

# C++编程规范

## 1 简介

为规范项目中以 C++ 为基础语言的代码风格，提到代码的健壮性、可维护性，提高开发效率，特制定本规范。

本规范具有法律效力，除特别说明，或者项目组得到职权部门书面核准，本规范必须执行。

### 1.1 目的

颁布本规范的目的是：

1. 增加代码的健壮性、可读性、易维护性；减少有经验和无经验开发人员编程所需的脑力工作；
2. 在项目范围内统一代码风格；
3. 通过人为以及自动的方式对最终软件应用质量标准；
4. 使新的开发人员快速适应项目环境；
5. 支持项目资源的复用：允许开发人员从一个项目区域（或子项目团队）移动到另一个，而不需要重新适应新的子项目团队的氛围。

### 1.2 适用范围

本规范适用于公司所有以 C++语言为基础的平台下开发的项目。

### 1.3 概述

本规范包括内容：

1. 如何组织项目代码；
2. 编程风格（如何编写实际的源代码）；
3. 如何记录源代码；
4. 代码内名称和源文件所使用的命名约定；
5. 何时使用某些语言结构以及何时应避免某些语言结构。

## 2 基本原则

1. 清晰、可理解的源代码是影响软件可靠性和可维护性的主要因素。清晰、可理解的代码可以表示为以下三个简单的基础原理：
  - 最小混淆：软件的生存期中，源代码的读远比写多，规范、标准更是这样。理想情况下，源代码读起来应该象英语一样描述了所要做的事，这同时还带来了它执行的好处。程序本质上是为编写，而不是为计算机编写的。阅读代码是一个复杂的脑力过程，它可由统一标准来简化，在本文中还包括最小混淆原则。整个项目中统一样式是软件开发团队在编程标准上达成一致的主要原因，它不应视为一种惩罚或对创造性和生产力的阻碍。
  - 维护的唯一性：只要可能，设计决策就应在源中只表述一点，它的多数后果应程序化的派生于此点。不遵守这一原则严重损害了可维护性、可靠性和可理解性。
  - 最小干扰：避免将源代码与可视干扰（如内容较少或对理解软件目的不起作用的信息）相混合；
2. 所表达的精神不过于苛刻；而对正确安全的使用语言特性提供指导。优秀软件的关键在

于:

- 了解每一个特性以及它的限制和潜在的危险;
- 确切了解此特性可安全的使用于哪一个环境中;
- 做出使用高度可视特性的决定;
- 在合适的地方小心适度的使用特性。

### 3 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)。

#### 3.1 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,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) 2004, 光庭导航数据(武汉)有限公司
```

```
* All rights reserved.
```

```
*
```

```
* 文件名称:filename.h
```

```
* 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:2004 年×月×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: 2004 年 月 日
```

```
**/
```

#### 【说明】

关于类的版权和版本申明要保持 C++工程和 Rose UML 模型的统一, 鉴于在 Rose UML 模型中编写这些声明比较麻烦导致工作量增加, 所以可以在 VC 中使用“VC 助手”工具帮助快速编写该类的版权和版本申明, 在 VC 中编写好申明后要将该 C++工程反转回到 Rose UML 模型中, 以保持 C++工程和 Rose UML 模型的一致。

使用 VC 助手的方法:

1. 点击助手工具栏的 Options 按钮
2. 点击 Completion 页面的 Edit 按钮
3. 找到

## 3.2 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。
  2. 预处理块。
  3. 函数和类结构声明等。
- **【规则 3-2—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 3-2—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 3—2-3】**用 `#include "filename.h"` 格式来引用非标准库的头文件(编译器将从用户的工作目录开始搜索)。
  - **【规则 3—2—4】**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,即使是缺省的构造函数和析构函数也不允许在头文件中定义。

- **【规则 3—2-5】**不提倡使用全局变量,尽量不要在头文件中出现 `extern int value` 这类声明.如果要保留全局变量,那么全局变量要保存在一个类中。

## 3.3 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。
2. 对一些头文件的引用。
3. 程序的实现体(包括数据和代码)

## 3.4 头文件的作用

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。

## 3.5 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。

## 4 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。

## 4.1 空行

空行起着分隔程序段落的作用.空行得体（不过多也过少）将使程序的布局更加清晰。空行不会浪费内存。。

- **【规则 4-1-1】** 在每个类声明之后、每个函数定义结束之后都要加一行空行。
- **【规则 4-1-2】** 在一个函数体内，逻辑上密切相关的语句之间不能加空行，而在逻辑上有区别的段落之间必须加空行。例如：

```
// 连接数据库
{
    .....
    .....
}

// 从数据库中读取数据
{
    .....
    .....
}
```

## 4.2 代码行

- **【规则 4-2-1】** 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 4—2-2】** if、for、while、do 等语句独占一行，执行语句不得紧跟其后。不论执行语句有多少都要加 {}。这样可以防止书写失误。
- **【规则 4-2-3】** 函数内定义变量都应放在函数的开始统一进行，以减少出现内存碎片的机会。
- **【建议 4—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。

## 4.3 代码行内的空格

- **【规则 4—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 4—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 4-3—3】** ‘(’ 向后紧跟，‘)’、‘,’、‘;’ 向前紧跟，紧跟处不留空格。
- **【规则 4-3-4】** ‘,’ 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 4—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 4—3-6】**一元操作符如“!”、“~”、“++”、“——”、“&”（地址运算符）等前后应当加空格。
- **【规则 4—3-7】**象“[]”、“.”、“—>”这类操作符前后不加空格。

#### 4.4 对齐

- **【规则 4—4-1】**程序的分界符‘}’应独占一行并且与引用它们的语句左对齐。‘{’可以另起一行，于‘}’左对齐，也可以放在引用它们的语句后面。
- **【规则 4-4-2】**{ }之内的代码块在引用语句的右边间隔一个“Tab”处左对齐。

#### 4.5 长行拆分

- **【规则 4—5-1】**代码行最大长度不能超过 80 个字符。
- **【规则 4-5-2】**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。

#### 4.6 修饰符的位置

- **【规则 4-6—1】**修饰符靠近数据类型和变量名。

例如：

```
int* x; // x 是 int 类型的指针。
```

```
&x = 100; // 给 x 赋值
```

#### 4.7 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。

- **【规则 4-7-1】**注释是对代码的“提示”，不是文档。程序中的注释应言简意赅,不说废话。注释的花样要少。
- **【规则 4—7—2】**如果代码本来就是清楚的，则不必加注释。
- **【规则 4-7—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 4—7-4】**注释应当准确、易懂，防止注释有二义性。
- **【规则 4-7—5】**尽量避免在注释中使用缩写,特别是不常用缩写。
- **【规则 4-7—6】**注释的位置应与被描述的代码相邻，放在代码的上方或右方（对单条语句的注释），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。
- **【规则 4-7-9】**对于所有有物理含义的变量、常量、数据结构和全局变量，如果其命名不

是充分自注释的，必须加以注释。

- 【规则 4-7—10】注释与所描述内容进行同样的缩排。
- 【规则 4-7—11】对于 switch 语句下的 case 语句,如果因为特殊情况需要处理完一个 case 后进入下一个 case 处理,必须在该 case 语句处理完、下一个 case 语句前加上明确的注释。

## 5 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。

### 5.1 共性规则

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

- 【规则 5-1-1】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 CurValue 写成 NowValue。
- 有一些几乎形成习惯用法的简写应该尽可能得到遵循。例如: Addr (地址)、Pnt(Point)、Memo (备注)、Cur(Current)等。
- 【规则 5—1—2】标识符的长度应当符合“min-length && max—information”原则。
- 虽然现在的 C++ 支持长标识符名称,但依然要尽可能设计短小有效的标识符。
- 【规则 5—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 5-1-4】程序中不允许出现仅靠大小写区分的相似的标识符。这非常危险。
- 【规则 5—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 5-1-6】变量的名字应当使用“名词”或者“形容词+名词”。
- 【规则 5—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)。类的成员函数应当尽可能使用“动词”,被省略掉的名词就是对象本身。
- 【规则 5—1—8】用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。
- 【建议 5-1—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。

这种命名风格对维护代码非常有害。

### 5.2 简单的 Windows 应用程序命名规则

- 【规则 5—2—1】类名和函数名用大写字母开头的单词组合而成。
- 【规则 5-2-2】在命名变量名的时候都应该加入其类型的简写。具体的简写对应表如下。假如类成员变量就命名为 m\_b\*\*, m\_ty, m\_w\*\*, m\_dw 等,一般临时变量就命名为 b\*\* , i\*\* , dw\*\* 等;

数据类型

简写(均小写)

BOOL	b
Byte	by
Word	w
DWord	dw
int	i
double	d
long	l
short	n
float	f
char	c
CString	s
Array	a
指针	p
KNTime	t
Handle 类型	h
COLORREF	cr
POINT	pt
Window 窗口类型	wnd
Unsigned long	un
Rect	r

- 【规则 5—2—3】常量全用大写的字母，用下划线分割单词。
- 【规则 5-2-4】静态变量加前缀 s\_（表示 static）。
- 【规则 5—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）。
- 【规则 5-2-6】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变量名混淆。
- 【规则 5-2—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。

## 6 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。

### 6.1 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 6—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	( ) [ ] →) 。	从左至右
	! ~ ++ -- (类型) sizeof	从右至左
	+ - * &	从右至左
	* / %	从左至右

高 到 低 排 列	+ -	从左至右
	<< >>	从左至右
	< <= > >=	从左至右
	== !=	从左至右
	&	从左至右
	^	从左至右
		从左至右
	&&	从左至右
		从右至左
	? :	从右至左
	= += -= *= /= %= &= ^=  = <<= >>=	从左至右

表 6-1 运算符的优先级与结合律

- **【规则 6-1—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 6-1 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。

## 6.2 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。同时建议尽量避免使用复合表达式。

- **【规则 6—2-1】** 不要编写太复杂的复合表达式。
- **【规则 6-2-2】** 不要有多用途的复合表达式。
- **【规则 6-2-3】** 不要把程序中的复合表达式与“真正的数学表达式”混淆。

### 6.2.1 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。

#### 6.2.2 布尔变量与零值比较

**【规则 6-3-1】** 不可将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较，而是直接判断该值。

#### 6.2.3 整型变量与零值比较

**【规则 6-3—2】** 应当将整型变量用“==”或“!=”直接与 0 比较,不可模仿布尔变量的风格而写。

#### 6.2.4 浮点变量与零值比较

**【规则 6-3—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。

### 6.2.5 指针变量与零值比较

【规则 6-3-4】应当将指针变量用“==”或“!=”与 NULL 比较。

### 6.2.6 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 6-4-1】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。
- 【建议 6-4—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 6-4(c)的程序比示例 6-4(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 6-4 (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 6—4(c)的写法比较好,因为程序更加简洁。

<pre>for (i=0; i&lt;N; i++) { if (condition)     DoSomething (); else     DoOtherthing (); }</pre>	<pre>if (condition) { for (i=0; i &lt;N; i++)     DoSomething(); } else { for (i=0; i &lt;N; i++)     DoOtherthing(); }</pre>
--	---

表 6-4 (c) 效率低但程序简洁

表 6-4 (d) 效率高但程序不简洁

## 6.3 for 语句的循环控制变量

- 【规则 6-5—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 6-5-1】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。如: 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。

## 6.4 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 6-6-1】每个 case 语句的结尾必须加 break，以免导致多个分支重叠。也不允许设计多个分支重叠。
- 【规则 6—6—2】switch 最后必须有 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。

## 6.5 goto 语句

【建议 6-7-1】避免使用 goto 语句。

## 7 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)。

### 7.1 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 7—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 PI 等等

### 7.2 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。但是前者比后者有很多的优点：

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。
  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。
  3. 但 #define 来定义常量也有其优点就是不用占用内存空间。
- 【建议 7-2—1】在对内存无过多要求的 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。而在对内存有很大限制的嵌入式开发环境中则需要使用宏常量。

- 【规则 7—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 7-3-2】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。

### 7.3 类中的常量

有时我们希望某些常量只在类中有效。由于 `#define` 定义的宏常量是全局的,不能达到目的.怎样才能建立在整个类中都恒定的常量呢? 应该用类中的枚举常量来实现.枚举常量不会占用对象的存储空间,它们在编译时被全部求值。枚举常量的缺点是：它的隐含数据类型是整数，其最大值有限，且不能表示浮点数(如  $PI=3.14159$ )。

## 8 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。

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，大家常常迷惑不解，容易引起混乱。

### 8.1 参数的规则

- 【规则 8—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。
- 【规则 8-1-2】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`。

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

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

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

- 【规则 8—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。
- 【规则 8—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,对于普通 `int` 型等等的变量则不需要采用“`const &`”方式。
- 【建议 8—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 8—1-2】尽量不要使用类型和数目不确定的参数。

### 8.2 返回值的规则

- 【规则 8-2-1】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 8—2—2】函数名字与返回值类型在语义上不可冲突。
- 【规则 8-2—3】不要将正常值和错误标志混在一起返回。正常值用输出参数获得，而错误标志用 return 语句返回。
- 【建议 8-2—1】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。但不是所有场合都适合用“引用传递”，而有些场合只能用“值传递”而不能“引用传递”，否则会出错。

### 8.3 函数内部实现的规则

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

- 【规则 8—3-1】在函数体的“入口处”，对参数的有效性进行检查。

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

- 【规则 8—3—2】在函数体的“出口处”，对 return 语句的正确性和效率进行检查。

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

注意事项如下：

1. return 语句不可返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。例如
2. 要搞清楚返回的究竟是“值”、“指针”还是“引用”。
3. 如果函数返回值是一个对象，要考虑 return 语句的效率。例如

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

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

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

```
return temp;
```

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

类似地，我们不要将

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

写成

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

```
return temp;
```

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

## 8.4 其它建议

- **【建议 8—4-1】** 函数的功能要单一, 不要设计多用途的函数。

- **【建议 8—4—2】** 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 8—4—3】** 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 8—4-4】** 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。

## 8.5 使用断言

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

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

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

- **【规则 8-5-1】** 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 8-5—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 8—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?” 一旦确定了假定, 就要使用断言对假定进行检查。

- **【建议 8-5—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, 则使用断言进行报警。

## 8.6 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。例如, `n` 是 `m` 的一个引用 (`reference`), `m` 是被引用物 (`referent`)。

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)。

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。

“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。

## 9 内存管理

### 9.1 内存分配方式

内存分配方式有三种:

1. 从静态存储区域分配。内存在程序编译的时候就已经分配好,这块内存在程序的整个运行期间都存在。例如全局变量, **static** 变量。
2. 在栈上创建。在执行函数时, 函数内局部变量的存储单元都可以在栈上创建, 函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中,效率很高,但是分配的内存容量有限。
3. 从堆上分配,亦称动态内存分配。程序在运行的时候用 **malloc** 或 **new** 申请任意多少的内存, 程序员自己负责在何时用 **free** 或 **delete** 释放内存。动态内存的生存期由我们决定,使用非常灵活,但问题也最多。

### 9.2 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, 将指针初始为 **NULL**, 那么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, 我们宁可信其无不可信其有。所以无论用何种方式创建数组, 都别忘了赋初值,即便是赋零值也不可省略,不要嫌麻烦。

- 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 9—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 9—2-2】任何指针在初始时必须赋值 NULL。

- 【规则 9-2-3】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 9—2—4】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 9-2-5】动态内存的申请与释放必须配对,防止内存泄漏。

- 【规则 9—2-6】用 free 或 delete 释放了内存之后，立即将指针设置为 NULL，防止产生“野指针”。

### 9.3 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.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。\_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。所以函数并不能输出任何东西。事实上,每执行一次函数就会泄露一块内存，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”。

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

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，因为该内存存在函数结束时自动消亡，应该返回指向“堆内存”的指针。

### 9.4 free 和 delete 把指针怎么啦？

`free` 和 `delete` 只是把指针所指的内存给释放掉，但并没有删除指针本身。

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，我们有时记不住 `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。

## 9.5 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。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完蛋。这是错觉！

我们发现指针有一些“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。

## 9.6 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：

3. 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。
4. 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。
5. 指针操作超越了变量的作用范围。这种情况让人防不胜防。

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

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

我们不要企图用 `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。

## 9.8 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。
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。
3. 为 `new` 和 `malloc` 设置异常处理函数.例如 `Visual C++` 可以用 `_set_new_handler` 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。

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。

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。

## 9.9 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);
```

我们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。我们通常记不住 `int`, `float` 等数据类型的变量的确切字节数。例如 `int` 变量在 16 位系统下是 2 个字节,在 32 位下是 4 个字节;而 `float` 变量在 16 位系统下是 4 个字节,在 32 位下也是 4 个字节.在 `malloc` 的“( )”中使用 `sizeof` 运算符是良好的风格,但要当心有时我们会昏了头,写出 `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。

## 9.10 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。

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。

在用 `delete` 释放对象数组时, 留意不要丢了符号 ‘[]’。

## 10 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。

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。

### 10.1 函数重载的概念

#### 10.1.1 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? 我们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, 我们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。

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。

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
```

### 10.1.2 当心隐式类型转换导致重载函数产生二义性

例如两个 `output` 函数,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。

## 10.2 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。

### 10.2.1 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。

### 10.2.2 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)。

据考察,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。

## 10.3 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 10—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。
- 【规则 10-3—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。

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。所以我们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。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。

## 10.4 运算符重载

### 10.4.1 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。

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。

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。总结了表 10-4-1 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
<code>= () [] -&gt;</code>	只能重载为成员函数
<code>+= -= /= *= &amp;=  = ~= %= &gt;&gt; = &lt;</code> <code>&lt;=</code>	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 10-4-1 运算符的重载规则

### 10.4.2 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。

## 10.5 函数内联

### 10.5.1 可以用内联取代宏代码

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。

让我们看看 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。“断言 **assert**”恐怕是个例外。**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。

- 【建议 10—5—1】基本上宏代码函数都可以用内联函数替代

如果编程人员仍然习惯用写宏代码函数，那么就应该遵守以下规则：

- 【规则 10-5-2】用宏定义表达式时，要使用完备的括号。
- 【规则 10—5—3】将宏所定义的多条表达式放在大括号中。
- 【规则 10—5-4】使用宏时，不允许参数发生变化。

### 10.5.2 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。**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，但我认为 **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。

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。

### 10.5.3 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？

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。

## 11 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。

### 11.1 构造函数与析构函数的起源

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。

但由于类的成员变量可能需要多次初始或销毁，如果把对象的初始和销毁工作放在构造函数和析构函数中则不能达到这个要求。

- **【建议 11-1-1】**每个类（尤其是实体类）都需要有两个函数 `Initial()` 和 `Release()`，主要做类成员变量的初始化和销毁，在 `Initial()` 中对指针变量设置为 `NULL`，对必要普通成员变量设置初始值 0，在 `Release()` 中 `Delete` 指针变量，并将指针变量重新设置为 `NULL`。在类的构造函数中调用 `Initial()`，析构函数中调用 `Release()`。

### 11.2 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。

## 12 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。

### 12.1 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：

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”。我们应当给“继承”立一些使用规则。

- **【规则 12-1-1】**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 12-1-2】**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。

- 【规则 12—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。

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。

## 12.2 聚合

- 【规则 12—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 12-2—1 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 12—2—1 Head 从 Eye、Nose、Mouth、Ear 派生而成

## 13 其它编程经验

### 13.1 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”。

#### 13.1.1 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。
- 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。

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低。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, 这是我们不期望的。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。

`const &` 修饰输入参数的用法总结如下:

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“ <code>const</code> 引用传递”, 目的是提高效率。例如将 <code>void Func (A a)</code> 改为 <code>void Func(const A &amp;a)</code> 。
---

对于内部数据类型的输入参数, 不要将“值传递”的方式改为“ <code>const</code> 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 <code>void Func(int x)</code> 不应该改为 <code>void Func(const int &amp;x)</code> 。
---

表 13—1—1 “`const &`” 修饰输入参数的规则

### 13.1.2 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。

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。

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。

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, 目的是为了实链式表达。

### 13.1.3 `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。

`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。

## 13.2 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 13-2—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 13—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 13—2-4】**先优化数据结构和算法，再优化执行代码。
- **【规则 13-2-5】**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 13-2-6】**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。

## 13.3 一些有益的建议

- **【建议 13-3-1】**当心那些视觉上不易分辨的操作符发生书写错误。

我们经常会把“==”误写成“=”，象“|”、“&&”、“<=”、“>=”这类符号也很容易发生“丢失”失误.然而编译器却不一定能自动指出这类错误。

- **【规则 13-3—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 13—3—3】**当心变量的初值、缺省值错误,或者精度不够。
- **【建议 13—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 13-3—5】**当心变量发生上溢或下溢，数组的下标越界。
- **【建议 13—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 13-3—7】**当心文件 I/O 有错误。
- **【建议 13-3-8】**避免编写技巧性很高代码。
- **【建议 13-3-9】**不要设计面面俱到、非常灵活的数据结构。
- **【建议 13-3—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 13—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 13-3-12】**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 13-3-13】**把编译器的选择项设置为最严格状态。
- **【建议 13—3—14】**如果可能的话，使用各种工具进行代码审查。