C基础与规范

**参考资料：**

Google\_Cpp\_Style\_guide\_CN.pdf

C语言编程规范--华为技术有限公司\_7812089.pdf

高质量C++\_C编程指南（林锐博士）

MISRA-C-2004 工业标准的C编程规范（中文版）

# 第1章 命名约定

不要企图去发明一种完美的命名规则，当然，好的命名风格能让人更易阅读代码，最终要是统一规范，并坚持下去。

良好的设计 > 实现（数据结构、算法） > 命名

## 通用命名规则（General Naming Rules）

* **函数命名、变量命名、文件命具有描述性，不要过度缩写。**
* **除了常见的通用缩写以外，不使用单词缩写，不得使用汉语拼音。**

说明：较短的单词可通过去掉“元音”形成缩写，较长的单词可取单词的头几个字母形成缩写，一些单词有大家公认的缩写，常用单词的缩写必须统一。

协议中的单词的缩写与协议保持一致。对于某个系统使用的专用缩写应该在注视或者某处做统一说明。

* **类型和变量应该是名词，函数名可以用“命令性”动词。**

类型和变量名一般为名词：如FileOpener、num\_errors。

函数名通常是指令性的，如OpenFile()、SetNumErrors()，访问函数需要描述的更细致，要与其访问的变量相吻合。

* **建议 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。**

add/remove begin/end create/destroy

insert/delete first/last get/release

increment/decrement put/get add/delete

lock/unlock open/close min/max

old/new start/stop next/previous

source/target show/hide send/receive

source/destination copy/paste up/down

* **建议 尽量避免名字中出现数字编号，除非逻辑上的确需要编号。**

示例：如下命名，使人产生疑惑。

示例：如下命名，使人产生疑惑。

#define EXAMPLE\_0\_TEST\_

#define EXAMPLE\_1\_TEST\_

应改为有意义的单词命名

#define EXAMPLE\_UNIT\_TEST\_

#define EXAMPLE\_ASSERT\_TEST\_

* **建议 标识符前不应添加模块、项目、产品、部门的名称作为前缀。**

说明：很多已有代码中已经习惯在文件名中增加模块名，这种写法类似匈牙利命名法，导致文件名不可读，并且带来带来如下问题：

（1） 第一眼看到的是模块名，而不是真正的文件功能，阻碍阅读；

（2） 文件名太长；

（3） 文件名和模块绑定，不利于维护和移植。若foo.c进行重构后，从a模块挪到b模块，若foo.c中有模块名，则需要将文件名从a\_module\_foo.c改为b\_module\_foo.c。

* **规则 产品/项目组内部应保持统一的命名风格。**

说明：Unix like和windows like风格均有其拥趸，产品应根据自己的部署平台，选择其中一种，并在产品内部保持一致。

例外：即使产品之前使用匈牙利命名法，新代码也不应当使用。

* **建议 平台/驱动等适配代码的标识符命名风格保持和平台/驱动一致。**

说明：涉及到外购芯片以及配套的驱动，这部分的代码变动（包括为产品做适配的新增代码），应该保持原有的风格。

* **建议 重构/修改部分代码时，应保持和原有代码的命名风格一致。**

说明：根据源代码现有的风格继续编写代码，有利于保持总体一致。

## 2 文件命名（File Names）

* **文件命名统一采用全部小写，可以包含下划线。**

说明：c语言本身不管文件名大小写的问题，它只是把文件名匹配的任务交给操作系统来完成，所以这一切都取决于操作系统的文件系统。如MS的DOS、Windows系统不区分大小写，但是Linux和Unix系统则区分。

* **文件名应能反映文件的功能。**

通常，尽量让文件名更加明确，http\_server\_logs.h就比logs.h要好。定义类时文件名一般成对出现，如foo\_bar.h和foo\_bar.cc，对应类FooBar。

* **源文件与对应的头文件命名应一致。**

如，init\_system.c / init\_system.h 。

可接受的文件命名：

my\_useful\_class.cc

my-useful-class.cc

myusefulclass.cc

## 3 类型命名（Type Names）

**类型命名 类型+标示符名（大小写），typdef定义的类型用\_t作为后缀，**

**类、结构体、类型定义（typedef）、枚举、联合**：

**/\* 结构体 \*/**

struct Student {

char name;

int num;

};

typedef struct {

char name;

int num;

} Studet\_t

**/\* 枚举 \*/**

enum Errors {

OK = 0,

ERROR\_OUT\_OF\_MEMORY,

ERROR\_MALFORMED\_INPUT,

};

typedef enum {

OK = 0,

ERROR\_OUT\_OF\_MEMORY,

ERROR\_MALFORMED\_INPUT,

} Errors\_t;

typedef void\* Handle\_t;

union Segment {

xxx;

};

typedef union {

xxx;

} Segment\_t;

## 4 变量命名（Variable Names）

**变量名一律小写，单词间以下划线相连**，类的成员变量以下划线结尾，如

my\_exciting\_local\_variable、 my\_exciting\_member\_variable\_。

* **使用名词或者形容词＋名词方式命名变量。**
* **不建议使用匈牙利命名法。**

说明：`变量命名需要说明的是变量的含义，而不是变量的类型`。在变量命名前增加类型说明，反而降低了变量的可读性；更麻烦的问题是，如果修改了变量的类型定义，那么所有使用该变量的地方都需要修改**。**

### 4.1 普通变量命名

举例：

string table\_name; // OK - uses underscore.

string tablename; // OK - all lowercase.

string tableName; // **Bad** - mixed case.

### 4.2 结构体成员变量

结构体的数据成员可以和普通变量一样，不用像类那样尾部接下划线：

struct UrlTableProperties {

string name;

int num\_entries;

}

### 4.3 全局变量与静态变量

* **全局变量应增加“g\_”前缀。**
* **静态变量（局部）应增加“s\_”前缀**。

说明：增加g\_前缀或者s\_前缀，原因如下：

* 首先，全局变量十分危险，通过前缀使得全局变量更加醒目，促使开发人员对这些变量的使用更加小心。
* 其次，从根本上说，应当尽量不适用全局变量，增加g\_和s\_前缀，会使得全局变量的名字显得很丑陋，从而促使开发人员尽量少使用全局变量。

## 5 常量命名（Constant Names）

* **在名称前加k，如 kDaysInAWeek。**

所有编译时常量（无论是局部的、全局的还是类中的）和其他变量保持些许区别， k后接**大写字母开头的单词**。

const int kDaysInAWeek = 7;

## 6 函数命名（Function Names）

* **普通函数（regular functions，译者注，这里与访问函数等特殊函数相对）** 大小写混合；
* **存取函数（accessors and mutators）** 则要求**与变量名匹配**：

MyExcitingFunction()、MyExcitingMethod()、my\_exciting\_member\_variable()、set\_my\_exciting\_member\_variable()

### 6.1 普通函数

* 函数名以**大写字母开头，每个单词首字母大写，没有下划线**：

AddTableEntry()

DeleteUrl()

* 函数名可以用**“命令性”动词**。

OpenFile()、SetNumErrors()

### 6.2 存取函数

* 存取函数要**与存取的变量名匹配，小写+下划线**。

这儿摘录一个拥有实例变量num\_entries\_的类：

class MyClass {

public:

...

int num\_entries() const { return num\_entries\_; }

void set\_num\_entries(int num\_entries) { num\_entries\_ = num\_entries; }

private:

int num\_entries\_;

};

其他**短小的内联函数名也可以使用小写字母**， 例如， 在循环中调用这样的函数甚至都不要

缓存其值，小写命名就可以接受。

### 6.3 内联函数

* 内联函数名使用**小写字母+下划线**。

### 6.4 驱动/模块类函数

对于驱动的接口函数，或者模块类（如算法的操作）函数，采用大写（可多个）、下划线、加上操作名的方式：**设备名/算法名\_操作名()**

* 操作名一般采用：谓语（此时设备名作为宾语或者标明操作所属的模块）或者谓语+宾语/表语（此时设备名作为主语或者标明操作所属的模块） 等形式，如：

LCD\_Display(); //大写\_

LED\_RELAY\_Display(); //大写\_大写\_

LedRelay\_Display();

LinkList\_Create(); //大小写\_

## 7 枚举命名（ Enumerator Names）

* **枚举值**应全部大写，单词间以下划线相连： MY\_EXCITING\_ENUM\_VALUE。
* **枚举名**称属于类型，因此大小写混合： UrlTableErrors。

enum UrlTableErrors {

OK = 0,

ERROR\_OUT\_OF\_MEMORY,

ERROR\_MALFORMED\_INPUT,

};

## 8 宏命名

* **对于数值或者字符串等等常量的定义，建议采用全大写字母，单词之间加下划线„\_‟的方式命名（枚举同样建议使用此方式定义）。**

#define PI\_ROUNDED 3.0

* **除了头文件或编译开关等特殊标识定义，宏定义不能使用下划线„\_‟开头和结尾。**

注：google编码规范中不建议使用宏。

## 9 命名规则之外（Exceptions to Naming Rules）

* 总体规则：不要随意缩写，如果说ChangeLocalValue写作ChgLocVal还有情可原的话，把ModifyPlayerName写作MdfPlyNm就太过分了，除函数名可适当为动词外，其他命名尽量使用清晰易懂的名词；
* 宏、枚举等使用全部大写+下划线；
* 变量（含类、结构体成员变量）、文件、命名空间、存取函数等使用全部小写+下划线，类成员变量以下划线结尾，全局变量以g\_开头；
* 普通函数、类型（含类与结构体、枚举类型）、常量等使用大小写混合，不含下划线

## 所有常用的变量缩写全部标识出来 用txt文件的形式

# 第2章 文件组织—头文件

**对于C语言来说，头文件的设计体现了大部分的系统设计。**不合理的头文件布局是编译时间过长的根因，不合理的头文件实际上是不合理的设计。

确使用头文件可令代码在可读性、文件大小和性能上大为改观。

## 1 作用

**对于C语言来说，头文件的设计体现了大部分的系统设计。**不合理的头文件布局是编译时间过长的根因，不合理的头文件实际上是不合理的设计。

确使用头文件可令代码在可读性、文件大小和性能上大为改观。

**说明：**

用 #include <filename.h> 格式来引用标准库的头文件（编译器将从标准库目录开始搜

索）；如果不在当前目录或默认目录，则应包含文件所在的相对路径。

用 #include “filename.h” 格式来引用非标准库的头文件（编译器将从用户的工作目录

开始搜索）；如果不在当前目录或默认目录，则应包含文件所在的相对路径。

## 2 禁止

* **避免私有头文件的名字与标准库头文件的名字一样。如：#include "math.h"**
* **内部使用的函数（相当于类的私有方法）声明不应放在头文件中。**
* **内部使用的宏、枚举、结构定义不应放入头文件中。**
* **.c/.h文件禁止包含用不到的头文件。**
* **禁止在头文件中定义变量。**

说明：在头文件中定义变量，将会由于头文件被其他.c文件包含而导致变量重复定义。

* **变量的声明尽量不要放在头文件中，亦即尽量不要使用全局变量作为接口。**

变量是模块或单元的内部实现细节，不应通过在头文件中声明的方式直接暴露给外部，应通过函数接口的方式进行对外暴露。即使必须使用全局变量，也只应当在.c中定义全局变量，在.h中仅声明变量为全局的。

* **禁止头文件循环依赖。**

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

* **头文件不要使用非习惯用法的扩展名，如.inc作为头文件扩展名。**
* **只能通过包含头文件的方式使用其他.c提供的接口，禁止在.c中通过extern的方式使用外部函数接口、变量。**

说明：若a.c使用了b.c定义的foo()函数，则应当在b.h中声明extern int foo(int input)；并在a.c中通过#include 来使用foo。禁止通过在a.c中直接写extern int foo(int input);来使用foo，后面这种写法容易在foo改变时可能导致声明和定义不一致。

* **规则 禁止在extern "C"中包含头文件（可在外部包含），针对C++包含C的用法。**

说明：在extern "C"中包含头文件，会导致extern "C"嵌套，Visual Studio对extern "C"嵌套层次有限制，嵌套层次太多会编译错误。 在extern "C"中包含头文件，可能会导致被包含头文件的原有意图遭到破坏。例如，存在a.h和b.h两个头文件：

|  |  |
| --- | --- |
| /\* a.h头文件 \*/  #ifndef A\_H\_\_  #define A\_H\_\_  #ifdef \_\_cplusplus  void foo(int);  #define a(value) foo(value)  #else  void a(int)  #endif  #endif /\* A\_H\_\_ \*/ | /\* b.h头文件 \*/  #ifndef B\_H\_\_  #define B\_H\_\_  #ifdef \_\_cplusplus  extern "C" {  #endif  #include "a.h"  void b();  #ifdef \_\_cplusplus  }  #endif  #endif /\* B\_H\_\_ \*/ |

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

/\* 示例：错误的使用方式：\*/

extern “C”

{

#include “xxx.h”

...

}

/\* 正确的使用方式：\*/

#include “xxx.h”

extern “C”

{

...

}

## 3 遵守

* **每一个.c文件应有一个同名.h文件，用于声明需要对外公开的接口。**

说明： 如果一个.c文件不需要对外公布任何接口，则其就不应当存在，除非它是程序的入口，如main函数所在的文件。

* **头文件应当职责单一**

说明：头文件过于复杂，依赖过于复杂是导致编译时间过长的主要原因。很多现有代码中头文件过大，职责过多，再加上循环依赖的问题，可能导致为了在.c中使用一个宏，而包含十几个头文件。

示例：如下是某平台**定义WORD类型**的头文件：

#include <VXWORKS.H>

#include <VXWORKS.H>

#include <KERNELLIB.H>

#include <SEMLIB.H>

#include <INTLIB.H>

#include <TASKLIB.H>

#include <MSGQLIB.H>

#include <STDARG.H>

#include <FIOLIB.H>

#include <STDIO.H>

#include <STDLIB.H>

#include <CTYPE.H>

#include <STRING.H>

#include <ERRNOLIB.H>

#include <TIMERS.H>

#include <MEMLIB.H>

#include <TIME.H>

#include <WDLIB.H>

#include <SYSLIB.H>

#include <TASKHOOKLIB.H>

#include <REBOOTLIB.H>

…

这个头文件不但定义了基本数据类型WORD，还包含了stdio.h syslib.h等等不常用的头文件。如果工程中有10000个源文件，而其中100个源文件使用了stdio.h的printf，由于上述头文件的职责过于庞大，而WORD又是每一个文件必须包含的，从而导致stdio.h/yslib.h等可能被不必要的展开了9900次，大大增加了工程的编译时间。

* **头文件应当自包含**

说明：简单的说，**自包含**就是**任意一个头文件均可独立编译**。如果一个文件包含的某个头文件，还要包含另外一个头文件才能工作的话，就会增加交流障碍，给这个头文件的用户增添不必要的负担。

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

* **总是编写内部#include保护符（#define 保护）。**

说明：多次包含一个头文件可以通过认真的设计来避免。如果不能做到这一点，就需要采取阻止头文件内容被包含多于一次的机制。通常的手段是为每个文件配置一个宏，当头文件第一次被包含时就定义这个宏，并在头文件被再次包含时使用它以排除文件内容。所有头文件都应当使用#define 防止头文件被多重包含，命名格式为FILENAME\_H，为了保证唯一性，更好的命名是:

<PROJECT>\_<PATH>\_<FILE>\_H\_

* 注：没有在宏最前面加上“"，即使用FILENAME\_H代替`\_FILENAME\_H`，是因为一般以**"\_"**和**"\_\_"开头的标识符为系统保留或者标准库使用**，在有些静态检查工具中，若全局可见的标识符以"\_"开头会给出告警。

定义包含保护符时，应该遵守如下规则：

1. **保护符使用唯一名称；**
2. **不要在受保护部分的前后放置代码或者注释。**

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

#ifndef VOS\_INCLUDE\_TIMER\_TIMER\_H

#define VOS\_INCLUDE\_TIMER\_TIMER\_H

...

#endif

/\* 也可以使用如下简单方式保护: \*/

#ifndef TIMER\_H

#define TIMER\_H

...

#endif

* 例外情况：头文件的版权声明部分以及头文件的整体注释部分（如阐述此头文件的开发背景、使用注意事项等）可以放在保护符(#ifndef XX\_H)前面。
* **建议 一个模块通常包含多个.c文件，建议放在同一个目录下，目录名即为模块名。为方便外部使用者，建议每一个模块提供一个.h，文件名为目录名。**

说明：需要注意的是，这个.h并不是简单的包含所有内部的.h，它是为了模块使用者的方便，对外整体提供的模块接口。

以Google test（简称GTest）为例，GTest作为一个整体对外提供C++单元测试框架，其1.5版本的gtest工程下有6个源文件和12个头文件。但是它对外只提供一个gtest.h，只要包含gtest.h即可使用GTest提供的所有对外提供的功能，使用者不必关系GTest内部各个文件的关系，即使以后GTest的内部实现改变了，比如把一个源文件c拆成两个源文件，使用者也不必关心，甚至如果对外功能不变，连重新编译都不需要。

　对于有些模块，其内部功能相对松散，可能并不一定需要提供这个.h，而是直接提供各个子模块或者.c的头文件。

　比如产品普遍使用的VOS，作为一个大模块，其内部有很多子模块，他们之间的关系相对比较松散，就不适合提供一个vos.h。而VOS的子模块，如Memory（仅作举例说明，与实际情况可能有所出入），其内部实现高度内聚，虽然其内部实现可能有多个.c和.h，但是对外只需要提供一个Memory.h声明接口。

* **建议 如果一个模块包含多个子模块，则建议每一个子模块提供一个对外的.h，文件名为子模块名。**

说明：降低接口使用者的编写难度。

## 4 头文件排序

* **（华为） 同一产品统一包含头文件排列方式。**

说明：常见的包含头文件排列方式：功能块排序、文件名升序、稳定度排序。

/\* 示例1： 以升序方式排列头文件可以避免头文件被重复包含，如： \*/

#include <a.h>

#include <b.h>

#include <c/d.h>

#include <c/e.h>

#include <f.h>

/\* 示例2：以稳定度排序，建议将不稳定的头文件放在前面，如把产品的头文件放在平台的头文件前面，如下： \*/

#include <product.h>

#include <platform.h>

　相对来说，product.h修改的较为频繁，如果有错误，不必编译platform.h就可以发现product.h的错误，可以部分减少编译时间。

* **Google （参照这种）**

将包含次序标准化可增强可读性、避免隐藏依赖（hiddendependencies，译者注：隐藏依赖主要是指包含的文件中编译时），次序如下：C库、C++库、其他库的.h、项目内的.h。

项目内头文件应按照项目源代码目录树结构排列，并且避免使用UNIX文件路径.（当前目录）和..（父目录）。例如，google-awesome-project/src/base/logging.h应像这样被包含：

#include"base/logging.h"

dir/foo.cc的主要作用是执行或测试dir2/foo2.h的功能，foo.cc中包含头文件的次序如下：

dir2/foo2.h（优先位置）

C系统文件

C++系统文件

其他库头文件

本项目内头文件

这种排序方式可有效减少隐藏依赖，我们希望每一个头文件独立编译。**最简单的实现方式是将其作为第一个.h文件包含在对应的.cc中。**

dir/foo.cc和dir2/foo2.h通常位于相同目录下（像base/basictypes\_unittest.cc和base/basictypes.h），但也可在不同目录下。

相同目录下头文件按字母序是不错的选择。

举例来说，google-awesome-project/src/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"

## 5 内联函数与头文件的关系

### 5.1内联函数

内联函数必须放在.h文件中，**如果内联函数比较短，就直接放在.h中**。如果代码比较长，可以放到以-inl.h结尾的文件中。对于包含大量内联代码的类，可以有三个文件：

url\_table.h //Theclassdeclaration

url\_table.cc //Theclassdefinition

url\_table-inl.h //Inlinefunctionsthatincludelotsofcode

**只有当函数只有10行甚至更少时才会将其定义为内联函数（inline function）。**

**定义（Definition）：**当函数被声明为内联函数之后，编译器可能会将其内联展开，无需按通常的函数调用机制调用内联函数。

**优点：**当函数体比较小的时候，内联该函数可以令目标代码更加高效。对于**存取函数（accessor、mutator）**以及其他一些比较短的关键执行函数。

**缺点：**滥用内联将导致程序变慢，内联有可能是目标代码量或增或减，这取决于被内联的函数的大小。内联较短小的存取函数通常会减少代码量，但内联一个很大的函数（译者注：如果编译器允许的话）将戏剧性的增加代码量。在现代处理器上，由于更好的利用**指令缓存（instruction cache）**，小巧的代码往往执行更快。

**结论：**一个比较得当的处理规则是，不要内联超过10行的函数。对于析构函数应慎重对待，析构函数往往比其表面看起来要长，因为有一些隐式成员和基类析构函数（如果有的话）被调用！

另一有用的处理规则：内联那些包含循环或switch语句的函数是得不偿失的，除非在大多数情况下，这些循环或switch语句从不执行。

重要的是，虚函数和递归函数即使被声明为内联的也不一定就是内联函数。通常，递归函数不应该被声明为内联的（**译者注：递归调用堆栈的展开并不像循环那么简单，比如递归层数在编译时可能是未知的，大多数编译器都不支持内联递归函数**）。析构函数内联的主要原因是其定义在类的定义中，为了方便抑或是对其行为给出文档。

### 5.2 -inl.h文件

**复杂的内联函数的定义，应放在后缀名为-inl.h的头文件中。**

在头文件中给出内联函数的定义，可令编译器将其在调用处内联展开。然而，实现代码应完全放到.cc文件中，我们不希望.h文件中出现太多实现代码，除非这样做在可读性和效率上有明显优势。

如果内联函数的定义比较短小、逻辑比较简单，其实现代码可以放在.h文件中。例如，存取函数的实现理所当然都放在类定义中。出于实现和调用的方便，较复杂的内联函数也可以放到.h文件中，如果你觉得这样会使头文件显得笨重，还可以将其分离到单独的-inl.h中。这样即把实现和类定义分离开来，当需要时包含实现所在的-inl.h即可。

-inl.h文件还可用于函数模板的定义，从而使得模板定义可读性增强。

要提醒的一点是，**-inl.h和其他头文件一样，也需要#define保护。**

# 第3章 文件组织—源文件

## 1 数据排列顺序

**宏常量、宏函数、typedef声明、枚举声明、extern外部变量、全局变量、静态变量。**

//参照STM32

/\* Private typedef -----------------------------------------------------------\*/

/\* Private define ------------------------------------------------------------\*/

/\* Private macro -------------------------------------------------------------\*/

/\* Private variables ---------------------------------------------------------\*/

/\* Private function prototypes -----------------------------------------------\*/

/\* Private functions ---------------------------------------------------------\*/

## 2 函数-广度优先

最后是函数，函数应该以一种有意义的顺序排列。

相似的函数应该放在一起。与深度优先相比，我们应该首选广度优先方法。这里需要相当多的判断。

如果定义大量本质上无关的工具函数，可考虑按字母表顺序排列。

* **深度优先：**函数定义尽可能在他们的调用者前后；
* **广度优先：**抽象层次相似的函数放在一起。

# 第4章 标识符与作用域

## 1 标示符长度

* **（MISRA） 规则5.1（强制）： 标识符（内部的和外部的）的有效字符不能多于31。**

ISO 标准要求在内部标识符之间前31 个字符必须是不同的以保证可移植性。即使编译器支持，也不能超出这个限制。

ISO 标准要求外部标识符之间前6 个字符必须是不同的（忽略大小写）以保证最佳的可移植性。然而这条限制相当严格并被认为不是必须的。本规则的意图是为了在一定程度上放宽ISO 标准的要求以适应当今的环境，但应当确保 31 个字符/ 大小写的有效性是可以由实现所支持的。

使用标识符名称要注意的一个相关问题是发生在名称之间只有一个字符或少数字符不同的情况，特别是名称比较长时，当名称间的区别很容易被误读时问题就比较显著，比如 1（数字1 ）和 l（L 的小写）、0 和O、2 和Z、5 和S，或者 n 和h 。建议名称间的区别要显而易见。

## 2 标示符隐藏

* **（MISRA） 规则5.2（强制）： 具有内部作用域的标识符不应使用与具有外部作用域**

**的标识符相同的名称，这会隐藏了外部标识符。**

**总结：**

**函数内与函数外：**全局变量不应与局部变量重名，局部变量在不同函数内允许重名。

**函数内与函数内：**更深层的局部变量也不应与外层的局部变量重名。

## 3 标示符唯一

### 3.1 typedef

* **（MISRA） 规则5.3（强制）： typedef的名字应当是唯一的标识符。**

typedef的名称不能重用，不管是做为其他 typedef或者任何目的。例如：

{

typedef unsigned char uint8\_t;

}

{

typedef unsigned char uint8\_t; /\* Not compliant – redefinition \*/

}

{

unsigned char uint8\_t; /\* Not compliant – reuse of uint8\_t \*/

}

### 3.2 标签tag

* **（MISRA） 规则5.4（强制）： 标签（tag ）名称必须是唯一的标识符。**

程序中标签的名字不可重用，不管是做为另外的标签还是出于其他目的。ISO 9899:1990[2] 没有定义当一个聚合体的声明以不同形式的类型标识符（struct 或union）使用同一个标签时的行为。标签的所有使用或者用于结构类型标识符，或者用于联合类型标识符，例如：

struct stag { uint16\_t a; uint16\_t b; };

struct stag a1 = { 0, 0 }; /\* Compliant – compatible with above \*/

union stag a2 = { 0, 0 }; /\* Not compliant – not compatible with

previous declarations \*/

void foo (void)

{

struct stag { uint16\_t a; }; /\* Not compliant – tag stag redefined \*/

}

如果类型定义是在头文件中完成的，且头文件被多个源文件包含，那么规则不算违背。

* **（MISRA） 规则5.6（建议）： 一个命名空间中不应存在与另外一个命名空间中的标**

**识符拼写相同的标识符，除了结构和联合中的成员名**

**字。 （这个过于严格，仅供参考）**

命名空间与作用域（scope）是不同的，本规则不考虑作用域。例如，ISO C 允许在一个作用域内为标签（tag）和 typedef使用相同的标识符（vector）

typedef struct vector ( uint16\_t x; uint16\_t y ; uint16\_t z ; ) vector;

/\* Rule violation ^^ ^^ \*/

ISO C 定义了许多不同的命名空间（见ISO 9899 :1990 6.1.2.3 [2] ）。技术上，在彼此独立的命名空间中使用相同的名字以代表完全不同的项目是可能的，然而由于会引起混淆，通常不赞成这种做法，因此即使是在独立的命名空间中名字也不能重用。

下面给出了违背此规则的例子，其中value在不经意中代替了 record.value ：

struct { int16\_t key ; int16\_t value ; } record ;

int16\_t value; /\* Rule violation – 2nd use of value \*/

record.key = 1;

value = 0; /\* should have been record.value \*/

相比之下，下面的例子没有违背此规则，因为两个成员名字不会引起混淆：

struct device\_q { struct device\_q \*next ; /\* ... \*/ }

devices[N\_DEVICES] ;

struct task\_q {struct task\_q \*next ; /\* … \*/ }

tasks[N\_TASKS];

device[0].next = &devices[1];

tasks[0].next = &tasks[1];

* **（MISRA） 规则5.7（建议）： 不能重用标识符名字。**

不考虑作用域，系统内任何文件中不应重用标识符。本规则和规则 5.2、5.3、5.4、5.5 和5.6一同使用。

struct air\_speed

{

uint16\_t speed; /\* knots \*/

} \*x;

struct gnd\_speed

{

uint16\_t speed; /\* mph \*/

/\* Not Compliant – speed is in different units \*/

} \*y;

x->speed = y->speed;

当标识符名字用在头文件且头文件包含在多个源文件中时，不算违背本规则。使用严格的命名规范可以支持本规则。

### 3.3 静态对象static

**（MISRA） 规则5.5（建议）： 具有静态存储期的对象或函数标识符不能重用。**

不管作用域如何，具有静态存储期的标识符都**不应在系统内的所有源文件中重用。**它包含带有外部链接的对象或函数，及带有静态存储类标识符的任何对象或函数。

由于编译器能够理解这一点而且决不会发生混淆，那么对用户来说就存在着把不相关的变量以相同名字联系起来的可能性。

这种混淆的例子之一是，在一个文件中存在一个具有内部链接的标识符，而在另外一个文件中存在着具有外部链接的相同名字的标识符。

**总结：**

不同函数内或函数外定义的静态变量命名都应该不一样（尽管不同文件中的同名静态变量定义合法且为不同变量，但不要这么做；同一个文件中函数外和函数内同样如此）。

### 3.4 单字节标示符

* **规则 禁止使用单字节命名变量，但允许定义i、j、k作为局部循环变量。**

# 第5章 格式

码风格和格式确实比较随意， 但一个项目中所有人遵循同一风格是非常容易的， 作为个人未必同意下述格式规则的每一处， 但整个项目服从统一的编程风格是很重要的， 这样做才能让所有人在阅读和理解代码时更加容易。

## 1. 行长度（ Line Length）

**每一行代码字符数不超过80。**

我们也认识到这条规则是存有争议的， 但如此多的代码都遵照这一规则， 我们感觉一致性更重要。

**优点：** 提倡该原则的人认为强迫他们调整编辑器窗口大小很野蛮。 很多人同时并排开几个窗口，根本没有多余空间拓宽某个窗口，人们将窗口最大尺寸加以限定，一致使用80列宽，

为什么要改变呢？

**缺点：**反对该原则的人则认为更宽的代码行更易阅读， 80列的限制是上个世纪60年代的

大型机的古板缺陷；现代设备具有更宽的显示屏，很轻松的可以显示更多代码。

**结论：** 80个字符是最大值。例外：

1) 如果一行注释包含了超过80字符的命令或URL，出于复制粘贴的方便可以超过80字符；

2) 包含长路径的可以超出80列，尽量避免；

3) 头文件保护（防止重复包含第一篇）可以无视该原则。

## 2. 非ASCII字符（ Non-ASCII Characters）

**尽量不使用非ASCII字符，使用时必须使用 UTF-8格式。**

哪怕是英文，也不应将用户界面的文本硬编码到源代码中，因此非ASCII字符要少用。特殊情况下可以适当包含此类字符，如，代码分析外部数据文件时，可以适当硬编码数据文件中作为分隔符的非ASCII字符串；更常用的是（不需要本地化的）单元测试代码可能包含非ASCII字符串。此类情况下，应使用UTF-8格式，因为很多工具都可以理解和处理其编码， 十六进制编码也可以， 尤其是在增强可读性的情况下——如"\xEF\xBB\xBF"是Unicode的zero-width no-break space字符，以UTF-8格式包含在源文件中是不可见的。

## 3. 空格还是制表位（Spaces vs. Tabs）

只使用空格，每次缩进4个空格。

使用空格进行缩进，不要在代码中使用tabs，设定编辑器将tab转为空格。

## 4. 函数声明与定义（ Function Declarations and Definitions）

注：原google修订版。

**返回类型和函数名在同一行，合适的话，参数也放在同一行。**

函数看上去像这样：

ReturnType ClassName::FunctionName(Type par\_name1, Type par\_name2)

{

DoSomething();

...

}

**如果同一行文本较多，容不下所有参数：**

ReturnType ClassName::ReallyLongFunctionName(Type par\_name1,

Type par\_name2,

Type par\_name3)

{

DoSomething();

...

}

**甚至连第一个参数都放不下：**

ReturnType LongClassName::ReallyReallyReallyLongFunctionName(

Type par\_name1, // 2 TAB indent

Type par\_name2,

Type par\_name3)

{

DoSomething(); // 1TAB

...

}

**注意以下几点：**

1) 返回值总是和函数名在同一行；

2) **函数的**左大括号（open curly brace）和右大括号（close curly brace）单独成行。

3 圆括号与参数间没有空格，参数与参数之间有空格；

4) 函数声明和实现处的所有形参名称必须保持一致；

5) 所有形参应尽可能对齐；

6) 缺省缩进为~~2个空格~~1 TAB；

7) 独立封装的参数保持~~4个空格~~2 TAB的缩进。

译者注：关于UNIX/Linux风格为什么要把左大括号置于行尾（ .cc文件的函数实现处，

左大括号位于行首），我的理解是代码看上去比较简约，想想行首除了函数体被一对大括

号封在一起之外，只有右大括号的代码看上去确实也舒服； Windows风格将左大括号置

于行首的优点是匹配情况一目了然。

## 5. 函数调用（ Function Calls）

**尽量放在同一行，否则，将实参封装在圆括号中。**

函数调用遵循如下形式：

bool retval = DoSomething(argument1, argument2, argument3);

如果同一行放不下，可断为多行，后面每一行都和第一个实参对齐，左圆括号后和右圆括号前不要留空格：

bool retval = DoSomething(averyveryveryverylongargument1,

argument2, argument3);

如果函数参数比较多，可以出于可读性的考虑每行只放一个参数：

bool retval = DoSomething(argument1,

argument2,

argument3,

argument4);

如果函数名太长，以至于超过行最大长度，可以将所有参数独立成行：

if (...) {

...

...

if (...) {

DoSomethingThatRequiresALongFunctionName(

very\_long\_argument1, //4 TAB

argument2,

argument3,

argument4);

}

## 6. 条件语句（ Conditionals）

注：原google修订版。

**if和左圆括号间有个空格，右圆括号和左大括号（如果使用的话）间也要有个空格；**

**else另起一行；**

**所有if、else语句单独成行，均加上括号（哪怕只有一句）**

/\* 成对 \*/

if (condition) {

...

}

else { // else另起一行

...

}

/\* 单独 \*/

if (condition) {

return ;

}

## 7. 循环和开关选择语句（ Loops and Switch Statements）

**switch语句可以使用大括号分块；空循环体应使用{}或continue。**

switch语句中的case块可以使用大括号也可以不用，取决于你的喜好，使用时要依下文所

述。

如果有不满足case枚举条件的值，要总是包含一个default（如果有输入值没有case去处理，编译器将报警）。如果default永不会执行，可以简单的使用assert：

switch (var) {

case 0: {

...

break;

}

case 1: {

...break;

}

default: {

assert(false);

}

}

空循环体应使用{}或 continue，而不是一个简单的分号：

while (condition) {

// Repeat test until it returns false.

}

for (int i = 0; i < kSomeNumber; ++i) {} // Good - empty body.

while (condition) continue; // Good - continue indicates no logic.

while (condition); // Bad - looks like part of do/while loop.

## 8. 指针和引用表达式（ Pointers and Reference Expressions）

句点（.）或箭头（->）前后不要有空格，指针/地址操作符（\*、&）后不要有空格。

下面是指针和引用表达式的正确范例：

x = \*p;

p = &x;

x = r.y;

x = r->y;

注意：

1) 在访问成员时，句点或箭头前后没有空格；

2) 指针操作符\*或&后**没有空格**。

**在声明指针变量或参数时，星号与类型或变量名紧挨**：

// These are fine, space preceding.

char \*c;

const string &str;

译者注：个人比较习惯与变量紧挨的方式。

## 9. 布尔表达式（ Boolean Expressions）

如果一个布尔表达式超过标准行宽（ 80字符），如果断行要统一一下。

下例中，逻辑与（ &&）操作符总位于行首：

if ((this\_one\_thing > this\_other\_thing)

&& (a\_third\_thing == a\_fourth\_thing)

&& ( yet\_another & last\_one)) {

...

}

两个逻辑与（ &&）操作符都位于行首，可以考虑额外插入圆括号，合理使用的话对增强可读性是很有帮助的。

译者注：个人比较习惯逻辑运算符位于行首，逻辑关系一目了然，各人喜好而已，至于加

不加圆括号的问题，如果你对优先级了然于胸的话可以不加，但可读性总是差了些。

## 10. 函数返回值（ Return Values）

**return表达式中不要使用圆括号。**

函数返回时不要使用圆括号：

return x; // not return(x);

## 11. 变量及数组初始化（ Variable and Array Initialization）

选择=还是()。

需要做二者之间做出选择，下面的形式都是正确的：

int x = 3;

int x(3);

string name("Some Name");

string name = "Some Name";

## 12. 预处理指令（ Preprocessor Directives）

**预处理指令不要缩进，从行首开始。**

即使预处理指令位于缩进代码块中，指令也应从行首开始。

// Good - directives at beginning of line

if (lopsided\_score) {

#if DISASTER\_PENDING // Correct -- Starts at beginning of line

DropEverything();

#endif

BackToNormal();

}

## 13. 垂直留白（ Vertical Whitespace）

**垂直留白越少越好。**

这不仅仅是规则而是原则问题了： 不是非常有必要的话就不要使用空行。 尤其是： 不要在两个函数定义之间空超过2 行，函数体头、尾不要有空行，函数体中也不要随意添加空行。

基本原则是：同一屏可以显示越多的代码，程序的控制流就越容易理解。当然，过于密集的

代码块和过于疏松的代码块同样难看，取决于你的判断，但通常是越少越好。

函数头、尾不要有空行：

void Function()

{

// Unnecessary blank lines before and after

}

代码块头、尾不要有空行：

错误范例：

while (condition) {

// Unnecessary blank line after

}

if (condition) {

// Unnecessary blank line before

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

译者：首先说明，对于代码格式，因人、因系统各有优缺点，但同一个项目中遵循同一标准

还是有必要的：

1. 行宽原则上不超过80列，把22寸的显示屏都占完，怎么也说不过去；

2. 尽量不使用非ASCII字符， 如果使用的话， 参考UTF-8格式 （尤其是UNIX/Linux

下， Windows下可以考虑宽字符），尽量不将字符串常量耦合到代码中，比如独立出资

源文件，这不仅仅是风格问题了；

3. UNIX/Linux下无条件使用空格， MSVC的话使用Tab也无可厚非；

4. 函数参数、逻辑条件、初始化列表：要么所有参数和函数名放在同一行，要么所有参数

并排分行；

5. 除函数定义的左大括号可以置于行首外，包括函数/类/结构体/枚举声明、各种语句的

左大括号置于行尾，所有右大括号独立成行；

6. ./->操作符前后不留空格， \*/&不要前后都留，一个就可，靠左靠右依各人喜好；

7. 预处理指令/命名空间不使用额外缩进，类/结构体/枚举/函数/语句使用缩进；

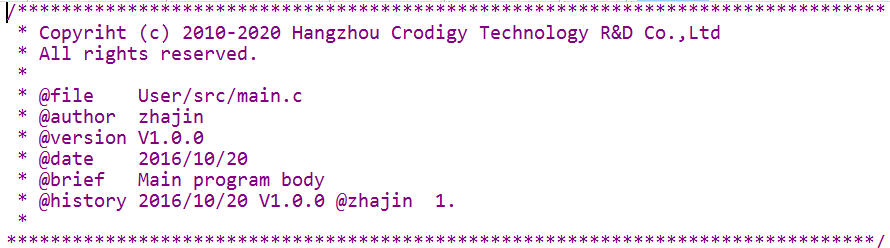
8. 初始化用=还是()依个人喜好，统一就好；

9. return不要加()；

10. 水平/垂直留白不要滥用，怎么易读怎么来。

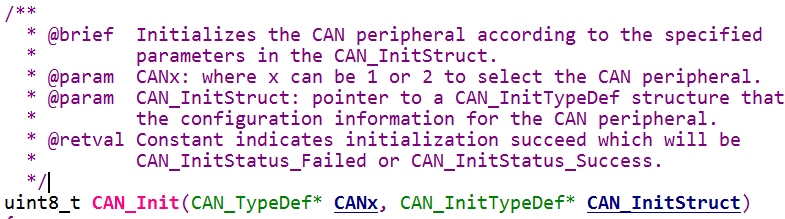
# 第6章 注释

## 1 文件顶部注释



## 函数注释

**函数头部**



**函数内部**

一般采用语句上面注释，/\* xxx \*/

注释暂时不做记录。

格式 工具

代码工具啥的

# 第7章 表达式和基本语句

读者可能怀疑：连if、for、while、goto、switch这样简单的东西也要探讨编程风格，是不是小题大做？

我真的发觉很多程序员用隐含错误的方式写表达式和基本语句，我自己也犯过类似的错误。

表达式和语句都属于C++/C的短语结构语法。它们看似简单，但使用时隐患比较多。本章归纳了正确使用表达式和语句的一些规则与建议。

## 1 运算符的优先级

C++/C语言的运算符有数十个，运算符的优先级与结合律如表4-1所示。注意一元运算符 + - \* 的优先级高于对应的二元运算符。

|  |  |  |
| --- | --- | --- |
| 优先级 | 运算符 | 结合律 |
| 从    高    到    低    排    列 | ( ) [ ] -> . | 从左至右 |
| ! ~ ++ -- （类型） sizeof  + - \* & | **从右至左** |
| \* / % | 从左至右 |
| + - | 从左至右 |
| << >> | 从左至右 |
| < <= > >= | 从左至右 |
| == != | 从左至右 |
| & | 从左至右 |
| ^ | 从左至右 |
| | | 从左至右 |
| && | 从左至右 |
| || | **从右至左** |
| ?: | **从右至左** |
| = += -= \*= /= %= &= ^=  |= <<= >>= | 从左至右 |

表4-1 运算符的优先级与结合律

* **【规则4-1-1】如果代码行中的运算符比较多，**用括号确定表达式的操作顺序，避免使用默认的优先级。

由于将表4-1熟记是比较困难的，为了防止产生歧义并提高可读性，应当用括号确定表达式的操作顺序。例如：

word = (high << 8) | low

if ((a | b) && (a & c))

## 2 复合表达式

如 a = b = c = 0这样的表达式称为复合表达式。允许复合表达式存在的理由是：（1）书写简洁；（2）可以提高编译效率。但要防止滥用复合表达式。

* **【规则4-2-1】**不要编写太复杂的复合表达式。

例如：

i = a >= b && c < d && c + f <= g + h ; // 复合表达式过于复杂

* **【规则4-2-2】**不要有多用途的复合表达式。

例如：

d = (a = b + c) + r ;

该表达式既求a值又求d值。应该拆分为两个独立的语句：

a = b + c;

d = a + r;

* **【规则4-2-3】**不要把程序中的复合表达式与“真正的数学表达式”混淆。

例如：

if (a < b < c) // a < b < c是数学表达式而不是程序表达式

并不表示

if ((a<b) && (b<c))

而是成了令人费解的

if ((a < b) < c)

* **自增或自减操作符**

x = b[i] + i++;

b[i] 的运算是先于还是后于 i ++ 的运算，表达式会产生不同的结果，把自增运算做为单独的语句，可以避免这个问题。 x = b[i] + i; i ++;

## 3 if语句

if语句是C++/C语言中最简单、最常用的语句，然而很多程序员用隐含错误的方式写if语句。本节以“与零值比较”为例，展开讨论。

### 3.1 布尔变量与零值比较

         **【规则4-3-1】**不可将布尔变量直接与TRUE、FALSE或者1、0进行比较。

根据布尔类型的语义，零值为“假”（记为FALSE），任何非零值都是“真”（记为TRUE）。TRUE的值究竟是什么并没有统一的标准。例如Visual C++ 将TRUE定义为1，而Visual Basic则将TRUE定义为-1。

假设布尔变量名字为flag，它与零值比较的标准if语句如下：

if (flag) // 表示flag为真

if (!flag) // 表示flag为假

**其它的用法都属于不良风格**，例如：

if (flag == TRUE)

if (flag == 1 )

if (flag == FALSE)

if (flag == 0)

### 3.2 整型变量与零值比较

         **【规则4-3-2】**应当将整型变量用“==”或“！=”直接与0比较。

假设整型变量的名字为value，它与零值比较的标准if语句如下：

if (value == 0)

if (value != 0)

不可模仿布尔变量的风格而写成

if (value) // 会让人误解 value是布尔变量

if (!value)

### 3.3 浮点变量与零值比较

         **【规则4-3-3】**不可将浮点变量用“==”或“！=”与任何数字比较。

**千万要留意，无论是float还是double类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。**

假设浮点变量的名字为x，应当将

if (x == 0.0) // 隐含错误的比较

转化为

if ((x>=-EPSINON) && (x<=EPSINON))

其中EPSINON是允许的误差（即精度）。

### 3.4 指针变量与零值比较

         **【规则4-3-4】应当将指针变量用“==”或“！=”与NULL比较**。

指针变量的零值是“空”（记为NULL）。尽管NULL的值与0相同，但是两者意义不同。假设指针变量的名字为p，它与零值比较的标准if语句如下：

if (p == NULL) // p与NULL显式比较，强调p是指针变量

if (p != NULL)

不要写成

if (p == 0) // 容易让人误解p是整型变量

if (p != 0)

或者

if (p) // 容易让人误解p是布尔变量

if (!p)

### 3.5 对if语句的补充说明

有时候我们可能会看到 if (NULL == p) 这样古怪的格式。不是程序写错了，是程序员为了防止将 if (p == NULL) 误写成 if (p = NULL)，而有意把p和NULL颠倒。编译器认为 if (p = NULL) 是合法的，但是会指出 if (NULL = p)是错误的，因为NULL不能被赋值。

程序中有时会遇到if/else/return的组合，应该将如下不良风格的程序

~~if (condition) {~~

~~return x;~~

~~}~~

~~return y;~~

改写为

if (condition) {

return x;

}

else {

return y;

}

或者改写成更加简练的

return (condition ? x : y);

## 4 循环语句的效率

C++/C循环语句中，for语句使用频率最高，while语句其次，do语句很少用。本节重点论述循环体的效率。提高循环体效率的基本办法是降低循环体的复杂性。

* **【建议4-4-1】**在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少CPU跨切循环层的次数。例如示例4-4(b)的效率比示例4-4(a)的高。

|  |  |
| --- | --- |
| for (row=0; row<100; row++) {  for (col=0; col<5; col++) {  sum = sum + a[row][col];  }  } | for (col=0; col<5; col++ ) {  for (row=0; row<100; row++) {  sum = sum + a[row][col];  }  } |

示例4-4(a) 低效率：长循环在最外层 示例4-4(b) **高效率：长循环在最内层**

* **【建议4-4-2】**如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体的外面。示例4-4(c)的程序比示例4-4(d)多执行了N-1次逻辑判断。并且由于前者老要进行逻辑判断，打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。如果N非常大，最好采用示例4-4(d)的写法，可以提高效率。如果N非常小，两者效率差别并不明显，采用示例4-4(c)的写法比较好，因为程序更加简洁。

|  |  |
| --- | --- |
| for (i=0; i<N; i++) {  if (condition)  DoSomething();  else  DoOtherthing();  } | if (condition) {  for (i=0; i<N; i++)  DoSomething();  }  else {  for (i=0; i<N; i++)  DoOtherthing();  } |

表4-4(c) 效率低但程序简洁 表4-4(d) 效率高但程序不简洁

## 5 for语句的循环控制变量

* **【规则4-5-1】**不可在for 循环体内修改循环变量，防止for 循环失去控制。

* **【建议4-5-1】**建议for语句的循环控制变量的取值采用**“半开半闭区间”**写法。

示例4-5(a)中的x值属于半开半闭区间“0 =< x < N”，起点到终点的间隔为N，循环次数为N。

示例4-5(b)中的x值属于闭区间“0 =< x <= N-1”，起点到终点的间隔为N-1，循环次数为N。

相比之下，示例4-5(a)的写法更加直观，尽管两者的功能是相同的。

|  |  |
| --- | --- |
| for (int x=0; x**<**N; x++) {  …  } /\* goog \*/ | for (int x=0; x**<=**N-1; x++) {  …  } /\* bad \*/ |

示例4-5(a) 循环变量属于半开半闭区间 示例4-5(b) 循环变量属于闭区间

## 6 switch语句

有了if语句为什么还要switch语句？

switch是多分支选择语句，而if语句只有两个分支可供选择。虽然可以用嵌套的if语句来实现多分支选择，但那样的程序冗长难读。这是switch语句存在的理由。

switch语句的基本格式是：

switch (variable) {

case value1 : …

break;

case value2 : …

break;

…

default : …

break;

}

* **【规则4-6-1】**每个case语句的结尾不要忘了加break，否则将导致多个分支重叠（除非有意使多个分支重叠）。
* **【规则4-6-2】**不要忘记最后那个default分支。即使程序真的不需要default处理，也应该保留语句 default : break; 这样做并非多此一举，而是为了防止别人误以为你忘了default处理。

## 7 goto语句

自从提倡结构化设计以来，goto就成了有争议的语句。首先，由于goto语句可以灵活跳转，如果不加限制，它的确会破坏结构化设计风格。其次，goto语句经常带来错误或隐患。它可能跳过了某些对象的构造、变量的初始化、重要的计算等语句，例如：

goto state;

String s1, s2; // 被goto跳过

int sum = 0; // 被goto跳过

…

state:

…

如果编译器不能发觉此类错误，每用一次goto语句都可能留下隐患。

很多人建议废除C++/C的goto语句，以绝后患。但实事求是地说，错误是程序员自己造成的，不是goto的过错。goto 语句至少有一处可显神通，它能从多重循环体中咻地一下子跳到外面，用不着写很多次的break语句; 例如

{ …

{ …

{ …

goto error;

}

}

}

error:

…

就象楼房着火了，来不及从楼梯一级一级往下走，可从窗口跳出火坑。所以我们主张少用、慎用goto语句，而不是禁用。

## 1 表达式——计算次序问题

 表达式的值在标准所允许的任何运算次序下都应该是相同的。

说明：除了少数操作符（函数调用操作符 ( )、 &&、 | |、 ? : 和 , （逗号）） 之外，子表达式所依据的运算次序是未指定的并会随时更改。注意，运算次序的问题不能使用括号来解决，因为这不是优先级的问题。

将复合表达式分开写成若干个简单表达式，明确表达式的运算次序，就可以有效消除非预期副作用。

1、 自增或自减操作符

示例：

x = b[i] + i++;

b[i] 的运算是先于还是后于 i ++ 的运算，表达式会产生不同的结果，把自增运算做为单独的语句，可以避免这个问题。

x = b[i] + i;

i ++;

## 2﹑ 函数参数

说明：函数参数通常从右到左压栈，但函数参数的计算次序不一定与压栈次序相同。

示例：

x = func( i++, i);

应该修改代码明确先计算第一个参数：

i++;

x = func(i, i);

## 3、 函数指针

说明：函数参数和函数自身地址的计算次序未定义。

示例：

p->task\_start\_fn(p++);

求函数地址p与计算p++无关，结果是任意值。必须单独计算p++：

p->task\_start\_fn(p);

p++;

## 4﹑ 函数调用

示例：

int g\_var = 0;

int fun1()

{

g\_var += 10;

return g\_var;

}

int fun2()

{

g\_var += 100;

return g\_var;

}

int x = fun1() + fun2();

编译器可能先计算fun1()，也可能先计算fun2()， 由于x的结果依赖于函数fun1()/fun2()的计算次序（fun1()/fun2()被调用时修改和使用了同一个全局变量），则上面的代码存在问题。

应该修改代码明确fun1/ fun2的计算次序：

int x = fun1();

x = x + fun2();

## 5、 嵌套赋值语句

说明：表达式中嵌套的赋值可以产生附加的副作用。不给这种能导致对运算次序的依赖提供任何机会的最好做法是，不要在表达式中嵌套赋值语句。

示例：

x = y = y = z / 3;

x = y = y++;

## 6、 volatile访问

说明：限定符volatile表示可能被其它途径更改的变量，例如硬件自动更新的寄存器。编译器不会优化对volatile变量的读取。

示例：下面的写法可能无法实现作者预期的功能：

/\* volume变量被定义为volatile类型\*/

UINT16 x = ( volume << 3 ) | volume; /\* 在计算了其中一个子表达式的时候， volume

的值可能已经被其它程序或硬件改变，导致另外一个子表达式的计算结果非预期，可能

无法实现作者预期的功能\*/

2 表达式——函数间调用

 函数调用不要作为另一个函数的参数使用，否则对于代码的调试、阅读都不利。

说明： 如下代码不合理， 仅用于说明当函数作为参数时，由于参数压栈次数不是代码可以控制的，可能造成未知的输出：

int g\_var;

int fun1()

{

g\_var += 10;

return g\_var;

}

int fun2()

{

g\_var += 100;

return g\_var;

}

int main(int argc, char \*argv[], char \*envp[])

{

g\_var = 1;

printf("func1: %d, func2: %d\n", fun1(), fun2());

g\_var = 1;

printf("func2: %d, func1: %d\n", fun2(), fun1());

}

上面的代码，使用断点调试起来也比较麻烦，阅读起来也不舒服，所以不要为了节约代码行，而写这种代码。

# 3 表达式——赋值语句

 赋值语句不要写在if等语句中，或者作为函数的参数使用。

说明： 因为if语句中，会根据条件依次判断，如果前一个条件已经可以判定整个条件，则后续条件语句不会再运行，所以可能导致期望的部分赋值没有得到运行。

示例：

int main(int argc, char \*argv[], char \*envp[])

{

int a = 0;

int b;

if ((a == 0) || ((b = fun1()) > 10)) {

printf("a: %d\n", a);

}

printf("b: %d\n", b);

}

作用函数参数来使用，参数的压栈顺序不同可能导致结果未知。

看如下代码，能否一眼看出输出结果会是什么吗？好理解吗？

int g\_var;

int main(int argc, char \*argv[], char \*envp[])

{

g\_var = 1;

printf("set 1st: %d, add 2nd: %d\n", g\_var = 10, g\_var++);

g\_var = 1;

printf("add 1st: %d, set 2nd: %d\n", g\_var++, g\_var = 10);

}

# 4 表达式——括号与默认优先级

 用括号明确表达式的操作顺序，避免过分依赖默认优先级。

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

1. 一元操作符，不需要使用括号

x = ~a; /\* 一元操作符，不需要括号\*/

x = -a; /\* 一元操作符，不需要括号\*/

2. 二元以上操作符，如果涉及多种操作符，则应该使用括号

x = a + b + c; /\* 操作符相同，不需要括号\*/

x = f ( a + b, c ) /\* 操作符相同，不需要括号\*/

if (a && b && c) /\* 操作符相同，不需要括号\*/

x = (a \* 3) + c + d; /\* 操作符不同，需要括号\*/

x = ( a == b ) ? a : ( a –b ); /\* 操作符不同，需要括号\*/

3 .即使所有操作符都是相同的，如果涉及类型转换或者量级提升，也应该使用括号控制计算的次序以下代码将3个浮点数相加：

/\* 除了逗号(,)，逻辑与(&&)，逻辑或(||)之外， C标准没有规定同级操作符是从左还是 从右开始计算，以上表达式存在种计算次序： f4 = (f1 + f2) + f3 或f4 = f1 + (f2 + f3)，

浮点数计算过程中可能四舍五入，量级提升，计算次序的不同会导致f4的结果不同， 以上表达式在不同编译器上的计算结果可能不一样，建议增加括号明确计算顺序\*/

f4 = f1 + f2 + f3;

.

5 表达式——赋值操作与布尔值

 赋值操作符不能使用在产生布尔值的表达式上。

说明：如果布尔值表达式需要赋值操作，那么赋值操作必须在操作数之外分别进行。 这可以帮助避免=和= =的混淆，帮助我们静态地检查错误。

示例：

/\* 正确示范 \*/

x = y;

if (x != 0) {

foo ();

}

不能写成：

if (( x = y ) != 0) {

foo ();

}

或者更坏的

if (x = y) {

foo ();

}