高质量 C++/C
编程指南及编码规范

计算机科学与技术系

2003/6

前 言

软件质量是被大多数程序员挂在嘴上而不是放在心上的东西！

一、编程老手与高手的误区

现在国内 IT 企业拥有学士、硕士、博士文凭的软件开发人员比比皆是，但他们在接受大学教育时就“先天不足”，岂能一到企业就突然实现质的飞跃。试问有多少软件开发人员对正确性、健壮性、可靠性、效率、易用性、可读性（可理解性）、可扩展性、可复用性、兼容性、可移植性等质量属性了如指掌？并且能在实践中运用自如？“高质量”可不是干活小心点就能实现的！

我们有充分的理由疑虑：

（1）编程老手可能会长期用隐含错误的方式编程（习惯成自然），发现问题后都不愿相信那是真的！

（2）编程高手可以在某一领域写出极有水平的代码，但未必能从全局把握软件质量的方方面面。

老手≠高手

事实证明如此!

编程老手和编程高手的定义：

定义 1：能长期稳定地编写出高质量程序的程序员称为编程老手。

定义 2：能长期稳定地编写出高难度、高质量程序的程序员称为编程高手。

二、内容简介

本书共分两部分，第一部分是高质量 C++/C 编程指南，第二部分是编码规范。其中这两部分之间有一些是互相重叠，但侧重点不同，您可根据需要和实际情况进行选择。

第一部分的第一章至第六章主要论述 C++/C 编程风格。难度不高，但是细节比较多。提高质量就是要从这些点点滴滴做起。世上不存在最好的编程风格，一切因需求而定。团队开发讲究风格一致，如果制定了大家认可的编程风格，那么所有组员都要遵守。

第七章至第十一章是专题论述，技术难度较高，要积极思考。特别是第七章“内存管理”。

第二部分则从命名原则、类型的使用、函数等不同专题进行分析讨论使用 C/C++ 语言的编码规范。

第一部分 高质量 C++/C 编程指南

第 1 章 文件结构

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

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

1.1 版权和版本的声明

版权和版本的声明位于头文件和定义文件的开头（参见示例 1-1），主要内容有：

（1）版权信息。

（2）文件名称，标识符，摘要。

（3）当前版本号，作者/修改者，完成日期。

（4）版本历史信息。

```
/*
* Copyright (c) 2001,XXXX有限公司XXXX部
* 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
*
* 当前版本：1.1
* 作　　者：输入作者（或修改者）名字
* 完成日期：2001年7月20日
*
* 取代版本：1.0
* 原作者　：输入原作者（或修改者）名字
* 完成日期：2001年5月10日
```

```
*/
```

示例1-1 版权和版本的声明

1.2 头文件的结构

头文件由三部分内容组成：

（1）头文件开头处的版权和版本声明（参见示例 1-1）。

（2）预处理块。

（3）函数和类结构声明等。

假设头文件名称为 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”

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 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 头文件的作用

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

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

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

1.5 目录结构

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

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

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

第 2 章 程序的版式

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

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

2.1 空行

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

- **【规则 2-1-1】**在每个类声明之后、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。

示例 2-2（a）为风格良好的代码行，示例 2-2（b）为风格不良的代码行。

```
int width;   // 宽度
int height;  // 高度
int depth;   // 深度
```

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

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

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

示例 2-2(a) **风格良好**的代码行

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

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

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

```
for (initialization; condition; update)
     dosomething();
other();
```

示例 2-2(b) **风格不良**的代码行

✧ 【建议 2-2-1】尽可能在定义变量的同时初始化该变量（就近原则）

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

```
int width = 10;    // 定义并初始化 width
int height = 10;   // 定义并初始化 height
```

```
int depth = 10;    // 定义并初始化 depth
```

2.3 代码行内的空格

- 【规则 2-3-1】关键字之后要留空格。象 const、virtual、inline、case 等关键字之后至少要留一个空格，否则无法辨析关键字。象 if、for、while 等关键字之后应留一个空格再跟左括号‘(’，以突出关键字。
- 【规则 2-3-2】函数名之后不要留空格，紧跟左括号‘(’，以与关键字区别。
- 【规则 2-3-3】‘(’向后紧跟，‘)’、‘,’、‘;’向前紧跟，紧跟处不留空格。
- 【规则 2-3-4】‘,’之后要留空格，如 Function(x, y, z)。如果‘;’不是一行的结束符号，其后要留空格，如 for (initialization; condition; update)。
- 【规则 2-3-5】赋值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如“=”、“+=” “>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<”,“^”等二元操作符的前后应当加空格。
- 【规则 2-3-6】一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。
- 【规则 2-3-7】象“[]”、“.”、“->”这类操作符前后不加空格。
- ✧ 【建议 2-3-1】对于表达式比较长的 for 语句和 if 语句，为了紧凑起见可以适当地去掉一些空格，如 for (i=0; i<10; i++)和 if ((a<=b) && (c<=d))

void Func1(int x, int y, int z);	// 良好的风格
void Func1 (int x,int y,int z);	// 不良的风格
if (year >= 2000)	// 良好的风格
if(year>=2000)	// 不良的风格
if ((a>=b) && (c<=d))	// 良好的风格
if(a>=b&&c<=d)	// 不良的风格
for (i=0; i<10; i++)	// 良好的风格
for(i=0;i<10;i++)	// 不良的风格
for (i = 0; i < 10; i ++)	// 过多的空格
x = a < b ? a : b;	// 良好的风格
x=a<b?a:b;	// 不好的风格
int *x = &y;	// 良好的风格
int * x = & y;	// 不良的风格

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

示例 2-3 代码行内的空格

2.4 对齐

- 【规则 2-4-1】程序的分界符‘{’和‘}’应独占一行并且位于同一列，同时与引用它们的语句左对齐。
- 【规则 2-4-2】{ }之内的代码块在‘{’右边数格处左对齐。

示例 2-4（a）为风格良好的对齐，示例 2-4（b）为风格不良的对齐。

void Function(int x)
{
  … // 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，否则眼睛看不过来，也不便于打印。
- **【规则 2-5-2】**长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

```
if ((very_longer_variable1 >= very_longer_variable12)
    && (very_longer_variable3 <= very_longer_variable14)
    && (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

修饰符 * 和 & 应该靠近数据类型还是该靠近变量名，是个有争议的活题。

若将修饰符 * 靠近数据类型，例如：int* x; 从语义上讲此写法比较直观，即 x 是 int 类型的指针。

上述写法的弊端是容易引起误解，例如：int* x, y; 此处 y 容易被误解为指针变量。虽然将 x 和 y 分行定义可以避免误解，但并不是人人都愿意这样做。

- **【规则 2-6-1】**应当将修饰符 * 和 & 紧靠变量名

  例如：

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

2.7 注释

C 语言的注释符为“/*…*/”。C++语言中，程序块的注释常采用“/*…*/”，行注释一般采用“//…”。注释通常用于：

（1）版本、版权声明；

（2）函数接口说明；

（3）重要的代码行或段落提示。

虽然注释有助于理解代码，但注意不可过多地使用注释。参见示例 2-6。

- **【规则 2-7-1】**注释是对代码的“提示”，而不是文档。程序中的注释不可喧宾夺主，注释太多了会让人眼花缭乱。注释的花样要少。
- **【规则 2-7-2】**如果代码本来就是清楚的，则不必加注释。否则多此一举，令人厌烦。例如

```
i++;    // i 加 1，多余的注释
```

- **【规则 2-7-3】**边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。
- **【规则 2-7-4】**注释应当准确、易懂，防止注释有二义性。错误的注释不但无益反而有害。
- **【规则 2-7-5】**尽量避免在注释中使用缩写，特别是不常用缩写。
- **【规则 2-7-6】**注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。
- **【规则 2-7-8】**当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处

加注释，便于阅读。

```
/*
* 函数介绍：
* 输入参数：
* 输出参数：
* 返回值  ：
*/
void Function(float x, float y, float z)
{
   …
}
```

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

示例 2-6 程序的注释

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。我们不可以滥用类的封装功能，不要把它当成火锅，什么东西都往里扔。

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 2-7（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 2-7（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）。

很多 C++教课书受到 Biarne Stroustrup 第一本著作的影响，不知不觉地采用了“以数据为中心”的书写方式，并不见得有多少道理。

建议读者采用“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。这是很多人的经验——“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。因为用户最关心的是接口，谁愿意先看到一堆私有数据成员！”

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

示例 2-7(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 2-7(b) 以行为为中心的版式

第 3 章 命名规则

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

“匈牙利”法最大的缺点是烦琐，例如

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

倘若采用“匈牙利”命名规则，则应当写成

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

如此烦琐的程序会让绝大多数程序员无法忍受。

据考察，没有一种命名规则可以让所有的程序员赞同，程序设计教科书一般都不指定命名规则。命名规则对软件产品而言并不是“成败悠关”的事，不要花费太多精力试图发明世界上最好的命名规则，而应当制定一种令大多数项目成员满意的命名规则，并在项目中贯彻实施。

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，应当在遵循这些共性规则的前提下，再扩充特定的规则，如 3.2 节。

- **【规则 3-1-1】**标识符应当直观且可以拼读，可望文知意，不必进行“解码”。

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

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

几十年前老 ANSI C 规定名字不准超过 6 个字符，现今的 C++/C 不再有此限制。一般来说，长名字能更好地表达含义，所以函数名、变量名、类名长达十几个字符不足为怪。那么名字是否越长约好？不见得！例如变量名 maxval 就比 maxValueUntilOverflow 好用。单字符的名字也是有用的，常见的如 i,j,k,m,n,x,y,z 等，它们通常可用作函数内的局部变量。

● 【规则 3-1-3】命名规则尽量与所采用的操作系统或开发工具的风格保持一致。

例如 Windows 应用程序的标识符通常采用“大小写”混排的方式，如 AddChild。而 Unix 应用程序的标识符通常采用“小写加下划线”的方式，如 add_child。别把这两类风格混在一起用。

● 【规则 3-1-4】程序中不要出现仅靠大小写区分的相似的标识符。

例如：

```
int   x,   X;          // 变量 x 与 X 容易混淆
void foo(int x);   // 函数 foo 与 F00 容易混淆
void F00(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;
float    newValue;
```

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

例如：

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

● 【规则 3-1-8】用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

例如：

```
int  minValue;
int  maxValue;

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

✧ 【建议 3-1-1】尽量避免名字中出现数字编号，如 Value1,Value2 等，除非逻辑

上的确需要编号。这是为了防止程序员偷懒，不肯为命名动脑筋而导致产生无意义的名字（因为用数字编号最省事）。

3.2 简单的 Windows 应用程序命名规则

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

- **【规则 3-2-1】**类名和函数名用大写字母开头的单词组合而成。

例如：

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

- **【规则 3-2-2】**变量和参数用小写字母开头的单词组合而成。

例如：

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

- **【规则 3-2-3】**常量全用大写的字母，用下划线分割单词。

例如：

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

- **【规则 3-2-4】**静态变量加前缀 s_（表示 static）。

例如：

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

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

例如：

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

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

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

  例如：

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

- **【规则 3-2-7】**为了防止某一软件库中的一些标识符和其它软件库中的冲突，可以为各种标识符加上能反映软件性质的前缀。例如三维图形标准 OpenGL 的所有库函数均以 gl 开头，所有常量（或宏定义）均以 GL 开头。

第 4 章 表达式和基本语句

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

发觉很多程序员用隐含错误的方式写表达式和基本语句。

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

4.1 运算符的优先级

C++/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**
	= += -= *= /= %= &= ^= |= <<= >>=	**从右至左**

表 4-1 运算符的优先级与结合律

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

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

```
word = (high << 8) | low
if ((a | b) && (a & c))
```

4.2 复合表达式

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

- **【规则 4-2-1】**不要编写太复杂的复合表达式。

  例如：

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

- **【规则 4-2-2】**不要有多用途的复合表达式。

  例如：

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

  该表达式既求 a 值又求 d 值。应该拆分为两个独立的语句：

  ```
  a = b + c;
  d = a + r;
  ```

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

  例如：

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

  并不表示

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

  而是成了令人费解的

  ```
  if ( (a<b)<c )
  ```

4.3 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)
```

4.3.5 对 if 语句的补充说明

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

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

```
if (condition)
    return x;
return y;
```

改写为

```
if (condition)
{
    return x;
}
else
{
    return y;
}
```

或者改写成更加简练的

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

4.4 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。并且由于前者老要进行逻辑判断，打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。如果 N 非常大，最好采用示例 4-4(d)的写法，可以提高效率。如果 N 非常小，两者效率差别并不明显，采用示例 4-4(c)的写法比较好，因为程序更加简洁。

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

表 4-4(c) 效率低但程序简洁

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

表 4-4(d) 效率高但程序不简洁

4.5 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

有了 if 语句为什么还要 switch 语句?

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

switch 语句的基本格式是：

```
switch (variable)
{
    case value1 :  …
                   break;
    case value2 :  …
                   break;
    …
        default :  …
                   break;
}
```

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

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; 例如

```
{ …
    { …
        { …
            goto error;
        }
    }
}
error:
…
```

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

第 5 章 常量

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

5.1 为什么需要常量

如果不使用常量，直接在程序中填写数字或字符串，将会有什么麻烦？

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

（2） 在程序的很多地方输入同样的数字或字符串，难保不发生书写错误。

（3） 如果要修改数字或字符串，则会在很多地方改动，既麻烦又容易出错。

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

  例如：

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

5.2 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。

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

5.3 常量定义规则

- **【规则 5-3-1】**需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。为便于管理，可以把不同模块的常量集中存放在一个公共的头文件中。
- **【规则 5-3-2】**如果某一常量与其它常量密切相关，应在定义中包含这种关系，而不应给出一些孤立的值。

  例如：

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

5.4 类中的常量

有时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 A
{…
    const int SIZE = 100; // 错误，企图在类声明中初始化 const 数据成员
    int array[SIZE];      // 错误，未知的 SIZE
};
```

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 A
{…
    A(int size);      // 构造函数
    const int SIZE ;
};
A::A(int size) : SIZE(size)   // 构造函数的初始化表
{
```

```
    …
}
A  a(100); // 对象 a 的 SIZE 值为 100
A  b(200); // 对象 b 的 SIZE 值为 200
```

怎样才能建立在整个类中都恒定的常量呢？别指望 const 数据成员了，应该用类中的枚举常量来实现。例如

```
class A
{…
    enum { SIZE1 = 100, SIZE2 = 200}; // 枚举常量
    int array1[SIZE1];
    int array2[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）。

第 6 章 函数设计

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

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

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);
```

  那么很难搞清楚究竟是把 str1 拷贝到 str2 中，还是刚好倒过来。

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

  还有一个问题，这两个参数那一个该在前那一个该在后？参数的顺序要遵循程序员的习惯。一般地，应将目的参数放在前面，源参数放在后面。

  如果将函数声明为：

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

  别人在使用时可能会不假思索地写成如下形式：

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

● 【规则 6-1-3】如果参数是指针，且仅作输入用，则应在类型前加 const，以防止该指针在函数体内被意外修改。

例如：

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

● 【规则 6-1-4】如果输入参数以值传递的方式传递对象，则宜改用“const &”方式来传递，这样可以省去临时对象的构造和析构过程，从而提高效率。

✧ 【建议 6-1-1】避免函数有太多的参数，参数个数尽量控制在 5 个以内。如果参数太多，在使用时容易将参数类型或顺序搞错。

✧ 【建议 6-1-2】尽量不要使用类型和数目不确定的参数。

C 标准库函数 printf 是采用不确定参数的典型代表，其原型为：

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

这种风格的函数在编译时丧失了严格的类型安全检查。

6.2 返回值的规则

● 【规则 6-2-1】不要省略返回值的类型。

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

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

● 【规则 6-2-2】函数名字与返回值类型在语义上不可冲突。

违反这条规则的典型代表是 C 标准库函数 getchar。

例如：

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

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

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

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

● **【规则 6-2-3】**不要将正常值和错误标志混在一起返回。正常值用输出参数获得，而错误标志用 return 语句返回。

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

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

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

函数 getchar 可以改写成 BOOL GetChar(char *c);

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

✧ **【建议 6-2-1】**有时候函数原本不需要返回值，但为了增加灵活性如支持链式表达，可以附加返回值。

例如字符串拷贝函数 strcpy 的原型：

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

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

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

✧ **【建议 6-2-2】**如果函数的返回值是一个对象，有些场合用“引用传递”替换“值传递”可以提高效率。而有些场合只能用“值传递”而不能用“引用传递”，否则会出错。

例如：

```
class String
{…
    // 赋值函数
    String & operator=(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   operator+(const String &s1, const String &s2);
```

```
private:
    char *m_data;
}
```

String 的赋值函数 operator = 的实现如下：

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

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

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

String 的相加函数 operator + 的实现如下：

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

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

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

此时 a + b 并不返回期望值，c 什么也得不到，留下了隐患。

6.3 函数内部实现的规则

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

- **【规则 6-3-1】**在函数体的“入口处”，对参数的有效性进行检查。

很多程序错误是由非法参数引起的，应该充分理解并正确使用“断言”（assert）来防止此类错误。详见 6.5 节“使用断言”。

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

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

注意事项如下：

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

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

（2）要搞清楚返回的究竟是“值”、“指针”还是“引用”。

（3）如果函数返回值是一个对象，要考虑 return 语句的效率。例如

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

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

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

实质不然，上述代码将发生三件事。首先，temp 对象被创建，同时完成初始化；然后拷贝构造函数把 temp 拷贝到保存返回值的外部存储单元中；最后，temp 在函数结束时被销毁（调用析构函数）。然而“创建一个临时对象并返回它”的过程是不同的，编译器直接把临时对象创建并初始化在外部存储单元中，省去了拷贝和析构

的化费，提高了效率。

类似地，不要将

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

写成

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

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

6.4 其它建议

✧ **【建议 6-4-1】**函数的功能要单一，不要设计多用途的函数。

✧ **【建议 6-4-2】**函数体的规模要小，尽量控制在 50 行代码之内。

✧ **【建议 6-4-3】**尽量避免函数带有“记忆”功能。相同的输入应当产生相同的输出。

带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态”。这样的函数既不易理解又不利于测试和维护。在 C/C++语言中，函数的 static 局部变量是函数的“记忆”存储器。建议尽量少用 static 局部变量，除非必需。

✧ **【建议 6-4-4】**不仅要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

✧ **【建议 6-4-5】**用于出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。

6.5 使用断言

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

断言 assert 是仅在 Debug 版本起作用的宏，它用于检查“不应该”发生的情况。示例 6-5 是一个内存复制函数。在运行过程中，如果 assert 的参数为假，那么程序就会中止（一般地还会出现提示对话，说明在什么地方引发了 assert）。

```
void  *memcpy(void *pvTo, const void *pvFrom, size_t size)
{
    assert((pvTo != NULL) && (pvFrom != NULL));  // 使用断言
```

```
    byte *pbTo = (byte *) pvTo;        // 防止改变 pvTo 的地址
    byte *pbFrom = (byte *) pvFrom;    // 防止改变 pvFrom 的地址
    while(size -- > 0 )
        *pbTo ++ = *pbFrom ++ ;
    return pvTo;
}
```

示例 6-5 复制不重叠的内存块

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

很少有比跟踪到程序的断言，却不知道该断言的作用更让人沮丧的事了。你花了很多时间，不是为了排除错误，而只是为了弄清楚这个错误到底是什么。有的时候，程序员偶尔还会设计出有错误的断言。所以如果搞不清楚断言检查的是什么，就很难判断错误是出现在程序中，还是出现在断言中。幸运的是这个问题很好解决，只要加上清晰的注释即可。这本是显而易见的事情，可是很少有程序员这样做。这好比一个人在森林里，看到树上钉着一块“危险”的大牌子。但危险到底是什么？树要倒？有废井？有野兽？除非告诉人们“危险”是什么，否则这个警告牌难以起到积极有效的作用。难以理解的断言常常被程序员忽略，甚至被删除。[Maguire, p8-p30]

- **【规则 6-5-1】**使用断言捕捉不应该发生的非法情况。不要混淆非法情况与错误情况之间的区别，后者是必然存在的并且是一定要作出处理的。
- **【规则 6-5-2】在**函数的入口处，使用断言检查参数的有效性（合法性）。
- **【建议 6-5-1】**在编写函数时，要进行反复的考查，并且自问：“我打算做哪些假定？”一旦确定了的假定，就要使用断言对假定进行检查。
- **【建议 6-5-2】**一般教科书都鼓励程序员们进行防错设计，但要记住这种编程风格可能会隐瞒错误。当进行防错设计时，如果“不可能发生”的事情的确发生了，则要使用断言进行报警。

6.6 引用与指针的比较

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

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

n 相当于 m 的别名（绰号），对 n 的任何操作就是对 m 的操作。例如有人名叫王小毛，他的绰号是“三毛”。说“三毛”怎么怎么的，其实就是对王小毛说三道四。所以 n 既不是 m 的拷贝，也不是指向 m 的指针，其实 n 就是 m 它自己。

引用的一些规则如下：

（1）引用被创建的同时必须被初始化（指针则可以在任何时候被初始化）。

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

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

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

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

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

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

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

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

```
void Func2(int *x)
{
    (* x) = (* x) + 10;
}
…
int n = 0;
```

```
Func2(&n);
cout << “n = ” << n << endl;     // n = 10
```

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

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

对比上述三个示例程序，会发现“引用传递”的性质象“指针传递”，而书写方式象“值传递”。实际上“引用”可以做的任何事情“指针”也都能够做，为什么还要“引用”这东西？

答案是“用适当的工具做恰如其分的工作”。

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

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

第 7 章 内存管理

欢迎进入内存这片雷区。伟大的 Bill Gates 曾经失言：

640K ought to be enough for everybody

— Bill Gates 1981

程序员们经常编写内存管理程序，往往提心吊胆。如果不想触雷，唯一的解决办法就是发现所有潜伏的地雷并且排除它们，躲是躲不了的。本章的内容比一般教科书的要深入得多，读者需细心阅读，做到真正地通晓内存管理。

7.1 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编程人员决定，使用非常灵活，但问题也最多。

7.2 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。

常见的内存错误及其对策如下：

◆ 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。

◆ 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，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。

◆ 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。

◆ 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）。

◆ 释放了内存却继续使用它。

有三种情况：

（1）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。

（2）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。

（3）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”。

- **【规则 7-2-1】**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 7-2-2】**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 7-2-3】**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 7-2-4】**动态内存的申请与释放必须配对，防止内存泄漏。
- **【规则 7-2-5】**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”。

7.3 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。

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。

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。

7.3.1 修改内容

示例 7-3-1 中，字符数组 a 的容量是 6 个字符，其内容为 hello\0。a 的内容可以改变，如 a[0]= ‘X’。指针 p 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 p[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 a[] = “hello”;
a[0] = ‘X’;
cout << a << endl;
char *p = “world”;      // 注意 p 指向常量字符串
p[0] = ‘X’;             // 编译器不能发现该错误
cout << p << endl;
```

示例 7-3-1 修改数组和指针的内容

7.3.2 内容复制与比较

不能对数组名进行直接复制与比较。示例 7-3-2 中，若想把数组 a 的内容复制给数组 b，不能用语句 b = a ，否则将产生编译错误。应该用标准库函数 strcpy 进行复制。同理，比较 b 和 a 的内容是否相同，不能用 if(b==a) 来判断，应该用标准库函数 strcmp 进行比较。

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

```
// 数组…
```

```
char a[] = "hello";
char b[10];
strcpy(b, a);          // 不能用  b = a;
if(strcmp(b, a) == 0)  // 不能用  if (b == a)
…
```

```
// 指针…
int len = strlen(a);
char *p = (char *)malloc(sizeof(char)*(len+1));
strcpy(p,a);           // 不要用 p = a;
if(strcmp(p, a) == 0)  // 不要用 if (p == a)
…
```

示例 7-3-2 数组和指针的内容复制与比较

7.3.3 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 7-3-3(a)中,sizeof(a)的值是 12（注意别忘了'\0'）。指针 p 指向 a，但是 sizeof(p)的值却是 4。这是因为 sizeof(p)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 7-3-3（b）中，不论数组 a 的容量是多少，sizeof(a)始终等于 sizeof(char *)。

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

示例 7-3-3（a） 计算数组和指针的内存容量

```
void Func(char a[100])
{
    cout<< sizeof(a) << endl; // 4 字节而不是 100 字节
}
```

示例 7-3-3（b） 数组退化为指针

7.4 指针参数是如何传递内存的？

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 7-4-1 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，为什么？

```
void GetMemory(char *p, int num)
{
    p = (char *)malloc(sizeof(char) * num);
}
```

```
void Test(void)
{
    char *str = NULL;
    GetMemory(str, 100);  // str 仍然为 NULL
    strcpy(str, "hello"); // 运行错误
}
```

示例 7-4-1 试图用指针参数申请动态内存

问题出在函数 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 p 的副本是 _p，编译器使 _p = p。如果函数体内的程序修改了_p 的内容，就导致参数 p 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，_p 申请了新的内存，只是把_p 所指的内存地址改变了，但是 p 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 7-4-2。

```
void GetMemory2(char **p, int num)
{
    *p = (char *)malloc(sizeof(char) * num);
}
```

```
void Test2(void)
{
    char *str = NULL;
    GetMemory2(&str, 100); // 注意参数是 &str，而不是 str
```

```
    strcpy(str, "hello");
    cout<< str << endl;
    free(str);
}
```

示例 7-4-2 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，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 7-4-3。

```
char *GetMemory3(int num)
{
    char *p = (char *)malloc(sizeof(char) * num);
    return p;
}
```

```
void Test3(void)
{
    char *str = NULL;
    str = GetMemory3(100);
    strcpy(str, "hello");
    cout<< str << endl;
    free(str);
}
```

示例 7-4-3 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 7-4-4。

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

```
void Test4(void)
{
    char *str = NULL;
```

```
    str = GetString(); // str 的内容是垃圾
    cout<< str << endl;
}
```

示例 7-4-4 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 str = GetString 语句后 str 不再是 NULL 指针，但是 str 的内容不是“hello world”而是垃圾。

如果把示例 7-4-4 改写成示例 7-4-5，会怎么样？

```
char *GetString2(void)
{
    char *p = "hello world";
    return p;
}
```

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

示例 7-4-5 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。

7.5 free 和 delete 把指针怎么啦？

free 和 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 7-5，发现指针 p 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，p 成了“野指针”。如果此时不把 p 设置为 NULL，会让人误以为 p 是个合法的指针。

如果程序比较长，有时记不住 p 所指的内存是否已经被释放，在继续使用 p 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。

```
char *p = (char *) malloc(100);
strcpy(p, “hello”);
free(p);        // p 所指的内存被释放，但是 p 所指的地址仍然不变
…
if(p != NULL)  // 没有起到防错作用
{
   strcpy(p, “world”); // 出错
}
```

示例 7-5　p 成为野指针

7.6 动态内存会被自动释放吗？

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 7-6 是正确的。理由是 p 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消亡。这是错觉！

```
void Func(void)
{
    char *p = (char *) malloc(100);   // 动态内存会自动释放吗？
}
```

示例 7-6 试图让动态内存自动释放

发现指针有一些“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。也许有人不服气，一定要找出可以草率行事的理由：

如果程序终止了运行，一切指针都会消亡，动态内存会被操作系统回收。既然如此，在程序临终前，就可以不必释放内存、不必将指针设置为 NULL 了。终于可以偷懒而不会发生错误了吧？

但是请仔细想想，如果别人把那段程序取出来用到其它地方怎么办？

7.7 杜绝“野指针”

“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：

（1）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 *p = NULL;
char *str = (char *) malloc(100);
```

（2）指针 p 被 free 或者 delete 之后，没有置为 NULL，让人误以为 p 是个合法的指针。参见 7.5 节。

（3）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 A
{
public:
    void Func(void){ cout << “Func of class A” << endl; }
};
void Test(void)
{
    A  *p;
        {
            A  a;
            p = &a; // 注意 a 的生命期
        }
    p->Func();      // p 是“野指针”
}
```

函数 Test 在执行语句 p->Func()时，对象 a 已经消失，而 p 是指向 a 的，所以 p 就成了“野指针”。但奇怪的是运行这个程序时居然没有出错，这可能与编译器有关。

7.8 有了 malloc/free 为什么还要 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。

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 7-8。

```
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  *a = (obj *)malloc(sizeof(obj)); // 申请动态内存
    a->Initialize();                      // 初始化
    //...
    a->Destroy();  // 清除工作
    free(a);       // 释放内存
}
```

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

示例 7-8 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。

既然 new/delete 的功能完全覆盖了 malloc/free，为什么 C++不把 malloc/free 淘汰出局呢？这是因为 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。

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。

7.9 内存耗尽怎么办？

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)
{
    A  *a = new A;
    if(a == 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)
{
    A  *a = new A;
    if(a == 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。

有一个很重要的现象要告诉大家。对于 32 位以上的应用程序而言，无论怎样使用 malloc 与 new，几乎不可能导致"内存耗尽"。在 Windows 98 下用 Visual C++编写了测试程序，见示例 7-9。这个程序会无休止地运行下去，根本不会终止。因为 32 位操作系统支持"虚存"，内存用完了，自动用硬盘空间顶替。只听到硬盘嘎吱嘎吱地响，Window 98 已经累得对键盘、鼠标毫无反应。

可以得出这么一个结论：对于 32 位以上的应用程序，"内存耗尽"错误处理程序毫无用处。这下可把 Unix 和 Windows 程序员们乐坏了：反正错误处理程序不起作用，就不写了，省了很多麻烦。

必须强调：不加错误处理将导致程序的质量很差，千万不可因小失大。

```
void main(void)
{
    float *p = NULL;
    while(TRUE)
    {
        p = new float[1000000];
        cout << "eat memory" << endl;
        if(p==NULL)
```

```
            exit(1);
        }
    }
```

示例 7-9 试图耗尽操作系统的内存

7.10 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  *p = (int *) malloc(sizeof(int) * length);
```

应当把注意力集中在两个要素上：“类型转换”和“sizeof”。

- 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。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，但要当心有时会昏了头，写出 p = malloc(sizeof(p))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。

7.11 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  *p1 = (int *)malloc(sizeof(int) * length);
int  *p2 = new int[length];
```

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 x);     // 带一个参数的构造函数
…
}
void Test(void)
{
    Obj  *a = new Obj;
    Obj  *b = new Obj(1);  // 初值为 1
    …
    delete a;
    delete b;
}
```

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。

7.12 一些心得体会

对于技术不错的 C++/C 程序员，很少有人能拍拍胸脯说通晓指针与内存管理。

初学 C 语言时可能会特别怕指针，导致开发应用软件时不用指针，全用数组来顶替指针。若恰当的使用指针可以使代码量缩小很多。

经验教训是：

（1）越是怕指针，就越要使用指针。不会正确使用指针，肯定算不上是合格的程序员。

（2）必须养成“使用调试器逐步跟踪程序”的习惯，只有这样才能发现问题的本质。

第 8 章 C++函数的高级特性

对比于 C 语言的函数，C++增加了重载（overloaded）、内联（inline）、const 和 virtual 四种新机制。其中重载和内联机制既可用于全局函数也可用于类的成员函数，const 与 virtual 机制仅用于类的成员函数。

重载和内联肯定有其好处才会被 C++语言采纳，但是不可以当成免费的午餐而滥用。本章将探究重载和内联的优点与局限性，说明什么情况下应该采用、不该采用以及要警惕错用。

8.1 函数重载的概念

8.1.1 重载的起源

自然语言中，一个词可以有许多不同的含义，即该词被重载了。人们可以通过上下文来判断该词到底是哪种含义。“词的重载”可以使语言更加简练。例如“吃饭”的含义十分广泛，人们没有必要每次非得说清楚具体吃什么不可。别迂腐得象孔已己，说茴香豆的茴字有四种写法。

在 C++程序中，可以将语义、功能相似的几个函数用同一个名字表示，即函数重载。这样便于记忆，提高了函数的易用性，这是 C++语言采用重载机制的一个理由。例如示例 8-1-1 中的函数 EatBeef,EatFish,EatChicken 可以用同一个函数名 Eat 表示，用不同类型的参数加以区别。

```
void EatBeef(…);        // 可以改为      void Eat(Beef …);
void EatFish(…);        // 可以改为      void Eat(Fish …);
void EatChicken(…);     // 可以改为      void Eat(Chicken …);
```

示例 8-1-1 重载函数 Eat

C++语言采用重载机制的另一个理由是：类的构造函数需要重载机制。因为 C++规定构造函数与类同名（请参见第 9 章），构造函数只能有一个名字。如果想用几种不同的方法创建对象该怎么办？别无选择，只能用重载机制来实现。所以类可以有多个同名的构造函数。

8.1.2 重载是如何实现的？

几个同名的重载函数仍然是不同的函数，它们是如何区分的呢？自然想到函数

接口的两个要素：参数与返回值。

如果同名函数的参数不同（包括类型、顺序不同），那么容易区别出它们是不同的函数。

如果同名函数仅仅是返回值类型不同，有时可以区分，有时却不能。例如：

```
void Function(void);
int  Function (void);
```

上述两个函数，第一个没有返回值，第二个的返回值是 int 类型。如果这样调用函数：

```
int  x = Function ();
```

则可以判断出 Function 是第二个函数。问题是在 C++/C 程序中，可以忽略函数的返回值。在这种情况下，编译器和程序员都不知道哪个 Function 函数被调用。

所以只能靠参数而不能靠返回值类型的不同来区分重载函数。编译器根据参数为每个重载函数产生不同的内部标识符。例如编译器为示例 8-1-1 中的三个 Eat 函数产生象_eat_beef、_eat_fish、_eat_chicken 之类的内部标识符（不同的编译器可能产生不同风格的内部标识符）。

如果 C++程序要调用已经被编译后的 C 函数，该怎么办？

假设某个 C 函数的声明如下：

```
void foo(int x, int y);
```

该函数被 C 编译器编译后在库中的名字为_foo，而 C++编译器则会产生像_foo_int_int 之类的名字用来支持函数重载和类型安全连接。由于编译后的名字不同，C++程序不能直接调用 C 函数。C++提供了一个 C 连接交换指定符号 extern“C”来解决这个问题。例如：

```
extern “C”
{
   void foo(int x, int y);
   … // 其它函数
}
```

或者写成

```
extern “C”
{
   #include “myheader.h”
   … // 其它 C 头文件
}
```

这就告诉 C++编译译器，函数 foo 是个 C 连接，应该到库中找名字_foo 而不是找_foo_int_int。C++编译器开发商已经对 C 标准库的头文件作了 extern“C”处理，所以可以用＃include 直接引用这些头文件。

注意并不是两个函数的名字相同就能构成重载。全局函数和类的成员函数同名不算重载，因为函数的作用域不同。例如：

```
void Print(…);     // 全局函数
class A
{…
    void Print(…);// 成员函数
}
```

不论两个 Print 函数的参数是否不同，如果类的某个成员函数要调用全局函数 Print，为了与成员函数 Print 区别，全局函数被调用时应加‘::’标志。如

```
::Print(…);    // 表示 Print 是全局函数而非成员函数
```

8.1.3 当心隐式类型转换导致重载函数产生二义性

示例 8-1-3 中，第一个 output 函数的参数是 int 类型，第二个 output 函数的参数是 float 类型。由于数字本身没有类型，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）。语句 output(0.5)将产生编译错误，因为编译器不知道该将 0.5 转换成 int 还是 float 类型的参数。隐式类型转换在很多地方可以简化程序的书写，但是也可能留下隐患。

```
# include <iostream.h>
void output( int x);  // 函数声明
void output( float x);// 函数声明

void output( int x)
{
    cout << " output int " << x << endl ;
}

void output( float x)
{
    cout << " output float " << x << endl ;
}

void main(void)
{
    int   x = 1;
```

```
    float y = 1.0;
    output(x);          // output int 1
    output(y);          // output float 1
    output(1);          // output int 1
//  output(0.5);        // error! ambiguous call, 因为自动类型转换
    output(int(0.5));   // output int 0
    output(float(0.5));    // output float 0.5
}
```

示例 8-1-3 隐式类型转换导致重载函数产生二义性

8.2 成员函数的重载、覆盖与隐藏

成员函数的重载、覆盖（override）与隐藏很容易混淆，C++程序员必须要搞清楚概念，否则错误将防不胜防。

8.2.1 重载与覆盖

成员函数被重载的特征：

（1）相同的范围（在同一个类中）；

（2）函数名字相同；

（3）参数不同；

（4）virtual 关键字可有可无。

覆盖是指派生类函数覆盖基类函数，特征是：

（1）不同的范围（分别位于派生类与基类）；

（2）函数名字相同；

（3）参数相同；

（4）基类函数必须有 virtual 关键字。

示例 8-2-1 中，函数 Base::f(int)与 Base::f(float)相互重载，而 Base::g(void)被 Derived::g(void)覆盖。

```
#include <iostream.h>
class Base
{
  public:
          void f(int x){ cout << "Base::f(int) " << x << endl; }
          void f(float x){ cout << "Base::f(float) " << x << endl; }
```

```
  virtual void g(void){ cout << "Base::g(void)" << endl;}
};
```

```
class Derived : public Base
{
  public:
  virtual void g(void){ cout << "Derived::g(void)" << endl;}
};
```

```
void main(void)
{
  Derived  d;
  Base *pb = &d;
  pb->f(42);        // Base::f(int) 42
  pb->f(3.14f);     // Base::f(float) 3.14
  pb->g();          // Derived::g(void)
}
```

示例 8-2-1 成员函数的重载和覆盖

8.2.2 令人迷惑的隐藏规则

本来仅仅区别重载与覆盖并不算困难，但是 C++的隐藏规则使问题复杂性陡然增加。这里“隐藏”是指派生类的函数屏蔽了与其同名的基类函数，规则如下：

（1）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）。

（2）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）。

示例程序 8-2-2（a）中：

（1）函数 Derived::f(float)覆盖了 Base::f(float)。

（2）函数 Derived::g(int)隐藏了 Base::g(float)，而不是重载。

（3）函数 Derived::h(float)隐藏了 Base::h(float)，而不是覆盖。

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

```
        void g(float x){ cout << "Base::g(float) " << x << endl; }
        void h(float x){ cout << "Base::h(float) " << x << endl; }
};
```

```
class Derived : public Base
{
public:
virtual void f(float x){ cout << "Derived::f(float) " << x << endl; }
        void g(int x){ cout << "Derived::g(int) " << x << endl; }
        void h(float x){ cout << "Derived::h(float) " << x << endl; }
};
```

示例 8-2-2（a）成员函数的重载、覆盖和隐藏

据作者考察，很多 C++程序员没有意识到有“隐藏”这回事。由于认识不够深刻，“隐藏”的发生可谓神出鬼没，常常产生令人迷惑的结果。

示例 8-2-2（b）中，bp 和 dp 指向同一地址，按理说运行结果应该是相同的，可事实并非这样。

```
  void main(void)
  {
    Derived  d;
    Base *pb = &d;
    Derived *pd = &d;
    // Good : behavior depends solely on type of the object
    pb->f(3.14f);    // Derived::f(float) 3.14
    pd->f(3.14f);    // Derived::f(float) 3.14

    // Bad : behavior depends on type of the pointer
    pb->g(3.14f);    // Base::g(float) 3.14
    pd->g(3.14f);    // Derived::g(int) 3        (surprise!)

    // Bad : behavior depends on type of the pointer
    pb->h(3.14f);    // Base::h(float) 3.14      (surprise!)
    pd->h(3.14f);    // Derived::h(float) 3.14
  }
```

示例 8-2-2（b） 重载、覆盖和隐藏的比较

8.2.3 摆脱隐藏

隐藏规则引起了不少麻烦。示例 8-2-3 程序中，语句 pd->f(10)的本意是想调用函数 Base::f(int)，但是 Base::f(int)不幸被 Derived::f(char *)隐藏了。由于数字 10 不能被隐式地转化为字符串，所以在编译时出错。

```
class Base
{
  public:
    void f(int x);
};
```

```
class Derived : public Base
{
  public:
    void f(char *str);
};
```

```
void Test(void)
{
  Derived *pd = new Derived;
  pd->f(10);   // error
}
```

示例 8-2-3 由于隐藏而导致错误

从示例 8-2-3 看来，隐藏规则似乎很愚蠢。但是隐藏规则至少有两个存在的理由：

- 写语句 pd->f(10)的人可能真的想调用 Derived::f(char *)函数，只是他误将参数写错了。有了隐藏规则，编译器就可以明确指出错误，这未必不是好事。否则，编译器会静悄悄地将错就错，程序员将很难发现这个错误，留下祸根。
- 假如类 Derived 有多个基类（多重继承），有时搞不清楚哪些基类定义了函数 f。如果没有隐藏规则，那么 pd->f(10)可能会调用一个出乎意料的基类函数 f。尽管隐藏规则看起来不怎么有道理，但它的确能消灭这些意外。

示例 8-2-3 中，如果语句 pd->f(10)一定要调用函数 Base::f(int)，那么将类 Derived 修改为如下即可。

```
class Derived : public Base
{
  public:
  void f(char *str);
```

```
    void f(int x) { Base::f(x); }
};
```

8.3 参数的缺省值

有一些参数的值在每次函数调用时都相同，书写这样的语句会使人厌烦。C++语言采用参数的缺省值使书写变得简洁（在编译时，缺省值由编译器自动插入）。

参数缺省值的使用规则：

- **【规则 8-3-1】**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。

  例如：

```
void Foo(int x=0, int y=0);   // 正确，缺省值出现在函数的声明中

void Foo(int x=0, int y=0)    // 错误，缺省值出现在函数的定义体中
{
…
}
```

为什么会这样？有两个原因：一是函数的实现（定义）本来就与参数是否有缺省值无关，所以没有必要让缺省值出现在函数的定义体中。二是参数的缺省值可能会改动，显然修改函数的声明比修改函数的定义要方便。

- **【规则 8-3-2】**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。

  正确的示例如下：

```
    void Foo(int x, int y=0, int z=0);
```

  错误的示例如下：

```
    void Foo(int x=0, int y, int z=0);
```

要注意，使用参数的缺省值并没有赋予函数新的功能，仅仅是使书写变得简洁一些。它可能会提高函数的易用性，但是也可能会降低函数的可理解性。所以只能适当地使用参数的缺省值，要防止使用不当产生负面效果。示例 8-3-2 中，不合理地使用参数的缺省值将导致重载函数 output 产生二义性。

```
#include <iostream.h>
void output( int x);
void output( int x, float y=0.0);
```

```
void output( int x)
{
    cout << " output int " << x << endl ;
}
```

```
void output( int x, float y)
{
    cout << " output int " << x << " and float " << y << endl ;
}
```

```
void main(void)
{
    int x=1;
    float y=0.5;
//  output(x);          // error! ambiguous call
    output(x,y);        // output int 1 and float 0.5
}
```

示例 8-3-2　参数的缺省值将导致重载函数产生二义性

8.4 运算符重载

8.4.1 概念

在 C++语言中，可以用关键字 operator 加上运算符来表示函数，叫做运算符重载。例如两个复数相加函数：

```
Complex Add(const Complex &a, const Complex &b);
```

可以用运算符重载来表示：

```
Complex operator +(const Complex &a, const Complex &b);
```

运算符与普通函数在调用时的不同之处是：对于普通函数，参数出现在圆括号内；而对于运算符，参数出现在其左、右侧。例如

```
Complex a, b, c;
…
c = Add(a, b); // 用普通函数
c = a + b;     // 用运算符 +
```

如果运算符被重载为全局函数，那么只有一个参数的运算符叫做一元运算符，有两个参数的运算符叫做二元运算符。

如果运算符被重载为类的成员函数，那么一元运算符没有参数，二元运算符只有一个右侧参数，因为对象自己成了左侧参数。

从语法上讲，运算符既可以定义为全局函数，也可以定义为成员函数。文献[Murray , p44-p47]对此问题作了较多的阐述，并总结了表 8-4-1 的规则。

运算符	规则
所有的一元运算符	建议重载为成员函数
= () [] ->	只能重载为成员函数
+= -= /= *= &= |= ~= %= >>= <<=	建议重载为成员函数
所有其它运算符	建议重载为全局函数

表 8-4-1 运算符的重载规则

由于 C++语言支持函数重载，才能将运算符当成函数来用，C 语言就不行。要以平常心来对待运算符重载：

（1）不要过分担心自己不会用，它的本质仍然是程序员们熟悉的函数。

（2）不要过分热心地使用，如果它不能使代码变得更加易读易写，那就别用，否则会自找麻烦。

8.4.2 不能被重载的运算符

在 C++运算符集合中，有一些运算符是不允许被重载的。这种限制是出于安全方面的考虑，可防止错误和混乱。

（1）不能改变 C++内部数据类型（如 int,float 等）的运算符。

（2）不能重载‘.’，因为‘.’在类中对任何成员都有意义，已经成为标准用法。

（3）不能重载目前 C++运算符集合中没有的符号，如#,@,$等。原因有两点，一是难以理解，二是难以确定优先级。

（4）对已经存在的运算符进行重载时，不能改变优先级规则，否则将引起混乱。

8.5 函数内联

8.5.1 用内联取代宏代码

C++ 语言支持函数内联，其目的是为了提高函数的执行效率（速度）。

在 C 程序中，可以用宏代码提高执行效率。宏代码本身不是函数，但使用起来象函数。预处理器用复制宏代码的方式代替函数调用，省去了参数压栈、生成汇编语言的 CALL 调用、返回参数、执行 return 等过程，从而提高了速度。使用宏代码最大的缺点是容易出错，预处理器在复制宏代码时常常产生意想不到的边际效应。例如

```
#define MAX(a, b)        (a) > (b) ? (a) : (b)
```

语句

```
result = MAX(i, j) + 2 ;
```

将被预处理器解释为

```
result = (i) > (j) ? (i) : (j) + 2 ;
```

由于运算符‘+’比运算符‘:’的优先级高，所以上述语句并不等价于期望的

```
result = ( (i) > (j) ? (i) : (j) ) + 2 ;
```

如果把宏代码改写为

```
#define MAX(a, b)        ( (a) > (b) ? (a) : (b) )
```

则可以解决由优先级引起的错误。但是即使使用修改后的宏代码也不是万无一失的，例如语句

```
result = MAX(i++, j);
```

将被预处理器解释为

```
result = (i++) > (j) ? (i++) : (j);
```

对于 C++ 而言，使用宏代码还有另一种缺点：无法操作类的私有数据成员。

看看 C++ 的“函数内联”是如何工作的。对于任何内联函数，编译器在符号表里放入函数的声明（包括名字、参数类型、返回值类型）。如果编译器没有发现内联函数存在错误，那么该函数的代码也被放入符号表里。在调用一个内联函数时，编译器首先检查调用是否正确（进行类型安全检查，或者进行自动类型转换，当然对所有的函数都一样）。如果正确，内联函数的代码就会直接替换函数调用，于是省去了函数调用的开销。这个过程与预处理有显著的不同，因为预处理器不能进行类型安全检查，或者进行自动类型转换。假如内联函数是成员函数，对象的地址（this）会被放在合适的地方，这也是预处理器办不到的。

C++ 语言的函数内联机制既具备宏代码的效率，又增加了安全性，而且可以自由操作类的数据成员。所以在 C++ 程序中，应该用内联函数取代所有宏代码，“断言 assert”恐怕是唯一的例外。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。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。如果 assert 是函数，由于函数调用会引起内存、代码的变动，那么将导致 Debug 版本与 Release 版本存在差异。所以 assert 不是函数，而是宏。（参见 6.5 节“使用断言”）

8.5.2 内联函数的编程风格

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。如下风格的函数 Foo 不能成为内联函数：

```
inline void Foo(int x, int y);    // inline 仅与函数声明放在一起
void Foo(int x, int y)
```

```
{
    …
}
```

而如下风格的函数 Foo 则成为内联函数：

```
void Foo(int x, int y);
inline void Foo(int x, int y) // inline 与函数定义体放在一起
{
    …
}
```

所以说，inline 是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了 inline 关键字，但一般认为 inline 不应该出现在函数的声明中。这个细节虽然不会影响函数的功能，但是体现了高质量 C++/C 程序设计风格的一个基本原则：声明与定义不可混为一谈，用户没有必要、也不应该知道函数是否需要内联。

定义在类声明之中的成员函数将自动地成为内联函数，例如

```
class A
{
public:
    void Foo(int x, int y) { … } // 自动地成为内联函数
}
```

将成员函数的定义体放在类声明之中虽然能带来书写上的方便，但不是一种良好的编程风格，上例应该改成：

```
// 头文件
class A
{
public:
    void Foo(int x, int y);
}
// 定义文件
inline void A::Foo(int x, int y)
{
    …
}
```

8.5.3 慎用内联

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数？

如果所有的函数都是内联函数，还用得着“内联”这个关键字吗？

内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。以下情况不宜使用内联：

（1）如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。

(2)如果函数体内出现循环,那么执行函数体内代码的时间要比函数调用的开销大。

类的构造函数和析构函数容易让人误解成使用内联更有效。要当心构造函数和析构函数可能会隐藏一些行为，如“偷偷地”执行了基类或成员对象的构造函数和析构函数。所以不要随便地将构造函数和析构函数的定义体放在类声明中。

一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这进一步说明了 inline 不应该出现在函数的声明中）。

8.6 一些心得体会

C++ 语言中的重载、内联、缺省参数、隐式转换等机制展现了很多优点，但是这些优点的背后都隐藏着一些隐患。正如人们的饮食，少食和暴食都不可取，应当恰到好处。要辨证地看待 C++的新机制，应该恰如其分地使用它们。虽然这会使编程时多费一些心思，少了一些痛快，但这才是编程的艺术。

第 9 章 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。它们太普通以致让人容易麻痹大意，其实这些貌似简单的函数就象没有顶盖的下水道那样危险。

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 A，如果不想编写上述函数，C++编译器将自动为 A 产生四个缺省的函数，如

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

这不禁让人疑惑，既然能自动生成函数，为什么还要程序员编写？

原因如下：

（1）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。

（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。

对于那些没有吃够苦头的 C++程序员，如果他说编写构造函数、析构函数与赋值函数很容易，可以不用动脑筋，表明他的认识还比较肤浅，水平有待于提高。

本章以类 String 的设计与实现为例，深入阐述被很多教科书忽视了的道理。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 & operator =(const String &other);// 赋值函数
  private:
    char    *m_data;              // 用于保存字符串
};
```

9.1 构造函数与析构函数的起源

作为比 C 更先进的语言，C++提供了更好的机制来增强程序的安全性。C++编译器具有严格的类型安全检查功能，它几乎能找出程序中所有的语法问题，这的确帮了程序员的大忙。但是程序通过了编译检查并不表示错误已经不存在了，在“错误”的大家庭里，“语法错误”的地位只能算是小弟弟。级别高的错误通常隐藏得很深，就象狡猾的罪犯，想逮住他可不容易。

根据经验，不少难以察觉的程序错误是由于变量没有被正确初始化或清除造成的，而初始化和清除工作很容易被人遗忘。Stroustrup 在设计 C++语言时充分考虑了这个问题并很好地予以解决：*把对象的初始化工作放在构造函数中，把清除工作放在析构函数中。当对象被创建时，构造函数被自动执行。当对象消亡时，析构函数被自动执行。这下就不用担心忘了对象的初始化和清除工作。*

构造函数与析构函数的名字不能随便起，必须让编译器认得出才可以被自动执行。Stroustrup 的命名方法既简单又合理：*让构造函数、析构函数与类同名，由于析构函数的目的与构造函数的相反，就加前缀‘~’以示区别。*

除了名字外，构造函数与析构函数的另一个特别之处是没有返回值类型，这与返回值类型为 void 的函数不同。构造函数与析构函数的使命非常明确，就象出生与死亡，光溜溜地来光溜溜地去。如果它们有返回值类型，那么编译器将不知所措。为了防止节外生枝，干脆规定没有返回值类型。（以上典故参考了文献[Eekel, p55-p56]）

9.2 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：

◆ 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 A
{...
    A(int x);        // A 的构造函数
};
class B : public A
{...
```

```
    B(int x, int y);// B 的构造函数
};
B::B(int x, int y)
 : A(x)             // 在初始化表里调用 A 的构造函数
{
  …
}
```

- 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（参见 5.4 节）。
- 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 A
{…
    A(void);                // 无参数构造函数
    A(const A &other);      // 拷贝构造函数
    A & operator =( const A &other);  // 赋值函数
};

class B
{
  public:
    B(const A &a); // B 的构造函数
  private:
    A  m_a;         // 成员对象
};
```

示例 9-2(a)中，类 B 的构造函数在其初始化表里调用了类 A 的拷贝构造函数，从而将成员对象 m_a 初始化。

示例 9-2 (b)中，类 B 的构造函数在函数体内用赋值的方式将成员对象 m_a 初始化。看到的只是一条赋值语句，但实际上 B 的构造函数干了两件事：先暗地里创建 m_a 对象（调用了 A 的无参数构造函数），再调用类 A 的赋值函数，将参数 a 赋给 m_a。

```
B::B(const A &a)
 : m_a(a)
{
   …
}
```

示例 9-2(a) 成员对象在初始化表中被初始化

```
B::B(const A &a)
{
    m_a = a;
    …
}
```

示例 9-2(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 F 的声明如下：

```
class F
{
  public:
    F(int x, int y);       // 构造函数
  private:
    int m_x, m_y;
    int m_i, m_j;
}
```

示例 9-2(c)中 F 的构造函数采用了第一种初始化方式，示例 9-2(d)中 F 的构造函数采用了第二种初始化方式。

```
F::F(int x, int y)
 : m_x(x), m_y(y)
{
   m_i = 0;
   m_j = 0;
}
```

示例 9-2(c) 数据成员在初始化表中被初始化

```
F::F(int x, int y)
{
   m_x = x;
   m_y = y;
   m_i = 0;
   m_j = 0;
}
```

示例 9-2(d) 数据成员在函数体内被初始化

9.3 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。[Eckel, p260-261]

9.4 示例：类 String 的构造函数与析构函数

```
// String 的普通构造函数
String::String(const char *str)
{
    if(str==NULL)
    {
        m_data = new char[1];
        *m_data = '\0';
    }
    else
    {
        int length = strlen(str);
        m_data = new char[length+1];
        strcpy(m_data, str);
    }
}

// String 的析构函数
String::~String(void)
{
    delete [] m_data;
    // 由于 m_data 是内部数据类型，也可以写成 delete m_data;
}
```

9.5 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：

◆ 本章开头讲过，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 a,b 为例，假设 a.m_data 的内容为“hello”，b.m_data 的内容为“world”。

现将 a 赋给 b，缺省赋值函数的“位拷贝”意味着执行 b.m_data = a.m_data。这将造成三个错误：一是 b.m_data 原有的内存没被释放，造成内存泄露；二是 b.m_data 和 a.m_data 指向同一块内存，a 或 b 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。

◆ 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，你分得清楚哪个调用了拷贝构造函数，哪个调用了赋值函数吗？

```
String  a(“hello”);
String  b(“world”);
String  c = a; // 调用了拷贝构造函数，最好写成 c(a);
        c = b; // 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 c(a) 以区别于第四个语句。

9.6 示例：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)
{
    // 允许操作 other 的私有成员 m_data
    int length = strlen(other.m_data);
    m_data = new char[length+1];
    strcpy(m_data, other.m_data);
}

// 赋值函数
String & String::operator =(const String &other)
{
    // (1) 检查自赋值
    if(this == &other)
        return *this;

    // (2) 释放原有的内存资源
```

```
    delete [] m_data;

    // （3）分配新的内存资源，并复制内容
    int length = strlen(other.m_data);
    m_data = new char[length+1];
    strcpy(m_data, 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。

类 String 的赋值函数比构造函数复杂得多，分四步实现：

（1）第一步，检查自赋值。你可能会认为多此一举，难道有人会愚蠢到写出 a = a 这样的自赋值语句！的确不会。但是间接的自赋值仍有可能出现，例如

// 内容自赋值	// 地址自赋值
b = a;	b = &a;
…	…
c = b;	a = *b;
…	
a = c;	

也许有人会说：“即使出现自赋值，我也可以不理睬，大不了化点时间让对象复制自己而已，反正不会出错！”

他真的说错了。看看第二步的 delete，自杀后还能复制自己吗？所以，如果发现自赋值，应该马上终止函数。注意不要将检查自赋值的 if 语句

```
if(this == &other)
```

错写成为

```
if( *this == other)
```

（2）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。

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

（4）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 。那么能否写成 return other 呢？效果不是一样吗？

不可以！因为不知道参数 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。

9.7 偷懒的办法处理拷贝构造函数与赋值函数

如果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，怎么办？

偷懒的办法是：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 A
{ ...
  private:
    A(const A &a);              // 私有的拷贝构造函数
    A & operator =(const A &a);   // 私有的赋值函数
};
```

如果有人试图编写如下程序：

```
A  b(a);    // 调用了私有的拷贝构造函数
b = 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。

9.8 如何在派生类中实现类的基本函数

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：

- 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。
- 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 * pB = new Derived;  // upcast
    delete pB;
}
```

输出结果为：

~Derived

~Base

如果析构函数不为虚，那么输出结果为

~Base

◆ 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 & operator =(const Base &other); // 类 Base 的赋值函数
  private:
    int  m_i, m_j, m_k;
};

class Derived : public Base
{
  public:
    …
    Derived & operator =(const Derived &other);  // 类 Derived 的赋值
函数
  private:
```

```
    int  m_x, m_y, m_z;
};

Derived & Derived::operator =(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::operator =(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_x = other.m_x;
    m_y = other.m_y;
    m_z = other.m_z;

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

9.9 一些心得体会

有些 C++程序设计书籍称构造函数、析构函数和赋值函数是类的“Big-Three”，它们的确是任何类最重要的函数，不容轻视。

第 10 章 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。

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。本章仅仅论述“继承”（Inheritance）和“组合”（Composition）的概念。

10.1 继承

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

```
class A
{
  public:
        void   Func1(void);
        void   Func2(void);
};

class B : public A
{
  public:
        void   Func3(void);
        void   Func4(void);
};

main()
{
    B   b;
    b.Func1();          // B 从 A 继承了函数 Func1
    b.Func2();          // B 从 A 继承了函数 Func2
    b.Func3();
    b.Func4();
}
```

这个简单的示例程序说明了一个事实：C++的“继承”特性可以提高程序的可复用性。正因为“继承”太有用、太容易用，才要防止乱用“继承”。应当给“继承”立一些使用规则。

- **【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。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。
- **【规则 10-1-2】**若在逻辑上 B 是 A 的“一种”（a kind of ），则允许 B 继承 A 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**

【规则 10-1-2】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 是什么东西？

```
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，岂非画蛇添足？

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。**

10.2 组合

- **【规则 10-2-1】**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 10-2-1 所示。

class Eye
{
  public:
  void Look(void);
};
	class Nose
	{
	  public:
	  void Smell(void);
	};

class Mouth
{
  public:
  void Eat(void);
};
	class Ear
	{
	  public:
	  void Listen(void);
	};

// 正确的设计，虽然代码冗长。
class Head
{
  public:
  void Look(void) { m_eye.Look(); }
  void Smell(void) { m_nose.Smell(); }
  void Eat(void) { m_mouth.Eat(); }
  void Listen(void) { m_ear.Listen(); }
  private:	

```
        Eye     m_eye;
        Nose    m_nose;
        Mouth   m_mouth;
        Ear     m_ear;
};
```

示例 10-2-1 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 10-2-2 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 10-2-2　Head 从 Eye、Nose、Mouth、Ear 派生而成

很多程序员经不起“继承”的诱惑而犯下设计错误。“运行正确”的程序不见得是高质量的程序，此处就是一个例证。

第 11 章 其它编程经验

11.1 使用 const 提高函数的健壮性

看到 const 关键字，C++程序员首先想到的可能是 const 常量。这可不是良好的条件反射。如果只知道用 const 定义常量，那么相当于把火药仅用于制作鞭炮。const 更大的魅力是它可以修饰函数的参数、返回值，甚至函数的定义体。

const 是 constant 的缩写，“恒定不变”的意思。被 const 修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。所以很多 C++程序设计书籍建议：“Use const whenever you need”。

11.1.1 用 const 修饰函数的参数

如果参数作输出用，不论它是什么数据类型，也不论它采用“指针传递”还是“引用传递”，都不能加 const 修饰，否则该参数将失去输出功能。

const 只能修饰输入参数：

◆ 如果输入参数采用“指针传递”，那么加 const 修饰可以防止意外地改动该指针，起到保护作用。

例如 StringCopy 函数：

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

其中 strSource 是输入参数，strDestination 是输出参数。给 strSource 加上 const 修饰后，如果函数体内的语句试图改动 strSource 的内容，编译器将指出错误。

◆ 如果输入参数采用“值传递”，由于函数将自动产生临时变量用于复制该参数，该输入参数本来就无需保护，所以不要加 const 修饰。

例如不要将函数 void Func1(int x) 写成 void Func1(const int x)。同理不要将函数 void Func2(A a) 写成 void Func2(const A a)。其中 A 为用户自定义的数据类型。

◆ 对于非内部数据类型的参数而言，象 void Func(A a) 这样声明的函数注定效率比较底。因为函数体内将产生 A 类型的临时对象用于复制参数 a，而临时对象的构造、复制、析构过程都将消耗时间。

为了提高效率，可以将函数声明改为 void Func(A &a)，因为“引用传递”仅借用一下参数的别名而已，不需要产生临时对象。但是函数 void Func(A &a) 存在一个缺点：“引用传递”有可能改变参数 a，这是不期望的。解决这个问题很容易，

加 const 修饰即可，因此函数最终成为 void Func(const A &a)。

以此类推，是否应将 void Func(int x) 改写为 void Func(const int &x)，以便提高效率？完全没有必要，因为内部数据类型的参数不存在构造、析构的过程，而复制也非常快，“值传递”和“引用传递”的效率几乎相当。

问题是如此的烦琐，只好将“const &”修饰输入参数的用法总结一下，如表 11-1-1 所示。

对于非内部数据类型的输入参数，应该将“值传递”的方式改为“const 引用传递”，目的是提高效率。例如将 void Func(A a) 改为 void Func(const A &a)。
对于内部数据类型的输入参数，不要将“值传递”的方式改为“const 引用传递”。否则既达不到提高效率的目的，又降低了函数的可理解性。例如 void Func(int x) 不应该改为 void Func(const int &x)。

表 11-1-1 “const &”修饰输入参数的规则

11.1.2 用 const 修饰函数的返回值

◆ 如果给以“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。

例如函数

```
const char * GetString(void);
```

如下语句将出现编译错误：

```
char *str = GetString();
```

正确的用法是

```
const char *str = GetString();
```

◆ 如果函数返回值采用“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。

例如不要把函数 int GetInt(void) 写成 const int GetInt(void)。

同理不要把函数 A GetA(void) 写成 const A GetA(void)，其中 A 为用户自定义的数据类型。

如果返回值不是内部数据类型，将函数 A GetA(void) 改写为 const A & GetA(void)的确能提高效率。但此时千万千万要小心，一定要搞清楚函数究竟是想返回一个对象的“拷贝”还是仅返回“别名”就可以了，否则程序会出错。见 6.2 节“返回值的规则”。

◆ 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。

例如

```
class A
{…
    A & operator = (const A &other);  // 赋值函数
};
A a, b, c;      // a, b, c 为 A 的对象
…
a = b = c;          // 正常的链式赋值
(a = b) = c;        // 不正常的链式赋值，但合法
```

如果将赋值函数的返回值加 const 修饰，那么该返回值的内容不允许被改动。上例中，语句 a = b = c 仍然正确，但是语句 (a = b) = c 则是非法的。

11.1.3 const 成员函数

任何不会修改数据成员的函数都应该声明为 const 类型。如果在编写 const 成员函数时，不慎修改了数据成员，或者调用了其它非 const 成员函数，编译器将指出错误，这无疑会提高程序的健壮性。

以下程序中，类 stack 的成员函数 GetCount 仅用于计数，从逻辑上讲 GetCount 应当为 const 函数。编译器将指出 GetCount 函数中的错误。

```
class Stack
{
  public:
    void    Push(int elem);
    int     Pop(void);
    int     GetCount(void)  const; // const 成员函数
  private:
    int     m_num;
    int     m_data[100];
};

int Stack::GetCount(void)  const
{
    ++ m_num;   // 编译错误，企图修改数据成员 m_num
    Pop();      // 编译错误，企图调用非 const 函数
    return m_num;
}
```

const 成员函数的声明看起来怪怪的：const 关键字只能放在函数声明的尾部，大概是因为其它地方都已经被占用了。

11.2 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 11-2-1】**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 11-2-2】**以提高程序的全局效率为主，提高局部效率为辅。

- **【规则 11-2-3】**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 11-2-4】**先优化数据结构和算法，再优化执行代码。

- **【规则 11-2-5】**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 11-2-6】**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。

11.3 一些有益的建议

✧ **【建议 11-3-1】**当心那些视觉上不易分辨的操作符发生书写错误。
经常会把“＝＝”误写成“＝”，象“||”、“&&”、“<=”、“>=”这类符号也很容易发生“丢 1”失误。然而编译器却不一定能自动指出这类错误。

✧ **【建议 11-3-2】**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 11-3-3】**当心变量的初值、缺省值错误，或者精度不够。

✧ **【建议 11-3-4】**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（让人们知道发生了什么事），避免让编译器轻悄悄地进行隐式的数据类型转换。

✧ **【建议 11-3-5】**当心变量发生上溢或下溢，数组的下标越界。

✧ **【建议 11-3-6】**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 11-3-7】**当心文件 I/O 有错误。

✧ **【建议 11-3-8】**避免编写技巧性很高代码。

✧ **【建议 11-3-9】**不要设计面面俱到、非常灵活的数据结构。

✧ **【建议 11-3-10】**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 11-3-11】**尽量使用标准库函数，不要“发明”已经存在的库函数。

✧ **【建议 11-3-12】**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 11-3-13】**把编译器的选择项设置为最严格状态。

✧ **【建议 11-3-14】**如果可能的话，使用 PC-Lint、LogiScope 等工具进行代码审查。

附录 A ： C++/C 代码审查表

文件结构		
重要性	审查项	结论
	头文件和定义文件的名称是否合理？	
	头文件和定义文件的目录结构是否合理？	
	版权和版本声明是否完整？	
重要	头文件是否使用了 ifndef/define/endif 预处理块?	
	头文件中是否只存放“声明”而不存放“定义”	
	……	
程序的版式		
重要性	审查项	结论
	空行是否得体？	
	代码行内的空格是否得体？	
	长行拆分是否得体？	
	“{”和“}”是否各占一行并且对齐于同一列？	
重要	一行代码是否只做一件事？如只定义一个变量，只写一条语句。	
重要	If、for、while、do 等语句自占一行，不论执行语句多少都要加“{}”。	
重要	在定义变量（或参数）时，是否将修饰符 * 和 & 紧靠变量名？	
	注释是否清晰并且必要？	
重要	注释是否有错误或者可能导致误解？	
重要	类结构的 public, protected, private 顺序是否在所有的程序中保持一致？	
	……	
命名规则		
重要性	审查项	结论
重要	命名规则是否与所采用的操作系统或开发工具的风格保持一致？	
	标识符是否直观且可以拼读？	
	标识符的长度应当符合“min-length && max-information”原则？	
重要	程序中是否出现相同的局部变量和全部变量？	

	类名、函数名、变量和参数、常量的书写格式是否遵循一定的规则？	
	静态变量、全局变量、类的成员变量是否加前缀？	
	……	
表达式与基本语句		
重要性	审查项	结论
重要	如果代码行中的运算符比较多，是否已经用括号清楚地确定表达式的操作顺序？	
	是否编写太复杂或者多用途的复合表达式？	
重要	是否将复合表达式与“真正的数学表达式”混淆？	
重要	是否用隐含错误的方式写 if 语句？例如 （1）将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较。 （2）将浮点变量用“==”或“！=”与任何数字比较。 （3）将指针变量用“==”或“！=”与 NULL 比较。	
	如果循环体内存在逻辑判断，并且循环次数很大，是否已经将逻辑判断移到循环体的外面？	
重要	Case 语句的结尾是否忘了加 break？	
重要	是否忘记写 switch 的 default 分支？	
重要	使用 goto 语句时是否留下隐患？例如跳过了某些对象的构造、变量的初始化、重要的计算等。	
	……	
常量		
重要性	审查项	结论
	是否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？	
	在 C++ 程序中，是否用 const 常量取代宏常量？	
重要	如果某一常量与其它常量密切相关，是否在定义中包含了这种关系？	
	是否误解了类中的 const 数据成员？因为 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。	
	……	
函数设计		
重要性	审查项	结论

	参数的书写是否完整？不要贪图省事只写参数的类型而省略参数名字。	
	参数命名、顺序是否合理？	
	参数的个数是否太多？	
	是否使用类型和数目不确定的参数？	
	是否省略了函数返回值的类型？	
	函数名字与返回值类型在语义上是否冲突？	
重要	是否将正常值和错误标志混在一起返回？正常值应当用输出参数获得，而错误标志用 return 语句返回。	
重要	在函数体的“入口处”，是否用 assert 对参数的有效性进行检查？	
重要	使用滥用了 assert？ 例如混淆非法情况与错误情况，后者是必然存在的并且是一定要作出处理的。	
重要	return 语句是否返回指向“栈内存”的“指针”或者“引用”？	
	是否使用 const 提高函数的健壮性？const 可以强制保护函数的参数、返回值，甚至函数的定义体。“Use const whenever you need”	
	……	
内存管理		
重要性	审查项	结论
重要	用 malloc 或 new 申请内存之后，是否立即检查指针值是否为 NULL？（防止使用指针值为 NULL 的内存）	
重要	是否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）	
重要	数组或指针的下标是否越界？	
重要	动态内存的申请与释放是否配对？（防止内存泄漏）	
重要	是否有效地处理了“内存耗尽”问题？	
重要	是否修改“指向常量的指针”的内容？	
重要	是否出现野指针？例如 （1）指针变量没有被初始化。 （2）用 free 或 delete 释放了内存之后，忘记将指针设置为 NULL。	
重要	是否将 malloc/free 和 new/delete 混淆使用？	
重要	malloc 语句是否正确无误？例如字节数是否正确？类型转换是否正确？	

重要	在创建与释放动态对象数组时，new/delete 的语句是否正确无误？	
	……	
C++ 函数的高级特性		
重要性	审查项	结论
	重载函数是否有二义性？	
重要	是否混淆了成员函数的重载、覆盖与隐藏？	
	运算符的重载是否符合制定的编程规范？	
	是否滥用内联函数？例如函数体内的代码比较长，函数体内出现循环。	
重要	是否用内联函数取代了宏代码？	
	……	
类的构造函数、析构函数和赋值函数		
重要性	审查项	结论
重要	是否违背编程规范而让 C++ 编译器自动为类产生四个缺省的函数：（1）缺省的无参数构造函数；（2）缺省的拷贝构造函数；（3）缺省的析构函数；（4）缺省的赋值函数。	
重要	构造函数中是否遗漏了某些初始化工作？	
重要	是否正确地使用构造函数的初始化表？	
重要	析构函数中是否遗漏了某些清除工作？	
	是否错写、错用了拷贝构造函数和赋值函数？	
重要	赋值函数一般分四个步骤：（1）检查自赋值；（2）释放原有内存资源；（3）分配新的内存资源，并复制内容；（4）返回 *this。是否遗漏了重要步骤？	
重要	是否正确地编写了派生类的构造函数、析构函数、赋值函数？注意事项： （1）派生类不可能继承基类的构造函数、析构函数、赋值函数。 （2）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。 （3）基类与派生类的析构函数应该为虚(即加 virtual 关键字)。 （4）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。	
	……	

类的高级特性		
重要性	审查项	结论
重要	是否违背了继承和组合的规则？ （1）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。 （2）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。	
	……	
其它常见问题		
重要性	审查项	结论
重要	数据类型问题： （1）变量的数据类型有错误吗？ （2）存在不同数据类型的赋值吗？ （3）存在不同数据类型的比较吗？	
重要	变量值问题： （1）变量的初始化或缺省值有错误吗？ （2）变量发生上溢或下溢吗？ （3）变量的精度够吗？	
重要	逻辑判断问题： （1）由于精度原因导致比较无效吗？ （2）表达式中的优先级有误吗？ （3）逻辑判断结果颠倒吗？	
重要	循环问题： （1）循环终止条件不正确吗？ （2）无法正常终止（死循环）吗？ （3）错误地修改循环变量吗？ （4）存在误差累积吗？	
重要	错误处理问题： （1）忘记进行错误处理吗？ （2）错误处理程序块一直没有机会被运行？ （3）错误处理程序块本身就有问题吗？如报告的错误与实际错误不一致，处理方式不正确等等。 （4）错误处理程序块是“马后炮”吗？如在被它被调用之前软件已经出错。	

重要	文件 I/O 问题： （1）对不存在的或者错误的文件进行操作吗？ （2）文件以不正确的方式打开吗？ （3）文件结束判断不正确吗？ （4）没有正确地关闭文件吗？	

第二部分 C++编码规范

第 0 章 为什么要用 C++

0.1 原因

为什么选择C++而不是C？或者更抽象一点，为什么选择面向对象语言，而不是面向过程语言或汇编语言?

这是一个很好的问题。有人可能心里知道一些，但说不清楚；有人可能会想到很多，并认为这是一个很泛泛的问题，说来话长。其实答案很简单：如果是一个技术人员在问这个问题，答案是“为了(更好地)复用代码”；如果是一个非技术人员在问(比如你的老板或是什么资本家)，回答只需两个字“省钱”，或者让他眼睛发亮的四个字“省很多钱”。

话虽不同，其背后的道理却是一样的。软件开发已经有几十年的历史了，每个人都知道这个行业最费人力，因为从开发到测试，再到维护，基本上以人的手工为主。我们还知道，软件开发人员从来都是高薪阶层。所以，软件的成本主要源于人的成本。那么如何降低成本?代码复用成了持续不断的主题。这是因为如果代码能够复用，则相应的开发时间、测试时间，以及分析修改时间都能节省下来，而这些时间都对应于软件人员的高薪。可见，代码复用率越高，成本削减的越多。

C++语言，或者说所有面向对象语言，就是针对代码复用设计的。我们可以列举一下面向对象语言的有名的特点：

封装：把具体实现封装在类内，而类内类外的代码只靠一些公共接口联系起来，类内实现接口的功能，类外使用接口的功能。目的是什么?类内实现变化了，可以不影响类外代码(复用)；类外使用代码变化了，可以不影响类内代码(也是复用)。

继承：子类可以继承父类的东西(复用)，也可以扩展自己新的特性，这些新特性不会影响父类，也不会影响使用父类的代码(复用)，甚至子类可以直接以父类的身份，使用所有父类可使用的代码(还是复用)。

多态：不同子类可以为同一个父类接口(复用)提供不同的实现，外部代码不需任何改动(复用)就可以拥有不同的特性。

0.2语言的发展和代码复用

一个主流语言的出现，或者说语言发展的每一次质的飞跃，其背后都有代码复用的影子。C语言取代汇编而流行，源于UNIX操作系统的开发。在这之前的操作系统，

基本上是用汇编写成的，而UNIX的90%是C，只有10％左右是汇编。带来的好处是，UNIX比其他操作系统更容易移植到不同的机器上，因为不必重写所有代码，90％只需重新编译(当然需要改动，现在看来改动应该还是一件艰巨的工作，但比起用汇编语言重头写要省事多了)。

C++较之C在代码复用上的能力更强。一方面，C++试图不加修改地整块(类)复用代码，而不像C那样需要逐行扫描修改(如同UNIX移植时)；另一方面，C++的复用接口(主要是类接口)更丰富、灵活、安全，而C主要依靠函数接口，包容性太窄，而函数间的联系也太弱。

我们再看看Java。它的流行不单是因为它也是一种面向对象的语言，还因为它在代码复用上有独到之处。我们基本上可以把一个网页看成一个程序，而浏览网页是将该程序从Internet上下载到本地机器上运行。但这个程序比较特殊，它要求能运行在所有平台上(尽可能)。我们很难用C++来写这样的网页，因为C++编译器只能编译出适应一种平台(CPU)的执行代码。另一种方式是用诸如HTML之类的语言，它们的特点是将源码下载到客户端，再由客户端解释执行。我们确实希望网页一次开发，能(复)用到所有平台，但有时我们不想公开源码(这实际上是不同组织间的代码复用问题，和一个组织内部的复用还不同，因为有商业利益或版权问题)。此时，Java可以帮助我们。它实际上是设计了一个虚拟平台(CPU)，所有Java语言源码都会编译成可运行在这个虚拟平台上的二进制执行程序，而客户端的Java解释器负责将这个虚拟平台的程序指令解释成真正平台的指令。可见，我们共享了目标级代码。

0.3 代码复用的特点

从以上的分析和我们自己的开发经验可以得出代码复用的两个特点。一是代码复用是一个由简到繁、从局部到整体的不断发展的过程。由于软件本身的复杂性，我们不可能一下子把代码做到复用率极高的程度。复用的经验和手段是一个逐渐积累的过程。具体到我们自己的程序，不要指望它们一下子成为代码复用的经典，从复用一个类、一个函数，甚至一两个好的编程风格或想法入手，日积月累，你手边会逐渐形成一个复用代码库，它将是你经验和财富的宝库。如果你所在的部门已经有几个人拥有复用代码库，那么恭喜你的部门，它可以成立一个技术委员会，负责收集和整合不同的复用代码库，这标志着你的部门已具备较高的专业水准和开发较大规模程序的能力。

代码复用的另一个特点是：为了复用，牺牲性能、“浪费”系统资源都不在话下。我们从汇编程序、C程序、C++程序，再到Java程序可以看出，复用性越强，性能越差(成倍下降)，程序尺寸越大(成倍增加)，但我们还是乐此不疲。究其原因，无非是性能、系统资源可以靠突飞猛进的硬件能力弥补，而硬件的成本比起人力资源根本不值一提。由此再顺便提醒大家一句，与其沉醉于程序性能的提高，不如关注代码的可复用性。

0.4 代码复用对我们的影响

回到本书的话题。本书列举了几百条编码原则，其实试图说明两个问题：一是如何防范错误；一是如何更好地发挥C++语言的特点。我自认为代码复用的境界更高，超出本书的范畴，但我还希望大家能思考一下，这几百条原则对代码复用有何帮助。

另一方面，大家不妨回忆一下自己开发的C++代码，看看它们有多大程度的可复用性。如果不高，我们能不能问一下自己：“我有充分的理由用C++吗?效率的损失、资源的浪费都值得吗?”我们还可以考虑一下今后的项目。

第1章　命名原则

好的命名原则在软件开发中是很重要的，尤其在可复用代码中更是如此。好的命名应该是直观而容易理解的，在移入新环境或上下文后仍能保持这种清晰的特点，并且不易与其他组件产生名字冲突。

原则1.1　关于类型名

1.1.1　说明

类型名中每个英文单词的第一个字母大写，其他小写，最后以_T结尾。类型名包括class、struct、union、typedef、enum以及namespace的名字。(同时参阅“原则1.9 命名时避免使用国际组织占用的格式”。)

注意：缩写字当作普通字处理，即只有首字母大写。

1.1.2　例子

```
//类名
class TnppCoverageArea_T
{
    //…
};
//枚举类型名
enum PageCode_T
{
    //…
);
//自定义类型名
typedef short Int16_T;
```

1.1.3　原因

●防止与变量名冲突(变量名定义请见“原则1.2　关于变量和函数名”)：

这种类型名定义方法和变量名会有两处不同，一是名字的第一个字母大小写不同，二是类型名以_T结尾。

●使得类型名更加清晰，尤其_T可以突出表示这是一个类名(T代表TYPE的意思)：

_T来源于POSIX的类犁命名：POSIX统一采用_t命名其类名。为沿用其思想而又防止和POSIX软件包冲突(请参阅“原则1.9　命名时避免使用国际组织占用的格式”)而采用_T。

●区分名字中各单词也可用下划线，但用大写字母会使得名字短些。

●缩写字当作普通字处理：

一是为了防止破坏该原则而造成混淆，请比较GPSReceiver_T和GpsReceiver_T

谁更清楚;二是为了防止和全大写的常量名等混淆(全大写的名字请见“原则1.4 关于宏、常量和模板名”)。

●因为namespace是表示一个逻辑组，与class或enum的某些用法类似，所以采用同样的命名原则。

原则1.2 关于变量和函数名

1.2.1 说明

变量和函数名中首字母小写，其后每个英文单词的第一个字母大写，其他小写。(同时参阅：“原则1.7 关于全局命名空间级标识符的前缀”、“原则1.9 命名时避免使用国际组织占用的格式”以及函数名的例外“原则1.3 关于全大写的函数名(建议)”。)

注意：缩写字当作普通字处理，即只有首字母大写。

1.2.2 例子

```
//变量名
int flexPageCount;
//函数名(length)
class String_T
{
    public:
        int length( void) const;
    //…
};
//比较类型名和变量名
GpsCommand_T gpsCommand;
```

原则1.3 关于全大写的函数名(建议)

1.3.1 说明

有一类函数，它们调用普通函数，只是对普通函数的错误返回做一般化处理。这些函数的名字要和所包含的函数名相同，只是全用大写字母(必要时用下划线分隔名字中的英文单词)。

1.3.2 例子

```
//罗嗦的用法
FILE* pFile = fopen("abc.txt", "rw+");
if(pFile == NULL)
{
    //错误处理：打印错误信息等
    abort();
}
```

```
    ret = fseek(pFile,0,SEEK_END);
    if( ret != 0)
    {
        //错误处理：打印错误信息等
        abort();
    }
    ret=fwrite(buffer, 100, 1, pFile );
    if( ret < 100 )
    {
        //错误处理：打印错误信息等
        abort();
    }
//简洁的用法
inline FILE* FOPEN( char const *fileName, char const* mode )
{
    FILE* pFile = fopen( fileName, mode);
    if( pFile == NULL )
    {
        //错误处理：打印错误信息等
        abort();
    }

    return pFile;     //正常则返回相应的文件句柄
 }
 //FSEEK()和FWRITE()依此类推，因而上面罗嗦的代码变成
 FILE* pFile = FOPEN( "abc.txt", "rw+");
 FSEEK(pFile, 0, SEEK_END);
 FWRITE(buffer, 100, 1, pFile);
```

1.3.3 原因

●阅读简洁，读者的思路不会总是被(非主流的)错误处理代码打断。

●此类函数与原函数名只有大小写的区别：

●不会改变原函数名的意思。

●由于区别小，阅读时有可能误以为就是原函数，从而减少了新加的函数给阅读者造成的额外负担(注意：区别不能太小，否则录入时容易出错)。

●阅读时即便误以为是原函数也不会出问题，因为它们的功能完全相同，区别只是错误处理方式。

●效率高：

●可用inline方式实现这类函数，相比上面“罗嗦的用法”效率　点也不损失。

●当在程序中要反复做错误检查时，这类函数会节省许多编程时间。

原则1.4　关于宏、常量和模板名

1.4.1　说明

这些名字要全部大写：如有多个单词，用下划线分隔。

宏指所有用宏形式定义的名字，包括常量类和函数类：常量也包括枚举中的常量成员。

1.4.2　例子

```
//常量类的宏PIE
#define PIE 3.1415926
//函数类的宏MAX
#define MAX(a, b)(/*…*/)
//常量LENGTH
const int LENGTH = 1024;
//枚举中的常量成员BLUE、RED、WHITE
enum
{
    BLUE,
    RED,
    WHITE
};
//模板名TYPE_T
templete<class TYPE_T>
class List_T
{
    public:
        void add(TYPE_T const& value);
        //…
};
```

1.4.3　原因

●使得此类名称更加清晰，防止与其他类型的名字(主要是变量名)冲突。

●函数宏之所以也用此原则是因为它在预编译时被展开(与普通函数有很大的不同)，需要阅读的人意识到这一点(这是沿袭了C语言的规范)。

●模板用此原则是因为此种类型名非常近似于“#define<name>”。

原则1.5　关于指针标识符名(供参考)

1.5.1 说明

建议以p开头或以Ptr结尾。

1.5.2 例子

```
//指针变量名pName
char* pName;
//函数指针类型的名字CallbackFunctionPtr_T
typedef int (*CallbackFunctionPtr_T)(int parameter);
```

1.5.3 原因

使阅读者不用查定义就能意识到这是一个指针：

对指针的操作与其他类型变量有很大不同，比如对其成员的访问要用“->”而不是“.”，特别是可能需要考虑内存回收的问题等，所以应给阅读者和使用者一个较明显的提醒。

原则1.6 关于变量名前缀(供参考)

1.6.1 说明

用下面不同的前缀来修饰变量名以区分不同的作用域：

i_ 类内数据成员(对象级成员)：

c_ 类内静态数据成员(类级成员)：

g_ 全局变量；

f_ 文件作用域变量(静态变量)。

函数内部等局部变量前不用前缀。

1.6.2 例子

```
class Message_T
{
    //类内静态数据成员c_id
    static int c_id;
    //类内普通数据成员(对象级)i_id
    int i_id;
  public:
    void someFunction( void )
    //函数内的局部变量id
    int id;
    //…
};
//全局变量g_id
int  g_id;
//静态变量(文件作用域)f_id
```

```
static int f_id;
```

1.6.3 原因

●减少作用域重叠时(不同作用域中的)变量名冲突问题。 比如在上例someFunction()中，变量id、i_id、c_id、f_id、g_id都可见，若无前缀，名字就都相同(冲突)了。

●这样也减少了为避免名字冲突而无规律可循地改变变量名。例如，将someFunction()中的变量id改名为myId,阅读者可能无法一目了然地知道为何要在Id前加my，会猜测有无特殊含义等。

●使得阅读者在读某个作用域代码时更清晰，知道碰到的变量是在哪个作用域中定义的。

●前缀的含义：

i_表示instance scope;

c_表示classscope;

f_表示filescope;

g_表示global scope。

●下划线前缀的用法：

有些人喜欢用下划线前缀修饰类内私有成员，而另一些人则用它表示全局变量。但遗憾的是它们都与ISO组织关于下划线前缀用法相冲突(见“原则1.9命名时避免使用国际组织占用的格式”)。

●常量前缀的处理方法：

常量和变量一样，也有作用域和名字冲突问题，因此也适用于此原则。不过由于常量名是全大写，前面加上小写的前缀稍嫌混淆。另一变通方法是将常量前缀改为大写，但造成此条规则复杂化(5项前缀变9项)，可能给阅读者造成更大的麻烦，建议还是不要这样做。

●类型前缀：

有些组织对于变量前缀定义得更细致，比如用n代表整数类型(int，short，long等)、用c代表字符型等，就如同本书用p作为指针类型的前缀一样。这些都是可以的。但需要提醒的是，变量命名(以及所有的命名)有两个极端，一个是什么前后缀都不加(精炼但有用信息过少)，另一个是加上所有相关的信息(Package名、作用域提示、类型提示等，信息全面但显得啰嗦，且重要信息不突出)，合理的命名方式肯定在这两个极端中间的某处，但具体在哪里有赖于大家自己的判断。不过，一个组织或部门内部最好统一意见，以方便大家相互理解和交流。

原则1.7 关于全局命名空间级标识符的前缀

1.7.1 说明

给全局命名空间(匿名，全局变量缺省所属的那个命名空间，以下同)级标识符一个公共前缀(如所属Package名或Library名，加下划线)，用来区别其他

提供类似功能的Packet或Library等。

全局命名空间中的标识符指的是全局或文件级变量名、常量名、宏名、类型名、函数名等。

前缀格式：全大写字母，(最好)少于3个字母。

1.7.2 例子

```
//HA项目中的代码
class HA_CheckPointTable_T …
Class HA_HashMap_T …
//UML函数库(第三方提供的标准函数库)中的代码
class UML_HashMap_T…
```

1.7.3原因

●如果希望代码复用，则全局命名空间级标识符就需要防止命名冲突。用Packet名或Library名是—个不错的选择。

●选择全大写是为了醒目，并尽量与其后的真正名字区别开来(因为真正名字才包含该标识符的意义、用途等)。

●限制其长度是为了防止它干扰对后面真正的名字的理解(要知道该前缀只是为了防止命名冲突，不能喧宾夺主)。

原则1.8 减少全局命名空间级标识符

1.8.1 说明

尽量减少全局命名空间级变量、常量、宏及函数等标识符。可以归类放在某个命名空间、类或函数中。

1.8.2 例子

```
class CommonDefinition_T …
{
public:
 const float PIE;     //现在常量PIE不再是全局的
//…
};
```

1.8.3 原因

●明显减少命名冲突。

●归类后使用更清晰。

●缩小其有效周期/范围，使之只存在于它应该发挥作用的有限时间/空间内，从而减少麻烦：更好记、更好维护、不会被误用或滥用等。

原则1.9 命名时避免使用国际组织占用的格式

1.9.1 已知的被占用的格式

●双下划线开头　　ISO C++、ANSI C；
●包含双下划线　　ISO C++
●单下划线开头　　ISO C++、ANSI C；
●E[0-9A-Z]开头　ANSI C；
●is[a_z]开头　　ANSI C；
●to[a_z]开头　　ANSI C；
●LC_开头　　ANSI C；
●SIG[_A_Z]开头　ANSI C；
●str[a_z]开头　　ANSI C；
●mem[a_z]开头　　ANSI C；
●wcs[a_z]开头　　ANSI C；
●_t结尾　　POSIX；
●其他国际组织占用的格式。

1.9.2　原因

●减少潜在的命名冲突。
●防止阅读者误以为是国际组织提供的代码。

原则 1.10 名字要本着清楚、简单的原则

1.10.1　说明

名字本身首先要做到清楚，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，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。

1.10.2　例子

```
//不好理解的名字
int shldwncnt;
int rs;
int num;
//比较一下
int shellDownloadCount;
int returnStatus;
int alarmNumber;
```

1.10.3　定量分析的参考

可以将名字长度限制在3到25个字符之间，并据此编写或利用现成的工具自动扫描代码，以检查名字是否做到“简单、清楚”。少于3个字符：通常不够清楚(选择3是因为lhs、rhs之类的名字应该足够清晰，见“原则1.26　关于函数的左值参数和右值参数名”)；大于25个字符则（感觉上)嫌不够简单。

下限的3个字符应该不包括公共前缀(如package名HA_、变量作用域i_、指针前缀p)、公共后缀(如类型名中的的_T)以及下划线等，为的是确保名字中真正核心的部分足够清晰；而上限的25个字符应包含名字中所有字符。

原则1.11 尽量用可发音的名字

1.11.1 例子

```
//不可发音的名字
class Ymdhms;
//可发音的名字
class Timestamp_T;
```

1.11.2 原因

可发音的名字更好读、更好懂，尤其是更便于交流。

原则1.12 尽量用英文命名

原因

英语是最通用的语言，特别是在程序语言中，其他语言(比如选择汉语拼音)可能造成阅读者理解上的困难，也不利于更大范围的代码复用。

原则1.13 尽量选择通用词汇并贯穿始终

1.13.1 说明

许多单词都能表达一个含义，要选择一个最通用的<大家在编写类似代码时约定俗成的)，并且始终保持这一用法。

1.13.2 例子

比如get、read、fetch、retrieve都能表达“取出”的意思，在定义一个有“取出”功能的函数时函数名用get或read较为常见。

一旦定下使用哪一个(比如read)就坚持用到底。如果一会儿用read，一会儿又用get，读者可能会误以为这两个函数有区别。

原则1.14 避免用模棱两可、晦涩或不标准的缩写

1.14.1 原因

好的缩写会明显简化名字，还能清楚地表达出原意。

不好的缩写则相反，不如不要，就算因此导致名字长度倍增也在所不惜。此时要记住：(任何情况下都是)录入易，维护难。

所谓好与不好，最简单的标准就是看该缩写是否通用，越通用越好。

1.14.2 例子

```
class Pgh_T;     //是paragraph的缩写吗?
```

原则1.15 避免使用会引起误解的词汇

例子

比如用portList来描述一组port，如果其数据结构不是链表就不合适，因为约定俗成List就是指链表，不要小看这一点，它会减少很多头疼的事。

原则1.16 减少名字中的冗余信息

1.16.1 例子

比如类的成员名不需要包含类名：

```
class Alarm_T …
{
 Severity_T alarmSeverity( void ) const;    //用severity()就足够了
};
```

1.16.2 原因

不要让阅读者花额外精力来区分哪些是有用信息，那些是冗余信息。

原则1.17 建议起名尽量通俗，太专一会限制以后的扩展

例子

比如定义了一个手机短语消息类，其中有一个域表示消息来源，用sourceSubscriberID做名字就不如用source通用。

要预知将来的扩展有时很难，但在初次定义时想一下今后可能的扩展不是坏事。

原则1.18 名字最好尽可能精确地表达其内容

例子

没人知道data、info或stuff的内容到底是什么。

原则1.19 避免名字中出现形状混淆的字母或数字

例子

```
/*---------------------------------------
*字母O和数字0形状类似，避免混用：实在无
*法避免，字母o最好总是用小写，这样和数字
*0区别还大一点
---------------------------------------*/
#define F00
#define FoO
/*---------------------------------------
```

```
*同样，字母l和数字1避免混用：无法避免时
*字母l最好总用大写形式L
----------------------------------------*/
const long VALUE = 0x43211;
const long VALUE = 0x4321L;
```

原则1.20 命名类和成员使得“object.method()”有意义

1.20.1 例子

```
timer.clear();
timer.Start();
```

1.20.2 原因

这是增加代码可读性、减少冗余信息的一种方法。

原则1.21 类和对象名应是名词

1.21.1 例子

```
class Alarm_T;
int length;
```

1.21.2 原因

类和对象名用来标明事物，而事物应是名词。

原则1.22 实现行为的类成员函数名应是动词

1.22.1 说明

实现行为的类成员函数应是动词，有时能还包含“直接对象”(如例子中的position)，此时要把“直接对象”名放在动词之前。

1.22.2 例子

```
class Gps_T
{
  void positionCalculate(void);
  void reset( void );
};
```

1.22.3 原因

“直接对象”很有可能是另—个实现类，放在前面可以提醒读者(提供线索)。

原则1.23 类的存取和查询成员函数名应是名词或形容词

1.23.1 说明

存取函数是用来读取和修改对象属性的，与属性本身同名显得自然而不罗嗦。

查询函数用于返回对象的信息。非布尔型查询函数应是名词，如size()。布尔型查询函数应是形容词，通常用is作前缀，如isEmpty()。

1.23.2 例子

```
class Shape_T
{
    public:
   //存取函数
   Color_T color( void ) const;    //colorGet()、getColor()显得罗嗦
   void color(Color_T const& aColor); //colorSet()、setColor()显得罗嗦
      //非布尔型查询函数
      int area( void ) const;
     //布尔型查询函数
     bool isVisible( void )const;
    };
```

原则1.24 变量名应是名词

1.24.1 说明

类的成员名(变量名、常量名等)应是名词。(可能带若干形容词作修饰，名词要放在最前面起突出作用。)

1.24.2 例子

```
    class Gps_T
    {
    //类的成员
     Almanac_T i_almanac;
    int i_satellitesVisible;
    };
```

原则1.25 布尔型的名字要直观

1.25.1 说明

is通常是一个不错的前缀，好不好可用if语句来检验。

1.25.2 例子

```
   class Queue_T
   {
     public:
     //布尔型函数名
     bool isFull( void ) const;
```

```
    bool contains( Object_T anObject ) const;
  };
  //用if语句检查，证明is开头不错
  if(queue.isFull())…
  //用if语句检查，此时用is不合适
  if(queue.contains(thisObject))…
```

原则1.26 关于函数的左值参数和右值参数名

1.26.1 说明

用lhs做左值参数的名字，用rhs做右值参数的名字。

1.26.2 例子

```
//类的拷贝构造函数
MyClass_T::MyClass_T( MyClass_T const& rhs );
//赋值函数
int operator=( String const& lhs, String const& rbs )
```

1.26.3 原因

●这是被广泛使用的一种编程约定。

●既通用又省事，不必再费神给此类参数起“清楚”、“简单”的名字了。

原则1.27 避免局部名和外层的名字冲突

1.27.1 例子

```
//time.h定义了time()函数
#include <time.h>
int test( void )
{
  //不好的定义，和外层的time()函数同名
  Time time;
  //好的定义
  Time timeStart;
};
```

1.27.2 原因

●这种冲突会给读者带来不必要的混淆。

●有可能成为真正的bug且非常难查。

原则1.28 用a、an、any区分重名 (参数)

1.28.1 例子

```
//函数名、参数名和类成员变量名类似
```

```
void MyClass_T::severity( int aSeverity )
{
  i_severity = aSeverity;
}
```

1.28.2 原因

●有效(避免冲突)。

●名字的含义不变。

●代价小。

原则1.29 模板类型名应有意义

1.29.1 例子

```
//用T1、T2不好
template<class T1, int T2>
class Vector_T
{
  T1 i_data[T2];
};
//用TYPE_T和SIZE就清楚多了
  template<class TYPE_T, int SIZE>
  class Vector_T
  {
  TYPE_T i_data[SIZE];
  };
```

1.29.2 注意

注意遵守“原则1.1 关于类型名”的约定(_T)。

第2章　类型的使用

类型是代码中的基本数据单元。正确定义和使用恰当的类型会避免许多问题。

原则2.1　避免隐式声明类型

2.1.1　说明

尽量用显式声明。

2.1.2　例子

```
main();     //是int main(void)的隐式表示法
void foo(const aValue);  //是void foo(const int aValue)的隐式表示法
```

原则2.2　慎用无符号类型

2.2.1　说明

●避免使用无符号类型，除非真的需要。

●按位访问的数据和设备寄存器通常要用无符号类型。

2.2.2　原因

●混用有符号和无符号类型会导致奇怪的结果，因为其中会发生隐式类型转换。

●不同的C++标准中有符号和无符号的转换规则不同。

原则2.3　少用浮点类型除非必须

2.3.1 例子

```
int baudRate=9600;
int symbolslnl5msec;
//使用了浮点类型，能避免吗
symbolslnl5msec=(int)(baudRate*0.015)；
//看来可以
symbolslnl5msec=(baudRate*15)/1000;
```

2.3.2　原因

●浮点数不精确：

因为计算机内部是二进制表示法，需要将代码中十进制浮点数转换成二进制。由于字长的限制，经常不得不丢掉最低的几位小数，所以结果是不精确的。

●浮点类型的异常处理复杂(比如上溢、下溢等的处理)。

●运算速度慢。

原则2.4　用typedef简化程序中的复杂语法

2.4.1 例子

```
//用typedef简化函数指针
typedef int(*CallbackFunctionPtr_T)(int parameter);
```

2.4.2 原因

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，使得阅读理解都更容易。

2.4.3 定量分析的参考

包含4个以上独立元素的语法应被视为复杂语法，如上例中的函数指针定义，不算typedef的独立元素数为5(int，*，CallbackFunctionPtr_T，int，parameter)。

原则 2.5 少用union

原因

- 由于union成员公用内存空间，所以容易出错，并且维护困难。
- 使用union通常意味着非面向对象的方法。

原则2.6 慎用位操作

原因

- 效率低下。
- 可能带来兼容性问题。

原则2.7 用enum取代(一组相关的)常量

2.7.1 例子

```
//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常量放在一起
enum
{
    EMPTY_ENTRY=-1,
    NOT_FOUND=-1,
    MAP_FULL=-2
};
```

2.7.2 原因

- 易于维护：

枚举可以对其成员自动编号，这样增加或减少成员等修改/维护工作就很方便。

●比#define或int const更安全：

因为编译器会检查每个枚举值是否位于取值范围内。

●在类中使用更方便：

如果用常量，要等到在构造函数中初始化后才能使用，(大家可以试一下，这在很多情况下不方便。)而枚举不用。

●符合“原则1.8　减少全局命名空间级标识符”。

原则2.8　使用内置bool类型

2.8.1　说明

使用内置bool类型，而不是自己定义或用int代替。

2.8.2　原因

● 内置bool类型比int更安全强壮。

● 只有true和false两个值，而不是零和非零。

原则2.9　（尽量）用引用取代指针

2.9.1　说明

在下述情况下用引用优于指针：

●被引用的对象永远不可能是空(NULL)。

●引用某一个对象后决不会再去引用其他对象。

●(某些)operator函数的返回值，如operator[]，operator+=等。

●函数的参数传递。

但在下述情况下用指针更好：

●NULL是合法的参数值。

●其他情况见“原则3.6　关于何时用指针传递参数”。

2.9.2　原因

●引用更安全，因为它一定不为空，且一定不会再指向其他目标。

●不需要检查非法值(如NULL)情况。

●使用更简洁(“.”比“->”好用)。

●不需要解除引用：

与此相反，当所指的内存被释放后，指针应有一个合理的值，一般是让它指向NULL。

第3章　函数

集中精力写出定义清晰、文档完备、行为良好的函数，会使你的代码易于使用和维护。

原则3.1　函数一定要做到先声明后使用

3.1.1　说明

C++必须这样做(否则编译通不过)。C程序没有强制要求，但也应该先提供原型，再使用函数。

3.1.2　原因

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)。

原则3.2　函数原型声明放在一个头文件中

原因

将同类/相关的(非成员)函数的原型声明集中存放在一个头文件中，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。

原则3.3　函数无参数一定要用void标注

3.3.1　例子

```
int foo(void);  //比“int foo();”好
```

3.3.2　原因

●C++和C对function()的解释不同。C++认为是不带参数；C认为是带任意参数(虽然ANSI C现在已经废除这一规则，但其他标准和编译器不一定能保证这一点)。

●显式使用(void)消除了C++和C混编时可能出现的潜在错误。

原则3.4　对于内置类型参数应传值(除非函数内部要对其修改)

3.4.1　说明

内置类型指int、char等<相对于自定义的class、struct、unit)。

3.4.2　原因

●传值既安全又简单。

●内置类型拷贝的代价与传指针或引用相同，因为(绝大多数)内置类型所占内存小于等于指针或引用所占内存。

原则3.5　对于非内置类型参数应传递引用(首选)或指针

3.5.1　说明

非内置类型指的是自定义的类(class)、结构(struct)和联合(union)。

如要防止参数被修改，可用const修饰。

引用/指针的选择请参看“原则2.9 (尽量)用引用取代指针”。

3.5.2 原因

●不用拷贝：

非内置类型的尺寸一般都大于引用和指针类型，尤其还要考虑非内置类型可能包含隐式的数据成员，比如虚函数指针等，所以拷贝的代价高于传递引用和指针。

●不用构造和析构：

如果拷贝对象，还要在传入时调用(拷贝)构造函数，函数退出时还要析构。

●有利于非内置类型的扩充：

对于小对象虽然传值代价也不大，但将来的修改/扩充可能使这一优势丧失，到时再漫山遍野地将函数接口从传值改成传引用/指针就太费劲了。

●有利于函数支持派生类：

若将派生类对象传给以基类为参数的函数(传值方式)，就会导致派生类对象被“切割”成基类对象，这样在函数内部实际上用的是一个基类对象的拷贝，而不是最初传入的派生类对象。这多半不是你的本意。

这种错误非常难查，因为它是编译器(隐式/自动)做的；编译器也不会报警，因为这是合法的。

如果传入的是引用或指针，则不会有对象“切割”现象。函数内部可以将传入的对象视作基类对象使用(因为任何派生类对象都可以作为基类对象)。如果传入的对象有虚函数，则恰当的实现版本还会被正确调用，而不局限于传入的参数类型(基类)。

原则3.6 关于何时用指针传递参数

3.6.1条件

●若函数内部须将自己的参数以指针形式传给其他的函数：

因为至少你不能确定那些(需要指针的)函数一定不需要传NULL。你若在外面强行用引用，它们就再也无法获得NULL(作为参数值了)。

●若参数是被new出来的，且将要在函数内被释放：

如果用引用，则会出现这样的语句“delete &reference”，看起来有点怪(但不是绝对不可接受)。

3.6.2 原因

●总的来说，引用比指针好，但指针也不是一无是处。

●取舍的一个关键是：NULL是否是一个合法的取值。

原则3.7 避免使用参数不确定的函数

3.7.1 例子

```
MyString_T hi("Hello, World.");
/*-------------------------------------------------------------
*虽然MyString_T提供将其对象转换成字符串的成员函
*数operator char const*(), printf()还是打印不出
*你想要的字符串"Hello, World", 因为printf()带的
*是可变长的参数，编译器不会将hi隐式转成字符串。
*这句话最终会把hi所占内存的内容打出来
-------------------------------------------------------------*/
printf("%s\n", hi);
/*-------------------------------------------------------------
*这样就行，因为这句话实际上是调用两次操作符<<函数
*每个函数都只带一个类型已知的参数，此时编译器会调
*用MyString_T的隐式转换函数，将其转成字符串类型，
*然后再传给操作符<<函数
-------------------------------------------------------------*/
cout << hi << endl;
```

3.7.2 原因

●参数不确定的函数有隐患：

因为参数不确定，编译器就不能检查参数的个数和类型，这会带来很多问题(比如上例中不能做隐式转换等)。

●C++有很好的解决办法：

用重载和链式函数。比如printf()可以变成一系列只带一个参数的函数(比如operator<<()函数)，每个函数都只接受一种printf()支持的类型，这些函数互为重载(函数名相同)。其结果是，任意一个printf()调用都可以转化为几个重载函数的连续调用。

可以看出，转化后更安全，因为每个重载函数的参数个数和类型都固定。同时也更灵活，因为增减一个类型只需增减一个重载函数，而不像printf()那样需要改函数实现和接口(接口上要相应增减类型指示符，如%s、%d等)。

这种用法的缺点是效率稍差。

原则3.8 若不得不使用参数不确定的函数，用<stdarg.h>提供的方法

3.8.1 说明

一定要明确这是万不得已的办法(参见"原则3.7 避免使用参数不确定的函数")。

3.8.2 例子

```
#include<stdarg.h>
```

```
void someFunction(char const*pFormat, va_list varArgs);
void varFunction(char const*pFormat, …)
{
  va_list varArgs;
  va_start(varArgs, pFormat);
  someFunction(pFormat, varArgs);
  va_end(varArgs);
}
void someFunction(char const*pFormat, vs_list varArgs)
{
  //first argument determined t0 be a char*
  char* pName=va_arg(varArgs, char*);
  //…
}
```

3.8.3 原因

<stdarg.h>中定义的宏提供了一种安全有效的访问参数链的方法。

原则3.9 避免函数的参数过多

3.9.1 原因

●使用麻烦。

●不易理解和维护。

●通常表明是一个不好的设计。

3.9.2 定量分析的参考

一个函数的参数应该限制在5个以内。

原则3.10 尽量保持函数只有唯一出口

3.10.1 原因

●单—出口易于维护：修改代码容易，不易因为忘记修改某处出口而产生问题。

●易于跟踪调试：可设单一端点跟踪函数(出口)。

3.10.2 缺点

可读性稍差(比如当有许多嵌套条件语句时)。

3.10.3 结论

根据情况自己权衡，但尽最大可能保持一个出口。

原则3.11 显式定义返回类型

3.11.1 例子

```
//隐含的返回类型是整型，不好
```

```
Password_T:: length(void);
//显式声明，好
int Password_T::length(void);
```

3.11.2 原因

●直观，所以容易阅读。

●提醒自己和阅读者注意正确的返回类型：

有可能阅读者没有仔细想，误以为没有返回值，结果造成理解上的偏差。

●避免让人怀疑是忘了写还是真想用缺省方式。

原则3.12 (非void)任何情况都要有返回值

3.12.1 说明

任何非void函数在任何情况下都要返回某个值。

3.12.2 例子

```
int valueGet(int const* pValue)
{
    if(pValue != NULL)
    {
      return *pValue;
    }
}//else时会返回什么
```

3.12.3 原因

这是一个程序错误，实际上返回值还是有，只是成为随机值。有的编译器会报错。

原则3.13 若函数返回状态，尝试用枚举作类型

原因

返回枚举类型可以使编译器对返回值做合法性检查(看看是不是枚举的合法成员)。

原则3.14 返回指针类型的函数应该用NULL表示失败

3.14.1 原因

●NULL是唯一合理的表示错误的指针返回值。

●同时也防止调用者用未初始化或已失效的指针进行随后的操作。

3.14.2 例子

```
int* myFunction(void)
{
  //…
  if(“something wrong”)     //如果遇到问题，函数需要失败返回
    {
```

```
        return NULL;     //返回NULL表示失败
      }
    }
    //…
    int *pCount=myFunction();
    if(pCount!=NULL)     //现在有办法知道函数是否成功
    //…
```

原则3.15　函数尽量返回引用(而不是值)

同时参见“原则3.16　若必须返回值，不要强行返回引用”。

3.15.1　例子

```
Item_T& List_T::head(void)     //返回的是引用(reference)
{
   return i_head;
}
```

3.15.2　原因

与“原则3.5　对于非内置类型参数应传递引用(首选)或指针”的原因相同。

原则3.16　若必须返回值，不要强行返回引用

3.16.1　说明

●不要返回局部变量的引用，因为当函数返回后它就不存在了。

●最好不要返回new出来的对象的引用，因为：

●new出来的对象总要被删掉，而函数的返回值是匿名的(临时的)，要保证不出内存泄漏(memory leak)，调用者必须记下每个返回值，因为它们引用的是new出来的内存，当不再使用时要回收。通常这是一件困难的事情(见例2的sum)，有时甚至根本就做不到(见例2的最后一句注释)。

●当new出来的对象被删掉后，那个引用已经是无效的了，此时只能靠操作者记住这件事，这样非常容易导致错误。

3.16.2　例子

●例1，错：返回局部变量的引用

```
Complex_T& operator+(Complex_T const& lhs, Complex_T const& rhs)
{
  Complex_T result(lhs.i_real+rhs.i_real, lhs.i_imag+rhs.i_mag);
  return result;
}
```

●例2，非常不好：返回系统堆内存的引用

```
Complex_T& operator+(Complex_T const& lhs, Complex_T const& rhs)
```

```
  {
      Complex_T* pResult=
      new Complex_T(lhs.i_real+rhs.i_real, ihs.i_imag+rhs.i_mag);
      return *pResult;
  }
Complex_T one(1), two(2), three(3), four(4);
Complex_T sum=one+two+three+four;
//…
  delete &sum;  //编程者要记得sum用完后要删掉
  //现在怎么办?谁来删掉另外3个new出来的中间变量
```

3.16.3 优化

返回值时也还是可以做一定的优化，参见“原则24.9 返回对象(值)的优化”。

原则3.17 当函数返回引用或指针时，用文字描述其有效期

3.17.1 说明

有效期是指引用或指针能有效地找到目标的时间段。文字描述最好以注释形式放在函数所在的头文件中(这样比较直接)。

3.17.2 原因

有利于后来者正确使用、维护该函数。

原则3.18 禁止成员函数返回成员(可读写)的引用或指针

3.18.1 例子

```
class Alarm_T
{
  private:
     String i_name;
  public:
   //和将i_name定义成public访问权有什么区别
  String& name(void){return i_name;};
}
```

3.18.2 原因

●这破坏了类的封装性，造成对象的成员被修改，而该对象对此一无所知。

●关于能否返回只读(常量)的引用或指针,请参见“原则8.3 类成员可以转换成常量形式暴露出来”。

原则3.19 重复使用的代码用函数替代

3.19.1 例子

```
int main(int argc, char*argv[])
{
 if(argc(1)
 {
     cerr<<"Usage:"<<argv[0]<<"<filename>"<<endl;//重复使用的代码
    exit(1);
 }
 //…
if(wantHelp)
 {
    cerr<<"Usage: "<<argv[0]<<"<filename>"<<endl;//重复使用的代码
}
 //…
}
```

3.19.2 原因

- ●简化代码：调用时一堆(重复)语句变成一个函数调用。
- ●易于维护：不用在代码各处做同样的修改，也不再担心是否会有遗漏。
- ●如担心效率问题，可以使用inline方式。

原则3.20 关于虚友元函数

3.20.1 说明

如果我们能把友元函数定义成虚函数，则子类既可以继承该函数而无需重复声明友元，(要知道，父类的友元不会自动成为子类的友元；而且友元会破坏封装，要少用，参见"原则12.1 少用友元"。)又能提供独特的子类实现。这个功能很诱人，然而C++语法不允许(非成员的)友元函数为虚函数(参见"原则4.12 恰当选择成员函数、全局函数和友元函数")。不过我们可以通过编程技巧达到虚友元想达到的效果。

3.20.2 例子

```
/*-----------------------------------------------------------
*一般情况下，基类和派生类要加入友元operator<<()以便
*打印输出自己的对象。这里我们换一种实现方法：在基类
*中定义一个虚函数print()，由它来完成打印输出
-----------------------------------------------------------*/
class Base_T
{
 public:
  virtual void print(ostream& output)=0;
```

```
  //…
};
void Base_T::print(ostream& output)
{
  //将基类需要打印的数据成员送给output
}
/*-----------------------------------------------
*派生类提供自己的print()
-----------------------------------------------*/
class Derived_T: public Base_T
{
  public:
virtual void print(ostream& output);
  //…
};
void
Derived_T::print(ostream& output)
{
  //先将基类数据成员送给output
  Base_T::print(output);
  //将派生类需要打印的数据成员送给output
}
/*-------------------------------------------------------------
*用这种方法，operator<<()不需是任何类的友元，只需带
*基类引用做其第二个参数，并是负责调用虚函数print()
*即可。这样，给operator<<()传什么对象，它就能打印相
*应的信息出来
-------------------------------------------------------------*/
ostream& operator<<(ostream& output, Base_T& anObject)
{
  //调用anObject相应的print()函数
  anObject.print(output);
}
```

3.20.3 原因

●若在每个类中都声明operator<<()是友元并实现之，有些罗嗦。

●友元打破了类的封装性，现在这种方法不需声明任何友元了。

●编程者可以把基类的print()声明成纯虚(pure virtual)函数，以强迫派生

类实现 print()，而友元方式无此类控制方法。就是说，如果用友元方式，使用者不能随便使用"cout << anObject"，因为你不知道这个对象的类型有没有定义并实现了相应的友元函数operator<<()(没人强迫它这么做)；与此相反，在"虚友元"方式下，任何派生类(不沦是现有的，还是将来可能要加的)使用者都可以随时使用"cout<<anObject"，所有机制都已建好，唯一需要新加派生类做的是输出自己特殊的成员，而这项工作也有编译器强迫编程者实现，不会漏掉。

原则3.21 关于虚构造函数

3.21.1 说明

C++语法不允许构造函数为虚函数，但有时需要有这样的函数:它们能根据不同情况构造不同的(派生)类。这类函数实际起到虚构造函数的作用。

3.21.2 例子

```
/*-----------------------------------------------------------------
 *最简单的虚构造函数是虚拷贝构造函数
 *(virtual copy constructor)
 -----------------------------------------------------------------*/
 class Base_T
 {
 //…
   public:
 //根据自身(this)类型，克隆出一个新对象
 virtual Base_T& clone(void);
 //…
}
/*-----------------------------------------------------------------
*baseRef可以引用仟何派生类刘象，anotherBaseRef
*可以克隆出相同的一个对象(做其他事情)
-----------------------------------------------------------------*/
Base_T& baseRef=aDerivedObj;
Base_T& anotherBaseRef=aDerivedObj.clone();
```

3.21.3 原因

这不是偷换概念的把戏，是为了打开一种新的思路：能不能基于现有的C++语法，扩展出新的功能，从而更好地为我所用?

第4章　类的设计和声明

类是一切面向对象设计的基础。一个好的类应该提供干净、清晰的接口，具备低耦合(类间)、高内聚(类内)的特点，并且很好地展现封装、继承、多态、模块化等特性。

原则4.1　类应是描述一组对象的集合

说明

类应是描述一组对象的集合，而不是用来包裹一个简单对象(int、char等)。

原则4.2　类成员应是私有的(private)

4.2.1　说明

每个类不应让其他人插手其内部实现(状态)，所以类的成员不应是public:甚至连派生类也不应插手基类的内部，所以protected也应避免。

4.2.2　原因

●非private成员破坏了类的封装性，导致类本身不知道其成员何时被修改。

●更糟的是，任何对类实现的修改都可能影响使用该类的代码，因为这些代码有权直接访问(被修改的)类成员。

●例外：我认为，属于接口而不属于实现的某些常量成员可以不受此规则限制，也就是说可以是protected甚至是public，因为不论类内类外都不会修改这些成员，除非强制/显式去掉只读属性，而这是要极力避免的。(请参见"原则8.6　不要将常量强制转换成非常量"。)

原则4.3　保持对象状态信息的持续性

4.3.1　说明

确保对象的状态信息(成员变量)在(对象的)整个生命周期都有效。

4.3.2　例子

```
class String_T
{
    //…
    private:
    char* i_buffer;
    /*-----------------------------------------------------
    *编程者要确保变量i_length在String_T对象的整个生命周期
    *都能精确地表示i_buffer中的字符串长度
    -----------------------------------------------------*/
```

```
    int i_length;
};
```

4.3.3 原因

●这种信息一旦失效，对象就进入不可预测状态。

●任何有这种可能的对象都是不能被信任及使用的。

原则4.4 提高类内聚合度

4.4.1 说明

高内聚是指在进行类设计时要强调专注于一件事或一个目标，并把它做好。

4.4.2 原因

●高内聚是要让类围绕一个核心去定义接口、提供服务、与其他类合作，从而易于实现、理解、使用和维护。

●很容易去设计一个功能非常强大的类(目标众多)，这种欲望是永无止境的，其结果是实现上顾此失彼、使用时错综复杂，因为不同目标接口不同，与之合作的周边类也不同，情况交织，理解、使用、维护都成问题。

●即使目标单一，也要仔细推敲什么需要类内实现，什么需要放在类外。编程者很容易忽略这一点，但这是进一步提高内聚性、降低耦合度的有效措施。

●简单完善的类扩充很容易，但要控制/削减一个已经"强大"的类很难。

4.4.3 定量分析的参考

包含10个以上数据成员的类应被怀疑内聚性较差(可能没有专注于一件事或一个目标)。统计数据成员时不包括继承来的成员、静态成员和枚举(成员)，但应包含常量成员。

原则4.5 降低类间的耦合度

4.5.1 说明

低耦合是指降低类间相互依赖程度。

4.5.2 例子

经常看到这样的例子：一个大功能模块中有许多不同类型的对象，为了协同工作，每类对象内部都多少带有其他对象的指针或引用，造成你中有我，我中有你，最终导致一锅粥，谁也离不开谁。

4.5.3 原因

●类间的耦合度越高(联系越紧密)，独立性越差，移植、修改能力越差，且易出错，所谓"牵一发而动全身"。

●降低了类的模块化和封装性，因为每个类的实现都与其他类相关。

原则4.6 努力使类的接口少而完备

4.6.1 说明

●一个完备的接口是指使用者可以通过它(对类)做任何合理的事情。

●接口少是指接口函数尽可能少。

4.6.2 原因

●对于使用者来说，不完备的接口是不可用的。

●但另一方面，接口函数太多会难于理解、使用和维护。

4.6.3 定量分析的参考

包含20个以上成员函数的类应被怀疑为接口不够精简。统计成员函数时不包括继承来的且未作修改的函数、静态成员函数、禁止使用的成员函数(没有函数体，请参见"原则4.14 显式禁止编译器自动生成不需要的函数")，但应包括自己提供实现的虚函数。另外，统计时不用考虑成员函数的外部访问权限(private、protected或public)。

原则4.7 保持类的不同接口在实现原则上的一致性

4.7.1 说明

防止阅读者或使用者有意外的感觉。

4.7.2 例子

```
/*--------------------------------------------------------------------
*没有考虑现有iString的空间是否能容纳aString。
*这是一种实现方法(不能算错)，因为可能String_T
*就设计成i_string空间足够大，这样效率高、处理
*简单
--------------------------------------------------------------------*/
void String_T::assign( char const* aString)
{
    strcpy(i_string, aString);
i_length = strlen(aString);
}
  /*--------------------------------------------------------------------
  *这是另一种设计方法，它的适用性更强，不会溢出，也
  *不会有内存浪费。但不应该在同一个类的不同函数中用
  *不同的实现方法

--------------------------------------------------------------------*/
  void String_T::concat(char const* aString)
  {
    int length=i_length + strlen(aString);
    char* newString=new char[length+1];
```

```
    strcpy(newString, i_string);
    strcat(newString, aString);
    delete[] i_string;
    i_length=length;
    i_string=newString;
  }
```

4.7.3 原因

接口背后的实现不一致会引起:

●增加代码学习的时间。

●增大出错的可能。

●导致使用者对该代码的不信任(怀疑是否存在其他的不一致；对能否完全找出并掌握这些不一致没有把握)，从而阻碍代码复用。

●使用接口的代码也会跟着有差异，这会使该类(在使用上)看起来很古怪。

原则4.8 保持不同类的接口在实现原则上的一致性

4.8.1 说明

许多相似的类，其接口(主要是相同名字的接口)的行为应保持一致。

4.8.2 例子

所有带insert()接口的类，该接口的实现原则应该保持一致。

4.8.3 原因

●易学好记。

●在这些类之间变换，操作相似。

●容易想到将某些类归并成从同一基类派生出来的。

原则4.9 避免为每个类成员提供访问函数

原因

●如果每个类成员都有访问函数，那就类似于将所有成员都定义成public。(请参见"原则4.2 类成员应是私有的(private)"。)

●对象的主要目的是捕捉发生在其上的行为，而不是像c语言的struct那样专注于属性(成员)；为每个类成员提供访问函数将导致类的退化并偏离其主要目的。

●专注于属性的类在多线程环境下效率低下。若专注于行为， 一次行为会修改一连串相关属性而只需上一次锁(在一个成员函数中)；而专注于属性，则不得不每修改一个属性就上一次锁(在每个类成员的访问函数中)，因为要防止多个线程同时调用访问函数(去修改同一个属性)。

原则4.10 不要在类定义时提供成员函数体

4.10.1 例子

```
class Type_T
{
  public:
    Type_T(void) { };  //不要在这里写函数体，哪怕是空的
//…
};
```

4.10.2 原因

●定义和实现要分开，使用者不关心实现(就算是空的也不关心)，让他看到不该或不需要看到的东西会干扰其思路，增加其阅读的负担。

●在类定义中的函数体自动是inline方式，将来若要该函数成为非inline方式，比分开写要麻烦。

原则4.11 函数声明(而不是实现) 时定义参数的缺省值

4.11.1 例子

```
class Type_T
{
  public:
  Type_T(int initValue=0);    //在这里给出缺省值
  //…
};
Type_T::Type_T(int initValue)  //而不要在这里给出缺省值
{
  //…
}
```

4.11.2 原因

●函数带缺省值类似于重载(overload)函数，我们应在声明中让使用者知道此事，而不是等到实现时才说。(使用者不该、而且也可能根本看不到实现，比如第三方的类库。)

●在实现时给出缺省值还会导致信息分散(比如当实现分散在多个文件中时)。

原则4.12 恰当选择成员函数、全局函数和友元函数

参考原则

●虚函数必为成员函数，因为非成员函数不能为虚。

●operator>>和operator<<必为非成员函数，因为具第一个参数是cin或cout等，而不是你自己的类。

若需访问类成员，它们还必须是友元函数。

●若函数第一个参数(包括隐含参数，比如this)需要做隐式类型转换，则其必不

能是成员函数，因为编译器不会对this做类型转换。

若需访问类成员，它们还必须是友元函数。

●对操作符函数的选择请参见"原则10.3 区分作为成员函数和作为友元的操作符"。

●其他情况首选成员函数，因为它最符合面向对象规则。

原则4.13 防范、杜绝潜在的二义性

4.13.1 例子

●例1，类型转换(不当)造成的二义性问题：

```
class Type_T
  public:
    void f(char c);
    void f(int i);  //重载函数
};
Type_T t;
t.f(0);   //二义性：编译器不知道是要调用Type_T::f(char)还是Type_T::f(int)
```

●例2，编译器重新命名(name resolution)函数时暴露出二义性问题：

```
class Type_T
{
  public:
    void f(char c);
    /*-------------------------------------------------------------
    *错：无法重载函数f()，因为编译器内部重新
    *命名(name resolution)两个函数时只看参
    *数而不看返回值，所以区分不出这两个函数有
    *何不同
    -------------------------------------------------------------*/
    int f(char c);
 };
```

4.13.2 原因

●C++所信奉的哲学是：二义性不是错误，编译器会容忍其存在(如例1所示，如果没有t.f(0)语句，该代码会编译通过，也可执行)，但我们自己要剔除这些隐患。

●绝大部分二义性是由类型转换(不当)或标识符重新命名(name resolution)造成的，因此在编程时遇到这两种情况要特别当心。

●有时二义性代码可以正常运行很长时间，直到二义性最终被碰到。与其让你的

代码一直带着这个"炸弹"，不如一开始就防止它产生。

原则4.14 显式禁止编译器自动生成不需要的函数

4.14.1 说明

编译器可自动/隐式生成缺省构造函数(default constructor)、拷贝构造函数(copy constructor)、赋值函数(operator=)等。如果你不需要，一定显式禁止，特别不能认为"现在不会用到，以后用到再说"。

4.14.2 例子

```
class Mutex_T
{
    //…
    private:
    //显式禁止拷贝构造函数和赋值函数;只声明，不定义函数体
    Mutex_T(const Mutex_T& rhs);
    Mutex_T& operator=(const Mutex_T& rhs);
};
```

4.14.3 显式禁止的原理

●编译器一旦发现用户自己声明了那些函数，就不再自动生成。

●声明为private将导致类外无权访问，从而达到禁止类外使用被禁函数的目的。

●不定义函数体则防止了类内及友元对其使用。

●最终编译器在编译时阻止所有使用被禁函数的企图。

原则4.15 当遇到错误时对象应该应对有度

说明

对于所有可能遇到的(合理)问题(比如文件打不开、通信请求超时等)，应避免没有应对措施或应对结果不可预料的情况。

原则4.16 用嵌套类的方法减少全局命名空间类的数量

4.16.1 说明

不需要把所有类的定义都放在全局命名空间中。嵌套在别的类中的类是不属于全局命名空间的。

4.16.2 原因

●减少命名冲突。

●使用者可以不再关心那些被嵌套的类(的具体实现)，因为它们是内部使用的。这就简化了阅读，且今后修改也方便(只牵扯到内部类的变化)。

第 5 章　(面向对象的)继承

继承是面向对象语言的一个基本特性。功能强大自不必多言，但"打破封装"的副作用却鲜为人知，所以必须运用得当，否则发挥不出其特点，甚至适得其反。

原则5.1　"公共继承(public inheritance)"意味着"派生类是基类"

5.1.1　说明

若类B公共继承于类A，则B的对象也就是A的对象，反之则不然。在决定一个类是否要公共继承于另一个类时，简单套用此规则验证一下。

5.1.2　例子

"白马是马，但马不是白马":"白马"相当于类B，并公共继承于"马"(类A)。换句话说，任何以"马"做参数的函数都可以不作任何修改，直接接受"白马"(作为参数传入)。

原则5.2　关于"有"和"由…实现"

5.2.1　说明

(一个类)包含(另一个类)意味着"有一个"或"由…实现"。"私有继承(private inheritance)"意味着"由…实现"。(和"包含"类似，完全不同于"公共继承"。)

5.2.2　例子

"马有四条腿":"马"这个类包含"腿"这个类。

5.2.3　"公共继承"、"私有继承"和"包含"的相互关系

- "公共继承"和"包含"的区别:

"包含"的两个类不能互相替换(马不是腿，腿也不是马)，而"公共继承"的派生类可以替换基类(白马是马，但反过来不行，马不是白马)。

- "公共继承"和"私有继承"的区别：

编译器不会将"私有继承"的派生类转换成基类，也就是说，"私有继承"的基类和派生类也没有谁是谁的关系。

- "私有继承"和"包含"的关系：
- 基本一样(了解这一点有助于理解和使用"私有继承")。
- "私有继承"可以访问基类的protected成员，而"包含"不能访问被包含类的protected成员。
- "私有继承"还可以重写基类的虚函数，而"包含"做不到。
- 但是应尽量用"包含"，只有在必要时才选择"私有继承"，因为它不如"包含"

简单直观，且更容易和"公共继承"混淆。

原则5.3 关于继承和模板(template) 的区别

5.3.1 说明

当类型差异不影响类的行为时用模板；否则用继承。

5.3.2 例1：应该选择模板

```
/*-------------------------------------------------------------------
---
* HA_HashMap_T不在乎KEY_T和VALUE_T具体是什么类
*型，即read()、write()、remove()的行为与KEY_T和
*VALUE T的具体类型无关
-------------------------------------------------------------------
---*/
template<class KEY_T, class VALUE_T>
class HA_HashMap_T
{
    public:
    //not support transaction
    virtual int read(const KEY_T& VALUE_T&);
    //not support transaction
    virtual int write(const KEY_T&, const VALUE_T);
    //not support transaction
    virtual int remove(const KEY T&);
};
```

5.3.3 例2：应该选择继承

```
/*-----------------------------------------------------------
*HA_SafetyHashMap_T的read()、write()，remove()的行
*为要和HA_HashMap_T有所不同，即HA_SafetyHashMap_T
*和HA_HashMap_T这两个不同的类型意味着不同的行为方式
-----------------------------------------------------------*/
template<class KEY_T,class VALUE_T>
class HA_SafetyHashMap_T:public HA_HashMap_T<KEY_T, VALUE_T>
{
  public:
   virtual int read(const KEY_T&, VALUE_T&);    //support transaction
  virtual int write(const KEY_T&, const VALUE_T);  //support transaction
  Virtual int remove(const KEY_T&);    //support transaction
```

5.3.4 原因

●模板必须能适合任何类型，所以它不能依赖具体类型来决定其行为。

●继承的目的是为了发扬光大(如果没有变化，和基类完全一样，直接用基类好了)，所以各个派生类(类型不同)必然有不同的行为。

原则5.4 关于继承接口和继承实现

5.4.1 说明

●纯虚函数(pure virtual function)：只继承接口并强制派生类必须提供自己的实现。

●一般的虚函数(virtual function)：继承接口并提供缺省实现。

●非虚函数：继承接口和实现(一点也不能改)。

●私有继承：只继承实现，因为接口在私有继承后都成为private。

5.4.2 另外

纯虚函数是可以有函数体的。许多人误以为纯虚函数一定不能有函数体，这是错误的(大家可以自己试一下)。该函数体的作用也是提供缺省实现。但是如果派生类要使用该缺省实现，必须在自己的函数体中显式调用(见下面的例子)。

5.4.3 例子：纯虚函数提供函数体

```
class MyBaseClass_T
{
  public:
    virtual void myFunction(void)=0;    //纯虚函数
    //  …
 };
 //纯虚函数体
  void
  MyBaseClass_T::myFunction(void)
  {
    //提供缺省实现
  }
  class MyDerivedClass_T:public MyBaseClass_T
  {
  public:
    virtual void myFunction(void);    //继承来的虚函数
    //  …
  };
  //派生类必须提供自己的实现
  void MyDerivedClass_T::myFunction(void)
```

```
//派生类自己要做的事情(如果有的话)
//显式调用纯虚函数体，用来引用缺省实现
MyBaseClass_T::myFunction(void);
//派生类自己要做的事情(如果有的话)
```

原则5.5 限制继承的层数

5.5.1 定量分析的参考

当继承的层数超过5层时问题就很严重了，需要有特别的理由和做特别解释。

5.5.2 原因

●很深的继承通常意味着未作通盘考虑的设计。

●会显著降低效率。

●可以尝试用(类的)组合替代(过多的)继承。

●与此类似，派生于同一父类的子类个数也不能太多(计划生育?)，否则应考虑是否要加一层父类，通过某种程度上的(新的)抽象，减少同层类的个数(这是一种平衡的艺术)。

原则5.6 继承树上非叶子节点的类应是虚基类

5.6.1例子

如果你有两个(非抽象)类C1和C2，且希望C2派生于C1，如图5-1左边所示，则应该增加一个抽象基类(abstract base class)A,且将C1和C2都从A派生出来，如图5-1右边所示。

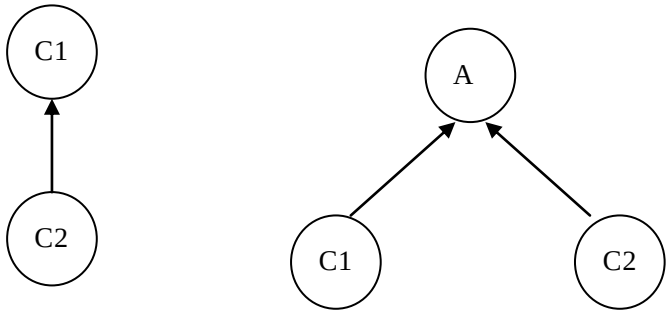


图 5-1

5.6.2 原因

●基类A的存在明确了C1和C2的公共部分和差异，即C2到底想从C1继承什么、改写什么。

●今后的修改很明确：对A的修改会影响C1和C2两个类。而对C1和C2各自的修改不会影响对方。

●进一步说，因为C++的类封装了具体实现，而只把接口露在外面，所以C1的接口应该是C2最想要继承的。换句话说，只要C2继承了C1的接口，C2的对象就可以被看作C1的对象而参与任何C1对象可参与的活动。由此可见，接口本身(而不是

具体实现)是参与活动的充分必要条件，这就是为什么A通常是只定义接口的完全的抽象基类(类似于Java中的Interface)。

●用完全抽象基类的另一个原因：继承存在一个很大的缺陷，就是打破封装，即把基类的实现细节暴露给派生类。导致的结果是对基类的修改将无法保证不会波及派生类，即基类和派生类是紧耦合关系(紧耦合的缺点请参见"原则4.5 降低类间的耦合度")，除非基类完全没有实现细节。

●用完全抽象基类的第三个原因：为了防止多重继承带来的各种问题(参见"原则5.14慎用多重继承")，继承树上非叶子节点的类最好没有成员。

原则 5.7 显式提供继承和访问修饰：public、protected或private

5.7.1例子

```
class Base_T
{
  public:    //显式指出访问权是public，好
    void put(void);
};
class Derived_T:Base_T    //没有显式给出，缺省为私有继承，不好
{
};
void foo(Derived const& derived)
{
  derived.put();    //编译错：此时put()是private访问权
}
```

5.7.2 原因

●使阅读者直接了解成员和成员函数的访问权限，而不用再回忆此处的缺省规则：

- ●缺省继承是私有继承。
- ●类(class)的缺省访问权是私有(private)。
- ●结构(struct)和联合(union)的缺省访问权是公共(public)。

●减少产生(使用)错误的可能性。

原则5.8 显式指出继承的虚函数

5.8.1 例子

```
class Base_T
{
```

```
    public:
    virtual void myFunction(void);      //虚函数
    //  …
};
class Derived_T:public Base_T
{
    public:
    /*---------------------------------------
    *派生类提供新的实现。由于继承于基类，
    *所以自动是虚函数，但为了更清晰，还是
    *把关键字virtual加上了
    ---------------------------------------*/
    virtual void myFunction(void);
    //  …
 };
```

5.8.2 原因

●方便自己和其他阅读者理解：

不用回头追踪基类的定义，以便知道某个函数是否为虚函数，特别是继承层数较多时更是如此。

●减少隐患：

比如，定义新派生类时不会将重载和虚函数搞混。

原则5.9 基类析构函数(destructor)首选是虚函数

5.9.1 说明

若决定不作虚函数，一定要有特殊理由。"导致效率下降"的理由不充分。

5.9.2原因

基类一派生类最常见的用法是：基类提供接口，派生类提供实现。这样，使用者就不用关心到底是哪个派生类提供实现，这就是所谓的多态。使用时用基类指针或引用指向/引用派生类的对象，而所有的操作是在这些基类指针/引用上进行的。若基类析构不是虚函数，这种用法就有问题：基类指针或引用只能调用基类的析构函数，从而使得派生类对象析构不成功(不完全)。

原则5.10 绝不要重新定义(继承来的)非虚函数

5.10.1 例子

```
class Base_T
{
public:
```

```
    void someMethod(void);  //非虚函数
   };
   class Derived_T: public Base_T
   {
   public:
    void someMethod(void); //重新定义!强行覆盖已经继承来的父类成员函数
    };
  Derived_T derivedObj;
  Base_T* pBase=&derivedObj;
  Derived_T* pDcrived=&derivedObj;
  pBase->someMethod();    //结果不令人满意：实际上在调
Base_T::someMethod()
  pDerived->someMethod();    //结果不令人满意：实际上在调
Derived_T::someMethod()
```

5.10.2 原因

●是什么(天大的)理由不用虚函数，而非得用这种方式重写函数实现?

●一般编译器都会对此给出警告，有的还认为是错误。

●如上例所示，指针或引用最终调用哪一个函数严重依赖指针或引用本身的类型，而与其所指或引用的对象无关(因为非虚函数是静态绑定)，这总是非常令人困惑的。

●这种用法有悖于C++的语义，见"原则5.4 关于继承接口和继承实现"。

原则5.11 绝不要重新定义缺省参数值

5.11.1 例子

```
   class Base_T
   {
   public:
   //基类提供了缺省参数值为WHITE
   virtual void someMethod(Color_T color=Color_T::WHITE);
   };
   class Derived_T:public Base_T
   {
   public:
   //又在派生类给出新的缺省参数值为RED
   virtual void someMethod(Color_T color=Color_T::RED);
   };
   Base_T* pBase=new Derived_T;
```

```
//你能确定缺省参数是WHITE还是RED吗
//那其他阅读/使用此代码的人呢
//你所用的编译器也这么认为吗
pBase->someMethod();
```

5.11.2原因

事实上编译器是静态绑定(statically bound)缺省参数值的，也就是说pBase的类型决定了缺省参数为WHITE。而调用哪个函数是动态绑定的，即由pBase所指的对象决定。其结果是：Derived_T::someMethod(WHITE)被调用。这种基类和派生类共同作用的产物太令人费解了。

原则5.12 不要将基类强制转换成派生类

5.12.1 说明

很多刚从C语言转过来的程序员习惯于这样的思路：若对象的类型是T1，则做某种处理。若是类型T2，则做另外的处理。等等。在C++中，基类指针可以指向派生类对象，所以当所指对象为某类派生对象时，自然想到将基类指针(强制)转换成相应派生类，然后调用派生类相应的函数。

5.12.2 原因

●C++提供了更好的解决方案：虚函数。

●使用虚函数更安全：不会出现(强制)转换错的情况。

●使用虚函数效率更高：用函数指针实现，从而免去了(每次的)条件判断。

●使用虚函数适用性更强：真正的动态绑定，而强制类型转换本身是静态绑定(编译而非运行时决定用什么)。

原则5.13 关于C++中的分支用法选择

5.13.1 说明

若分支依赖于对象的值，用普通的分支语句(if-else，switch等)。若分支依赖于对象的类型，用虚函数方式。

5.13.2 引入虚函数分支法的原因

●更符合C++风格。

●效率更高：

普通的分支语句需要CPU的指令流水线等待条件语句的计算结果，然后才能确定下一步调哪一条指令进入流水线，这实际上打断了流水线(的流水)。新型的CPU有分支预测功能以尽量减小流水线被打断的可能，但实现复杂且浪费资源；如果接连有多个条件分支，解决的复杂性和所需资源会呈几何级数增加，在这种情况下，CPU很快就不得不放弃分支预测。

虚函数是靠函数指针(指向什么函数体)来确定运行什么函数，不需要任何条件判断。

●代码更简单，维护更容易：

普通分支语句(方式)在增加或去掉某个类型时，要增加或去掉相应分支，而虚函数方式不用。

原则5.14 慎用多重继承

原因

●多重继承会显著增加代码的复杂性，还会带来潜在的混淆：

比如菱形继承问题：类A派生类B和类C，类D又多重继承于类B和类C，导致类D的对象中会有两份类A对象(一份来自类B，一份来自类C)。

可以看出，多重继承若使用不当会造成很多麻烦(麻烦到诸如Java等面向对象语言根本不支持多重继承)。

●另一方面，多重继承(偶尔)又是不可或缺的：

比如要设计马、驴和骡子这三个类时，没有多重继承，无法(直接了当地)实现"骡子既是马，又是驴"这一功能。具体地说，如果有一类函数以马为参数，则这类函数应该不需任何修改就能接受骡子作为传入参数。同样地，所有以驴为参数的函数也可直接接受骡子作为传入参数。可以看出，若只允许单继承，马或驴必有一个不是骡子的父类，则相应的一类函数接受骡子为参数就有问题。

原则5.15 所有多重继承的基类析构函数都应是虚函数

原因

目的是当析构派生类对象时，确保所有析构函数都被正确调用。在此特别强调这一点是因为菱形继承的特殊性。

第6章　内存分配和释放

内存分配和释放是几乎所有程序的基本需求，同时也是最常出问题的地方之一，有谁没有听说过内存泄漏的问题呢?通过遵循几条简单的规则，你可以避免很多常见的内存分配问题。

原则6.1　用new、delete取代malloc、calloc、realloc和free

原因

●更安全：

malloc、calloc、realloc和free是C语言的用法，它们不意识(理会)到对象的存在与否，更不会去调用构造和析构函数，所以在C++中经常会引起麻烦。

●混用这些函数会造成更大的麻烦：

比如要防止用malloc分配、用delete释放，这些都需要花费额外的精力。

原则6.2　new、delete和new[]、delete[]要成对使用

原因

●调用new所包含的动作：

●从系统堆中申请恰当的一块内存。

●若是对象，调用相应类的构造函数，并以刚申请的内存地址作为this参数。

●调用new[n]所包含的动作：

●从系统堆中申请可容纳n个对象外加一个整型的一块内存。

●将n记录在额外的那个整型内存中(其位置依赖于不同的实现，有的在申请的内存块开头，有的在末尾)。

●调用n次构造函数初始化这块内存中的n个连续对象。

●调用delete所包含的动作：

●若是对象，调用相应类的析构函数(在delete参数所指的内存处)。

●将该内存返还系统堆。

●调用delete[]所包含的动作：

●从new[]记录n的地方将n值找出。

●调用n次析构函数析构这块内存中的n个连续对象。

●将这一整块内存(包括记录n的整型)归还系统堆。

可以看出，分配和释放单个元素与数组的操作不同，这就是为什么一定要成对使用这些操作符的原因。

原则6.3　确保所有new出来的东西适时被delete掉

原因

不需要的内存不能被及时释放(回系统)就是大家常听到的内存泄漏(memory leak)。狭义的内存泄漏是指分配的内存不再使用，但却永远不被释放。从更广的意义上说，没有及时释放也是内存泄漏，只是程度较轻而已。

内存泄漏不管有多小，最终都会耗尽所有内存，只要运行的时间足够长。

内存泄漏的问题极难查找，因为当出现问题时，内存已然耗尽，此时CPU正在运行什么程序(哪怕是系统程序)，什么程序就崩溃。可以看出，崩溃时报告的出错信息与引起问题的代码毫无关联。另外内存耗尽的时间是不确定的，且一般会是一个较长的时间(几天或几星期都可能)，这就更增加了再现和定位问题的难度。

目前有一些工具可以帮助查找内存泄漏问题，比如Rational公司的Purify。有些编译器也带有类似的功能，比如SUN的Forte编译器。这些工具大大提高了查找内存泄漏的能力。但防患于未然才是治本之道(本章稍后会给出一些防范的原则)。

原则6.4 谁申请谁释放

说明

●原则上，哪个对象、函数申请的系统堆内存，在用完后应由原申请者释放，所谓解铃还需系铃人。

这个说起来容易，其实要完全做到很麻烦。比如经常遇到这样的情况：程序内部要传递一些信息，这些信息一般都是需要时产生，用完消除，所以应该存放在系统堆内存中。但是由于传递，导致该内存的产生者必须知道什么时候所有人都不再使用该内存，即需要回收了，这可不是一句两句能实现的，拿捏不好就会发生回收之后又有人要用、或所有人都不再用了却没有及时回收的情况。

当然，传递可以用内存拷贝方式(而不是上述的传递指针或引用方式)，这样，就不存在回收的问题了。但当信息很大或引用者很多时，拷贝的代价就成了问题。这引出另一个有名的计算机问题，即程序运行时，数据像波浪一样从内存中的一处移动到另一处，可以想象，这种移动毫无意义，肯定不是好的解决方案。

更为严重的是，如果程序被异常(exception)打断，程序正常的回收机制(比如析构函数)可能根本得不到机会运行，此时又如何确保资源的回收?

●所有涉及资源管理的代码应使用诸如自动指针(auto pointer)或引用计数(count reference)之类的技术。

●自动指针的原理(源码请见："附录 供参考的源代码")：

●自动指针在其构造函数中接受或自己申请一个系统堆内存，并在其析构函数中释放该内存。

●这样，当该自动指针所在对象消失或所在函数退出时，自动指针的析构函数会被系统自动调用(局部变量要被系统自动释放)，从而最终自动释放其所管辖的系统堆内存。实际上，这是把一个系统堆内存的生命周期(原本"无限"长)限制

在一个生命周期有限的局部变量上

●如果异常发生，C++的异常机制保证所有已构造成功的局部变量会被自动析构。由于自动指针是局部变量，如果构造成功，则系统堆内存已分配，析构函数能将其回收。如果没有构造成功，系统堆资源尚未分配，不能执行析构函数也无妨。

●自动指针间可以通过拷贝或赋值传递其拥有的内存：它们把所管辖的系统堆内存从源自动指针转移到目的自动指针，此后源自动指针不再拥有该系统堆内存的管辖权(析构时要释放的将是一个空指针)。可以看出，此种拷贝或赋值函数比较特殊，它们的源(lhs)也需要被修改(将其拥有的系统堆内存指针置成空)，而一般拷贝和赋值函数的源不需要任何改动。(请参见"原则7.8　拷贝构造函数和赋值函数尽量用常量参数"。)

●通过自动指针的拷贝和赋值，可以使系统堆内存能生存于申请者之外而不需申请者释放之。

●当最后一个拥有者(对象或函数)将要消失且不再转移系统堆内存的管辖权时，最后的自动指针将释放该系统堆内存。

●可以看出，编程者不再关心何时、由谁来释放申请的内存(这是最麻烦的)，他只要指定下一步谁(对象、函数)要拥有该内存，并作简单的转移(赋值或拷贝)即可。拥有者内部的自动指针将完成适时的回收动作。

●引用计数的原理：

●引用计数类似于自动指针，只是每次拷贝或赋值(引用计数对象)时都只是将引用数加1。而在每个引用计数对象析构时引用数减1，只有当引用数为0时，析构函数才释放该内存。

●引用计数(相对于自动指针)的特点是：

●许多拥有者(对象；指针)是同时共享一个系统堆内存，所以可以并行工作。但要注意，多线程环境下，只读没问题，有读有写要做互斥保护(上锁等)。

●众多拥有者放弃使用权的先后可以随意，即谁都可以是最后一个拥有者，但只有当最后一个拥有者放弃时，被共享的系统堆内存才会、且一定会被释放。此后决不会有人能再引用该内存，因为所有知道该内存地址的人都不存在了。

●可以看出，编程者更加不用分神去考虑此类内存的回收，他甚至都无法断定谁将是最后的回收者。

●计算机系统中引用计数的应用很多，如多用户打开同一个文件、内存页面的引用等。

原则6.5 当对象消亡时确保指针成员指向的系统堆内存全部被释放

原因

●否则会造成内存泄漏。

●将相应的指针成员替换成自动指针(auto pointer)对象是一个一劳永逸的办法：

●因为若对象正常消亡，系统会在其析构函数执行完成后自动调用所有成员的析构函数(如果有的话)，所以自动指针成员会在此时释放其拥有的系统堆内存。

●无论发生什么异常，只要对象能被析构，自动指针就会释放其拥有的系统堆内存。

●对象在构造时发生异常，其析构函数是不被调用的，因为该对象还未构造完成。但即便如此，C++的异常处理机制也会依次调用已构造好的成员的析构函数，所以已申请的内存还是能被释放掉的，见下面的代码和图6-1：

例如：

```
class MyClass_T
{
public:
    MyClass_T(void);
    ~MyClass_T(void);
private:
    A* aPtr;
    B* bPtr;
    C* cPtr;
};
MyClass_T::MyClass_T(void):
    aPtr(NULL),
    bPtr(NULL),
    cPtr(NULL)
{
    aPtr=new A();
    bPtr=new B();
    /*-----------------------------------------
    *如果"new C()"失败，谁来释放aPtr和bPtr
    *如果用自动指针技术，自动指针会释放它们
    -----------------------------------------*/
    cPtr=new C();
}
MyClass_T::~MyClass_T(void)
{
    delete aPtr;
    delete bPtr;
```

```
    delete cPtr;
}
```

原则6.6 自定义类的new/delete操作符一定要符合原操作符的行为规范

6.6.1 说明

就是说自定义的new/delete在使用上应该和系统的new/delete没什么两样，比如带同样的参数、new要返回一个内存指针、若没有足够的内存供分配则返回空指针(NULL)且要产生一个异常(throw an exception)，等等。

6.6.2 例子(接口格式)

```
class MyClass_T
{
 public:
//...
    void* operator new(size t);    //申请一个元素
    vold* operator delere(void* );    //释放一个元素
    void* operator new[](size_t);    //申请一个数组
    void* operator delete[](void*);    //释放一个数组
    //...
    };
```

6.6.3原因

系统new/delete的操作方式实际上是此类操作符的标准接口，许多程序都是基于这种标准接口进行编程的。如果有一天使用者发现这个接口改变了，那可不仅是感觉上别扭，不知有多少相关代码要重新修订(遗漏一处就是一个隐患)。其实已经无法修订了，因为已经没有大家都遵守的公共标准了(标准本身被破坏了)。

原则6.7 自定义类的new操作符一定要自定义类的delete操作符

原因

内存资源从哪里、以何种方式申请到的，还应以相同的方式返还。

原则6.8 当所指的内存被释放后，指针应有一个合理的值

6.8.1 说明

除非该指针变量本身将要消失(out of scope)，否则应置为空(NULL)。

6.8.2 例子

```
void Channel_T::disconnect(void)
```

```
{
    delete i_pConnection;     //释放资源
    i_pConnection=NULL;     //指针指向一个合理的值
    //...
    /*-------------------------------------------------
    * 注意：delete不会也不可能将i_Connection置成
    * NULL，因为传给delete的是i_Connection的一个
    * 拷贝，而不是它本身
    -------------------------------------------------*/
}
```

6.8.3原因

●防止其后再次使用该指针(使用NULL会立刻导致系统错误，好查)。

●防止其后再次删除该指针。

●这些防范措施不仅针对现有代码，还使将来(未知)的代码修改更安全。

●举一反三，指针在其声明周期的全程都要指向一个合理的值，比如指针的定义和初始化一定要放在一起，以保证指针一旦生成就指向合理的值。

原则6.9 记住给字符串结束符申请空间

6.9.1例子

```
pName=new char [strLen+1];     //+1是为了给"\0"留出空间
```

6.9.2说明

字符串有隐含的结束符"\0"，申请空间时别忘了。

这种数组越界的错误很难查。因为一方面越界的概率一般很小，不好再现。另一方面，即使越界，也不一定导致程序出错，因为只有越界后要访问其他进程的内存空间时，系统才意识到有问题。或者越界修改自己的其他内存值而造成严重问题，比如导致该值成为非法，否则程序还能向前推进。越界后是否出错还跟编译器或编译时的参数设定有关，比如有的编译器或参数设定(如打开优化功能)会要求内存变量一个紧挨一个排列。另一些却选择诸如字对齐方式，这会留下一些空隙，如果越界正好落在空隙中，就什么问题也没有。所以，有时发现，调试版(打开debug编译选项)不出问题，优化版总出问题。

第7章　初始化和清除

你必须确保你的程序中的对象总是处于完整、可用、定义明确的状态。这就要求初始化工作一定要仔细、完善。同样地，当需要清除这些对象时也必须做到及时、干净、彻底。中国有两句话，一句叫"良好的开端等于成功的一半"，用在这里意思是说初始化工作占成功的50%；另一句叫"行百里者半九十"，用在这里可以理解为清除工作占成功的另一个50%(而中间的工作，由于相对不容易做的不好，在成功中的比例忽略不计)。

构造函数和析构函数是初始化和清除的主角。由于是隐式调用，它们容易被忽视。

构造函数和析构函数还有一些普通函数没有的限制、特性和语法。

请遵循这样的法则：善始善终。

原则7.1　声明后就初始化强于使用前才初始化

7.1.1　例子

```
//使用前才初始化
String_T systemName;     //声明时未初始化：调用缺省构造函数
systemName="Kitchen sink";     //随后再作初始化：调用赋值操作符函数
//声明后就初始化
String_T systemName("Kitchen sink");     //只调用一次构造函数
```

7.1.2原因

- 减少使用未初始化对象的几率。
- 减少临时状态。
- 提高效率。

原则7.2　初始化要彻底

7.2.1　例子

```
struct Circle_T
{
  Color_T i_color;
  int 1_x;
  int i_y;
  int i_radius;
};
//只初始化了一个成员，还有三个没初始化
Circle_T blackCircle={Color_T::BLACK};
```

7.2.2 原因

●初始化不彻底的对象不是可用状态。

●稍不注意，就有可能在完全初始化前使用。

原则7.3 确保每一个构造函数都实现完全的初始化

7.3.1 原因

一个初始化不完全的对象就像一个等待被踩响的地雷，被触发只是时间和幸运程度的问题。

7.3.2 用init()函数导致构造函数初始化不完全

有些编程者喜爱单独提供一个(普通的成员)函数init()来做初始化动作而部分或全 部替代构造函数的功能(不是在构造函数中调用init()函数，而是在构造函数返回后，通过已存在的对象显式调用init()来完成最终的初始化)。他们认为这样做有至少两个好处：

一是init()能返回初始化成功与否的状态，而构造函数不能有返回值；二是在构造一个非 常费时的对象时，可单起一个线程执行init()，原线程可以继续执行其他初始化动作，从而加快整个程序的初始化过程(并行初始化)。但是，这样做是违反本准则的，并且在大部分情况下是得不偿失的。

●构造函数是特殊函数，C++保证对象在被使用前一定会调用某个构造函数：而init()是普通函数，除了自觉遵守没有别的办法保证被调用，特别是隐式构造对象时可能根本无法适时调用init()。

●构造函数是特殊函数，C++保证对象在被使用前一定只会执行一次构造函数，也就是确保初始化只作一次；而init()是普通函数，除了自觉遵守没有别的办法保证只被调用一次。

● 构造函数能调用基类构造函数并自动保证所有基类构造函数的调用顺序，而init()必须手工来保证这些。

●构造函数中的初始化列表是真正"初始"化(类成员)的地方，而init()中的"初始"化顶多算是第二次对这些类成员的操作，它们的区别一是效率，二是有些成员(比如常量成员)根本不允许二次初始化。

●这里所说的初始化并不排斥(初始化之后的)对象重新复位(如reset()之类的函数)功能，因为初始化和复位从本质上说是两个不同的概念。

7.3.3 结论

●尽量遵守本准则。

●若非要产生构造不完全的对象，先问自己，为什么非要?这是唯一的解决办法吗?

●确实不得已，一定要清楚所有可能的后果，并做全面的防范——包括代码中的防范、使用说明、今后修改时的注意事项等都要列举清楚。

原则7.4 尽量使用初始化列表

7.4.1 例子

```
class Square_T
{
  public:
  Square_T(int aHeight, int aWidth, Color_T aColor);
  //...
  private:
  int i_height;
  int i_width;
  Color_T i_color;
};
Square_T::Square_T(int aHeight, int aWidth, Color_T aColor):
  //使用初始化列表给各成员赋初值
  i_height(aHeight),
  i_width(aWidth),
  i_color(aColor)
  {
//...
}
```

7.4.2 原因

●效率更高：

以上述i_color为例，若在初始化列表中初始化，则只调用类Color_T的某个构造函数一次；若在构造函数体中初始化，则Color_T的缺省构造函数已被隐式调用，而在构造函数体中的初始化实际上是在调用operator=。

●必不可少：

常量 、引用等的初始化只能用初始化列表。

原则7.5 初始化列表要按成员声明顺序初始化它们

7.5.1 说明

同时记住，要先初始化直接父类，然后是成员(非静态，参见"原则7.7 不要用构造函数初始化静态成员")。

父类的初始化是通过调用父类某个构造函数完成的；它也会先初始化它自己的直接父类，从而保证继承链上正确的初始化顺序。

7.5.2 原因

不论你在初始化列表中用什么顺序初始化类的成员，编译器都会按成员在类定

义时声明的顺序做初始化，所以初始化列表的顺序实际上是不起作用的。但打乱顺序会造成阅读和理解上的混淆，特别是当你需要用一个成员初始化另一个成员时更是如此。

原则7.6 构造函数没结束，对象就没有构造出来

说明

构造函数结束以前，系统将相应对象视为不存在(因为不完全)，此时很多事情不能做：

●不能调用析构函数：

不但编程者不能，编译器也不能：比如隐式调用的构造函数未结束(被打断)，编译器永远不会隐式调用析构函数，因为编译器认为该对象没有构造出来。

●构造函数全程不要使用this成员，因为this所指的对象还不完整。

●不要在构造函数中调用虚函数成员：

因为构造函数会先调所有的基类构造函数，等到所有的基类都构造好之后再构造自己；但所有的基类构造函数所用的this参数，其类型均为派生类(而非该基类)，所以(基类)构造函数若调用虚函数，执行的函数体可能是派生类的成员函数，但此时派生类自己还未开始构造。这一点一定要切记！

原则7.7 不要用构造函数初始化静态成员

原因

●构造函数是用来构造对象的；静态成员不属于对象，它属于类本身。

●可以通过类名直接初始化静态成员，或者用静态成员函数初始化静态成员。

原则7.8 拷贝构造函数和赋值函数尽量用常量参数

7.8.1 例子

```
calss MyClass_T
{
    public:
    MyClass_T(const MyClass_T& rhs);     //拷贝构造函数
    MyClass_T& operator=(const MyClass_T& rhs);     //赋值函数
};
```

7.8.2 原因

●一般来说，拷贝和赋值过程中源对象不会被修改，只有目的对象被修改，所以构造函数和赋值函数的参数应该是常量。

●变量可以当常量用，所以变量可以传给参数是常量的拷贝或赋值函数(编译器会将变量隐式转换成常量，再传给函数)。

原则7.9 让赋值函数返回当前对象的引用

7.9.1 例子

```
MyClass_T& MyClass_T::operator=(const MyClass_T& rhs)
{
  //...
  return *this;
}
```

7.9.2原因

符合赋值的常见用法和习惯，如objectl=object2=“hello world.”等。

原则7.10 在赋值函数中防范自己赋值自己

7.10.1 例子

```
MyClass_T& MyClass_T::operator=(const MyClass_T& rhs)
{
    //防止自己赋值自己
    if(this!=&rhs)
    {
      //…
    }
    returll *this;
}
```

7.10.2 原因

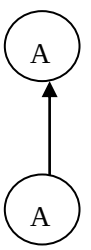
●自己赋值自己会和普通的赋值有很多不同，若不防范会出问题。

●由于一般不太用到自己赋值自己，所以问题会被隐藏很久，出了问题也很难查。

●比较源和目的对象是否位于内存的同一地址是最简单、高效、安全的方法。

顺便说一句，内存地址相同并不绝对意味着是同一个对象：

如图7-1所示，类B继承于类A，则类B对象中会有一个完整的类A对象，这就是为什么类B对象可以当作类A对象用的实现原理。可见在同一个内存(起始)位置有两个不同的对象。如果是图7-1右面的菱形继承，则在同一个内存起始位置上有更多不同的对象。



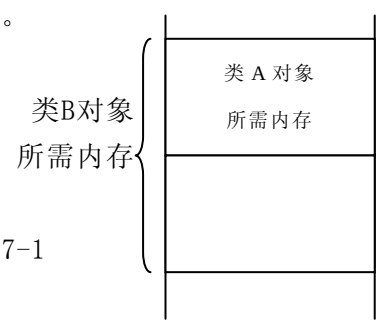
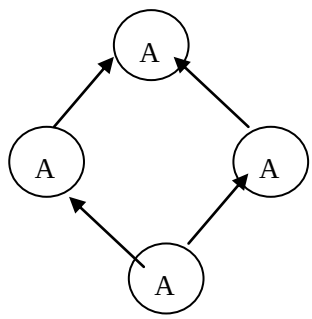


图7-1



即便如此，从上面的举例也可以看出，用比较内存(地址)的方法防止自我复制还是安全有效的。

●不能用对象本身做比较(*this!=rhs)，因为这往往意味着比较两对象的内容是否相同，而不是确定这两个对象是否为同一对象。

原则7.11 拷贝和赋值要确保彻底

7.11.1 说明

包括所有基类，每个成员都要被拷贝/赋值到。

7.11.2原因

确保拷贝/赋值目标对象的完整性。

原则7.12 关于构造函数、析构函数、赋值函数、相等或不等函数的格式

7.12.1 格式

```
/*-------------------------------------------------------------
*拷贝构造函数：一般参数(源对象)用常量，见〝原则7.8 拷贝
*构造函数和赋值函数尽量用常量参数〞
-------------------------------------------------------------*/
MyClass_T::MyClass_T(coast MyClass_T& rhs);
//析构函数：不带任何参数
HyClass_T::~MyClass_T(void);
/*-------------------------------------------------------------
*赋值函数：一般参数(源对象)用常量，见〝原则7.8 拷贝构造
*函数和赋值函数尽量用常量参数〞
-------------------------------------------------------------*/
MyClass_T& MyClass_T::operator=(const MyClass_T& rhS);
//相等函数：参数都用常量，因为不会(不需要)修改它们
bool operator==(const MyClass_T& lhs，const MyClass_T&rhs);
//不等函数：参数都用常量，因为不会(不需要)修改它们
bool operator!=(const MyClass_T& lhs，const MyClass_T& rhs);
```

7.12.2 原因

这些函数是自定义类中最常见的。应尽量保持格式的统一、尽量使其符合惯例，这样可以令使用更容易、流畅，同时减少误解和错误。

原则7.13 为大多数类提供缺省和拷贝构造函数、析构函数、赋

值函数、相等函数

原因

●包含全部这些函数的类通常被视为符合惯例：

●缺省构造函数在定义对象数组等情况下是必不可少的。

●如需以值形式向函数传递对象，必须有拷贝构造函数。

●析构函数永远都有，编程者不提供，编译器也会自动生成一个空的析构函数。需要注意的是，析构函数经常是虚函数，但自动生成的析构函数不能保证这一点(如果父类析构是虚函数，则自动生成的析构函数也是虚函数，否则是普通函数)。

●自定义类就像普通类型一样，经常需要相互赋值和判断相等与否，所以赋值函数和相等函数经常是不可或缺的。

●不符合惯例的类在理解和使用上多少有些限制。

原则7.14　只有在有意义时才提供缺省构造函数

说明

有些类在构造时不能没有参数，此时不必画蛇添足。

有人会想，我先将就提供一个缺省构造函数，从而保证诸如生成对象数组能够成功。然后，再用其他方法(比如用init()函数)做最终的初始化。这样做的问题请参见"原则7.3　确保每一个构造函数都实现完全的初始化"。

原则7.15　包含资源管理的类应自定义拷贝构造函数、赋值函数和析构函数

原因

如果类不提供这三种函数，编译器可以自动生成(见"原则24.14　如果合理，使用编译器生成的函数")。但自动生成的拷贝构造函数和赋值函数只能将所有源对象的成员简单赋值给目的对象，所谓浅拷贝(shallow copy)，而自动生成的析构函数根本就是空的。这对于包含资源管理的类是不够的：比如包含从系统堆中申请的内存(资源)，浅拷贝会使源和目的对象的成员指向同一(申请)内存，而空析构函数没有将申请的内存释放掉。此时需要自定义这些函数，以完成正确的动作。

除了系统堆内存以外，文件、管道、(网络)连接等都是资源。

原则7.16　拷贝构造函数、赋值函数和析构函数要么全自定义，要么全生成

原因

如果你要自己提供这三个函数中任何一个，那就意味着你认为编译器没有聪明

到能为你生成该函数的正确实现；如果编译器在其中一个函数生成上不够聪明，那它(很可能)在所有函数生成上都不够聪明，因为编译器生成这三类函数的策略类似。

这条原则不是强迫，是提醒。

原则7.17 类应有自己合理的拷贝原则：或浅拷贝或深拷贝

说明

拷贝原则是类内实现的问题，使用者应该不用关心(不管实现，只关心接口)。但是什么是合理的拷贝原则切是从使用者角度判断的，所以开发者在选择时应本着让(大多数)使用者感觉最自然、最方便为前提。

一般一个类不会既需要浅拷贝，又需要深拷贝。

如果不能决断，要在类定义/声明(而不是实现)处用注释声明。

原则7.18 若编译时会完全初始化，不要给出数组的尺寸

7.18.1例子

```
//不需要给出4，画蛇添足
int array[4]={1，2，3，4)
//完全初始化，这样就行了
int array[]={1，2，3，4}
```

7.18.2原因

●让编译器计算数组的尺寸会使你的代码能自适应数组的变化。

●举一反三，任何想指定数组尺寸时都问一下自己：不给尺寸行不行?

●当没有指定尺寸，但随后又需要用时，可用sizeof(array)/sizeof(array[0]算出数组大小。由于sizeof(array)和sizeof(array[0])大多数情况下都是常量，编译器会直接把运算结果放在生成的代码中，一点也不影响效率。

原则7.19 将循环索引的初值定在循环点附近

7.19.1例子

假设循环队列下标从0到MAX-1(MAX为该队列可容纳的元素个数)，则队首游标应设在MAX-3处。

7.19.2 原因

●循环索引最常出问题的地方就在循环点附近，将队首放在这里可以很早就发现问题。

●问题常出现在：插入元素使得游标从MAX-1绕回到0、删除元素使得游标从0绕回到MAX-1、插入头一两个元素，以及删除最后一两个元素时。选择MAX-3可以一口气将这些情况都覆盖住。

●队列越长、插入删除越频繁，查错效果越好(相比将队首设在0处，则队列越长，

越不容易到达循环点；同样，插入删除越频繁，队列也越不容易插满，从而将问题隐藏起来)。

●但是，这样做会使效率有所下降。不过以一点效率的损失换取问题的及早发现，我认为是值得的。

原则7.20 确保全局变量在使用前被初始化

7.20.1 说明

这个问题其实难在全局变量的初始化顺序上：全局变量a的初始化需要全局变量b，所以b的初始化一定要在a之前；但因为a和b都是全局变量，编译器无法保证b一定在a之前初始化。

注意：类的静态成员也是全局变量。

7.20.2 例子

```
#include<iostream.h>
int func(void)
{
    int input;
    cin>>input;    //cin是一个全局变量
    return input;
}
int i=func( );    //i是另一个全局变量，它依赖于cin
void main(void)
{
    //...
}
```

7.20.3 仅有的编译器初始化全局变量的规则(国际标准)

●所有全局变量在被初始化之前的值都为0。

●同一文件中，若全局变量a在全局变量b之前定义，则a一定会在b之前初始化。

●同一文件中，所有全局变量一定会在该文件的任何函数或对象被使用前初始化(而不论该变量的位置是在被使用函数或对象之前还是之后)。

●用#include包括进来的文件，包括嵌套#include，都算作同一文件。

●不同文件中的全局变量没有固定的初始化顺序。

7.20.4 确保全局变量在使用前被初始化的原理

每个全局变量在使用前都要先声明，这声明应该位于一个头文件中(如cin会在 iostream.h中声明一样)。将该全局变量的初始化代码也放在该头文件中。这样，使用该全局变量的代码肯定会在初始化代码之后，从而保证了先初始化再使用的原则。为了防止该全局变量在不同文件中被多次初始化，头文件中的初始

化代码会有一个计数器，只有当计数器为零时才作初始化动作，否则只将计数器加1。为了防止同一初始化代码出现在多个文件中而引起的连接错误(link error)，使用静态方法(见下面的例子)。

7.20.5 确保全局变量在使用前被初始化的具体方法

●假设全局变量g的类型是G，g的声明在头文件g.h中，则在g.h中定义一个新的类GI，它包含一个静态整型计数器i_count、一个构造函数和一个析构函数。

●文件g.cpp是g.h对应的实现文件，类GI的成员i_count在此定义(值为0)；

类GI的构造函数在i_count为0时初始化g，否则只将i_count加1；GI的析构函数将i_count减1，若为0，则调用g的清除代码(如果有的话)。

●在g.h中定义一个GI的静态变量gi，并将其放在变量g的声明之后。

7.20.6 上面例子的延续

```
//文件iostream.h
class istream_withassign    //全局变量cin的类定义(相当于类G)
{
   //…
};
extern istream_withassign cin;    //全局变量cin的声明(相当于g)
class IostreamInit_T    //相当于类CI的定义
{
  public:
    IostreamInit_T(void);
    ~IostreamInit_T(void);
  private:
    static unsigned int i_count;
};
static IostreamInit_T iostreamInit;    //相当于gi
//文件iostream.cc
istream_withassign cin;    //全局变量cin的定义
unsigned int IostreamInit_T i_count;    //类静态成员的定义
IostreamInit_T::IostreamInit_T(void)    //构造函数
{
    if (count++==0)
    {
    //初始化cin的代码
}
}
IostreamInit_T::~IostreamInit_T(void)  //析构函数
```

```
{
    if(--count==0)
    {
    //清除cin的代码
    }
}
```

第8章 常量

很多编程者要么完全忽略关键字const的作用，要么认为它用处不大。其实不然，关键字const的首要功能是通过类型检查帮助编译器查找程序中(更多)的错误；其次是借助编译器强迫代码符合最初的设计；另外，在设计可复用的代码时关键字const也显得很重要。

原则8.1 关于常量修饰符的含义

8.1.1 例子

```
char* p="Hello.";     //指针不是常量，指针指向的也不是常量
char const* p="Hello.";     //指针不是常量，指针指向的是常量
char* const p="Hello.";     //指针是常量，指针指向的不是常量
char const* const p="Hello";     //指针是常量，指针指向的也是常量
```

8.1.2 说明

●"char* p"的意思是：p是一个指针，它指向字符类型。

●"char const* p"的意思是：p是一个指针，它指向一个常量，该常量是字符类型。

●"char* const p"的意思是：p是一个常量，它是一个指针常量，该常量指针指向字符类型。

●"char const* const p"的意思是：p是一个常量，它是一个指针常量，该常量指针指向一个常量，而被指向的常量是字符类型。

原则8.2 在设计函数原型时，对那些不可能被修改的参数用常量修饰

8.2.1 说明

此处参数指的是指针或引用类型。

8.2.2例子

```
void blockCopy( void* pDest, const void* pSrc, sizc_t length);
```

8.2.3 原因

编译器会永远记住此事。如果今后对blockCopy()函数体的任何修改(有意或无意)要对pSrc所指数据进行写操作，就会引起编译错误，从而提示该函数的后继开发者遵循最初的设计。

原则8.3 类成员可以转换成常量形式暴露出来

8.3.1说明

一般来说，类成员是私有的(见"原则4.2 类成员应是私有的(private)")。但有时需要在类外部读取某些成员，此时若用常量形式暴露出来，可以使外部既高效访问到成员，又防止绕过类直接对成员的修改。这是一种巧妙的方法。

8.3.2 例子

```
class String
{
public:
/*-------------------------------------------------------
*将String_T隐式转换成char*，使得String_T可以自接用于打
*印输出、串拷贝等
-------------------------------------------------------*/
operator const char*(void)const {return i_data;}
//...
private:
char* i_data;
```

8.3.3 注意

需要反复强调的是，类成员通常都属于类的实现范畴，所以此种方法有暴露实现的嫌疑，使用前一定要确认。(比如上例就没有暴露实现，使用者根据返回的字符串不能推断该类的实现方法。)

原则8.4 关于常量成员函数

8.4.1 说明

常量成员函数是指函数体不会对任何成员(包括基类成员，但不包括静态成员)进行修改的成员函数。其函数声明以关键字const结尾。其实，常量成员函数就是其隐含的this参数为指向常量对象的指针，但因为this隐含，无法将const修饰符加上，只能放在结尾。由此可推出以下原则：

●若成员函数在设计时不会修改任何成员，则应定义成常量成员函数。这样常量对象和非常量对象都可以使用该函数。

●常量成员函数和同名同参数(显式)的非常量成员函数可重载，这是因为它们的第一个(this)参数类型不同。

这一点常被人忽视，但很有用：比如类需要在多线程中使用，则其成员函数要上锁，不论该函数是要修改某些成员还是只是读取成员(因为虽然你只读，别人可能同时在修改)。但常量对象是不能上锁的，因为锁本身也是成员，上锁是修改锁成员的动作，在常量对象中是不能做的。

那么是不是常量对象在多线程中就不能使用了呢?显然不是。其实你只要再提供(重载)一个常量成员函数，其中不需上锁，直接读取成员做相应的操作即可。

这是因为在这个新提供的成员函数中，this指向常量对象，不用担心读取时

有人修改，上锁也就不必要了。

8.4.2例子

```
class MyClass_T
{
  public:
    //相当于int func(MyClass_T const* this，int aParameter);
    int func(int aParameter)const;
    //重载：相当于int func(MyClass_T* this，int aParameter);
    int func(int aParameter);
    //...
};
```

原则8.5 不要让常量成员函数修改程序的状态

8.5.1 说明

意思是说，常量成员函数不要修改成员、静态成员、全局变量、其他对象，以及所有会造成该对象或程序状态改变的东西。

8.5.2 原因

造成程序状态改变的成员函数不是绝对意义上的常量成员函数。虽然不一定违反C++语法，但语义有问题,且不符合大多数人对常量成员函数的预期。

原则8.6 不要将常量强制转换成非常量

原因

如果要这么做，先想一想该常量是否根本就应是变量，否则给出足够的理由。要防止想利用常量的优点，却将变量的缺点带进来。

原则8.7 任何变量和成员函数，首选用const修饰

原因

经过以上的讨论可以看出，先尝试用const修饰是一种明智的选择。

第 9 章　重载

C++的重载功能使得同名函数可以有多种实现方法。这可以简化接口的设计和使用。这是C++极为重要的特性之一。但是，要恰如其分地运用，以防带来二义性等潜在问题。

原则9.1　仔细区分带缺省值参数的函数和重载函数

9.1.1 说明

你可以将带缺省值参数的函数看成是重载函数的一个变种。

如果可以找出合理的缺省值，且函数的实现不依赖于参数的个数和参数值，则应选择带缺省值参数这一方式。否则，用重载方式。

9.1.2　例子

```
//用重载方式显得很笨重
class Event_T
{
  public:
    void report(char const* pMessage);
    void report(char const* pMessage, bool doLog);
void report(char const* pMessage, bool doLog, bool doPring);
};
//用缺省参数方式更好
class Event_T
{
  public:
    void report(
    char const* pMessage,
    bool doLog=false,
    bool doPring=false
);
};
```

9.1.3　原因

●简化实现和维护：

缺省参数方式减少了函数的个数，通过把相似的代码进行合并，简化了实现，也使得理解和维护变得简单。

●同理，在使用重载函数方式下，各重载函数的公共部分最好也提出来作为一个单独的函数。

原则9.2 确保重载函数的所有版本有共同的目的和相似的行为

原因

C++(相对于C)之所以提供重载这一新特性(允许用相同的函数名)，就是为了让这些具有共同目的的和相似行为的函数能归为一类，方便理解和使用。如果目的和行为不同，却要重载，除了令人费解，还会有其他作用吗?

原则9.3 避免重载在指针和整型类型上

9.3.1 例子

```
class NonPortable_T
{
public:
 void foo(int someInt);
void foo(char* charPtr);
};
foo(0);    //试图调用foo(int)
foo(NULL);    //试图调用foo(char*)
```

9.3.2原因

因为0既是整型的合法值，也是指针类型的合法值，所以此时就有二义性。最终结果如何有赖于编译器：有些编译器会报错，有些是根据NULL的定义选择一个函数，但即便如此也不能保证编译器选择的正是编程者想要的(感觉上好象经常与编程者想要的相反)，就算编译器总能正确"猜到"编程者的意图，这种代码的兼容性也成问题。

原则9.4 尽量避免重载在模板类型上

9.4.1例子

```
template<class TYPE_T>
class SomeTemplate_T
{
  public:
    void function(int parameter);
    void function(TYPE_T parameter);//若TYPE_T实例化为整型怎么办
};
```

9.4.2 原因

因为你不能控制用户怎样实例化模板，所以存在潜在的二义性，最好不这样设计。

第10章　操作符

C++允许自定义类重写操作符函数。这是一个非常强大的功能，但要做到自己实现的操作符能和缺省操作符一样自然顺畅并不容易。

可自定义的操作符有：

operator new

operator delete

operator new[]

operator delete[]

+ - * / %

+= -= *= /= %=

& | ^ ~

&= |= ^=

>> <<

>>= <<=

! = < >

!= == <= >=

&& ||

++ --

-> ->*

, () []

不可自定义的操作符有：

.

.*

::

?:

new delete sizeof typeid

static_cast dynamic_cast const_cast reinterpret_cast

原则10.1　遵守操作符原本的含义，不要创新

10.1.1　说明

和前面所说的new和delete一样(见“原则6.6　自定义类的new/delete操作符一定要　符合原操作符的行为规范”)，操作符原本的行为方式(系统缺省)实际上是该操作符的标准接口。保持该接口的稳定一致是使用该操作符(不论是系统缺省还是自定义的)的基础和前提。

10.1.2例子

让operator!()打印告警信息看似很有新意，但它违反常理，出乎使用者的意料，并不是好的实现方法。

原则10.2 确保自定义操作符能和其他操作符混合使用

例子

如果将异或操作符operator^()重定义成求幂操作符，则x^y+b将被编译器解释为x^(y+b)，而不是所希望的(x^y)+b，因为编译器是按异或的优先级确定运算顺序的。

混用的方式很多，任何对操作符功能的扩充都要谨慎，以防混用时出问题。

"原则10.1 遵守操作符原本的含义，不要创新"和本原则有一定的重叠，但前者侧重于完全破坏原有接口时的问题;后者更注重在保持操作符原有接口的情况下增添其他功能所需注意的问题(例子不太恰当，但实在想不出更合适的了)。

原则10.3 区分作为成员函数和作为友元的操作符

10.3.1 说明

若操作符需要左值(lvalue，将被赋值)，应将其定义为成员函数，否则应是友元。

10.3.2 例子

操作符operator+=()、operator=()等都需要左值，应是成员函数;而操作符operator==()、operator+()不需要左值，应是友元。

10.3.3 原因

●操作符作为成员函数可以确保左值(被赋值者)一定是该类的对象。

●操作符作为友元，使得第一个操作数可以是任何类型，不一定非要是自定义类，比如：若是友元，obj+1或1+obj均可；若是成员函数，1+obj不行。

原则10.4 关于前后缀操作符

10.4.1 说明

操作符operator++()和operator--()既可以是前缀操作符，又可以是后缀操作符，且运 算动作不同。早期(八十年代)的C++编译器无法区分自定义类的前、后缀操作符函数：它们名字一样，参数一样(都只有隐含参数this)。现在的编译器会让后缀操作符多带一个int类型的参数，具体格式为：

```
//后缀，参数无用，只是为区分<重载)前后缀用，其值为零
MyClass_T const operator++(int);
//后缀，参数无用，只是为区分<重载)前后缀用，其值为零
MyClass_T coast operator--(int);
//前缀
MyClass_T& operator-+(void);
//前缀
```

```
MyClass_T& operator--(void);
```

10.4.2返回值类型

后缀操作符应返回一个常量对象(如上所示)，目的是为了防止类似于myObj++++的用法：第二个后缀操作符试图对一个临时(匿名)变量做加1操作，这是不合理的(你可以试一下int++++)。同时，该返回值也不能是对象本身的引用，因为根据后缀操作符的定义，操作数(对象)本身不参与随后的运算，而只会产生一个原操作数的副本，且由这个副本参与随后的运算。

前缀操作符应返回对象本身(的引用)，因为++++myObj是合理的。

原则10.5 确保相关的一组操作符行为统一

10.5.1 例子

●如果实现了operator==()，也应实现operator!=();

●如果a为真，则!a应为假。

10.5.2 从实现上确保行为统一

为了确保相关的一组操作符行为统一，最好能基于同一个(核心)实现。比如，为保持前、后缀操作的一致性，应该基于前缀操作符函数来实现后缀函数，然后再补充其他需要的动作(见下例)。这样也好维护：加1功能的改变只需改动前缀函数，而后缀函数自动与之保持一致。

同样的，operator+()可基于operator+=()、operator!=()可基于operator==()。

10.5.3 例子

```
//后缀++:
MyChss_T const
MyClass_T::operator++(int)
{
    MyClass_T oldValue=*this;
    ++(*this);    //调用前缀++来实现加1这一核心功能
return (oldValue);
}
```

原则10.6 绝不要自定义Operator&&()、Operator‖()和operator,()

10.6.1 例子

```
char* name;
//...
if((name!=NULL)&&(name[0]=='y'))
```

```
{
//...
}
```

10.6.2　原因

(绝大部分主流的)C和C++编译器都执行布尔表达式的短路判断(这是国际标准)。如上例所示，若name!=NULL为假，整个表达式肯定为假，此时编译器不会再执行剩余表达式的计算了。但是，如果你自定义了operator&&()，则：

if(expression1 && expression2)就应该是(当operator&&()是成员函数时)：

if(expression1.operator&&(expression2))

或是(当operator&&()是友元函数时)：

if(operator&&(expression1,expression2))

不论那种形式，函数operator&&()在被调用前，所有参数都要准备好。也就是说，　expression1和expression2都要被计算完毕。这就意味着不再有短路判断。另外，由于　expression1和expression2都是函数的参数，再也不能确定谁将被先计算，谁将被后计算；　而原先是从左向右计算的，即expression1肯定被先执行。

这种变化是隐含的，会令使用者在很长一段时间百思不得其解。

10.6.3　缺省operator,()的动作描述

operator,()看上去比较特殊，其实和&&以及||类似：

●先计算逗号左侧的表达式。

●然后计算逗号右侧的表达式。

● 整个逗号表达式的返回值是右侧表达式的返回值。

第11章　类型转换

C++提供了比C语言强大得多的类型检查功能，目的是让编译器尽可能多地在编译时发现错误，而不是把它们漏到运行时。要知道，错误越早发现，付出的代价越小。当然，你也可以部分或全部绕过这些类型检查。有时候这是需要的，但通常是有害的。对类型转换，尤其是隐式的类型转换保持警觉是必要的素养。

原则11.1　尽量避免强制类型转换

原因

●如果你在自己的代码中经常做强制类型转换，通常意味着你的设计有问题。

●尤其是static_cast、const_cast、reinterpret_cast和C风格的强制类型转换，会绕过编译器的类型检查，从而可能将真正的问题隐藏起来。

原则11.2　如果不得不做类型转换，尽量用显式方式

11.2.1　原因

●隐式的类型转换容易被开发者和阅读者忽略，从而可能隐藏错误。

●显式方式更清晰、直观，可控性好。

11.2.2　例子

```
/*------------------------------------------------------
*tableSize的类型是long
*KEY_T和VALUE_T是模板类型
------------------------------------------------------*/
//隐式类型转换，可以试一下，结果不对
long hashTableSize=(
tableSize*sizeof(VALUE_T)/5-   //<=20% of value array size
tableSize*sizeof(KEY_T)    //minus key node array size
)/sizeof(size_t);    //unit size of hash array
//提供显式的类型转换，可控性好
long hashTableSize=(
long(tableSize*sizeof(VALUE_T)/5)-    //<=20% of value array size
long(tableSize*sizeof(KEY_T))    //minus key node array size
)/long(sizeof(size_t));    //unit size of hash array
```

原则11.3　使用新型的类型转换并确保选择正确

11.3.1　说明

新型的类型转换包括：

●const_cast：将常量或易失量(volatile)转换成非常量或非易失量。

●dynamic_cast：将继承类转换成派生类或相反，运行时确定转换是否成功。

●static_cast：非常类似于C风格的强制类型转换，但不能将常量转换成非常量，因为const_cast是专做此事的。

●reinterpret_cast：例如将指针转换成整型就需要该转换符；它依赖于编译器的实　　现，用处不多。

11.3.2　原因

新型的类型转换减少了原先(C风格)的内在缺陷。它们允许用户选择适当级别的转换符，而不是像C那样全用一个转换符。

如果你的编译器不支持新型的类型转换，那很可能是更换编译器的时候了，因为很古老的C++标准(1992年之前)就支持该功能了。

原则11.4　用虚函数方式取代dynamic_cast

原因

●虚函数不需要在编码时确定对象的真实类型(虚函数指针指哪就执行哪个函数体)；而dynamic_cast必须告知要转成的类型，运行时若类型不当还要抛出一个异常(throw an exception)，所以还需提供失败检测机制。

●虚函数的执行效率远高于dynamic_cast，因为dynamic_cast的转换机制比虚函数要复杂得多。

●当增加或删除一个派生类时，dynamic_cast用法还得增减相应的代码，见“原则5.13　关于C++中的分支用法选择”。

●提供完善的dynamic_cast失败检测机制非常困难，因为运行时各种情况的组合可能非常多。

原则11.5　自定义类最好提供显式而不是隐式转换函数

11.5.1　例子

```
class LibraryBook_T
{
  public:
    int operator int(void);     //隐式的转换函数
    //...
};
LibraryBook_T book;
cout<<book<<end1;     //输出一个整型也许不是使用者的本意
class LibraryBook_T
{
  public:
```

```
    int asInt(void);    //显式的类型转换
//...
};
LibraryBook_T book;
cout<<book.asInt()<<end1;    //使用者明确表示要输出一个整型
```

11.5.2 原因

●隐式类型转换函数是在编译器认为合适时，由编译器执行的，有时候这并不是使用者的本意，由此可能会引起代码的误动作。

●显式类型转换的发生时刻和形式都是使用者明确认可的。

原则11.6 用关键字explicit防止单参数构造函数的类型转换功能

11.6.1 说明

单参数构造函数的一个隐含意思是：将该参数转换成该类的一个对象。这实际上是
一个隐式的类型转换。要注意，参数定义缺省值的构造函数，只要在调用时能形成单参数形式，同样具有隐式类型转换功能。

关键字explicit用来禁止编译器隐式调用该构造函数，以防止隐式的类型转换，但对显式调用该函数则没有问题。

11.6.2例子

```
class MyClass_T
{
  public:
    //将int隐式转换成MyClass_T
    MyClass_T(int);
    //可将float隐式转换成MyClass_T
    MyClass_T(float f1, float f2=0.0, float f3=0.0);
    //禁止将char隐式转换成MyClass_T
    explicit Myclass_T(char);
    //...
};
void myFunction(MyClass_T anObj);
myFunction(15);    //你能想到这样会成功吗
myFunction(3.14);    //这样也行
//这句不行，编译器报错，因为显式禁止将字符型传成MyClass_T
myFunction('A');
```

11.6.3 原因

同"原则11.5 自定义类最好提供显式而不是隐式转换函数"。

注意，这种隐式转换更隐蔽(因为可能连编程者都没意识到某个构造函数还隐含着类型转换的功能)，所以更要仔细防范。

原则11.7 限制隐式类形转换的类型数

例子

```
From_T::operator To1_T(void);    //将类型From_T隐式转换成类型To1_T
From_T::operator To2_T(void);   //将类型From_T隐式转换成类型To2_T
void ambiguous(To1_T to);    //以To1_T对象为参数的函数
void ambiguous(To2_T to);    //重载，以To2_T对象为参数的函数
Prom_T from;
ambiguous(from);    //二义性：哪个重载函数应该被调用
```

原则11.8 避免多个函数提供相同的类型转换

例子

```
//转换操作符函数：将类型From_T隐式转换成类型To_T
Prom_T::operator To_T(void);
//转换构造函数：将类型From_T隐式转换成类型To_T
To_T::To_T(const From_T& from);
void ambiguous(To_T to);
From_T from;
smbiguous(from);    //二义性：哪个隐式类型转换应该被调用
```

第12章　友元

和其名字相反，如果使用不当的话，友元有可能成为敌人而不是朋友。这是因为友元会令C++的封装和保护机制失效，从而失去面向对象语言的一些重要的特性。

原则12.1　少用友元

12.1.1　说明

不鼓励用友元，除非：

●某些操作符(成员)函数，见"原则10.3 区分作为成员函数和作为友元的操作符"。

●协作类之间的内部通信，如container(容器模型)或iterator(迭代器)等。

12.1.2　原因

任何使用友元的地方都必须忍受：

●类间高耦合(参见"原则4.5　降低类间的耦合度")：

打破封装，暴露出具体实现，从而使友元和类的实现紧密耦合在一起，失去了各自独立变化而不互相影响的特性。

●降低可继承性(面向对象语言的另一个基本特性)：

因为友元是不能继承的(父亲的朋友不意味着是儿子的朋友)，所以子类必须提供自己的友元以实现类似父类友元要做的工作。

原则12.2　减少拥有友元特权的个数

12.2.1　说明

如果类间必须用友元，只将确实需要的(成员)函数指定为另一个类的友元即可，而不要将整个类都指定为(另一个类的)友元。

12.2.2　原因

减少友元带来的负面影响。

第 13 章　模板

(通过)模板可以衍生出一系列的类和函数。这是一种形式的代码复用。但要注意，这种形式的复用是源码级而不是目标代码级的。也就是说，对模板的每一次实例化都会产生一份新的源代码。与此相反，继承(另一种形式的代码复用)允许复用基类的大部分目标代码。

滥用模板会造成三个后果：第一，代码规模的过度膨胀；第二，当修改(已存在的)模板时，很难预料是否会对原先正常工作的代码造成不良影响；第三，很难确保模板所有可能的合法实例化都能正常工作，因为这是一个无穷集合，往往一个事先未预料到的特例就会引出问题。所以，模板的使用要仔细且有节制。

原则13.1 使用模板如果有限制条件一定要在注释和文档中描述清楚

13.1.1　例子

```
  //TYPE_T必须是能做小于运算且返回布尔值的类型
template<class TYPE_T>
void mySort(TYPE_T[] items, int itemNum)
{
    TYPE_T* pMinItem, pCurltem;
    //...
    if(*pMinItem < *pCurItem)
    //...
}
```

13.1.2　原因

●如果模板不是适用于所有类型，一定要在第一时间、清晰、彻底地提示使用者。

●注意，有些限制条件是隐含的，它们同样要在第一时间、清晰、彻底地提示使用者。比如:

```
template<class ELEMENT_T>
class Array_T
{
  public:
    explicit Array_T(int sizc);
    //…
  private:
```

```
    ELEMENT_T i_element[];
};
template<class ELEMENT_T>
Array_T<ELEMENT_T>::Array_T(int size):
i_element(new ELEMENT_T[size])    //隐含要求ELEMENT_T必须有缺省构造函数
{
     //...
}
```

原则13.2 模板类型应传引用/指针而不是值

原因

实例化时的类型可能是规模很大的自定义类，传值代价很高(参见"原则3.5 对于非内置类型参数应传递引用(首选)或指针")；另一方面，即便实例化时的类型是内置类型，传引用/指针也没有重大缺陷(参见"原则3.4 对于内置类型参数应传值(除非函数内部要对其修改)")。

原则13.3 注意模板编译的特殊性

13.3.1 说明

●延迟编译，导致错误被隐藏：

对于模板本身，编译器只是检查模板语法是否正确，并不作真正的编译；真正的编译要等到实例化时，生成了具体的代码后才做，所以错误可能被隐藏起来。

●增加编译时间：

实例化模板是指在代码中用具体的类替换模板，此时编译器要根据模板生成相应于具体类的代码，然后再参与其他代码的编译，可见需要更长的编译时间。

●模板的错误提示比较晦涩：

编译实例化代码产生的错误要映射回模板后才显示给编程者看，但映射毕竟不是出错代码本身，所以有时很难看懂，有时甚至失真。

●中断编译可能导致错误：

编译在被异常中断后(比如发现了很多错误，用户不想等到编译完成，而直接强行中断编译)，再重新编译时可能出连接错误。这是因为编译器没有识别出上次异常中断造成的模板实例化失败(自动生成的代码不全)。解决的办法是，清除/忽略所有编译的中间结果，然后重新编译全部源代码。

13.3.2例子

```
/* ------------------------------------------------
*延迟编译，导致错误被隐藏：单独编译此段代码
*不会有编译错误，但是如果有一天用一个不支持
```

```
*operator<<()的类实例化该模板，就会产生编译错
-------------------------------------------------------*/
template<class TYPE_T>
void
myFunction(TYPE_T& aParameter)
{
cout<<aParameter<<endl;
}
/*-------------------------------------------------------
*模板错误提示的例子请见"原则13.4  嵌套template
*的>>中间要加空格以区别于operator>>"
-------------------------------------------------------*/
```

原则13.4 嵌套template的>>中间要加空格以区别于operator>>

13.4.1 例子

```
template<POINTEE_T> class Z_AutoPtr_T;
template<ELEMENT_T> class Z_Array_T;
//int之后的两个右尖括号，中间要加空格
const Z_AutoPtr_T< Z_Array_T<int> >apValueArray;
/*-----------------------------------------------
*否则我的编译器会提示
* "`', 'expected instead of`>>'"
*你能理解吗？我是费了半天劲才发现是什么问题
-----------------------------------------------*/
```

13.4.2 原因

否则编译器可能理解为是操作符>>，而不是模板的右尖括号。

第 14 章　表达式和控制流程

实际上，程序的大部分工作是建立在表达式上的，而大部分表达式又最终转换成流程控制。所以，保证表达式和控制流程的清晰、易读和健壮，会使开发和维护工作变得简单。

原则14.1　让表达式直观

14.1.1　例1

```
if(strlen(someString))            //不直观
if(strlen(someString)!=0)         //这样较好
if(!strcmp(string1, string2))     //费解
if(strcmp(string1, string2)==0)   //直接了当
```

14.1.2　例2

```
if(queue.isEmpty())               //布尔表达式，足够清晰
```

14.1.3　例3

```
if(!queue.isNotFull())            //没有必要用否定之否定
if(queue.isFull())                //开门见山
```

14.1.4　原因

●阅读者不需心算一下(或几下)才能明白表达式的含义。

●更直观的表示方法，本身实现起来也不麻烦。

原则14.2　避免在表达式中用赋值语句

14.2.1　例子

```
//实在没必要这样写，理解起来不顺畅
  area = (width = width * SCALE_FACTOR) * (height = height * SCALE_FACTOR);
  //常见，但也不是好风格
  while( (c=getchar()) != EOF )
```

14.2.2原因

●表达式中用赋值语句令表达式不直观。

●危险：

编译器计算表达式的顺序是依照很复杂的优先级和左/右扫描规则，可能导致赋值语句不按你以为的先后顺序运算，从而产生错误。

原则14.3　不能将枚举类型进行运算后再赋给枚举变量

14.3.1　例子

```
enum Bit_T{
```

```
    BIT_ZERO=1，
    BIT_ONE=2，
    BIT_TWO=4
}b1，b2，b3;
b1=BIT_ZERO;//可以
b2=BIT_ONE;  //可以
b3=b1|b2;//许多编译器会报告诸如"can't assign int to enum Bit_T"之类的错误
```

14.3.2 原因

●在C++中，枚举进行运算后自动变成整型，若再赋给枚举变量，就会导致编译错误"类型不匹配"。

●C++之所以这样处理，是因为枚举在运算后，其结果可能不再是枚举类型的合法值。所以，C++只允许在枚举上做一种操作：赋值。

原则14.4 避免对浮点类型做等于或不等于判断

14.4.1 例子

```
float myFunction(void);
//...
if(myFunction()==3.14)  //精确比较浮点数
//...
if(myFunction()<=3.14)  //范围比较浮点数
//...
```

14.4.2 原因

●由于浮点数在计算机内部表达是用二进制，它不能精确地表示所有的十进制浮点数(有舍入问题)，所以看似相等或不等的两个浮点数，在计算机内部的表示可能不同(比如最后一位不等)，此时用等于或不等这种精确比较方法就可能得出与期望相反的结果。

●用大于等于或小于等于是比较明智的替代方法。

原则14.5 尝试用范围比较代替精确比较

例子

```
//i!=LENGTH是精确比较
for( i=0; i!=LENGTH; ++i )
//i<LENGTH是范围比较，更安全
for( i=0;i<LENGTH;++i )
```

原则14.6 范围用包含下限不包含上限方式表示

14.6.1 例子

```
//下限0在范围内，上限LENGTH在范围外
for( i=0;i<LENGTH;++i )
```

14.6.2 原因

●上限减去下限即为范围大小。

●当上限等于下限时，范围为空。

●上限永远不小于下限，即使范围为空。

●范围的统一表示会容易理解，减少问题。

原则14.7 关于goto

原因

由于goto的权力太大(能从循环内跳出跳入；能从函数中跳出跳入；能跳过初始化语句等等)，它打破了语言中许多原则，贻害无穷。另外，经数学的严格证明，所有goto语句都可由非goto的语句替换，也就是说，goto不是编程必须的。因此，绝大多数相关书籍都强烈建议编程者不要用goto。本书也持类似观点。

但另一方面，我们也需要正反两方面的信息，才能全面了解它：

●goto本身的效率很高。

●若运用得当，用goto的实现可读性好于强行用非goto替代的实现。

合理的态度是：慎重再慎重地把goto(以及break、continue等goto的变种)用在的确恰当的地方；如果拿捏不准，不要使用。

这里给出几个使用goto的限制条件供参考：

●不能用goto跳出/跳入循环体。

●不能用goto跳出/跳入程序块(block)。

原则14.8 在循环过程中不要修改循环计数器

14.8.1 例子

```
for(i=0;i<length;i++)
{
    if(someConditions)
    {
   i=newVahe;     //不要在循环体内修改循环计数器
       //...
}
}
```

14.8.2 原因

- 使得该循环不好理解。
- 本身就容易出错：其后的维护也容易出错。

第15章 宏

在C++中必须用宏的地方很少，因为C++提供了更强大、更安全的替代方法。但就在这些为数不多的使用宏的地方，也要小心避免出问题。

原则15.1 彻底用常量替代(类似功能的)宏

15.1.1 例子

```
//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;
```

15.1.2 原因

●宏是在预编译时用定义值(如例中的3.14)全程替换宏名(如例中的PIE)的，这样在编译时编译器不知道宏名，报错时直接报相应值的错误，不易理解。

●跟踪调试时也是显示的值，而不是宏名。

●宏没有类型，也就不能做类型检查，不安全。

●宏没有作用域。

●常量和宏的效率同样高：像上例中的常量PIE，编译器在最终生成代码时会用3.14全程替换掉；若PIE还和其他常量做运算，编译器也会先计算好，然后把结果放在目标代码中。

原则15.2 代码中的数值应由一个有意义的标识符代替

15.2.1 例子

```
//宏定义的方法
#define PIE 3.14
//常量定义的方法
const float PIE=3.14;
//枚举的方法
enum
{
     PIE=3
};
```

15.2.2原因

●用有意义的标识符比直接用值更好理解。

●以后修改该值也方便，只需改动定义即可，而不需在代码中改动所有直接引用该值的地方。

●效率和直接用值一样。

●例外情况是，一般不必对值0和1定义标识符。

15.2.3疑问

●不是说不再用宏做这种事情了吗?

这条原则(过去)在C语言中应用的结果是：用宏的方式替代值。从本原则的目的来说(参见上面的"原因"一节)用宏也能达到。为了在此基础上更进一步，才有用常量替代宏的做法。所以"原则15.1　彻底用常量替代(类似功能的)宏"的前提是执行了本条原则。

●枚举又是怎么回事?

在某些情况下，枚举又是常量的改进方法：它可以将一组常量归类，同时又减少了匿名空间的标识符数量：另外在类中使用尤其方便，具体说明请参见"原则2.7用cnum取代(一组相关的)常量"。

但是，正如你所看到的，枚举也有其局限：只能表示(有符号)整型值，其他就无能为力了。

原则15.3　若宏值多于一项，一定要使用括号

15.3.1　说明

这是一条C语言的原则。由于建议在C++中尽量少用宏，所以不太可能有这种用法，但为以防万一，还是顺手把它放在这里。

15.3.2　例子

```
//若宏值多于一项，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
malloc(5*ERROR_STRING_LENGTH)
//应这样定义
#define ERROR+STRINC_LENGTH(100+1)
```

15.3.3　原因

当宏用在表达式中时，防止出现计算混乱。

原则15.4 不要用分号结束宏定义

例子

```
/*-------------------------------------------
*用分号结尾，结果导致下面第二条语句变成
*"channel=16;-1;"，编译出错
-------------------------------------------*/
#define CHANNEL_MAX 16;
int channel=CHANNEL_MAX-1;
```

原则15.5 彻底用inline函数替代(类似功能的)宏函数

15.5.1 例子

```
//宏定义的方法
#define MAX(x, y)( (x)>(y) ? (x ): (y) )
int a=1, b=0;
MAX(a++, b);    //相当于((a++)>(b)?(a++):(b))
MAX(a, "Hello");    //相当于((a++)>("Hello")?(a++):("Hello"))
//inline函数的方法
template<class TYPE_T>
inline TYPE_T& max(TYPE_T& a, TYPE_T& b)
{
return(a>b)?a:b;
}
```

15.5.2 原因

●宏函数也是在预编译时处理的，编译有错时错误信息不易理解。

●无法单步跟踪调试宏函数本身(编译器一般也不能单步跟踪调试inline函数，但有不少编译器可以为调试自动将inline函数传成普通函数，但从来没听说能将宏函数传成可调试的某种方式)。

●没有类型，参数可以是任何东西，极不安全。

●没有作用域的概念。

●inline函数会在目标代码中展开，和宏效率一样高。

原则15.6 不好被替代的宏函数

15.6.1 说明

宏函数的个别用法很独特，inline函数取代不了。

15.6.2 例子

```
/*-----------------------------------------------------------------
*通过设置不同的编译条件，可以产生调试版和正式发布版。在调试版中可以
*加入很多调试代码，帮助分析、追踪程序；而在正式版中可以完全去掉这些
*代码。这样既不影响正式版的效率，又能在调试版提供丰富的调试信息，
*可谓两全其美。
*下面的statement就是调试代码，而最终是否参加编译由
*Z_EXECUTE_WHEN_DEBUG()宏函数和编译条件Z_DEBUG OPEN(也是一个宏)决定。
*可以看出，statement不是普通的参数，这个宏函数很难用inline函数取代
-----------------------------------------------------------------*/
#if defined(Z DEBUG_OPEN)
```

```
#define Z_EXECUTE_WHEN_DEBUG(statement)\
    statement\
    //if close debug
 #else
     #define Z_EXECUTE_WHEN_DEBUG(statement)
 #endif
```

15.6.3 原因

当不想指明或不能指明参数类型时，宏函数比inline函数更适合。

原则15.7 函数宏的每个参数都要括起来

例子（这也是C语言的原则，姑且放在这吧）

```
//参数w未括起来
#define WEEKS_TO_DAYS(w) (w*7)
//结果是1+2*7=15，而不是(1+2)*7=21
totalDays=WEEKS_TO_DAYS(1+2);
//应这样定义
#define WEEKS_TO_DAYS(w)((w)*7)
```

原则 15.8 不带参数的宏函数也要定义成函数形式

15.8.1 例子

```
//这种形式较好
#define HELLO()printf ("Hello.")
//HELLO不带括号不清晰
#define HELLO ()  printf("Hello.")
```

15.8.2 原因

括号会暗示阅读者此宏是一个函数

原则15.9 用{ }将函数宏的函数体括起来

15.9.1 例子

```
//未给函数体加{}
#define MY_ASSERT(condition)\
if (!(condition))\
{\
    assertError(_FILE_,_LINE_);\
}
//出问题了
if((x<0)&&(y>0))
```

```
        MY_ASSERT(x>y);  //MY_ASSERT()的if和下面的else接到一起了，错
    else                 //想和if((x<0)&&(y>0))连用，但被MY_ASSERT()破坏了
        MY_ASSERT(y>x);
    //应当加上{ }
#define MY_ASSERT(condition)\
{\
    if(!(condition))\
    {\
        assertError(_FILE_,_LINE_);\
    }\
}
```

15.9.2 原因

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。

15.9.3 例外

细心的读者可能注意到了，“原则15.6不好被替代的宏函数”的例子中，宏函数Z_EXECUTE_ WHEN_DERUGO 并未用{}将函数体括起来。这是一个例外，因为参数statement有可能是半条语句：

```
void myFunction(void)
{
    //...
    Z_EXECUTE_WHEN_DEBUG(ssize_t ret=)
    read(fileDescriptor, buffer, length);
    //display error information
    Z_EXECUTE_WHEN_DEBUG
    (
     if(ret<0)
     {
      cout << "Fatal error in myFunction():system read()"
      "function return error=" << ret;
     }
    );
    //...
}
```

原则15.10 彻底用typedef代替宏定义新类型

15.10.1 例子

```
//不要用宏定义新类型
```

```
#define LongPtr_T long*
//下面这句话相当于"long* aPtr, bPtr;"，所以bPtr是长整型，而不是长整指针
LongPtr_T aPtr, bPtr;
//用typedef可以消除宏的弊病
typedef long* LongPtr_T;
LongPtr_T aPtr, bPtr;
```

15.10.2 原因

宏的最大弱点(特点)就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。这是导致期望和效果不同的主要原因。

原则15.11 不要在公共头文件中定义宏

15.11.1 说明

除非是用作防止头文件被多次引用的宏(参见"原则20.1 头文件多次引用的防范")。

15.11.2 原因

●增大命名冲突的可能：

由于宏不受作用域的控制，所以它可以出现在任何作用域中，这就大大增加了命名冲突的可能。对于不得不出现在公共头文件中的宏(如防止头文件被多次引用的宏)，要加前缀以减少冲突，参见"原则1.7 关于全局命名空间级标识符的前缀"。

●会将宏的负面影响扩散到所有引用公共头文件的代码中。

原则15.12 不要用宏改写语言

15.12.1 例子

```
//不要定义这类的宏
#define FOREVER for(;;)
#define BEGIN  {
#define END    }
```

15.12.2 原因

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。

第 16 章　异常(exception)处理

C++的异常处理是该浯言中最难于掌握的部分之一，同时又是最容易被忽略的地方。很多程序员都有这样的想法：如果出现问题的可能性不大，就先不予考虑，以后有机会再说。这是非常有害的想法，因为编程是一种非常严谨的工作，而面向对象语言又把程序复用提升到前所未有的高度，所以任何小的疏漏或隐患都会造成该代码不能被信任，从而不能在其上进行进一步的迭代开发。

原则16.1　确保代码在异常出现时能正确处理

16.1.1　说明

C++的异常分两大类，一类是系统产生的，比如用new申请内存失败(系统资源不足)；一类是编程者自己生成的(用来表示自定义的某种异常情况)。所有异常都有缺省处理方法，就是程序退出。对于系统产生的异常，(绝大部分情况下)程序退出是可接受的解决办法，即编程者几乎不用考虑此类异常。对于自定义的异常，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，一定要有说明文字，并能在第一时间告知使用者。

16.1.2　原因

●系统异常，要么意味着程序有错，要么意味着系统资源不够，总之是程序本身很难在运行时解决的问题，所以退出运行就成为简单而自然的事情。

●异常处理代价很高，所以对无能为力的系统异常(比如系统资源耗尽)，不予考虑是明智的。

●自定义的异常却要扩散出去(让别人处理)，应是罕见且必须事先约定好，所以必须有文字描述。

原则16.2　正确注释代码的异常处理能力

16.2.1　说明

一块代码(一般以类或函数为单位)根据是否对外扩散异常可注释为(不考虑系统异常)：

●不扩散：包括根本不产生异常，或所有异常都能被处理掉。

●扩散：要注明扩散出来的异常的详细描述。

●不知道(不产生、不处理)：代码本身不产生或所有自己产生的异常都被处理掉，但是会调用其他代码，而被调用的代码是否扩散异常已超出本代码的控制范围：可能现在不扩散，但将来难说。所以只能归为"不知道"类。

16.2.2　原因

正确且统一的异常注释能够帮助使用者正确地使用该代码，减少隐患。

原则16.3　减少不必要的异常处理

原因

●异常处理会使编译器向目标代码中插入相应的处理机制，由于C++的异常处理机制很强大也很完善，所以会(显著)增大目标代码的尺寸。

●异常处理机制需要占用额外的处理时间。

原则16.4 不要利用异常处理机制处理其他功能

原因

异常处理机制代价高昂，不要用来处理其他非异常情况，比如消息传递等。

原则16.5 注意模板类型可能会破坏异常处理的一些约定

16.5.1 例子

```
/*-----------------------------------------------
*该函数想表明它不向外扩散任何异常(通过throw()语句)，但实际上做不
*到，因为如果TYPE_T是一个自定义类，则TYPE_T可以提供自己的
*operator&()，并可以在函数operator==()传递参数时产生异常，由于这
*些异常不是在函数体中产生的，所以throw()抓不住它们
-----------------------------------------------*/
template<class TYPE_T>
bool
operator==(TYPE_T const& lhs, TYPE_T const& rhs)
  throw()
{
   //...
}
```

16.5.2 原因

●其实，破坏上例关于throw()约定的不是模板，而是自定义类型。但是模板会将这个问题隐藏起来，有雪上加霜的嫌疑，所以更要留神。

●另外，没人知道模板类型可能会抛出什么异常，所以包含模板的类或函数的异常处理能力只能归于"不知道"类。(参见"原则16.2 正确注释代码的异常处理能力"。)除非它们用"catch(...)"捕捉所有异常类型。但是一般代码很少用"catch(...)"，因为异常类型不明，所以可供选择的(对所有异常都是安全的)处理方式可能只有一种：终止运行。如果是这样，"catch(...)"就成为多余的举动，因为系统缺省就是干这件事的。

原则16.6 确保异常发生后资源还能被回收

原因

●因为异常发生后，当前代码运行序列被打断，转到相应的异常处理代码

(catch语句处)继续执行，所以正常的资源回收机制被跳过(在断点之后的某处)。如果没有应对措施，资源可能再也不会被回收，从而造成内存泄漏。

常见的用法是在构造函数中分配资源，在析构函数中回收，此时一定要考虑构造函数被打断怎么办(你能保证你的程序永远不会被强行终止吗?)。

●解决办法可以参见"原则6.4　谁申请谁释放"中关于自动指针和引用计数的说明。

如果(异常最终导致)程序退出运行，有的操作系统(如SUN公司的Solaris)能回收全部分配给该程序的系统堆资源；而另一些操作系统(如微软的WINDOWS系列，但我不能肯定WINDOWS2000和WINDOWSXP怎样处理此事)却没有能力回收分配给该程序的系统堆资源，其结果是一旦有内存泄漏，重新启动程序不能消除，只有重新启动计算机才行(所以此类操作系统上的程序需要花更大的精力防范内存泄漏)。

原则16.7　特别当心析构时发生异常

原因

●c++的异常处理机制规定，若在异常处理期间(catch语句块中)又发生异常，程序将被终止。

● 如果析构时发生异常，而这个异常又不能被该析构函数处理，则导致该析构函数被打断(异常将向外扩散)。

●如果该析构函数本身就是由于某个异常而被C++异常处理机制自动调用的，则再次被打断就意味着"异常处理期间又发生异常"，程序肯定会被终止。

●可见析构时的异常可能导致程序退出运行，所以要特别当心。

原则16.8　抛出的异常最好是一个对象

16.8.1　说明

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。

16.8.2　原因

●若抛出的异常只是一个内置类(的值)，则表达的错误信息可能不够充分：

比如，你无法断定该值是否是全局唯一的，因而无法断定接到的异常是否是你所知道的那种异常。

●扩展性也较差：

对象可以包含多个数据成员，并能继续扩充。

原则16.9　捕捉异常时绝不要先基类后派生类

例子

```
try
{
```

```
    //  …
   }
   //先捕捉基类异常
   catch(Dase_T const& anException)
   {
    //...
   }
   /*----------------------------------------
   *再捕捉派生类异常，永远也抓不住了，都被上面的catch拦截了
   ----------------------------------------*/
   catch (Derived_T const& anException)
   {
    //...
   }
```

原则16.10　捕捉异常时用引用

16.10.1　例子

```
  try
  {
    //...
  }
  //捕捉异常用常量引用
  catch(MyException T const& anException)
  {
    //...
  }
```

16.10.2　原因

●如果用拷贝，由于不知道异常是何种原因造成的，所以拷贝有可能不成功(比如异常是由于内存耗尽造成的)。

● 拷贝不能得到派生类对象，因为在拷贝时，派生类对象会被切削成(sliced)基类对象(关于切削请见"原则3.5　对于非内置类型参数应传递引用(首选)或指针")。

●如果是传指针，那么用完之后是否需要清除(回收)?谁来清除?

● 需要判断指针是空吗?如何得知该指针所指对象是否还有效?

第17章　代码格式

统一、合理、美观的代码格式能显著增强可读性和可维护性，还可以帮助预防和发现代码错误(问题越早发现代价越小)。因此养成良好的编码风格是有理由的。当然，什么样的格式才是好的格式没有统一的标准。此处介绍的原则源于Bjarne Stroustrup—C++的创始人。我们认为他的风格很不错，同时我们也相信他对C++的理解比许多人都更深入。

原则17.1　水平缩进每次用两个空格

17.1.1　说明

其他字间填充也用空格。不建议使用制表符(tab)。

17.1.2　例子

```
class Z_AbstractMutex_T
{
    public:
    Z_AbstractMutex_T(void);    //default constructor
    virtual ~Z_AbstractMutex_T(void);    //for destruct derived obj.
    virtual int lock(void)=0;    //lock interface
    virtual int unlock(void)=0;    //unlock interface
    virtual int trylock(void)=0;    //trylock interface
    //------------------------------------------------------------
  private:    //DISALLOW USE ANYWHERE
    Z_AbstractMutex_T(       const       Z_AbstractMutex_T&);    //copy
constructor
    Z_AbstractMutex_T& operator=(const Z_AbstractMutex_T&);//operator=
};
```

17.1.3　原因

●中国人的书写习惯说明用两个空格的水平缩进就已经足够清晰。

●水平缩进过多会使深层嵌套代码过于偏向右边。

●水平缩进过多还会影响右边的行末注释(行末注释希望对齐，且要有足够的空间写内容，另参见“原则17.13　每一行不超过78个字符”)。

●上例中public缩进两格，正文再缩进两格，正好；否则正文也太靠右了(类似的例子还有switch-case的缩进等)。

●制表符的宽度是可自定义的，如果用在代码中会造成在不同环境下，代码的缩进和对齐不同，从而造成(显式)混乱的情况，因此不建议使用；但是在编辑代码时，可将制表符定义成(如果可以)“自动替换为两个空格”方式，以加快录入

速度。

原则17.2 不要在引用操作符前后加空格

17.2.1 说明

引用操作符指“.”“->”和“[]”。

17.2.2 例子

```
foo.bar，pFoo->bar，foo[bar]
```

17.2.3 原因

不要打断语义的连贯性。

原则17.3 不要在单目操作符和其操作对象间加空格

17.3.1 例子

```
!foe，~foo，++foe，-foo，+foo，*pFoo，&foo，sizeof(foo)，(int)foo
```

17.3.2 原因

●不要打断语义的连贯性。

●在阅读时帮助区分符号相同的单目和多目运算符。

原则17.4 不要在“::”前后加空格

17.4.1 例子

```
int result=Base_T::someFunction();
```

17.4.2 原因

“::”前后的标识符紧密相关，加入空格会打断其语义的连贯性。

原则17.5 在“,”“:”之后(而不是之前)加空格

17.5.1 例子

```
for(i=0, j=0; i<l0; i++, j++)
```

17.5.2 原因

符合英语书写和阅读理解的习惯。

原则17.6 在关键字和其后的“(”间加一个空格

17.6.1 原因

●加空格会更明显地表明这不是一个函数调用。

●有几个关键字例外：

它们是asm、catch、operator、sizeof、typeid；另外强制类型转换符const_cast、 dynamic_cast、reinterpret_cast、static_cast之后也不要空格，虽然跟随它们的不是“(”而是“<”。这样做的原因是，将这些关键字看成函数调用更好理解。

17.6.2 例子

```
//关键字if之后加一个空格
if (a==b)
//...
//关键字catch之后不要加空格
try
{
    //...
}
catch(MyException const& anException)
{
  //...
}
```

原则17.7 文件中的主要部分用空行分开

说明

文件的主要部分是指“原则19.7 关于文件的段落安排”中排列的那些项目，它们之间要用空行分开。

原则17.8 函数间要用空行分开

例子

```
void function1(void)
{
    //...
}
void function2(void)
{
    //...
}
```

原则17.9 组局部变量声明和代码之间用空行分开

17.9.1 例子

```
void  myFunction( void )
{
     int  i;
     char  c;
for(i=0;i<LENGTH;  i++)
```

```
    //...
}
```

17.9.2 原因

声明和执行代码分开显得清晰。

原则17.10 用空行将代码按逻辑片断划分

原因

就像文章要分段一样。这样便于阅读。

原则17.11 可以考虑将if块和else/else if块用空行分开

17.11.1 说明

特别是每块都很复杂时。

17.11.2 例子

```
if (a>b)
{
  //...
}

else if (a=b)
{
 //...
}

else
{
//...
}
```

原则17.12 函数返回语句要和其他语句用空行分开

17.12.1 例子

```
MyClass_T&
MyClass_T::operator=(MyClass_T const& rhs)
{
 if(this!=rhs)
  {
    //...
  }
```

```
return(*this);//返回语句之前加一空行
}
```

17.12.2 例外

除非该函数过于简单（比如只有一两条语句）。

原则17.13 每一行不超过78个字符

原因

●因为标准显示/输出设备（字符终端，缺省宽度的显式窗口和打印机等）的宽度都大于等于80个字符，所以每行不超过78个字符可以保证代码在显示、编辑和打印等情况下都不会发生因为宽度不够而折行的现象。

●限制为78个是因为行尾还隐含一个 至二个回车换行符（0x0d 或0x0d0a）。

原则17.14 当一条语句超过78个字符时按逻辑划分成不同行

例子

```
//类声明超长的处理方法
template< cladd KEY_T ,class VALUE_T>
class HA_CheckPointTable_T:public HA_AbstractTable_T<KEY_T,VALUE_T>
{
 public:
//函数声明超长的处理方法
HA_CheckPointTable_T(
const  char* pStoreFileName,
  size_t tableSize=0,
  size_t  syncInterval=60,
  float performanceCoeff=1.0
);
//...
};
//函数实现超长的处理方法
template<class KEY_T,class VALUE_T>
inline
HA_CheckPointTable_T<KEY_T,VALUE_T>::HA_CheckPointTable_T(
 //初始化列表超长的处理方法
    const char* pStoreFileName,
    size_t tableSize,
    size_t syncInterval,
```

```
    float performanceCoeff
    ) :
    i_DATA_FILE_NAME(new char[strlen(pStoreFileName)+20]),
    i_IDX1_FILE_NAME(new char[strlen(pStoreFileName)+20]),
    //...
  {
    //...
  }
  //条件语句超长的处理方法
  if((table.syncInterval()!=0)&&
    (table.syncInterval()!=size_t(-1)))
```

原则17.15　花括号{}要单独占一行

17.15.1　说明

并且和其控制语句的缩进相同。有个别例外：

●在do-while循环中，“}”要和while在一行。

●结构(struct)和联合(union)结尾紧跟变量定义，要和“}”放在一行。

●任何“}”后紧跟分号要放在一行。

17.15.2　例子

```
  //{}单独占一行，并且和if/for语句缩进相同
  if(clearRequested)
  {
    for(i=0; i<LENGTH; i++)
    {
      //...
    }
  }
else
{
    //...
  }
  //do_while例外
  do
  {
    //...
  }while(length-- >0);    //放在一行
  //结构和联合的例外
```

```
typedef struct
{
  //...
}Message;    //放在一行
//花括号后紧跟分号的情况
class MyClass_T
{
  //...
};    //放在一行
```

17.15.3 原因

●关于花括号的放置方法有很多风格，争论也有很长时间了。但我们坚持认为，单独放置更明显一点，查找和配对也更直接。

●另一种流行的风格是将左花括号和前面的语句放在一行，这样代码更紧凑，尤其是当花括号中的语句不多时更明显。这种风格源于杂志和书籍在排版时的美观要求，作为编程者，我们看问题的角度与排版不尽一致。

原则17.16 花括号中没有或只有一条语句时也不省略花括号

17.16.1 例子

```
//这种方式表达空循环不够清楚
for(i=0;i<TIMEOUT;i++);
//即便是空循环，也保留{}，这样就清楚多了
for(i=0;i<TIMEOUT;i++)
{
}
```

17.16.2 原因

●所有情况都是统一格式。

●阅读更简单直观。

●当增添或删除语句时不需要增减花括号，简单、不易出错。

●不会把不同层的if和else错误地配在一起。

原则17.17 不要在一行中放多于一条语句

17.17.1 例子

```
//两条语句放在一行
if (buffer.empty()) status=ERROR_EMPTY;
//这样要清晰一些
if(buffer.empty())
{
```

```
    status = ERROR_EMPTY;
}
```

17.17.2 原因

●简化每一行，清晰好读。

●便于统计工具计算代码行数。

●合并多条语句于一行没有明显的好处。

原则17.18 语句switch中的每个case各占一行

17.18.1 例子

```
switch(debugLevel)
{
    //这样不好
    case DEBUC_LEVEL_VERBOSE:case DEBUG_LEVEL_INFO:
    //...
    break;
    case DEBUG_LEVEL_ONE:
    //...
    break;
    default:
    //...
}
//应该这样
switch(debugLevel)
{
    case DEBUC_LEVEL_VERBOSE:
    case DEBUC_LEVEL_INFO:
    //...
    break;
    case DEBUG_LEVEL_ONE:
    //...
    break;
    default:
    //...
}
```

17.18.2 原因

●不容易漏掉信息。

●子语句也要符合“原则17.17 不要在一行中放多于一条语句”。

原则17.19 语句switch中的case按字母顺序排列

原因

●(对于比较复杂的switch语句)增减和查找case都比较容易，且不易遗漏。

●简单的switch语句也按此办理，可以简化规则，从而使阅读更顺畅。

●有人喜欢按调用频率排列case语句，把频繁使用的放在前面，以提高效率。总的来说，和减少出错几率相比，这点效率问题不太值得考虑。(其实，编译器会对switch-case做一些优化，其效果和相应的if-“else if”不同，所以不能保证放在前面的case 一定会先被判断。)

原则17.20 为所有switch语句提供default分支

原因

●安全：

对于那些确实不可能有default分支的switch语句，在default分支中打印出错信息，然后退出程序好了。这样，今后一旦增加新分支(比如枚举增加了新的值)，不必担心是否会遗漏相关的switch语句。(遗漏会造成程序退出，马上就能发现。)

●统一。

原则17.21 若某个case不需要break一定要加注释声明

17.21.1 说明

但case中没有执行语句的除外。

17.21.2 例子

```
switch(ch)
{
    case 'a':
    case 'A':
     action=Action_T::ABORT;
    break;
    case 'V':
    case 'V':
    verbosity++;
    //Fall through
    case 'd':
    //...
     break;
    default:
    //...
```

```
}
```

17.21.3 原因

帮助阅读者明白这是故意的而不是忘了，防止出错。

原则17.22 变量定义应集中放置、各占一行，并按字母顺序排列

17.22.1 例子

```
//不好理解每个变量的类型
int* pCount, flexPages, reflexPages;
//这样就清楚多了
int flexPages;
int*pCount;
int reflexPages;
```

17.22.2 原因

●不易出错。

●容易阅读。

●便于增减。

●符合"原则17.17 不要在一行中放多于一条语句"。

●C++允许变量定义出现在一个代码块(block)的任意位置，但为了方便阅读和维护，建议还是集中放在代码块的开头。

原则17.23 定义指针和引用时*和&紧跟类型

17.23.1 例子

```
int* pCount;
int& count;
```

17.23.2 原因

和类型放在一起更好理解(习惯后会把它当成一个新的类型)。

原则17.24 按编译器解析顺序放置变量声明的修饰符

17.24.1 例子

```
//两句话意思完全一样，但后者更清楚
const char* pName;
char const* pName;
```

17.24.2 原因

●编译器在解析变量定义时，是以变量名为核心向外扫描的，编程者也要习惯这种方式。如上例所示，pName首先是一个指针；其次，它要指向一个常量;第

三，这个常量是字符类型的。仔细品味这些修饰，你会发现它们是有优先顺序的，优先级越高，越要靠近变量名。(还可参考"原则8.1　关于常量修饰符的含义"。)

●保持和编译器一致不易出现理解上的偏差，特别是在定义一个复杂的变量时。

原则17.25　关于函数声明和定义的格式

17.25.1　说明

函数声明原则上放在一行。但函数定义(实现)应按下列顺序放在多行中：

●模板描述。

●修饰符inline和返回值类型。

●函数名及其参数，若需要，参数可放在多行中。

●函数体。

17.25.2　例子

```
//函数声明
int function(void);
//函数定义(实现)
template<class KEY_T, class VALUE_T>
inline void
HA_CheckPointTable_T<KEY_T, VALUE_T>::myFunction(
  const char* pStoreFileName,
  size_t tableSize,
  size_t syncInterval,
  float performanceCoeff
)
{
//...
}
```

17.25.3　原因

因为按逻辑层次进行了划分，所以容易理解，特别是当函数名前的修饰很复杂时。

原则17.26　函数名和左括号间不要空格

17.26.1　例子

```
int function(int argument);
```

17.26.2　原因

●清晰。

●符合出版以及业界的惯例。

原则17.27 声明函数时给出参数的名字,除非没有用处

17.27.1 例子

```
//argument就是参数的名字
void function(int argument);
//后缀++的参数只用来区分前后缀，所以不需要名字
MyClass_T const MyClass_T::operator++(int);
template<class ELEMENT_T>
class Z_Array_T
{
  public:
  //第一个参数需要名字index，因为类型size_t不能说清问题
  //第二个参数可以不要名字，因为已经足够清晰了
  virtual void read(size_t index, ELEMENT_T&);
  //...
};
```

17.27.2 原因

●一切从阅读者容易理解的角度出发：

如果给出名字更清楚，则给出名字；如果不给名字更清楚，就不给名字：如果拿不准(界限不明显)，就给出名字(更安全);如果不给名字会使编译器给出警告，就给出名字。

●当给出参数的名字时，注意声明和实现的名字要一样。

原则17.28 关于类内不同级别的元素排列顺序

17.28.1 顺序

在类定义的.hpp文件中各元素的排列顺序，按类的内外(使用)次序由外而内排列，依次为：

●类的公共(public)部分：

所有对类的访问都应通过它们完成，它们在类的最外层；公共部分按类内定义的类型(其他类、枚举等)、公共成员函数、公共成员的顺序排列。

●被禁止使用函数部分：

见"原则4.14 显式禁止编译器自动生成不需要的函数"，这是因为它们是用来告知外界该类不能做的事情，因而也属于类的接口(而不是实现)部分。

●注释横线：

在.hpp文件中，类的接口部分和其他部分要用横线(见下面例子)分开，使得阅读者马上知道只有横线上方的部分才是他关心的，其余都是使用时不可见也不关心的(注意，不要在保护部分和私有部分再加注释横线，否则显得乱。而且当

一个类缺少公共、受保护或私有部分时，注释横线的含义容易混淆)。

●类的保护(protected)部分：

派生类对基类的访问可以通过它们完成，它们属于类的中间层。

●类的私有(private)成员函数部分：

它们只能被本类内部使用，属于类的核心部分。

●类的私有成员：

它们属于类核心中最基本的实现细节；相比之下，私有成员函数还有一点类内接口的味道。

●最后是类的友元声明：

友元声明不属于实现，因为它没有为所在类的实现做任何贡献；它倒像是接口，但这个接口没有开在类的最外层，而是开在最核心(核心后门，所以放在最后)。

另外提一句，友元声明不受public、private等访问控制权的限制，所以放在哪种权限下或者干脆不显式给出访问权限都行。

17.28.2 例子

```
class MyClass_T
{
  //类的公共(public)部分
  public:
  //接口函数、常量成员、枚举等
  //被禁止使用函数部分
  private:
  //拷贝构造函数、赋值函数等
  // ---------------------------------------------
  //类的保护(protected)部分
  protected:
  //派生类可以使用的接口函数等
  //类的私有(private)成员函数部分
  private:
  //只能本类内部使用的函数
  //类的私有成员
  private:
  //变量、常量等
  //类的友元声明
  friend MyFriendClass_T;
};
#include"MyClass_I.hpp"  //包含inline函数实现
```

原则17.29 关于类成员函数的排列顺序

17.29.1 说明

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。

17.29.2 原因

符合一般阅读者的思维方式。

原则17.30 类成员变量按字母顺序排列

原因

●方便插入、删除和查找。

●但要将静态和非静态分开。实际上，最好把所有静态成员(包括函数)放在一起，因为它们是有关类级的实现，而普通成员是有关对象级的实现，完全不同。

原则17.31 关于静态成员的访问

17.31.1 说明

用操作符::，避免用.或->。

17.31.2 例子

```
Class MyClass_T
{
  //...
  static int count(void)const;
};
MyClass_T* pMyObj=new MyClass_T;
//不要用这种方式访问静态成员
i=pMyObj->count();
//这种方式较好
i=MyClass_T::count();
```

17.31.3 原因

因为静态成员是类级的，用对象去访问会误导阅读者。

原则17.32 关于字符常量

17.32.1 例子

```
char c='x';     //不要用0x78、0170或120
if(c == '\b')     //用定义好的标识符号，而不是直接用值
```

17.32.2 原因

更清晰易懂。

原则17.33 用带颜色的编辑器

17.33.1 说明

诸如微软的VisualC++、UNIX中的Emacs等代码编辑器(不是编译器)，都能自动按不同颜色显式不同的语法元素，比如用蓝色表示保留字、黄色表示自定义标识符、红色表示语法有问题等。

17.33.2 原因

●颜色表示法实际上是一种(初级的)语法分析，它可以帮助我们在程序录入时发现一些问题(主要是笔误)，此时发现问题也符合"问题发现越早修改代价越低"的原理。

●有颜色更清晰整洁。

●阅读者用带颜色的编辑器也能帮助理解。

第 18 章　注释

注释应当是编码的一部分。换句话说，没有注释，编码不能算完。写出好的注释如同写出好的代码一样，需要很高的水平。注释要清晰、简洁，并且有价值。

原则18.1　用英语写全部的注释

原因

英语是所有程序员中最通用的语言，不论国籍。

原则18.2　确保注释完善你的代码，而不是重复你的代码

18.2.1　说明

注释不应包含代码本身显而易见的信息，而应给出其他有用的信息。这些内容应是重要的，否则不如不加。

18.2.2　例子

```
//这种注释没有用处
index++;     //Increment index
```

18.2.3　原因

没有额外价值的注释只能增加阅读者和维护者的负担。

原则18.3　注释用词要精确，不能有二义性

注意

人与人交流最大的问题是误解：你以为你说清楚了，或他们以为他们看懂了你的注释，其实没有。所以注释要像代码一样经过审查(inspection)。

原则18.4　注释中的术语要通用

原因

不通用的术语会造成理解上的困难。

原则18.5　注释要简单、清楚、切中要害

例子

●代码的修改历史不应出现在该代码的注释中，可将其放在文件结尾的总体说明(注释)中。

●代码的作者、修改时间以及相关的记录号(如PR/CR/DDTS号)等不应出现在该代码的注释中。

原则18.6　注释不能超出被注释代码所包含的内容

18.6.1　例1，没必要说二叉树，此处代码不包含实现细节：

```
//Search the set for the item using a binary-tree search algorithm
bool contains(Item_T anltem);
```

18.6.2　例2，这样说是恰当的：

```
//Return true if the specified item is contained in the set
bool contains(Item_T anltem);
```

18.6.3　原因

●增加阅读负担；

当前代码不包含的信息不是阅读该代码时应该获得的，如果给出会增加阅读的负担，干扰对该代码本身功能的理解。

●削弱C++的封装性：

让使用者知道实现方法，有可能诱使他编写出和实现方法配套的代码(比如进行相应的优化或做出依赖于该实现方法的某种假设)，这不利于使用和实现(代码)的互相独立、互不干扰。

●增加维护负担：

额外信息还需和真正对应的(实现)代码保持一致，这也是维护的负担：并且通常不容易保持一致(因为它们不在一处)，从而误导阅读者。

原则18.7　注释中避免引用容易变化的信息

18.7.1　例1，引用了容易变化的值12：

```
//Don't allow more than 12 attempts
if(attempts>12)
```

18.7.2　例2，这样就好多了：

```
//Restrict the number of attempts
if (attempts>12)
```

18.7.3　原因

●防止代码变更后忘记修改注释。

●好的注释会尽量减少维护的代价。

原则18.8　确保所有注释(随代码)及时更新

原因

●注释是编码的一部分，所以修改代码时，相应的注释也要改。

●没有及时更新的注释会误导阅读者。

●不能保证及时更新的注释不如没有注释。

原则18.9　注释不具备约束使用者行为的能力

18.9.1例子

```
class MyClass_T
{
    public:
    //...
    int init(void);    //Must call this function before using this class
//...
}
/*---------=-------------------------------------------------------
*上面的注释没有强制力(基本上没用)，用构造函数代替init()函数才是正路
-----------------------------------------------------------------------*/
```

18.9.2 原因

●注释的作用在于说明情况，没有强制力，所以不能希望通过注释约束使用者的行为。

●如果希望约束使用者，一定要通过代码本身来实现。

原则18.10 注释不要嵌套

18.10.1 例子

```
/*
 if(verbosity>0)
 {
    /*这是嵌套注释*/
    cerr << "Opening file<" << filename << ">" << endl ;
 }
*/
```

18.10.2 原因

不同编译器对嵌套注释的处理不同，有些不支持，从而导致错误。

原则18.11 不要用/* */注释掉(大块)代码，应该用#if 0

18.11.1 例子

```
//temporarily disabled section of code
#if 0
if(debugLevel>0)
{
    //...
}
#endif//#if 0
```

18.11.2 原因

●嵌套问题：

用/**/做大段的注释要防止被注释掉的代码中有嵌套的/**/，这会导致注释掉的代码　区域不是你想要的范围，当被注释掉的代码很大时容易出现这种情况，特别是过一段时间后又修改该处代码时更是如此。

●#if 0方式不允许嵌套。

●#if 0是注释掉，#if1是打开，非常方便。

原则18.12　区分段落注释和单行注释

18.12.1　说明

注释分段落和单行两大类：

●段落注释用来说明一段代码，因而要放在该段代码之前。通常，此类注释较长(超过一行)，建议用/**/风格。

●单行注释用来说明一句代码，通常放在该代码之后(行末注释)。建议用//风格。

●段落注释用横线上下框住(见例子或附录的大段代码)，目的是一目了然地将代码和注释区分开来，便于阅读。(单行注释这样做显得累赘，请读者自己权衡。)

18.12.2　段落注释的例子(格式)

```
/*-------------------------------------
*This is"strategic"comment
*used for block comments.
-------------------------------------*/
```

原则18.13　行末注释尽量对齐

18.13.1　说明

一般从第46个字符位置开始：不同文件中可以不同，但同一个文件中最好保持一致。

18.13.2　原因

●看上去整齐。

●注释和被注释语句保持足够的距离以便阅读时能轻易将其分开。

原则18.14　单独的注释行和被注释语句缩进相同的空格

原因

习惯上认为这样更清晰。

原则18.15　减少不必要的单独占一行的注释

原因

●单独占一行的注释有打断代码(阅读连贯性)的嫌疑，所以如果能用行末注

释，就不要用单独占一行的注释。

●类定义和函数前的注释(段落注释)不会打断代码，因为它们在一个完整的逻辑段之外。由此也可以揣摩该原则进退的分寸。

原则18.16　对每个#else或#endif给出行未注释

18.16.1　例子

```
#if(defined(Z_EXECUTE_WHEN_DEBUG))
    //...
#else //not defined Z_EXECUTE_WHEN_DEBUG
    //...
#endif //#if(defined(Z_EXECUTE_WHEN_DEBUG))
```

18.16.2　原因

●方便阅读者阅读。

●也提醒自己别配对配错了。

●大块的条件语句也应做类似处理。

原则18.17　对每个引用的头文件给出行末注释

18.17.1　例子

```
#include<iostream.h>   //use cout, etc.
#include<pthread.h>    //use POSIX thread library
#include"ZDebug.hpp"  //use macro Z_DEBUG()
#include"ZErrLog.hpp" //use macro Z_LOG_…()
#include"ZMutex.hpp"  //use class Z_Mutex_T
```

18.17.2　原因

●方便阅读者理解引用该头文件的目的。

●方便今后代码维护时增减这些头文件，从而保证头文件的自足性，并减少不必要的文件依赖。(请参见"原则20.2　确保公共头文件的自足性"和"原则20.3　只引用需要的头文件"。)

原则18.18　对每个空循环体给出确认性注释

18.18.1　例子

```
while(str[ctr++] != EOS)
{
  //This is an empty body
}
```

18.18.2　原因

●提示自己和别人，这是空循环体，不是忘了。

●防止阅读者误将其后紧跟的语句当成循环体。

原则18.19 关于函数注释

18.19.1 说明

●关于使用(某函数)方法的注释：

(头文件中)函数声明之前，要给出恰当的注释，从而让使用者无需了解实现细节就能够快速获得足够的信息来使用该函数：同时这些信息又是精炼的，没有不相关的内容。

●关于(某函数)实现的注释：

在函数实现之前，要给出和实现有关的足够而精炼的注释信息。

18.19.2 例子

```
//在ZFileArray，hpp头文件中的使用注释
template<class ELEMENT_T>
class Z_Array_T
{
  public:
    //read from Z_Array_T[index]
    virtual void read(size_t index, ELEMENT_T&);
    //...
};
//在ZFileArray.cpp文件中的实现注释
//*********************************************************************
//Copy Read()    virtual public
//*********************************************************************
//Description:  Return element in file array.
//Parameters:  index:array index;
//    anElement:save return value in it.
//Return:    None.
//Exception:    Index out of bound:process abort.
//MT Safe:    False
//Note:    In header file because of template.

//*********************************************************************
template<class ELEMENT_T>
void
Z_FileArray_T<ELEMENT_T>::read(size_t index, ELEMENT_T& anElement)
{
```

```
    //...
}
```

原则18.20　关于注释频率

说明

通常的规则是:大约每5行代码应至少有一行注释，否则应视为注释过少；但没听说过有关注释的上限的定量标准。

第19章　文件和目录

源代码是保存在头文件(header file)和源文件(source file)中的。如果逻辑结构组织得好,封装和模块化恰当，并且仔细地减少命名空间的冲突，可使你的代码易读，并增加被引用的几率。

原则19.1　使用统一而且通用的文件名后缀

19.1.1　说明

●文件后缀为.hpp：C++的头文件。

●文件后缀为_I.hpp：C++的inline函数实现(头)文件。

●文件后缀为.cpp：C++的源文件。

●文件后缀为.h: C的头文件(几乎成为标准)。

●文件后缀为.c：C的源文件(几乎成为标准)。

19.1.2　例子

#include"MyClass_I.hpp"　　//inline函数实现所在的头文件，比MyClassI.hpp清晰一点

19.1.3原因

●为何要统一：

使用统一而且通用的文件名后缀可增加可读性和移植性，还能简化诸如make等工具的依赖规则。

●为何选择hpp和cpp：

hpp和cpp是微软的VC编译器支持的、且比较通用的C++文件后缀，同时UNIX编译器也能接受，所以若想让代码能跨尽可能多的平台(编译器)，这是一个好的方案。

●为何不用.h:

头文件是包含在源文件中编译的，所以用.h做C++头文件的后缀也不会出错，但.hpp能直观地区分是C还是C++的头文件，所以相对还是清楚一些。

●为何用_I.hpp而不是.i或I.hpp：

用_I.hpp做C++中inline函数实现的头文件后缀比用.i要好，因为大多数编译器不认识.i这种后缀；另外有的编译器(如SUN的Forte编译器)用.i做预编译结果文件的后缀名，所以可能有冲突。

_I.hpp比没有下划线的I.hpp要清晰一点(见例子)。

●为何都是小写：

有的操作系统不区分大小写，有的区分。为了尽量提高可移植性，建议在命名和引用时按区分大小写方式处理，这样在不区分大小写的环境也能用。后缀用小写是常见用法，统一(而不是大小写随便)后也能帮助简化理解和使用。

原则19.2 关于文件名的选择

要求

●清晰地表达其内容：

比如可以用类名做文件名。

●防止名字冲突：

为了提高可复用性，名字要尽可能独特，要考虑当移入一个新环境时名字不会(不容易)冲突；用目录也可以帮助防止名字冲突，比如将某个类库源代码放在一个以类库命名的子目录下。

●要精炼：

文件名经常要被用到(编辑、编译、拷贝等)，如果名字很长，用起来不方便；

可以适当用缩写，但要以服从前两条要求(清晰、防止冲突)为前提。

●公共前缀：

有时候可用公共前缀对文件分组，从而降低名字冲突的可能性，并能帮助查找相关文件，但要防止文件名过长和冗余信息过多。

原则19.3 关于文件/目录名的字符集选择

19.3.1 说明

文件或目录名可用的字符集是：[A-Za-z0-9._-]。

19.3.2 原因

●这是POSIX指定的字符集。

●如果用到其他字符，不能保证在所有系统上都没问题(比如WINDOWS系统中文件名可以用"&"，但在UNIX系统中"&"有特殊含义，因而可能遇到麻烦)。

原则19.4 关于每个类的文件组成

说明

●类的接口：

必须有一个.hpp(不是_I.hpp，以下同)头文件，用来声明类的结构和接口。

所有有关类使用的注释应全部在该头文件中。

该头文件尽量不包含任何实现细节(除非是C++语法导致的，比如类成员应是实现细节，但不得不放在该头文件中)。

●类的实现：

至少有一个_I.hpp或.cpp(可以两个都有)文件，用来提供类(函数等)的实现。

●inline函数的实现：

在.hpp文件的最后包含(#include)_I.hpp文件(如果有的话)。

●在_I.hpp和.cpp文件中的排列顺序：

函数实现按其在类声明中的顺序排列。

●不要在一个文件中定义一个以上的类。

●文件名用精简的类名：

去掉类名的后缀(_T等)：如果清晰的话，类名中的单词还可以缩写；最后加上.hpp、_I.hpp或.cpp作为后缀。

原则19.5 关于模板类的文件安排

说明

模板类的非inline函数，它们的实现(函数体)表面上看应该放在相应的.cpp中，但是由于有些编译器不支持这种做法，为了最大限度地提高可移植性，目前还是放在_I.hpp中较好。

其实，由于模板是源代码级的复用(见"第13章 模板")，所有代码，包括函数实现(源码而不是目标码)都要在模板实例化时被引用，不论它们在头文件中还是源文件中。所以就算你把实现放在.cpp中，实例化时，编译器还是要你提供.cpp文件，这就如同模板的所有代码都在头文件中一样。所以放在_I.hpp中还是合理的(除非以后出现了相应的国际标准)。

原则19.6 保持文件前言的简洁性

19.6.1 说明

开门见山。给出版权说明和该文件的简介后即可切入正题——编码(文件的前言是指代码之前的所有东西)。

19.6.2 例子

```
//Copyright(c)1999-2000 Motorola, lnc.All right reserved.
//
//"filename":
//A brief description of the purpose of the collection of items in the
//file.
```

19.6.3 原因

●先入为主：

所以越重要的越要往前放。代码本身无疑非常重要，能比它还重要的东西应该不多。

●其他重要信息：

很多注释(比如限制条件、使用说明、多线程安全等)也很重要，但放在代码后面足矣。要知道，随着代码存在时间的延续，需要加的注释会越来越多。都放在代码之前会直线增加阅读者的心理负担：有这么多比代码还重要的信息要先看，到什么时候才能看到代码呀！

●版权问题：

为确保代码受到合法的保护，@或"Copyright(c)"是必要的。@更好，但微软的VC集成环境不支持，为了通用，只好选择"Copyright(c)"。另外，All right reserved也最好加上。

原则19.7 关于文件的段落安排

19.7.1 说明

我们建议这样的顺序：

●前言。

●防止重复引用设置：

只用在头文件中，见"原则20.1 头文件多次引用的防范"。

●#include部分。

●#define部分。

●常量声明。

●类型声明和定义：

包括struct、union、class、typedef等。

●全局变量声明。

●文件级变量定义。

●文件级函数声明。

●函数实现：

按声明顺序。另外，inline函数应放在单独的头文件中，并在类声明头文件的最后#include进来，见"原则19.4 关于每个类的文件组成"。

●结束注释：

包括(按需选择)：基类名称、兼容性、属于哪种通用的设计模式、多线程安全与否、能否处理异常(exception)、使用需要注意的问题(如限制条件)、实现的一些关键想法(修改时需要注意)、重要文档名称、修改历史等。

类的结束注释一般只需集中放在其头文件.hpp中，而在_I.hpp和.cpp中就不必了。

19.7.2原因

●坚持在所有项目的所有代码中都用这样的顺序，可以减少适应新项目的时间、提高项目成员间的沟通。

●防范循环依赖关系的产生。

原则19.8 关于目录组织

说明

●目录：

一般一个软件包或一个逻辑组件的所有头文件和源文件应放在一个单独的目录下，这样有利于查找相关文件，简化make等编译工具的设置。

●公共头文件：

整个项目需要的公共头文件(自己实现的代码，不是系统头文件)应放在一个单独的目录下(比如在myProject/include下)，这些头文件只是一些连接(file link)，它们指向各个逻辑组件目录下真正的文件。这样做既保证了文件的一致性(没有同一个文件有两份以上拷贝的情况)，又避免了别人引用时来源目录太分散的问题。

第 20 章　头文件

头文件的主要功能是为多个源文件提供信息，这些信息通常定义了相应代码的接口部分。实际上，头文件是代码中唯一需要最终用户了解的部分。所以，保证头文件的良好设计、易于理解和易于使用显得尤为重要。

原则20.1　头文件多次引用的防范

20.1.1　说明

由于头文件可被其他头文件引用，所以在某个文件中同一个头文件可能被引用两次或更多次。比如A.hpp引用B.hpp和C.hpp，而B.hpp和C.hpp又分别引用D.hpp，因此A.hpp中实际上引用D.hpp两次，这会造成重复定义之类的编译错误。

由于头文件的嵌套引用没有层数限制，所以要追踪某个文件的所有引用、以检查是否有重复引用基本上不可能：一是路径太多太复杂，二是任何头文件引用的增减都应重新扫描所有文件以防修改造成新的重复引用。

使用条件编译技术可以防止多次引用引起的问题，而且简单易行。请见例子。

20.1.2　例子

```
#ifndef MY_CLASS_HPP
#define MY_CLASS_HPP
//真正的头文件内容
#endif//ifndef MY_CLASS HPP
```

20.1.3　原因

●如果在某个文件中多次引用出现，条件编译判断就会为假，编译器就会将被该条件编译包裹的代码去掉，从而防止了同一代码出现多次。

●这个办法只需增加三条语句，实现简单，代价不高。

●所有追踪、扫描以及程序变动后的重新追踪、扫描都交给编译器来完成，保证安全和高效。

20.1.4　此类宏的命名方式

●基本上可以和该头文件的文件名相同。

●此类宏如果重名，会造成严重后果，所以尽量给出各种信息以保证唯一性，比如加前后缀等。

●头文件.hpp的宏用_HPP做后缀;_I.hpp的宏用_INLINE做后缀。

●此类宏的前缀请参见"原则1.7　关于全局命名空间级标识符的前缀"。

●由于此类宏只有在这三句代码中用到，长一点也无妨，所以不要用缩写等。

原则20.2　确保公共头文件的自足性

20.2.1　说明

自足性是指该头文件包含全部自己所需的其他头文件，因此使用者无需再引用其他文件就能使用该公共头文件所提供的所有功能。注意，只需引用直接需要的头文件，间接用到的头文件应由其直接使用者引用。比如A.hpp直接用到B.hpp定义的某些功能，B.hpp直接用到C.hpp的某些功能，但A.hpp没有直接用到C.hpp的功能，则A.hpp只需引用B.hpp，而C.hpp应由B.hpp引用。

20.2.2 原因

●公共头文件自己(而不是使用者)应该关心自己还需要什么。

●虽然这会加重文件间的相互依赖关系，从而导致一个文件的修改造成大面积的文件需要重新编泽，但权衡易用性和编译时间，还是前者更重要。

●为了减少这种类型依赖造成的重新编译，让编译器的依赖关系分析工具更智能化可能才是正确的解决思路(比如，如果用这种工具分析能精确得知某个头文件的某处改动最终会影响哪个文件的哪个部分，则会大大减少不必要的重新编译)。

原则20.3 只引用需要的头文件

原因

●否则会增加维护者理解的难度。

●增加不必要的依赖关系，导致一处修改引起不相关的文件被迫重新编译，浪费资源。

原则20.4 引用时""和<>的用法

20.4.1 说明

<>只用在引用系统或语言本身提供的头文件，其他情况一律用""。

20.4.2例子

```
#include <iostream.h>     //标准头文件
#include"MyClass.hpp"     //自己的头文件
```

20.4.3原因

●使得自定义和非自定义头文件一目了然。

●这是被广泛接受的用法。

原则20.5 引用头文件的顺序

20.5.1 顺序

●系统头文件(用<>扩起来的)；并按文件名字母顺序排列。

●自定义头文件(用""扩起来的)；并按文件名字母顺序排列。

20.5.2例子

```
#include <iostream.h>     //use cout,etc.
#include <pthread.h>     //use POSIX thread library
```

```
#include"ZDebug.hpp"     //use macro Z_DEBUG()
#include"ZErrLog.hpp"     //use macro Z_ERROR_LOG()
#include"ZMutex.hpp"     //use class Z_Mutex_T
```

20.5.3 原因

●系统头文件提供基础服务，习惯上放在前面。

●字母顺序可以方便查找，特别是在头文件很多时更是有用。

原则20.6 引用时不要用绝对路径

20.6.1 例子

```
/*--------------------------------------------------
*使用绝对路径，如果需要移动目录，必须修改所有
*相关代码，繁琐且不安全
--------------------------------------------------*/
#include "/usr/project/src/MyClass.hpp"
/*--------------------------------------------------
*用相对路径，当需要移动目录时，只需修改编译器
*的某个选项(-I选项)即可
--------------------------------------------------*/
#include"MyClass.hpp"
```

20.6.2 原因

当需要改动目录结构或目录名字时，修改编译器的一个-I选项当然要比修改文件中所有引用该路径的语句来得轻松。

原则20.7 将函数库放在一个单独的目录下引用

20.7.1 例子

```
#include"Z_Library/ZErrLog.hpp"
```

20.7.2 原因

●减少名字冲突。

●更清晰。

●有助于查找相关文件。

原则20.8 不要在头文件中定义常量/变量

原因

头文件中定义常量或变量，会使得所有引用该头文件的源文件都产生一个变量或常量。这通常不是编程者的初衷。实际上，头文件是用来声明而不是定义的。

原则20.9 任何声明若被多个源文件引用则应在一个头文件中

原因

这样，如果修改该声明，只需修改相应的头文件即可，而不必修改多个源文件。

原则20.10 在源文件中不要用关键字extern

原因

关键字extern表明这是一个声明而不是一个定义。所有声明都应该在头文件中，就像所有实现都应在源文件中一样。

第 21 章　条件编译

条件编译允许在编译时裁剪代码，从而生成不同的执行程序。这种裁剪不需要代码物理上的改变，而是通过设置不同的编译条件实现的。这是一个非常有用的功能，特别是处理某些对症的问题时更是如此。但是，不能滥用，因为条件编译会影响代码的可读性和可维护性。

原则21.1　最小化条件编译的使用范围

21.1.1　原因

●条件编泽会显著影响代码的可读性：

因为它把代码分割成不连续的部分。

●提高维护难度：

维护逻辑上不连续的代码是困难的，并且容易因为遗漏(某块条件编译代码)而出错。

●测试困难：

编译条件的所有组合都要通过编译和测试。

●最小化条件编译的使用范围：

所以，当确实需要使用条件编译时，也要将其限制在尽可能少的几个文件中，使得阅读和维护都只需在一个小范围内关注条件编译。

21.1.2　允许使用条件编译的一种情况：包裹调试代码

条件编译最常见的用法是包裹调试代码。这样做可以在调试时打开编译条件，增加额外的代码帮助调试。而在真正运行时关闭编译条件，使代码能全速运行(不因为源程序中存在调试代码而损失任何效率)。从这一点来说，条件编译好于用普通的if语句包裹调试代码，因为后者的调试代码在任何时候都存在于执行程序中，导致程序尺寸增大，效率下降。但另一方面，用普通的if语句可以实现运行时打开调试的功能，这是条件编译做不到的地方。

那么何时用条件编译，何时用普通的if浯句包裹调试代码呢?总的来说，两种方法要结合使用：

●从开发阶段角度看：

条件编译包裹的调试代码用于开发者自测阶段；而用if语句包裹的调试代码用于系统测试以及投入运行后对发生的问题的调试。

●从发生问题的性质看：

由于机器本身或严重的编码错误造成的问题应该用条件编译方式，比如内存资源耗尽的检查、(大部分)系统调用失败的检查、程序关键配置错误的检查等；而程序运行的位置、相应的中间信息应用if语句方式。

●从使用的限制条件看：

由于用if语句方式是在运行期间进行跟踪调试，它必须尽可能减少对同时进行的正常业务流程的打扰，所以此类代码要少占系统资源(包括CPU、内存、I/O等)，慎重做影响正常业务的动作(比如修改运行状态等)；而条件编译方式的限制条件要宽松的多。

●有时无法区分哪种方式更好：

这就要具体问题具体分析，一条参考原则是：用if语句方式做错误的粗定位(和中级定位)，用条件编译方式做精确定位。

21.1.3　允许使用条件编译的一种情况：包裹与特定机器/实现相关的代码

有时候条件编译也用来包裹与特定机器/实现相关的代码。比如，如果是Windows平台，则编译某块代码，如果是Solaris平台则编译另一块代码等。

但这种方法逐渐被面向对象的抽象工厂(abstract　factory)模式替代。比如，用如图21-1所示的类的继承关系来实现互斥锁的不同解决方案：

所有需要互斥锁的代码只使用父类(Abstract)指针或引用(做加锁、解锁等操作)，只是在初始化这些父类指针或引用时用某个特定的派生类对象。比如，当需要用POSIX型的互斥锁时，将“POSIX Mutex”类型的对象赋给父类指针或引用；当运行在单线程环境时，将“Empty Mutex”类型的对象赋给父类指针或引用，等等。这样，使用者只需面对同一接口：“Abstract”类，因而相应的代码无需依赖于互斥锁的不同解决方案(低耦合)。另一方面，不同解决方案只会引用(#include)不同的头文件(“POSIX Mutex”.hpp 或“Empty Mutex”.hpp)来做初始化，不相关或不兼容的代码根本不参与编译。

这种方法之所以好于条件编译，是因为条件编译要在所有实现代码中区分各种情况，比如加锁函数中要包含“POSIX Mutex”的实现方法(用一个条件编译包裹)、“Empty　Mutex”的实现方法(用另外一个条件编译包裹)，以及所有需要支持的解决方案；解锁也一样。可见，每个函数都非常冗长，并且阅读者不能完整地看到一套特定实现(包括加锁、解锁、初始化、清除等)的所有代码(它们都分散在各个函数内)，理解起来以及今后的维护都要困难一些。

原则21.2　若使用#if或#ifdef，不要遗漏#else

21.2.1例子

```
#if defined(Z_POSIX_MUTEX)
//...
#elif defined(Z_EMPTY_MUTEX)
//...
#else
#error Hust select a kind of mutex    //确保指定一种类型的互斥锁
#endif
```

21.2.2原因

●确保条件编译的完整性。

●使得编译条件选择错误出现在编译时而不是运行时。

原则21.3 编译条件的含义要具体

21.3.1 例子

```
#if defined(LINUX)              //不好，含义不清
#if defined(HAS_SYSV_IPC)       //好，含义足够清晰
```

21.3.2 原因

●含义不清会导致理解、维护和代码重用的困难。

●这是一条很通用的原则，注意举一反三。

原则21.4 对复杂的编译条件用括号使其清晰

21.4.1 例子

```
//复杂的编译条件用括号使其清晰
#if( ((  STDC_-0)==0) && !defined(_NO_LONCLONC) )
//简单的编译条件不需要括号已足够清晰了
#if 0
```

21.4.2 原因

●消除二义性和潜在错误。

●理解更容易。

原则21.5 条件编译和普通条件语句不要混合使用

21.5.1 例子

```
#define DEBUG_LEVEL 4
//不好，条件编译和普通条件语句混用
if((status!=OK)&&(DEBUG_LEVEL>0))
{
    //这里的语句可能不参加编译，如果DEBUG_LEVEL未定义的话
}
/*-------------------------------------------------------------
*如果DEBUG_LEVEL未定义，包括if语句本身的条件语句块都不会参加编译
-------------------------------------------------------------*/
```

第 22 章　编译

编译器是检测程序问题的最强大的工具之一。它对于发现编程初期大量的语法错误和部分语义错误特别有效。由于编译是在运行之前，此时发现的问题解决起来代价相对要小。要知道，问题发现的越晚，解决的代价越大。

原则22.1　关注编译时的警告(warning)错误

原因

●警告是编译器认为可能会带来潜在问题的地方，所以关注警告所描述的信息，会及早发现问题(隐患)，及早消除。

●一个好的程序，应该不包含任何警告错误。

●一个好的编译习惯是将编译器的所有警告开关都打开。

原则22.2　把问题尽量暴露在编译时而不是运行时

22.2.1　原因

问题发现的越晚，解决的代价越大。

22.2.2例子

有很多编程手段和方法可以让编译器帮助查找问题，从而避免将这些问题遗留到运行时：

●防止函数返回非法状态，请见“原则3.13　若函数返回状态，尝试用枚举作类型”。

●防止类拷贝或赋值引起问题，请见“原则4.14　显式禁止编译器自动生成不需要的函数”。

●防止函数内部对其参数进行修改，请见“原则8.2　在设计函数原型时，对那些不可能被修改的参数用常量修饰”。

●防止隐式类型转换，请见“原则11.6　用关键字explicit防止单参数构造函数的类型转换功能”。

●防止头文件的重复引用，请见“原则20.1　头文件多次引用的防范”。

原则22.3　减少文件的依赖程度

22.3.1　说明

文件的依赖程度高是指文件间的相互引用(#include)过多，其中很多可能是不必要的。这会造成某个头文件的修改会导致一大片相关文件的重新编译，造成时间和机器资源的浪费。

但要注意，本原则的优先级较低，执行时一定不能伤害诸如“原则20.2　确保公共头文件的自足性”等原则。

22.3.2 例子

```
class AnotherClass_T;
class MyClass_T
{
    private:
     /*---------------------------------------------------------------
     *如果MyClass_T只用到AnotherClass_T的指针或引用，不必用#include
     *形式引用AnotherClass_T所在的头文件，用"class AnotherClass_T; "
     *的声明方式即可。这会减少文件依赖.但要注意，MyClass_T的实现文
     *件(_I.hpp或.cpp)中还是需要引用AnotherClass_T的头文件
     ---------------------------------------------------------------*/
     AnotherClass_T& i_object;
     //...
 };
```

原则22.4 减少编译时间

22.4.1 说明

程序开发就是一个修改、编译、再修改、再编译不断循环的过程，减少编译时间可以提高工作效率。在大型软件开发中，源文件有许多不同的版本、一个执行程序可能由几十万甚至上百万行程序编译而成(比如微软的winword.exe是一个8MB多的执行文件，比较一下自己手边的程序，你就可以倒推出它大致有多少行代码)，因而编译时间可能很长,减少编译时间就显得尤为重要。

22.4.2 减少编译时间的一些建议

●编译优化选项的开关：

编译优化会显著增加编译时间，所以在代码调试阶段不要打开，因为此时的目的是查错、改错，基本上不关心性能。但在性能测试和系统测试阶段一定要打开优化，因为此时要测试的程序应该和正式发布版完全一样(正式版应该打开所有优化选项以取得最佳的性能效果)，如果还拿测试版，有些问题可能会被隐藏起来(比如"原则6.9 记住给字符串结束符申请空间"中提到的数组越界问题)。

●优化make规则以减少编译时间：

诸如make这样的编译工具是用来制定文件依赖关系的，即某个文件被修改后，什么文件需要重新编译。若这个工具优化的好，会大大减少不必要的重新编译，从而节省编译时间。

●在生成库中预先实例化模板，例如：

```
template<class TYPE_T>
class Z_Array_T
{
```

```
    //...
};
/*--------------------------------------------------------------
*在库文件中这样声明会实例化模板，参加编译并放
*在库中，以后用到这个库时，Z_Array_T<int>将不
*再需要重新编译，从而节省编译时间
--------------------------------------------------------------*/
template class Z_Array_T<int>;
```

●可将不需要模板的代码从模板中提出来，例如：

```
template<class TYPE_T>
class Z_Array_T
{
   public:
     size_t arraySize(void)const;
   private:
     size_t i_arraySize;
     //...
};
template<class TYPE_T>
Z_Array_T<TYPE_T>::arraySize(void)const
{
    return i_arraySize;
}
/*--------------------------------------------------------
*函数arraySize()若放在模板类中，则每次实例化模板时该函数都要
*被重复生成，而其实每份代码都是一样的，所以如果提出来放在非模
*板方式的基类中可以节省每次重新生成、重新编译的时间。另外，不
*是所有的c++编译器都能在优化时识别出此类代码并做相应的优化，
*为了兼容性，需要编程者手工做这件提取工作。
--------------------------------------------------------*/
```

原则22.5 透彻研究编译器

原因

现代的编译器越来越强大，集成的功能也越来越多(它已不是简单的编译连接工具)，仔细研究编译器提供的功能和各种选项，会帮助我们更多、更快地发现问题，更好地优化代码(包括性能和程序尺寸等)，以及提高对平台的针对性或兼容性等。

第 23 章　兼容性

这里所说的兼容性(高)是指源代码能适应不同的编译器、操作系统、硬件平台等。兼容性高可以增加代码可复用性。通过遵循几条简单的规则，可使兼容性大幅提升。

原则23.1　遵守ANSI C和ISO C++国际标准

原因

●避免使用某个编译器自定义的语法规则。可能很好用，但不利于移植。

●慎用最新的国际标准，因为有可能主流编译器还不支持。

原则23.2　将不符合国际标准的代码与其他代码分开

原因

●分开放置：

如果不得不使用不符合标准的代码，一定要和其他(符合标准的)代码分开(放在不同文件中)，从而使得它们容易被找到，移植时容易被替换掉。

●封装：

最好将不符合国际标准的代码封装起来，以减小它们(变化时)对其他(符合标准的)代码的影响；这也同时提高了其他(符合标准的)代码的可复用性。

●和特定系统密切相关的代码：

更常见的问题是，有时不得不依赖特定硬件或操作系统以实现某些功能。同理于"不符合标准的代码"，它们也要分开放置，并最好封装起来(可参见"21.1.3 允许使用条件编译的一种情况：包裹与特定机器/实现相关的代码"中的有关描述)。

原则23.3　不要假设字符类型是否是有符号类型

原因

有符号字符和无符号字符区别很大，且(隐式)相互转换算法有赖于编译器的实现(没有标准)，所以任何假设可能导致不兼容问题。

原则23.4　运算时显式转换有符号和无符号类型

23.4.1　例子

```
unsigned long possitionA, possitionB;
//...
long distance=long(possitionA) - long(possitionB);
```

23.4.2　原因

●不同的国际标准(Classic C/ANSI C/ISO C++)对隐式转换有符号和无符号类型的规则不同，有可能导致不同的结果。

●使阅读和理解更清晰、受控。

原则23.5 注意双字节字符的兼容性

例子

```
//双字节字符"xy"在不同编译器中的值可能不同
Uintl6 magic='xy';
```

原则23.6 恰当使用位操作符

23.6.1 说明

只有绝对需要时，才使用位操作符，比如操作硬件寄存器时。

23.6.2 原因

●位操作通常依赖于硬件，所以兼容性不好。

●位操作比较晦涩(比如"原则24.2 不要用移位代替乘除运算")。

原则23.7 注意位域(bitfield)变量的兼容性问题

原因

例如，有的编译器视此类变量为有符号的，另一些编译器视其为无符号的。如果你在乎符号，请显式指定。

原则23.8 不要强制引用/指针指向尺寸不同的目标

23.8.1 例子

```
int badFunction(const char* pChar)
{
    //危险
    int* pInt=(const int*)pChar;
    return(*pInt);
}
```

23.8.2 原因

●可能引起(CPU的)越界访问等错误。

●这种用法本身也没有道理。

原则23.9 确保类型转换不会丢失信息

23.9.1 例子

```
//在大多数环境下，函数fork()的返回值大于short的长度，从而导致信息丢失
short pid = fork( );
```

23.9.2 原因

●从大尺寸类型转换成小尺寸类型一定会丢失某些数据，要确保丢失的数据是不需要的。

●大尺寸类型的值对于小尺寸类型来说可能是超界的(溢出)，此时的类型转换涉及溢出的处理，要确保结果如你所料。

原则23.10 不要假设类型的存储尺寸

23.10.1 说明

诸如整型、长整、短整、浮点数、双精浮点数等的内存尺寸依系统不同而不同;而指向不同类型的指针也不能保证尺寸一定一样(比如虚函数的指针)。

23.10.2 例子

```
/* ------------------------------------------------------
*假设整型的内存尺寸为4个字节，并据此申请内存，一旦
*运行平台改变，而新的运行平台的整型尺寸不同(比如64
*位平台应是8个字节而不是4个字节)，则需要直接修改代
*码，兼容性不好
------------------------------------------------------*/
buffer = new char[4*length];
/* ------------------------------------------------------
*让编译器在编译时计算(通过sizeof())要灵活得多，因
*为编译器在计算时会根据不同平台的要求给出相应的值
------------------------------------------------------*/
buffer=new char[sizeof(int)*length];
```

23.10.3 原因

●用sizeof()函数可以消除此类的不兼容问题。

●很多主流编译器都对sizeof()函数做了优化，使其性能(几乎)不逊于直接使用尺寸(值)。

原则23.11 如果一定要规定类型的存储尺寸

23.11.1 说明

如果一定要规定类型的存储尺寸，使用下面例子所示的方法。

23.11.2 例子

```
typedef char Int8;              //8位有符号整数
typedef short Int16;            //16位有符号整数
typedef int Int32;              //32位有符号整数
typedef unsigned char Uint8;    //8位无符号整数
typedef unsigned short Uint16;  //16位无符号整数
```

```
typedef unsigned int Uint32;     //32位无符号整数
```

23.11.3 原因

●明确告知阅读者/使用者代码希望得到的整数位数。

●如果目标环境的类型不符合该长度要求(比如int不是16位而是32位)，只需给出新的类型定义(typedef)即可(比如用long代替int)。

原则23.12 不要假设对象(等数据结构)的存储结构

原因

●不同编译器的对象存储格式会有差别：

对象是由编译器最终决定共存储结构的。你可以大致想象出某个编译器怎样在内存中放置一个对象，但你永远不能精确说出每个编译器放置对象的方法，比如对象中的虚函数指针放在前面还是后面?菱形继承类型对象的数据怎样安排?编译器是否要做边界对齐以优化访问性能等。所以在你的程序中，不要依赖自己的推测写代码，否则兼容性肯定有问题。

●举一反三：

C++中还有不少存储结构是由编译器决定的，比如用new生成的数组，编码时也不能依赖它们的存储结构。

原则23.13 注意运算溢出问题

23.13.1 例子

```
if ((a+b) < 0 )
{
  logError("Overflow.");
  /*-----------------------------------------------
  *a，b永远是非负整数，但用判断a+b是否小于0来判
  *断溢出不保险，因为溢出(a+b的结果超过正整数上限)
  *后的处理方法是没有统一标准的，也就是说，a+b溢出
  *后编译器/CPU可不一定直接将运算结果(负数)返回给
  *调用者(比如采用置溢出标志，并将结果清零)，所以
  *这里用的方法不保险，至少兼容性不好
   ----------------------------------------------- */
}
short abs(short n)
{
  if(n<0)
   {
     n=-n;
```

```
    /*-----------------------------------------------------
    *短整型的范围是-32768到+32767，当n=-32768时，-n就溢出了
    -----------------------------------------------------*/
    }
return n;
}
```

23.13.2 原因

要记住，溢出的处理方法没有标准，完全依赖于编译器(或CPU)，所以要非常小心。

原则23.14 不要假设表达式的运算顺序

23.14.1 例子

```
int x=(
(getchar()<<24)|
(getchar()<<16)|
(getchar()<<8)|
(getchar())
);
```

23.14.2 例子中的疑问

- ●如果输入的是"1234"，你能肯定x是0x31323334还是0x34333231?
- ●阅读者也能肯定吗?
- ●至少理解上不直观。
- ●所有编译器也如你所愿吗?
- ●没有清晰而简单的其他方法吗?

原则23.15 不要假设函数参数的计算顺序

23.15.1 例子

```
int array[]={1, 2, 3, 4};
int* pArray=&array[0];
/*-----------------------------------------------
*不同编泽器，执行结果可能不同，有可能是
    * function(1, 2)，也可能是function(2, 1)
    -----------------------------------------*/
    function(*pArray++, *pArray++);
```

23.15.2 原因

函数参数的计算顺序没有统一标准，即从左到右、从右到左、甚至从中间到两边或随机选择，在理论上都是合法的。

原则23.16 不要假设不同源文件中静态或全局变量的初始化顺序

原因

●因为没有统一的标准。

●如果想确保初始化顺序，请显式指明。(参见"原则7.20　确保全局变量在使用前被初始化"。)

原则23.17　不要依赖编译器基于实现、未明确或未定义的功能

23.17.1　说明

●基于实现：

指的是编译器必须实现、且实现必须在文档中明确说明、但实现没有公共标准的那些功能，比如有符号整数右移(对符号位的处理)、负数取模等。

●未明确：

指的是编译器必须实现、但无需明示、也没有公共标准的那些功能，比如函数参数的计算顺序等。

●未定义：

指的是编译器无法处理、却又一定会有某种结果的行为，比如释放一个非法指针等。

23.17.2　原因

●基于这些不确定/不通用的功能，其兼容性显然很差。

●这是前面许多具体原则的总结。

原则23.18　注意数据文件的兼容性

原因

●字节排列顺序：

有的系统从右向左组织字节，有的从左向右；从网络上接收数据也有同样的问题：有的系统先传数据的低字节，有的先传高字节；(现在有标准函数可以做转换，请查阅操作系统的有关文档)。

●数据对齐：

为了保证访问效率，数据存放会采用不同的对齐格式，比如字对齐或双字对齐等，所以可能会有无用的空隙在相邻数据之间，编程者也需留神。

●行尾标志：

有的系统用回车做行尾(比如一般的UNIX系统)，有的用回车换行两个字符做行尾(比如Windows系列)。

原则23.19 注意引用公共库的兼容性

说明

尽量选择兼容性好(支持多平台)的库。一般来说，标准库(Standard Library)要好于系统库(操作系统提供的)或自定义库，因为它们是被仔细设计以保证兼容性、高效性等一系列需求的。

原则23.20 一定不要重新实现标准库函数

23.20.1 例子

```
//自己实现memcpy()
void* memcpy{vold* pDes, void* pSrc, size_t length}
{
  //...
}
```

23.20.2 原因

●重新实现标准库函数一定会使人糊涂。

●带来兼容性问题。

原则23.21 将所有#include的文件名视为大小写敏感

23.21.1 例子

```
#include <iostream.h>    //正确
#include <Iostream.h>    //兼容性不好
```

23.21.2 原因

在大小写敏感的系统中，如果乱用文件名的大小写，有可能找不到需引用的文件(文件名不匹配)，从而导致编译失败。

原则23.22 代码中用到的路径只用"/"而不要用"\"

23.22.1 例子

```
#include"myLib/myClass.hpp"
```

23.22.2 原因

●C++标准要求用"/";

●POSIX也要求用"/"。

原则23.23 确保main()函数总是返回一个整型

23.23.1 例子

```
int main(int argc, char* argv[])
{
```

```
    int status=0;
        //...
return status;    //return appropriate status
}
```

23.23.2 原因

在很多系统中，返回一个整型是必须的(用来确定程序的退出状态)，所以如果返回其他类型(比如void)，在某些系统中可能编译不过。

原则23.24 不要依赖pragmas

原因

pragmas的行为是编译器定义的(有的编译器会忽略它，有些实现的作用截然不同)。

第24章　性能

编写出性能优异的程序是几乎每个项目的要求，也是开发者本能的愿望。通过遵从一些简单规则，你可以用很小的代价，换来性能的显著提高。

原则24.1　使用性能追踪分析工具

24.1.1　说明

不要主观想象推测，使用性能追踪分析工具(如Rational公司的Quantify，SUN公司的Forte编译器也集成有这样的工具)，定量性能瓶颈，然后采取相应措施。

24.1.2　原因

●最大的性能瓶颈经常会出现在你预料不到的地方。

●主观推测往往费时且效果不大。相比之下，工具更直接、快速和切中要害。

原则24.2　不要用移位代替乘除运算

24.2.1　例子

```
//不要试图做编译器的优化工作
handOptimized>>=2;
//如果想做除4运算，直接做
handOptimized/=4;
```

24.2.2　原因

●当前主流的编译器都会自动做此类的优化，程序员不必越俎代庖。

●使得代码不直截了当，增加理解难度。

●对于有符号数的移位操作是没有统一标准的，所以结果不确定，降低了兼容性。

原则24.3　如无必要，不要用非int的整型类型

24.3.1　例子

用short、long等取代int可能会影响效率。

24.3.2　原因

●对任何平台，int都是效率最佳的类型，因为它总和该平台的字长(16位、32位或64位等)匹配。

●任何非int类型都可能导致无法使用某些优化手段。

原则24.4　不要使用关键字register

原因

●让编译器来安排register的使用——编译器几乎总能比你做得好。

●使用register可能会妨碍编译器对代码进行全面优化。

原则24.5 避免在循环体内部定义对象

24.5.1 例子

```
for(i=0;i<lengrh;++i)
{
    MyObject_T object;
    //...
  }
```

24.5.2 原因

在循环体内部定义对象会导致每一次循环都要构造和析构该对象。应该把它放在循环体外面。

原则24.6 减少代价很高的对象拷贝

说明

这类对象包括尺寸很大、派生类、包含其他对象的对象等，在拷贝时会花很多时间或调动许多(成员)函数，属于代价很高的拷贝。替代方法请参见“原则3.5 对于非内置类型参数应传递引用(首选)或指针”。

原则24.7 减少临时对象

24.7.1 说明

C++函数经常会产生临时对象，它们都需要创建和删除。对于那些非小型的对象，创建和删除在处理时间和内存两方面的代价不菲。一般的编译器在优化时会尽量减少这些临时对象，但肯定无法彻底消除它们，所以如果编程者自己能够留意、并消除那些不必要的临时对象，会弥补编译器这方面的不足，提高程序的性能。

24.7.2 方法

在保持程序的清晰顺畅的前提下，尽量减少(大型)临时对象的使用，方法有：

●用非临时对象替代临时对象。

●用引用或指针类型的参数代替直接传值。

●用诸如+=代替+(只为减少临时对象)，例如：

```
//可能产生临时对象存储objectA+objectg的结果，然后再赋值给objectA
objectA=objectA+objectB;
//这种方法消除了临时对象
objectA+=objectB;
```

●使用匿名的临时对象：

它会为编译器优化提供更大的灵活性(“原则24.9 返回对象(值)的优化”就是一例)，这是因为如果是有名的临时对象，编译器有时不能确定其他代码是否还会使用

该对象，所以优化时不敢去掉它。但如果是匿名，则其他代码肯定无法引用它，编译器就可以采用诸如将它和其他变量合并等的优化手段。

●关注只带一个显式参数的构造函数：

这种构造函数具有隐式类型转换功能，如果在函数调用时隐式调用它，则会构造出一个临时对象，例如：

```
class MyClass_T
{
  public:
    MyClass_T(int);  //具有将整形转换成MyClass_T的功能
    //...
};
void myFunction(MyClass_T);
/*---------------------------------------------------------------
*下面的函数调用会隐式调用MyClass_T(int)，从而将整数4隐式转换/构
*造成一个MyClass_T的(临时)对象，并传入函数体内
---------------------------------------------------------------*/
myFunction(4);
```

如果你并不想使用此类隐式转换，可以显式禁止，(参见"原则11.6　用关键字explicit防止单参数构造函数的类型转换功能"。)以消除临时对象，前提是合理。

原则24.8　注意大尺寸对象数组

原因

当生成或删除一个由大尺寸对象构成的数组时，每个数组成员的构造函数或析构函数都要被调用一次，这会花费相当长的时间(有时会出乎意料)。

原则24.9　返回对象(值)的优化

24.9.1　说明

当你不得不以值(而不是指针或引用)的形式返回一个对象时，尝试用编译器可优化的方式进行编码会提高效率。这里推荐在返回时直接构造(匿名的)临时变量。

24.9.2　例子

```
//普通的实现方法
Complex_T operator+(Complex_T& lhs，Complex_T& rhs)
{
    Complex_T tmp;  //需要一块内存和一次构造函数调用
    tmp.real=lhs.real+rhs.real;
    tmp.mag=lhs.imag+rhs.imag;
```

```
        return tmp;
    }
/*-------------------------------------------------------------------------
*需要将临时对象tmp赋值给c，然后析构tmp，有些编译器不能对这种做法进行优化
-------------------------------------------------------------------------*/
Complex_T c=a+b:
//优化的实现方法
Complex_T operator+(Complex_T& lhs, Complex_T& rhs)
{
        return Complex_T(lhs.real+rhs.real, lhs.imag+rhs.imag);
}
/*-----------------------------------------------
*编译器此时可借用c作为临时变量，从而在operatar+()
*函数体内直接构造c，从而消除了一个临时变量
-----------------------------------------------*/
Complex_T c=a+b;
```

24.9.3 原因

●从上述例子的比较可以看出，优化算法比不优化少用一块内存、少执行一次构造、一次operator=()和一次析构。

●若Complex_T更复杂一些(如包含其他类的对象作为成员)，则优化效果更明显。

24.9.4 举一反三：传入对象(值)的优化

可以用类似的方法优化传入的对象(值)，即不是构造好一个(临时)对象，然后将其传入函数，而是直接在调用函数时在函数参数处构造一个匿名对象：

```
void myFunction(Complex_T);
Complex_T  a, b;
//优化方法
myFunction(a+b);
//普通方法
Complex_T tmp=a+b;
myFunction(tmp);
```

原则24.10 前缀++和--的效率更高

24.10.1 原因

●后缀++和--会(隐式)产生临时变量(若该临时变量是对象，还会有一次构造，一次operator=()和一次析构)，而前缀不会。

●但注意避免矫枉过正，该用后缀++和--时还得用。

●有的编译器在做优化时会判断是否可以用前缀取代后缀，但不能分辨出所有情

况。

24.10.2 相关描述

对于前后缀操作符的其他描述，请见"原则10.4 关于前后缀操作符"。

原则24.11 恰当使用递归

说明

递归的好处在于易于理解，实现简单。不好的地方是效率不高，需要很多的堆栈空间(当递归层数非常多时要严防堆栈溢出，尤其是在小型系统中)，查错困难。数学证明，递归是可以用非递归的算法替代的(有标准步骤，但变换结果非常费解)。在使用时，应尽量避免将递归用在可能会导致递归层数非常多的地方。

原则24.12 恰当地使用inline函数

24.12.1 说明

将小而轻快的函数定义为inline比普通函数方式效果要好——目标代码增加不多而性能显著提高。反之，将大或耗时很多的函数定义为inline比普通函数方式效果要差——目标代码倍增而性能无甚变换甚至下降。

24.12.2 原因(inline的缺陷)

●编译：

一旦inline函数定义改变，则所有调用该函数的代码都要重新编译，因为(编译时)inline要在调用点就地展开。

●暴露实现：

inline还不太符合面向对象的程序风格：将实现暴露出来，因为inline函数体要放在头文件中。

●调试：

有很多主流编译器(如UNIX上的编译器)不能在inline函数中设置断点，也不能自动将inline函数转换成非inline函数(Visual C++或Borland C++在调试时会自动转换，从而能够设断点)，所以调试时很不方便。

24.12.3 判断方法

●当用inline展开后的代码小于函数调用(隐含)的代码时，用inline。

●当用inline会明显提高效率时，用inline。

●当你预见到inline函数的修改会造成大面积的代码(重新)编译，不用inline。

●对于某些情况，有的编译器会拒绝使用inline方式，此时应不用inline，这是因为inline函数是在头文件中的，如果编译器决定不(能)用inline，则要解决inline函数的多重实现问题，即可能有多个源文件中都有该inline函数体(它们都引用了该inline函数所在的头文件)，如果不处理，连接时会发现多个同名函数。

编译器可以解决这个问题，但是非常复杂，其结果是，inline没用成，还增加了编译时间。

另外，如果考虑兼容性问题，这里列举的所有情况最好都不用inline方式，因为即便你当前使用的编译器支持这种用法，(移植后)其他编译器也可能不行：

●函数包含复杂的控制结构，诸如循环、分支(switch)、try-catch等。若非想用inline方式，可将函数分成两部分：(循环)内部处理部分和外部判断部分，然后分别inline。即使编译器支持inline复杂的控制结构，这样做后效率也会提高。

●展开后的函数体很大或很复杂：有时这是因为该函数调用了其他inline函数，有时是因为隐式调用了构造函数或析构函数(而且派生类的构造/析构函数还会隐式调用基类的构造/析构函数)。

●函数的参数大或复杂。比如函数的参数是一个表达式的结果，或是另一个inline函数的返回值等。

●虚函数不能用inline方式：

inline的虚函数语法是对的(不会有编译错)，但语义有问题：inline是静态绑定的，即在编译时展开。而虚函数是动态绑定的，要在运行时才知道要调用哪个具体函数。所以在绝大多数情况下，编译器会拒绝将虚函数inline，因为在(绝大多数情况下)编译时还不能确定要展开哪个函数(体)。

另外，虚函数要隐式生成唯一一个虚函数表，它应该在某个源文件生成的目标文件中。若虚函数为inline，则其应在头文件中，从而可能被多个源文件引用(#include)，这就造成多份虚函数表的存在，编译器在连接(link)时还要做特别处理。

原则24.13　虚函数和虚继承效率会有一点损失

24.13.1　说明

●函数和虚继承提供了强大的功能，但任何事都是有得有失，使用它们会有一点效率的损失。当然，相对于所得，这点损失是值得的。不过，了解一下虚函数和虚继承的缺点可以使我们更全面地认识它们。

24.13.2　缺点

●增大对象的尺寸：

虚函数和虚继承会(隐式)增加对象的大小，因为每个对象对每个虚函数要增加一个(虚函数)指针。

●增加程序尺寸、降低运行速度：

每类虚函数都需要一张单独的虚函数表，表中每项都是一个函数指针，指向该虚函数在基类或派生类中的一个不同实现(的函数入口)。这样具体对象才能通过它保存的(虚函数)指针(实际上是虚函数表的一个索引值)，找到虚函数表的某一项，然后根据该项中的指针跳到相应的函数入口执行。

但据我所知，这是为达到动态绑定而能采取的最佳方法。同时，编译器还会对虚函数表做特殊优化(比如放在CPU的Cache中)以便进一步提高效率。

●虚函数在绝大多数情况下不能是inline方式。

●对象不能跨进程共享：

包含虚函数或虚继承的对象不能在进程之间共享，因为它们都(隐式)需要指针，而指针实际上是一个进程的内部地址，这个值在另一个进程空间中毫无意义，也就是说，另一个进程不能通过这些指针找到它们想找的东西。其实，所有指针和引用都不能在进程间共享，而不单是虚函数和虚继承，但虚函数和虚继承的指针是隐式加入的，容易被忽略。

原则24.14　如果合理，使用编译器生成的函数

24.14.1　说明

一个类若没有声明缺省构造函数(无参数或全部参数都有缺省值的构造函数)、拷贝构造函数、赋值(operator＝)函数或析构函数，编译器会自动生成。

24.14.2　例子

在C++中使用struct大多数都和C语言的用法一样，但其实C++编译器自动生成上述四个函数。

24.14.3　原因

●效率高：

编译器生成的函数运行效率极高，它甚至可以直接生成汇编代码并做全套优化。

●兼容性好：

这是效率和可移植性的完美结合，因为新平台的编译器会生成在新平台上效率最高的代码，而通常效率和可移植性总是矛盾的。

●如何防止带来隐患：

有些C++书籍不建议用编译器生成的函数，因为阅读者不能区分开发者是忘了还是有意这样做，这会带来隐含的错误。解决的办法是：开发者在有意使用自动生成函数时加一段注释说明此事。

24.14.4　判断方法

●编译器生成的函数是inline函数，所以在inline不适宜的地方不用(见"原则24.12　恰当地使用inline函数")。

●缺省构造函数只调用基类的缺省构造函数，可用自动生成函数。

●拷贝构造函数或赋值函数只是成员赋值，而不涉及更深层次的拷贝(比如成员是指针类型，简单赋值会使两个对象的成员实际上指向同一个目标，这通常是不行的，需要"更深层次的拷贝")，可用自动生成函数。

●析构函数为空，可用自动生成函数。

●最后不要忘记"原则4.14　显式禁止编译器自动生成不需要的函数"。

原则24.15 如果合理，构造直传类

24.15.1 说明

非引用或指针方式下，C++的类(class)、结构(struct)和联合(union)在函数传入和返回时是用值拷贝方式。这些值拷贝若是能像C对待其结构(struct)那样处理，相应的类型就称为直传类。若有自定义的拷贝构造函数，则C++编译器会用该函数构造一个对象的拷贝，传入或返回该拷贝的指针/引用，并在函数完全返回后析构此拷贝，这就是非直传类。

一般情况下，编译器会视满足下述条件之一的类为非直传类：

●有自定义的拷贝构造函数。

●有自定义的析构函数。

●有基类是非直传类。

●有非静态成员是非直传类。

●其他情况都视为直传类。

但有的编译器会视满足下述条件之一的类为非直传类：

●有自定义的拷贝构造函数。

●有虚函数。

●有抽象基类。

●有基类是非直传类。

●有非静态成员是非直传类。

●其他情况都视为直传类。

24.15.2 原因

●直传类传递效率高于非直传类，因为直传类在生成拷贝时更像一次memcpy()调用。而非直传类是每个成员的一系列赋值操作，它要比连续的内存拷贝慢很多。

●尺寸足够小的直传类对象甚至可以(被编译器)放在寄存器中传递。

●直传类的拷贝不需析构。

原则24.16 关于缓存(cache)类成员

24.16.1 说明

编译器/CPU必须经常(通过指针this)从内存中调入类成员。因为是通过指针间接得到的，编译器/CPU有时不能确定上次调入的值在这次是否还有效。此时，只能选择安全但低效的方式——重新调入该值。

你可以用局部变量缓存类成员，使间接引用变成直接引用，从而避免这些不必要的重复动作。

24.16.2 缺点

修改缓存(的类成员)后，要记得适时更新类成员本身，这很容易被忘记，造成很难查的错误，建议少对可修改的类成员做缓存。

原则24.17 关于标准库的性能

例子

C语言的IO函数库(头文件是stdio.h)在性能上要高于C++的IO函数库(头文件是iostream.h)。所以心里要清楚不同的函数库，其性能是有差异的，有时差距还很大。

但是在选择时不能光考虑性能，比如C++的IO库在安全性、可读性、可扩展性上远高于C，所以除非对性能有极端苛刻的要求，否则C++的IO库应是首选。

原则24.18 关于偷懒

24.18.1 说明

偷懒是提高性能的有力方法之一。它的含义是：对(耗时很多的)请求予以延期或简化执行，或是借用别人已有的成果等，但就是不老老实实从头到尾自己做，然后立即回复说请求已被执行。直到请求结果被真正要求使用时才把未作的动作补齐。这一优化技术在计算机领域广泛使用。

24.18.2 前提

请求的结果(较大可能)不会被立刻用到，或者可以借用到。

24.18.3 例子：延迟执行

用户程序在要求操作系统写盘时，操作系统其实只是写在内存的缓冲区中，并未真正写盘。这种拖延会极大提高写盘效率：

●用户写请求本身耗时缩短。

●若用户要读取刚写的内容不用到盘上读。

●若用户随后又写入更新的内容，则第一次写盘完全没有必要。

●若用户写的是连续空间，还可将多次写合并成一次扇区写。

可以看出，拖延时间越长，效果越好。但是拖延时间往往受制于安全性和(内存等的)额外开销。

24.18.4 例子：引用计数

用户程序在要求调用一个动态连接库中的函数时，该函数代码在内存中只有一份拷贝，只是引用数加1。这会极大地节省内存资源，同时降低系统运行开销(因为不用做调入、维护、调出等动作了)。

操作系统在处理许多共享资源时(比如内存页面、文件的访问等)都采用类似的做法。

需要注意：

●引用技术一般适用于被引用对象较复杂或尺寸较大的情况，较小对象恐怕得不偿失。

●当要修改被引用对象时，要提前上锁保护或拷贝一份私用(不再共享)。

24.18.5 例子：延迟判断

```
String_T s = "Hello world.";
```

```
cout << s[3];    //调用operator[]去读s[3]
s[3] = 'L';      //调用operator[]去写s[3]
```

一般来说，operator[]的返回值应该只允许读而不允许写(否则String_T的内部值就被外部修改了，而String_T却不知道)。但对于operatorl]来说，要在返回时确定将来用户是要读还是要写是不可能的。此时就用到了延迟判断：用一个特殊的类包住返回值，如上例中的s[3]，此时它不是char类型，而是一个自定义类的对象，因此读操作(比如cout<<s[3]时，该对象是某个函数(如operator<<())的参数，此时该自定义类只需提供将本类型隐式转换成常量字符型即可，由于是常量，不用担心被修改。如果是写操作(比如s[3]='L')，该自定义类的某个函数(比如operator=())将被调用，此时该自定义类可以做禁止写或通知String_T等动作了。

原则24.19 关于勤快

24.19.1 说明

勤快也是提高性能的有力方法之一。它的含义是：用户还未请求，你已经先把结果预备出来了。这一优化技术在计算机领域同样被广泛使用。

24.19.2 例子：预先执行

用户程序在要求操作系统读盘时，操作系统会一口气读入连续的一个扇区。因为根据统计和试验，用户读取位置邻近数据的概率很大。这种预取会极大提高读盘效率。

24.19.3 例子：分步运算

用户对一个表进行插入、删除和查找。对于表中现存的记录个数，"勤快"的做法是在每次操作后都做计算(插入加1、删除减1)。这比用户请求时再遍历全表要快得多。

24.19.4 前提

请求的结果一定<或很大可能)会被用到。

注意：在具体情况下是用"偷懒"的优化准则还是"勤快"的优化准则，完全看请求的结果被使用的概率。

原则24.20 80—20原则

24.20.1 说明

80-20原则是计算机技术中极为重要的一条原则，具体到性能方面是指：要用20%(小)的代价获得80%(大)的性能提高——所谓事半功倍。而对剩余20%的性能提高余地，不值得用80%的代价去换取，否则就是事倍功半。这一点在做任何优化时一定要切记。

24.20.2 性能与安全性的关系

更进一步说，其实性能在编程时的优先级远低于安全，就是说宁可牺牲性能，

也要保证安全第一。这里的安全有多方面的含义：

●使用安全——本着一切以使用者方便为宜(比如避免出现很难查的错误，尽量把错误暴露在编译而不是运行时)。

●阅读安全——本着以阅读者容易看懂为原则，包括：

●不易曲解代码的含义和用法。

●代码清晰直观而不是晦涩难懂。

●扩展容易——本着以后续开发者容易增减修改的原则(比如不能牵一发而动全身，更不能有不易被察觉的影响)。

当然，最终境界是鱼(性能)和熊掌(安全)兼得，而且是相辅相承、缺一不可。但这种境界太难达到了。必须警醒的是，编程者往往(本能地)沉醉于对性能的追逐，这极易倒置本末而入歧途，大家(特别是高手们)千万小心。

24.20.3　最后

把这条原则放在性能这一章的最后，是为了使大家在最后留下最新鲜的印象，并反思前面所有性能准则。在做优化前问问自己：

●这条优化会破坏安全性吗?

●　这条优化是事半功倍的吗?

第25章 其 他

还有一些没有归到前面章节的原则，都放在这里。

原则25.1 避免产生全局数据对象

25.1.1 说明

全局数据对象几乎没有存在的理由。可将其放在类中，或局限在文件级(static)。如果许多地方需要引用，给出访问函数即可。

25.1.2 原因

●减少命名冲突：

详细说明，请参见“原则1.8 减少全局命名空间级标识符”。

●破坏封装：

使用全局数据对象不符合面向对象的封装原则，因为不可能所有人都需要该数据对象，所以没有必要放在全局，使得大家都能看见。

●在多线程环境下很危险：

因为任何人、任何时候都可以访问它(外部没有一点防范，只能指望数据对象内部能控制并发访问了)。

原则25.2 确保任何定义只发生一次

说明

C语言的编译器允许对同一事物多次定义，这在C++中是被明确禁止的。无论怎样，多重定义都是一个很不好的习惯。

原则25.3 指针操作中用NULL代替0

25.3.1 例子

```
//不好
int* p=0;
//好
int* p=NULL;
```

25.3.2 原因

●NULL能更清楚地表明这是一个空指针。

●ISO C++已经明确将NULL指定为表示空指针的宏。

●另外，一定不要自己重新定义NULL，需要使用时引用标准头文件。

原则25.4 不要把NULL用在指针以外的领域

例子

```
//不要使阅读者困惑
char fileName[LENGTH]=NULL;
```

原则25.5 不要弄巧成拙

25.5.1 说明 ，

编程时不要试图展示自己的非凡手法，比如使用语言的某些罕见的功能，或生成一些巧妙但明显有悖常理的算法或接口。(请结合“原则24.20 80—20原则”，其中有类似的建议。)

25.5.2 原因

软件开发越来越是一项协作工程，让别人容易看懂、容易使用、容易维护是基本的素养。

原则25.6 将不再使用的代码删掉

说明

当项目持续很长时间后，代码中的无用垃圾会越来越多。要及时清除它们。现在的版本管理工具很强大，完全没有必要为其他目的而在当前代码中存放信息(将它们放在它应该在的版本中)。

原则25.7 运行时不要改变进程的环境变量

25.7.1 例子

在程序中执行chdir()或umask()等操作。

25.7.2 原因

如果是多线程或引用了多线程库的程序，可能会造成严重后果。

原则25.8 该用volatile时一定要用

25.8.1 说明

关键字volatile的含义是：其所修饰的变量(的值)可能会发生变化(比如中断程序修改该变量)，而这种变化是编译器察觉不到的。

25.8.2例子

```
char c = ioRegisters.input;  //应该用volatile修饰ioRegisters.input
     c = ioRegisters.input;  //否则编译器优化时可能删掉此句
```

25.8.3 原因

如果不用volatile，编译器的优化算法有可能做不该做的优化，从而导致运行错误。

原则25.9 不要使用鲜为人知的替换符号

例子

ISO C++标准允许用诸如<%替代{，或_eq替代&=。不要用它们，因为没什么人知道这种用法。

原则25.10 关于代码审查(code inspection/review)规范

25.10.1 说明

代码审查是一种制度，它通过相关人员分头阅读代码，并在会上讨论，以发现一些大家常犯的错误、笔误、或不符合管理/规范的代码等。事实证明，这是一种非常有效的手段，并被国际上公认为软件开发必须的过程之一。

25.10.2 状态流程图(图25-1)

25.10.3说明

●分清角色：

作者本人必须认识到代码审查是在帮自己提高效率和质量，这是自己的事，不是大家的事、不是上司的事、也不是开发组的事。

●代码审查单元：

应是逻辑上相对完整、独立的一小块代码，一般在100行左右；另外注意，代码审查要趁热打铁。

●编译通过：

当且仅当编译通过之后，才能进行代码审查。这是因为编译是第一轮的错误扫除，主要是检查拼写、语法和部分语义等初级错误；用编译器做效率高，成效大，其他手段成本高于编译器。代码审查基本上属于第二轮的错误扫描，目的是检查通用语义、用法等中级错误。还有第三轮错误扫描，那是在测试阶段，目的是检查实现等更深层的错误。

对于第二轮扫描，目前有一些自动工具，比如Telelogic公司的Logsicope等，它们可以辅助人工的代码审查(比如发现一些问题、通过一些分析模型勾勒出错误在代码中可能的分布情况等)，所以也应在代码审查之前(编译之后)。

●保存审查前的代码：

通过编译后立刻保存该版本(比如用Rational公司的Clearcase等版本管理工具)，并注明是在某某代码审查之前，使得代码审查是基于一个正式(保存)的版本。

●代码审查的资料：

若是全新的代码，给出程序清单；若是原代码修改，给出对照清单；若需提供其他相关代码，划出要做审查的部分；在散发清单时要做一次概要介绍。

●参加审查的人员：

作者必须找能认真看的人，如交叉使用者、该模块替补开发者、直接技术领导、技术高手、好朋友等(事先认真看占代码审查效果的三分之二，所以一定要保证质量)；同时，只要保证一个(从头到尾)认真看的人就行(最好不要超过两个，否则浪费人力，也不现实)，再加上一个能事先大概看看的技术高手参加审查会

议就行了。

●判断准备是否充分的标准：

作者根据每个人发现问题的数量(参与者要提交发现错误的清单)推测大家看的效果，从而决定是否可以召开下一步的审查会议(这比大家自己报准备时间更有效)。

●审查会议的人数：

参加审查会议的人以3～4个为好，绝对不能超过7人(否则是浪费，且效果极差)。

●会后：

根据审查会议的结果修改代码并编译通过，然后再次保存该版本，并注明是在某某代码审查之后。这样以后可以根据保存的两个版本比较出此次代码审查都发现了什么问题，为今后统计记录提供线索。这样做本身也不费什么时间，当进度很紧时可以既保证代码审查的质量，又节省时间。如果能开发相应的工具，根据保存的两个版本自动产生结果，并辅以少量的人工帮助，将会大大提高代码审查记录、统计的效率。