7. 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.

7.1. 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.

7.2. 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.

7.3. 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 { ...
```

7.4. 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.

7.5. 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.

7.6. 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)`.

7.7. 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`).

7.8. 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,
};
```

2009 年 1 月之前, 我们一直建议采用 宏 的方式命名枚举值. 由于枚举值和宏之间的命名冲突, 直接导致了很多问题. 由此, 这里改为优先选择常量风格的命名方式. 新代码应该尽可能优先使用常量风格. 但是老代码没必要切换到常量风格, 除非宏风格确实会产生编译期问题.

7.9. 宏命名

总述

你并不打算 使用宏, 对吧? 如果你一定要用, 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
```

7.10. 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`

译者（acgtyrant）笔记

1. 感觉 Google 的命名约定很高明, 比如写了简单的类 QueryResult, 接着又可以直接定义一个变量 query_result, 区分度很好; 再次, 类内变量以下划线结尾, 那么就可以直接传入同名的形参, 比如 `TextQuery::TextQuery(std::string word) : word_(word) {}` , 其中 `word_` 自然是类内私有成员.