# 编程规范

**编程规范**

版本： 1.0

日期： 2020-09-18

**修订记录**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **日期** | **修订版本** | **修改描述** | **作者** | **审核** |
| 2020-09-18 | 1.0 | 草拟 |  |  |
| 2021-09-07 | 1.1 | 修改 |  |  |

目录

[1 目标 4](#_Toc51622862)

[2 代码规范 4](#_Toc51622863)

[2.1 命名 4](#_Toc51622864)

[2.2 头文件 4](#_Toc51622865)

[2.3 作用域 5](#_Toc51622866)

[2.4 静态和全局变量 5](#_Toc51622867)

[2.5 Thread\_local变量 6](#_Toc51622868)

[2.6 类 6](#_Toc51622869)

[2.6.1 构造函数 6](#_Toc51622870)

[2.6.2 继承 6](#_Toc51622871)

[2.6.3 多重继承 6](#_Toc51622872)

[2.6.4 接口 7](#_Toc51622873)

[2.6.5 操作符重载 7](#_Toc51622874)

[2.6.6 类声明顺序 7](#_Toc51622875)

[2.6.7 函数 7](#_Toc51622876)

[2.7 C++特性 8](#_Toc51622877)

[2.8 注释 8](#_Toc51622878)

[2.8.1 程序文件头注释 8](#_Toc51622879)

[2.8.2 方法头注释 9](#_Toc51622880)

[2.8.3 类注释 9](#_Toc51622881)

[3 代码格式 9](#_Toc51622882)

[4 补充 9](#_Toc51622883)

# 1 目标

编程规范是为了使软件代码具有可读性，保证软件质量，因此制定本文档。

* 编程规范需要能让大部分工程师记住；

* 编程规范是为了让代码具有可读性；

* 编程规范是为了避免危险的操作从而出现bug；

* 编程规范是为了保证代码容易维护。

# 2 代码规范

代码规范可以配合使用google的cpplint工具检查，看是否满足google规范要求。

## 2.1 命名

* 局部变量名单词之间使用下划线隔开；

* 类的变量成员用下划线作为前缀如 \_file\_name;

* 类的函数名使用驼峰类型；如doSomething();

* 类的成员存取使用 如get\_file\_name() set\_file\_name()；

* 类名是PASCAL风格，即首字母大写 如MyClass；

* 常量用k作为前缀后面是PASCAL风格如 kFileName;

* 全局变量用g作为前缀后面是PASCAL风格如 gFileName;

* 接口类必须以Interface为后缀；

* 宏定义全大写，中间用下划线隔开 FILE\_NAME;

* 源文件名和类名保持一致，采用首字母大写PASCAL风格， 如MyClass.cc MyClass.h。

## 2.2 头文件

* 头文件应该自包含；

模板和inline函数的定义应该和声明一起放在头文件中。

* 使用#define Guard防止头文件重复包含；

*<PROJECT>*\_*<PATH>*\_*<FILE>*\_H\_。

* 避免使用前向声明，尽量使用包含头文件；

* 少于10行的代码定义为inline。

* 头文件应该自包含顺序如下

Related header,（cc文件对应的.h文件）

C system headers,

C++ standard library headers,

other libraries' headers,

your project's headers

## 2.3 作用域

* Namespace不要用using namespace xxx，避免名字污染；

* 不要使用inline namespace；

* Namespace结尾加上注释表明namespace的名字；

* 不要在头文件中使用namespace别名，除非内部使用；

* 对于.cc文件中不再其他文件中使用的定义，把它放入匿名空间或者static修饰。

* 局部变量定义在一个local scope，并尽量靠近使用的地方；

* 局部变量定义和声明在一起；

* 对于if，while，for语句中的变量，可以声明在这些语句中；

* 如果一个变量在for循环中每次循环触发构造和析构则放到循环外面。

## 2.4 静态和全局变量

* 由于全局变量的初始化和释放执行顺序是不确定的，因此全局变量的使用不应该依赖于其他条件；

* 除非析构是trivially destructible（即什么都没做），全局变量才能是类的对象；

* 全局变量初始化不能依赖类的构造，或者只能是constexpr；

* Global Strings，考虑使用简单的字符数组，或者字符指针；

* Maps，sets，或者其他容器，可以考虑用数组，或者用function-local static指针变量；

* 智能指针(unique\_ptr，shared\_ptr)由于在结束时析构指针，因此不能作为全局变量；

* 对于用户自定义类类型作为全局变量，需要使用constexpr构造函数，以及trivial析构函数；

* 最后，如果以上都做不到，则可以创建一个function-local static pointer或者reference。static const auto& impl = \*new T(args...)。

## 2.5 Thread\_local变量

* Thread\_local变量的生命周期从线程创建时存在，在线程结束时终止，每个线程有自己的变量副本；

* Thread\_local变量类似于全局变量，但它是在线程启动时初始化，另外它无须是trivial析构的；

* Thread\_local变量在函数内使用没有限制，但作为全局变量或者class静态变量则初始化必须是在编译时确定，即constexpr。

## 2.6 类

### 2.6.1 构造函数

* 构造函数中不要使用类的virtual函数，因为在基类中可能不会调用子类的实现；

* 构造函数中避免初始化失败的情形出现；

* 避免隐式转换，单变量constructor使用explicit关键字，对于多变量的constructor如果不想使用initialization list，也可以使用explicit关键字；

* 类的copy和move构造函数如果不支持，需要声明为deleted。如果一个类会被继承，则避免支持move操作；

* 除了只包含公有数据的类使用struct，其他使用class；

* 没有构造函数时应提供默认构造函数。

### 2.6.2 继承

* 所有继承必须是public；

* 如果需要使用private继承，使用组合比继承更合适；

* 基类的析构函数需为virtual；

* 虚函数加上override或者final，可以检查出虚函数的继承错误。

### 2.6.3 多重继承

* 最多一个基类中含有实现，其他类作为接口才用多重继承。

### 2.6.4 接口

* 只有纯虚函数和静态函数，没有非静态数据成员；

* 提供虚析构函数；

* 不提供构造函数，提供的话，声明为protected。

### 2.6.5 操作符重载

* 谨慎使用重载操作符，尤其是赋值操作；

* 使用重载操作符，需要让用户使用起来直观，如定义了==，就应该同时支持!=；

* 不要使用用户自定义literals，即不要重载操作符””。

### 2.6.6 类声明顺序

* public在private之前；

* 成员函数在数据成员前；

* 按照public protected private的顺序；

* 按照typedef，using和enums，常量，构造函数，拷贝构造函数，移动构造函数，析构函数，赋值操作符，成员存取set函数及get函数，其他成员函数含静态成员函数，数据成员含静态数据成员的顺序。

### 2.6.7 函数

* 函数应该尽量短小精悍；

* 函数参数如果不会在函数内发生修改则使用引用时加上const；

* 使用重载时需要能让读者一眼看出什么情况会调用哪个重载；

* 禁止在虚函数中使用缺省函数参数；

* 尽量少用拖尾返回类型；

* 优先使用std::unique\_ptr，不用std::auto\_ptr。

## 2.7 C++特性

* 使用右值引用支持move和完美转发操作；

* 合理使用友元；

* 不要使用C++异常；

* 避免使用RTTI（Run-Time Type Information）；

* 使用C++类型转换；

* 只在记录日志时使用流，其他使用printf；

* 使用前置而不是后置++；

* 使用constexpr来作为更鲁棒的常数接口；

* 需要确定位数时可以在stdint.h中定义有int16\_6，uint32\_t，int64\_t；

* 指针和整型的转换用intptr\_t，这样能保证32和64位平台的可移植性；

* 使用nullptr作为void pointer；

* sizeof(var)而不是sizeof(type)；

* 只在类型对读代码不重要时使用auto；

* lambda指定直接的capture；

* 只用成熟的boost库；

* 不要特化std::hash；

* 使用using创建别名，优于typedef。

## 2.8 注释

采用DoxyGen风格的注释。

### 2.8.1 程序文件头注释

应该包含如下：

* 文件描述（Description）：描述此类的作用；

* 作者（Author）：创建者或者修改者名；

* 日期（Date）：创建或者修改时的日期，使用“-”进行年月日分割。

### 2.8.2 方法头注释

方法头如以下模板：

/\*\*

\* @brief push item.

\*

\* push an item to stack.

\*

\* @param a the item which will be pushed.

\* @retval true or false.

\* @note test doxy note.

\* @attention test doxy attention.

\* @warning test doxy warning.

\* @exception test doxy exception.

\*/

### 2.8.3 类注释

应该包含如下信息

* 类的简要描述；

* 类的详细描述 。

* 类的成员变量描述

/\*\*

\* @brief my test stack.

\*

\* This is detail stack description.

\*/

template<typename T>

class MyStack

{

vector<T> stack = {}; ///< stack vector

# 3 代码格式

采用ClangFormat工具，按照google风格配置，并格式化代码即可。

# 4 多工具编译

## 4.1 避免头文件的依赖编译

不要随便在头文件尤其是公有头文件中引入依赖，有些头文件可以直接在cpp中include即可。

## 4.2 文件命名和类命名

增荣老师在集成编译多个工具时，由于工具之间出现了文件重名和类重名问题，在编译时暴露了很多问题，请各位同学以后在文件命名和类命名时最好加上模块名前缀，没有小模块就加上大模块比如Cts等等。

## 4.3 Cmake

链接时需要区分include\_directory和target\_include\_directory，前者会传递，后者不会，如果不想传递给其他工具就用后者，这样能减少编译冲突问题。

# 5 补充

后续还需要不断补充该规范。