# C++编码规范

# 1 前言

本编码规范针对C++语言。制定本规范的目的：

提高代码的健壮性，使代码更安全、可靠；

提高代码的可读性，使代码易于查看和维护。

本文档分别对C++程序的格式、注释、标识符命名、语句使用、函数、类、程序组织、公共变量等方面做出了要求。规范分为两个级别——规则和建议。规则级的规范要求开发人员必须要遵守，建议级的规范开发人员应尽量遵守。

各项目组在用C++语言或基于C++语言的开发工具开发项目时，要遵守本规范。

# 2 编码规范正文

     以下是各条规范的具体内容。

## 2.1格式

     对代码书写格式的要求。

### 2.1.1 空行的使用

**规范级别：**规则

**规则描述：**

●在头文件和实现文件中，各主要部分之间要用空行隔开。

所谓文件的主要部分，包括：序言性注释、防止被重复包含部分（只在头文件中）、# include部分、#define部分、类型声明和定义部分、实现部分等等。

●在一个函数中，完成不同功能的部分，要用空行隔开。

**理由：**

段落分明，提高代码的可读性。

### 2.1.2 哪里应该使用空格

**规范级别：**规则

**规则描述：**

●在使用赋值运算符、逻辑运算符、位运算符、算术运算符等二元操作符时，在其两边各加一个空格。

例： nCount = 2；而不是  nCount=2；

● 函数的各参数间要用“，”和一个空格隔开。

 例：void  GetDate(int x, int y)；

而不是void  GetDate(int x,int y)或void  GetDate(int x ,int y)。

**理由：**

提高代码的可读性。

### 2.1.3 哪里不应该使用空格

**规范级别：**规则

**规则描述：**

●不要在引用操作符前后使用空格，引用操作符指“．”和“->”，以及“[]”。

●不要在“：：”前后使用空格。

●不要在一元操作符和其操作对象之间使用空格，一元操作符包括“++”、“--”“！”、“&”“\*”等。

**理由：**

提高代码的可读性。

**举例:**

// 不要象下面这样写代码：

m\_pFont -> CreateFont()；

//应该写成这样

m\_pFont->CreateFont()；

### 2.1.4 缩进

**规范级别：**规则

**规则描述：**

对程序语句要按其逻辑进行水平缩进，以两个空格为单位，使同一逻辑层次上的代码在列上对齐。

**理由：**

提高代码的可读性。

### 2.1.5 长语句的书写格式

**规范级别：**规则

**规则描述：**

较长的语句要分成多行书写。长表达式要在低优先级操作符处分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，缩进长度以两个空格或Tab符长度为单位。

**理由：**

提高代码的可读性。

**举例:**

// 下面是一个处理的较为合理的例子

nCount = Fun1(n1, n2, n3)

+ (nNumber1 \* GetDate(n4, n5, n6)) \* nNumber1;

### 2.1.6 清晰划分控制语句的语句块

**规范级别：**规则

**规则描述:**

控制语句(if , for , while , do...whule)的语句部分一定要用 ‘{ ’和‘ }’括起来(即使只有一条语句)，并且‘{ ’和‘ }’应处在同一列上。

**理由：**

这样做，能够划分出清晰的语句块，使语句的归属明确，使代码更加容易阅读和修改。

**举例:**

//不要象下面这样写代码：

if (x == 0)

return;

else

while (x > min)

x--;

// 应该这样写

if (x == 0)

{

return;

}

else

{

while (x > min)

{

x--;

}

}

### 2.1.7 一行只写一条语句

**规范级别：**规则

**规则描述：**

一行只写一条程序语句。

**理由：**

提高代码的可读性。

**举例：**

// 不要这样写

x = x0; y = y0;

while(IsOk(x)) {x++;}

// 应该这样写代码

x = x0;

y = y0;

while(IsOk(x))

{

x++;

}

### 2.1.8 一次只声明、定义一个变量

**规范级别：**规则

**规则描述：**

一次（一条声明、定义语句）只声明、定义一个变量。

理由：

提高代码的可读性。

**举例：**

// 应该这样写

int width;

int length;

// 不要这样写

int width, length;

### 2.1.9 在表达式中使用括号

**规范级别：**建议

**规则描述：**

对于一个表达式，在一个二元、三元操作符操作的操作数的两边，应该放置“（”和“）”。

**理由：**

避免出现不明确的运算、赋值顺序，提高代码的可读性。

**举例:**

// 下面这行代码：

result = fact / 100 \* number + rem;

//最好写成这样

result = ((fact / 100) \* number) + rem;

### 2.1.10将操作符“\*”、“&”和类型写在一起

**规范级别：**规则

**规则描述：**

在定义指针、引用变量时，将操作符“\*”、“&”和类型写在一起。

**理由：**

统一格式，提高代码的可读性。

**举例:**

// 不要象下面这样写代码：

 char  \*s;

       //而应该写成这样

char\*  s;

## 2.2注释

这一部分对程序注释提出了要求。

程序中的注释是程序与日后的程序读者之间通信的重要手段。良好的注释能够帮助读者理解程序，为后续阶段进行测试和维护提供明确的指导。

下面是关于注释的基本原则：

1．  注释内容要清晰明了，含义准确，防止出现二义性。

2．  边写代码边注释，修改代码的同时修改相应的注释，保证代码与注释的一致性。

### 2.2.1 对函数进行注释

**规范级别：**规则

**规则描述：**

●在函数的声明之前，要给出精练的注释（不必牵扯太多的内部细节），让使用者能够快速获得足够的信息使用函数。格式不做具体要求。

●在函数的定义之前，要给出足够的注释。注释格式要求如下：

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

【函数名称】       （必需）

【函数功能】       （必需）

【参数】           （必需。标明各参数是输入参数还是输出参数。）

【访问变量】       （必需。列出该函数访问的全局变量、成员变量。）

【返回值】         （必需。解释返回值的意义。）

【使用情况】       （必需。调用其它函数的情况，被调用的情况）

【开发者及日期】   （必需）

【版本】           （必需）

【更改记录】       （若有修改，则必需注明）

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

**理由：**

提高代码的可读性。

### 2.2.2 对类进行注释

**规范级别：**规则

**规则描述：**

● 在类的声明之前，要给出足够而精练的注释。注释格式要求如下：

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

【类名】             （必需）

【功能】            （必需）

【接口说明】           （必需）

【开发者及日期】       （必需）

【版本】              （必需）

【版权信息】           （可选）

【更改记录】          （若修改过则必需注明）

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

**理由：**

提高代码的可读性。

### 2.2.3 对文件进行注释

**规范级别：**规则

**规则描述：**

在头文件、实现文件的首部，一定要有文件注释，用来介绍文件内容。注释格式要求如下：

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

【文件名】                 （必需）

【功能模块和目的】      （必需）

【开发者及日期】           （必需）

【版本】                  （必需）

【版权信息】               （必需）

【更改记录】              （若修改过则必需注明）

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

**理由：**

提高代码的可读性。

### 2.2.4 对每个空循环体要给出确认性注释

**规范级别：**建议

**规则描述：**

 建议对每个空循环体给出确认性注释。

**理由：**

        提示自己和别人，这是空循环体，并不是忘了。

**举例：**

        while(g\_bOpen == FALSE)

        {

           //空循环

         }

### 2.2.5 对多个case语句共用一个出口的情况给出确认性注释

**规范级别：**建议

**规则描述：**

建议对多个case语句共用一个出口的情况给出确认性注释。

**理由：**

 提示自己和别人，这几个case语句确实是共用一个出口，并不是遗漏了。

**举例：**

        switch(nNumber)

        {

           case 1:

nCount++;

break;

           case 2:

           case 3:

nCount--;

break;       // 当nNumber等于2或3时，进行同样的处理

default:

break;

         }

### 2.2.6 其它应该考虑进行注释的地方

**规范级别：**建议

**规则描述：**

除上面说到的，对于以下情况，也应该考虑进行注释：

l 变量的声明、定义。通过注释，解释变量的意义、存取关系等；

    例如：int m\_nNumber;  //记录图形个数。被SetDate( )、GetDate( )使用。

l 数据结构的声明。通过注释，解释数据结构的意义、用途等；

 例如：

//定义结构体，存储元件的端点。用于将新旧的端点对应。

typedef struct

{

int nBNN;

int nENN;

int nBNO;

int nENO;

}Element;

l分支。通过注释，解释不同分支的意义；

 例如：

                if(m\_iShortRadio == 0)       //三相的情况

                {

                    strvC.Format("%-10.6f", vC);

                    straC.Format("%-10.6f", aC);

                }

                else if(m\_iShortRadio == 1)    //两相的情况

                {

                    strvC = \_T("");

                    straC = \_T("");

                    }

l调用函数。通过注释，解释调用该函数所要完成的功能；

  例如：

    SetDate(m\_nNumber );  //设置当前的图形个数。

l 赋值。通过注释，说明赋值的意义；

    例如：

          m\_bDraw = true;  //将当前设置为绘图状态

l 程序块的结束处。通过注释，标识程序块的结束。

  例如：

                           if (name = = White)

                           {

                                                …

if (age = = 20)

{

…

                                                              }//年龄判断、处理结束

…

                   }//姓名判断、处理结束

l其它有必要加以注释的地方；

理由：

提高代码的可读性。

### 2.2.7 行末注释尽量对齐

**规范级别：**建议

**规则描述：**

同一个函数或模块中的行末注释应尽量对齐。

理由：

提高代码的可读性。

**举例:**

  nCount = 0;                   //计数器，表示正在处理第几个数据块

  BOOL bNeedSave;          //是否保存从服务器返回的数据

  BOOL bReturnCache;      //是否将Cache中的内容返回客户端

    DWORD  BytesWritten;   //写入的数据长度

### 2.2.8 注释量

**规范级别：**规则

**规则描述：**

注释行的数量不得少于程序行数量的1/3。

## 2.3命名

    对标识符和文件的命名要求。

### 2.3.1 标识符命名要求

**规范级别：**规则

**规则描述：**

在程序中声明、定义的变量、常量、宏、类型、函数，在对其命名时应该遵守统一的命名规范。具体要求如下：

<!  l一般变量

一般变量名应以小写字母打头，各英文描述单词的首字母分别大写，其他字母一律小写。对于不同作用域的变量，其命名要求如下表所示：

表2-1 变量命名

|  |  |  |
| --- | --- | --- |
| **变量种类** | **前缀要求** | **示例** |
| 全局变量 | g\_ | g\_Number |
| 全局指针变量 | g\_p | g\_pNumber |
| 对象级变量（类内数据成员）  文件作用域变量（文件中静态变量） | m\_ | m\_Number |
| 对象级指针变量（类内指针数据成员）  文件作用域指针变量（文件中静态指针变量） | m\_p | m\_pNumber |
| 函数级变量（局部变量） | 无要求 | number |
| 函数级指针变量（局部指针变量） | p | pNumber |

    上表列出了对变量命名的基本要求。项目组或程序员可在该要求上再进行细化，但必须保证符合该要求。

<!  l常量

        常量的名字要全部大写。常量指：

const修饰的量。如const int NUMBER = 100；

枚举量。如enum{ ONE，TWO，THREE }；

<!  l宏

        所有用宏形式定义的名字，包括宏常量和宏函数，名字要全部大写。

<!  l类型

自定义类型名应以大写字母打头。C++中自定义类型包括：class、struct、enum、union、typedef声明的类型、namespace。

例如：typedef struct Student；

      class CMsgDialog;

<!  l函数

函数名应以大写字母打头。

例如：void GetCount()；

下面还有一些在命名时应该遵守的基本规范：

<!  l名中含多于一个单词时，每个单词的第一个字母大写。

例如：m\_LastCount 中要大写L和C；

<!  l不要使用以下划线“\_”打头的标识符。

    例如：\_bFind 是不允许出现的变量；

<!  l不要使用仅用大小写字母区分的名称。

    例如：m\_bFind 和 M\_BFIND；

<!  l尽量使用有意义的名字。应做到见其名知其意。

例如：m\_strError 表示错误的字符串；

**理由：**

减少命名冲突；提高代码的可读性。

### 2.3.2 标识符长度要求

**规范级别：**规则

**规则描述：**

在程序中声明、定义的变量、常量、宏、类型、函数，它们的名字长度要在4至25个字符之内（下限不包括前缀，上限包括名字中所有的字符）。

对于某些已经被普遍认同的简单命名，可不受本规则的限制。如for循环的循环记数变量，可使用 i 、j 等简单字符命名。

**理由：**

名字长度应该在一个恰当的范围内，名字太长不够简洁，名字太短又不能清晰表达含义。

### 2.3.3 文件命名要求

规范级别：建议

规则描述：

代码文件的名字要与文件中声明、定义的类的名字基本保持一致，使类名与类文件名建立联系。

**理由：**

使应用程序容易理解。

**举例:**

将类CMsgDialog的头文件和实现文件命名为msgdialog.h和msgdialog.cpp就是一种比较简单、恰当的方法。

## 2.4语句

    对具体程序语句的使用要求。

### 2.4.1 一条程序语句中只包含一个赋值操作符

规范级别：建议

规则描述：

在一条程序语句中，只应包含一个赋值操作符。赋值操作符包括：=, +=, -=, \*=, /=, %=, >>=, <<=, &=, |=,^=, ++, --。

**理由：**

避免产生不明确的赋值顺序。

**举例:**

// 不要这样写

b = c = 5;

a = (b \* c) + d++;

// 应该这样写

c = 5;

b = c;

a = (b \* c) + d;

d++;

### 2.4.2 不要在控制语句的条件表达式中使用赋值操作符

**规范级别：**建议

**规则描述：**

不要在控制语句if, while, for 和 switch的条件表达式中使用赋值操作符。赋值操作符包括：=, +=, -=, \*=, /=, %=, >>=, <<=, &=, |=, ^=,++,--。

**理由：**

一个类似于 if (x = y)这样的写法是不明确、不清晰的，代码的作者也许是想写成这样： if (x == y)。

**举例:**

//不要象下面这样写代码：

if (x -= dx)

{

 ...

        }

//应该这样写：

x -= dx;

if (x)

{

...

}

### 2.4.3 赋值表达式中的规定

**规范级别：**建议

**规则描述：**

在一个赋值表达式中：

• 一个左值，在表达式中应该仅被赋值一次。

• 对于多重赋值表达式，一个左值在表达式中仅应出现一次，不要重复出现。

**理由：**

避免产生不明确的赋值顺序。

**举例:**

//不要象下面这样写代码：

i = t[i++]; //一个左值，在表达式中应该仅被赋值一次

        a = b = c + a;    //对于多重赋值表达式，一个左值在表达式中仅应出现一次，不能重复出

现。

i = t[i] = 15;    //对于多重赋值表达式，一个左值在表达式中仅应出现一次，不能重复出现。

### 2.4.4 使用正规格式的布尔表达式

**规范级别：**建议

**规则描述：**

对于if, while, for等控制语句的条件表达式，建议使用正规的布尔格式。

**理由：**

使代码更容易理解。

**举例:**

//不要象下面这样写代码：

while(1)

{

...

}

if(test)

{

...

}

for(i = 1; function\_call(i); i++)

{

...

}

//最好这样写：

AlwaysTrue = true;

while(AlwaysTrue == true)

{

...

}

if(test == true)

{

...

}

for(i = 1; function\_call(i) == true; i++)

{

...

}

### 2.4.5 禁用Goto语句

**规范级别：**规则

**规则描述：**

程序中不要使用goto语句。

**理由：**

这条规则的目的是为了确保程序的结构化，因为滥用goto语句会使程序流程无规则，可读性差。

Goto语句只在一种情况下有使用价值，就是当要从多重循环深处跳转到循环之外时，效率很高，但对于一般要求的软件，没有必要费劲心思追求多么高的效率，而且效率主要是取决于算法，而不在于个别的语句技巧。

### 2.4.6 程序中禁用break、continue

**规范级别：**规则

**规则描述:**

在控制语句 (for, do, while) 块中，禁止使用Break和continue。

在switch中的case语句块不受该规则限制。

**理由：**

在控制语句 (for, do, while) 块中使用Break和continue，会打乱代码结构化的流程，使代码的可读性降低。

### 2.4.7 字符串的赋值

**规范级别：**规则

**规则描述:**

字符串的赋值应采用\_T(“”)模式。

**理由：**

改善可移植性。

**举例:**

//不要象下面这样写代码：

        Cstring strError = “syntax error”；

//应该这样写：

Cstring strError = \_T(“syntax error”)；

### 2.4.8 避免对浮点数值类型做精确比较

**规范级别：**规则

**规则描述:**

不要对浮点类型的数据做等于、不等于这些精确的比较判断，要用范围比较代替精确比较。

**理由：**

由于存在舍入的问题，计算机内部不能精确的表示所有的十进制浮点数，用等于、不等于这种精确的比较方法就可能得出与预期相反的结果。所以应该用大于、小于等范围比较的方法代替精确比较的方法。

**举例:**

//不要象下面这样写代码：

float number;

… …

        if (number = = 0)              //精确比较

… …

### 2.4.9 new 和 delete

**规范级别：**规则

**规则描述:**

局部的new 和 delete 要成对出现；

new要与delete对应，new[]要与delete[]对应。

**理由：**

防止内存泄露。

### 2.4.10对switch语句中每个分支结尾的要求

规范级别：规则

规则描述：

switch语句中的每一个case分支，都要以break作为分支的结尾（几个连续的空case语句允许共用一个）。

**理由：**

使代码更容易理解；减少代码发生错误的可能性。

### 2.4.11switch语句中的default分支

规范级别：规则

规则描述：

在switch语句块中，一定要有default分支来处理其它情况。

**理由：**

用来处理switch语句中默认、特殊的情况。

### 2.4.12对指针的初始化

**规范级别：**建议

**规则描述：**

在定义指针变量的同时，对其进行初始化。如果定义时还不能为指针变量赋予有效值，则使其指向NULL。

**理由：**

减少使用未初始化指针变量的几率。

**举例:**

// 不要这样写代码

int\* y ;

y = &x ;

...

// 应该这样写

int\* y = &x;

...

### 2.4.13释放内存后的指针变量

**规范级别：**建议

**规则描述：**

当指针变量所指的内存被释放后，应该赋予指针一个合理的值。除非该指针变量本身将要消失这种情况下不必赋值，否则应赋予NULL。

**理由：**

保证指针变量在其生命周期的全过程都指向一个合理的值。

### 2.4.14指针指向的数据成员的访问方式

规范级别：规则

规则描述：

在代码中用ptr->fld的形式代替(\*ptr).fld的形式。

## 2.5函数

对函数的要求。

### 2.5.1 明确函数功能

**规范级别：**规则

**规则描述：**

函数体代码长度不得超过100行（不包括注释）。

**理由：**

明确函数功能（一个函数仅完成一件事情），精确（而不是近似）地实现函数设计。

### 2.5.2 将重复使用的代码编写成函数

**规范级别：**建议

**规则描述：**

将重复使用的简单操作编写成函数。

**理由：**

对于重复使用的功能，虽然很简单，也应以函数的形式来处理，这样可以简化代码，使代码更易于维护。

### 2.5.3 尽量保持函数只有唯一出口

**规范级别：**规则

**规则描述：**

应该尽量保证一个函数只有一个出口。

**理由：**

增加函数的可靠性。

### 2.5.4 函数声明和定义的格式要求

**规范级别：**规则

**规则描述：**

在声明和定义函数时，在函数参数列表中为各参数指定类型和名称。

**理由：**

提高代码的可读性，改善可移植性。

**举例:**

// 不要象下面这样写代码：

f(int, char\*);        //函数声明

f(a, b)               //函数定义

int a;

char\* b

{

...

}

// 应该这样写：

f(int a, char\* b);

f(int a, char\* b)

{

...

}

### 2.5.5 为函数指定返回值

**规范级别：**规则

**规则描述：**

要为每一个函数指定它的返回值。如果函数没有返回值，则要定义返回类型为void。

**理由：**

提高代码的可读性；改善代码的可移植性。

### 2.5.6 在函数调用语句中不要使用赋值操作符

**规范级别：**建议

**规则描述：**

函数调用语句中，在函数的参数列表中不要使用赋值操作符。赋值操作符包括=, +=, -=, \*=, /=, %=, >>=, <<=, &=, |=, ^=,++,--。

**理由：**

避免产生不明确的赋值顺序。

**举例:**

// 不要象下面这样写代码：

void fun1(int a);

void fun2(int b)

{

         fun1(++b);   //注意这里!

}

### 2.5.7 保护可重入函数中的全局变量

**规范级别：**规则

**规则描述：**

编写可重入函数时，若操作全局变量，则应加以保护。

**举例:**

如果全局变量不加以保护，当多个线程调用此函数时，很可能使此变量变为不可知状态。

         例如：假设 Exam是int型全局变量，函数Squre\_Exam返回Exam平方值，如下函数不具有可重入性。

               Unsigned int example (int para)

               {

                  unsigned int temp;

<!--

                  Exam = para;

                  temp = Square\_Exam();

                 return temp;

             }

           此函数若被多个线程调用，  Exam 可能成为未知的。

可改为如下方式：

Unsigned int example (int para)

              {

                   unsigned int temp;

[申请信号量操作]

Exam = para;

temp = Square\_Exam();

                    [释放信号量操作]

return temp;

    }

## 2.6类

对类的要求。

### 2.6.1 关于默认构造函数

**规范级别：**规则

**规则描述:**

为每一个类显示定义默认构造函数。

**理由：**

确保类的编写者考虑在类对象初始化时，可能出现的各种情况。

**举例:**

class CMyClass

{

CMyClass();

...

};

### 2.6.2 关于拷贝构造函数

**规范级别：**规则

**规则描述:**

当类中包含指针类型的数据成员时，必须显示的定义拷贝构造函数。建议为每个类都显示定义拷贝构造函数。

**理由：**

确保类的编写者考虑类对象在被拷贝时可能出现的各种情况。

**举例:**

class CMyClass

{

...

CMyClass(CMyClass& object);

...

};

### 2.6.3 为类重载“=”操作符

规范级别：规则

规则描述：

当类中包含指针类型的数据成员时，必须显示重载“=”操作符。建议为每个类都显示重载“=”操作符。

**理由：**

确保类的编写者考虑将一个该类对象赋值给另一个该类的对象时，可能出现的各种情况。

**举例:**

// 应该这样写代码

class CMyClass

{

...

operator = (const CMyClass& object);

...

};

### 2.6.4 关于析构函数

**规范级别：**规则

**规则描述:**

为每一个类显示的定义析构函数。

**理由：**

确保类的编写者考虑类对象在析构时，可能出现的各种情况。

**举例:**

class CMyClass

{

...

~CMyClass (CMyClass& object);

...

};

### 2.6.5 虚拟析构函数

该规则参考自《Effective C++》中的条款 14。

**规范级别：**规则

**规则描述：**

基类的析构函数一定要为虚拟函数（virtual Destructor）。

**理由：**

保证类对象内存被释放之前，基类和派生类的析构函数都被调用。

### 2.6.6 不要重新定义继承来的非虚函数

**规范级别：**规则

**规则描述：**

在派生类中不要对基类中的非虚函数重新进行定义。如果确实需要在派生类中对该函数进行不同的定义，那么应该在基类中将该函数声明为虚函数；

**理由：**

不要忘了，当通过一个指向对象的指针调用成员函数时，最终调用哪个函数取决于指针本身的类型，而不是指针当前所指向的对象。

### 2.6.7 用内联函数代替宏函数

规范级别：建议

规则描述：

用内联函数代替宏函数。

**理由：**

同宏函数相比，内联函数不但具有宏函数的效率，而且使用起来更安全。

### 2.6.8 如果重载了操作符"new"，也应该重载操作符 "delete"

该规则参考自《Effective C++》中的条款10。

**规范级别：**规则

**规则描述：**

如果你为一个类重载了操作符new，那你也应该为这个类重载操作符delete。

**理由：**

操作符new和操作符delete需要一起合作。

### 2.6.9 类数据成员的访问控制

**规范级别：**规则

**规则描述：**

类对外的接口应该是完全功能化的，类中可以定义Public的成员函数，但不应该有Public的数据成员。

**理由：**

要想改变对象的当前状态，应该通过它的成员函数来实现，而不应该通过直接设置它的数据成员这种方法。一个类的数据成员应该声明为private的，最起码也应该是protected的。

### 2.6.10限制类继承的层数

**规范级别：**建议

**规则描述：**

当继承的层数超过5层时，问题就很严重了，需要有特别的理由和解释。

**理由：**

●很深的继承通常意味着未做通盘的考虑；

●会显著降低效率；

●可以尝试用类的组合代替过多的继承；

●与此类似，同层类的个数也不能太多，否则应该考虑是否要增加一个父类，以便做某种程度上的新的抽象，从而减少同层类的个数。

### 2.6.11慎用多继承

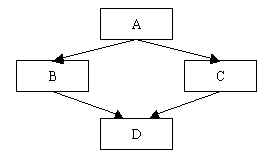
**规范级别：**建议

**规则描述：**

C++提供多继承的机制。多继承在描述某些事物时可能是非常有利的，甚至是必须的，但我们在使用多继承的时，一定要慎重，在决定使用多继承时，确实要有非常充分的理由。

**理由：**

多继承会显著增加代码的复杂性，还会带来潜在的混淆。比如在很多C++书籍中提到的菱形继承问题，如下图所示：

  
图2-1 菱形继承

A派生子类B、C，D多继承于B、C。这种情况下，会导致类D的对象中有两个类A子对象。

### 2.6.12考虑类的复用

**规范级别：**建议

**规则描述：**

类设计的同时，考虑类的可复用性。

## 2.7程序组织

对程序组织的要求。

### 2.7.1 一个头文件中只声明一个类

**规范级别：**规则

**规则描述：**

在一个头文件中，只应该包含对一个类的声明（嵌套类的情况除外）。头文件是指以.h、.hh、.H、.hxx、.hpp为后缀的文件。

**理由：**

提高代码的可读性。

### 2.7.2 一个源文件中只实现一个类

**规范级别：**规则

**规则描述:**

在一个源文件中定义的每一个函数，都应该属于同一个类，即对一个类的实现描述要独占一个文件。源文件指以\*.cc, \*.cxx, \*.cpp, \*.C or \*.c为后缀的代码文件。

**理由：**

提高代码的可读性。

### 2.7.3 头文件中只包含声明，不应包含定义

规范级别：规则

规则描述：

在头文件中只包含声明，不要包含全局变量和函数的定义。头文件指以.h、 .hh、.H、 .hxx、.hpp为后缀的代码文件。

内联函数的情况除外。

**理由：**

在头文件中只应该包含各种声明，而不应该包含具体的实现。

### 2.7.4 源文件中不要有类的声明

**规范级别：**规则

**规则描述:**

在源文件中只应该包含对类的实现，不应该包含任何类的声明。类声明应该统一放到头文件中去。源文件指以\*.cc, \*.cxx, \*.cpp, \*.C or \*.c为后缀的代码文件。

**理由：**

提高代码的可读性。

### 2.7.5 可被包含的文件

规范级别：规则

规则描述：

只允许头文件被包含到其它的代码文件中去。

**理由：**

改善程序代码的组织结构。

### 2.7.6 避免头文件的重复包含

**规范级别：**规则

**规则描述：**

头文件的格式应该类似于：

#ifndef <IDENT>

#define <IDENT>

...

...

#endif

或者

#if !defined (<IDENT>)

#define <IDENT>

...

...

#endif

上面的<IDENT>是一个标识字符串，要求该标识字符串必须唯一。建议使用该文件的答谢文件名。

**理由：**

避免对同一头文件的重复包含。

**举例:**

// 对于文件audit.h，它的文件结构应该为：

#ifndef AUDIT\_H

#define AUDIT\_H

...

...

#endif

## 2.8 公共变量

对公共变量（全局变量）的要求。

### 2.8.1 严格限制公共变量的使用

**规范级别：**建议

**规则描述：**

在程序中要尽可能少的使用公共变量。在决定使用一个公共变量时，要仔细考虑，权衡得失。

**理由：**

公共变量会增大模块间的耦合。

### 2.8.2 明确公共变量的定义

**规范级别：**规则

**规则描述：**

当你真的决定使用公共变量时，要仔细定义并明确公共变量的含义、作用、取值范围、与其它变量间的关系。明确公共变量与操作此公共变量的函数之间的关系，如访问、修改和创建等。

### 2.8.3 防止公共变量与局部变量重名

**规范级别：**规则

**规则描述：**

防止公共变量与局部变量重名。

## 2.9 其它

    下面这几条要求，不适合合并到上面任何一类，所以单独作为一部分。

### 2.9.1 不要使用结构体

规范级别：建议

规则描述：

在C++中，不要再使用struct。

**理由：**

以符合面向对象的思想。

### 2.9.2 不要使用联合体

规范级别：建议

规则描述：

在C++中，不要再使用union。

**理由：**

●由于union成员公用内存空间所以容易出错，并且维护困难；

●使用union通常意味着非面向对象的方法。

### 2.9.3 用常量代替宏

**规范级别：**规则

**规则描述：**

使用const来定义常量，代替通过宏来定义常量的方法。

**理由：**

在不损失效率的同时，使用const常量比宏更加安全。

**举例:**

//宏定义的方法

#define string "Hello world!"

#define value  3

//常量定义的方法可以代替宏，且要更好

const char\* string = "Hello world!";

const int value = 3;

### 2.9.4 括号在宏中的使用

**规范级别：**规则

**规则描述：**

对于宏的展开部分，在宏的参数出现的地方要加括号“（）”。

**理由：**

保证宏替换的安全，同时提高代码的可读性。

**举例:**

// 不要这样写

#define GET\_NAME(obj,ind) obj->name[ind]

// 应该这样写

#define GET\_NAME(obj,ind) (obj)->name[ind]

### 2.9.5 尽量使用C++风格的类型转换

该规则参考自《More Effective C++》中的条款2。

**规范级别：**建议

**规则描述：**

用C++提供的类型转换操作符（static\_cast，const\_cast， dynamic\_cast和reinterpret\_cast）代替C风格的类型转换符。

**理由：**

C风格的类型转换符有两个缺点：

1 允许你在任何类型之间进行转换，即使在这些类型之间存在着巨大的不同。

2 在程序语句中难以识别。

### 2.9.6 将不再使用的代码删掉

**规范级别：**规则

**规则描述：**

将程序中不再用到的、注释掉的代码及时清除掉。

**理由：**

理由不用做太多的解释了吧？没有用的东西就应该清理掉。如果觉得这些代码你可能以后会用到，可以备份到其它地方，而不要留在正式的版本里。

# 3 结束

以上就是我们要求C++程序遵守的规范的全部内容。