????

- □□□□□□□□
- C & C++ □□□□

????????

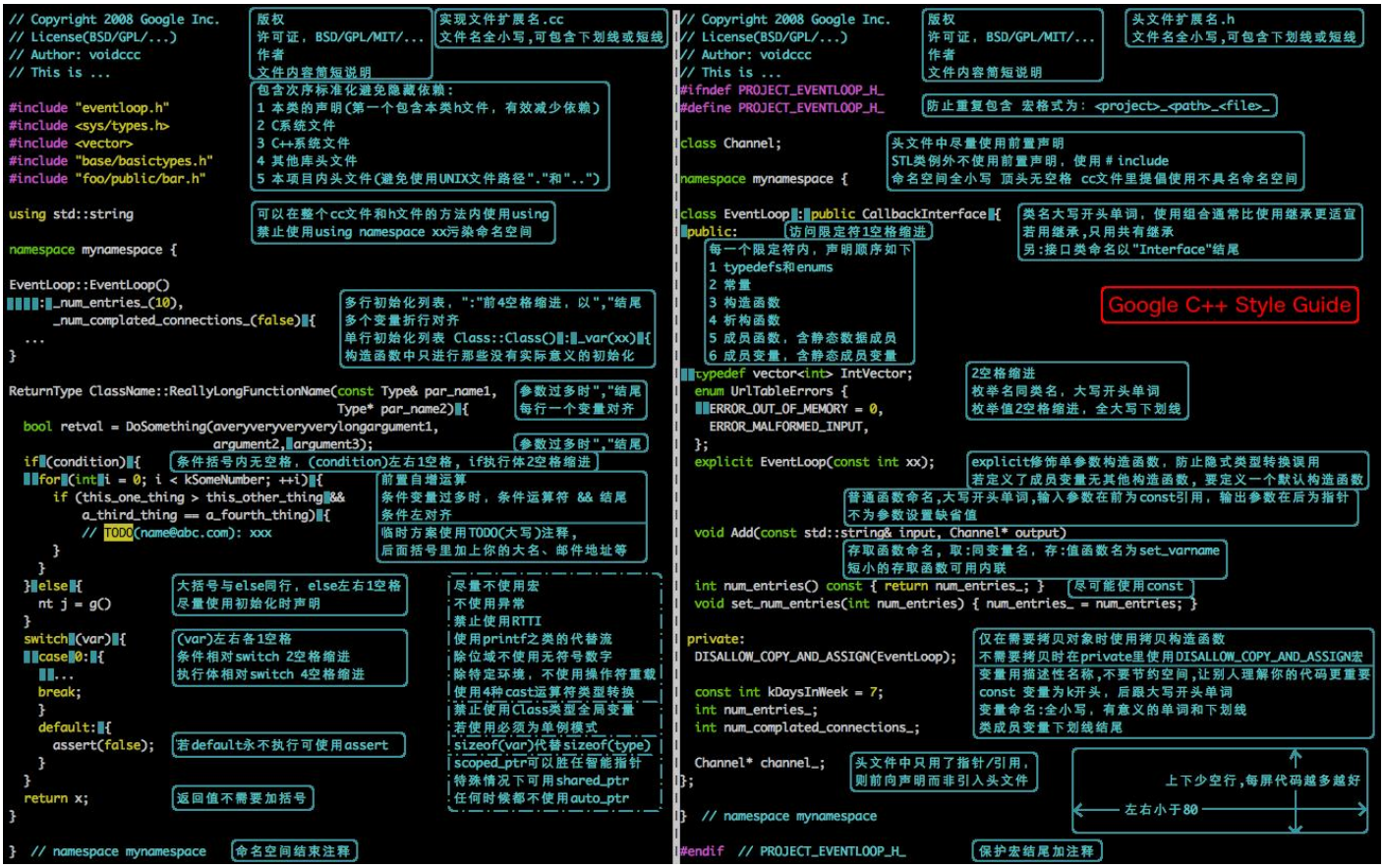
vscode ????

```
{
// Place your snippets for cpp here. Each snippet is defined under a snippet name and has a prefix, body and
// description. The prefix is what is used to trigger the snippet and the body will be expanded and inserted. Possible variables are:
// $1, $2 for tab stops, $0 for the final cursor position, and ${1:label}, ${2:another} for placeholders. Placeholders with the
// same ids are connected.
// Example:
"copyright info":{
        "prefix": "//file",
        "body": [
            "//-------------------------------------------------------------------------------
",
            "//! Copyright(C) 2020-2022, netflt.com",
            "//! All rights reserved.",
            "// ",
            "// name: $TM_FILENAME",
            "// author: Danny.Zhu"
            "// date: $CURRENT_YEAR-$CURRENT_MONTH-$CURRENT_DATE $CURRENT_HOUR:$CURRENT_MINUTE:$CURRENT_SECOND",
            "//-------------------------------------------------------------------------------"
        ],
        "description": "add copyright information"
    },
    "simple comment":{
        "prefix": "//",
        "body": [
            "//<! ${1}"
        ],
        "description": "add simple comment"
    },
```

```
    "block comment":{
        "prefix": "///",
        "body": [
            "/***",
            "//<! ${1}",
            "***/"
        ],
        "description": "add block comments"
    },
  "TODO": {
    "prefix": "//td",
    "body": [
      "//<! TODO (Danny.Zhu): $1"
    ],
    "description": "Log output to console"
  }

}
```

clang-format ??



.clang-format

```
---
Language:        Cpp
# BasedOnStyle:  Chromium
SortIncludes:    true
AccessModifierOffset: -4
DerivePointerAlignment: false
PointerAlignment: Left
ConstructorInitializerIndentWidth: 4
AlignEscapedNewlines: Left
AlignAfterOpenBracket: Align
AlignConsecutiveMacros: Consecutive
AlignConsecutiveAssignments: None
AlignConsecutiveBitFields: Consecutive
AlignConsecutiveDeclarations: false
AlignOperands:   Align
AlignTrailingComments: true
AllowShortBlocksOnASingleLine: Never
AllowShortFunctionsOnASingleLine: Empty
AllowAllArgumentsOnNextLine: true
AllowAllConstructorInitializersOnNextLine: false
AllowAllParametersOfDeclarationOnNextLine: false
AllowShortEnumsOnASingleLine: false
AllowShortLambdasOnASingleLine: Empty
AllowShortIfStatementsOnASingleLine: Never
AllowShortCaseLabelsOnASingleLine: false
AllowShortLoopsOnASingleLine: false
AlwaysBreakTemplateDeclarations: true
BinPackArguments: false
AlwaysBreakBeforeMultilineStrings: false
BreakBeforeBinaryOperators: NonAssignment
BreakBeforeTernaryOperators: true
BreakConstructorInitializersBeforeComma: false
BinPackParameters: false
ColumnLimit:     96
ConstructorInitializerAllOnOneLineOrOnePerLine: false
DerivePointerBinding: false
ExperimentalAutoDetectBinPacking: false
```

```
IndentCaseLabels: false
MaxEmptyLinesToKeep: 1
NamespaceIndentation: None
ObjCSpaceBeforeProtocolList: true
PenaltyBreakBeforeFirstCallParameter: 1
PenaltyBreakComment: 300
PenaltyBreakString: 1000
PenaltyBreakFirstLessLess: 120
PenaltyExcessCharacter: 1000000
PenaltyReturnTypeOnItsOwnLine: 1000
AlwaysBreakAfterDefinitionReturnType: None
AlwaysBreakAfterReturnType: None
PointerBindsToType: true
Cpp11BracedListStyle: false
Standard:        Auto
IndentWidth:     4
TabWidth:        4
UseTab:          Never
BreakBeforeBraces: Custom
AttributeMacros:
  - __capability
BraceWrapping:
  AfterCaseLabel:  false
  AfterClass:      true
  AfterControlStatement: false
  AfterEnum:       false
  AfterFunction:   true
  AfterNamespace:  true
  AfterObjCDeclaration: true
  AfterStruct:     false
  AfterUnion:      false
  AfterExternBlock: true
  BeforeCatch:     true
  BeforeElse:      true
  BeforeLambdaBody: false
  BeforeWhile:     false
  IndentBraces:    false
  SplitEmptyFunction: false
  SplitEmptyRecord: false
```

```
  SplitEmptyNamespace: false
SpacesInParentheses: false
SpacesInAngles:  false
BreakBeforeConceptDeclarations: true
BreakBeforeInheritanceComma: false
BreakInheritanceList: BeforeComma
BreakConstructorInitializers: BeforeComma
BreakAfterJavaFieldAnnotations: false
BreakStringLiterals: true
CommentPragmas:  '^ IWYU pragma:'
CompactNamespaces: false
ContinuationIndentWidth: 4
DeriveLineEnding: true
DisableFormat:   false
EmptyLineBeforeAccessModifier: LogicalBlock
FixNamespaceComments: true
ForEachMacros:
  - foreach
  - Q_FOREACH
  - BOOST_FOREACH
StatementAttributeLikeMacros:
  - Q_EMIT
IncludeBlocks:   Regroup
IncludeCategories:
  - Regex:           '^<ext/.*\.h>'
    Priority:        2
    SortPriority:    0
    CaseSensitive:   false
  - Regex:           '^<.*\.h>'
    Priority:        1
    SortPriority:    0
    CaseSensitive:   false
  - Regex:           '^<.*'
    Priority:        2
    SortPriority:    0
    CaseSensitive:   false
  - Regex:           '.*'
    Priority:        3
    SortPriority:    0
```

```
    CaseSensitive:   false
IncludeIsMainRegex: '([-_](test|unittest))?$'
IncludeIsMainSourceRegex: ''
IndentCaseBlocks: false
IndentGotoLabels: true
IndentPPDirectives: BeforeHash
IndentExternBlock: AfterExternBlock
IndentRequires:  false
IndentWrappedFunctionNames: true
InsertTrailingCommas: None
JavaScriptQuotes: Leave
JavaScriptWrapImports: true
KeepEmptyLinesAtTheStartOfBlocks: true
MacroBlockBegin: ''
MacroBlockEnd:   ''
ObjCBinPackProtocolList: Auto
ObjCBlockIndentWidth: 2
ObjCBreakBeforeNestedBlockParam: true
ObjCSpaceAfterProperty: false
PenaltyBreakAssignment: 2
PenaltyBreakTemplateDeclaration: 10
PenaltyIndentedWhitespace: 0
RawStringFormats:
  - Language:        Cpp
    Delimiters:
      - cc
      - CC
      - cpp
      - Cpp
      - CPP
      - 'c++'
      - 'C++'
    CanonicalDelimiter: ''
    BasedOnStyle:    google
  - Language:        TextProto
    Delimiters:
      - pb
      - PB
      - proto
```

```
      - PROTO
    EnclosingFunctions:
      - EqualsProto
      - EquivToProto
      - PARSE_PARTIAL_TEXT_PROTO
      - PARSE_TEST_PROTO
      - PARSE_TEXT_PROTO
      - ParseTextOrDie
      - ParseTextProtoOrDie
      - ParseTestProto
      - ParsePartialTestProto
    CanonicalDelimiter: ''
    BasedOnStyle:    google
ReflowComments:  true
SortJavaStaticImport: Before
SortUsingDeclarations: true
SpaceAfterCStyleCast: false
SpaceAfterLogicalNot: false
SpaceAfterTemplateKeyword: true
SpaceBeforeAssignmentOperators: true
SpaceBeforeCaseColon: false
SpaceBeforeCpp11BracedList: true
SpaceBeforeCtorInitializerColon: true
SpaceBeforeInheritanceColon: true
SpaceBeforeParens: ControlStatements
SpaceAroundPointerQualifiers: Default
SpaceBeforeRangeBasedForLoopColon: true
SpaceInEmptyBlock: false
SpaceInEmptyParentheses: false
SpacesBeforeTrailingComments: 1
SpacesInConditionalStatement: false
SpacesInContainerLiterals: false
SpacesInCStyleCastParentheses: false
SpacesInSquareBrackets: false
SpaceBeforeSquareBrackets: false
BitFieldColonSpacing: Both
StatementMacros:
  - Q_UNUSED
  - QT_REQUIRE_VERSION
```

```
UseCRLF:         false
WhitespaceSensitiveMacros:
  - STRINGIZE
  - PP_STRINGIZE
  - BOOST_PP_STRINGIZE
  - NS_SWIFT_NAME
  - CF_SWIFT_NAME
...
```

clang-tidy ??

.clang-tidy

```
---
# Configure clang-tidy for this project.

# Here is an explanation for why some of the checks are disabled:
#
#  -google-readability-namespace-comments: the *_CLIENT_NS is a macro, and
#      clang-tidy fails to match it against the initial value.
#
#  -modernize-use-trailing-return-type: clang-tidy recommends using
#      `auto Foo() -> std::string { return ...; }`, we think the code is less
#      readable in this form.
#
#  -modernize-return-braced-init-list: We think removing typenames and using
#      only braced-init can hurt readability.
#
#  -modernize-avoid-c-arrays: We only use C arrays when they seem to be the
#      right tool for the job, such as `char foo[] = "hello"`. In these cases,
#      avoiding C arrays often makes the code less readable, and std::array is
```

```
#      not a drop-in replacement because it doesn't deduce the size.
#
#  -performance-move-const-arg: This warning requires the developer to
#      know/care more about the implementation details of types/functions than
#      should be necessary. For example, `A a; F(std::move(a));` will trigger a
#      warning IFF `A` is a trivial type (and therefore the move is
#      meaningless). It would also warn if `F` accepts by `const&`, which is
#      another detail that the caller need not care about.
#
#  -readability-redundant-declaration: A friend declaration inside a class
#      counts as a declaration, so if we also declare that friend outside the
#      class in order to document it as part of the public API, that will
#      trigger a redundant declaration warning from this check.
#
#  -readability-function-cognitive-complexity: too many false positives with
#      clang-tidy-12. We need to disable this check in macros, and that setting
#      only appears in clang-tidy-13.
#
#  -bugprone-narrowing-conversions: too many false positives around
#      `std::size_t`  vs. `*::difference_type`.
#
#  -bugprone-easily-swappable-parameters: too many false positives.
#
#  -bugprone-implicit-widening-of-multiplication-result: too many false positives.
#     Almost any expression of the form `2 * variable` or `long x = a_int * b_int;`
#     generates an error.
#
#  -bugprone-unchecked-optional-access: too many false positives in tests.
#     Despite what the documentation says, this warning appears after
#     `ASSERT_TRUE(variable)` or `ASSERT_TRUE(variable.has_value())`.
#
Checks: >
  -*,
  abseil-*,
  bugprone-*,
  google-*,
  misc-*,
  modernize-*,
  performance-*,
  portability-*,
```

```
  readability-*,
  -google-readability-braces-around-statements,
  -google-readability-namespace-comments,
  -google-runtime-references,
  -misc-non-private-member-variables-in-classes,
  -misc-const-correctness,
  -misc-use-anonymous-namespace,
  -modernize-return-braced-init-list,
  -modernize-use-trailing-return-type,
  -modernize-use-nodiscard,
  -modernize-avoid-c-arrays,
  -performance-move-const-arg,
  -readability-braces-around-statements,
  -readability-identifier-length,
  -readability-magic-numbers,
  -readability-named-parameter,
  -readability-redundant-declaration,
  -readability-function-cognitive-complexity,
  -readability-convert-member-functions-to-static,
  -readability-qualified-auto,
  -readability-implicit-bool-conversion,
  -bugprone-narrowing-conversions,
  -bugprone-easily-swappable-parameters,
  -bugprone-implicit-widening-of-multiplication-result,
  -bugprone-unchecked-optional-access,
  -bugprone-branch-clone

# Turn all the warnings from the checks above into errors.
WarningsAsErrors: "*"

CheckOptions:
  - { key: readability-identifier-naming.ClassCase,             value: CamelCase  }
  - { key: readability-identifier-naming.StructCase,            value: CamelCase  }
  - { key: readability-identifier-naming.TemplateParameterCase, value: CamelCase  }
  - { key: readability-identifier-naming.FunctionCase,          value: camel_back  }
  - { key: readability-identifier-naming.PrivateMemberPrefix,   value: _          }
  - { key: readability-identifier-naming.ProtectedMemberPrefix, value: _          }
  - { key: readability-identifier-naming.EnumConstantCase,        value: CamelCase }
  - { key: readability-identifier-naming.EnumConstantPrefix,      value: k         }
  - { key: readability-identifier-naming.ConstexprVariableCase,   value: CamelCase }
```

```
- { key: readability-identifier-naming.ConstexprVariablePrefix,  value: k         }
- { key: readability-identifier-naming.GlobalConstantCase,       value: CamelCase }
- { key: readability-identifier-naming.GlobalConstantPrefix,     value: k         }
- { key: readability-identifier-naming.MemberConstantCase,       value: CamelCase }
- { key: readability-identifier-naming.MemberConstantPrefix,     value: k         }
- { key: readability-identifier-naming.StaticConstantCase,       value: CamelCase }
- { key: readability-identifier-naming.StaticConstantPrefix,     value: k         }
- { key: readability-implicit-bool-conversion.AllowIntegerConditions,  value: 1   }
- { key: readability-implicit-bool-conversion.AllowPointerConditions,  value: 1   }
- { key: readability-function-cognitive-complexity.IgnoreMacros,  value: 1   }
```

C & C++ ????

1. ??

这份编码规范基于 Google C++ Style Guide，适用于 C。 C++ 作为一门非常强大的语言在各大厂被广泛使用，然而一直以来都没有一套公认的编程规范，不同人写出的代码风格迥异，导致代码在可读性、可维护性方面都有一些问题；为了解决这些问题，我们编写了编程规范，并且会强制 C++ 程序员遵循本规范；本文档风格约定部分可能跟你的喜好有冲突，请尽量用包容的心态来阅读。有任何问题或建议，欢迎跟我们讨论：shjborage@gmail.com

2. ??

本节主要规定和建议大家使用/不使用某些 C++ 语法，从而提高代码质量、可读性、可维护性。

2.1. ??????

本小节关注 C++ 作为面向对象编程语言时（所谓的 Object-Oriented C++）的使用规范。

2.1.1. struct ? class ??????

·[建议] `struct` 表示数据的简单集合，公开定义数据成员，只定义用于初始化数据成员的方法（比如构造/析构函数，`initialize()`，`reset()`，`validate()`）。 ·[强制] `class` 表示被封装的用户自定义类型，不公开定义非静态数据成员，一般通过成员方法进行交互。

解释 在 C++ 中 `struct` 与 `class` 只有默认成员权限上的差别。但 `struct` 是从 C 语言中借来的概念，因此应该尽可能保持其语义与 C语言中的一致——单纯的数据集合。示例

```
// 简单的数据聚合，没有动作
struct Coordinate {
    int x;
    int y;
    int z;
};

// 有动作的对象
class Cat {
```

```
public:
    void meow();
private:
    ...
};
```

2.1.2. ??????

- [强制] 必须使用构造函数初始化列表显式初始化直接基类与所有的基本类型数据成员
- [强制] 没有复制意义的类必须用 `DISALLOW_COPY_AND_ASSIGN` 宏禁止拷贝构造函数和赋值构造函数。
- [强制] 禁止在构造函数中进行可能出错的复杂操作(比如申请资源)，复杂操作用额外的 `init()` 函数实现

定义

构造函数

在面向对象编程中通过在对象创建时自动被调用而使对象进入合法状态的一个特别方法。

初始化列表

构造函数定义中，参数列表与函数体之间的列表，用于为数据成员赋初值。

解释

显式初始化能够使代码更清晰、不易错误调用，还能避免二次赋值造成的效率问题。

当类编写者没有显式声明构造函数时，C++ 编译器会自动生成默认构造函数与复制构造函数，而其行为往往与程序员的期望不一致。

示例

没有名字的人不是合法的人，而且人的名字不应该能够随便变化，所以名字适合在构造函数中传递。

```
class Person {
public:
    // 单参数构造函数尽量加上 explicit
    explicit Person(const std::string& name) : _name(name) {}  // 尽可能使用初始化列表初始化数据成员
private:
    std::string _name;
};
DISALLOW_COPY_AND_ASSIGN宏，以及其用法
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
        TypeName(const TypeName&); \
        TypeName& operator=(const TypeName&)

class Foo {
public:
    explicit Foo(int f);
    ~Foo();
private:
```

```
    DISALLOW_COPY_AND_ASSIGN(Foo);
};
```

参考

> Effective C++, item 04: Make sure that objectes are initialized before they’re used More Effective C++, Item 04: Avoid gratuitous default constructors 也提到不使用 non-trival的构造函数带来的风险和代价 More Effective C++, item 10: Prevent resource leaks in constructors

2.1.3. ??????

·[强制] 有默认值语意的类，必须显式定义默认构造函数

例外：对于没有成员变量的类，例如虚基类或仿函数类，不需要显式定义默认构造函数

·[强制] 没有默认值语意的类，必须显式定义其它构造函数或 private 声明默认构造函数，并不给出实现。

定义

默认构造函数

default constructor。形如 `MyClass::MyClass()` 的构造函数

当程序员没有显式声明/定义任何构造函数时，编译器将自动生成一个默认构造函数

解释

当类编写者没有显式声明构造函数时，C++编译器会自动生成默认构造函数与复制构造函数，而其行为往往与程序员的期望不一致。因此，类编写者必须注意，对于没有默认状态的类，应该 private 声明默认构造函数，并不给出实现。使用显式写出的默认构造函数（而非编译器生成的）具有更好的可读性，也可以使用文档化注释向使用者解释默认构造的对象状态

参考

> More Effective C++, Item 04: Avoid gratuitous default constructors Effective C++, item 06: Explicitly disallow the use of compiler-generated functions you do not want

示例

一个字符串对象，默认值理所当然是空串（”“），所以可以定义默认构造函数

```
class String {
public:
    String() : // 默认构造函数
        _str(_S_EMPTY_C_STR) {}  // 使用_S_EMPTY_C_STR作占位符
```

```

    ~String() {
        if (_str != _S_EMPTY_C_STR) {
            free(_str);
        }
    }

    const char* c_str() const {
        return _str;
    }

private:
    char* _str;
    static char _S_EMPTY_C_STR[1];  // = ""
};
```

2.1.4. ??????

·[强制]

只有一个参数的构造函数，除非是复制构造函数，或者希望提供隐式转换功能，否则必须声明为显式构造函数

◦ ·[建议]

在极少的情况下，当该类的隐式转换很有意义时，可以不把单参数构造函数声明为显式构造函数，但必须十分小心，并使用文档化注释说明

定义

隐式构造函数

implicit constructor。形如 `MyClass::MyClass(MyArg arg)` 的单参数构造函数 这种函数会定义 `MyArg` 类型到 `MyClass` 的隐式转换功能，可能会带来相当大的风险。 显式构造函数

explicit constructor。形如 `explicit MyClass::MyClass(MyArg arg)` 的单参数构造函数

解释 单参数构造函数如果不用 `explicit` 关键字修饰，则可能被编译器用来做隐式类型转换。

错误地使用了类型间的隐式转换，轻则带来效率降低，重则导致不正确的行为与难以查出的 bug

，因此应尽可能避免使用 示例

`Person( const std::string& )` 是一个单参数构造函数，如果没有使用 `explicit`，会导致 `string` 能够隐式转换为 `Person`，这是违反语意的。

```
class Person {
public:
    explicit Person(const std::string& name) : _name(name) {}
private:
    std::string _name;
};
```

`string` 字符串类和 C 风格字符串都表示同样的意思，可以使用隐式构造函数

```
class String {
public:
    String(const char* cs);                    // 该函数提供隐式转换功能。

private:
    ...
};
```

参考

> ❝ More Effective C++, Item 05: Be wary of user-defined conversion functions

2.1.5. ??????

·[强制] 有复制意义的类必须显式给出复制构造函数，并小心指定其行为（浅复制、深复制等）

定义

复制构造函数

copy constructor。形如 `MyClass::MyClass(const MyClass&)` 的构造函数

当程序员没有显式声明/定义任何构造函数时，编译器将自动生成一个复制构造函数

解释

托管了资源的类，往往是没有复制意义的。此时应当防止用户错误调用而导致资源泄漏、重复释放的后果。

编译器默认生成的复制构造函数，对指针数据成员使用浅复制策略，但这种策略常常不是程序员希望的。

示例

一个可以复制的类

```
class Point {
public:
    Point(int x, int y) : _x(x), _y(y) {}
    Point(const Point& other) :   // 复制构造函数
            _x(other._x), _y(other._y) {}
private:
    int _x;
    int _y;
}
```

一个不可以复制的类 (RAII类一般都是不可复制的)

```
class File {
public:
    explicit File(const char* file_name) :
            _fp(NULL) {
        ...  // open file, and set _fp
    }
    ~File() {
        fclose(_fp);
    }
private:
    FILE *_fp;
    DISALLOW_COPY_AND_ASSIGN(File);
};
```

参考

> Effective C++, item 06: Explicitly disallow the use of compiler-generated functions you do not want Effective C++, item 14: Think carefully about copying behavior in resource-managing classes

2.1.6. ???????

·[强制　] 有复制赋值意义的类必须显式定义复制赋值运算符，并小心指定其行为（浅复制赋值、深复制赋值等）

定义

复制赋值运算符

assignment operator=：形如 `MyClass& MyClass::operator=(const MyClass&)` 的运算符重载

当程序员没有定义该运算符重载时，编译器将自动生成一个赋值运算符函数

解释

托管了资源的类，往往是没有赋值意义的。此时应当防止用户错误调用而导致资源泄漏、重复释放的后果。

编译器默认生成的赋值运算符，对指针数据成员使用浅复制策略，但这种策略常常不是程序员希望的。

示例

一个坐标点，是有赋值语意的

```
class Point {
public:
    Point(int x, int y) : _x(x), _y(y) {}
    Point(const Point& other) :
            _x(other._x), _y(other._y) {}
```

```
    Point& operator=(const Point& other) {   //复制赋值运算符
        _x = other._x;
        _y = other._y;
        return *this;
    }
private:
    int _x;
    int _y;
}
```

一个人，在克隆人技术和借尸还魂技术被发明出来之前，是没有复制或赋值语意的

```
class Person {
public:
    explicit Person(const std::string& name) : _name(name) {}
private:
    std::string _name;
    DISALLOW_COPY_AND_ASSIGN(Person);
};
```

参考

> Effective C++, item 06: Explicitly disallow the use of compiler-generated functions you do not want Effective C++, item 14: Think carefully about copying behavior in resource-managing classes

2.1.7. ????

·[强制　] 若类定义了虚函数，必须定义虚析构函数 ·[强制　]
若类设计为可被继承的，应该定义公开的虚析构函数或保护的非虚析构函数 ·[强制　] RULE010
含有指针成员，必须显式给出析构函数，并小心指定其行为（是否销毁指针，如何销毁等） ·[强制　]
绝不允许让异常离开析构函数。 ·[建议　]
析构函数应该用于释放资源，销毁对象，避免执行复杂的操作，尤其避免执行可能失败的操作。

定义　destructor。形如 `MyClass::~MyClass()` 的函数

解释　若基类的析构函数没有声明为虚函数，则 `delete pBase;` 将不会调用子类析构函数，从而导致错误行为和内存泄漏，因此，避免继承具有公开非虚析构函数的类。

编译器默认生成的析构函数不会析构指针成员指向的对象，更不会回收其内存。如果忘记处理的话会导致资源泄漏

程序抛出异常时，会导致栈展开，局部对象依次析构。如果析构过程中再次抛出异常。程序将会立即中止。

示例 析构函数示例：一个数据库链接的RAII类，能够自动关闭连接，释放资源

```
class Connection {
public:
    void close();    // 关闭数据库连接，具有"无抛掷保证"（定义参见"编程实践->异常安全性"一节）
    ~Connection() {
        if (_connection_state == CONNECTED) {
            close();  // 注意：不要让异常离开析构函数，如果调用了会抛出异常的函数，一定要接住异常。
        }
    }
};
```

参考

> Effective C++, item 07: Declare destructors virtual in polymorphic base classes
> Effective C++, item 08: Prevent exceptions from leaving destructors More
> Effective C++, item 11: Prevent exceptions from leaving destructors

2.1.8. ??

- ·[强制] 公开继承的类，必须表示“是一个”的关系，禁止仅仅为复用某些方法而使用继承
- ·[强制] 派生类不允许覆盖基类中的非虚函数
- ·[强制] [RULE014] 使用公有继承，禁用保护继承和私有继承

解释

继承是一个经常被滥用的概念。最常见的滥用情形是仅仅为了可以默认地利用某些方法而使用继承。这是有很大危害的。因为这样做往往提供了不一致的抽象，导致用户难以理解、容易误用。

覆盖非虚函数会导致多态性失效，引起难以发现的bug 对于不满足“是一个”关系的两个类，若要复用代码，请使用组合代替继承

示例 一个表示”是一个”关系的例子

```
class Chinese {
public:
    explicit Chinese(const std::string& name) : _name(name) {}

    // 可以被继承的类析构函数必须是虚的
    virtual ~Chinese() {}

    // 这个函数不是虚函数，不可被覆盖
    void say(const std::string& word);
```

```
    // 这个函数可以被覆盖
    virtual void greet() { say("吃了吗? "); }

private:
    std::string _name;
};

// 一个广东人"是一个"中国人

class Cantonese : public Chinese {
public:
    explicit Cantonese(const std::string& name) : Chinese(name) {}
    virtual ~Cantonese() {}

    // 广东人问候的方式不一样，覆盖本方法
    virtual void greet() { say("食咗未啊?"); }
};
```

参考

> Effective C++, Item 36: Never redefine an inherited non-virtual function Effective C++, Item 32: Make sure public inheritance models “is-a” Effective C++, Item 38: Model “has-a” or “is-implemented-in-terms-of” through composition Effective C++, Item 39: Use private inheritance judiciously C++ Coding Standards, 第 37条: 公用继承即可替换性。继承，不是为了重用，是为了被重用。 C++ Coding Standards, 第 34条: 用组合代替继承

2.1.9. ????

- [强制] 尽量避免使用多重继承。如果使用，最多只能有一个基类是实现继承，其它只能是接口继承。

解释

客观世界里一个事物同时既是一个事物又是另外一个事物的情况极少，而且不直观（人通常习惯于用分类的方法来理解事物之间的关系的），如果要进行多重实现继承，考虑使用组合代替。

参考

> Effective C++, Item 40: Use multiple inheritance judiciously

2.1.10. ??????

- [建议] 在满足需求与接口完整性的前提下，必须为所有方法成员提供尽可能严格的访问控制
- [强制] 类不能定义 public非静态数据成员，不应该定义 protected数据成员
- [强制] 类可以定义 public 静态数据成员，但必须是 const的。否则，应该通过静态 getter/setter访问，并通过文档指定其是否线程安全。
- [强制] 除非嵌入类是接口的一部分，否则禁止将其声明为 public

解释　当类成员通过 `public/protected` 的方式暴露给使用者 / 派生类后，就失去了通过重构优化代码的能力，因此，适当的访问控制是必须的。数据成员的耦合比方法成员的耦合更大，而且经常有一些微妙的一致性约束，使用者往往难以全部顾及。因此数据成员应尽可能声明为 `private`，另外定义适当的 `public/protected` 的 `getter/setter` 来访问类的静态成员除了面临封装性问题外，还须面临线程安全问题。因此，应该将常数参数定义为 `const` 的，并使用 `getter/setter` 保护可变的参数

参考

> ❝ Effective C++, Item 22, Declare data members private

2.1.11. ??????

- [强制] 按照 `public、protected、private` 的顺序声明成员。按照类型、方法、数据成员的顺序声明成员，`DISALLOW_COPY_AND_ASSIGN` 放在 `private` 区段的末尾

解释　一致的顺序有利于阅读。一般来说，排在前面的，是读者更可能关心的信息。

示例　一个资源池，能托管 C风格资源或 C++风格对象

```
// 以真实的bsl::ResourcePool为原型的无敌简化版
// 以一致的顺序排序，并且将读者最可能关心的信息排在前面
class ResourcePool {
public:
    // 类型定义
    typedef void(*DestructorFunc)(void * p_object);  // 清理函数类型定义
    // 构造函数、析构函数声明/定义
    ResourcePool();
    ~ResourcePool() {
        // 如果函数很短，也可以在声明同时给出定义（自动inline）
        reset();
    }
```

```
    // 公有方法定义
    void attach(void * p_object, DestructorFunc destructor);  // 托管C风格资源如内存buffer、文件指针等
    ...                                                       // attach系列有多个类似的方法，排在一起

    template <typename T>
    T& create();                                // 构造出托管对象
    ...                                         // create系列有多个类似的方法，排在一起
    ...                                         // 再排列其它次要的方法，如
create_raw(),clone_cstring(),reset()等
private:
    // private成员不是接口的一部分，用户一般不关心，放在后面
    // 类型定义
    struct ObjectInfo;                          // 各种内部使用的类型定义
    ...
    // 方法定义
    void push_info();                         // 现在的实现有push_info()和pop_info()函数，但可能会随着重构而改变
    void pop_info();                          // 因此声明为private函数，并且放在类定义的后面
    ...
    // 数据成员定义
    ObjectInfo *_p_object_info_list;
    ...
    DISALLOW_COPY_AND_ASSIGN(ResourcePool);  // 这个放最后
};
```

2.1.12. 友元 ·[建议] 不是非常必要的话，避免使用友元。

解释 友元破坏了类的封装性，增加了类之间的耦合，因此应该避免使用。
适宜使用友元的例子如容器与它的迭代器，还有类与涉及该类的运算符重载等。

示例 一个字典容器类，定义 DictIterator迭代器类为它的友元类，定义 `operator ==( const Dict&, const Dict& )` 为它的友元方法

```
class DictIterator;
class Dict {
public:
    typedef DictIterator iterator;
    friend class DictIterator;                          // 容器类与其迭代器天生是紧密耦合的，一般声明为friend
    friend bool operator ==(const Dict&, const Dict&);  // 对象和相关的重载运算符有可能紧密耦合，可以声明为friend
private:
    ...
```

```
};

class DictIterator {
    ...
};
```

参考

> Effective C++, Item 23, Prefer non-member non-friend functions to member functions

2.1.13. ????

·[建议] 建议不要使用异常，除非已有项目/底层库使用了异常，这时候必须要 `catch` 所有异常

解释

优点：异常允许上层应用决定如何处理在底层嵌套函数中“不可能出现的”失败，不像错误码记录那么含糊又易出错；很多现代语言都使用异常．引入异常使得 C++ 与 Python, Java 以及其它 C++ 相近的语言更加兼容．许多第三方 C++ 库使用异常，禁用异常将导致很难集成这些库．异常是处理构造函数失败的唯一方法．虽然可以通过工厂函数或 `initialize()` 方法替代异常，但他们分别需要堆分配或新的“无效”状态；

在测试框架中使用异常确实很方便．

缺点：在现有函数中添加 `throw` 语句时，你必须检查所有调用点．所有调用点得至少有基本的异常安全保护，否则永远捕获不到异常，只好“开心的”接受程序终止的结果．例如，如果 `f()` 调用了 `g()`, `g()` 又调用了 `h()`，`h` 抛出的异常被 `f` 捕获，`g` 要当心了，很可能会因疏忽而未被妥善清理．更普遍的情况是，如果使用异常，光凭查看代码是很难评估程序的控制流：函数返回点可能在你意料之外．这回导致代码管理和调试困难．你可以通过规定何时何地如何使用异常来降低开销，但是让开发人员必须掌握并理解这些规定带来的代价更大．异常安全要求同时采用 RAII 和不同编程实践．要想轻松编写正确的异常安全代码，需要大量的支撑机制配合．另外，要避免代码读者去理解整个调用结构图，异常安全代码必须把写持久化状态的逻辑部分隔离到“提交”阶段．它在带来好处的同时，还有成本(也许你不得不为了隔离“提交”而整出令人费解的代码). 允许使用异常会驱使我们不断为此付出代价，即使我们觉得这很不划算．启用异常使生成的二进制文件体积变大，延长了编译时间(或许影响不大), 还可能增加地址空间压力．异常的实用性可能会怂恿开发人员在不恰当的时候抛出异常，或者在不安全的地方从异常中恢复．例如，处理非法用户输入时就不应该抛出异常．如果我们要完全列出这些约束，这份风格指南会长出很多！结论：从表面上看，使用异常利大于弊，尤其是在新项目中．但是对于现有代码，引入异常会牵连到所有相关代码．如果新项目允许异常向外扩散，在跟以前未使用异常的代码整合时也将是个麻烦，引入带有异常处理的新代码相当困难．所以建议：不要使用异常，除非已有项目/底层库使用了异常，这时候必须要 catch 所有异常。

2.1.14. ???????

·[强制] 如果使用 int 作为返回值，必须遵循 ”负值表示失败，非负值表示成功” ·[强制] 若调用了可能返回错误码 / 空指针的函数，必须检查其返回值。

解释 由于错误码检查会大大增加代码量，很多程序员常常忽略对 ”不大可能” 出错的函数的错误码的检查，造成错误被掩盖，从而造成了严重而难以追查的错误。因此，只要函数有错误码返回值，必须进行检查，无论错误看起来多么不可能发生。

2.2. ?? C++ ??

2.2.1. ????

·[强制] [RULE022] 所有的代码应该定义在 `namespace` 中 C/C++ 任何有效代码必须都被 `namespace` 封装起来，除 `main()` 函数外. ·[强制] [RULE023] 头文件中禁止在函数、方法、类作用域外的地方使用 using

·[强制] [RULE024] 源文件中禁止 `using namespace`，但允许 `using class`

解释

命名空间能够有效避免命名冲突、使代码以更灵活的方式组织（把多个相关的类、函数组织为包），代码更具有可读性，因此规定必须使用命名空间。这几条规则确保在编写代码时只 using 了自己需要的类 如果遵循使用 <项目 >::<模块 > 为 root 的方式，一般需要 using 的其他名空间 class 的情况不会很多。如出现大量 using 语句，请先考虑该 cpp 是否放错了名空间 or 对外耦合是否过于严重。

示例

```
// 具名命名空间
namespace saick {
...
// 嵌套的具名命名空间
namespace my_module {
...
}  // namespace my_module
}  // namespace saick
```

对于下面的代码（ BadCase: 名空间污染) :

```
// a.h
#include <string>
using namespace std;  // 禁止在头文件中使用using namespace

class A {
public:
    string name() const;
}
```

```
// a.cpp
#include “bsl/bsl_string.h”
#include “a.h”

using namespace bsl;  // 禁止在.cpp文件中使用using namespace

int main() {
    string str; // 编译器无法区分这里是使用bsl::string还是std::string

    A a;
    cout << a.name() << endl;  // 用户不知道std名空间已经被打开了
}
```

建议改为:

```
// a.h
#include <string>

class A {
public:
    std::string name() const;  // 使用std::string替换string，显式指明所属空间
};

// a.cpp
#include "bsl/bsl_string.h"
#include "a.h"

using std::cout;  // 可以使用using class显式说明来源
using std::endl;

int main() {
    bsl::string str;  // 明确类型来源

    A a;
    cout << a.name() << endl;
}
```

2.2.2. ????

·[强制] [RULE025] 禁止内联虚函数 ·[建议] 不应该内联过大、过于复杂的函数 (建议 10行以内)

解释　内联虚函数不能达到函数展开的目的，还会导致代码膨胀。

内联函数本身通过减少函数调用开销来起到优化目的。内联过大的函数不但起不到优化效果，还会导致代码膨胀、增加内存换页次数，得不偿失。

注意在类定义里给出定义的方法会自动内联

示例　适合使用内联函数的示例

```
class Person {
public:
    std::string get_name() const {  // 注意在类里边定义的函数会自动内联
        return _name;
    }
    int get_age() const {
        return _age;
    }
private:
    std::string _name;
    int _age;
};

inline void greet(const Person& person) { //通过inline关键字来内联函数
    std::cout << "Hello, "<< person.get_name() << "!" <<std::endl;
}
```

参考

> Effective C++, item 30: understand the ins and outs of inlining

2.2.3. ?????

·[建议] 只有当重载运算符对该类很自然时才重载，绝不贪图使用方便而盲目重载运算符。

解释　盲目重载运算符会导致代码难以理解，容易用错。

示例　为迭代器重载 ++运算符

```
class MyContainer {
public:
    class iterator {
    public:
        iterator& operator++() {
            ...
```

```
                return *this;
            }
        };
    };
```

2.2.4. ??/????

·[强制] 只有在参数类型不同，而表示意义相同时才使用函数重载

·[建议] 必须非常小心重载函数的参数顺序，防止用户误调用错误版本的函数

·[强制] 特别地，避免整形数值与指针类型之间的重载

解释 在 C++里 NULL只是定义为 0的一个宏。如果重载整形数值与指针，当传入 NULL 时，就会调用错误版本的函数。

2.2.5. ????

·[强制] [RULE028] 禁止使用默认参数

解释 默认参数是编译期绑定的，不具有多态性。重定义默认参数会造成难以发现的 bug

参考

> Effective C++, Item 37: Never redefine a function’s inherited default parameter value

2.2.6. ????????RTTI?

·[强制] 除非用于跟踪或调试，不应该使用 RTTI。如果为了实现多态，应该使用虚函数代替。

·[建议] 尽可能避免使用 dynamic_cast。

解释 如果使用 RTTI 来模拟多态，会破坏对象的封装性，无法扩展，而且效率很低。dynamic_cast 有额外的运行时开销。而且很可能破坏了封装性，增加了代码的耦合度，因此应该尽量避免使用。

2.2.7. ????

·[建议] 尽可能减少类型转换操作

·[强制] 必须使用 C++风格类型转换。

定义 C++ 风格的类型转换：指 C++ 引入并推荐的 `static_cast`、`dynamic_cast`、`const_cast`、`reinterpret_cast` 运算符

解释 类型转换是容易出错的地方，尽可能避免使用。容易出错的地方必须显眼，而且尽可能清晰明了。C++ 风格的类型转换可以明确指明转换的方式，还可以通过 `_cast` 来方便地找到

示例 使用 `static_cast` 把无类型指针（`void *`）转换为为有类型指针

```
void foo(const void* network_buf) {
    const MyBuf * buf = static_cast<const MyBuf*>(network_buf); //无类型数据转换为有类型，是相当危险的，所以要明确指明。
}
```

使用 `const_cast` 来调用旧风格的程序库

```
// 正确的声明应该是void old_copy(void *dest, const void *src, size_t n);
// 但由于在另外一个库里，无法改写
void old_copy(void* dest, void* src, size_t n);

// 重新封装一个函数
void new_copy(void* dest, const void* src, size_t n) {
    old_copy(dest, const_cast<void *>(src), n);   // 把有const保护的强制转换为没有const保护的，是相当危险的，所以要明确指明。
}
```

参考

> Effective C++, item 27: Minimize casting More Effective C++, item 02: Prefer C++ style casts

2.2.8. ++/—???

·[强制] 在不考虑返回值的前提下，数值类型、指针、迭代器，统一使用前置自增 (自减).

解释 后置的 `++/--` 产生了额外的复制成本。如果迭代器使用了动态内存，还会造成额外动态内存分配。

参考

> More Effective C++, item 06: Distinguish between prefix and postfix forms of increment and decrement operators

2.3. C/C++ ????

2.3.1. ?????

·[建议] 避免使用非布尔型的变量或表达式作为分支语句条件

·[强制] 避免在条件表达式中赋值

·[建议] 作相等比较时，建议把常量或右值变量放在 ==运算符的右边

·[强制] [RULE033] 避免对浮点数进行相等或不等比较

解释　非布尔值的表达式比布尔值表达式难理解，应避免。

条件表达式中的赋值常常是因为手误造成的，应避免。常量 / 右值放在右边更符合人的阅读逻辑。现代编译器可以检测出在 `if` 条件里把 `==` 误写为 `=` 的情况。

浮点数在运算中经常会产生少量的误差，这个误差会导致 `==` 或 `!=` 运算返回与期望相反的结果。

示例

```
// 布尔型变量
if (is_valid) {
    ...
}
if (!is_finished) {
    ...
}
// 整型变量
if (my_value == 0) {
    ...
}
// 指针变量
if (p_value == NULL) {
    ...
}
// 浮点数（避免==和!=判断）
const double EPSILON = 1e-9; // 比如说，接受1e-9以内的误差
if (fabs(my_value) < EPSILON) {
    ...
}
```

2.3.2. NULL, nullptr?0

·[强制] 分别使用 `0`, `0.0`, `NULL`, `'\0'` 表示整数、浮点数、指针、字符中的 0。 ·[建议] 在支持 C++11 的环境中，建议使用系统关键字 `nullptr` 作为空指针常量。

解释　使用不同的字面量形式具有更好的可读性。

示例

```
int i = 0;
double i = 0.0;
void* p = NULL;
char ch = '\0';
```

2.3.3. sizeof ???

·[强制] 若要取得变量的大小，必须使用 sizeof() 关键字，不能自己估算。 ·[建议] 尽可能使用 sizeof(变量) 而非 sizeof(类型名) 解释 变量的大小可能会由于 32/64位环境、对齐、重构等因素而变化。因此必须使用 sizeof()关键字来避免此类错误。 进行代码重构时，变量的类型与大小可能会发生改变， sizeof(类型名) 无法自动更新大小值，从而造成难以发现的 bug。

示例

```
int g_my_id;

void func() {
    size_t size_of_my_id = sizeof(g_my_id);   // OK
}
int g_my_id;

void func() {
    size_t size_of_my_id = sizeof(int);       // NO! g_my_id有可能改为long long或者size_t！
}
```

2.3.4. typedef ???

·[建议] 除非用作类的 type tag，或者作为模板类的常用实例化类的别名外，应尽可能限制 typedef 的作用域 ·[强制] 禁止定义 struct 的同时 typedef 一个类型名，除非该头文件必须兼容 gcc

解释 滥用 typedef 会造成类型体系的混乱，因此必须审慎地使用 typedef。

示例 YES:

```
class MyContainer {
public:
    typedef char value_type;    // type tag

    void my_method() {
        typedef std::vector<int>::iterator iterator_type;   //在函数作用域里把长类型名缩短
        ...
    }
}
```

NO:

```
typedef tagMyStruct {
    ...
```

```
} MyStruct; // 在C++中没必要这样写，这是C语言时代的写法
```

2.3.5. goto

·[强制] [[RULE037]] 禁止 goto 语句。

解释 goto 会使代码流程变得混乱而难以理解。

2.3.6. ?

·[强制] 除非用于条件编译、跟踪调试，或者能够显著减少代码量并确保可读性的宏（如 `DISALLOW_COPY_AND_ASSIGN`, `CFATAL_LOG` etc.)，否则不应使用 .

·[建议] 宏代码尽可能使用 do...while(0) 块包围。

·[强制] 如果必须编写宏函数，宏参数必须使用小括号包围。 例外：使用 `##` 连接宏参数或使用 `#` 把宏参数转换为字符串

·[强制] 如果必须编写宏函数，避免宏函数引入控制语句，如 `break`, `continue`, `return` 等。

·[强制] 如果必须编写宏 /宏函数，避免宏 /宏函数依赖全局变量。

·[强制] 如果必须使用宏函数，避免宏函数作为赋值表达式的左值。

解释

宏的使用非常不直观，编译器不能做有效的合法性检查，也非常不利于调试，宏还会造成名称冲突，以及各种诡异的错误。 相比于宏，替代方法往往有很大的好处： inline 函数不会引起名称冲突，具有强类型检查，不怕函数调用的副作用，可以做为类的方法。 常量和枚举变量能够限制在名空间内，不易造成名空间污染，且具备强类型检查。

示例 使用 inline 函数代替宏函数

```
inline int abs(int i) {
    return (i >=0 ? i : -i);
}
// 或者
template <typename T>
inline T abs( T value ) {
    return (value >= 0 ? value : -value);
}
```

使用枚举代替宏做常量定义

```
enum Color {
    RED,
    BLUE,
    GREEN,
    COLOR_COUNT
};
```

使用 const常量代替宏做常量定义

```
const size_t BUFFER_SIZE = 1048576;
```

参考

> Effective C++, item 02: Prefer consts, enums, and inlines to #define

2.3.7. ????

·[建议] 鼓励使用前向声明，以减少文件之间的依赖关系

解释

文件之间产生循环依赖是应该尽可能避免的事情，但有时候的确无法避免。使用前向声明能够减少文件之间的
依赖关系。

示例 前向声明一个类

```
class Iterator;  //前向声明
class Container {
    Iterator* begin();
};
```

参考

> Effective C++, item 31: Minimize compilation dependencies between files

2.3.8. ??/????/??

·[强制] 尽可能避免使用全局变量，如有必要应使用 singleton模式代替。 ·[强制] 禁止使用 class
类型的静态或全局变量 ·[强制] 禁止全局变量、静态变量之间存在依赖关系。 ·[强制] 不建议编写除 `main()`
函数、动态链接库入口函数以外的全局函数，使用类和 /或命名空间把它们组织起来。 ·[强制]
内部使用的全局函数，必须声明为静态函数，不能在目标文件中出现外部可见的符号 ·[强制]
内部使用的全局变量，必须声明为静态变量，不能在目标文件中出现外部可见的符号

解释 全局变量跟所有的代码都发生了耦合，如果出错非常难跟踪。因此应该尽可能避免 静态变量的构造函数 ,
析构函数以及初始化操作的调用顺序在 C++ 标准中未明确定义 ，甚至每次编译构建都有可能会发生变化 ,
从而导致难以发现的 bug. 比如 ，结束程序时 ，某个静态变量已经被析构了 ,
但代码还在跑，此时其它线程很可能还在试图访问该变量 ，直接导致崩溃 . C++
对各编译单元的静态、全局变量的初始化顺序是没有保证的。因此不应该存在这些变量之间的依赖关系。案例 :
some_compiler_1-3-8 版本生成的代码就会有全局变量的顺序依赖，程序退出时有 crash 的风险。
全局函数污染了命名空间，难以重用，因此应该用类和 /或命名空间把它们组织起来
历史上曾经多次出现过由于内部使用的全局函数，没有声明为静态函数，从而发生错误链接的事件，因此必须

避免。 假如必须使用全局变量，而该变量是只有内部可见的，应声明为静态变量，以免发生错误链接。

2.3.9. ????

·[强制] [RULE050] 不应在头文件定义类 /结构体类型的全局常量，以减少代码膨胀 ·[强制] [RULE051] 使用 enum 或 const 定义常量，不使用 define ·[强制] 定义 enum 时，只有符号意义的枚举值由编译器分配值，否则应显式指定值 ·[强制] 可能用于跨模块间通讯或者涉及存储的枚举值必须显式指定值，避免版本不一致造成诡异的错误。 ·[建议] 建议由编译器分配值的枚举定义最后一个枚举值为 XXX_COUNT(用于循环枚举所有枚举值) ·[强制] 禁止使用魔术数字，必须替换为符号常量

解释 define 定义的常量不能使用类 /命名空间等限定作用域，容易造成莫名其妙的错误。此外， define 定义的常量如果是通过表达式求值得到的，还会造成多次求值，增大运行时开销。只有符号意义的枚举值由编译器分配值，避免产生间断的或重合的取值 魔术数字不容易理解，而且在数值失效后不容易查觉（比如某处使用 8 来代替 `strlen("filename")` ，结果后来字符串变为 `file_name` 了，数值就是错的了）若多处使用了同一个魔术数字，更新时更容易造成改漏、改错。

示例 使用枚举做常量定义

```
enum Color {
    RED,
    BLUE,
    GREEN,
    COLOR_COUNT
};
```

使用 const常量做常量定义

```
const size_t BUFFER_SIZE = 1048576;
```

参考

> Effective C++, item 02: Prefer consts, enums, and inlines to #define

2.3.10. const???

·[强制] 在下列情况下，加上 const关键字: 不修改内部状态的成员方法必须声明为 const 返回不可修改的指针或引用必须声明为 const 不被修改的引用 /指针形参必须声明为 const 除非保证状态不会被修改，不应使用 const_cast 来去掉 const 属性 ·[强制] const 关键字用在类型名前面而非后面

解释 尽可能的使用 const 可以使编译器更好地检查代码，避免手误造成的难以发现的状态改变 bug
const_cast 一般只用在跟非常老旧的 C库（不包括 C标准库）交互上。那里有部分输入函数没有声明为 const
const 关键字放在类型名前面主要是为了统一风格。 示例

```
class MyClass {
public:
    void my_const_method(const OtherClass& arg) const;
};

const MyClass my_obj;   // const 对象
const MyClass* p1;      // 指向常量的指针
const MyClass* const p2 = &my_obj; // 指向常量的指针常量
```

参考

> Effective C++, item 03: Use const whenever possible

2.3.11. ??????

·[建议] 避免使用可变长度数组 ·[强制] 避免定义超过 4KB 的局部变量数组。

定义 可变长度数组 (variable-length array):（在栈上分配的）变长数组 ，属于 GCC 扩展语法。

解释 在栈上分配可变长度数组或过大的局部变量数组容易造成栈溢出错误出 crash，而且这种 crash 由于栈信息破坏而无法调试。因此应避免使用。

3. ??/??

3.1. ?????

3.1.1. ???????

·[强制] 必须按以下顺序引用头文件: .cpp对应的头文件 (如果有), C(标准)库， C++(标准) 库，其它库，自己项目的 .h文件

解释 在这种引用顺序的要求下， 如果 cpp对应的头文件遗漏了必要的引用文件， 编译将失败，问题会立刻暴露出来 ；否则， 问题有可能被掩盖， 给未来的改动埋雷 ；

示例 举例来说 , some-project/foo/internal/fooserver.cc 的包含次序如下 :

```
#include "foo/public/fooserver.h" // 优先位置
#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>
#include "base/basictypes.h"
#include "base/commandlineflags.h"
#include "foo/public/bar.h"
```

3.1.2. #include<>?#include”“

·[强制] 系统库使用 #include <>, 其他库使用 #include ""

3.1.3. #include?????????????

·[建议] 建议 include的路径、文件名与命名空间保持一致。

解释 `#include` 使用与命名空间对应的相对路径具有更好的可读性，容易记住，并且减少文件冲突的机会

示例

```
#include "bsl/ResourcePool.h"
#include "bsl/var/IVar.h"
#include "mylib/MyClass.h"
```

3.1.4. ??include guards

·[强制] [RULE060] 头文件必须使用 include guards ·[强制] [RULE061] include guards必须使用 `<SVNPATH>_<FILE>_H`（或HPP）的全大写形式。 `SVNPATH` 中需要去掉标示主干 /分支的关键词 如 ：trunk, branch-xxx;

定义 include guards 防止头文件使用者重复 include 同一个头文件而使用的 `#ifndef`, `#define`, `#endif` 预处理指令

解释 使用良好编写的 include guards 可以减少文件名冲突导致一些莫名奇妙的编译错误 svn 路径作为 include guards 前缀不会冲突 ；示例

// 例如 , svn 中的头文件 `https://svn.saick.net/ps/se/trunk/ac/strategy/StrategyQueue/default/time_open_cluster_data.h` 必须按如下方式保护 :

```
#ifndef PS_SE_AC_STRATEGY_STRATEGYQUEUE_DEFAULT_TIME_OPEN_CLUSTER_DATA_H
#define PS_SE_AC_STRATEGY_STRATEGYQUEUE_DEFAULT_TIME_OPEN_CLUSTER_DATA_H
…
#endif // PS_SE_AC_STRATEGY_STRATEGYQUEUE_DEFAULT_TIME_OPEN_CLUSTER_DATA_H
```

3.2. ??

3.2.1. ????

·[强制] 头文件按以下顺序组织

1. 文件声明
2. include guards
3. 引用的头文件
4. 自定义类型声明 ，函数声明 (一般函数 ,模板函数), 内联函数 /模板函数 ·[强制] CPP文件按以下顺序组织
5. 文件声明
6. 引用的头文件
7. 代码部分 （注意：涉及的函数和类方法按照头文件的顺序组织） ·[建议]
   对于一个库，可给出一个与库同名的头文件，包含该库下所有的公开接口

解释 一致的文件组织方式方便理解与查找。

示例 文件申明标准实例：

```
// Copyright 2017 Saick net. All Rights Reserved.
// Author: LastName FirstName (eric@saick.net)
//
// <注意：上面空一行。此处开始描述文件功能>
如果多个Author，按照如下写法
// Copyright 2017 Saick net. All Rights Reserved.
// Author: LastName FirstName (eric@saick.net)
//         LastName2 FirstName2 (x@saick.net)
//
// <注意：上面空一行。此处开始描述文件功能>
```

3.2.2. ?????

·[强制] [RULE064] 一条单独的语句必须独立成行 ·[强制] 使用适当的空行来分组代码的逻辑 ·[强制] [RULE066] 避免连续的空行 ·[强制] 左大括号不独立，右大括号独立成行。并且左大括号所在行进行垂直对齐

例外 ：空函数 左右大括号可以放在函数签名末尾 ；例外 ：对于带分支的 if语句 ，右大括号后面还有 else/else if 分支的 ，需要与 else/else if语句合并成行 .

解释 适当的空行，能够更好地对代码逻辑进行分组，增加可读性

连续的空行，没有实际的意义，使代码显得松散，常常是编写代码不认真的副产品。

示例

```
// mylib/AwesomeClass.h
namespace mylib {
```

```
class AwesomeClass {
    void foo() { //内联函数
        ...
    }
    void bar();
};  // MyClass

}   //namespace mylib

// mylib/AwesomeClass.cpp
namespace mylib {
void AwesomeClass::bar() {
    while (...) {
        if (...) {
            ...
        } else {
        ...
        }
    }  // while loop
}

}  // namespace mylib
```

3.2.3. ??

·[强制] [RULE068] 使用空格缩进，不使用制表符 ·[建议] 避免超过 5重的缩进 ·[强制] [RULE069] 以 4 个空格为单位缩进

解释 我们没有也不可能限制程序员使用的编辑器，不同的编辑器（还有 `terminal` ）对制表符长度的定义不尽相同，为免因此造成混乱的代码缩进，不应使用制表符。

过多的缩进会造成代码偏在一边，在窄屏幕上看非常不便；此外，过多的缩进也是代码划分不合理的一个征兆。4格的缩进比较清晰，也是一直以来的习惯。

3.2.3.1. ??????

·[强制] [RULE070] 命名空间不缩进

解释 namespace 缩进会让每行可写入的代码变少 (每行最多 100字节 ,包括缩进); 而 namespace 不缩进并不会影响阅读 ;

示例 不缩进的例子

```
namespace bsl {
namespace var {
class IVar {
};

}  // namespace var
}  // namespace bsl
```

3.2.3.2. ?????

·[强制] [RULE071] 预处理指令不要缩进

3.2.3.3. ?????

·[强制] [RULE072] public/protected/private关键字与类声明对齐

示例

```
class MyClass {
public:
    void foo();
protected:
    void bar();
private:
    void meow();
    int _haha;
};
```

3.2.3.4. ?????????

·[强制] 构造函数初始化列表放在同一行或按至少八格缩进并排几行 , 如果需要换行 , 把 : 放在第一行 . 解释 这样做主要因为函数体缩进为 4格缩进, 为了容易区分函数体第一行函数, 因而采用至少 8格缩进 示例

```
// 如果一行可以放得下
MyClass::MyClass(int var) : _some_var(var), _some_other_var(var + 1) {}

// 如果一行放不下，至少缩进8个空格，把冒号放在第一行最后
// the first initializer line:
MyClass::MyClass(int var) :
        _some_var(var),              // 8 space indent
        _some_other_var(var + 1) {   // lined up
     do_something();
```

```
    ...
}
```

3.2.3.5. ????

·[强制] 比较短的函数声明，整个声明占一行；过长的函数声明，令每一参数占一行，并且垂直对齐，换行后的参数至少保持 8个空格缩进

示例

```
void foo();
void this_is_a_very_long_function_name(
        VeryLongLongDogType *p_my_dog,
        const std::map<std::string, int>& a_wierd_dictionary);
```

3.2.3.6. ??????

·[强制] 比较短的函数调用语句，整个语句占一行；过长的函数调用，控制折行确保每行不要超过 100列，换行后至少额外缩进 8个空格。

示例

```
grey_wolf.eat(white_sheep);
hungry_lion.eat(
        big_grey_wolf,
        little_white_sheep,
        quick_brown_fox,
        lazy_dog);
```

3.2.3.7. if/while ????

·[强制] [RULE076] if语句的分支部分必须使用大括号包围 ·[强制] 如果条件表达式很长很复杂，在低优先级运算符前换行，换行后运算符 (`||` or `&&`)放在新行最前面，并额外缩进 8个空格。 ·[强制] 连续的 `if ... else if ...` 判断，原则上应该以 `else` 语句结束。若该条件合法但没有相应动作，通过注释标明 `//pass` 或 `//do nothing`；若是不合法情况，应报错。

解释 尽管有时候 `if` 里只有一条语句，但以后重构时可能会改变，使用大括号可以避免不必要的错误。 连续的 `if ... else if ...` 常常是对情况作完全的分类讨论。为了使遗漏的情况能够容易发现，应使用一个无条件的 `else` 语句来检查一下。

示例

```
if (you.come_from(EARTH)) {
    return "hello";                  // 单语句也要用{}括起来
} else if (you.come_from(MARS)) {
    return "@%!%#@!~&$%$";           // 火星文
} else {                             // 虽然不大可能，但假如用户在骗我们呢?
    throw YouCheatMeException() << BSL_EARG << "your name:" << you.name(); // 留下案底了
}

// Demo 条件表达式换行
if (you.com_from(EARTH)
        || you.com_from(MARS)
        || you.com_from(VENUS)) {
    return "you come from solar system";
}
```

参考

> 1TBS风格

3.2.3.8. switch??

- ·[强制] switch 语句的每一个 `case` 语句都必须以 `break` 语句或 `return` 语句结束。
- ·[强制] 如果使用分组，每组除最后一个 case 之外不允许有任何代码。
- ·[强制] switch 语句原则上应该有一个 `default` 语句，如果没有动作，通过注释标明 `//pass` 或 `//do nothing`
- ·[强制] switch 对应的 `case` 语句需要与 `switch` 对齐， 不要增加额外缩进

解释 `case` 语句使用 `break` 或 `return` 来避免代码跨 `case` 执行导致错误。 switch 用于对一个变量或表达式的值进行完全的分类讨论。为避免遗漏，应通过一个 `default` 语句检查一下。

这样写是推荐的:

```
switch (cur_char) {
case '\"':
case '\'': {
    return "it's a string!";
}
default:
    printf("not found yet");
    break;
}
```

这样写是危险的:

```
switch (cur_char) {
case '\"':
    printf(“wow!”); //没有break或return
case '\'':
    return “it’s a string!”;
}  // 没有default
```

3.2.3.9. ????

·[强制] [RULE083] 循环语句的循环体必须使用大括号包围。 ·[强制] [RULE084] `do-while` 循环里中的 `while` 与右大括号在同一行。

解释 尽管有时候循环体只有一条语句，但以后重构时可能会改变，使用大括号可以避免不必要的错误。 若 `do-while` 循环中的 `while` 语句独自一行，容易错看成与上边的 `do{...}` 无关的独立语句

示例 空循环

```
while (iter.next()) {
    // pass
}
```

一条语句的循环

```
for (size_t i = 0; i < N; ++i) {
    sum += arr[i];
}
// do-while
do {
    ...
} while (++it != end);  // 正确
do {
    ...
}
while (++it != end);    // 错误，容易看成一条和上边语句无关的独立循环语句。
```

参考

> 1TBS风格

3.2.4. ?????

·[强制] [RULE085] `if/switch/while/for/catch`

与后边的圆括号之间加一个空格，圆括号内侧与判断表达式之间不加空格 ·[强制] [RULE086] `for`

语句圆括号内分号前不加空格，分号后加一个空格 ·[强制] [RULE087]

函数调用中，函数名和圆括号之间不要加空格 ·[强制] [RULE088]

类继承与构造函数的初始化列表的冒号前后加一个空格 ·[强制] [RULE089]

逗号表达式或参数列表中，逗号前不加空格，逗号后加一个空格 ·[强制] [RULE090]

一元运算符前后不加空格。二元、三元表达式前后各加一个空格。 ·[强制] [RULE091]

成员访问或作用域运算符前后不加空格。如: `a.b, a->b, a.*b, a->*b, a::b` 。 ·[强制] [RULE092]

圆括号和方括号运算符前后和内部都不加空格。

示例

```
if (is_good && is_powerful)    // 正确
if(is_good && is_powerful)     // 错误 if之后没有空格
if ( is_good && is_powerful ) // 错误, 括号内加了额外空格

switch (type)       // 正确
while (condition)   // 正确
catch (std::exception& ex)        // 正确
for (int i = 0; i < 10; ++i)      // 正确
for (int i = 0;i<10;++i)          // 错误, 分号后无空格
for (int i = 0 ; i < 10 ; ++i)    // 错误, 分号前有空格

call_some_func(arg1, arg2, arg3);   // 正确
call_some_func (arg1, arg2, arg3);  // 错误, 函数名和圆括号之间有额外空格
call_some_func( arg1, arg2, arg3 ); // 错误, 圆括号内部有额外空格
call_some_func(arg1,arg2,arg3);     // 错误, 逗号后没有空格
call_some_func(arg1 , arg2 , arg3); // 错误, 逗号前有额外空格

template <typename T>     // 正确
template<typename T>      // 错误, template和<之间没有空格

class Derived : public Base // 正确
class Derived:public Base   // 错误, 冒号前后没有空格

MyClass::MyClass() : _member_var(0) // 正确
MyClass::MyClass():_member_var(0)   // 错误, 冒号前后没有空格

++i; !i; ~i; *i; &i;        // 正确
++ i; ! i; ~ i; * i; & i;   // 错误,一元运算符不应该加空格
```

```
a + b; a ~ b; a || b; a << b; a = b; a %= b;    // 正确
a+b; a~b; a||b; a<<b; a=b; a%=b;                // 错误， 二元运算符前后应该加空格

a ? b : c    // 正确
a?b:c        // 错误， 三元运算符前后应该加空格

a.b; a->b; a.*b; a->*b; a::b;               // 正确
a . b; a -> b; a .* b; a ->* b; a :: b;     // 错误， 成员访问或作用域运算符前后都不应该有空格

xx_comparator(a, b); xx_map["key"];         // 正确
xx_comparator (a, b); xx_map ["key"];       // 错误，圆括号运算符和方括号运算符前边不应该加空格
xx_comparator( a, b ); xx_map[ "key" ];     // 错误，圆括号运算符和方括号运算符内部不应该加空
```

3.2.5. ????

·[强制] [RULE093] 一行代码不应该超过 100 个字符，对于格式化字符串，可以选择在适当位置截断，也可以不截断。

解释 过长的代码有几个坏处：

一般长代码都是由多个语句组成的，这些混合在一行的语句往往难以单步调试，难以跟踪

还有相当一部分程序员在使用小液晶屏或者 12寸的 laptop，请考虑一下其阅读代码时的感受。

对于格式化字符串，可以通过截断来减少语句长度，但必须谨记绝不要误加 ,

示例 如何截断格式化字符串（以把异常信息打印到日志为例）

```
UB_LOG_WARNING(
    "An exception[%s] was thrown from[%s:%d:%s] "     // 被截断的格式化字符串之间不能有','号!
    "with a message[%s]! stack_trace:%s%s",
    e.name(), e.file(), int(e.line()), e.function(),              // 推荐的做法：格式化字符串与可变参数逐行对应
    e.what(), e.get_line_delimiter(), e.stack()
    );
```

3.2.6. ????

·[建议] 编写小型的、功能单一的函数，建议拆分超过 100行的函数 ·[强制] .h 中不要定义复杂的函数 ，复杂函数如果必须在头文件中 ，应该放在 .hpp 后缀的头文件中

解释

短小的函数比长函数更容易理解，也更容易保证正确性。其中局部变量的作用域更小，耦合度更低，也更有利于单测和重构。

3.2.7. ?????

·[强制] 函数 return返回值不要用圆括号包围

解释 return 不是一个函数，所以不要加 ()

示例

```
return (ret_val);
```

3.2.8. ????

·[建议] 函数的参数顺序，建议先安排输入参数，再安排输出参数。 ·[建议] 输入参数用 const 引用（对象）值类型可以直接值传递。输出参数使用指针。如果输入参数可能为空，允许使用 const 指针类型
·[强制] 没有使用到的函数参数，把参数名注释起来，不允许未命名参数出现 ·[建议]
如果函数传入参数是指针，需要检查指针是否非空

解释 限制传入传出参数的类型，让其他人阅读代码的时候，很容易看到哪些参数是输入，哪些是输出；哪些是会被修改的

示例

```
/*
C/C++ 函数参数分为输入参数， 输出参数， 和输入/输出参数三种.
输入参数如果为对象时，需要使用 ``const`` 引用. 值类型（基本类型+指针）即可以使用值传递，也可以使用const引用，优先保持代码上下文一致。
如果函数需要修改变量的值(通常输出参数或者输入输出参数为多), 则必须采用非 ``const`` 指针
定义函数时， 参数推荐的顺序依次为: 输入参数， 然后是输出参数. 输入/输出两用参数 (通常是类/结构体变量) 把事情变得复杂， 为保持和相关函数的一致性， 也允许使用.
*/
// 输入参数对象类型使用const引用
void foo(const std::string& input1, const MyClass& input2);


// 输入值类型可直接传递
void foo(int input1, float input2);


// 输入参数可以是const指针，特别对于有可能是空指针的输入
void foo(const MyClass* input1);


// 输出参数不论是对象还是值类型都是用指针传递
void foo(int* output1, MyClass* output2);


// 对于字符串类型的传出参数，建议使用std::string*，不要使用char*。因为使用char*需要保证可用的buffer长度，需要额外的传入参数。
void foo(std::string* out);


// 同时具有输入参数和输出参数
```

```
void foo(const InClass& input1, OutClass* out1);


// 多个输入输出参数
void foo(const InClass& input1,  // 输入使用const引用
        int input2,  // 输入使用POD类型传值
        OutClass* outpu1);  // 使用指针传出对象


// 如果存在输入输出参数，输入输出参数放在输入参数后，输出参数前
void foo(const InClass& input1,
        int* inout,  // 输入输出参数
        OutClass* output1)
禁止未命名参数：
// Always have named parameters in interfaces.
class Shape {
public:
    virtual void rotate(double radians) = 0;
}


// Always have named parameters in the declaration.
class Circle : public Shape {
public:
    virtual void rotate(double radians);
}


// Comment out unused named parameters in definitions.
void Circle::rotate(double /*radians*/) {}
```

3.3. ??/??

3.3.1. ??????

·[建议] 命名应当尽可能有描述性，不使用非通用的缩写 (尤其是省一个字符的缩写如 `creat`,`usr`等)
，不使用有歧义的缩写，不使用任意的无意义的字符 ·[建议] 标识符的作用域越大，命名就应该越清晰 ·[强制]
全局可见的，却又无法通过命名空间约束的标识符命名，必须以库名作为前缀，以避免冲突。 ·[建议]
可能使用多个单位名称的变量（如表示时间的变量），应以单位名称缩写为后缀。 ·[建议]
例外情况：如果为某种类似于已存在的实体命名，使用它们的命名规则（比如说模仿 STL 的容器使用 STL
的命名规则）

解释 好的命名应该是容易理解、记忆且不容易冲突的。非通用的缩写不便于理解和记忆。
很多编程规范没有指出标识符命名与作用域的关系，但其实这样是很重要的。比如说，一个 `VScriptEvaluator`
对象，作为一个短函数的局部变量时，命名为 `vse` 是合适的，但做为类数据成员就很牵强了（此时可命名为

`_evaluator` [illegible] *[FALL BACK]* [illegible] `extern “C”` [illegible] /
[illegible]
[illegible]

[illegible]

```
// [illegible] [illegible] [illegible];
// [illegible]
int num_errors;                  // Good.
int num_completed_connections;   // Good.

// [illegible]
int n;              // Bad - [illegible]
int nerr;           // Bad - [illegible]
int n_comp_conns;   // Bad - ambiguous abbreviation.

// [illegible]: [illegible];
// [illegible]:
int num_dns_connections;  // [illegible]"DNS"[illegible]
int price_count_reader;   // OK, price count. Makes sense.

// [illegible]
int wgc_connections;      // wgc [illegible]??
int pc_reader;            // pc[illegible]!

// [illegible]
int error_count;          // Good!
int error_cnt;            // Bad! [illegible], [illegible];
```

3.3.2. ????

·[illegible]] [illegible] .h [illegible] .hpp [illegible] C++ [illegible] .cpp [illegible] C [illegible] .c[illegible] ·[illegible]] [RULE100] [illegible] , [illegible] _[illegible] ; [illegible] :this_is_my_awesome_file.cpp ·[illegible]] [illegible] `<[illegible]>_test.h(.cpp)` [illegible] ·[illegible]] [illegible] .h [illegible] .hpp [illegible] ·[illegible]] [illegible] lib[illegible] `<[illegible]>_<[illegible]>.h` [illegible] api [illegible] include [illegible] api [illegible] `mylib.h,mylib_utils.h,mylib_api.h`

[illegible] [illegible]

3.3.3. ??????

·[强制] [RULE102] 使用下划线分隔的全小写命名法命名命名空间 ·[建议] [RULE103]

命名空间名可以使用缩写，同时应保证简短、不易冲突但同时富有意义 ·[建议] 推荐使用库名作为命名空间名

解释 命名空间名是该命名空间下所有成员（类、游离函数、常量等）的公共前缀。由于不允许使用 using namespace 语句，过长的命名空间名会带来使用上的不便，因此应保证简短、不易冲突。

库名与命名空间名保持一致易于记忆。

3.3.4. ????

·[强制] [RULE103] 函数命名使用下划线分隔的全小写命名法 ·[建议] 函数通常使用动词短语命名 ·[强制]

类私有变量的 accessor 命名 : `getter: my_member_variable()`, setter: `set_my_member_variable(), mutable: mutable_my_member_variable()`

解释 accessor 命名主要考虑兼容 Google protobuf 的命名形式

示例

```
class Person {
public:
    ...
    const IdCard& id_card() const { return _id_card; }               // getter
    IdCard* mutable_id_card() { return &_id_card; }
    void set_id_card(const IdCard& id_card) { _id_card = id_card; }  // setter

private:
    IdCard _id_card;
    int _age;
    int get_age() const { return _age; }
};
```

3.3.5. ?????????

·[强制] [RULE105] 自定义类类型使用首字母大写的驼峰命名法命名，一般不使用前缀 ·[强制] [RULE106]

枚举类型成员，使用全大写蛇形命名法（即全部字母大写，单词间用下划线分隔）。 ·[建议] 接口类建议使用 Interface 作为后缀命名

定义 下述自定义类型指 `enum`、`struct`、`class`、`typedef` 的各种类型、 `template` 参数类型。

示例

```
enum Color {
    RED,
    BLUE,
    GREEN,
```

```
    COLOR_COUNT
};
struct Point {
    int x;
    int y;
};
```

3.3.6. ??????

·[强制] [RULE107] 尽可能不使用全局变量，如果必须使用，必须以 g为前缀，而且必须足够长以避免名字冲突

·[强制] [RULE108] 全局变量使用下划线分隔的全小写命名法命名

3.3.7. ??????

·[强制] [RULE109] 局部变量名使用下划线分隔的全小写命名法命名 ·[建议] 假如局部变量作用域很小，可以适当使用缩写

解释

变量的命名，在清晰性与简洁性之有一个折衷，其主要标准是它的作用域大小，比如说，在一个短循环里面，用 i,j,k 表示循环变量是问题不大的，但如果在一个几百行的函数的开头定义了变量 i,j,k ，就很容易导致冲突和误用了。这时就应当选择更有意义的名字，比如 `cur_user_index` 等

3.3.8. ??????

·[强制] [RULE110] 全局静态变量 `s_` 作为前缀 . ·[强制] [RULE111] 类静态成员使用 `_s_` 前缀的下划线分隔的全小写命名法命名

解释 `s_` 中的 s 代表 static, 这样命名可以让读代码的人很快知道这个变量是什么类型的 ; 使用 `_` 前缀表示类变量是规范，使用 `_s_` 前缀以区分静态数据成员与非静态数据成员

3.3.9. ?/?????????

·[强制] [RULE112] 类数据成员使用下划线作前缀表示 ,命名用小写单词中间下划线分割，如 `MyClass::_my_attr` ·[强制] [RULE113] 结构体数据成员一般不使用下划线前缀，命名用小写单词中间下划线分割 ,如 `MyStruct::my_attr`

3.3.10. ????

·[强制] const 常量与枚举常量都使用下划线分隔的全大写命名法命名

3.3.11. ?/?????

·[强制] 尽可能不定义宏，如果必须，使用下划线分隔的全大写命名法，并且必须足够长，以库名为前缀，使人们不至误用，并尽可能减少冲突（如 `MY_LIB_MY_MACRO_THAT_SCARES_SMALL_CHILDREN`）

3.3.12. ????

·[建议] 尽可能使用原生类型 int。 ·[建议] 确定需要不同大小的变量，推荐使用 `<stdint.h>` 中定义的长度明确的整形类型，例如 int64_t, int32_t 等。

解释 没有长度语义的整形，可以使用 int，例如一些用于函数内部的表示状态的变量等。避免使用形如 `unsigned long long` 这样的整型类型来表示有长度语义的整形。

3.4. ??

3.4.1. ??????

·[建议] 注释语言与编码 ：鼓励采用英文书写 ，并建议采用 UTF-8 编码 . ·[建议]
注释用语应该清晰明确，避免有二义性，避免不常用的缩写。 ·[强制]
多行注释必须写在被解释内容的上方。单行注释可以写在被注释语句的上文或右方 ·[建议]
推荐在编写代码的同时编写注释，修改代码的同时修改注释，无用的注释要及时删除，以保证注释与代码同步

解释

鉴于目前程序员的平均英语水平，若必须使用英文注释，应避免使用难以理解的语句，并保证用语准确且无二义性。 错误的或过时的注释比没有注释更糟，应保证注释与代码的同步。

3.4.2. ????

·[建议] 注释应该是有意义的，说明语句背后的意图，而非语句的简单重复 ·[建议] 使用 `TODO`
来标记出临时添加的代码，或者能用但未臻完美的代码 ; ·[建议] `TODO` 注释格式 : 全大写字符串 ，
在随后的圆括号里写上负责人邮件地址 ，多个负责人用 *连接 ·[建议] 用 `DEPRECATED`
注释表明不推荐使用的过时接口 ; 新增代码不应该调用被 `DEPRECATED` 标示的接口 ·[建议]
诡异的、不明显的、有趣的、重要的代码应该伴有注释 ·[建议]
建议对较长的块，包括命名空间、类、函数定义、循环、分支等，在其末尾使用注释标明。 ·[强制]
会抛异常的函数调用使用注释注明

3.4.3. ?????

·[强制] 每个文件都必须有文件声明放在文件头部 ，按如下内容组织 : 1)版权声明 ; 2)许可证 (可选) 3)作者 4)
代码功能 /用法说明 ·[强制]
对基础库，以及涉及各模块边界、接口的文件、类、公有函数、类数据成员、全局常量、全局变量都必须有文档化注释 ·[强制]
文档化注释必须标明作者和简要介绍，公有函数必须解释其输入参数、输入参数合法取值、输出参数、返回值、抛出的异常。 ·[建议]
特别的函数，如有前置条件的函数、没有强异常安全保证的函数、线程不安全的函数、不检查指针的函数、较高时空复杂度的函数、不推荐使用的函数、有其它注意事项的函数，必须在文档化注释中标明。

解释 文档化注释是最贴近代码的文档，是使用者了解程序的最重要途径之一，因此必须重视文档化注释。

3.5. ????

3.5.1. ??????

·[强制] [RULE122] 一行内只应声明一个变量

解释 混合声明值变量、指针变量、引用变量难于理解，容易出错。 一行只声明一个变量便于阅读，不易出错

3.5.2. ??????

·[强制] 局部变量应尽量延迟到第一次使用处声明

例外：若类变量第一次使用在循环内部，且其构造、析构代价较大的话，可以在循环前声明。 ·[强制] [RULE123] 局部变量应在声明处赋初值，指针类型局部变量必须在声明处赋初值。

解释 变量作用域越小，出错的可能性就越小，也更容易理解和记住。

避免没有赋初值而处于非法状态的变量，尤其要避免野指针。

参考

> Effective C++, item 26: Postpone variable definitions as long as possible

3.6. ???????

·[强制] [RULE124] 禁止使用 `auto_ptr` ·[强制] [RULE125] 禁止使用 `strcpy()/strcat()/strdup()/sprintf()` 等没有越界检查的函数。 考虑使用 `snprintf()/strncat()/strndup()/snprintf()` 替代之 ·[强制] [RULE126] 禁止使用 `strncpy` 实现有问题的函数。 用 `ullib` 中的 `ul_strlcpy` 函数替换

解释 `auto_ptr` 的赋值会导致被赋值者的值被改变，同时也使其无法与容器兼容。不当使用 `auto_ptr` 会造成严重的后果。 `strncpy()` 在缓冲区长度不足时不保证以 `\0` 结尾，缓冲区长度过长时又会填满整个缓冲区而造成性能问题，而且由于是逐字节复制，效率很低，应避免使用。 `snprintf` 能够解决前两个问题，但复制效率依然很低。

4. ????

4.1. ????/?????

·[强制] 本规范不适用于第三方代码 ，修改时请遵守第三方代码自身的编码风格 ·[强制] 显式或者通过宏等隐式的定义派生自第三方代码中所申明的类 ,必须遵守对应第三方代码自身的编码风格

解释 有部分第三方库需要使用通过继承的其提供功能类的方式使用 . 比如 ，编写单元测试中最频繁使用到的框架 -- gtest. 为了保持代码风格的一致性 ，我们强制要求 ， 派生自第三方代码中定义的类必须遵循对应第三方代码的编程风格 .

代码示例 ：

```
class FooEnvironment : public testing::Environment {
public:
    virtual void SetUp() {
        std::cout << "Foo FooEnvironment SetUP" << std::endl;
    }
    virtual void TearDown() {
        std::cout << "Foo FooEnvironment TearDown" << std::endl;
    }
};

// Example:
// template
class FooTest : public testing::Test {
 public:

  ...
  typedef std::list List;
  static T shared_;
  T value_;
};
```

4.2. ????????

·[建议] 如果修改一个风格非常一致的老模块，而且它的风格跟我们的编码规范冲突，请保持老模块的风格一致性 ;

4.3. Windows ??

Windows 下写 C++ 代码请遵循本节的规范要求。对于本节没有要求的部分，请遵循 C++ 编码规范。

4.3.1. ????

·[强制] [WCPP001] [WCPP002] [WCPP003]文件名、函数名 ,自定义类型采用大驼峰命名法 ·[强制] [WCPP004] 一般变量名称采用小驼峰命名法 ,成员变量采用前缀 m_,后跟小驼峰命名法 ·[建议] 类名以 C 作为前缀，后跟大驼峰命名法 ·[建议] C++程序不建议采用匈牙利命名法，有两个匈牙利命名法可以保留: `m_xxxx` 表示类的成员变量， `g_xxx` 表示全局变量 ·[建议] 接口名以 `I` 作为前缀，后跟大驼峰命名法

定义 大驼峰命名法 ：每一个单词都首字母大写 ，如 EatApple 小驼峰命名法 ：第一个单词首字母小写，之后的单词首字母大写 ，如 eatApple

4.3.2. ????

4.3.2.1. ??

·[强制] [WCPP006] 大括号 `{`、`}` 应各独占一行并且位于同一列，同时与引用它们的语句左对齐 ·[强制] [WCPP005] 在其结构体内即使只有一条语句，也必须使用大括号，不能省略

示例：

```
if (i == j)
{
  //在函数体的开始、类的定义、结构的定义、枚举的定义
  //以及i、for、do、while、switch、case语句中的程序都要采用如上的缩进方式。
  i++;
}
```

·[强制] 用括号 ()明确表达式的操作顺序，避免使用默认优先级 ，比如 `int x = 3 & (4<<32)`;

4.3.3. ?WINAPI???????VS?IDE????

·[强制] 对于 WINAPI 接口使用时保持相同的风格，如 DWORD，HANDLE 等 ·[建议] 按 C++ 规范要求使用 include guard，亦可使用 #pragma once

示例：

```
#ifndef ABC_H
#define ABC_H
#endif
```

4.4. ????"??"??

·[强制] 如果需要违反本规范约定的 "建议 "级别的规则， 必须给出注释说明原因