C语言书写规范指南

第 1 章文件结构

每个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程序的头文件以“.h”为后缀，C程序的定义文件以“.c”为后缀。

1.1 版权和版本的声明

版权和版本的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，标识符，摘要。

（3）当前版本号，作者/修改者，完成日期。

（4）版本历史信息。

```
/*
*Copyright(c)2001,吉林大学物理学院无线电
*Allrightsreserved.
*
*文件名称：filename.h
*文件标识：
*摘　　要：简要描述本文件的内容
*
*当前版本：1.1
*作　　者：输入作者（或修改者）名字
*完成日期：2007年7月20日
*
*取代版本：1.0
*原作者　：输入原作者（或修改者）名字
*完成日期：2007年5月10日
*/
```

示例1-1版权和版本的声明

1.2 头文件的结构

头文件由三部分内容组成：
（1）头文件开头处的版权和版本声明（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 SCL_SPI.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。
【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）。
【规则1-2-4】 #include 后面使用TAB键控制排版 。

【规则1-2-5】头文件中只存放“声明”而不存放“定义”
【规则1-2-6】全局变量在头文件中声明,在.c文件中定义
.h extern in tvalue; 声明。
.c in tvalue=0x10; 定义。
【规则1-2-7】局部变量在.c中定义 (static) unsigned in tvalue; 定义。

```
//版权和版本声明见示例1-1，此处省略。

#ifndef          SCL_SPI_H          //防止SCL_SPI.h被重复引用
#define          SCL_SPI_H
#include         <p30f6014A.h>      //引用标准库的头文件
…
#include               “SCL_CAN.h”    //引用非标准库的头文件
…
void Function1(…);//全局函数声明
…
extern unsign int value;//全局变量声明


#endif
```

示例1-2C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）定义文件开头处的版权和版本声明（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SCL_SPI.c，定义文件的结构参见示例1-3。

```
//版权和版本声明见示例1-1，此处省略。

#include “SCL_SPI.h” //引用头文件
…
//全局变量定义
unsign int value = 0x10;
//全局函数的实现体
void Function1(…)
{
  …
}
```

示例1-3C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。

第 2 章程序的版式

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

可以把程序的版式比喻为“书法”。好的“书法”可让人对程序一目了然，看得兴致勃勃。差的程序“书法”如螃蟹爬行，让人看得索然无味，更令维护者烦恼有加。请程序员们学习程序的“书法”，弥补大学计算机教育的漏洞，实在很有必要。

2.1 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。所以不要舍不得用空行。

【规则2-1-1】在每个函数定义结束之后都要加空行。参见示例2-1（a）

【规则2-1-2】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2-1（b）

```
    //空行
    void Function1(…)
    {
      …
    }
    //空行
    void Function2(…)
    {
      …
    }
```

示例2-1(a)函数之间的空行

```
//空行
while (condition)
{
  statement1;
  //空行
  if (condition)
  {
      statement2;
  }
  else
  {
      statement3;
```

```
    }
    //空行

    statement4;
}
```

示例2-1(b)函数内部的空行

2.2 代码行

【规则2-2-1】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2-2-2】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。

示例2-2（a）为风格良好的代码行，示例2-2（b）为风格不良的代码行。

```
int     width;//宽度
int     height;//高度
int     depth;//深度
中间使用TAB键控制距离
```

```
x = a+b; //等号左右用空格控制距离

y = c+d;

z = e+f;
```

```
if (width < height)//<左右用空格控制距离
{
        dosomething();//使用TAB键控制距离
}
```

```
for (initialization; condition; update)
{
     dosomething();//使用TAB键控制距离
}
//空行

other();
```

示例2-2(a)风格良好的代码行

```
int     width,height,depth;//宽度高度深度
```

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

```
if(width<height)dosomething();
```

```
for(initialization;condition;update)

     dosomething();

other();
```

示例2-2(b)风格不良的代码行

【建议2-2-1】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。例如

```
int             width = 10;  //定义并初始化width
int             height= 10;  //定义并初始化height
int             depth = 10;  //定义并初始化depth
  使用TAB键控制距离
```

2.3 代码行内的空格

【规则2-3-1】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2-3-2】函数名之后不要留空格，紧跟左括号‘（’，以与关键字区别。
【规则2-3-3】‘（’向后紧跟，‘）’、‘，’、‘;’向前紧跟，紧跟处不留空格。
【规则2-3-4】‘，’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2-3-5】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2-3-6】一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。
【规则2-3-7】象“[]”、“.”、“->”这类操作符前后不加空格。
【建议2-3-1】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);      //良好的风格
void Func1 (int x,int y,int z);     //不良的风格

if (year >= 2000)                   //良好的风格
if(year>=2000)                      //不良的风格
if ((a>=b) && (c<=d))               //良好的风格
if(a>=b&&c<=d)                      //不良的风格

for (i=0; i<10; i++)                //良好的风格
for(i=0;i<10;i++)                   //不良的风格
for(I = 0; I < 10; I++)             //过多的空格

x = a < b ? a : b;                  //良好的风格
x=a<b?a:b;                          //不好的风格

int *x = &y;                        //良好的风格
int*x=&y;                           //不良的风格

array[5] = 0;                       //不要写成array [ 5 ] = 0;
a.Function();                       //不要写成a . Function();
b->Function();                      //不要写成b -> Function();
```

示例2-3代码行内的空格

2.4 对齐

【规则2-4-1】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2-4-2】{}之内的代码块在‘{’右边一个TAB键处左对齐。

示例2-4（a）为风格良好的对齐，示例2-4（b）为风格不良的对齐。

void Function (int x)
{
…//programcode
}
	Void Function(intx){
	…//programcode
	}

if (condition)
{
…//programcode
}
else
{
…//programcode
}
	if(condition){
	…//programcode
	}
	else{
	…//programcode
	}

for (initialization; condition; update)
{
…//programcode
}
	for(initialization;condition;update){
	…//programcode
	}

while (condition)
{
…//programcode
}
	while(condition){
	…//programcode
	}

如果出现嵌套的{}，则使用缩进对齐，如：
{
…
{
…
}
…
}	

示例2-4(a)风格良好的对齐　　示例2-4(b)风格不良的对齐

2.5 长行拆分

【规则2-5-1】代码行最大长度宜控制在70至80个字符以内。代码行不要过长，否则眼睛看不过来，也不便于打印。

【规则2-5-2】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)  使用TAB键控制距离
      &&(very_longer_variable5 <= very_longer_variable16))
{
    dosomething();
}
Virtual CMatrixCMultiplyMatrix (CMatrixleftMatrix,
                                CMatrixrightMatrix); 使用TAB键控制距离
```

```
For (very_longer_initialization;
            very_longer_condition; 使用TAB键控制距离
            very_longer_update)
{
    dosomething();
}
```

示例2-5长行的拆分

2.6 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2-6-1】应当将修饰符*和＆紧靠变量名例如：

```
char        *name;
int         *x,y;    //此处y不会被误解为指针
```

2.7 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2-6。

【规则2-7-1】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2-7-2】如果代码本来就是清楚的，则不必加注释。否则多此一举，令人厌烦。例如

```
i++;    //i加1，多余的注释
```

【规则2-7-3】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2-7-4】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2-7-5】尽量避免在注释中使用缩写，特别是不常用缩写。

【规则2-7-6】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2-7-8】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。

```
/*
*函数介绍：
*输入参数：
*输出参数：
*返回值    ：
*/
Void Function(floatx, floaty, floatz)
{
    …
    if(…)
    {
        …
        while(…)
        {
            …
        }//endofwhile
        …
    }//endofif
}
```

示例2-6程序的注释

第 3 章命名规则

比较著名的命名规则当推Microsoft公司的“匈牙利”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。

据考察，没有一种命名规则可以让所有的程序员赞同，程序设计教科书一般都不指定命名规则。命名规则对软件产品而言并不是“成败悠关”的事，我们不要化太多精力试图发明世界上最好的命名规则，而应当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。绝对杜绝汉语拼音命名法

3.1 共性规则

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3.2节。

【规则3-1-1】标识符应当直观且可以拼读，可望文知意，不必进行“解码”。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会太复杂，用词应当准确。例如不要把CurrentValue写成NowValue。

【规则3-1-2】标识符的长度应当符合“min-length&&max-information”原则。

几十年前老ANSIC规定名字不准超过6个字符，现今的C不再有此限制。一般来说，长名字能更好地表达含义，所以函数名、变量名、类名长达十几个字符不足为怪。那么名字是否越长约好？不见得!例如变量名maxval就比maxValueUntilOverflow好用。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。

【规则3-1-3】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。

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用。

【规则3-1-4】程序中不要出现仅靠大小写区分的相似的标识符。例如：

```
int x,  X;      //变量x与X容易混淆
voidfoo(intx); //函数foo与FOO容易混淆

voidFOO(floatx);
```

【规则3-1-5】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3-1-6】变量的名字应当使用“名词”或者“形容词＋名词”。例如：

```
float       value;
float       oldValue;
float       newValue;
```

【规则3-1-7】全局函数的名字应当使用“动词”或者“动词＋名词”（动宾词组）。类的成员函数应当只使用“动词”，被省略掉的名词就是对象本身。

例如：

```
DrawBox();      //全局函数
box->Draw();    //类的成员函数
```

【规则3-1-8】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。例如：

```
int         minValue;

int         maxValue;
```

```
int         SetValue(…);
int         GetValue(…);
```

【建议3-1-1】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。这是为了防止程序员偷懒，不肯为命名动脑筋而导致产生无意义的名字（因为用数字编号最省事）。

3.2 简单的 Windows 应用程序命名规则

作者对“匈牙利”命名规则做了合理的简化，下述的命名规则简单易用，比较适合于Windows应用软件的开发。

【规则3-2-1】类名和函数名用大写字母开头的单词组合而成。
例如：

```
class                       Node;       //类名
class                       LeafNode;   //类名
void  Draw(void);           //函数名
void  SetValue(int value);//函数名
```

【规则3-2-2】变量和参数用小写字母开头的单词组合而成。例如：

```
BOOL        flag;
int         drawMode;
```

【规则3-2-3】常量全用大写的字母，用下划线分割单词。例如：

```
const int   MAX = 100;
const int   MAX_LENGTH = 100;
```

【规则3-2-4】静态变量加前缀s_（表示static）。例如：

```
void Init(…)
{
    static int     s_initValue;//静态变量
    …
}
```

【规则3-2-5】如果不得已需要全局变量，则使全局变量加前缀g_（表示global）。例如：

```
int                     g_howManyPeople;      //全局变量
int                     g_howMuchMoney;//全局变量
```

【规则3-2-6】类的数据成员加前缀m_（表示member），这样可以避免数据成员与成员函数

的参数同名。
例如：

```
Void Object::SetValue(int width, int height)
{
    m_width = width;
    m_height = height;
}
```

【规则3-2-7】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。

第 4 章表达式和基本语句

表达式和语句都属于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。

4.1 运算符的优先级

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))
```

4.2 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)
```

4.3if 语句

if语句是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。

4.3.1 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VisualC++将TRUE定义为1，而VisualBasic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)
```

4.3.2 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)
```

4.3.3 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）。

4.3.4 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)
```

4.3.5 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)
    returnx;
returny;
```

改写为

```
if(condition)
{
    returnx;
}
else
{
    returny;
}
```

或者改写成更加简练的

```
return(condition?x:y);
```

4.4 循环语句的效率

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++)
    {
        fum = sum+a[row][col];
    }
}
```

示例4-4(a)低效率：长循环在最外层

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

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();
   }
}
```

表4-4(c)效率低但程序简洁

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

表4-4(d)效率高但程序不简洁

4.5for 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(intx=0;x<N;x++)
{
  …
}
```

示例4-5(a)循环变量属于半开半闭区间

```
for(intx=0;x<=N-1;x++)
{
  …
```

```
}
```
示例4-5(b)循环变量属于闭区间

4.6switch 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。

4.7goto 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

```
{…
    {…
        {…
            gotoerror;
        }
    }
}
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。

第 5 章常量

常量是一种标识符，它的值在运行期间恒定不变。C语言用#define来定义常量（称为宏常量）。C++语言除了#define外还可以用const来定义常量（称为const常量）。

5.1 为什么需要常量

如果不使用常量，直接在程序中填写数字或字符串，将会有什么麻烦？

（1）程序的可读性（可理解性）变差。程序员自己会忘记那些数字或字符串是什么意思，用户则更加不知它们从何处来、表示什么。

（2）在程序的很多地方输入同样的数字或字符串，难保不发生书写错误。

（3）如果要修改数字或字符串，则会在很多地方改动，既麻烦又容易出错。

【规则5-1-1】尽量使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串。

例如：

```
#define         MAX     100     /*  C语言的宏常量    */
constint        MAX=100;        // C++语言的const常量
constfloat  PI=3.14159;//  C++语言的const常量
```

5.2const 与#define 的比较

C++语言可以用const来定义常量，也可以用#define来定义常量。但是前者比后者有更多的优点：

（1）const常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误（边际效应）。

（2）有些集成化的调试工具可以对const常量进行调试，但是不能对宏常量进行调试。

【规则5-2-1】在C++程序中只使用const常量而不使用宏常量，即const常量完全取代宏常量。

5.3 常量定义规则

【规则5-3-1】需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。为便于管理，可以把不同模块的常量集中存放在一个公共的头文件中。

【规则5-3-2】如果某一常量与其它常量密切相关，应在定义中包含这种关系，而不

应给出一些孤立的值。

例如：

```
const  float  RADIUS=100;
const  float  DIAMETER=RADIUS*2;
```

第 6 章函数设计

函数是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（passbyvalue）和指针传递（passbypointer）。C++语言中多了引用传递（passbyreference）。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”。

6.1参数的规则

【规则6-1-1】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 width, int height);//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6-1-2】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,“HelloWorld”);//参数顺序颠倒
```

【规则6-1-3】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6-1-4】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6-1-1】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6-1-2】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。

6.2 返回值的规则

【规则6-2-1】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6-2-2】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：

```
charc;
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6-2-3】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。

回顾上例，C标准库函数的设计者为什么要将getchar声明为令人迷糊的int类型呢？

他会那么傻吗？

在正常情况下，getchar的确返回单个字符。但如果getchar碰到文件结束标志或发生读错误，它必须返回一个标志EOF。为了区别于正常的字符，只好将EOF定义为负数（通常为负1）。因此函数getchar就成了int类型。

我们在实际工作中，经常会碰到上述令人为难的问题。为了避免出现误解，我们应该将正常值和错误标志分开。即：正常值用输出参数获得，而错误标志用return语句返回。

函数getchar可以改写成BOOL GetChar(char *c);

虽然gechar比GetChar灵活，例如putchar(getchar());但是如果getchar用错了，它的灵活性又有什么用呢？

【建议6-2-1】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：

```
charstr[20];
int length=strlen(strcpy(str,“HelloWorld”));
```

【建议6-2-2】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;
    deletem_data;
    m_data=newchar[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：

```
Stringa,b,c;
...
a=b;        //如果用“值传递”，将产生一次 *thi 拷贝
a=b=c;      //如果用“值传递”，将产生两次 *thi 拷贝
```

String的相加函数operate+的实现如下：

```
String operate+(const String &s1,const String &s2)
{
    String temp;
    deletetemp.data;//temp.data是仅含‘\0’的字符串
    temp.data=newchar[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。

6.3 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6-3-1】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6.5节“使用断言”。

【规则6-3-2】在函数体的“出口处”，对return语句的正确性和效率进行检查。

如果函数有返回值，那么函数的“出口处”是return语句。我们不要轻视return语句。如果return语句写得不好，函数要么出错，要么效率低下。

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[]=“helloworld”;//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。

6.4 其它建议

【建议6-4-1】函数的功能要单一，不要设计多用途的函数。

【建议6-4-2】函数体的规模要小，尽量控制在50行代码之内。

【建议6-4-3】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6-4-4】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6-4-5】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。

6.5 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6-5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++;
    returnpvTo;
}
```

示例6-5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。

很少有比跟踪到程序的断言，却不知道该断言的作用更让人沮丧的事了。你化了很多时间，不是为了排除错误，而只是为了弄清楚这个错误到底是什么。有的时候，程序员偶尔还会设计出有错误的断言。所以如果搞不清楚断言检查的是什么，就很难判断错误是出现在程序中，还是出现在断言中。幸运的是这个问题很好解决，只要加上清晰的注释即可。这本是显而易见的事情，可是很少有程序员这样做。这好比一个人在森林里，看到树上钉着一块“危险”的大牌子。但危险到底是什么？树要倒？有废井？有野兽？除非告诉人们“危险”是什么，否则这个警告牌难以起到积极有效的作用。难以理解的断言常常被程序员忽略，甚至被删除。

【规则6-5-1】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6-5-2】在函数的入口处，使用断言检查参数的有效性（合法性）。

【建议6-5-1】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6-5-2】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。

6.6 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。例如有人名叫王小毛，他的绰号是“三毛”。说“三毛”怎么怎么的，其实就是对王小毛说三道四。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;

}

…

intn=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”。实际上“引用”可以做的任何事情“指针”也都能够做，为什么还要“引用”这东西？

答案是“用适当的工具做恰如其分的工作”。

指针能够毫无约束地操作内存中的如何东西，尽管指针功能强大，但是非常危险。就象一把刀，它可以用来砍树、裁纸、修指甲、理发等等，谁敢这样用？

如果的确只需要借用一下某个对象的“别名”，那么就用“引用”，而不要用“指针”，以免发生意外。比如说，某人需要一份证明，本来在文件上盖上公章的印子就行了，如果把取公章的钥匙交给他，那么他就获得了不该有的权利。