C++编程规范

[SGDP编程规范(C++语言) 1](#_Toc303167696)

[1. 前言 2](#_Toc303167697)

[1.1. 编码三准则 2](#_Toc303167698)

[1.2. 环境设置 2](#_Toc303167699)

[2. 文件结构 2](#_Toc303167700)

[3. 程序和格式 3](#_Toc303167701)

[3.1. 空行和空格 3](#_Toc303167702)

[3.2. 对齐和缩进 4](#_Toc303167703)

[3.3. 长行拆分 5](#_Toc303167704)

[3.4. 注释 5](#_Toc303167705)

[3.5. 类版式 6](#_Toc303167706)

[4. 命名规则 9](#_Toc303167707)

[5. 表达式和基本语句 11](#_Toc303167708)

[6. 常量 11](#_Toc303167709)

# 前言

## 编码三准则

正确性 > 可靠性 > 健壮性 > 可读性 > 执行效率。代码主要是给人看的，顺带才是给机器编译执行。代码是写给别人看的，然后才给自己看。

## 环境设置

|  |  |
| --- | --- |
| 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，在显示器中呈现的影像更为清晰。 * 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。 |

# 文件结构

1. 所有文件(.h/.inl/.cpp)都应该有统一的文件头。

// SHANDA GAMES PROPRIETARY INFORMATION

//

// This software is supplied under the terms of a license agreement or

// nondisclosure agreement with Shanda Games and may not

// be copied or disclosed except in accordance with the terms of that

// agreement.

//

// Copyright (c) 1999-2010 Shanda Games.

// All Rights Reserved.

//

// Shanda Games Limited, Shanghai, China, 201203

// <http://www.shandagames.com>

#ifndef \_THISFILE\_H\_

#define \_THISFILE\_H\_

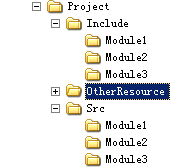
//

//文件内容

//

#endif //#ifndef \_THISFILE\_H\_

1. 为避免头文件被重复包含，必须在头文件中使用ifndef/define/endif产生预处理块。预处理块紧跟在文件头注释之后，包含除文件头之外的所有文件内容。
2. 代码中需要使用到的基本数据类型定义，全部使用“sdtype.h”中定义的类型，不直接使用C++基本类型保留关键字。
3. 具有不同功能的文件，应该按照功能或者模块，存放在不同目录。对外头文件可以按照代码结构同样存放。如下图所示



1. 面向跨平台开发，工程中所有目录以及源代码文件必须使用英文小写命名。
2. 面向未来64位编程的兼容性，代码中不允许将指针类型强制转换为UINT32等其他类型，如果有转换的需要，只能强制转换为UINT64类型。

# 程序和格式

## 空行和空格

* 1. 定义文件中，在每个类声明之后，每个函数定义结束之后都要加一个空行。
  2. 在一个函数体内，逻辑上密切相关的语句之间不加空行，其他地方应加空行分隔。
  3. 一行代码只做一件事情，每行一个语句。容易阅读且方便注释。
  4. if，for，while，do等语句自成一行，执行语句不得紧跟其后，且不管执行语句是否只有一句，都要加{}和缩进，使条理清晰。
  5. 尽量在定义变量的同时初始化，可以减少引用未初始化变量的隐患。
  6. ’(’向后紧跟，右括号逗号分号向前紧跟，紧跟处不留空格。
  7. 参数之间的逗号后面需要留出空格。
  8. 二元操作符前后都应当加空格。
  9. 一元操作符前后不加空格
  10. [], ., ->这类操作符前后不加空格。
  11. 对于比较长的if，for等语句，可以适当去掉空格。

void Func1(INT32 x, INT32 y, INT32 z); // 良好的风格

void Func1 (INT32 x, INT32 y, INT32 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

// ===========================================================

INT32\* x = &y; // 良好的风格

INT32\* 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();

## 对齐和缩进

* 1. {}应该独占一行并且位于同一列，且和引用它们的语句左对齐。
  2. {}之内的代码块，应当相对’{’使用4个空格的缩进。

void Function(INT32 x)

{

for (初始化; 终止条件; 更新)

{

// 相关代码之间紧密相连，不相关代码中间加上空行

}

try

{

// ...

}

catch (const exception& err)

{

// ...

}

// ...

}

* 1. 代码当中的缩进，统一使用Tab，并定义Tab为4个空格。

## 长行拆分

* 1. 代码行长度尽量控制为120字符以内，过长代码行可以进行适当的分行。
  2. 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，拆分的新行要进行适当的缩进，使排版整齐，语句可读。

if ((very\_longer\_variable1 >= very\_longer\_variable2)

&& (very\_longer\_variable3 <= very\_longer\_variable4)

&& (very\_longer\_variable5 <= very\_longer\_variable6))

{

dosomething();

}

## 注释

* 1. 提倡将代码写成自注释的，条理清楚则不必额外添加注释。
  2. 注释是对代码的额外提示和解释，注释应该达到代码的20%到40%。
  3. 如果不是通过伪码来编码，则提倡边写代码边注释。修改代码同时修改注释，保持两者一致性。多余的注释及时删除。
  4. 注释应当准确易懂，没有二义性。使用中文注释。
  5. 注释放在被注释代码的上方。（如果注释的字符串长度短小简单，也可以放在对应代码的右边，但要注意和上下的注释保持局部对齐）
  6. 代码段内注释的格式如下。

// 单行注释，用双斜线，放在被注释代码之上

被注释代码

//

// 多行注释

// 在被注释代码之上，前后都增加一个空的注释行。

//

被注释代码

## 类版式

* 1. 采用支持Doxygen自动生成说明文档的注释格式。
  2. 正常的函数注释风格如下

/\*\*

\* @brief

\* 函数功能介绍

\* @param 参数名1 : 参数1功能介绍

\* @param 参数名2 : 参数2功能介绍

\* 如需要断行，第二行从起始处空一格位置继续书写

\* @return 返回值介绍

\*/

函数声明

例如：

/\*\*

\* @brief

\* 获取当前文件的属性

\* @param pStatus : 获取的文件属性结构体

\* @return 如果成功，返回TRUE，否则返回FALSE

\*/

BOOL SDAPI GetFileStatus(SFileStatus &pStatus)

* 1. 正常的结构体注释的格式如下：

/\*\*

\*@brief 结构体功能介绍

\*/

struct \*\*\*

{

成员1定义; /\*\*<成员1功能介绍\*/

成员2定义; /\*\*<成员2功能介绍\*/

};

例如：

/\*\*

\*@brief 文件状态结构体

\*/

struct SFileStatus

{

UINT32 mode; /\*\*<文件对应的模式，文件，目录等\*/

UINT32 inode; /\*\*<i-node节点号, linux-specific\*/

};

* 1. 正常的类注释的格式如下：

/\*\*

\*@brief 类功能介绍

\*/

class \*\*\*\*

{

public：

公共接口；

protect:

保护类函数；

private：

私有成员；  
};

例如：

/\*\*

\*@brief 文件操作类，完成最基本的文件操作,或使用FileMapping方式读取文件(具体功能参照接口)

\*/

class CSDFile

{

public：

/\*\*

\* @brief

\* Open a file, type参数同fopen接口

\* @param pszFileName : 文件名称

\* @param type : 文件访问权限

\* @return 成功返回TRUE，失败返回FALSE

\*/

BOOL SDAPI Open(const CHAR\* pszFileName, const CHAR\* type, BOOL);

protect:

private：

BOOL m\_bFileMapping; /\*\*<是否使用FileMapping方式读取文件\*/

};

* 1. 正常的命名空间注释的格式如下：

namespace \*\*\*\*

{

/\*\*

\* @defgroup 子类别 子类别介绍

\* @ingroup 父类别

\* @{

\*/

……

}

例如：

namespace SGDP

{

/\*\*

\* @defgroup groupfile 文件操作

\* @ingroup SGDP

\* @{

\*/

……

}

* 1. 正常的文件注释的格式如下：

/\*\*

\* @file 文件名

\* @author 作者名

\* @brief 文件功能介绍

\*

\*\*/

例如：

/\*\*

\* @file sdfile.h

\* @author 王奎

\* @brief 文件工具类

\*

\*\*/

* 1. 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，体现以方法为中心的设计思想，而不是以数据为中心。

# 命名规则

* 1. 标识符应当直观而且可以拼读，可望文知意，不必进行解码。
  2. 标识符应当遵循”MinLength-MaxInfo”的原则。用最小长度来表达最多的信息。
  3. 命名规则尽量和采取的操作系统和开发工具的风格一致。
  4. 程序中不要出现仅仅靠大小写区别的标识符。如x和X。
  5. 程序中不要出现标识符完全相同的局部变量和全局变量。
  6. 变量名字应该是“名词”或者“形容词+名词”。如Value或者MaxValue。
  7. 全局函数的名字应该使用“动词+名词”，而类的函数则使用动词。如DrawBox（）和box->Draw（）。
  8. 尽量避免在标识符当中出现数字编号。
  9. 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。如果是全局变量，需要在类型名称后面加上模块名称，以示区别。

作用域前缀表明一个变量的可见范围。作用域有如下几种：

|  |  |
| --- | --- |
| **前缀** | **说明** |
| *无* | 局部变量 |
| m\_ | 类和结构的成员变量（member） |
| sm\_ | 类和结构的静态成员变量（static member） |
| s\_ | 静态变量（static） |
| g\_ | 外部全局变量（global） |
| sg\_ | 静态全局变量（static global） |
| gg\_ | 进程间共享的共享数据段全局变量（global global） |

类型前缀表明一个变量的数据类型，可以有如下几种：

|  |  |
| --- | --- |
| **前缀** | **说明** |
| ll  qw  n  dw  sh  w  c  by | 八字节有符号整数必须使用“sdtype.h”中定义的INT64类型  八字节无符号整数必须使用“sdtype.h”中定义的UINT64类型  四字节有符号整数必须使用“sdtype.h”中定义的INT32, LONG类型  四字节无符号整数必须使用“sdtype.h”中定义的UINT32, ULONG类型  两字节有符号整数必须使用“sdtype.h”中定义的INT16类型  两字节无符号整数必须使用“sdtype.h”中定义的UINT16类型  一字节有符号整数必须使用“sdtype.h”中定义的CHAR类型  一字节无符号整数必须使用“sdtype.h”中定义的UCHAR类型 |
| en | 枚举型变量（enumeration）举例如下  /\*\*  \*@brief 服务器状态  \*/  enum ESvrStat  {  SRV\_INVALID = 0,  SRV\_STARTING = 1,  SRV\_STARTED,  SRV\_PAUSING,  SRV\_PAUSED,  SRV\_STOPPING,  SRV\_STOPPED  }; |
| b | 布尔型变量必须使用“sdtype.h”中定义的BOOL类型，取值使用“sdtype.h”中定义的TRUE、FALSE |
| f | 浮点型变量必须使用“sdtype.h”中定义的FLOAT类型 |
| p  pst  po  psz  pfn  pc | 指针型变量和迭代子（pointer），后面再加上指针的类型  pointer of struct  pointer of object  pointer of string with zero end  pointer of function  pointer of char array |
| st  tag | 结构struct SDateTime stDateTime;  typedef struct tagAbc { }SAbc; |
| a  adw | 数组（array）  UINT32 array |
| sz  ac  str | string with zero end一般用来存放0结尾的字符串  CHAR array字符数组，字符缓冲区，不一定是字符串  String，CString等 |
| C  S  I | class类名称如CTimerModule  struct结构名称如SDateTime  接口定义，如ISDNet |
| o  itr  map  lst  vec  str | 类的实例（object）  iterator迭代器  std::map  std::list  std::vector  std::string  **关于模版的typedef, 应按如下方式定义： typedef std::map<int, int> CIntPairMap**  **以C开头，模版容器类型结尾, 其他类型容器的typedef也应遵守此规则。** |

模块名称是为了避免在大型的开发项目当中不同模块之间出现命名冲突。通过增加模块标识，可以更醒目。比如g\_ulTimer\_MaxElapse.

* 1. 提倡使用名字空间，并且名字空间的名称不应该过长，通常都使用缩写的形式来命名。

# 表达式和基本语句

* 1. 避免使用运算符的默认优先级，用括号来明确指出运算顺序。

word = (high << 8) | low

if ((a | b) && (a & c))

* 1. 不要编写太复杂的复合表达式。
  2. 不要写有多用途的表达式。

d = ( a = b + c) + e

应该写成

a = b + c

d = a + e

* 1. 不要直接将Bool型变量跟TRUE或者1，0等值直接进行比较。
  2. 应该将整型变量用==或者!=直接与0比较，而不要写成if(nValue)或者if(!nValue).
  3. 不可将浮点变量用等于或者不等于跟任何数字比较，而应该设法转化成大于小于的形式。
  4. 指针变量尽量跟NULL而不是0来比较。
  5. switch的每个case语句结尾都需要添加break，除非是逻辑的确需要多个分支重叠，此时应加以注释。不管是否需要default分支，都应该加上，并且需要break。

# 常量

宏常量和const常量的区别：const常量有数据类型，而宏常量没有。编译器可以对const常量进行类型安全检查，对宏常量只进行字符替换，没有类型检查，并且字符替换可能会有意想不到的错误。

* 1. 尽量使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串
  2. 在C++程序中只使用const常量而不使用宏常量。
  3. 需要对外公开的常量放在头文件中，不需要公开的常量放在定义文件中。为便于管理，可以将不同模块的常量几种存放在一个公共的头文件中。
  4. 如果某一常量与其他常量密切相关，应在定义中包含这种关系，而不是直接给出值。
  5. 常量的命名由类型前缀+模块+全大写字母组成，单词间通过下划线来界定，如：TIMER\_NAME\_STRING, TIMER\_MAX\_LENGTH