# C++编码规范与指导

版本：1.31

作者：白杨

http://baiy.cn

推荐浏览设置：

- 屏幕分辨率：≥ 1024x768
- 字体：中（Ctrl＋鼠标滚轮设置）
- 最大化本窗口

## 文档控制

| 版本号 | 修改时间 | 修改内容 | 修改人 | 审稿人 |
|---|---|---|---|---|
| 1.0 | 2004-07-22 | • 创建 | 白杨 | 田振军 |
| 1.1 | 2004-08-05 | • 根据审稿意见修改 | 白杨 | 田振军、马浩军、叶晓峰 |
| 1.2 | 2004-08-09 | • 根据审稿意见修改<br>• 新增RTTI、虚函数和虚基类的开销分析及使用指导 | 白杨 | 田振军、马浩军、叶晓峰 |
| 1.3 | 2004-08-10 | • 重写目录；一些小改动 | 白杨 | |
| 1.4 | 2004-08-10 | • 新增C++成长篇 :-) | 白杨 | 广大CSDN上的网友，鸣谢 :-) |
| 1.5 | 2004-08-28 | • 根据网友审稿意见修改<br>• 在函数中增加“让相同的代码只出现一次”的条目<br>• 在函数头中增加复杂性描述 | 白杨 | |
| 1.6 | 2004-11-22 | • 修正了一些笔误 | 白杨 | |
| 1.7 | 2005-03-30 | • 新增成员函数的下划线后缀命名 | 白杨 | |
| 1.8 | 2005-05-04 | • 小改动，修正了一些笔误；在类命名规范中增加了界面类的概念 | 白杨 | |
| 1.9 | 2005-05-11 | • 再次调整类命名规范，引入界面、类型和类的概念 | 白杨 | |
| 1.10 | 2005-06-06 | • 新增修改标记规则 | 白杨 | |
| 1.11 | 2005-06-07 | • 新增类型和界面的使用策略 | 白杨 | |
| 1.12 | 2005-06-28 | • 修正一些笔误 | 白杨 | |
| 1.13 | 2005-08-20 | • 小幅修正和调整 | 白杨 | |
| 1.14 | 2005-11-08 | • 对全文再次进行修正及调整 | 白杨 | |
| 1.15 | 2005-11-14 | • 增加数值前缀的特别记法 | 白杨 | |
| 1.16 | 2005-11-16 | • 增加对复杂的宏实行缩进 | 白杨 | |
| 1.17 | 2005-11-21 | • 补全修改标记规则 | 白杨 | |

| | | | | |
|---|---|---|---|---|
| | | • 在常用注释中增加一种新型组注释 | | |
| 1.18 | 2005-12-02 | • 在文件头中增加多线程和异常时安全性描述 | 白杨 | |
| 1.19 | 2005-12-25 | • 细化异常过滤器规则<br>• 增加特别代码段注释规则 | 白杨 | CCF上的网友，特别感谢smartsl |
| 1.20 | 2006-04-03 | • 根据网友意见修正一些笔误 | 白杨 | |
| 1.21 | 2007-02-26 | • 新增 DUMMY 表意宏 | 白杨 | |
| 1.22 | 2007-07-18 | • **新增 C++异常机制的实现方式和开销分析** | 白杨 | |
| 1.23 | 2007-11-22 | • 新增私有成员函数的层次结构表示 | 白杨 | |
| 1.24 | 2008-01-17 | • 措辞和表达上的细微调整 | 白杨 | CCF 上的 Jiang Haibin |
| 1.25 | 2008-03-12 | • 根据审稿人建议修正一些错别字，感谢 Haibin 兄 :) | 白杨 | |
| 1.26 | 2008-06-26 | • 新增“针对C程序员的快速回顾”一节 | 白杨 | |
| 1.27 | 2008-07-27 | • 修正一些错别字 | 白杨 | |
| 1.28 | 2009-02-04 | • 更新和一些小修正 | 白杨 | |
| 1.29 | 2009-03-31 | • 新增函数对象的命名规则<br>• 新增 OWNER 表意宏对返回值的修饰 | 白杨 | |
| 1.30 | 2009-04-17 | • 新增 WRKBUF 表意宏<br>• 新增 new / delete 时的异常处理 | 白杨 | |
| 1.31 | 2009-05-07 | • **新增 多处理器环境和线程同步的高级话题** | 白杨 | |
| | | | | |

---

# 目录


- 版权声明
- 概述
- 针对 C 程序员的快速回顾
- 语法高亮与字体
  - 字体
  - 语法高亮
- 文件结构
  - 文件头注释
  - 头文件
  - 内联函数定义文件

- 实现文件
- 文件的组织结构

- 命名规则
  - 类/结构
  - 函数
  - 变量
  - 常量
  - 枚举、联合、typedef
  - 宏、枚举值
  - 名空间
- 代码风格与版式
  - 类/结构
  - 函数
  - 变量、常量
  - 枚举、联合、typedef
  - 宏
  - 名空间
  - 异常
  - 修改标记
- 版本控制
- 自动工具与文档生成
- 英文版
- 关于本规范的贯彻实施
- 术语表
- 参考文献
- C++成长篇
- 与我联系

附件


- 常用注释一览
- 常用英文注释一览
- 文件头例子
- 头文件例子
- 实现文件例
- 内联函数定义文件例子
- 类/结构的风格与版式例子
- 函数的风格与版式例子
- RTTI、虚函数和虚基类的开销分析和使用指导
- C++异常机制的实现方式和开销分析
- 多处理器环境和线程同步的高级话题


# 版权声明

本文档版权归作者所有。您可以以任意形式免费使用本文档的任意部分，并且无需通知作者。作者对使用本文档所造成的任何直接或者间接的损失不负任何责任。

# 概述

对于任何工程项目来说，统一的施工标准都是保证工程质量的重要因素。堪称当今人类最抽象、最复杂的工程——软件工程，自然更加不能例外。

高品质、易维护的软件开发离不开清晰严格的编码规范。本文档详细描述C++软件开发过程中的编码规范。本规范也适用于所有在文档中出现的源码。

除了“语法高亮”部分，本文档中的编码规范都以：

| 规则（或建议） | 解释 |
|---|---|

的格式给出，其中强制性规则使用黑色，建议性规则使用灰色。

# 针对 C 程序员的快速回顾

本节旨在较高层面上快速回顾 C 与 C++ 的主要区别。专门针对 C 思想根深蒂固的老咖和经常需要在 C / C++ 项目间频繁切换的 coder。C 与 C++ 的主要区别包括：

- **空参函数声明**的默认参数类型为 void 而不是 int [编译时]。
- **强类型检查和**专门的类型转换操作 [编译时，除 dynamic_cast 操作]。
- **名空间**：用于归类接口和模块以及防止重名。名空间有自动向父级查询匹配和凯氏匹配。名空间的一个副作用是名称粉碎，可以用 extern "C" 声明解决 [编译时]。
- **类**：类将一个 C 结构体和与之相关的函数打包在一起，并且提供了编译时的访问控制检查，为了拟真内置类型，还提供了操作符重载 [编译时]。
- **类层次结构**：类可以通过相互间的继承和派生形成层次结构，派生类继承了基类的数据结构和方法 [编译时]。
- **模板**：本质上是类型参数化。在编译时生成 C++ 代码的过程叫做模板实例化，默认的实例化点在当前编译单元第一次使用该模板时，也可以通过显式实例化来增加编译速度。通过部分或完全的专门化可以为指定类型提供优化算法 [编译时]。
- 抽象类和虚方法：可以通过基类指针 或引用访问的动态重载技术 [运行时]。
- 多重继承和虚基类：一个类可以有多个父亲，为了避免层次结构中出现重复的基类，C++ 提供了虚基类 [运行时]。
- 运行时类型信息（RTTI）：允许程序员在类层次结构中漫游；完成动态转换（向上、向下和交叉强制）；获取指定类型的 typeid 信息，用户可以以此为基础实现反射、高级调试等各类功能。
- 异常处理：本质上是一种安全的 longjmp 机制，在退栈期间能够完成必要的对象析构动作。主要用于在发生错误时跳转到相应的错误处理分支。

# 语法高亮与字体

## 字体

文字是信息的载体；文字使我们能够把个人的经验和思想长久的保存下来；文字使我们得以站在前人的肩膀上向前发展；文字的诞生标志着人类文明的开始……

扯的太离谱了？好吧，至少你应该承认：

- 没有文字就不可能出现计算机（先不管他是哪国字 ）
- 没有文字大家就不可能（也没必要）学会如何写程序
- 在过去、现在和可见的将来，使用文字符号都是编写计算机软件的主要方式方法

既然文字如此重要，它的长相自然会受到广泛的关注。如今这个连MM都可以“千面”的年头，字体的种类当然是数不胜数。

然而，前辈先贤们曾经用篆体教导偶们： 。想让大家读到缩进、对齐正确一致，而且不出现乱码的源文件，我们就要使用相互兼容的字体。

字体规范如下：

| | |
|---|---|
| 使用等宽字体 | 由于非等宽字体在对齐等方面问题多多，任何情况下，源码都必须使用等宽字体编辑和显示。 |

| | |
|---|---|
| 每个制表符（TAB）的宽度为4个半角字符 | 不一致的缩进宽度会导致行与行之间的参差不齐，进而严重影响代码的可读性。 |
| 优先使用Fixedsys | 在Windows平台中，应该优先使用字体：**Fixedsys**，这也是操作系统UI（所有的菜单、按钮、标题栏、对话框等等）默认使用的字体。该字体的好处很多：<br>• 兼容性好：所有Windows平台都支持该字体<br>• 显示清晰：该字体为点阵字体，相对于矢量字体来说在显示器中呈现的影像更为清晰。矢量字体虽然可以自由缩放，但这个功能对于纯文本格式的程序源码来说没有任何实际作用。<br>而且当显示字号较小（12pt以下）时，矢量字体还有一些明显的缺陷：<br>○ 文字的边缘会有严重的凹凸感。<br>○ 一些笔画的比例也会失调。<br>○ 开启了柔化字体边缘后，还会使文字显得模糊不清。<br>说句题外话，这也是Gnome和KDE等其它GUI环境不如Windows的一个重要方面 （2009年更新：现如今这些环境的界面和字体也不比 Windows 差了）。<br>• 支持多语言：Fixedsys可以兼容其它UNICODE等宽字体，所以支持世界上几乎所有的文字符号。这对编写中文注释是很方便的。 |

## 语法高亮

几乎所有的现代源码编辑器均不同在程度上支持语法高亮显示的功能。缤纷的色彩不但可以吸引MM们的目光，还可以在很大程度上帮助我们阅读那些奥涩如咒语般的源代码。

统一的语法高亮规则不仅能让我们望色生意，还可以帮助我们阅读没有编码规范，或者规范执行很烂的源码。

所有在文档中出现的代码段均必须严格符合下表定义的语法高亮规范。在编辑源码时，应该根据编辑器支持的自定义选项最大限度地满足下表定义的高亮规范。

| 类型 | 颜色 | 举例 |
|---|---|---|
| 注释 | R0;G128;B0（深绿） | // 注释例子 |
| 关键字 | R0;G0;B255（蓝） | typedef, int, dynamic_cast class ... |
| 类、结构、联合、枚举等其它自定义类型 | R0;G0;B255（蓝） | class CMyClass, enum ERRTYPE, typedef int CODE ... |
| 名空间 | R0;G0;B255（蓝） | namespace BaiY |
| 数字 | R255;G0;B0（红） | 012 119u 0xff ... |
| | | |

| | | |
|---|---|---|
| 字符、字符串 | R0;G128;B128（深蓝绿） | "string", 'c ... |
| 宏定义、枚举值 | R255;G128;B0（橙黄） | #define UNICODE, enum { RED, GREEN, BLUE }; |
| 操作符 | R136;G0;B0（棕色） | < > , = + - * / ; { } ( ) [ ] ... |
| 方法/函数 | R136;G0;B0（棕色） | MyFunc() |
| 变量 | R128;G128;B128（中灰色） | int nMyVar; |
| 背景 | R255;G255;B255（白色） | |
| 其它 | R0;G0;B0（黑色） | other things（通常是一个错误） |

# 文件结构

## 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>
-------------------------------------------------------------------------------
多线程安全性 : <是/否>[, 说明]
异常时安全性 : <是/否>[, 说明]
-------------------------------------------------------------------------------
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>
*******************************************************************************/
```

如果该文件有其它需要说明的地方，还可以专门为此扩展一节 ，节与节之间用长度为80的“=”带分割：

```
/*! @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>
```

```
--------------------------------------------------------------------------------
多线程安全性 :<是/否>[, 说明]
异常时安全性 :<是/否>[, 说明]
--------------------------------------------------------------------------------
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>
*******************************************************************************

* 项目1
  - 项目1.1
  - 项目1.2

===============================================================================
* 项目2
  - 项目2.1
  - 项目2.2
....

*******************************************************************************/
```

每行注释的长度都不应该超过80个半角字符。还要注意缩进和对齐，以利阅读。

**注意**：将多线程和异常时安全性描述放在文件头，而不是类或者函数注释中，是为了体现以下设计思想：同一个模块中的界面，其各方面的操作方式和使用风格应该尽量保持一致。

关于文件头的完整例子，请参见：文件头例子

关于文件头的模板，请参见：文件头注释模板

## 头文件

头文件通常由以下几部分组成：

| | |
|---|---|
| 文件头注释 | 每个头文件，无论是内部的还是外部的，都应该由一个规范的文件头注释作为开始。 |
| 预处理块 | 为了防止头文件被重复引用，应当用ifndef/define/endif结构产生预处理块。 |
| 函数和类/结构的声明等 | 声明模块的接口 |
| 需要包含的内联函数定义文件（如果有的话） | 如果类中的内联函数较多，或者一个头文件中包含多个类的定义（不推荐），可以将所有内联函数定义放入一个单独的内联函数定义文件中，并在类声明之后用“#include”指令把它包含进来。 |

头文件的编码规则：

| | |
|---|---|
| 引用文件的格式 | 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）。 |
| 分割多组接口（如果有的话） | 如果在一个头件中定义了多个类或者多组接口（不推荐），为了便于浏览，应该在每个类/每组接口间使用分割带把它们相互分开。 |

关于头文件的完整例子，请参见：头文件例子

## 内联函数定义文件

如上所述，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，然后再用#include指令将它包含到类声明的后面。这样的文件称为一个内联函数定义文件。

按照惯例，应该将这个文件命名为“*filename*.inl”，其中“filename”与相应的头文件和实现文件相同。

内联函数定义文件由以下几部分组成：

| | |
|---|---|
| 文件头注释 | 每内联函数定义文件都应该由一个规范的文件头注释作为开始 |
| 内联函数定义 | 内联函数的实现体 |

内联函数定义文件的编码规则：

| | |
|---|---|
| 分割多组接口（如果有的话） | 如果在一个内联函数定义文件中定义了多个类或者多组接口的内联函数（不推荐），必须在每个类/每组接口间使用分割带把它们相互分开。 |
| 文件组成中为什么没有预处理块？ | 与头文件不同，内联函数定义文件通常不需要定义预处理块，这是因为他们总是被包含在与其相应的头文件预处理块内。 |

关于内联函数定义文件的完整例子，请参见：内联函数定义文件例子

## 实现文件

实现文件包含所有数据和代码的实现体。实现文件的格式为：

| | |
|---|---|
| 文件头注释 | 每个实现文件都应该由一个规范的文件头注释作为开始 |
| 对配套头文件的引用 | 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（如果有的话） | 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 。 |

| | |
|---|---|
| 程序的实现体 | 数据和函数的定义 |

实现文件的编码规则：

| | |
|---|---|
| 分割每个部分 | 在本地（静态）定义和外部定义间，以及不同接口或不同类的实现之间，应使用分割带相互分开。 |

关于实现文件的完整例子，请参见：实现文件例子

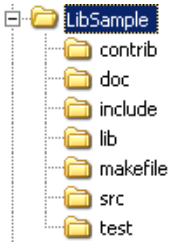
## 文件的组织结构

由于项目性质、规模上存在着差异，不同项目间的文件组织形式差别很大。但文件、目录组织的基本原则应当是一致的：使外部接口与内部实现尽量分离；尽可能清晰地表达软件的层次结构……

为此提供两组典型项目的文件组织结构范例作为参考：

### 功能模块/库的文件组织形式

显而易见，编写功能模块和库的主要目的是为其它模块提供一套完成特定功能的API，这类项目的文件组织结构通常如下图所示：




其中：

| | |
|---|---|
| contrib | 当前项目所依赖的所有第三方软件，可以按类别分设子目录。 |
| doc | 项目文档 |
| include | 声明外部接口的所有头文件和内联定义文件。 |
| lib | 编译好的二进制库文件，可以按编译器、平台分设子目录。 |
| makefile | 用于编译项目的makefile文件和project文件等。可以按编译器、平台分设子目录。 |
| src | 所有实现文件和声明内部接口的头文件、内联定义文件。可按功能划分；支持编译器、平台等类别分设子目录。 |
| test | 存放测试用代码的目录。 |

### 应用程序的文件组织形式

与功能模块不同，应用程序是一个交付给最终用于使用的、可以独立运行并提供完整功能的软件产品，它通常不提供编程接口，应用程序的典型文件组织形式如下图所示：

| | |
|---|---|
| contrib | 当前项目所依赖的所有第三方软件，可以按类别分设子目录。 |
| doc | 项目文档 |
| makefile | 用于编译项目的makefile文件和project文件等。可以按编译器、平台分设子目录。 |
| setup | 安装程序，以及制作安装程序所需要的项目文件和角本。 |
| src | 所有源文件。可按功能划分；支持编译器、平台等类别分设子目录。 |
| test | 存放测试用代码的目录。 |

# 命名规则

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 | 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”等等。 |
| 避免在不同级别的作用域 | 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 |

| | |
|---|---|
| 中重名 | 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）。 |

## 类/结构

除了异常类等个别情况（不希望被用户看作一个普通的、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。<br><br>特别地，由于**界面**与其它类概念上的巨大差别，规定界面类要以大写字母“I”开头。界面类描述一个服务（一组被命名的操作集合），在C＋＋中，界面与其它类间的最大区别在于，界面类中不包含任何数据结构（属性），也不包括任何具体的操作和实现，界面类通常仅包含一组纯虚函数的声明而不包含任何实现和数据。在一些其它语言中，一个界面也被称作一个接口及其实现契约。<br><br>另一个与接口相似的概念是**类型**，类型与接口的不同点在于，类型可以包含部分接口的实现或包含一些接口默认的或不完整的实现，一个类型也可以包含一些属性。规定类型类要以大写字母“T”开头。例如：轿车类型 "TCar"、线程类型 "TThread" 等等。在C＋＋种，类型类也叫做结点类。<br><br>在现实世界中，类型和界面的区别往往比较微妙。在真实代码中，有些类除了包含纯虚函数以外，也可能同时包含几个带简单默认实现的普通虚函数。例如：某个类中可能包含一个（非纯虚）虚方法 IsLoadable，并定义了该方法的默认实现：return false;。我们不难找出很多类似的例子。<br><br>以下是一些类型和界面的界定策略：<br><br>• 如果一个类中包含静态成员，则一定不是界面<br>• 如果一个类中包含属性，则一定不是界面<br>• 如果一个类中包含非虚方法，则一定不是界面<br>• 如果一个类中包含非公有成员，则一定不是界面<br>• 如果一个类中包含模板方法，则一定不是界面。<br>这里的模板方法是指那些调用了该类中其它虚函数的成员，这样的方法通常用于实现针对某种应用的算法框架，这显然超出了界面的范畴。<br>在C＋＋中，模板方法的另一个意思通常指使用函数模板的成员，由于C＋＋函数模板只能是非虚的，所以包含这种方法的类也一定不是界面。<br>• 通常定义那些不十分明确的接口时，先将其指定为一个界面，必要时再把它提升为一个类型。 |

| | |
|---|---|
| | **模板类**的命名规范与实体类相同。<br><br>为了更好地表示代码段之间的相关性和增加程序的可读性。我们经常会把一段仅在某个函数内反复使用的代码片段封装为一个**函数对象**，并定义在这个函数体内。对于这类实现功能简单，并且主要通过 operator() 来使用的类/结构，其名称应当以大写字母“FO”开头，如："FONameChecker" 等。 |
| 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，例如："CAnalyzer", "CFastVector", "IUnknown", "IDBWriter", "TTimer", "TThread" .... |

不同于C++类的概念，传统的C结构体只是一种将一组数据捆绑在一起的方式。传统C结构体的命名规则为：

| | |
|---|---|
| 传统C结构体的命名 | 传统C结构体的名称全部由大写字母组成，单词间使用下划线界定，例如："SERVICE_STATUS", "DRIVER_INFO" .... |

## 函数

| | |
|---|---|
| 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。例如："GetName()", "SetValue()", "Erase()", "Reserve()" .... |
| 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，例如："_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“__”，例如："__DestroyImp()" .... |
| 私有成员函数的层次结构表示 | 通常来说，在一个类中，公有方法、保护方法和私有方法所完成的任务总是呈现一种逐级依次细化的层次结构（意即：保护方法所实现的功能通常比该类中的公有方法更为细小琐碎；类似地，私有方法的功能也比其保护方法更具原子性）。<br><br>因此，对于遵循以上规则，并且功能较为复杂的类，在按照“公有、保护、私有”的三级形式划分以后，如果其私有成员中仍然存在明显不同的功能粒度，则可以通过追加更多下划线前缀的形式予以表示。<br><br>例如：由三个下划线开头的私有方法“___PushCdr”就要比同一类中，仅由两个下划线开头的私有方法“__MergeConCall”所完成的功能粒度更细小、更琐碎；而四个下划线开头的“____CalcCompensate”则比“___PushCdr”完成的功能 更具原子性。 |

| | |
|---|---|
| | 如果发现类中的功能层数太多（从公有方法到最“原子”的私有方法间，一般不应该超过 7 层），那通常反应一个不良的设计。此时请检查这个类的功能是否过于臃肿，已使接口显得不太清晰。另外一个常见的问题是将无需访问该类中 私有或保护成员的功能定义成了方法。第一个问题可以通过重新划分类层次结构或将一个类分裂为多个类等方法解决。对于第二个问题，由于这些方法无需访问 受限成员，大多数时候都可以把它们转变成局部函数（放在无名空间或使用“static”前缀定义）。 |
| 成员函数的下划线后缀命名 | 对一些本应该作为保护或私有成员的函数，由于设计方面的其它考虑（例如：灵活性、功能等方面）将其提升为公有成员的，应该在其后面添加与其原本访问控制级别相应的下划线后缀。<br><br>另外，对于其它不推荐直接使用的成员函数（例如：会引起兼容性或可移植性方面问题的函数），也应当在其后面加相应下划线提示。<br><br>例如：″ioctl_()″, ″SetSysOpt_()″, ″GetSysOpt_()″, ″PreParser__()″ .... |
| 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()″ .... |
| 虚函数 | 回调函数以外的虚函数习惯以“Do”开头，如：″DoRefresh()″, ″_DoEncryption()″ .... |

## 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。<br><br>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, x, y, z .... |
| 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：<br>**前缀** — **说明**<br>*无* — 局部变量<br>m_ — 类的成员变量（member）<br>sm_ — 类的静态成员变量（static member）<br>s_ — 静态变量（static）<br>g_ — 外部全局变量（global）<br>sg_ — 静态全局变量（static global） |

| | |
|---|---|
| gg_ | 进程 或动态链接库间共享的全局变量（global global） |

除非不得已，否则应该尽可能少使用全局变量。

关于全局变量和局部静态变量的依赖性问题和初始化时的线程安全性问题，请参考：多处理器环境和线程同步的高级话题 一节

**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 |
|---|---|
| n | 整型和位域变量（number） |
| e | 枚举型变量（enumeration） |
| c | 字符型变量（char） |
| b | 布尔型变量（bool） |
| f | 浮点型变量（float） |
| p | 指针型变量和迭代子（pointer） |
| pfn | 指向函数的指针变量或指向函数对象的指针（pointer of function） |
| pm | 指向成员的指针（pointer of member） |
| g | 数组（grid） |
| fo | 函数对象（Function Object） |
| i | 类的实例（instance）<br><br>对于经常用到的类，也可以定义一些专门的前缀，如：std::string和std::wstring类的前缀可以定义为"st"，std::vector类的前缀可以定义为"v"等等。 |

类型前缀可以组合使用，例如"gc"表示字符数组，"ppn"表示指向整型的指针的指针等等。

**数值前缀的特别记法**

以“n”作为所有整形前缀是由于大多数情况下，编写程序时不需要过多考虑整形的宽度，但在某些场合中，整形宽度是需要特别注意并且仔细加以区分的，这时可使用如下记法代替“n”前缀：

| 前缀 | 说明 |
|---|---|
| b | 字节（8bit，byte） |
| w | 字（16bit，word） |
| dw | 双字（32bit，double word） |
| qw -或- nn | 四字（64bit，quad word） |
| bf | 位域（bit field） |

对浮点型变量也有类似记法如下：

| 前缀 | 说明 |
|---|---|
| f | 单精度浮点（32bit，float） |
| d | 双精度浮点（64bit，double） |

| | | |
|---|---|---|
| | ld | 扩展精度浮点（80bit，long double） |
| 推荐的组成形式 | 变量的名字应当使用"名词"或者"形容词＋名词"。例如："nCode"，"m_nState"，"nMaxWidth" .... | |

## 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 ....<br><br>类型前缀的定义与变量命名规则中的相同。 |

## 枚举、联合、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：

| | |
|---|---|
| 枚举、联合的命名 | 由枚举、联合语句定义的类型名由全大写字母组成，单词间通过下划线来界定，如：FAR_PROC，ERROR_TYPE .... |
| typedef的命名 | 通常情况下，typedef语句定义的类型名，其命名规范与枚举及联合语句相同，也采用大写字母加下划线的原则。但是在定义一个模板类实例的别名时，为清晰起见，可以考虑酌情使用类的命名原则，例如：<br><br>`typedef CWriter< CSysFile > CSysFileWriter;`<br>`typedef std::vector< int > VINT;` |

## 宏、枚举值

| | |
|---|---|
| 宏、枚举值的命名 | 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 .... |

## 名空间

C++名空间是“类”概念的一种退化（大体相当于只包含静态成员且不能实例化的类）。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捷，同时大大降低了名字冲突的可能性。

名空间的命名规则包括：

| | |
|---|---|
| 名空间的命名 | 名空间的名称不应该过长，通常都使用缩写的形式来命名。 |

| | |
|---|---|
| | 例如，一个图形库可以将其所有外部接口存放在名空间"GLIB"中，但是将其换成"GRAPHIC_LIBRARY"就不大合适。<br><br>如果碰到较长的名空间，为了简化程序书写，可以使用：<br><br>`namespace new_name = old_long_name;`<br><br>语句为其定义一个较短的别名。 |

# 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就是不可读的。

先来看一下这方面的整体原则：

| | |
|---|---|
| 空行的使用 | 空行起着分隔程序段落的作用。空行得体（不过多也不过少）将使程序的布局更加清晰。空行不会浪费内存，虽然打印含有空行的程序是会多消耗一些纸张，但是值得。所以不要舍不得用空行。<br><br>• 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。<br><br>• 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。 |
| 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。<br><br>• "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 和修改代码时出现失误。 |
| 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。<br><br>• "{ }" 之内的代码块在 "{" 右边一个制表符（4个半角空格符）处左对齐。如果出现嵌套的 "{ }"，则使用缩进对齐。<br><br>• 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。<br><br>例如：<br><br>`void`<br>`Function(int x)`<br>`{`<br>`  CSessionLock iLock(mxLock);`<br><br>`  for (初始化; 终止条件; 更新)`<br>`  {`<br>`  // ...`<br>`  }`<br><br>`  try`<br>`  {`<br>`  // ...`<br>`  }` |

| | |
|---|---|
| | <pre>catch (const exception& err)<br>{<br>// ...<br>}<br>catch (...)<br>{<br>// ...<br>}<br><br>// ...<br>}</pre> |
| 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，否则眼睛看不过来，也不便于打印 （2009年更新：随着GUI开发环境和高分宽屏的普及，此规则可以视情况适当放宽）。 |
| 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。<br><br>例如：<br><br><pre>if ((very_longer_variable1 >= very_longer_variable2)<br>&& (very_longer_variable3 <= very_longer_variable4)<br>&& (very_longer_variable5 <= very_longer_variable6)<br>)<br>{<br>DoSomething();<br>}</pre> |
| 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。<br><br>• 函数名之后不要留空格，紧跟左括号 "(" ，以与关键字区别。<br><br>• "(" 向后紧跟。而 ")"、","、";" 向前紧跟，紧跟处不留空格。<br><br>• "," 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)。<br><br>• 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"="、"+="<br>">="、"<="、"+"、"*"、"%"、"&&"、"\|\|"、"<<"，"^" 等二元操作符的前后应当加空格。<br><br>• 一元操作符如 "!"、"~"、"++"、"--"、"&"（地址运算符）等前后不加空格。<br><br>• 象"[]"、"."、"->"这类操作符前后不加空格。<br><br>• 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();
```

| | |
|---|---|
| 修饰符的位置 | 为便于理解，应当将修饰符 "*" 和 "&" 紧靠数据类型。<br><br>例如：<br><br>`char* name;`<br><br>`int* x;`<br>`int  y;  // 为避免y被误解为指针，这里必须分行写。`<br><br>`int* Function(void* p);`<br><br>参见：变量、常量的风格与版式 -> 指针或引用类型的定义和声明 |
| 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。<br>• 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。<br>• 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。<br>• 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。 |
| 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
{
    // ...
}
```

**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 |
| UNDONE | 已取消的接口、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 << "...";
```

## 类/结构

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品质。

**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>, [yyy], [zzz] ...（作者和逗号分割的修改者列表）
</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
{
    // ...
};
```

| | |
|---|---|
| 继承 | 基类直接跟在类名称之后，不换行，访问说明符（public, private, 或protected）不可省略。如：<br><br>`class CXXX : public CAAA, private CBBB`<br>`{`<br>`  // ...`<br>`};` |
| 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。<br><br>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。 |
| 访问说明符 | 访问说明符（public, private, 或protected）应该独占一行，并与类声明中的‘class’关键字左对齐。 |
| 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。<br><br>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。<br><br>如果一类声明中有很多组功能不同的成员，还应该用分组注释将其分组。分组注释也要与 ″class″ 关键字对齐。<br><br>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），成员之间左对齐。<br><br>例如：<br><br>`class CXXX`<br>`{`<br>`public:`<br>`/////////////////////////////////////////////////////////////////// 类型定义`<br>`  typedef vector<string> VSTR;`<br><br>`public:`<br>`/////////////////////////////////////////////////////////// 构造、析构、初始化`<br>`  CXXX();`<br>`  ~CXXX();`<br><br>`public:`<br>`/////////////////////////////////////////////////////////////////// 公用方法`<br><br>`// [[ 功能组1`<br>`  void Function1(void) const;`<br>`  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);
};
```

**正确地使用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;
    // ...
};
```

也就是说，应当尽量使所有逻辑上只读的操作成为const方法，然后使用mutable解决那些存在逻辑冲突的属性。

**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
    {
        // ...
    };
};
```

**初始化**

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

| | |
|---|---|
| 列表 | 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。<br><br>例如：<br><br>`CXXX::CXXXX(IN int nA, IN bool bB)`<br>`  : m_nA(nA), m_bB(bB)`<br>`{`<br>`  // ...`<br>`};`<br><br>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。<br><br>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）。<br><br>例如：<br><br>`// ...`<br><br>`class CXXXX : public CAA, public CBB`<br>`{`<br>`  // ...`<br>`  CYY m_iA;`<br>`  CZZ m_iB;  // m_iA必须在m_iB之前声明`<br>`};`<br><br>`CXXX::CXXXX(IN int nA, IN int nB, IN bool bC)`<br>`  : CAA(nA), CBB(nB), m_iA(bC), m_iB(m_iA) // 先基类，后成员，`<br>`  // 分别按照声明顺序书写`<br>`{`<br>`  // ...`<br>`};` |
| 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。 |

关于类声明的例子，请参见：类/结构的风格与版式例子

关于类声明的模板，请参见：类声明模板

## 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。

| | |
|---|---|
| 函数原型 | 函数原型的格式为：<br><br>[存储类] 返回值类型<br>[名空间或类::]函数名(参数列表) [const说明符] [异常过滤器]<br><br>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 及类型转换操作符外，"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

**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 类的成员函数时，为了紧凑，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。

**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
            ...
-----------------------------------------------------------------------------
复杂度    : <描述函数的复杂度/开销（可选）>
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>, [yyy], [zzz] ...（作者和逗号分割的修改者列表）
</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。

**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 |
| DUMMY | 哑元参数－不使用参数的值，仅为帮助函数重载解析等目的而设置的参数 |
| 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 | 保留参数－这个参数当前未被支持，留待以后扩展；或者该参数为内部使用，用户无需关心 |
| OWNER | 获得参数的所有权，调用者不再负责销毁实参指定的对象 ；如果用来修饰返回值，则表示调用者获得返回值的所有权，并负责将其销毁 |
| UNUSED | 标明这个参数在此版本中已不再使用 |
| CHANGED | 参数类型或用途 与较早版本相比发生了变化 |
| ADDED | 新增的参数 |
| NOTE | 需要注意的参数－参数意义发生变化 或者与习惯用法不同 |
| WRKBUF | 工作缓冲区－为避免频繁分配临时资源而传入的临时工作区 |

其中：

- 除了空参数 "void" 和哑元参数以外，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：只要一个描述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。

**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* piDB, OPTIONAL IN OUT int* pnRecordCount = NULL

IN OUT string& stRuleList, RESERVED IN int nOperate = 0

...
```

其中：

- "参数描述宏" 见上文
- 参数命名规范与变量的命名规范相同

**存储类**

"extern", "static", "inline" 等函数存储类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，也不允许一个类型声明仅存在于函数的声明或定义中。

**成员函数的存储类**

由于C++语言的限制， 类中成员函数的 "static", "virtual", "explicit" 等存储类说明不允许出现在函数定义中。

但是为了明确起见，这些存储类应以注释的形式在相应的成员定义中给出。

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。

**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
{
    // ...
}
```

**异**

对于任何可能抛出异常的函数，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

**常过滤器**

中列举该函数可能抛出的异常。

例如：

```
int
Function(IN const char* pcName) throw(byExp, exception);
```

如果一个函数本身及其**直接**调用的函数不会显式抛出异常（没有指定异常过滤器），那么该函数可以省略异过滤器。

特别地：如果一个函数内部显式地捕获了任何可能的异常（例如：使用了 "catch (...)" ），并且保证不抛出任何异常，那么应该在其声明和定义中显式地指定一个空异常过滤器："throw()"。

例如：

```
int
Function(IN const char* pcName) throw();
```

特别地：进程入口函数（"main()"）**不**应当使用异常过滤器。

**代码段注释**

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(mxLock);

        //
==================================================================
        // = 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 = sm_iProcTbl.begin();
            for (; 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（超过100行）或代码的意图不明确时，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。

特别地，对由于如加密及性能优化等特殊原因无法提取为一个单独函数的代码段，应当使用特别代码段注释显式分割。当然，类似情况应当尽量使用内联函数或编译器提供的强制性内联函数代替。

例如：

```
    void
    CXXX::Function(void)
    {
        // ...

        //
    @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        // @@ 获取首网卡的 MAC 地址
        typedef CTmpHandle<IP_ADAPTER_INFO, FreeDeletor<IP_ADAPTER_INFO> >
                THADAPTERINFO;

        byteEx btAddr;
        THADAPTERINFO thAdapterInfo;
        thAdapterInfo = (IP_ADAPTER_INFO*) malloc(sizeof(IP_ADAPTER_INFO));

        ULONG ulOutBufLen = sizeof(IP_ADAPTER_INFO);

        // Make an initial call to GetAdaptersInfo to get
        // the necessary size into the ulOutBufLen variable
        if (ERROR_SUCCESS != ::GetAdaptersInfo(thAdapterInfo, &ulOutBufLen))
        {
            thAdapterInfo = (IP_ADAPTER_INFO*) malloc(ulOutBufLen);
        }

        if (NO_ERROR != ::GetAdaptersInfo(thAdapterInfo, &ulOutBufLen))
        {
        #ifdef DEBUG
            CLog::DebugMsg(byT("lic verifier"), byT("failed verifying
    license"));
        #endif
            CProcess::Exit(-97);
        }
        btAddr.assign(thAdapterInfo->Address, thAdapterInfo->AddressLength);
        // @@ 获取首网卡的 MAC 地址
        //
    @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

        // ...
    }
```

| | |
|---|---|
| 调用系统API | 所有系统API调用前都要加上全局名称解析符 "::"。<br><br>例如：<br><br>`::MessageBoxA(NULL, gcErrorMsg, "!FATAL ERROR!", MB_ICONSTOP\|MB_OK);`<br><br>`if (0 == ::GetTempFileName(m_basedir.c_str(), byT("bai"), 0, stR.ref()))`<br>`{`<br>`  // ...`<br>`}` |
| 让相同的代码只出现一次 | 为了使程序更容易调试、修改，尽量降低日后维护的复杂性，应该把需要在一个以上位置使用的代码段封装成函数。哪怕这段代码很短，为了以后维护方便着想，也应当将其封装为内联函数。 |

关于函数的例子，请参见：函数的风格与版式例子

关于函数的模板，请参见：函数模板

## 变量、常量

| | |
|---|---|
| 声明格式 | 变量、常量的声明格式如下：<br><br>［存储类］类型 变量名;<br><br>其中：<br><br>• 以 "[ ]" 括住的为可选项目。<br>• "存储类" 的说明见下文 |
| 定义格式 | 变量、常量的定义格式如下：<br><br>［存储类］类型 变量名 = 初始值;<br><br>其中：<br><br>• 以 "[ ]" 括住的为可选项目。<br>• "存储类" 的说明见下文 |
| | |

| | |
|---|---|
| 存储类 | 除 "auto" 类型以外，诸如 "extern", "static", "register", "volatile" 等存储类均不可省略，且必须在声明和定义中一致地使用（即：不允许仅在声明或定义中使用）。 |
| 成员变量的存储类 | 由于C++语言的限制，成员变量的 "static" 等存储类说明不允许出现在变量定义中。<br><br>但是为了明确起见，这些存储类应以注释的形式在定义中给出。<br><br>例如：<br><br>`/*static*/ int CThread::sm_nPID = 0;` |
| 指针或引用类型的定义和声明 | 在声明和定义多个指针或引用变量/常量时，每个变量至少占一行。例如：<br><br>`int* pn1,`<br>`  * pn2 = NULL,`<br>`  * pn3;`<br><br>`char* pc1;`<br>`char* pc2;`<br>`char* pc3;`<br><br>`// 错误的写法：`<br>`int* pn11, *pn12, *pn13;` |
| 常指针和指针常量 | 声明/定义一个常指针（指向常量的指针）时，"const" 关键字一律放在类型说明的左侧。<br><br>声明/定义一个指针常量（指针本身不能改变）时， "const" 关键字一律放在变量左侧、类型右侧。<br><br>例如：<br><br>`const char* pc1;  // 常指针`<br>`char* const pc2;  // 指针常量`<br>`const char* const pc3; // 常指针常量`<br><br>`// 错误的写法：`<br>`char const* pc1;  // 与 const char* pc1 含义相同，但不允许这样写` |
| 全局变量、常量的注释 | 全局变量、常量的注释独占一行，并用 "//!" 开头。<br><br>例如：<br><br>`//! 当前进程的ID`<br>`static int sg_nPID = 0;`<br><br>`//! 分割符`<br>`static const char* pcDTR = "\\/";` |
| | |

<table>
<tr><td>类型转换</td><td>禁止使用C风格的 ″(类型)″ 类型转换，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。其中：
<table>
<tr><td>static_cast</td><td>static_cast 用于编译器认可的，安全的静态转换，比如将 ″char″ 转为 ″int″ 等等。该操作通常在编译时完成，但有可能调用用户定义的类型转换操作或非 explicit 的单参（或至少从第二个参数开始带缺省值的）构造函数 。</td></tr>
<tr><td>reinterpret_cast</td><td>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″ 等等。这种转换有可能产生可移植性方面的问题，该操作在编译时完成 。（注意：reinterpret_cast 比 C 风格的类型转换还要野蛮，它不进行任何地址对其和调整，也不调用任何用户定义的类型转换操作）</td></tr>
<tr><td>const_cast</td><td>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。这种转换可能伴随潜在的错误。该操作在编译时完成 。</td></tr>
<tr><td>dynamic_cast</td><td>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 。反之，若编辑器关闭了 RTTI 支持，或被转换的类层次结构中没有抽象类存在，则此操作在编译时完成（有些编译器会给出警告）。</td></tr>
</table>
此外，对于定义了单参构造函数或类型转换操作的类来说，应当优先使用构造函数风格的类型转换，如：’string(″test″)’ 等等。<br><br>通常来说，″xxx_cast″ 格式的转换与构造函数风格的类型转换之间，最大的区别在于：构造函数风格的转换经常会生成新的临时对象，可能伴随相当的时间和空间开销。而 ″xxx_cast″ 格式的转换只是告诉编译器，将指定内存中的数据当作另一种类型的数据看待，这些操作一般在编译时完成，不会对程序的运行产生额外开销。当然，″dynamic_cast″ 和某些 ″static_cast″ 则例外。<br><br>参见：RTTI、虚函数和虚基类的开销分析和使用指导</td></tr>
</table>

## 枚举、联合、typedef

| | |
|---|---|
| 枚举、联合的定义格式 | 枚举、联合的定义格式为： |

| | |
|---|---|
| | //! 说明（可选）<br>enum\|union 名称<br>{<br>内容 // 注释（可选）<br>};<br><br>例如：<br><br>//! 服务的状态<br>enum SRVSTATE<br>{<br>SRV_INVALID = 0, // 无效（尚未启动）<br>SRV_STARTING = 1,<br>SRV_STARTED,<br>SRV_PAUSING,<br>SRV_PAUSED,<br>SRV_STOPPING,<br>SRV_STOPPED<br>};<br><br>//! 32位整数<br>union INT32<br>{<br>unsigned char cByte[4];<br>unsigned short nShort[2];<br>unsigned long nFull;<br>}; |
| typedef的定义格式 | typedef 的定义格式为：<br><br>//! 说明（可选）<br>typedef 原类型 类型别名;<br><br>例如：<br><br>//! 返回值类型<br>typedef int EXITCODE;<br><br>//! 字符串数组类型<br>typedef vector<string> VSTR; |

## 宏

宏是C/C++编译环境提供给用户的，在编译开始前（编译预处理阶段）执行的唯一可编程逻辑。

| | |
|---|---|
| 何时使用宏 | 应当尽量减少宏的使用，在所有可能的地方都使用常量 、模版和内联函数来代替宏。 |
| 边界效应 | 使用宏的时候应当注意边界效应，例如，以下代码将会得出错误的结果：<br><br>#define PLUS(x,y) x+y<br><br>cout << PLUS(1,1) * 2; |

| | |
|---|---|
| | 以上程序的执行结果将会是 "3"，而不是 "4"，因为 "PLUS (1,1) * 2" 表达式将会被展开为："1 + 1 * 2"。<br><br>因此在定义宏的时候，只要允许，就应该为它的替换内容括上 "( )" 或 "{ }"。例如：<br><br>`#define PLUS(x,y) (x+y)`<br><br>`#define SAFEDELETE(x) {delete x; x=0}` |
| 对复杂的宏实行缩进 | 有时为了实现诸如：对编译器和目标平台自适应；根据用户选项编译不同模块等机制，需要使用大量较为复杂的宏定义块。在宏比较复杂（代码块多于5行）的地方，为了便于阅读和理解，应当遵循与普通C++代码相同的原则进行缩进和排版。<br><br>为了区别于其他语句和便于阅读，宏语句的 "#" 前缀**不要**与语句本身一起缩进，例如：<br><br>`//! Windows`<br>`#if defined(__WIN32__)`<br>`#  if defined(__VC__) \|\| defined(__BC__) \|\| defined (__GNUC__) // ...`<br>`#  define BAIY_EXPORT __declspec(dllexport)`<br>`#  define BAIY_IMPORT __declspec(dllimport)`<br>`#  else // 编译器不支持 __declspec()`<br>`#  define BAIY_EXPORT`<br>`#  define BAIY_IMPORT`<br>`#  endif`<br><br>`//! OS/2`<br>`#elif defined(__OS2__)`<br>`#  if defined (__WATCOMC__)`<br>`#  define BAIY_EXPORT __declspec(dllexport)`<br>`#  define BAIY_IMPORT`<br>`#  elif !(defined(__VISAGECPP__) && (__IBMCPP__<400 \|\| __IBMC__<400))`<br>`#  define BAIY_EXPORT _Export`<br>`#  define BAIY_IMPORT _Export`<br>`#  endif`<br><br>`//! Macintosh`<br>`#elif defined(__MAC__)`<br>`#  ifdef __MWERKS__`<br>`#  define BAIY_EXPORT __declspec(export)`<br>`#  define BAIY_IMPORT __declspec(import)`<br>`#  endif`<br><br>`// Others`<br>`#else`<br>`#  define BAIY_EXPORT`<br>`#  define BAIY_IMPORT`<br><br>`#endif` |

## 名空间

| | |
|---|---|
| 名空间的使用 | 名空间可以避免名字冲突、分组不同的接口以及简化命名规则。应当尽可能地将所有接口都放入适当的名字空间中。 |

| | |
|---|---|
| 将实现和界面分离 | 提供给用户的界面和用于实现的细节应当分别放入不同的名空间中。<br><br>例如：如果将一个软件模块的所有接口都放在名空间 "MODULE" 中，那么这个模块的所有实现细节就可以放入名空间 "MODULE_IMP" 中 或者 "MODULE" 内的 "IMP" 中。 |

## 异常

异常使C++的错误处理更为结构化；错误传递和故障恢复更为安全简便；也使错误处理代码和其它代码间有效的分离开来。

| | |
|---|---|
| 何时使用异常 | 异常机制只用在发生错误的时候，仅在发生错误时才应当抛出异常。这样做有助于错误处理和程序动作两者间的分离，增强程序的结构化，还保证了程序的执行效率。<br><br>确定某一状况是否算作错误有时会很困难。比如：未搜索到某个字符串、等待一个信号量超时等等状态，在某些情况下可能并不算作一个错误，而在另一些情况下可能就是一个致命错误。<br><br>有鉴于此，仅当某状况必为一个错误时（比如：分配存储失败、创建信号量失败等），才应该抛出一个异常。而对另外一些模棱两可的情况，就应当使用返回值等其它手段报告 。<br><br>此外，在发生错误的位置，已经能够获得足够的信息处理该错误的情况不属于异常，应当对其就地处理。只有无法获得足够的信息来处理发生的错误时，才应该抛出一个异常。 |
| 用异常代替goto等其它错误处理手段 | 曾经被广泛使用的传统错误处理手段有goto风格和do...while风格等，以下是一个goto风格的例子：<br><br>反面教材<br><br>//! 使用goto进行错误处理的例子<br>bool<br>Function(void)<br>{<br>int nCode, i;<br>bool r = false;<br><br>// ...<br><br>if (!Operation1(nCode))<br>{<br>goto onerr;<br>}<br><br>try<br>{<br>Operation2(i);<br>}<br>catch (...)<br>{<br>r = true; |

```
            goto onerr;
        }

        r = true;

    onerr:
        // ... 清理代码
        return r;
    }
```

由上例可见，goto风格的错误处理至少存在问题如下：

- 错误处理代码和其它代码混杂在一起，使程序不够清晰易读 。
- 函数内的变量必须在第一个 "goto" 语句之前声明，违反就近原则。
- 多处跳转的使用破坏程序的结构化，影响程序的可读性，使程序容易出错 。
- 对每个会抛出异常的操作都需要用额外的 try...catch 块检测和处理。
- 稍微复杂一点的分类错误处理要使用多个标号和不同的goto跳转（如："goto onOp1Err"，"goto onOp2Err" ...）。这将使程序变得无法理解和错误百出。

再来看看 do...while 风格的错误处理：

反面教材

```
//! 使用do...while进行错误处理的例子
bool
Function(void)
{
    int nCode, i;
    bool r = false;

    // ...

    do
    {
        if (!Operation1(nCode))
        {
            break;
        }

        do
        {
            try
            {
                Operation2(i);
            }
            catch (...)
            {
                r = true;
                break;
            }
        } while (Operation3())

        r = true;

    } while (false);

    // ... 清理代码
    return r;
```

```
}
```

与 goto 风格的错误处理相似 ，do...while 风格的错误处理有以下问题：

- 错误处理代码和其它代码严重混杂，使程序非常难以理解 。
- 需要进行分类错误处理 时非常困难，通常需要事先设置一个标志变量，并在清理时使用 "switch case" 语句进行分检。
- 对每个会抛出异常的操作都需要用额外的 try...catch 块检测和处理 。

此外，还有一种更糟糕的错误处理风格——直接在出错位置就地完成错误处理：

超级反面教材

```
//! 直接进行错误处理的例子
bool
Function(void)
{
    int nCode, i;

    // ...

    if (!Operation1(nCode))
    {
        // ... 清理代码
        return false;
    }

    try
    {
        Operation2(i);
    }
    catch (...)
    {
        // ... 清理代码
        return true;
    }

    // ...

    // ... 清理代码
    return true;
}
```

这种错误处理方式所带来的隐患可以说是无穷无尽，这里不再列举。

与传统的错误处理方法不同，C++的异常机制很好地解决了以上问题。使用异常 完成出错处理时，可以将大部分动作都包含在一个try块中，并以不同的catch块捕获和处理不同的错误：

```
//! 使用异常进行错误处理的例子
bool
Function(void)
{
    int nCode, i;
    bool r = false;

    try
    {
        if (!Operation1(nCode))
```

```
            {
                throw false;
            }

            Operation2(i);
        }
        catch (bool err)
        {
            // ...
            r = err;
        }
        catch (const excption& err)
        {
            // ... excption类错误处理
        }
        catch (...)
        {
            // ... 处理其它错误
        }

        // ... 清理代码
        return r;
    }
```

以上代码示例中，错误处理和动作代码完全分离，错误分类清晰明了，好处不言而喻。

**构造函数中的异常**

在构造函数中抛出异常将中止对象的构造，这将产生一个没有被完整构造的对象。

对于C++来说，这种不完整的对象将被视为尚未完成创建动作而不被认可，也意味着其析构函数永远不会被调用。这个行为本身无可非议，就好像公安局不会为一个被流产的婴儿发户口然后再开个死亡证明书一样。但有时也会产生一些问题，例如：

```
class CSample
{
    // ...

    char* m_pc;
};

CSample::CSample()
{
    m_pc = new char[256];
    // ...
    throw -1;  // m_pc将永远不会被释放
}

CSample::~CSample()  // 析构函数不会被调用
{
    delete[] m_pc;
}
```

解决这个问题的方法是在抛出异常以前释放任何已被申请的资源。一种更好的方法是使用 一个满足“资源申请即初始化（RAII）”准则的类型（如：句柄类、灵巧指针类等等）来代替一般的资源申请与释放方式，如：

```
templete <class T>
struct CAutoArray
{
    CAutoArray(T* p = NULL) : m_p(p) {};
    ~CAutoArray() {delete[] m_p;}
    T* operator=(IN T* rhs)
    {
```

```
        if (rhs == m_p)
            return m_p;
        delete[] m_p;
        m_p = rhs;
        return m_p;
    }
    // ...

    T* m_p;
};

class CSample
{
    // ...

    CAutoArray<char> m_hc;
};

CSample::CSample()
{
    m_hc = new char[256];
    // ...
    throw -1;  // 由于m_hc已经成功构造，m_hc.~CAutoPtr()将会
               // 被调用，所以申请的内存将被释放
}
```

注意：上述CAutoArray类仅用于示范，对于所有权语义的通用自动指针，应该使用C++标准库中的 "auto_ptr" 模板类。对于支持引用计数和自定义销毁策略的通用句柄类，可以使用白杨工具库中的 "CHandle" 模板类。

**析构函数中的异常**

析构函数中的异常可能在2种情况下被抛出：

1. 对象被正常析构时
2. 在一个异常被抛出后的退栈过程中——异常处理机制退出一个作用域，其中所有对象的析构函数都将被调用。

由于C++不支持异常的异常，上述第二种情况将导致一个致命错误，并使程序中止执行。例如：

```
class CSample
{
    ~CSample();
    // ...
};

CSample::~CSample()
{
    // ...
    throw -1;  // 在 "throw false" 的过程中再次抛出异常
}

void
Function(void)
{
    CSample iTest;
    throw false;  // 错误，iTest.~CSample()中也会抛出异常
}
```

如果必须要在析构函数中抛出异常，则应该在异常抛出前用 "std::uncaught_exception()" 事先判断当前是否存在已被抛出但尚未捕获的异常。例如：

```
// uncaught_exception() 函数在这个头文件中声明
#include <exception>

class CSample
{
    ~CSample();
    // ...
};

CSample::~CSample()
{
    // ...
    if (!std::uncaught_exception()) // 没有尚未捕获的异常
    {
        throw -1;  // 抛出异常
    }
}

void
Function(void)
{
    CSample iTest;
    throw false;  // 可以，iTest.~CSample()不会抛出异常
}
```

new 时的异常

在 C++ 标准（ISO/IEC 14882:2003）第 15.2 节中明确规定，在使用 new 或 new[] 操作创建对象时，如对象的构造函数抛出了异常，则该对象的所有成员和基类都将被正确析构，**如果存在一个与使用的 operator new 严格匹配的 operator delete，则为这个对象所分配的内存也会被释放**。例如：

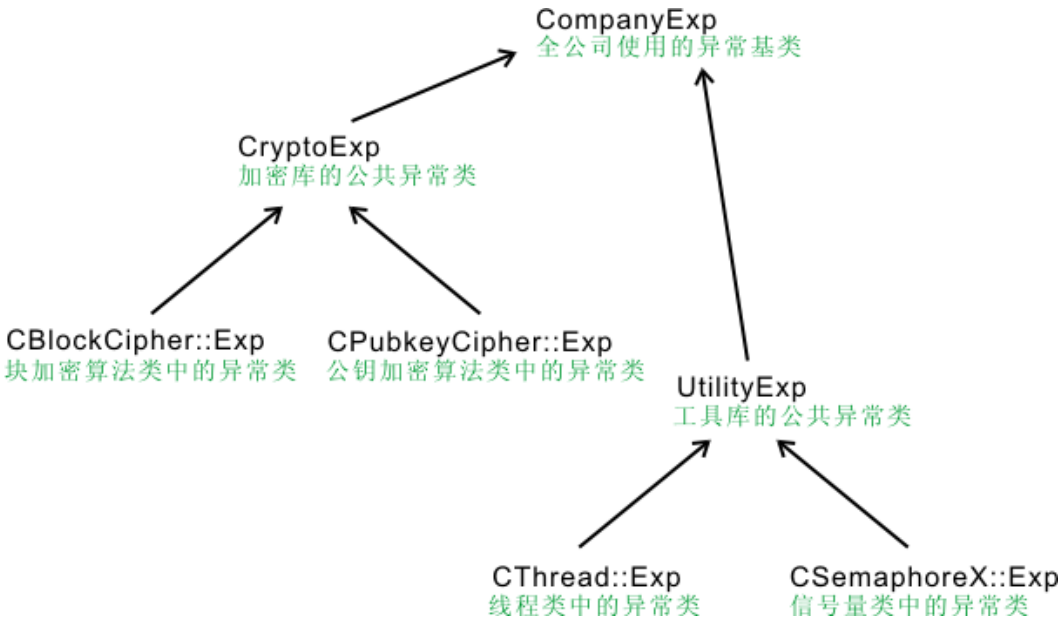
```
class CSample
{
    CSample() { throw -1; }

    static void* operator new(IN size_t n)
        { return malloc(n); }

    static void operator delete(IN void* p)
        { free(p); }

    static void* operator new(IN size_t n, IN CMemMgr& X)
        { return X.Alloc(n); } // 缺少匹配的 operator delete
};


void
Function(void)
{
    CSample* p1 = new CSample; // 有匹配的 operator delete，为 p1 分配
的内存会被释放
    CSample* p2 = new(iMyMemMgr) CSample; // 没有匹配的 operator
delete，内存泄漏！为 p2 分配的内存永远不会被释放
}


// 编译器实际生成的代码像这样：
void
Function(void)
{
    CSample* p1 = CSample::operator new(sizeof(CSample));
    try { p1->CSample(); } catch(...) {CSample::opertaor delete(p1);
throw(); }

    CSample* p2 = CSample::operator new(sizeof(CSample), iMyMemMgr);
    p2->CSample();
```

```
}
```

这里顺便提一句，delete 操作只会匹配普通的 operator delete（即：全局或类中的 operator delete(void*) 和类中的 operator delete(void*, size_t)），**如果像上例中的 p2 那样使用了一个高度自定义的 operator new，用户就需要自己完成析构和释放内存的动作**，例如：

```
// ...
p2->~CSample();
CSample::operator delete(p2, iMymemMgr);
```

| | |
|---|---|
| delete 时的异常 | C++ 标准中明确规定，如果在一个析构函数中中途返回（不管通过 return 还是 throw），该析构函数不会立即返回，而是会逐一调用所有成员和基类的析构函数后才会返回。但是标准中并没有说明如果这个异常是在 delete 时发生的（即：该对象是由 new 创建的），此对象本身所占用的堆存储是否会被释放（即：在 delete 时析构函数抛出异常会不会调用 operator delete 释放这个对象占用的内存）。<br><br>**在实际情况中，被 delete 的对象析构函数抛出异常后，GCC、VC 等流行的 C++ 编译器都不会自动调用 operator delete 释放对象占用的内存**。这种与 new 操作不一致的行为，其背后的理念是：在构造时抛出异常的对象尚未成功创建，系统应当收回事先为其分配的资源；而析构时抛出异常的对象 并未成功销毁，系统不能自动回收它使用的内存。<br><br>例如：如果一个对象在构造时申请了系统资源（比如：打开了一个设备）并保留了与这个设备对应的句柄，但在析构时归还该资源失败（例如：关闭设备失败），则自动调用 operator delete 会销毁这个尚未关闭的句柄，导致 用户永远失去向系统归还资源或者进行进一步处理的机会。反之，如果这个对象在构造时就没能成功地申请到相应资源，则自动回收预分配给它的内存空间是安全的，不会产生任何资源泄漏。<br><br>应当注意到，如果一个对象在析构时抛出了异常，则这个对象很可能已经处于一个不完整的状态。此时访问该对象中的任何非静态成员都是不安全的。因此，应当在被抛出的异常中包含完成进一步处理的足够信息。这样捕获到这个异常的用户就可以安全地释放该对象占用的内存，并且仅使用异常对象完成后续处理。例如：<br><br>`//! delete 时异常处理的例子`<br>`void`<br>`Function(void)`<br>`{`<br>`  CSample* p1 = new CSample;`<br>`  // ...`<br>`  try`<br>`  {`<br>`  delete p1;`<br>`  }`<br>`  catch (const sampleExp& err)`<br>`  {`<br>`  CSample::operator delete(p1); // 释放 p1 所占用的内存`<br>`  // 使用 err 对象完成后续的错误处理...`<br>`  }`<br>`}` |
| 异常的组织 | 异常类应该以继承的方式组织成一个层次结构，这将使以不同粒度分类处理错误成为可能。<br><br>通常，某个软件生产组织的所有异常都从一个公共的基类派生出来。而每个类的异常则从该类所属模块的公共异常基类中派生。例如： |




**异常捕获和重新抛出**

- 异常捕获器的书写顺序应当由特殊到一般（先子类后基类），最后才是处理所有异常的捕获器（"catch(...)"）。否则将使某些异常捕获器永远不会被执行。
- 为避免捕获到的异常被截断，异常捕获器中的参数类型应当为常引用型或指针型。
- 在某级异常捕获器中无法被彻底处理的错误可以被重新抛出。重新抛出采用一个不带运算对象的 "throw" 语句。重新抛出的对象就是刚刚被抛出的那个异常，而不是处理器捕获到的（有可能被截断的）异常。

例如：

```
try
{
    // ...
}
// 公钥加密错误
catch (const CPubKeyCipher::Exp& err)
{
    if (可以恢复)
    {
        // 恢复错误
    }
    else
    {
        // 完成能做到的事情
        throw;  // 重新抛出
    }
}
// 处理其它加密库错误
catch (const CryptoExp& err)
{
    // ...
}
// 处理其它本公司模块抛出的错误
catch (const CompanyExp& err)
{
    // ...
}
// 处理 dynamic_cast 错误
catch (const bad_cast& err)
```

| | |
|---|---|
| | `{`<br>`  // ...`<br>`}`<br>`// 处理其它标准库错误`<br>`catch (const exception& err)`<br>`{`<br>`  // ...`<br>`}`<br>`// 处理所有其它错误`<br>`catch (...)`<br>`{`<br>`  // 完成清理和日志等基本处理...`<br>`  throw;  // 重新抛出`<br>`}` |
| 异常和效率 | 对于绝大部分现代编译器来说，在不抛出异常的情况下，异常处理的实现在运行时 几乎不会有任何额外开销。相反，很多时候，异常机制比传统的通过返回值判断错误的开销还来得稍微小些。<br><br>相对于函数返回和调用的开销来讲，异常抛出和捕获的开销通常会大一些。不过错误处理代码通常不会频繁调用， 再说传统的错误处理方式也不是没有代价的。所以错误处理时开销稍大一点基本上不是什么问题。这也是我们提倡仅将异常用于错误处理的原因之一。<br><br>更多关于效率的讨论，参见：C++异常机制的实现方式和开销分析 和 RTTI、虚函数和虚基类的开销分析和使用指导 等小节。 |

## 修改标记

在代码交叉审查，或使用带完整源代码的第三方库时，经常需要为某些目的修改源码。这时应当为被改动的部分添加修改标记。

| | |
|---|---|
| 何时使用修改标记 | 修改标记通常仅用于修改者不是被修改模块（或项目）的主要作者时，但也可以用于在调试、重构或添加新特性时进行临时标注。<br><br>在交叉审查中使用的修改标记，当原作者已经确认并将其合入主要版本之后，应当予以消除，以避免由于多次交叉审查累积的标记混乱。但是相应的修改应当记入文件头的修改记录中。 |
| 修改标记的格式 | 修改标记分为单行标记和段落标记两种，单行标记用于指示对零星的单行代码进行的修改，段落标记则用于指出对一组任意长度的代码作出的修改。它们的格式如下：<br><br>`// 单行标记:`<br>`// code ...; // by <修改者> - <目的> [@ YYYY-MM-DD（可选的修改日期）]`<br><br>`// 段落标记:`<br>`// [[ by <修改者> - <目的> [@ YYYY-MM-DD（可选的修改日期）]`<br>`//  详细说明（可选，可多行）`<br>`// ... // 被修改的代码段落`<br>`// ]] [by <修改者>]` |

| | |
|---|---|
| | 注意段落标记结尾的 "by <修改者>" 字段是可选的。<br><br>此外，在比较混乱或较长的代码段中，可以将段落开始（"// [[" ）和段落结束（"// ]]" ）标记扩展层次结构更为明显的："// ---- [[" 和 "// ---- ]]"<br><br>例如：<br><br>`// [[ by BaiYang - limit @ 2005-03-29`<br>`//  add pre compile and delay binding support to "limit [s,]n".`<br>`void setStatementLimit(dbQuery const& q) {`<br>`  // ...`<br>`}`<br>`// ]]`<br><br>`// ...`<br><br>`// ---- [[ by Mark - multithread`<br>`void dbCompiler::compileLimitPart(dbQuery& query)`<br>`{`<br>`  // ...`<br>`  int4* lp1 = INVPTR; // by BaiYang - limit`<br>`  switch (scan())`<br>`  {`<br>`  case tkn_iconst:`<br>`  // ...`<br>`}`<br>`// ---- ]] by Mark` |
| 修改标记的语言 | 修改标记当中的说明性文字应当尽量选择与被修改项目一致的语言书写。例如在全英文的项目中应当尽量避免添加中文注释。<br><br>否则能完全看懂修改后项目的程序员将会被限制于同时掌握多种自然语言的人。 |

## 版本控制

- 源代码的版本按文件的粒度进行维护。
- 创建一个新文件时，其初始版本为 "1.0"，创建过程中的任何修改都不需要增加修改记录。
- 从软件第一次正式发布开始，对其源文件的每次修改都应该在文件头中加入相应的修改记录，并将文件的子版本加1。
- 升级软件的主版本时，其源文件的相应主版本号随之增加。与创建新文件时一样，在该主版本第一 次发布之前，对文件的任何修改都不需要再增加修改记录。

## 英文版

对于为海外用户编写的代码，所有注释都统一使用英文。关于各标准注释的英文模板，请参考：常用英文注释一览

## 自动工具与文档生成

纵观 MSDN、unix/linux manaul(man)、wxWindows Doc等享有盛誉的开发文档都是手工或半手工编写的。相反，那些完全由自动工具生成的文档基本上都是被广大程序员唾弃的 。

由此可以看出，以现今的人工智能科技，完全由机器生成的文档，仍然无法满足人类阅读的需要。但是一份注释详实、版式规范的源代码配合一些简单的工具确实可以大大降低文档编写的工作量。从这样的源码中抽取出来的信息，通常只要稍加整理和修改就可以得到一份媲美MSDN的文档了。

详情参见：软件模块用户文档模板

## 关于本规范的贯彻实施

像这样一套完整、详细、繁琐又重要的规范，最难的恐怕就是贯彻实施的环节了。有鉴于此，特提供几点意见：

| | |
|---|---|
| 设立代码审查小组 | 对于大型开发团队，可以设立专门的代码审查小组，对项目的所有源码进行有计划的审查和评估。 |
| 设立交叉审查制度 | 在程序员间设立源码的交叉审查制度，鼓励大家互相督促。 |
| 提供专门的检查工具 | 以IDE插件和独立应用程序的形式提供专门的自动化批量代码检查工具，提高审查与自我检查的效率和精度。 |
| 设立奖惩和激励机制 | 不用多说了，金钱加大棒。坚决执行的公开表扬、发奖金；发证书，顽冥不化的K一顿然后废了他 |

## 术语表

| 术语 | 解释 |
|---|---|
| API | 应用程序编程接口 |
| UI | 用户界面 |
| GUI | 图形用户界面 |
| OOP | 面向对象的程序开发 |
| IDE | 集成开发环境 |
| 没有规矩 不成方圆 | 没有规矩 否成方圆 |

## 参考文献

| 名称 | 作者 | 发布/出版日期 |
|---|---|---|
| C++程序设计语言——特别版 | Bjarne Stroustrup | 2002 |
| ANSI/ISO C++ Professional Programmer's Handbook | Denny Kalev | 1999 |
| ISO/IEC 14882 Programming Languages -- C++ | the ANSI C++ community | 1998 |
| Microsoft MSDN | 微软公司 | 期刊，参照版本为2004年4月 |
| linux/unix在线手册（man） | - | - |
| wxWindows 2.4.2 Doc | wxWindows | September 2003 |
| 高质量C++/C编程指南 | 林锐 博士 | 2001年7月24日 |
| 人月神话——20周年纪念版 | Frederick P. Brooks Jr. | 2002 |
| C/C++编程规范（华为） | 苏信南 | 1997-5-5 |
| C++编码规范（中兴） | - | - |
| 前台软件编程细则（中兴） | - | - |
| 软件评审 | PMT Community | 2002 |
| UML 用户指南 | Grady Booch<br>James Rumbaugh<br>Ivar Jacobson | 2001年6月 |
| 重构——改善现有代码的设计 | Martin Fowler | 2003年8月 |
| 设计模式 | Erich Gamma 等 | 2000年9月 |

# C++成长篇

本篇归纳了一C++程序员成长中的各个阶段，以及踏入该阶段的最佳武林秘笈。本篇仅供各位大侠茶余饭后时拍砖用 。

这里只围绕**纯粹的C++程序设计语言**进行讨论。当然，要成为一个称职的程序员，计算机原理、操作系统、数据库等其它方面的专业知识也是十分重要的。

| 秘笈 | 作者 |
|---|---|
| 初入江湖——惨不忍睹 | |
| C++程序设计教程<br>— 或 —<br>C++语言程序设计（第二版） | 钱能<br><br>郑莉 董渊 |
| 小有名气——将就着用 | |
| Thinking in C++ 2nd edition | Bruce Eckel |
| 名动一方——在大是大非的问题上立场坚定 | |

| | |
|---|---|
| Effective C++(第二版) 和 More Effective C++ | Scott Meyers (Lostmouse、候捷 等译) |
| 天下闻名——正确的使用C++的每个特性 | |
| C++程序设计语言——特别版<br>— 和 —<br>ANSI/ISO C++ Professional Programmer's Handbook | Bjarne Stroustrup (裘宗燕 译)<br><br>Denny Kalev |
| 一代宗师——掌握通用程序设计思想 | |
| 范型编程与STL | Matthew H. Austem (候捷 译) |
| 超凡入圣——清楚C++的每个细节 | |
| ISO/IEC 14882: Programming Languages－C++ | ISO/IEC |
| 天外飞仙——透过C++的军大衣，看到赤裸裸的汇编码 | |
| GCC的源码烂熟于胸，有事没事的随便写个编译器玩玩～ | |

# 与我联系

如有任何建议或意见，请发邮件至：asbai@msn.com，或通过MSN：asbai@msn.com 与我联系。