6. 命名约定

6.1. 通用命名规则

驼峰风格(CamelCase) 大小写字母混用，单词连在一起，不同单词间通过单词首字母大写来分开。按连接后的首字母是否大写，又分：大驼峰(**UpperCamelCase**)和小驼峰(**lowerCamelCase**)。

类型	命名风格
类类型，结构体类型，枚举类型，联合体类型等类型定义， 作用域名称	大驼峰
函数(包括全局函数，作用域函数，成员函数)	小驼峰
全局变量(包括全局和命名空间域下的变量，类静态变量)，局部变量，函数参数，结构体和联合体中的成员变量	小驼峰
宏，常量(const)，枚举值，goto 标签	全大写，下划线分割
类成员变量	m 开头+大驼峰

注意：上表中常量是指全局作用域、namespace 域、类的静态成员域下，以 const 或 constexpr 修饰的基本数据类型、枚举、字符串类型的变量，不包括数组和其他类型变量。上表中变量是指除常量定义以外的其他变量，均使用小驼峰风格。

6.2. 文件命名

建议 6.2.1. C++文件以.cpp 结尾，头文件以.h 结尾

我们推荐使用.h 作为头文件的后缀，这样头文件可以直接兼容 C 和 C++。 我们推荐使用.cpp 作为实现文件的后缀，这样可以直接区分 C++代码，而不是 C 代码。

目前业界还有一些其他的后缀的表示方法：

头文件： .hh, .hpp, .hxx

cpp 文件：.cc, .cxx, .c

如果当前项目组使用了某种特定的后缀，那么可以继续使用，但是请保持风格统一。 但是对于本文档，我们默认使用.h 和.cpp 作为后缀。

建议 6.2.2. C++文件名和类名保持一致

C++的头文件和 cpp 文件名和类名保持一致，使用下划线小写风格。

如果有一个类叫 DatabaseConnection，那么对应的文件名：

- database_connection.h
- database_connection.cpp

结构体，命名空间，枚举等定义的文件名类似。

在目录的命名上，选用大驼峰的风格去命名，比如后处理设备

```
|--PostProcess
   |--post_process.h
   |--post_process.cpp
```

6.3. 类型命名

类型命名采用大驼峰命名风格。

所有类型命名——类、结构体、联合体、类型定义（typedef）、枚举一一使用相同约定，例如：

```
// classes, structs and unions
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...
union Packet { ...
// typedefs
typedef std::map<std::string, UrlTableProperties*> PropertiesMap;
// enums
enum UrlTableErrors { ...
```

建议 6.3.1. 避免滥用 typedef 或者#define 对基本类型起别名

除有明确的必要性，否则不要用 typedef/#define 对基本数据类型进行重定义。优先使用<cstdint>头文件中的基本类型。

有符号类型	无符号类型	描述
int8_t	uint8_t	宽度恰为 8 的有/无符号整数类型
int16_t	uint16_t	宽度恰为 16 的有/无符号整数类型
int32_t	uint32_t	宽度恰为 32 的有/无符号整数类型
int64_t	uint64_t	宽度恰为 64 的有/无符号整数类型
intptr_t	uintptr_t	足以保存指针的有/无符号整数类型

6.4. 变量命名

通用变量命名采用小驼峰，包括全局变量、函数形参、局部变量。

```
std::string tableName;  // Good: 推荐此风格
```

```
std::string tablename;  // Bad: 禁止此风格
std::string path;       // Good: 只有一个单词时，小驼峰为全小写
```

规则 6.4.1. 全局变量应增加‘g’前缀，静态变量命名不需要加特殊前缀

全局变量是应当尽量少使用的，使用时应特别注意，所以加上前缀用于视觉上的突出，促使开发人员对这些变量的使用更加小心。

- 全局静态变量命名与全局变量相同；
- 函数内的静态变量命名与普通局部变量相同；
- 类的静态成员变量和普通成员变量相同；

```
int gActiveConnectCount;
void func()
{
    static int packetCount = 0;
    ...
}
```

规则 6.4.2. 类的成员变量命名以”m”加大驼峰组成

```
class Foo {
private:
    std::string mFileName;   // 添加 m 前缀
};
```

对于 struct/union 的成员变量，仍采用小驼峰不加后缀的命名方式，与局部变量命名风格一致。

6.5. 常量命名

全局作用域内，有名和匿名 namespace 内的 const 常量，类的静态成员常量，全大写，下划线连接；函数局部 const 常量和类的普通 const 成员变量，使用小驼峰命名风格。

```
int func(...)
{
    const unsigned int bufferSize = 100;    // 函数局部常量
    char *p = new char[bufferSize];
    ...
}
namespace Utils
{
```

```
9.      const unsigned int DEFAULT_FILE_SIZE_KB = 200;  // 全局常量
10. } // Utils
```

6.6. 函数命名

函数命名统一使用小驼峰风格，一般采用动词或者动宾结构。

```
1.  class List
2.  {
3.  public:
4.      void addElement(const Element& element);
5.      Element getElement(const unsigned int index) const;
6.      bool isEmpty() const;
7.  };
8.  namespace Utils
9.  {
10.     void deleteUser();
11. } // Utils
```

6.7. 命名空间命名

对于命名空间的命名，建议使用大驼峰：

```
1.  // namespace
2.  namespace OsUtils
3.  {
4.  namespace FileUtils
5.  {
6.  } // FileUtils
7.  } // OsUtils
```

6.8. 枚举命名 && 枚举命名

宏、枚举值采用全大写，下划线连接的格式。

```
1.  #define MAX(a, b)   (((a) < (b)) ? (b) : (a)) // 仅对宏命名举例，并不推荐用宏实现此类功能
2.   // 注意，枚举类型名用大驼峰，其下面的取值是全大写，下划线相连
3.  enum TintColor
4.  {
5.      RED,
6.      DARK_RED,
```

```
    GREEN,
    LIGHT_GREEN
};
```

6.9. 命名规则特例

建议 6.9.1. Qt Creator 中 ui 文件拖拽控件的命名为控件类型加下划线加小驼峰

在 Qt Creator 这个 IDE 中通过拖拽来设计 ui 界面时，IDE 会自动生成对应控件的名称。一般情况下，为了方便，在生成的命名后面加下划线再加小驼峰来命名。

```
ui->comboBox_deviceList;
ui->pushButton_ppTaskDeliver;
ui->label_lldDefaultMode;
ui->lineEdit_lightEffect;
ui->checkBox_applyRemap;
ui->pushButton_ppTaskDeliver;
```

7. 格式

7.1. 行长度

建议 7.1.1. 行长度不要超过 120 个字符

建议每行字符数不要超过 120 个，如果超过 120 个字符，请选择合理的方式进行换行。例外:

- 如果一行注释包含了超过 120 个字符的命令或 URL，则可以保持一行，以方便复制、粘贴和通过 grep 查找；
- 包含长路径的 #include 语句可以超出 120 个字符，但是也需要尽量避免；
- 编译预处理中的 error 信息可以超出一行。预处理的 error 信息在一行便于阅读和理解，即使超过 120 个字符。

```
#ifndef XXX_YYY_ZZZ
#error Header aaaa/bbbb/cccc/abc.h must only be included after xxxx/yyyy/zzzz/xyz.h, because xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
#endif
```

7.2. 缩进

规则 7.2.1. 使用空格进行缩进，每次缩进 4 个空格

只允许使用空格(space)进行缩进，每次缩进为 4 个空格。不允许使用 Tab 符进行缩进。当前几乎所有的集成开发环境（IDE）都支持配置将 Tab 符自动扩展为 4 空格输入；请配置你的 IDE 支持使用空格进行缩进。

7.3. 函数声明与定义

规则 7.3.1. 函数声明和定义的返回类型和函数名在同一行；函数参数列表超出行宽时要换行并合理对齐

在声明和定义函数的时候，函数的返回值类型应该和函数名在同一行；如果行宽度允许，函数参数也应该放在一行；否则，函数参数应该换行，并进行合理对齐。参数列表的左圆括号总是和函数名在同一行，不要单独一行；右圆括号总是跟随最后一个参数。

换行举例：

```
// Good：全在同一行
ReturnType functionName(ArgType paramName1, ArgType paramName2)
{
    ...
}

// 行宽不满足所有参数，进行换行
// Good：和上一行参数对齐
ReturnType veryVeryVeryLongFunctionName(ArgType paramName1,
                                        ArgType paramName2,
                                        ArgType paramName3)
{
    ...
}

// 行宽限制，进行换行
// Good: 换行后 4 空格缩进
ReturnType longFunctionName(ArgType paramName1,
    ArgType paramName2,  ArgType paramName3)
{
    ...
}

// 行宽不满足第 1 个参数，直接换行
// Good: 换行后 4 空格缩进
ReturnType reallyReallyReallyReallyLongFunctionName(
```

```
    ArgType paramName1, ArgType paramName2, ArgType paramName3)
{
    ...
}

// 返回类型和函数名在一行放不下，直接换行
// Good: 函数名不需要缩进
ReturnType
reallyReallyReallyReallyReallyLongFunctionName(
    ArgType paramName1, ArgType paramName2, ArgType paramName3)
{
    ...
}
```

注意以下几点:

- 只有在参数未被使用或者其用途非常明显时，才能省略参数名；
- 如果返回类型和函数名在一行放不下，分行；
- 如果返回类型与函数声明或定义分行了，不要缩进；
- 左圆括号总是和函数名在同一行；
- 函数名和左圆括号间永远没有空格；
- 圆括号与参数间没有空格；
- 左大括号总在最后一个参数的单独新一行；
- 右大括号总是单独位于函数最后一行，或者与左大括号同一行；
- 所有形参应尽可能对齐；
- 缺省缩进为 4 个空格；
- 换行后的参数保持 4 个空格的缩进；

未被使用的参数，或者根据上下文很容易看出其用途的参数，可以省略参数名:

```
class Foo {
public:
    Foo(Foo&&);
    Foo(const Foo&);
    Foo& operator=(Foo&&);
    Foo& operator=(const Foo&);
};
```

未被使用的参数如果其用途不明显的话，在函数定义处将参数名注释起来:

```
class Shape
{
public:
    virtual void rotate(double radians) = 0;
};

class Circle : public Shape
```

```
{
public:
    void rotate(double radians) override;
};

// Good: 直接将变量名注释
void Circle::rotate(double /*radians*/)
{}

// Bad: 如果将来有人要实现，很难猜出变量的作用.
void Circle::rotate(double)
{}
```

7.4. 函数调用

规则 7.4.1. 函数调用入参列表应放在一行，超出行宽换行时，保持参数进行合理对齐

函数调用时，函数参数列表放在一行。参数列表如果超过行宽，需要换行并进行合理的参数对齐。 左圆括号总是跟函数名，右圆括号总是跟最后一个参数。

换行举例：

```
// Good: 函数参数放在一行
ReturnType result = functionName(paramName1, paramName2);

// Good: 保持与上方参数对齐
ReturnType result = functionName(paramName1,
                                 paramName2,
                                 paramName3);

// Good: 参数换行，4 空格缩进
ReturnType result = functionName(paramName1, paramName2,
    paramName3, paramName4, paramName5);

// 行宽不满足第 1 个参数，直接换行
ReturnType result = veryVeryVeryLongFunctionName(
    paramName1, paramName2, paramName3);  // 换行后，4 空格缩进
```

如果函数调用的参数存在内在关联性，按照可理解性优先于格式排版要求，对参数进行合理分组换行。

```
// Good：每行的参数代表一组相关性较强的数据结构，放在一行便于理解
my_widget.Transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

7.5. lambda 表达式

- lambda 表达式对形参和函数体的格式化和其他函数一致；
- 捕获列表同理，表项用逗号隔开,若用引用捕获，在变量名和 & 之间不留空格；
- **必须去完成 lambda 表达式的返回值类型声明**

```
std::set<int> blacklist = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(),
    [&blacklist](int i)->bool {
        return blacklist.find(i) != blacklist.end();
    }),
digits.end());
```

7.6. 条件语句

规则 7.6.1. if 语句必须要使用大括号

我们要求 if 语句都需要使用大括号，即便只有一条语句。
理由：

- 代码逻辑直观，易读；
- 在已有条件语句代码上增加新代码时不容易出错；
- 对于在 if 语句中使用函数式宏时，有大括号保护不易出错（如果宏定义时遗漏了大括号）。

```
if (objectIsNotExist) // Good：单行条件语句也加大括号
{
    return createNewObject();
}
```

规则 7.6.2. 禁止 if/else/else if 写在同一行

条件语句中，若有多个分支，应该写在不同行。
如下是正确的写法：

```
if (someConditions)
{
    doSomething();
```

```
    ...
}
else // Good: else 与 if 在不同行
{
    ...
}
```

下面是不符合规范的案例：

```
if (someConditions) { ... } else { ... } // Bad: else 与 if 在同一行
```

7.7. 循环和 switch

规则 7.7.1. 循环语句必须使用大括号

和条件表达式类似，我们要求 for/while 循环语句必须加上大括号，即便循环体是空的，或循环语句只有一条。

```
for (int i = 0; i < someRange; i++)  // Good: 使用了大括号
{
    doSomething();
}

// Good: 循环体是空，使用大括号
while (condition) { }

// Good: continue 表示空逻辑，使用大括号
while (condition)
{
    continue;
}

// Bad: 应该加上括号
for (int i = 0; i < someRange; i++)
    doSomething();

// Bad: 使用分号容易让人误解是 while 语句中的一部分
while (condition);
```

规则 7.7.2 switch 语句的 case/default 要缩进一层

switch 语句的缩进风格如下：

- Case 条件内的代码段，必须实现在大括号内；
- 代码段的 break，写在大括号内；
- Case 的代码段超过 10 行时，建议使用函数代替；
- 必须包含 default 条件

```
switch (var)
{
    case 0:                 // Good: 缩进
        doSomething1(); // Good: 缩进
        break;
    case 1:
    {           // Good: 带大括号格式
        doSomething2();
        break;
    }
    default:
        break;
}

switch (var)
{
case 0:                 // Bad: case 未缩进
    doSomething();
    break;
default:                // Bad: default 未缩进
    break;
}
```

7.8. 指针和引用表达式

建议 7.8.1. 指针类型“*”跟随变量名或者类型，不要两边都留有或者都没有空格

指针命名:*靠左靠右都可以，但是不要两边都有或者都没有空格。

```
int* p = NULL;  // Good
int *p = NULL;  // Good
int*p = NULL;   // Bad
int * p = NULL; // Bad
```

例外：当变量被 const 修饰时，”*” 无法跟随变量，此时也不要跟随类型。

```
const char * const VERSION = "V100";
```

建议 7.8.2. 引用类型“&”跟随变量名或者类型，不要两边都留有或者都没有空格

引用命名：&靠左靠右都可以，但是不要两边都有或者都没有空格。

```
int i = 8;
int& p = i;      // Good
int &p = i;      // Good
int*& rp = pi;   // Good，指针的引用，*& 一起跟随类型
int *&rp = pi;   // Good，指针的引用，*& 一起跟随变量名
int* &rp = pi;   // Good，指针的引用，* 跟随类型，& 跟随变量名
int & p = i;     // Bad
int&p = i;       // Bad
```

7.9. Bool 表达式

建议 7.9.1. 表达式换行要保持换行的一致性，运算符放行末

较长的表达式，不满足行宽要求的时候，需要在适当的地方换行。一般在较低优先级运算符或连接符后面截断，运算符或连接符放在行末。**运算符、连接符放在行末，表示“未结束，后续还有”**。 例：

// 假设下面第一行已经不满足行宽要求

```
if (currentValue > threshold &&  // Good：换行后，逻辑操作符放在行尾
    someConditionsion)
{
    doSomething();
    ...
}

int result = reallyReallyLongVariableName1 +    // Good
             reallyReallyLongVariableName2;
```

表达式换行后，注意保持合理对齐，或者 4 空格缩进。参考下面例子:

```
// Good: 4 空格缩进
int sum = longVaribleName1 + longVaribleName2 + longVaribleName3 +
    longVaribleName4 + longVaribleName5 + longVaribleName6;

// Good: 保持对齐
int sum = longVaribleName1 + longVaribleName2 + longVaribleName3 +
          longVaribleName4 + longVaribleName5 + longVaribleName6;
```

7.10. 变量及数组初始化

规则 7.10.1. 多个变量定义和赋值语句不允许写在一行

每行只有一个变量初始化的语句，更容易阅读和理解。

```
int maxCount = 10;
bool isCompleted = false;
```

下面是不符合规范的示例：

```
// Bad：多个变量初始化需要分开放在多行，每行一个变量初始化
int maxCount = 10; bool isCompleted = false;
int x, y = 0;  // Bad：多个变量定义需要分行，每行一个
int pointX;
int pointY;
...
pointX = 1; pointY = 2;  // Bad：多个变量赋值语句放同一行
```

例外：for 循环头、if 初始化语句（C++17）、结构化绑定语句（C++17）中可以声明和初始化多个变量。这些语句中的多个变量声明有较强关联，如果强行分成多行会带来作用域不一致，声明和初始化割裂等问题。

规则 7.10.2. 初始化换行时要有缩进，并进行合理对齐

结构体或数组初始化时，如果换行应保持 4 空格缩进。从可读性角度出发，选择换行点和对齐位置。

```
const int rank[] = {
    16, 16, 16, 16, 32, 32, 32, 32,
    64, 64, 64, 64, 32, 32, 32, 32
};
```

7.11. 函数返回值

- 不要在 return 表达式里加上非必须的圆括号。
- 只有在写 x = expr 要加上括号的时候才在 return expr; 里使用括号。
- **返回空数据时，尽量不要用 return {}; 这种方式，不好理解。**

```
return result; // 返回值很简单，没有圆括号。
// 可以用圆括号把复杂表达式圈起来，改善可读性。
return (some_long_condition && another_condition);
return (value); // 你应该不会写这种代码吧 var = (value);
```

7.12. 预处理命令

规则 7.12.1. 编译预处理的”#”统一放在行首，嵌套编译预处理语句时，”#”可以进行缩进

编译预处理的”#”统一放在行首，即使编译预处理的代码是嵌入在函数体中的，”#”也应该放在行首。

```
#if defined(__x86_64__) && defined(__GCC_HAVE_SYNC_COMPARE_AND_SWAP_16) // Good：”#”放在行首
#define ATOMIC_X86_HAS_CMPXCHG16B 1  // Good：”#”放在行首
#else
#define ATOMIC_X86_HAS_CMPXCHG16B 0
#endif
int FunctionName()
{
    if (someThingError)
    {
        ...
#ifdef HAS_SYSLOG        // Good：即便在函数内部，”#”也放在行首
        writeToSysLog();
#else
        writeToFileLog();
#endif
    }
}
```

内嵌的预处理语句”#”可以按照缩进要求进行缩进对齐，区分层次。

```
#if defined(__x86_64__) && defined(__GCC_HAVE_SYNC_COMPARE_AND_SWAP_16)
    #define ATOMIC_X86_HAS_CMPXCHG16B 1  // Good：区分层次，便于阅读
#else
    #define ATOMIC_X86_HAS_CMPXCHG16B 0
#endif
```

下面是不合适的缩进：

```
// 差 - 指令缩进
if (lopsided_score)
{
    #if DISASTER_PENDING // 差 - "#if" 应该放在行开头
    dropEverything();
    #endif // 差 - "#endif" 不要缩进
```

```
7.      backToNormal();
8.  }
```

7.13. 类格式

规则 7.13.1. 类访问控制块的声明依次序是 public, protected, private，缩进和 class 关键字对齐

```
1.  class MyClass : public BaseClass
2.  {
3.  public:          // 注意没有缩进
4.      MyClass();  // 标准的4 空格缩进
5.      explicit MyClass(int var);
6.      ~MyClass() {}
7.      void someFunction();
8.      void someFunctionThatDoesNothing()
9.      {
10.     }
11.     void setVar(int var) { mSomeVar = var; }
12.     int getVar() const { return mSomeVar; }
13.
14. private:
15.     bool someInternalFunction();
16.     int mSomeVar;
17.     int mSomeOtherVar;
18. };
```

注意事项:

- 所有基类名应在 120 列限制下尽量与子类名放在同一行；
- 关键词 public:, protected:, private: 要与 class 对齐；
- 除第一个关键词 (一般是 public) 外，其他关键词前要空一行，如果类比较小的话也可以不空；
- 这些关键词后不要保留空行；
- public 放在最前面，然后是 protected，最后是 private。
- 关于声明顺序的规则请参考以下：

在各个部分中，建议将类似的声明放在一起，并且建议以如下的顺序：类型（包括 typedef, using 和嵌套的结构体与类），常量，工厂函数，构造函数，赋值运算符，析构函数，其它成员函数，数据成员。

规则 7.13.2 构造函数初始化列表放在同一行或按四格缩进并排多行

```
// 如果所有变量能放在同一行:
MyClass::MyClass(int var) : mSomeVar(var)
{
    doSomething();
}

// 如果不能放在同一行,
// 必须置于冒号后, 并缩进 4 个空格
MyClass::MyClass(int var)
    : mSomeVar(var), mSomeOtherVar(var + 1)  // Good: 逗号后面留有空格
{
    doSomething();
}

// 如果初始化列表需要置于多行, 需要逐行对齐
MyClass::MyClass(int var)
    : mSomeVar(var),              // 缩进 4 个空格
      mSomeOtherVar(var + 1)
{
    doSomething();
}
```

7.14. 命名空间格式化

命名空间不要增加额外的缩进层次, 例如:

```
namespace
{
void foo()  // 正确. 命名空间内没有额外的缩进.
{
    ...
}
} // namespace
```

不要在命名空间内缩进:

```
namespace
{
    // 错, 缩进多余了.
```

```
    void foo()
    {
        ...
    }
} // namespace
```

声明嵌套命名空间时，每个命名空间都独立成行.

```
namespace foo {
namespace bar {
```

7.15. 空格与空行

规则 7.15.1 水平空格应该突出关键字和重要信息，避免不必要的留白

水平空格应该突出关键字和重要信息，每行代码尾部不要加空格。总体规则如下：

- 小括号内部的两侧，不要加空格；
- 大括号内部两侧有无空格，左右必须保持一致；
- 一元操作符（& * + - ~ !）之后不要加空格；
- 二元操作符（= + - < > * / % | & ^ <= >= == != ）左右两侧加空格；
- 三目运算符（? :）符号两侧均需要空格；
- 前置和后置的自增、自减（++ —）和变量之间不加空格；
- 结构体成员操作符（. ->）前后不加空格；
- 逗号(,)前面不加空格，后面增加空格；
- 对于模板和类型转换(<>)和类型之间不要添加空格；
- 域操作符(::)前后不要添加空格；
- 冒号(:)前后根据情况来判断是否要添加空格；

常规情况

```
int i = 0;  // Good：变量初始化时，=前后应该有空格，分号前面不要留空格
int buf[BUF_SIZE] = {0};    // Good：大括号内两侧都无空格
```

函数定义和函数调用

```
int result = foo(arg1,arg2);
                    ^    // Bad: 逗号后面需要增加空格
int result = foo( arg1, arg2 );
                  ^         ^  // Bad: 函数参数列表的左括号后面不应该有空格，右括号前面不应该有空格
```

指针和取地址

```
x = *p;     // Good：*操作符和指针 p 之间不加空格
```

```
p = &x;      // Good：&操作符和变量x之间不加空格
x = r.y;     // Good：通过.访问成员变量时不加空格
x = r->y;    // Good：通过->访问成员变量时不加空格
```

操作符：

```
x = 0;     // Good：赋值操作的=前后都要加空格
x = -5;    // Good：负数的符号和数值之前不要加空格
++x;       // Good：前置和后置的++/--和变量之间不要加空格
x--;
if (x && !y)  // Good：布尔操作符前后要加上空格，！操作和变量之间不要空格
v = (w * x) + (y / z); // Good：加括号去突出运算顺序，方便理解
v = w * (x + z);     // Good：括号内的表达式前后不需要加空格
int a = (x < y) ? x : y;  // Good：三目运算符，  ？和：前后需要添加空格
```

循环和条件语句

```
if (condition)
{
    ...
}
else
{
    ...
}
while (condition) {}   // Good：while关键字和括号之间加空格，括号内条件语句前后不加空格
for (int i = 0; i < someRange; ++i) // Good：for关键字和括号之间加空格，分号之后加空格
{
    ...
}
switch (condition) // Good：switch 关键字后面有1空格
{
    case 0:     // Good：case 语句条件和冒号之间不加空格
        ...
        break;
    ...
    default:
        ...
        break;
}
```

模板和转换，**尖括号内的”*”和”&”统一贴在类型名称的右边，中间不带空格**。

```
// 尖括号(< and >) 不与空格紧邻, < 前没有空格, > 和 ( 之间也没有.
vector<string> x;
y = static_cast<char*>(x);
// 在类型与指针操作符之间留空格也可以, 但要保持一致.
vector<char*> x;
```

域操作符

```
std::cout;    // Good: 命名空间访问，不要留空格
int MyClass::getValue() const {}  // Good: 对于成员函数定义，不要留空格
```

冒号

```
// 添加空格的场景
// Good: 类的派生需要留有空格
class Sub : public Base
{
};
// 构造函数初始化列表需要留有空格
MyClass::MyClass(int var) : mSomeVar(var)
{
    doSomething();
}
// 位域表示也留有空格，冒号对齐
struct XX {
    char defaultSector       : 4;
    char defaultSize         : 5;
    char defaultLongLongName : 4;
};
```

```
// 不添加空格的场景
// Good: 对于public:, private:这种类访问权限的冒号不用添加空格
class MyClass
{
public:
    MyClass(int var);
private:
    int mSomeVar;
};

// 对于switch-case 的case 和default 后面的冒号不用添加空格
switch (value)
{
    case 1:
        doSomething();
```

```
16.         break;
17.     default:
18.         break;
19. }
```

注意：当前的 IDE 可以设置删除行尾的空格，请正确配置。

建议 7.15.2 合理安排空行，保持代码紧凑

减少不必要的空行，可以显示更多的代码，方便代码阅读。下面有一些建议遵守的规则：

- 根据上下内容的相关程度，合理安排空行；
- 函数内部、类型定义内部、宏内部、初始化表达式内部，不使用连续空行
- 不使用连续 3 个空行，或更多
- 大括号内的代码块行首之前和行尾之后不要加空行，但 namespace 的大括号内不作要求。

```
1.  int foo()
2.  {
3.      ...
4.  }
5.
6.
7.  int bar()  // Bad：最多使用连续 2 个空行。
8.  {
9.      ...
10. }
11.
12. if (...)
13. {
14.         // Bad：大括号内的代码块行首不要加入空行
15.     ...
16.         // Bad：大括号内的代码块行尾不要加入空行
17. }
18.
19. int foo(...)
20. {
21.         // Bad：函数体内行首不要加空行
22.     ...
23. }
```

8. 注释

一般的，尽量通过清晰的架构逻辑，好的符号命名来提高代码可读性；需要的时候，才

辅以注释说明。注释是为了帮助阅读者快速读懂代码，所以要从读者的角度出发，按需注释。

注释内容要简洁、明了、无二义性，信息全面且不冗余。

注释跟代码一样重要。

写注释时要换位思考，用注释去表达此时读者真正需要的信息。在代码的功能、意图层次上进行注释，即注释解释代码难以表达的意图，不要重复代码信息。

修改代码时，也要保证其相关注释的一致性。只改代码，不改注释是一种不文明行为，破坏了代码与注释的一致性，让阅读者迷惑、费解，甚至误解。

使用**你觉得舒服的语言**进行注释。

8.1. 注释风格

在 C++ 代码中，使用 /**/和 // 都是可以的。

按注释的目的和位置，注释可分为不同的类型，如文件头注释、函数头注释、代码注释等等；

同一类型的注释应该保持统一的风格。

注意：本文示例代码中，大量使用 '//' 后置注释只是为了更精确的描述问题，并不代表这种注释风格更好。

8.2. 文件注释

规则 8.2.1. 文件头注释必须包含版权许可

这个东西，以后补充

8.3. 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。

```
// Iterates over the contents of a GargantuanTable.
// Example:
//     GargantuanTableIterator* iter = table->NewIterator();
//     for (iter->Seek("foo"); !iter->done(); iter->Next())
//     {
//         process(iter->key(), iter->value());
//     }
// delete iter;
class GargantuanTableIterator
{
...
};
```

类注释应当为读者理解如何使用与何时使用类提供足够的信息，同时应当提醒读者在正确使用此类时应当考虑的因素。如果类有任何同步前提，请用文档说明。如果该类的实例可被多线程访问，要特别注意文档说明多线程环境下相关的规则和常量使用。

如果你想用一小段代码演示这个类的基本用法或通常用法，放在类注释里也非常合适。

如果类的声明和定义分开了（例如分别放在了 .h 和 .cc 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。

8.4. 函数注释

规则 4.3.1 禁止空有格式的函数头注释

并不是所有的函数都需要函数头注释；函数签名无法表达的信息，加函数头注释辅助说明。

函数头注释统一放在函数声明或定义上方，使用如下风格之一：

使用//写函数头

```
// 单行函数头
int func1(void);
// 多行函数头
// 第二行
int func2(void);
```

使用/* */写函数头

```
/* 单行函数头 */
int func1(void);
/*
 * 另一种单行函数头
 */
int func2(void);
/*
 * 多行函数头
 * 第二行
 */
int func3(void);
```

函数尽量通过函数名自注释，按需写函数头注释。不要写无用、信息冗余的函数头；不要写空有格式的函数头。

函数头注释内容可选，但不限于：功能说明、返回值，性能约束、用法、内存约定、算法实现、可重入的要求等等。模块对外头文件中的函数接口声明，其函数头注释，应当将重要、有用的信息表达清楚。

例：

```
/*
 * 返回实际写入的字节数，-1 表示写入失败
```

```
3.   * 注意，内存 buf 由调用者负责释放
4.   */
5.  int writeString(const char *buf, int len);
```

坏的例子：

```
1.  /*
2.   * 函数名：writeString
3.   * 功能：写入字符串
4.   * 参数：
5.   * 返回值：
6.   */
7.  int writeString(const char *buf, int len);
```

上面例子中的问题：

- 参数、返回值，空有格式没内容
- 函数名信息冗余
- 关键的 buf 由谁释放没有说清楚

如果函数参数的意义不明显，考虑用下面的方式进行弥补:

- 如果参数是一个字面常量，并且这一常量在多处函数调用中被使用，用以推断它们一致，你应当用一个常量名让这一约定变得更明显，并且保证这一约定不会被打破；
- 考虑更改函数的签名，让某个 bool 类型的参数变为 enum 类型，这样可以让这个参数的值表达其意义；
- 如果某个函数有多个配置选项，你可以考虑定义一个类或结构体以保存所有的选项，并传入类或结构体的实例。这样的方法有许多优点，例如这样的选项可以在调用处用变量名引用，这样就能清晰地表明其意义。同时也减少了函数参数的数量，使得函数调用更易读也易写。除此之外，以这样的方式，如果你使用其他的选项，就无需对调用点进行更改；
- 用具名变量代替大段而复杂的嵌套表达式；
- 万不得已时，才考虑在调用点用注释阐明参数的意义。

我们可以比较这两个例子：

```
1.  // 变量代表着啥。。。
2.  const DecimalNumber product =
3.      calculateProduct(values, 7, false, nullptr);
```

```
1.  ProductOptions options;
2.  options.setPrecisionDecimals(7);
3.  options.setUseCache(ProductOptions::kDontUseCache);
4.  const DecimalNumber product =
5.      calculateProduct(values, options, /*completion_callback=*/nu
    llptr);
```

8.5. 变量注释

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：

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

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。比如：

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

还有一些数值上的有系数或者倍数的变量，需要去注明系数的大小，避免使用时被忽略系数，导致差值很大。

8.6. 实现注释

对于代码中巧妙的，晦涩的，有趣的，重要的地方加以注释，免得你也不记得了。

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++)
{
    x = (x << 8) + (*result)[i];
    (*result)[i] = x >> 1;
    x &= 1;
}
```

比较隐晦的地方要在行尾加入注释，在行尾空两格进行注释。比如：

```
// If we have enough memory, mmap the data portion too.
mmapBudget = max<int64>(0, mmapBudget - mIndex->length());
if (mmapBudget >= mDataSize &&
    !mmapData(mmapChunkBytes, mLock))
{
    return; // Error already logged.
}
```

注意，这里用了两段注释分别描述这段代码的作用，和提示函数返回时错误已经被记入日志。

8.7. 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。

8.8. 弃用注释

规则 8.8.1 不用的代码段直接删除，不要注释掉

被注释掉的代码，无法被正常维护；当企图恢复使用这段代码时，极有可能引入易被忽略的缺陷。正确的做法是，不需要的代码直接删除掉。若再需要时，考虑移植或重写这段代码。

这里说的注释掉代码，包括用 // 和 //，还包括 #if 0， #ifdef NEVER_DEFINED 等等。