C++ 编程规范

作者： PENDLE
时间： 2020/07/16

说明：此文档是综合《Google Cpp Style Guid》和
《Effective C++》而来的编程规范，部分规范相关的地方
也参考了《华为技术有限公司 C 语言编程规范》

修改记录：

① 2020/10/26 ：调整部分说明位置，合并部分内容

② 2020/11/14 ：整合原来几个拷贝函数的相关内容到 3.2 中

目录

1、头文件

通常，每个.cc 文件都有一个对应的.h 文件。

1.1 #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_
```

1.2 内联函数（E-5.30）

（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）包含循环和 switch 语句的函数不应该为内联函数，递归函数不应该为内联函数（inline 只是一种给编译器的建议，太复杂的语法逻辑，编译器会将其作为一个普通函数处理），虚函数不应该为内联（虚函数是在运行时展开的，但是 inline 是在编译时给编译器提供建议）。

（3）不要只因为 function templates 出现在头文件，就将它们声明为 inline。

1.3 -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。

1.4 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。

1.5 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”

C++标准库中除了定义 C++语言特有的功能之外，也兼容了 C 语言的标准库。C 语言的头文件形如 name.h, C++则将这些文件命名为 cname。也就是去掉了.h 后缀，而在文件名前面加上了字母 c，这里 c 表示这是一个属于 C 语言标准库的头文件。

2、作用域

2.1 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

（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。

在文件中进行 static 的做法已经被 C++标准取消了，现在的做法是使用未命名的命名空间。

不能在.h 文件中使用不具名命名空间。

（3）具名命名空间

命名空间将“除了文件包含、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。

2.2 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，但将嵌套类的声明置于命名空间中是更好的选择

2.3 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。

2.4 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。

2.5 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，请使用单例模式（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”;

3、类

3.1 构造和析构函数

（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）明确为类定义一个构造函数和析构函数，不论是否有成员存在。

（3）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。

（4）绝不在构造函数和析构函数中使用 virtual 函数，使用了也无法生效。（E-2.9）

3.2 拷贝函数

（1）若需要拷贝函数，则标准写法如下（E-2.12）

示例如下（参考 https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom）：

```
Class anotest { … };
class test { … };
class test1 : public test
{
public:
    …

    //关键的 swap 函数
    friend void swap(test1& first, test1& second)
    {
        using std::swap;

        swap(first.one, second.one);
        swap(first.two, second.two);
        swap(first.three, second.three);
    }

    //拷贝构造函数
    test1(const test1& rhs)
        : test(rhs)    //处理基类的成员拷贝
        , one(rhs.one)
        , two(rhs.two)
        , three(rhs.three)
    {
        //do nothing
    }

    拷贝赋值运算符
    test1& operator=(const test1& rhs)
    {
        test1 temp(rhs);
        test::operator=(rhs);    //处理基类的成员拷贝
        swap(*this, temp);
```

```
        return *this;
    }

private:
    int one;
    double* two;
    anotest* three;
};
```

（2）若不需要拷贝函数，则在类的 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&);                 \
TypeName& operator=(const TypeName&)
```

标准写法如下：

```
class Foo
{
public:
    Foo(int f);
    ~Foo();
private:
    DISALLOW_COPY_AND_ASSIGN(Foo);
};
```

或者，在 C++11 及以上版本中，使用下面写法更好：

```
#define DISUSE_COPY_AND_ASSIGN(TypeName) \
TypeName(const TypeName&) = delete;      \
TypeName& operator=(const TypeName&) = delete

class Foo
{
public:
    Foo(int f);
    ~Foo();
    DISUSE_COPY_AND_ASSIGN(Foo);
};
```

3.3 初始化（E-1.4）

（1）手工初始化内置类型对象，内置类型以外的，都由构造函数进行初始化。

（2）所有成员统一在构造函数中使用初始化列表进行初始化。初始化列表的顺序，和 class 中的声明次序相同。

（3）为免除“跨编译单元之初始化次序”问题，以 local-static（定义在函数内的 static）对象替换 non-local-static（定义在函数外的 static）对象。

3.4 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。

3.5 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。

3.6 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：

①typedefs 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。

3.7 类的设计（E-4.23）

Class 的设计就是 type 的设计。在定义一个新的 type 之前，请确定已经考虑过下面这些问题：

(1)新 type 的对象应该如何被创建和销毁

(2)对象的初始化和对象的赋值应该有什么样的差别

(3)新 type 的对象如果被值传递，意味着什么

(4)什么是新 type 的“合法值”

(5)你的新 type 需要配合某个继承图系吗

(6)你的新 type 需要什么样的转换

(7)什么样的操作符和函数对此新 type 而言是合理的

(8)什么样的标准函数应该驳回

(9)谁该取用新 type 的成员

(10)什么是新 type 的“未声明接口”

(11)你的新 type 有多么一般化

(12)你真的需要一个新 type 吗

3.8 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。

3.9 成员和非成员函数（E-4.23）（E-4.24）

（1）考虑使用 non-member non-friend 函数替换 member 函数。因为 non-member non-friend 函数不能访问私有成员，这样做可以增加封装性、包裹弹性和机能扩充性。同时使用命名空间进行辅助扩充。（参考 STL）

（2）如果你需要为某个函数的所有参数（包括被 this 指针所指的那个隐喻参数）进行类型转换（指的是包含隐式的类型转换的情况，一般是不允许类包含隐式的类型转换，但是有一些例外，此条就是针对这些例外设计的），那么这个函数必须是个 non-

member。（https://www.cnblogs.com/lasnitch/p/12764171.html）

3.10 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();

4、继承和面向对象设计

4.1 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 private。

（6）当重定义派生的虚函数时，在派生类中明确声明其为 virtual.

4.2 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。

4.3 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。

4.4 公有继承（E-6.32）

（1）确定你的 public 继承塑模出 is-a 关系。“public 继承”意味着 is-a。适用于 base classes 身上的每一件事情一定适用于 derived classes 身上，因为每一个 derived class 对象也都是一个 base class 对象。

4.5 避免遮掩继承而来的名称（E-6.33）

（1）derived classes 内的名称会遮掩 base classes 内的名称。在 public 继承下从来没有人希望如此。为了让被遮掩的名称再见天日，可使用 using 声明式或转交函数。

4.6 区分接口继承和实现继承（E-6.34）

（1）接口继承和实现继承不同。在 public 继承之下，derived classes 总是继承 base class 的接口。纯虚函数只具体制定接口继承。非纯虚函数具体执行接口继承及缺省实现继承。non-virtual 函数具体指定接口继承以及强制性实现继承

4.7 考虑 virtual 函数以外的其他选择（E-6.35）

（1）virtual 函数的替代方法包含 NVI 手法以 Strategy 设计模式的多种形式。NVI 手法自身是一个特殊形式的 Template Method 设计模式。

（2）将机能从成员函数移到 class 外部函数，带来的一个缺点是，非成员函数无法访问 class 的 non-public 成员。

（3）tr1::function 对象的行为就像一般函数指针。这样的对象可接纳“与给定之目标签名式兼容”的所有可调用物。

4.8 绝不重新定义继承而来的 non-virtual 函数（E-6.36）

（1）绝对不要重新定义继承而来的 non-virtual 函数

4.9 绝不重新定义继承而来的缺省参数值（E-6.37）

（1）绝对不要重新定义一个继承而来的缺省参数值，因为缺省参数值都是静态绑定，而 virtual 函数——你唯一应该覆写的东西——却是动态绑定。

4.10 重合（E-6.38）

（1）复合的意义和 public 继承完全不同。通过重合塑模出 has-a 或“根据某物实现出”

（2）在应用域，复合意味着 has-a。在实现域，复合意味着 is-implemented-in-terms-of。

4.11 私有继承（E-6.39）

（1）明智而审慎地提出私有继承。private 继承意味着 is-implemented-in-terms-of。它通常比复合的级别低。但是当派生类需要访问 protected 的基类成员，或需要重新定义继承而来的虚函数时，这么设计是合理的。

（2）和复合不同，private 继承可造成 empty 基类最优化。这对致力于“对象尺寸最小化”的程序库开发者而言，可能很重要。

4.12 多重继承（E-6.40）

（1）明智而审慎地提出多重继承。多重继承比单一继承复杂。它可能导致新的歧义性，以及对 virtual 继承的需要。

（2）virtual 继承会增加大小、速度、初始化复杂度等等成本。如果虚基类不带任何数据，将是最具实用价值的情况。

（3）多重继承的确有正当用途。其中一个情节涉及“public 继承某个 Interface class”和“private 继承某个协助实现的 class”的两相组合。

5、设计和声明

5.1 让接口容易被使用，不易被误用

（1）好的很容易被正确使用，不容易被误用。你应该在你的所有接口中努力达成这些性质。

（2）“促进正确使用”的办法包括接口的一致性，以及与内置类型的行为兼容。

（3）“阻止误用”的办法包括建立新类型、限制类型上的操作，束缚对象值，以及消除客户的资源管理责任。

（4）tr1::shared_ptr 支持定制型删除器。这可防范 DLL 问题，可被用来自动解除互斥锁等等。

5.2 以 pass-by-reference-to-const 替换 pass-by-value（E-4.20）

（1）只有三种情况下使用 pass-by-value。其余情况使用 pass-by-reference-to-const。

①内置类型②STL 的迭代器③函数对象

（2）使用 pass-by-reference-to-const 有两个好处。

①回避很多无效的构造和析构函数，因为 pass-by-value 会产生对象，因此会多次调用构造和析构函数，而 pass-by-reference 不产生对象，因此可以回避这些，提高效率。②避免对象切割问题。

5.3 避免返回 handles 指向的对象内部成分（E-5.28）

（1）避免返回 headles（包括 references、指针、迭代器）指向对象内部。遵守这个条款可增加封装性，帮助 const 成员函数的行为像个 const，并将发生“虚吊号码牌”的可能性降至最低。

5.4 将文件之间的编译依存关系降至最低（E-5.31）

（1）支持“编译依存性最小化”的一般构想是：相依于声明式，不要相依于定义式。基于此构想的两个手段是 Handle classes 和 Interface classes。

（2）程序库头文件应该以“完全且仅有声明式”的形式存在。这种做法不论是否涉及 templates 都适用。

5.5 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。

5.6 考虑写一个不抛出异常的 swap 函数（E-4.25）

（1）当 std::swap 对你的类型效率不高时，提供一个 swap 成员函数，并确定这个函数不抛出异常

（2）如果你提供一个 member swap，也该提供一个 non-member swap 用来调用前者。对于 class（而非 templates），也请特化 std::swap

（3）调用 swap 时应针对 std::swap 使用 using 声明式，然后调用 swap 并且不带任何“命名空间资格修饰”

（4）为“用户定义类型”进行 std templates 全特化是好的，但千万不要尝试在 std 内加入某些对 std 而言全新的东西。

5.7 以对象管理资源（E-3.13）

（1）为防止资源泄露，请使用 RALL 对象，它们在构造函数中获得资源并在析构函数中释放资源。

（2）两个常被使用的 RALL class 分别是 tr1::shared_ptr 和 auto_ptr。前者通常是较佳选择，因为其 copy 行为比较直观。若选择 auto_ptr，复制动作会使它（被复制）指向 null。

6、异常

6.1 别让异常逃离析构函数（E-2.8）

（1）析构函数绝对不要吐出异常。如果一个被析构函数调用的函数可能抛出异常，析构函数就应该捕捉任何异常，然后吞下他们（不传播）或者结束程序。

（2）如果客户需要对某个操作函数运行期间抛出的异常做出反应，那么 class 应该提供一个普通函数（而非析构函数）中执行该操作。

6.2 为“异常安全”而努力是值得的（E-5.29）

（1）异常安全函数即使发生异常也不会泄露资源或允许任何数据结构败坏。这样的函数区分为三种可能的保证：基本型、强烈型、不抛异常型。

（2）“强烈保证”往往能够以 copy-and-swap 实现出来，但“强烈保证”并非对所有函数都可实现或具备现实意义。

（3）函数提供的“异常安全保证”通常最高只等于其所调用之各个函数的“异常安全保证”中的最弱者。

6.3 以 by-reference 的方式捕捉异常（ME-14）

（1）示例如下：

```
Void DoSomthing()
{
    Try{
        someFunction();
    }
    Catch(exception& ex)    //这里使用引用捕获异常
    {
       …
    }
}
```

7、其他 C++特性

7.1 引用函数

（1）函数形参表中，输入参数为值或者常数引用，输出参数为引用。

7.2 new 和 delete

（1）new 和 delete；new[]和 delete[]成对使用。

7.3 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。

7.4 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

7.5 运行时类型识别

（1）禁止使用 RTTI

7.6 类型转换（E-5.27）

（1）如果可以，尽量避免类型转换，特别是在注重效率的代码中避免 dynamic_cast。如果有个设计需要类型转换，试着发展无需类型转换的替代设计。

（2）如果类型转换是必要的，试着将它隐藏于某个函数背后。客户随后可以调用该函数，而不需要将类型转换放进他们自己的代码内。

（3）如果要进行类型转换，使用 C++风格而不要使用 C 风格的类型转换

①static_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

②const_cast:移除 const 属性

③reinterpret_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

④dynamic_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

7.7 流

（1）不要使用流，除非是日志接口需要，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，不要在代码中使用流

7.8 智能指针

（1）不使用 auto_ptr，所有能用 auto_ptr 的地方，均使用 unique_ptr 替代

（2）使用 unique_ptr 管理具备专属所有权的资源

（3）使用 shared_ptr 管理具备共享所有权的资源

（4）类似 shared_ptr 但有可能为空的指针，使用 weak_ptr

7.9 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。

7.10 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。

7.11 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）

7.12 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不要在.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。

7.13 0 和 nullptr

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

7.14 sizeof

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

7.15 Boost 库

（1）C++标准程序库的主要机能由 STL、iostream、locales 组成。并包含 C99 标准程序库。TR1 添加了智能指针、一般化函数指针、hash-based 容器、正则表达式以及另外 10 个组件的支持。TR1 自身只是一份规范。为获得 TR1 提供的好处，你需要一份实物。一个好的实物来源是 Boost。因此，要熟悉 Boost。

（2）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：

①Compressed Par: boost/compressed_pair.hpp

②Pointer Container: boost/ptr_container 不包括 ptr_array.hpp 和序列化还有一些别的。

7.16 前置版本的递减递增运算符和后置版本的递减递增运算符

（1）建议养成使用前置版本的习惯。前置版本的递增运算符避免了不必要的工作，它把值加 1 后直接返回改变了的运算对象。与之相比，后置版本需要将原始值存储下来以便于返回这个未修改的内容。如果我们不需要修改之前的值，那么后置版本的操作就是一种浪费。对于整数和指针类型来说，编译器可能对这种额外的工作进行一定的优化；但是对于相对复杂的迭代器类型，这种额外的工作就消耗巨大了。

7.17 位运算

（1）位运算中，关于符号位如何处理并没有明确的规定，所以强烈建议仅将位运算用于处理无符号类型。

7.18 以 const,enum,inline 代替#define（E-1.2）

（1）对于单纯变量，以 const,enum 代替#define

（2）对于形似函数的宏，改用 inline 函数代替#define

7.19 使用限定作用域的枚举型别

（1）限定作用域枚举型别如下：

```
enum class Color { black, white, red };
```

使用这种替代之前的非限定作用域的枚举型别(enum Color { black, white, red };)

7.20 使用 using 替换 typedef

（1）using 比 typedef 可读性更强，并且 using 可以模板化，typedef 不行。两种写法分别如下：

```
using FP = void (*) (int, const std::string&);
typedef void (*FP) (int, const std::string&);
```

7.21 操作符重载

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

等函数进行替代。

（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。

8、模板和泛型编程

8.1 了解隐式接口和编译期多态

（1）class 和 templates 都支持接口和多态。

（2）对 class 而言接口是显式，以函数签名为中心。多态则是通过 virutal 函数发生于运行期。对 template 参数而言，接口是隐式的，基于有效表达式。多态则是通过 template 具现化和函数重载解析发生于编译器。

8.2 了解 typename 的双重意义

（1）声明 template 参数时，前缀关键字 class 和 typename 可互换

（2）请使用关键字 typename 标识嵌套从属类型名称；但不得在 base class lists(基类列)或参数初始化列表内以它作为基类修饰符。

8.3 学习处理模板化基类内的名称

（1）可在 derived class templates 内通过“this->”指涉 base class templates 内的成员名称，或藉由一个明白写出的“基类资格修饰符”完成。

8.4 将于参数无关的代码抽离 templates

（1）templates 生成多个 classes 和多个函数，所以任何 template 代码都不该与某个造成膨胀的 template 参数产生相依关系。

（2）因非类型模板参数而造成的代码膨胀，往往可消除，做法是以函数参数或 class 成员变量替换 template 参数。因类型参数而造成的代码膨胀，往往可降低，做法是让带有完全相同二进制表述的具现类型共享实现码。

8.5 运用成员函数模板接收所有兼容类型

（1）请使用 member function templates（成员函数模板）生成“可接受所有兼容类型”的函数。

（2）如果你声明 member templates 用于“泛化拷贝构造函数”或“拷贝赋值操作”，你还是需要声明正常的拷贝构造函数和拷贝赋值操作符。

8.6 需要类型转换时请为模板定义非成员函数

（1）当我们编写一个 class template，而它所提供的之“与此 template 相关的”函数支持“所有参数之隐式类型转化”时，请将那些函数定义为“class template 内部的 friend 函数”

8.7 请使用 traits classes 表现类型信息

（1）traits classes 使得“类型相关信息”在编译期可用。它们以 templates 和“tempaltes 特化”完成实现。

（2）整合重载技术后，traits classes 有可能在编译器对类型执行 if...else 测试。

8.8 认识 template 元编程

（1）模板元编程可将工作由运行期推往编译器，因而得以实现早期错误侦测和更高的执行效率。

（2）模板元编程可被用来生成“基于政策选择组合”的客户定制代码，也可用来避免生成对某些特殊类型并不适合的代码。

9、命名约定

9.1 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。

9.2 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    //内联函数的定义
```

9.3 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 {…}
```

9.4 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;
}
```

9.5 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;
```

9.6 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 get_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。

9.7 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

9.8 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
};
```

9.9 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
```

9.10 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

10、注释

10.1 注释风格

（1）使用//或者/**/都行，统一就行。

10.2 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 …
```

10.3 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。

10.4 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.
*/
```

10.5 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。

10.6 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。

10.7 标点、拼写和语法

（1）留意标点、拼写和语法等的书写。

10.8 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
```

11、格式

11.1 行长度

（1）每一行的代码字符数不超过 80。

11.2 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。

11.3 空格还是制表位

（1）只使用空格，每次缩进 4 个空格。

11.4 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返回值总是和函数名在同一行

②函数名和左圆括号之间没有空格

③圆括号和参数之间没有空格

④所有形参尽可能对齐，换行的形参缩进 4 个空格

⑤如果函数是 const 的，关键字 const 应与最后一个参数位于同一行

11.5 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);
```

11.6 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
{
}
```

11.7 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
}
```

11.8 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;
```

11.9 布尔表达式

（1）如果一个布尔表达式超过标准行宽，换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))
{
    …
}
```

11.10 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);

11.11 变量及数组初始化

（1）初始化使用=或者()都可以,比较常用=

```
string name = “some name”;
```

或者 string name(“some name”);

11.12 预处理指令

（1）预处理指令要顶着行首写，不要缩进。

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

11.13 类格式

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

```
class MyClass : public OtherClass
{
public:
    MyClass();
```

```
    ~MyClass();

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

protected:
    void SomeInterFunction();

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

11.14 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)
{
    …
}
```

11.15 命名空间格式化

（1）命名空间的内容不缩进。

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

11.16 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;        //…
```

[1]C++ Primer P265