Google C++ 编程规范

一、头文件

通常，每个.cc 文件都有一个对应的.h 文件。

1、 #define 的保护

（1）所有头文件都应该使用#define 防止头文件被多重包含，命名格式当时:<PROJECT>_<PATH>_<FILE>_H_。

例如：

#ifndef FOO_BAR_BAZ_H_

#define FOO_BAR_BAZ_H_

…

#endif FOO_BAR_BAZ_H_

2、 头文件依赖

（1）使用前置声明，尽量减少.h 文件中#include 的数量

举例说明：头文件中用到了类 File，但不需要访问 File 的声明，则头文件中只需声明 class File；无需#include “file/base/file.h”

在头文件如何做到使用类 Foo 而无需访问类的定义？

1.将数据成员类型声明为 Foo*或者 Foo&

2.参数、返回值类型为 Foo 的函数只是声明（但不定义实现）

3.静态数据成员的类型可以被声明为 Foo，因为静态数据成员的定义在类定义之外。

另一方面，如果你的类是 Foo 的之类，或者含有类型为 Foo 的非静态数据成员，则必须为之包含头文件。

3、 内联函数

（1）只有当函数只有 10 行甚至更少时才会将其定义为内联函数。对于析构函数应该慎重对待，析构函数往往比其表面看起来要长，因为有一些隐式成员和基类析构函数（如果有的话）被调用。

（2）一般不要内联包含循环和 switch 语句的函数。

（3）虚函数和递归函数即使被声明为内联的也不一定就是内联函数

（4）通常，递归函数不应该被声明为内联的。

4、 -inl.h 文件

（1）复杂的内联函数的定义，应放在后缀名为-inl.h 的头文件中。

5、 函数参数顺序

（1）定义函数的时候，参数顺序是：输入参数在前，输出参数在后。

6、 包含文件的名称和次序

（1）将包含次序标准化可增强可读性，避免隐藏依赖，次序如下：C 库、C++库、其他库的.h、项目内的.h

（2）项目内头文件应该按照项目源代码目录树结构排列，并且避免使用 UNIX 文件路径。例如：google_prj/src/base/logging.h 应该像这样被包含：#include “base/logging.h”

（3）如果 dir/foo.cc 的主要作用是执行或者测试 dir2/foo2.h 的功能，foo.cc 中包含文件的次序如下：

dir2/foo2.h //优先排列

C 系统文件

C++系统文件

其他库文件

本项目内头文件

（4）相同目录下的头文件，按照字母序进行排列是比较好的选择。

举例来说，google-prog/src/foo/internal/fooserver.cc 的包含次序如下：

#include “foo/public/fooserver.h” //优先位置

#include <sys/types.h>

```
#include <unistd.h>
#include <hash_map>
#include <vector>

#include “base/basictypes.h”
#include “base/commandlineflags.h”   //按照字母序进行排列
#include “foo/public/bar.h”
```

二、作用域

1、 命名空间

（1）在.cc 文件中，提倡使用不具名的命名空间，使用具名命名空间时，其名称可基于项目或者路径名称，不要使用 using 指示符。

（2）不具名命名空间：

```
namespace   //.cc 文件中
{
//命名空间的内容无需缩进
enum { UNUSED, EOF, ERROR };   //经常使用的符号
bool AtEof() { return pos_ == EOF; }   //使用本命名空间内的符号 EOF
}  //namespace
```

然而，与特定类相关联的文件作用域声明在该类中被声明为类型、静态数据成员或者静态成员函数，而不是不具名命名空间的成员。

（3）具名命名空间

命名空间将除文件包含、全局标识的声明/定义以及类的前置声明外的整个源文件封装起来，同其他命名空间相区分。

所有的声明都置于命名空间中，注意不要使用缩进。

不要声明命名空间 std 下面的任何内容，包括标准库类的前置声明。声明 std 下的实体会导致不明确的行为。

最好不要使用 using 指示符，以保证命名空间下的所有名称都可以正常使用。

2、 嵌套类

（1）不要将嵌套类定义为 public，除非它们是接口的一部分。

（2）当公开嵌套类作为接口的一部分时，虽然可以直接将他们保持在全局作用域中，但将嵌套类的声明至于命名空间中是更好的选择

3、 非成员函数、静态成员函数和全局函数

（1）使用命名空间中的非成员函数和静态函数，尽量不要使用全局函数。

4、 局部变量

（1）将函数变量尽可能置于最小作用域内，在声明变量时将其初始化。变量的声明离第一次使用越近越好，特别的，应使用初始化代替声明+赋值的方式。

5、 全局变量

（1）class 类型的全局变量是被禁止的，内建类型的全局变量是允许的，当然多线程代码中非常数全局变量也是被禁止的。

（2）永远不要使用函数返回值初始化全局变量

（3）如果一定要使用 class 类型的全局变量，请使用单件模式（singleton pattern）

（4）对于全局的字符串常量，使用 C 风格的字符串，不要使用 STL 的字符串。 Const char kFrogSays[] = “ribbet”;

三、类

1、 构造函数的职责

（1）构造函数中只进行那些没有实际意义的初始化，可能的话，使用 Init()方法集中初始化为有意义的数据。

2、 默认构造函数

（1）如果一个类定义了若干成员变量但是没有其他构造函数，需要定义一个默认的构造函数，否则编译器将自动生产默认构造

函数。

3、 明确的构造函数

（1）对单参数构造函数使用 C++关键字 explicit[1]，避免构造函数被调用造成隐式转换。

（2）隐式转换：从构造函数形参类型到类类型的转换。

4、 拷贝构造函数

（1）仅在代码中需要拷贝一个类对象的时候使用拷贝构造函数；不需要拷贝时应使用 DISALLOW_COPY_AND_ASSIGN。大量的类并不需要可拷贝，也不需要一个拷贝构造函数或赋值操作。不幸的是，如果你不主动声明它们，编译器会为你自动生成，而且是 public 的。

（2）可以考虑在类的 private 中添加空的拷贝构造函数和赋值操作，只有声明，没有定义。由于这些空程序声明为 private,当其他代码试图使用他们的时候，编译器将报错。为了方便，可以使用宏 DISALLOW_COPY_AND_ASSIGN:

```
//禁止使用拷贝构造函数和赋值操作的宏
//应在类的 private:中使用
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
    TypeName(const TypeName&);                    \
    void operator=(const TypeName&)

    class Foo
    {
    Public:
        Foo(int f);
        ~Foo();
    Private:
        DISALLOW_COPY_AND_ASSIGN(Foo)
    };
```

如上所述，绝大多数情况都应使用 DISALLOW_COPY_AND_ASSIGN，如果类确实需要可拷贝，应该在类的头文件中说明原由，并适当定义拷贝构造函数和赋值操作，注意在 operator=中检测自赋值情况。

5、 结构体和类

（1）仅当只有数据的时候使用 struct。其他一概使用 class。

6、 继承

（1）所有继承必须是 public 的，如果想私有继承的话，应该采取包含基类实例作为成员的方式作为替代。

（2）不要过多使用实现继承，组合通常更合适一些。努力做到只在“是一个”的情况下使用继承：如果 Bar 的确“是一种”Foo,才令 Bar 是 Foo 的子类。

（3）如果类具有虚函数，令析构函数为 virtual.

（4）限定仅在子类访问的成员函数为 protected

（5）数据成员应始终为私有。

（6）当重定义派生的虚函数时，再派生类中明确声明其为 virtual.

7、 多重继承

（1）只有当所有超类除第一个外都是纯接口时才能使用多重继承。为确保它们是纯接口，这些类必须以 Interface 为后缀。

8、 接口

（1）接口是指满足特定条件的类，这些类以 Interface 为后缀

（2）定义：当一个类满足以下要求时，称之为纯接口：

(1)只有纯虚函数（“=0”）和静态函数（下文提到的析构函数除外）

(2)没有非静态数据成员

(3)没有定义任何构造函数。如果有，也不含参数，并且为 protected

(4)如果是子类，也只能继承满足上述条件并以 Interface 为后缀的类

（3）接口类不能被直接实例化，因为它声明了纯虚函数。为确保接口类的所有实现可被正确销毁，必须为之声明虚析构函数。

9、 操作符重载

（1）一般不要重载操作符，尤其是赋值操作（operator=）比较阴险，应避免重载。如果需要的话，可以定义类似 Equals()，CopyFrom()等函数。

（2）然而，极少数情况下需要重载操作符以便与模板或“标准”C++类衔接，如果被证明是正当的尚可接受，但要尽可能避免这样做。尤其是不要仅仅为了在 STL 容器中作为 key 使用就重载 operator==或 operator<，取而代之，你应该在声明容器的时候，创建相等判断和大小比较的仿函数类型。

（3）有些 STL 算法确实需要重载 operator==时可以这么做，不要忘了提供文档说明原因。

10、 存取控制

（1）将数据成员私有化，并提供相关存取函数。例如：定义变量 foo_及取值函数 get_foo()、赋值函数 set_foo()。

（2）存取函数的定义一般内联在头文件中。

11、 声明次序

（1）在类中使用特定的声明次序：public 在 private 之前，成员函数在数据成员（变量）前。

（2）定义次序如下：public、protected、private，如果哪一块没有，可以直接忽略。

（3）每一块中，声明次序一般如下：

(1)typedefs 和 enums

(2)常量

(3)构造函数

(4)析构函数

(5)成员函数，含静态成员函数

(6)数据成员，含静态数据成员

（4）.CC 文件中函数的定义应尽可能和声明次序一致。

（5）不要将大型函数内联到类的定义中，通常，只有那些没有特别意义的或者性能要求高的，并且是比较短小的函数才被定义为内联函数。

12、 编写短小函数

（1）倾向选择短小的、凝练的函数。

（2）尽量控制函数长度在 40 行，超过 40 行，可以考虑分割一下。

四、 Google 特有的风格

1、 智能指针

（1）尽量避免使用智能指针。如果需要使用，scoped_ptr 完全可以胜任。在非常特殊的情况下，例如对 STL 容器中的对象，应该只使用 std::tr1::shared_ptr，任何情况下不要使用 auto_ptr.

（2）一般来说，倾向于设计对象隶属明确的代码，最明确的对象隶属是根本不需要指针，直接将一个对象作为一个域或局部变量使用。

五、 其他 C++特性

1、 引用参数

（1）函数形参表中，所有的引用必须是 const 的。

（2）这是一个硬性规定：输入参数为值或者常数引用，输出参数为指针；输入参数可以是常数指针，但不能使用非常数应用形参。

2、 函数重载

（1）仅在输入参数不同、功能相同时使用重载函数（含构造函数），不要使用函数重载模仿缺省函数参数。

（2）如果想重载一个函数，考虑让函数名包含参数信息，例如，使用 AppendString(),AppendInt()而不是 Append();

3、 缺省参数

（1）禁止使用缺省函数参数。

（2）所有参数必须明确指定，强制程序员考虑 API 和传入的各参数值，避免使用可能不为程序员所知的缺省参数。

4、 变长数组和 alloca

（1）禁止使用变长数组和 alloca()

（2）使用安全的分配器，如 scoped_ptr/scoped_array

5、 友元

（1）通常将友元定义在同一文件下，避免读者跑到其他文件中查找其对某个类私有成员的使用。

（2）经常用到友元的一个地方是将 FooBuilder 声明为 Foo 的友元，FooBuilder 以便可以正确构造 Foo 的内部状态，而无需将该状态暴露出来。某些情况下，将一个单元测试用类声明为待测类的友元会很方便。

6、 异常

（1）不要使用 C++异常。

7、 运行时类型识别

（1）禁止使用 RTTI

8、 类型转换

（1）使用 C++风格而不要使用 C 风格的类型转换

(1)static_cast:和 C 风格转换相似可做值的强制转换，或指针的父类到子类的明确的向上转换

(2)const_cast:移除 const 属性

(3)reinterpret_cast:指针类型和整型或者其他指针之间不安全的相互转换，仅在你对所做的一切了然于心时使用

(4)dynamic_cast:除了测试外不要使用，除了单元测试之外，如果你需要在运行时确定类型信息，说明设计有缺陷

9、 流

（1）不要使用流，除非是日子接口需要，使用 printf 之类的代替

（2）使用流还有很多利弊，代码一致性胜过一切，不要再代码中使用流

10、 前置自增和自减

（1）对简单数值（非对象）来说，两种都无所谓，对迭代器和模板类型来说，要使用前置自增（自减）。

11、 const 的使用

（1）const 变量、数据成员、函数和参数为编译时类型检测增加了一层保障，更好的尽早发现错误。因此，在任何可以使用的情况下使用 const。

12、 整型

（1）<stdint.h>定义了 int_16_t、uint32_t、int64_t 等整型，在需要确定大小的整型时可以使用它们替代 short, unsigned long long 等，在 C 整型中，只使用 int，适当情况下，推荐使用的标准类型如 size_t 和 ptrdiff_t。

（2）最常使用的是，对整数来说，通常不会用到太大，如循环计数等。可以使用普通的 int,你可以认为 int 至少为 32 位，但不要认为它会多于 32 位，需要 64 位整型的话，可以使用 int64_t 或 uint64_t。

（3）对于大整数，使用 int64_t

（4）不要使用 uint32_t 等无符号整数，除非你是在表示一个位组而不是一个数值，即使数值不会为负值也不要使用无符号类型使用断言来保护数据。无符号后续会带来很多隐形 bug，因此，使用断言声明变量为非负数，不要使用无符号数。

13、 64 位下的可移植性

（1）printf()指定的一些类型在 32 位和 64 位系统上可移植性不是很好，C99 标准定义了 一些可移植的格式。不幸的是，MSVC 7.1 并非全部支持，而且标准中也有所遗漏。所以 有时我们就不得不自己定义丑陋的版本（使用标准风格要包含文件 inttypes.h）：

```
// printf macros for size_t, in the style of inttypes.h
#ifdef _LP64
#define __PRIS_PREFIX "z"
#else
```

```
#define __PRIS_PREFIX
#endif

// Use these macros after a % in a printf format string
// to get correct 32/64 bit behavior, like this:
// size_t size = records.size();
// printf("%"PRIuS"\n", size);

#define PRIdS __PRIS_PREFIX "d"
#define PRIxS __PRIS_PREFIX "x"
#define PRIuS __PRIS_PREFIX "u"
#define PRIXS __PRIS_PREFIX "X"
#define PRIoS __PRIS_PREFIX "o"
```

类型	不要使用	使用	备注
Void*(或者其他指针类型)	%lx	%p	
Int64_t	%qd, %lld	%”PRId64”	
Uint64_t	%qu, %llu, %llx	%”PRIu64”, %”PRIx64”	
Size_t	%u	%”PRIuS”, %”PRIxS”	C99 指定%zu
Ptrdiff_t	%d	%”PRIdS”	C99 指定%zu

注意宏 PRI*会被编译器扩展为独立字符串，因此如果使用非常量的格式化字符串，需要将宏的值而不是宏名插入格式中，在使用宏 PRI*时同样可以在%后指定长度等信息。例如，

printf("x = %30"PRIuS"\n", x)在 32 位 Linux 上将被扩展为 printf("x = %30" "u" "\n", x)，编译器会处理为 printf("x = %30u\n", x)。

（2）记住 sizeof(void *) != sizeof(int)，如果需要一个指针大小的整数要使用 intptr_t

（3）需要对结构对齐加以留心，尤其是对于存储在磁盘上的结构体。大多数编译器提供了调整结构体对齐的方案，GCC 中可使用__attribute__((packed))，MSVC 提供了#pragma pack()和__declspec(align())

（4）创建 64 位常量时使用 LL 或 ULL 作为后缀，如：

```
Int64_t my_value = 0x123456789LL;
Uint64_t my_mask = 3ULL << 48;
```

（5）如果确实需要 32 位和 64 位系统具有不同代码，可以在代码变量前使用。（尽量不要这么做，使用时尽量使修改局部化）

14、 预处理宏

（1）使用宏要谨慎，尽量以内联函数、枚举和常量代替。

下面给出的用法模式可以避免一些使用宏的问题，供使用宏时参考：

（1）不要再.h 文件中使用宏

（2）使用前正确#define，使用后正确#undef

（3）不要只是对已经存在的宏使用#undef，选择一个不会冲突的名称

（4）不使用会导致不稳定的 C++构造的宏，至少文档说明其行为。

15、 0 和 NULL

（1）整数用 0，实数用 0.0，指针用 NULL，字符（串）用 “\0”。

16、 sizeof

（1）尽量用 sizeof(varname) 代替 sizeof(type)

17、 Boost 库

（1）只使用 Boost 中被认可的库。为了向阅读和维护代码的人员提供更好的可读性，我们只允许使用 Boost 特性的一个成熟子

集，当前，这些库包括：

(1) Compressed Par: boost/compressed_pair.hpp

(2) Pointer Container: boost/ptr_container 不包括 ptr_array.hpp 和序列化

还有一些别的。

六、命名约定

1、 通用命名规则

（1）函数命名、变量命名、文件命名应具有描述性，不要过度缩写，类型和变量应该是名词，函数名可以用“命令性”动词。

2、 文件命名

（1）文件名要全部小写，可以包含下划线(_)或短线(-)，按照项目的约定来。

（2）C++文件以.cc 结尾，头文件以.h 结尾。

（3）不要使用已经存在于/usr/include 下的文件名（对 linux/unix 等系统而言）

（4）通常，尽量让文件名更加明确，http_server_logs.h 就比 logs.h 要好，定义类时文件名一般成对出现，如 foo_bar.h 和 foo_bar.cc，对应类 FooBar。

（5）内联函数必须放在.h 文件中，如果内联函数比较短，就直接放在.h 中，如果代码比较长，可以放到以-inl.h 结尾的文件中。

（6）对于包含大量内联代码的类，可以有三个文件：

url_table.h //类的声明

url_table.cc //类的定义

url_table-ini.h //

3、 类型命名

（1）类型命名每个单词以大写字母开头，不包含下划线: MyExcitingClass、MyExcitingEnum。

（2）所有类型命名——类、结构体、类型定义(typedef)、枚举——使用相同约定，例如：

```
//class and structs
class UrlTable {…}
class UrlTableTester {…}
struct UrlTableProperties {…}

//typedefs
Typedef hash_map<UrlTableProperties *, string> PropertiesMap;

//enums
enum UrlTableErrors {…}
```

4、 变量命名

（1）变量名一律小写，单词间以下划线相连，类的成员变量以下划线结尾，如 my_exciting_local_veriable 、my_exciting_member_veriable_。

（2）普通变量命名：string table_name;

（3）结构体的数据成员可以和普通变量一样，不用像类那样接下划线：

```
struct UrlTableProperties
{
string name;
int num_entries;
}
```

5、 常量命名

（1）在名称前加 k：kDaysInAWeek

（2）所有编译时常量（无论是局部的、全局的还是类中的）和其他变量保持些许区别，k 后加大写字母开头的单词：

```
const int kDaysInAWeek = 7;
```

6、 函数命名

（1）普通函数:函数名以大写字母开头，每个单词首字母大写，没有下划线：AddTableEntry(), DeleteUrl()

（2）存取函数要与存取的变量名匹配，举例如下：

```
class MyClass
{
public:
…
int num_entries() const { return num_entries_ ; }
void set_num_entries(int num_entries) { num_entries_ = num_entries; }

private:
int num_entries_;
};
```

（3）其他短小的内联函数也可以使用小写字母。小写的函数名意味着可以直接内联使用。

7、 命名空间

（1）命名空间的名称是全小写的，其命名基于名称和目录结构 ： google_awesome_project

8、 枚举命名

（1）枚举值应全部大写，单词之间以下划线连接，MY_EXCITING_ENUM_VALUE

（2）枚举名称属于类型，因此大小写混合:UrlTableErrors.

```
Enum UrlTableErrors
{
OK = 0,
ERROR_OUT_OF_MEMORY,
ERROR_MALFORMED_INPUT
};
```

9、 宏命名

（1）宏命名要像枚举命名一样全部大写，使用下划线。

```
#define MY_EXCITING_ENUM_VALUE 3.0
```

10、 命名规则例外

（1）当命名与现有 C/C++实体相似的对象时，可参考现有命名约定：

bigopen() : 函数名，参考 open()

uint : typedef 类型定义

bigpos : struct 或 class，参考 pos

sparse_hash_map : STL 相似实体，参考 STL 命名约定

LONGLONG_MAX : 常量，类似 INT_MAX

七、注释

1、 注释风格

（1）使用//或者/**/都行，统一就行。

2、 文件注释

（1）在每一个文件开头加入版权公告，然后是文件内容描述。

法律公告和作者信息：每一项包含以下项，依次是：

(1)版权：如 Copyright 2008 Google Inc.;

(2)许可版本：为项目选择合适的许可证版本，如 Apache 2.0、BSD、LGPL、GPL;

(3)作者：标识文件的原始作者

如果对其他人创建的文件做出了重大修改，需要将自己的信息增加到作者信息中，这样当其他人对文件有疑问时可以知道应该联系谁。

示例：

//Copyright 2008 Google Inc.

//Licence(BSD/GPL/…)

//Author: voidccc

//This is …

3、 类注释

（1）每个类的定义要附着描述类的功能和用法的注释。

（2）如果类有任何同步前提，文档说明之。如果该类的实例可被多线程访问，使用时务必注意文档说明。

4、 函数注释

（1）函数声明处注释描述函数功能，定义处描述函数实现。

函数注释例子如下(主要有 4 项：函数功能描述、输入的参数、带回的参数、返回值)：

/**

* @DESCRIPTION： 获取合适的目录，存放要进行恢复的文件;一般为 cellibrary/dbRecover，如果这个目录存在，

就按照 cellibrary/dbRecover(1)这样的规律继续往上找，直到找到未被用的名称为止

* @PARAM [IN] : find_file_path ,在此路径下进行搜索文件，并判断是否进行拷贝和重命名

* @PARAM [OUT] : path_copy_to，搜索到的合格的文件拷贝到此路径下，并重命名

* @RETURN： 操作成功，返回 true;否则，返回 false.

*/

5、 变量注释

（1）正常的变量名足以说明变量用途，一般不需要进行注释。

（2）全局变量或者类的数据成员应注释说明用途。

6、 实现注释

（1）对于实现代码中巧妙的、晦涩的、有趣的、重要的地方要加注释。

（2）注意不要用自然语言翻译代码作为注释，要假设读你代码的人的 C++比你强。

7、 标点、拼写和语法

（1）留意标点、拼写和语法等的书写。

8、 TODO 注释

（1）对于那些临时的、短期的解决方案，或已经够好但并不完美的代码使用 TODO 注释。

例子如下：

//TODO（wsrelea@aliyun.com）: Use a “*” here for concatenation operation

八、格式

1、 行长度

（1）每一行的代码字符数不超过 80。

2、 非 ASCII 字符

（1）尽量不使用非 ASCII 字符，使用时必须使用 UTF-8 格式。

3、 空格还是制表位

（1）只使用空格，每次缩进 4 个空格。

4、 函数声明和定义

（1）返回类型和函数名要在同一行，合适的话，参数也放在同一行。

看上去像这样：

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2)
{
    DoSomething();
    …
}
```

（2）如果同一行文本较多，容不下所有的参数：

```
ReturnType ClassName::ReallyLongFunctionName(
    Type par_name1,
    Type par_name2,
    Type par_name3) const
{
    DoSomething();
    …
}
```

（3）注意以下几点：

(1)返回这总是和函数名在同一行

(2)左圆括号总是和函数名在同一行

(3)函数名和左圆括号之间没有空格

(4)圆括号和参数之间没有空格

(5)所有形参尽可能对齐，换行的形参缩进 4 个空格

(6)如果函数是 const 的，关键字 const 应与最后一个参数位于同一行

5、 函数调用

（1）尽量放在同一行，否则，将实参封装在圆括号中。

如下形式：

```
bool retval = DoSomething(argument1, argument2, argument3);
```

（2）参数比较多，可以一行放一个参数：

```
bool retval = DoSomething(
    argument1,
    argument2,
    argument3);
```

6、 条件语句

（1）按照下面的格式：

```
if (condition)
{
    …
}
```

（2）条件较多的按照下面的格式：

```
if (condition1)
{
}
else if (condition2)
{
}
else
```

```
{
}
```

7、 循环和开关选择语句

（1）switch 语句按照下面的写法：

```
switch (var)
{
case 0:
{
}
break;
case 1:
{
}
break;
…
default:
break;
}
```

（2）空循环体按照下面的格式：

```
while (condition)
{
    //do nothing
}

for (int i = 0; i < kSomeNum; i++)
{
    //do nothing
}
```

8、 指针和引用表达式

（1）句点（.）和箭头（->）前后不要有空格，指针/地址操作符（*, &）后面不要有空格。

实例如下：

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

（2）在声明指针变量或者参数时，星号与类型紧挨，表示这个变量类型是指针。

```
char* x;
```

9、 布尔表达式

（1）如果一个布尔表达式超过标准行宽，断行的时候需要统一下，尽量加上()增强可读性。

实例如下：

```
if ((one < two)
   && (three < four)   //行首后退四格
   && (five < six))
{
```

```
        …
    }
```

10、 函数返回值

（1）return 表达式中不要使用圆括号。

用 return x; 不要用 return (x);

11、 变量及数组初始化

（1）初始化使用=

```
string name = “some name”;
```

12、 预处理指令

（1）预处理指令要定着行首写，不要缩进。

```
if (condition)
{
#if DISASTER_PENDING
        DropEverything();
        …
#endif
        …
    }
```

13、 类格式

（1）声明属性依次是 public:、protected:、private:，每行不缩进。

```
calss MyClass : public OtherClass
{
public:
    MyClass();
    ~MyClass();

    Void SomeFunction();
    Void someFunctionThatDoesNothing()
    {
        …
    }

protected:
    void SomeInterFunction();

private:
    int some_var_;
    int some_other_var_;
}
```

14、 初始化列表

（1）初始化列表可以放在同一行或者缩进四格并排几行。

（2）如果一行可放下：

```
MyClass::MyClass(int var) : some_var_(var), some_other_var_(var + 1)
{
```

```
    …
}
```

（3）如果放不下：

```
MyClass::MyClass(int var)
    : some_var_(var),
      Some_other_var_(var + 1)
{
    …
}
```

15、 命名空间格式化

（1）命名空间的内容不缩进。

```
namespace
{
void foo()  //不缩进
{
    …
}
}
```

16、 水平留白

（1）水平留白的使用要因地制宜，不要在行尾添加无谓的留白。

（2）一般是空两个格。或者多行统一空格。

```
int I = 0;  //…
```

（3）或者多行统一空格。

```
int I = 1;              //…
string s = “hello”;  //…
char a = ‘m’;        //…
double b = 2.3;      //…
```

17、 垂直留白

（1）垂直留白越少越好。

（2）不是非常有必要的话，不要使用空行。

九、规则之外

1、 现有不统一代码

（1）当修改使用其他风格的代码时，为了与代码原有风格保持一致可以不使用本规范。

2、 Windows 代码

（1）Windows 程序员有自己的编码习惯，主要源于 Windows 的一些头文件和其他 Microsoft 代码。

（2）下面是一些 Windows 指南：

(1)不要使用匈牙利命名法（Hungarian notation，如定义整型变量为 iNum），使用 Google 命名约定，包括对源文件使用.cc 扩展名；

(2) Windows 定义了很多原有内建类型的同义词（译者注，这一点，我也很反感），如 DWORD、HANDLE 等等，在调用 Windows API 时这是完全可以接受甚至鼓励的，但还是尽量使用原来的 C++类型，例如，使用 const TCHAR *而不是 LPCTSTR；

(3)使用 Microsoft Visual C++进行编译时，将警告级别设置为 3 或更高，并将所有 warnings 当作 errors 处理；

(4)不要使用#pragma once;作为包含保护，使用 C++标准包含保护，包含保护的文件路径包含到项目树顶层（译者注，#include<prj_name/public/tools.h>）；

(5)除非万不得已，否则不使用任何不标准的扩展，如#pragma 和__declspec，允许使用 __declspec(dllimport)和

__declspec(dllexport)，但必须通过 DLLIMPORT 和 DLLEXPORT 等宏，以便其他人在共享使用这些代码时容易放弃这些扩展。

（3）在 Windows 上，只有很少一些偶尔可以不遵守的规则：

(1)通常我们禁止使用多重继承，但在使用 COM 和 ATL/WTL 类时可以使用多重继承，为了执行 COM 或 ATL/WTL 类及其接口时可以使用多重实现继承；

(2)虽然代码中不应使用异常，但在 ATL 和部分 STL（包括 Visual C++的 STL）中异常被广泛使用，使用 ATL 时，应定义 _ATL_NO_EXCEPTIONS 以屏蔽异常，你要研究一下是否也屏蔽掉 STL 的异常，如果不屏蔽，开启编译器异常也可以，注意这只是为了编译 STL，自己仍然不要写含异常处理的代码；

(3)通常每个项目的每个源文件中都包含一个名为 StdAfx.h 或 precompile.h 的头文件方便头文件预编译，为了使代码方便与其他项目共享，避免显式包含此文件（precompile.cc 除 外），使用编译器选项/FI 以自动包含；

(4)通常名为 resource.h、且只包含宏的资源头文件，不必拘泥于此风格指南。

十、团队合作

（1）参考常识，尽量与团队保持一致。

[1]C++ Primer P265