0 前言

目的

规则并不是完美的，通过禁止在特定情况下有用的特性，可能会对代码实现造成影响。但是我们制定规则的目的__“为了大多数程序员可以得到更多的好处”__， 如果在团队运作中认为某个规则无法遵循，希望可以共同改进该规则。

参考该规范之前，希望您具有相应的 C++基础能力，而不是通过该文档来学习 C++。1. 了解 C++的 ISO 标准； 2. 熟知 C++的基本语言特性，包括 C++ 03/11/14/17 相关特性； 3. 了解 C++的标准库；

重点关注

1. 约定 C++的编程风格，比如命名，排版等。
2. C++的模块化设计，如何设计头文件，类，接口和函数。
3. C++相关特性的优秀实践，比如常量，类型转换，资源管理，模板等。
4. 现代 C++的优秀实践，包括 C++11/14/17 中可以提高代码可维护性，提高代码可靠性的相关约定。

约定

规则：编程时必须遵守的约定(must)

建议：编程时应该遵守的约定(should)

本规范适用通用 C++标准, 如果没有特定的标准版本，适用所有的版本(C++03/11/14/17)。

例外

无论是'规则'还是'建议'，都必须理解该条目这么规定的原因，并努力遵守。 但是，有些规则和建议可能会有例外。

在不违背总体原则，经过充分考虑，有充足的理由的前提下，可以适当违背规范中约定。 例外破坏了代码的一致性，请尽量避免。'规则'的例外应该是极少的。

下列情况，应风格一致性原则优先：
修改外部开源代码、第三方代码时，应该遵守开源代码、第三方代码已有规范，保持风格统一。
某些特定领域，优先参考其行业规范。

1 原则

好代码的原则

我们参考 Kent Beck 的简单设计四原则来指导我们的如何写出优秀的代码，如何有效地判断我们的代码是优秀的。 1. 通过所有测试（Passes its tests） 2. 尽可能消除重复 (Minimizes duplication) 3. 尽可能清晰表达 (Maximizes clarity) 4. 更少代码元素 (Has fewer elements) 5. 以上四个原则的重要程度依次降低。 这组定义被称做简单设计原则。

第一条强调的是外部需求，这是代码实现最重要的；第二点就是代码的模块架构设计，保证代码的正交性，保证代码更容易修改；第三点是代码的可阅读性，保证代码是容易阅读的；最后一点才是保证代码是简洁的，在简洁和表达力之间，我们更看重表达力。

类和函数设计指导原则

C++是典型的面向对象编程语言，软件工程界已经有很多 OOP 原则来指导我们编写大规模的，高可扩展的，可维护性的代码： - 高内聚，低耦合的基本原则 - SOLID 原则 - 迪米特法则 - “Tell，Don’t ask”原则 - 组合/聚合复用原则

保证静态类型安全

我们希望 C++应该是静态类型安全的，这样可以减少运行时的错误，提高代码的健壮性。但是由于 C++的下面的特性存在，会破坏 C++静态类型安全，我们针对这部分特性要仔细处理。 - unions 联合体 - 类型转换 cast - 缩窄转换 narrowing conversions - 类型退化 type decay - 范围错误 range errors - `void*`类型指针

我们可以通过约束这些特性的使用，或者使用 C++的新特性，比如 variant(C++17)，GSL 的 span，narrow_cast 等来解决这些问题，提高 C++代码的健壮性。

遵循 C++ ISO 标准

希望通过使用 ISO C++标准的特性来编写 C++代码，对于 ISO 标准中未定义的或者编译器实现的特性要谨慎使用，对于 GCC 等编译器的提供的扩展特性也需要谨慎使用，这些特性会导致代码的可移植性比较差。

注意：如果模块中需要使用相关的扩展特性来，那么尽可能将这些特性封装成独立的接口，并且可以通过编译选项关闭或者编译这些特性。对于这些扩展特性的使用，请模块制定特性编程指南来指导这些特性的使用。

优先编译时检查错误

通过编译器来优先保证代码健壮性，而不是通过编写错误处理代码来处理编译就可以发现的异常，比如：

- 通过 const 来保证数据的不变性，防止数据被无意修改。
- 通过 gsl::span 等来保证 char 数组不越界，而不是通过运行时的 length 检查。
- 通过 static_assert 来进行编译时检查。

使用命名空间来限定作用域

全局变量，全局常量和全局类型定义由于都属于全局作用域，在项目中，使用第三方库中容易出现冲突。

命名空间将作用域细分为独立的，具名的作用域，可有效地防止全局作用域的命名冲突。 1. class，struct 等都具有自己的类作用域。 2. 具名的 namespace 可以实现类作用域更上层的作用域。 3. 匿名 namespace 和 static 可以实现文件作用域。

对于没有作用域的宏变量，宏函数强烈建议不使用。

作用域的一些缺点： 1. 虽然可以通过作用域来区分两个命名相同的类型，但是还是具有迷惑性。 2. 内联命名空间会让命名空间内部的成员摆脱限制，让人迷惑。 3. 通过多重嵌套来定义 namespace，会让完整的命名空间比较冗长。

所以，我们使用命名空间的建议如下： - 对于变量，常量和类型定义尽可能使用 namespace，减少全局作用域的冲突 - 不要在头文件中使用 using namespace - 不要使用内联命名空间 - 鼓励在.cpp 文件中通过匿名 namespace 或者 static 来封装，防止不必要的定义通过 API 暴露出去。

优先使用 C++特性而不是 C 特性

C++比起 C 语言更加类型安全，更加抽象。我们更推荐使用 C++的语言特性来编程，比如使用 string 而不是 char*, 使用 vector 而不是原生数组，使用 namespace 而不是 static。

2 命名

通用命名

常见命名风格有： **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(UperCamelCase)和小驼峰(lowerCamelCase)

内核风格(unix_like) 单词全小写，用下划线分割。 如：‘test_result’

匈牙利风格 在‘大驼峰’的基础上，加上前缀；前缀用于表达类型或用途。 如：‘uiSavedCount’, ‘bTested’

规则 2.1.1 标识符命名使用驼峰风格

不考虑匈牙利命名，在内核风格与驼峰风格之间，根据存量代码的情况，我们选择驼峰风格。

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	大驼峰
函数(包括全局函数，作用域函数，成员函数)	大驼峰（接口部分可加前缀，如 XXX_函数名）
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，类、结构体和联合体中的成员变量	小驼峰
常量(const)，枚举值	k+大小写混合
宏	大写+下划线
命名空间	全小写

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。 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。

文件命名

建议 2.2.1 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

对于本文档，我们默认使用.h 和.cpp 作为后缀。

建议 2.2.2 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。

如下： - database_connection.h - database_connection.cpp

结构体，命名空间，枚举等定义的文件名类似。

函数命名

函数命名统一使用大驼峰风格，一般采用动词或者动宾结构。接口部分可加前缀，如 XXX_函数名。

```
class List {
 public:
  void AddElement(const Element& element);
  Element GetElement(const unsigned int index) const;
  bool IsEmpty() const;
  bool MCC_GetClass();
};

namespace utils {
void DeleteUser();
}
```

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——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 { ...
```

对于命名空间的命名，建议全小写：

```
// namespace
namespace osutils {

namespace fileutils {

}

}
```

建议 2.4.1 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

如果模块有自己的定义，请使用统一的 typedef 来定义类型：

```
typedef signed char VOS_INT8;
typedef unsigned char VOS_UINT8;

#if __WORDSIZE == 64
typedef unsigned long int   VOS_UINTPTR;
#else
typedef unsigned int        VOS_UINTPTR;
#endif
```

如果模块为了封装某个类型的信息，方便后续的扩展，可以使用 typedef 来重新定义。

```
typedef uint8_t DeviceID;
// ...
// 若干版本后扩展成 16-bit
typedef uint16_t DeviceID;
```

有特殊作用的类型 `typedef void *Handle;` 注意：**不要使用 #define 进行别名定义，并且在 C++11 以后推荐使用 using 来定义类型。**

除上述理由外，应避免给其本数值类型别名定义。因为类型别名可读性并不好，隐藏了基本数值类型信息，如位宽，是否带符号。 滥用举例：

```
typedef uint16_t MyCounter;
// ...
int Foo(...) {
  MyCounter c;
  // ...
  while (c >= 0) {
    printf("counter = %d\n", c);
    // ...
  }
  // ...
}
```

对'MyCounter'是否可能小于 0，打印时用'%d'还是'%u'都不是很直观，极容易引入上述类似缺陷。

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，成员变量。

```
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 2.5.1 类的成员变量命名使用小驼峰。

```
class Foo {
 private:
  std::string fileName;    // 不添加任何作用域前缀或者后缀
};
```

当构造函数参数和成员变量重名时，可通过 this->来引用成员变量。

```
class MyClass {
 public:
  MyClass(int myVar) : myVar(myVar) { // OK，初始化列表允许同名入参初始化同名成员
```

```
    if (NeedNewVar()) {
      this->myVar = GetValue();  // 注意不要漏掉this->，否则就成了给入参
赋值
    }
  }

 private:
  int myVar;
};
```

宏、常量、枚举命名

宏采用全大写，下划线连接的格式。常量、枚举值使用 k+大小写混合。 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。

```
#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

enum TintColor {    // 注意，枚举类型名用大驼峰，其下面的取值是k+大小写混合
  kRed,
  kDarkRed,
  kGreen,
  kLightGreen
};

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

namespace utils {
const unsigned int kFileSize = 200;      // 全局常量
}
```

3 格式

尽管有些编程的排版风格因人而异，但是我们强烈建议和要求使用统一的编码风格，以便所有人都能够轻松的阅读和理解代码，增强代码的可维护性。

行宽

建议 3.1.1 行宽不超过 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/yyy
y/zzzz/xyz.h, because xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
#endif
```

缩进

规则 3.2.1 使用空格进行缩进，每次缩进 2 个空格

只允许使用空格(space)进行缩进，每次缩进为 2 个空格。

大括号

规则 3.3.1 除函数外，使用 K&R 缩进风格

函数左大括号跟随语句放行末。 右大括号独占一行，除非后面跟着同一语句的剩余部分，如 do 语句中的 while，或者 if 语句的 else/else if，或者逗号、分号。

如：

```
struct MyType {      // 跟随语句放行末，前置 1 空格
  ...
};

int Foo(int a) {                    // 函数左大括号跟随语句放行末
  if (...) {
    ...
  } else {
    ...
  }
}
```

推荐这种风格的理由：

- 代码更紧凑；
- 相比另起一行，放行末使代码阅读节奏感上更连续；
- 符合后来语言的习惯，符合业界主流习惯；
- 现代集成开发环境（IDE）都具有代码缩进对齐显示的辅助功能，大括号放在行尾并不会对缩进和范围产生理解上的影响。

对于空函数体，可以将大括号放在同一行：

```
class MyClass {
 public:
  MyClass() : value(0) {}

 private:
  int value;
};
```

函数声明和定义

规则 3.4.1 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：

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

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

ReturnType LongFunctionName(ArgType paramName1, ArgType paramName2, // 行宽限制，进行换行
    ArgType paramName3, ArgType paramName4, ArgType paramName5) {     // Good: 换行后 4 空格缩进
  ...
}

ReturnType ReallyReallyReallyReallyLongFunctionName(             // 行宽不满足第 1 个参数，直接换行
    ArgType paramName1, ArgType paramName2, ArgType paramName3) { // Good: 换行后 4 空格缩进
  ...
}
```

函数调用

规则 3.5.1 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：

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

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

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

ReturnType result = VeryVeryVeryLongFunctionName(            // 行宽不满足第 1 个参数，直接换行
    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
int result = DealWithStructureLikeParams(left.x, left.y,     // 表示一组相关参数
                                         right.x, right.y);  // 表示另外一组相关参数
```

if 语句

规则 3.6.1 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 3.6.2 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
```

循环语句

规则 3.7.1 循环语句要求使用大括号

和 if 语句类似，我们要求 for/while 循环语句必须加上的大括号，即使循环体是空的，或者循环语句只有一条。

```
for (int i = 0; i < someRange; i++) {
  DoSomething();
}
```

如果循环体是空的，应该使用空的大括号，而不是使用单个分号。 单个分号容易被遗漏，也容易被误认为是循环语句中的一部分。

```
for (int i = 0; i < someRange; i++) { } // Good： for 循环体是空，使用大括号，而不是使用分号

while (someCondition) { }  // Good：while 循环体是空，使用大括号，而不是使用分号

while (someCondition) {
  continue;  // Good：continue 表示空逻辑，可以使用大括号也可以不使用
}
```

坏的例子：

```
for (int i = 0; i < someRange; i++) ;   // Bad： for 循环体是空，也不要只使用分号，要使用大括号
```

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

switch 语句

规则 3.8.1 switch 语句的 case/default 要缩进一层

switch 语句的缩进风格如下：

```
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;
}
```

表达式

建议 3.9.1 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

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

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

变量赋值

规则 3.10.1 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：

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

初始化

初始化包括结构体、联合体、及数组的初始化

规则 3.11.1 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
};
```

指针与引用

建议 3.12.1 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。

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

建议 3.12.2 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 & p = i;    // Bad
int&p = i;      // Bad
```

编译预处理

规则 3.13.1 编译预处理的“#”统一放在行首，嵌套编译预处理语句时，“#”不缩进

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“#”不缩进

```
#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
```

空格和空行

建议 3.14.1 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：

- if, switch, case, do, while, for 等关键字之后加空格；
- 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：

```
void Foo(int b) {  // Good：大括号前应该留空格

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[kBufSize] = {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：二元操作符前后要加空格
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) {  // Good：if 关键字和括号之间加空格，括号内条件语句前后不加空格
  ...
} else {          // Good：else 关键字和大括号之间加空格
  ...
}

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(< 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) : someVar(var) {
  DoSomething();
}

// 位域表示也留有空格
struct XX {
  char a : 4;
  char b : 5;
  char c : 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 someVar;
};

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

注意：当前的集成开发环境（IDE）可以设置删除行尾的空格，请正确配置。

建议 3.14.2 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。

```
int Foo() {
  ...
}


// Bad：两个函数定义间超过了一个空行
int Bar() {
  ...
}


if (...) {
        // Bad：大括号内的代码块行首不要加入空行
  ...
        // Bad：大括号内的代码块行尾不要加入空行
}

int Foo(...) {
        // Bad：函数体内行首不要加空行
  ...
}
```

类

规则 3.15.1 类访问控制块的声明依次序是 public:, protected:, private:，每个都缩进 1 个空格

```
class MyClass : public BaseClass {
 public:       // 注意没有缩进
  MyClass();  // 标准的4 空格缩进
  explicit MyClass(int var);
  ~MyClass() {}

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void SetVar(int var) { someVar = var; }
  int GetVar() const { return someVar; }

 private:
  bool SomeInternalFunction();

  int someVar;
  int someOtherVar;
};
```

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 3.15.2 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) : someVar(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)
    : someVar(var), someOtherVar(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)
    : someVar(var),             // 缩进4 个空格
      someOtherVar(var + 1) {
  DoSomething();
}
```

4 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。

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。**

文件头注释

规则 4.2.1 文件头注释必须包含版权许可

```
/*
 * Copyright (c) [2019] [name of copyright holder]
 * [Software Name] is licensed under the Mulan PSL v1.
 * You can use this software according to the terms and conditions of t
he Mulan PSL v1.
 * You may obtain a copy of Mulan PSL v1 at:
 *   http://license.coscl.org.cn/MulanPSL
 * THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OF
 ANY KIND, EITHER EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO NON-INFRINGEMENT, MERCHANTABIL
ITY OR FIT FOR A PARTICULAR
 * PURPOSE.
 * See the Mulan PSL v1 for more details.
 */
```

函数头注释

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
 * 注意，内存 buf 由调用者负责释放
 */
int WriteString(const char *buf, int len);
```

坏的例子：

```
/*
 * 函数名：WriteString
 * 功能：写入字符串
 * 参数：
 * 返回值：
 */
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

代码注释

规则 4.4.1 代码注释放于对应代码的上方或右边

规则 4.4.2 注释符与注释内容间要有 1 空格；右置注释与前面代码至少 1 空格

代码上方的注释，应该保持对应代码一样的缩进。 选择并统一使用如下风格之一：
使用//

```
// 这是单行注释
DoSomething();

// 这是多行注释
// 第二行
DoSomething();
```

使用/*' '*/

```
/* 这是单行注释 */
DoSomething();

/*
 * 另一种方式的多行注释
 * 第二行
 */
DoSomething();
```

代码右边的注释，与代码之间，至少留 1 空格，建议不超过 4 空格。 通常使用扩展后的 TAB 键即可实现 1-4 空格的缩进。

选择并统一使用如下风格之一：

```
int foo = 100;  // 放右边的注释
int bar = 200;  /* 放右边的注释 */
```

右置格式在适当的时候，上下对齐会更美观。 对齐后的注释，离左边代码最近的那一行，保证 1-4 空格的间隔。 例：

```
const int kConst = 100;           /* 相关的同类注释，可以考虑上下对齐 */
const int kAnotherConst = 200;    /* 上下对齐时，与左侧代码保持间隔*/
```

当右置的注释超过行宽时，请考虑将注释置于代码上方。

规则 4.4.3 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。

建议 4.4.1 正式交付给客户的代码不能包含 TODO/TBD/FIXME 注释

TODO/TBD 注释一般用来描述已知待改进、待补充的修改点 FIXME 注释一般用来描述已知缺陷 它们都应该有统一风格，方便文本搜索统一处理。如：

```
// TODO(<author-name>): 补充XX 处理
// FIXME: XX 缺陷
```

5 头文件

头文件职责

头文件是模块或文件的对外接口，头文件的设计体现了大部分的系统设计。 头文件中适合放置接口的声明，不适合放置实现（内联函数除外）。对于 cpp 文件中内部才需要使用的函数、宏、枚举、结构定义等不要放在头文件中。 头文件应当职责单一。头文件过于复杂，依赖过于复杂还是导致编译时间过长的主要原因。

建议 5.1.1 每一个.cpp 文件应有一个对应的.h 文件，用于声明需要对外公开的类与接口

通常情况下，每个.cpp 文件都有一个相应的.h，用于放置对外提供的函数声明、宏定义、类型定义等。 如果一个.cpp 文件不需要对外公布任何接口，则其就不应当存在。 例外：**程序的入口（如 main 函数所在的文件），单元测试代码，动态库代码。**

示例:

```
// Foo.h

#ifndef FOO_H
#define FOO_H

class Foo {
 public:
  Foo();
  void Fun();

 private:
  int value;
};

#endif

// Foo.cpp
#include "Foo.h"

namespace { // Good: 对内函数的声明放在.cpp 文件的头部，并声明为匿名 namespa
ce 或者 static 限制其作用域
void Bar()
{
}
}

...

void Foo::Fun() {
  Bar();
}
```

头文件依赖

规则 5.2.1 禁止头文件循环依赖

头文件循环依赖，指 a.h 包含 b.h，b.h 包含 c.h，c.h 包含 a.h， 导致任何一个头文件修改，都导致所有包含了 a.h/b.h/c.h 的代码全部重新编译一遍。
而如果是单向依赖，如 a.h 包含 b.h，b.h 包含 c.h，而 c.h 不包含任何头文件，则修改 a.h 不会导致包含了 b.h/c.h 的源代码重新编译。

头文件循环依赖直接体现了架构设计上的不合理，可通过优化架构去避免。

规则 5.2.2 禁止包含用不到的头文件

用不到的头文件被包含的同时引入了不必要的依赖，增加了模块或单元之间的耦合度，只要该头文件被修改，代码就要重新编译。

很多系统中头文件包含关系复杂，开发人员为了省事起见，直接包含一切想到的头文件，甚至发布了一个 god.h，其中包含了所有头文件，然后发布给各个项目组使用，这种只图一时省事的做法，导致整个系统的编译时间进一步恶化，并对后来人的维护造成了巨大的麻烦。

规则 5.2.3 头文件应当自包含

简单的说，自包含就是任意一个头文件均可独立编译。如果一个文件包含某个头文件，还要包含另外一个头文件才能工作的话，给这个头文件的用户增添不必要的负担。

示例： 如果 a.h 不是自包含的，需要包含 b.h 才能编译，会带来的危害： 每个使用 a.h 头文件的.cpp 文件，为了让引入的 a.h 的内容编译通过，都要包含额外的头文件 b.h。 额外的头文件 b.h 必须在 a.h 之前进行包含，这在包含顺序上产生了依赖。

规则 5.2.4 头文件必须编写#define 保护，防止重复包含

为防止头文件被重复包含，所有头文件都应当使用 #define 保护；不要使用 #pragma once

定义包含保护符时，应该遵守如下规则： 1）保护符使用唯一名称； 2）不要在受保护部分的前后放置代码或者注释，文件头注释除外。

示例：假定 VOS 工程的 timer 模块的 timer.h，其目录为 VOS/include/timer/Timer.h,应按如下方式保护：

```
#ifndef VOS_INCLUDE_TIMER_TIMER_H
#define VOS_INCLUDE_TIMER_TIMER_H
...
#endif
```

也可以不用像上面添加路径，但是要保证当前工程内宏是唯一的。

```
#ifndef TIMER_H
#define TIMER_H
...
#endif
```

建议 5.2.1 禁止通过声明的方式引用外部函数接口、变量

只能通过包含头文件的方式使用其他模块或文件提供的接口。 通过 extern 声明的方式使用外部函数接口、变量，容易在外部接口改变时可能导致声明和定义不一致。同时这种隐式依赖，容易导致架构腐化。

不符合规范的案例：

// a.cpp 内容

```
extern int Fun();    // Bad: 通过 extern 的方式使用外部函数

void Bar() {
  int i = Fun();
  ...
}
```

// b.cpp 内容

```
int Fun() {
  // Do something
}
```

应该改为：

// a.cpp 内容

```
#include "b.h"    // Good: 通过包含头文件的方式使用其他.cpp 提供的接口

void Bar() {
  int i = Fun();
  ...
}
```

// b.h 内容

```
int Fun();
```

// b.cpp 内容

```
int Fun() {
  // Do something
}
```

例外，有些场景需要引用其内部函数，但并不想侵入代码时，可以 extern 声明方式引用。 如： 针对某一内部函数进行单元测试时，可以通过 extern 声明来引用被测函数； 当需要对某一函数进行打桩、打补丁处理时，允许 extern 声明该函数。

规则 5.2.5 禁止在 extern “C”中包含头文件

在 extern “C” 中包含头文件，有可能会导致 extern “C” 嵌套，部分编译器对 extern “C” 嵌套层次有限制，嵌套层次太多会编译错误。

在 C，C++混合编程的情况下，在 extern “C”中包含头文件，可能会导致被包含头文件的原有意图遭到破坏，比如链接规范被不正确地更改。

示例，存在 a.h 和 b.h 两个头文件：

// a.h 内容

```
...
#ifdef __cplusplus
void Foo(int);
#define A(value) Foo(value)
#else
void A(int)
#endif
```

// b.h 内容

```
...
#ifdef __cplusplus
extern "C" {
#endif

#include "a.h"
void B();

#ifdef __cplusplus
}
#endif
```

使用 C++预处理器展开 b.h，将会得到

```
extern "C" {
  void Foo(int);
  void B();
}
```

按照 a.h 作者的本意，函数 Foo 是一个 C++ 自由函数，其链接规范为 “C++”。 但在 b.h 中，由于 `#include "a.h"` 被放到了 `extern "C"` 的内部，函数 Foo 的链接规范被不正确地更改了。

例外： 如果在 C++ 编译环境中，想引用纯 C 的头文件，这些 C 头文件并没有 `extern "C"` 修饰。非侵入式的做法是，在 `extern "C"` 中去包含 C 头文件。

建议 5.2.2 尽量避免使用前置声明，而是通过#include 来包含头文件

前置声明（forward declaration）是类、函数和模板的纯粹声明，没伴随着其定义。

- 优点：
  1. 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。
  2. 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。
- 缺点：
  1. 前置声明隐藏了依赖关系，头文件改动时，用户的代码会跳过必要的重新编译过程。
  2. 前置声明可能会被库的后续更改所破坏。前置声明函数或模板有时会妨碍头文件开发者变动其 API. 例如扩大形参类型，加个自带默认参数的模板形参等等。
  3. 前置声明来自命名空间 `std::` 的 symbol 时，其行为未定义（在 C++11 标准规范中明确说明）。
  4. 前置声明了不少来自头文件的 symbol 时，就会比单单一行的 include 冗长。
  5. 仅仅为了能前置声明而重构代码（比如用指针成员代替对象成员）会使代码变得更慢更复杂。
  6. 很难判断什么时候该用前置声明，什么时候该用`#include`，某些场景下面前置声明和`#include` 互换以后会导致意想不到的结果。

所以我们尽可能避免使用前置声明，而是使用#include 头文件来保证依赖关系。

建议 5.2.3 头文件包含顺序：首先是.cpp 相应的.h 文件，其它头文件按照稳定度排序

使用标准的头文件包含顺序可增强可读性, 避免隐藏依赖，建议按照稳定度排序：cpp 对应的头文件, C/C++标准库, 系统库的.h, 其他库的.h, 本项目内其他的.h。

举例，Foo.cpp 中包含头文件的次序如下:

```
#include "Foo/Foo.h"

#include <cstdlib>
#include <string>

#include <linux/list.h>
#include <linux/time.h>

#include "platform/Base.h"
#include "platform/Framework.h"
```

```
#include "project/public/Log.h"
```

将 Foo.h 放在最前面可以保证当 Foo.h 遗漏某些必要的库，或者有错误时，Foo.cpp 的构建会立刻中止，减少编译时间。 对于头文件中包含顺序也参照此建议。

例外： 平台特定代码需要条件编译，这些代码可以放到其它 includes 之后。

```
#include "foo/public/FooServer.h"

#include "base/Port.h"  // For LANG_CXX11.

#ifdef LANG_CXX11
#include <initializer_list>
#endif  // LANG_CXX11
```

6 作用域

命名空间

命名空间里的内容不缩进。

建议 6.1.1 对于 cpp 文件中不需要导出的变量，常量或者函数，请使用匿名 namespace 封装或者用 static 修饰

在 C++ 2003 标准规范中，使用 static 修饰文件作用域的变量，函数等被标记为 deprecated 特性，所以更推荐使用匿名 namespace。

主要原因如下： 1. static 在 C++中已经赋予了太多的含义，静态函数成员变量，静态成员函数，静态全局变量，静态函数局部变量，每一种都有特殊的处理。 2. static 只能保证变量，常量和函数的文件作用域，但是 namespace 还可以封装类型等。 3. 统一 namespace 来处理 C++的作用域，而不需要同时使用 static 和 namespace 来管理。 4. static 修饰的函数不能用来实例化模板，而匿名 namespace 可以。

但是不要在 .h 中使用中使用匿名 namespace 或者 static。

```
// Foo.cpp

namespace {
const int kMaxCount = 20;
void InternalFun(){};
}

void Foo::Fun() {
  int i = kMaxCount;
```

```
  InternalFun();
}
```

规则 6.1.1 不要在头文件中或者#include 之前使用 using 导入命名空间

说明：使用 using 导入命名空间会影响后续代码，易造成符号冲突，所以不要在头文件以及源文件中的#include 之前使用 using 导入命名空间。 示例：

```
// 头文件 a.h
namespace namespacea {
int Fun(int);
}

// 头文件 b.h
namespace namespaceb {
int Fun(int);
}

using namespace namespaceb;

void G() {
  Fun(1);
}

// 源代码 a.cpp
#include "a.h"
using namespace namespacea;
#include "b.h"

void main() {
  G(); // using namespace namespacea 在#include “b.h”之前，引发歧义：name
spacea::Fun，namespaceb::Fun 调用不明确
}
```

对于在头文件中使用 using 导入单个符号或定义别名，允许在模块自定义名字空间中使用，但禁止在全局名字空间中使用。

```
// foo.h

#include <fancy/string>
using fancy::string;  // Bad，禁止向全局名字空间导入符号

namespace foo {
using fancy::string;  // Good，可以在模块自定义名字空间中导入符号
using MyVector = fancy::vector<int>;  // Good，C++11 可在自定义名字空间中
定义别名
}
```

全局函数和静态成员函数

建议 6.2.1 优先使用命名空间来管理全局函数，如果和某个 class 有直接关系的，可以使用静态成员函数

说明：非成员函数放在名字空间内可避免污染全局作用域， 也不要用类+静态成员方法来简单管理全局函数。 如果某个全局函数和某个类有紧密联系， 那么可以作为类的静态成员函数。

如果你需要定义一些全局函数，给某个 cpp 文件使用，那么请使用匿名 namespace 来管理。

```
namespace mynamespace {
int Add(int a, int b);
}

class File {
 public:
  static File CreateTempFile(const std::string& fileName);
};
```

全局常量和静态成员常量

建议 6.3.1 优先使用命名空间来管理全局常量，如果和某个 class 有直接关系的，可以使用静态成员常量

说明：全局常量放在命名空间内可避免污染全局作用域， 也不要用类+静态成员常量来简单管理全局常量。 如果某个全局常量和某个类有紧密联系， 那么可以作为类的静态成员常量。

如果你需要定义一些全局常量，只给某个 cpp 文件使用，那么请使用匿名 namespace 来管理。

```
namespace mynamespace {
const int kMaxSize = 100;
}

class File {
 public:
  static const std::string kName;
};
```

全局变量

建议 6.4.1 尽量避免使用全局变量，考虑使用单例模式

说明：全局变量是可以修改和读取的，那么这样会导致业务代码和这个全局变量产生数据耦合。

```
int counter = 0;

// a.cpp
counter++;

// b.cpp
counter++;

// c.cpp
cout << counter << endl;
```

使用单实例模式

```
class Counter {
 public:
  static Counter& GetInstance() {
    static Counter counter;
    return counter;
  }  // 单实例实现简单举例

  void Increase() {
    value++;
  }

  void Print() const {
    std::cout << value << std::endl;
  }

 private:
  Counter() : value(0) {}

 private:
  int value;
};

// a.cpp
Counter::GetInstance().Increase();

// b.cpp
Counter::GetInstance().Increase();
```

```
// c.cpp
Counter::GetInstance().Print();
```

实现单例模式以后，实现了全局唯一一个实例，和全局变量同样的效果，并且单实例提供了更好的封装性。

例外：有的时候全局变量的作用域仅仅是模块内部，这样进程空间里面就会有多个全局变量实例，每个模块持有一份，这种场景下是无法使用单例模式解决的。

7 类

如果仅有数据成员，使用结构体，其他使用类

构造，拷贝构造，赋值和析构函数

构造，拷贝，移动和析构函数提供了对象的生命周期管理方法： - 构造函数（constructor）： `X()` - 拷贝构造函数（copy constructor）： `X(const X&)` - 拷贝赋值操作符（copy assignment）： `operator=(const X&)`
- 移动构造函数（move constructor）： `X(X&&)` *C++11 以后提供* - 移动赋值操作符（move assignment）： `operator=(X&&)` *C++11 以后提供* - 析构函数（destructor）： `~X()`

规则 7.1.1 类的成员变量必须显式初始化

说明：如果类有成员变量，没有定义构造函数，又没有定义默认构造函数，编译器将自动生成一个构造函数，但编译器生成的构造函数并不会对成员变量进行初始化，对象状态处于一种不确定性。

例外： - 如果类的成员变量具有默认构造函数，那么可以不需要显式初始化。

示例：如下代码没有构造函数，私有数据成员无法初始化：

```
class Message {
 public:
  void ProcessOutMsg() {
    //…
  }
 private:
  unsigned int msgID;
  unsigned int msgLength;
  unsigned char* msgBuffer;
  std::string someIdentifier;
};

Message message;    // message 成员变量没有初始化
message.ProcessOutMsg();    // 后续使用存在隐患
```

```
// 因此，有必要定义默认构造函数，如下：
class Message {
 public:
  Message() : msgID(0), msgLength(0) {
  }

  void ProcessOutMsg() {
    // …
  }

 private:
  unsigned int msgID;
  unsigned int msgLength;
  unsigned char* msgBuffer;
  std::string someIdentifier; //具有默认构造函数，不需要显式初始化
};
```

建议 7.1.1 成员变量优先使用声明时初始化（C++11）和构造函数初始化列表初始化

说明：C++11 的声明时初始化可以一目了然的看出成员初始值，应当优先使用。如果成员初始化值和构造函数相关，或者不支持 C++11，则应当优先使用构造函数初始化列表来初始化成员。相比起在构造函数体中对成员赋值，初始化列表的代码更简洁，执行性能更好，而且可以对 const 成员和引用成员初始化。

```
class Message {
 public:
  Message() : msgLength(0) {  // Good，优先使用初始化列表
    msgBuffer = NULL;     // Bad，不推荐在构造函数中赋值
  }

 private:
  unsigned int msgID{0};  // Good，C++11 中使用
  unsigned int msgLength;
  unsigned char* msgBuffer;
};
```

规则 7.1.2 为避免隐式转换，将单参数构造函数声明为 explicit

说明：单参数构造函数如果没有用 explicit 声明，则会成为隐式转换函数。 示例：

```
class Foo {
 public:
  explicit Foo(const string& name): name(name) {
  }
 private:
  string name;
};
```

```
void ProcessFoo(const Foo& foo){}

int main(void) {
  std::string test = "test";
  ProcessFoo(test);  // 编译不通过
  return 0;
}
```

上面的代码编译不通过，因为 ProcessFoo 需要的参数是 Foo 类型，传入的 string 类型不匹配。

如果将 Foo 构造函数的 explicit 关键字移除，那么调用 ProcessFoo 传入的 string 就会触发隐式转换，生成一个临时的 Foo 对象。往往这种隐式转换是让人迷惑的，并且容易隐藏 Bug，得到了一个不期望的类型转换。所以对于单参数的构造函数是要求 explicit 声明。

规则 7.1.3 如果不需要拷贝构造函数、赋值操作符 / 移动构造函数、赋值操作符，请明确禁止

说明：如果用户不定义，编译器默认会生成拷贝构造函数和拷贝赋值操作符， 移动构造和移动赋值操作符（移动语义的函数 C++11 以后才有）。 如果我们不要使用拷贝构造函数，或者赋值操作符，请明确拒绝：

1. 将拷贝构造函数或者赋值操作符设置为 private，并且不实现：

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

2. 使用 C++11 提供的 delete, 请参见后面现代 C++的相关章节。

规则 7.1.4 拷贝构造和拷贝赋值操作符应该是成对出现或者禁止

拷贝构造函数和拷贝赋值操作符都是具有拷贝语义的，应该同时出现或者禁止。

```
// 同时出现
class Foo {
 public:
  ...
  Foo(const Foo&);
  Foo& operator=(const Foo&);
  ...
};

// 同时 default， C++11 支持
```

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

// 同时禁止，C++11 可以使用 delete
class Foo {
 private:
  Foo(const Foo&);
  Foo& operator=(const Foo&);
};
```

规则 7.1.5 移动构造和移动赋值操作符应该是成对出现或者禁止

在 C++11 中增加了 move 操作，如果需要某个类支持移动操作，那么需要实现移动构造和移动赋值操作符。

移动构造函数和移动赋值操作符都是具有移动语义的，应该同时出现或者禁• 止。

```
// 同时出现
class Foo {
 public:
  ...
  Foo(Foo&&);
  Foo& operator=(Foo&&);
  ...
};

// 同时 default， C++11 支持
class Foo {
 public:
  Foo(Foo&&) = default;
  Foo& operator=(Foo&&) = default;
};

// 同时禁止，使用 C++11 的 delete
class Foo {
 public:
  Foo(Foo&&) = delete;
  Foo& operator=(Foo&&) = delete;
};
```

规则 7.1.6 禁止在构造函数和析构函数中调用虚函数

说明：在构造函数和析构函数中调用当前对象的虚函数，会导致未实现多态的行为。在 C++中，一个基类一次只构造一个完整的对象。

示例：类 Base 是基类，Sub 是派生类

```
class Base {
 public:
  Base();
  virtual void Log() = 0;     // 不同的派生类调用不同的日志文件
};

Base::Base() {          // 基类构造函数
  Log();                // 调用虚函数 Log
}

class Sub : public Base {
 public:
  virtual void Log();
};
```

当执行如下语句： `Sub sub;` 会先执行 Sub 的构造函数，但首先调用 Base 的构造函数，由于 Base 的构造函数调用虚函数 Log，此时 Log 还是基类的版本，只有基类构造完成后，才会完成派生类的构造，从而导致未实现多态的行为。 同样的道理也适用于析构函数。

继承

规则 7.2.1 基类的析构函数应该声明为 virtual

说明：只有基类析构函数是 virtual，通过多态调用的时候才能保证派生类的析构函数被调用。

示例：基类的析构函数没有声明为 virtual 导致了内存泄漏。

```
class Base {
 public:
  virtual std::string getVersion() = 0;

  ~Base() {
    std::cout << "~Base" << std::endl;
  }
};

class Sub : public Base {
 public:
  Sub() : numbers(NULL) {
  }

  ~Sub() {
    delete[] numbers;
    std::cout << "~Sub" << std::endl;
  }
```

```
  int Init() {
    const size_t numberCount = 100;
    numbers = new (std::nothrow) int[numberCount];
    if (numbers == NULL) {
      return -1;
    }

    ...
  }

  std::string getVersion() {
    return std::string("hello!");
  }
private:
  int* numbers;
};

int main(int argc, char* args[]) {
  Base* b = new Sub();

  delete b;
  return 0;
}
```

由于基类 Base 的析构函数没有声明为 virtual，当对象被销毁时，只会调用基类的析构函数，不会调用派生类 Sub 的析构函数，导致内存泄漏。

规则 7.2.2 禁止虚函数使用缺省参数值

说明：在 C++中，虚函数是动态绑定的，但函数的缺省参数却是在编译时就静态绑定的。这意味着你最终执行的函数是一个定义在派生类，但使用了基类中的缺省参数值的虚函数。为了避免虚函数重载时，因参数声明不一致给使用者带来的困惑和由此导致的问题，规定所有虚函数均不允许声明缺省参数值。 示例：虚函数 display 缺省参数值 text 是由编译时刻决定的，而非运行时刻，没有达到多态的目的：

```
class Base {
 public:
  virtual void Display(const std::string& text = "Base!") {
    std::cout << text << std::endl;
  }

  virtual ~Base(){}
};

class Sub : public Base {
 public:
  virtual void Display(const std::string& text  = "Sub!") {
    std::cout << text << std::endl;
```

```
  }

  virtual ~Sub(){}
};

int main() {
  Base* base = new Sub();
  Sub* sub = new Sub();

  ...

  base->Display();  // 程序输出结果: Base! 而期望输出: Sub!
  sub->Display();   // 程序输出结果: Sub!

  delete base;
  delete sub;
  return 0;
};
```

规则 7.2.3 禁止重新定义继承而来的非虚函数

说明：因为非虚函数无法实现动态绑定，只有虚函数才能实现动态绑定：只要操作基类的指针，即可获得正确的结果。

示例：

```
class Base {
 public:
  void Fun();
};

class Sub : public Base {
 public:
  void Fun();
};

Sub* sub = new Sub();
Base* base = sub;

sub->Fun();    // 调用子类的Fun
base->Fun();   // 调用父类的Fun
//...
```

多重继承

在实际开发过程中使用多重继承的场景是比较少的，因为多重继承使用过程中有下面的典型问题： 1. 菱形继承所带来的数据重复，以及名字二义性。因此，C++引入了 virtual 继承来解决这类问题; 2. 即便不是菱形继承，多个父类之间的名字也可能

存在冲突，从而导致的二义性; 3. 如果子类需要扩展或改写多个父类的方法时，造成子类的职责不明，语义混乱; 4. 相对于委托，继承是一种白盒复用，即子类可以访问父类的 protected 成员, 这会导致更强的耦合。而多重继承，由于耦合了多个父类，相对于单根继承，这会产生更强的耦合关系。

多重继承具有下面的优点： 多重继承提供了一种更简单的组合来实现多种接口或者类的组装与复用。

所以，对于多重继承的只有下面几种情况下面才允许使用多重继承。

建议 7.3.1 使用多重继承来实现接口分离与多角色组合

如果某个类需要实现多重接口，可以通过多重继承把多个分离的接口组合起来，类似 scala 语言的 traits 混入。

```
class Role1 {};
class Role2 {};
class Role3 {};

class Object1 : public Role1, public Role2 {
  // ...
};

class Object2 : public Role2, public Role3 {
  // ...
};
```

在 C++标准库中也有类似的实现样例：

```
class basic_istream {};
class basic_ostream {};

class basic_iostream : public basic_istream, public basic_ostream {

};
```

重载

重载操作符要有充分理由,而且不要改变操作符原有语义，例如不要使用 '+' 操作符来做减运算。 操作符重载令代码更加直观，但也有一些不足： - 混淆直觉，误以为该操作和内建类型一样是高性能的，忽略了性能降低的可能； - 问题定位时不够直观，按函数名查找比按操作符显然更方便。 - 重载操作符如果行为定义不直观(例如将'+' 操作符来做减运算)，会让代码产生混淆。 - 赋值操作符的重载引入的隐式转换会隐藏很深的 bug。可以定义类似 Equals()、CopyFrom()等函数来替代=,==操作符。

8 函数

函数设计

建议 8.1.1 避免函数过长，函数不超过 50 行（非空非注释）

函数应该可以一屏显示完 (50 行以内)，只做一件事情，而且把它做好。

过长的函数往往意味着函数功能不单一，过于复杂，或过分呈现细节，未进行进一步抽象。

例外：某些实现算法的函数，由于算法的聚合性与功能的全面性，可能会超过 50 行。

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。 建议将其拆分为更加简短并易于管理的若干函数，以便于他人阅读和修改代码。

内联函数

建议 8.2.1 内联函数不超过 10 行（非空非注释）

说明：内联函数具有一般函数的特性，它与一般函数不同之处只在于函数调用的处理。一般函数进行调用时，要将程序执行权转到被调用函数中，然后再返回到调用它的函数中；而内联函数在调用时，是将调用表达式用内联函数体来替换。

内联函数只适合于只有 1~10 行的小函数。对一个含有许多语句的大函数，函数调用和返回的开销相对来说微不足道，也没有必要用内联函数实现，一般的编译器会放弃内联方式，而采用普通的方式调用函数。

如果内联函数包含复杂的控制结构，如循环、分支(switch)、try-catch 等语句，一般编译器将该函数视同普通函数。 **虚函数、递归函数不能被用来做内联函数**。

函数参数

建议 8.3.1 函数参数使用引用取代指针

说明：引用比指针更安全，因为它一定非空，且一定不会再指向其他目标；引用不需要检查非法的 NULL 指针。

选择 const 避免参数被修改，让代码阅读者清晰地知道该参数不被修改，可大大增强代码可读性。

建议 8.3.2 使用强类型参数，避免使用 void* 尽管不同的语言对待强类型和弱类型有自己的观点，但是一般认为 c/c++是强类型语言，既然我们使用的语言是强类型的，就应该保持这样的风格。 好处是尽量让编译器在编译阶段就检查出类型不匹配的问题。

使用强类型便于编译器帮我们发现错误，如下代码中注意函数 FooListAddNode 的使用：

```
struct FooNode {
  struct List link;
  int foo;
};

struct BarNode {
  struct List link;
  int bar;
}

void FooListAddNode(void *node) { // Bad: 这里用 void * 类型传递参数
  FooNode *foo = (FooNode *)node;
  ListAppend(&fooList, &foo->link);
}

void MakeTheList() {
  FooNode *foo = nullptr;
  BarNode *bar = nullptr;
  ...

  FooListAddNode(bar);        // Wrong: 这里本意是想传递参数 foo，但错传了
 bar，却没有报错
}
```

1. 可以使用模板函数来实现参数类型的变化。
2. 可以使用基类指针来实现多态。

建议 8.3.3 函数的参数个数不超过 5 个

函数的参数过多，会使得该函数易于受外部变化的影响，从而影响维护工作。函数的参数过多同时也会增大测试的工作量。

如果超过可以考虑: - 看能否拆分函数 - 看能否将相关参数合在一起，定义结构体

9 C++其他特性

常量与初始化

不变的值更易于理解、跟踪和分析，所以应该尽可能地使用常量代替变量，定义值的时候，应该把 const 作为默认的选项。

建议 9.1.1 不允许使用宏来表示常量

说明：宏是简单的文本替换，在预处理阶段时完成，运行报错时直接报相应的值；跟踪调试时也是显示值，而不是宏名；宏没有类型检查，不安全；宏没有作用域。

```
#define MAX_MSISDN_LEN 20    // 不好

// C++请使用const 常量
const int kMaxMsisdnLen = 20; // 好

// 对于C++11 以上版本，可以使用constexpr
constexpr int kMaxMsisdnLen = 20;
```

建议 9.1.2 一组相关的整型常量应定义为枚举

说明：枚举比#define 或 const int 更安全。编译器会检查参数值是否位于枚举取值范围内，避免错误发生。

```
// 好的例子:
enum Week {
  kSunday,
  kMonday,
  kTuesday,
  kWednesday,
  kThursday,
  kFriday,
  kSaturday
};

enum Color {
  kRed,
  kBlack,
  kBlue
};

void ColorizeCalendar(Week today, Color color);

ColorizeCalendar(kBlue, kSunday); // 编译报错，参数类型错误

// 不好的例子:
```

```
const int kSunday = 0;
const int kMonday = 1;

const int kRed = 0;
const int kBlack = 1;

bool ColorizeCalendar(int today, int color);
ColorizeCalendar(kBlue, kSunday); // 不会报错
```

当枚举值需要对应到具体数值时，须在声明时显式赋值。否则不需要显式赋值，以避免重复赋值，降低维护(增加、删除成员)工作量。

```
// 好的例子：S 协议里定义的设备 ID 值，用于标识设备类型
enum DeviceType {
  kUnknown = -1,
  kDsmp = 0,
  kIsmg = 1,
  kWapportal = 2
};
```

建议 9.1.3 不允许使用魔鬼数字

所谓魔鬼数字即看不懂、难以理解的数字。

魔鬼数字并非一个非黑即白的概念，看不懂也有程度，需要自行判断。 例如数字 12，在不同的上下文中情况是不一样的： type = 12; 就看不懂，但 `month = year * 12;` 就能看懂。 数字 0 有时候也是魔鬼数字，比如 `status = 0;` 并不能表达是什么状态。

解决途径： 对于局部使用的数字，可以增加注释说明 对于多处使用的数字，必须定义 const 常量，并通过符号命名自注释。

禁止出现下列情况： 没有通过符号来解释数字含义，如 `const int kZero = 0` 符号命名限制了其取值，如 `const int kXxTimerInterval = 300`，直接使用 `kXxTimerInterval` 来表示该常量是定时器的时间间隔。

规则 9.1.1 常量应该保证单一职责

说明：一个常量只用来表示一个特定功能，即一个常量不能有多种用途。

```
// 好的例子：协议 A 和协议 B，手机号(MSISDN)的长度都是 20。
const unsigned int kAMaxMsisdnLen = 20;
const unsigned int kBMaxMsisdnLen = 20;

// 或者使用不同的名字空间：
namespace namespace1 {
const unsigned int kMaxMsisdnLen = 20;
}
```

```
namespace namespace2 {
const unsigned int kMaxMsisdnLen = 20;
}
```

建议 9.1.4 禁止用 memcpy_s、memset_s 初始化非 POD 对象

说明：POD 全称是 Plain Old Data，是 C++ 98 标准(ISO/IEC 14882, first edition, 1998-09-01)中引入的一个概念，POD 类型主要包括 int, char, float，double，enumeration，void，指针等原始类型以及聚合类型，不能使用封装和面向对象特性（如用户定义的构造/赋值/析构函数、基类、虚函数等）。

由于非 POD 类型比如非聚合类型的 class 对象，可能存在虚函数，内存布局不确定，跟编译器有关，滥用内存拷贝可能会导致严重的问题。

即使对聚合类型的 class，使用直接的内存拷贝和比较，破坏了信息隐蔽和数据保护的作用，也不提倡 memcpy_s、memset_s 操作。

对于 POD 类型的详细说明请参见附录。

建议 9.1.5 变量使用时才声明并初始化

说明：变量在使用前未赋初值，是常见的低级编程错误。使用前才声明变量并同时初始化，非常方便地避免了此类低级错误。

在函数开始位置声明所有变量，后面才使用变量，作用域覆盖整个函数实现，容易导致如下问题：* 程序难以理解和维护：变量的定义与使用分离。* 变量难以合理初始化：在函数开始时，经常没有足够的信息进行变量初始化，往往用某个默认的空值(比如零)来初始化，这通常是一种浪费，如果变量在被赋于有效值以前使用，还会导致错误。

遵循变量作用域最小化原则与就近声明原则， 使得代码更容易阅读,方便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。

```
// 不好的例子：声明与初始化分离
string name;         // 声明时未初始化：调用缺省构造函数
name = "zhangsan";   // 再次调用赋值操作符函数；声明与定义在不同的地方，理解相对困难

// 好的例子：声明与初始化一体，理解相对容易
string name("zhangsan");  // 调用构造函数
```

表达式

规则 9.2.1 含有变量自增或自减运算的表达式中禁止再次引用该变量

含有变量自增或自减运算的表达式中，如果再引用该变量，其结果在 C++标准中未明确定义。各个编译器或者同一个编译器不同版本实现可能会不一致。 为了更好的可移植性，不应该对标准未定义的运算次序做任何假设。

注意，运算次序的问题不能使用括号来解决，因为这不是优先级的问题。

示例：

```
x = b[i] + i++; // Bad: b[i]运算跟 i++，先后顺序并不明确。
```

正确的写法是将自增或自减运算单独放一行：

```
x = b[i] + i;
i++;            // Good: 单独一行
```

函数参数

```
Func(i++, i);   // Bad: 传递第 2 个参数时，不确定自增运算有没有发生
```

正确的写法

```
i++;            // Good: 单独一行
x = Func(i, i);
```

规则 9.2.2 switch 语句要有 default 分支

大部分情况下，switch 语句中要有 default 分支，保证在遗漏 case 标签处理时能够有一个缺省的处理行为。

特例： 如果 switch 条件变量是枚举类型，并且 case 分支覆盖了所有取值，则加上 default 分支处理有些多余。 现代编译器都具备检查是否在 switch 语句中遗漏了某些枚举值的 case 分支的能力，会有相应的 warning 提示。

```
enum Color {
  kRed = 0,
  kBlue
};

// 因为 switch 条件变量是枚举值，这里可以不用加 default 处理分支
switch (color) {
  case kRed:
    DoRedThing();
    break;
  case kBlue:
    DoBlueThing();
```

```
    ...
    break;
}
```

建议 9.2.1 表达式的比较，应当遵循左侧倾向于变化、右侧倾向于不变的原则

当变量与常量比较时，如果常量放左边，如 if (MAX == v) 不符合阅读习惯，而 if (MAX > v) 更是难于理解。 应当按人的正常阅读、表达习惯，将常量放右边。写成如下方式：

```
if (value == MAX) {

}

if (value < MAX) {

}
```

也有特殊情况，如：`if (MIN < value && value < MAX)` 用来描述区间时，前半段是常量在左的。

不用担心将 '==' 误写成 '='，因为 `if (value = MAX)` 会有编译告警，其他静态检查工具也会报错。让工具去解决笔误问题，代码要符合可读性第一。

建议 9.2.2 使用括号明确操作符的优先级

使用括号明确操作符的优先级，防止因默认的优先级与设计思想不符而导致程序出错；同时使得代码更为清晰可读，然而过多的括号会分散代码使其降低了可读性。下面是如何使用括号的建议。

- 二元及以上操作符, 如果涉及多种操作符，则应该使用括号

```
x = a + b + c;          /* 操作符相同，可以不加括号 */
x = Foo(a + b, c);      /* 逗号两边的表达式，不需要括号 */
x = 1 << (2 + 3);       /* 操作符不同，需要括号 */
x = a + (b / 5);        /* 操作符不同，需要括号 */
x = (a == b) ? a : (a - b);     /* 操作符不同，需要括号 */
```

类型转换

避免使用类型分支来定制行为：类型分支来定制行为容易出错，是企图用 C++编写 C 代码的明显标志。这是一种很不灵活的技术，要添加新类型时，如果忘记修改所有分支，编译器也不会告知。使用模板和虚函数，让类型自己而不是调用它们的代码来决定行为。

建议避免类型转换，我们在代码的类型设计上应该考虑到每种数据的数据类型是什么，而不是应该过度使用类型转换来解决问题。在设计某个基本类型的时候，请考虑： - 是无符号还是有符号的 - 是适合 float 还是 double - 是使用 int8，int16，int32 还是 int64，确定整形的长度

但是我们无法禁止使用类型转换，因为 C++语言是一门面向机器编程的语言，涉及到指针地址，并且我们会与各种第三方或者底层 API 交互，他们的类型设计不一定是合理的，在这个适配的过程中很容易出现类型转换。

例外：在调用某个函数的时候，如果我们不想处理函数结果，首先要考虑这个是否是你的最好的选择。如果确实不想处理函数的返回值，那么可以使用(void)转换来解决。

规则 9.3.1 如果确定要使用类型转换，请使用有 C++提供的类型转换，而不是 C 风格的类型转换

说明：

C++提供的类型转换操作比 C 风格更有针对性，更易读，也更加安全，C++提供的转换有： - 类型转换： 1. `dynamic_cast`：主要用于继承体系下行转换，`dynamic_cast` 具有类型检查的功能，请做好基类和派生类的设计，避免使用 dynamic_cast 来进行转换。 2. `static_cast`：和 C 风格转换相似可做值的强制转换，或上行转换(把派生类的指针或引用转换成基类的指针或引用)。该转换经常用于消除多重继承带来的类型歧义，是相对安全的。如果是纯粹的算数转换，那么请使用后面的大括号转换方式。 3. `reinterpret_cast`：用于转换不相关的类型。`reinterpret_cast` 强制编译器将某个类型对象的内存重新解释成另一种类型，这是一种不安全的转换，建议尽可能少用 `reinterpret_cast`。 4. `const_cast`：用于移除对象的 `const` 属性，使对象变得可修改，这样会破坏数据的不变性，建议尽可能少用。

- 算数转换：（C++11 开始支持）对于那种算数转换，并且类型信息没有丢失的，比如 float 到 double， int32 到 int64 的转换，推荐使用大括号的初始方式。

```
double d{ someFloat };
int64_t i{ someInt32 };
```

建议 9.3.1 避免使用 `dynamic_cast`

1. `dynamic_cast` 依赖于 C++的 RTTI， 让程序员在运行时识别 C++类对象的类型。
2. `dynamic_cast` 的出现一般说明我们的基类和派生类设计出现了问题，派生类破坏了基类的契约，不得不通过 `dynamic_cast` 转换到子类进行特殊处理，这个时候更希望来改善类的设计，而不是通过 `dynamic_cast` 来解决问题。

建议 9.3.2 避免使用 reinterpret_cast

说明：reinterpret_cast 用于转换不相关类型。尝试用 reinterpret_cast 将一种类型强制转换另一种类型，这破坏了类型的安全性与可靠性，是一种不安全的转换。不同类型之间尽量避免转换。

建议 9.3.3 避免使用 const_cast

说明：const_cast 用于移除对象的 const 和 volatile 性质。

使用 const_cast 转换后的指针或者引用来修改 const 对象，行为是未定义的。

```
// 不好的例子
const int i = 1024;
int* p = const_cast<int*>(&i);
*p = 2048;      // 未定义行为

// 不好的例子
class Foo {
 public:
  Foo() : i(3) {}

  void Fun(int v) {
    i = v;
  }

 private:
  int i;
};

int main(void) {
  const Foo f;
  Foo* p = const_cast<Foo*>(&f);
  p->Fun(8);  // 未定义行为
}
```

资源分配和释放

规则 9.4.1 单个对象释放使用 delete，数组对象释放使用 delete []

说明：单个对象删除使用 delete， 数组对象删除使用 delete []，原因：

- 调用 new 所包含的动作：从系统中申请一块内存，并调用此类型的构造函数。
- 调用 new[n]所包含的动作：申请可容纳 n 个对象的内存，并且对每一个对象调用其构造函数。
- 调用 delete 所包含的动作：先调用相应的析构函数，再将内存归还系统。
- 调用 delete[]所包含的动作：对每一个对象调用析构函数，再释放所有内存

如果 new 和 delete 的格式不匹配，结果是未知的。对于非 class 类型， new 和 delete 不会调用构造与析构函数。

错误写法：

```
const int KMaxArraySize = 100;
int* numberArray = new int[KMaxArraySize];
...
delete numberArray;
numberArray = NULL;
```

正确写法：

```
const int KMaxArraySize = 100;
int* numberArray = new int[KMaxArraySize];
...
delete[] numberArray;
numberArray = NULL;
```

建议 9.4.1 使用 RAII 特性来帮助追踪动态分配

说明：RAII 是“资源获取就是初始化”的缩语(Resource Acquisition Is Initialization)，是一种利用对象生命周期来控制程序资源(如内存、文件句柄、网络连接、互斥量等等)的简单技术。

RAII 的一般做法是这样的：在对象构造时获取资源，接着控制对资源的访问使之在对象的生命周期内始终保持有效，最后在对象析构的时候释放资源。这种做法有两大好处： - 我们不需要显式地释放资源。 - 对象所需的资源在其生命期内始终保持有效。这样，就不必检查资源有效性的问题，可以简化逻辑、提高效率。

示例：使用 RAII 不需要显式地释放互斥资源。

```
class LockGuard {
 public:
  LockGuard(const LockType& lockType): lock(lockType) {
    lock.Aquire();
  }

  ~LockGuard() {
    lock.Relase();
  }

 private:
  LockType lock;
};


bool Update() {
  LockGuard lockGuard(mutex);
```

```
  if (...) {
    return false;
  } else {
    // 操作数据
  }

  return true;
}
```

标准库

STL 标准模板库在不同模块使用程度不同，这里列出一些基本规则和建议。

规则 9.5.1 不要保存 std::string 的 c_str()返回的指针

说明：在 C++标准中并未规定 string::c_str()指针持久有效，因此特定 STL 实现完全可以在调用 string::c_str()时返回一个临时存储区并很快释放。所以为了保证程序的可移植性，不要保存 string::c_str()的结果，而是在每次需要时直接调用。

示例：

```
void Fun1() {
  std::string name = "demo";
  const char* text = name.c_str();  // 表达式结束以后，name 的生命周期还在，指针有效

  // 如果中间调用了string 的非const 成员函数，导致string 被修改，比如operator[], begin()等
  // 可能会导致text 的内容不可用，或者不是原来的字符串
  name = "test";
  name[1] = '2';

  // 后续使用text 指针，其字符串内容不再是"demo"
}

void Fun2() {
  std::string name = "demo";
  std::string test = "test";
  const char* text = (name + test).c_str(); // 表达式结束以后，+号产生的临时对象被销毁，指针无效

    // 后续使用text 指针，其已不再指向合法内存空间
}
```

例外：在少数对性能要求非常高的代码中，为了适配已有的只接受 const char*类型入参的函数，可以临时保存 string::c_str()返回的指针。但是必须严格保证 string

对象的生命周期长于所保存指针的生命周期，并且保证在所保存指针的生命周期内，string 对象不会被修改。

建议 9.5.1 使用 std::string 代替 char*

说明：使用 string 代替 char*有很多优势，比如： 1. 不用考虑结尾的'\0'； 2. 可以直接使用+, =, ==等运算符以及其它字符串操作函数； 3. 不需要考虑内存分配操作，避免了显式的 new/delete，以及由此导致的错误；

需要注意的是某些 stl 实现中 string 是基于写时复制策略的，这会带来 2 个问题，一是某些版本的写时复制策略没有实现线程安全，在多线程环境下会引起程序崩溃；二是当与动态链接库相互传递基于写时复制策略的 string 时，由于引用计数在动态链接库被卸载时无法减少可能导致悬挂指针。因此，慎重选择一个可靠的 stl 实现对于保证程序稳定是很重要的。

例外： 当调用系统或者其它第三方库的 API 时，针对已经定义好的接口，只能使用 char*。但是在调用接口之前都可以使用 string，在调用接口时使用 string::c_str()获得字符指针。 当在栈上分配字符数组当作缓冲区使用时，可以直接定义字符数组，不要使用 string，也没有必要使用类似 vector<char>等容器。

规则 9.5.2 禁止使用 auto_ptr

说明：在 stl 库中的 std::auto_ptr 具有一个隐式的所有权转移行为，如下代码：

```
auto_ptr<T> p1(new T);
auto_ptr<T> p2 = p1;
```

当执行完第 2 行语句后，p1 已经不再指向第 1 行中分配的对象，而是变为 NULL。正因为如此，auto_ptr 不能被置于各种标准容器中。 转移所有权的行为通常不是期望的结果。对于必须转移所有权的场景，也不应该使用隐式转移的方式。这往往需要程序员对使用 auto_ptr 的代码保持额外的谨慎，否则出现对空指针的访问。使用 auto_ptr 常见的有两种场景，一是作为智能指针传递到产生 auto_ptr 的函数外部，二是使用 auto_ptr 作为 RAII 管理类，在超出 auto_ptr 的生命周期时自动释放资源。 对于第 1 种场景，可以使用 std::shared_ptr 来代替。 对于第 2 种场景，可以使用 C++11 标准中的 std::unique_ptr 来代替。其中 std::unique_ptr 是 std::auto_ptr 的代替品，支持显式的所有权转移。

例外： 在 C++11 标准得到普遍使用之前，在一定需要对所有权进行转移的场景下，可以使用 std::auto_ptr，但是建议对 std::auto_ptr 进行封装，并禁用封装类的拷贝构造函数和赋值运算符，以使该封装类无法用于标准容器。

建议 9.5.2 使用新的标准头文件

说明： 使用 C++的标准头文件时，请使用<cstdlib>这样的，而不是<stdlib.h>这种的。

const 的用法

在声明的变量或参数前加上关键字 const 用于指明变量值不可被篡改 (如 `const int foo`). 为类中的函数加上 const 限定符表明该函数不会修改类成员变量的状态 (如 `class Foo { int Bar(char c) const; };`)。 const 变量, 数据成员, 函数和参数为编译时类型检测增加了一层保障， 便于尽早发现错误。因此, 我们强烈建议在任何可能的情况下使用 const。 有时候， 使用 C++11 的 constexpr 来定义真正的常量可能更好。

规则 9.6.1 对于指针和引用类型的形参，如果是不需要修改的，请使用 const

不变的值更易于理解/跟踪和分析，把 const 作为默认选项，在编译时会对其进行检查，使代码更牢固/更安全。

```
class Foo;

void PrintFoo(const Foo& foo);
```

规则 9.6.2 对于不会修改成员变量的成员函数请使用 const 修饰

尽可能将成员函数声明为 const。 访问函数应该总是 const。只要不修改数据成员的成员函数，都声明为 const。

```
class Foo {
 public:

  // ...

  int PrintValue() const { // const 修饰成员函数，不会修改成员变量
    std::cout << value << std::endl;
  }

  int GetValue() const { // const 修饰成员函数，不会修改成员变量
    return value;
  }

 private:
  int value;
};
```

建议 9.6.1 初始化后不会再修改的成员变量定义为 const

```
class Foo {
 public:
  Foo(int length) : dataLength(length) {}
 private:
```

```
  const int dataLength;
};
```

异常

建议 9.7.1 C++11 中，如果函数不会抛出异常，声明为 noexcept

理由 1. 如果函数不会抛出异常，声明为 noexcept 可以让编译器最大程度的优化函数，如减少执行路径，提高错误退出的效率。 2. vector 等 STL 容器，为了保证接口的健壮性，如果保存元素的 move 运算符没有声明为 noexcept，则在容器扩张搬移元素时不会使用 move 机制，而使用 copy 机制，带来性能损失的风险。如果一个函数不能抛出异常，或者一个程序并没有截获某个函数所抛出的异常并进行处理，那么这个函数可以用新的 noexcept 关键字对其进行修饰，表示这个函数不会抛出异常或者抛出的异常不会被截获并处理。例如：

```
extern "C" double sqrt(double) noexcept;  // 永远不会抛出异常

// 即使可能抛出异常，也可以使用 noexcept
// 这里不准备处理内存耗尽的异常，简单地将函数声明为noexcept
std::vector<int> MyComputation(const std::vector<int>& v) noexcept {
  std::vector<int> res = v;    // 可能会抛出异常
  // do something
  return res;
}
```

示例

```
RetType Function(Type params) noexcept;   // 最大的优化
RetType Function(Type params);            // 更少的优化

// std::vector 的 move 操作需要声明 noexcept
class Foo1 {
 public:
  Foo1(Foo1&& other);  // no noexcept
};

std::vector<Foo1> a1;
a1.push_back(Foo1());
a1.push_back(Foo1());  // 触发容器扩张，搬移已有元素时调用copy constructo
r

class Foo2 {
 public:
  Foo2(Foo2&& other) noexcept;
};

std::vector<Foo2> a2;
```

```
a2.push_back(Foo2());
a2.push_back(Foo2());  // 触发容器扩张，搬移已有元素时调用move constructo
r
```

注意 默认构造函数、析构函数、swap 函数，move 操作符都不应该抛出异常。

模板

模板能够实现非常灵活简洁的类型安全的接口，实现类型不同但是行为相同的代码复用。

模板编程的缺点：

1. 模板编程所使用的技巧对于使用 c++不是很熟练的人是比较晦涩难懂的。在复杂的地方使用模板的代码让人更不容易读懂，并且 debug 和维护起来都很麻烦。
2. 模板编程经常会导致编译出错的信息非常不友好: 在代码出错的时候, 即使这个接口非常的简单, 模板内部复杂的实现细节也会在出错信息显示. 导致这个编译出错信息看起来非常难以理解。
3. 模板如果使用不当，会导致运行时代码过度膨胀。
4. 模板代码难以修改和重构。模板的代码会在很多上下文里面扩展开来, 所以很难确认重构对所有的这些展开的代码有用。

所以， 建议__模板编程最好只用在少量的基础组件，基础数据结构上面__。并且使用模板编程的时候尽可能把__复杂度最小化__，尽量__不要让模板对外暴露__。最好只在实现里面使用模板, 然后给用户暴露的接口里面并不使用模板, 这样能提高你的接口的可读性。 并且你应该在这些使用模板的代码上写尽可能详细的注释。

宏

在 C++语言中，我们强烈建议尽可能少使用复杂的宏 - 对于常量定义，请按照前面章节所述，使用 const 或者枚举； - 对于宏函数，尽可能简单，并且遵循下面的原则，并且优先使用内联函数，模板函数等进行替换。

```
// 不推荐使用宏函数
#define SQUARE(a, b) ((a) * (b))

// 请使用模板函数，内联函数等来替换。
template<typename T> T Square(T a, T b) { return a * b; }
```

如果需要使用宏，请参考 C 语言规范的相关章节。 **例外**：一些通用且成熟的应用，如：对 new, delete 的封装处理，可以保留对宏的使用。

10 现代 C++特性

随着 ISO 在 2011 年发布 C++11 语言标准，以及 2017 年 3 月发布 C++17，现代 C++(C++11/14/17 等)增加了大量提高编程效率、代码质量的新语言特性和标准库。本章节描述了一些可以帮助团队更有效率的使用现代 C++，规避语言陷阱的指导意见。

代码简洁性和安全性提升

建议 10.1.1 合理使用 auto

理由

- auto 可以避免编写冗长、重复的类型名，也可以保证定义变量时初始化。
- auto 类型推导规则复杂，需要仔细理解。
- 如果能够使代码更清晰，继续使用明确的类型，且只在局部变量使用 auto。

示例

```
// 避免冗长的类型名
std::map<string, int>::iterator iter = m.find(val);
auto iter = m.find(val);

// 避免重复类型名
class Foo {...};
Foo* p = new Foo;
auto p = new Foo;

// 保证初始化
int x;    // 编译正确，没有初始化
auto x;   // 编译失败，必须初始化
```

auto 的类型推导可能导致困惑：

```
auto a = 3;           // int
const auto ca = a;    // const int
const auto& ra = a;   // const int&
auto aa = ca;         // int, 忽略 const 和 reference
auto ila1 = { 10 };   // std::initializer_list<int>
auto ila2{ 10 };      // std::initializer_list<int>

auto&& ura1 = x;      // int&
auto&& ura2 = ca;     // const int&
auto&& ura3 = 10;     // int&&

const int b[10];
```

```
auto arr1 = b;         // const int*
auto& arr2 = b;        // const int(&)[10]
```

如果没有注意 auto 类型推导时忽略引用，可能引入难以发现的性能问题:

```
std::vector<std::string> v;
auto s1 = v[0];  // auto 推导为 std::string，拷贝 v[0]
```

如果使用 auto 定义接口，如头文件中的常量，可能因为开发人员修改了值，而导致类型发生变化。

规则 10.1.1 在重写虚函数时请使用 override 关键字

理由 override 关键字保证函数是虚函数，且重写了基类的虚函数。如果子类函数与基类函数原型不一致，则产生编译告警。

如果修改了基类虚函数原型，但忘记修改子类重写的虚函数，在编译期就可以发现。也可以避免有多个子类时，重写函数的修改遗漏。

示例

```
class Base {
 public:
  virtual void Foo();
  void Bar();
};

class Derived : public Base {
 public:
  void Foo() const override; // 编译失败: derived::Foo 和 base::Foo 原型不一致，不是重写
  void Foo() override;       // 正确: derived::Foo 重写 base::Foo
  void Bar() override;       // 编译失败: base::Bar 不是虚函数
};
```

总结 1. 基类首次定义虚函数，使用 virtual 关键字 2. 子类重写基类虚函数，使用 override 关键字 3. 非虚函数，virtual 和 override 都不使用

规则 10.1.2 使用 delete 关键字删除函数

理由 相比于将类成员函数声明为 private 但不实现，delete 关键字更明确，且适用范围更广。

示例

```
class Foo {
 private:
  // 只看头文件不知道拷贝构造是否被删除
  Foo(const Foo&);
```

```
};

class Foo {
 public:
  // 明确删除拷贝赋值函数
  Foo& operator=(const Foo&) = delete;
};
```

delete 关键字还支持删除非成员函数

```
template<typename T>
void Process(T value);

template<>
void Process<void>(void) = delete;
```

规则 10.1.3 使用 nullptr，而不是 NULL 或 0

理由 长期以来，C++没有一个代表空指针的关键字，这是一件很尴尬的事：

```
#define NULL ((void *)0)

char* str = NULL;   // 错误: void* 不能自动转换为 char*

void(C::*pmf)() = &C::Func;
if (pmf == NULL) {} // 错误: void* 不能自动转换为指向成员函数的指针
```

如果把 NULL 被定义为 0 或 0L。可以解决上面的问题。

或者在需要空指针的地方直接使用 0。但这引入另一个问题，代码不清晰，特别是使用 auto 自动推导：

```
auto result = Find(id);
if (result == 0) {  // Find() 返回的是 指针 还是 整数?
  // do something
}
```

0 字面上是 int 类型(0L 是 long)，所以 NULL 和 0 都不是指针类型。 当重载指针和整数类型的函数时，传递 NULL 或 0 都调用到整数类型重载的函数:

```
void F(int);
void F(int*);

F(0);       // 调用 F(int)，而非 F(int*)
F(NULL);    // 调用 F(int)，而非 F(int*)
```

另外，sizeof(NULL) == sizeof(void*)并不一定总是成立的，这也是一个潜在的风险。

总结： 直接使用 0 或 0L，代码不清晰，且无法做到类型安全；使用 NULL 无法做到类型安全。这些都是潜在的风险。

nullptr 的优势不仅仅是在字面上代表了空指针，使代码清晰，而且它不再是一个整数类型。

nullptr 是 std::nullptr_t 类型，而 std::nullptr_t 可以隐式的转换为所有的原始指针类型，这使得 nullptr 可以表现成指向任意类型的空指针。

```
void F(int);
void F(int*);
F(nullptr);   // 调用 F(int*)

auto result = Find(id);
if (result == nullptr) {  // Find() 返回的是 指针
  // do something
}
```

建议 10.1.2 使用 using 而非 typedef

在 C++11 之前，可以通过 typedef 定义类型的别名。没人愿意多次重复 std::map<uint32_t, std::vector<int>>这样的代码。

```
typedef std::map<uint32_t, std::vector<int>> SomeType;
```

类型的别名实际是对类型的封装。而通过封装，可以让代码更清晰，同时在很大程度上避免类型变化带来的散弹式修改。 在 C++11 之后，提供 using，实现声明别名 (alias declarations):

```
using SomeType = std::map<uint32_t, std::vector<int>>;
```

对比两者的格式：

```
typedef Type Alias;   // Type 在前，还是 Alias 在前
using Alias = Type;   // 符合'赋值'的用法，容易理解，不易出错
```

如果觉得这点还不足以切换到 using，我们接着看看模板别名(alias template):

```
// 定义模板的别名，一行代码
template<class T>
using MyAllocatorVector = std::vector<T, MyAllocator<T>>;

MyAllocatorVector<int> data;       // 使用 using 定义的别名

template<class T>
class MyClass {
 private:
  MyAllocatorVector<int> data_;   // 模板类中使用 using 定义的别名
};
```

而 typedef 不支持带模板参数的别名，只能“曲线救国”:

```
// 通过模板包装 typedef，需要实现一个模板类
template<class T>
struct MyAllocatorVector {
  typedef std::vector<T, MyAllocator<T>> type;
};

MyAllocatorVector<int>::type data;  // 使用 typedef 定义的别名，多写 ::type

template<class T>
class MyClass {
 private:
  typename MyAllocatorVector<int>::type data_;  // 模板类中使用，除了 ::type，还需要加上 typename
};
```

规则 10.1.4 禁止使用 std::move 操作 const 对象

从字面上看，std::move 的意思是要移动一个对象。而 const 对象是不允许修改的，自然也无法移动。因此用 std::move 操作 const 对象会给代码阅读者带来困惑。
在实际功能上，std::move 会把对象转换成右值引用类型；对于 const 对象，会将其转换成 const 的右值引用。由于极少有类型会定义以 const 右值引用为参数的移动构造函数和移动赋值操作符，因此代码实际功能往往退化成了对象拷贝而不是对象移动，带来了性能上的损失。

错误示例：

```
std::string gString;
std::vector<std::string> gStringList;

void func() {
  const std::string myString = "String content";
  gString = std::move(myString); // bad:并没有移动myString，而是进行了复制
  const std::string anotherString = "Another string content";
  gStringList.push_back(std::move(anotherString));    // bad:并没有移动anotherString，而是进行了复制
}
```

智能指针

建议 10.2.1 优先使用智能指针而不是原始指针管理资源

理由 避免资源泄露。

示例

```
void Use(int i) {
  auto p = new int {7};                // 不好: 通过 new 初始化局部指针
  auto q = std::make_unique<int>(9);   // 好: 保证释放内存
  if (i > 0) {
    return;                            // 可能 return，导致内存泄露
  }
  delete p;                            // 太晚了
}
```

例外 在性能敏感、兼容性等场景可以使用原始指针。

规则 10.2.1 优先使用 unique_ptr 而不是 shared_ptr

理由 1. shared_ptr 引用计数的原子操作存在可测量的开销，大量使用 shared_ptr 影响性能。 2. 共享所有权在某些情况（如循环依赖）可能导致对象永远得不到释放。 3. 相比于谨慎设计所有权，共享所有权是一种诱人的替代方案，但它可能使系统变得混乱。

规则 10.2.2 使用 std::make_unique 而不是 new 创建 unique_ptr

理由 1. make_unique 提供了更简洁的创建方式 2. 保证了复杂表达式的异常安全

示例

```
// 不好：两次出现 MyClass，重复导致不一致风险
std::unique_ptr<MyClass> ptr(new MyClass(0, 1));
// 好：只出现一次 MyClass，不存在不一致的可能
auto ptr = std::make_unique<MyClass>(0, 1);
```

重复出现类型可能导致非常严重的问题，且很难发现：

```
// 编译正确，但new 和delete 不配套
std::unique_ptr<uint8_t> ptr(new uint8_t[10]);
std::unique_ptr<uint8_t[]> ptr(new uint8_t);
// 非异常安全: 编译器可能按如下顺序计算参数:
// 1. 分配 Foo 的内存,
// 2. 构造 Foo,
// 3. 调用 Bar,
// 4. 构造 unique_ptr<Foo>.
// 如果 Bar 抛出异常, Foo 不会被销毁，产生内存泄露。
F(unique_ptr<Foo>(new Foo()), Bar());

// 异常安全: 调用函数不会被打断.
F(make_unique<Foo>(), Bar());
```

例外 std::make_unique 不支持自定义 deleter。 在需要自定义 deleter 的场景，建议在自己的命名空间实现定制版本的 make_unique。 使用 new 创建自定义 deleter 的 unique_ptr 是最后的选择。

规则 10.2.3 使用 std::make_shared 而不是 new 创建 shared_ptr

理由 使用 std::make_shared 除了类似 std::make_unique 一致性等原因外，还有性能的因素。 std::shared_ptr 管理两个实体： * 控制块(存储引用计数，deleter 等) * 管理对象

std::make_shared 创建 std::shared_ptr，会一次性在堆上分配足够容纳控制块和管理对象的内存。而使用 std::shared_ptr<MyClass>(new MyClass)创建 std::shared_ptr，除了 new MyClass 会触发一次堆分配外，std::shard_ptr 的构造函数还会触发第二次堆分配，产生额外的开销。

例外 类似 std::make_unique，std::make_shared 不支持定制 deleter

Lambda

建议 10.3.1 当函数不能工作时选择使用 lambda(捕获局部变量，或编写局部函数)

理由 函数无法捕获局部变量或在局部范围内声明；如果需要这些东西，尽可能选择 lambda，而不是手写的 functor。 另一方面，lambda 和 functor 不会重载；如果需要重载，则使用函数。 如果 lambda 和函数都可以的场景，则优先使用函数；尽可能使用最简单的工具。

示例

```
// 编写一个只接受 int 或 string 的函数
// -- 重载是自然的选择
void F(int);
void F(const string&);

// 需要捕获局部状态，或出现在语句或表达式范围
// -- Lambda 是自然的选择
vector<Work> v = LotsOfWork();
for (int taskNum = 0; taskNum < max; ++taskNum) {
  pool.Run([=, &v] {...});
}
pool.Join();
```

规则 10.3.1 非局部范围使用 lambdas，避免使用按引用捕获

理由 非局部范围使用 lambdas 包括返回值，存储在堆上，或者传递给其它线程。局部的指针和引用不应该在它们的范围外存在。lambdas 按引用捕获就是把局部对象的引用存储起来。如果这会导致超过局部变量生命周期的引用存在，则不应该按引用捕获。

示例

```
// 不好
void Foo() {
  int local = 42;
  // 按引用捕获 local.
  // 当函数返回后，local 不再存在，
  // 因此 Process() 的行为未定义!
  threadPool.QueueWork([&]{ Process(local); });
}

// 好
void Foo() {
  int local = 42;
  // 按值捕获 local。
  // 因为拷贝，Process() 调用过程中，local 总是有效的
  threadPool.QueueWork([=]{ Process(local); });
}
```

建议 10.3.2 如果捕获 this，则显式捕获所有变量

理由 在成员函数中的[=]看起来是按值捕获。但因为是隐式的按值获取了 this 指针，并能够操作所有成员变量，数据成员实际是按引用捕获的，一般情况下建议避免。如果的确需要这样做，明确写出对 this 的捕获。

示例

```
class MyClass {
 public:
  void Foo() {
    int i = 0;

    auto Lambda = [=]() { Use(i, data_); };    // 不好: 看起来像是拷贝/按值捕获，成员变量实际上是按引用捕获

    data_ = 42;
    Lambda(); // 调用 use(42);
    data_ = 43;
    Lambda(); // 调用 use(43);

    auto Lambda2 = [i, this]() { Use(i, data_); }; // 好，显式指定按值捕获，最明确，最少的混淆
  }

 private:
  int data_ = 0;
};
```

建议 10.3.3 避免使用默认捕获模式

理由 lambda 表达式提供了两种默认捕获模式：按引用（&）和按值（=）。 默认按引用捕获会隐式的捕获所有局部变量的引用，容易导致访问悬空引用。相比之下，显式的写出需要捕获的变量可以更容易的检查对象生命周期，减小犯错可能。 默认按值捕获会隐式的捕获 this 指针，且难以看出 lambda 函数所依赖的变量是哪些。如果存在静态变量，还会让阅读者误以为 lambda 拷贝了一份静态变量。 因此，通常应当明确写出 lambda 需要捕获的变量，而不是使用默认捕获模式。

错误示例

```
auto func() {
  int addend = 5;
  static int baseValue = 3;

  return [=]() {     // 实际上只复制了 addend
    ++baseValue;     // 修改会影响静态变量的值
    return baseValue + addend;
  };
}
```

正确示例

```
auto func() {
  int addend = 5;
  static int baseValue = 3;

  return [addend, baseValue = baseValue]() mutable {  // 使用 C++14 的捕获初始化拷贝一份变量
    ++baseValue;     // 修改自己的拷贝，不会影响静态变量的值
    return baseValue + addend;
  };
}
```

参考：《Effective Modern C++》： Item 31: Avoid default capture modes.

接口

建议 10.4.1 不涉及所有权的场景，使用 T*或 T&作为参数，而不是智能指针

理由 1. 只在需要明确所有权机制时，才通过智能指针转移或共享所有权. 2. 通过智能指针传递，限制了函数调用者必须使用智能指针(如调用者希望传递 `this`)。 3. 传递共享所有权的智能指针存在运行时的开销。

示例

```
// 接受任何 int*
void F(int*);

// 只能接受希望转移所有权的 int
void G(unique_ptr<int>);

// 只能接受希望共享所有权的 int
void G(shared_ptr<int>);

// 不改变所有权，但需要特定所有权的调用者
void H(const unique_ptr<int>&);

// 接受任何 int
void H(int&);

// 不好
void F(shared_ptr<Widget>& w) {
  // ...
  Use(*w); // 只使用 w -- 完全不涉及生命周期管理
  // ...
};
```

建议 10.4.2 在接口层面明确指针不会为 nullptr

理由 1. 避免解引用空指针的错误。 2. 避免重复检查空指针，提高代码效率。

建议使用 gsl::not_null，或参考实现自己的版本（如使用 NullObject 模式）。对于集合，在不违反已有的接口约定的情况下，建议返回空集合而避免返回空指针，当返回字符串时，建议返回空串""。

示例

```
int Length(const char* p);     // 不清楚 length == nullptr 是否是有效的
Length(nullptr);               // 可以吗?

int Length(not_null<const char*> p);  // 更好：可以认为 p 不会是空指针
int Length(const char* p);            // 必须假设 p 可能是空指针
```

通过在代码中表明意图(指针不能为空)，工具可以提供更好的诊断，如通过静态分析找到一些错误。也可以实现代码优化，如移除判空的测试和分支。

注意 not_null 由 guideline support library(gsl)提供。