内联函数

* 对于inline的内联函数，由于G++编译器本身的原故，必须要指定 –O以上优化才有效果
* 内联函数如果是作为对外接口，需要把整个函数写到头文件中才能起作用。
* 如果内联函数逻辑比较复杂，建议把实现写到 .in-h文件中。

引用

* 在函数参数中使用引用，容易不经意修改了参数的值从而引起错误，另一个方面引用没有空引用的情况，不能像指针那样返回NULL，所以在函数参数中不建议使用非const的引用，作传出参数使用的时候，建议使用指针。

如:

foo(myclass &s); //不建议

foo(const myclass &s); //允许

foo(myclass \*s); //允许

考虑一些特殊情况，比如需要实现一些特殊的兼容第三方风格接口（比如 一些STL接口)的情况，返回值可以采用非const的引用，重载了[]运算符的情况，但重载运算符除非必要也是不推荐使用的

命名空间

* 在C语言编码规范中对于.cpp内部使用的函数和变量要求用static开头，在C++还可以使用匿名的命名空间达到相同的作用，而且好处在于不需要像static那样，一个个函数加过去，不小心会有遗漏，只需要用一个 namespace{ }就可以把它们都包括起来。
* 对外接口必须使用具名命名空间
* 匿名命名空间不允许在.h文件中使用，从逻辑上我们要把对内使用和对外使用的接口，类型分开。
* 命名空间的名字不能使用std
* 基础库中以后增加的C++库都会有对应命名空间，这些名字都不能允许被项目组随便使用，比如现在已经确定的bsl
* 不允许使用 using namespace XXX的形式

这样写虽然方便，但是不注意的话，还是免不了命名空间污染的问题，另外阅读也不太方便，std::sort一看就知道是STL提供的sort,而不是其它。

* 命名空间的名字按照产品线区分，避免各产品线之间出现混乱，特别是有代码需要公开被其他产品线使用
* 对于使用了全局函数比如snprintf等函数推荐加上::的前缀

STL的使用

* 使用STL算法的时候，请注意算法中使用的容器是否考虑到了线程安全问题
* 我们使用的STL版本中，32位平台下的string 类即使在一个线程一个实例的情况下也存在线程安全问题，多线程程序中不允许使用std::string，后续COM组会推出我们自己的STL容器。
* 建议看过Effective STL等书对STL本身有所了解后再使用，避免陷入STL的一些陷阱中。

类

* 构造函数
  + 不要在构造函数中做复杂的初始化工作，复杂的初始化工作使用专门的函数代替，
  + 构造函数中只做诸如变量赋初值等工作
  + 初始化列表中不能使用成员变量进行初始化,不到万不得已不使用引用成员 \* 构造函数没有返回值，不易捕捉到错误，而且在构造函数抛出异常是非常危险的事情。如果需要抛异常，需要自己注意清理现场。 \* 对于使用参数的构造函数，要用explicit显示调用。不经意的隐式转化的容易产生混乱
* class 与 struct

C++中 struct 可以认为是默认public 的class, 考虑与C语言语法的兼容， struct就只能表示数据，不能存在任何成员函数。

* 重载

不推荐使用运算符重载，因为这样会不经意间改变符号的行为，甚至产生混淆，比如 p 重载了 + 号， 在p+1 与 &p+1 的情况会混乱。

* const

在C编程规范中已经有强调const的使用，这里强调的是对于类的成员函数，能在函数后面加上const 就尽量加上。

其他注意

* 不推荐使用函数重载，即使需要使用函数重载也应该是那种只有变量不同，但行为一致的情况。

不推荐原因是不利于代码阅读，尤其是重载加默认参数非常容易出错，另外错误的使用也会导致函数重载的二义性。当心隐式类型转换导致重载函数产生二义性，另外重载和省略参数的混合使用情况下，也要注意二义性的问题。特别是NULL很容易被识别为0。编写模版的时候也要特别注意。

* 不推荐C++的异常机制

C++的异常机制，可以使上层函数方便的处理由底层函数发生的异常状态，而且也可以方便将异常集中处理。但异常的滥用会导致更多的问题，所以如果要使用异常机制，需要确认自己对异常机制有足够的了解，特别的对于析构函数不能抛异常

* 不推荐使用RTTI

如果需要使用RTTI，不能出现对一种类型进行一种处理的情况，这会给代码维护带来不便。

* new的异常

new的时候在C++失败会抛出bad\_alloc的异常，除非使用 (std::nothrow)， 如果对使用异常没有把握，可以使用(std::nothrow)，然后通过判断指针是否为NULL来判断是否分配内存失败。

* ++i 和 i++

当i 是class 重载了 ++, 推荐使用++i， 因为 i++会多进行一次拷贝（构造）操作，比如在使用迭代器的情况下。

* 强制类型转化

推荐使用C++风格的强制类型转化，包括static\_cast, dynamic\_cast, reinterpret\_cast, const\_cast 。 C风格的强制类型转化有时会造成混乱。像int(3.5)是强制转化，而int”hello”是在做类型转化，意义不一样。使用C++风格的强制类型转化可以明确转化的意义。而且C++的强制转化还有一个好处是可以方便查找。

* 不推荐使用多重继承

代码风格

* 常量的定义使用大写，单词间用‘\_’连接。例如

const int MAX\_NUM = 100;

* 在类中，由于普通常量无法进行类似 int array[MAX\_XXX]的定义，此时可采用enum的方式实现，或者采用static const的方式也可以
* 特别的对于字符串常量建议使用下面的形式

const char STR[] = “xxxx”

不要用

const char \*STR = “xxxx”;

* 变量命名习惯上有两种方式，linux下习惯的命名方式为小写字母 + 下划线的方式；例如：current\_count 或者大小写字母混合的方式，例如：currentCount;两种命名方式各有千秋吧，但在基础库中，一律使用linux的命名方式。没有特别的好处，只是符合大多数linux程序员的习惯，保持统一。
* 全局变量需要以“g\_”开头。例如：g\_thread\_data。
* 命名空间的名字与普通变量一直，使用大小写+下划线的形式
* 结构体struct命名的尾部以\_t表示为类型定义,如命名的尾部以\_t表示为类型定义
* 类的命名方式使用PASCAL的命名方式，使用首字母大写的形式如

class MyClass {

…

}

与结构体可以很容易区分

* 类的私有成员变量使用 \_开头， 成员函数中的局部变量不使用\_开头

私有成员函数也以\_开头

class MyClass {

public:

int foo(int arg);

private:

int \_my\_foo(int i);

int \_arg;

};

λ 每个类之前都要用doxygen的风格注释对类进行相关说明，如:

/\*\*

\* @brief MyClass类

\*

\* XXXX 类的说明

\*\*/

class MyClass {

…

}

作为基础需要保证doxygen注释的解析正确。

* 一个.h文件对应一个class, 类名与文件名一一对应
* 所有的数据成员都应该是私有的，如果需要使用，可以采用存取函数的方法以 set\_foo() 和get\_foo()的形式给外部接口调用。如

class MyClass {

public:

...

inline int get\_num() const { return \_num; }

inline int set\_num(int num) { \_num = num; return 0;}

private:

int \_num;

};

* 类的声明属性依次序使用public:、protected:、private

推荐使用的缩进格式：

统一缩进采用4个空格进行缩进

public, private使用2格缩进，比4少，这样比较容易阅读

使用template的地方和后面跟函数名或类名对齐。

namespace 顶格写，namespace里的函数也顶格

这些缩进主要考虑到程序的美观和阅读问题。