# 1 文件头注释

|  |  |
| --- | --- |
| 所有C++的源文件均必须包含一个规范的文件头，文件头包含了该文件的名称、功能概述、作者、版权和版本历史信息等内容。标准文件头的格式为：   |  | | --- | | /\*! @file \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* <PRE> 模块名       : <文件所属的模块名称> 文件名       : <文件名> 相关文件     : <与此文件相关的其它文件> 文件实现功能 : <描述该文件实现的主要功能>  作者         : <作者部门和姓名> 版本         : <当前版本号> -------------------------------------------------------------------------------- 备注         : <其它说明> -------------------------------------------------------------------------------- 修改记录 :  日 期        版本     修改人              修改内容 YYYY/MM/DD   X.Y      <作者或修改者名>    <修改内容> </PRE> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ | |

# 2 命名规则

## 2.1 整体原则

如果想要有效的管理一个稍微复杂一点的体系，针对其中事物的一套统一、带层次结构、清晰明了的命名准则就是必不可少而且非常好用的工具。

活跃在生物学、化学、军队、监狱、黑社会、恐怖组织等各个领域内的大量有识先辈们都曾经无数次地以实际行动证明了以上公理的正确性。除了上帝（设它可以改变世间万物的秩序）以外，相信没人有实力对它不屑一顾。

在软件开发这一高度抽象而且十分复杂的活动中，命名规则的重要性更显得尤为突出。一套定义良好并且完整的、在整个项目中统一使用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。

在引入细节之前，先说明一下命名规范的整体原则：

|  |  |
| --- | --- |
| 同一性 | 尽量与所采用的操作系统或开发工具的风格保持一致。例如Windows应用程序的标识符通常采用“大小写”混排的方式，如AddChild。而Unix应用程序的标识符通常采用“小写加下划线”的方式，如add\_child。别把这两类风格混在一起用。  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 | 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。 |
| 最小化长度  最大化信息量 | 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”等。 |
| 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。 |
| 正确命名具有互斥意义的标识符 | 用正确的反义词组命名具有互斥意义的标识符，如："nMinValue"和"nMaxValue"，"GetName()" 和 "SetName()" .... |
| 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。这是为了防止程序员偷懒，不肯为命名动脑筋而导致产生无意义的名字（因为用数字编号最省事）。 |
| 使用库标志 | 在开发动态库时，为了防止软件库中的一些标识符和其它软件库中标识符冲突，可以为各种标识符加上能反映软件性质的前缀。例如三维图形标准OpenGL的所有库函数均以gl开头，所有常量（或宏定义）均以GL开头。 |
| 避免魔鬼数字 | 在代码中使用魔鬼数字（没有具体含义的数字、字符串等）将会导致代码难以理解，应该将数字定义为名称有意义的常量。将数字定义为常量的最终目的是为了使代码更容易理解，所以并不是只要将数字定义为常量就不是魔鬼数字了。如果常量的名称没有意义，无法帮助理解代码，同样是一种魔鬼数字。在个别情况下，将数字定义为常量反而会导致代码更难以理解，此时就不应该强求将数字定义为常量。 |

## 2.2 自定义类型

### 2.2.1 C++类/结构体

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 除了异常类等个别情况（不希望用户把该类看作普通的、正常的类之情况）外，C++类/结构 的命名应该遵循以下准则：   |  |  | | --- | --- | | C++类/结构的命名 | 类的名称要以大写字母“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。接口类(含纯虚函数的抽象类)以大写"I"开头，代表Interface。 | | 推荐的组成形式 | 类的命名推荐用"名词"或"形容词＋名词"的形式，如："CAnalyzer",“CFastVector" | |

### 2.2.2 C结构体、联合体、枚举、typedef

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 枚举、联合及typedef语句都是定义新类型的简单手段，它们的命名规则为：   |  |  | | --- | --- | | c结构体 | 结构体名称以大写字母“S”开头，后跟一个或多个单词。每个单词首字母要大写。 | | 联合体 | 联合体名称以大写字母“U”开头，后跟一个或多个单词。每个单词首字母要大写。 | | 枚举 | 枚举名称以大写字母“E”开头，后跟一个或多个单词。每个单词的首字母要大写。 | | typedef | 枚举名称以大写字母“T”开头，后跟一个或多个单词。每个单词的首字母要大写。 | |

## 2.3 函数

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 自由函数的命名 | 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。 | | 推荐的组成形式 | 函数名应当使用"动词"或者"动词＋名词"（动宾词组）的形式。如："GetName()", "SetValue()",类成员函数应当只使用“动词”，被省略掉的名词就是对象本身。例如： " box->Draw();"。 | | 全局函数 | 以小写前缀"g"开头。 | | 保护成员函数 | 保护成员函数的开头应当加上一个下划线“\_”以示区别，如："\_SetState()" . | | 私有成员函数 | 类似地，私有成员函数的开头应当加上两个下划线“\_\_”，如："\_\_DestroyImp()" | | 虚函数 | 虚函数习惯以“Do”开头，如："DoRefresh()", "\_DoEncryption()" .... | | 回调和事件处理函数 | 回调和事件处理函数习惯以单词“On”开头。例如："\_OnTimer()", "OnExit()" .... | |

## 2.4 变量

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 变量应该是程序中使用最多的标识符了，变量的命名规范可能是一套C++命名准则中最重要的部分   |  |  | | --- | --- | | 变量的命名 | 变量名由作用域前缀+类型前缀＋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。对于某些用途简单明了的局部变量，也可以使用简化的方式，如：i, j, k, .... | | 组成形式 | 变量的名字应当使用"名词"或者"形容词＋名词"。例如："nCode", "m\_nState"。 | | 作用域前缀 | 作用域前缀标明一个变量的可见范围。作用域可以有如下几种：   |  |  |  |  | | --- | --- | --- | --- | | **前缀** | **说明** | **前缀** | **说明** | | *无* | 局部变量 | sm\_ | 结构体数据成员 | | s\_ | 静态变量（static） | um\_ | 联合体数据成员 | | g\_ | 外部全局变量（global） | em\_ | 枚举数据成员 | | sg\_ | 静态全局变量（static global） | csm\_ | 类的静态数据成员 | | gg\_ | 进程间共享的共享数据段全局变量 | ssm | 结构体的静态数据成员 | | cm\_ | 类的数据成员 | usm | 联合体的静态数据成员 |   说明：作用域前缀不同于下面的类型前缀，应该坚决执行。原因是：  1）变量作用域和链接性改变的情况是很少的，  2）编程中使用的工具常常不会直观的显示变量的作用域和链接性，  3）除非不得已，否则应该尽可能少使用全局变量。  4）作用域前缀也可以组合使用，如csm\_ 表示类的静态数据成员  5)类的数据成员与函数成员一样，公用成员以字母开头，保护成员以“\_”开头，私有成员 以“\_\_”开头 | | 类型前缀 | 类型前缀标明一个变量的类型，可以有如下几种：   |  |  |  |  | | --- | --- | --- | --- | | **前缀** | **说明** | **前缀** | **说明** | | c | Const，不单独使用 | w | WORD | | b | bool, BOOL | dw | DWORD | | ch | char， TCHAR | lpsz | LPSTR、LPCSTR、LPCTSTR(vc) | | n | int | str | string | | s | Short,不单独使用 | ar | array | | l | Long,不单独使用 | ve | vector | | u | Usinged，不单独使用 | de | deque | | f | float | li | list | | d | double | fl | forward\_list | | p | pointer | se | set | | r | Reference，不单独使用 | mse | multiset | | e | enum | ma | map | | u | union | mma | multimap | | s | Struct（c中结构体） | uo | unorder无序，不单独使用 | | c | Class/struct（c++类语义的结构体） | st | Stack | | g | 数组（grid） | qu | queue | | lp | 远指针 | fq | Priority\_queue | | pfn | 指向函数对象指针 | ap | auto\_ptr | | it | Iterator | up | unique\_ptr | | r | Reverse，不单独使用 | sp | shared\_ptr | | h | handle (windows) | wp | weak\_ptr |   类型前缀可以组合使用，如"gc"表示字符数组，"ppn"表示指向整型的二级指针等等。 | |

## 2.5 常量

|  |  |  |
| --- | --- | --- |
| C++中引入了对常量的支持，常量的命名规则如下：   |  |  | | --- | --- | | 常量的命名 | 常量名由类型前缀＋全大写字母组成，单词间通过下划线来界定，如：cDELIMITER, nMAX\_BUFFER ....，类型前缀的定义与变量命名规则中的相同。 | |

## 2.6 宏

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 宏 | 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 .... | |

# 3 代码风格与版式

代码风格的重要性怎么强调都不过分。一段稍长一点的无格式代码基本上是不可读的。

## 3.1 整体原则

### 3.1.1 空行的使用

在每个类声明之后、每个函数定义结束之后都要加2行空行。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。

### 3.1.2 语句与代码行

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。

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

### 3.1.3 缩进和对齐

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。　 "{ }" 之内的代码块在 "{" 右边一个制表符（4个半空格符）处左对齐。如果出现嵌套的 "{ }"，则使用缩进对齐。　如果一条语句会对其后的多条语句产生影响的话，应该只对该语句做半缩进（2个半角空格符），以突出该语句。

### 3.1.4 代码行最大长度

代码行最大长度宜控制在70至80个字符以内。代码行不要过长，不便于阅读和打印。

### 3.1.5 长行拆分

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。

|  |
| --- |
| if ((very\_longer\_variable1 >= very\_longer\_variable2)     && (very\_longer\_variable3 <= very\_longer\_variable4)     && (very\_longer\_variable5 <= very\_longer\_variable6)) {     dosomething(); } |

### 3.1.6 空格的使用

关键字之后要留空格。象 "const"、"virtual"、"inline"、"case" 等关键字之后至少要留一个空格，否则无法辨析关键字。象 "if"、"for"、"while"、"catch" 等关键字之后应留一个空格再跟左括号 "("，以突出关键字。

函数名之后不要留空格，紧跟左括号 "(" ，以与关键字区别。

"(" 向后紧跟。而 ")"、","、";" 向前紧跟，紧跟处不留空格。

"," 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)。

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。

一元操作符如 "!"、"~"、"++"、"--"、"&"（地址运算符）等前后不加空格。

象"[]"、"."、"->",”::”、“\*”、“&”这类访问操作符前后不加空格。但继承权限说明符”:” 前后要空格。

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，如for (i=0; i<10; i++)和if ((a<=b) && (c<=d))

例如：

|  |
| --- |
| 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 for (i = 0; I < 10; i ++) // 过多的空格  // =========================================================== x = a < b ? a : b;        // 良好的风格 x=a<b?a:b;                // 不好的风格  // =========================================================== 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(); |

### 3.1.7 修饰符的位置

为便于理解，应当将修饰符 "\*" 和 "&" 紧靠数据类型，但是要注意分行，也可以紧靠变量名称。一个程序应当只统一使用一种。

例如：

|  |
| --- |
| char\* name; int\* x,      y;    // 为避免y被误解为指针，这里必须分行写。 int\* Function(void\* p); |

参见：[变量、常量的风格与版式](#代码风格与版式_变量、常量) -> 指针或引用类型的定义和声明

### 3.1.8 注释

注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。

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。

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。

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。

### 3.1.9 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 == p)  // 如果把 "==" 错打成 "="，编译器就会报错 {     // ... } |

### 3.1.10 为增强代码的可读性而定义的宏

以下预定义宏对程序的编译没有任何影响，只为了增加代码的可读性：

|  |  |
| --- | --- |
| 宏 | 说明 |
| NOTE | 需要注意的代码 |
| TODO | 尚未实现的接口、类、算法等 |
| FOR\_DBG | 标记为调试方便而临时增加的代码 |
| OK | 仅用于调试的标记 |

例如：

|  |
| --- |
| TODO class CMyClass; TODO void Function(void);  FOR\_DBG cout << "..."; |

## 3.2 类/结构体

类是C++中最重要也是使用频率最高特性之一。类的版式好坏将极大地影响代码品质。

### 3.2.1 注释头与类声明

与文件一样，每个类应当有一个注释头用来说明该类的各个方面。

类声明换行紧跟在注释头后面，"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。

|  |
| --- |
| /\*! @class \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* <PRE> 类名称   : CXXX 功能     : <简要说明该类所完成的功能> 异常类   : <属于该类的异常类（如果有的话）>  -------------------------------------------------------------------------------- 备注     : <使用该类时需要注意的问题（如果有的话）>  典型用法 : <如果该类的使用方法较复杂或特殊，给出典型的代码例子>  -------------------------------------------------------------------------------- 作者     : <xxx> </PRE> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ class CXXX {     // ... }; |

对于功能明显的简单类（接口小于10个），也可以使用简单的单行注释头：

|  |
| --- |
| //! <简要说明该类所完成的功能> class CXXX {     // ... }; |

### 3.2.2 继承

基类直接跟在类名称之后，不换行，继承方式（public, private, 或protected）不可省略。

|  |
| --- |
| class CXXX : public CAAA, private CBBB {     // ... }; |

### 3.2.3 以行为为中心

没人喜欢上来就看到一大堆私有数据，大多数用户关心的是类的接口与其提供的服务，而不是其实现。

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，而私有的摆在最后。

### 3.2.4 访问权限

访问权限（public, private, 或protected）应该独占一行，并与类声明中的‘class’关键字左对齐。

### 3.2.5 类成员的声明版式

对于比较复杂（成员多于20个）的类，其成员必须分类声明。

每类成员的声明由访问说明符（public, private, 或protected）+ 全行注释开始。注释不满全行（80个半角字符）的，由 "/" 字符补齐，最后一个 "/" 字符与注释间要留一个半角空格符。

如果一类声明中有很多组功能不同的成员，还应该用[分组注释](file:///F:\王道训练营\常用注释一览.htm#语句/函数组)将其分组。分组注释也要与 "class" 关键字对其。

每个成员的声明都应该由 "class" 关键字开始向右缩进一个制表符（4个半角空格符），成员之间左对其。

例如：

|  |
| --- |
| class CXXX { 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 int n); // ]] 功能组1  // [[ 功能组2     void Function3(void) const;     bool Function4(OUT int& n); // ]] 功能组2  private: /////////////////////////////////////////////////////////////////////////// 属性     // ...  private: ///////////////////////////////////////////////////////////////////// 禁用的方法     // 禁止复制     CXXX(IN const CXXX& rhs);     CXXX& operator=(IN const CXXX& rhs); }; |

### 3.2.6 使用const和mutable

把不改变对象逻辑状态的成员都标记为const成员不仅有利于用户对成员的理解，更可以最大化对象使用方式的灵活性及合理性（比如通过const指针或const引用的形式传递一个对象）。

如果某个属性的改变并不影响该对象逻辑上的状态，而且这个属性需要在const方法中被改变，则该属性应该标记为 "mutable"。

例如：

|  |
| --- |
| class CString { public:     //! 查找一个子串，find()不会改变字符串的值所以为const函数     int find(IN const CString& str) const;     // ...  private:     // 最后一次错误值，改动这个值不会影响对象的逻辑状态，     // 像find()这样的const函数也可能修改这个值     mutable int m\_nLastError;      // ... }; |

### 3.2.7 嵌套的类声明

在相应的逻辑关系确实存在时，类声明可以嵌套。嵌套类可以使用简单的单行注释头：

|  |
| --- |
| class CXXX {     //! 嵌套类说明     calss CYYY     {         // ...  }; }; |

### 3.2.8 初始化列表

初始化列表应该写在类外实现时，而不是类内声明时，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制表符（4个半角空格）的缩进。

例如：

|  |
| --- |
| CXXX::CXXXX(IN int nA, IN bool bB)  : m\_nA(nA), m\_bB(bB)  { // ...  }; |

　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。

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）。

### 3.2.9 内联函数的实现体

定义在类声明之中的函数将自动成为内联函数。但为了使类的声明更为清晰明了，应尽量避免直接在声明中直接定义成员函数的编程风格。鼓励使用 "inline" 关键字将内联函数放在类声明的外部定义。

Inline应在类外实现时，而不是类内声明时，如果还有条件、开关、循环语句则不要使用inline，编译器会自动忽略。

## 3.3 函数

函数是程序执行的最小单位，任何一个有效的C/C++程序都少不了函数。

### 3.3.1 函数原型

函数原型的格式为：

|  |
| --- |
| [存储类型] 返回值类型 [名称空间或类]::函数名(参数列表) [const说明符] [异常过滤器] |

例如：

|  |
| --- |
| static inline void Function1(void)  int CSem::Function2(IN const char\* pcName) const throw(Exp) |

其中：

以 "[ ]" 括住的为可选项目。

除了构造/析构函数外，"返回值类型" 和 "参数列表" 项不可省略（可以为 "void" ）。

"const说明符" 仅用于成员函数中

"存储类", "参数列表" 和 "异常过滤器" 的说明见下文。

### 3.3.2 函数声明

函数声明的格式为：

|  |
| --- |
| //! 函数功能简单说明（可选） 函数原型; |

例如：

|  |
| --- |
| //! 执行某某操作 static void Function(void); |

函数声明和其它代码间要有空行分割。

声明成员函数时，返回值类型和函数名之间不用换行，也可以适当减少声明间的空行。

### 3.3.3 函数定义

函数定义使用如下格式：

|  |
| --- |
| /\*! @function \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* <PRE> 函数名   : <函数名> 功能     : <函数实现功能>  参数     : <参数类表及说明（如果有的话），格式为：>             [IN|OUT] 参数1 : 参数说明            [IN|OUT] 参数2 : 参数说明            ... 返回值   : <函数返回值的意义（如果有的话）> 抛出异常 : <可能抛出的异常及其说明（如果有的话），格式为：>            类型1 : 说明            类型2 : 说明            ... -------------------------------------------------------------------------------- 备注     : <其它注意事项（如果有的话）>  典型用法 : <如果该函数的使用方法较复杂或特殊，给出典型的代码例子>  -------------------------------------------------------------------------------- 作者     : <xxx> </PRE> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 函数原型 {     // ... } |

对于返回值、参数意义都很明确简单函数（代码不超过20行），也可以使用单行函数头：

|  |
| --- |
| //! 函数实现功能 函数原型 {     // ... } |

函数定义和其它代码之间至少分开2行空行。

### 3.3.4 参数描述宏

以下预定义宏对程序的编译没有任何影响，只为了增强对参数的理解：

|  |  |
| --- | --- |
| 宏 | 说明 |
| IN | 输入参数 |
| OUT | 输出参数 |
| OPTIONAL | 可选参数－通常指可以为NULL的指针参数，带默认值的参数不需要这样标明 |
| RESERVED | 这个参数当前未被支持，留待以后扩展 |
| OWNER | 获得参数的所有权，调用者不再负责销毁参数指定的对象 |
| UNUSED | 标明这个参数在此版本中已不再使用 |
| CHANGED | 参数类型发出变化 |
| ADDED | 新增的参数 |
| NOTE | 需要注意的参数－参数意义发生变化 |

其中：

除了空参数 "void" 以外，每个参数左侧都必须有 "IN" 和/或 "OUT" 修饰

既输入又输出的参数应记为："IN OUT"，而不是 "OUT IN"

IN/OUT的左侧还可以根据需要加入一个或多个上表中列出的其它宏

参数描述宏的使用思想是：只要一个宏可以用在指定参数上（即：对这个参数来说，用这个描述宏修饰它是贴切的），那么就应当使用它。

也就是说，应该把能用的描述宏都用上，以期尽量具体地描述一个参数。

### 3.3.5 参数列表

参数列表的格式为：

|  |
| --- |
| 参数描述宏1 参数类型1 参数1, 参数描述宏2 参数类型2 参数2, ... |

例如：

|  |
| --- |
| IN const int nCode, OUT string& nName  OWNER IN CDatabase\* pDB, OPTIONAL IN OUT int\* pRecordCount = NULL  IN OUT string& stRuleList, RESERVED IN int nOperate = 0  ... |

其中：

["参数描述宏" 见上文](#_3.3.4_参数描述宏)

参数命名规范与[变量的命名规范](#_2.4_变量)相同

### 3.3.6 无参函数

对于明确知道无参的函数，要表明void。

### 3.3.7 存储类型

"extern", "static", “explicit”等函数存储类型说明应该在声明和定义中一致并且显式地使用。不允许隐式地使用一个类型声明，也不允许一个存储类型仅存在于函数的声明或定义中。

### 3.3.8 成员函数的存储类型

由于C++语言的限制，成员函数的 "static", "virtual", "explicit" 等存储类型说明不允许出现在类外的函数定义中。 但是为了明确起见，这些存储类型应以注释的形式在定义中给出。

成员函数在类内定义或声明不需加inline，在类外定义时要加inline。

例如：

|  |
| --- |
| /\*virtual\*/ CThread::EXITCODE CSrvCtl::CWrkTrd::Entry(void) {     // ... }  /\*static\*/ inline void  stringEx::regex\_free(IN OUT void\*& pRegEx) {     // ... } |

特别地，为缩短声明的长度，"inline" 关键字可以在成员函数声明中省略。

### 3.3.9 默认参数

类似地，参数的默认值只能出现在函数声明中，但是为了明确起见，这些默认值应以注释的形式在定义中给出。

例如：

|  |
| --- |
| bool  stringEx::regex\_find(OUT VREGEXRESULT& vResult,  IN stringEx stRegEx，                                    IN size\_t nIndex      /\*= 0\*/,   IN size\_t nStartPos   /\*= 0\*/,                                    IN bool bNoCase       /\*= false\*/,                       IN bool bNewLine      /\*= true\*/,                       IN bool bExtended     /\*= true\*/,                       IN bool bNotBOL       /\*= false\*/,                       IN bool bNotEOL       /\*= false\*/,                       IN bool bUsePerlStyle /\*= false\*/) const  {     // ... } |

### 3.3.10 异常过滤器

对于任何肯能抛出异常的函数，必须在其声明和定义中显式地指定异常过滤器，并在过滤器中列举该函数可能抛出的异常。

例如：

|  |
| --- |
| int Function(IN const char\* pcName) throw(byExp, exception); |

### 3.3.11 代码注释段

如果函数体中的代码较长，应该根据功能不同将其分段。代码段间以空行分离，并且每段代码都以代码段分割注释作为开始。

例如：

|  |
| --- |
| void CXXX::Function(IN void\* pmodAddr) {     if (NULL == pmodAddr)         return;      {  CSessionLock iLock(\*sm\_hSELock);          // =====================================================================         // = 判断指定模块是不是刚刚被装入，由于在NT系列平台中，“A”系列函数都是         // = 由“W”系列函数实现的。所以可能会有一次LoadLibrary产生多次本函数调         // = 用的情况。为了增加效率，特设此静态变量判断上次调用是否与本次相同。         static PVOID pLastLoadedModule = NULL;         if (pLastLoadedModule == pmodAddr)         {             return;  // 相同，忽略这次调用         }         pLastLoadedModule = pmodAddr;          // =====================================================================         // = 检查这个模块是否在旁路模块表中         stringEx stModName;         if (!BaiY\_IMP::GetModuleNameByAddress(pmodAddr, stModName))         {             return;         }                  if (CHookProc::sm\_sstByPassModTbl.find(stModName)             != CHookProc::sm\_sstByPassModTbl.end())         {             return;         }          // =====================================================================         // = 在这个模块中HOOK所有存在于HOOK函数表中的函数         PROCTBL::iterator p;         for (p=sm\_iProcTbl.begin(); p!=sm\_iProcTbl.end(); ++p)         {             p->HookOneModule(pmodAddr);         }     } // SessionLock } |

明显地，如果需要反复用到一段代码的话，这段代码就应当作为一个函数实现。

当一个函数过长时（超过100行），为了便于阅读和理解，也应当将其中的一些代码段实现为单独的函数。

### 3.3.12 调用系统API

所有系统API或者全局函数调用前都要加上全局名称解析符 "::"。

例如：

|  |
| --- |
| ::MessageBoxA(NULL, gcErrorMsg, "!FATAL ERROR!", MB\_ICONSTOP|MB\_OK);  if (0 == ::GetTempFileName(m\_basedir.c\_str(), byT("bai"), 0, stR.ref())) {     // ... } |