**C++编码规范**

1. **代码注释**

**1.文件注释**

在C++的代码都是分布在一个个的头文件和源文件中，如果是用的VS则是h后缀文件和cpp后缀文件。当然了有时你用以内联函数还可能会有inl文件，每个文件的开头部分你可以添加注释信息。

内容一般是:版权，作者，编写日期，功能描述等例如：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Copyright：MySchool

Author:杨鹤星

class：信1604-2班

number：20163792

Data：2019/6/14

Description：输入三个顶点，判定是否能组成三角形，若能，则判定组成的三角形是普通三角形还是特殊三角形

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**2.类注释**

一般是简单的说下这个类的功能，如果你在文件开头已经描述了这里也可以省略。如果要注释的话就可以这样简单的描述下：

//point

class point //顶点类定义

{

.......

}

**3.函数注释**

**总述：**函数声明处的注释描述函数功能；定义处的注释描述函数实现。

**说明：**函数声明处注释的内容例如：

函数的输入输出。

对类成员函数而言: 函数调用期间对象是否需要保持引用参数，是否会释放这些参数。

函数是否分配了必须由调用者释放的空间。

参数是否可以为空指针。

是否存在函数使用上的性能隐患。

如果函数是可重入的，其同步前提是什么?

举例如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

输入输出：输入顶点坐标，输出各边长。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void triangle::lenth(point&a)//计算边长函数

{

........

}

**4.变量注释**

**总述：**

通常变量名本身足以很好说明变量用途。某些情况下，也需要额外的注释说明。

**说明**

类数据成员：

每个类数据成员 (也叫实例变量或成员变量) 都应该用注释说明用途。如果有非变量的参数(例如特殊值，数据成员之间的关系，生命周期等)不能够用类型与变量名明确表达，则应当加上注释。然而，如果变量类型与变量名已经足以描述一个变量，那么就不再需要加上注释。

特别地，如果变量可以接受 NULL 或 -1 等警戒值，须加以说明。比如:

public://公有类

double x1,x2,x3; //三个顶点横坐标

double y1,y2,y3;//三个顶点纵坐标

全局变量：和数据成员一样，所有全局变量也要注释说明含义及用途， 以及作为全局变量的原因。

1. **#define 保护**

所有头文件都应该使用 #define 来防止头文件被多重包含，命名格式当是: <PROJECT>\_<PATH>\_<FILE>\_H\_ 。

为保证唯一性，头文件的命名应该基于所在项目源代码树的全路径。例如：项目point中的头文件可按如下方式保护:

#ifndef point\_H //#define保护，防止头文件被重复包含

#define point\_H

....

#endif//point\_H

1. **前置声明**

尽可能地避免使用前置声明。使用 #include 包含需要的头文件即可。

尽量避免前置声明那些定义在其他项目中的实体。

函数：总是使用#include。

类模板：优先使用#include。

#include "point.h"

前置声明能够节省编译时间，多余的 #include 会迫使编译器展开更多的文件，处理更多的输入。前置声明能够节省不必要的重新编译的时间。 #include 使代码因为头文件中无关的改动而被重新编译多次。

1. **条件语句**

倾向于不在圆括号内使用空格。关键字if和else另起一行。

if(AB==AC&&AC==BC){// 圆括号里没有空

x=1; //空格缩进

}else if(AB==AC||AB==BC||AC==BC){// else 与 if 的右括号同一行

y=1;//该三角形为等腰三角形

}

1. **类格式**

总述：

访问控制块的声明依次序是 public:， protected:，private: 每个都缩进 1 个空格。

说明：

类声明 (下面的代码中缺少注释，参考类注释) 的基本格式如下:

class triangle{//边类定义

public://公有类// 注意有一个空格的缩进

void lenth(point&);//计算边长//标准的两空格缩进

void compare();//比较边长

private://私有类

double AB,AC,BC;//三角形三条边

double AB2,AC2,BC2;//三角形三条边的平方

};

1. **局部变量**

将函数变量尽可能置于最小作用域内，并在变量声明时进行初始化。

C++ 允许在函数的任何位置声明变量。我们提倡在尽可能小的作用域中声明变量，离第一次使用越近越好。 这使得代码浏览者更容易定位变量声明的位置，了解变量的类型和初始值。特别是，应使用初始化的方式替代声明再赋值，比如:

int i;

i = f(); // 坏—初始化和声明分离

int j = g(); // 好—初始化时声明

vector<int> v;

v.push\_back(1); // 用花括号初始化更好

v.push\_back(2);

vector<int> v = {1，2}; // 好—v 一开始就初始化

属于 if，while 和 for 语句的变量应当在这些语句中正常地声明，这样子这些变量的作用域就被限制在这些语句中了，举例而言:

while (const char\* p = strchr(str，'/')) str = p + 1;

在循环作用域外面声明这类变量要高效的多

Foo f; // 构造函数和析构函数只调用 1 次

for (int i = 0; i < 1000000; ++i) {

f.DoSomething(i);

}

1. **函数**

函数的参数顺序为: 输入参数在先，后跟输出参数。

**说明：**

C/C++ 中的函数参数或者是函数的输入，或者是函数的输出，或兼而有之。 输入参数通常是值参或 const 引用，输出参数或输入/输出参数则一般为非 const 指针。在排列参数顺序时，将所有的输入参数置于输出参数之前。 特别要注意，在加入新参数时不要因为它们是新参数就置于参数列表最后，而是仍然要按照前述的规则，即将新的输入参数也置于输出参数之前。

这并非一个硬性规定。输入/输出参数 (通常是类或结构体) 让这个问题变得复杂。并且，有时候为了其他函数保持一致，你可能不得不有所变通。

1. **引用参数**

**总述：**

所有按引用传递的参数必须加上 const。

**定义：**

在 C 语言中，如果函数需要修改变量的值，参数必须为指针，如 int foo(int \*pval)。在 C++ 中，函数还可以声明为引用参数: int foo(int &val)。

优点

定义引用参数可以防止出现 (\*pval)++ 这样丑陋的代码。引用参数对于拷贝构造函数这样的应用也是必需的。同时也更明确地不接受空指针。

函数参数列表中，所有引用参数都必须是 const:

void Foo(const string &in，string \*out);

事实上这在 Google Code 是一个硬性约定: 输入参数是值参或 const 引用，输出参数为指针。输入参数可以是 const 指针，但决不能是非 const 的引用参数，除非特殊要求，比如 swap()。

有时候，在输入形参中用 const T\* 指针比 const T& 更明智。比如:

可能会传递空指针。

函数要把指针或对地址的引用赋值给输入形参。

总而言之，大多时候输入形参往往是 const T&。若用 const T\* 则说明输入另有处理。所以若要使用 const T\*，则应给出相应的理由，否则会使得读者感到迷惑。

1. **函数重载**

**总述：**

若要使用函数重载，则必须能让读者一看调用点就胸有成竹，而不用花心思猜测调用的重载函数到底是哪一种。这一规则也适用于构造函数。

**定义**

你可以编写一个参数类型为 const string& 的函数，然后用另一个参数类型为 const char\* 的函数对其进行重载:

class MyClass {

public:

void Analyze(const string &text);

void Analyze(const char \*text，size\_t textlen);

};

优点

通过重载参数不同的同名函数，可以令代码更加直观。模板化代码需要重载，这同时也能为使用者带来便利。

缺点

如果函数单靠不同的参数类型而重载 (acgtyrant 注：这意味着参数数量不变)，读者就得十分熟悉 C++ 五花八门的匹配规则，以了解匹配过程具体到底如何。另外，如果派生类只重载了某个函数的部分变体，继承语义就容易令人困惑。

结论

如果打算重载一个函数，可以试试改在函数名里加上参数信息。例如，用 AppendString() 和 AppendInt() 等，而不是一口气重载多个 Append()。如果重载函数的目的是为了支持不同数量的同一类型参数，则优先考虑使用 std::vector 以便使用者可以用 列表初始化 指定参数。

1. **缺省参数**

**总述**

只允许在非虚函数中使用缺省参数，且必须保证缺省参数的值始终一致。缺省参数与 函数重载 遵循同样的规则。一般情况下建议使用函数重载，尤其是在缺省函数带来的可读性提升不能弥补下文中所提到的缺点的情况下.

优点

有些函数一般情况下使用默认参数，但有时需要又使用非默认的参数。缺省参数为这样的情形提供了便利，使程序员不需要为了极少的例外情况编写大量的函数。和函数重载相比，缺省参数的语法更简洁明了，减少了大量的样板代码，也更好地区别了 “必要参数” 和 “可选参数”。

缺点

缺省参数实际上是函数重载语义的另一种实现方式，因此所有 不应当使用函数重载的理由 也都适用于缺省参数。

虚函数调用的缺省参数取决于目标对象的静态类型，此时无法保证给定函数的所有重载声明的都是同样的缺省参数。

缺省参数是在每个调用点都要进行重新求值的，这会造成生成的代码迅速膨胀。作为读者，一般来说也更希望缺省的参数在声明时就已经被固定了，而不是在每次调用时都可能会有不同的取值。

缺省参数会干扰函数指针，导致函数签名与调用点的签名不一致。而函数重载不会导致这样的问题。

结论

对于虚函数，不允许使用缺省参数，因为在虚函数中缺省参数不一定能正常工作。如果在每个调用点缺省参数的值都有可能不同，在这种情况下缺省函数也不允许使用。(例如，不要写像 void f(int n = counter++); 这样的代码。)

在其他情况下，如果缺省参数对可读性的提升远远超过了以上提及的缺点的话，可以使用缺省参数。如果仍有疑惑，就使用函数重载。

1. **命名约定**

最重要的一致性规则是命名管理。命名的风格能让我们在不需要去查找类型声明的条件下快速地了解某个名字代表的含义: 类型，变量，函数，常量，宏，等等，甚至。我们大脑中的模式匹配引擎非常依赖这些命名规则。

命名规则具有一定随意性，但相比按个人喜好命名，一致性更重要，所以无论你认为它们是否重要，规则总归是规则。

* 1. **通用命名规则**

总述

函数命名，变量命名，文件命名要有描述性; 少用缩写。

说明

尽可能使用描述性的命名，别心疼空间，毕竟相比之下让代码易于新读者理解更重要。不要用只有项目开发者能理解的缩写，也不要通过砍掉几个字母来缩写单词。

int price\_count\_reader; // 无缩写

int num\_errors; // "num" 是一个常见的写法

int num\_dns\_connections; // 人人都知道 "DNS" 是什么

int n; // 毫无意义.

int nerr; // 含糊不清的缩写.

int n\_comp\_conns; // 含糊不清的缩写.

int wgc\_connections; // 只有贵团队知道是什么意思.

int pc\_reader; // "pc" 有太多可能的解释了.

int cstmr\_id; // 删减了若干字母.

注意，一些特定的广为人知的缩写是允许的，例如用 i 表示迭代变量和用 T 表示模板参数。

模板参数的命名应当遵循对应的分类: 类型模板参数应当遵循 类型命名 的规则，而非类型模板应当遵循 变量命名 的规则.

* 1. **文件命名**

总述

文件名要全部小写，可以包含下划线 (\_) 或连字符 (-)，依照项目的约定。如果没有约定，那么 “\_” 更好。

说明

可接受的文件命名示例:

my\_useful\_class.cc

my-useful-class.cc

myusefulclass.cc

myusefulclass\_test.cc // \_unittest 和 \_regtest 已弃用。

C++ 文件要以 .cc 结尾，头文件以 .h 结尾。专门插入文本的文件则以 .inc 结尾，参见头文件自足。

不要使用已经存在于 /usr/include 下的文件名 (Yang.Y 注: 即编译器搜索系统头文件的路径)，如 db.h.

通常应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，对应于类 FooBar。

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。

* 1. 类型命名

总述

类型名称的每个单词首字母均大写，不包含下划线: MyExcitingClass，MyExcitingEnum。

说明

所有类型命名 —— 类，结构体，类型定义 (typedef)，枚举，类型模板参数 —— 均使用相同约定，即以大写字母开始，每个单词首字母均大写，不包含下划线。例如:

// 类和结构体

class UrlTable { ...

class UrlTableTester { ...

struct UrlTableProperties { ...

// 类型定义

typedef hash\_map<UrlTableProperties \*，string> PropertiesMap;

// using 别名

using PropertiesMap = hash\_map<UrlTableProperties \*，string>;

// 枚举

enum UrlTableErrors { ...

* 1. 变量命名

总述

变量 (包括函数参数) 和数据成员名一律小写，单词之间用下划线连接。类的成员变量以下划线结尾，但结构体的就不用，如: a\_local\_variable，a\_struct\_data\_member，a\_class\_data\_member\_。

说明

普通变量命名

举例:

string table\_name; // 好 - 用下划线.

string tablename; // 好 - 全小写

string tableName; // 差 - 混合大小写

类数据成员

不管是静态的还是非静态的，类数据成员都可以和普通变量一样，但要接下划线.

class TableInfo {

...

private:

string table\_name\_; // 好 - 后加下划线.

string tablename\_; // 好.

static Pool<TableInfo>\* pool\_; // 好.

};

结构体变量

不管是静态的还是非静态的，结构体数据成员都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;

static Pool<UrlTableProperties>\* pool;

};

结构体与类的使用讨论，参考 结构体 vs。类.

* 1. 常量命名

总述

声明为 constexpr 或 const 的变量，或在程序运行期间其值始终保持不变的，命名时以 “k” 开头，大小写混合。例如:

const int kDaysInAWeek = 7;

说明

所有具有静态存储类型的变量 (例如静态变量或全局变量，参见 存储类型) 都应当以此方式命名。对于其他存储类型的变量，如自动变量等，这条规则是可选的。如果不采用这条规则，就按照一般的变量命名规则.

* 1. 函数命名

总述

常规函数使用大小写混合，取值和设值函数则要求与变量名匹配: MyExcitingFunction()，MyExcitingMethod()，my\_exciting\_member\_variable()，set\_my\_exciting\_member\_variable().

说明

一般来说，函数名的每个单词首字母大写 (即 “驼峰变量名” 或 “帕斯卡变量名”)，没有下划线。对于首字母缩写的单词，更倾向于将它们视作一个单词进行首字母大写 (例如，写作 StartRpc() 而非 StartRPC()).

AddTableEntry()

DeleteUrl()

OpenFileOrDie()

(同样的命名规则同时适用于类作用域与命名空间作用域的常量，因为它们是作为 API 的一部分暴露对外的，因此应当让它们看起来像是一个函数，因为在这时，它们实际上是一个对象而非函数的这一事实对外不过是一个无关紧要的实现细节.)

取值和设值函数的命名与变量一致。一般来说它们的名称与实际的成员变量对应，但并不强制要求。例如 int count() 与 void set\_count(int count)

* 1. 命名空间命名

总述

命名空间以小写字母命名。最高级命名空间的名字取决于项目名称。要注意避免嵌套命名空间的名字之间和常见的顶级命名空间的名字之间发生冲突。

顶级命名空间的名称应当是项目名或者是该命名空间中的代码所属的团队的名字。命名空间中的代码，应当存放于和命名空间的名字匹配的文件夹或其子文件夹中。

注意 不使用缩写作为名称 的规则同样适用于命名空间。命名空间中的代码极少需要涉及命名空间的名称，因此没有必要在命名空间中使用缩写。

要避免嵌套的命名空间与常见的顶级命名空间发生名称冲突。由于名称查找规则的存在，命名空间之间的冲突完全有可能导致编译失败。尤其是，不要创建嵌套的 std 命名空间。建议使用更独特的项目标识符 (websearch::index，websearch::index\_util) 而非常见的极易发生冲突的名称 (比如 websearch::util)

对于 internal 命名空间，要当心加入到同一 internal 命名空间的代码之间发生冲突 (由于内部维护人员通常来自同一团队，因此常有可能导致冲突)。在这种情况下，请使用文件名以使得内部名称独一无二 (例如对于 frobber.h，使用 websearch::index::frobber\_internal).

* 1. 枚举命名

总述

枚举的命名应当和 常量 或 宏 一致: kEnumName 或是 ENUM\_NAME.

说明

单独的枚举值应该优先采用 常量 的命名方式。但 宏 方式的命名也可以接受。枚举名 UrlTableErrors (以及 AlternateUrlTableErrors) 是类型，所以要用大小写混合的方式

enum UrlTableErrors {

kOK = 0,

kErrorOutOfMemory,

kErrorMalformedInput,

};

enum AlternateUrlTableErrors {

OK = 0,

OUT\_OF\_MEMORY = 1,

MALFORMED\_INPUT = 2,

};

* 1. 宏命名

总述

一定要用宏，像这样命名: MY\_MACRO\_THAT\_SCARES\_SMALL\_CHILDREN

说明

参考 预处理宏; 通常 不应该 使用宏。如果不得不用，其命名像枚举命名一样全部大写，使用下划线:

#define ROUND(x) ...

#define PI\_ROUNDED 3.0

* 1. 命名规则的特例

总述

如果你命名的实体与已有 C/C++ 实体相似，可参考现有命名策略.

bigopen(): 函数名，参照 open() 的形式

uint: typedef

bigpos: struct 或 class，参照 pos 的形式

sparse\_hash\_map: STL 型实体; 参照 STL 命名约定

LONGLONG\_MAX: 常量，如同 INT\_MAX