C 语言开发规范

在研究项目团队协作开发的情况下（这里的团队协作也适合于应用项目的开发），编程时应该强调的一个重要方面是程序的易读性，在保证软件速度等性能指标能满足用户需求的情况下，能让其他程序员容易读懂你所编写的程序。若研究项目小组的所有开发人员都遵循统一的、鲜明的一套编程风格，可以让协作者、后继者和自己一目了然，在很短的时间内看清楚程序结构，理解设计的思路，大大提高代码的可读性、可重用性、程序健壮性、可移植性、可维护性。

制定本编程规范的目的是为了提高软件开发效率及所开发软件的可维护性，提高软件的质量。本规范由程序风格、命名规范、注释规范、程序健壮性、可移植性、错误处理以及软件的模块化规范等部分组成。

本软件开发规范适合讨论 C/C++程序设计。

1 文件结构

每个 C++/C 程序通常分为两个文件。一个文件用于保存程序的声明（declaration），称为头文件。另一个文件用于保存程序的实现（implementation），称为定义（definition）文件。

C++/C 程序的头文件以“.h”为后缀，C 程序的定义文件以“.c”为后缀，C++程序的定义文件通常以“.cpp”为后缀（也有一些系统以“.cc”或“.cxx”为后缀）。

1.1 文件信息声明

文件信息声明位于头文件和定义文件的开头（参见示例 1-1），主要内容有：

（1） 版权信息；
（2） 文件名称，项目代码，摘要，参考文献；
（3） 当前版本号，作者/修改者，完成日期；
（4） 版本历史信息；
（5） 主要函数描述。

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
// Copyright (c) 2004, Department of Mathematics, Zhejiang University
// All rights reserved.
//
// Filename     : filename.h
// Project Code : The project code about this file
// Abstract     : Describe the content of this file summarily
// Reference    : ......
//
// Version      : 1.1
// Author       : the name of author(mender)
```

```
// Accomplished date :  September 2, 2004
//
// Replaced version  :  1.0
// Original Author   : the name of original author(mender)
// Accomplished date : September 10, 2003
//
// Main functions    :
//  Function 1 Return code      Function name(Parameter Explain)
//  Function 2 Return code      Function name(Parameter Explain)
//  ...
//  Function n Return code      Function name(Parameter Explain)
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

示例 1-1 文件信息声明

☆ **【规则 1.1-1】** 文件信息声明以两行斜杠开始，以两行斜杠结束，每一行都以两个斜杠开始；

☆ **【规则 1.1-2】** 文件信息声明包含五个部分，各部分之间以一空行间隔；

☆ **【规则 1.1-3】** 在主要函数部分描述了文件所包含的主要函数的声明信息，如果是头文件，这一部分是可以省略的。

1.2 头文件的结构

头文件由三部分内容组成：

(1) 头文件开头处的文件信息声明(参见示例 1-1)；

(2) 预处理块；

(3) 函数和类结构声明等。

假设头文件名称为 filesystem.h，头文件的结构参见示例 1-2。

☆ **【规则 1.2-1】** 为了防止头文件被重复引用，应当用 ifndef/define/endif 结构产生预处理块；“#ifndef”或者“#define”后以 TAB 键代替 SPACE 键做空格；如果头文件名称是由多个单词组成，则各单词间以下划线“_”连接，例如有头文件名称为“filesystem.h”，则定义如下：“#ifndef _FILE_SYSTEM_H_”；

☆ **【规则 1.2-2】** 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)；

☆ **【规则 1.2-3】** 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)；

☆ **【建议 1.2-1】** 头文件中只存放“声明”而不存放“定义”；

☆ **【建议 1.2-1】** 头文件中应包含所有定义文件所定义的函数声明，如果一个头文件对应多个定义文件，则不同定义文件内实现的函数要分开声明，并作注释以解释所声明的函数从属于那一个定义文件；

☆ **【建议 1.2-3】** 宏定义和函数声明分离，在两个头文件中定义，如果没有类成员函数，可以将类和结构的定义与函数声明分离，也就是说一个头文件专用于宏定义，一个头文件专用于类和结构的定义，一个头文件专

用于函数声明；

☆ **【建议 1.2-4】** 在 C++ 语法中，类的成员函数可以在声明的同时被定义，并且自动成为内联函数。这虽然会带来书写上的方便，但却造成了风格不一致，弊大于利。建议将成员函数的定义与声明分开，不论该函数体有多么小。

头文件的结构如下：

```
//文件信息声明见示例 1-1，此处省略。
#ifndef _FILE_SYSTEM_H_ //avoid referencing the file filesystem.h repeat
#define _FILE_SYSTEM_H_

#include <math.h>          //reference standard head file
…
#include “myheader.h”      //reference non-standard head file
…
void Function1(…);         //global function declare
…
class CBox                 //class structure decalre
{
…
};
#endif
```

示例 1-2 C++/C 头文件的结构

1.3 定义文件的结构

定义文件有三部分内容：

(1) 定义文件开头处的文件信息声明(参见示例 1-1)；
(2) 对一些头文件的引用；
(3) 程序的实现体（包括数据和代码）。

假设定义文件的名称为 filesystem.c，定义文件的结构参见示例 1-3。

```
//文件信息声明见示例 1-1，此处省略。

#include “filesystem.h”      //reference a head file
…

//global function realization
void Function1(…)
{
  …
}

//class member function realization
void CBox::Draw(…)
{
  …
}
```

示例 1-3 C++/C 定义文件的结构

1.4 头文件的作用

早期的编程语言如 Basic、Fortran 没有头文件的概念，C++/C 语言的初学者虽然会用使用头文件，但常常不明其理。这里对头文件的作用略作解释：

（1） 通过头文件来调用库功能。在很多场合，源代码不便（或不准）向用户公布，只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能，而不必关心接口怎么实现的。编译器会从库中提取相应的代码；

（2） 头文件能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，这一简单的规则能大大减轻程序员调试、改错的负担。

1.5 目录结构

如果一个软件的头文件数目比较多（如超过十个），通常应将头文件和定义文件分别保存于不同的目录，以便于维护。

例如可将头文件保存于 include 目录，将定义文件保存于 source 目录（可以是多级目录）。

如果某些头文件是私有的，它不会被用户的程序直接引用，则没有必要公开其“声明”。为了加强信息隐藏，这些私有的头文件可以和定义文件存放于同一个目录。

2 命名规则

比较著名的命名规则当推“匈牙利” 命名法，该命名规则的主要思想是“在变量和函数名中加入前缀以增进人们对程序的理解”。例如所有的字符变量均以 ch 为前缀，若是指针变量则追加前缀 p。如果一个变量由 ppch 开头，则表明它是指向字符指针的指针。

“匈牙利”法最大的缺点是烦琐，例如

```
int    i,  j,  k;
float  x,  y,  z;
```

倘若采用“匈牙利”命名规则，则应当写成

```
int    iI,  iJ,  ik;  // 前缀 i 表示 int 类型
float  fX,  fY,  fZ;  // 前缀 f 表示 float 类型
```

如此烦琐的程序会让绝大多数程序员无法忍受。

总的说来，没有一种命名规则可以让所有的程序员赞同，且命名规则对软件产品而言并不是“成败悠关”的事，而且在不同的平台和不同的环境下编写的程序所应遵循的规则也不尽相同，所以我们只是追求制定一种令大多数项目成员满意的命名规则，并在项目中贯彻实施。

2.1 共性原则

本节论述的共性规则是被大多数程序员采纳的，我们应当在遵循这些共性规则的前提下，再扩充特定的规则，如 2.2 节

☆ **【规则 2.1-1】** 标识符应当直观且可以拼读，可望文知意，不必进行“解码”；
☆ **【规则 2.1-2】** 标识符的长度应当符合“min-length && max-information”原则；
☆ **【规则 2.1-3】** 命名规则尽量与所采用的操作系统或开发工具的风格保持一致；
☆ **【规则 2.1-4】** 程序中不要出现仅靠大小写区分的相似的标识符。
☆ **【规则 2.1-5】** 程序中不要出现标识符完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但会使人误解；
☆ **【规则 2.1-6】** 变量的名字应当使用“名词”或者“形容词＋名词”；
☆ **【规则 2.1-7】** 全局函数的名字应当使用“动词”或者“动词＋名词”(动宾词组)；
☆ **【规则 2.1-8】** 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；
☆ **【建议 2.1-1】** 尽量避免名字中出现数字编号，如 Value1,Value2 等，除非逻辑上的确需要编号；

注：

2.1.1 标识符最好采用英文单词或其组合，便于记忆和阅读，切忌使用汉语拼音来命名，程序中的英文单词一般不要太复杂，用词应当准确，例如不要把 CurrentValue 写成 NowValue；

2.1.2 标示符的长度应当以最小的长度实现最多信息，一般来说，长名字能更好地表达含义，但并非长的变量名就一定要比短的变量名要好，此外单字符的名字也是有用的，常见的如 i,j,k,m,n,x,y,z 等，它们通常可用作函数内的局部变量；

2.1.3 不同的操作系统的程序设计风格是不一样的，例如 Windows 应用程序的标识符通常采用“大小写”混排的方式，如 AddChild，而 Unix 应用程序的标识符通常采用“小写加下划线”的方式，如 add_child，别把这两类风格混在一起使用；

2.2 Windows 变量命名规则

☆ 【规则 2.2-1】 变量的命名规则要求采用“匈牙利法则”，即开头字母用变量的类型，其余部分用变量的英文意思或其英文意思的缩写，尽量避免采用中文拼音，要求单词的第一个字母大写；

即：变量名＝变量类型＋变量英文意思(或缩写)

变量类型请参见附表 1－变量类型表；

☆ 【规则 2.2-2】 类名和函数名用大写字母开头的单词组合而成；对 struct、union、class 变量的命名要求定义的类型用大写，结构采用 S 开头，联合体采用 U 开头，类采用 C 开头；

例如：

```
struct SPoint
{
    int   m_nX;
    int   m_nY;
};
union URecordLen
{
    BYTE    m_byRecordNum;
    BYTE    m_byRecordLen;
}
class CNode
{
    //类成员变量或成员函数
};
```

☆ 【规则 2.2-3】 指针变量命名的基本原则为：

一重指针变量的基本原则为：

变量名＝ “p”＋变量类型前缀＋命名

对多重指针变量的基本原则为：

二重指针：

变量名＝“pp”＋变量类型前缀＋命名

三重指针：

变量名＝“ppp”＋变量类型前缀＋命名

.....

例如一个 short*型的变量应该表示为 pnStart；

☆ 【规则 2.2-4】 全局变量用 g_开头；例如一个全局的长型变量定义为 g_lFileNum，

即：变量名＝g_＋变量类型＋变量的英文意思(或缩写)；

☆ 【规则 2.2-5】 静态变量采用 s_开头；例如一个静态的指针变量定义为 s_plPrevInst，

即：变量名＝s_＋变量类型＋变量的英文意思(或缩写)；

☆ 【规则 2.2-6】 类成员变量采用 m_开头；例如一个长型成员变量定义为 m_lCount，

即：变量名＝m_＋变量类型＋变量的英文意思(或缩写)；

☆ 【规则 2.2-7】 对 const 的变量要求在变量的命名规则前加入 c_(若作为函数的输入参数，可以不加)，

即：变量名＝c_＋变量命名规则，例如：

const char* c_szFileName；

☆ 【规则 2.2-8】 对枚举类型(enum)中的变量，要求用枚举变量或其缩写做前缀，且用下划线隔离变量名，所有枚举类型都要用大写，例如：

```
enum      EMDAYS
{
    EMDAYS_MONDAY;
    EMDAYS_TUESDAY;
```

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

☆ **【规则 2.2-9】** 对常量(包括错误的编码)命名，要求常量名用大写，常量名用英文意思表示其意思，用下划线分割单词，例如：

```
#define  CM_7816_OK      0x9000;
```

☆ **【规则 2.2-10】** 为了防止某一软件库中的一些标识符和其它软件库中的冲突，可以为各种标识符加上能反映软件性质的前缀。例如三维图形标准 OpenGL 的所有库函数均以 gl 开头，所有常量（或宏定义）均以 GL 开头。

3 程序风格

程序风格虽然不会影响程序的功能，但会影响程序的可读性，追求清晰、美观，是程序风格的重要构成因素。

3.1 空行

空行起着分隔程序段落的作用。空行得体（不过多也不过少）将使程序的布局更加清晰。空行不会浪费内存，虽然打印含有空行的程序是会多消耗一些纸张，但是值得。

☆ **【规则 3.1-1】** 在每个类声明之后、每个函数定义结束之后都要加空行。参见示例 3.1(a)；

☆ **【规则 3.1-2】** 在一个函数体内，逻辑上密切相关的语句之间不加空行，其它地方应加空行分隔。参见示例 3.1(b)；

```
// blank line
void Function1(…)
{
  …
}
// blank line
void Function2(…)
{
  …
}
// blank line
void Function3(…)
{
  …
}
```

示例 3.1(a) 函数之间的空行

```
// blank line
while (condition)
{
    statement1;
    // blank line
    if (condition)
    {
        statement2;
    }
    else
    {
        statement3;
    }
    // blank line
    statement4;
}
```

示例 3.1(b) 函数内部的空行

3.2 代码行

☆ **【规则 3.2-1】** 一行代码只做一件事情，如只定义一个变量，或只写一条语句,这样的代码容易阅读，并且方便于写注释；

☆ **【规则 3.2-2】** if、for、while、do 等语句自占一行，执行语句不得紧跟其后，不论执行语句有多少都要加{}，这样可以防止书写失误；

☆ **【规则 3.2-3】** if、for、while、do 等语句的“{”要单独占用一行；

☆ **【建议 3.2-1】** 所有函数内的变量都在函数开始处定义；

☆ **【建议 3.2-2】** 尽可能在定义变量的同时初始化该变量(就近原则)，如果变量的引用

处和其定义处相隔比较远，变量的初始化很容易被忘记。如果引用了未被初始化的变量，可能会导致程序错误，本建议可以减少隐患。

示例 3.2(a)为风格良好的代码行，示例 3.2(b)为风格不良的代码行。

int nWidth; // width
int nHeight; // height
int nDepth; // depth
	int nWidth,nHight,nDepth;//width,height,depth

x = a + b;
y = c + d;
z = e + f;
	X = a + b; y = c + d; z = e + f;

if (nWidth < nHight)
{
  DoSomething();
}
	if (nWidth < nHight) DoSomething();

for (initialization; condition; update)
{
  DoSomething();
}
// blank line
Other();
	for (initialization; condition; update)
	  DoSomething();
	Other();

示例 3.2(a) **风格良好**的代码行　　示例 3.2(b) **风格不良**的代码行

3.3 代码行内的空格

☆ **【规则 3.3-1】** 关键字之后要留空格，象 const、virtual、inline、case 等关键字之后至少要留一个空格，否则无法辨析关键字，象 if、for、while 等关键字之后应留一个空格再跟左括号‘(’，以突出关键字；

☆ **【规则 3.3-2】** 函数名之后不要留空格，紧跟左括号‘(’，以与关键字区别；

☆ **【规则 3.3-3】** ‘(’向后紧跟，‘)’、‘,’、‘;’向前紧跟，紧跟处不留空格；

☆ **【规则 3.3-4】** ‘,’之后要留空格，如 Function(x, y, z)，如果‘;’不是一行的结束符号，其后要留空格，如 for (initialization; condition; update)；

☆ **【规则 3.3-5】** 赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如“=”、“+=” “>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<”，“^”等二元操作符的前后应当加空格；

☆ **【规则 3.3-6】** 一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格；

☆ **【规则 3.3-7】** 象“[]”、“.”、“->”这类操作符前后不加空格；

☆ **【建议 3.3-1】** 对于表达式比较长的 for 语句和 if 语句，为了紧凑起见可以适当地去掉一些空格，如 for (i=0; i<10; i++)和 if ((a<=b) && (c<=d))

```
void Func1(int x, int y, int z);          // favorable style
void Func1 (int x,int y,int z);           // ill style
```

if (year >= 2000)	// favorable style
if(year>=2000)	// ill style
if ((a>=b) && (c<=d))	// favorable style
if(a>=b&&c<=d)	// ill style
for (i=0; i<10; i++)	// favorable style
for(i=0;i<10;i++)	// ill style
for (i = 0; I < 10; i ++)	// favorable style
x = a < b ? a : b;	// favorable style
x=a<b?a:b;	// ill style
int *x = &y;	// favorable style
int * x = & y;	// ill style
array[5] = 0;	// Do not use array [5] = 0;
a.Function();	// Do not use a . Function();
b->Function();	// Do not use b -> Function();

示例 3.3 代码行内的空格

3.4 对齐

☆ **【规则 3.4-1】** 程序的分界符‘{’和‘}’应独占一行并且位于同一列，同时与引用它们的语句左对齐；

☆ **【规则 3.4-2】** {}之内的代码块在‘{’右边数格处左对齐；

☆ **【规则 3.4.3】** 代码的的对齐采用 TAB 键而不采用空格键对齐，一般 TAB 键设置为向后空 4 个空格。

示例 3.4(a)为风格良好的对齐，示例 3.4(b)为风格不良的对齐。

void Function(int x)
{
… // program code
}
	void Function(int x){
	… // program code
	}

if (condition)
{
… // program code
}
else
{
… // program code
}
	if (condition){
	… // program code
	}
	else {
	… // program code
	}

for (initialization; condition; update)
{
… // program code
}
	for (initialization; condition; update){
	… // program code
	}

While (condition)
{
  … // program code
}
	while (condition){
	  … // program code
	}

如果出现嵌套的{}，则使用缩进对齐，如：
{
…
{
…
}
…
}	

示例 3.4(a) **风格良好**的对齐　　示例 3.4(b) **风格不良**的对齐

3.5 长行拆分

☆ **【规则 3.5-1】** 代码行最大长度宜控制在 70 至 80 个字符以内；

☆ **【规则 3.5-2】** 长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符），拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

```
if ((very_longer_variable1 >= very_longer_variable12)
    && (very_longer_variable3 <= very_longer_variable14)
    && (very_longer_variable5 <= very_longer_variable16))
{
    DoSomething();
}
```

```
virtual CMatrix CMultiplyMatrix (CMatrix leftMatrix,
                                 CMatrix rightMatrix);
```

```
for (very_longer_initialization;
     very_longer_condition;
     very_longer_update)
{
    DoSomething();
}
```

示例 3.5 长行的拆分

3.6 修饰符的位置

修饰符 * 和 & 应该靠近数据类型还是该靠近变量名，是个有争议的活题，若将修饰符 * 靠近数据类型，例如：int* x; 从语义上讲此写法比较直观，即 x 是 int 类型的指针，上述写法的弊端是容易引起误解，例如：int* x, y; 此处 y 容易被误解为指针变量。虽然将 x 和 y 分行定义可以避免误解，但并不是人人都愿意这样做。

☆ 【规则 3.6-1】　　应当将修饰符 * 和 & 紧靠变量名；

3.7 注释

C 语言的注释符为“/*…*/”。C++语言中，程序块的注释常采用“/*…*/”，行注释一般采用“//…”。注释通常用于：
（1）版本、版权声明；
（2）函数接口说明；
（3）重要的代码行或段落提示。
虽然注释有助于理解代码，但注意不可过多地使用注释。参见示例 3.7。

☆ 【规则 3.7-1】　　注释是对代码的“提示”，而不是文档，程序中的注释不可喧宾夺主，注释太多了会让人眼花缭乱，注释的花样要少；
☆ 【规则 3.7-2】　　如果代码本来就是清楚的，则不必加注释；例如
```
i++;    // i 加 1，多余的注释
```
☆ 【规则 3.7-3】　　边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性，不再有用的注释要删除；
☆ 【规则 3.7-4】　　注释应当准确、易懂，防止注释有二义性，错误的注释不但无益反而有害；
☆ 【规则 3.7-5】　　尽量避免在注释中使用缩写，特别是不常用缩写；
☆ 【规则 3.7-6】　　注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方；
☆ 【规则 3.7-8】　　当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读；
☆ 【建议 3.7-1】　　对于多行代码的注释，尽量不采用“/*...*/”，而采用多行“//”注释，这样虽然麻烦，但是在做屏蔽调试时不用查找配对的“/*...*/”。

示例 3.7 程序的注释

```
////////////////////////////////////////////////////////////////////
// Function capacity：
// Parameter declare：
// Return value ：
////////////////////////////////////////////////////////////////////
void Function(float x, float y, float z)
{
   …
}
```

```
if (…)
{
   …
   while (…)
   {
      …
   } // end of while
   …
} // end of if
```

3.7.1 文件头的注释

文件头的注释请参见 1.1，文件头的注释是以两行斜杠开始，以两行斜杠结束(以区别于函数的注释)。

3.7.2 函数头的注释

一般说来每个函数都应该做详细的注释，函数头的注释是以一行斜杠开始，以一行斜杠

结束，注释的内容包括“功能”，“参数”，“返回值”，“设计思想”，“调用函数”，“日期”，“修改记录”等几个方面，函数头的注释格式如下：

```
//////////////////////////////////////////////////////////////////////////////////////////
// Function capacity   ：Describe the function capacity
// Parameter declare   ：
//     parameter 1：        Describe the function of parameter ( input/output parameter )
//     parameter 2：Describe the function of parameter ( input/output parameter )
//     ......
// Return value     ：  Describe the possible return value
// Designed idea   ：  Describe designed idea about the function
// Author             ：
// Creation date     ：  Creation date(YY-MM-DD)
// Transferred function：List the sub-function in the function
// Modification record：
//     (一)Mender 1：      Modified date：      modified content
//////////////////////////////////////////////////////////////////////////////////////////
```

4 函数设计

函数是 C++/C 程序的基本功能单元，其重要性不言而喻。函数设计的细微缺点很容易导致该函数被错用，所以光使函数的功能正确是不够的。本章重点论述函数的接口设计和内部实现的一些规则。

函数接口的两个要素是参数和返回值。C 语言中，函数的参数和返回值的传递方式有两种：值传递（pass by value）和指针传递（pass by pointer）。C++ 语言中多了引用传递（pass by reference）。由于引用传递的性质象指针传递，而使用方式却象值传递，初学者常常迷惑不解，容易引起混乱，请先阅读 6.6 节“引用与指针的比较”。

4.1 参数的规则

☆ **【规则 4.1-1】** 参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字，如果函数没有参数，则用 void 填充；例如：

```
void SetValue(int nWidth, int nHeight);// 良好的风格
void SetValue(int, int);               // 不良的风格
float GetValue(void);                  // 良好的风格
float GetValue();                      // 不良的风格
```

☆ **【规则 4.1-2】** 参数命名要恰当，顺序要合理；

例如编写字符串拷贝函数 StringCopy，它有两个参数，如果把参数名字起为 str1 和 str2，例如：

```
void StringCopy(char *str1, char *str2);
```

那么我们很难搞清楚究竟是把 str1 拷贝到 str2 中，还是刚好倒过来，可以把参数名字起得更有意义，如叫 strSource 和 strDestination。这样从名字上就可以看出应该把 strSource 拷贝到 strDestination。还有一个问题，这两个参数那一个该在前那一个该在后？参数的顺序要遵循程序员的习惯。一般地，应将目的参数放在前面，源参数放在后面。如果将函数声明为：

```
void StringCopy(char *strSource, char *strDestination);
```

别人在使用时可能会不假思索地写成如下形式：

```
char str[20];
StringCopy(str, “Hello World”);  // 参数顺序颠倒
```

☆ **【规则 4.1-3】** 如果参数是指针，且仅作输入用，则应在类型前加 const，以防止该指针在函数体内被意外修改。例如：

```
void StringCopy(char *strDestination, const char *strSource);
```

☆ **【规则 4.1-4】** 如果输入参数以值传递的方式传递对象，则宜改用“const &”方式来传递，这样可以省去临时对象的构造和析构过程，从而提高效率；

☆ **【建议 4.1-1】** 避免函数有太多的参数，参数个数尽量控制在 5 个以内。如果参数太多，在使用时容易将参数类型或顺序搞错；

☆ **【建议 4.1-2】** 尽量不要使用类型和数目不确定的参数；

C 标准库函数 printf 是采用不确定参数的典型代表，其原型为：

```
int printf(const chat *format[, argument]…);
```

这种风格的函数在编译时丧失了严格的类型安全检查。

4.2 返回值的规则

☆ 【规则 4.2-1】 不要省略返回值的类型；

C 语言中，凡不加类型说明的函数，一律自动按整型处理，这样做不会有什么好处，却容易被误解为 void 类型；

C++语言有很严格的类型安全检查，不允许上述情况发生。由于 C++程序可以调用 C 函数，为了避免混乱，规定任何 C++/ C 函数都必须有类型。如果函数没有返回值，那么应声明为 void 类型

☆ 【规则 4.2-2】 函数名字与返回值类型在语义上不可冲突；

违反这条规则的典型代表是 C 标准库函数 getchar。

例如：

```
char c;
c = getchar();
if (c == EOF)
    …
```

按照 getchar 名字的意思，将变量 c 声明为 char 类型是很自然的事情。但不幸的是 getchar 的确不是 char 类型，而是 int 类型，其原型如下：

```
int getchar(void);
```

由于 c 是 char 类型，取值范围是[-128，127]，如果宏 EOF 的值在 char 的取值范围之外，那么 if 语句将总是失败，这种“危险”人们一般哪里料得到！导致本例错误的责任并不在用户，是函数 getchar 误导了使用者

☆ 【规则 4.2-3】 不要将正常值和错误标志混在一起返回。正常值用输出参数获得，而错误标志用 return 语句返回；

☆ 【建议 4.2-1】 有时候函数原本不需要返回值，但为了增加灵活性如支持链式表达，可以附加返回值；

例如字符串拷贝函数 strcpy 的原型：

```
char *strcpy(char *strDest, const char *strSrc);
```

strcpy 函数将 strSrc 拷贝至输出参数 strDest 中，同时函数的返回值又是 strDest。这样做并非多此一举，可以获得如下灵活性：

```
char str[20];
int  nLength = strlen( strcpy(str, “Hello World”) );
```

☆ 【建议 4.2-2】 如果函数的返回值是一个对象，有些场合用“引用传递”替换“值传递”可以提高效率。而有些场合只能用“值传递”而不能用“引用传递”，否则会出错；

对于建议 4.2-2，如果函数的返回值是一个对象，有些场合用“引用传递”替换“值传递”可以提高效率，而有些场合只能用“值传递”而不能用“引用传递”，否则会出错，例如：

```
class String
{…
    // 赋值函数
    String & operate=(const String &other);
    // 相加函数，如果没有 friend 修饰则只许有一个右侧参数
    friend  String   operate+( const String &s1, const String &s2);
private:
    char *m_data;
};
```

String 的赋值函数 operate = 的实现如下：

```
String & String::operate=(const String &other)
{
    if (this == &other)
        return *this;
    delete m_data;
    m_data = new char[strlen(other.data)+1];
    strcpy(m_data, other.data);
    return *this;   // 返回的是 *this 的引用，无需拷贝过程
}
```

对于赋值函数，应当用“引用传递”的方式返回 String 对象。如果用“值传递”的方式，虽然功能仍然正确，但由于 return 语句要把 *this 拷贝到保存返回值的外部存储单元之中，增加了不必要的开销，降低了赋值函数的效率。例如：

```
String a,b,c;
…
a = b;      // 如果用“值传递”，将产生一次 *this 拷贝
a = b = c;  // 如果用“值传递”，将产生两次 *this 拷贝
```

String 的相加函数 operate + 的实现如下：

```
String  operate+(const String &s1, const String &s2)
{
    String temp;
    delete temp.data;   // temp.data 是仅含‘\0’的字符串
    temp.data = new char[strlen(s1.data) + strlen(s2.data) +1];
    strcpy(temp.data, s1.data);
    strcat(temp.data, s2.data);
    return temp;
}
```

对于相加函数，应当用“值传递”的方式返回 String 对象。如果改用“引用传递”，那么函数返回值是一个指向局部对象 temp 的“引用”。由于 temp 在函数结束时被自动销毁，将导致返回的“引用”无效。例如：

```
c = a + b;
```

此时 a + b 并不返回期望值，c 什么也得不到，流下了隐患。

4.3 函数内部实现的规则

不同功能的函数其内部实现各不相同，看起来似乎无法就“内部实现”达成一致的观点。但根据经验，我们可以在函数体的“入口处”和“出口处”从严把关，从而提高函数的质量。

☆ **【规则 4.3-1】** 在函数体的“入口处”，对参数的有效性进行检查；
很多程序错误是由非法参数引起的，我们应该充分理解并正确使用“断言”（assert）来防止此类错误。详见 4.5 节“使用断言”

☆ **【规则 4.3-2】** 在函数体的“出口处”，对 return 语句的正确性和效率进行检查；

注意事项如下：

(1) return 语句不可返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁，例如：

```
char * Func(void)
{
    char str[] = "hello world"; // str 的内存位于栈上
    …
    return str;      // 将导致错误
}
```

(2) 要搞清楚返回的究竟是“值”、“指针”还是“引用”；

(3) 如果函数返回值是一个对象，要考虑 return 语句的效率，例如：

```
return String(s1 + s2);
```

这是临时对象的语法，表示“创建一个临时对象并返回它”，不要以为它与“先创建一个局部对象 temp 并返回它的结果”是等价的，如

```
String temp(s1 + s2);
return temp;
```

实质不然，上述代码将发生三件事。

首先，temp 对象被创建，同时完成初始化；

然后拷贝构造函数把 temp 拷贝到保存返回值的外部存储单元中；

最后，temp 在函数结束时被销毁（调用析构函数）。

然而“创建一个临时对象并返回它”的过程是不同的，编译器直接把临时对象创建并初始化在外部存储单元中，省去了拷贝和析构的化费，提高了效率。

类似地，我们不要将

```
return int(x + y);  // 创建一个临时变量并返回它
```

写成

```
int temp = x + y;
return temp;
```

由于内部数据类型如 int,float,double 的变量不存在构造函数与析构函数，虽然该“临时变量的语法”不会提高多少效率，但是程序更加简洁易读。

4.4 其它建议

☆ **【建议 4.4-1】** 函数的功能要单一，不要设计多用途的函数；

☆ **【建议 4.4-2】** 函数体的规模要小，尽量控制在 150 行代码之内；

☆ **【建议 4.4-3】** 尽量避免函数带有“记忆”功能。相同的输入应当产生相同的输出带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态”。这样的函数既不易理解又不利于测试和维护。在 C/C++语言中，函数的 static 局部变量是函数的“记忆”存储器。建议尽量少用 static 局部变量，除非必需。

☆ **【建议 4.4-4】** 不仅要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等；

☆ **【建议 4.4-5】** 用于出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。

4.5 使用断言

程序一般分为 Debug 版本和 Release 版本，Debug 版本用于内部调试，Release 版本发行给用户使用。

断言 assert 是仅在 Debug 版本起作用的宏，它用于检查“不应该”发生的情况。示例 4.5 是一个内存复制函数。在运行过程中，如果 assert 的参数为假，那么程序就会中止（一般地还会出现提示对话，说明在什么地方引发了 assert）。

```
void  *memcpy(void *pvTo, const void *pvFrom, size_t size)
{
    assert((pvTo != NULL) && (pvFrom != NULL));     // 使用断言
    byte *pbTo = (byte *) pvTo;     // 防止改变 pvTo 的地址
    byte *pbFrom = (byte *) pvFrom; // 防止改变 pvFrom 的地址
    while(size -- > 0 )
        *pbTo ++ = *pbFrom ++ ;
    return pvTo;
}
```

示例 4.5 复制不重叠的内存块

assert 不是一个仓促拼凑起来的宏。为了不在程序的 Debug 版本和 Release 版本引起差别，assert 不应该产生任何副作用。所以 assert 不是函数，而是宏。程序员可以把 assert 看成一个在任何系统状态下都可以安全使用的无害测试手段。**如果程序在 assert 处终止了，并不是说含有该 assert 的函数有错误，而是调用者出了差错，assert 可以帮助我们找到发生错误的原因。**

☆ **【规则 4.5-1】** 使用断言捕捉不应该发生的非法情况，不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的；

☆ **【规则 4.5-2】** 在函数的入口处，使用断言检查参数的有效性（合法性）；

☆ **【建议 4.5-1】** 在编写函数时，要进行反复的考查，并且自问：“我打算做哪些假定？”一旦确定了的假定，就要使用断言对假定进行检查；

☆ **【建议 4.5-2】** 一般教科书都鼓励程序员们进行防错设计，但要记住这种编程风格可能会隐瞒错误。当进行防错设计时，如果“不可能发生”的事情的确发生了，则要使用断言进行报警。

4.6 引用与指针的比较

引用是 C++中的概念，初学者容易把引用和指针混淆一起。一下程序中，n 是 m 的一个引用（reference），m 是被引用物（referent）。

```
int m;
int &n = m;
```

n 相当于 m 的别名（绰号），对 n 的任何操作就是对 m 的操作。所以 n 既不是 m 的拷贝，也不是指向 m 的指针，其实 n 就是 m 它自己。

引用的一些规则如下：

（1） 引用被创建的同时必须被初始化（指针则可以在任何时候被初始化）；

（2） 不能有 NULL 引用，引用必须与合法的存储单元关联（指针则可以是 NULL）；

（3） 一旦引用被初始化，就不能改变引用的关系（指针则可以随时改变所指的对象）。

以下示例程序中，k 被初始化为 i 的引用。语句 k = j 并不能将 k 修改成为 j 的引用，只是把 k 的值改变成为 6。由于 k 是 i 的引用，所以 i 的值也变成了 6。

```
int i = 5;
int j = 6;
int &k = i;
k = j;  // k 和 i 的值都变成了 6;
```

上面的程序看起来象在玩文字游戏，没有体现出引用的价值。引用的主要功能是传递函数的参数和返回值。C++语言中，函数的参数和返回值的传递方式有三种：值传递、指针传递和引用传递。

以下是“值传递”的示例程序。由于 Func1 函数体内的 x 是外部变量 n 的一份拷贝，改变 x 的值不会影响 n，所以 n 的值仍然是 0。

```
void Func1(int x)
{
    x = x + 10;
}
…
int n = 0;
Func1(n);
cout << “n = ” << n << endl;  // n = 0
```

以下是“指针传递”的示例程序。由于 Func2 函数体内的 x 是指向外部变量 n 的指针，改变该指针的内容将导致 n 的值改变，所以 n 的值成为 10。

```
void Func2(int *x)
{
    (* x) = (* x) + 10;
}
…
int n = 0;
Func2(&n);
cout << “n = ” << n << endl;      // n = 10
```

以下是“引用传递”的示例程序。由于 Func3 函数体内的 x 是外部变量 n 的引用，x 和 n 是同一个东西，改变 x 等于改变 n，所以 n 的值成为 10。

```
void Func3(int &x)
{
    x = x + 10;
}
…
int n = 0;
Func3(n);
    cout << “n = ” << n << endl;      // n = 10
```

对比上述三个示例程序，会发现“引用传递”的性质象“指针传递”，而书写方式象“值传递”。

5 附录

5.1 变量类型定义

类　型	规　则	范　例
bool(BOOL)	用 b 开头	bIsParent
byte(BYTE)	用 by 开头	byFlag
~~short(SHORT)~~	用 n 开头	nFileLen
int(INT)	用 n 开头	nStepCount
long(LONG)	用 l 开头	lSize
char(CHAR)	用 ch 开头	chCount
unsigned short(WORD)	用 w 开头	wLength
unsigned long(DWORD)	用 dw 开头	dwBroad
void(VOID)	用 v 开头	vVariant
用 0 结尾的字符串	用 sz 开头	szFileName
LPCSTR(LPCTSTR)	用 str 开头	strString
HANDLE(HINSTANCE)	用 h 开头	hHandle
struct	用 blk 开头	blkTemplate
BYTE*	用 pb 开头	pbValue
WORD*	用 pw 开头	pwValue
LONG*	用 pl 开头	plValue