软件部 C/C++编程规范

编制：	日期： 2006-3-6
审核：	日期：
会签：	日期：
批准：	日期：

文档历史发放及记录

序号	变更（+/-）说明	作者	版本号	日期	批准
1	初稿		V1.0	2006-3-6	

目　　录

第1章 概述

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 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：

- 版权信息
- 文件名称，标识符，摘要
- 当前版本号，作者/修改者，完成日期
- 版本历史信息。

```
/*********************************************************************
/*********************************************************************
* Copyright (c) 2004,ShenZhen SDMC Microelectronic.
* All rights reserved.
*
* File Name：filename.h
* File Identify：reference to Configuration Manager
* Summary： brief description of the file
*
* Current Version：1.1
* Author(s)：lists all the authors who participated in composion and modification, the reason of modification and date.
* Example:
/----------------------------------------------------------------------/
revision        author          reason          date
1.1             He XingChao     Addion to JVM   August 17,2004
```

```
1.0                Yuan Song          Original          August 11,2003
/--------------------------------------------------------------------------/
```

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 graphics.h，头文件的结构参见示例 1-2。

- 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。
- 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)。
- 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】头文件中只存放“声明”而不存放“定义”

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。

【建议 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 FunctionFeed(…);  // 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 定义源文件的结构

定义源文件有三部分内容：

（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 FunctionOne(…)
{
    …
}

// 类成员函数的实现体
void Box::m_Draw(…)
{
```

```
        …
    }
```

示例 1-3 C++/C 定义文件的结构

1.4 头文件的作用

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。

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 源程序文件的划分

（1） 按分层划分原则

在同一个文件中最好只包含处在同一层次的函数集。例如，应将一个模块中对硬件直接进行访问的底层函数和协议层的函数放在两个源文件里面。

（2） 按功能划分原则

将一些功能类似的函数归到同一个文件中。比如：将对器件进行读写的一系列函数应放在同一个文件中。

（3） 按规模划分原则

保证每个源程序的大小在一定的规模。一般来说，一个 C 语言的源程序的大小不应超过 2000 行。

1.6 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 代码格式

2.1 缩进

（1） 程序块要采用缩进风格编写，缩进的空格数为 4 个空格。

函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case 语句下的情况处理语句也要遵从语句缩进要求。

由开发工具自动生成的代码可以有不一致。

（2） 对齐只使用空格键，不使用 TAB 键

以免用不同的编辑器阅读程序时，因 TAB 键所设置的空格数目不同而造成程序布局不整齐。如果编辑器支持，可在编辑代码时将 TAB 键定义成“自动替换为四个空格”方式，以加快录入速度。

（3） 程序块的分界符（如 C/C++语言的大括号‘{’和‘}’）应各独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、类的定义、结构的定义、枚举的定义以及 if、for、do、while、switch、case 语句中的程序都要采用如上的缩进方式。

示例：如下例子不符合规范。

```
for (...) {
    ... // program code
}

if (...)
    {
    ... // program code
    }


void exampleFun( void )
    {
```

```
    ... // program code
    }
```

应如下书写。

```
for (...)
{
    ... // program code
}

if (...)
{
    ... // program code
}

void ExampleFun( void )
{
    ... // program code
}
```

2.2 相对独立的程序块之间、变量说明之后必须加空行

（1） 函数间要用空行分开

函数之间至少有两个连续空行，但函数内部不应该有连续空行。

（2） 局部变量声明和代码之间用空行分开

（3） 用空行将代码按逻辑片段划分

例如

如下示例不符合规范。

```
if (!valid_ni(ni))
{
    ... // program code
}
repssn_ind = SsnData[index].m_RepssnIndex;
repssn_ni  = SsnData[index].m_Ni;
```

应如下书写：

```
if (!valid_ni(ni))
{
    ... // program code
}

repssn_ind = SsnData[index].m_RepssnIndex;
repssn_ni  = SsnData[index].m_Ni;
```

2.3 空格

（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。

（2） 函数名之后不要留空格，紧跟左括号‘(’，以与关键字区别。

（3） ‘(’向后紧跟，‘)’、‘,’、‘;’向前紧跟，紧跟处不留空格。

（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)。

（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。

（6） 一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。

（7） 象“[]”、“.”、“->”这类操作符前后不加空格。

【建议】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 TestFunc(int nFir, int nSec, int nThr); // 良好的风格 *void TestFunc (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*
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 *pSdt = &sdtSearch; // 良好的风格 *int * x = & y; // 不良的风格*
aFlash[5] = 0; // *不要写成 aFlash [5] = 0;* EPG_Upd.m_Function(); // *不要写成* EPG_Upd . *m_Function();* EPG_Upd ->m_Function(); // *不要写成* EPG_Upd -> *m_Function();*

2.4 不必空格之处

由于留空格所产生的清晰性是相对的，所以，在已经非常清晰的语句中就没有必要再留空格，如：

（1） 如果语句已足够清晰，则括号内侧（即左括号后面和右括号前面）不需要加空格；

（2） 多重括号间不必加空格，因为在 C/C++语言中括号已经是最清晰的标志了；

（3） 在长语句中如果需要加的空格非常多，则应该保持整体清晰，而在局部不加空格。

2.5 语句分行

（1） 一行程序不要写得过长，以小于 80 字符为宜

（2） 较长的语句要分成多行书写

较长的语句（>80 字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，

操作符放在新行之首，划分出的新行采用 4 个空格进行缩进，使排版整齐，语句可读。

例如

```
PermCountMsg.CHead.m_len = NO7_TO_STAT_PERM_COUNT_LEN
    + STAT_SIZE_PER_FRAM * sizeof(_UL);


ActTaskTtable[frame_id * STAT_TASK_CHECK_NUMBER
    + index].m_Occupied = CStat_Poi[index].m_Occupied;


ActTaskTable[taskno].m_DurationTrueOrFalse
    = SysGetSccpStatisticsState(nStat_item);


bReportOrNotFlag = ((nTaskNo < MAX_ACT_TASK_NUMBER)
    && (n7statStatItemValid (statItem))
    && (ActTaskTable[nTaskNo].m_ResultData != 0));
```

（3） 循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分

循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，长表达式要在低优先级操作符处划分新行，操作符放在新行之首并采用 4 个空格进行缩进。

例如

```
if ((nTaskNo < MAX_ACT_TASK_NUMBER)
    && (n7StatStaItemValid (nStatItem)))
{
    ... // program code
}


for (i = 0, j = 0; (i < BufferKeyWord[wordIndex].m_WordLength)
     && (j < NewKeyWord.m_WordLength); i++, j++)
{
    ... // program code
}
```

```
for (i = 0, j = 0;
   (i < nFirstWordLength) && (j < nSecondWordLength);
    i++, j++)
{
    ... // program code
}
```

（4） 若函数中的参数较长，则要进行适当的划分并采用 4 个空格进行缩进

例如

```
n7StatStrCompare((UINT8 *) & m_StatObject,
   (UINT8 *) & (ActTaskTable[nTaskNo].m_StatObject),
   sizeof (_STAT_OBJECT));
```

（5） 一行只写一条短语句

不允许把多个短语句写在一行中，即一行只能写一条语句。

例如

如下示例不符合规范。

```
rect.length = 0;  rect.width = 0;
```

应如下书写

```
Rect.m_Length = 0;
Rect.m_Width  = 0;
```

（6） if、for、do、while、case、switch、default 等语句应自占一行

2.6 括号

（1） 避免使用默认优先级，应用括号明确表达式的操作顺序

注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

例如

好的书写方式：

```
nWord = (nHigh << 8) | nLow                          (1)
if ((nValueA | nValueB) && (nValueA & nValueC))      (2)
```

```
if ((nValueA | nValueB) < (nValueC & nValueD))        (3)
```

如果以上语句中的表达式书写为：

```
nHigh << 8 | nLow                                    (1')
nValueA | nValueB && nValueA & nValueC               (2')
nValueA | nValueB < nValueC & nValueD                (3')
```

对于(1')和(2')：

```
nHigh << 8 | nLow = (nHigh << 8) | nLow,
nValueA | nValueB && nValueA & nValueC =
   ((nValueA | nValueB) && (nValueA & nValueC));
```

(1')(2')不会出错，但语句不易理解。而对于(3')：

```
nValueA | nValueB < nValueC & nValueD =
nValueA | （nValueB < nValueC） & nValueD;
```

(3')造成了判断条件出错。

（2） if、for、do、while 等语句的执行语句部分无论多少都要加花括号{}

花括号中没有或只有一条语句时也不省略花括号。采用这种方式有如下益处：

（a） 所有情况都是统一格式；

（b） 阅读更简单直观；

（c） 当增添或删除语句时不需要增减花括号，简单、不易出错；

（d） 不会把不同层次的 if 和 else 错误地配在一起。

例如 1

如下示例不符合规范。

```
if (pUserCR == NULL) return;
```

应如下书写：

```
if (pUserCR == NULL)
{
   return;
}
```

例如 2

如下方式表达空循环不够清楚：

```
for (i=0; i<nTIMEOUT; i++);
```

应如下书写：

```
for (i=0; i<nTIMEOUT; i++)
{
    ...//program code
}
```

（3） 一般情况下，花括号{}要单独占一行，且与其控制语句左对齐

在函数体的分界、类的定义、结构的定义、枚举的定义以及及循环、判断等语句（如 if、for、do、while、switch、case 语句）中的{}一般都要采用如上的方式。

例如

如下示例不符合规范。

```
for (...) {
    ... // program code
}

if (...)
    {
    ... // program code
    }

void ExampleFun(void)
    {
    ... // program code
    }
```

应如下书写。

```
for (...)
{
```

```
        ... // program code
    }

    if (...)
    {
        ... // program code
    }

    void ExampleFun(void)
    {
        ... // program code
    }
```

（4） 花括号中的右括号“}”不独占一行的情况

在下列情况下，花括号中的右括号“}”不独占一行：

（a） 结构和联合结尾紧跟变量定义，要和“}”放在一行；

（b） 任何“}”后紧跟分号要放在一行。

例如

```
// 结构结尾紧跟变量定义的情况
typedef struct
{
  //…
}Message;          // 放在一行

// 花括号后紧跟分号的情况
class MyClassTar
{
  //…
};                 // 放在一行
```

2.7 相邻

源程序中关系较为紧密的代码应尽可能相邻，以便于程序阅读和查找。

例如

以下代码布局不太合理。

```
Rect.m_Length = 10;
cCharPoi = str;
Rect.m_Width = 5;
```

若按如下形式书写，会更清晰一些。

```
Rect.m_nLength = 10;
Rect.m_nWidth = 5; // 矩形的长与宽关系较密切，放在一起。
cCharPoi = str;
```

2.8 修饰符的位置

应当将修饰符 * 和 & 紧靠变量名

例如：

```
char  *pcName;
int   *pnPat
int   nConst;
```

2.9 类的版式

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（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（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）,推荐使用这种类的版式。

class Tmplate *{* *private:*	class Tmplate { public:

```
        int       nLength,nWidth;
        float    crdntX, crdntY;
…
    public:
        void FuncOne(void);
        void FuncSec(void);
…
}
```

```
        void FuncOne(void);
        void FuncSec(void);
        …
    private:
        int    nLength;
        int    nWidth;
        float    crdntX;
        float    crdntY;
        …
}
```

示例(a) 以数据为中心版式　　示例 (b) 以行为为中心的版式

2.10 类成员函数的排列顺序

先是构造函数和析构函数，然后是操作符函数和其他函数，并按功能归类排列。

第3章 注释

基本要求一般情况下，源程序有效注释量必须在 20％以上。

注释的原则是有助于对程序的阅读理解，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

3.1 应加注释之处

（1） 说明性文件（如头文件.h 文件、.inc 文件、.def 文件、编译说明文件.cfg 等）头部应进行注释，注释必须列出：版权和版本的声明、修改日志等，头文件的注释中还应有函数功能简要说明。

示例：下面这段头文件的头注释比较标准，当然，并不局限于此格式，但下述信息建议要包含在内。

```
/*************************************************************************
* Copyright (c) 2004,ShenZhen SDMC Microelectronic.
* All rights reserved.
*
* File Name：filename.h
* File Identify：reference to Configuration Manager
* Summary： brief description of the file
*
* Current Version：1.1
 * Author(s)：lists all the authors who participated in composion and modification, the reason of modification and date.
* Example:
/----------------------------------------------------------------------/
revision author reason date
1.1 He XingChao Addion to JVM August 17,2004
1.0 Yuan Song Original August 11,2003
/----------------------------------------------------------------------/
```

（2） 源文件头部应进行注释，列出：版权和版本的声明、修改日志等。

例如：下面这段源文件的头注释比较标准，当然，并不局限于此格式，但下述信息建议要包含在内。

/***

* Copyright (c) 2004,ShenZhen SDMC Microelectronic

* All rights reserved.

*

* File Name：filename.h

* File Identify：reference to Configuration Manager

* Summary： brief description of the file

*

* Current Version：1.1

* Author(s)：lists all the authors who participated in composion and modification, the reason of modification and date.

* Example:

/---/

revision	author	reason	date
1.1	He XingChao	Addion to JVM	August 17,2004
1.0	Yuan Song	Original	August 11,2003

/---/

（3） 函数头部应进行注释，列出：函数的目的/功能、输入参数、输出参数、返回值、调用关系（函数、表）等。

例如：下面这段函数的注释比较标准，当然，并不局限于此格式，但下述信息建议要包含在内。

```
/*************************************************

  Function:         function name

  Description:      describes function  about the
                    functions,performance

  Input:            discribes every parameter, including
                    meaning,parameter domain and relation with
```

```
                    other parameter

    Output:         discribes output

    Return:         discribes returning value

    Others:         other discriptions

***********************************************/
```

（4） 对于所有有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。

例如：

```
/* active statistic task number */

#define MAX_ACT_TASK_NUMBER （1000）

#define MAX_ACT_TASK_NUMBER （1000） /* active statistic task
    number */
```

（5） 数据结构声明(包括数组、结构、类、枚举等)，如果其命名不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释放在此域的右方。

例如：可按如下形式说明枚举/数据/联合结构。

```
/* sccp interface with sccp user primitive message name */

enum  SCCP_USER_PRIMITIVE

{

    N_UNITDATA_IND, /* sccp notify sccp user unit data come */

    N_NOTICE_IND,   /* sccp notify user the No.7 network can
       not transmission this message */

    N_UNITDATA_REQ, /* sccp user's unit data

       transmission request*/
```

```
};
```

（6） 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时建议等的说明。

例如：

```
/* The ErrorCode when SCCP translate */

/* Global Title failure, as follows */     // 变量作用、含义

/* 0 — SUCCESS   1 — GT Table error */

/* 2 — GT error  Others — no use  */      // 变量取值范围

/* only  function  SCCPTranslate() in */

/* this modual can modify it,  and  other */

/* module can visit it through call */

/* the  function GetGTTransErrorCode() */    // 使用方法

g_nGTTranErrorCode;
```

（7） 对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。

这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

（8） 对于 switch 语句下的 case 语句，如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理，必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。

这样比较清楚程序编写者的意图，有效防止无故遗漏 break 语句。

例如（注意斜体加粗部分）：

```
case CMD_UP:

    ProcessUp();

    break;



case CMD_DOWN:
```

```
    ProcessDown();
    break;

case CMD_FWD:
    ProcessFwd();

if (...)
{
    ...
    break;
}
else
{
    ProcessCFW_B();   // now jump into case CMD_A
}

case CMD_A:
    ProcessA();
    break;

case CMD_B:
    ProcessB();
```

```
        break;


    case CMD_C:
        ProcessC();
         break;


    case CMD_D:
        ProcessD();
        break;
    ...
```

3.2 注释与代码的关系

（1） 尽量使代码成为自注释的

通过对函数、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的。清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

（2） 在代码的功能及意图层次上进行注释，提供有用且额外的信息

注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

例如

如下注释意义不大。

/ if receive_flag is TRUE */*

if (bReceive_flag)

而如下的注释则给出了额外有用的信息。

```
/* if mtp receive a message from links */
if (bReceiveFlag)
```

（3） 边写代码边注释，注释应与代码保持一致性

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。

（4） 注释应有针对性，且遵守惯例

对类、方法、变量的注释要求的内容不同。例如， 在注释方法时，要用第三人称描述性语言， 不能用第二人称命令性语言。

3.3 注释内容要求

（1） 注释的内容要清楚明了：注释的内容要清楚明了，含义准确，防止注释二义性。错误的注释不但无益反而有害。

（2） 避免在注释中使用缩写，特别是非常用缩写：在使用缩写时或之前，应对缩写进行必要的说明。

（3） 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。

（4） 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
```

（5） 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。

3.4 注释位置及格式

（1） 避免在一行代码或表达式的中间插入注释

除非必要，不应在代码或表达式中间插入注释，否则容易使代码可理解性变差。

（2） 将注释与其上面的代码用空行隔开

例如

如下示例，显得代码过于紧凑。

```
/* code one comments */
program code one
/* code two comments */
program code two
```

应如下书写

```
/* code one comments */

program code one
```

```
/* code two comments */
program code two
```

（3） 注释应与其描述的代码相近

注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面。若注释放于代码上方则需与其上面的其他代码用空行隔开。

例如

如下示例不符合规范。

例1：

```
/* get replicate sub system index and net indicator */



repssn_ind = ssn_data[index].m_RepssnIndex;
repssn_ni = ssn_data[index].m_Ni;
```

例2：

```
repssn_ind = ssn_data[index].m_RepssnIndex;
repssn_ni = ssn_data[index].m_Ni;
/* get replicate sub system index and net indicator */
```

应如下书写

```
/* get replicate sub system index and net indicator */
repssnInd = ssnData[index].m_RepssnIndex;
repssnNi = ssnData[index].m_Ni;
```

（4） 注释与所描述内容进行同样的缩排：可使程序排版整齐，并方便注释的阅读与理解。

例如

如下示例，排版不整齐，阅读稍感不方便。

```
void ExampleFun(void)
{
```

/ code one comments */*

CodeBlock One

/ code two comments */*

CodeBlock Two

}

应改为如下布局。

```
void ExampleFun(void)
{
    /* code one comments */
    CodeBlock One;

    /* code two comments */
    CodeBlock Two;
}
```

第4章 命名原则

4.1 命名规定

（1） 匈牙利命名法

匈牙利命名法通过在变量名前面加上相应的小写字母的符号标识作为前缀，标识出变量的作用域、类型等。

匈牙利命名法关键是：标识符的名字以一个或者多个小写字母开头作为前缀；前缀之后的是首字母大写的一个单词或多个单词组合，该单词要指明变量的用途。

这些前缀符号可以多个同时使用，顺序是先 m_(成员变量)，再指针，再简单数据类型，再其他。例如：m_lpszStr, 表示指向一个以 0 字符结尾的字符串的长指针成员变量。

匈牙利命名法中常用的小写字母的前缀如下表：

前缀	类型
a	数组 (Array)
b	布尔值 (Boolean)
cr	颜色参考值 (ColorRef)
c/n8	有符号字符 (Char)
uc/u8	无符号字符 (Char Byte)
n16	16 位有符号数
w/u16	16 位无符号数
n/n32	32 位有符号数
dw/u32	32 位无符号数
ll/n64	64 位有符号数
u64	64 位无符号数

fn	函数指针
h	Handle 32 位
s_	静态变量
m_	类的成员
g_	全局变量
p	Pointer
sz	以 null 做结尾的字符串型 (String with Zero End)

（2） 宏、常量和枚举

宏指所有用宏形式定义的名字，包括常量类和函数类。宏、常量和枚举常量的名字要全部大写，如果有多个单词，用下划线分隔。这可以使此类名称更加清晰，防止与其他类型的名字（主要是变量名）冲突。

例如

```
// 常量类的宏 PIE
#define  PIE  （3.1415926）
// 函数类的宏 MAX
#define  MAX (nObjOne, nObjTwo)  (/*…*/)
// 常量 LENGTH
const int LENGTH = 1024;
// 枚举中的常量成员 BLUE、RED、WHITE
enum
{
    BLUE,
    RED,
    WHITE
};
```

（3） 布尔型的变量命名用 b 作为前缀

（4） 变量和参数用小写字母开头的前缀或单词开始，命名中其余单词的首写字母大

写，但每个单词的非首写字母均小写。变量名称中的每个单词和缩写至少要包含两个字母。缩写要全部大写，尽量避免连续使用缩写，若无法避免，则两个相邻缩写单词之间用下划线隔开。

（5） 函数命名

（a） 函数接口命名采用首字母大写的英文加下划线的方法，并加前缀 CS_模块名称(缩写)_ ;若表述意思清晰，则可不加下划线。

例如：

```
void CS_PVR_Count_TimerWait(void);
HRESULT CSStringCbCat( LPTSTR pszDest, size_t cbDest
    ,LPCTSTR pszSrc);
```

（b） 所有非接口函数的第一个字母均大写，命名中其余单词的首写字母大写，但每个单词的非首写字母均小写。在命名中所有的单词和缩写至少包含两个字母。 缩写要全部大写，尽量避免连续使用缩写，若无法避免，则两个相邻缩写单词之间用下划线隔开。

（c） 其他函数命名约定：

i. wait: 具有悬挂当前运行程序的执行，直到某种条件满足后，当前程序继续执行，这样的函数名中应该包含‘wait’

ii. request: 需要满足一些条件才可执行，但不会悬挂进程的函数名中应包含‘request’

iii. go: 包含无限循环的函数，如进程的主函数，其命名中应该包含‘go’

（d） 所有函数名均使用宋体字

（6） 类名的首字母为大写字母，命名中其余单词的首写字母大写，但每个单词的非首写字母均小写。名称中的每个单词和缩写至少要包含两个字母。缩写要全部大写，尽量避免连续使用缩写，若无法避免，则两个相邻缩写单词之间用下划线隔开。

例如：

```
class Node;                 // 类名

class LeafNode;             // 类名
```

4.2 名字要求

（1） 名字要本着清晰简单的原则

名字本身首先要做到清晰明了，有明确含义，从而帮助（而不是混淆）对代码的理解；其次在清楚的前提下尽量简单(定量参考限制在 3 到 2 5 个字符之间)，简单本身也是要使阅读者更容易理解．理解之后才能谈到使用，使用之后才有修改和提高。

尽可能使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。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；一些单词用大家公认的缩写。

避免用模棱两可、晦涩或不标准不通用的缩写。

例如 1：

```
// 不好理解的名字
int  shldwncnt;
int  rs;
int  num;
// 比较一下
int  nShellDownloadCount;
int  nReturnStatus;
int  nAlarmNunber;
```

例如 2

如下单词的缩写能够被大家基本认可。

temp 可缩写为 tmp ;

flag 可缩写为 flg ;

statistic 可缩写为 stat ;

increment 可缩写为 inc ;

message 可缩写为 msg ;

（2） 命名中若使用特殊约定或缩写，则要有注释说明

应该在源文件的开始之处，对文件中所使用的缩写或约定，特别是特殊的缩写，进行必要的注释说明。

（3） 用英文命名

英语是最通用的语言，特别是在程序语言中，禁止使用其他语言（比如汉语拼音）。

（4） 具有互斥意义的变量或函数应当用正确的反义词组命名。

应当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。

下面是一些在软件中常用的反义词组。

```
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
cut / paste       up / down
```

例如

```
int  nMinSum;
int  nMaxSum;
int  AddUser(UINT8 *pu8UserName);
int  DeleteUser(UINT8 *pu8UserName);
```

4.3 命名共性规则

标识符应当直观且可以拼读，可望文知意，不必进行“解码”。

（1） 标识符的长度应当符合“min_length && max_information”原则（即使用最少的标识符而达到表达最多信息）。

（2） 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);
```

（3） 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。

（4） 变量的名字应当使用“名词”或者“形容词＋名词”。

例如：

```
int  nValue;
int  nOldValue;
int  nNewValue;
```

（5） 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->m_Draw();        // 类的成员函数
```

✧ 【**建议**】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）。

第5章 表达式和基本语句

表达式和语句都属于 C++/C 的短语结构语法。

5.1 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

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：

```
nWord = (nHigh << 8) | nLow
if ((nValueA | nValueB) && (nValueA & nValueC))
```

5.2 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。

（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
```

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;
```

（2） 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 )
```

5.3 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,主要列举如下：

（1） 布尔变量与零值比较

（2） 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 (bFlag)   // 表示 bFlag 为真
if (!bFlag)  // 表示 bFlag 为假
```

其它的用法都属于不良风格，例如：

```
if (bFlag == TRUE)
if (bFlag == 1 )
if (bFlag == FALSE)
if (bFlag == 0)
```

（3） 整型变量与零值比较

（4） 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 nValue，它与零值比较的标准 if 语句如下：

```
if (nValue == 0)
if (nValue != 0)
```

不可模仿布尔变量的风格而写成

```
if (nValue)      // 会让人误解 value 是布尔变量
if (!nValue)
```

（5） 浮点变量与零值比较

（6） 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 tmp，应当将

```
if   (tmp == 0.0)    // 隐含错误的比较
```

转化为

```
if ((tmp>=-EPSINON) && (tmp<=EPSINON))
```

其中 EPSINON 是允许的误差（即精度）。

（7） 指针变量与零值比较

（8） 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 pTdt，它与零值比较的标准 if 语句如下：

```
if (pTdt == NULL)   // pTdt 与 NULL 显式比较，强调 p 是指针变量
if (pTdt != NULL)
```

不要写成

```
if (pTdt == 0)      // 容易让人误解 p 是整型变量
if (pTdt != 0)
```

或者

```
if (pTdt)           // 容易让人误解 p 是布尔变量
if (!pTdt)
```

（9） 对 if 语句的补充说明

有时候我们可能会看到 if (NULL == pTdt) 这样古怪的格式。不是程序写错了，是程序员为了防止将 if (pTdt == NULL) 误写成 if (pTdt = NULL)，而有意把 p 和 NULL 颠倒。编译器认为 if (pTdt = NULL) 是合法的，但是会指出 if (NULL = pTdt)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 nVal;
return nErrorVal;
```

改写为

```
if (bCondition)
{
    return nVal;
}
else
{
```

```
return nErrorVal;
}
```

或者改写成更加简练的

```
return (bCondition ? nVal : nErrorVal);
```

5.4 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。提高循环体效率的基本办法是降低循环体的复杂性。

（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 5-4(b)的效率比示例 5-4(a)的高。

```
for (nRow=0; nRow <100; nRow ++)
{
    for ( nCol=0; nCol <5; nCol ++ )
    {
        nSum = nSum + aProgNo[nRow][ nCol];
    }
}
```

示例 5-4(a) 低效率：长循环在最外层

```
for (nCol=0; nCol <5; nCol ++ )
{
    for (nRow=0; nRow <100; nRow ++)
    {
        nSum = nSum + aProgNo[nRow][ nCol];
    }
}
```

示例 5-4(b) 高效率：长循环在最内层

（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 5-4(c)的程序比示例 5-4(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 5-4(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 5-4(c)的写法比较好，因为程序更加简洁。

```
for (i=0; i<N; i++)
{
    if (bCondition)
    {
        DoSomething();
    }
```

```
if (bCondition)
{
    for (i=0; i<N; i++)
    {
        DoSomething();
    }
```

```
        else
        {
            DoOtherThing();
        }
    }
```

表 5-4(c) 效率低但程序简洁

```
    }
    else
    {
        for (i=0; i<N; i++)
        {
            DoOtherThing();
        }

    }
```

表 5-4(d) 效率高但程序不简洁

5.5 for 语句的循环控制变量

（1） 不可在 for 循环体内修改循环变量，防止 for 循环失去控制。

（2） 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,但禁止在 for 条件中定义变量类型。

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

示例 5-5(b)中的 x 值属于闭区间“0 =< nTemp <= 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 5-5(a)的写法更加直观，尽管两者的功能是相同的。

```
    for ( nTemp =0; nTemp <N; nTemp ++)
    {
        …
    }
```

示例 5-5(a) 循环变量属于半开半闭区间

```
    for (nTemp =0; nTemp <=N-1; nTemp ++)
    {
        …
    }
```

示例 5-5(b) 循环变量属于闭区间

5.6 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 valueOne :  …
       break;
   case valueSec :  …
       break;
   …
   default :     …
       break;
}
```

（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）。

（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。

5.7 禁止使用 goto 语句

第6章 函数

6.1 函数声明

（1） 函数一定要做到先声明后使用

先声明使得编译器能够在编译时就检查和找出错误，而不是等到连接或运行时。C 程序没有强制要求，但也应先提供原型，再使用函数，C++必须这样做，否则编译通不过。

（2） 函数原型声明放在一个头文件中

将同类或相关的函数原型声明集中放在一个头文件中，有利于引用和修改，因为引用只引用一个熟知的头文件，修改也只在一个地方修改。

6.2 函数参数的规则

（1） 函数参数的书写要完整，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 SetValue(void);                   // 良好的风格
float SetValue();                       // 不良的风格
```

（2） 函数的参数命名要恰当，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 pcStr1 和 pcStr2，例如

```
void StringCopy(char *pcStr1, char *pcStr2);
```

那么我们很难搞清楚究竟是把 pcStr1 拷贝到 pcStr2 中，还是刚好倒过来。

可以把参数名字起得更有意义，如叫 pcStrSource 和 pcStrDestination。这样从名字上就可以看出应该把 pcStrSource 拷贝到 pcStrDestination。

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 *pcStrSource, char *pcStrDestination);
```

别人在使用时可能会不假思索地写成如下形式：

char caStr[20];

StringCopy(caStr, "Hello World");// 参数顺序颠倒

（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 *pcStrDestination , const char *pcStrSource);
```

（4） 如果输入参数以值传递的方式传递对象，则应使用“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。

（5） 避免函数有太多的参数，参数个数尽量控制在 5 个以内。因为如果参数太多，在使用时容易将参数类型或顺序搞错。

（6） 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。

6.3 返回值的规则

（1） 不要省略返回值的类型。

（a） 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。

（b） 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。

（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 cTmp;

cTmp = getchar();

if (cTmp == EOF)

    …
```

按照 getchar 名字的意思，将变量 cTmp 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 cTmp 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。

【建议】：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。

（3） 当函数的返回值是一个对象时，“引用传递”、“值传递”的使用原则。

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 &strObj,
        const String &strOrig);
    private:
        char *m_pcData;
}
```

String 的赋值函数 operate = 的实现如下：

```
String & String::operate=(const String &other)
{
    unsigned int u16MaxBytesNum;
    unsigned int nLength;
    unsigned int u16bDest;
    if (this == &other)
    {
        return *this;
    }
    delete m_pcData;
    u16MaxBytesNum = (unsigned int)sizeof(other.m_Data)
```

```
        + 1;
        if(CSStringCbLength(other.m_Data, u16MaxBytesNum
        , pu16RealCnt)==S_OK)
        {
            nLength = *pu16RealCnt;
            u16bDest = nLength + 1;
            m_pcData = new char[u16bDest];
            CSStringCbCopy(m_pcData, u16bDest, other.m_Data);
        }
        return *this; // 返回的是 *this 的引用，无需拷贝过程
    }
```

（4） 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 strObjA,strObjB,strObjC;
…
strObjA  = strObjB;            // 如果用“值传递”，将产生一次 *this 拷贝
strObjA = strObjB = strObjC;   // 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 &strObj, const String
  &strOrig)
{
    unsigned int u16MaxBytesNumOne;
    unsigned int u16MaxBytesNumSec;
    unsigned int * pu16RealCntOne;
    unsigned int * pu16RealCntSec;
    pu16RealCntOne = NULL;
    pu16RealCntSec = NULL;
    unsigned int nLength;
    String tmpStr;
```

```
    delete tmpStr.m_TmpData;   // tmpStr.m_TmpData 是仅含
        '\0'的字符串
    u16MaxBytesNumOne = (unsigned int)sizeof(strObj.m_Data)
        + 1;
   u16MaxBytesNumSec=(unsigned int)sizeof(strOrig.m_Data)
       + 1;
    if((CSStringCbLength(strObj.m_Data,u16MaxBytesNumOne
        ,pu16RealCntOne)==S_OK)
        &&(CSStringCbLength(strOrig.m_Data
        ,u16MaxBytesNumSec, pu16RealCntSec) == S_OK))
    {
         nLength = *pu16RealCntOne + *pu16RealCntSec;
         u16bDest = nLength + 2;
         tmpStr.m_TmpData = new char[u16bDest + 1];
         CSStringCbCopy(tmpStr.m_TmpData, cbDest
             , strObj.m_Data);
         CSStringCbCat(tmpStr.m_TmpData, cbDest
             , strOrig.m_Data);
    }
    return tmpStr;
}
```

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 tmpStr 的“引用”。由于 tmpStr 在函数结束时被自动销毁，将导致返回的“引用”无效。

6.4 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。

（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 6.10 节“使用断言”。

（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。

（3） 注意事项如下：

（a） 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 * pcFunc(void)
{
    char caStr[] = “hello world”;      // caStr 的内存位于栈上
    …
    return caStr;          // 将导致错误
}
```

（b） 要搞清楚返回的究竟是“值”、“指针”还是“引用”。

（c） 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(strObj + strOrig);
```

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(strObj + strOrig);
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。

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。

6.5 函数独立性要求

（1） 一个函数仅完成一个功能

（2） 函数的功能应该是可以预测的

函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出。

（a） 带有内部“存储器”的函数的功能可能是不可预测的，因为它的输出可能取决于内部存储器（如某标记）的状态。这样的函数既不易于理解又不利于测试和维护。

（b） 在 C/C++语言中，函数的 static 局部变量是函数的内部存储器，有可能使

函数的功能不可预测，然而，当某函数的返回值为指针类型时，则必须是STATIC 的局部变量的地址作为返回值，若为 AUTO 类，则返回为错误指针。

6.6 函数内容

(1) 防止把没有关联的语句放到一个函数中

（a) 防止函数内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数中。随机内聚给函数的维护、测试及以后的升级等造成了不便，同时也使函数的功能不明确。使用随机内聚函数，常常容易出现在一种应用场合需要改进此函数，而另一种应用场合又不允许这种改进，从而陷入困境。

（b) 在编程时，经常遇到在不同函数中使用相同的代码，许多开发人员都愿把这些代码提出来，并构成一个新函数。若这些代码关联较大并且是完成一个功能的，那么这种构造是合理的，否则这种构造将产生随机内聚的函数。

例如

如下函数就是一种随机内聚。

```
void InitVar(void)
{
    Rect.m_Length = 0;
    Rect.m_Width = 0; /* 初始化矩形的长与宽 */

    Point.m_X = 10;
    Point.m_Y = 10;   /* 初始化“点”的坐标 */
}
```

矩形的长、宽与点的坐标基本没有任何关系，故以上函数是随机内聚。

应如下分为两个函数：

```
void InitRect(void)
{
    Rect.m_Length = 0;
    Rect.m_Width = 0; /* 初始化矩形的长与宽 */
}
```

```
void InitPoint(void)
{
    Point.m_X = 10;
    Point.m_Y = 10;   /* 初始化“点”的坐标 */
}
```

(2) 若函数中对较长变量引用较多时，可用宏代替

当一个函数中对较长变量（一般是结构的成员）有较多引用时，可以用一个意义相当的宏代替。这样可以增加编程效率和程序的可读性。

例如

在某函数中较多引用 TheReceiveBuffer[FirstSocket].m_ByDataPtr，

则可以通过以下宏定义来代替：

```
# define pSOCKDATA (TheReceiveBuffer[FirstScoket]
    . m_ByDataPtr)
```

6.7 函数调用

（1） 尽量避免在调用函数填写参数时进行数据类型转换

在调用函数填写参数时，应尽量减少没有必要的默认数据类型转换或强制数据类型转换。因为数据类型转换或多或少存在危险。

（2） 对于提供了返回值的函数，在调用时最好使用其返回值

6.8 函数设计

（1） 为简单功能编写函数

虽然为仅用一两行就可完成的功能去编函数好象没有必要，但用函数可使功能明确化，增加程序可读性，亦可方便维护及测试。

例如

如下语句的功能不很明显。

```
nValue = (nValueA > nValueB) ? nValueA : nValueB ;
```

改为如下就很清晰了。

```
int Max(int nValueA, int nValueB)
{
```

```
    return (nValueA > nValueB) ? nValueA : nValueB);
}

nValue = Max(nValueA, nValueB);
```

或改为如下。

```
#define MAX(nValueA, nValueB) (((nValueA) > (nValueB)) ?
    (nValueA) : (nValueB))

nValue = MAX(nValueA, nValueB);
```

（2） 若多段代码重复做同一件事情，则函数划分可能存在问题

如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题。若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段代码构造成一个新的函数。

（3） 功能不明确的较小函数，应考虑把它合并到上级函数中

功能不明确较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在。模块中函数划分的过多，一般会使函数间的接口变得复杂。所以过小的函数，功能不明确的函数，不值得单独存在。

（4） 慎用函数本身或函数间的递归调用

递归调用特别是函数间的递归调用（如 funcA->funcB->funcC->funcA），影响程序的可理解性；递归调用一般都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。故除非为某些算法或功能的实现方便，应尽量慎用没必要的递归调用。应注明递归调用的层数，而且程序编写人员应该对栈的占用情况比较清晰。

（5） 仔细分析模块的需求，进行模块的函数划分与组织

仔细分析模块的功能及性能需求，并进一步细分，同时若有必要画出有关数据流图，据此来进行模块的函数划分与组织。函数的划分与组织是模块的实现过程中很关键的步骤，如何划分出合理的函数结构，关系到模块的最终效率和可维护性、可测性等。根据模块的功能图或/及数据流图映射出函数结构是常用方法之一。

（6） 明确函数功能，精确（而不是近似）地实现函数设计

（7） 优化模块中函数结构，降低函数间的耦合度

对初步划分后的函数结构应进行改进、优化，使之更为合理。应改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、效率和可维护性。

（8） 优化函数结构时，要遵守以下原则：

（a） 不能影响模块功能的实现

（b） 仔细考查模块或函数出错处理及模块的性能要求并进行完善。

（c） 通过分解或合并函数来改进软件结构。

（d） 考查函数的规模，过大的要进行分解。

（e） 降低函数间接口的复杂度。

（f） 函数功能应可预测。

（g） 提高函数内聚。（单一功能的函数内聚最高）

6.9 可重入函数

（1） 在多任务操作系统下编程，要注意函数可重入性的构造

（a） 可重入性是指函数可以被多个任务进程调用。在多任务操作系统中，函数是否具有可重入性是非常重要的，因为这是多个进程可以共用此函数的必要条件。

（b） 编译器是否提供可重入函数库，与它所服务的操作系统有关，只有操作系统是多任务时，编译器才有可能提供可重入函数库。如 DOS 下 BC 和 MSC 等就不具备可重入函数库，因为 DOS 是单用户单任务操作系统。

（2） 编写可重入函数时，应注意局部变量的使用

编写可重入函数时，应注意局部变量的使用。如编写 C/C++语言的可重入函数时，应使用 auto 即缺省态局部变量或寄存器变量，不应使用 static 局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

（3） 编写可重入函数时，若使用全局变量则应加以保护

编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即 P、V 操作）等手段对其加以保护。若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使有关全局变量变为不可知状态。

例如

假设 g_nExam 是 int 型全局变量，函数 Squre_Exam 返回 Exam 平方值。那么如下函数不具有可重入性。

```
unsigned int Example(int nPara)
{
    unsigned int u16Temp;
    g_nExam = nPara; // （**）
    u16Temp = SquareExam();
    return u16Temp;
}
```

此函数若被多个进程调用的话，其结果可能是未知的，因为当（**）语句刚执行完后，另外一个使用本函数的进程可能正好被激活，那么当新激活的进程执行到此函数时，将使 Exam 赋与另一个不同的 para 值，所以当控制重新回到“u16Temp = Square_Exam()”后，计算出的 u16Temp 很可能不是预想中的结果。此函数应如下改进。

```
unsigned int Example(int nPara)
{
    unsigned int u16Temp;

    /* 若申请不到“信号量”，说明另外的进程正处于给 Exam 赋值并计算 */
    /* 其平方过程中（即正在使用此信号），本进程必须等待其释放信号后，*/
    /*  才可继续执行。若申请到信号，则可继续执行，但其它进程必须等待*/
    /*  本进程释放信号量后，才能再使用本信号。*/
    [申请信号量操作]
    g_nExam = nPara;
    u16Temp = SquareExam();
    [释放信号量操作]

    return u16Temp;
}
```

6.10 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，nRfc 是 nRft 的一个引用（reference），nRft 是被引用物（referent）。

```
int nRft;
int &nRfc = nRft;
```

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

（1） 引用的一些规则如下：

（a） 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）。

（b） 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）。

（c） 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）。

（2） “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，

（3） 使用“引用”的必要性：

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;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。

6.11 其它建议

（1） 函数的功能要单一，不要设计多用途的函数。

（2） 函数体的规模要小，尽量控制在 100 行代码之内。

（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。

（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。

（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。

第7章 宏

7.1 若宏值多于一项，一定要使用括号

这是 C 语言的一条原则，当宏用在表达式中时，防止出现计算混乱。

// 若值多于一项，这样做危险

#define ERROR_STRING_LENGTH 100+1

*// malloc() 的参数变成 5*100+1=501，而不是想要的5*(100+1)=505 了*

malloc (5 * ERROR_STRIING_LENGTH)

// 应这样定义

#define ERROR+STRING_LENGTH (100+1)

7.2 不要用分号结束宏定义

用分号结尾，会导致错误。

#define CHANNEL_MAX 16;

int nChannel = CHANNEL_MAX – 1;

结果上面的语句变成“channel = 16; -1;”,编译出错。

7.3 使用宏时，不允许参数发生变化

如下用法可能导致错误。

```
#define SQUARE( nSqr ) ((nSqr) * (nSqr))


int nSqr = 5;
int nRslt;
nRslt = SQUARE(nSqr ++ ); // 结果：nSqr = 7，即执行了两次增1。
```

正确的用法是：

```
nRslt = SQUARE(nSqr );
nSqr ++; // 结果：nSqr = 6，即只执行了一次增1。
```

7.4 不要用宏改写语言

语言已经有完善且众所周知的语法，试图将其改变成类似于其他语言的形式会使阅读者混淆和难于理解。

例如

// 不要定义这类的宏

define FOREVER for (;;)

define BEGIN {

define END }

7.5 关于宏函数

（1） 函数宏的每个参数都要括起来

// 参数 nWk 未括起来

*# define WEEKS_TO_DAYS (nWk) (nWk * 7)*

*// 结果是 1+2*7=15，而不是(1+2)*7=21*

nTotalDays = WEEKS_TO_DAYS (1+2);

// 应这样定义

define WEEKS_TO_DAYS (nWk) ((nWk)*7)

例如：如下定义的宏都存在一定的风险。

```
#define RECTANGLE_AREA( nLenth, nWidth ) nLenth * nWidth
#define RECTANGLE_AREA( nLenth, nWidth ) (nLenth * nWidth)
#define RECTANGLE_AREA( nLenth, nWidth ) (nLenth) * (nWidth)
```

正确的定义应为：

```
#define RECTANGLE_AREA( nLenth, nWidth ) ((nLenth) * (nWidth))
```

（2） 不带参数的宏函数也要定义成函数形式

因为括号会暗示阅读者此宏是一个函数。

// 这种形式较好

define　HELLO ( )　printf (“Hello.”);

// 不带括号不清晰

#define　HELLO　printf(“Hello.”)

（3）　一般要将函数宏的函数体用{ }括起来

除特殊情况外，应当将函数宏所定义的多条表达式放在大括号中。尽量保证宏函数能作为一条独立的语句，不干扰别人，也不受别人干扰。。

例如

// 未给函数体加上｛｝

#define　MY-ASSERT (bCondition)

*if (!(bCondition))　　*

{

*　　assertError (_FILE_,　_LINE_)*

}

//　出问题了

if　((nValA<0) && (nValB>0))

MY_ASSERT (nValA>nValB);　//　MY_ASSERT()的 if 和下面的 else 接到一起了，错

else　　//　想和 if ((nValA<0) && (nValB>0)) 连用，但被 MY_ASSERT()破坏了

MY_ASSERT (nValB>nValA);

//　应当加上｛｝

#define　MY_ASSERT (bCondition)　　\

{ \

　　if (!(bCondition))　　\

　　{

　　　　assertError (_FILE_,　_LINE_);　　\

　　} \

}

7.6 关于宏的替代方法

（1） 尽量用常量替代类似功能的宏

常量和宏的效率同样高，但宏没有作用域，也没有类型，不能做类型检查，不安全，跟踪调试时显示的是值而不是宏名，报错时直接报相应错误值，不易理解。

```
// 宏定义的方法
#define  PIE  (3.14)
// 常量定义的方法可以替代宏，且更好
const  float  PIE = (3.14);
```

（2） 编程人员禁止使用 inline 函数，但编译等工具自动生成的 inline 允许使用

（3） 尽量用 typedef 代替宏定义新类型

宏的最大弱点（特点）是在预编译时做简单的串替换，没有任何语法约束在里面，所以看似被宏归为一类的一串语法元素，其实是一盘散沙，和宏之外的其他元素没有任何差别，这是导致期望与效果不同的主要原因。

```
//  不要用宏定义新类型
# define  LongPtr_T  long*
//  下面这句话相当于“long*  pPtrA, pPtrB;”，所以 pPtrB 是长整型，而不是长整指针
LongPtr_T  pPtrA, pPtrB;

//  用 typedef 可以消除宏的弊病
typedef  long* LongPtr_T;
LongPtr_T  pPtrA;
LongPtr_T  pPtrB;
```

第8章 常量、变量、结构

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）。

8.1 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，将会有如下麻烦：

（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。

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
```

8.2 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。

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。

8.3 常量定义规则

（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。

（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;
```

8.4 类中的常量

有时我们希望某些常量只在类中有效。由于#define 定义的宏常量是全局的，不能达到目的，于是想当然地觉得应该用 const 修饰数据成员来实现。const 数据成员的确是存在的，但其含义却不是我们所期望的。const 数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的，因为类可以创建多个对象，不同的对象其 const 数据成员的值可以不同。

不能在类声明中初始化 const 数据成员。以下用法是错误的，因为类的对象未被创建时，编译器不知道 SIZE 的值是什么。

```
class Example
{…
    const int nSize = 100;    // 错误，企图在类声明中初始化 const 数据成员
    int naTmp[nSize];         // 错误，未知的 nSize
};
```

const 数据成员的初始化只能在类构造函数的初始化表中进行，例如

```
class Example
{…
    Example (int nSize);            // 构造函数
    const int SIZE ;
};
Example:: Example (int nSize) : SIZE(nSize)      // 构造函数的初始化表
{
   …
}
Example   objA(100);   // 对象 objA 的 SIZE 值为 100
```

Example　objB(200);　// 对象 objB 的 SIZE 值为 200

应该用类中的枚举常量来实现。例如

```
class Example
{…
    enum { SIZE1 = 100, SIZE2 = 200}; // 枚举常量
    int naArrayOne[SIZE1];
    int naArrayOther[SIZE2];
};
```

枚举常量不会占用对象的存储空间，它们在编译时被全部求值。枚举常量的缺点是：它的隐含数据类型是整数，其最大值有限，且不能表示浮点数（如 PI=3.14159）。

8.5　关于公共变量

（1）　去掉没必要的公共变量

公共变量是增大模块间耦合的原因之一，故应减少没必要的公共变量以降低模块间的耦合度。

（2）　仔细定义公共变量的含义、作用、取值范围及公共变量间的关系

应当仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。在对变量声明的同时，应对其含义、作用及取值范围进行注释说明，同时若有必要还应说明与其它变量的关系。

（3）　明确公共变量与操作此公共变量的函数的关系

应当明确公共变量与操作此公共变量的函数的关系，如访问、修改及创建等。明确过程操作变量的关系后，将有利于程序的进一步优化、单元测试、系统联调以及代码维护等。这种关系的说明可在注释或文档中描述。

例如

在源文件中，可按如下注释形式说明。

RELATION	System_Init	Input_Rec	Print_Rec	Stat_Score
Student	Create	Modify	Read	Read
Score	Create	Modify	Read	Read,Modify

注：RELATION 为操作关系；System_Init、Input_Rec、Print_Rec、Stat_Score 为四个不同的函数；Student、Score 为两个全局变量；Create 表示创建，Modify 表示修改，Read 表示读取。其中，函数 Input_Rec、Stat_Score 都可修改变量 Score，故此变量将引起函数间较大的耦合，并可能增加代码测试、维护的难度。

（4） 尽量降低公共变量的耦合度

应当尽量降低公共变量的耦合度。设计公共变量与操作此公共变量的函数（或模块）的关系时，尽量设计成仅有一个函数（或模块）可以修改及创建此公共变量，而其余有关函数（或模块）只能访问。防止多个不同函数（或模块）都可以修改、创建同一公共变量的现象。

（5） 向公共变量传递数据要十分小心

当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生。若有必要应进行合法性检查，以提高代码的可靠性与稳定性。

8.6 变量避免事项

（1） 防止局部变量与公共变量同名

（2） 禁使用未经初始化的变量作为右值

严禁使用未经初始化的变量作为右值。特别是在 C/C++中引用未经赋值的指针，经常会引起系统崩溃。

8.7 结构设计

（1） 结构的功能要单一

结构的功能要单一，是针对一种事务的抽象。设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

例如

如下结构不太清晰、合理。

```
typedef struct STUDENT_STRU
{
    unsigned char u8Name[8]; /* student's name */
    unsigned char u8Age;     /* student's age */
    unsigned char u8Sex;     /* student's sex, as follows */
                             /* 0 - FEMALE; 1 - MALE */
    unsigned char u8TeacherName[8]; /* the student teacher's name */
    unisgned char u8TeacherSex;     /* his teacher sex */
}STUDENT;
```

若改为如下，会更合理些。

```
typedef struct TEACHER_STRU
{
    unsigned char u8Name[8]; /* teacher name */
    unisgned char u8Sex;        /* teacher sex, as follows */
                                /* 0 - FEMALE; 1 - MALE */
}TEACHER;

typedef struct STUDENT_STRU
{
    unsigned char u8Name[8];        /* student's name */
    unsigned char u8Age;            /* student's age */
    unsigned char u8Sex;            /* student's sex, as follows */
                                    /* 0 - FEMALE; 1 - MALE */
    unsigned int u16TeacherInd;     /* his teacher index */
}STUDENT;
```

（2） 不要设计面面俱到的数据结构

面面俱到、非常灵活的数据结构反而容易引起误解和操作困难。

（3） 不同结构间的关系不要过于密切或复杂

若两个结构间的关系较密切或复杂，那么应合为一个结构。

例如

如下两个结构的构造不合理。

```
typedef struct PERSON_ONE_STRU
{
    unsigned char u8Name[8];
    unsigned char u8Addr[40];
    unsigned char u8Sex;
    unsigned char u8City[15];
}PERSON_ONE;
```

```
typedef struct PERSON_TWO_STRU
{
    unsigned char u8Name[8];
    unsigned char u8Age;
    unsigned char u8Tel;
}PERSON_TWO;
```

由于两个结构都是描述同一事物的，那么不如合成一个结构。

```
typedef struct PERSON_STRU
{
    unsigned char u8aName[8];
    unsigned char u8Age;
    unsigned char u8Sex;
    unsigned char u8aAddr[40];
    unsigned char u8aCity[15];
    unsigned char u8Tel;
}PERSON;
```

（4） 结构中元素的个数应适中

结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同的子结构，以减少原结构中元素的个数。这增加结构的可理解性、可操作性和可维护性。

例如

假如认为如上的_PERSON 结构元素过多，那么可如下对之划分。

```
typedef struct PERSON_BASE_INFO_STRU
{
    unsigned char u8aName[8];
    unsigned char u8Age;
    unsigned char u8Sex;
}PERSON_BASE_INFO;

typedef struct PERSON_ADDRESS_STRU
```

```
{
    unsigned char u8aAddr[40];
    unsigned char u8aCity[15];
    unsigned char u8Tel;
}PERSON_ADDRESS;

typedef struct PERSON_STRU
{
    PERSON_BASE_INFO person_base;
    PERSON_ADDRESS person_addr;
}PERSON;
```

（5） 仔细设计结构中元素的布局与排列顺序

仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象。合理排列结构中元素顺序，可节省空间并增加可理解性。

例如

如下结构中的位域排列，将占较大空间，可读性也稍差。

```
typedef struct EXAMPLE_STRU
{
    unsigned int nValid: 1;
    PERSON person;
    unsigned int nSetFlg: 1;
}EXAMPLE;
```

若改成如下形式，不仅可节省一个 int 的空间，可读性也变好了。

```
typedef struct EXAMPLE_STRU
{
    unsigned int u16Valid: 1;
    unsigned int u16SetFlg: 1;
    PERSON person;
}EXAMPLE;
```

8.8 类型转换

（1） 要注意强制的数据类型转换

编程时，要注意数据类型的强制转换。当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，而这些细节若考虑不周，就很有可能留下隐患。

（2） 要注意编译系统默认的数据类型转换

对编译系统默认的数据类型转换，也要有充分的认识。有时尽管编译器不产生告警，但值的含义还是稍有变化。

例如

如下赋值，多数编译器不产生告警，但值的含义会有变化。

```
char cPaint;
unsigned  int u16Exam;
cPaint = -1;
u16Exam = cPaint; // 编译器不产生告警，此时u16Exam 为0xFFFF。
```

（3） 尽量减少数据类型默认转换与强制转换

非必要时，应尽量减少数据类型默认转换与强制转换。应当合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。

8.9 类型使用优化建议

（1） 应对自定义数据类型进行恰当命名

使用自定义类型，可以弥补编程语言提供类型少、信息量不足的缺点，并能使程序清晰、简洁。对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性。注意其命名方式在同一产品中的统一。

例如

可参考如下方式声明自定义数据类型。下面的声明可使数据类型的使用简洁、明了。

```
typedef unsigned char   UINT8;
typedef unsigned short  UINT16;
typedef short INT16;
typedef unsigned int UINT32;
```

下面的声明可使数据类型具有更丰富的含义。

```
typedef float DISTANCE;
typedef float SCORE;
```

（2） 用 typedef 简化程序中的复杂语法

当语法非常复杂时（比如函数指针的定义很难读，尤其当该函数比较复杂时），用 typedef 可以大大简化复杂度，使阅读理解更容易。一般包含 4 个以上独立元素的语法应被视为复杂语法。

例如

// 用 typedef 简化函数指针

typedef int(*CallbackFunctionPtr_T) (int parameter);

上述函数指针定义，不算 typedef 的独立元素为 5 (int, *, CallbackFunctionPtr_T, int , parameter).

（3） 用 enum 取代一组相关的常量

枚举可以对其成员编号，可以方便增加或减少成员等修改／维护工作，比#define 或 int const 更安全且使用更方便。

例如

```
// 为每个成员自动编号
enum
{
    BLUE,
    RED,
    WHITE
};

// 用一个匿名的枚举把这一组相关的常量放在一起
enum
{
    EMPTY_ENTRY = -1,
    NOT_FOUND = -1,
    MAP-FULL = -2
};
```

8.10 类型避免事项

（1） 避免隐式声明类型

尽量用显式声明。

例如

下面例子的隐式表示法不好。

main (); // *int main (void) 的隐式表示法*

void foo (const nValue); // *是 void foo (const int nValue)的隐式表示法*

（2） 慎用无符号类型

除非必要，应避免使用无符号类型。混用有符号和无符号类型常会导致奇怪的结果，因为其中会发生隐式类型转换，而不同的编译器有符号和无符号的转换规则不同。

但按位访问的数据和设备寄存器通常要用无符号类型。

（3） 尽量少用浮点类型

计算机内部将代码中十进制浮点数转换成二进制时，由于字长限制会丢掉最后几位小数，造成结果不精确，且浮点类型异常处理复杂（比如上溢、下溢等处理），此外，浮点类型运算速度较慢。

例如

```
int  nBaudRate = 9600;
int  nSymbolsIn15msec;
// 尽量少用以下浮点运算方式
nSymbolsIn15msec = (int)(nBaudRate * 0.015);

// 可用以下方式表达
nSymbolsIn15msec = (nBaudRate * 15) / 1000;
```

（4） 少用 union

union 成员公用内存空间，容易出错，且维护困难。

第9章 类

9.1 类的独立性和一致性

（1） 不要在类定义时提供成员函数体。

（2） 保持类的不同接口或不同类的接口在实现原则上的一致性

（3） 提高类内聚合度，降低类间耦合度；防范、杜绝类内函数潜在的二义性。

（4） 显式禁止编译器自动生成不需要的函数。

（5） 限制继承的层数，慎用多重继承，继承树上非叶子节点的类应是虚基类。

（6） 显式提供继承和访问修饰，及显式指出继承的虚函数。

（7） 基类析构函数首选是虚函数，所有多重继承的基类析构函数都应是虚函数。

（8） 绝不要重新定义（继承来的）非虚函数，以及绝不要重新定义缺省参数值；不要将基类强制转换成派生类。

9.2 类的成员变量和函数

（1） 类的成员变量

类成员变量应是私有的，避免为每个类成员提供访问函数。

（2） 类的构造函数、析构函数与赋值函数

构造函数、析构函数与赋值函数是每个类最基本的函数。每个类只有一个析构函数和一个赋值函数，但可以有多个构造函数（包含一个拷贝构造函数，其它的称为普通构造函数）。对于任意一个类 CatSch，如果不想编写上述函数，C++编译器将自动为 CatSch 产生四个缺省的函数，如

```
CatSch (void);                          // 缺省的无参数构造函数
CatSch (const CatSch &a);               // 缺省的拷贝构造函数
~CatSch (void);                         // 缺省的析构函数
CatSch & operate =(const CatSch &a); // 缺省的赋值函数
```

（3） 编写构造函数，析构函数原因如下：

（a） 如果使用“缺省的无参数构造函数”和“缺省的析构函数”，等于放弃了自主“初始化”和“清除”的机会，C++发明人 Stroustrup 的好心好意白费了。

（b） （2）“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，倘若类中含有指针变量，这两个函数注定将出错。

在此以类 String 的设计与实现为例，深入阐述很多被忽视了的道理。String 的结构如下：

```
class String
{
    public:
        String(const char *str = NULL);   // 普通构造函数
        String(const String &other);  // 拷贝构造函数
        ~ String(void);                   // 析构函数
        String & operate =(const String &other);    // 赋值函数
     private:
        char   *m_pcData;                 // 用于保存字符串
};
```

（4） 构造函数的初始化表

构造函数有个特殊的初始化方式叫“初始化表达式表”（简称初始化表）。初始化表位于函数参数表之后，却在函数体 {} 之前。这说明该表里的初始化工作发生在函数体内的任何代码被执行之前。

构造函数初始化表的使用规则：

（a） 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。

例如

```
class CatSch
{…
    CatSch (int nCnt);      // CatSch 的构造函数
};
class CatSchB : public CatSch
{…
    CatSchB (int nCntX, int nCntY);// CatSchB 的构造函数
};
CatSchB:: CatSchB(int nCntX, int nCntY)
   : CatSch (nCntX)            // 在初始化表里调用 CatSch 的构造函数
```

```
{
 ….
}
```

（b） 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。

（c） 类的数据成员的初始化可以采用初始化表或函数体内赋值两种方式，这两种方式的效率不完全相同。非内部数据类型的成员对象应当采用第一种方式初始化，以获取更高的效率。例如

```
class CatSch
{…
    CatSch (void);                  // 无参数构造函数
    CatSch (const CatSch &other);     // 拷贝构造函数
    CatSch & operate =( const CatSch &other);   // 赋值函数
};

class CatSchB
{
    public:
       CatSchB(const CatSch &paramtr);   // CatSchB 的构造函数
     private:
       CatSch  m_CatSch;              // 成员对象
};
```

示例 9-5(a)中，类 CatSchB 的构造函数在其初始化表里调用了类 CatSch 的拷贝构造函数，从而将成员对象 m_CatSch 初始化。

示例 9-5 (b)中，类 CatSchB 的构造函数在函数体内用赋值的方式将成员对象 m_CatSch 初始化。我们看到的只是一条赋值语句，但实际上 CatSchB 的构造函数干了两件事：先暗地里创建 m_CatSch 对象（调用了 CatSch 的无参数构造函数），再调用类 CatSch 的赋值函数，将参数 paramtr 赋给 m_CatSch。

CatSchB:: CatSchB(const CatSch & paramtr) : m_CatSch (paramtr) {	CatSchB:: CatSchB(const CatSch & paramtr) { m_CatSch = paramtr;

… }	… }

示例 9-5(a) 成员对象在初始化表中被初始化　　示例 9-5(b) 成员对象在函数体内被初始化

对于内部数据类型的数据成员而言，两种初始化方式的效率几乎没有区别，但后者的程序版式似乎更清晰些。若类 Flight 的声明如下：

```
class Flight
{
    public:
        Flight (int nCntX, int nCntY);        // 构造函数
    private:
        int m_nCntX;
        int m_nCntY;
        int m_nRstA;
        int m_nRstB;
}
```

示例 9-5(c)中 Flight 的构造函数采用了第一种初始化方式，示例 9-5(d)中 Flight 的构造函数采用了第二种初始化方式。

```
Flight:: Flight (int nCntX, int nCntY)
    : m_nCntX (nCntX), m_nCntY(nCntY)
{
    m_nRstA = 0;
    m_nRstB = 0;
}
```

```
Flight:: Flight (int nCntX, int nCntY)
{
    m_nCntX = CntX;
    m_nCntY = nCntY;
    m_nRstA = 0;
    m_nRstB = 0;
}
```

示例 9-5(c) 数据成员在初始化表中被初始化　　示例 9-5(d) 数据成员在函数体内被初始化

（5） 构造和析构的次序

构造从类层次的最根处开始，在每一层中，首先调用基类的构造函数，然后调用成员对象的构造函数。析构则严格按照与构造相反的次序执行，该次序是唯一的，否则编译器将无法自动执行析构过程。

一个有趣的现象是，成员对象初始化的次序完全不受它们在初始化表中次序的影响，只

由成员对象在类中声明的次序决定。这是因为类的声明是唯一的，而类的构造函数可以有多个，因此会有多个不同次序的初始化表。如果成员对象按照初始化表的次序进行构造，这将导致析构函数无法得到唯一的逆序。

例如：类 String 的构造函数与析构函数

```
// String 的普通构造函数
String::String(const char *pcStr)
{
    unsigned int nLength;
    if(pcStr==NULL)
    {
        m_pcData = new char[1];
        *m_pcData = '\0';
    }
    else
    {
        if(CSStringCbLength(pcStr, u16MaxBytesNum
            , pu16RealCnt) == S_OK)
        {
            nLength = *pu16RealCnt;
            u16bDest = nLength + 1;
            m_pcData = new char[u16bDest];
            CSStringCbCopy(m_pcData, u16bDest, pcStr);
        }
    }
}

// String 的析构函数
    String::~String(void)
    {
        delete []m_pcData;
```

```
// 由于 m_Data 是内部数据类型，也可以写成 delete m_pcData;
}
```

（6） 不要轻视拷贝构造函数与赋值函数

由于并非所有的对象都会使用拷贝构造函数和赋值函数，程序员可能对这两个函数有些轻视。请先记住以下的警告，在阅读正文时就会多心：

（a） 如果不主动编写拷贝构造函数和赋值函数，编译器将以“位拷贝”的方式自动生成缺省的函数。倘若类中含有指针变量，那么这两个缺省的函数就隐含了错误。以类 String 的两个对象 StrA,StrB 为例，假设 StrA.m_Data 的内容为“hello”，StrB.m_Data 的内容为“world”。现将 StrA 赋给 StrB，缺省赋值函数的“位拷贝”意味着执行 StrB.m_Data = StrA.m_Data。这将造成三个错误：一是 StrB.m_Data 原有的内存没被释放，造成内存泄露；二是 StrB.m_Data 和 StrA.m_Data 指向同一块内存，StrA 或 StrB 任何一方变动都会影响另一方；三是在对象被析构时，m_Data 被释放了两次。

（b） 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。以下程序中，第三个语句和第四个语句很相似，注意哪个调用了拷贝构造函数，哪个调用了赋值函数

```
String  StrA(“hello”);
String  StrB(“world”);
String  StrC = StrA; // 调用了拷贝构造函数，
    最好写成 StrC(StrA);
StrC = StrB;  // 调用了赋值函数
```

本例中第三个语句的风格较差，宜改写成 String StrC(StrA) 以区别于第四个语句。

例如：类 String 的拷贝构造函数与赋值函数

```
// 拷贝构造函数
String::String(const String &other)
{
    unsigned int nLength;
    // 允许操作 other 的私有成员 m_Data
    if(CSStringCbLength(other.m_Data, u16MaxBytesNum
        , pu16RealCnt) == S_OK)
        {
            nLength = *pu16RealCnt;
```

```
            u16bDest = nLength + 1;
            m_pcData = new char[u16bDest];
            CSStringCbCopy(m_pcData, u16bDest, other.m_Data);
        }
    }

// 赋值函数
    String & String::operate =(const String &other)
    {
        unsigned int nLength;
        // (1) 检查自赋值
        if(this == &other)
        {
            return *this;
        }

        // (2) 释放原有的内存资源
        delete []m_pcData;

        // （3）分配新的内存资源，并复制内容
        if(CSStringCbLength(other.m_Data, u16MaxBytesNum
            , pu16RealCnt) == S_OK)
        {
            nLength = *pu16RealCnt;
            u16bDest = nLength + 1;
            m_pcData = new char[u16bDest];
            CSStringCbCopy(m_pcData, u16bDest, other.m_Data);
        }
        // （4）返回本对象的引用
        return *this;
```

```
}
```

类 String 拷贝构造函数与普通构造函数（参见 9.4 节）的区别是：在函数入口处无需与 NULL 进行比较，这是因为“引用”不可能是 NULL，而“指针”可以为 NULL。

（7） 类 String 的赋值函数比构造函数复杂得多，分四步实现：

（a） 第一步，检查自赋值。

（b） 第二步，用 delete 释放原有的内存资源。如果现在不释放，以后就没机会了，将造成内存泄露。

（c） 第三步，分配新的内存资源，并复制字符串。注意函数 CSStringCbLength 返回的是有效字符串长度，不包含结束符‘\0’。函数 CSStringCbCopy 则连‘\0’一起复制。

（d） 第四步，返回本对象的引用，目的是为了实现象 a = b = c 这样的链式表达。注意不要将 return *this 错写成 return this 。不可以！因为我们不知道参数 other 的生命期。有可能 other 是个临时对象，在赋值结束后它马上消失，那么 return other 返回的将是垃圾。

（8） 处理拷贝构造函数与赋值函数

如果我们实在不想编写拷贝构造函数和赋值函数，又不允许别人使用编译器生成的缺省函数，只需将拷贝构造函数和赋值函数声明为私有函数，不用编写代码。

例如：

```
class StdTable
{ …
  private:
    StdTable (const StdTable &a);     // 私有的拷贝构造函数
    StdTable & operate =(const StdTable &a);// 私有的
        赋值函数
};
```

如果有人试图编写如下程序：

```
StdTable  StdTblB(a);   // 调用了私有的拷贝构造函数
StdTblB = a;        // 调用了私有的赋值函数
```

编译器将指出错误，因为外界不可以操作 A 的私有函数。

（9） 在派生类中实现类基本函数的注意事项

基类的构造函数、析构函数、赋值函数都不能被派生类继承。如果类之间存在继承关系，在编写上述基本函数时应注意以下事项：

（a） 派生类的构造函数应在其初始化表里调用基类的构造函数。

（b） 基类与派生类的析构函数应该为虚（即加 virtual 关键字）。例如

```
#include <iostream.h>
class Base
{
   public:
       virtual ~Base() { cout<< "~Base" << endl ; }
};

class Derived : public Base
{
    public:
       virtual ~Derived() { cout<< "~Derived" << endl ; }
};

void main(void)
{
   Base * pTmp = new Derived;  // upcast
   delete pTmp;
}
```

输出结果为：

```
~Derived
~Base
```

如果析构函数不为虚，那么输出结果为

```
~Base
```

（c） 在编写派生类的赋值函数时，注意不要忘记对基类的数据成员重新赋值。例如：

```
class Base
{
```

```
    public:
…
        Base & operate =(const Base &other); // 类Base的赋值函
数
    private:
        int  m_nLength;
        int  m_nWidth;
        int  m_nHeight;
};

class Derived : public Base
{
    public:
…
        Derived & operate =(const Derived &other);  // 类
            Derived的赋值函数
    private:
        int  m_nArea;
        int  m_nLength
        int  m_nWidth;
};

Derived & Derived::operate =(const Derived &other)
{
    //（1）检查自赋值
    if(this == &other)
    {
        return *this;
    }
```

```
        //（2）对基类的数据成员重新赋值
        Base::operate =(other); // 因为不能直接操作私有数据成员

        //（3）对派生类的数据成员赋值
        m_nArea = other.m_nOtherAr;
        m_nLength = other.m_nOtherLength;
        m_nWidth = other.m_nOtherWidth;

        //（4）返回本对象的引用
        return *this;
    }
```

9.3 类的继承与组合

对象（Object）是类（Class）的一个实例（Instance）。如果将对象比作房子，那么类就是房子的设计图纸。所以面向对象设计的重点是类的设计，而不是对象的设计。

（1） 继承

如果 Base 是基类，DeriveBase 是 Base 的派生类，那么 DeriveBase 将继承 Base 的数据和函数。例如：

```
class Base
{
    public:
        void  FuncSearch(void);
        void  FuncLock(void);
};

class DeriveBase : public Base
{
    public:
        void  FuncPlay(void);
        void  FuncDisplay(void);
```

```
};


main()
{
    DeriveBase  tmpObj;
    tmpObj.FuncSearch();        // DeriveBase 从 Base 继承了
        函数 funcSearch
    tmpObj.FuncLock();          // DeriveBase 从 Base 继承了
        函数 funcLock
    tmpObj.FuncPlay();
    tmpObj.FuncDisplay();
}
```

（2） “继承”使用规则。

（a） 如果类 Base 和类 DeriveBase 毫不相关，不可以为了使 DeriveBase 的功能更多些而让 DeriveBase 继承 Base 的功能和属性。不要觉得“白吃白不吃”，让一个好端端的健壮青年无缘无故地吃人参补身体。

（b） 若在逻辑上 DeriveBase 是 Base 的“一种”（a kind of ），则允许 DeriveBase 继承 Base 的功能和属性。例如男人（Man）是人（Human）的一种，男孩（Boy）是男人的一种。那么类 Man 可以从类 Human 派生，类 Boy 可以从类 Man 派生。

```
class Human
{
    …
};
class Man : public Human
{
    …
};
class Boy : public Man
{
    …
```

```
};
```

◆ **注意事项**

看起来很简单，但是实际应用时可能会有意外，继承的概念在程序世界与现实世界并不完全相同。

例如从生物学角度讲，鸵鸟（Ostrich）是鸟（Bird）的一种，按理说类 Ostrich 应该可以从类 Bird 派生。但是鸵鸟不能飞，那么 Ostrich::Fly 就不对了。

```
class Bird
{
   public:
       virtual void Fly(void);
       …
};


class Ostrich : public Bird
{
    …
};
```

例如从数学角度讲，圆（Circle）是一种特殊的椭圆（Ellipse），按理说类 Circle 应该可以从类 Ellipse 派生。但是椭圆有长轴和短轴，如果圆继承了椭圆的长轴和短轴，画蛇添足了。

所以更加严格的继承规则应当是：若在逻辑上 B 是 A 的“一种”，并且 A 的所有功能和属性对 B 而言都有意义，则允许 B 继承 A 的功能和属性。

（3） 组合

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。

例如眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类 Head 应该由类 Eye、Nose、Mouth、Ear 组合而成，不是派生而成。如示例 9-2-1 所示。

```
class Eye
{
     public:
          void    m_Look(void);
};
```

```
class Nose
{
      public:
           void    m_Smell(void);
};
```

```
class Mouth
{
    public:
        void    m_Eat(void);
};
```

```
class Ear
{
    public:
        void    m_Listen(void);
};
```

```
// 正确的设计，虽然代码冗长。
class Head
{
    public:
        void        m_HdLook(void)
        {
            m_Eye.m_Look();
        }
        void        m_HdSmell(void)
        {
            m_Nose.m_Smell();
        }
        void        m_HdEat(void)
        {
            m_Mouth.m_Eat();
        }
        void        m_HdListen(void)
        {
            m_Ear.m_Listen();
        }
    private:
        Eye         m_Eye;
        Nose        m_Nose;
        Mouth       m _Mouth;
```

```
    Ear         m_Ear;
};
```

示例 9-2-1 Head 由 Eye、Nose、Mouth、Ear 组合而成

如果允许 Head 从 Eye、Nose、Mouth、Ear 派生而成，那么 Head 将自动具有 Look、Smell、Eat、Listen 这些功能。示例 9-2-2 十分简短并且运行正确，但是这种设计方法却是不对的。

```
// 功能正确并且代码简洁，但是设计方法不对。
class Head : public Eye, public Nose, public Mouth, public Ear
{
};
```

示例 9-2-2　Head 从 Eye、Nose、Mouth、Ear 派生而成

第10章 内存管理

10.1 内存分配方式

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。

10.2 常见的内存错误及其对策

发生内存错误是件非常麻烦的事情。编译器不能自动发现这些错误，通常是在程序运行时才能捕捉到。而这些错误大多没有明显的症状，时隐时现，增加了改错的难度。有时用户怒气冲冲地把你找来，程序却没有发生任何问题，你一走，错误又发作了。

（1） 常见的内存错误及其对策如下：

（a） 内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为 NULL。如果指针 p 是函数的参数，那么在函数的入口处用 assert(p!=NULL)进行检查。如果是用 malloc 或 new 来申请内存，应该用 if(p==NULL) 或 if(p!=NULL)进行防错处理。

（b） 内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。

内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

（c） 内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多 1”或者“少 1”的操作。特别是在 for 循环语句中，循环次数很容易搞错，导致数组操作越界。

（d） 忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然死掉，系统出现提示：内存耗尽。

动态内存的申请与释放必须配对，程序中 malloc 与 free 的使用次数一定要相同，否则肯定有错误（new/delete 同理）。

（e） 释放了内存却继续使用它。

有三种情况：

- ✧ 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。
- ✧ 函数的 return 语句写错了，注意不要返回指向“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。
- ✧ 使用 free 或 delete 释放了内存后，没有将指针设置为 NULL。导致产生“野指针”。

（2） 内存管理规则：

（a） 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。

（b） 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。

（c） 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。

（d） 动态内存的申请与释放必须配对，防止内存泄漏。

（e） 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”。

10.3 指针与数组的对比

C++/C 程序中，指针和数组在不少地方可以相互替换着用，让人产生一种错觉，以为两者是等价的。

（1） 数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。数组名对应着（而不是指向）一块内存，其地址与容量在生命期内保持不变，只有数组的内容可以改变。

（2） 指针可以随时指向任意类型的内存块，它的特征是“可变”，所以我们常用指针来操作动态内存。指针远比数组灵活，但也更危险。

下面以字符串为例比较指针与数组的特性。

（3） 修改内容

示例 10-3-1 中，字符数组 cArray 的容量是 6 个字符，其内容为 hello\0。cArray 的内容可以改变，如 cArray [0]= ‘X’。指针 pChar 指向常量字符串“world”（位于静态存储区，内容为 world\0），常量字符串的内容是不可以被修改的。从语法上看，编译器并不觉得语句 pChar [0]= ‘X’有什么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。

```
char acArray [] = “hello”;
acArray [0] = ‘X’;
cout << acArray << endl;
char *pChar = “world”;        // 注意 pChar 指向常量字符串
pChar [0] = ‘X’;              // 编译器不能发现该错误
cout << pChar << endl;
```

示例 10-3-1 修改数组和指针的内容

（4） 内容复制与比较

不能对数组名进行直接复制与比较。示例 10-3-2 中，若想把数组 aOne 的内容复制给数组 aOther，不能用语句 aOther = aOne ，否则将产生编译错误。应该用标准库函数 CSStringCbCopy 进行复制。同理，比较 aOther 和 aOne 的内容是否相同，不能用 if(aOther==aOne) 来判断，应该用标准库函数 strcmp 进行比较。

语句 pTmp = aOne 并不能把 aOne 的内容复制指针 pTmp，而是把 aOne 的地址赋给了 pTmp。要想复制 aOne 的内容，可以先用库函数 malloc 为 pTmp 申请一块容量为 CSStringCbLength(aOne)+1 个字符的内存，再用 CSStringCbCopy 进行字符串复制。同理，语句 if(pTmp==aOne) 比较的不是内容而是地址，应该用库函数 strcmp 来比较。

```
int nLength;
unsigned int u16bDest;
unsigned int u16MaxBytesNum;
// 数组…
char aOne[] = "hello";
char aOther[10];
u16MaxBytesNum = (unsigned int)sizeof(aOne) + 1;
u16bDest = (unsigned int)sizeof(aOther) + 1;
CSStringCbCopy(aOther, u16bDest, aOne); // 不能用      aOther = aOne;
if (strcmp(aOther, aOne) == 0)                  // 不能用   if (aOther == aOne)
{
```

```
        …Program Code
    }
```

```
    // 指针…
    if (CSStringCbLength(aOne, u16MaxBytesNum, pu16RealCnt) == S_OK)
        {
            nLength = *pu16RealCnt;
            u16bDest = nLength + 1;
            pTmp = new char[u16bDest];
            CSStringCbCopy(pTmp, u16bDest, aOne);
        }
    if (strcmp(pTmp, aOne) == 0)      // 不要用 if (pTmp == aOne)
    {
        …Program Code
    }
```

示例 10-3-2 数组和指针的内容复制与比较

（5） 计算内存容量

用运算符 sizeof 可以计算出数组的容量（字节数）。示例 10-3-3（a）中，sizeof(aOne)的值是 12（注意别忘了'\0'）。指针 p 指向 a，但是 sizeof(pTmp)的值却是 4。这是因为 sizeof(pTmp)得到的是一个指针变量的字节数，相当于 sizeof(char*)，而不是 p 所指的内存容量。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。

注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。示例 10-3-3（b）中，不论数组 a 的容量是多少，sizeof(aOne)始终等于 sizeof(char *)。

```
    char acOne[] = "hello world";
    char *pcTmp  = acOne;
    cout<< sizeof(acOne) << endl;     // 12 字节
    cout<< sizeof(pcTmp) << endl;     // 4 字节
```

示例 10-3-3（a） 计算数组和指针的内存容量

```
    void Func(char aOne[100])
```

```
{
    cout<< sizeof(aOne) << endl;     // 4 字节而不是 100 字节
}
```

示例 10-3-3（b） 数组退化为指针

10.4 指针参数传递内存注意事项

如果函数的参数是一个指针，不要指望用该指针去申请动态内存。示例 10-4-1 中，Test 函数的语句 GetMemory(str, 200)并没有使 str 获得期望的内存，str 依旧是 NULL

```
void GetMemory(char *pcMemo, int nNum)
{
    pcMemo = (char *)malloc(sizeof(char) * nNum);
}
```

```
void Test(void)
{
    Unsigned int u16Constant;
    Unsigned char cbDest;
    u16Constant = 100;
    char *pcStr = NULL;
    GetMemory(pcStr, u16Constant);  // pcStr 仍然为 NULL
    cbDest = u16Constant;
    CSStringCbCopy(pcStr, cbDest, "hello");     // 运行错误
}
```

示例 10-4-1 试图用指针参数申请动态内存

毛病出在函数 GetMemory 中。编译器总是要为函数的每个参数制作临时副本，指针参数 *pcMemo* 的副本是 *_pcMemo*，编译器使 *_pcMemo = pcMemo*。如果函数体内的程序修改了*_ pcMemo* 的内容，就导致参数 *pcMemo* 的内容作相应的修改。这就是指针可以用作输出参数的原因。在本例中，*_pcMemo* 申请了新的内存，只是把*_ pcMemo* 所指的内存地址改变了，但是 *pcMemo* 丝毫未变。所以函数 GetMemory 并不能输出任何东西。事实上，每执行一次 GetMemory 就会泄露一块内存，因为没有用 free 释放内存。

如果非得要用指针参数去申请内存，那么应该改用“指向指针的指针”，见示例 10-4-2。

```
void GetMemory2(char ** pcMemo, int nNum)
{
    * pcMemo = (char *)malloc(sizeof(char) * nNum);
}
```

```
void Test2(void)
{
    Unsigned int u16Constant;
    Unsigned char cbDest;
    u16Constant = 100;
    char *pcStr = NULL;
    GetMemory2(&pcStr, u16Constant);// 注意参数是 &pcStr，而不是 pcStr
    cbDest = u16Constant;
    CSStringCbCopy(pcStr, cbDest, "hello");
    cout<< pcStr << endl;
    free(pcStr);
}
```

示例 10-4-2 用指向指针的指针申请动态内存

由于“指向指针的指针”这个概念不容易理解，我们可以用函数返回值来传递动态内存。这种方法更加简单，见示例 10-4-3。

```
char *GetMemory3(int nNum)
{
    char *pcMemo = (char *)malloc(sizeof(char) * nNum);
    return pcMemo;
}
```

```
void Test3(void)
{
    Unsigned int u16Constant;
```

```
    Unsigned char cbDest;
    u16Constant = 100;
    char *pcStr = NULL;
    pcStr = GetMemory3(u16Constant);
    cbDest = u16Constant;
    CSStringCbCopy(pcStr, cbDest, "hello");
    cout<< pcStr << endl;
    free(pcStr);
}
```

示例 10-4-3 用函数返回值来传递动态内存

用函数返回值来传递动态内存这种方法虽然好用，但是常常有人把 return 语句用错了。这里强调不要用 return 语句返回指向“栈内存”的指针，因为该内存在函数结束时自动消亡，见示例 10-4-4。

```
char *GetString(void)
{
    char pcMemo[] = "hello world";
    return pcMemo;     // 编译器将提出警告
}
```

```
void Test4(void)
{
    char *pcStr = NULL;
    pcStr = GetString();      // pcStr 的内容是垃圾
    cout<< pcStr << endl;
}
```

示例 10-4-4 return 语句返回指向“栈内存”的指针

用调试器逐步跟踪 test4，发现执行 pcStr = GetString 语句后 pcStr 不再是 NULL 指针，但是 pcStr 的内容不是“hello world”而是垃圾。

如果把示例 10-4-4 改写成示例 10-4-5，则：

```
char *GetString2(void)
```

```
{
    char *pcMemo = "hello world";
    return pcMemo ;
}
```

```
void Test5(void)
{
    char *pcStr = NULL;
    pcStr = GetString2();
    cout<< pcStr << endl;
}
```

示例 10-4-5 return 语句返回常量字符串

函数 test5 运行虽然不会出错，但是函数 getString2 的设计概念却是错误的。因为 getString2 内的“hello world”是常量字符串，位于静态存储区，它在程序生命期内恒定不变。无论什么时候调用 getString2，它返回的始终是同一个“只读”的内存块。

10.5 free 和 delete 指针

别看 free 和 delete 的名字恶狠狠的（尤其是 delete），它们只是把指针所指的内存给释放掉，但并没有把指针本身干掉。

用调试器跟踪示例 10-5，发现指针 pcTest 被 free 以后其地址仍然不变（非 NULL），只是该地址对应的内存是垃圾，pcTest 成了“野指针”。如果此时不把 pcTest 设置为 NULL，会让人误以为 pcTest 是个合法的指针。

如果程序比较长，我们有时记不住 pcTest 所指的内存是否已经被释放，在继续使用 pcTest 之前，通常会用语句 if (p != NULL)进行防错处理。很遗憾，此时 if 语句起不到防错作用，因为即便 p 不是 NULL 指针，它也不指向合法的内存块。

```
Unsigned int u16Constant;
Unsigned char cbDest;
u16Constant = 100;
char *pcTest = (char *) malloc(u16Constant);
cbDest = u16Constant;
CSStringCbCopy(pcTest, cbDest, “hello”);
```

```
free(pcTest);          // p 所指的内存被释放，但是 p 所指的地址仍然不变
…
if(pcTest != NULL) // 没有起到防错作用
{
    CSStringCbCopy(pcTest, cbDest, “world”); // 出错
}
```

示例 10-5　pcTest 成为野指针

10.6 动态内存不会被自动释放

函数体内的局部变量在函数结束时自动消亡。很多人误以为示例 10-6 是正确的。理由是 pcTest 是局部的指针变量，它消亡的时候会让它所指的动态内存一起完蛋。这是错觉！

```
void Func(void)
{
    char *pcTest = (char *) malloc(100);    // 动态内存不会自动释放
}
```

示例 10-6 试图让动态内存自动释放

指针有一些“似是而非”的特征：

（1） 指针消亡了，并不表示它所指的内存会被自动释放。

（2） 内存被释放了，并不表示指针会消亡或者成了 NULL 指针。

这表明释放内存并不是一件可以草率对待的事。

10.7 杜绝指针的非法操作

“野指针”不是 NULL 指针，是指向“垃圾”内存的指针。人们一般不会错用 NULL 指针，因为用 if 语句很容易判断。但是“野指针”是很危险的，if 语句对它不起作用。

“野指针”的成因主要有两种：

（a） 指针变量没有被初始化。任何指针变量刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为 NULL，要么让它指向合法的内存。例如

```
char *pcTest = NULL;
```

```
char *pcStr = (char *) malloc(100);
```

指针 pcTest 被 free 或者 delete 之后，没有置为 NULL，让人误以为 pcTest 是个合法的指针。

（b） 指针操作超越了变量的作用范围。这种情况让人防不胜防，示例程序如下：

```
class Person
{
    public:
        void Func(void)
        {
            cout << “Func of class Person” << endl;
        }
};


void Test(void)
{
    Person  *pTmp;
        {
            Person  kaolin;
            PTmp = &kaolin;  // 注意 kaolin 的生命期
        }
    pTmp->Func();    // pTmp 是“野指针”
}
```

函数 Test 在执行语句 pTmp->Func()时，对象 kaolin 已经消失，而 pTmp 是指向 kaolin 的，所以 pTmp 就成了“野指针”。

10.8 malloc/free 与 new/delete

malloc 与 free 是 C++/C 语言的标准库函数，new/delete 是 C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用 maloc/free 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。

因此 C++语言需要一个能完成动态内存分配和初始化工作的运算符 new，以及一个能完成清理与释放内存工作的运算符 delete。注意 new/delete 不是库函数。

我们先看一看 malloc/free 和 new/delete 如何实现对象的动态内存管理，见示例 10-8。

```
class Obj
{
    public :
        Obj(void)
        {
            cout << “Initialization” << endl;
        }
        ~Obj(void)
        {
            cout << “Destroy” << endl;
        }
        void Initialize(void)
        {
            cout << “Initialization” << endl;
        }
        void    Destroy(void)
        {
            cout << “Destroy” << endl;
        }
};
```

```
void UseMallocFree(void)
{
    Obj    *pObj = (obj *)malloc(sizeof(obj));    // 申请动态内存
    pObj ->Initialize();                          // 初始化
    //…
    pObj ->Destroy();  // 清除工作
```

```
    free(pObj);          // 释放内存
}
```

```
void UseNewDelete(void)
{
    Obj   * pObj = new Obj;      // 申请动态内存并且初始化
    //…
    delete pObj;                 // 清除并且释放内存
}
```

示例 10-8 用 malloc/free 和 new/delete 如何实现对象的动态内存管理

类 Obj 的函数 Initialize 模拟了构造函数的功能，函数 Destroy 模拟了析构函数的功能。函数 UseMallocFree 中，由于 malloc/free 不能执行构造函数与析构函数，必须调用成员函数 Initialize 和 Destroy 来完成初始化与清除工作。函数 useNewDelete 则简单得多。

所以我们不要企图用 malloc/free 来完成动态对象的内存管理，应该用 new/delete。由于内部数据类型的“对象”没有构造与析构的过程，对它们而言 malloc/free 和 new/delete 是等价的。

虽然 new/delete 的功能完全覆盖了 malloc/free，由于 C++程序经常要调用 C 函数，而 C 程序只能用 malloc/free 管理动态内存，所以 malloc/free 不能被 new/delete 完全取代。

如果用 free 释放“new 创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用 delete 释放“malloc 申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差。所以 new/delete 必须配对使用，malloc/free 也一样。

10.9 内存耗尽的处理方式

如果在申请动态内存时找不到足够大的内存块，malloc 和 new 将返回 NULL 指针，宣告内存申请失败。通常有三种方式处理“内存耗尽”问题。

（1） 判断指针是否为 NULL，如果是则马上用 return 语句终止本函数。例如：

```
void Func(void)
{
    Obj  *pObjA = new Obj;
    if(pObjA == NULL)
    {
        return;
    }
```

```
    …
}
```

（2） 判断指针是否为 NULL，如果是则马上用 exit(1)终止整个程序的运行。例如：

```
void Func(void)
{
    Obj  * pObjA = new Obj;
    if(pObjA == NULL)
    {
        cout << “Memory Exhausted” << endl;
        exit(1);
    }
    …
}
```

（3） 为 new 和 malloc 设置异常处理函数。例如 Visual C++可以用_set_new_hander 函数为 new 设置用户自己定义的异常处理函数，也可以让 malloc 享用与 new 相同的异常处理函数。详细内容请参考 C++使用手册。

上述（1）（2）方式使用最普遍。如果一个函数内有多处需要申请动态内存，那么方式（1）就显得力不从心（释放内存很麻烦），应该用方式（2）来处理。

很多人不忍心用 exit(1)，问：“不编写出错处理程序，让操作系统自己解决行不行？”

不行。如果发生“内存耗尽”这样的事情，一般说来应用程序已经无药可救。如果不用 exit(1) 把坏程序杀死，它可能会害死操作系统。道理如同：如果不把歹徒击毙，歹徒在老死之前会犯下更多的罪。

10.10 malloc/free 的使用要点

函数 malloc 的原型如下：

```
void * malloc(size_t size);
```

用 malloc 申请一块长度为 length 的整数类型的内存，程序如下：

```
int  *pTmp = (int *) malloc(sizeof(int) * nLength);
```

我们应当把注意力集中在两个要素上：“类型转换”和“sizeof”。

（1） malloc 返回值的类型是 void *，所以在调用 malloc 时要显式地进行类型转换，将 void * 转换成所需要的指针类型。

malloc 函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。我们通

常记不住 int, float 等数据类型的变量的确切字节数。例如 int 变量在 16 位系统下是 2 个字节，在 32 位下是 4 个字节；而 float 变量在 16 位系统下是 4 个字节，在 32 位下也是 4 个字节。最好用以下程序作一次测试：

```
cout << sizeof(char) << endl;
cout << sizeof(int) << endl;
cout << sizeof(unsigned int) << endl;
cout << sizeof(long) << endl;
cout << sizeof(unsigned long) << endl;
cout << sizeof(float) << endl;
cout << sizeof(double) << endl;
cout << sizeof(void *) << endl;
```

在 malloc 的“()”中使用 sizeof 运算符是良好的风格，但要当心有时我们会昏了头，写出 pTmp = malloc(sizeof(pTmp))这样的程序来。

函数 free 的原型如下：

```
void free( void * memblock );
```

为什么 free 函数不象 malloc 函数那样复杂呢？这是因为指针 p 的类型以及它所指的内存的容量事先都是知道的，语句 free(p)能正确地释放内存。如果 p 是 NULL 指针，那么 free 对 p 无论操作多少次都不会出问题。如果 p 不是 NULL 指针，那么 free 对 p 连续操作两次就会导致程序运行错误。

10.11 new/delete 的使用要点

运算符 new 使用起来要比函数 malloc 简单得多，例如：

```
int  *pTest = (int *)malloc(sizeof(int) * nLength);
int  *pTest = new int[nLength];
```

这是因为 new 内置了 sizeof、类型转换和类型安全检查功能。对于非内部数据类型的对象而言，new 在创建动态对象的同时完成了初始化工作。如果对象有多个构造函数，那么 new 的语句也可以有多种形式。例如

```
class Obj
{
   public :
      Obj(void);       // 无参数的构造函数
```

```
        Obj(int nPara);      // 带一个参数的构造函数
        …
}
void Test(void)
{
    Obj  *pObjA = new Obj;
    Obj  *pObjB = new Obj(1);  // 初值为 1
    …
    delete pObjA;
    delete pObjB;
}
```

如果用 new 创建对象数组，那么只能使用对象的无参数构造函数。例如

```
Obj  *objects = new Obj[100]; // 创建 100 个动态对象
```

不能写成

*Obj *objects = new Obj[100](1);// 创建 100 个动态对象的同时赋初值 1*

在用 delete 释放对象数组时，留意不要丢了符号‘[]’。例如

delete []objects; // 正确的用法

delete objects; // 错误的用法

后者相当于 delete objects[0]，漏掉了另外 99 个对象。

第11章 可测试性

11.1 测试准备

（1） 编写代码前应设计好程序调试与测试的方法和手段

（a） 在编写代码之前，应预先设计好程序调试与测试的方法和手段，并设计好各种调测开关及相应测试代码如打印函数等。

（b） 程序的调试与测试是软件生存周期中很重要的一个阶段，如何对软件进行较全面、高效率的测试并尽可能地找出软件中的错误就成为很关键的问题。因此在编写源代码之前，除了要有一套比较完善的测试计划外，还应设计出一系列代码测试手段，为单元测试、集成测试及系统联调提供方便。

（2） 编程的同时要为单元测试选择恰当的测试点

编程的同时要为单元测试选择恰当的测试点，并仔细构造测试代码、测试用例，同时给出明确的注释说明。测试代码部分应作为（模块中的）一个子模块，以方便测试代码在模块中的安装与拆卸（通过调测开关）。

（3） 集成测试所需测试准备

在进行集成测试/系统联调之前，要构造好测试环境、测试项目及测试用例，同时仔细分析并优化测试用例，以提高测试效率。好的测试用例应尽可能模拟出程序所遇到的边界值、各种复杂环境及一些极端情况等。

11.2 联调调测开关及调测信息串

（1） 要有一套统一的联调调测开关及打印函数

要有一套统一的为集成测试与系统联调准备的调测开关及相应打印函数，调测打印出的信息串的格式要有统一的形式，并且要有详细的说明。

（2） 调测信息串中至少要有所在模块名及行号

调测打印出的信息串的格式要有统一的形式。信息串中至少要有所在模块名（或源文件名）及行号。统一的调测信息格式便于集成测试。

（3） 调测开关应分为不同级别和类型

调测开关的设置及分类应从以下几方面考虑：针对模块或系统某部分代码的调测；针对模块或系统某功能的调测；出于某种其它目的，如对性能、容量等的测试。这样做便于软件功能的调测，并且便于模块的单元测试、系统联调等。

11.3 断言

断言是一个定义正确执行的必要条件的布尔表达式

（1） 断言的一般用途包括：

（a） 检查特定于实现的假设；

（b） 检查在方法的入口必须为真的条件（前置条件）；

（c） 检查在方法的出口必须为真的条件（后置条件）；

（d） 检查在任何时候对于一对象必须为真的条件（不变式）；

（2） 基于实现的断言

（a） 假设检查器：

- ✧ 断言假设是正确执行的必要条件，它应反映正常发生的、具有代表性的条件和期望，断言不应当用于输入检查或异常处理；
- ✧ 若果一个变量必须临时违背入口-出口约束，那么应当卡在改变化应当恢复原状的那一点之后使用外部约束来断言；
- ✧ 如果代码需要过长的解释，可以考虑将这个解释形式化为断言。

（b） 潜伏的故障和探测器：

- ✧ 输入域的大小与输出域的大小相比很大；
- ✧ 依赖于由 5 个或更多个变量定义的边界条件的手段；
- ✧ 依赖于不熟悉的超类中的方法的结果或方法的副作用的段；
- ✧ 依赖于全局可见变量的段。

（3） 基于责任的断言：

（a） 责任、契约和子类型

- ✧ 契约隐喻 断言能够提供公共类的责任和有关实现的内嵌测试；
- ✧ 类契约的元素是方法和类范围中的断言、继承的断言和异常；
- ✧ 断言的强度和弱度 较强的断言是更多的限制，弱断言更普遍

（b） 方法范围

- ✧ 前置条件的断言
- ✧ 对正确循环控制的条件进行断言
- ✧ 后置条件的断言

（c） 类范围

类不变式说明了一个类的每个对象都必须为真的特性，因此类不变式实例化后在每个方法的入口和出口、析构之前必须为真，对所有的方法、活动顺序和所有输入自变量的和法值均必须成立，即需要断言来处理。

（d） 顺序约束

- ✧ 状态不变式
- ✧ 接受条件和结果条件
- ✧ 运行顺序检查

（e） 超类/子类范围

- ✧ 子契约和类型替换
- ✧ 错误的继承

（4） 实现

断言动作：当出现断言相悖时，必须产生断言动作，该动作有两个责任：通告和继续。处理方式为：

（a） 将相悖看作输入错误并继续处理；

（b） 标识相悖，生成诊断程序转储，并终止进程；

（c） 利用相悖触发交互调试；

（d） 标识相悖为成功的测试实例，然后继续处理；

（e） 生成错误消息并终止这个进程；

（f） 将相悖看作异常，然后试着恢复

（5） 限制和告诫

（a） 不是对所有系统都可行；

（b） 无法排除的测试；

（c） 断言不能检查出所有的错误；

（d） 覆盖分析的曲解；

（e） 与执行空间和时间相关的代价和副作用；

（f） 人的因素，断言必须有人来确定和编程，因此断言受社会的、心理的因素制约。

11.4 软件的 DEBUG 版和正式版

（1） 正式软件产品中应把断言及其它调测代码去掉

正式软件产品中应把断言及其它调测代码去掉（即把有关的调测开关关掉），以加快软

件运行速度。

（2） 设置与取消测试代码的软件，在功能行为上应一致。

即在软件系统中设置与取消有关测试手段，不能对软件实现的功能等产生影响。

（3） 用调测开关来切换软件的 DEBUG 版和正式版

用调测开关来切换软件的 DEBUG 版和正式版，而不要同时存在正式版本和 DEBUG 版本的不同源文件，以减少维护的难度。

（4） 软件的 DEBUG 版本和发行版本应该统一维护

软件的 DEBUG 版本和发行版本应该统一维护，不允许分家，并且要时刻注意保证两个版本在实现功能上的一致性。

第12章 C++异常处理

12.1 异常的种类：

C++异常分为两大类，一类是系统产生的，比如 new 申请内存失败（系统资源不足）；一类是编程者自己产生的，即自定义异常。

12.2 自定义异常的处理方法

对于自定义异常，一定要提供异常处理函数，且要就地予以解决（尽量不要扩散）；不得不扩散时，一定要有文字说明，并能在第一时间告知使用者。

（1） 正确注释代码的异常处理能力，以类或函数为单位，注明下列三种情况；

（a） 不扩散

（b） 扩散

（c） 不知道

（2） 减少不必要的异常处理；

（3） 确保异常发生后资源还能被回收；

（4） 抛出的异常最好是一个对象；

异常的类型代表错误的类型，并且应是一个只被异常处理机制使用的类。

（5） 捕捉异常的方法：

（a） 捕捉异常遵照从里到外的顺序，即先派生类后基类；

（b） 捕捉异常时使用引用；

（6） 重要异常处理应写入日志文件中。

第13章 字符串函数

13.1 目的

在许多涉及缓冲溢出的安全问题中，主要是老的缓冲处理函数的原因。在 CSStrsafe.h 中定义的函数，为代码正确地处理缓冲提供了一种方法。如果使用这些 API 函数，则在处理字符串时将获得更安全的代码。

13.2 API 函数列表

（1） 使用字符计数的函数如下：

Index	Name	Description
1	CSStringCchCat	
2	CSStringCchCatEx	
3	CSStringCchCatN	
4	CSStringCchCatNEx	
5	CSStringCchCopy	
6	CSStringCchCopyEx	
7	CSStringCchCopyN	
8	CSStringCchCopyNEx	
9	CSStringCchGets	
10	CSStringCchGetsEx	
11	CSStringCchPrintf	
12	CSStringCchPrintfEx	
13	CSStringCchVPrintf	

14	CSStringCchVPrintfEx	
15	CSStringCchLength	

（2） 使用字节计数的函数如下：

Index	Name	Description
1	CSStringCbCat	
2	CSStringCbCatEx	
3	CSStringCbCatN	
4	CSStringCbCatNEx	
5	CSStringCbCopy	
6	CSStringCbCopyEx	
7	CSStringCbCopyN	
8	CSStringCbCopyNEx	
9	CSStringCbGets	
10	CSStringCbGetsEx	
11	CSStringCbPrintf	
12	CSStringCbPrintfEx	
13	CSStringCbVPrintf	
14	CSStringCbVPrintfEx	
15	CSStringCbLength	

（3） 新老函数对照表（老函数是禁用的）

序号	新的 CSStrSafe API 函数名	对应老的 API 函数名
1.	CSStringCchCat	strcat
2.	CSStringCchCatEx	strcat

3.	CSStringCchCatN	strncat
4.	CSStringCchCatNEx	strncat
5.	CSStringCchCopy	strcpy
6.	CSStringCchCopyEx	strcpy
7.	CSStringCchCopy	strncpy
8.	CSStringCchCopyNEx	strncpy
9.	CSStringCchGets	gets
10.	CSStringCchGetsEx	gets
11.	CSStringCchPrintf	sprintf
12.	CSStringCchPrintfEx	sprintf
13.	CSStringCchVPrintf	vsprintf
14.	CSStringCchVPrintfEx	vsprintf
15.	CSStringCchLength	strlen
16.	CSStringCbCat	strcat
17.	CSStringCbCatEx	strcat
18.	CSStringCbCatN	strncat
19.	CSStringCbCatNEx	strncat
20.	CSStringCbCopy	strcpy
21.	CSStringCbCopyEx	strcpy
22.	CSStringCbCopyN	strncpy
23.	CSStringCbCopyNEx	strncpy
24.	CSStringCbGets	gets
25.	CSStringCbGetsEx	gets
26.	CSStringCbPrintf	sprintf
27.	CSStringCbPrintfEx	sprintf
28.	CSStringCbVPrintf	vsprintf
29.	CSStringCbVPrintfEx	vsprintf
30.	CSStringCbLength	strlen