码缩进

代码块要采用缩进风格编写，缩进使用空格，空格数为4个。

代码缩进不使用TAB键，由于不同的编辑器对于TAB键的默认缩进字符数是不同的，所以使用TAB键作为缩进会为代码查看者带来很多不便。

程序块的分界大括号起始符号‘{’位于其引用语句末尾，结束符‘}’应独占一行并且与引用它们的语句左对齐，这样在保证对齐清晰的同时提升一屏可阅读的代码行数。在函数体的开始、类的定义、结构的定义、枚举的定义以及if、for、do、while、switch、case语句中的程序都要采用如上的缩进方式。

#### 代码对齐

1. if语句的对齐

if (...){

//Coding

}else if (){

//Coding

}else{

//Coding

}

1. for语句的对齐

for (...){

//Coding

}

1. switch/case语句的对齐

switch (...){

case 1:{

break ;

}

default:{

break ;

}

}

1. 其他比如do/while/try/catch等语句也采用for等语句的对齐方式。

### 空行的使用

1. 空行起着分隔程序段落的作用，空行得体将使程序的布局更加清晰。
2. 相对独立的程序块之间、变量说明之后必须加空行。
3. 在每个类声明之后、每个函数定义结束之后都要加1行空行。
4. 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。

### 语句与代码行

1. 一行代码只做一件事情。

不允许把多个短语句写在一行中，即一行只写一条语句。

例如： int key = 0; int value =1 ;不符合规范，应该写成

int key = 0 ;

int nValue = 1 ;

1. if、for、do、while、case、switch、default等语句自占一行，且if、for、do、while等语句的执行语句部分无论多少都要加括号{}。

如下例子不符合规范:

if(NULL == pData) return ;

应该用如下写法：

if(NULL == pData){

return ;

}

1. 较长的语句（超过1屏）要分成多行书写

较长的表达式要在低优先级操作符处划分新行，操作符放在上一行的行尾，划分出的新行要进行适当的缩进，使排版整齐，语句可读。比如：

nValue = sizeof(TCHAR)\*strValue.size() +

sizeof(DWORD) +

sizeof(DWORD) +

sizeof(HLOCAL) +

sizeof(LPVOID) +

sizeof(long) ;

1. 循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，长表达式要在低优先级操作符处划分新行，操作符上一行的行尾。例如：

if ((stricmp(wfd.cFileName,"..") == 0) ||

(stricmp(wfd.cFileName,".") == 0) &&

(wfd.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY)){

continue ;

}

1. 函数或过程中的参数较长，则要进行适当的划分。

RegSetValueEx(hKey,

"Default\_Page\_URL",

0,

REG\_SZ,

(LPCBYTE)szStartPage,

(DWORD)strlen(szStartPage));

### 空格的使用

1. 语句中的逗号、分号只在后面加空格。

例如：for(int nIndex; nIndex < nCount; ++nIndex, ++nCounter)

1. 比较操作符, 赋值操作符"="、 "-="，算术操作符"-"、"+"，逻辑操作符"&&"、"||"，位域操作符"<<"、"^"等双目操作符的前后加空格。

例如：c = a + b ;

1. "\*"、"~"、"++"、"--"、"&"（地址运算符）等单目操作符前后不加空格。

例如：\*pszValue = 'A' ;

1. "->"、"."、"[]" 前后不加空格。

例如： pWnd->SetWindowText(szText) ;

### 修饰符的位置

为便于理解，应当将修饰符 "\*" 和 "&" 紧靠数据类型。例如：

int\* Function(void\* p, const std::string& strValue) ;

### 与常量的比较

在与宏、常量进行 "==", "!=", ">=", "<=" 等比较运算时，应当将常量写在运算符左边，而变量写在运算符右边。这样可以避免因为偶然写错把比较运算变成了赋值运算的问题。

例如：

if (NULL == pszText){ // 如果把"==" 错打成"="，编译器就会报错

// ...

}

### 类/结构

#### 类的声明

"class" 关键字由行首开始书写，后跟类名称。界定符 "{" 和 "};" 应独占一行，并与 "class" 关键字左对齐。例如：

class CStatusWnd

{

//声明代码

} ;

#### 继承类

基类的类名换行写，每个基类占一行，多个基类之间的分隔逗号放在上一行的行尾，访问说明符（public, private, 或protected）不可省略。例如：

class CStatusWnd:

public A,

private B

{

//声明代码

} ;

#### 嵌套的类声明

如果一个类中需要声明嵌套类，以一个缩进位置的方式进行声明，嵌套类的类定义方式与其他类相同。例如：

class CStatusWnd:

public A,

private B

{

public:

class C:

public D

{

//嵌套类声明代码

} ;

} ;

#### 初始化列表

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。初始化列表至少独占一行，并且与构造函数的定义保持一个缩进。例如

CStatusWnd::CStatusWnd(): //分号放在第一行行尾

m\_nValue(0), //逗号放在上一行的行尾，每个成员占一行

m\_pValue(NULL)

{

}

类成员变量的初始化列表中的顺序应该与成员在类的声明的顺序一致。

#### 类成员的声明顺序

类的类型定义，成员变量，成员函数等在类的声明中，需要按照一定的顺序声明。

class CX

{

public:

/\*\* 类型定义

\*/

typedef char CHAR ;

public:

/\*\* 构造函数

\*/

CX();

/\*\* 初始化构造函数

\*/

CX(int nValue) ;

/\*\* 拷贝构造函数

\*/

CX(const CX& rhs) ;

/\*\* 赋值操作符

\*/

CX& operator=(const CX& rhs) ;

/\*\* 析构函数

\*/

~CX();

public:

/\*\* 公用方法

\*/

protected:

/\*\* 保护方法

\*/

private:

/\*\* 私有方法

\*/

public:

/\*\* 公用成员，一般来说，类不应该有公有成员

\*/

protected:

/\*\* 保护成员

\*/

private:

/\*\* 私有成员

\*/

};

## 函数

函数的声明采用以下方式：

*[存储类] 返回值类型 [名空间或类]::函数名(参数列表) [const说明符]*

函数声明代码不分行，如果参数列表比较长，可以根据参数列表适当进行分行。

例如：

* void Function(void) ;
* const int Function(void) const ;
* static int Function(void) ;
* inline void CX::Function(void) const
* void CX::Function(int nValue,

double fValue,

const char\* pszText,

const std::string& strValue) const

## 代码注释

### 注释格式

#### 接口文件注释格式

接口文件(.h等)的注释格式采用doxygen格式。

doxygen的网址：<https://www.doxygen.nl/index.html>

doxygen命令参考网址：<https://www.doxygen.nl/manual/commands.html>

#### 实现文件的注释格式

实现文件（.cpp， .inl等）的注释格式可以不用doxygen格式的注释，但是注释风格要求统一。

### 注释要求

1. 注释统一采用中文。
2. 一般情况下，源程序有效注释量必须在20％以上。
3. 所有文件（接口文件、实现文件）都要文件头信息。
4. 所有公共接口（包括类，结构，枚举，方法，变量，常量，typedef等）都要有注释。
5. 注释与所描述内容进行同样的缩进排版。

### 接口文件注释

1. 接口文件的注释，统一使用下面的格式：

/\*\* 注释内容

\*/

注释是以“/\*\*”开始，**后面跟一个空格**，然后写注释内容。

注释以“\*/”结束，另起一行。

1. 接口文件中的注释，应与所描述代码相近，注释应放在代码上方。

无论注释内容多少，禁止把注释放在代码的右方或者下方。

1. 类/结构/枚举等的功能注释

class/struct/enum 等类型定义要有功能说明，注释的格式示例如下：

/\*\* 类的功能注释说明

\*/

class CXX

{

};

1. 类/结构/枚举等类型的成员变量注释

class/struct/enum 等类型的成员变量，包括public/protected/private的成员变量，必须有注释说明，注释的格式示例如下：

/\*\* 成员变量说明

\*/

long m\_lValue ;

1. 函数的注释

对于类的成员函数，结构的成员函数，普通函数必须有详细的注释。注释内容主要包括以下几种：

* + 函数的功能说明：必须有
  + 函数参数说明：如果没有参数，可以没有
  + 函数返回值说明：如果返回类型是void，可以没有
  + 异常说明：如果函数不抛出异常，可以没有

一个完整的函数说明示例如下：

/\*\* 将本地编码转换成UTF8编码

@param[in] pSrc 需要转换的字符串地址

@param[in] nSrcLen 需要转换的字符串长度

@param[out] pBuf 转换结果保存的缓冲区地址

@param[in] nBufLen 结果缓冲区长度

@return 返回转换结果的字符串长度

@throw CConvertException 如果转换失败，失败原因在异常信息中保存

\*/

std::size\_t ConvertLocalToUTF8(const char\* pSrc,

std::size\_t nSrcLen,

char\* pBuf,

std::size\_t nBufLen) ;

参数修饰有传入[in]、传出[out]，传入传出[in,out]三种类型。

1. 在类内部的定义的嵌套类/enum等注释方法与普通类相同
2. 在类/结构内部定义的typedef等类型定义，注释方法与类的成员变量相同。

### 实现文件注释

实现文件内部的注释目的是提高代码的可读性。

* 1. 单行注释和多行注释采用不同的方式：
     + 单行注释采用“// 注释内容”来注释。
     + 多行注释采用下面的格式来注释：

/\*\* 注释内容

\*/

* 1. 注释应与其描述的代码相近，对代码的注释应放在其上方。

无论注释内容多少，禁止把注释放在代码的右方或者下方，程序块的结束行例外。

* 1. 类/结构等的实现函数功能，参数等说明不需要写。

由于对应说明在相应接口文件中已经有注释，所以不需要重复写，以减少注释维护量。

* 1. 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。不再有用的注释要及时从代码中删除。
  2. 避免在一行代码或表达式的中间插入注释。这种注释会破坏代码的可读性。
  3. 在程序块的结束行右方加注释标记，以表明某程序块的结束。

if (...){

//这里是很长的语句

} // end of if (...), 指明是哪条if语句结束

## 代码易读性

1. 注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。
2. 避免使用不易理解的数字，用有意义的标识来替代。
3. 对于代表状态的数值等，可以采用enum，const 常量来代替。

## 质量保证

1. 不管是变量还是函数，可以声明为const的，都要声明为const。

使用const可以避免错误改变变量值而导致的意外程序错误。

1. 类成员变量控制访问基本是private，有理由时可以为protected，尽量避免public。
2. 基类的析构函数必须用virtual声明
3. 如果类不允许拷贝，必须明确把copy constructor, operator=()声明为private，并且不提供实现。或者采取其他方法禁止拷贝或者赋值。
4. 对于单参构造函数，要使用explicit关键字修饰，避免错误的隐式转换。
5. 类/结构的析构函数和赋值操作（operator=）函数中禁止抛出异常。
6. 局部变量声明时要初始化。禁止表达式中使用未经初始化的右值。
7. if或while的条件式中避免使用赋值。
8. 函数退出之前，需要释放申请的内存，关闭打开的资源句柄。从而避免内存泄露和资源泄露。
9. 避免内存操作访问越界，必要时要进行越界检查。
10. 函数的参数，对于类/结构对象优先使用引用，对于基本类型优先使用传值方式。
11. 函数的入口处，应该对参数的有效性进行检查。
12. 函数的返回值，如果是类/结构对象，可以返回引用的优先返回const引用。
13. 函数的声明避免使用异常规范，使用不抛异常的声明（throw()）例外。
14. 禁止使用C风格的 "(类型)" 格式转换。

应当优先使用C++的 "xxx\_cast" 风格的类型转换。C++风格的类型转换可以提供丰富的含义和功能，以及更好的类型检查机制，这对代码的阅读、修改、除错和移植有很大的帮助。

* static\_cast

static\_cast用于编译器认可的，安全的静态转换，比如将 "char" 转为 "int" 等。该操作在编译时完成。

* reinterpret\_cast

reinterpret\_cast用于编译器不认可的，不安全的静态转换，比如将 "int\*" 转为 "int" 等。这种转换有可能产生移植性方面的问题，不建议使用，该操作在编译时完成。

* const\_cast

const\_cast用于将一个常量转化为相应类型的变量，比如将 "const char\*" 转换成 "char\*" 等等。这种转换通常伴随潜在的错误，使用时需要注意。该操作在编译时完成。

* dynamic\_cast

dynamic\_cast是C++RTTI机制的重要体现，用于在类层次结构中漫游。dynamic\_cast可以对指针和引用进行自由度很高的向上、向下和交叉转换。被正确使用的dynamic\_cast操作将在运行时完成。dynamic\_cast在运行时有一定的开销，也不应该频繁使用。

1. 多线程的程序，在程序退出时，应主动停止所有执行线程。
2. 所有系统API调用前都要加上全局名称解析符 "::"，例如：::MessageBox(...) ;
3. 编译代码时，警告开关开到最高级，正常情况下代码不应该有编译警告信息，如果可以确认警告是正常的，应通过一些方法把警告去掉。
4. 避免函数实现过长，最好控制在200行（不包括空行和注释）以内，代码层次上避免嵌套过深。
5. 避免跨编译单元的初始化依赖。对于全局变量和静态变量，写代码的时候要注意变量的初始化顺序。
6. 优先使用++和—操作符的标准形式。优先调用前缀形式，因为前缀形式执行效率较高。
7. 禁止用==对浮点数进行相等判断。
8. 严格分层，不要在视图层中加入任何复杂的逻辑。根据 MVC 理论，视图的职责是展示，不要抢模型和控制器的活。

## 宏的使用

1. 不推荐使用宏

宏存在很多问题，比如代码可读性较差；编译好的程序比较难调试；忽略作用域；忽略类型系统；忽略所有其他的语言特性等。

所以在能够避免使用宏的地方，尽量避免使用宏。

1. 用宏定义表达式时，要使用完备的括号。

示例：下面的宏有一定的风险：

#define max(a,b) ((a) > (b)) ? (a) : (b)

#define max(a,b) a > b ? a) : b

正确的写法应该为：

#define max(a,b) (((a) > (b)) ? (a) : (b))

1. 将宏所定义的多条语句放在大括号中。

若宏的内容包含多条语句，应该放在一个大括号中。

1. 使用宏时，不允许参数发生变化。

示例：如下用法可能导致错误。

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

int a = 5 ;

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

正确的用法是：

int b = SQUARE( a );

++a ; // 结果：a = 6，即只执行了一次增。

**修订履历**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 序号 | 文件名称 | 修改章节 | 修改后版次 | 修改人 | 修改日期 | 备注 |
| 01 | 《C++语言编程规范》 | 首次编辑 | A/0 | 崔俊杰 | 2024.10.17 |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |