SUNGARD 金仕达

编码规范

组织标准软件过程文档

文档标识

文档名称	编码规范
版本号	<3SPE-GC++-V1.00>
状况	☐ 草案 ☐ 评审过的 ☐ 更新过的 ☑ 定为基线的

文档修订历史

版本	日期	描述	文档所有者
1.00	2005-7-25	创建	李明辉

此版本文档的正式核准

姓名	签字	日期

分发控制

副本	接受人	机构

目 录

1. 版权声明 1
2. 概述 1
3. 语法高亮与字体 1
3.1 字体 1
3.2 语法高亮 2
4. 文件结构 3
4.1 文件头注释 3
4.2 头文件 5
4.3 内联函数定义文件 6
4.4 实现文件 6
4.5 文件的组织结构 7
5. 命名规则 9
5.1 类/结构 10
5.2 函数 10
5.3 变量 10
5.4 常量 11
5.5 枚举、联合、typedef 11
5.6 宏、枚举值 12
5.7 名空间 12
6. 代码风格与版式 12
6.1 类/结构 17
6.2 函数 21
6.3 变量、常量 27
6.4 枚举、联合、typedef 30
6.5 宏(尽量不使用，除非没有其它更好的方法) 32
6.6 名空间 32
6.7 异常 32
7. 版本控制 41
8. 英文版 41
9. 自动工具与文档生成 41
10. 术语表 41
11. 参考文献 42
12. C++成长篇 42
13. 附录 44
13.1 常用注释一览 44
13.1.1 文件头注释 44
13.1.2 标准类注释 44
13.1.3 标准函数注解 46
13.1.4 语句/函数组 46
13.1.5 语句块 47
13.1.6 分割带 47

13.2 常用英文注释一览 48
13.2.1 文件头注释 48
13.2.2 标准类注释 49
13.2.3 标准函数注解 50
13.2.4 语句/函数组 51
13.2.5 语句块 51
13.2.6 分割带 51
13.3 C/C++文件头例子 52
13.4 头文件例子 58
13.5 实现文件例子 59
13.6 内联函数定义文件例子 61
13.7 类/结构的风格与版式例子 62
13.8 函数的风格与版式例子 64
13.9 RTTI、虚函数和虚基类的开销分析及使用指导 67

1. 版权声明

2. 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。

VC++下编程要优先使用 C++自带的编码规则，保证同一性。第三方代码不按照编码规范进行修改。

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。

3. 语法高亮与字体

想让大家读到缩进、对其正确一致，而且不出现乱码的源文件，我们就要使用相互兼容的字体。

3.1 字体

规范如下：

使用等宽字体	由于非等宽字体在对其等方面问题多多，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：

	兼容性好：所有 Windows 平台都支持该字体 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。 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： 文字的边缘会有严重的凹凸感。 一些笔画的比例也会失调。 开启了柔化字体边缘后，还会使文字显得模糊不清。 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。 支持多语言：Fixedsys 是 UNICODE 字体，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。

3.2 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。缤纷的色彩可以在很大程度上帮助我们阅读那些晦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）

4. 文件结构

4.1 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>
**********************************************************/
```

相关文件和版本部分不作强制要求。新版本应该在前面。

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：

```
/*! @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>
**********************************************************************************
**********
* 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。

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

4.2 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

4.3 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，这是因为它通常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

4.4 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，除作者、时间和修改记录外不作强制要求。
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
临时代码	对未完成代码，加注释为// TODO ： 说明； 对暂时加入的代码，加注释为//FixMe ： 说明

实现改变如果不影响头文件，可以不修改相应头文件的注释，但必须要有修改指南。

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

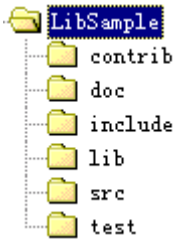
4.5 文件的组织结构

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：



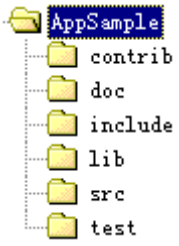
其中：

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。makefile 文件放到 src 文件夹中，可以按编译器分设子目录。不同平台用不同的后缀名，比如 linux 为 makefile.lnx。
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：



注：是否使用 makefile 文件夹不作要求。

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。makefile 文件放到 src 文件夹中，可以按编译器分设子目录。不同平台用不同的后缀名，比如 linux 为 makefile.lnx。
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。

5. 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。

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。
单词缩写	一个单词的缩写不能小于 3 个字母。Cmd、Id 等常用的缩写单词的缩写不计入缩写计数;一个命名里缩写不应超过 2 个（有些尽人皆知的如 IP、HTTP、FTP 等不计入缩写计数）。
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：″MinValue″ 和 ″MaxValue″，″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。如果不能避免，则数字前加下划线。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）。

5.1 类/结构

除了异常类等个别情况（不希望用户把该类看作一个普通的、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	每个单词的首字母要大写。
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："Analyzer"，"FastVector" ....

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" ....

5.2 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()" ....
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()" ....

5.3 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＋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

<table>
<tr><td></td><td>式，如：i，j，k，x，y，z ....</td></tr>
<tr><td>作用域前缀</td><td>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：
<table>
<tr><th>前缀</th><th>说明</th></tr>
<tr><td>无</td><td>局部变量</td></tr>
<tr><td>m_</td><td>类的成员变量（member）</td></tr>
<tr><td>sm_</td><td>类的静态成员变量（static member）</td></tr>
<tr><td>s_</td><td>静态变量（static）</td></tr>
<tr><td>g_</td><td>外部全局变量（global）</td></tr>
<tr><td>sg_</td><td>静态全局变量（static global）</td></tr>
<tr><td>gg_</td><td>进程间共享的共享数据段全局变量（global global）</td></tr>
</table>
类的成员变量作用域前缀“m_”可以为后缀“_”。但在同一项目中只能选用其中一种。
除非不得已，否则应该尽可能少使用全局变量。</td></tr>
<tr><td>推荐的组成形式</td><td>变量的名字应当使用"名词"或者"形容词＋名词"。例如："Code"，" State"，"MaxWidth" ....</td></tr>
</table>

5.4 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	常量名由全大写字母组成，单词间通过下划线来界定，如： DELIMITER，MAX_BUFFER ....

5.5 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、typedef 的命名	枚举、联合语句生成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 命名遵守 class struct 等类型的命名规则。

5.6 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 ....

5.7 名空间

C++名空间是“类”概念的一种退化（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。 如果碰到较长的名空间，为了简化程序书写，可以使用： `namespace new_name = old_long_name;` 语句为其定义一个较短的别名。

6. 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。

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。空行不得超

	过 3 行。 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。 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。 "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。
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 个空格符）处左对齐。 如果出现嵌套的 "{ }"，则使用缩进对齐。 例如： `void Function(int x)` `{`

	`` // ... } ```
最大长度	代码行最大长度宜控制在 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。
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。 例如： ``` if ((very_longer_variable1 >= very_longer_variable2) && (very_longer_variable3 <= very_longer_variable4) && (very_longer_variable5 <= very_longer_variable6)) { dosomething(); } ```
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"[]"、"."、"->"这类操作符前后不加空格。 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();
```

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：

```
char* name;
```

<table>
<tr><td></td><td>

```
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 -> 指针或引用类型的定义和声明</td></tr>
<tr><td>注释</td><td>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。
注释应避免使用“/*”。</td></tr>
<tr><td>与常量的比较</td><td>在与宏、常量进行 ″==″, ″!=″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
{
    // ...
}
```
</td></tr>
<tr><td>为增强代码的可读性而定义的宏（不作要求）</td><td>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：
<table>
<tr><th>宏</th><th>说明</th></tr>
<tr><td>NOTE</td><td>需要注意的代码</td></tr>
<tr><td>TODO</td><td>尚未实现的接口、类、算法等</td></tr>
</table>
</td></tr>
</table>

	TODO class CMyClass; TODO void Function(void); FOR_DBG cout << "...";

6.1 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" 关键字左对其。

```
/*! @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
{
    // ...
};
```

■ 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：

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

■ 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’关键字左对其。

■ 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);
};
```

■ 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;
    // ...
};
```

■ 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制表符（4 个半角空格）的缩进。

例如：

```
CXXX::CXXXX(IN int A, IN bool B)
: m_A(nA), m_B(B)
{
// ...
};
```

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）。

例如：

```
// ...

class CXXXX : public CAA, public CBB
{
// ...
    CYY m_A;
    CZZ m_B;  // m_iA 必须在 m_iB 之前声明
};
CXXX::CXXXX(IN int A, IN int B, IN bool C)
: CAA(A), CBB(B), m_A(C), m_B(m_A) // 先基类，后成员，
                                   // 分别按照声明顺序书写
{
 // ...
};
```

■ 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

6.2 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，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>（如果程序员负责的最小单位是文件，此项可选）
</PRE>
********************************************************************************/
函数原型
{
// ...
}
```

函数名到返回值作为强制要求，抛出异常后的部分作为推荐要求。

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 1 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
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。

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

参数命名规范与变量的命名规范相同

成员函数的存储类	由于C++语言的限制，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
{
    // ...
}
```

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 Lock(*sm_SELock);` `// ==========` `//=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 LastLoadedModule = NULL;` `if (LastLoadedModule == modAddr)` `{` `return; // 相同，忽略这次调用` `}` `LastLoadedModule = modAddr;` `//==========`

```
        {
            return;
        }
        if (CHookProc::sm_ByPassModTbl.find(ModName)
            != CHookProc::sm_ByPassModTbl.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_ProcTbl.begin(); p != sm_ProcTbl.end(); ++p)
        {
            p->HookOneModule(modAddr);
        }
    } // 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 150 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。

调用系统 API	所有系统 API 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()))` `{` `// ...` `}`
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

6.3 变量、常量

<table>
<tr><td>声明格式</td><td>变量、常量的声明格式如下：

<pre>[存储类] 类型 变量名;</pre>其中：
以 "[ ]" 括住的为可选项目；
变量声明后必须初始化；
"存储类" 的说明见下文。</td></tr>
<tr><td>定义格式</td><td>变量、常量的定义格式如下：
<pre>[存储类] 类型 变量名 = 初始值;</pre>其中：
以 "[ ]" 括住的为可选项目。
"存储类" 的说明见下文
定义时一定要进行初始化</td></tr>
<tr><td>存储类</td><td>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）。Register 不推荐使用。</td></tr>
<tr><td>成员变量的存储类</td><td>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。
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。
例如：
<pre>/*static*/ int CThread::sm_PID = 0;</pre></td></tr>
<tr><td>指针或引用类型的定义和声明</td><td>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：

<pre>int* pn1,
   * pn2 = NULL,
   * pn3;</pre></td></tr>
</table>

	`` char* pc1; char* pc2; char* pc3; // 错误的写法： int* pn11, *pn12, *pn13; ```
指针的指针	指向指针的指针，用 Int** x 表示
常指针和指针常量及常引用	声明/定义一个常指针或常引用（指向常量的指针）时，"const" 关键字一律放在类型与*之间。 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。 例如： ``` char const* pc1; // 常指针 string const& str; //常引用 char* const pc2; // 指针常量 char const* const pc3; // 常指针常量 // 错误的写法： const char* 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。 例如： ``` //! 当前进程的 ID static int sg_nPID = 0; //! 分割符 static const char* DTR = "\\/"; ```
传入参数	非简单类型推荐用"const&"。
类型转换	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。其中：

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" 等等。这种转换有可能产生移植性方面的问题，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。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

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。

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通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" 则是一个例外。 参见：RTTI、虚函数和虚基类的开销分析和使用指导

6.4 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） enum|union 名称 { 内容 }; ``` 例如： ``` //! 服务的状态 enum SRVSTATE { SRV_INVALID = 0, ```

```
        SRV_STOPPING,

        SRV_STOPPED

    };

    //! 32 位整数

    union INT32

    {

        unsigned char    cByte[4];

        unsigned short   nShort[2];

        unsigned long    nFull;

    };
```

typedef 的定义格式

typedef 的定义格式为：

```
//! 说明（可选）
```

例如：

	typedef vector<string> VSTR;

6.5 宏(尽量不使用，除非没有其它更好的方法)

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： #define PLUS(x,y) x+y 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"。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： #define PLUS(x,y) (x+y) #define SAFEDELETE(x) {delete x; x=0}

6.6 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。 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。

6.7 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。 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。 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 。
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： `//! 使用 goto 进行错误处理的例子` `bool Function(void)` `{` `int nCode, i;` `bool r = false;`

```
    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

变量必须在 "goto" 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："onOp1Err"，"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：

```
//! 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
        {
```

```
                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：

错误处理代码和其它代码严重混杂，使程序非常难以理解

无法进行分类错误处理

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，还有一种更为糟糕的错误处理风格——直接在出错的位置完成错误处理：

```
//! 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
    {
```

```
    }
    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。使用异常做出错处理时，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并为创建而不被认可，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流产的婴儿发户口一样。但是这有时也会产生一些问题，由此应绝对避免构造函数中的异常。
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： 对象被正常析构时 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：

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

CSample::~CSample()
{
// ...
```

	} } void Function(void) { CSample iTest; throw false; // 可以，iTest.~CSample()不会抛出异常 }
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)
{
    // ...
}
// 处理其它标准库错误
catch (const exception& err)
{
    // ...
}
// 处理所有其它错误
catch (...)
{
    throw;  // 重新抛出
}
```

异常和效率	对于几乎所有现代编译器来说，在不抛出异常的情况下，异常处理的实现在运行时不会有任何额外开销，也就是说：正常情况下，异常机制比传统的通过返回值判断错误的开销还来得小。 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来得大一些。不过错误处理代码通常不会频繁调用，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。

	更多关于效率的讨论，参见：RTTI、虚函数和虚基类的开销分析和使用指导

7. 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 "0.1"，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。

8. 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

9. 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

10. 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
IDE	集成开发环境

11. 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
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

12. 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。

书名	作者
初入江湖——惨不忍睹	
C++程序设计教程 — 或 — C++语言程序设计（第二版）	钱能 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	Bjarne Stroustrup (裘宗燕 译)
一代宗师——掌握通用程序设计思想	
范型编程与 STL	Matthew H. Austem (侯捷 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～	

13. 附录

13.1 常用注释一览

13.1.1 文件头注释

```
/*! @file
**********************************************************************************
<PRE>
模块名          :
文件名          :
相关文件        :
文件实现功能    :
作者            : <xxx>
版本            : 1.0
----------------------------------------------------------------------------------
备注            :
----------------------------------------------------------------------------------
修改记录 :
日 期          版本    修改人          修改内容
YYYY/MM/DD     1.0     <xxx>            创建
</PRE>
**********************************************************************************

* 版权所有(c) YYYY, <xxx>, 保留所有权利

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

13.1.2 标准类注释

以下定义的各种成员类型可以根据实际需要增删。

```
/*! @class
**********************************************************************************
```

```
<PRE>
类名称    :
功能      :
--------------------------------------------------------------------------------
作者      :<xxx>
</PRE>
*********************************************************************************/
class CXXX
{
public:
////////////////////////////////////////////////////////////////////////// 类型定义

public:
////////////////////////////////////////////////////////////////////// 构造、析构、初始化

public:
////////////////////////////////////////////////////////////////////////// 虚函数

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

public:
////////////////////////////////////////////////////////////////////////// 静态方法

protected:
////////////////////////////////////////////////////////////////////////// 内部方法

private:
////////////////////////////////////////////////////////////////////////// 私有类型定义

private:
////////////////////////////////////////////////////////////////////////// 私有方法

private:
////////////////////////////////////////////////////////////////////////// 属性
```

```
private:
////////////////////////////////////////////////////////////////////////// 静态属性

private:
////////////////////////////////////////////////////////////////////////// 禁用的方法

};
```

13.1.3 标准函数注解

```
/*! @function
********************************************************************************************
<PRE>
函数名    :
功能      :
参数      :
返回值    :
--------------------------------------------------------------------------------
作者      : <xxx>
</PRE>
********************************************************************************************/
```

13.1.4 语句/函数组

```
// [[ 这组语句或函数的功能

    ...

// ]] 这组语句或函数的功能
```

13.1.5 语句块

```
//
=============================================================================
// = 说明由此以下一系列语句执行的操作
```

13.1.6 分割带

```
//
###############################################################################
//
###############################################################################


//
###############################################################################
// ########################################################### 本地数据和
函数
    ...
// ########################################################### 本地数据和
函数
//
###############################################################################


//
###############################################################################
// ########################################################### Cxxx 类成员
定义
    ...
// ########################################################### Cxxx 类成员
定义
//
```

```
################################################################################


//
################################################################################
// ############################################################################ 其它部分开始
    ...
// ############################################################################ 其它部分结束
//
################################################################################
```

13.2 常用英文注释一览

13.2.1 文件头注释

```
/*! @file
**********************************************************************************************
<PRE>
Module       :
File         :
Related Files:
Intro        :
Author       : <xxx>
Version      : 1.0
----------------------------------------------------------------------------------
Notes        :
----------------------------------------------------------------------------------
Change History :
Date         Version  Changed By      Changes
YYYY/MM/DD   1.0      <xxx>           Create
</PRE>
**********************************************************************************************
```

```
* Copyright(c) YYYY, by <xxx>, All rights reserved

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

13.2.2 标准类注释

以下定义的各种成员类型可以根据实际需要增删。

```
/*! @class
*********************************************************************************
<PRE>
Class    :
Desc     :
Exception:

--------------------------------------------------------------------------------
Author   : <xxx>
</PRE>
*********************************************************************************/
class CXXX
{
public:
////////////////////////////////////////////////////////////////////// defines

public:
////////////////////////////////////////////////////////////////////// big three

public:
////////////////////////////////////////////////////////////////////// virtual funcs

public:
////////////////////////////////////////////////////////////////////// public funcs

public:
////////////////////////////////////////////////////////////////////// static funcs
```

```
protected:
////////////////////////////////////////////////////////////////// internal funcs

private:
////////////////////////////////////////////////////////////////// private defines

private:
////////////////////////////////////////////////////////////////// private funcs

private:
////////////////////////////////////////////////////////////////// properties

private:
////////////////////////////////////////////////////////////////// static properties

private:
////////////////////////////////////////////////////////////////// disabled method

};
```

13.2.3 标准函数注解

```
/*! @function
********************************************************************************
<PRE>
Function  :
Desc      :
Params    :
Return    :
Exception :
--------------------------------------------------------------------------------
Author    : <xxx>
</PRE>
********************************************************************************/
```

13.2.4 语句/函数组

```
// [[ Group Title

    ...

// ]] Group Title
```

13.2.5 语句块

```
//
=============================================================================
// = Action Title
```

13.2.6 分割带

```
//
################################################################################
//
################################################################################


//
################################################################################
// ############################################################ Local data and
funcs
    ...
// ############################################################ Local data and
funcs
//
################################################################################


//
################################################################################
```

```
//  ##########################################################################  class
CXXX
    ...
//  ##########################################################################  class
CXXX
//
##############################################################################


//
##############################################################################
//   ##########################################################################  Other
Title
    ...
//   ##########################################################################  Other
Title
//
##############################################################################
```

13.3 C/C++文件头例子

```
/*! @file
**********************************************************************************
<PRE>
模块名          : 白杨 string 扩展库
文件名          : stringEx.h
相关文件        : stringEx.cpp
文件实现功能    : C++ string 类功能扩展
作者            : 白杨
版本            : 1.10
----------------------------------------------------------------------------------
备注            : 在 string 的基础上构建，未添加任何数据成员
----------------------------------------------------------------------------------
修改记录 :
日  期          版本            修改人          修改内容
```

2003/08/27	1.0	白杨	创建
2003/12/29	1.1	白杨	新增正则表达式、operator<<操作符等功能
2004/01/28	1.2	白杨	新增部分函数
2004/02/02	1.3	白杨	新增部分函数
2004/03/11	1.6	白杨	新增部分函数
2004/03/29	1.7	白杨	新增 ref(), set_size()等函数
2004/06/28	1.8	白杨	移植到 namespace BaiY
2004/07/15	1.9	白杨	新增换码/还原函数；为所有正则函数增加 perl 风格的字符类
2004/07/19	1.10	白杨	新增 vformat()函数；重写、改进 format()函数

</PRE>

****

* 版权所有(c) 2003, 2004, 白杨, 保留所有权利

==========

* to_str、from_str、to_wstr、from_wstr 等函数只能做标准^ASCII^字符集的宽/窄字符转换

==========

* format 和 vformat 函数所能处理的最大字符串长度限制为 MAX_STRING，如需临时改变，可
  以使用：format(maxLength, fmt, ...)函数和 vformat(maxLength, fmt, arglist)函数

==========

* 支持如下 ostream 风格的流式输入操作，可以方便地用于将个类数据追加到串：
  operator<<(char)
  operator<<(BYTE)
  operator<<(int)
  operator<<(unsigned)

```
    opeartor<<(long)
    operator<<(unsigned long)
    operator<<(float)
    operator<<(double)
    operator<<(const stringEx&)
    operator<<(const char*)
    operator<<(const BYTE*)

    operator<<(wchar_t)
    operator<<(wstringEx)



=============================================================================
```

* 支持兼容 IEEE 1003.2-1992（“POSIX.2”）标准，并且能够正确处理包含 0 值之字符串的正则表达式功能。

  同时也支持 perl 风格的字符类。

  为此提供了以下函数：

  ```
    regex_find();
    regex_find_replace();
    regex_find_replace_all();

    regex_comp();
    regex_find_with_comp();
    regex_free();
  ```

  - 也可以通过定义宏“#define BaiY_USE_REGEX 0”禁用此功能。

  - 为了提高搜索效率，避免大量重复的表达式编译：regex_find()会自动缓存最近一次使用的正则表达式（缓存的内容可以通过用空表达式调用 regex_find()函数释放）。如果需要高效率的交替搜索多个表达式，请使用 regex_comp()、regex_find_with_comp()、regex_free()函数。

  - 对于 wstringEx，还保证能够正确识别 UNICODE 字符类（如空白符，大小写等）。所以在

构建一个正则表达式的时候，应该尽量使用如[:blank:]、[:upper:]这样的字符类，而不是如[A-Z]这样的自定义集合。

- 以下是所有 POSIX 字符类描述，及其对应 perl 字符类

POSIX 类	perl 类	描述
[:alnum:]		字母和数字
[:alpha:]	\a	字母
[:lower:]	\l	小写字母
[:upper:]	\u	大写字母
[:blank:]		空白字符（空格和制表符）
[:space:]	\s	所有空格符（比[:blank:]包含的范围广）
[:cntrl:]		不可打印的控制字符（退格、删除、警铃...）
[:digit:]	\d	十进制数字
[:xdigit:]	\x	十六进制数字
[:graph:]		可打印的非空白字符
[:print:]	\p	可打印字符
[:punct:]		标点符号

- 此外，perl 还有以下特殊字符类：

perl 类	等效 POSIX 表达式	描述
\o	[0-7]	八进制数字
\O	[^0-7]	非八进制数字
\w	[[:alnum:]_]	单词构成字符
\W	[^[:alnum:]_]	非单词构成字符
\A	[^[:alpha:]]	非字母
\L	[^[:lower:]]	非小写字母
\U	[^[:upper:]]	非大写字母
\S	[^[:space:]]	非空格符
\D	[^[:digit:]]	非数字
\X	[^[:xdigit:]]	非十六进制数字
\P	[^[:print:]]	非可打印字符

- POSIX 字符类必须在方括号中工作，相反，perl 字符类则必须在方括号外工作。

- 在允许 perl 风格的正则搜索和替换中，还可以使用以下特殊字符换码序列：

  \r - 回车
  \n - 换行
  \b - 退格
  \t - 制表符
  \v - 垂直制表符
  \″ - 双引号
  \’ - 单引号

==========

* 支持以给定的换码符和换码表进行换码和还原操作，为此提供了 2 个函数：

```
stringEx& escape_translate(IN const char cEscape,
                           IN const stringEx vstTransTable[][2]);

stringEx& escape_restore(IN const char cEscape,
                         IN const stringEx vstTransTable[][2],
                         IN bool bEraseUnknownEscapeSequence = true,
                         IN bool bUntouchEscEsc = false);
```

- 使用范例：

```
// 定义换码表
const stringEx tbl[][2] = {
    { "<", "LS" },
    { ">", "GT" },
    { "=", "EQ" },
    { "",  "" }   // 换码表结束符
};
// 定义换码符
const char ESC = '%';
```

```
stringEx test = "<TEST>aaaa%==%bbbb</TEST>";
cout << test.escape_translate( ESC, tbl ) << endl;
cout << test.escape_restore( ESC, tbl ) << endl;
```

程序输出：

```
%LSTEST%GTaaaa%%%EQ%EQ%%bbbb%LS/TEST%GT
<TEST>aaaa%==%bbbb</TEST>
```

==========

* 为了增加与 C 函数一起使用的效率，提供了以下方法：

```
char_type* ref( void )
void set_size( IN size_t len )
void set_length( IN size_t len )
```

- 可以这样使用：

```
stingEx a;
const char* pcStr = "test";
len = strlen(pcStr);

a.reserve(len);
strcpy(a.ref(), pcStr);
a.set_size(len);
```

- 但是如果有另一个 stringEx 类对象与“a”引用同一块存储的话，则会导致两个对象的内容同时被改变，为防止这种情况发生，应该这样使用它们：

```
stringEx a,b;
a = "something";
b = a;                    // b 和 a 引用同一块内存
const char* pcStr = "test";
len = strlen(pcStr);

a.resize(len);            // 对象 a 发生写时拷贝
strcpy(a.ref(), pcStr);
```

不过很显然，前一种方法效率较高。

- set_length() 方法是 set_size 方法的一个别名

- 使用以上函数时需要在 basic_string 模板类中添加两个保护成员函数，具体请参看“安装事项”文档。

**********/

13.4 头文件例子

```
/*! @file
**********************************************************************************
<PRE>
模块名          : 代码风格示例模块
文件名          : sample.h
相关文件        : sample.cpp, sample.inl
文件实现功能    : 演示正确的代码风格
作者            : 白杨
版本            : 1.0
----------------------------------------------------------------------------------
备注            : -
----------------------------------------------------------------------------------
修改记录 :
日  期          版本          修改人          修改内容
2004/07/22      1.0           白杨            创建
</PRE>
**********************************************************************************

* 版权所有(c) 2004, 白杨, 保留所有权利

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

```
#ifndef __BaiY_Sample_H__      // 防止 sample.h 被重复引用
#define __BaiY_Sample_H__

#include <stdlib.h>        // 引用标准库的头文件
// ...

#include "../common.h"     // 引用当前工程中的头文件
// ...


extern
void MyFunction( void );    // 全局函数声明


class CMyClass          // 类/结构声明
{
    // ...
};

#include "sample.inl" // 定义所有 CMyClass 中的内联函数

#endif //__BaiY_Sample_H__
```

13.5 实现文件例子

```
/*! @file
*********************************************************************************
<PRE>
模块名          : 代码风格示例模块
文件名          : sample.cpp
相关文件        : sample.h, sample.inl
文件实现功能    : 演示正确的代码风格
作者            : 白杨
版本            : 1.0
```

```
--------------------------------------------------------------------------------
备注            : -
--------------------------------------------------------------------------------
修改记录 :
日  期          版本            修改人          修改内容
2004/07/22      1.0             白杨            创建
</PRE>
*********************************************************************

* 版权所有(c) 2004, 白杨, 保留所有权利

*********************************************************************/


#include <sample.h>   // 引用相关的头文件

#include <vector>     // 引用仅用于实现的头文件

// ###########################################################################
// ############################################################ 本地数据和函数

static int Internal = 1;

static
void InternalFunc( void )
{
    // ...
}

// ############################################################ 本地数据和函数
// ###########################################################################


// ###########################################################################
// ############################################################ 外部数据和函数
```

```
extern int Code = 0; // 数据定义

extern
void MyFunction( IN int Arg )  // 函数实现体
{
    // ...
}

// ########################################################## 外部数据和函数
// ########################################################################
```

13.6 内联函数定义文件例子

```
/*! @file
**********************************************************************************
<PRE>
模块名          : 代码风格示例模块
文件名          : sample.inl
相关文件        : sample.h
文件实现功能    : 演示正确的代码风格
作者            : 白杨
版本            : 1.0
----------------------------------------------------------------------------------
备注            : -
----------------------------------------------------------------------------------
修改记录 :
日  期          版本            修改人          修改内容
2004/07/22      1.0             白杨            创建
</PRE>
**********************************************************************************

* 版权所有(c) 2004, 白杨, 保留所有权利
```

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


inline
void CMyClass::MyInlineFunc1( void )
{
    // ...
}


inline
void CMyClass::MyInlineFunc2( void )
{
    // ...
}

// ...
```

13.7 类/结构的风格与版式例子

```
/*! @class
*********************************************************************************************
<PRE>
类名称   : CSem
功能     : 封装信号量相关操作
异常类   : CSem::Exp
---------------------------------------------------------------------------------------------
备注     : -
典型用法 : -
---------------------------------------------------------------------------------------------
作者     : 白杨
</PRE>
*********************************************************************************************/
class CSem
{
```

```
public:
////////////////////////////////////////////////////////////////////// 类型定义

    //! 异常类
    struct Exp : public byExp
    {
        enum
        {
            ERR_CREATE = 1,
            ERR_OPEN
        };

        Exp(IN UINT Err = 0, IN const stringEx& Description = "")
            : byExp(Err, Description)
            {}
    };

public:
////////////////////////////////////////////////////////////// 构造、析构、初始化

    //! 构造函数 - 创建一个新的信号量
    explicit CSem(IN long InitCount,
                  IN long MaxCount = 1,
                  IN LPCTSTR pstrName = NULL) throw(Exp);

    //! 构造函数 - 打开一个已创建的命名信号量
    explicit CSem(IN LPCTSTR pstrName = NULL) throw(Exp);

    ~CSem();

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

    //! 申请资源
    bool p(IN DWORD TimeOut = INFINITE);
```

```
    //! 归还资源
    bool v(IN long ReleaseCount = 1);

private:
////////////////////////////////////////////////////////////////////////////// 属性
    HANDLE m_Semaphore;

private:
////////////////////////////////////////////////////////////////////////////// 禁用的方法
    CSem(IN const CSem& rhs);
    CSem& operator=(IN const CSem& rhs);
};
```

13.8 函数的风格与版式例子

```
/*! @function
********************************************************************************
<PRE>
函数名    : GetAllSubDirs
功能      : 获得所有符合条件的子目录列表
参数      : [OUT] vResult       : 用来储存结果
            [IN]  dirPattern    : 要查找的子目录通配符
            [IN]  includeSubdir : 是否包含子目录
            [IN]  includeHidden : 是否包含隐含目录
            [IN]  includeDot    : 是否包含“.”和“..”系统目录
返回值    : 符合条件的子目录数量
抛出异常  : -
--------------------------------------------------------------------------------
备注      : 无论是否成功，vResult 中的原内容都将被清空
典型用法  : -
--------------------------------------------------------------------------------
作者      : 白杨
</PRE>
********************************************************************************/
```

```
ULONG CDir::GetAllSubDirs(OUT VSTR& vResult,
                          IN tstringEx dirPattern /*= byT("*")*/,
                          IN bool includeSubdir    /*= false*/,
                          IN bool includeHidden    /*= true*/,
                          IN bool includeDot       /*= false*/) const
{
    // ========================================================
    // = 初始化
    vResult.clear();
    if (dirPattern.empty() || !IsValid())
    {
        return 0;
    }

    WIN32_FIND_DATA FindFileData;
    HANDLE hFind;
    tstringEx Pattern = m_basedir + DELIMITER + dirPattern;

    // ============================================================
    // = 匹配子目录
    hFind = ::FindFirstFile(Pattern.c_str(), &FindFileData);
    if(INVALID_HANDLE_VALUE == hFind)
    {
        return 0;
    }

    int n;
    tstringEx stDir;
    if (-1 != (n=dirPattern.find_last_of(ALLDELIMITERS)))
    {
        stDir = m_basedir + DELIMITER + dirPattern.substr(0, n) + DELIMITER;
    }
    else
    {
        stDir = m_basedir + DELIMITER;
```

```
}

do
{
    if (!(FindFileData.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
    {
        goto findnext;
    }

    if (IsDotDir(FindFileData.cFileName) && !includeDot)
    {
        goto findnext;
    }

    if ((FindFileData.dwFileAttributes & FILE_ATTRIBUTE_HIDDEN)
        && !includeHidden)
    {
        goto findnext;
    }

    // 递归搜索子目录
    if (includeSubdir && !IsDotDir(FindFileData.cFileName))
    {
        VSTR rr;
        tstringEx rdir;
        if (-1 != n)
        {
            rdir = dirPattern.substr(0, n)+DELIMITER+FindFileData.cFileName
                    +DELIMITER+dirPattern.substr(n+1, -1);
        }
        else
        {
            rdir = FindFileData.cFileName+DELIMITER+dirPattern;
        }
```

```
            GetAllSubDirs(rr, rdir, includeSubdir, includeHidden, includeDot);
            const ULONG count = rr.size();
            for (ULONG i=0; i<count; ++i)
            {
                vResult.push_back(rr[i]);
            }
        } // if (includeSubdir && !IsDotDir(FindFileData.cFileName))

        vResult.push_back(stDir + FindFileData.cFileName);

findnext:
        if (!::FindNextFile(hFind, &FindFileData))
        {
            break;
        }
    } while(true);

    ::FindClose(hFind);
    return vResult.size();
}
```

13.9 RTTI、虚函数和虚基类的开销分析及使用指导

“在正确的场合使用恰当的特性” 对称职的 C++程序员来说是一个基本标准。想要做到这点，首先要了解语言中每个特性的实现方式及其开销。本文主要讨论相对于传统 C 而言，对效率有影响的几个 C++新特性。

C++引入的额外开销体现在以下两方面：

■ 编译时开销

模板、类层次结构、强类型检查等新特性，以及大量使用了这些新特性的 C++模板、算法库都明显地增加了 C++编译器的负担。但是应当看到，这些新机能在不增加程序执行效率的前提下，明显降低了广大 C++程序员的工作量。

用几秒钟的 CPU 时间换取程序员几小时的辛勤劳动，附带节省了日后 debug 和维护代码的时间，这点开销当算超值。

当然，在使用这些特性的时候，也有不少优化技巧。比如：编译一个大型软件时，几条显式实例化指令就可能使编译速度提高几十倍；恰当地组合使用部分专门化和完全专门化，不但可以最优化程

序的执行效率，还可以让同时使用多种不同参数实例化模板的软件体积显著减小……

■ 运行时开销

运行时开销恐怕是程序员最关心的问题之一了。相对与传统 C 程序而言，C++中有可能引入额外运行时开销的新特性包括：

虚基类

虚函数

RTTI（dynamic_cast 和 typeid）

异常

对象的构造和析构

关于其中第四点 异常，对于几乎所有的现代编译器来说，在正常情况（未抛出异常）下，try 块中的代码执行效率和普通代码一样高，而且由于不再需要使用传统上通过返回值或函数调用来判断错误的方式，代码的实际执行效率还会进一步提高。抛出和捕捉异常的效率也只是在某些情况下会高于函数返回和函数调用的效率，何况对于一个编写良好的程序，抛出和捕捉异常的机会应该不多。关于异常使用的详细讨论，参见：C++编码规范正文。

而第五点对象的构造和析构开销也不总是存在。对于不需要初始化/销毁的类型，并没有构造和析构的开销，相反对于那些需要初始化/销毁的类型来说，即使用传统的 C 方式实现，也至少需要与之相当的开销。

对能够真正用于开发的编译器而言，C++本身就是使用 C/汇编 加以千锤百炼的优化才实现的。也就是说，想用 C 甚至汇编更高效地实现某个 C++特性几乎是不可能的。要是真能做到这一点的话，大侠就应该去写个编译器造福广大程序员才对～

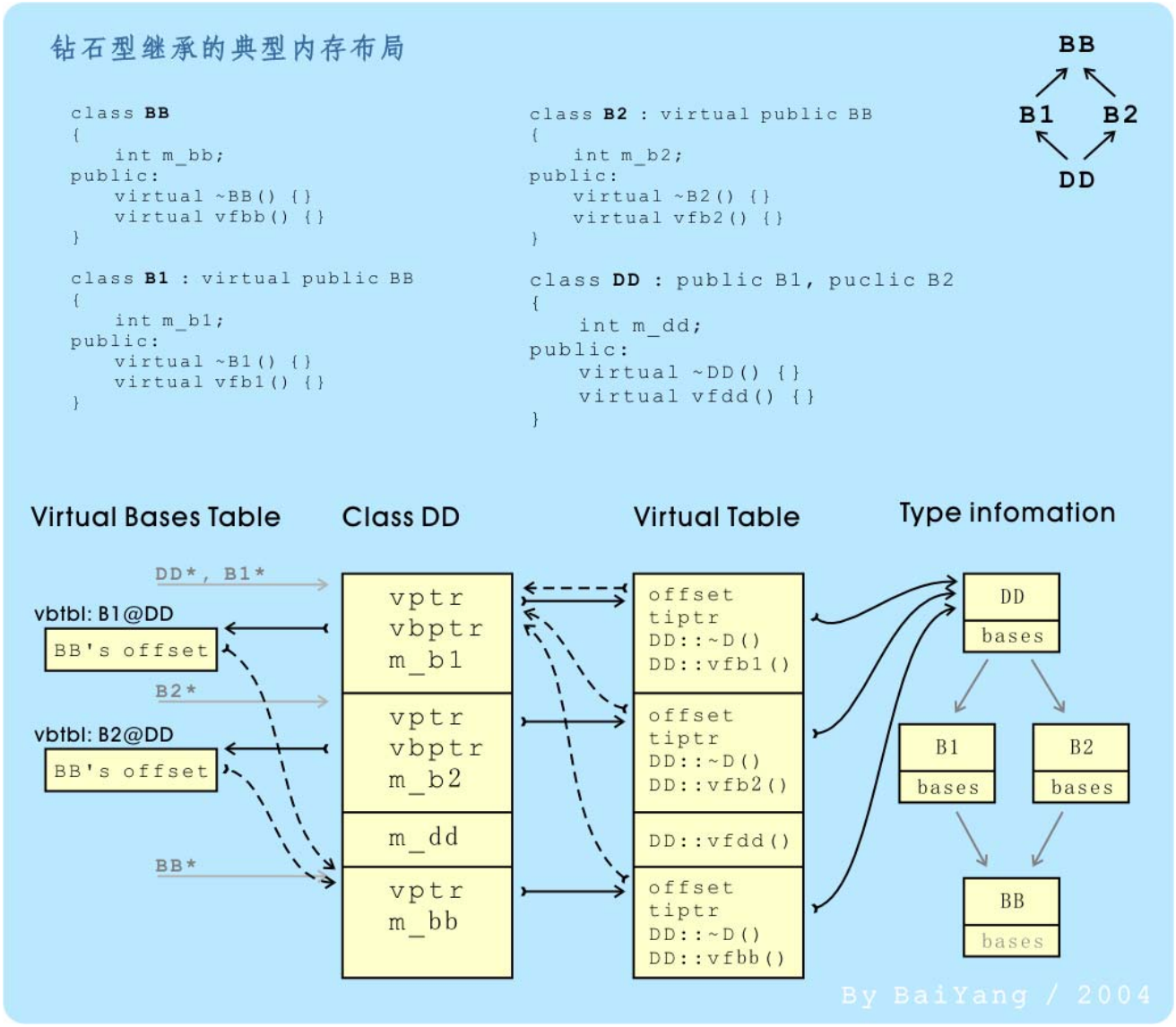
C++之所以比 C“低效”，其根本原因在于：由于对某些特性的实现方式及其开销不够了解，导致程序员在错误的位置使用了错误的特性。而这些错误基本都集中在：

把异常当作另一种流程控制机制，而不是仅将其用于错误处理中

滥用或不正确地使用 RTTI、虚函数和虚基类机制

其中第一点上文已经 讲过，下面讨论第二点。

为了说明 RTTI、虚函数和虚基类的实现方式，首先给出一个实例及其具体实现（为了便于理解，这里故意忽略了一些无关紧要的优化）：



图中虚箭头代表偏移，实箭头代表指针

由上图得到每种特性的运行时开销如下：

特性	时间开销	空间开销
RTTI	几次整形比较和一次取址操作（可能还会有 1、2 次整形加法）	每类型一个 type_info 对象（包括类型 ID 和类名称），

		典型情况下小于 32 字节
虚函数	一次整形加法和一次指针引用	每类型一个虚表，典型情况下小于 32 字节 每对象若干个（大部分情况下是一个）虚表指针，典型情况下小于8字节
虚基类	从直接虚继承的子类（例如上图中的 "B1" 和 "B2"，但不包括 "DD" ）中访问虚基类的数据成员或其虚函数时，将增加两次指针引用（大部分情况下可以优化为一次）和一次整形加法。	每类型一个虚基类表，典型情况下小于 16 字节 每对象若干虚基类表指针，典型情况下小于8字节
* 其中“每类型”或“每对象”是指用到该特性的类型/对象。对于未用到这些功能的类型及其对象，则不会增加上述开销		

可见，关于老天爷“饿时掉馅饼、睡时掉老婆”等美好传说纯属谣言。但凡人工制品必不完美，总有设计上的取舍，有其适应的场合也有其不适用的地方。

C++中的每个特性，都是从程序员平时的生产生活中逐渐精化而来的。在不正确的场合

使用它们必然会引起逻辑、行为和性能上的问题。对于上述特性，应该只在必要、合理的前提下才使用。

"dynamic_cast" 用于在类层次结构中漫游，对指针或引用进行自由的向上、向下或交叉强制。"typeid" 则用于 获取一个对象或引用的确切类型,与 "dynamic_cast" 不同，将 "typeid" 作用于指针通常是一个错误， 要得到一个指针指向之对象的 type_info，应当先将其解引用（例如：typeid(*p)）。

一般地讲，能用虚函数解决的问题就不要用 "dynamic_cast"，能够用 "dynamic_cast" 解决的就不要用 "typeid"。比如：

```
void rotate(IN const CShape& iS)
{
    if (typeid(iS) == typeid(CCircle))
    {
        // ...
    }
    else if (typeid(iS) == typeid(CTriangle))
    {
        // ...
    }
    else if (typeid(iS) == typeid(CSqucre))
    {
        // ...
    }

    // ...
}
```

以上代码用 "dynamic_cast" 写会稍好一点,当然最好的方式还是在其中每个类里定义名为 "rotate" 的虚函数。

虚函数是 C++运行时多态特性中开销最小，也最常用的机制。虚函数的好处和作用这里不再多说，应当注意在对性能有苛刻要求的场合，或者需要频繁调用，对性能影响较大的地方（比如每秒钟 要调用上百次的事件处理函数）要慎用虚函数。

作为一种支持多继承的面向对象语言,虚基类有时是保证类层次结构正确的一种必不可少的手段。在需要频繁使用基类提供的服务，又对性能要求较高的场合，应该避免使用虚基类。

在基类中没有数据成员的场合，也可以解除使用虚基类。例如，在上图中，如果类 "BB" 中不存在数据成员，那么 "BB" 就可以作为一个普通基类分别被 "B1" 和 "B2" 继承。

这样的优化在达到相同效果的前提下，解除了虚基类引起的开销。不过这种优化也会带来一些问题：从 "DD" 向上强制到 "BB" 时会引起歧义；破坏了类层次结构的逻辑关系。上述特性的空间开销一般都是可以接受的，当然也存在一些特例，比如：在存储布局需要和传统 C 结构兼容的场合、在考虑对齐的场合、在需要为一个本来很小的类同时实例化许多对象的场合等等。