中软国际公司内部技术规范

C++语言编程规范

中软国际科技服务有限公司

修订声明

参考:华为 C++语言编码规范。

0.

日期	修订版本	修订描述	作者
2017-07-28	1.0	整理创建初稿	陈丽佳
2017-08-02	1.1	统一整理格式，并修正部分错误	陈丽佳

目录

说明

前言

随着业务的发展和产品架构的演进，越来越多的传统电信产品使用 C++语言，很多新型产品更是把 C++作为首选。C++继承于 C，包含 C 的所有特性，同时又增加了新语言特性，如面向对象、泛型设计等。目前 C++使用现状是：基础技能薄弱，陷入很多误区，不能很好地发挥 C++的作用。为了帮助团队合理使用 C++，规避语言陷阱，特制定本规范。

代码总体原则

跟C语言编程一样，C++编程遵循通用原则：

1、清晰第一。清晰性是易于维护、易于重构的程序必需具备的特征。

2、简洁为美。简洁就是易于理解并且易于实现。

3、选择合适的风格，与代码原有风格保持一致。

除此之外，C++编程还应该注意以下方面：

1、正确使用C++

面向对象技术使得程序结构清晰、简单，提高了代码的重用性，但又隐藏了很多内部实现细节，内存模型复杂，不小心会误入陷阱，比如：拷贝构造函数，赋值操作符，析构函数，重载等。

为了简化代码，改善代码结构，提高编程效率，一些团队引入新特性和第三方库，如：模板技术、STL、Boost等，由于缺乏足够的理解，使用中屡次发生问题，比如对迭代器(Iterator)使用不当导致功能失常，甚至程序崩溃。所以，必须深入理解C++对象布局、内存模型等，了解编译器背后所做的处理，才能在编程中知道如何正确使用。

2、安全高效

跟其他流行的高级语言、脚本语言相比，C++运行速度快，天然适合开发核心通信部件，但是这些部件对稳定性的要求非常高，不容许发生异常、失效以及崩溃。C++具有直接操作硬件、访问内存的能力，提供了指针、地址运算等灵活特性，程序员可以任意发挥，增加了出错的几率。所以在追求速度与灵活性的同时，一定要注意保持程序的健壮性。在增强代码稳定性过程中，程序员通常采用if-else等防御式编程，使得代码非常臃肿，可适当采用RAII、智能指针等技术。

规范实施、解释

本规范制定了编写C++语言程序的基本原则、规则和建议。

本规范适用于公司内使用C++语言编码的所有软件。本规范自发布之日起生效，对以后新编写的和修改的代码应遵守本规范。

在某些情况下需要违反本规范给出的规则时，相关团队必须通过一个正式的流程来评审、决策规则违反的部分，个体程序员不得违反本规范中的相关规则。

术语定义

原则：编程时必须坚持的指导思想。
规则：编程时强制必须遵守的约定。
建议：编程时必须加以考虑的约定。
说明：对此原则/规则/建议进行必要的解释。
示例：对此原则/规则/建议从好、不好两个方面给出例子。
延伸阅读材料：建议进一步阅读的参考材料。

1. 常量

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。

规则1.1 使用const常量取代宏

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。

示例：

```
#define MAX_MSISDN_LEN (20)  //不好的例子
const int MAX_MSISDN_LEN = 20; //好的例子
```

规则1.2 一组相关的整型常量应定义为枚举

说明：之所以使用枚举，基于：

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

示例：

```
//好的例子：
enum Color{black, blue, white, red, purple};

enum DayOfWeek{sunday, monday, tuesday, wednesday, thursday, friday,
saturday};

BOOL ColorizeCalendar(DayOfWeek today, Color todaysColor);
ColorizeCalendar(blue, sunday); //编译报错，Blue和Sunday位置错误

//不好的例子：
const int sunday = 0;
const int monday = 1;
const int black  = 0;
const int blue   = 1;

BOOL ColorizeCalendar(int today, int todaysColor);
ColorizeCalendar(blue, sunday); //不会报错
```

- 当枚举值需要对应到具体数值时，须在声明时显示赋值。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。

示例：

```
//好的例子：S协议里定义的设备ID值，用于标识设备类型
enum TDeviceType
{
    DEV_UNKNOWN = -1,
    DEV_DSMP = 0,
```

```
    DEV_ISMG = 1,
    DEV_WAPPORTAL = 2
};
```

程序内部使用，仅用于分类的情况，不应该进行显式的赋值。

示例：

```
//好的例子：程序中用来标识会话状态的枚举定义
enum TSessionState
{
    SESSION_STATE_INIT,
    SESSION_STATE_CLOSED,
    SESSION_STATE_WAITING_FOR_RSP
};
```

- 应当尽量避免枚举值重复，如必须重复也要用已定义的枚举来修饰，例如：

```
typedef enum
{
    RTCP_SR = 200,
    RTCP_MIN_TYPE = RTCP_SR,        //must be lowest known type
    RTCP_RR    = 201,
    RTCP_SDES  = 202,
    RTCP_BYE   = 203,
    RTCP_APP   = 204,
    RTCP_RTPFB = 205,
    RTCP_PSFB  = 206,
    RTCP_XR  = 207,
    RTCP_RSI = 208,
    RTCP_PUBPORTS = 209,
    RTCP_MAX_TYPE = RTCP_PUBPORTS  //must be highest known type
} rtcp_type_t;
```

规则1.3 不相关的常量，即使取值一样，也必须分别定义

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。

示例：

```
//好的例子：协议A和协议B，手机号(MSISDN)的长度都是20。
unsigned const int A_MAX_MSISDN_LEN = 20;
unsigned const int B_MAX_MSISDN_LEN = 20;
//或者使用不同的名字空间：
namespace alib
{
    unsigned const int MAX_MSISDN_LEN = 20;
}
namespace blib
{
    unsigned const int MAX_MSISDN_LEN = 20;
}
```

建议1.1 尽可能使用const

说明：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 。类成员函数加上 const 限定符表明该函数不会修改类成员变量的状态。
使用 const 常见的场景：

- 函数参数：传递引用时，如果函数不会修改传入参数，该形参应声明为const。
- 成员函数：访问函数(如get函数)；不修改任何数据成员的函数；未调用非const函数、未返回数据成员的非const指针或引用的函数。
- 数据成员：如果数据成员在对象构造之后不再发生变化，可将其定义为const。

2. 初始化和类型转换

2.1 声明、定义与初始化

规则2.1 禁止用memcpy、memset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，指针等原始类型及其集合类型，不能使用封装和面对对象特性（如用户定义的构造/赋值/析构函数、基类、虚函数等）。

由于非POD类型比如非集合类型的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。

即使对集合类型的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、memset操作。

示例：×××产品程序异常退出(core dump)。

经过现场环境的模似，程序产生COREDUMP，其原因是：在初始化函数内使用memset(this, 0, sizeof(*this))进行了类的初始化，将类的虚函数表指针被清空，从而导致使用空指针。

解决方案：使用C++构造函数初始化，不要使用memset函数初始化类对象。

建议2.1 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。**

示例：

```
//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=”nicole”; //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(”nicole”); //调用一次构造函数
```

建议2.2 避免构造函数做复杂的初始化，可以使用“init”函数

说明：正如函数的变量都在函数内部初始化一样，类数据成员最好的初始化场所就是构造函数，数据成员都应该尽量在构造函数中初始化。
以下情况可以使用 init()函数来初始化：

- 需要提供初始化返回信息。
- 数据成员初始化可能抛异常。
- 数据成员初始化失败会造成该类对象初始化失败，引起不确定状态。
- 数据成员初始化依赖this指针：构造函数没结束，对象就没有构造出来，构造函数内不能使用this成员；
- 数据成员初始化需要调用虚函数。在构造函数和析构函数中调用虚函数，会导致未定义的行为。

示例：数据成员初始化可能抛异常：

```
class CPPRule
{
public:
    CPPRule():size_(0), res (null) {};  //仅进行值初始化
    long init(int size)
    {
        //根据传入的参数初始化size_, 分配资源res
    }
private:
    int size_;
    ResourcePtr* res;
};
//使用方法：
CPPRule a;
a.init(100);
```

建议2.3 初始化列表要严格按照成员声明顺序来初始化它们

说明：编译器会按照数据成员在类定义中声明的顺序进行初始化，而不是按照初始化列表中的顺序，如果打乱初始化列表的顺序实际上不起作用，但会造成阅读和理解上的混淆；特别是成员变量之间存在依赖关系时可能导致 BUG。
示例：

```
//不好的例子：初始化顺序与声明顺序不一致
class Employee
{
public:
    Employee(const char* firstName, const char* lastName)
        : firstName_(firstName), lastName_(lastName)
        , email_(firstName_ + "." + lastName_ + "@company.com") {};
private:
```

```
    string email_, firstName_, lastName_;
};
```

类定义 email_是在 firstName_, lastName_之前声明，它将首先初始化，但使用了未初始化的 firstName_和 lastName_，导致错误。
在成员声明时，应按照成员相互依赖关系按顺序声明。

建议2.4 明确有外部依赖关系的全局与静态对象的初始化顺序

说明：如果全局对象 A 的成员变量有外部依赖，比如依赖另外一个全局变量 B，在 A 的构造函数中访问 B，隐含的规则就是 B 先于 A 初始化，然而全局与静态对象的初始化与析构顺序未有严格定义，无法确保 B 已经完成初始化，而每次生成可执行程序都可能发生变化，这类 BUG 难以定位。
通常采用单件(Singleton)模式或者把有依赖关系的全局对象放在一个文件中定义来明确初始化顺序。同一个文件中，若全局对象 a 在全局对象 b 之前定义，则 a 一定会在 b 之前初始化；但是不同文件中的全局对象就没有固定的初始化顺序。可以在 main()或 pthread_once() 内初始化一个运行期间不回收的指针。

2.2 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。

规则2.2 使用C++风格的类型转换，不要使用C风格的类型转换

说明：C++的类型转换由于采用关键字，更醒目，更容易查找，编程中强迫程序员多停留思考片刻，谨慎使用强制转换。

C++使用const_cast, dynamic_cast, static_cast, reinterpret_cast等新的类型转换，它们允许用户选择适当级别的转换符，而不是像C那样全用一个转换符。

dynamic_cast：主要用于下行转换，dynamic_cast具有类型检查的功能。dynamic_cast有一定的开销，建议在调测代码中使用。

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。下行转换(把基类的指针或引用转换成派生类的指针或引用)时，由于没有动态类型检查，所以不安全的，不提倡下行转换。

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，相关代码可移植不好。建议对reinterpret_cast<> 的用法进行注释，有助于减少维护者在看到这种转换时的顾虑。

const_cast：用于移除对象的 const属性，使对象变得可修改。

示例：

```
extern void Fun(DerivedClass* pd);
void Gun(BaseClass* pb)
```

```
{
    //不好的例子：C风格强制转换，转换会导致对象布局不一致，编译不报错，运行时可能会崩
溃
    DerivedClass* pd = (DerivedClass *)pb;

    //好的例子：C++风格强制转换，明确知道pb实际指向DerivedClass
    DerivedClass* pd = dynamic_cast< DerivedClass *>(pb);

    if(pd)
        Fun(pd);
}
```

建议2.5 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2.6 避免使用const_cast

说明：const_cast用于移除对象的 const性质。

const 属性提供一种安全感，让程序员知道这个定义是固定不变的，从而不需要担心后面的变化。如果 const 属性在程序员不知道的地方被消除，会带来很多严重的后果。

示例：不好的例子

```
unsigned const int arraySize = 1024;
int &newArraySize = const_cast<int&>(arraySize);
newArraySize = 2048;
```

这里如果不通过引用或者指针访问 arraySize，那么 arraySize 的值始终是 1024。可是如果被作为一个指针或者引用传给其他函数进行取值的话，会发现值变成了 2048。

示例：不好的例子：强制去掉入参的const属性，导致函数可以对入参进行修改。

```
void setObj(TBase const *obj)
{
    //m_pObj的定义为:
    TBase *m_pObj;
    m_pObj = const_cast<TBase*>(obj);
    m_pObj->value = 123;
}
```

建议2.7 使用虚函数替换dynamic_cast

说明：很多刚从 C 语言转过了的程序员习惯这样的思路：若对象的类型是 T1，则做某种处理；若对象的类型是 T2，则做另外的处理等等。但 C++提供了更好的解决方案：虚函数。

虚函数与 dynamic_cast 类型转换相比：

- 虚函数更安全，不会出现强制转换错的情况；

- 虚函数效率更高：用函数指针，避免条件判断；
- 虚函数不需要在编码时确定对象的真实类型，而dynamic_cast必须告知要转成的类型，运行时若类型不当返回空指针或者抛异常；
- 虚函数适用性更强：虚函数是真正动态绑定；类型转换当增加或删除一个派生类时，dynamic_cast必须增减相应的代码。

3. 函数

3.1 内联函数

规则3.1 内联函数(inline function)小于10行

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。**

规则3.2 使用内联函数代替函数宏

说明：C++中也支持宏的功能，但是宏有其自身固有的缺陷(例如无法对参数进行类型检查)，因此，能使用内联函数的地方，一定不使用宏。
示例：

```
//较好的例子:
template <class TYPE_T> inline TYPE_T& max(TYPE_T& x, TYPE_T& y)
{
   return (x > y) ? x : y;
}
//不好的例子:
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

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。

建议3.1 内联函数应该放在头文件中声明，而且在函数前添加inline关键字

说明：内联函数的定义对编译器而言必须可见，以便在调用点将函数展开。放在头文件中可以保证对编译器可见，修改或者删除内联函数时，重新编译使用该头文件的所有源文件。

建议3.2 内联函数的实现放在独立的文件

说明：确保接口清晰。如果使用者和维护者看见声明包含大量的内联实现，会干扰他们的思维，降低声明的可读性和可维护性。所以除了最简单的成员存取函数外，其他较为复杂内联函数的实现放到独立的头文件中(建议使用.inl 为扩展名)，在声明头文件的最后 include。

```
//cpp_rule.h
#ifndef CPP_RULE_H
#define CPP_RULE_H

class CppRule
{
public:
```

```
    inline inlineFunction();
};

#include "cpp_rule.inl"
#endif  //CPP_RULE_H

//cpp_rule.inl
#ifndef CPP_RULE_INL
#define CPP_RULE_INL

inline CppRule::inlineFunction()
{
    //内联函数实现
}

#endif //CPP_RULE_INL
```

3.2 函数参数

建议3.3 入参尽量用const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。

如果是基于老平台开发的产品，则优先顺从原有平台的处理方式。

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3.4 消除未使用函数参数

说明：检查未使用的函数参数，确认是否需要使用该函数参数，如果不需要直接删除参数名。当实现接口时，有一些参数没有被引用是相当常见的。编译器会发现未使用的参数，并产生一个警告，有些组件甚至会认为这是一个错误。为避免发生如此情况，将未使用的参数使用 /* 参数名 */ 语法将其注释掉。

示例：

```
//好的例子：将localityHint参数名去掉，在注释中保留参数名以备参考和理解参数含义
pointer allocate(size_type numObjects, const void * /* localityHint */ =
0)
{
    return static_cast<pointer>(mallocShared(numObjects * sizeof(T)));
}
```

建议3.5 尽量少用缺省参数

说明：使用参数的缺省值仅仅方便函数的使用，没有赋予函数新的功能，但降低函数的可理解。缺省参数使得拷贝粘贴以前函数调用的代码难以呈现所有参数，当缺省参数不适用于新代码时可能导致重大问题。

3.3 函数指针

建议3.6 尽量少用函数指针

说明：不少代码中还是用函数指针来实现一些功能扩展（如封装），但函数指针难以理解和难以维护，建议使用C++中派生与继承的基本用法，少用函数指针。

4. 类

4.1 类的设计

类是面向对象设计的基础，一个好的类应该职责单一，接口清晰、少而完备，类间低耦合、类内高内聚，并且很好地展现封装、继承、多态、模块化等特性。

原则4.1 类职责单一

说明：类应该职责单一。如果一个类的职责过多，往往难以设计、实现、使用、维护。随着功能的扩展，类的职责范围自然也扩大，但职责不应该发散。

用小类代替巨类。小类更易于编写，测试，使用和维护。用小类体现简单设计的概念；巨类会削弱封装性，巨类往往承担过多职责，试图提供“完整”的解决方案，但往往难以真正成功。

如果一个类有 10 个以上数据成员，类的职责可能过多。

原则4.2 隐藏信息

说明：封装是面向对象设计和编程的核心概念之一。隐藏实现的内部数据，减少调用者代码与具体实现代码之间的依赖。

- 尽量减少全局和共享数据；
- 禁止成员函数返回成员可写的引用或者指针；
- 将数据成员设为私有的(struct除外)，并提供相关存取函数；
- 避免为每个类数据成员提供访问函数；
- 运行时多态，将内部实现(派生类提供)与对外接口(基类提供)分离。

原则4.3 尽量使类的接口正交、少而完备

说明：应该围绕一个核心去定义接口、提供服务、与其他类合作，从而易于实现、理解、使用、测试和维护。接口函数功能正交，尽量避免一个接口功能覆盖另一个接口功能。接口函数太多，会难以理解、使用和维护。如果一个类包含 20 个以上的非私有成员函数，类的接口可能不够精简。

规则4.1 模块间对外接口类不要暴露私有和保护成员

说明：对外接口类暴露受保护或者私有成员则破坏了封装，一旦因为类的设计变更(增加，删除，修改内部成员)会导致关联组件或系统的代码重新编译，从而增加系统编译时间，也产生了二进制兼容问题，导致关联升级和打补丁。所以除非必要，不要在接口类中暴露私有和保护成员。

有如下几种做法：

- 使用纯虚类作为接口类，用实现类完成实现，使用者只看到接口类，这种做法缺点是：
  - 代码结构相对复杂。
  - 新增接口必须放在原有接口后面，不能改变原有接口的顺序。否则，因为虚函数表的原因，会导致客户代码重新编译。

- 接口类使用PIMPL模式(只有一个指向实现类指针的私有数据成员)，所有私有成员都封装在实现类中(实现类可以不暴露为头文件，直接放在实现文件中)。
  - 代码结构简单，容易理解。
  - 可以节省虚函数开销，但是有间接访问开销。
  - 修改实现不会导致客户代码重新编译。

```
class Interface
{
public:
    void function();
private:
    Implementation* impl_;
};

class Implementation
{
public:
    int i;
    int j;
};

void Interface:: function ()
{
    ++impl_->i;
}
```

规则4.2 避免成员函数返回成员可写的引用或者指针

说明：破坏了类的封装性，对象本身不知道的情况下对象的成员被修改。

示例：不好的例子

```
class Alarm
{
public:
    string&  getname(){return name;}  //破坏类的封装性，成员name被暴露
private:
    string name;
};
```

例外：某些情况下确实需要返回可写引用或者指针的，例如单件模式的一种写法：

```
Type& Type::Instance()
{
    static Type instance;
    return instance;
}
```

规则4.3 禁止类之间循环依赖

说明：循环依赖会导致系统耦合度大大增加，所以类之间禁止循环依赖。类 A 依赖类 B，类 B 依赖类 A。
出现这种情况需要对类设计进行调整，引入类 C：

- 升级：将关联业务提到类C，使类C依赖类A和类B，来消除循环依赖
- 降级：将关联业务提到类C，使类A和类B都依赖类C，来消除循环依赖。

示例：类 Rectangle 和类 Window 互相依赖

```
class Rectangle
{
public:
    Rectangle(int x1, int y1, int x2, int y2);
    Rectangle(const Window& w);
};

class Window
{
public:
    Window(int xCenter, int yCenter, int width, int height);
    Window(const Rectangle& r);
};
```

可以增加类 BoxUtil 做为转换，不用产生相互依赖

```
class BoxUtil
{
public:
    static Rectangle toRectangle(const Window& w);
    static Window toWindow(const Rectangle& r);
};
```

建议4.1 将数据成员设为私有的(struct除外)，并提供相关存取函数

说明：信息隐藏是良好设计的关键，应该将所有数据成员设为私有，精确的控制成员变量的读写，对外屏蔽内部实现。否则意味类的部分状态可能无法控制、无法预测，原因是：

- 非private成员破坏了类的封装性，导致类本身不知道其数据成员何时被修改；
- 任何对类的修改都会延伸影响到使用该类的代码。

将数据成员私有化，必要时提供相关存取函数，如定义变量 foo_及取值函数 foo()、赋值操作符 set_foo()。 存取函数一般内联在头文件中定义成内联函数。如果外部没有需求，私有数据成员可以不提供存取函数，以达到隐藏和保护的目的。不要通过存取函数来访问私有数据成员的地址(见规则 4.2)。

建议4.2 使用PIMPL模式，确保私有成员真正不可见

说明：C++将私有成员成员指定为不可访问，但还是可见的，可以通过 PIMPL 惯用法使私有成员在当前类的范围中不可见。PIMPL 主要是通过前置声明，达到接口与实现的分离的效果，降低编译时间，降低耦合。

示例：

```
class Map
{
private:
    struct  Impl;
    shared_ptr<Impl> pimpl_;
};
```

4.2 构造、赋值和析构

规则4.4 包含成员变量的类，须定义构造函数或者默认构造函数

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：如果这个类是从另一个类继承下来，且没有增加成员变量，则不用提供默认构造函数

示例：如下代码没有构造函数，私有数据成员无法初始化：

```
class CMessage
{
public:
    void ProcessOutMsg()
    {
        //…
    }
private:
    unsigned int  msgid;
    unsigned int  msglen;
    unsigned char *msgbuffer;
};
CMessage msg;   //msg成员变量没有初始化
msg.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 CMessage
{
public:
    CMessage ():
      msgid(0),
      msglen (0),
      msgbuffer (NULL)
      {
      }
      //...
};
```

规则4.5 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 explict 声明，则会成为隐式转换函数。
示例：

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

ProcessFoo("zhangsan"); //函数调用时，编译器报错，因为显式禁止隐式转换
```

定义了 Foo::Foo(string &name)，当形参是 Foo 对象实参为字符串时，构造函数 Foo::Foo(string &name)被调用并将该字符串转换成一个 Foo 临时对象传给调用函数，可能导致非预期的隐式转换。解决办法：在构造函数前加上 explicit 限制隐式转换。

规则4.6 包含资源管理的类应自定义拷贝构造函数、赋值操作符和析构函数

说明：如果用户不定义，编译器默认会生成拷贝构造函数、赋值操作符和析构函数。自动生成的拷贝构造函数、赋值操作符只是将所有源对象的成员简单赋值给目的对象，即浅拷贝(shallow copy)；自动生成析构函数是空的。这对于包含资源管理的类来说是不够的：比如从堆中申请的资源，浅拷贝会使得源对象和目的对象的成员指向同一内存，会导致资源重复释放。空的析构函数不会释放已申请内存。
如果不需要拷贝构造函数和赋值操作符，可以声明为 private 属性，让它们失效。
示例：如果结构或对象中包含指针，定义自己的拷贝构造函数和赋值操作符以避免野指针。

```
class GIDArr
{
public:
    GIDArr()
    {
        iNum = 0;
        pGid = NULL;
    }
    ~GIDArr()
    {
        if (pGid)
        {
            delete [] pGid;
        }
    }
private:
    int iNum;
    char *pGid;
```

```
    GIDArr(const GIDArr& rhs);
    GIDArr& operator = (const GIDArr& rhs) ;
}GIDArr;
```

规则4.7 让operator=返回*this的引用

说明：符合连续赋值的常见用法和习惯。

示例：

```
String& String::operator=(const String& rhs)
{
    //...
    return *this;           //返回左边的对象
}
string w, x, y, z;
w = x = y = z = "Hello";
```

规则4.8 在operator=中检查给自己赋值的情况

说明：自己给自己赋值和普通赋值有很多不同，若不防范会出问题。

示例：

```
class String
{
public:
    String(const char *value);
    ~String();
    String& operator=(const String& rhs);
private:
    char *data;
};

//自赋值，合法
String a;
a=a;
//不好的例子：忽略了给自己赋值的情况，导致访问野指针
String& String::operator=(const String& rhs)
{
    delete [] data;    //删除data

    //分配新内存，将rhs的值拷贝给它
    data = new char[strlen(rhs.data) + 1]; //rhs.data已经删除，变成野指针
    strcpy(data, rhs.data);

    return *this;
}

//好的例子：检查给自己赋值的情况
```

```
String& String::operator=(const String& rhs)
{
    if(this != &rhs)
    {
        delete [] data;
        data = new char[strlen(rhs.data) + 1];
        strcpy(data, rhs.data);
    }
    return *this;
}
```

规则4.9 在拷贝构造函数、赋值操作符中对所有数据成员赋值

说明：确保构造函数、赋值操作符的对象完整性，避免初始化不完全。

规则4.10 通过基类指针来执行删除操作时，基类的析构函数设为公有且虚拟的

说明：只有基类析构函数是虚拟的，才能保证派生类的析构函数被调用。
示例：基类定义中无虚析构函数导致的内存泄漏。

```
//如下平台定义了基类A，完成获取版本号的功能。
class A
{
public:
    virtual std::string getVersion()=0;
};
//产品派生类B，实现其具体功能，其定义如下：
class B:public A
{
public:
    B()
    {
        cout<<"B()"<<endl;
        m_int = new int [100];
    }

    ~B()
    {
        cout<<"~B()"<<endl;
        delete [] m_int;
    }
    std::string getVersion(){ return std::string("hello!");}

private:
    int *m_int;
};
```

```
//模拟该接口的调用代码如下：
int main(int argc, char* args[])
{
    A *p = new B();
    delete p;
    return 0;
}
```

派生类 B 虽然在析构函数中进行了资源清理，但不幸的是该派生类析构函数永远不会被调用。由于基类 A 没有定义析构函数，更没有定义虚析构函数，当对象被销毁时，只会调用系统默认的析构函数，故导致内存泄漏。

规则4.11 避免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虚函数，会导致未定义的行为。
在 C++中，一个基类一次只构造一个完整的对象。
示例：类 BaseA 是基类，DeriveB 是派生类

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

BaseA::BaseA()          //基类构造函数
{
    log();          //调用虚函数 log
}

class DeriveB:public BaseA //派生类
{
public:
    virtual void log() const;
};
```

当执行如下语句：
DeriveB B;
会先执行 DeriveB 的构造函数，但首先调用 BaseA 的构造函数，由于 BaseA 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，从而导致未定义的行为。
同样的道理也适用于析构函数。

建议4.3 拷贝构造函数和赋值操作符的参数定义成const引用类型

说明：拷贝构造函数和赋值操作符不可以改变它所引用的对象。

建议4.4 在析构函数中集中释放资源

说明：使用析构函数来集中处理资源清理工作。如果在析构函数之前，资源被释放（如 release 函数），请将资源设置为 NULL，以保证析构函数不会重复释放。

4.3 继承

继承是面向对象语言的一个基本特性。理解各种继承的含义：“public 继承”意味"是...一个"，纯虚函数只继承接口，一般的虚函数继承接口并提供缺省实现，非虚函数继承接口和实现但不允许修改。继承的层次过多导致理解困难；多重继承会显著增加代码的复杂性，还会带来潜在的混淆。

原则4.4 用组合代替继承

说明：继承和组合都可以复用和扩展现有的能力。如果组合能表示类的关系，那么优先使用组合。

继承实现比较简单直观，但继承在编译时定义，无法在运行时改变；继承对派生类暴露了基类的实现细节，使派生类与基类耦合性非常强。一旦基类发生变化，派生类随着变化，而且因为派生类无法修改基类的非虚函数，导致修改基类会影响到各个派生类。

而组合的灵活性较高，代码耦合小，所以优先考虑组合。

但是并非绝对，往往组合和继承是一起使用的，例如组合的元素是抽象的，通过实现抽象来修改组合的行为。

继承在一般情况下有两类：实现继承(implementation inheritance)和接口继承(interface inheritance)，尽可能不要使用实现继承而考虑用组合替代。

接口继承：只继承成员函数的接口（也就是声明），例如纯虚（pure virtual）函数；实现继承：继承成员函数的接口和实现，例如虚函数同时继承接口和缺省实现，又能够覆写它们所继承的实现；非虚函数继承接口，强制性继承实现。

示例：组合是指一个类型嵌入另一个类型的成员变量，即"有一个" 或 "由...来实现"。例如：

```
class Address{ ... };          //某人居住之处
class PhoneNumber{ ... };      //某人电话号码
class Person
{
private:
   string name;                //组合成员变量
   Address address;            //同上
   PhoneNumber voiceNumber;    //同上
   PhoneNumber faxNumber;      //同上
};
```

原则4.5 避免使用多重继承

说明： 相比单继承，多重实现继承可重用更多代码；但多重继承会显著增加代码的复杂性，程序可维护性差，且父类转换时容易出错，所以除非必要，不要使用多重实现继承，使用组合来代替。

多重继承中基类都是纯接口类，至多只有一个类含有实现。

规则4.12 使用public继承而不是protected/private继承

说明：public 继承与 private 继承的区别：

- private继承体现"由...来实现"的关系。编译器不会把private继承的派生类转换成基类，也就是说，私有继承的基类和派生类没有"是...一个"的关系。
- public继承体现"是...一个"的关系，即类B public继承于类A，则B的对象就是A的对象，反之则不然。例如“白马是马，但马不是白马”。

对继承而言，努力做到"是...一个"的关系，否则使用组合代替。

private 继承意味"由...来实现"，它通常比组合的级别低，与组合的区别：

- private继承可以访问基类的protected成员，而组合不能。
- private继承可以重新定义基类的虚函数，而组合不能。
- 尽量用组合代替private继承，因为private继承不如组合简单直观，且容易和public继承混淆。

规则4.13 继承层次不超过4层

说明：当继承的层数超过 4 层时，对软件的可维护性大大降低，可以尝试用组合替代继承。

规则4.14 虚函数绝不使用缺省参数值

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。因此只要在基类中定义缺省参数值即可，绝对不要在派生类中再定义缺省参数值。

示例：虚函数display缺省参数值strShow 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& strShow = "I am Base class !")
    {
        std::cout << strShow << std::endl;
    }
    virtual ~Base(){}
};
class Derive: public Base
{
public:
    virtual void display(const std::string& strShow  = "I am Derive class !")
    {
        std::cout << strShow << std::endl;
    }
    virtual ~Derive(){}
};
int main()
{
    Base* pBase = new Derive();
    Derive* pDerive = new Derive();
```

```
    pBase->display();  //程序输出结果：I am base class ！ 而期望输出：I am Derive class ！
    pDerive->display();//程序输出结果：I am Derive class ！
    delete pBase;
    delete pDerive;
    return 0;
};
```

规则4.15 绝不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：pB->mf()和 pD->mf()两者行为不同。

```
class B
{
public:
    void mf();
    //...
};

class D:public B
{
public:
    void mf();
    //...
};

D x;                              //x is an object of type D
B *pB = &x;                       //get pointer to x
D *pD = &x;                       //get pointer to x

pB->mf();                         //calls B::mf
pD->mf();                         //calls D::mf
```

建议4.5 避免派生类中定义与基类同名但参数类型不同的函数

说明：参数类型不同的函数实际是不同的函数。

示例：如下三个类，类之间继承关系如下：类 Derive2 继承类 Derive1，类 Derive1 继承类 Base。三个类之中均实现了FOO函数，定义如下：

Base类：`virtual long FOO(const A , const B , const C)=0;`

Derive1类：`long FOO(const A , const B , const C);`

Derive2类：`long FOO(const A , B , const C);`

代码中存在如下的调用：

```
Base* baseptr = new Derive2();
baseptr -> FOO(A,B,C);
```

代码原意是期望通过如上的代码调用Derive2::FOO函数，但是由于Derive2::FOO与Base::FOO的参数类型不一致，Derive2::FOO对Base类来说不可见，导致实际运行的时候，调用到Derive1::FOO，出现调用错误。使得代码逻辑异常。

解决方案：确保派生类 Derive2::FOO 定义和 Base::FOO 一致。

建议4.6 派生类重定义的虚函数也要声明virtual关键字

说明：当重定义派生的虚函数时，在派生类中明确声明其为 virtual。如果遗漏 virtual 声明，阅读者需要检索类的所有祖先以确定该函数是否为虚函数。

4.4 重载

C++的重载功能使得同名函数可以有多种实现方法，以简化接口的设计和使用。但是，要合理运用防止带来二义性以及潜在问题。保持重载操作符的自然语义，不要盲目创新。

原则4.6 尽量不重载操作符，保持重载操作符的自然语义

说明：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。

规则4.16 仅在输入参数类型不同、功能相同时重载函数

说明：使用重载，导致在特定调用处很难确定到底调用的是哪个函数；当派生类只重载函数的部分变量，会对继承语义产生困惑，造成不必要的费解。

如果函数的功能不同，考虑让函数名包含参数信息，例如，使用AppendName()、AppendID()而不是Append()。

建议4.7 使用重载以避免隐式类型转换

说明：隐式转换常常创建临时变量；如果提供类型精确匹配的重载函数，不会导致转换。

示例：

```
class String
{
```

```
    //…
    String( const char* text );      //允许隐式转换
};
bool operator==( const String&, const String& );
//…代码中某处…
if( someString == "Hello" ) {... }
```

上述例子中编译器进行隐式转换，好像someString == String("Hello")一样，形成浪费，因为并不需要拷贝字符。使用操作符重载即可消除这种隐式转换：

```
bool operator==( const String& lhs, const String& rhs );     //#1
bool operator==( const String& lhs, const char* rhs );       //#2
bool operator==( const char* lhs, const String& rhs );       //#3
```

建议4.8 C/C++混用时，避免重载接口函数

说明：目前很多产品采用C模块与C++模块混合的方式，在这个情况下，应该避免模块之间接口函数的重载。比如：传递给用C语言实现的组件的函数指针。

```
stPdiamLiCallBackFun.pfCreateConn = PDIAM_COM_CreatConnect;
stPdiamLiCallBackFun.pfDeleteConn = PDIAM_COM_DeleteConnect;
stPdiamLiCallBackFun.pfSendMsg = PDIAM_COM_SendData;
stPdiamLiCallBackFun.pfRematchConn = PDIAM_COM_RematchConn;
stPdiamLiCallBackFun.pfSuCloseAcceptSocket =
PDIAM_COM_CloseAcceptSocket;

//注册系统底层通讯函数
ulRet = DiamRegLiFunc(&stPdiamLiCallBackFun);
if (DIAM_OK != ulRet)
{
    return PDIAM_ERR_FAILED_STACK_INIT;
}
```

上面的代码中，由于组件通过 C 语言实现，如果重载 PDIAM_COM_CreatConnect 等函数，将会导致该组件无法初始化。

5. 作用域、模板和 C++其他特性

5.1 作用域

原则5.1 使用名字空间进行归类，避免符号冲突

说明：名字空间主要解决符号冲突问题。

示例：两个不同项目的全局作用域都有一个类 Foo，这样在编译或运行时造成冲突。如果每个项目将代码置于不同名字空间中

```
namespace project1
{
    class Foo;
    //…
}
namespace project2
{
    class Foo;
    //…
}
```

作为不同符号自然不会冲突：project1::Foo 和 project2::Foo。

建议名字空间的名称采用全小写，为避免冲突，可采用项目(产品部件)或目录结构。

用名字空间把整个源文件封装起来，以区别于其它名字空间，但是这些内容除外：文件包含(include)，全局标识的声明/定义以及类的前置声明。

示例：

```
#include "a.h"
DEFINE_bool(someflag, false, "dummy flag");
class C;                    //全局名字空间中类C的前置声明
namespace a{class A;}    //a::A 的前置声明
//以上代码就不要放在名字空间b中。
namespace b
{
    //b中的代码…
} //namespace b
```

规则5.1 不要在头文件中或者#include之前使用using指示符

说明：使用using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。

示例：

```
//头文件a.h
namespace A
```

```
{
    int f(int);
}
//头文件b.h
namespace B
{
    int f(int);
}
using namespace B;
void g()
{
    f(1);
}

//源代码a.cpp
#include "a.h"
using namespace A;
#include "b.h"
void main()
{
    g();//using namespace A在#include "b.h"之前，引发歧意：A::f，B::f调用不明
确
}
```

建议5.1 尽量少使用嵌套类

说明：一个类在另一个类中定义，这样的类被称为嵌套类。嵌套类是其外围类的成员，嵌套类也被称为成员类。

```
class Foo
{
private:
    //Bar是嵌套在Foo中的成员类
    class Bar
    {
        //…
    };
};
```

跟其它类一样，嵌套类可以声明为public、priate和protected属性，因此，从外部访问嵌套类，遵循类成员的访问规则。

一般来说，应该尽量少用嵌套类。嵌套类最适合用来对它们的外层类实现细节建模(如：方便实现链表、容器等算法)，且在这个类需要访问外围类所有成员(包括私有成员)时才使用嵌套类。 不要通过嵌套类来进行命名分组，应该使用名字空间。

建议5.2 尽可能不使用局部类

说明：定义在函数体内的类称为局部类。局部类只在定义它的局部域内可见。
局部类的成员函数必须被定义在类定义中，且不能超过 15 行代码。否则，代码将变得很难理解。

建议5.3 使用静态成员函数或名字空间内的非成员函数，避免使用全局函数

说明：非成员函数放在名字空间内可避免污染全局作用域。

如果你必须定义非成员函数，又只是在 .cpp 文件中使用它，也可使用static关键字（如 static int Foo() {...}）限定其作用域。

建议5.4 避免class类型的全局变量，尽量用单件模式

说明：静态生存周期的对象，包括全局变量，静态变量，静态类成员变量，以及函数静态变量，都必须是原生数据类型（POD : Plain Old Data)。

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,从而导致难以发现的 bug。比如作用域结束时，某个静态变量已经被析构了，但其他代码还试图访问该变量，导致系统崩溃。所以只允许 POD 类型的静态变量。同样，不允许用函数返回值来初始化静态变量。

5.2 模板

模板可以衍生出一系列的类和函数，这是一种形式的代码复用。但要注意，这种形式的复用是源代码级而不是目标代码级的。也就是说，对模板的每一次实例化都会产生一份新的源代码。与此相反，继承允许复用基类的大部分目标代码。

滥用模板会造成三个后果：第一，代码规模的过度膨胀；第二，当修改模板时，很难预料是否会对原先正常工作的代码造成不良影响;第三,很难确保模板所有可能的合法实例化都能正常工作。所以模板的使用要仔细且有节制。

建议5.5 谨慎使用模板，只使用模板的基础特性

说明：模板对编译器的要求很高，当前对模板100%支持的编译器几乎没有，因此，使用前应作测试，特别是涉及偏特化，模板参数等高级特性时。

模板的错误提示比较难懂，产生的错误要映射回模板后才能显示给编程者看，但映射毕竟不是错代码本身，所以有时很难看懂，有时甚至失真，对于人员素质要求也高，增加了维护难度。

建议5.6 注意使用模板的代码膨胀

说明：模板会为每个类型产生一个实例，如果使用不当，会产生过多实例，特别是根据常量实例化时。

建议5.7 模板类型应该使用引用或指针

说明：实例化和参数传递复杂类型(结构体，对象)，传值的代价很高；引用和指针可以提高效率。

建议5.8 模板如果有约束条件，请在模板定义处显式说明

说明：编译对模板的检查较弱，很难保证检察所有错误，因此进行显式说明可以减少模板使用错误。

建议5.9 两个模块之间接口中尽量不要暴露模板

说明：因为在编译器优化选项打开的情况下，其编译的符号为不确定，导致依赖。

5.3 其他

规则5.2 不要在extern "C"内部使用#include包含其他头文件

说明：在 C++代码中调用 C 的库文件，需要用 extern "C"来告诉编译器：这是一个用 C 写成的库文件，请用 C 的方式来链接它们。
严格的讲，只应该把函数、变量以及函数类型这三种对象放置于 extern "C"的内部。
如果把#include 指令放在 extern "C"里面，可能会导致 extern "C"嵌套而带来如下几个风险：
extern "C"嵌套过深，导致编译错误
extern "C"嵌套后，改变其他某些文件的编译链接方式

建议5.10 避免使用友元

说明：友元扩大了（但没有打破）类的封装边界。友元会导致类间强耦合，打破封装，暴露出具体实现，从而使友元和类的实现紧耦合；友元不可继承，降低可继承性。
例外：某些情况下，相对于将类成员声明为public，使用友元是更好的选择，尤其是你只允许另一个类访问该类的私有成员时。

建议5.11 避免使用RTTI

说明：RTTI允许在运行时识别对象的类型。使用RTTI很容易违反“开放封闭原则”。如果要根据派生类的类型来确定执行不同逻辑代码，虚函数无疑更合适，在对象内部就可以处理类型识别问题；如果要在对象外部的代码中判断类型，考虑使用双重分派方案，如访问者模式，在对象本身之外确定类的类型。所以，不建议使用RTTI，除非在一些特定场合如某些单元测试中会用到。

建议5.12 使用sizeof(变量)而不是sizeof(类型)

说明：使用 sizeof(varname)，当代码中变量类型改变时会自动更新。

```
struct DATA data;
memset(&data, 0, sizeof(data));
memset(&data, 0, sizeof(DATA));  //当data改为其他类型时，出错。
```

另外，对数组来说，sizeof(数组变量) 并不一定是元素的个数，元素个数是sizeof(数组变

量)/sizeof(数组变量[0])。

6. 资源分配和释放

原则6.1 明确产品动态内存的申请与释放原则

说明：之所以存在大量内存问题，主要原因是申请与释放内存的规则混乱：

- 申请内存后，传入子程序中，由子程序使用并释放；
- 由子程序申请内存并返回父程序，层层调用后在某一个函数内释放。

内存申请与释放一般原则：

- 对象在退出其作用域时，就应该立即被释放，而且要做到：谁申请，谁释放。
- 函数内分配的内存， 函数退出之前要释放，避免跨函数释放；
- 类中数据成员的内存，在析构函数中确认并释放；
- 全局变量、静态变量的内存空间则在进程退出时，或相应的共享库被卸载时，由操作系统回收；
- 如果程序分支很多或内存资源的分配与释放不在同一个地方，要考虑使用RAII等资源跟踪管理技术。

规则6.1 明确operator new的行为和检查策略

说明：当operator new无法满足某一内存分配需求时，默认会抛异常，也可以返回空指针(通过编译选项设置)。团队明确operator new的操作。在申请内存后，要立即检查指针是否为NULL或进行异常处理。

示例：捕获异常来处理申请内存失败情况

```
char* pBuffer = NULL;
try
{
    pBuffer = new char[BUFFER_SIZE];
}
catch (...)
{
    pBuffer = NULL;
    return EF_FAIL;
}
```

或进行非空判断：

```
char* pBuffer = new char[BUFFER_SIZE];
if (NULL == pBuffer)
{
    return ER_FAIL;
}
```

规则6.2 释放内存后，要立即将指针设置为NULL，防止产生野指针

说明： free 或 delete 释放内存后，立即将指针设置为 NULL，防止产生“野指针”。这种判

断最好能够封装起来，见建议 6.2。

示例：

```
char* pBuffer = new char[BUFFER_SIZE];
//…
delete [] pBuffer;
pBuffer = NULL;
```

规则6.3 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，若是对象调用相应的构造函数。
调用 new[n]所包含的动作：申请可容纳 n 个对象外加一点内存来保存数组的元素的数量；
调用 n 次构造函数初始化这块内存中的 n 个对象。
调用 delete 所包含的动作：若是对象调用相应的析构函数；将内存归还系统。
调用 delete[]所包含的动作：从 new[]将找出的 n 值；调用 n 次相应的析构函数；将内存归还给系统。

示例：

```
std::string *string = new std::string;
std::string *stringArray = new std::string[100];

delete string;
string = NULL;

delete [] stringArray;
stringArray = NULL;
```

如果使用 delete stringArray；会导致 stringArray 指向的 100 个 string 对象中的 99 个未被销毁，因为它们的析构函数根本没有被调用，并且结果是不可预测的，编译器不同，结果也不同。

规则6.4 释放结构(类)指针时，首先释放其成员指针的内存空间

示例：下面是一段有内存泄漏的产品代码：

```
struct STORE_BUF_S
{
    ULONG   ulLen;
    UCHAR   *pcData;
}STORE_BUF_T;

void func()
{
    STORE_BUF_T *pstStorageBuff = NULL;
    //申请结构内存….
    //程序处理…
    free(pstStorageBuff);
    return;
}
```

先删除了 pstStorageBuff，pstStorageBuff->pcData 永远不可能被删除了。删除结构指针时，必须

从底层向上层顺序删除。即：

```
free (pstStorageBuff->pcData);
free(pstStorageBuff);
```

规则6.5 释放指针数组时，首先释放数组每个元素指针的内存

说明：在释放指针数组时，确保数组中的每个元素指针是否已经被释放了，这样才不会导致内存泄漏。

示例：

```
struct dirent **namelist;
int n = scandir(path.c_str(), &namelist, 0, alphasort);//【1】

int i = 0;
for(i ; i < n; ++i)
{
    string name = namelist[i]->d_name;
    free(namelist[i]); //【2】
    if(name != ".." && name != ".")
    {
        //.........
        ++fileNum;
        if(MAX_SCAN_FILE_NUM == fileNum )//MAX_SCAN_FILE_NUM=1000
        {
            break;
        }
    }
}
free(namelist); //【3】
return ;
```

从上面的代码可以看是指针数组 `namelist` 由系统函数进行分配内存(如【1】所示)，内存释放时时分别由【2】释放数组单个元素和【3】释放数组本身内存一起完成的。

但是中间有个条件，每次只取 1000 个文件，如果目录下的文件大于 1000 就跳出，后面的就不会再处理(【2】没有执行到)。当本地目录下文件数比较小，小于等于 1000 时没有内存泄漏；而当本地目录下的文件大于 1000 时，就会导致内存泄漏。所以释放指针数组时，请注意首先释放其每个元素的内存空间。

正确的做法是在【3】之前加上：

```
for(int j = i ; j < n; ++j)
{
    free(namelist[i]);
}
```

规则6.6 不要返回局部对象指针

说明：局部对象在定义点构造，在同一作用域结束时立即被销毁。

示例：

```
char* GetParameter ()
```

```
{
    CDBConnect DBConnect;
    //……..
    return DBConnect.GetString("ParamValue");
}
```

由于对象 DBConnect 已经析构，对应的指针已经被释放，从而后续访问非法内存，导致系统coredump。

规则6.7 不要强制关闭线程

说明：线程被强制关闭，导致线程内部资源泄漏。用事件或信号量通知线程，确保线程调用自身的退出函数。线程死锁需要强制关闭的情况除外。

示例：强制关闭线程，导致线程资源泄漏。

```
CShakeHand::CShakeHand()
{
    m_hdShakeThreadrecv = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)ThreadProc_ShakeHands,
        this, NULL, &m_ulShakeThreadID);
}
CShakeHand::~CShakeHand()
{
    TerminateThread(m_hdShakeThreadrecv, 0); //强制关闭
    CloseHandle(m_hdShakeThreadrecv);
}
```

建议6.1 使用new，delete的封装方式来分配与释放内存

说明：推荐使用如下宏，可以在一定程度上避免使用空指针，野指针的问题。

```
#define HW_NEW(var, classname) \
    do { \
    try \
{ \
    var = new classname; \
} \
    catch (...) \
{ \
    var = NULL; \
} \
    break; \
    } while(0)

//(1) 该宏会将var置为NULL，所以调用该宏之后，不再需要置var为NULL
//(2) HW_DELETE宏与NEW对应，用来释放由HW_NEW分配的对象
//    注意：如果以数组方式分配对象(见对HW_NEW的描述)，则必须使用宏HW_DELETE_A
//    来释放，否则可能导致问题，参见：规则6.3
#define HW_DELETE(var) \
```

```
    do \
{ \
    if (var != NULL) \
{ \
    delete var; \
    var = NULL; \
} \
    break; \
} while(NULL == var)

//(1) 这个宏用来删除一个由HW_NEW分配的数组，删除之后也会将var置为NULL
#define HW_DELETE_A(var) \
    do \
{ \
    if (var != NULL) \
{ \
    delete []var; \
    var = NULL; \
} \
    break; \
} while(NULL == var)
```

直接使用HW_DELETE，HW_DELETE_A宏来释放指针内存空间，就不会出现遗忘将指针置为NULL了。

建议6.2 避免在不同的模块中分配和释放内存

说明：在一个模块中分配内存，却在另一个模块中释放它，会使这两个模块之间产生远距离的依赖，使程序变得脆弱。

模块在C++是一个不清晰的概念，小到一个类，大到一个库。如果在不同的类之间分配、释放内存，需要考虑两个类的初始化、销毁顺序；如果在不同的库之间分配、释放内存，需要考虑两个库的加载或卸载顺序。这种远距离的依赖，容易导致遗漏和重复操作，引发严重问题。有时，在通信机制下，两个实体(如线程)之间交换数据或消息，考虑到程序执行效率，不会采用拷贝的方式交换数据，而是通过指针交换数据，仍然会在不同位置分配、释放内存。这种情况下，只有数据交换成功以后，才会由对方负责释放，否则应遵循“谁申请、谁释放”的原则。为了降低处理复杂性，可以适当地采用RAII或智能指针。

建议6.3 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。

C++类库的智能指针就是其中之一：

auto_ptr是标准C++库提供的一种模板类，使用指针进行初始化，其访问方式也和指针一样。在auto_ptr退出作用域时，所指对象能被隐式的自动删除。这样可以象使用普通指针一样使用auto_ptr，而不用考虑释放问题。注意：auto_ptr的复制会造成它本身的修改，原有的auto_ptr将不再指向任何对象，而由新的auto_ptr接管对象内存，并负责自动删除。因此auto_ptr复制后不能再使用，且不能复制const auto_ptr。

boost库中提供了一种新型的智能指针shared_ptr，它解决了多个指针间共享对象所有权的问题，同时也满足容器对元素的要求，因而可以安全地放入容器中。shared_ptr解决了auto_ptr移动语义的破坏性。

关于auto_ptr与shared_ptr使用请参考C++标准库的相关书籍。

示例：使用RAII不需要显式地释放互斥资源。

```
class My_scope_lock
{
public:
    My_scope_lock(LockType& _lock):m_lock(_lock)
    {
        m_lock.occupy();
    }
    ~My_scope_lock()
    {
        m_lock.relase();
    }
protected:
    LockType    m_lock;
}
bool class Data::Update()
{
    My_scope_lock l_lock(m_mutex_lock);
    if()
    {
        return false;
    }
    else
    {
        //execute
    }
    return true;
}
```

7. 异常与错误处理

7.1 异常

异常是C++语言的一个强大特性，在正确使用之前需要深入了解，以及使用异常代码的上下文。

原则7.1 减少不必要的异常

说明：异常对编码技能要求更高，使用中容易出错，首先从安全性角度考虑，尽量少用或者不用异常。

相比返回错误，异常的优点：

- 异常可以集中捕捉，错误检测与算法处理相分离，算法逻辑更清晰；而返回错误在每个返回点都要进行检测与错误处理，代码逻辑分散。
- 异常的约束更强，用户不能忽略抛出的异常，否则程序默认会被终止，而返回错误则可能被忽略。

异常的缺点也很明显：

- 必须检查所有调用点是否可能抛出异常，在抛出后必须正确处理状态和资源变量等，否则可能导致对象状态不正确或者资源泄露等。例如：如果f()依次调用了g()和h()，h抛出被f捕获的异常，g就要当心了，避免资源泄露。
- 必须清楚可能抛出的所有异常，并在合适的地方捕捉，如果遗漏通常会导致程序被终止。
- 使用异常很难评估程序的控制流，代码很难调试。
- 目标文件变大，编译时间延长，性能下降。

若对异常缺乏充分理解，可能会在不恰当的时候抛出异常，或在不安全的地方从异常中恢复。

适用异常的几个场景：

- 出现“不应该出现的”失败，且不能被忽略必须处理，比如分配内存失败。
- 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。
- 错误码难以通过函数的返回值或参数返回，比如流。
- 许多第三方C++库使用异常，必须在系统边界与第三方C++库结合处使用异常便于跟这些库集成。
- 在测试框架中使用异常很方便。

规则7.1 构造和析构函数不能抛出异常

说明：如果构造和析构函数执行失败则无法安全地撤销和回滚，故这些函数不能向外抛出异常。

为了降低复杂性，建议在这类函数中实现最简单的逻辑。

规则7.2 通过传值的方式抛出，通过引用的方式捕获

说明：抛出异常时，如果抛出指针，谁释放指针就成为问题。捕捉时如果是传值，会存在拷贝，拷贝可能不成功(比如异常是由于内存耗尽造成的)，而且拷贝得不到派生类对象，因为在拷贝时，派生类对象会被切片成为基类对象。

规则7.3 确保抛出的异常一定能被捕捉到

说明：异常未被捕捉到，系统的默认行为是终止程序运行，所以要确保程序产生的异常都能被捕捉。

规则7.4 确保异常发生后资源不泄漏

说明：异常发生后，当前代码执行序列被打断，需要查看分配的内存、文件和内核句柄等资源是否正确释放，避免资源泄漏，尤其每个可能的返回点是否正确释放资源。

示例：如下代码存在内存泄漏

```
int PortalTransformer::transRLS
{
    RLS_Service* service = NULL;
    NEW( service, RLS_Service );
    parser->adoptDocument();//失败时会抛异常
     //....
    delete service;
    service =NULL;
    return 0;
}
```

调用adoptDocument出现的异常没有在函数transRLS里面被捕获，而是在父函数里面捕获了异常的派生类。如果发生异常，则NEW(service, RLS_Service)分配的内存泄漏。

解决方案:在函数transRLS里面捕获adoptDocument的异常,如果发生异常,则删除指针service 。

规则7.5 独立编译模块或子系统的外部接口禁止抛异常

说明：异常处理没有普遍通用的二进制标准，所以不允许跨模块抛异常。

7.2 错误处理策略

原则7.2 建立合理的错误处理策略

说明：这里所说的错误指运行时错误，并非模块内部的编程和设计错误。模块内部的编程和设计错误应该通过断言标记。

在设计早期确定错误处理策略，包括：鉴别，严重程度，错误检查，错误处理，错误传递，错误报告方案。

错误鉴别：对每个实体(函数、类、模块)，记录该实体内部和外部的不变式、前置条件、后置条件以及它支持的错误安全性保证。

错误严重程度：对于每个错误，标明严重级别。
错误检查：对于每个错误，记载哪些代码负责检查它。
错误处理：对于每个错误，标明负责处理它的代码。
错误报告：对于每个错误，标明合适的报告方法。
错误传递：对每个模块，标明使用什么编程机制传递错误，如 C++异常、CORBA 异常、返回值。
错误处理策略应该只在模块边界改变。如果模块内外所使用的策略不同，则所有模块入口函数都要直接负责由内到外的策略转换。例如，在一个内部使用 C++异常，但提供 C 语言的 API 边界的模块中，所有 C 语言的 API 必须用 catch(…)捕获所有异常并将其转换为错误代码。

原则7.3 离错误最近的地方处理错误或转换错误

说明：当函数检查到一个自己无法解决的错误，而且会使函数无法继续执行的时候，就应该报告错误。如果缺乏处理的上下文，应该向上传播错误。

规则7.6 错误发生时，至少确保符合基本保证；对于事务处理，至少符合强保证；对于原子操作，符合无错误保证

说明：基本保证是指访问对象时的状态都是正确的；强保证是对基本保证的增强，不仅要状态正确，而且当失败时状态要回滚到操作前的状态，要么成功要么什么都不做；无错误保证是不能出现失败。编码中严格遵循此原则，会极大提升程序的健壮性。
符合基本保证的代码示例：如下代码是解析输入流到对象，流抛出异常方式呈报错误

```
void CMessage::Parse(IStream* input)
{
    try
    {
        m_uiMessageLen = input.ReadInteger();//失败会抛出异常
        if (0 == m_uiMessageLen || m_uiMessageLen>MAX_MESSAGE_LEN)
        {
            throw invalid_argument("Invalid message len in CMessage::Parse
");
        }
        m_pMessage = new char[m_uiMessageLen];//失败抛出异常
        input.Read(m_pMessage, m_uiMessageLen);//失败抛出异常
        //.....
    }
    catch (const exception &exp){
        ResetContent();//把对象的所有字段都设置为无效
        throw exp;
    }
    return;
}
```

上例确保对象字段的值要么都有效要么都无效，不会出现部分有效部分无效的情况，否则必须处理异常被抛出时的对象状态，给调用者带来麻烦，容易遗漏。故接口应该至少符合基本保证。
可以把该函数改造为符合强保证，如下：

```
void CMessage::Parse(IStream* input)
{
    CMessage temp;
    try
    {
        temp .m_uiMessageLen = input.ReadInteger();//失败会抛出异常
        if (0 == temp .m_uiMessageLen ||
temp .m_uiMessageLen>MAX_MESSAGE_LEN)
        {
            throw invalid_argument("Invalid messageLen in CMessage::Parse
");
        }
        temp .m_pMessage = new char[temp .m_uiMessageLen];//失败抛出异常
        input.Read(temp .m_pMessage, temp .m_uiMessageLen);//失败抛出异常
        //.....
    }
    catch (const exception &exp){
        temp .ResetContent();//把对象的所有字段都设置为无效
        throw exp;
    }
    swap(temp);//成功后执行swap
    return;
}
```

失败后，对象的状态不受影响；而成功后，执行的 swap 必须符合无错误保证，否则此函数是无法支持强保证的。

8. 标准库

STL标准模板库在不同产品使用程度不同，这里列出一些基本规则和建议，供各团队参考。

规则8.1 避免使用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 boost::shared_ptr 或者是 std::tr1::shared_ptr(在 C++11 标准中是 std::shared_ptr)来代替。

对于第 2 种场景，可以使用 boost::scoped_ptr 或者 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。

规则8.2 仅将scoped_ptr、shared_ptr和unique_ptr用于管理单个对象

说明：boost::scoped_ptr、boost::shared_ptr、std::tr1::shared_ptr(在 C++11 标准中是 std::shared_ptr)和 std::unique_ptr(C++11 标准)都是用于管理单一对象的智能指针。当这些智能指针在销毁所指向的对象时使用的都是 delete 而不是 delete[]，而使用 delete 删除数组是 undefined 行为，因此不可使用上述智能指针管理数组。

当需要一个具有 RAII 特性的数组时，可以使用 boost::scoped_array、boost::shared_array、std::vector<std::tr1::shared_ptr>或 std::vector<std::unique_ptr>(C++11 标准)代替。

shared_ptr 是基于引用计数的智能指针，可以安全用于大部分场景中，更新引用计数时需要略微消耗一些性能，一般来说对性能不会有显著的影响。使用 shared_ptr 的另外一个需要注意的问题是不要产生循环引用，基于引用计数的智能指针当出现循环引用时会造成内存泄漏。当需要循环引用时，其中一个智能指针请使用 weak_ptr。

boost::shared_ptr、std::tr1::shared_ptr(std::shared_ptr)的线程安全与 stl 中常见类型的线程一致，即：

1. 多个线程可以同时读同一个 shared_ptr 对象；
2. 多个线程可以同时写不同的 shared_ptr 对象。

注：当多个不同的 shared_ptr 对象指向同一个底层对象时，同时写这些 shared_ptr 对象本身是线程安全的，但是需要额外操作保证底层对象的线程安全。

规则8.3 如果涉及循环引用，使用weak_ptr解开循环

说明：当使用各种基于引用计数的 shared_ptr 时，会遇到循环引用的问题，例如：

```
#include <memory>
class TChild;
class TParent
{
public:
    void SetChild(std::shared_ptr<TChild> const& Child)
    {
        Child_ = Child;
    }

private:
    std::shared_ptr<TChild> Child_;
};

class TChild
{
public:
    void SetParent(std::shared_ptr<TParent> const& Parent)
    {
        Parent_ = Parent;
    }

private:
    std::shared_ptr<TParent> Parent_;
};

int main()
{
    std::shared_ptr<TParent> Parent = std::make_shared<TParent>();
    std::shared_ptr<TChild> Child = std::make_shared<TChild>();

    Parent->SetChild(Child);
    Child->SetParent(Parent);

    //到这里Parent和Child产生了循环引用，当Parent、Child超出作用域后将产生内存泄漏。
}
```

为了解决循环引用导致的内存泄漏，需要引入 weak_ptr。将代码修改成下面这样：

```
class TChild
{
public:
    void SetParent(std::shared_ptr<TParent> const& Parent)
    {
```

```
        Parent_ = Parent;
    }

    void UseParent()
    {
        //使用weak_ptr指向的对象之前通过lock()获得shared_ptr。
        std::shared_ptr<TParent> Parent = Parent_.lock();
    }

private:
    std::weak_ptr<TParent> Parent_;
};
```

注意红色部分以及新增的 UseParent()函数，演示了如何使用 weak_ptr。另外需要注意的是 SetParent()函数中形参依然是 shared_ptr，这样将使用了 weak_ptr 的细节隐藏了起来，从外部看全部是 shared_ptr。举例使用的是 C++11 标准中的 std::shared_ptr，但是对于 boost::shared_ptr 和 std::tr1::shared_ptr 同样适用。

规则8.4 使用make_shared代替new生成shared_ptr

说明：在代码中，我们可以使用形如 std::shared_ptr<T> A(new T)的方式初始化 shared_ptr。但是在涉及到 shared_ptr 的地方使用 new 涉及到 3 个潜在的风险。

一是容易出现下面的代码，访问悬空指针：

```
T* A = new T;
std::shared_ptr<T> B(A);
A->xxxxx; //当B超出作用域后：A指向的内存被释放，访问出错
```

二是容易出现下面的代码，引起重复 delete：

```
T* A = new T;
std::shared_ptr<T> B(A);

//在许多代码之后再次出现：
std::shared_ptr<T> C(A);
```

当使用一个原生指针初始化一个 shared_ptr 时，引用计数会被置为 1，于是出现了 2 组独立的引用计数，当这 2 组引用计数到达 0 时都会引发销毁对象的操作，于是就会出现重复 delete 的问题。

三是可能出现内存泄漏的风险，考虑如下代码：

```
int func1();
void func2(std::shared_ptr<T> const& P1, int P2);

int main()
{
    func2(std::shared_ptr<T>(new T), func1());
}
```

在以上调用 func2 的代码中，根据编译器不同，可能会以以下顺序执行代码：

1. new T
2. func1()

3. std::shared_ptr<T>构造

在这种情况下如果在 func1()中抛出了异常，将会造成 new T 泄漏。所以建议使用 new 初始化的 shared_ptr 要放入单独的语句中，即将调用 func2 的代码修改为：

```
std::shared_ptr<T> temp(new T);
func2(temp, func1());
```

使用以上方法相对略微烦琐，多引入了一个变量。跟 shared_ptr 配套存在的 make_shared 可以解决使用 new 初始化的问题。上述两种情况下的代码可以修改为如下：

```
std::shared_ptr<T> B = std::make_shared<T>();
func2(std::make_shared<T>(), func1());
```

make_shared 模板会构造一个指定类型的对象和一个 shared_ptr 并用该对象初始化 shared_ptr，以上操作是一步完成的，不存在中间抛出异常的风险。对于构造时需要参数的类型，将参数加在 make_shared 模板后面的括号中即可。

延伸阅读材料：《Effective C++中文版 第三版》 [美]Scott Meyers 著 侯捷译 2006 电子工业出版社 75 页 条款 17: 以独立语句将 newed 对象置入智能指针

规则8.5 对于同一个对象一旦使用shared_ptr，后续就要处处使用shared_ptr

说明：规则 8.4 描述了混用原生指针和 shared_ptr 容易导致问题：使用悬空指针和重复释放。所以，同一对象的指针要统一用法，要么使用原生指针，要么使用 shared_ptr，不要混用。

规则8.6 对于返回自身的shared_ptr指针的对象，要从enable_shared_from_this类派生

说明：对于需要使用 shared_ptr 管理的对象，当需要 this 指针时也需要使用对应的 shared_ptr，但是从 3.4 中可以看出，如果直接使用 shared_ptr<T>(this)构造一个 shared_ptr 将会导致严重错误。为此，boost 和 stl 都提供了对应的 enable_shared_from_this 类，该类提供了一个 shared_from_this()函数返回 this 指针对应的 shared_ptr。

示例：

```
class TClass : public std::enable_shared_from_this<TClass>
{
public:
    std::shared_ptr<TClass> GetSelf()
    {
        return shared_from_this();
    }

    std::shared_ptr<TClass const> GetSelf() const
    {
        return shared_from_this();
    }
};
```

规则8.7 不要将使用不同版本stl、boost等模板库编译的模块连接在一起

说明：模板库大量使用了内联函数，不同版本的模板库编译的模块对同一种数据类型的操作都已固化在该模块中。如果不同版本的模板库中同一种数据类型的结构或者内存布局不同，在一个模块中定义的对象被另外一个模块操作时可能会产生严重的错误。因为静态连接的模块常常不会划分出明确的接口，常常会相互访问其它模块中定义的对象。

例外：
有些现存的模块已经无法获得源码，不可能重新使用同一版本的模板库重新编译。在这种情况下，如果模块定义了清晰的接口，且接口中没有传递存在风险的数据类型，可以谨慎地混用，但是一定要进行充足的验证。

规则8.8 不要保存string::c_str()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移植性，一定不要保存 string::c_str()的结果，而是在每次需要时直接调用。
示例：
```
//不好的例子：
std::string DemoStr = "demo";
const char* buf = DemoStr.c_str();
//在这里buf指向的位置有可能已经失效。
strncpy（info_buf，buf，INFOBUF_SIZE - 1）；
```

建议8.1 不要将stl、boost等模板库中的数据类型传递到动态链接库或者其它进程中

说明：跨动态链接库或者其它进程还存在另外一个更复杂的问题，一般来说，内存分配与释放要在同一个模块中，如果将容器或者其它数据类型传递到其它模块中由其它模块进行了内存分配释放操作，将会造成不可预知的问题，通常是程序崩溃或者数据消失等但是也有可能在某些情况下程序完全正常运行，因此定位错误会比较困难。

建议8.2 使用容器时要评估大量插入删除是否会生成大量内存碎片

说明：不同的操作系统和运行时库分配内存的策略各不相同，由于容器内存多是动态分配而来，对于反复大量插入删除的操作，有可能会造成大量的内存碎片或者内存无法收回。
对于长期运行的服务程序，建议在使用容器时要对容器是否会造成内存碎片进行评估和测试，如果存在风险的，可以使用内存池(如 boost::pool)来避免这个问题。

建议8.3 使用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。

建议8.4 使用stl、boost等知名模板库提供的容器，而不要自己实现容器

说明：stl、boost 等知名模板库已经提供较完善的功能，与其自行设计并维护一个不成熟且不稳定的库，不如掌握和使用标准库，标准库的使用经验在业界已有成熟的经验和使用技巧。

建议8.5 使用新的标准库头文件

说明：使用 stl 的时候，头文件采用<vector>、<cstring>等，而不是<vector.h>、<string.h>。

9. 程序效率

9.1 C++语言特性的性能分级

影响软件性能的因素众多，包括软件架构、运行平台(操作系统/编译器/硬件平台)等。很多时候，程序的性能在框架设计完成时就已经确定了。因此当一个程序的性能需要提高时，首先需要做的是用性能检测工具对其运行的时间分布进行一个准确的测量，找出关键路径和真正的瓶颈所在，然后针对瓶颈进行分析和优化，而不是一味盲目地将性能低劣归咎于所采用的语言。事实上，如果框架设计不做修改，即使用 C 语言或者汇编语言重新改写，也并不能保证提高总体性能。

C++对性能的设计原则是零开销(zero overhead)，其含义是你不用为你不选择的而付费(you don't pay for what you don't use)，因此需要了解各种 C++语言特性的性能开销。C++语言特性的性能描述采用 FREE/CHEAP/EXPENSIVE 的分级。

- FREE：性能开销很小，甚至有优化，可以放心使用；
- CHEAP：性能开销有一定程度，多数情况下可以使用，在性能关键地方需要注意；
- EXPENSIVE：性能开销较大，需要按照情况使用，在性能关键地方需要慎重使用。

C++语言特性	性能分级	备注
封装	FREE	class 和 C 的 struct 在使用空间上是相同的，class 中的成员函数的时间开销也和 C 等效代码是一致的。
多态	FREE	含有虚函数的 class 在空间上需要增加虚表指针(4 字节)，在虚函数的执行上需要间接寻址的开销。虽然有微量开销，但等同于 C 等效代码。
名字空间	FREE	Namespace 会带来符号名字符串长度的增加，但 C 等效代码也需要增加前缀字符串(比如模块名)。
隐含内联	FREE	没有函数调用的开销，没有指令跳转的顺序执行能让编译器进行更好的优化，但会增加程序大小。
重载	FREE	等效类成员函数的开销
构造和析构	FREE	等效类成员函数的开销
引用	FREE	等效或优于指针的使用 引用可以避免指针的间接寻址开销。
模板	FREE~ EXPENSIVE	模板具有静态多态的优点，部分逻辑提前到编译阶段以提升性能；但会导致代码膨胀，程序尺寸增大。
RTTI	CHEAP~ EXPENSIVE	执行时间有一定耗时，gcc/VC 中 dynamic_cast 的开销小于 10 倍函数调用，程序尺寸增大。
异常	EXPENSIVE	异常捕捉很耗时，其包括栈展开等操作，gcc/VC 中异常处理开销大于 300 倍函数调用。
STL	CHEAP~ EXPENSIVE	STL 提供线性(list)、对数级(map)和常量级(hash_map)不同性能的容器，建议根据应用实际需求选用。

更多的信息请参见延伸阅读材料：

1、《Technical Report on C++ Performance》ISO/IEC PDTR 18015，2003-08-11

2、《The Inefficiency of C++, Fact or Fiction?》Anders Lundgren, IAR Systems

3、《提高C++性能的编程技术》，布尔卡(Dov Bulka),梅休(David Mayhew) 著。

9.2 C++语言的性能优化指导

原则9.1 先测量再优化，避免不成熟的优化

说明：性能涉及到非常多因素，目标不明确的语言层面优化很难显著地改善性能。建议首先测量到性能瓶颈，再对这些性能瓶颈进行针对性的优化。初学者常犯的一个错误，就是编写代码时着迷于过度优化，牺牲了代码的可理解性。

原则9.2 选用合适的算法和数据结构

说明：代码优化应该从选择合适的算法和数据结构开始。对于元素个数超过 1000 的数据，其顺序查找算法效率要远低于对数性能的查找算法。std::vector<int>的占用空间约是 N*sizeof(int)，而 std::list<int>的占用空间约是 3N*sizeof(int)。std::map 提供了对数级的查找算法，而 hash_map 则更高，提供了常数级的查找算法，但 hash_map 需要选择合适的 hash 算法和桶大小。

建议9.1 在构造函数中用初始化代替赋值

说明：通过成员初始化列表来进行初始化总是合法的，效率也高于在构造函数体内赋值。

示例：

```
class A
{
   string s1_;

public:
   A(){s1_ = "Hello,world ";}
};
```

实际上，生成的构造函数代码类似如下：

```
A():s1_(){s1 = "Hello, world";}
```

成员 s1_的缺省构造函数已被隐式调用，构造函数体中的初始化实际上上在调用 operator=而初始化列表只需调用一次 s1_的构造函数，相对效率更高。

```
A():s1_("Hello,world "){}
```

建议9.2 当心空的构造函数或析构函数的开销

说明：空构造函数的开销不一定是0，空构造函数也包括基类构造、类内部成员对象的构造等。

如果对象的构造函数或析构函数有相当的开销，建议避免临时对象的使用，并在性能关键路径上考虑避免非临时对象的构造和析构，比如Lazy/Eager/Partial Acquisition设计模式。

```
class Y
{
   C c;
   D d;
};
class Z : public Y
```

```
{
    E e;
    F f;
public:
    Z() { };
};
Z z;    //initialization of c, d, e, f
```

建议9.3 对象参数尽量传递引用(优先)或指针而不是传值

说明：对于数值类型的int、char等传值既安全又简单；但对于自定义的class、struct、union等对象来说，传引用效率更高：

- 不需要拷贝。class等对象的尺寸一般都大于引用，尤其可能包含隐式的数据成员，虚函数指针等，所以传值的拷贝的代价远远大于引用。
- 不需要构造和析构。如果传值，传入是调用拷贝构造函数，函数退出时还要析构。
- 有利于函数支持派生类。

之所以首选引用，见建议 3.2。

示例：

```
void f(T x)          //bad
void f(T const* x)      //good
void f(T const& x)      //good, prefer
```

建议9.4 尽量减少临时对象

说明：临时对象的产生条件包括：表达式、函数返回、默认参数以及后缀++等等，临时对象都需要创建和删除。对于那些非小型的对象，创建和删除在处理时间和内存方面的代价不菲。采用如下方法可以减少临时对象的产生。

- 用引用或指针类型的参数代替直接传值；
- 用诸如+=代替+
- 使用匿名的临时对象；
- 避免隐式转换；

示例：

```
Matrix a;
a = b + c;      //bad: (b + c) creates a temporary
Matrix a = b;
a += c;         //good: no temporary objects created
```

建议9.5 优先采用前置自增/自减

说明：后置++/--是将对象拷贝到临时对象中，自身++/--，然后再返回临时对象。此过程在非简单数据类型时较为耗时(简单数据类型编译器可以优化)

示例：

```
for (list<X>::iterator it = mylist.begin();
    it != mylist.end();
    ++it) //good: rather than it++
```

```
{
    //...
}
```

建议9.6 简单访问方法尽量采用内联函数

说明：小函数频繁跳转会带来性能上的损失，内联可以避免性能损失。

```
class X
{
private:
    int value_;
    double* array_;
    size_t size_;
public:
    inline int value() { return value_; }
    inline size_t size() { return size_; }
};
```

建议9.7 要审视标准库的性能规格

说明：std::string 是个巨大类，若使用不当(比如大量的+操作)会导致性能迅速下降；std::list<T>::size()在某些实现版本中是线性的，所以在 if(myList.size()==0)时可以考虑用 if(myList.empty())替换；标准输入输出是性能瓶颈，如果不混用 C++和 C 的标准输入输出库，可以考虑关掉同步：std::ios_base::sync_with_stdio(false)。

建议9.8 用对象池重载动态内存管理器

说明：系统调用 new 和 delete 涉及到系统调用等复杂处理，时间和空间开销都较大，对于 DOPRA 的内存申请机制也有类似的情况。建议对于特定类的申请和释放，采用自定义的对象池机制管理该对象的申请和释放，而不用操作系统或 DOPRA 的内存管理机制。对象池的大小需要预先确定或是采用类似 std::vector.resize()的方法可以动态增长。

建议9.9 注意大尺寸数组的初始化效率

说明：我们常这么初始化数组:

```
char szAccount[MAX_ACCOUNT_LEN] = {0};//数组大小16
```

对于字符串确保有结束符即可，故要初始化也应该用“szAccount[0] = 0;”替换之，当然在非关键路径，上述的一行代码完成初始化代码更简洁，也能够被接受。

```
char chTempBuff[MAX_MSG_LEN]    = {0};//数组大小为40K
```

而对于如此大的一块内存清 0，应该用 memset 等库函数替代之。编译器不优化情况下，对于{0}的初始化，通常是一个字节逐一赋值为 0，而 memset 在 64 位平台下很可能是一次 8 个字节。故当数组大于 10 个字节时，两者的性能差距在 2~8 倍，根据数组大小而定。
特别是生成或者删除大尺寸的对象数组，每个数组成员的构造函数或析构函数度要被调用一次，花费的时间更多。

建议9.10 避免在函数内部的小块内存分配

例子 某函数内部，根据消息长度分配内存，然后做相关操作，函数退出前释放内存，如下：

```
char *pMsg = new char[msgLen];
```

其实 pMsg 的生命周期仅仅在函数内，它所指向的内存其实应该是一个临时变量，如果我们能够预测其最大长度远小于线程栈空间，比如最大几十 K，或者只有几十字节，那么就应该声明一个足够大的临时数组，如下：

```
assert(msgLen <= MAX_MSG_LEN);//某些情况下可能需要if检测，而断言检测可能不充分。
char chMsg[MAX_MSG_LEN];//不会有失败和释放的处理，效率也完全不在一个数量级
```

10. 并发

规则10.1 多线程、进程并行访问共享资源时，一定要加锁保护

说明：共享资源包括全局变量，静态变量，共享内存，文件等。

建议封装像智能指针一样的对象对锁进行管理，比如我们就封装了一个auto_lock，在构造时申请锁，析构中释放锁，保证不会忘记“解锁”。如果锁的作用范围有限，则可以这样：

```
do
{
    auto_lock lock(&lock);
    //....
}while(0);
```

规则10.2 锁的职责单一

说明：每个锁只锁一个唯一共享资源；这样，才能保证锁应用的单一，也能更好的确保加锁的范围尽量小。

对于共享全局资源，应该根据实际需要，每类或每个资源，有一把锁。这样，这把锁只锁对这个资源访问的代码，通常这样的代码都会是比较简单的资源操作代码，不会是复杂的函数调用等。相反，如果我们对几类或几个资源共用一把锁。这把锁的责任范围就大了，使用复杂，很难理清锁之间的关系(有没有释放锁，或者锁之间的嵌套加锁等)，容易导致死锁问题。

规则10.3 锁范围尽量小，只锁对应资源操作代码

说明：使用锁时，尽量减少锁的使用范围。我们使用锁，为了方便，会大范围的加锁，如：直接锁几个函数调用。这种使用，一方面会导致多线程执行效率的低下，容易变成串行执行；另一方面，容易出现锁未释放，或者锁的代码中再加锁的场景，最后导致死锁。

所以，对锁操作的最好办法，就是只锁简单资源操作代码。对应资源访问完后，马上释放锁。尽量在函数内部靠近资源操作的地方加锁而不是靠近线程、函数外部加锁。

规则10.4 避免嵌套加锁；如果必须加锁，务必保证不同地方的加锁顺序是一样的

说明：加上一把锁之后，在释放之前，不能再加锁。

典型的锁中加锁的场景：OMU 代码中对几个容器的同时遍历，每个容器一把锁，就导致需要加多把锁。这种场景的解决方法：先加一把锁，对一个容器遍历，选择出合乎要求的数据，并保存在临时变量中；再加另一把锁，使用临时变量，再对其他容器遍历。

锁中加锁，必须保证加锁的顺序是一样的，比如先加的锁后解锁，

Lock1

Lock2

Unlock2

Unlock1

则其他地方的加锁顺序，必须与这里的顺序一样，避免死锁，不允许出现：
lock2
lock1
unlock2
unlock1

建议10.1 进程间通讯，使用自己保证互斥的数据库系统、共享内存，或socket消息机制；尽量避免使用文件等进程无法管理的资源

说明：由于文件在不同进程间访问，无法保证互斥。当然，可以在进程间加进程锁，但只受限于我们能加锁的进程，对于第三方进程等无法保证。这样，当多个进程同时对文件进行写操作时，将会导致文件数据破坏，或文件写失败等问题。
当数据库系统本身的访问接口带有互斥机制，当多个进程同时访问时，可以保证数据库数据的完整。
共享内存，只限制于使用共享内存的几个进程，需要我们对这些访问共享内存的进程加锁。但由于共享内存，第三方进程等无法访问，这也能比较好的保护数据，避免文件系统存在的问题。
socket 消息机制，由操作系统 socket 通讯机制保证互斥，在多个进程间，通过消息来保证数据的互斥。进程的消息都是操作系统转发而来的独立数据，属于进程私有数据，不存在进程间并行访问的问题。

建议10.2 可重入函数尽量只使用局部变量和函数参数，少用全局变量、静态变量

说明：支持多线程并行访问的函数称之为可重入函数。设计可重入函数时，尽量使用局部变量和函数参数来传递数据，在多线程并行访问时，互相之间不会受影响。相反，如果使用全局变量、静态变量，就需要同步。
示例：

```
int iTotalCnt = 10;
void WriteFile()
{
    for (int i=0; i<iTotalCnt; i++)
    {
        //写个数据到内存;
    }

    iTotalCnt = 0;      //写完数据之后，赋值计数为0
}
```

上面的函数，如果是并行访问，将会导致有部分调用 WriteFile 的线程，不执行 for 循环；因为 iTotalCnt 可能被其他线程修改为 0。
引申：一些库函数也是非线程安全，调用时可能会出现多线程并发访问问题。

建议10.3 锁中避免调用函数；如果必须调用函数，务必保证不会造成死锁

说明：这条规则是对加锁范围尽量小(只锁对应资源操作代码)规则的补充。不能把调用函数也加到加锁范围中。因为被调用函数的内部到底做了什么事情，是如何做的，调用者可能不是很清楚。尤其是当被调用函数内部又加锁的情况，就容易导致两个锁互饿，导致死锁。
示例：

```
Callfunc()
{
    Lock2;
    //….
    Unlock2;
}
Thread_func1()
{
    Lock1;
    Callfunc();
    Unlock1;
}
Thread_func2()
{
    Lock2;
    Lock1;
    //…
    Unlock1;
    Unlock2;
}
```

当上述线程函数 Thread_func1()和 Thread_func2()并行执行时，就很有可能导致死锁。而且这种死锁情况还是比较难分析。因为我们调用函数，很多时候只关注函数实现的功能，而忽略函数内部的具体实现。

其次，锁中调用函数，也会把对资源操作的代码扩大化，不利于并行效率。更主要的是，这种操作，由于加锁的范围变大，引起死锁的可能就增大。

建议10.4 锁中避免使用跳转语句

说明：跳转语句包含 return、break、continue、goto 等。如果锁中有宏调用的代码，要特别注意，分析宏中是否存在隐含的跳转语句。

在函数返回时忘记把锁释放，特别是存在很多分支都可能返回的时候，可能一些分支会忘记释放锁。

11. 风格

遵守《C语言编程规范》，仅仅增加C++相关的内容。

11.1标示符命名与定义

建议11.1 类命名以大写字母开头，中间单词也以大写开头

示例：

```
class UrlTable;
class UrlTableProperties;
```

11.2排版

建议11.2 类的声明按照一定的次序进行，关键字不缩进

说明：类的声明按照一定的次序和规范进行。建议的次序如下：

- 按照存取控制特性段排序：public、protected、private，如果没有可以忽略。这个排序是让声明的时候使用者首先看到最关心的特性(对外接口)。
- 在每一段中按照如下顺序声明(先定义后续用到的类型，常量和enums；然后定义行为(构造，析构，成员函数)，最后定义成员变量):

1. typedefs和enums；
2. 常量；
3. 构造函数；
4. 析构函数；
5. 成员函数，含静态成员函数；
6. 数据成员，含静态数据成员。

建议11.3 构造函数初始化列表在同一行或按4格缩进并排几行

示例：初始化列表放在同一行：

```
class TaskProgressTimer
{
public:
    TaskProgressTimer(CHRCollectTask & parent):m_delayTime(30)
    {
    }
};
```

初始化列表并排多行：

```
class TaskProgressTimer
```

```
{
public:
    TaskProgressTimer(CHRCollectTask & parent):
        m_parTask(parent) ,
        m_delayTime(30) ,
        m_intervalTime(50) ,
        timer_handler_( reactor(), *this, m_delayTime, m_intervalTime )
    {
    }
};
```

11.3注释

建议11.4 使用‘//’注释方式，而不是‘/* */’

说明：说明：使用 ‘//’ 而不是 ‘/* */’ 来注释 C++代码。使用‘/* */’的缺点就是容易产生交叉错乱。

即使注释多行代码， //仍然是首选，现在的编辑器提供多行加‘//’和去‘//’的功能。

建议11.5 为重要的类或结构体作注释，注释格式支持工具自动生成

说明：为重要的类或结构体作注释，不宜写太多无用的注释。注释主要表达代码难以直接表达的意思。

建议11.6 为所有重要的成员函数作注释

建议11.7 正式发布的代码不能使用TODO 注释

说明：对那些临时的、短期的解决方案，或待改进的代码使用TODO注释。使用全大写的字符串TODO，后面括号里加上姓名、工号等信息。

```
//TODO(姓名：  工号：  ):注释内容
```

11.4文件组织

建议11.8 整个项目需要的公共头文件应放在一个单独的目录下

建议11.9 一个模块的目录层次不宜太深，以1～2层为宜，接口文件放在最外层

建议11.10 保持文件前言的简洁性

说明：给出版权说明和该文件的简介后即可切入正题——编码。

12. 可移植性(兼容性)

本章讨论C++可移植性问题主要关注：32位移植到64位，不同CPU架构之间的移植。
移植中一些关键问题如下：

1. 指针截断
2. 数据类型字节对齐
3. 对内存地址的错误假设
4. 对复合数据类型成员地址的错误假设
5. 大小端，网络字节序问题

遵守《C语言编程规范》，仅仅增加C++相关的内容。

更多的信息请参见延伸阅读材料：《C++跨平台开发技术指南》，Syd Logan(美) 著。

建议12.1 不直接使用C++的基本数据类型，不要假定其存储尺寸长度

说明：C++标准没有明确基本数据类型的大小与存储格式， 这些基本类型包括：short， int, long, long long, float double等。这些基本数据类型在不同的编译器中，实现有所不同，如：
long类型在32位编译模式下为4字节长度，在64位编译模式下为8字节长度。
所以建议不要直接使用基本数据类型。推荐如下两种使用方式：
1、重定义基本数据类型

```
typedef int32_t int;
typedef int64_t long long;
```

使用重定义后的基本类型好处是：如果程序需要移植，可以大大减少移植的工作量。
2、使用C99标准中定义的标准类型

```
int64_t my_value = 0x123456789LL;
uint64_t my_mask = 3ULL << 48;
```

使用这些标准类型长度的好处是，它们规定了固定的长度，这个长度不会随编译器变化而变化，所以我们可以放心的使用。

建议12.2 避免指针截断

说明：指针截断是从32位移植到64位系统时，经常会碰到的问题。

```
int *i = &int_val;
short *w = (short*)((int)i + 2);
```

上面的代码在32位环境下运行是没有问题的，但在64位环境下，发生了地址截断：无法把64位长的数据接到32位的数据空间里面。
上面的代码中可以使用intptr_t类型解决：

```
int *i = &int_val;
short *w = (short*)((intptr_t)i + 2);
```

建议12.3 注意数据类型对齐问题

说明：需要对结构对齐加以留心，尤其是对于存储在磁盘上的结构体。

在64位系统中，任何拥有int64_t/uint64_t成员的类/结构体将默认被处理为8字节对齐。如果32位和64 位代码共用磁盘上的结构体，需要确保两种体系结构下的结构体的一致对齐。
另外，大多数编译器提供了调整结构体对齐的方案：
gcc 中可使用__attribute__((packed))，MSVC 提供了#pragma pack()和__declspec(align())。
由于各个平台和编译器的不同，所以在不同编译器与平台移植代码时，一定要特别关注编译器关于对齐的参数设置与默认值。因为字节对齐不仅影响性能，而且会导致一些不可预知的问题。

建议12.4 在涉及网络字节序处理时，要注意进行网络字节序与本地字节序的转换

说明：小端法(Little-Endian)
低位字节排放在内存的低地址端即起始地址，高位字节排放在内存的高地址端。
大端法(Big-Endian)
高位字节排放在内存的低地址端即起始地址，低位字节排放在内存的高地址端。
不同cpu平台上字节序通常也不一样：
X86、AMD64平台使用小端法、而HP-IA， IBM AIX的CPU采用的是大端法。
而网络字节序是大端法，如常见网络发送的码流。
涉及网络字节需要注意处理网络字节序与本地字节序的转换，即使本地字节序采用的也是大端法，为了程序可移植性，建议也调用转换函数进行转换。
库函数提供了16，32位整型int的网络字节序与本地字节序的转换函数：

```
htonl, htons, ntohl, ntohs - convert values between host and network byte order
```

建议12.5 避免无符号数与有符号数的转换

说明：不同的国际标准(ANSI C/ISO C++等)对隐式转换有符号和无符号类型的规则不同，有可能导致不同的执行结果。

```
unsigned short usNumber = xxx;
long lNum = usNumber;
```

将unsigned short赋值给long需要经过两次类型转换，ANSI标准中没有规定多次类型转换的顺序。
大多数编译器(例如VC)在高位优先填充0，按照下面的顺序进行转换：

```
lNum = (long) (unsigned long) usNumber;
```

个别编译器(例如BSD的一些编译器)在高位优先填充1，即使用下面的顺序进行转换：

```
lNum = (long) (signed short) usNumber;
```

如果是后一种转换顺序，并且正好usNumber的高位为1，则首先被转换成一个负数的long，接着转换成unsigned long时就成了很大的数。
usNumber永远不可能为负数，没有必要使用signed修饰。
修改办法是定义lNum的类型为unsigned long，并更改名字为ulNum：

```
unsigned short usNumber = xxx;
unsigned long ulNum = usNumber;
```

尽量避免无符号数与有符号数的转换，特别是长度不同数值的类型转换。请首先考虑设计上是否需要这种转换。

建议12.6 创建64 位常量时使用LL 或ULL 作为后缀

说明：指定LL或ULL后缀说明，能让代码更加清晰。

```
int64_t my_value = 0x123456789LL;
uint64_t my_mask = 3ULL << 48;
```

尤其是在>>操作时，无符号与有符号有很大差异的，如果操作数是无符号数，则右移操作符>>，从左边开始插入0， 否则插入符号位的拷贝或者插入0 ，这由编译器决定。

建议12.7 区分sizeof(void *)和sizeof(int)

说明：64位下sizeof(void *) != sizeof(int)，而在32位下是相同的。
如果需要一个指针大小的整数请使用intptr_t。

建议12.8 编译器，操作系统相关的代码独立出来

说明：为了程序的可移植性，建议将编译器，操作系统相关的代码从产品代码中独立出来。编译器特有的东西，如 gcc 的编译参数__thread， __attribute__等，如果需要做到支持多个平台，需要封装为宏或函数。
例如 gcc 的__thread 对应的 VC 开关为__declspec(thread)

```
__thread int number;
__declspec(thread) int number;
```

均表示 number 为线程私有存储变量，可采用如下宏来封装：

```
#ifdef WIN32
#define THREAD __declspec(thread)
#else
#define THREAD __thread
#endif

THREAD int number;
```

13. 全球化

本章提供给使用全球化特性的系统使用，凡是涉及全球化，可参照本章节的条款。

13.1多语言输入输出

原则13.1 使用正确的数据类型和类处理多语言字符和字符串

说明：使用wchar支持utf-16，char支持UTF-8字符集；

使用wchar_t保存字符时，使用wstring类处理字符串；

使用char保存字符时，使用string类处理字符串。

原则13.2 对字符进行处理时，需分配足够的内存空间并确保字符完整

说明：

1.使用char类型存储UTF-8多语言字符串时，注意分配足够的内存空间。UTF-8的每个字符占用空间长度在1-4字节之间，世界常用语言的字符长度在1-3个字节中，如英文字母长度为一个字节；阿拉伯语中字符长度为两个字节；汉字长度则为三个字节；

2.使用指针操作在char数组中存储的UTF8字符串时，需要根据存储的数据正确的增加和减少，确保指针始终指向字符的开始位置。

当一个字节前两位为10时，表示该字节不是UTF8字符的首字节；

UTF8字符的第一个字节第一位是0，表示字符长1字节；

UTF8字符的第一个字节前三位是110，表示字符长2字节；

UTF8字符的第一个字节前四位为1110，表示字符长3字节；

UTF8字符的第一个字节前五位为11110，表示字符长4字节。

规则13.1 使用标准库函数判断字符属性，使应用能够支持多区域

说明：在进行字符属性判断时，应使用locale::ctype类别里的标准库函数ctype::is来进行判断，而不要使用自己写的函数。

示例：

```
//好的例子：判断一个字符是否为字母
char char_val ='a';
locale loc1("English_Australia");
bool isalpha = use_facet<ctype<char>> (loc1).is(ctype_base::alpha, char_val);
```

规则13.2 对字符串进行本地化排序、比较和查找时使用有区域参数的locale::collate<>函数

说明：在不同的区域下，字符排序的规则不同，如在德语中字母“ß”排在s和t中间,同按照英语排序方式应当排在z之后。

示例：

```
//好的例子：按区域进行字符比较
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;
int main()
{
    locale loc ( "German_germany" );
    wchar_t * s1 = _T("Das ist weiß.");// ß the German sharp-s, it comes before z in the
German alphabet
    wchar_t * s2 = _T("Das ist weizzz.");
    int result1 = use_facet<collate<wchar_t> > ( loc ).
        compare ( s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
    cout << result1 << endl;

    locale loc2 ( "C" );
    int result2 = use_facet<collate<wchar_t> > ( loc2 ).
        compare (s1, &s1[_tcslen( s1 )-1 ],  s2, &s2[_tcslen( s2 )-1 ] );
    cout << result2 << endl;
}
```

建议13.1 处理Unicode数据时，注意检测BOM是否存在，避免产生多余字符或解码错误

说明：部分Unicode数据文件会使用BOM头用于表示文件的格式(UTF8的BOM为EF BB BF，UTF-16 BE中为FEFF，而UTF-16LE中为FFFE)。在读取数据时应当跳过BOM头，而在写文件的时候需要根据实际情况决定是否写入BOM头。

13.2单一版本

规则13.3 资源与程序逻辑代码分离

说明：资源指的是应用程序需要使用的区域敏感的数据。这些区域敏感的数据包括界面文字、快捷键和热键、消息提示信息、状态信息、图片、软件中包含的语音文件等。资源都是随区域的不同而变化的。因此，软件开发人员应当将资源独立于软件业务逻辑(核心代码)之外，同时两者应当无耦合关系，一套核心代码可以加载多种不同的本地化资源而使软件正常运行。

规则13.4 保持资源的语义完整性，不要拼接字符串资源

说明：拼接字符串资源往往造成句子的语义不完整、含义模糊，翻译人员无法准确理解。但是，在实际过程中，有些资源并不能够完全事先定义好，其中可能会用到运行时的一些结果。因此，强调不要采用拼接的方式来组装资源，而采用与语言无关的格式化方式来组装资源。

示例：

```
//不好的例子：
wchar_t * file = L"C:\\test.xml ";
wchar_t msg[200] =L"The file ";
wcscat(msg, file);
wcscat(msg, L"has some errors.");
wprintf(L"%s", msg);
//好的例子：
wchar_t buf[200];
wchar_t* msg = L"The file %s has some errors.";
wchar_t* file = L"C:\\test.xml";
int len = swprintf_s( buf, 200, msg, file );
wprintf(buf );
```

建议13.2 资源的键名中包含该资源的应用场景信息

说明：这些应用场景描述信息能够帮助翻译人员理解上下文，有助于提高翻译质量。例如，在资源键名的命名中包含界面元素的类型。

示例：

```
btnOK = OK; //btn是button的缩写，代表按钮的意思，表示【OK】这个单词是按钮上的文本
lblName = Name; //lbl是label的缩写，代表标签的意思，表示【Name】是标签上的文本
```

13.3时区夏令时

规则13.5 确保本地时间中时区夏令时信息完整

说明：保存、传递本地时间时应该确保夏令时信息完整，由于夏令时跳变可能导致本地时间空白或者重复，丢失夏令时信息将导致保存的时间与实际时间不符，出现计算结果异常

使用字符串表达的本地时间，应遵循如下格式：

YYYY-MM-DD<1SP>hh:mm:ss[.s’’s’’s’’][UTC{+|-}Hh:Mm]<1SP>[DST]

示例：

```
//不好的例子(复制tm结构体)
void copytm(struct tm* source, struct tm* target)
{
    target->year = source->Year;
    target->month = source->Month;
    target->day = source->Day;
    target->hour = source->Hour;
    target->minute = source->Minute;
    target->second = source->Second;
}
```

举例夏令时跳变在2:59:59后跳回2:00:00,期间经历了两次2:30:00，如果丢失了夏令时标志

(isDST设置为0则默认为非夏令时，isDST设置为-1则由系统自动判断，但在重复时间段内如2:30:00系统判断为夏令时内时间)，则在夏令时跳变重复时间中的一段无法保存

好的例子：在保存本地时间时必须同时保存夏令时信息，并在时间计算时作为信息使用。如上例至少须加上时区与夏令时信息

```
target->isDST = source->isDST;//夏令时信息
target-> tm_gmtoff= source-> tm_gmtoff; //时区偏移量
```

补充说明：如果操作系统支持时区夏令时，获取UTC时间后(time_t)，使用localtime函数可以自动填充tm结构体各项值。

规则13.6 避免使用本地时间直接进行时间计算

说明：进行时间差计算、时间增减等时间运算时，应避免直接使用本地时间直接进行计算。使用本地时间进行直接计算，可能因为夏令时跳变导致预期结果异常。应当基于UTC时间来计算避免夏令时跳变带来的影响。同样，不同时区的本地时间之间的计算，同样也必须转换为UTC时间后才可进行。

示例：

```
void AddOneHour(struct tm &localtime)
{
    localtime->hour += 1;
}
```

上述代码通过直接取本地时间小时数加上1来获1小时后的而本地时间。然而如果当本地时间1小时内发生了夏令时跳变，则可能导致预期结果不正确。

正确做法：先将本地时间转换为UTC时间，基于UTC进行计算，完成后转换回本地时间。

```
void AddOneHour(struct tm &tmlocaltime)
{
    time_t tt = mktime(&tmlocaltime);
    tt = tt + 3600;
    tmlocaltime = localtime(&tt);
}
```