参考并采用中译版谷歌C++代码规范

基本规范

1. Google 公司提供了代码规范检查风格错误,python2可以直接复制代码,python3运行该程序有问题,笔者已进行改正,现已上传并提供下载地址

   使用方法:

   ```
   python cpplint.py xxx/xxx.cpp
   ```

   编写完代码检查明显语法错误后请使用该文件判断,请根据其中的错误信息做出修改

   编写完代码检查明显语法错误后请使用该文件判断,请根据其中的错误信息做出修改

   编写完代码检查明显语法错误后请使用该文件判断,请根据其中的错误信息做出修改
2. 务必仔细阅读`7.命名约定`,`8.注释`,`9.格式`.**这一部分十分重要**
3. 项目代码命名规定:
   - 文件命名:下划线连接单词,例`useful_class.c`
   - 类型命名:类型名称的每个单词首字母均大写,例`MyExcitingClass`
   - 变量命名:
     - 普通变量命名:下划线连接,例`table_name`
     - 类数据成员:下划线连接,下划线结尾,例

       ```
       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:同普通变量,下划线连接
   - 常量命名: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:`const int kDaysInAWeek = 7;`
   - 函数命名: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”)

     ```
     AddTableEntry()
     DeleteUrl()
     OpenFileOrDie()
     ```

   - 命名空间命名:不使用缩写,下划线连接,例`websearch::index::frobber_internal`
   - 宏/枚举命名:全大写,下划线连接,例`MY_MACRO_THAT_SCARES_SMALL_CHILDREN`

4. 项目代码注释规定,统一使用 `//` 进行注释,不使用 `/**/`:

   - 文件注释:在每一个文件开头加入版权公告.该篇代码主要实现的功能,代码的版权,作者,编写日期等。如果你对原始作者的文件做了重大修改, 请考虑删除原作者信息.
   - 类注释:每个类的定义都要附带一份注释, 描述类的功能和用法, 除非它的功能相当明显.如果类的声明和定义分开了(例如分别放在了 `.h` 和 `.c` 文件中), 此时, 描述类用法的注释应当和接口定义放在一起, 描述类的操作和实现的注释应当和实现放在一起.
   - 函数注释:基本上每个函数声明处前都应当加上注释, 描述函数的功能和用途. 只有在函数的功能简单而明显时才能省略这些注释
     - 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?

     ```
     // Returns an iterator for this table.  It is the client's
     // responsibility to delete the iterator when it is done with it,
     // and it must not use the iterator once the GargantuanTable object
     // on which the iterator was created has been deleted.
     //
     // The iterator is initially positioned at the beginning of the table.
     //
     // This method is equivalent to:
     //    Iterator* iter = table->NewIterator();
     //    iter->Seek("");
     //    return iter;
     // If you are going to immediately seek to another place in the
     // returned iterator, it will be faster to use NewIterator()
     // and avoid the extra seek.
     Iterator* GetIterator() const;
     ```

     同时也应避免啰嗦,如在函数命名中体现

     ```
     // Returns true if the table cannot hold any more entries.
     bool IsTableFull();
     ```

     如果函数的实现过程中用到了很巧妙的方式, 那么在函数定义处应当加上解释性的注释. 例如, 你所使用的编程技巧, 实现的大致步骤, 或解释如此实现的理由
   - 变量注释: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.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:

     ```
     private:
     // Used to bounds-check table accesses. -1 means
     // that we don't yet know how many entries the table has.
     int num_total_entries_;
     ```

     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

     ```
     // The total number of tests cases that we run through in this
     regression test.
     const int kNumTestCases = 6;
     ```

- 行注释:如果你需要连续进行多行注释, 可以使之对齐获得更好的可读性

```
DoSomething();                  // Comment here so the comments line up.
DoSomethingElseThatIsLonger();  // Two spaces between the code and the
comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  DoSomethingElse();  // Two spaces before line comments normally.
}
std::vector<string> list{
                    // Comments in braced lists describe the next
element...
                    "First item",
                    // .. and should be aligned appropriately.
"Second item"};
DoSomething(); /* For trailing block comments, one space is fine. */
```

- **禁止的行为**:不要描述显而易见的现象, 永远不要 用自然语言翻译代码作为注释, 除非即使对深入理解 C++ 的读者来说代码的行为都是不明显的. 要假设读代码的人 C++ 水平比你高, 即便他/她可能不知道你的用意:

  你所提供的注释应当解释代码为什么要这么做和代码的目的, 或者最好是让代码自文档化.对比一下三者代码注释,最后一份显然最好

```
// Find the element in the vector.  <-- 差: 这太明显了!
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

```
// Process "element" unless it was already processed.
auto iter = std::find(v.begin(), v.end(), element);
if (iter != v.end()) {
  Process(element);
}
```

```
if (!IsAlreadyProcessed(element)) {
   Process(element);
}
```

- TODO注释:对那些临时的, 短期的解决方案, 或已经够好但仍不完美的代码使用 `TODO` 注释

  TODO 注释要使用全大写的字符串 TODO, 在随后的圆括号里写上你的名字, 邮件地址, bug ID, 或其它身份标识和与这一 TODO 相关的 issue. 主要目的是让添加注释的人 (也是可以请求提供更多细节的人) 可根据规范的 TODO 格式进行查找. 添加 TODO 注释并不意味着你要自己来修正, 因此当你加上带有姓名的 TODO 时, 一般都是写上自己的名字.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
// TODO(bug 12345): remove the "Last visitors" feature
```

5. 格式:对于代码格式, 因人, 系统而异各有优缺点, 但同一个项目中遵循同一标准还是有必要的

- 每一行代码字符数不超过 80.
- `return` 不要加 `()`;
- `windows` 操作系统推荐使用 `Tab(4)` 制表符缩进
- 返回类型和函数名在同一行, 参数也尽量放在同一行, 如果放不下就对形参分行

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
     Type par_name1,  // 4 space indent
     Type par_name2,
     Type par_name3) {
   DoSomething();  // 2 space indent
   ...
}
```

- 大括号位置不做特殊要求,结尾或另起一行皆可.(本人推崇结尾)
- 注意所有情况下 `if` 和左圆括号间都有个空格. 右圆括号和左大括号之间也要有个空格

```
if (condition) {  // 好 - IF 和 { 都与空格紧邻.
```

如果能增强可读性, 简短的条件语句允许写在同一行. 只有当语句简单并且没有使用 `else` 子句时使用

```
if (x == kFoo) return new Foo();
if (x == kBar) return new Bar();
```

如果语句有 `else` 分支则不允许

- 通常, 单行语句不需要使用大括号, 如果你喜欢用也没问题; 复杂的条件或循环语句用大括号可读性会更好.
- 句点或箭头前后不要有空格. 指针/地址操作符 `(*, &)` 之后不能有空格.
  - 在访问成员时, 句点或箭头前后没有空格.
  - 指针操作符 * 或 & 后没有空格
  - 指针定义使用 `int *p` 而不是 `int* p` .新手会误以为后者的 `p` 是 `int *` 变量, 但前者就不一样了, 高下立判.

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

C++规范

- 0 背景

  C++ 是 Google 大部分开源项目的主要编程语言. 正如每个 C++ 程序员都知道的, C++ 有很多强大的特性, 但这种强大不可避免的导致它走向复杂,使代码更容易产生 bug, 难以阅读和维护.

  本指南的目的是通过详细阐述 C++ 注意事项来驾驭其复杂性. 这些规则在保证代码易于管理的同时, 也能高效使用 C++ 的语言特性.

  风格, 亦被称作可读性, 也就是指导 C++ 编程的约定. 使用术语 “风格” 有些用词不当, 因为这些习惯远不止源代码文件格式化这么简单.

使代码易于管理的方法之一是加强代码一致性. 让任何程序员都可以快速读懂你的代码这点非常重要. 保持统一编程风格并遵守约定意味着可以很容易根据 “模式匹配” 规则来推断各种标识符的含义. 创建通用, 必需的习惯用语和模式可以使代码更容易理解. 在一些情况下可能有充分的理由改变某些编程风格, 但我们还是应该遵循一致性原则,尽量不这么做.

本指南的另一个观点是 C++ 特性的臃肿. C++ 是一门包含大量高级特性的庞大语言. 某些情况下, 我们会限制甚至禁止使用某些特性. 这么做是为了保持代码清爽, 避免这些特性可能导致的各种问题. 指南中列举了这类特性, 并解释为什么这些特性被限制使用.

Google 主导的开源项目均符合本指南的规定.

注意: 本指南并非 C++ 教程, 我们假定读者已经对 C++ 非常熟悉.

- 1.头文件
  - 除单元测试代码和主程序入口,每一个`.cpp`文件都需要对应一个`.h`文件
  - 为了保护唯一性,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`foo`中的头文件`foo/src/bar/baz.h`可按如下方式保护

    ```
    #ifndef FOO_BAR_BAZ_H_
    #define FOO_BAR_BAZ_H_
    ...
    #endif // FOO_BAR_BAZ_H_
    ```

  - 尽可能避免使用前置声明,使用`#include`包含所需的头文件,详见1.3前置声明
  - 项目内头文件应按照源代码目录树结构排列,不应使用`.`(当前目录),`..`(上级目录)
  - 如`dir/foo_test.c`的作用是实现或测试`dir/foo.h`的功能,其头文件引用次序应该为
    - `dir/foo.h`(优先位置)
    - c系统文件
    - c++系统文件
    - 其他库`.h`文件
    - 本项目内`.h`文件
- 2.作用域

  鼓励使用匿名命名空间或`static`命名.使用具名的命名空间时, 其名称可基于项目名或相对路径. 禁止使用`using`指示禁止使用内联命名空间(inline namespace).
  - 不要在`.h`中定义不需要被外部引用的变量
  - 使用静态成员函数或命名空间内的非成员函数, 尽量不要用裸的全局函数. 将一系列函数直接置于命名空间中,不要用类的静态方法模拟出命名空间的效果,类的静态方法应当和类的实例或静态数据紧密相关.应使用

    ```
    namespace myproject {
    namespace foo_bar {
    void Function1();
    void Function2();
    }  // namespace foo_bar
    }  // namespace myproject
    ```

    而非

```
namespace myproject {
class FooBar {
 public:
  static void Function1();
  static void Function2();
};
}  // namespace myproject
```

- 函数变量应置于最小的作用域内,并在变量声明时进行初始化

```
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

有一个例外, 如果变量是一个对象, 每次进入作用域都要调用其构造函数, 每次退出作用域都要调用其析构函数. 这会导致效率降低.

```
// 低效的实现
for (int i = 0; i < 1000000; ++i) {
Foo f;                  // 构造函数和析构函数分别调用 1000000 次!
f.DoSomething(i);
}
```

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);
}
```

- 3.类
  - 不要在构造函数中调用虚函数,构造函数中可以进行各种初始化操作,不在构造函数中做太多逻辑相关的初始化
  - 不要定义隐式类型转换. 对于转换运算符和单参数构造函数, 请使用`explicit`关键字
  - 仅当只有数据成员时使用`struct`, 其它一概使用`class`.
  - 将相似的声明放在一起, 将`public`部分放在最前.类定义一般应以`public:`开始, 后跟`protected:`, 最后是`private:`. 省略空部分.
- 4.函数
  - 我们倾向于按值返回,否则按引用返回,避免返回指针,除非它可以为空.

- 我们倾向于编写简短, 凝练的函数.我们承认长函数有时是合理的, 因此并不硬性限制函数的长度. 如果函数超过**40**行, 可以思索一下能不能在不影响程序结构的前提下对其进行分割.
  即使一个长函数现在工作的非常好, 一旦有人对其修改, 有可能出现新的问题, 甚至导致难以发现的 bug. 使函数尽量简短, 以便于他人阅读和修改代码.
- 所有按引用传递的参数必须加上`const`.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).`

  `Google Code`中有一个硬性约定: 输入参数是值参或 const 引用, 输出参数为指针.

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

  `const &`指针有很多好处,比如避免值传递时的复制/调用构造函数,并且保证参数不被修改
- 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.
- 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.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()`
- 只有在常规写法 (返回类型前置) 不便于书写或不便于阅读时使用返回类型后置语法.

- 5.C++特性
  - 所有按引用传递的参数必须加上`const.`
  - 我们不允许使用缺省函数参数,少数极端情况除外。尽可能改用函数重载.
  - 我们不使用 C++ 异常(Google要求,但个人使用或团队项目还是推荐使用异常处理或异常捕获).

    异常处理的态度,值得一读
  - 使用 C++ 的类型转换, 如`static_cast<>()`. 不要使用`int y = (int)x`或`int y = int(x)`等转换方式;
  - 对于迭代器和其他模板对象使用前缀形式 (++i) 的自增, 自减运算符.

    ```
    不考虑返回值的话, 前置自增 (++i) 通常要比后置自增 (i++) 效率更高.
    因为后置自增 (或自减) 需要对表达式的值 i 进行一次拷贝.
    如果 i 是迭代器或其他非数值类型, 拷贝的代价是比较大的.
    ```
  - 我们强烈建议你在任何可能的情况下都要使用 const. 此外有时改用 C++11 推出的 constexpr 更好.在 C++11 里,用 constexpr 来定义真正的常量,或实现常量初始化。
  - 使用宏时要非常谨慎, 尽量以内联函数, 枚举和常量代替之.

最后特别鸣谢该网站的翻译作者,谢谢