C++ 编码规范

👋 说明：

- 本编码规范仅在新的 C++ 工程中使用。在一个已有的工程中编写代码时，以与该工程现有编码风格保持一致为原则。
- 本编码规范中的部分条款代表着更好的编码习惯，而另一些仅仅是为了统一风格而做的人为规定。
- 本编码规范中的所有条款均可以在特殊情况下违背，但必须加上必要的注释说明理由。

一. 编码规范

1. 命名

1.1 文件命名

- 文件名全部使用小写字母。

👋 **示例：**kctlinechartview.h

1.2 类命名

- 类的命名 **使用前缀＋驼峰命名法**。
- 当一个类需要暴露给其它工程使用时，应加上工程约定的前缀，前缀全部使用大写字母。
- 类命名尽量使用完整英文单词，除非是常见的缩写

👋 **示例：**

C/C++

```
KCTLineChartView
```

- Com 接口命名在普通前缀前加 I 作为标志。

👋 **示例：**IKCTLineChartView

1.3 其他类型命名

- **struct**、**typedef** 与 **enum** 的命名原则与类相同。
- enum 成员**采用大写字母开头的驼峰写法**，**且应在命名中体现 enum 名称**，具体位置不做强制要求，但同一个 enum 内的命名规则要保持一致。

👋 **示例 枚举类型**

C/C++

```
enum ActionType
{
    InsertActionType,
    DeleteActionType,
    MoveActionType
};
```

1.4 变量命名

- 变量命名**使用小写字母开头的驼峰写法**，不加类型信息前缀，命名应使用能表达变量涵义的完整英文单词。

👋 **示例：**

int index = 0;

KCTLineChartView *lineChartView = NULL;

- **仅允许在 for 单层循环中的 i 和表示坐标的 x、y、z 上使用单字母变量，其他情况下不允许使用。**
- **非 static 的 类成员变量** 前加上小写字母 **m_前缀**，**static 的 类成员变量前** 加上 **小写字母 s_前缀**，这两种情况下变量名的 **第一个单词首字母小写，后面单词首字母大写**。

👋 **示例**

C/C++

```
class GoodClass
{
private:
    int m_dataMember;
    static int s_staticDataMember;
};
```

1.5 函数命名

- 除构造函数和析构函数外，**类成员函数使用小写字母开头的驼峰写法**。
- 非类成员函数使用大写字母开头的驼峰写法。

> 👋 **示例：**
>
> ```
> lineChartView->lineCount();
> GetObjectCountInDocument();
> ```

- 名词属性的类成员函数命名分两种情况：
  - 若欲获取的信息当成返回值返回，则命名方式为名词或名词＋修饰语 的样式。

> 👋 **示例：**
>
> ```
> QString title = lineChartView->getTitle();
> ```

  - 若欲获取的信息通过某个参数返回，则命名方式为 get+ 名词（＋修饰语）的样式。

> 👋 **示例：**
>
> ```
> lineChartView->title();
> lineChartView->subViewAtIndex(index);
> lineChartView->getSubViews(&outPointer);
> ```

- 动词属性的类成员函数命名方式为动词＋宾语，若宾语是 this, 应省略。

> 👋 **示例：**
>
> ```
> lineChartView->removeAllSubViews();
> lineChartView->setTitle(L"Hello World");
> lineChartView->initialize();
> ```

- 形容词属性的类成员函数命名方式为 is/has 等修饰语＋形容词 / 名词。

👋 **示例：**

```
lineChartView->isVisible();
lineChartView->hasTitle();
```

- 返回对象指针的函数若内部分配了内存，应使用 copy, create 等关键字把该信息反映在函数名上。

👋 **示例：**

```
lineChartView->subViews();        // 未重新分配内存
lineChartView->copySubViews();    // 重新分配了内存
```

1.6 宏命名

- 原则上**宏的命名全部大写，如有必要单词间用下划线分隔**。如无必要，不把一段代码定义成宏。

👋 示例：

```
#define SOMETHING_USED
```

1.7 全局变量

- 关于使用全局变量的注意事项见后文。
  - **全局变量前加小写字母 g_ 前缀**
  - **static 全局变量前加小写字母 gs_前缀。**

👋 **示例：**

```
bool g_inFileOpening = false;
```

```
static bool gs_staticInFileOpening = true;
```

1.8 名称空间命名

- 名称空间使用剪短的全小写英文单词命名。

> 👋 **示例：**

C/C++

```
namespace  chart
{
    ....
}
```

1.9 不允许使用 My 或自己的姓名作为以上命名的前缀

- **禁止下列写法**

C/C++

```

Class MyStringProject
{
public:
...
};
```

2. 头文件

2.1 头文件包含保护

- 在所有头文件中均使用 **#ifndef + #define + #endif** 来避免该文件被重复包含
- 宏的命名方式为 **__PROJECTNAME_FILENAME_H__**。

👋 **示例：**

- KProject 项目有头文件名为 KHeader.h，则文件头部加上

  ```
  #ifndef __KPROJECT_KHEADER_H__
  #define __KPROJECT_KHEADER_H__
  ```

- 末尾加上

  ```
  #endif //__KPROJECT_KHEADER_H__
  ```

2.2 前置声明

- 当使用一两个前置声明就能编译通过时，不要在头文件里包含另一个头文件。

2.3 头文件包含顺序

- 在代码文件里包含多个头文件时，应按照如下顺序将头文件分组，每组之间使用空行隔开：
  a. 预编译头文件，通常是 stdafx.h
  b. 与代码文件同名的头文件
  c. 系统头文件
  d. WPS 内部其他工程头文件
  e. 本工程其他头文件

示例：

- KProject 工程中 用到如下头文件
  - kclass.cpp 实现了声明在 kclass.h 内的类，
  - C++ 标准库中的 Vector 容器、
  - ExternalProject 工程中的 externalclass.h 和 externalutils.h
  - KProject 工程中的 kfriendclass.h、kfilelohelper.h 与 kstringutils.h，其包含顺序如下：

C/C++

```
#include "stdafx.h"
#include "KClass.h"

#include <vector>

#include "externalclass.h"
#include "externalutils.h"

#include "kfirendclass.h"
#include "kfileiohelper.h"
#include "kstringutils.h"
```

3. 类

3.1 初始化

- **类的所有成员变量必须初始化，只有成员变量没有函数的类，必须定义默认构造函数。**
- 除非有特殊需求，在类的构造函数中仅进行不涉及具体功能的初始化操作，例如为成员变量赋零。
- 较复杂的初始化操作，应放在一个单独的 init() 方法中，**由类实例的创建者负责调用**。

> 👋 **解释：**
>
> - **构造函数内难以报告错误**，且构造函数中对虚函数的调用不会派发给子类，所以较复杂的初始化操作应在构造完成之后进行

3.2 初始化

- 若没有明确的将单参构造函数用于隐式类型转换的需求，应使用 explicit 关键字。

3.3 拷贝构造函数、赋值运算符

- **若没有明确的对类进行拷贝的需求，应在 private 段中声明拷贝构造函数和赋值运算符。**

3.4 继承

- **只使用 public 继承。**
- 在语义明确的时候从父类继承具体实现，其他时候从抽象接口继承。
- 若类有虚函数，则析构函数也定义为虚函数。
- **父类中声明为 virtual 的函数，子类声明中要明确标明为 virtual 以及 override。**

3.5 运算符重载

- **尽量避免运算符重载**，除非是为了在容器类中使用而必须实现的。

3.6 访问权限声明

- 类声明中按 public, protected, private 的顺序声明函数和变量，
- 每个关键字仅占用一段，每一段中的声明顺序为：
  - typedef, enum,
  - Q_OBJECT 及类似声明
  - 嵌套类，
  - 常量，
  - 构造函数， 虚构函数，成员函数，数据成员。

3.7 友元

- **友元的定义和友类的定义应放在同一个文件中。**

4. 函数与实现

4.1 参数

- **参数排列顺序为输入参数在前，输出参数在后。**
- **引用作为输入参数时应配合 const 使用。不使用缺省参数。**

- **不允许一个参数即作为传入参数，又作为传出参数**。

4.2 内联函数

- 内联函数不超过 10 行，推荐仅对 1 行的函数进行 inline。
- 不使用循环或 switch...case，不进行递归。
- 在类定义体内实现的函数，不要加 inline 关键字。

4.3 异常处理

- **避免使用异常（try...catch）。使用 ATL 和 STL 时，关闭异常或限制异常作用域**。

4.4 类型编程

- 不使用运行时类型信息。
- 明确使用 static_cast，const_cast 进行类型转换，避免使用 dynamic_cast，小心使用 reinterpret_cast。

4.5 宏

- 尽量使用 const、enum、inline 替代 #define。

4.6 类数据成员的引用

- **避免成员函数返回指向类成员的指针或引用**。

5. 注释

5.1 文件头

文件头注释应包括文件名，创建者，创建时间，功能描述和版权信息。

👋 **示例**

C/C++

```
// ------------------------------------------------------------
// kctcodingguideline.h
// 创建者： Tom Cat
// 创建时间： 2013/5/24
// 功能描述： The coding guideline for Chart project in WPS
// Copyright 2013 Kingsoft
// ------------------------------------------------------------
```

5.2 代码注释

- **注释使用 // 风格，应简洁清晰，不写没必要的注释。代码中不是很简单直观的地方需要增加注释**，

💡 **例如：**

- 故意违背编码规范之处。
- 分成多步完成的任务。
- 较复杂的逻辑。
- 和常理不符的代码。
- 比较重要，需要引起注意的地方。

5.3 注释语言

- 默认注释语言为 **中文**。

  a. **注释中的工程师姓名**

- 在且仅在两种情形下把姓名加入注释中：

  a. 文件头中的创建者信息。

b. 做 TODO 注释时。

> 👋 **示例：**
>
> //TODO(Tommy Zhang): 删掉这里的全局变量！

5.4 预处理宏注释

- 中间代码段较长的#else 与#endif 之后用注释标明宏的名字。

> 👋 **示例：**

C/C++

```
#if TARGET_PLATFORM_WINDOWS
.................
#else //TARGET_PLATFORM_WINDOWS
.................
#endif //TARGET_PLATFORM_WINDOWS
```

6. 格式

6.1 对齐与缩进

- 使用 Tab 进行对齐。
- **花括号上下对齐，不允许将左花括号放在 if 等语句的末尾**：

> 👋 示例如下：

C/C++

```
void KFunction(InputType type)
{
        static const int someInt[] = {80, 40, 40, 50, ... 94, 94, 47};
        static const ScopeType scopeType[] =
        {
                wpsFindScope_MainText,
                wpsFindScope_Selection,
                wpsFindScope_HeaderFooters,
                wpsFindScope_Footnotes,
                wpsFindScope_Endnotes,
                ...
                wpsFindScope_Comments
        };
        switch (type)
        {
        case TypeA:
                break;
        case TypeB:
                break;
        default:
                break;
        }

        if (...)
        {
        }
        else
        {
        }

        do
        {
        }
        while (...);

        for (int i = 0; i < 10; ++ i)
        {
```

```
38          }
39  }
```

- **构造函数初始化列表和多个基类的中每一行逗号放前面，和冒号对齐。**

> 👋 **示例：**

C/C++

```
 1  KGroupGrid::KGroupGrid(QWidget *parent)
 2                  : QWidget(parent)
 3                  , m_scrollBar(Qt::Vertical, this)
 4                  , m_headerHeight()
 5                  , m_itemSize(80, 64)
 6                  , m_showTooltip(false)
 7                  , m_flatFrame(true)
 8                  , m_showSeperatorLine(true)
 9                  , m_scrollBarPolicy(Qt::ScrollBarAsNeeded)
10          #if X_OS_WINDOWS
11                  , m_firstRow(0)
12          #endif
13          {
14          }
15          class KxWppViewPages
16                  : public QStackedWidget
17                  , public KFakeUnknown<IShellPagesGetter>
18                  , public KFakeUnknown<IROShellPages>
19                  , public KxWppViewPagesCoreNotify
20          {
21          };
```

- **只有一行代码的 if 语句不加花括号。**

C/C++

```
        if (inputType == badType)
                DoSomethingBad();

        if (inputType == badType)
                DoSomethingBad();
        else
                DoSomethingElse();
```

- **当 if 语句中某一个分支有一行的以上代码时，if 的所有子句都要加括号。**

C/C++

```
if (inputType == badType)
{
        DoSomethingBad();
}
else
{
        DoSomethingElse1();
        DoSomethingElse2();
}
```

- 推荐一行代码不超过 120 个字符。
- 函数参数列表过长时，应换行：**返回值类型、左括号应保持与函数名同一行，具体换行的方案有两种：**

C/C++

```
HRESULT KClass::method(ParameterOne param1,
                       ParameterTwo param2,
                       ParameterThree param3)

HRESULT KClass::method(ParameterOne param1,
        ParameterTwo param2, ParameterThree param3)
```

- **函数类型为 const 时，const 关键字与最后一个参数同行**

C/C++

```
HRESULT KClass::constMethod(ParameterOne param1,
                            ParameterTwo param2,
                            ParameterThree param3) const
```

- **所有预编译宏顶格对齐。**

C/C++

```
        void KClass::methodWithMacros()
        {
            static int integer1 = 0;
            integer1 ++;
            if(integer1 >= 10)
            {
        #if PLATFORM_1
                integer1 = 1;
        #else
                integer1 = 2;
        #endif
            }
        }
```

- 分为多行的布尔表达式中 && 与 || 置于行首。
- **换行应在运算符优先级最低的地方进行，尽可能避免把配对的括号分成两行**。
- 当条件比较复杂时，应将高优先级的运算用括号明确标识出来。

C/C++

```
1 if ((valueGood >= somethingGood || otherContion)
2         && (valueBad + 1) <= somethingRealyBad)
```

- **命名空间（namespace)内容不缩进**

6.2 空格

- **函数调用的括号前后之间不加空格，参数之间逗号之后加一个空格**。

C/C++

```
1 void KFunction(ParameterOne param1, ParameterTwo param2)
```

- if、while、for 等关键字与括号之间加一个空格，括号后不加空格。

C/C++

```
1 if (NULL == thePointer)
2
3 for (int index = 0; index < maxIntValue; index++)
```

- 使用 Tab 键无法刚好对齐时，使用空格补齐。
- 没有参数的函数括号内不加空格。

- **二元运算符前后各加一个空格；**
- 自增减运算符与分号间不加空格，与变量之间也不加空格。

C/C++

```
1 if (inputType1 == goodType && inputType2 == badType)
2
3 for (int index = 0; index < MAX_INT; index++)
```

- 引用符号（. ->）前后不加空格。

C/C++

```
1 theSmartPointer->value();
```

- 地址、引用运算符（*, &）后不加空格；

C/C++

```
1  *integerPointer = 1;
```

- 声明指针类型的 * 号前加一空格，后边不加空格

C/C++

```
1 KClass *goodClass;
```

7. 作用域

7.1 嵌套类

- **不需要暴露嵌套类作为接口的时候，将嵌套类声明为 private。**

7.2 局部变量

- **局部变量应在将要使用时进行声明，声明的同时初始化。**

C/C++

```
        void SomeMagicalFunction()
        {
                const int magicNumber = 1;
                int integer1 = magicNumber;
                integer1 ++;
                if (integer1 <= 0)
                        DoSomethingWeird();

                float floatingNumer1 = 1.0;
                float *floatingPointer = &floatNumber1;
                if (floatingPointer == NULL)
                        HowCouldThisHappen();
        }
```

7.3 全局变量

- **除非特殊情况，不使用 Class 类型的全局变量，不使用 Class 类型的 static 类数据成员。**
- **可以用单例模式替代 Class 类型的全局变量。**

- 全局的字符串变量使用 C 风格的 char，不使用各种字符串类。

7.4 名称空间（namespace）

- 允许在 cpp 文件中使用匿名名称空间进行保护，**不允许在头文件中使用匿名名称空间。**
- **避免使用 using namespace 将一个名称空间中的所有名称全部导入。**

8. 模板

8.1 模板声明

- 使用 typename 关键字声明模板，不使用 class 关键字。

8.2 模板适用范围

- **一般情况下仅使用模板来实现容器类或通用算法，如果需要用作其他用途，请增加注释说明必要性**。

8.3 模板中的嵌套类

- 模板中使用嵌套类时，加上 typename 关键字声明。

C/C++

```
1 template<typename T>
2 void someFunction(const T& container)
3 {
4     typename T::Iterator iter(container.begin());
5     iter.next();
6 }
```

8.4 模板继承

- 在模板派生类中调用模板基类函数时，明确在函数调用前声明该函数属于基类。

C/C++

```
template<typename T>
class DerivedClass::public BaseClass<T>
{
public:
    void doAction()
    {
        BaseClass<T>::doBaseAction();
    }
}
```

9. 内存

9.1 优先使用静态内存

- 编译期能够确定的常量数组，应声明为 static。

C/C++

```
void func(int type)
{
    static const int someInt[] = {80, 40, 40, 50, ... 94, 94, 47};
        ......
}
```

9.2 其次为栈内存

- 编译期能够确定长度的非常量数组，应使用栈内存。

C/C++

```
void func(int type)
{
    const int length = 1024;
    unsigned char buffer[length + 1];
    DWORD dwReadSize = length;

    while (::ReadFile(hFile, buffer, length, &dwReadSize, NULL))
    {
        ......
        dwReadSize = length;
    }
}
```

9.3 使用智能指针和容器管理堆内存

- 单个堆对象的生命周期用 std::unique_ptr 来管理。
- 连续的内存空间用 std::vector 或 QVector 来分配和管理，**不允许在代码中出现 new[] 和 delete[] 的显示调用。**

C/C++

```
void func(int type)
{
    std::unque_ptr<CSomeObject> spObject = new CSomeObject();

    int bufferLength = getBufferLength();
    std::vector<SomeType> vecBuffer(bufferLength);

    readBuffer(&vecBuffer[0], bufferLength);
}
```

- 字符串用 std::basic_string<T> 或 QString 来管理。

C/C++

```
void func(int type)
{
    int stringLength = getStringLength();
     std::string str;
     str.resize(stringLength);
                readStringInCStyleFunction(&str[0], stringLength);
}
```

- delete 不应该出现在析构函数或用于清理内存的函数之外的地方出现。
- 不允许使用 malloc/free 来分配和释放内存。

10. 跨平台

10.1. Windows 内建类型

- 调用 Windows API 时，使用 DWORD、LPCTSTR 等 Windows 类型，其它情况下避免使用，但可以使用 HRESULT。

10.2. Windows API

- 避免使用 Windows 系统 API，只能在限定模块内使用，比如一个封装操作系统 API 的中间层。

10.3. C++ 扩展库

- 不允许使用 Boost 库，除非是已经进入 C++11 标准的部分。

10.4. C++ 语法

- 可以使用 C++11 新语法中 Visual C++ 支持的部分，但须遵守如下原则：
  - 不使用 lambda 表达式。

- 仅使用 auto 来简化模板变量的声明，不将基本类型或者表达式返回值赋给 auto 对象。
- 不使用尾部返回类型声明，例如 auto Function() -> int;
- 不使用 Raw String。
- 禁用模版元编程。
- 不允许使用逗号表达式。

- **不能使用 Visual C++ 专有语法**，例如：
  - __finally、__super、__forceinline 等关键字。
  - 省略类静态成员的外部定义。
  - 将函数指针 cast 成整型指针。
  - 重复包含有 extern 全局变量定义(非声明)的头文件。
  - goto 关键字跨越变量定义。
- **如果有可能，在 Visual Studio 中设置 /Za 标志关闭 Microsoft Extension to C++。**

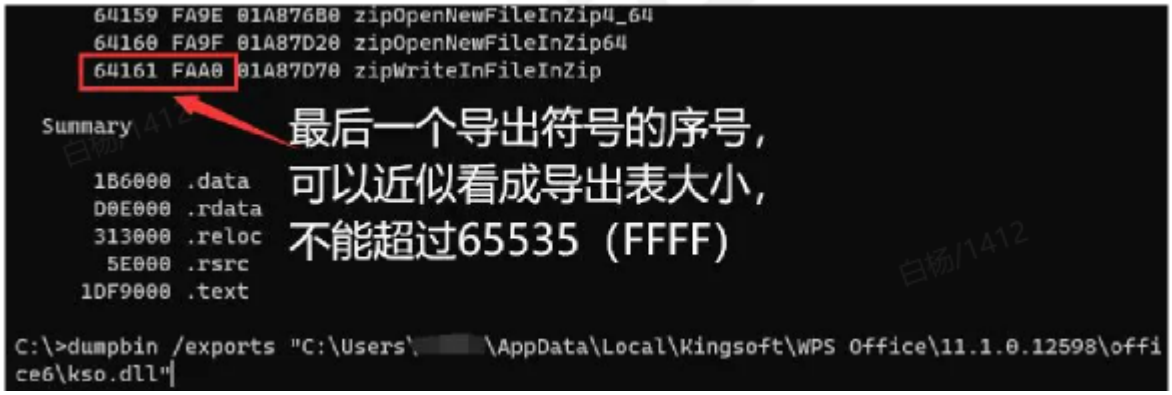
11.Qt 相关

- **推荐使用 Qt5 的方式 connect 信号与槽，不推荐使用 Qt4 时代的 SIGNAL 与 SLOT 宏包含的方式**

二、公司红线

2.1 禁止新增导出符号（增加导出表大小）

- 导出符号包括导出类造成的类成员函数、静态成员变量的导出，或者单独导出的函数或全局变量
- Windows 下可以通过 Visual Studio 提供的 dumpbin 工具检查对应可执行文件的导出表大小。



2.1.1 基于当前工程结构，禁止新增导出类

- Shell Command 与 Qt Widget 体系除外；
- 建议通过接口、导出函数、纯虚抽象类的方式跨模块使用来代替导出类的需求。

2.1.2 基于当前工程结构，公共模块禁止新增导出符号

- 确有必须新增导出符号的，采用置换策略（增加几个就删除几个已经不需要导出的），

确保导出符号总数不能增长。

2.2 信息收集使用要求

- 新增信息收集：统一使用数仓，并遵循数仓的流程与要求
- 禁止未经审核，使用数仓同步的方式发送数据
- 禁止未经审核，在启动、打开过程中发送信息收集
- 信息收集的数据获取逻辑：禁止磁盘、网络、打印机等耗时信息的获取

2.3 禁止拷贝(库)代码

- 需要此类代码文件的能力时，建议通过公共 static 或者 share 类型模块提供
- 禁止拷贝基础通用库代码（例：MD5、Crc、Base64 等基础算法库）

2.4 启动流程的禁止操作

- 禁止未经对应模块负责人审核，在启动、打开等非用户主动触发的功能中，增加创建定时

器

- 禁止未经对应模块负责人审核，在启动流程中增加创建线程

- 禁止未经对应模块负责人审核，在启动流程中增加文件、注册表的写操作
- 禁止未经对应模块负责人审核，在启动流程中增加模块链接依赖
- 禁止未经版本管理审核，在启动流程中增加 auto 类型插件加载

2.5 安装目录的操作要求

- 禁止未经安装包 PackageOwner 审核，在安装目录增加文件
- 安装目录中已有文件（文件夹）大小变更超过 1M，需要经过安装包 PackageOwner 审核

（ 文件夹指 office6/addons 下的文件夹）

2.6 注册表的禁止操作

- **禁止多次实时获取注册表的值**
  - 采用监听注册表变更或者本地 RPC 跨进程通信代替
- **禁止使用注册表明文存储 url、模块路径、程序敏感信息等**
  - 建议本地加密、auth 配置或者云端存储

2.7 URL 地址及 MSVC 的禁止操作

- **禁止 URL 地址硬编码（包括域名、IP）**
  - 建议通过 auth 配置文件获取
- **禁止新增使用 MSVC 的 DelayLoad Hook 机制**
  - 建议通过设计模块使用动态加载的方式代替

2.8 静态变量、全局变量、常量的禁止操作

- **禁止在头文件中使用 const 定义全局常量**
  - 可以在编译期求值（constexpr）

C/C++

```cpp
// 错误写法
const int X_MAX_INT = std::numeric_limits<int>::max();

// 正确写法
constexpr int X_MAX_INT = std::numeric_limits<int>::max();
```

C/C++

```cpp
// 错误写法
const int g_const_array = {0,1,2,3,4,5,6,7,8,9};

// 正确写法
// 在头文件中声明
extern const int g_const_array[10];

// 在模块的某个编译单元中定义
const int g_const_array = {0,1,2,3,4,5,6,7,8,9};

// 正确写法
// 在头文件中声明
class KFoo
{
public:
    static const int g_const_array[10];
};

// 在模块的某个编译单元中定义
const int KFoo::g_const_array[10] = {0,1,2,3,4,5,6,7,8,9};
```

- **禁止在头文件定义。不能在编译期求值的常量。**

C/C++

```
// 错误写法
// 直接在头文件写
const UINT CF_LINKSOURCE = RegisterClipboardFormatW(L"Link Source");

// 正确写法
// 在头文件中声明
extern const UINT CF_LINKSOURCE;

// 在模块的某个编译单元中定义
const UINT CF_LINKSOURCE = RegisterClipboardFormatW(L"Link Source");
```

- **禁止在头文件中使用 static 定义全局变量**

C/C++

```
// 错误写法
static int g_var

// 正确写法
// 在头文件中声明
extern int g_var;

// 正确写法
int g_var;

// 错误写法
static int g_nonconst_array[];

// 正确写法
// 在头文件中声明 (建议指明数组大小并增加 static assert)
extern int g_nonconst_array[xx];

// 在模块的某个编译单元中定义
int g_nonconst_array[xx]

// 正确写法
// 在头文件中声明
class KFoo
{
public:
    static const int g_nonconst_array[10];
};

// 在模块的某个编译单元中定义
const int KFoo::g_nonconst_array[10] = {0,1,2,3,4,5,6,7,8,9};

```

- **禁止在头文件中使用 static 定义全局常量 (同时使用 static 和 const)**

C/C++

```cpp
// 错误写法
static const int X_MAX_INT = std::numeric_limits<int>::max();

// 正确写法
constexpr int X_MAX_INT = std::numeric_limits<int>::max();
```

C/C++

```cpp
// 错误写法
static const int g_const_array = {0,1,2,3,4,5,6,7,8,9};

// 正确写法
constexpr int g_const_array = {0,1,2,3,4,5,6,7,8,9};
```

C/C++

```cpp
// 错误信息
static const UINT CF_LINKSOURCE = RegisterClipboardFormatW(L"Link Source");

// 正确写法
// 在头文件中声明
extern const UINT CF_LINKSOURCE;

// 在模块的某个编译单元中定义
const UINT CF_LINKSOURCE = RegisterClipboardFormatW(L"Link Source");

```

- **禁止在头文件中使用匿名名字空间定义全局变量**

C/C++

```
// 错误信息
namespace
{
    int g_var;
}

// 正确写法
// 在头文件声明
extern int g_var;

// 在模块的某个编译单元中定义
int g_var;
```

2.9 全局变量、常量的初始化问题

- 全局常量如需动态初始化，且耗时久（比如表达式中调用复杂 non-constexpr 函数，或类常量初始化时调用复杂构造函数）
- **建议通过函数调用来替代全局变量以延迟初始化，函数实现通过局部静态变量来避免多次调用**。

C/C++

```
// 错误写法
const  int g_var = func1();

....
func2(g_var)

// 正确写法
int getxxx()
{
    static int s_var = func1();
     return s_var;
}

......
func2(getxxx())
```

- **全局变量如果初始化耗时久，就不要在定义时初始化，延迟到使用时再初始化。**