密　　级：普通
文件编号：NO.3
文件类别：测试管理体系文件
发 放 号：1003

C/C++ 编程规范

版本：1.0

目　录

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）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 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) 2003,北京梅梅出品有限公司
* All rights reserved.
*
* 文件名称：filename.h
* 文件标识：见配置管理计划书
* 摘要：简要描述本文件的内容
*/
#ifndef GRAPHICS_H      // 防止graphics.h 被重复引用
#define GRAPHICS_H
下面其它的声明代码
……

下面是原作者、版本、完成、日期和当前版本的信息
/* 当前版本：1.1.2
* 作者：输入作者（或修改者）名字
* 完成日期：2003年5月20日
*
* 取代版本：1.1.1
* 原作者：输入原作者（或修改者）名字
* 完成日期：2003年4月10日
*/
```

示例 1-1 版权和版本的声明

版本标识 采用 <主版本号>.<次版本号>.<修订号> 来命名自己产品的编号。Linux 核心还有一个约定，就是如果次版本号是偶数（如 0、2、4 等），代表正式版本，如果次版本号是奇数(如 1、3、5 等)，代表的是开发过程中的测试版本。修订号则相当于 Build 号，用来标识一些小的改动。

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graphics.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）。

【建议1-2-1】头文件中只存放“ 声明 ”而不存放“ 定义 ”

【建议1-2-2】不提倡使用全局变量，尽量不要在头文件中出现象extern int value 这类声明。

```
// 版权和版本声明见示例1-1，此处省略。
#ifndef GRAPHICS_H      // 防止graphics.h 被重复引用
#define GRAPHICS_H
#include <math.h>       // 引用标准库的头文件
…
#include “myheader.h”   // 引用非标准库的头文件
…
void Function1(…);      // 全局函数声明
…
class Box    // 类结构声明
{ …
};
#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graphics.cpp，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 “ graphics.h”     // 引用头文件
…
// 全局函数的实现体
void Function1(…)
{
    ...
}
// 类成员函数的实现体
void Box::Draw(...)
{
  ...
}
```

示例1-3 C++/C 定义文件的结构

1.4 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. 程序版式

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，所以不要舍不得用空行。

【规则2-1-1】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(.)
{
  ...
}
// 空行
void Function3(.)
{
  ...
}
```

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。

【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
```

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

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

if (width < height)
{
  dosomething();
}
	if (width < height) dosomething();

for (initialization; condition; update)
{
  dosomething();
}
// 空行
other();
	for (initialization; condition; update)
	  dosomething();
	  other();

示例2-2(a) 风格良好的代码行　　示例2-2(b) 风格不良的代码行

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();

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‘{’右边数格处左对齐。

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)
     && (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2-5 长行的拆分

2.6 修饰符的位置

【规则2-6-1】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。

【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。

【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。

【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。

2.8 类的版式

类可以将数据和函数封装在一起，其中函数表示了类的行为（或称服务）。类提供关键字public、protected 和private，分别用于声明哪些数据和函数是公有的、受保护的或者是私有的。这样可以达到信息隐藏的目的，即让类仅仅公开必须要让外界知道的内容，而隐藏其它一切内容。

类的版式主要有两种方式：

（1）将private 类型的数据写在前面，而将public 类型的函数写在后面，如示例8-3（a）。采用这种版式的程序员主张类的设计“ 以数据为中心 ”，重点关注类的内部结构。

（2）将public 类型的函数写在前面，而将private 类型的数据写在后面，如示例8.3（b）采用这种版式的程序员主张类的设计“ 以行为为中心 ”，重点关注的是类应该提供什么样的接口（或服务）。

【建议2-8-1】建议采用“ 以行为为中心 ”的书写方式，即首先考虑类应该提供什么样的函数。这样做不仅让自己在设计类时思路清晰，而且方便别人阅读。

```
class A
{
  private:
          int  i, j;
          float x, y;
          …
  public:
          void Func1(void);
          void Func2(void);
      …
}
```

示例8.3(a) 以数据为中心版式

```
class A
{
  public:
        void Func1(void);
        void Func2(void);
        …
  private:
        int i, j;
        float x, y;
        …
}
```

示例8.3(b) 以行为为中心的版式

3 命名规则

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。

命名两个基本原则：1．含义清晰，不易混淆；2．不和其它模块、系统API的命名空间相冲突即可。

【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。

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。但名字并不是越长约好！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
void foo(int x);     // 函数foo 与FOO 容易混淆
void FOO(float x);
```

【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;
```

【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。

3.2 Windows 应用程序命名规则

【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】如果需要定义全局变量，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。

3.3 Linux 应用程序函数命名规则

函数命名应遵循下面两个原则：

1）属于某一模块的函数，加上前缀，前缀为模块缩写；

2）函数名应该表明函数意义，格式为"前缀_名词_动词"；

4．表达式和基本语句

4.1 运算符的优先级

【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。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)

4.3 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。

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)
```

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)
```

4.4 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];
    }
}
```

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++)
    {
        sum = 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。并且由于前者总要进行逻辑判断，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

4.5 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++)
{
   ...
}
```

示例4-5(a) 循环变量属于半开半闭区间

```
for (int x=0; x<=N-1; x++)
{
   ...
}
```

示例4-5(b) 循环变量属于闭区间

4.6 switch 语句

【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;

4.7 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。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，而不是完全禁用。

5. 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 const 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5-1-1】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*/
const int MAX = 100;  // C++ 语言的const 常量
const float PI = 3.14159; // C++ 语言的const 常量
```

【规则5-1-2】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。

5.2 常量定义规则

【规则5-2-1】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5-2-2】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;
```

6. 函数设计

一个函数的注释信息如下例：

```
/*********************************************************************************
*                    Function:      calculate   The area of rectangle                *
*                    parameter:     the Length and Width of rectangle                *
*                    outout:        the area of   rectangle                          *
*********************************************************************************/
int     GetValue(int   iLength,int   iWidth)
{
   ……..
   return   iArea;
}
/*

Error:
        1 描述在单元测试中出现的错误
        2…….
*/
```

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。另外，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。

【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。

【规则 6-1-5】参数缺省值只能出现在函数的声明中，而不能出现在定义体中。

【规则 6-1-6】如果函数有多个参数，参数只能从后向前挨个儿缺省，否则将导致函数调用语句怪模怪样。

【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。

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。

【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。

【规则 6-2-4】给以“ 指针传递 ”方式的函数返回值加 const 修饰，那么函数返回值（即指针）的内容不能被修改，该返回值只能被赋给加 const 修饰的同类型指针。

【规则 6-2-5】函数返回值采用“ 值传递方式 ”，由于函数会把返回值复制到外部临时的存储单元中，加 const 修饰没有任何价值。

【规则 6-2-6】函数返回值采用“ 引用传递 ”的场合并不多，这种方式一般只出现在类的赋值函数中，目的是为了实现链式表达。

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。

【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。

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。

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。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）。

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，容易把引用和指针混淆。

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）。

7 重载和内联

7.1 普通函数重载

【规则 7-1-1】重载函数中的参数不同（包括类型、顺序不同），才是重载函数，而仅仅返回值不同则不行。

【规则 7-1-2】当心隐式类型转换导致重载函数产生二义性，数字本身没有类型，将数字当作参数时将自动进行类型转换（称为隐式类型转换）。

成员函数的重载、覆盖与隐藏

【规则 7-2-1】成员函数的重载、覆盖（override）与隐藏很容易混淆，注意区分。

【规则 7-2-2】注意如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。

【规则 7-2-3】注意如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

7.2 内联函数

【规则 7-3-1】尽量用内联取代宏代码，提高函数的执行效率（速度）。

【规则 7-3-2】关键字 inline 必须与函数定义体放在一起才能使函数成为内联，仅将 inline 放在函数声明前面不起任何作用。

【规则 7-3-3】如果函数体内的代码比较长或函数体内出现循环，则不宜使用内联。

8. 内存管理

内存分配方式有三种：

（1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static 变量。

（2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3） 从堆上分配，亦称动态内存分配。程序在运行的时候用malloc 或new 申请任意多少的内存，程序员自己负责在何时用free 或delete 释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

【规则8-1-1】用malloc 或new 申请内存之后，应该立即检查指针值是否为NULL。防止使用指针值为NULL的内存。

【规则8-1-2】不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。

【规则8-1-3】避免数组或指针的下标越界，特别要当心发生“多1”或者“少1”操作。

【规则8-1-4】动态内存的申请与释放必须配对，防止内存泄漏。

【规则8-1-5】用free 或delete 释放了内存之后，立即将指针设置为NULL，防止产生“野指针”。

9 类的构造函数、析构函数、成员函数与赋值函数

9.1 类的构造函数

【规则 9-1-1】“缺省的拷贝构造函数”和“缺省的赋值函数”均采用“位拷贝”而非“值拷贝”的方式来实现，若类中含有指针变量，不能采用缺省的方式。

【规则 9-1-2】如果类存在继承关系，派生类必须在其初始化表里调用基类的构造函数。

【规则 9-1-3】类的 const 常量只能在初始化表里被初始化，因为它不能在函数体内用赋值的方式来初始化。

【规则 9-1-4】非内部数据类型的成员对象采用初始化表的方式初始化较好。

【规则 9-1-5】拷贝构造函数和赋值函数非常容易混淆，常导致错写、错用。拷贝构造函数是在对象被创建时调用的，而赋值函数只能被已经存在了的对象调用。

9.2 成员函数

【规则 9-2-1】任何不会修改数据成员的函数都应该声明为 const 类型。

10. 类的继承和组合

对于 C++程序而言，设计孤立的类是比较容易的，难的是正确设计基类及其派生类。

【规则 10-1-1】如果类 A 和类 B 毫不相关，不可以为了使 B 的功能更多些而让 B 继承 A 的功能和属性。

【规则 10-1-2】若在逻辑上 B 是 A 的“一种情况”，则允许 B 继承 A 的功能和属性。

【规则 10-1-3】若在逻辑上 A 是 B 的“一部分”(a part of)，则不允许 B 从 A 派生，而是要用 A 和其它东西组合出 B。

11. 其他规范及建议

11.1 提高程序的效率

程序的时间效率是指运行速度，空间效率是指程序占用内存或者外存的状况。全局效率是指站在整

个系统的角度上考虑的效率，局部效率是指站在模块或函数角度上考虑的效率。

【规则11-1-1】不要一味地追求程序的效率，应当在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率。

【规则11-1-2】以提高程序的全局效率为主，提高局部效率为辅。

【规则11-1-3】在优化程序的效率时，应当先找出限制效率的“瓶颈”，不要在无关紧要之处优化。

【规则11-1-4】先优化数据结构和算法，再优化执行代码。

【规则11-1-5】有时候时间效率和空间效率可能对立，此时应当分析那个更重要，作出适当的折衷。例如多花费一些内存来提高性能。

【规则11-1-6】不要追求紧凑的代码，因为紧凑的代码并不能产生高效的机器码。

11.2 一些有益的建议

【建议11-2-1】当心那些视觉上不易分辨的操作符发生书写错误。我们经常会把“==”误写成“=”，象“||”、“&&”、“<=”、“>=”这类符号也很容易发生“丢失”失误。然而编译器却不一定能自动指出这类错误。

【建议11-2-2】变量（指针、数组）被创建之后应当及时把它们初始化，以防止把未被初始化的变量当成右值使用。

【建议11-2-3】当心变量的初值、缺省值错误，或者精度不够。

【建议11-2-4】当心数据类型转换发生错误。尽量使用显式的数据类型转换，避免让编译器轻悄悄地进行隐式的数据类型转换。

【建议11-2-5】当心变量发生上溢或下溢，数组的下标越界。

【建议11-2-6】当心忘记编写错误处理程序，当心错误处理程序本身有误。

【建议11-2-7】当心文件I/O 有错误。

【建议11-2-8】避免编写技巧性很高代码。

【建议11-2-9】不要设计面面俱到、非常灵活的数据结构。

【建议11-2-10】如果原有的代码质量比较好，尽量复用它。但是不要修补很差劲的代码，应当重新编写。

【建议11-2-11】尽量使用标准库函数，不要“发明”已经存在的库函数。

【建议11-2-12】尽量不要使用与具体硬件或软件环境关系密切的变量。

【建议11-2-13】把编译器的选择项设置为最严格状态。

【建议11-2-14】如果可能的话，使用LogiScope 等工具进行代码审查。