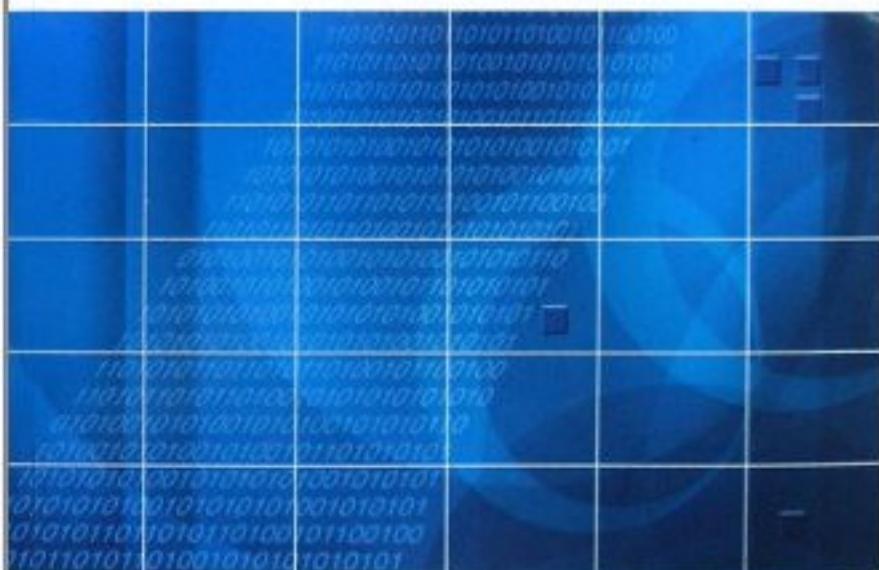




重点大学计算机教材

C++高级进阶教程

主编 陈刚



武汉大学出版社

图书在版编目(CIP)数据

C++高级进阶教程/陈刚主编. —武汉:武汉大学出版社, 2008. 10
重点大学计算机教材
ISBN 978-7-307-06563-5

I. C… II. 陈… III. C 语言—程序设计—高等学校—教材 IV. TP312

中国版本图书馆 CIP 数据核字(2008)第 158016 号

责任编辑: 黄金文 韩光朋

责任校对: 黄添生

版式设计: 支 笛

出版发行: 武汉大学出版社 (430072 武昌 珞珈山)

(电子邮件: wdp4@whu.edu.cn 网址: www.wdp.whu.edu.cn)

印刷: 湖北金海印务公司

开本: 787×1092 1/16 印张: 27.25 字数: 691 千字 插页: 1

版次: 2008 年 10 月第 1 版 2008 年 10 月第 1 次印刷

ISBN 978-7-307-06563-5/TP·317 定价: 38.00 元

版权所有, 不得翻印; 凡购买我社的图书, 如有缺页、倒页、脱页等质量问题, 请与当地图书销售部门联系调换。

重点大学计算机教材

编 委 会 (一)

执行主任: 陈 刚

编 委: 胡启平 王树良 桂 浩

执行编委: 黄金文



内 容 提 要

本书在假定读者有一定的 C++ 编程能力的基础上，进一步加强了对一些基本概念（如文字常量与常变量、指针与引用、作用域与生命期、分离编译模式、声明与定义、静态联编与动态联编等）的解释，介绍了一些不太常用的关键字（如 volatile、mutable、static_cast、dynamic_cast、const_cast、reinterpret_cast 等）的用法。同时，介绍了一些 C++ 语言机制的底层实现方案，如引用是怎样实现的、对象上的实例成员函数是怎样被调用的、虚函数表是如何存储以及如何被访问的、new 和 delete 的实现过程是怎样的，等等。另外，还进一步加深了对一些常用的 C++ 语言机制的讲解，如 sizeof 的用法、typedef 的用法、命名空间的定义和使用、多维数组与多重指针、各种操作符的重载等，同时也对一些高级话题，如怎样调试程序、为什么需要设计模式、怎样应对 C++ 语言的复杂性等进行了探讨。

希望通过这些内容的学习，使读者能够在微观和宏观两个方面进一步拓展对 C++ 语言的认识，从而能够更好地利用它进行程序开发。

本书可作为高年级本科生、研究生的程序设计语言教材，也可供相关的工程技术人员参考。



前 言

本书是为学过一遍C++语言的学生和程序员编写的。因此，本书并没有对C++的入门知识作详细的介绍。所谓“学过一遍”的含义是：至少跟着一本教科书从头到尾学习过一遍，对C++的基础知识和基本概念有所了解。对于初学者来说，普遍的情形是：虽然也可以用C++语言进行编程，但是对C++语言的很多地方仍然感到模糊，想有进一步的提高却找不到合适的教材，只能按部就班地在原来的教材上再走一遍。结果是花费了时间，却达不到好的效果，因为学习的针对性太差。

“学过一遍”C++的人，如何才能有进一步的提高呢？本书打算从五个方面入手：

第一，纠正一些模糊不清的概念和观点。由于编程实践的限制，也由于一般的教材讲述重点的限制，某些要点只是匆匆交代了一下就结束了，导致学生留下一些似是而非的观点。例如，main()函数一定是C++程序第一个执行的函数吗？是由对象的构造函数为对象分配内存空间吗？等等。

第二，介绍一些一般的教材上不介绍或没有深入介绍的内容。如关键字mutable的用法，static_cast、dynamic_cast、const_cast、reinterpret_cast的用法，名字空间(namespace)的定义和使用，等等。这些内容往往由于缺乏实际的例子而无法让学生有贴近的观察。而当读到别人的程序中使用了这些机制的时候，会有一种自己不是真正的C++程序员的感觉。

第三，了解一些C++语言的底层实现机制。例如，细心的程序员都会遇到一些难以解释的“怪”现象，如执行cout<<++i<<-i<<i++;语句，程序的执行结果让人感到惊讶不已，却又一时难以解释。再如，引用是变量吗？引用在底层是怎样实现的呢？这就需要了解一些C++语言的底层实现机制。有时，仅仅停留在高级语言的层面很难把一个问题解释清楚。

第四，介绍一些与编程操作相关的内容。没有足够的编程实践是无法学好一门计算机编程语言的。通常的教科书都忽视了编程操作的问题，认为那是与具体开发平台相关的内容，不适合在教材中讲解。实际上，有些问题是在编程实践中带有共性的问题，无论在哪种开发平台下都回避不了。例如，分离编译模式的问题，以及模板在分离编译模式下产生的典型编译错误应如何解决等，都值得仔细讲解，为学生进一步深入学习扫清障碍。

第五，介绍一些有用的C++编程思想。仅仅掌握C++的语法规则是远远不够的，就像一个只懂得下棋规则的棋手，不经过实战的磨练，不会成为一名出色的大师。把一些优秀的编程思想介绍给读者，能使学习者少走弯路，在较短的时间内进行有针对性的训练从而得到快速的提高。

本书的内容就是针对上述五个目标而展开的。为了加强学习效果，绝大部分的内容都是通过具体的程序实例进行讲解，有的还展示了对应的汇编代码以帮助理解。所有的程序实例都在Visual Studio 2005（在书中简称VS 2005）平台下调试通过。每个例子的规模都不大，但都针对一个具体的问题，因而很有说服力。

需要说明的是，C++语言的底层实现机制大多不属于C++标准的一部分，因而不同的编

译器的实现细节可能是不同的。本书展示的汇编语言代码是在VS 2005平台下反汇编的结果，这些代码能够帮助学习者了解一些在高级语言层面之下的实现方案，从而加深对计算机系统（不仅仅是编程语言本身）的理解。

通过本书的学习，能使学生对C++语言的掌握有一个质的飞跃。不仅能够解释程序的各种行为发生的原因，避免一些编程错误的发生，还可以有意识地编写出高效率的代码来。本书适用于对C++语言有一定程度的掌握而想进一步大幅度提高的读者，也适合用做软件及相关专业高年级本科生或研究生教材。

在本书的编写过程中，武汉大学软件学院的胡启平和桂浩老师提出了很多宝贵的建议，武汉大学出版社的黄金文副编审也提供了大力的支持，在此深表感谢！

本书中的例子大多数是编者自己实际教学的案例，部分参考了一些C++教材和专业网站的资料。书中不足甚至是错误的地方，欢迎来信批评指正。联系邮箱是：

chenzuolin@yahoo.cn

如需要配套的资料，也可以直接与武汉大学出版社或编者本人联系。

作 者

2008年6月于珞珈山



目 录

第1章 C++基础知识	1
1.1 关于C++标准	1
1.2 文字常量和常变量	2
1.3 const的用法	4
1.4 const_cast的用法	10
1.5 mutable的用法	12
1.6 求余运算符	14
1.7 sizeof的用法	15
1.8 引用与指针常量	18
1.9 左值的概念	22
1.10 关于goto语句	24
1.11 volatile的用法	26
1.12 typedef的用法	28
1.13 关于字符串	31
1.14 什么是链式操作	37
1.15 关于名字空间	40
1.16 怎样定义复杂的宏(Macro)	46
1.17 explicit的用法	48
第2章 数据类型与程序结构	51
2.1 C++的数据类型	51
2.2 C++中的布尔类型	54
2.3 void的用法	55
2.4 枚举类型的定义和使用	58
2.5 结构与联合体	60
2.6 数据类型转换	65
2.7 声明与定义的区别	72
2.8 关于初始化	75
2.9 作用域和生命周期	80
2.10 关于头文件	82
2.11 什么是分离编译模式	87
第3章 函数	91
3.1 关于main()函数	91



3.2 函数参数是如何传递的.....	94
3.3 实现函数调用时堆栈的变化情况.....	97
3.4 关于函数参数的默认值.....	100
3.5 如何禁止传值调用.....	102
3.6 定义和使用可变参数函数.....	103
3.7 关于函数指针.....	106
3.8 关于函数重载.....	110
3.9 关于操作符重载.....	113
3.10 类的成员函数与外部函数（静态函数）的区别.....	116
3.11 关于内联函数.....	120
3.12 函数的返回值放在哪里.....	122
3.13 extern “C” 的作用.....	126
第4章 类与对象.....	131
4.1 类与对象概述.....	131
4.2 类定义后面为什么一定要加分号.....	135
4.3 关于初始化列表.....	137
4.4 对象的生成方式.....	144
4.5 关于临时对象.....	147
4.6 关于点操作符.....	150
4.7 嵌套类与局部类.....	153
4.8 对象之间的比较.....	156
4.9 类的静态成员的定义和使用.....	160
4.10 类的设计与实现规范.....	164
4.11 抽象类与纯虚函数.....	169
4.12 类对象的内存布局.....	172
4.13 为什么说最好将基类的析构函数定义为虚函数.....	177
4.14 对象数据成员的初始值.....	179
4.15 对象产生和销毁的顺序.....	180
4.16 关于拷贝构造函数.....	182
第5章 数组与指针.....	186
5.1 数组名的意义.....	186
5.2 什么是指针.....	187
5.3 数组与指针的关系.....	189
5.4 数组的初始化.....	193
5.5 多维数组与多重指针.....	195
5.6 成员数据指针.....	198
5.7 关于 this 指针.....	201
5.8 什么是悬挂指针.....	203
5.9 什么是解引用.....	204

5.10 指针与句柄.....	205
第6章 模板与标准模板库	209
6.1 关于模板参数.....	209
6.2 关于模板实例化.....	215
6.3 函数声明对函数模板实例化的屏蔽.....	217
6.4 将模板声明为友元.....	218
6.5 模板与分离编译模式.....	223
6.6 关于模板特化.....	225
6.7 输入/输出迭代子的用法.....	229
6.8 bitset 的简单用法.....	230
6.9 typename 的用法.....	232
6.10 什么是仿函数.....	233
6.11 什么是引用计数.....	234
6.12 什么是 ADL	238
第7章 内存管理	249
7.1 C++程序的内存布局	249
7.2 理解 new 操作的实现过程.....	254
7.3 怎样禁止在堆（或栈）上创建对象.....	257
7.4 new 和 delete 的使用规范	259
7.5 delete 和 delete[] 的区别	261
7.6 什么是定位放置 new	265
7.7 在函数中创建动态对象.....	266
7.8 什么是内存池技术.....	268
第8章 继承与多态	273
8.1 私有成员会被继承吗.....	273
8.2 怎样理解构造函数不能被继承.....	275
8.3 什么是虚拟继承.....	276
8.4 怎样编写一个不能被继承的类.....	280
8.5 关于隐藏.....	282
8.6 什么是 RTTI	288
8.7 虚调用的几种具体情形.....	296
8.8 不要在构造函数或析构函数中调用虚函数.....	299
8.9 虚函数可以是私有的吗.....	302
8.10 动态联编是怎样实现的.....	304
8.11 !操作符重载	310
8.12 []操作符重载	313
8.13 *操作符重载	316
8.14 赋值操作符重载	317



8.15 输入、输出操作符重载.....	320
第 9 章 流类库与输入/输出.....	323
9.1 什么是 IO 流.....	323
9.2 IO 流类库的优点.....	325
9.3 endl 是什么.....	326
9.4 实现不带缓冲的输入.....	329
9.5 提高输入输出操作的稳健性.....	330
9.6 为什么要设定 locale.....	333
9.7 char* 和 wchar_T* 之间的转换.....	340
9.8 获取文件信息.....	344
9.9 管理文件和目录的相关操作.....	346
9.10 二进制文件的 IO 操作	349
第 10 章 异常处理.....	353
10.1 C++ 为什么要引入异常处理机制	353
10.2 抛出异常和传递参数的不同.....	355
10.3 抛出和接收异常的顺序.....	365
10.4 在构造函数中抛出异常.....	369
10.5 用传引用的方式捕捉异常.....	370
10.6 在堆栈展开时如何防止内存泄漏.....	371
第 11 章 程序开发环境与实践.....	374
11.1 关于开发环境.....	374
11.2 在 IDE 中调试程序时查看输出结果	376
11.3 使用汇编语言.....	377
11.4 怎样调试 C++ 程序.....	379
11.5 关于编码规范.....	382
11.6 正确使用注释.....	385
11.7 静态库与动态库	387
第 12 章 编程思想与方法	395
12.1 C 与 C++ 最大的区别	395
12.2 一个代码重构的例子.....	396
12.3 实现代码重用需要考虑的问题.....	401
12.4 为什么需要设计模式	414
12.5 再论 C++ 的复杂性	419
参考文献	424



pa_to pa_ban to_lapdmp pldc_lapd ldn_fms 识别中等音量人首果碱性。=, =
量级, 声弱大。中弱 G++虽不讲行进方式要不, 翻译或映射上替开来
。语音前面不痛苦。容内首人段中非常 C99 预期 I-1-1
<mem>> sbufonish
<image>> iso900-out
nring unnessesbig
JOpind int
&=, &=, &=, &=, &
<:bb>> <:bb>> <:bb>> <:bb>> <:bb>>

第1章 C++基础知识

1.1 关于 C++标准

美国 AT&T 贝尔实验室的 Bjarne Stroustrup 博士在 20 世纪 80 年代初发明并实现了 C++（最初这种语言被称作“C with classes”）。一开始 C++是作为 C 语言的增强版出现的，从给 C 语言增加类开始，不断地增加新特性。虚函数（virtual function）、运算符重载（operator overloading）、多重继承（multiple inheritance）、模板（template）、异常（exception）、RTTI、名字空间（name space）逐渐被加入标准。1998 年国际标准化组织（ISO）颁布了 C++程序设计语言的国际标准 ISO/IEC 14882-1998。C++是具有国际标准的编程语言，通常称作 ANSI/ISO C++。1998 年是 C++标准委员会成立的第一年，以后每 5 年视实际需要更新一次标准，下一次标准更新将在 2009 年，目前一般称该标准为 C++ 0x。遗憾的是，由于 C++ 语言过于复杂，并且经历了长年的演变，直到现在只有少数几个编译器完全符合这个标准。实际上，从严格的角度来说，至今为止没有任何一款编译器完全支持 ISO C++。

C++语言发展大概可以分为三个阶段：第一阶段从 20 世纪 80 年代到 1995 年。这一阶段 C++语言基本上是传统类型上的面向对象语言，并且凭借着接近 C 语言的效率，在工业界使用的开发语言中占据了相当大的份额；第二阶段从 1995 年到 2000 年，这一阶段由于标准模板库（STL）和后来的 Boost 等程序库的出现，泛型程序设计在 C++中占据了越来越多的比重。同时，由于 Java 等语言的出现和硬件价格的大规模下降，C++受到了一定的冲击；第三阶段从 2000 年至今，由于以 Loki、MPL 等程序库为代表的产生式编程和模板元编程的出现，C++出现了发展历史上又一个新的高峰，这些新技术的出现以及和原有技术的融合，使 C++已经成为当今主流程序设计语言中最复杂的一员。

实际上，C 语言也有自己的标准。从 20 世纪 70 年代初期的早期 C 语言到后来的 K&R C、ANSI C、C89，在将近 20 年中 C 语言多次发展演化，一直到 1999 年 C 语言又重新定案，成为新的 C 语言标准，这就是 C99 标准。

实际上，可以认为 C 与 C++是两门独立的语言。首先，它们有各自的标准和标准委员会；其次，尽管 C++支持 C 语言的几乎全部功能，在语法上与 C 语言还是有极微妙的差别，编译器对全局变量名和函数名的编译方式也不同；使用 C++语言，应该重点使用它的面向对象的特性，而这一点是 C 所完全做不到的。

对于初学者而言，有 C 语言的基础对于掌握 C++有一定的帮助，但这不是必须的。C++并不依赖于 C 语言，完全可以直接学习 C++。根据《C++编程思想》（Thinking in C++）一书所评述的，C++与 C 的效率往往相差在正负 5% 之间。所以，在开发大型程序时 C++完全可以取代 C 语言，仅在非常强调效率、内存空间有限、直接操作硬件等场合使用 C 语言。

使用过 C/C++语言的人都很熟悉下列这些逻辑运算符：&&、||、!、&、|、~、^、&=、|=、



$\wedge =$ 、 $!=$ 。但如果有人在程序中使用 `and`、`or`、`not`、`bitand`、`bitor`、`compl`、`xor`、`and_eq`、`or_eq`、`xor_eq`、`not_eq` 来代替上述逻辑运算符，不要认为他们写的不是 C/C++ 程序。实际上，这是 C99 标准中引入的内容。考察下面的程序。

```
/// Program 1.1-1 ///
#include <iostream>
#include <iso646.h>
using namespace std;
int main(){
    int i=5,j=6,k=7;
```

if(i<j and j<k) cout<<"i<j and j<k"<<endl;
if(i<j or j<k) cout<<"i<j or j<k"<<endl;

此程序会顺利输出：

i<j and j<k
i<j or j<k

其实，像 `and`、`or` 等符号并不是真正新加的什么关键字，而是利用宏替换实现的。打开文件 `iso646.h`，可以看到如下的语句：

```
#define and  &&
#define and_eq  &=
#define bitand  &
#define bitor   |
#define compl  ~
#define not    !
#define not_eq !=
```

所以，要想使用这些逻辑运算符的替代符号，要包含文件 `iso646.h`。假设 Visual Studio 2005 安装在目录 `C:\Program Files\Microsoft Visual Studio 8` 下，则可以在 `C:\Program Files\Microsoft Visual Studio 8\VC\include` 目录下找到头文件 `iso646.h`。有时，直接查看系统头文件可以帮助我们了解一些语言现象的底层实现机制。

1.2 文字常量和常变量

常量可以直观地理解成“值不可改变的量”。在 C++ 语言中，常量分为两种：文字常量（literal constant）和常变量（constant variable）。概括地说，它们之间的关系是这样的：

①文字常量又称为“符号常量”、“字面常量”，经编译之后写在代码区，是不可寻址的。而常变量同其他变量一样被分配空间，是可以寻址的。

例如，在Visual C++中，语句 `int i=3;` 所对应的汇编代码为 `mov DWORD PTR _i$[ebp],3;`，其中 `_i$` 表示在一帧数据中，变量 `i` 距帧指针 `ebp` 的偏移量。文字常量 `3` 被直接写在代码区，在数据区无法找到它。利用预编译指令`#define` 定义的常量也属于文字常量。

整型文字常量前可加 `0`，表示八进制数，加 `0x` 表示十六进制数，如 `0x14` 表示十进制的 `20`。整型文字常量后可加 `L`（或 `l`，但推荐用大写字母，不易和数字混淆）表示 `long` 类型，加 `U`（或 `u`）表示无符号数，如 `1024UL`。

科学计数法中，指数可写作 `e` 或 `E`，如 `3e-3` 表示 3×10^{-3} 。浮点型文字常量缺省为 `double` 型，其后可加 `f` 或 `F` 来表示单精度文字常量；扩展精度由后面跟 `l` 或 `L` 来指示。

字符文字前加 `L` 表示宽字符文字，类型为 `wchar_t`。在字符串后加 “\” 表示在下一行继续。字符串前加 `L` 表示宽字符串文字，以空字符结束。两个（宽）字符串在程序中相邻，C++会把它们连接在一起，如 “two” “some”，结果为 “twosome”。

程序中有些特殊的标识符或表达式，由于同时满足这样两个条件：不可寻址、值不可变，所以可以将它们视为文字常量。它们是：静态数组名、枚举常量、全局（静态变量）首地址。

一个变量，如果限定它在定义的时候就必须同时为它赋初值，且此后它的值就不能再改变，那么这个变量就是一个常变量。常变量由普通变量在前面加 `const` 关键字来定义。常变量的值在初始化后不能改变，是在高级语言的语义这一层面上定义的，由编译器所做的语法检查进行保障。但由于运行时常变量并不是放在只读内存中，而是和一般变量一样放在数据区，所以在运行时如果能获得常变量的地址，一样可以通过特殊的途径对它们进行修改。如下面的程序。

```
/// Program 1.2-1 ///
#include <iostream>
using namespace std;
void ShowValue(const int& i){
    cout<<i<<endl;
}
int main(){
    const int j=5;
    int *ptr;
    void *p=(void *)&j;
    ptr=(int *)p;
    (*ptr)++;
    ShowValue(j);
}
/// End of Program 1.2-1 ///
```

程序的输出结果是 `6`，表明常变量 `j` 的值在运行时的确被修改了。这就证明：常变量是一种加了特殊限制的变量。将常变量理解成“只读”变量会更准确一些。

注意：在上面的程序中，如果将 `main()` 函数中的 `ShowValue(j);` 改成 `cout<<j<<endl;`，那么输出结果仍然是 `5`。这并不是说明常变量 `j` 的值没有改变，而是编译器在代码优化的过程中已经将 `j` 替换成了文字常量 `5` 的缘故。

②出于语法规则或代码优化的需要，在某些情况下文字常量和常变量会由编译器进行转



换。例如，考察下面的程序的输出结果。

```
/// Program 1.2-2 ///
#include <iostream>
using namespace std;
void DefineArray(const int n){
    int B[n]={};
    cout<<B[0]<<endl;
}
int main(){
    const int m=5;
    int A[m]={};
    cout<<A[0]<<endl;
}
/// End of Program 1.2-2 ///
```

实际上，在函数 DefineArray() 体内，`int B[n]={};` 这条语句出现编译错误。而如果去掉函数 DefineArray() 的定义，程序则可以正常运行并输出结果 0。原因是在定义数组 A 的时候，编译器将常变量 m 替换成了文字常量 5；在函数 DefineArray() 体内，无法将常变量 n 替换成相应的文字常量，而在 C++ 中，数组的大小是在编译时决定的，所以出现编译错误。

再看如下的语句：`int &r=5;`

这条语句会出现编译错误，原因是文字常量是不可寻址的，因而无法为文字常量建立引用。

而下面这条语句又是合法的：`const int &r=5;`

这里实际上有一个将文字常量转化成常变量的过程。即先在数据区开辟一个值为 5 的无名整型量，然后将引用 r 与这个整型量进行绑定。

1.3 const 的用法

`const` 是 C++ 语言引入的一个关键字，是“不变的”、“常量”的意思。用 `const` 定义一个变量，实际上是定义了一个“常变量”（即只读变量），关于常变量的有关内容请参见 1.2 节。但是，`const` 的用法远不是这么简单，因为 C++ 中有指针、引用、函数等多种机制，所以和 `const` 组合在一起，就会遇到很多实际问题。下面从 4 个方面总结一下 `const` 的用法。

(1) `const` 的位置

`const` 的位置比较灵活。一般说来，除了修饰一个类的成员函数外，`const` 不会出现在一条语句的最后。以下是一个使用 `const` 的例子。

```
/// Program 1.3-1 ///
#include <iostream>
using namespace std;
int main(){
    int i=5;
    const int v1=1;
}
```

```

int const v2=2; //常量声明语句，义含“只读”。“只读”既表示其中的值
const int *p1; //不能被修改，又表示该指针不能指向其他常量。也就是说，该指针
int const *p2; //所指向的同名变量（即 p1 所指向的变量）是不能被修改的。也就是说，该指针
//以下三条语句都会报编译错误，为什么？ //常量声明语句只能出现在变量前面，不能出现在变量
//const * int p3; //之后。也就是说，const 只能修饰类型，不能修饰某一个变量。于
//int * const p3=&v1; //是，const 不能放在指针后面，也不能放在引用后面。
//int * const p3;
int * const p3=&i;
const int * const p4=&v1;
int const * const p5=&v2;
const int &r1=v1;
int const &r2=v2;
//以下语句会报编译错误，为什么？
//const & int r3;
//以下语句会报警告，并忽略const
int & const r4=i;
cout << *p4 << endl;
}
/// End of Program 1.3-1 ///

```

程序的输出结果是：1. 当然，读者也可以自行考察一下其他变量的值。以上程序演示的是 `const` 的位置与它的语义之间的关系。表面上看起来很复杂，但实际上是有规律可循的。也就是说，`const` 和数据类型结合在一起，形成所谓“常类型”，然后利用常类型来声明或定义变量，这样就产生了常变量。`const` 用来修饰类型时，既可以放在类型的前面，也可以放在类型的后面；用常类型声明或定义变量时，`const` 只会出现在变量前面。`const` 和被修饰类型之间不能有其他标识符存在。

在理解有 `const` 存在的声明语句时，关键是要搞清楚到底什么是“不可变”的。对一个具体的变量来说，如果 `const` 直接出现在该变量的前面，则该变量的值一旦被初始化就不能再改变。所以，`const int v1;` 和 `int const v1;` 都是合法的，而且是等价的。由于 `const` 直接出现在 `v1` 前，所以 `v1` 是只读变量（常变量）。而 `int const *p` 和 `int * const p` 则是不同的声明语句，原因是前者的 `const` 修饰的是 `int`，而后的 `const` 修饰的是 `int *`。前者表示指针 `p` 指向的是一个只读的整型量（指针所指单元的内容不允许修改），而指针本身可以指向其他的常变量；而后者表示指针 `p` 本身的值不能修改（`const` 直接出现在 `p` 的前面），即一旦指针 `p` 指向某个整数之后，就不能再指向其他整型量。如果指针被定义为指针常量，那么在定义它的时候必须同时初始化，否则编译器报错。

为了方便叙述，本书中将“常指针”和“指针常量”定义为不同的术语。常指针代表“指向常量的指针”，“指针常量”则代表指针变量本身是只读的。这种区分是为了统一全书的叙述，而在其他很多书籍中二者都是后一种含义。

引用本身可以理解成一个指针常量，所以在引用前使用关键字 `const` 没有意义。上例中 `int & const r4=i;` 语句的 `const` 是多余的，编译器在给出警告信息后，自动忽略 `const` 的存在。常引用是指将被引用对象当做一个常量，也就是不允许通过引用来修改被引用对象的值。本

书中其他地方提到“常引用”，都代表这种含义。常引用的定义有它独特的特点，具体可参阅 1.9 节。要清楚的一点是：普通变量可以当做常变量看待，但反过来不可以。

在很多情况下，为了表达同一种语义，可以将 `const` 放在不同的位置，如前面的几个例子。但在某些情况下，`const` 只能放在特定的位置，否则意义就会完全不一样。下面是一个 `const` 配合二重指针的例子，在这个程序中，`const` 的位置是不能随意变动的。

/// Program 1.3-2 ///

```
#include <iostream>
using namespace std;
int main(){
    int const **p1;
    int * const *p2;
    int i=5;
    int j=6;
    const int *ptr1=&i;
    int * const ptr2=&j;
    p1=&ptr1;
    p2=&ptr2;
    cout << **p1 << endl;
    cout << **p2 << endl;
}
```

/// End of Program 1.3-2 ///

程序的运行结果是：

```
5
6
```

在程序中定义了两个二重指针 `p1` 和 `p2`，它们的声明分别是 `int const **p1` 和 `int * const *p2`，但含义完全不同。`p1` 不是指针常量，它所指向的变量的类型是 `int const *`（指向整型常量的指针）；`p2` 也不是指针常量，但它所指向的变量是指针常量（`int * const`，即指向整型的指针常量）。所以，为 `p1` 和 `p2` 赋值是有讲究的。如果在上面的程序中，使用这样的赋值 `p1=&ptr2` 或 `p2=&ptr1`，都会产生编译错误。有兴趣的读者可以自行试验一下，并分析产生错误的原因。

(2) `const` 对象和对象的 `const` 成员

用 `const` 来定义一个基本类型的变量，是指不允许显式修改该变量的值。如果用 `const` 来定义某个类的对象，则情况还要复杂一些，因为对象除了有成员数据之外，还有成员函数。用 `const` 修饰的对象称为常对象，而用 `const` 修饰的类的成员函数称为常函数，在常函数中不允许对任何成员变量进行修改。通过常对象，只能调用该对象的常函数。下面是一个关于常对象和常函数的例子。

/// Program 1.3-3 ///

```
#include <iostream>
```

```
using namespace std;
```

```
class A{
```



```

int num;
public:
    A(){num=5;}
    void disp();
    void disp() const;
    void set(int n){num=n;}
};

void A::disp() const{
    cout << num << endl;
}

void A::disp(){
    cout << "Another version of disp()" << endl;
}

int main(){
    A a1;
    a1.set(3);
    a1.disp();
    A const a2;
    a2.disp();
}
/// End of Program 1.3-3 ///

```

程序的执行结果是：

Another version of disp()

5

阅读这个程序，要注意以下几个要点：

①如果将常函数的声明和定义分开进行，那么在两边都要使用 `const` 关键字，否则发生编译错误，如上面程序中 `void disp() const` 的声明和定义。将一个成员函数声明为常函数，只要在函数声明的末尾加上关键字 `const` 即可。只有类的非静态成员函数可以被声明为常函数，其他类型的函数（如外部函数等）不能被声明为常函数。

②一个类的两个成员函数，如果函数的返回值类型、函数名、函数的参数列表完全相同，一个是常函数，一个是普通函数，那么它们之间构成重载关系，如上例中的 `void disp()` 和 `void disp() const`。这是因为，`void disp()` 和 `void disp() const` 具备不同的签名，也就是说，C++ 把 `this` 也作为参数评估的一部分，由于上面的函数被定义成 `class A` 的成员函数，那么它们最终会被看作 `void disp(A*)` 和 `void disp(const A*)`。由于函数参数的类型不同，从而构成函数重载。

非只读对象（如 `a1`）调用某个函数（如 `disp()` 函数）时，先寻找它的非 `const` 函数版本，如果没有找到，再调用它的 `const` 函数版本。而常对象（如 `a2`），只能调用类中定义的常函数，否则编译器会给出错误信息。

如果一个非只读对象（如 `a1`）调用某个函数（如 `disp()` 函数）时，这个函数同时存在非 `const` 函数版本和 `const` 函数版本，而我们希望调用它的 `const` 函数版本，应该怎样做呢？我们必须通过建立该对象的常引用或指向该对象的常指针来达到目的。在上面的程序中，要调



用对象 a1 的常函数 disp(), 可以使用如下的语句:

((const A&)a1).disp();

或者:

((const A*)&a1)->disp();

这两种方式都可以达到目的, 只不过前者显得更简洁一些。

③常对象在创建之后, 其数据成员的值便不允许再修改。所以, 常对象的数据成员的初始化工作很重要, 必须定义合适的构造函数使对象的数据成员拥有有意义的初始值。如果一个类没有显式定义默认构造函数, 而由编译器提供, 在程序中调用默认构造函数生成该类的常对象, 这种做法会遭到很多编译器的拒绝。因为编译器提供的默认构造函数并没有给类对象的数据成员提供任何有意义的初始值。

定义一个常对象, 是将该对象的全体数据成员当做常量看待。而一个非只读对象中, 也可以将部分数据成员定义为常量。这就是类对象的常量成员。类对象的非静态常量成员必须在构造函数中初始化, 而且只能借助初始化列表进行, 具体方法可参见 4.3 节。

(3) 用 const 修饰函数的参数和函数的返回值

在定义函数时常用到 const, 主要是用来修饰参数和返回值。这样做的目的是让编译器为程序员做变量只读性的检查, 以使程序更加健壮。下面是一个例子。

```
/// Program 1.3-4 ///
#include <iostream>
using namespace std;
void disp1(const int &ri){
    cout << ri << endl;
}
void disp2(const int i){
    cout << i << endl;
}
const int disp3(const int &ri){  
    cout << ri << endl;  
    return ri;  
}
int &disp4(int &ri){  
    cout << ri << endl;  
    return ri;  
}
const int &disp5(const int &ri){  
    cout << ri << endl;  
    return ri;  
}
int main(){  
    int n=5;  
    disp1(n);  
}
```



```

    disp2(n);
    disp3(n);
    disp4(n)=6;
    disp5(n); //disp5(n)=6;是错误的
}

```

/// End of Program 1.3-4 ///

程序的输出结果是：

```

5
5
5
5
5

```

阅读上面的程序要注意这样几个要点：

①用 `const` 修饰函数的参数时，主要将被引用对象或被指向的对象声明为常量，如果是将传值调用的形参声明为常量，则没有多大实用价值。在上例中，`void disp2(const int i)` 这样的声明就没有多少意义，因为形参 `i` 是否在函数体内改变，并不影响实参的值，所以将 `i` 声明为常量虽然在语法上没有错，但没有实用价值。不但如此，如果同时定义一个相似的（没有用 `const` 修饰参数）的函数，还会引起重定义错误，如下面的程序。

/// Program 1.3-5 ///

```
#include <iostream>
```

```
using namespace std;
```

```
void f(int i){
```

```
    cout << "int" << endl;
```

```
}
```

```
void f(const int i){
```

```
    cout << "const int" << endl;
```

```
} //主函数
```

```
int main(){
```

```
    f(3);
```

```
    int num=6;
```

```
    f(num);
```

```
}
```

/// End of Program 1.3-5 ///

在这个程序当中，定义了两个名为 `f` 的函数，一个函数原型为 `void f(int)`，另一个函数原型为 `void f(const int)`，由于都是采用传值调用，而任意的整型表达式的值都可以传送给 `int` 型参变量，也可以传送给 `const int` 型参变量，所以这两个函数之间并不构成重载，而是发生重定义错误。

②当函数的返回值是一个普通数据，而不是引用时，用 `const` 修饰函数返回值也没有多少意义。因为这时函数的返回值是一个非左值，本来就不能够改变其值。在 program 1.3-4 例

中, `const int disp3(const int &ri)` 对返回值的 `const` 限定是多余的。

③如果函数的返回值为引用, 那么用 `const` 修饰返回值可以阻止对被引用对象的修改。例如 `disp5(n)=6;` 是错误的, 原因是函数 `disp5()` 的返回值被声明为常引用。

(4) 常见的对 `const` 的误解

在学习 `const` 的用法时, 要留意几种常见的误解。对初学者来说更是这样。

①误解一: 用 `const` 修饰的变量的值一定是不能改变的。在一些特殊情况下, 可能需要临时改变一下常变量的值, 而且这也是做得到的。具体例子如 Program 1.2-1 和 Program 1.4-1。

②误解二: 常引用或常指针, 只能指向常变量。这是一个极大的误解。引用被声明为常引用(或指针被声明为指向常量的指针), 只是说明不能通过该引用(或该指针)去修改被引用对象(或被指向单元)的值。至于被引用对象原来是什么性质, 或者是否会被其他引用修改, 是无法由常引用(或常指针)来决定的。看下面这个小例子。

//// Program 1.3-6 ////

```
#include <iostream>
```

```
using namespace std;
```

```
int main(){
```

```
    int i=5;
```

```
    const int &ri=i;
```

```
    cout << ri << endl;
```

```
    int *p=&i;
```

```
*p=6;
```

```
    cout << ri << endl;
```

```
}
```

//// End of Program 1.3-6 ////

程序的运行结果是:

5

6

在程序中, 虽然 `ri` 被定义为常引用, 但这只是规定了不能通过 `ri` 来修改 `i` 的值, 但并没有阻止通过其他的引用、指针修改 `i` 的值, 甚至直接写 `i=6` 也是没有问题的。所以, 要理解 C++ 设立常引用(常指针)的用意, 主要是保证在一个函数体内部, 不应变化的变量不被意外修改, 从而提高程序的健壮性。

1.4 `const_cast` 的用法

`const_cast` 是一种 C++ 运算符, 作用是去除复合类型中的 `const` 或 `volatile` 属性。一般来说, 定义成 `const` 的变量的值是不能被修改的。但凡事总有例外。C++ 提供了这种转换, 以便在特殊的情况下使用。大量使用 `const_cast` 显然是不明智的, 只能说明程序中存在着设计缺陷。变量本身的 `const` 属性是不能去掉的。要想修改常变量的值, 一般是去除指向该变量的指针(或引用)的 `const` 属性。见下面的例子。

//// Program 1.4-1 ////

```
#include <iostream>
```



```

using std::cout;
using std::cin;
using std::endl;

void ConstTest1(){
    const int a=5;
    int *p;
    p=const_cast<int*>(&a);
    (*p)++;
    cout << a << endl;
}

void ConstTest2(){
    int i;
    cout << "please input a integer:";
    cin >> i;
    const int a=i;
    int &r = const_cast<int &>(a);
    r++;
    cout << a << endl;
}

int main(){
    ConstTest1();
    ConstTest2();
}
/// End of Program 1.4-1 ///

```

从键盘上输入 5，则程序输出：

```

5
6

```

在函数 ConstTest1() 中输出 5，并不表明常变量 a 的值没有改变，而是编译器在优化代码时将 a 替换成了文字常量 5。实际上 a 的值已经变成了 6。在函数 ConstTest2() 中，由于常变量 a 无法转化成对应的文字常量，所以它显示出被修改之后的值。

使用 `const_cast` 进行类型转换时要注意以下几点：

① `const_cast` 操作符的语法形式是：`const_cast<type>(expression)`，其中 `expression` 外面的括号不能省略，哪怕 `expression` 是一个简单变量也是这样。

② 通过 `const_cast` 操作符，只能将 `const type *` 类型转换成 `type *` 类型，或者将 `const type &` 类型转换成 `type &` 类型。也就是说，源类型和目标类型除了 `const` 属性之外，其他方面必须相同，否则不能使用 `const_cast` 操作符进行类型转换。例如，下面的转换就是错误的：

```

const int A[]={1,2,3};
char *p = const_cast<char*>(A); //不能由 const int[] 转换为 char*

```

③ 一个变量本身被定义为只读变量（常变量），那么它就永远是常变量。`const_cast` 取消的是对间接引用时的改写限制（即只针对指针或引用），而不能改变变量本身的 `const` 属性。

如下面的语句就是错误的：int j=const_cast<int>(i)。

④利用传统的 C 语言中的强制类型转换，也可以将 const type *类型转换成 type *类型，或者将 const type &类型转换成 type &类型。但是使用 const_cast 会更好一些。原因是：任何类型转换都存在风险，const_cast 的书写形式比 C 风格的强制类型转换更复杂，因此提醒程序员不要轻易使用类型转换；const_cast 能够更明确地表达程序员的意图，因为它的能力较弱，只能去除复合类型中的 const 属性，而 C 风格的强制转换能力要大得多，风险更大；const_cast 更容易识别，当程序中出现 bug 时，更容易让程序员从类型转换中查找原因。所以，如果确有必要，建议使用 const_cast 操作，而不是采用 C 风格的强制转换。

1.5 mutable 的用法

mutable 是一个使用相对较少的关键字。那么它到底有什么用处呢？看下面的程序。

/// Program 1.5-1 ///

```
#include <iostream>
#include <string>
using namespace std;
class Student{
    string Name;
    int times_of_getname;
public:
    Student(char *name){
        Name=name;
        times_of_getname=0;
    }
    string get_name(){
        times_of_getname++;
        return Name;
    }
    void PrintTimes() const{
        cout<<times_of_getname;
    }
};
```

int main(){
 const Student s("张三");
 cout<<s.get_name()<<endl;
 cout<<s.get_name()<<endl;
 s.PrintTimes();
}

/// End of Program 1.5-1 ///



这段程序创建了一个常量对象 s，代表某一个信息不能被修改的实体，并利用成员函数 get_name()获取学生的姓名，并统计获取学生姓名的次数。很可惜，这段程序会出现编译错误，原因是 s 是一个常量对象，却调用了类 Student 中的非 const 函数 get_name()。假设 s 保持常量不变，将 get_name()改成常函数，那么在 get_name()当中就不能修改成员变量 times_of_getname 的值。

如果要统计常量对象 s 的成员函数 s.get_name()的调用次数，虽然可以增加一个与 s 相关联的外部变量或静态变量来实现，但这显然不是一个很好的办法。

mutable 关键字就是用来解决常函数中不能修改对象的数据成员的问题。如果在某些情况下，希望在常函数中仍然可以修改某个成员变量的值，那么就在该成员变量的声明前面加上关键字 **mutable**，这样不管在什么情况下，这个成员变量的值都可以修改。将上面的程序修改如下。

```
/// Program 1.5-2 //////////////////////////////////////////////////////////////////
#include <iostream>
#include <string>
using namespace std;

class Student{
    string Name;
    mutable int times_of_getname; //在需要修改的成员变量前加关键字mutable
public:
    Student(char *name){
        Name=name;
        times_of_getname=0;
    }
    string get_name() const{
        times_of_getname++; //尽管是在常函数中，用mutable声明的变量仍然可以修改
        return Name;
    }
    void PrintTimes() const{
        cout<<times_of_getname;
    }
};

int main(){
    const Student s("张三");
    cout<<s.get_name()<<endl;
    cout<<s.get_name()<<endl;
    s.PrintTimes();
}

/// End of Program 1.5-2 //
```

这样就能够正确地输出：

张三

张三麻扎，林寒的生学的西游记不息那个一某秀才，老怀是常个一下课的书包朋友，2课出会来骑摩托车。曾巨野，魏大山的生学的西游记，李我的生学的西游记，老怀是常个一下课的书包朋友，所以，**mutable** 关键字能够在保持常量对象中大部分数据成员仍然是“只读”的情况下，实现对个别数据成员的修改。使用关键字 **mutable**，要注意以下几点：

①**mutable** 只能作用于类的非静态和非常量数据成员。

②**mutable** 关键字提示编译器该变量可以被类的 **const** 函数修改。

③在一个类中，用 **mutable** 修饰的变量只能是少数，或者根本不使用 **mutable**。大量使用 **mutable** 表明程序存在设计上的缺陷。

1.6 求余运算符

% 是求余运算符，也叫模除运算符（modulus operator），用于求余数。% 的优先级与 * 及 / 相同，其结合律也是从左到右。% 要求两个操作数均为整数（或可以隐式转化成整数的类型），例如 14.2%3 就是错误的，因为 double 型无法隐式转化成整型。考察下面的程序。

//// Program 1.6-1 ////

```
#include <iostream>
```

```
using namespace std;
```

```
int main(){
```

```
    char c=253;
```

```
    int i=5;
```

```
    cout << c%2 << endl;
```

```
    cout << i%c << endl;
```

```
    cout << 19%10%5 << endl;
```

```
}
```

//// End of Program 1.6-1 ////

这个程序可以正常编译并顺利运行，输出结果是：

-1

2

4

在 C/C++ 语言中，char 类型可以视为单字节整型，其取值范围是 -128~127。所以，char 类型的变量可以直接参与求余运算。253 对应的二进制数是 0xFD，也就是单字节整数 -3 的补码表示。所以，c=253 与 c=-3 是等价的。

C99 标准规定：如果 % 左边的操作数是正数，则模除的结果为正数或 0；如果 % 左边的操作数是负数，则模除的结果为负数或 0。根据上述规则，可以得出如下结论：

$$a\%(-b) == a\%b$$

$$(-a)\%b == -(a\%b)$$

也就是说，当除数或被除数中有负数时，可以做适当的转换再求余数。例如，在上面的程序中，因为 c 的值为 -3，所以 $c\%2 == (-3)\%2 == -(3\%2) == -1$ ， $5\%c == 5\%(-3) == 5\%3 == 2$ 。

又因为 % 运算是从左向右结合的，所以 $19\%10\%5$ 等同于 $(19\%10)\%5$ ，其结果为 4。



1.7 sizeof 的用法

sizeof 是 C/C++ 语言当中的一种基本运算，用来求一个数据类型或变量在内存中占据空间的大小。大家对它也非常熟悉。考察如下的程序。

```
//// Program 1.7-1 ////
#include <iostream>
using namespace std;
int main(){
    cout<<"sizeof(char)="<<sizeof(char)<<endl;
    cout<<"sizeof(short)="<<sizeof(short)<<endl;
    cout<<"sizeof(int)="<<sizeof(int)<<endl;
    cout<<"sizeof(long)="<<sizeof(long)<<endl;
    cout<<"sizeof(float)="<<sizeof(float)<<endl;
    cout<<"sizeof(double)="<<sizeof(double)<<endl;
    int i=9;
    cout<<"i="<<i<<endl;
    cout<<"sizeof(i)="<<sizeof(i)<<endl;
    cout<<"sizeof(i=5)="<<sizeof(i=5)<<endl;
    cout<<"i="<<i<<endl;
}
```

//// End of Program 1.7-1 ////

在 32 位 Windows 操作系统下运行此程序，得到的结果是：

```
sizeof(char)=1
sizeof(short)=2
sizeof(int)=4
sizeof(long)=4
sizeof(float)=4
sizeof(double)=8
i=9
sizeof(i)=4
sizeof(i=5)=4
i=9
```

C++ 基本数据类型的变量占据内存字节数的多少跟运行平台有关。目前占主导地位的是 32 位平台，此例子程序演示了各种基本类型的变量所占空间的大小。

sizeof 运算符号除了接受类型作为参数外，还接受变量或表达式作为参数。在上面的程序中，最容易使人迷惑的是最后一行输出。如果不明白编译器对 sizeof 运算的处理方式，可能会认为输出结果是：i=5，而实际上正确的输出结果却是：i=9。这引申出 sizeof 运算的一个最大特点：sizeof 是一种在编译时进行的运算，而与运行时无关。换句话说，在可执行代码中不包含 sizeof 运算。既然 sizeof 运算与运行时无关，所以当 sizeof 的参数是变量或表达式时，其值在编译时就已确定，与运行时无关。也就是说，sizeof 运算的值在编译时就已经确定，与运行时无关。也就是说，sizeof 运算的值在编译时就已经确定，与运行时无关。

式时，`sizeof` 运算真正关心的是变量或表达式的类型，而不是变量或表达式的值。因为变量 `i` 的类型是 `int`，所以 `sizeof(i)` 等价于 `sizeof(int)`；因为表达式 `i=5` 的类型是 `int`，所以 `sizeof(i=5)` 也等价于 `sizeof(int)`。也就是说，在可执行代码中，并不包含 `i=5` 这个表达式，它早在编译阶段就被处理了。这就是为什么在程序结束时，`i` 的值是 9 而不是 5。

`sizeof` 作用于基本数据类型，在特定的平台上，结果是确定的。如果用 `sizeof` 计算类对象或结构的大小，情况要复杂一些。考察下面的程序。

/// Program 1.7-2 ///

#include <iostream>

using namespace std;

class Small{};

class LessFunc{

 int num;

 void func1(){}

};

class MoreFunc{

 int num;

 void func1(){}

 int func2(){return 1;}

};

class NeedAlign{

 char c;

 double d;

 int i;

};

class Virtual{

 int num;

 virtual void func(){};

};

int main(){

 cout<<sizeof(Small)<<endl;

 cout<<sizeof(LessFunc)<<endl;

 cout<<sizeof(MoreFunc)<<endl;

 cout<<sizeof(NeedAlign)<<endl;

 cout<<sizeof(Virtual)<<endl;

}

/// End of Program 1.7-2 ///

程序的输出结果是：

1 4 4 16 4

24
8

从上面的程序中，可以得出如下结论：

①不允许有长度为 0 的数据类型存在，即使在类中不定义任何数据成员，该类的对象在内存中仍要占据 1 字节。上面的程序中的类 Small 就属于这种情形。

②类的成员函数并不影响类对象所占空间的大小。类对象的大小是由它的数据成员决定的。如上面程序中的 LessFunc 类对象和 MoreFunc 类对象占用相同大小的空间。

③当类的成员变量具有不同类型时，类对象的大小并不是等于各成员变量类型的大小之和。如类 NeedAlign，各成员变量大小之和为 13，但 sizeof(NeedAlign)的值却是 24。这里涉及到一个成员变量对齐的问题，对齐的目的是为了提高 CPU 的存储速度。在默认情况下，VC++规定各成员变量存放的起始地址相对于对象的起始地址的偏移量必须为该变量的类型所占用的字节数的倍数，而整个对象的大小必须是其成员变量中最大尺寸的整数倍。在 NeedAlign 类对象中，成员变量 c 的偏移量是 0，是 sizeof(char)的整数倍，不必调整。接下来，由于成员变量 d 的类型是 double，d 的起始位置必须是 sizeof(double)的整数倍，所以 d 的偏移量被调整为 8（而不是压缩存储时的 1）。同样的原因，成员变量 i 的偏移量被调整为 16。由于在各成员变量中，d 所占据的空间最大，所以 sizeof(NeedAlign)必须是 sizeof(double)的整数倍，最后得到 sizeof(NeedAlign)为 24。从宏观上看，对齐的结果使得有些成员变量占据了更多的空间。例如在类 NeedAlign 中，成员变量 c 占据的空间是 8 字节（其实只有第一个字节有用，其余 7 字节只起到一个填充的作用）。

④如果一个类包含虚函数，那么编译器会在该类对象中插入一个指向虚函数表的指针，以帮助实现虚函数的动态调用。所以，该类对象的大小至少比不包含虚函数时多 4 字节。如果考虑到对齐的因素，可能还要多些。如果使用数据成员之间的对齐，那么至少包含一个数据成员的、拥有虚函数的类对象的大小最小为 8，读者可自行推导一下。

当 sizeof 作用于数组时，求取的是数组全体元素所占空间的大小。而 sizeof 作用于指针时，不管是什类型的指针，大小一律为 4（在 32 位运行平台下），它代表的是指针变量本身在内存中所占空间的大小，函数指针也不例外。下面是一个例子。

```
/// Program 1.7-3 ///
#include <iostream>
using std::cout;
using std::endl;
char testfunc(){
    return 'k';
}
int main(){
    int A[3][5];
    char c[]="123456";
    int a=0;
    int *q;
    double* (*d)[3][6];
    cout <> sizeof(A)<< endl; //输出：60
```

```

cout << sizeof(A[4])<<endl; //输出: 20
cout << sizeof(A[0][0])<<endl; //输出: 4
cout << sizeof(c)<<endl; //输出: 7
cout << sizeof(a)<<endl; //输出: 4
cout << sizeof(&testfunc)<<endl; //输出: 4
cout << sizeof(testfunc())<<endl; //输出: 1
cout << sizeof((*testfunc)())<<endl; //输出: 1
q = new int[10]; //输出: 40
cout << sizeof(q) << endl; //输出: 4
cout << sizeof(d)<<endl; //输出: 4
cout << sizeof(*d)<<endl; //输出: 72
cout << sizeof(**d)<<endl; //输出: 24
cout << sizeof(***d)<<endl; //输出: 4
cout << sizeof(****d)<<endl; //输出: 8
}
/// End of Program 1.7-3 ///

```

程序的运行结果已经写在注释中。下面再做一些补充说明：

- ① A 的数据类型为 int[3][5]，A[4]的数据类型是 int[5]，A[0][0]的数据类型是 int。所以， $\text{sizeof}(A) == \text{sizeof}(\text{int}[3][5]) == 3 * 5 * \text{sizeof}(\text{int}) == 60$ ， $\text{sizeof}(A[4]) == \text{sizeof}(\text{int}[5]) == 5 * \text{sizeof}(\text{int}) == 20$ ， $\text{sizeof}(A[0][0]) == \text{sizeof}(\text{int}) == 4$ 。尽管 A[4]的下标越界，但并不会造成运行时错误，因为 sizeof 运算只关心数据类型，而且在编译阶段就已经完成了。
- ② 由于字符串要以'\0'结尾，所以 c 的数据类型是 char[7]，所以 $\text{sizeof}(c) == \text{sizeof}(\text{char}[7]) == 7$ 。

③ sizeof 如果作用于变量，则有两种写法，如 $\text{sizeof}(a)$ 和 $\text{sizeof } a$ ，都是合法的。而 sizeof 作用于数据类型，只有一种写法。例如， $\text{sizeof}(\text{int})$ 是正确的，而 $\text{sizeof } \text{int}$ 则是错误的。

④ &testfunc 代表一个函数指针，而指针的大小都为 4，所以 $\text{sizeof}(\&testfunc) == 4$ 。testfunc() 代表一次函数调用，返回值类型为 char，所以 $\text{sizeof}(\text{testfunc}()) == \text{sizeof}(\text{char}) == 1$ 。同样，testfunc 名本身就是一个函数指针，所以 $(\ast \text{testfunc})()$ 也是一次函数调用，返回值的类型是 char，所以 $\text{sizeof}((\ast \text{testfunc})()) == 1$ 。

⑤ q 是一个指针，不管它申请了多大的空间，它自身的大小永远是 4。所以 $\text{sizeof}(q) == 4$ 。

⑥ d 是一个指针，所以不管它所指的对象是什么数据类型，它的大小是固定的，所以 $\text{sizeof}(d) == 4$ 。 $\text{sizeof}(*d)$ 的数据类型是 double*[3][6]，所以 $\text{sizeof}(*d) == \text{sizeof}(\text{double}^*[3][6]) == 3 * 6 * \text{sizeof}(\text{double}^*) == 18 * 4 == 72$ 。依据同样的分析，可以推算出： $\text{sizeof}(**d) == \text{sizeof}(\text{double}^*[6]) == 6 * \text{sizeof}(\text{double}^*) == 24$ ， $\text{sizeof}(***d) == \text{sizeof}(\text{double}^*) == 4$ ， $\text{sizeof}(****d) == \text{sizeof}(\text{double}^*) == 8$ 。

1.8 引用与指针常量

引用变量是 C++ 语言引入的一个重要机制，它的出现使得原来在 C 语言中必须用指针实现的功能有了另一种实现的选择，而且在书写形式上更为简便。那么，引用的本质是什么呢？

它与指针有什么关系呢？

很多教科书上说，引用就是被引用对象的一个别名。在高级语言的层面上，或者说在概念上，这种说法没错，但并没有揭示引用的实现方式。还有些资料上说，“引用”自身根本就不是一个变量，它不能脱离被引用对象独立存在，甚至编译器可以不为引用分配内存空间。这种说法是含糊不清的。考察下面的代码：

```
int i=5;
int &ri=i;
ri=8;
```

经过代码优化，可以转换成如下代码：

```
int i=5;
int &ri=i;
ri=8;
```

也就是说，既然 ri 是 i 的别名，那么凡是 ri 出现的地方，都可以直接用它的“原名” i，或者说，在编译的时候，在符号表中 ri 和 i 对应于相同的变量地址，这样的确就没有必要存在一个名为 ri 的变量了。

但是，这种在编译阶段进行静态替换的做法并不是在任何情况下都能进行的。看下面的函数定义：

```
void swap(int &a, int &b){
    int t;
    t=a;
    a=b;
    b=t;
}
```

这是一个经典的交换两个变量的值的函数，在介绍引用的用法时经常被展示。由于函数 swap() 是在不同的上下文环境中被调用的，每次调用的时候，传进函数的实参都可能不相同。所以，在编译阶段将引用替换成被应用对象原名的做法在这里就行不通了。既然没有办法把引用替换成别的内容，那么就必须把引用本身当作实体来看待。

实际上，引用本身的确是一个变量，只不过这个变量的定义和使用都跟普通的变量有显著的不同。为了考察引用变量的底层实现机制，我们先来看一段代码：

```
int i=5;
int &ri=i;
ri=8;
```

在 Visual Studio .NET 2005 环境下反汇编，得到对应的汇编代码如下：

```
mov DWORD PTR _i$[ebp], 5      ; 将整型文字常量 5 送入变量 i
lea  eax, DWORD PTR _i$[ebp]    ; 将变量 i 的地址送入寄存器 eax
mov DWORD PTR _ri$[ebp], eax   ; 将寄存器 eax 的内容（也就是变量 i 的地址）送入变量 ri
mov eax, DWORD PTR _ri$[ebp]   ; 将变量 ri 的值送入寄存器 eax
mov DWORD PTR [eax], 8        ; 将数值 8 送入以寄存器 eax 的内容为地址的单元中
```

可以看到，在汇编代码中，ri 的数据类型为 dword，也就是说，ri 要在内存中占据 4 个字节的位置。所以，ri 的确是一个变量，它存放的是被引用对象的地址。由于通常情况下，地址是由指针变量存放的，那么，指针变量和引用变量有什么区别呢？再来看另外一段代码：



```

int i=5;
int * const pi=&i;
*pi=8;

```

同在 Visual Studio 2005 环境下反汇编，得到对应的汇编代码如下：

```

mov DWORD PTR _i$[ebp], 5
lea eax, DWORD PTR _i$[ebp]
mov DWORD PTR _pi$[ebp], eax
mov eax, DWORD PTR _pi$[ebp]
mov DWORD PTR [eax], 8

```

只要将标识符 pi 换成 ri，所得汇编代码与第一段代码所对应的汇编代码完全一样。所以，引用变量在功能上等同于一个指针常量，即一旦指向某个单元就不能再指向别处。在底层，引用变量也是按指针常量的方式来实现的。

在高级语言层面上，指针常量与引用变量的关系如下：

①在内存中都是占据 4 字节的空间，存放的都是被引用对象的地址，都必须在定义的同时进行初始化。

②指针常量本身（以 p 为例）允许寻址，即 &p 返回指针变量（常变量）本身的地址，被引用对象用 *p 表示；引用变量本身（以 r 为例）不允许寻址，&r 返回的是被引用对象的地址（就是变量 r 自身的值），而不是变量 r 的地址（r 的地址由编译器掌握，程序员无法直接对它进行存取），被引用对象直接用 r 表示。

③凡是使用了引用变量的代码，都可以转换成使用指针常量的对应形式的代码，只不过书写形式上要繁琐一些。反过来，由于对引用变量使用方式上的限制，使用指针常量能够实现的功能，却不一定能够用引用来实现。例如，下面的代码是合法的：

```

int i=5, j=6;
int * const array[]={&i, &j};
int i=5, j=6;
int & array[]={i, j};

```

也就是说，数组元素允许是指针常量，却不允许为引用。这种限制是必要的，它可以避免 C++ 的语法变得过于晦涩。假设允许定义一个“引用的数组”（如上面的 array），那么 array[0]=8; 这条语句该如何理解？是将数组元素 array[0] 本身的值变成 8 呢，还是将 array[0] 所引用的对象的值变成 8？对程序员来说，这种解释上的二义性对正确编程将是一种严重的威胁，毕竟程序员在编写程序的时候，不可能每次使用数组时都要回过头去查数组的原始定义。

C++ 语言规定，引用变量在定义的时候就必须初始化，也就是将引用变量与被引用对象进行绑定。而且这种引用关系一旦确定就不允许改变，直到引用变量结束其生命期。这种规定是在高级语言的层面上，由 C++ 语法和编译器所做的检查来保障实施的。在特定的环境下，利用特殊的手段，我们还是可以在运行时动态地改变一个引用变量与被引用对象的对应关系，使引用变量指向一个别的对象。见下面的程序。

```

/// Program 1.8-1 ///
#include <iostream>

```

念错怕真玉 e.t

```

using std::cout;
using std::endl;

int main(){
    int i=5;
    int j=6;
    int &r=i;
    void *pi,*pj;
    int *addr;
    int dis;
    pi=&i;           //取整型变量i的地址
    cout << pi << endl; //显示整型变量i的地址
    pj=&j;           //取整型变量j的地址
    cout << pj << endl; //显示整型变量j的地址
    dis = (int)pj-(int)pi; //计算连续两个整型变量的内存地址之间的差异
    cout << dis << endl; //显示地址差值
    addr=(int *)((int)pj+dis); //计算引用变量r在内存中的地址
    (*addr) += dis;          //将引用r所指向的对象由变量i转变成变量j
    r=10;
    cout << i << " " << j << endl;
}

```

/// End of Program 1.8-1 ///

这个程序的输出结果是：

0012FF60

0012FF54

-12

5 10

根据 C++ 的语法规则，引用变量 r 在初始化的时候就与变量 i 相绑定，这种绑定关系在 r 的有效期内是不能解除的。执行语句 `r=10;` 后，i 的值应该是 10 而 j 的值应该保持不变。但是，程序的执行结果显示，i 和 j 最后的值分别是 5 和 10，而不是预期的 10 和 6。也就是说，由语句 `r=10;` 改变的是 j 的值，而不是 i 的值。这说明了引用变量 r 所指向的对象由 i 变成了 j。

这是一个非常诡异的程序。在实际编程中绝对不提倡大家模仿。在这里，利用此程序可以看出“引用”本身的确是一个变量，它存放被引用对象的地址。而且，利用特殊的手段能够找到这个引用变量的地址并修改其自身在内存中的值，从而实现重新指派被引用对象的目的。

这个程序的可移植性是很差的。首先，在 64 位平台上，由指针转换为 int 可能会发生截断从而丢失数据；其次，如果引用变量前的变量不是 int 型，考虑到对齐等因素，要准确计算引用变量在内存中的地址也不是一件容易的事，很可能跟具体的编译器和运行环境相关。所以，研究此程序的目的是为了对引用变量的底层实现机制有所了解。在实际使用中，还是要遵循 C++ 语言所制定的使用引用的规范。

1.9 左值的概念

左值 (lvalue) 是 C++ 中的一个基本概念。凡是可以在赋值运算左边的表达式都是左值。与左值相对的是右值 (rvalue)，凡是可以出现在赋值运算右边的表达式都是右值。左值一定可以作为右值，而反过来不一定成立。

可以给左值下一个定义：值为可寻址的非只读单元的表达式称为左值。因此，理解左值的概念，要注意以下几点：

①左值一定是可以寻址的表达式，不能寻址的表达式不能作为左值。例如，表达式 3+5 是一个符号常量表达式，它不能被寻址，因此就不能作为左值。

②常变量虽然可以寻址，但由于只读的限制，也不能作为左值。

③如果表达式的运算结果是一个临时的无名对象，则此表达式不能作为左值，如下面的例子。

```
/// Program 1.9-1 ///
#include <iostream>
using namespace std;
int func(){
    return 0;
}
int main(){
    int i;
    i+1=5;           //statement1
    func()=5;        //statement2
}
/// End of Program 1.9-1 ///
```

这个程序中的 statement1 和 statement2 都是非法的语句，原因是 i+1 的运算结果是一个临时无名对象，函数 func() 的返回值也是一个临时无名对象，所以它们都不能作为左值。注意：这里提到“无名对象”，是指任何没有标识符与之关联的数据实体，包括类对象和基本数据类型的实体。

④如果表达式的运算结果是一个引用，则此表达式可以作为左值，如下面的例子。

```
/// Program 1.9-2 ///
#include <iostream>
using namespace std;
int global;
int &func(){
    return global;
}
int main(){
    int i=2;
    (i+=1)=5;      //statement1
}
```

```

func()=6;           //statement2
cout << i << " ";
cout << global << endl;
}

/// End of Program 1.9-2 ///

```

程序的输出结果是：5 6。在 statement1 中，由于表达式 $i+=1$ 的运算结果是对 i 的引用，所以它可以作为左值。而在 statement2 中，函数调用 $func()$ 的返回结果是对全局变量 $global$ 的引用，所以该表达式也可以作为左值。

由此可知，并不是只有单个变量才能作为左值，也不能仅由表达式的外在形式判断它是否为左值。要根据一个表达式的运算结果的性质进行判断。

结合引用的性质可知：能建立引用的表达式一定是左值；不能作为左值的表达式只能建立常引用，而不能建立一般的引用。由于引用变量中实际上存放的是被引用对象的地址，所以，对非左值建立常引用，首先要考虑该表达式结果是否能寻址，其次还要考虑表达式结果的数据类型与引用数据类型是否一致，只有在满足了这两个条件的基础上，才能将表达式结果的地址送入引用变量。否则，只能另外创建一个无名变量，该变量中存放非左值表达式的运算结果，然后再建立对该无名变量的常引用。

在 C++ 语言中，经常把函数的参数声明为引用，这样在发生函数调用时可以减少运行时开销。但要特别注意的是，将函数的参数声明为一般的引用还是声明为常引用，是有讲究的。在应该将函数参数声明为常引用的时候，却把它声明为一般的引用，很可能造成函数无法正常使用。考察下面的程序。

```
/// Program 1.9-3 ///
```

```

#include <iostream>

using namespace std;
int Max(int &a,int &b){return (a>b)?a:b;}
int main(){
    int i=2;
    cout << Max(i,5) << endl;
}

/// End of Program 1.9-3 ///

```

这个程序无法通过编译。在函数调用 $Max(i,5)$ 中，由于 5 不是左值，不能为它建立引用，所以出现编译错误。在这种情况下，必须修改函数 $Max()$ 的定义，也就是把它的参数声明为常引用： $int Max(const int &a,const int &b)$ ，这样问题就解决了。可见，将函数的参数声明为常引用，不完全是因为参数的值在函数体内不能修改，还考虑到了接受非左值作为函数实参的情况。

另外，对某个变量（或表达式）建立常引用，允许发生类型转换，而一般的引用则不允许，见下面的程序。

```
/// Program 1.9-4 ///
#include <iostream>
```

```

using namespace std;
int Max(const int &a,const int &b){
    return (a>b)?a:b;
}
int main(){
    char c='a';
    const int &rc=c;
    cout << (void*)&c << endl;
    cout << (void*)&rc << endl;
    int i=7;
    cout << Max(rc,5.5) << endl;
}
/// End of Program 1.9-4 /////

```

程序的输出结果是：

```

0012FF63
0012FF48
0012FF3C
0012FF3C
97

```

在这个程序中，如果将语句 `const int &rc=c;` 中的 `const` 去掉，将发生编译错误。原因是一般的引用只能建立在相同数据类型的变量上。同样，之所以允许 `Max(i,5.5)` 这样的函数调用，也是因为函数 `Max()` 的第二个参数是常引用，因此可以将实参 `5.5` 先转换为 `int` 类型的无名变量，然后再建立对该无名变量的常引用。

所以，对一个表达式建立常引用，如果该表达式的结果可以寻址，并且表达式的数据类型与引用类型相同，那么可以直接将表达式结果的地址送入引用变量。此例中，`&i` 和 `&ri` 的值相等就说明了这一点。否则，若是表达式的数据类型与引用类型不相同，或者是表达式结果不可寻址，那么只能另外建立一个无名变量存放表达式结果（或其转换之后的值），然后将引用与无名变量绑定，此例中 `&c` 与 `&rc` 的值不相同正好说明了这一点。

1.10 关于 goto 语句

`goto` 语句是一种无条件跳转语句，相当于汇编语言中的 `jmp` 指令。由于 `goto` 语句有可能破坏结构化程序设计的准则，所以一般都提倡尽量不要使用 `goto` 语句。

在 20 世纪 60 年代末和 70 年代，关于 `goto` 语句发生了比较激烈的争论。有很多人主张从高级语言中去除 `goto` 语句，理由是：`goto` 语句使程序的静态结构和程序的动态执行之间有很大的差别，这使得程序难以阅读，难以查错。毕竟对于大型程序而言，正确性是第一位的。而不同的观点认为：`goto` 语句使用起来比较灵活，有较高的执行效率。如果一味强调删除 `goto`



语句，在某些情况下反而会使程序过于复杂，增加一些不必要的计算量。

1974年，D. E. Knuth 对于 goto 语句的争论作了全面的公正的评述，他的基本观点是：不加限制地使用 goto 语句，特别是使用往回跳的 goto 语句，会使程序的结构难于理解，这种情形应该尽量避免使用 goto 语句；另外，为了提高程序的效率，同时又不破坏程序的良好结构，有控制地使用一些 goto 语句是有必要的。用他的话来说：“有些情形，我主张废除转向语句，有些情形我主张引进转向语句。”（见 D.E.Knuth 的大作：《带有转向语句的结构化程序设计》）。

在 C++ 语言中，如同 C 语言一样，保留了 goto 语句。并且，没有在语法规则上对 goto 语句的使用做任何限制。对 goto 语句的正确使用全凭程序员“自觉”完成。由于早在 20 世纪，结构化程序设计理论就证明了：任何程序都可以用顺序结构、条件结构和循环结构表示出来。所以，goto 语句是基本上不被程序员使用的。从多重循环内部跳出来，算是合理使用 goto 语句的一个较为典型的例子，见下面的程序。

```
/// Program 1.10-1 ///
#include <iostream>
using namespace std;
int main(){
    int A[2][3][4]={1,2,3,4,5,6,7,8,9};
    int i,j,k,order=0;
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            for(k=0;k<4;k++){
                if(A[i][j][k]==8)
                    goto disp;
}
cout << order << endl;
}
cout << order << endl;
```

disp: “跳来用字”，字数关个一拍+**C++** 显 **iostream**。意思是“宝熟木”，“前委易”是 **endl**。
此程序在一个 3 维数组中寻找第一个值为 8 的元素，然后跳出循环，打印出这个元素的线性序号。其中 **disp** 是一个“标号”，它代表程序中的某条语句在内存中的位置。标号后面要跟冒号，而且一定要放在某个可执行语句之前。goto 语句只能跳转到同一个函数体内的标号处，跳到其他函数体内是没有任何意义的。

如果不使用 goto 语句，那么就要用 break 语句跳出多重循环。也就是说，程序应该这样写：

```
/// Program 1.10-2 ///
#include <iostream>
using namespace std;
int main(){
    int A[2][3][4]={1,2,3,4,5,6,7,8,9};
```



```

int i,j,k,order=0;
bool out=false;
for(i=0;i<2;i++){
    for(j=0;j<3;j++){
        if(out) break;
    }
    for(k=0;k<4;k++){
        order++;
        if(A[i][j][k]==8){
            out=true;
            break;
        }
    }
    cout << order << endl;
}
/// End of Program 1.10-2 ///

```

由于 `break` 语句一次只能从一个循环中退出，所以，要多次使用 `break` 语句才能完全从多重循环中退出。很显然，这个版本较之上一个版本，不仅程序的执行效率要低一些，而且可读性也要差一些。所以，不加限制地使用 `goto` 语句固然有害，完全取消 `goto` 语句也是不必要的。正确的做法是：在特定的情况下，安全而高效地使用 `goto` 语句。

1.11 volatile 的用法

`volatile` 是“易变的”、“不稳定”的意思。`volatile` 是 C++的一个关键字，它用来解决变量在“共享”环境下容易出现的读取错误的问题。

在单任务环境中，在一个函数体内部，如果在两次读取变量的值之间的语句没有对变量的值进行修改，那么编译器就会设法对可执行代码进行优化。因为，由于访问寄存器的速度要快过 RAM，所以编译器一般都会作减少存取外部 RAM 的优化。除了第一次读取变量的值时将访问 RAM（从 RAM 中读取变量的值到寄存器），以后只要变量的值“没有改变”，就一直从寄存器中读取变量的值，而不对外存进行访问。

而在多任务环境中，虽然在一个函数体内部，在两次读取变量之间没有对变量的值进行修改，但该变量仍然有可能被其他程序（如中断服务程序、另外的线程等）所修改。如果这时还是从寄存器而不是从 RAM 中读取变量的值，就会出现被修改了的变量的值不能及时反映的问题。下面的程序对这一现象进行了模拟。

```

/// Program 1.11-1 ///
#include <iostream>
using std::cout;
using std::endl;

```



```

int main(){
    int i=10;
    int a = i;
    cout << a << endl;
    _asm{
        mov dword ptr [ebp-4], 80
    }
    int b = i;
    cout << b << endl;
}

```

/// End of Program 1.11-1 ///

此程序在 Visual Studio 2005 环境下生成 release 版本，得到的运行结果是：

```

10
10

```

而在语句 `b=i;` 之前，其实已经通过内联汇编代码修改了 `i` 的值（改成 80，变量 `i` 在内存中的地址是 `[ebp-4]`），所以，送入变量 `b` 的值是原来的 `i` 值，`i` 的变化没有反映到 `b` 中来。如果变量 `i` 是一个被多个任务共享的变量，这种优化带来的错误很可能是致命的。

为了让编译器在生成可执行代码时，抑制对读取变量的这种优化，就要使用 `volatile` 关键字。将上面的程序修改如下。

```

/// Program 1.11-2 /////
#include <iostream>
using std::cout;
using std::endl;
int main(){
    volatile int i=10;
    int a = i;
    cout << a << endl;
    _asm{
        mov dword ptr [ebp-4], 80
    }
    int b = i;
    cout << b << endl;
}

```

/// End of Program 1.11-2 ///

仍然生成 release 版本，程序的输出结果变成：

```

10
80

```

也就是说，第二次读取变量 `i` 的值的时候，已经获得了变化之后的值。跟踪汇编代码可知，凡是声明为 `volatile` 的变量，每次都是从内存中读取变量的值，而不是在某些情况下直接从寄存器中取值。阅读上面的程序，注意这样几点：

①上面的程序一定要在 release 版本下考察，因为只有 release 版本才会对汇编代码作充分的优化。而正是这种优化在变量共享的环境下容易引发问题。

②凡是需要被多个任务共享的变量（如可能被中断服务程序访问的变量、被其他线程访问的变量等），都应该声明为 volatile 变量。而且，为了提高执行效率，要减少对 volatile 变量的不必要的引用。

③由于优化可能会将一些“无用”的代码彻底去除，所以，如果确实想在执行文件中保留这部分代码的话，也可以将其中的变量声明为 volatile。例如下面的程序。

/// Program 1.11-3 ///

```
#include <iostream>
using namespace std;
int main()
```

```
    int i,j,k;
    int s;
```

```
    for(i=0;i<5;i++)
```

```
        for(j=0;j<5;j++)
            for(k=0;k<5;k++)
                cout << s << endl;
```

/// End of Program 1.11-3 ///

在生成 release 版的可执行文件的时候，由于循环体每次送入变量 s 的量都没有变，所以循环实际上是不会被执行的，三重循环被直接优化成了一条赋值语句 s=5。这一点可以跟踪可执行文件的汇编代码看出来。但是，如果程序员的意图是为了让这三重循环“拖延时间”，或者简单地说，想让这三重循环在最终的可执行代码中确实存在，就可以将变量 s 声明为 volatile int s，这样就可以达到目的了。有兴趣的读者可自行跟踪并确认一下。

1.12 **typedef** 的用法

typedef 是 C++语言中的一个关键字，用来为现有类型创建一个新的名字。在 C 语言中就大量使用了 **typedef**，在 C++中，由于存在更为丰富的数据类型，所以 **typedef** 的使用更为频繁。程序员常使用 **typedef** 来编写更美观和可读的代码。所谓美观，是指 **typedef** 能隐藏笨拙的语法构造以及与平台相关的数据类型，从而增强可移植性以及程序的可维护性。

typedef 并不产生新的类型，只是用来将复杂繁琐的类型定义用一个简单的标识符表示。先看下面这个例子。

/// Program 1.12-1 ///

```
#include <iostream>
using namespace std;
```

```
typedef int* pInt;
int main()
{
    int i=2,j=3;
```



```

int* p1, p2;
pInt q1, q2;
p2=4; // 将由变量 p2 的值赋给指针 p1
p1=&p2;
q1=&i; // 将单指针 i 的地址赋给指针 q1
q2=&j;
cout << *q1 << " " << *q2 << " " << *p1 << endl;
}

/// End of Program 1.12-1 /////

```

程序的输出结果是：2 3 4。根据 C++的语法，定义一个指针，可以将*写在靠近类型的一边，也可以把*写在靠近变量的一边。在上面的程序中，int* p1, p2;容易让初学者产生 p1 和 p2 的类型都是 int*的错觉。而实际上，p1 是一个指向整型的指针，而 p2 是一个普通的整型变量。因此，当要在一行定义多个指针时，每个变量前面都必须加上*，否则前面没有*的变量就是一个普通变量而不是一个指针变量。用 **typedef** 将 int *类型用 pInt 标识后，pInt 可以当作一个真正的数据类型看待，而不是仅仅像宏（Macro）一样起一个替换的作用。语句 pInt q1, q2;将 q1 和 q2 都定义成 int*类型，省去了写多个*的麻烦。所以，一定要将由 **typedef** 声明的标识符当做一个“单一”的数据类型看待，而不能机械地用字符替换的方式进行理解。例如，声明 **typedef int* pInt;**，然后定义 **const pInt ptr;**，则 ptr 是一个指针常量，在定义的同时就必须初始化。不可以简单地将 **const pInt ptr;**还原成 **const int* ptr;**进行理解，而应该把 pInt 作为一个整体进行理解。

那么如何书写 **typedef** 声明呢？有很多人简单地把 **typedef** 声明理解成下述格式：

typedef 原数据类型 新名称

按照这种理解，下面几个 **typedef** 声明都应该是合法的：

```

typedef int[8] intarray8;
typedef void (*)(int) pfunc;

```

而这两个 **typedef** 声明都是错误的，不能通过编译。正确的写法应是：

```

typedef int intarray8[8];
typedef void (*pfunc)(int);

```

从中可以看出使用 **typedef** 的一个技巧：如果一个定义变量的语句是合法的，那么在这条语句之前加上 **typedef** 就一定是合法的，并且原来的变量名就成为类型的新名。所以，能够正确使用 **typedef** 的关键，是能够正确理解 C++中的复杂类型说明，然后就可以给这种复杂类型起一个新名字，见下面的例子。

```

/// Program 1.12-2 /////
#include <iostream>
using namespace std;
// 声明四个函数
int Add(int, int);
int Sub(int, int);
int Mul(int, int);
int Div(int, int);

```

```

// 定义指向这类函数的指针类型
typedef int (*FP_CALC)(int, int);

// 声明一个函数s_calc_func，它根据操作字符op 返回指向相应的计算函数的指针
int (*s_calc_func(char op))(int, int);

// 声明一个函数calc_func，它的作用与s_calc_func相同，但声明语句要简单得多
FP_CALC calc_func(char op);

// 根据op 返回相应的计算结果值
int calc(int a, int b, char op);
int Add(int a, int b){
    return a + b;
}
int Sub(int a, int b){
    return a - b;
}
int Mul(int a, int b){
    return a * b;
}
int Div(int a, int b){
    return b ? a / b : -1;
}

//这个函数的用途与下一个函数的用途和调用方式的完全相同，参数为op，而不是最后的两个整型
int (*s_calc_func(char op))(int, int){
    return calc_func(op);
}

FP_CALC calc_func(char op){
    switch (op){
        case '+': return Add;
        case '-': return Sub;
        case '*': return Mul;
        case '/': return Div;
    }
    return NULL;
}

int calc(int a, int b, char op){
    FP_CALC fp = calc_func(op);
    //下面这行是不用typedef，而直接实现指向函数的指针的例子，可读性较差
    int (*s_fp)(int, int) = s_calc_func(op); // ASSERT(fp == s_fp);可以断言它们是相等的
    if (fp) return fp(a, b);
    else return -1;
}

```



```

}

int main()
{
    int a = 100, b = 20;

    cout << "calc(" << a << ", " << b << ", +)=" << calc(a, b, '+') << endl;
    cout << "calc(" << a << ", " << b << ", -)=" << calc(a, b, '-') << endl;
    cout << "calc(" << a << ", " << b << ", *)=" << calc(a, b, '*') << endl;
    cout << "calc(" << a << ", " << b << ", /) =" << calc(a, b, '/') << endl;

    ///////////////////////////////////////////////////
    // End of Program 1.12-2 /////////////////////
}

// 程序的运行结果是:
calc(100, 20, +)=120
calc(100, 20, -)=80
calc(100, 20, *)=2000
calc(100, 20, /)=5

```

这个例子充分显示了利用 `typedef` 给程序的可读性带来的好处。函数 `s_cal_func()` 和 `cal_func()` 的返回值都是一个函数指针，而该函数指针的类型是 `int (*)(int, int)`。由于用了 `typedef` 将此函数指针类型声明为 `FP_CALC`，所以函数 `cal_func()` 的定义非常清晰，即从函数头 `FP_CALC calc_func(char op)` 立即可以知道定义了一个新函数，该函数的名称是 `cal_func`，带一个 `char` 类型的参数，返回值类型为 `FP_CALC`。而函数头 `int (*s_calc_func(char op))(int, int)` 则显得非常复杂，初学者甚至看不懂到底声明了些什么内容。在 C++ 中，模板、函数指针等内容组合在一起，可能形成非常复杂的类型，这时利用 `typedef` 为我们层层分解，对理解复杂类型大有好处。

因此，`typedef` 并不是“可有可无”的、只能起到美化代码的作用的关键字，在某些情况下它是必需的。例如，如下的代码有多少人能够轻松地读懂：

```

int (*Register(int (*pf)(const char *, const char *)))(const char *, const char *);

而如果换成：

typedef int (*PF)(const char *, const char *);
PF Register(PF pf);

```

读懂代码就不是一件困难的事情了。

1.13 关于字符串

字符串是任何一门高级语言都要处理的一种数据。在 C 语言中，字符串是由指向字符串的首字符的指针来表示的，沿着首字符一直向后查找，第一个 ASCII 码为 0 的字符就是字符串结束的位置。C++ 语言沿用了这种表示方式。C/C++ 语言的这种字符串的表示方式不是非常的直观，操作起来也比较繁琐，容易出错。例如，声明两个字符串 `char *s1` 和 `char *s2`，要把 `s2` 中的内容拷贝到 `s1` 中去，要借助函数调用 `strcpy(s1, s2)`。但如果字符串 `s2` 太长，超过了指针 `s1` 所指空间的最大有效长度，就会出现运行时错误。为了避免错误的发生，需要程序员自己进行某些判断，这样就加重了程序员的负担。在 VC++ 2005 中使用 `strcpy()` 函数，会得到一条警告信息：“`strcpy`”被声明为否决的，就是提醒程序员使用该函数可能存在的风险，

并暗示在 VC++将来的版本中可能放弃对该函数的支持。

要处理字符串，就要对计算机系统中字符的编码有所了解。目前广泛使用的有 3 种编码模式，对应于 3 种字符类型。

第一种编码类型是单字节字符集 (single-byte character set or SBCS)。在这种编码模式下，所有的字符都只用一个字节表示。ASCII 是 SBCS。一个字节表示的 0 用来标志 SBCS 字符串的结束。

第二种编码模式是多字节字符集 (multi-byte character set or MBCS)。一个 MBCS 编码包含一些一个字节长的字符，而另一些字符大于一个字节的长度。用在 Windows 里的 MBCS 包含两种字符类型，单字节字符 (single-byte characters) 和双字节字符 (double-byte characters or DBCS)。由于 Windows 里使用的多字节字符绝大部分是两个字节长，所以 MBCS 常被 DBCS 代替。

在 DBCS 编码模式中，一些特定的值被保留用来表明他们是双字节字符的一部分。例如，在 Shift-JIS 编码中（一个常用的日文编码模式），0x81-0x9f 之间和 0xe0-0xfc 之间的值表示“这是一个双字节字符，下一个字节是这个字符的一部分”。这样的值被称做“leading bytes”，他们都大于 0x7f。跟随在一个 leading byte 子节后面的字节被称做“trail byte”。在 DBCS 中，trail byte 可以是任意非 0 值。像 SBCS 一样，DBCS 字符串的结束标志也是一个单字节表示的 0。

第三种编码模式是 Unicode。Unicode 是一种所有的字符都使用两个字节编码的编码模式。Unicode 字符有时也被称作宽字符，因为它比单字节字符宽（使用了更多的存储空间）。注意，Unicode 不能被看作 MBCS。MBCS 的独特之处在于它的字符使用不同长度的字节编码。Unicode 字符串使用两个字节表示的 0 作为它的结束标志。

单字节字符包含拉丁文字母表， accented characters 及 ASCII 标准和 DOS 操作系统定义的图形字符。双字节字符被用来表示东亚及中东的语言。Unicode 被用在 COM 及 Windows NT 操作系统内部。

在 C/C++语言程序中，单字节字符和双字节字符用 `char` 类型表示。Unicode 字符用 `wchar_t` 来表示。Unicode 字符和字符串常量用前缀 `L` 来表示。例如：

```
wchar_t wch = L"1"; // 2 bytes, 0x0031
wchar_t* wsz = L"Hello"; // 12 bytes, 6 wide characters
```

关于 `wchar_t` 更详细的讨论参见 2.1 节。在这里，我们只讨论 `char` 字符类型。

为了更安全、有效地处理字符串，C++标准库引入了一个字符串类 `string`。利用 `string` 类的强大功能可以完成传统 C 风格字符串的全部操作，而且更为安全、方便。下面是一个使用 `string` 类创建字符串对象的例子。

```
/// Program 1.13-1 ///
#include <string>
#include <iostream>
using namespace std;
int main(){
    char chs[] = "Hello";
    string s0; // 创建空字符串
    string s1(chs); // 利用字符指针创建字符串
```

```

string s2("World"); //利用字符串常量创建字符串
string s3(chs,1,3); //指定从chs的索引1开始，一共复制3个字符
string s4(s2); //使用复制构造函数创建新的字符串
string s5(chs,3); //利用chs前3个字符创建字符串
string s6(10,'k'); //创建包含10个相同字符'k'的字符串
cout << s1 << endl;
cout << s2 << endl;
cout << s3 << endl;
cout << s4 << endl;
cout << s5 << endl;
cout << s6 << endl;
cout << "size of s1 is " << s1.size() << endl;
cout << "length of s1 is " << s1.length() << endl;
cout << "capacity of s1 is " << s1.capacity() << endl;
cout << "max_size of s1 is " << s1.max_size() << endl;
cout << "length of s0 is " << s0.length() << endl;
cout << "capacity of s0 is " << s0.capacity() << endl;
if(s0.empty()) cout << "s0 is empty" << endl;
}
/// End of Program 1.13-1 ///

```

程序的执行结果是：

```

Hello
World
ell
World
Hel
kkkkkkkk
size of s1 is 5
length of s1 is 5
capacity of s1 is 15
max_size of s1 is 4294967294
length of s0 is 0
capacity of s0 is 15
s0 is 15 empty

```

可以看到，`string` 类对象一共可有 6 种不同的构造方式。其实，`string` 类把传统的 C/C++ 字符串封装在内部，提供更为安全和强大的操作。要使用 `string` 类，必须包含 `string` 头文件。该类的 `size()` 和 `length()` 是等价的两个成员函数，返回当前字符串的长度。成员函数 `capacity()` 的返回值代表在下一次赋值之前，字符串对象最多所能容纳的字符数。可以看到，创建一个空字符串对象的时候，其 `capacity` 值为 15。换句话说，只有当字符串对象需要容纳超过 15 个字符时，才会重新申请空间以加大其 `capacity` 值。并且，`capacity` 值的加大是单向的，一旦



变得更大，就不会回到较小的值。成员函数 `max_size()` 返回一个 `string` 类对象字符串所能容纳的最大长度。由于 `string` 字符串的长度是由一个无符号 32 位整数表示的，而且字符串用'\0'结尾，所以字符串的最大长度为 4294967294（4294967295 是 32 位无符号整数所能表示的最大值）。成员函数 `empty()` 判断字符串是否为空。

`string` 类是 C++ 标准库提供的一个容器类，只不过该容器装载的是字符。下面的程序演示了对 `string` 类对象进行插入、修改、删除等操作的基本方法。

```
//// Program 1.13-2 /////
#include <string>
#include <iostream>
using namespace std;
int main(){
    char chs[] = "Hello";
    string s1(chs);
    cout << s1 << endl;
    string s2("World");
    s1 = "some";           //重新为字符串指定内容
    cout << s1 << endl;
    s1.assign("another"); //重新为字符串指定内容
    cout << s1 << endl;
    s1.assign("looking",5); //重新分配指定字符串的前5个字符
    cout << s1 << endl;
    s1.swap(s2);          //交换s1和s2的内容
    cout << "s1=" << s1 << endl;
    cout << "s2=" << s2 << endl;
    s2 += "ng";            //添加字符串
    cout << s2 << endl;
    s2.append(" at");     //append()方法也可以添加字符串
    cout << s2 << endl;
    s2.push_back('k');    //push_back()方法只能添加一个字符
    cout << s2 << endl;
    s2.replace(8,3,"at me"); //从索引8开始3个字节的字符替换成"at me"
    cout << s2 << endl;
    s2.insert(0,"You are "); //在索引处插入字符串，原来的内容后移
    cout << s2 << endl;
    //s2.insert(s2.size()," now"); //在字符串末尾插入字符串，原来的内容保持不动
    cout << s2 << endl;
    s2.erase(0,4);          //从索引开始删除个字符，即删除掉了"You"
    cout << s2 << endl;
    s2.replace(0,4,"");    //将指定范围内的字符替换成""，即变相删除了
    cout << s2 << endl;
```



```

s1.clear();           //清空字符串
cout << "s1=" << s1 << "" << endl;
s1 = "World";
s1.erase(0,s1.length()); //用erase()方法删除全部字符
cout << "s1=" << s1 << "" << endl;
s1 = "World";
s1.replace(0,s1.length(),""); //用replace()方法删除全部字符
cout << "s1=" << s1 << "" << endl;
}

/// End of Program 1.13-2 ///

```

程序的执行结果是：

```

Hello
some
another
looki
s1= World
s2= looki
looking
looking at
looking atk
looking at me
You are looking at me
You are looking at me now
are looking at me now
looking at me now
s1=
s1=
s1=

```

可直接利用赋值语句改变字符串对象中存放的字符串值。这实际上是对赋值运算符 operator= 进行重载的结果。如何对赋值运算符进行重载可参见 8.14 节。成员函数 assign() 也可以为字符串赋值，而且有多种重载形式，使用起来较为灵活。其他的如交换、插入、替换、删除等操作，都可对照注释了解相关用法。

由于 string 是一个类，所以 string 类对象之间进行比较必须重载关系运算符。C++ 标准库已经重载了这些关系运算符，所以可以直接使用。见下面的例子。

```

/// Program 1.13-3 ///
#include <string>
#include <iostream>
using namespace std;
int main()
{
    string s1 = "abcdefg";

```

```

string s2 = "abcdefg";
if(s1==s2) cout << "s1 == s2" << endl;
else cout << "s1 != s2" << endl;
if(s1!=s2) cout << "s1 != s2" << endl;
else cout << "s1 == s2" << endl;
if(s1>s2) cout << "s1 > s2" << endl;
else cout << "s1 <= s2" << endl;
if (s1<=s2) cout << "s1 <= s2" << endl;
else cout << "s1 > s2" << endl;
}

```

/// End of Program 1.13-3 ///

程序的运行结果是：

```

s1 == s2
s1 == s2
s1 <= s2
s1 <= s2

```

两个字符串比较大小，是按照字典序进行的。在字典中排列靠前的字符串，被认为是较小的字符串，否则就是较大的字符串。

为了实现与传统的 C 风格的字符串类似的操作，string 类允许以[]操作或 at()函数访问字符串中的字符。可以将 string 类对象转换成传统的 C 风格的字符串（字符指针），还可以迭代器的方式遍历字符串。见下面的例子。

/// Program 1.13-4 ///

```

#include <string>
#include <iostream>
using namespace std;
int main(){
    int i;
    string s = "abcdefghijkl";
    cout << "use []:" << endl;
    for(i=0; i<s.length(); i++)
        cout << s[i];
    cout << endl;
    cout << "use at():" << endl;
    for(i=0; i<s.length(); i++)
        cout << s.at(i);
    cout << endl;
    s[s.length()-1] = '2';
    cout << s << endl;
    const char * chs1 = s.c_str();
    const char * chs2 = s.data();
}

```

```

    cout << "c_str() : " << chs1 << endl;
    cout << "data() : " << chs2 << endl;
    string str = s.substr(5,3); //从索引5开始3个字节
    cout << str << endl;
    string pattern = "fg";
    string::size_type pos;
    pos = s.find(pattern,0); //从索引0开始,查找第一次出现"fg"的索引值
    cout << pos << endl;
    string str1 = s.substr(pos,pattern.size());
    cout << str1 << endl;
    for(string::iterator iter = s.begin(); iter!=s.end(); iter++)
        cout << *iter;
    cout << endl;
}

```

/// End of Program 1.13-4 ///

程序的运行结果是：

```

use []:
abcdefg1111
use at():
abcdefg1111
abcdefg1112
c_str() : abcdefg1112
data() : abcdefg1112
fg1
5
fg
abcdefg1112

```

在上面的程序中，`s.c_str()`函数返回的是一个 C 风格的字符串，也就是说最后一个字符是 '`\0`'，而 `s.data()` 则返回一个字符数组的首地址，并不保证这个数组中包含元素 '`\0`'。由于字符串的内容是由 `string` 类对象拥有的，所以这两个函数只提供读取字符串的功能，而不允许利用函数的返回值对字符串的内容进行修改。

使用 `[]` 运算符与使用 `at()` 成员函数都可以读写字符串中的某个字符，它们的区别在于：使用 `[]` 运算符不检查下标是否越界，而使用 `at()` 函数时，当下标越界就会抛出异常。

至于以迭代器的方式遍历字符串，那是 STL 的标准做法，可参阅 STL 的相关内容。

1.14 什么是链式操作

链式操作是利用运算符进行的连续运算（操作），它的特点是在一条语句中出现两个或两个以上相同的操作符，如连续的赋值操作、连续的输入操作、连续的输出操作等都是链式操作的例子。链式操作一定涉及到结合律的问题，例如链式赋值操作满足右结合律，即 `a=b=c`

被解释成 `a=(b=c)`, 而链式输出操作则满足左结合律, 即 `cout << a << b` 被解释成 `(cout << a) << b`。基本数据类型的链式操作都有明确的定义, 而涉及到类类型的链式操作则往往需要进行操作符的重载。而且, 为了使链式操作能够进行, 操作符的重载必须满足一定的要求:

①操作符重载函数一定不能返回 `void` 类型。

因为 `void` 类型不能参与任何运算, 所以, 操作符重载函数返回 `void` 类型实际上是阻止了链式操作的可能性。

②对赋值操作符进行重载, 如果返回的是类的对象, 那么链式赋值操作必须借助于拷贝构造函数才能进行。这样不但会有较大的运行开销, 还要编写正确的拷贝构造函数。考察下面的程序。

```
/// Program 1.14-1 ///
#include <iostream>
using namespace std;
class Complex{
    double real;
    double image;
public:
    Complex(double r=0.0,double i=0.0){
        real=r;
        image=i;
    }
    Complex(const Complex& c){
        cout << "Copy Constructor" << endl;
        real=c.real;
        image=c.image;
    }
    void Show(){
        cout << real << "+" << image << "i" << endl;
    }
    Complex operator=(const Complex& c){
        real=c.real;
        image=c.image;
        return c;
    }
}
int main(){
    Complex c1(2.3,4.5),c2,c3;
    c1.Show();
    c3=c2=c1;
    c2.Show();
}
```

该程序首先输出一个复数 `c1`, 然后输出一个复数 `c3`, 其中 `c3` 的值等于 `c2`。由于 `c2` 的值等于 `c1`, 因此输出结果为 `2.3+4.5i`。但问题是, 在 `c3=c2=c1;` 语句中, 由于先执行了 `c3=c2`, 因此 `c3` 的值等于 `c2`, 而 `c2` 的值等于 `c1`, 因此 `c3` 的值等于 `c1`。所以输出结果为 `2.3+4.5i`。这说明, 在链式赋值操作中, 拷贝构造函数的返回值不能是类对象。



```
c3.Show();
}

/// End of Program 1.14-1 ///
```

程序的输出结果是：

```
2.3+4.5i
Copy Constructor
Copy Constructor
2.3+4.5i
2.3+4.5i
```

可以看到，在连续的两次赋值操作中，一共两次调用拷贝构造函数。第一次发生在执行 `c2=c1` 的操作中，函数的返回值（临时对象）是由 `c1` 构造的，这时发生一次拷贝构造函数的调用；第二次发生在为 `c3` 赋值的时候，赋值运算的返回值仍然是一个 `Complex` 类的对象，这时又发生一次拷贝构造函数的调用。让赋值操作依赖于拷贝构造函数，显然不是一种明智的做法。

思考：如果把赋值操作符的重载定义为 `Complex Complex::operator=(Complex &c){...}`，会有什么后果？

如果把程序改造一下，使赋值操作符函数返回引用，则有：

```
/// Program 1.14-2 ///
```

```
#include <iostream>
using namespace std;

class Complex{>
public:
    Complex(double r=0.0,double i=0.0){real=r;image=i;}
    Complex(const Complex& c){
        cout << "Copy Constructor" << endl;
        real=c.real;
        image=c.image;
    }
    Complex& operator=(const Complex& c){
        cout << real << "+" << image << "i" << endl;
        real=c.real;
        image=c.image;
        return *this;
    }
    void Show(){
        cout << real << "+" << image << "i";
    }
};

Complex& operator=(const Complex &c){
    Complex& Complex::operator=(const Complex &c){
        real=c.real;
        image=c.image;
        return *this;
    }
}
```

```

image=c.image;
    :c3.Show();
return *this;
}

int main(){
    Complex c1(2.3,4.5),c2,c3;
    c1.Show();
    c3=c2=c1;
    c2.Show();
    c3.Show();
}
} //End of Program 1.14-2 ///

```

程序的执行结果是：

也就是说，一次拷贝构造函数都没有调用，原因是赋值操作符函数返回的是 Complex 类的引用，不用产生一个新的临时对象，这样大大提高了程序的运行效率。所以，赋值运算符的重载几乎无一例外地采用返回引用的方式编写。

思考：像 $x+y+z$ 这样的表达式是链式操作吗？重载 operator+ 时返回值类型是类对象还是引用？

③为了实现输入、输出的链式操作，输入操作符 ($>>$) 和输出操作符 ($<<$) 的重载函数必须返回引用，否则链式操作无法完成。

一般来说，实现输入操作符重载，一律采用如下函数原型：

```
istream& operator>>(istream&,classNam&);
```

而实现输出操作符重载，一律采用如下函数原型：

```
ostream& operator<<(ostream&,const classNam&);
```

如果操作符函数返回的是 istream 或 ostream 类的对象，而不是引用，会出现编译错误。原因以及具体示例可参见 8.15 节。

1.15 关于名字空间

名字空间 (namespace) 是随标准 C++ 引入的，是一种新的作用域级别。原来 C++ 标识符的作用域分成三级：代码块 ({……})，如复合语句和函数体）、类和全局。现在，在类作用域和全局作用域之间，标准 C++ 又添加了命名空间这一个作用域级别。

名字空间的引入是为了解决日益严重的名称冲突问题。随着可重用代码的增多，各种不同的代码体系中的标识符之间同名的情况就会显著增多。例如，两个部门都用 C++ 开发了一些功能组件，形成两个不同的代码库。当这两个代码库要相互调用或合并工作时，很有可能出现名称冲突的现象。解决的办法是将它们放到不同的名字空间中去，访问一个具体的标识符的时候，使用如下的形式：space_name::identifier。即用作用域指示符 “::” 将名字空间的名称和该空间下的标识符连接起来。这样，即使有同名的标识符出现，由于它们处于不同的



名字空间中，也不会发生冲突。

关于名字空间的定义和使用，要注意以下几点：

①一个名字空间可以在多个头文件或 cpp 源文件中实现。例如，标准 C++ 库中的所有组件都是在一个被称为 std 的名字空间中声明和定义的。这些组件当然分散在不同的头文件和源文件当中。

②名字空间内部可以定义类型、函数、变量等内容，但名字空间不能定义在类和函数的内部。

③在一个名字空间中可以自由地访问另外一个名字空间中的内容，因为名字空间并没有保护级别的限制。

④虽然经常可以见到 using namespace std; 这样的用法，我们也可以用同样的方式将其他名字空间中的内容一次性地“引入”到当前的名字空间中来，但这并不是一个好的用法。因为这样做相当于取消了名字空间的定义，使发生名称冲突的机会增多。所以，用 using 单独引入需要的内容，这样会更有针对性。例如，要使用标准输入对象，只需用 using std::cin; 就可以了。

⑤在名字空间的内部还可以定义另一个名字空间，这样就形成了名字空间的嵌套。

以下是一个定义和使用名字空间的例子。

```
/// Program 1.15-1 ///
```

```
/** ns1.cpp **/
```

```
#include <iostream>
```

```
namespace myspace1{
```

```
extern int gvar;
```

```
using std::cout;
```

```
using std::endl;
```

```
class myclass{
```

```
public:
```

```
    void print(){// 定义成员函数 print()，输出 gvar 的值}
```

```
    cout<<"in space1,gvar="<<gvar<<endl;
```

```
} // 定义类 myclass 结束。注意：类名与空间名不同，但空间名是类的外层空间
```

```
}
```

```
namespace myspace2{
```

```
    using std::cout;
```

```
    using std::endl;
```

```
    class myclass{ // 虽然类名也是 myclass，但并不与名字空间 myspace1 中的类发生冲突
```

```
    public:
```

```
        void print(){
```

```
            cout<<"in space2"<<endl;
```

```
}
```

```
};
```

```
namespace nestedspace{ // 定义名字空间 nestedspace，它包含一个全局变量 gvar 和一个成员函数 print()
```

```

void ExternFunc(){
    cout<<"you can reference a function within another namespace"<<endl;
}

int main()
{
    myspace1::myclass obj1;
    obj1.print();

    myspace2::myclass obj2;
    obj2.print();
}

/** end of ns1.cpp ***/

```

```

/** ns2.cpp ***/
namespace myspace1{
    int gvar=100;
}

/** end of ns2.cpp ***/
/// End of Program 1.15-1 ///

```

程序的执行结果是：

```

in space1,gvar=100
in space2
you can reference a function within another namespace

```

可以看到，名字空间 myspace1 不是一次性定义完毕，而是分段定义的，把各段的内容组合在一起形成一个名字空间。main()函数位于全局名字空间中，如果将它移到某个名字空间（例如 myspace1）中，它就会变成一个普通函数，而不能作为整个程序的入口函数。当名字空间发生嵌套时，为了访问位于“较深”的名字空间中的内容，必须连续使用作用域指示符“::”，如 main()函数中的函数调用：

```
myspace2::nestedspace::ExternFunc();
```

⑥为了避免命名空间的名字与其他的命名空间同名，可以用较长的标识符作为命名空间的名字。但是这样在引用命名空间内的标识符时就可能要重复书写较长的命名空间名，从而加重程序员的负担。所以，在特定的上下文环境可以给命名空间起一个相对简单的别名，这样引用起来就要方便得多。下面就是一个例子。

```

/// Program 1.15-2 ///
#include <iostream>
namespace MyNewlyCreatedSpace{
    void show(){
        std::cout << "a function within a namespace" << std::endl;
    }
}
```



```

    }
}

int main(){
    namespace sp = MyNewlyCreatedSpace;
    sp::show();
}

/// End of Program 1.15-2 ///

```

在main()函数中，要调用用户自定义命名空间MyNewlyCreatedSpace中的函数show()，本来是要采用如下的写法MyNewlyCreatedSpace::show()，由于给该命名空间起了一个别名sp，函数调用变成了sp::show()。

对于一些规模较小的程序，可以不定义自己的名字空间。但对于大型项目来说，使用名字空间将是一种组织和管理代码的有效手段。

在C++中，还有一种特殊的名字空间，那就是匿名名字空间。没有给出名字的名字空间，就是匿名名字空间。在一个源文件中，可以定义多个匿名名字空间。在概念上，同一源文件中的多个匿名名字空间的内容会被合并在一起形成一个统一的匿名名字空间。那么，定义匿名名字空间有什么用处呢？

在C/C++语言中，如果两个源文件中定义了相同的全局变量（或函数），就会发生重定义错误。如果将它们声明为全局静态变量（函数），就可以避免重定义错误。考察下面的程序。

```

/// Program 1.15-3 ///
/** a.cpp ***/
#include <iostream>
#include <iostream>
using namespace std;
static int gvar=8;
void show1(){
    cout << gvar << endl;
}
/** end of a.cpp **/ 

/** b.cpp ***/
#include <iostream>
using namespace std;
static int gvar=4;
void show2(){
    cout << gvar << endl;
}
int main(){
    void show1();
    show1();
}


```

```

    show2();
}

/** end of b.cpp */
/// End of Program 1.15-3 ///

```

程序的执行结果是：

```

8
4

```

本例这个程序由两个源文件组成，在a.cpp中定义了全局变量gvar，在b.cpp中也定义了一个名为gvar的全局变量。由于在两个源文件中都将它们定义为静态（static）变量，从而避免了重定义错误。

在C++中，除了可以使用static关键字避免全局变量（函数）的重定义错误，还可以通过匿名名字空间的方式实现。如下面的程序。

```
/// Program 1.15-4 ///
```

```
/** a.cpp **/
```

```
#include <iostream>
```

```
using namespace std;
```

```
namespace{
```

```
    double dvar=3.8;
```

```
    extern int gvar;
```

```
}
```

```
void show2(){
```

```
    cout << gvar << endl;
```

```
    cout << dvar << endl;
```

```
}
```

```
namespace{
```

```
    int gvar=91;
```

```
}
```

```
/** end of a.cpp **/
```

```
/** b.cpp **/
```

```
#include <iostream>
```

```
using namespace std;
```

```
namespace{
```

```
    int gvar=5;
```

```
    extern double dvar;
```

```
}
```

```
void show1(){
```

```
    cout << gvar << endl;
```

```
    cout << dvar << endl;
```

```
}
```



```

int main(){
    void show2();
    show1();
    show2();
}

namespace{
    double dvar=7.6;
}

/** end of b.cpp ***/
/// End of Program 1.15-4 ///

```

程序的执行结果是：

```

5
7.6
91
3.8

```

通过匿名名字空间，同样实现了对不同源文件中同名全局变量（函数）的保护，使它们不至于发生重定义冲突。在这一点上，匿名名字空间和static的作用是相同的。

但是，用static修饰的变量（函数）具有内部连接特性，而具有内部连接特性的变量（函数）是不能用来实例化一个模板的。见下面的例子。

```
/// End of Program 1.15-5 ///
```

```
#include <iostream>
using namespace std;
template <char *p>
```

class Example{
public:
 void display(){
 cout << *p << endl;
 }
};

static char c='a';
int main(){
 Example<&c> a;

a.display();
}

/// End of Program 1.15-5 ///

此程序不能够通过编译，因为静态变量c不具有外部连接特性，因此不是真正的“全局”变量。而类模板的非类型参数要求用常量类实例化，只有真正的全局变量的地址才能算作常量。为了实现既能保护全局变量（函数）不受重定义错误的干扰，又能够使它们具有外部连接特性的目的，必须使用匿名名字空间机制。同样是上面这个程序，修改之后如下。

```
/// End of Program 1.15-6 ///
```

```
#include <iostream>
using namespace std;
template <char *p>
class Example{
public:
    void display(){
        cout << *p << endl;
    }
};

namespace{
    char c='a';
}

int main(){
    Example<&c> a;
    a.display();
}
```

【白话】此函数（函数）是全局空间中非静态的，即没有类空间，即全局空间。此程序能正常通过编译并输出：a。由此可以看出匿名名字空间与static的区别：包含在匿名名字空间中的全局变量（函数）具有外部连接特性，而用static修饰的全局变量具有内部连接的特性，不能用来实例化模板的非类型参数。

1.16 怎样定义复杂的宏（Macro）

宏（Macro）是C语言中提供的机制，在C++中继续沿用。由于C++引入了内联函数、重载、const、模板等机制，加之C语言的宏本身所固有的缺陷（如不进行类型检查、容易产生副作用、容易带来边际效应等），在C++编程中大多数情况下可以不必使用宏。但在某些特殊的情况下，使用宏仍然可以为我们带来不少方便。例如：

①利用宏可以防止头文件在一个源文件中被包含多次。具体参见2.10节。

②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。在程序运行过程中，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）。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。

③实现一些特殊操作。例如利用内联汇编获取类的非静态成员函数的地址这样的操作，如果用函数来实现就非常困难，而使用宏来实现则方便得多（参见3.10节）。宏是用#define指令定义的，从这个意义上说，用#define指令定义的符号常量就是一种最简单的宏。但是有时我们需要定义“复杂”的宏，也就是说，宏可以带参数，宏可以使用多行代码来定义，宏的参数可以做某些特殊处理。下面分别加以说明。

定义带参数的宏，可以采用如下的格式：

```
#define MacroName(MacroParamList) MacroBody
```

其中 *MacroParamList* 是宏的参数列表，*MacroBody* 是实现宏的代码。例如：

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

在编译器对源程序做预处理时，会自动将宏调用展开成对应的实现代码，这个过程叫宏展开。在宏展开的过程中，宏的调用参数会自动替换宏定义代码中的参数。

如果宏的实现代码比较复杂，不适合在一行当中书写完毕，则可以分成多行书写，在需要续行的那一行的末尾加一个“\”，表示下一行代码仍然是宏定义的一部分。例如：

```
#define Compare(a,b) if(a>b)\n    printf("The first one is bigger.");\\\nelse\\n    printf("The first one is not bigger.");
```

需要注意的是，如果用多行来定义一个宏，那么最后一行不要使用“\”。并且，每个“\”后面不要再跟任何字符，连空格也不要，否则会引发编译错误。

尽管一个宏可以分多行定义，但是在宏展开时，被展开的宏在源程序中仍然是被书写在一行当中的。这在某些情况下会引发编译错误。例如，在3.10节中介绍了可以利用内联汇编来获取一个类的非静态成员函数的入口地址。假设A::f是类A的一个非静态成员函数，p是void*类型的指针，那么通过如下代码：

```
_asm{\n    mov eax,A::f\n    mov p, eax}
```

然后通过宏调用GetAddress(p,A::f)来获取函数A::f的地址，则会得到编译错误。原因是C++预编译会将宏调用GetAddress(p,A::f)在一行上展开，从而得到：

```
_asm{ mov eax,A::f  mov p, eax }
```

而这样的内联汇编代码是不能够通过编译的，一条独立的汇编语句只能写在单独的一行上。为了解决这个问题，可将每一条汇编语句单独放进一个_asm块中，即改写成：

```
#define GetAddress(ptr,function) _asm{\n    mov eax,function}\\n    _asm{mov ptr,eax}
```

这样问题就解决了。

宏参数本身还可以做一些特殊处理，如将宏参数置于字符串中，将宏参数与别的字符组合形成新的标识符等。以上两项功能可以分别由#和##操作符完成。见下面的程序。

```
/// Program 1.16-1 ///\n#include <iostream>\nusing std::cout;\nusing std::endl;
```

```
#define PrintVar(v,index) cout<<#v#index" is "<<v##index<<endl
int main(){
    int i1=1,i2=2,i3=3;
    PrintVar(i,1);
    PrintVar(i,2);
    PrintVar(i,3);
}
/// End of Program 1.16-1 ///
```

程序的输出结果是：

```
i1 is 1
i2 is 2
i3 is 3
```

在对PrintVar(i,1)进行宏展开时，#i#1" is "被处理成"i""1"" is "，也就是"i1 is "，而i##1则被处理成标识符i1，这样才能得到合法的C++语句。对PrintVar(i,1)和PrintVar(i,2)的宏展开过程与此相同。

1.17 explicit 的用法

explicit关键字的作用是：禁止隐式调用类的单参数的构造函数。这实际上有两种情形：一是禁止隐式调用拷贝构造函数，二是禁止类对象之间的隐式转换。类对象之间的隐式转换是指利用一个已经存在的其他类型的对象来创建本类的新对象，且不显式调用本类的构造函数。如A a1 = 37;就是一个隐转换，它利用整数37来创建一个A类对象，且不显式调用构造函数A(37)。像A(37)这样的构造函数涉及到类型转换，因此这种构造函数又被称为“转换构造函数”(Converting Constructor)。以下是一个隐式调用单参数构造函数的例子。

```
/// Program 1.17-1 ///
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){num=0;}
    A(int num){this->num=num;}
    A(const A &a){num=a.num;}
    friend void show(const A &a);
};
void show(const A &a){
    cout << a.num << endl;
}
int main()
```



```

A a1=37;
A a2=a1;
show(a1);
show(a2);
show(47);
}

```

//// End of Program 1.17-1 ////

程序的输出结果是：

```

37
37
47

```

程序中有三条语句，`A a1=37;`和`A a2=a1;`以及`show(47);`隐式调用了类A的单参数的构造函数。这种隐式调用在C++语法当中是允许的，但很多人对这种表示方式不习惯，觉得程序的可读性较差。

为了禁止对类的单参数构造函数的隐式调用，C++引入了关键字`explicit`。在类的定义中，在任何一个单参数构造函数前加`explicit`，就可以禁止对该构造函数的隐式调用。考察下面的程序。

//// Program 1.17-2 ////

```

#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){num=0;}
    explicit A(int num){this->num=num;}
    explicit A(const A &a){num=a.num;}
    friend void show(const A &a);
};
void show(const A &a){
    cout << a.num << endl;
}
int main(){
    A a1=37;
    A a2=a1;
    show(a1);
    show(a2);
    show(47);
}

```

//// End of Program 1.17-2 ////

该程序由于将类A的所有单参数构造函数声明为`explicit`，从而导致了3个编译错误。为了

能让程序正常运行，必须将

```
A a1=37;
```

```
A a2=a1;
```

```
show(47);
```

分别改写成：

```
A a1(37);
```

```
A a2(a1);
```

```
show(A(47));
```

这样程序就可以正常运行，并具有更好的可读性。

使用**explicit**关键字，要注意以下几点：

- ①**explicit**关键字只需用于单参数的构造函数前面。由于无参数的构造函数和多参数（参数个数为2或2以上）的构造函数总是显式调用的，所以在这种构造函数前加**explicit**没有意义。
- ②如果想禁止类A对象被隐式转换为类B对象，可在类B中使用关键字**explicit**，即定义这样的转换构造函数**explicit B (A a){...}**，或者**explicit B (const A &a){...}**。

第2章 | 数据类型与程序结构

2.1 C++的数据类型

C++是一种强类型语言。C++程序中的任何变量（或函数）必须遵循“先说明后使用”的原则。定义数据类型有两个方面的作用：一是决定该类型的数据在内存中如何存储，二是决定可对该类型的数据进行哪些合法的运算。

C++的数据类型分为基本数据类型和非基本数据类型。其中非基本数据类型称为复合数据类型或构造数据类型。为了能够体现C++语言与传统C语言在非基本数据类型上的区别，在这里把能够体现面向对象特性的非基本数据类型称为构造数据类型，而将其他非基本数据类型称为复合数据类型。C++的数据类型如图2-1所示。

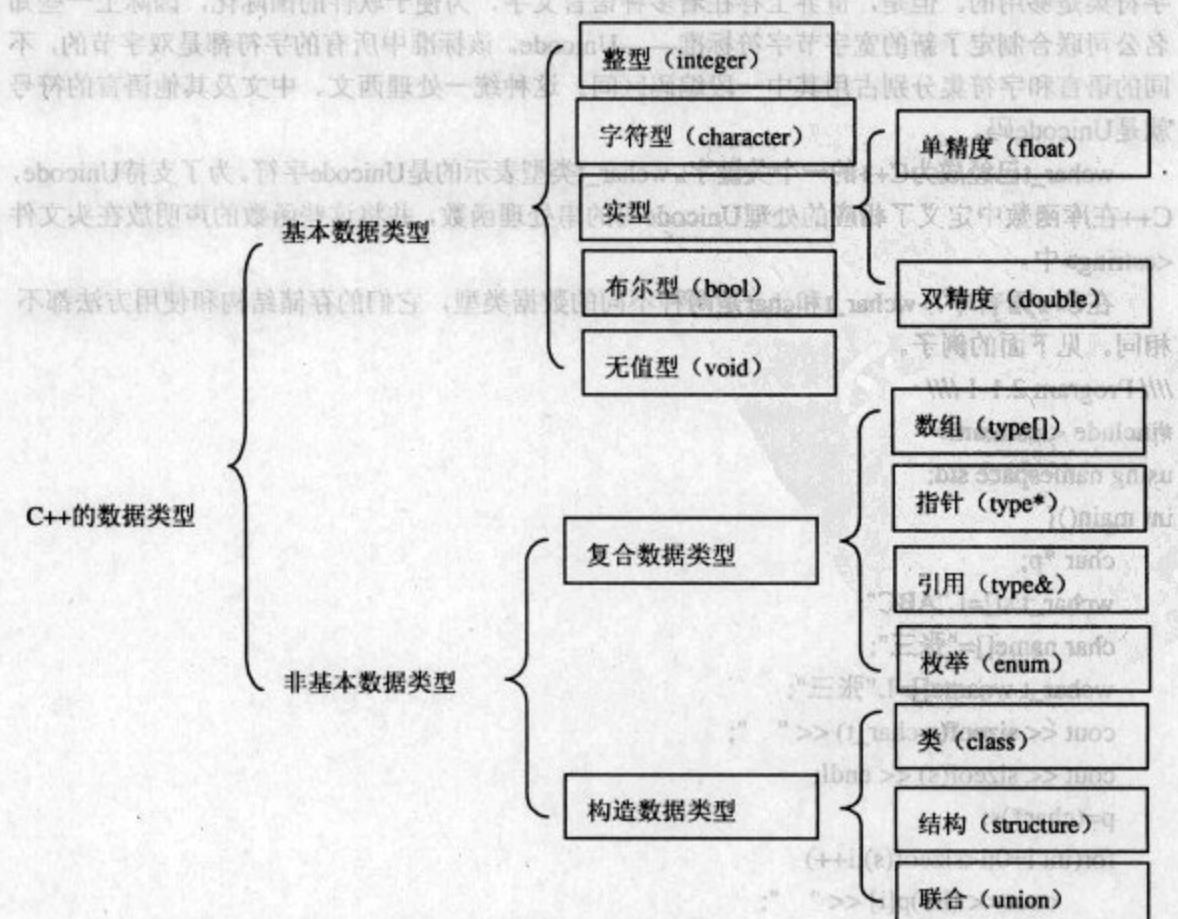


图 2-1 C++的数据类型



基本数据类型是C++内部预定义的，又叫做内置（built-in）数据类型。非基本数据类型则是用户根据需要按照C++语法规则创建的数据类型。在这里，构造数据类型和复合数据类型的区别在于：构造数据类型的实例叫做对象，它是属性和方法的集合。真正的构造数据类型是由C++语言引入的，它体现了面向对象的程序设计思想。构造数据类型的一个显著特征是：在生成该数据类型的一个实例的时候，会自动调用该类型定义的构造函数。也就是说，构造数据类型的变量的初始化工作是由构造函数完成的。

注意：用基本数据类型定义变量时，类型出现在前面，变量直接跟在类型之后。但用复合数据类型定义变量时，变量却不一定完全位于类型之后。例如，定义一个数组int a[8]，标识符a的数据类型是int[8]，但它出现在数据类型的中间部位。另外，定义或声明变量时，类型外一定不能加括号，例如用这种方式定义一个指针是不对的：(int*) p;，它表达的真实含义是将p转换为int*类型，是强制类型转换的语法形式。

传统的字符型char是单字符型，存储的是该字符的ASCII码，占用一个字节。也可以把char理解成单字节整型，它的取值范围是-128~127。而单字节无符号整数可以用unsigned char表示，取值范围是0~255。如果在一个字符串中包含汉字，那么每个汉字占用2字节，每字节的最高位都是1。单个字符变量是无法容纳一个汉字字符的，如定义char c='好';将得到一条编译警告信息，并且只有低字节编码会存放在字符变量c中。

C++语言同时也支持宽字符类型（wchar_t），或称为双字节字符型。用ASCII码表示英文字符集是够用的。但是，世界上存在着多种语言文字，为便于软件的国际化，国际上一些知名公司联合制定了新的宽字节字符标准——Unicode。该标准中所有的字符都是双字节的，不同的语言和字符集分别占用其中一段编码区间。这种统一处理西文、中文及其他语言的符号就是Unicode码。

wchar_t已经成为C++的一个关键字。wchar_t类型表示的是Unicode字符。为了支持Unicode，C++在库函数中定义了相应的处理Unicode码的串处理函数，并将这些函数的声明放在头文件<cstring>中。

在C++语言中，wchar_t和char是两种不同的数据类型，它们的存储结构和使用方法都不相同。见下面的例子。

```
/// Program 2.1-1 ///
#include <iostream>
using namespace std;
int main(){
    char *p;
    wchar_t s[] = L"ABC";
    char name[] = "张三";
    wchar_t wname[] = L"张三";
    cout << sizeof(wchar_t) << " ";
    cout << sizeof(s) << endl;
    p = (char*)s;
    for(int i=0; i<sizeof(s); i++)
        cout << (int)p[i] << " ";
    cout << endl;
```

```

cout << s << " ";
wcout << s << endl;
for(int i=0;i<sizeof(name);i++)
    cout << (int)name[i] << " ";
cout << endl;
p=(char*)wname;
for(int i=0;i<sizeof(wname);i++)
    cout << (int)p[i] << " ";
cout << endl;
cout << name << endl;
wcout << wname << endl;
}

```

//// End of Program 2.1-1 ////

程序的输出结果是：

```

2 8
65 0 66 0 67 0 0 0
0012FF4C ABC
-43 -59 -56 -3 0
32 95 9 78 0 0
张三

```

研究这段程序，要注意这样几个要点：

①wchar_t和char的数据宽度是不一样的，即sizeof(wchar_t)==2而sizeof(char)==1。

②定义一个wchar_t类型的字符串常量时，要以L开头，例如 wchar_t s[] = L"ABC"，如果去掉起始的L，就会出现编译错误。定义一个wchar_t类型的字符常量时，也应以L开头，例如 wchar_t c=L'A'，如果去掉起始的L，编译器会执行由char到wchar_t的转换。

③对于西文字符（如'A'、'B'、'C'等）来说，在wchar_t型的变量中，高字节存放的是0X00，低字节存放的是西文字符的ASCII码。

④char类型的字符串以单字节'\0'结束，wchar_t类型的字符串以双字节'\0\0'结束。

⑤在char类型的字符串中，一个汉字字符用两个字节表示，这两个字节的最高位都是1，只有这样才能将它们与西文字符区分开来，所以将它们的ASCII码输出时得到两个负数。在 wchar_t类型的字符串中，每个字符（包括汉字字符）都用双字节表示，只不过这两个字符的最高位没有必要为1。

⑥要想直接将wchar_t类型的字符或字符串向标准输出设备输出，必须使用对象wcout，它在头文件iostream中声明。如果将wchar_t类型的字符向cout输出，得到的是将该字符作为双字节整数时的值；如果将wchar_t类型的字符串向cout输出，打印的是该字符串在内存的首地址（字符型指针的值）。

⑦在上面的程序中，语句cout << name << endl;的输出结果是“张三”，而语句wcout << wname << endl;却无法看到输出。如果字符串wname中全是西文字符，则仍然可以看到输出。这是在控制台程序中的一个现象，与控制台的缺省语言环境的设置有关。为了能够更好地支持多种语言符号的显示，IO流类库依靠一个locale类的对象来设置语言环境。为了能够显示汉



字，必须将控制台语言环境设置成显示中文，即，使用如下的语句：

```
std::wcout.imbue(std::locale("chs"));
```

然后再用wcout输出含有中文字符的宽字符串，就不会出现显示问题了。有关locale的更详细的介绍，请参阅9.6节。

有时，需要在char类型的字符串和wchar_t类型的字符串之间进行转换。这同样涉及到语言环境的设置问题。具体做法参见9.7节。

2.2 C++中的布尔类型

bool是C++中表示布尔类型的关键字。与其他严格限制布尔类型的使用场合的语言（如C#等）相比，C++将布尔类型视为只能在0和1当中取值的整数类型。因此，下面的程序能顺利通过编译，不会产生任何编译警告、编译错误或运行时错误。

```
//// Program 2.2-1 /////
#include <iostream>
using namespace std;
int main(){
    bool bvar;
    int i=5,j=5;
    bvar=i==j;
    cout << bvar << endl;
    cout << bvar+i << endl;
}
//// End of Program 2.2-1 /////

```

程序的执行结果是：

```
1
6
```

在表达式bvar+i中，布尔变量bvar是当作整数1来对待的。

如果显式地将一个整型量的值赋给布尔变量，编译器会给出一个警告。并且，一定要清楚：布尔变量的值只可能是0或1。见下面的例子。

```
//// Program 2.2-2 /////
#include <iostream>
using namespace std;
int main(){
    bool bvar;
    int i=5;
    bvar = i;
    cout << bvar+i << endl;
}
//// End of Program 2.2-2 /////

```

此程序在编译时，会得到一条警告信息：



warning C4800: “int”：将值强制为布尔值“true”或“false”(性能警告)

要注意语句**bvar=i;**的执行结果，变量bvar的值变为1（而不是5！），这样程序的输出结果就是6。

在if语句、while语句等需要布尔表达式的场合，任何基本数据类型的表达式（如char、int、double等）和指针变量都可以隐式地转换为布尔量，**if(a)**和**if(a!=0)**是等价的。但是对于像double等表示实数的类型，直接使用**if(a)**这样的表达式显然是不合适的，这样会造成计算的不稳定。

2.3 void 的用法

void是C++语言的一个关键字，表示“无类型”的意思。有趣的是，**void**的出现位置却经常是数据类型应该出现的位置。

void几乎只有“注释”和限制程序的作用。由于**void**不是一种数据类型，所以不能用它来定义一个变量。如果在程序中定义：

```
void a;
```

编译的时候会得到出错信息：非法使用“**void**”类型。不过，即使**void a**的编译不会出错，它也没有任何实际意义。

void可以用在如下几种场合：

(1) 对函数返回值的限定

如果一个函数没有返回值，应在定义（或声明）函数时使用**void**对其进行修饰。例如：

```
void f(int);
```

这个函数接受一个整型变量作为它的参数，它没有返回值。

没有返回值的函数一般不使用**return**语句。如果有需要，可以使用**return**；提前退出函数。也就是说，**return**后面不能写返回值。

在定义（或声明）一个函数时，如果不声明返回值，并不代表该函数没有返回值。如定义函数：

```
Add(int x,int y){.....}
```

在C编译器中，会认为函数Add()的返回值为int。而C++的类型检查更为严格，较新的C++编译器会拒绝这种写法，要求程序员必须显式写出函数的返回值类型。如果没有返回值，就必须显式使用**void**。

这样，在C++程序中，任何函数都必须指定返回类型。如果函数没有返回值，一定要声明为**void**类型。这样的程序具有良好的可读性，充分发挥代码的“自注释”功用。

在定义一个类时，有三种函数是不必（也不允许）书写返回值类型的，它们是类的构造函数、析构函数和类型转换操作符函数（参见2.6节）。用**void**修饰这三种函数会导致编译错误。

(2) 对函数参数的限定

在C语言中，如果在定义（或声明）一个函数时，不给出任何参数的说明，如定义函数：
int f0(...)

那么该函数实际上可以接收任意数量的参数。如果要定义（或声明）一个不接收任何参数的函数，则必须使用**void**。如：

```
int f(void){...}
```

则函数f()不接收任何参数，类似f(1)这样的调用都会导致编译错误。

在C++语言中，定义（或声明）一个函数时，不给出任何参数说明，或者使用void声明参数，表示同样的语义，即该函数不接收任何参数。如int f()和int f(void)都表示函数f()不接收任何参数。在这一点上，C++与C语言是有差异的。

所以，如果指明参数为void，则无论在C还是C++中，都代表函数不接受任何参数。

(3) 使用void*类型的指针

指针代表的是内存中某处的地址。一般定义一个指针，除了说明指针变量中存储的值的意义（即代表地址）之外，还要同时说明该地址处存放的数据（变量）的类型。指向不同类型变量的指针之间不能赋值。例如下面的程序不能通过编译。

```
/// Program 2.3-1 ///
#include <iostream>
using namespace std;
int main(){
    int i=9;
    float *p=&i;
    cout << *p << endl;
}
/// End of Program 2.3-1 ///
```

因为表达式&i的数据类型是int *，所以它只能赋给int*类型的指针，而不能赋给float*类型的指针。编译时的出错信息是：无法从“int *__w64”转换为“float *”。如果一定要完成这样的操作，则必须使用强制类型转换。修改后的程序如下。

```
/// Program 2.3-2 ///
#include <iostream>
using namespace std;
int main(){
    int i=9;
    float *p=(float *)&i;
    cout << *p << endl;
}
/// End of Program 2.3-2 ///
```

此程序能够通过编译，输出结果是1.26117e-044。可见指针之间的强制转换意味着对同一地址处的数据做出不同的解释，因此要格外小心。

C++语言中允许定义void *类型的指针，它“纯粹”代表一个地址，而不对该地址处的内容作任何解释。因此，任何类型的指针都可以直接赋值给它，无需进行强制类型转换。

反过来，void *类型的指针却不能直接赋给其他类型的指针，而必须使用强制类型转换。

按照ANSI(American National Standards Institute)标准，不能对void指针进行加减运算，即下列操作都是不合法的：

```
void * pvoid;
pvoid++; //ANSI: 错误
pvoid += 1; //ANSI: 错误
```



ANSI标准之所以这样规定，是因为它认为，进行加减运算的指针必须知道其指向数据类型的大小。例如：

```
int *pint;
pint++; //ANSI: 正确
pint++ 的结果是使指针pint的值增大sizeof(int)。
```

实际上，改变void*类型指针的值并不是一件十分困难的事情，只要将它先转换成char*类型的指针，然后再转换回void*类型就可以了。见下面的程序。

```
/// Program 2.3-3 ///
#include <iostream>
using namespace std;
int main(){
    char str[]="abc";
    void *p=str;
    cout << str << endl;
    cout << p << endl;
    p = (void*)((char*)p+1);
    cout << p << endl;
}
/// End of Program 2.3-3 ///
```

此程序的运行结果是：

```
abc
0012FF60
0012FF61
```

因此，尽管有些编译器可能允许把void *的算法操作与char *等同起来，但是为了提高程序的可移植性，我们还是不应该直接在void*类型的指针上直接实施算法操作。在上面的程序中可以看到，cout << str << endl;输出的结果是字符串的值，而cout << p << endl;的输出结果是指针变量中存储的地址值。所以，如果想确保查看到一个指针所代表的地址值，应该将其转换成void*指针后再输出。

在C/C++提供的标准库中，有两个典型的使用void*指针进行操作的函数：

```
void * memcpy(void *dest, const void *src, size_t len);
void * memset ( void * buffer, int c, size_t num );
```

任何类型的指针都可以传入memcpy和memset中，这也真实地体现了内存操作函数的意义，因为它操作的对象仅仅是一片内存，而不论这片内存是什么类型。下面是一个利用这两个函数完成数组和对象的初始化工作的例子。

```
/// Program 2.3-4 ///
#include <iostream>
using namespace std;
class A{
    int arr[2];
    int i;
```



```
double d;
public:
A(){
    memset(this,0,sizeof(A));
}
void set(int,int,int,double);
A(const A &a){
    memcpy(this,&a,sizeof(A));
}
void show(){
    cout << arr[0] << " " << arr[1] << " " << i << " " << d << endl;
}
void A::set(int a0,int a1,int iv,double dv){
    arr[0]=a0;
    arr[1]=a1;
    i = iv;
    d = dv;
}
int main(){
    A obj;
    obj.show();
    obj.set(1,2,3,4.8);
    A another=obj;
    another.show();
}
/// End of Program 2.3-4 ///
```

程序的运行结果是：

0 0 0

1 2 3 4.8

函数memset()用来将一片连续的内存区域中的每个字节置为某个特定的值，而memcpy()则可以将一片连续的内存区域拷贝到另一块同等大小的区域中去。可以认为使用memcpy()是实现类的默认拷贝构造函数的一种可行方案。同时可以看出，如果类A中存在只读成员变量，函数memset()和memcpy()的执行会改变该只读成员变量的值。

2.4 枚举类型的定义和使用

枚举类型的主要作用是提高程序的可读性。定义一个枚举类型之后，相当于提供了若干代表整数的符号常量。这些符号常量通常具有明显的字面意义，可以在枚举类型的有效作用域内自由使用。例如下面的程序。

```

/// Program 2.4-1 ///
#include <iostream>
using namespace std;
enum weekDay{Sun,Mon,Tue,Wed,Thu,Fri,Sat};
int main(){
    weekDay day;
    cout << Sun << " ";
    cout << Mon << " ";
    cout << Tue << " ";
    cout << Wed << " ";
    cout << Thu << " ";
    cout << Fri << " ";
    cout << Sat << endl;
    day = Sat;
    cout << day << endl;
}
/// End of Program 2.4-1 ///

```

程序的运行结果是：

```

0 1 2 3 4 5 6
6

```

枚举常量代表一个整数，通常按照在类型声明中出现的次序，从0开始编号。在使用枚举常量时，只需直接使用它的标识符即可，并不需要在枚举常量前面给出枚举类型的名字。被声明为枚举类型的变量（如程序中的day），只能将枚举常量赋给它，而不能直接将整数值赋给它。枚举变量在输出时，直接输出它的值所对应的整数值。如果用sizeof运算符考察枚举变量的长度，会发现其值为4，说明枚举常量是32位的整数（int类型）。

可以在定义枚举类型时显式地指明某个枚举常量的整数值（可以是负数、0或正数），没有显式指明值的枚举常量，其值为前一个枚举常量的值加1。枚举类型可以定义在文件作用域内，也可以定义在类作用域内（即某个类的类体内部）。考察下面的程序。

```

/// Program 2.4-2 ///
#include <iostream>
using std::cout;
using std::endl;
class someClass{
public:
    static const int count=4;
    enum weekDay{Mon=2,Tue,Wed,Thu=100,Fri,Sat,Sun};
    typedef int intarray[7];
};
int main(){
    someClass::intarray ia;
}

```

```

ia[0]=someClass::Mon;
ia[1]=someClass::Tue;
ia[2]=someClass::Wed;
ia[3]=someClass::Thu;
ia[4]=someClass::Fri;
ia[5]=someClass::Sat;
ia[6]=someClass::Sun;
cout<<ia[0]<<" "<<ia[1]<<" "<<ia[2]<<" "<<ia[3]<<" "<<ia[4]<<" "<<ia[5]<<
" "<<ia[6]<<endl;
cout << someClass::coint << endl;
}
/// End of Program 2.4-2 ///

```

程序的输出结果是：

2 3 4 100 101 102 103

4

阅读上面这段程序要注意以下几个要点：

- ①静态常变量coint的初始化设定可以在类comeClass的类体内进行，详细介绍参见4.9节。
- ②类型weekDay和intarray都是在类comeClass内部定义的，如果它们的访问限定符是private，则只能在该类的成员函数中使用。
- ③因为Thu代表的整数是100，所以Fri的值是Thu的值再加1（即101），其余枚举常量的值依此类推。
- ④枚举常量的作用类似于类中的静态整型常变量，但要注意它们只是符号常量，是不可寻址的，在编译阶段就已经替换为相应的整数值。

在一个作用域内可以定义多个枚举类型，这样，在这个作用域内就会存在多个枚举常量。允许两个枚举常量对应相同的整数值，但不允许任何两个枚举常量同名，否则会出现编译错误。若想避免在全局作用域中定义枚举类型可能带来的枚举常量和其他标识符之间的冲突，可以将枚举类型定义在特定的名字空间或类中。如果仅仅是想使用枚举常量，而并不需要定义枚举变量，则在定义枚举类型时枚举类型名也可以省略。

2.5 结构与联合体

结构（struct）与联合体（union）是C语言中就已经存在的数据类型。C++语言对它们进行了扩展，最大的变化是允许在结构和联合体中定义成员函数。以下是一个使用了结构的C++程序。

```

/// Program 2.5-1 ///
#include <iostream>
using namespace std;
struct Room{
    int floor;
    int No;
}

```



```

};

struct Student{
    int age;
    int score;
    Student(int a,int s){
        age=a;
        score=s;
    }
};

int main(){
    Room r[3]={ {1,101}, {2,201}, {3,301} };
    Student s(18,80);
    cout << "The rooms are:" ;
    cout << r[0].floor << "-" << r[0].No << ",";
    cout << r[1].floor << "-" << r[1].No << ",";
    cout << r[2].floor << "-" << r[2].No << endl;
    cout << "The student is " << s.age;
    cout << " years old and he's score is " << s.score << endl;
}

```

//// End of Program 2.5-1 ////

程序的执行结果是：

The rooms are:1-101,2-201,3-301

The student is 18 years old and he's score is 80

在C++中使用结构（struct）要注意这样几点：

①在C++中，结构是一种真正的数据类型，所以在利用结构定义变量时，不需要像在C语言中那样带上struct关键字，或先使用typedef struct structname structname的方式进行声明。

②C++对C语言当中的struct进行了扩展，允许在struct中定义成员函数。struct中的成员变量和成员函数也有访问权限。在class中，默认的访问权限是private，而在struct中，默认的访问权限是public。这是结构和类的唯一差别，在其他方面，二者是相同的。将struct成员的默认访问权限设定为public，是C++保持与C语言兼容而采取的一项策略。

③如果在struct中没有显式定义任何构造函数，那么结构变量可以像在C语言中那样用花括号中顺序指明数据成员的值的办法进行初始化，如上例中的数组r。而一旦显式定义了任何一个构造函数，就不能用这种方式初始化了。例如，在上例中将变量s的初始化语句写成：Student s={18,80};就会报告编译错误。同样地，如果在class中只有若干public型的数据成员，而没有显式定义任何构造函数，也可以利用花括号进行初始化。

④用sizeof()运算符计算结构的大小时，要考虑到结构体内部变量对齐的问题，详细介绍请参阅1.7节。

联合（union）是一种特殊的类，很少用但很有用。它是从C语言中继承而来的，其基本语义没有发生变化，只是具有了类的特性而已（允许定义成员函数）。与结构不同的是，联合中的数据成员共享同一段内存，以达到节省空间的目的。通过下面的例子，可以考察union

变量的体积是怎样决定的，为union变量的成员赋值，会怎样改变其他成员的值。

```
/// Program 2.5-2 ///
#include <iostream>
using namespace std;
union testunion{
    char c;
    int i;
};
int main(){
    cout << sizeof(testunion) << endl;
    testunion *pt= new testunion;
    char *p= reinterpret_cast<char*>(pt);
    for(int i=0;i<sizeof(*pt);i++){
        cout << int(p[i]) << " ";
    }
    cout << endl;
    cout << pt->i << endl;
    pt->c='A';
    cout << pt->c << endl;
    for(int i=0;i<sizeof(*pt);i++){
        cout << int(p[i]) << " ";
    }
    cout << endl;
    cout << pt->i << endl;
    delete pt;
}
/// End of Program 2.5-2 ///
```

程序的执行结果是：

```
4
-51 -51 -51 -51
-842150451
```

```
A
65 -51 -51 -51
-842150591
```

可以看到，union testunion变量的体积为4，它是由其两个数据成员中体积较大的一个（int类型）来决定的。对其中一个数据成员的修改，一定会同时改变所有其他数据成员的值。不过，对其中体积较小的数据成员的修改，只会影响到该成员应该占据的那些字节，对超出部分（高位字节）没有影响。在上面的例子中，对pt->c的修改，只会改变*pt中的第一个字节的值，其他3个字节的值不受影响。

下面是另一个使用union的例子。



```
/// Program 2.5-3 ///
#include <iostream>
using namespace std;
class someClass{
    int num;
public:
    void show(){
        cout << num << endl;
    }
};

union A{
    char c;
    int i;
    double d;
    someClass s;
};

union B{
    char c;
    int i;
    double d;
    B(){
        d=8.9;
    }
};

union{
    char c;
    int i;
    double d;
    void show(){
        cout << c << endl;
    }
}u={'U'};

int main(){
    A a={'A'};
    B b;

    cout << a.c << endl;
    cout << b.d << endl;
    a.s.show();
    u.show();
}
```

```

union{           //定义一个匿名联合
    int p;
    int q;
};

p=3;
cout << q << endl;
}

```

//// End of Program 2.5-3 ////

程序的执行结果是：

A
8.9
65
U
3

阅读以上程序，请注意这样几个要点：

①union 可以指定成员的访问权限，默认情况下，与 struct 具有一样的权限（public）。

②union 也可以定义成员函数，包括构造函数和析构函数。但是，它不能作为基类使用，成员函数不能为虚函数。union 也不能有静态数据成员或引用成员，因为静态数据成员实际上并不是联合体的数据成员，它无法和联合体的其他数据成员共享空间，而引用本质上是一个指针常量，它的值一旦初始化就不允许修改。如果允许联合体有引用成员，那么联合体对象一创建就无法修改，只能作为一个普通的引用使用，这样就失去了联合体存在的意义。

③union 允许其他类的对象成为它的数据成员，但是要求该对象的所属类不能定义构造函数、析构函数或者赋值操作符函数。union 的数据成员若为对象，那么它与 union 的其他数据成员共享内存，union 的构造函数在执行的时候，不能调用该对象的构造函数（否则也改变了 union 的其他数据成员的值）。同样，调用 union 的对象成员的析构函数也是没有意义的，因为其他数据成员的有意义的值对于对象成员而言可能毫无意义。还是由于内存共享的原因，union 的对象成员的赋值操作应该维持其原始语义，而不要进行重载。因为赋值操作符的重载一般使用在对象的“深拷贝”等场合，而在对象空间与其他变量共享的情况下，“深拷贝”等操作的安全性是无法得到保障的。

④默认情况下 union 对象是未初始化的。如果 union 没有显式定义任何构造函数，就可以采用 C-style 初始化方法显式初始化 union 对象。但是，只能为第一个成员提供初始化式。如上例中的变量 a 的初始化。

⑤由于 union 对象内部成员共用一段空间，如果使用不当可能产生错误，如错误地使用当前有效成员。为了避免这类错误，往往把 union 嵌套在一个类中，该类附加一个枚举成员，用来指明 union 成员中当前哪个成员有效。当然，带来安全的同时，也带来了额外的开销。因为每次使用时都要进行判断，特别是当 union 内成员较多时，就显得十分繁琐。

⑥如果一个 union 类型只在定义该类型的同时使用一次，以后就再不使用了，那么也可以不给出 union 的名称。如上例中的变量 u 就是这种情况。u 的类型是一个 union，但该 union 类型的作用只是定义变量 u，以后就不再使用了，所以可以不给出类型名。只不过在这种情况下，无法为该 union 定义构造函数。



⑦还可以定义匿名联合（anonymous union），也就是给出一个不带名称的联合体的声明后，并不定义任何该 union 的变量，而是直接以分号结尾。如上面程序中的语句 u.show()之后定义的那个 union。严格地说，匿名联合并不是一种数据类型，因为它不用来定义变量，它只是指明若干个变量共享一片存储区域。例如在上例中，对变量 p 的修改实际上就是修改了变量 q，因为这两个变量共享一片存储单元。可以看到，匿名联合中的变量，尽管被定义在一个联合声明中，它们与同一个程序块中的任何其他局部变量具有相同的作用域级别。这意味着匿名联合内的成员的名称不能与同一个作用域内的其他标志符相冲突。另外，对匿名联合还存在如下限制：包含在匿名联合内的元素必须是数据，而不允许有成员函数；匿名联合也不能包含私有或者保护成员；最后，全局匿名联合中的成员必须是全局或静态变量。

2.6 数据类型转换

虽然有可能带来风险，但数据类型转换仍是编程中经常需要进行的一个操作。先来看一个例子。

```
/// Program 2.6-1 ///
#include <iostream>
using namespace std;
int main(){
    int i;
    double d=4.88;
    i=d;
    cout << i << endl;
}
/// End of Program 2.6-1 ///
```

在 VC++ 2005 下编译，会得到一条警告信息：warning C4244：“=”：从“double”转换到“int”，可能丢失数据。但程序还是能通过编译，并输出：4。

数据类型转换（如程序中的 i=d;）到底做了些什么事情呢？实际上，数据类型转换的工作相当于一条函数调用，若有一个函数专门负责从 double 类型到 int 类型数据的转换（假设函数名为 dtoi），则上面那条语句等价于 i=dtoi(d)。函数 dtoi 的原型应该是：int dtoi(double) 或者 int dtoi(const double&)。

有些类型的数据转换是绝对安全的，所以可以自动进行，编译器不会给出任何警告，如由 int 型转换成 double 型。另一些转换会造成数据丢失，但编译器只是给出警告，并不算一个语法错误，如上面的例子，由 double 型转换成 int 型就属于这种情况。各种基本数据类型（不包括 void）之间的转换基本都属于以上两种情况。

如果考虑复合数据类型和其他数据类型的转换，情况就会复杂一些。考察下面的程序。

```
/// Program 2.6-2 ///
#include <iostream>
using namespace std;
int main(){
    short arr[]={65,66,67,0};
```

```
wchar_t *s;  
s = arr;  
wcout << s << endl;  
}  
/// End of Program 2.6-2 ///
```

由于 short int 和 wchar_t 是不同的数据类型，直接把 arr 代表的地址赋给 s 会导致一个编译错误：error C2440：“=”：无法从“short [4]”转换为“wchar_t *”。

为了解决这种“跨度较大”的数据类型转换，可以使用“强制类型转换”机制，把语句 s=arr; 改写成 s=(wchar_t*)arr; 就能顺利通过编译，并输出：ABC。

强制类型转换在 C 语言中就已经存在，到了 C++ 语言中可以继续使用。在 C 风格的强制类型转换中，目标数据类型被放在一对圆括号中，然后置于源数据类型的表达式前。在 C++ 语言中，允许将目标数据类型当作一个函数来使用，将源数据类型表达式置于一对圆括号中，这就是所谓“函数风格”的强制类型转换。以上两种强制转换没有本质区别，只是书写形式上略有不同，即：

(T) expression // C-style cast

T(expression) // function-style cast

可将它们称为旧风格的强制类型转换。在上面的程序中，可用以下两种书写形式实现强制类型转换：

```
s = (wchar_t*) arr;
```

```
typedef wchar_t* LS PTR; s = LS PTR(arr);
```

思考：如果这样进行类型转换 s=wchar_t *(arr)，是否可行？为什么？

在 C++ 语言中，增加了四种内置的类型转换操作符：const_cast、static_cast、dynamic_cast 和 reinterpret_cast。它们具有统一的语法形式：type_cast_operator<type>(expression)。下面分别介绍。

(1) const_cast 主要用于解除常指针和常量引用的 const 属性。也就是说，把 const type * 类型转换成 type * 类型或将 const type & 转换成 type & 类型。详细用法和示例参见 1.4 节。

(2) static_cast 相当于传统的 C 语言中那些“较为合理”的强制类型转换，较多地使用于基本数据类型之间的转换、基类对象指针（或引用）和派生类对象指针（或引用）之间的转换、一般的指针和 void* 类型的指针之间的转换等。static_cast 操作对于类型转换的合理性会作出检查，对于一些过于“无理”的转换会加以拒绝。例如下面的转换：

```
double d=3.14;  
double *p=static_cast<double*>(d);
```

就是一种非常怪异的转换，在编译时会遭到拒绝。

另外，对于一些看似合理的转换，也可能被 static_cast 拒绝。这时要考虑别的办法。如下面的程序。

```
/// Program 2.6-3 ///  
#include <iostream>  
using namespace std;  
class A{  
    char ch;
```

```

int n;
public:
    A(char c,int i):ch(c),n(i){}
};

int main(){
    A a('s',2);
    char *p = static_cast<char*>(&a);
    cout << *p << endl;
}

/// End of Program 2.6-3 ///

```

这个程序无法通过编译，得到的错误信息是：error C2440：“static_cast”：无法从“A * __w64”转换为“char *”。就是说，直接将A*类型转换为char*类型是不允许的，这时可以通过void*类型为中介实现转换。修改后的程序如下。

```

/// Program 2.6-4 ///
#include <iostream>
using namespace std;
class A{
    char ch;
    int n;
public:
    A(char c,int i):ch(c),n(i){}
};

int main(){
    A a('s',2);
    void *q = &a;
    char *p = static_cast<char*>(q);
    cout << *p << endl;
}

/// End of Program 2.6-4 ///

```

首先将&a的值保存在void*型指针q中，然后将q的值传递给char*型指针p。程序的输出结果是：s。可见，如果指针类型之间进行转换，一定要注意转换的合理性，这一点必须由程序员自己负责。指针类型的转换意味着对原数据实体的内容重新作出解释。

在实践中，static_cast多用于类类型之间的转换。这时，被转换的两种类类型之间一定要存在派生与继承的关系，否则，在两个互不相关的类之间用static_cast进行转换，一定会报编译错误。如下面的程序。

```

/// Program 2.6-5 ///
#include <iostream>
using namespace std;
class A{};
class B{};

```



```
int main(){
    A *pa;
    B *pb;
    A a;
    pa = &a;
    pb = static_cast<B*>(pa);
}
```

//// End of Program 2.6-5 ////

该程序无法通过编译，原因是类 A 与类 B 没有任何关系。

综上所述，使用 static_cast 进行类型转换时要注意如下几点：

① static_cast 操作符的语法形式是：static_cast<type>(expression)，其中 expression 外面的圆括号不能省略，哪怕 expression 是一个简单变量也是这样。

② 通过 static_cast 操作符，只能进行一些相关类型之间的“合理”的转换。如果是进行类类型之间的转换，源类型和目标类型之间必须存在继承关系，否则会得到编译错误。

③ static_cast 所进行的转换是一种静态转换，是在编译时决定的。通过编译后，空间和时间效率实际上等价于 C 方式强制转换。

④ 在 C++ 语言中，指向派生类对象的指针可以隐式地转换为指向基类对象的指针。而要把指向基类对象的指针转换为指向派生类对象的指针，就必须借助 static_cast 操作来完成，而且必须由程序员自己来承担这种转换带来的风险。

既然用 static_cast 将指向基类对象的指针转换为指向派生类对象的指针有时是不安全的，那么为什么不干脆禁止这种转换呢？原因是，既然指向派生类对象的指针可以隐式地转换为指向基类对象的指针，那么也应该允许在适当的时候再转换回来，以恢复指针所指对象的“本来面目”。

在满足一定条件的情况下，基类对象指针（或引用）与派生类对象指针（或引用）之间的转换还可以通过 dynamic_cast 来实现，具体细节参见 8.6 节。

(3) reinterpret_cast 是一种最为“狂野”的转换，它在 4 种转换操作中能力是最强的。例如，在 Program 2.6-3 中，语句 char *p = static_cast<char*>(&a); 会引发编译错误，但如果把它改为 char *p = reinterpret_cast<char*>(&a); 就可以直接通过编译，由此也可以看出 reinterpret_cast 的能力之“强大”。

reinterpret_cast 是 C++ 语言中最不提倡使用的一种类型转换操作符，应该尽量避免使用。不过，在一些特殊的场合，在确保安全性的情况下，也可以适当使用它。函数指针类型之间的转换便是一例。考察下面的程序。

```
//// Program 2.6-6 /////
#include <iostream>
using std::cout;
using std::endl;
typedef void (*pfunc)();
void func1()
{
    cout << "this is func1(), return void" << endl;
```

```

}

int func2()
{
    cout<<"this is func2(),return int"<<endl;
    return 1;
}
int main()
{
    pfunc FuncArray[2];
    FuncArray[0]=func1;
    FuncArray[1]=reinterpret_cast<pfunc>(func2);
    for(int i=0;i<2;i++)
        (*FuncArray[i])();
}
/// End of Program 2.6-6 ///

```

程序的输出结果是：

```

this is func1(),return void
this is func2(),return int

```

由函数指针类型 `int (*)()` 转换为 `void (*)()`，只能通过 `reinterpret_cast` 进行，用其他的类型转换方式都会遭到编译器的拒绝。而且从程序的意图来看，这里的转换是“合理”的。不过，C++ 是一种强调类型安全的语言，即使是用 `reinterpret_cast`，也不能任意地将某种类型转换为另一种类型。C++ 编译器会设法保证“最低限度”的合理性。

在各种各样的类型转换中，用户自定义的类类型与其他数据类型之间的转换要引起注意。这里要重点考察这样几种情况：

(1) 由一种类对象转换成另一种类对象。这种转换无法自动进行，必须定义相关的转换函数，其实这种转换函数就是类的构造函数，或者将类类型作为类型转换操作符函数进行重载（参见 3.9 节中的 Program 3.9-3）。例如，要将 class A 的对象转换成 class B 的对象，就要在 class B 中定义一个构造函数，该构造函数接受一个 class A 的对象或引用作为参数。考察下面的程序。

```

/// Program 2.6-7 ///
#include <iostream>
using namespace std;
class Student{
    char name[20];
    int age;
public:
    Student(){};
    Student(char *s,int a){
        strcpy(name,s);
        age=a;
    }
}

```

```

    }
    friend class Team;
};

class Team{
    int members;
    Student monitor;
public:
    Team(){};
    Team(const Student&s):monitor(s),members(0){};
    void Display() const{
        cout << "The number of the members is: ";
        if(members==0)
            cout << "unknown" << endl;
        else
            cout << members << endl;
        cout << "The name of the monitor is: " << monitor.name << endl;
        cout << "The age of the monitor is: " << monitor.age << endl;
    }
};

ostream& operator<<(ostream& out,const Team &t){
    t.Display();
    return out;
}

int main(){
    Student s("王威",18);
    cout << s;
}
/// End of Program 2.6-7 /////

```

该程序的输出结果是：

The number of the members is: unknown

The name of the monitor is: 王威

The age of the monitor is: 18

本来，输出操作符函数 `operator<<` 并不接受 `Student` 类对象作为参数，但由于可通过类 `Team` 的构造函数将 `Student` 类对象转换成 `Team` 类对象，所以输出操作仍然可以成功进行。类的带一个参数的构造函数实际上就充当了类型转换函数。

(2) 由基本数据类型转换成类对象。这种转换仍然是借助于类的构造函数进行的。也就是说，在类的若干重载的构造函数中，有一些接受一个基本数据类型作为参数，这样就可以实现从基本类型数据到类对象的转换。

(3) 由类对象转换为基本数据类型。由于无法为基本数据类型定义构造函数，所以由类对象向基本数据类型的转换必须借助于显式的转换函数。这些转换函数名由 `operator` 后跟

基本数据类型名构成。下面是一个具体的例子。

```
/// Program 2.6-8 ///
#include <iostream>
using namespace std;
class A{
public:
    operator int(){
        return 1;
    }
    operator double(){
        return 0.5;
    }
};
int main(){
    A obj;
    cout << "Treating obj as an interger, its value is: " << (int)obj << endl;
    cout << "Treating obj as a double, its value is: " << (double)obj << endl;
}
/// End of Program 2.6-8 ///
```

程序的输出结果是：

```
Treating obj as an interger, its value is: 1
Treating obj as a double, its value is: 0.5
```

在一个类中定义向基本数据类型转换的函数，要注意以下几点：

- ①类型转换函数只能定义为一个类的成员函数，而不能定义为外部函数。类型转换函数与普通成员函数一样，也可以在类体中声明函数原型，而将函数体定义在类外部。
- ②类型转换函数通常是提供给类的客户使用的，所以应将访问权限设置为 public，否则无法被显式地调用，隐式的类型转换也无法完成。
- ③类型转换函数既没有参数，也不显式给出返回类型。

④转换函数中必须有“return 目的类型数据；”的语句，即必须送回目的类型数据作为函数的返回值。

⑤一个类可以定义多个类型转换函数。C++编译器将根据目标数据类型选择合适类型转换函数。在可能出现二义性的情况下，应显式地使用类型转换函数进行类型转换。如 Program 2.6-8 中输出 obj 时，就必须显式指明将其转换成哪种数据类型进行输出。

综上所述，数据类型转换相当于一次函数调用。调用的结果是生成一个新的数据实体，或者生成一个指向原数据实体但解释方式发生变化的指针（或引用）。编译器不给出任何警告也不报错的隐式转换总是安全的，否则必须使用显式的转换，必要时还要编写类型转换函数。使用显式的类型转换，程序员必须对转换的安全性负责，这一点可以通过两种途径实现：一是利用 C++语言提供的数据类型动态检查机制；一是利用程序的内在逻辑保证类型转换的安全性。



2.7 声明与定义的区别

在 C++ 的教材或参考资料中，经常看到“声明”或“定义”这两个术语。搞清楚它们之间的区别，对于正确理解 C++ 语言的程序结构非常重要。通常将“声明”或“定义”理解成动词，它们所作用的对象有三种：变量、函数和数据类型，我们姑且把这三种对象称为“程序元素”。对一个程序元素而言，声明和定义的最大区别是：声明能够出现多次，而定义只能出现一次。

以下是 C++ 中一些常用的声明语句：

```
int foo(int,int); //函数前置声明  
typedef int Int; //typedef 声明  
class bar; //类前置声明  
extern int g_var; //外部引用声明  
friend test; //友员声明  
using std::cout; //名字空间引用声明
```

声明一个程序元素，是通知编译器在当前作用域内存在这样一个程序元素（从而将该程序元素引入作用域），并提供关于该程序元素的必要信息（而不是全部实现细节）。而定义一个程序元素，则是提供该程序元素在当前作用域内的惟一描述，包括它的全部实现细节。对于三种不同的程序元素，声明和定义还伴随着一些其他语法现象。

(1) 关于变量（基本数据类型的变量或类对象）

对一个变量而言，其定义式是编译器为它分配内存的地方。所以，对于全局变量（全局静态变量）或局部变量而言，数据类型后跟变量名，就是它的定义式。局部变量是不需要声明的，因为 C++ 不允许在定义局部变量之前提前使用它。而全局变量（全局静态变量）则允许在定义之前使用，不过必须先声明。见下面的程序。

```
//// Program 2.7-1 ////  
#include <iostream>  
using namespace std;  
  
extern int i; //statement1  
extern int j; //statement2  
  
int main(){  
    cout << i << endl;  
    cout << j << endl;  
}  
  
int i=5; //statement3  
static int j=6; //statement4  
//// End of Program 2.7-1 ////
```

在这里，statement1 是对全局变量 i 的声明语句，statement2 是对全局静态变量 j 的声明语句。而 statement3 则是全局变量 i 的定义语句，statement4 是全局静态变量 j 的定义语句。对全局静态变量而言，其声明语句和定义语句必须出现在同一个源文件内，而对于全局变量而言，其声明语句和定义语句允许出现在不同的源文件内。



(2) 关于函数

对于函数或函数模板而言，其定义式是提供函数体（function body）的地方。而函数的声明，在C++中有一个专门的术语叫做“函数原型”。一个函数原型是在函数定义中去除函数体后剩下的部分（即函数头），包含函数的返回类型、函数名、形参的个数以及每个形参的数据类型（含缺省值）等信息。下面是一个使用了函数声明的例子。

```
/// Program 2.7-2 ///
#include <iostream>
#include <string>
using namespace std;
void greet(string);           //statement1
int main(){
    string name("张三");
    greet(name);
}
static void greet(string name){           //statement2
    cout << "Hello," << name << "!" << endl;
}
/// End of Program 2.7-2 ///
```

在程序中，`void greet(string);`就是函数`greet()`的声明。而`statement2`才是定义该函数的地方。需要注意的是，由于声明语句是出现在全局作用域，所以将函数`greet()`引入了全局作用域。也可以在一个函数内部声明另一个函数，那么这个时候只能局限在该函数内部使用被声明的函数，否则会引发编译错误。

在函数原型中，函数参数名是可以省略的，因为只有在定义的时候它们才有用。在声明一个函数的时候，从调用函数的需要出发，只需知道函数参数的类型就可以了。

(3) 关于数据类型

基本数据类型是既不需要定义也不需要声明的。构造数据类型（结构、联合、类）需要定义，如果提前使用也需要声明。对构造数据类型（或其模板）而言，其定义式是列出其全部成员的地方，也就是提供类体（class body）的地方。要注意的是，在类的定义中，所有数据成员的说明语句都视为声明，而不是定义。所以，不能在类体内直接为数据成员赋初值，而只能借助于构造函数来完成类对象的初始化工作。类的成员函数只能在类体内声明，可以在类体内定义，也可以在类体之外定义。

一个类被声明了之后，可以用它来声明其他的标识符。但是不允许利用它来定义对象，也不允许使用类的任何成员。见下面的例子。

```
/// Program 2.7-3 ///
#include <iostream>
using namespace std;
class A;
int main(){
    A a;
    a.show();
}
```

```

}
class A{
    int num;
public:
    A(){num=5;}
    void show(){
        cout << num << endl;
    }
};

/// End of Program 2.7-3 ///

```

该程序无法通过编译。类 A 具有全局作用域，但它的所有成员的有效范围是从定义类的地方开始的。同时，由于不知道 sizeof(A) 的大小，也不知道类的构造函数的定义，所以在定义类 A 之前就使用它来定义对象是不允许的。

一般情况下，如果改变类定义的位置，就可以避免提前声明该类。但这并不是在任何时候都行得通的。下面就是一例。

```

/// Program 2.7-4 ///
#include <iostream>
using namespace std;
class A;
class B{
public:
    void show(A &);
};

class A{
private:
    int i;
public:
    A(int in){i=in;}
    friend void B::show(A&);
};

void B::show(A &obj){
    cout << obj.i << endl;
}

int main(){
    A a(5);
    B b;
    b.show(a);
};

/// End of Program 2.7-4 ///

```

在这个程序中，要将函数 B::show() 声明为 A 的友元函数，却采用了非常“迂回”的方式



来实现。如果最先提供 class A 的定义，由于 friend void B::show(A&); 这条语句“暴露”了类 B 的成员细节（即存在成员函数 show()），所以编译器会报错。那么，只能先提供 class B 的定义，而且由于在 class B 中用到了 class A，所以在此之前还必须有类 A 的声明。并且，由于类 A 的声明并没有提供类 A 的实现细节，所以，函数 B::show() 只能在类 A 的定义完成之后再定义，而不能在 class B 的类体内定义。这些复杂的关系说明了：程序元素的声明有时是无法避免的。

一般而言，在同一个作用域内，同样的声明是可以出现多次的。但也有例外：类的成员（包括静态成员）在类体内只能声明一次。

使用声明语句，是将某个程序元素引入了作用域。换句话说，是对某个标识符有了特定含义的解释。所以，在使用声明语句时，要防止不同声明之间的冲突。考察下面的程序。

```
/// Program 2.7-5 ///
#include <iostream>
using namespace std;
int main(){
    int i=5;
    double d=6.6;
    typedef int *ptr;
    ptr p1=&i;
    cout << *p1 << endl;
    typedef double *ptr;
    ptr p2=&d;
    cout << *p2 << endl;
}
/// End of Program 2.7-5 ///
```

这个程序无法通过编译，原因是 ptr 开始被解释成 int *，随后又被解释成 double *，这是不允许的。typedef 在 C++ 中被当做声明语句，原因是它并不产生新的数据类型。使用 typedef 时要防止对同一个表示数据类型的标识符做出不同的解释。

2.8 关于初始化

初始化是程序设计中一项重要的操作，又是一项容易被误解和忽视的操作。使用未初始化的变量（或内存区域）是程序产生 bug 的重要原因。正确理解和使用初始化操作，要弄清以下几个问题。

（1）什么是初始化

初始化是创建一个变量（或对象）并使之具备有意义的值的过程。所以，完整的初始化应该包括两个密不可分的动作：给变量（对象）分配空间；向该空间写入特定的初始值。由于分配空间的操作往往是不能回避的（如由编译器负责分配或者程序员使用 malloc 申请空间），初始化也常指变量（或对象）刚刚获得空间之后向该空间写入特定初始值的操作。

在程序中定义一个变量、用 new 操作创建一个动态对象、函数调用时传递参数、向调用者传递返回值，这些操作都是初始化操作。

(2) 初始化与赋值不同

初始化与赋值是不同的操作。在概念上，初始化是使变量（对象）第一次具备有意义的值的过程，也就是“创建一个有意义的对象”的过程；而赋值则是改变一个已经存在的变量（对象）的值的过程。

对于基本数据类型的变量来说，变量的初始化与赋值在实现方式上区别不大。例如：

```
int i=5; //初始化
int i; i=5; //赋值
```

都是利用赋值符号表示将特定的值写入到变量 i 中。但对于类对象来说，初始化和赋值的操作在实现方式上有很大的差别。例如下面的例子。

```
/// Program 2.8-1 ///
#include <iostream>
using namespace std;
class String{
    char *s;
    unsigned int len;
    unsigned int capacity;
public:
    String(char *str){
        len=strlen(str);
        capacity=len+1;
        s = new char[capacity];
        strcpy(s,str);
    }
    String & operator=(char *str){
        if(strlen(str)+1>capacity){
            delete s;
            capacity=strlen(str)+1;
            s = new char[capacity];
        }
        strcpy(s,str);
        len=strlen(str);
        return *this;
    }
    void show(){
        cout << s << endl;
    }
};
int main(){
    String name("John");
    name.show();
```



```

    name="Johnson";
    name.show();
}
/// End of Program 2.8-1 ///

```

这个程序编写了一个非标准的字符串类 String。该类对象可以由 C 风格的字符串初始化，也可以将 C 风格的字符串赋值给该类对象。从程序中可以看出，对于对象而言，初始化语句的语法形式与赋值不同。赋值只能通过赋值符号“=”进行，而对象的初始化一般采用在圆括号中给出初始化参数的形式来完成。更为重要的是，赋值操作是使用默认的按位复制的方式或者是由重载 operator= 操作来完成，而对象的初始化必须由构造函数来完成。在 String 类的设计中，构造函数只需要根据传入的参数字符串的长度来分配空间就可以了，而赋值操作符重载函数则要考虑传入的参数字符串的长度，然后决定是否要释放原有空间并申请新的空间。可见，构造函数和赋值操作的逻辑流程也是有很大的差别的。

由于模板机制的引入，C++ 允许基本数据类型的变量采用类似默认构造函数的形式初始化。例如 int i=int(); 和 double d=double(); 等，这时变量的初值为 0。

(3) 避免初始化带来的问题

C++ 语言并没有强行规定在创建变量（对象）的时候“必须”为它们赋初始值，这实际上导致很多变量的初始化工作只“完成了一半”。也就是说，变量获得了空间，但并没有获得有意义的值。如果程序员在使用变量（即读取变量的值）之前还是没有为它们赋予有意义的值，那么很可能导致运行时错误。考察下面的程序。

```

/// Program 2.8-2 ///
#include <iostream>
using namespace std;
void DispStar(char * &p,int width){
    int i;
    if(!p || p[0]<width+2){
        delete[] p;
        p=new char[width+2];
        p[0]=width+2;
    }
    if(p){
        for(i=1;i<=width;i++)
            p[i]='*';
        p[i]='\0';
        cout << p+1 << endl;
    }
}
int main(){
    int i;
    char *p=NULL;
    for(i=0;i<10;i++)

```



```
    DispStar(p,rand()%20);
    delete p;
}
```

```
/// End of Program 2.8-2 ///
```

在主函数 main() 中定义了一个指针 p，p 所指向的空间中会连续存放若干个星号（'*'），星号的个数由随机数发生器 rand() 产生，并由取模运算保证不会大于 20，然后将其存在 p[0] 中。程序会利用 10 次循环，输出由不等长的多行星号形成的图案。

由于函数 DispStar() 负责为指针 p 分配空间，当 p 为 NULL 或者 p 所指向的空间“不够用”时，就会为它分配更大的空间，所以，进入函数时 p 的状态就非常关键。在以上程序中，在定义指针 p 时将其初始化为 NULL，这样在第一次调用函数 DispStar() 时，就能保证该函数会为 p 分配空间。否则，在程序中将定义 p 的语句改为 char* p;（也就是说不对它进行初始化），就将导致严重的运行时错误，读者可以自行试验一下。

所以，对指针变量而言，在定义它的时候就将其初始化为 NULL 是一种很好的编程习惯，是防止指针变成“悬挂指针”（参见 5.8 节）的一种有效手段。另外一些对程序运行起关键作用的变量，也应该在定义的时候完成初始化。

在 C++ 中，用 new 操作生成类对象的时候，会自动调用该类的构造函数，而使用 malloc 为对象分配空间，就不会自动调用构造函数，从而导致对象的初始化工作没有完成。所以，C++ 中提倡用 new 和 delete 来生成和销毁对象，而不要用 malloc() 和 free()。

（4）编译时与初始化相关的错误

在某些时候，初始化是强制性进行的，没有初始化会导致编译错误。定义常变量的时候，必须同时为它初始化；由于引用相当于指针常量，所以定义引用时也必须同时初始化；定义常对象的时候，相应的构造函数必须显式定义。考察下面的程序。

```
/// Program 2.8-3 ///
#include <iostream>
using namespace std;
class A{
    int num;
public:
    void show() const{
        cout << num << endl;
    }
};
int main(){
    const A a;
    a.show();
}
/// End of Program 2.8-3 ///
```

在此程序中定义了一个常对象 a，然后调用其常函数 show()。但是，类 A 并没有显式定义参数个数为 0 的构造函数，而编译器提供的默认构造函数实际上什么初始化的工作都没有做。由于 a 是常对象，如果构造函数不作初始化的工作，它就永远没有机会获得有意义的值。



所以，在很多编译器（如 DJGPP）下，以上程序无法通过编译。在 VS 2005 下，程序虽然能够通过编译，但运行结果也没有任何意义。所以，如果要生成常对象，必须显式定义其对应的构造函数，完成对象的初始化工作。

还有一种情况，由于程序的控制结构可能导致某些变量无法初始化，也将引发编译错误。最常见的是 goto 语句和 switch 语句。见下面的程序。

```
//// Program 2.8-4 /////
#include <iostream>
using namespace std;
int main(){
    int i;
    cin >> i;
    if(i==8)
        goto disp;
    int j=9;
disp:
    cout << i+j << endl;
}
//// End of Program 2.8-4 ////
```

这个程序在很多编译器下无法通过编译，即使通过编译运行时也会出问题。原因是 goto disp 语句会跳过变量 j 的初始化语句，即使 j 被分配空间（很多编译器集中分配临时变量的空间），它也无法获得有意义的初值，这样会导致运行时错误。下面是另外一例。

```
//// Program 2.8-5 /////
#include <iostream>
using namespace std;
int main(){
    int i;
    cin >> i;
    switch(i){
        case 1: int j=6;break;
        case 2: cout << "Hello" << endl;
    }
}
//// End of Program 2.8-5 ////
```

这个程序也无法通过编译。在 VS 2005 下，编译器给出的出错信息是：“j”的初始化操作由“case”标签跳过。由于 C++ 没有强制 switch 语句的各 case 分支使用 break，所以在一个 case 分支中定义的变量是可能被其他分支的语句所使用的。由于 case 分支被执行的随机性，无法保证变量有合适的初值。所以，除非只有一个 case 分支，否则不要在 case 分支中定义局部变量。

2.9 作用域和生命期

作用域和生命期是两个完全不同的概念。在英文中，作用域用“scope”表示，生命期则用“duration”表示。作用域是一个静态概念，只在编译源程序的时候有用。一个标识符的作用域是指在源文件中该标识符能够独立地合法出现的区域。而生命期则是一个运行时（Runtime）概念，它是指一个变量在整个程序从载入到结束运行的过程中，在哪一个时间区间有效。因此，C++的所有标识符都有作用域，而只有变量（或动态分配的数据实体）有生命期的概念。

作用域是一个范围，而范围不加界定就容易产生混淆。在本书中，严格将作用域限定在单个C++源文件内，并且对标识符的访问是指“直接”访问，即不需要借助于作用域指示符（::）或点操作符。

在C++中，存在5种作用域级别：函数原型域、块域(局部域)、类域、名称空间域和文件域。这些区域都是C++源文件的自然组织单位，任何一个标识符的作用域必然位于这5种范围之一。

考虑到C++源程序的书写形式，需要对与作用域相关的概念做进一步的明确。以下是几个与作用域相关的概念。

①声明域（declaration region），即声明标识符的区域，也就是上面提到的5种作用域级别之一。如在函数外面声明的全局变量，它的声明域为声明所在的文件。在函数内声明的局部变量，它的声明域为声明所在的代码块（例如整个函数体或整个复合语句）。

②潜在作用域（potential scope），即从声明点开始，到声明域的末尾的区域。C++中的标识符采用的是先声明后使用的原则，所以在声明点之前的声明域中，标识符是不能用的。因此，标识符的潜在作用域，一般会小于其声明域。

③作用域（scope），即标识符对程序（直接）可见的范围，也称有效作用域。标识符在其潜在作用域内，并非在任何地方都是可见的。例如，局部变量可以屏蔽全局变量、嵌套层次中的内层变量可以屏蔽外层变量，从而被屏蔽的全局或外层变量在其被屏蔽的区域内是不可见的。所以，一个标识符的作用域可能小于其潜在作用域。

所以，在相关的论述中提及作用域，一般是指标识符的有效作用域，或者是其潜在作用域。当然，“作用域”也可能是指标识符的声明域，这需要读者小心分辨。

对于变量而言，其作用域与生命期有时是一致的。例如在某个块（Block）中定义的局部变量，当这个块在运行结束的时候，变量也随之失效了。而在另外一些时候，变量的作用域与生命期毫无关系。典型的如静态变量，它的生命期与全局变量是一样的，而它的作用域则局限在某个特定的函数、类或文件内。

在同一个声明域内定义同名的两个变量，会发生编译错误（重定义错误）。不同声明域中的变量允许同名。如果两个变量的声明域是相互分离的，它们之间不会发生任何干扰。如果它们的声明域是重叠的，例如在一个块中定义了一个变量，然后又在其内部的嵌套块中定义一个与之同名的变量，则发生同名变量的屏蔽现象。如下面的程序。

```
/// Program 2.9-1 ///
#include <iostream>
using namespace std;
```



```

int i=8;
int main(){
    int i=5;
    {
        int i=6;
        cout<<i<<endl;           //在内层块中无法访问到外层块中的变量i
    }
    cout<<i<<endl;
    cout<<::i<<endl;         //在任何地方都能访问到全局变量i
}
/// End of Program 2.9-1 ///

```

其输出结果是 6、5 和 8。在内层块中，无法访问外层块中定义的变量 i，因为内层块中定义的变量 i 将其屏蔽了。C/C++语言之所以不提供这种访问机制，倒不是因为实现起来有多大的困难，而是从实用的角度说实在没有多大必要。在程序的任何地方，都能方便地访问到全局变量 i，只要在变量前加作用域指示符::即可。

变量的作用域识别起来比较简单，一般不会有什么问题。需要注意的是，在 for 语句中定义的变量，其作用域在新旧标准上有一定的差异。如下面的程序。

```

/// Program 2.9-2 ///
#include <iostream>
using namespace std;
int main(){
    for(int i=0;i<5;i++);
    cout << i;
}
/// End of Program 2.9-2 ///

```

根据旧的标准，在 for 语句中定义的变量 i 离开循环体之后继续有效，所以应该输出 5。而根据新的标准，变量 i 只在循环体内有效，在循环体外访问 i 是非法的。所以，如果在离开循环之后还要访问某个变量，就应该在 for 语句之前定义它。

除了变量有作用域外，在 C++ 源程序中定义（或声明）的标识符都有作用域。理解标识符的作用域，要注意以下几点：

①一般情况下，标识符的潜在作用域要小于其声明域。也就是说，标识符的有效范围是从定义（声明）标识符的位置开始，到标识符所在的声明域结束的位置终止。如变量、用户自定义类型等都遵循这样的原则。

②有些特殊的标识符，其潜在作用域与其声明域相同。如标号，其作用域遍布定义它的函数体内的任何位置，而出了这个函数，标号就失效。考察下面的程序。

```

/// Program 2.9-3 ///
#include <iostream>
using namespace std;
void func1(){
    goto Label2;
}

```



```
Label1:  
    cout << "Hello!" << endl;  
Label2:  
    cout << "in func1" << endl;  
}  
void func2(){  
Label1:  
    cout << "in func2" << endl;  
}  
int main(){  
    func1();  
    func2();  
}  
/// End of Program 2.9-3 ///
```

这个程序能顺利地通过编译并正确运行。在函数 func1() 中，在还没有定义 Label2 的时候就可以使用它，说明 Label2 的有效范围遍布 func1() 的函数体。而在函数 func1() 和 func2() 中，都有一个叫做 Label1 的标号，它们并不发生冲突，说明标号的作用域局限在函数范围以内。

③ 标识符的作用域不会超出其声明域的范围。

④ 作用域范围可以是一个连续的代码范围，如块域、文件域，也可以是分段定义的各段代码的集合，如命名空间和类域。命名空间可以分段定义，但各段属于同一个作用域范围；类的成员函数可以定义在类体外部，但它仍然属于该类域的一部分。

⑤ 标识符的作用域与标识符的可用区域并不完全相同，因为某些标识符可以借助于特殊的机制（如作用域指示符、点操作符等）在作用域以外的范围被合法地使用。例如，用户自定义类如果置于一个函数内部，那么它只在该函数内部有效。此时，该类的作用域与其可用区域是一致的。而如果在一个全局类的内部定义嵌套类，只要其访问权限是 public，仍然可以在全局范围内使用（通过作用域指示符）。另外一个可以在标识符的作用域以外访问该标识符的例子，是借助于点操作符访问类对象的成员。

2.10 关于头文件

头文件是 C/C++ 程序不可缺少的组成部分。使用头文件，应该注意哪些问题呢？下面从头文件的内容、系统提供的头文件、避免头文件重复包含等方面对其进行讨论。

(1) 头文件的内容

头文件中应该放置什么内容，应该从设立头文件的原因谈起。头文件是由 #include 指令引入某个源文件的，编译器在预编译阶段会把头文件的内容“插入”到包含它的源文件中，然后统一进行编译。所以，仅从编译过程的角度看，可以把任何代码放入头文件。一些不了解头文件作用的初学者，有时也会随意将一部分代码移入头文件。

需要注意的是，C++ 编译器采用的是分离编译模式（参见 2.11 节）。如果程序员觉得一个源文件过长，可以把它分割成两个源文件实现，从这个角度来看，没有必要利用头文件从一个较长的源文件中“转移”出一部分代码。



那么，设立头文件的目的是什么呢？观察在一个项目下的多个源文件，发现它们总会有一些内容是相同的，如使用了相同的用户自定义类型、使用了相同的全局变量等。因此，将这些相同的内容“抽取”出来放到头文件中去，然后再提供给各个源文件包含，就可以避免这些内容的重复书写，提高编程效率和代码安全性。所以，设立头文件的目的主要是提供全局变量、全局函数的声明或提供公用数据类型的定义，从而实现分离编译或代码复用的目的。在这里，有一个判断头文件中的内容是否合适的简单准则：规范的头文件应该可以被多个源文件包含而不引发编译错误。

因此，下面这些内容应该放在头文件中：外部函数原型声明、全局变量声明、自己定义的宏和类型等。而下面这些内容则不应该放到头文件中去：全局变量的定义、外部函数的定义、静态变量和静态函数的定义、在类体之外的类的成员函数的定义等。这样做的目的，就是为了让头文件能够被多个源文件包含。如果一个头文件，在一个项目中只能被包含一次，就起不到联络和共享的作用，这样的头文件就没有必要存在了。而有很多内容，比如全局变量的定义、外部函数的定义等，在全局范围内是唯一的，而头文件又必须满足被多次包含的要求，所以这些内容是不宜放进头文件的。

概括地说，头文件有如下三个作用：

①加强类型检查，提高代码的类型安全性。

在 C++ 中使用头文件，对于自定义类型的安全也是非常重要的。虽然，在语法上，同一个数据类型（如一个 class）在不同的源文件中书写多次是允许的，程序员也认为它们是同一个自定义类型。但是，由于用户自定义类型不具有外部连接特性，编译器并不关心该类型的多个版本之间是否一致，这样会导致逻辑错误的发生。考察下面的程序。

```
/// Program 2.10-1 ///
/** source1.cpp */
#include <iostream>
class A{
    char num;
public:
    A();
    void show();
};
void A::show(){
    std::cout << num << std::endl;
}
void see(A &a){
    a.show();
}
/** end of source1.cpp **/


```

```
/** source2.cpp **/
```

```
class A{
    int num;
```

```
public:  
    A(){num=5;}  
    void show();  
};  
void see(A&);  
int main(){  
    A a;  
    see(a);  
}  
/** end of source2.cpp **/  
/// End of Program 2.10-1 ///
```

在构成项目的两个源文件 source1.cpp 和 source2.cpp 中，由于某种原因，对 class A 的定义出现了一点小小的不一致。在 source1.cpp 中，成员 num 的类型是 char，而在另一个源文件 source2.cpp 中，成员 num 的类型是 int。本来是要将整数 5 输出到屏幕上，但却输出了一个“怪”字符。

如果将 class A 的定义放到一个头文件中，用到 class A 的源文件都将这个头文件包含进来，就可以绝对保证数据类型的安全性。

②减少代码的重复书写，提高编写和修改程序的效率。

在程序开发的过程中，对某些数据类型或接口进行修改是难免的。有了头文件之后，只需修改头文件的内容，就可以保证修改在所有源文件中生效，从而避免了繁琐易错的重复修改。

③提供保密和代码重用的手段。

头文件也是 C++ 代码重用机制中不可缺少的一种手段。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) 系统提供的头文件

熟悉 C 语言编程的人都知道，由 C 语言系统环境提供的头文件都是以.h 结尾的，如最常使用的 stdio.h 等。C++ 语言最初的目的之一是成为“更好的 C”，所以 C++ 语言沿用了 C 语言头文件的命名习惯，而且这些头文件也位于全局名字空间中，如 iostream.h 等。但是，随着 C++ 语言的发展，C++ 发展了自己的标准库，而且为了避免与 C 发生冲突，C++ 标准库使用了新的名字空间 std，在这个空间中的头文件都不带.h 作为扩展名。同时，新的 C++ 标准库把 C 语言的头文件也移植进来，只不过在文件名前面加上 c 而去掉后面的.h，如 C 语言中的头文件 stdio.h（位于全局名字空间）变成了 std 名字空间中的头文件 cstdio。

于是，在一段时间里，很多头文件有两个版本，一个以.h 结尾，而另一个则不是，如 iostream.h（位于全局名字空间）和 iostream（位于名字空间 std）。程序员编写程序也有不同的选择，很多 C++ 源程序以这样的语句开始：

```
#include <iostream.h>
```

而另外一些，则以这样两条语句开始：

```
#include <iostream>  
using namespace std;
```



这种现象多少有些混乱。根据 C++ 标准委员会的意见，旧的 C 头文件（如 stdio.h）可以继续使用，以保持对 C 语言的兼容，新的 C 头文件（如 cstdio）也可以使用；旧的 C++ 头文件（如 iostream.h）则明确反对使用，C++ 标准已明确提出不再对它们提供支持。所以，C++ 程序员应使用名字空间 std 中的头文件，而放弃旧的 C++ 头文件。实际上，在 VC++ 2005 中已经不提供对旧的 C++ 头文件的支持。

另外，在包含系统头文件的时候，应该使用 <>（尖括号）而不是 ""（双引号）。例如应该这样包含头文件 iostream：

```
#include <iostream>
```

而不是这样：

```
#include "iostream"
```

双引号 "" 用来包含自定义的头文件，用它来包含系统头文件是一种不良的编程习惯。

(3) 避免头文件的重复包含

C++ 语言中的内容，有的在项目一级的范围内只能出现一次，如全局变量的定义、函数的定义等；有的可以在项目一级的范围出现多次，但在一个源文件中只能出现一次，如 class 的定义等；还有的在一个源文件中也可以出现多次，如函数声明等。由于我们事先无法确切知道头文件中的内容，所以应该避免在一个源文件中对同一个头文件包含多次，以免引起重定义错误。考察下面的程序。

```
/// Program 2.10-2 /////
/** header1.h ***/
class A{
    int num;
public:
    A();
    void show();
};

/** end of header1.h **/


/** header2.h ***/
#include "header1.h"
class B{
    A a;
public:
    void disp();
};

/** end of header2.h **/


/** main.cpp ***/
#include <iostream>
#include "header1.h"
#include "header2.h"
```

```
A::A(){  
    num=5;  
}  
void A::show(){  
    std::cout<<num<<std::endl;  
}  
int main(){  
    A a;  
    a.show();  
}  
/** end of main.cpp ***/  
/// End of Program 2.10-2 ///
```

这个程序无法通过编译，原因是class A被定义了两次。class A在header1.h中定义了一次，而header1.h被header2包含了一次，而这两个头文件都被main.cpp所包含，导致class A被定义了两次。一个头文件被别的头文件包含是经常发生的事，所以必须想办法阻止头文件的重复包含。其实，利用条件编译能很轻松地解决这个问题。原理是：当某个头文件被包含之后，就定义一个相关的条件变量，下一次包含同一个头文件时，检查相应的条件变量是否已经定义，如果已经定义了，就不要重复包含。将以上程序改造如下。

```
/// Program 2.10-3 ///  
/** header1.h ***/  
#ifndef HEADER1  
#define HEADER1  
class A{  
    int num;  
public:  
    A();  
    void show();  
};  
#endif  
/** end of header1.h ***/
```

```
/** header2.h ***/  
#include "header1.h"  
class B{  
    A a;  
public:  
    void disp();  
};  
/** end of header2.h ***/
```



```
/** main.cpp ***/
#include <iostream>
#include "header1.h"
#include "header2.h"
A::A(){
    num=5;
}
void A::show(){
    std::cout<<num<<std::endl;
}
int main(){
    A a;
    a.show();
}
/** end of main.cpp ***/
/// Program 2.10-3 ///
```

改造之后的程序可以顺利通过编译，并输出：5。在这里要注意这样几点：

①条件编译指令#ifndef HEADER1和#endif的意思是：如果条件编译标志HEADER1没有定义的话，就编译#ifndef和#endif之间的程序段，否则就忽略它。可见，头文件header1.h只要被包含一次，条件编译标志（宏）HEADER1就会被定义，这样就不会再次被包含。

②iostream是系统提供的头文件，所以被包含时在文件名两边使用尖括号<>；而header1.h和header2.h是用户自定义的头文件，包含时在文件名两边使用双引号""。

③程序中，如果要防止header2.h被包含多次，也可以用同样的方法进行。有兴趣的读者可以自己试一试。

2.11 什么是分离编译模式

分离编译模式源于C语言，在C++语言中继续沿用。简单地说，分离编译模式是指一个程序（项目）由若干个源文件共同实现，而每个源文件单独编译生成目标文件，最后将所有目标文件链接起来形成单一的可执行文件。

分离编译模式是C/C++组织源代码和生成执行文件的方式。在实际开发大型项目的时候，不可能把所有的源程序都放在一个文件中，而是分别由不同的程序员开发不同的模块，再将这些模块汇总成为最终的可执行程序。这里就涉及到不同的模块（源文件）定义的函数和变量之间的相互调用问题。C/C++语言所采用的方法是：只要给出函数原型（或外部变量声明），就可以在本源文件中使用该函数（或变量）。每个源文件都是一个独立的编译单元，在本源文件中使用但未在此定义的变量或函数，就假设在其他的源文件中定义好了。每个源文件生成独立的目标文件（obj文件），然后通过链接（link）将目标文件组成最终的可执行文件。理解分离编译模式，要注意几点：

①每个函数或外部变量只能被定义一次，但可以被多次“声明”。见下面的程序。

/// Program 2.11-1 ///



```
#include <iostream>
using namespace std;
void func();           //函数func()被声明了一次
void func();           //函数func()被再次声明
void func(){          //函数func()只能定义一次
    cout << "This is a demo" << endl;
}
int main(){
    func();
}
/// End of Program 2.11-1 ///
```

函数func()被多次声明，并不影响程序的正常编译和运行。其实这正是C++分离编译模式的特点之一。在一个源文件中允许同时包含定义和声明同一个标识符的语句，这样有利于头文件内容的组织。

②函数声明也是有作用域的。类的成员函数只能在类体中声明。对于外部函数，如果是在一个函数体内声明另外一个外部函数，那么该函数声明的作用域就是从声明处起到函数体结束为止。在别的位置要调用这个函数，还必须再次声明。如下面的程序，由两个cpp源文件组成，一个是a.cpp，定义了一个函数func()；另一个是b.cpp，在此源文件中有两个函数show()和main()，这两个函数都调用了在a.cpp中定义的函数func()。如果坚持将函数声明放在函数体内部，则在函数show()和main()中都必须分别对函数func()进行声明，否则编译器报错。程序代码如下。

```
/// Program 2.11-2 ///
/** a.cpp ***/
#include <iostream>
using namespace std;
void func(){
    cout << "This is a demo" << endl;
}
/** end of a.cpp **/


/** b.cpp ***/
void show(){
    void func();           //这个函数声明必不可少，否则编译出错
    func();
}
int main(){
    void func();           //这个函数声明必不可少，否则编译出错
    func();
    show();
}
```

```
/** end of b.cpp **/
```

```
/// End of Program 2.11-2 ///
```

通常情况下，将外部函数或外部变量的说明放在.h头文件之中。对于不在本源文件中定义的函数（或变量），只要将相应的头文件通过#include指令包含进来，就可以正常使用了。

③函数声明只是表明在当前的源文件中有可能用到该函数，但并不一定会用到。因此，一个函数被声明却从未定义，也是允许的。只要没有发生函数调用，在生成执行文件阶段就不会发生连接错误。见下面的程序。

```
/// Program 2.11-3 ///
```

```
#include <iostream>
```

```
using namespace std;
```

```
class Demo{
```

```
public:
```

```
    void func1();
```

```
    void func2();
```

```
};
```

```
void Demo::func1(){
```

```
    cout << "This is a demo" << endl;
```

```
}
```

```
int main(){
```

```
    Demo obj;
```

```
    obj.func1();
```

```
}
```

```
/// End of Program 2.11-3 ///
```

表面上看，类Demo的定义是不完全的，因为在类体中声明了函数func2()，却从未对该函数加以定义。但这并不影响程序的编译和运行，因为实际上并没有发生对func2()的调用。从分离编译模式的角度看，函数Demo::func2()有可能是在别的源文件中定义的。下面的程序就说明了这一点。

```
/// Program 2.11-4 ///
```

```
/** a.cpp **/
```

```
#include <iostream>
```

```
using namespace std;
```

```
class Demo{
```

```
public:
```

```
    void func1();
```

```
    void func2();
```

```
};
```

```
void Demo::func2(){
```

```
    cout << "This is func2" << endl;
```

```
}
```

```
/** end of a.cpp **/
```

```

/** b.cpp ***/
#include <iostream>
using namespace std;
class Demo{
public:
    void func1();
    void func2();
};
void Demo::func1(){
    cout << "This is func1" << endl;
}
int main(){
    Demo obj;
    obj.func2();
}
/** end of b.cpp ***/
/// End of Program 2.11-4 ///

```

类Demo有两个成员函数，它们分别在a.cpp和b.cpp源文件中实现。类Demo是被“分离”实现的。所以，分离编译模式关心的是函数的调用规范（函数原型），至于函数是否真正实现要到连接的时候才被发现。

由分离编译模式也可以得到头文件的书写规范。那就是，头文件的目的是提供其他源文件中定义的，可以被当前源文件使用的内容（函数、变量等）的说明。因此，有个基本的假设是：头文件要多次被不同源文件包含。因此，一般都不在头文件中定义函数、定义外部变量，因为这样的头文件只能被包含一次，没有被第二次包含的可能性，背离了设立头文件的初衷（详细讨论参见2.10节）。

在一个源文件中定义函数，而在另一个源文件中调用该函数，是分离编译模式下十分普遍的现象。但如果定义的不是一个普通函数，而是一个函数模板，却可能发生严重的问题，具体的描述和解决方案参见6.5节。

第3章 函数

3.1 关于main()函数

main()函数是C++程序的入口函数。最常见的是把main()函数的返回值类型定义为int或void（C++标准要求main()函数的返回值类型为int）。当main()函数的返回值类型为int，而函数体内并没有出现return语句时，同样可以通过编译并正常运行。这是因为编译器在main()函数的末尾自动添加了return 0;的语句。所以，main()函数是C++程序中经过特殊处理的函数。其他的返回值类型不是void的函数，如果没有使用return语句，编译器将报错。

其实，如果我们在Visual Studio .NET 2005中作个实验的话，就会发现main()函数的返回值类型可以是任意的类型。如下述程序，可以正常运行。

```
/// Program 3.1-1 ///
#include <iostream>
using namespace std;
float main(){
    cout << "Hello!" << endl;
    return 0.0;
}
/// End of Program 3.1-1 ///
```

上面的程序只是在VS 2005环境下的一个特例，并不具有可移植性。从程序员的角度说，main()函数的主要作用是提供C++程序的入口，它的返回值通常并不为程序员所关心。正是基于这种考虑，Visual C++并没有强制性要求main()函数的返回值类型是int。但是，main()函数的返回值可以向操作系统提供程序的运行状态信息。返回0表明程序运行成功，返回非0值则代表程序运行出了问题。在某些C++编译器中（如djgpp），严格要求main()函数的返回值类型为int，否则会出现编译错误。所以，应该将main()函数的返回值类型设定为int，这样才符合C++标准，使程序具有可移植性。

在Windows平台下，可通过环境变量errorlevel获取用C++开发的应用程序的main()函数的返回值，并据此返回值做出不同的响应。例如，编写如下的程序。

```
/// Program 3.1-2 ///
#include <iostream>
int main(){
    int i;
    std::cout << "please input a number:" ;
    std::cin >> i;
```



```
    return i;  
}  
/// End of Program 3.1-2 ///
```

将此程序编译之后生成returnval.exe，然后编写一个批处理文件（如test.bat），内容如下：

```
@echo off  
returnval  
if %errorlevel% == 3 echo third  
if %errorlevel% == 2 echo second  
if %errorlevel% == 1 echo first
```

则当我们运行此批处理文件，从控制台输入1，得到输出first；从控制台输入2，得到输出second；从控制台输入3，得到输出third。这个实验说明了当程序returnval.exe运行时，main()函数的返回值被存放在环境变量errorlevel中，我们可以在批处理文件中利用这个返回值采取不同的行动。

在上面的程序中，如果将语句return i；改成函数调用exit(i)；这个程序的执行结果不发生变化。exit(i)的执行效果是返回操作系统，并将i作为程序的返回结果。在main()函数中，return i；和exit(i)；能达到同样的效果。但是，exit()并不只是可以用在main()函数中，它可以用在程序的任何地方。在C语言程序当中，当程序出现无法恢复的错误时，就有可能使用exit()函数退出程序。但是，在C++程序中，exit()函数的使用会破坏程序对对象的析构函数的调用，控制得不好还会引发其他程序设计问题。在C++程序设计中，应利用异常处理机制（参见第10章）来取代对exit()函数的使用。

系统在提交命令行参数时，会自动在数组argv[]的最后一个有效参数后面加一个空指针。这样，即使不知道argc的值，通过字符指针数组中空指针所处的位置，也能推断出命令行中有多少个参数。考察如下程序的输出结果。

```
/// Program 3.1-3 ///  
#include <iostream>  
using namespace std;  
int main(int argc,char* argv[]){  
    int i=0;  
    while(argv[i])  
        cout << argv[i++] << " ";  
    cout << endl;  
    cout << "argc=" << argc << endl;  
    cout << "i=" << i << endl;  
}  
/// End of Program 3.1-3 ///
```

编译此程序生成执行文件nullp.exe，然后执行nullp one two three，程序的输出是：

```
nullp one two three  
argc=4  
i=4
```

也就是说，由于空指针出现在argv[4]的位置，所以命令行参数的个数为4，这正好和系统

提供的参数argc的值相同。

在C/C++语言程序中，main()函数被称为“入口函数”。那么，main()函数一定是程序中第一个被执行的函数吗？考察下面的程序。

```
/// Program 3.1-4 ///
#include <iostream>
using namespace std;
class A{
public:
    A(){
        cout << "In constructor." << endl;
    }
};
A a;
int main(){
    cout << "In main()." << endl;
}
/// End of Program 3.1-4 ///
```

在这个程序中，先输出的是“In constructor.”，后输出的是“In main().”。这表明对象a的构造函数是先于main()函数执行的。实际上，所有的外部对象的构造函数都是先于main()函数执行的。如果要对类中的静态成员对象进行初始化，那么这些对象的构造函数也是在main()函数之前执行的。如果在这些构造函数中还调用了其他函数的话，就可以使更多的函数先于main()函数执行。再者，可以编写一个函数，该函数的返回值可以用来为静态、全局变量提供初值（参见4.9节）。所以，尽管沿用惯例称main()函数为“入口函数”，但main()函数却不一定是在C++程序中第一个被执行的函数。

main()函数的另一个重要特性，是可以带参数，以处理由用户输入的命令行参数。main()函数所带的参数有固定格式，即int main(int argc,char *argv[]），其中argc代表参数的个数，而argv数组中的每一个元素则是一个保存命令行参数内容的字符串。考察下面的程序。

```
/// Program 3.1-5 ///
#include <iostream>
using namespace std;
int main(int argc,char *argv[]){
    if(argc>1)
        cout << "Hello, " << argv[1] << "!" << endl;
}
/// End of Program 3.1-5 ///
```

假设此程序经过编译之后生成test.exe，则通过键盘输入“test Jack”，会在屏幕上输出“Hello, Jack!”。在使用命令行参数时要注意这样几个问题：

①命令行参数包括用户输入的所有内容。如在命令行输入test Jack，在main()函数中字符串argv[0]的内容是“test”。如果在命令行输入test.exe Jack，则在main()函数中字符串argv[0]的内容是“test.exe”。在其他的编程语言（如C#语言）中，命令行参数并不包含执行文件的

名字。

②在命令行中，空格被认为是命令行参数的分隔符。也就是说，同一个参数内部不允许出现空格。如果确实需要在一个参数中使用空格，可以将该参数置于一对双引号之间。如在上例中，输入test “Jack and Mary”，则会输出“Hello, Jack and Mary!”。

3.2 函数参数是如何传递的

要实现函数调用，除了要知道函数的入口地址外，还要向函数传递合适的参数。向被调用函数传递参数，可以采用不同的方式实现。这些方式被称为“调用规范”或“调用约定”。`_cdecl`是C和C++程序默认的函数调用约定，参数按从右到左的顺序压入堆栈，由主调函数（caller）负责维护堆栈（在调用之前将参数压入堆栈，调用完成后将参数弹出栈）。也正是因为用来传送参数的堆栈是由主调函数维护的，所以实现可变参数的函数只能使用这种函数调用约定。因为每一个主调函数都要包含清理堆栈的代码，所以编译后的可执行文件的大小要比使用其他调用约定（如`_stdcall`）的程序大。下面，通过一个非常简单的小程序考察一下函数调用的实现过程。

```
//// Program 3.2-1 /////
#include <stdio.h>
int main(){
    int i=0,j=1;
    printf("%d %d",i,j);
}
/// End of Program 3.2-1 ///
```

其中函数调用语句`printf("%d %d",i,j);`所对应的汇编语言代码是：

```
mov    eax, DWORD PTR _j$[ebp]
push   eax
mov    ecx, DWORD PTR _i$[ebp]
push   ecx
push   OFFSET ??_C@_05OKMLJOMC@?$_CFd?5?$_CFd?$_AA@
call   DWORD PTR __imp__printf
add    esp, 12
```

可以看到，在主调函数（此处是`main()`）中，通过三个`push`语句，依次把参数`j`、`i`和格式字符串首地址压入堆栈，通过`call`指令完成函数调用后，再通过`add esp, 12`将12个字节的参数数据（两个整型量各占4个字节，一个字符指针也占4个字节，加起来一共12字节）从堆栈中清除。

但是，如果传递给函数的参数不是简单的变量，而是表达式，就涉及到表达式求值的问题。要先将表达式化简，然后再实现函数调用。如下面的程序。

```
//// Program 3.2-2 /////
#include <stdio.h>
int main(){
    int i=0,j=1;
```



```
    printf("%d %d", i+1, j+1);
}
```

//// End of Program 3.2-2 ////

其中函数调用语句printf("%d %d", i+1, j+1);所对应的汇编语言代码是：

```
mov    eax, DWORD PTR _j$[ebp]
add    eax, 1
push   eax
mov    ecx, DWORD PTR _i$[ebp]
add    ecx, 1
push   ecx
push   OFFSET ??_C@_05OKMLJOMC@?$_CFd?5?$_CFd?$_AA@
call   DWORD PTR __imp__printf
add    esp, 12
```

也就是说，从右向左计算各个参数表达式的值，依次将计算结果入栈。当参数表达式中带有自增或自减运算时，情况会变得更为复杂一些。如下面的程序。

//// Program 3.2-3 ////

```
#include <stdio.h>
int main(){
    int i=0,j=1;
    printf("%d %d", i++, j++);
}
```

//// End of Program 3.2-3 ////

其中函数调用语句printf("%d %d", i++, j++);所对应的汇编语言代码是：

```
mov    eax, DWORD PTR _j$[ebp]
mov    DWORD PTR tv66[ebp], eax
mov    ecx, DWORD PTR _j$[ebp]
add    ecx, 1
mov    DWORD PTR _j$[ebp], ecx
mov    edx, DWORD PTR _i$[ebp]
mov    DWORD PTR tv69[ebp], edx
mov    eax, DWORD PTR _i$[ebp]
add    eax, 1
mov    DWORD PTR _i$[ebp], eax
mov    ecx, DWORD PTR tv66[ebp]
push   ecx
mov    edx, DWORD PTR tv69[ebp]
push   edx
push   OFFSET ??_C@_05OKMLJOMC@?$_CFd?5?$_CFd?$_AA@
call   DWORD PTR __imp__printf
add    esp, 12
```



代码中tv66[ebp]处存放的是表达式j++的计算结果，tv69[ebp]处存放的是表达式i++的计算结果。由于变量j和i的自增运算是函数调用之前完成的，所以最后入栈的是tv66[ebp]和tv69[ebp]处存放的整数值。

在Visual C++的函数调用规范中，如果函数的任何一个参数表达式包含自增（自减）运算，所有这些运算会在第一个push操作之前全部完成，然后再完成其他的运算并将结果入栈。接下来考察另一个程序。

```
//// Program 3.2-4 /////
#include <iostream>
using std::cout;
int main(){
    int i=10;
    cout<<++i<<-i<<i++;
}
/// End of Program 3.2-4 ///
```

按照“正常”的思维，标准输出操作符<<是从左向右结合的，所以应该依次计算表达式++i, -i和i++的值，那么最终应该依次输出11, 10和10。但在Visual C++中运行的输出结果却是11, 11和10。考察此程序的汇编代码，发现语句cout<<++i<<-i<<i++;所对应的汇编语言代码是：

```
mov    eax, DWORD PTR _i$[ebp]
mov    DWORD PTR tv75[ebp], eax
mov    ecx, DWORD PTR _i$[ebp]
add    ecx, 1
mov    DWORD PTR _i$[ebp], ecx
mov    edx, DWORD PTR _i$[ebp]
sub    edx, 1
mov    DWORD PTR _i$[ebp], edx
mov    eax, DWORD PTR _i$[ebp]
add    eax, 1
mov    DWORD PTR _i$[ebp], eax
mov    ecx, DWORD PTR tv75[ebp]
push   ecx
mov    edx, DWORD PTR _i$[ebp]
push   edx
mov    eax, DWORD PTR _i$[ebp]
push   eax
mov    ecx, OFFSET ?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A
call   ??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@H@Z
mov    ecx, eax
call   ??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@H@Z
mov    ecx, eax
```



```
call    ??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@H@Z
```

这段汇编代码比较复杂，现把几个关键的地方解释一下。首先，虽然`<<`运算是从左向右结合，但在由`<<`运算构成的链式操作中，各表达式的入栈顺序却是从右向左，只有这样才能实现`<<`运算从左向右进行。所以，先计算的是表达式`i++`的值。因为`i`自增之后无法提供入栈的值，所以另外开辟了一个内存单元`tv75[ebp]`来存放第一个入栈的表达式的值。接着计算`-i`的值，自减运算完成之后，编译器认为`i`的值可以直接作为参数入栈，所以并没有开辟别的内存单元存放这一个入栈参数的值。再接下来计算`++i`，情形跟计算`-i`类似。这些操作完成之后，利用三个`push`语句分别将`tv75[ebp]`处的整数值、`i`和`i`入栈，再三次调用`cout.operator<<`函数将它们输出。所以，程序的最终输出结果是：11、11和10。在程序中，`cout.operator<<`函数名用了一个很长的标识符来表示，该函数执行完成之后，会将对象`cout`的地址存放在寄存器`eax`中作为该函数的返回值。由于在Visual C++中，调用对象的成员函数之前会先将对象的地址存放在寄存器`ecx`中，所以在下一次调用`cout.operator<<`函数之前，会先将`eax`的值送入`ecx`中。

如果生成上述程序的release版，会发现输出结果变成：10、10和10。这是编译器对代码所作的优化导致的结果。因为编译器事先“知道”变量的`i`值为10，所以在将`i++`、`-i`、`++i`依次入栈时，它直接将三次入栈的值计算出来。计算的规则是这样的：由于`i++`的值是自增运算前的值，所以直接将`i`的值（10）入栈；而`-i`和`++i`的值是自减（自增）运算完成之后的值，所以先做完所有这些运算，再将`i`的值入栈。这样就导致连续有3个10入栈。编译器在处理这些自增（自减）运算时，认为它们之间是相互独立的。如果`i`的值不是通过文字常量赋予的，而是通过键盘将10送入`i`，则程序的输出结果依然是：11、11和10。

从上面的程序中，我们可以看出，自增（自减）运算虽然可以使表达式更为紧凑，但很容易带来副作用。过分追求小的技巧正是很多程序缺陷的根源。应该编写那些可读性较好的代码，避免那些看似简练但蕴藏危险的表达式。

假设`i`的值为10，执行语句`i=i++`之后，`i`的值是多少？其实，这样的代码在不同的编译器中有着不同的实现，输出结果是不一样的。所以，编写这样的代码是没有意义的。

3.3 实现函数调用时堆栈的变化情况

函数的正常运行必然要利用堆栈，至少，函数的返回地址是保存在堆栈上的。函数一般要利用参数，而且内部也会用到局部变量，在对表达式进行求值时，编译器还会生成一些无名临时变量，这些变量都是存放在堆栈上的。那么，C++编译器是如何对堆栈进行管理的呢？我们将以Visual C++编译器为例进行研究。考察下面的程序。

```
//// Program 3.3-1 /////
#include <stdio.h>
int MixAdd(int i, char c){
    int keepi;
    char keepc;
    keepi=i;
    keepc=c;
    return keepi+keepc;
}
```

```
int main(){
    printf("%c",MixAdd(4,'A'));
}
```

//// End of Program 3.3-1 ////

在Visual Studio 2005环境下，生成该程序的调试版本(debug版)的汇编代码。在main()函数中，对应于函数调用MixAdd(4,'A');的汇编代码是：

```
push    65          ; 00000041H
push    4
call    ?MixAdd@@YAHHD@Z      ; MixAdd
add    esp, 8
```

其意义非常明确，先是把'A'的ASCII码65压入堆栈，然后把整数4压入堆栈，再用call指令实施函数调用。在X86平台下，堆栈的栈底在高地址处，而栈顶在低地址处。栈指针esp总是指向最后一个入栈的字节。所以，当有新元素被压入堆栈时，栈指针esp的值会减小。在函数调用结束后，add esp,8是将所有压入堆栈的参数“出栈”。可以看到，由于“对齐”的原因，第一个参数实际上也占用了4个字节，所以最后是让8字节的数据出栈。下面是一次实际跟踪的记录：在Visual Studio环境中，切换到汇编代码，然后单步跟踪，可以发现，在执行call语句之前，堆栈指针的值为0x0012FE94，进入到函数MixAdd体内后，esp的值变成0x0012FE90。这说明在进入函数体之前，有4字节的内容入栈。其实，这就是函数的返回地址，也就是汇编代码中call语句的下一条指令的地址。这一地址是函数执行结束后返回主调函数时使用的。

函数MixAdd的汇编代码如下：

```
?MixAdd@@YAHHD@Z PROC           ; MixAdd, COMDAT
    push    ebp
    mov     ebp, esp
    sub    esp, 216      ; 000000d8H
    push    ebx
    push    esi
    push    edi
    lea    edi, DWORD PTR [ebp-216]
    mov    ecx, 54        ; 00000036H
    mov    eax, -858993460   ; ccccccccH
    rep    stosd
    mov    eax, DWORD PTR _i$[ebp]
    mov    DWORD PTR _keepi$[ebp], eax
    mov    al, BYTE PTR _c$[ebp]
    mov    BYTE PTR _keepc$[ebp], al
    movsx  eax, BYTE PTR _keepc$[ebp]
    add    eax, DWORD PTR _keepi$[ebp]
    pop    edi
    pop    esi
    pop    ebx
```

```

mov    esp, ebp
pop    ebp
ret    0
?MixAdd@@YAHHD@Z ENDP ; MixAdd

```

在进入函数MixAdd后，可以马上看到这样三条汇编指令：

```

push ebp
mov ebp, esp
sub esp, xxx

```

这是所有的C/C++函数的汇编代码所共同遵循的规范。其中，`ebp`被称为“帧指针”。每一个函数，都有“属于”函数自身的局部数据，这些数据被称为该函数的“帧”。一帧数据的起始位置由帧指针`ebp`指明，而帧的另一端由栈指针`esp`动态维护。在函数的运行期间，帧指针`ebp`的值保持不变。`push ebp`就是保留主调函数的帧指针，`mov ebp,esp`是建立本函数的帧指针，而`sub esp,xxx`（`xxx`代表某个正整数）则是为函数的局部变量分配空间。在调试版本下，一个C/C++函数即使没有定义一个局部变量，仍然会分配192个字节（16进制数为`0C0H`）的空间，这一部分可以给临时变量使用。如果在函数中定义了局部变量，则会为每个局部变量分配12字节的空间（大于任何基本数据类型变量的需要）。在函数MixAdd中定义了两个局部变量，所以给局部变量和临时变量预留空间的大小是 $192 + 12 + 12 = 216$ 。

所有的参变量和局部变量都分配在栈上，在汇编代码中，函数的局部变量（或参变量）的地址用帧指针加偏移量表示。如，变量`keepi`表示成`DWORD PTR _keepi$[ebp]`，`_keepi$`就是变量`keepi`距帧指针的偏移量。参变量`i`表示成`DWORD PTR _i$[ebp]`。参变量与局部变量的区别在于，参变量是在主调函数的代码中被压入栈的，而局部变量是在进入本函数后分配空间，所以参变量的偏移量（如`_i$`）是正数，而局部变量的偏移量（如`_keepi$`）是负数。在执行函数MixAdd时，先入栈的是参数`c`，后入的是参数`i`，再加上函数返回地址入栈，主调函数的帧指针`ebp`入栈，所以参数`i`的偏移量`_i$`是8，参数`c`的偏移量`_c$`是12。对于局部变量`keepi`和`keepc`而言，先分配空间的是`keepi`，后分配空间的是`keepc`。这两个变量都得到了12个字节的空间。由于单个变量实际上用不了12个字节，所以会先留出4个字节，然后再开始存放变量。所以，`keepi`的偏移量`_keepi$`是 $-(4+4)=-8$ ，`keepc`的偏移量`_keepc$`是 $-(12+4+1)=-17$ 。

接下来的三条汇编语句是：

```

push    ebx
push    esi
push    edi

```

它们保存了本函数可能改变的几个寄存器的值。这些寄存器应该在函数执行结束后恢复到刚进入本函数的时候的值。

再接下来的四条汇编语句是：

```

lea     edi, DWORD PTR [ebp-216]
mov    ecx, 54          ; 00000036H
mov    eax, -858993460   ; ccccccccH
rep    stosd

```

通过设置寄存器`edi`、`ecx`、`eax`的值，利用`stosd`指令，将预分配的空间以双字为单位，全部填充16进制数`0cccccccH`。`stosd`指令将`eax`的内容保存到`edi`所指向的空间，同时在`edi`上加



或减4，同时ecx递减，直到ecx的值变为0。由于edi默认的变化方向是递增，所以可以从低地址到高地址将所有的预留空间填满0cccccccH。这样就可以解释，如果局部变量没有赋初始值，则在调试时可发现它们的值为0cccccccH。其实，0cccccccH正是int 3指令的机器码，int 3是断点中断指令。当程序执行到int3时，便会停下来，从而可以进入单步跟踪调试。

函数MixAdd的最后几步操作，是将运算结果保留在寄存器eax中，然后恢复几个重要寄存器的值，随后通过ret 0指令返回。

要特别注意的是，以上汇编代码是在程序的调试版本下得到的。如果是程序的发行版（release版），编译器会对代码做大幅度的优化，例如，有些参数不通过堆栈传递，而是利用eax等寄存器传递；如果将文字常量传递给函数的参数，有时可由编译器计算函数的返回值，然后直接把计算结果写在主调函数中（从而根本不发生函数调用）；如果没有改变某些寄存器的值，那么开始根本不将它们入栈，等等。如果要清楚地了解C/C++语言的较为规范的底层实现机制，应该阅读调试版本下的汇编代码。

3.4 关于函数参数的默认值

C++语言允许指定函数参数的默认值。在一个程序中，如果需要多次用同样的参数值去调用函数，可将此参数值指定为函数参数的默认值，从而不必每次调用时都给出这个参数值。调用时如果没有给定参数值，则使用默认值。使用函数参数的默认值时要注意以下几点：

(1) 最好在函数的声明中指明参数的默认值，而在函数定义时添加注释。这样会有较好的程序结构。否则，在分离编译模式下可能会出现问题。考察下面的程序。

```
//// Program 3.4-1 /////
/** defaultparm.cpp ***/
#include <iostream>
#include "d.h"
using namespace std;
void f(int x=10){
    cout<<x<<endl;
}
int main(){
    f(29);
    f();
    UseDefault();
}
/** end of defaultparm.cpp **/


/** d.h ***/
void f(int x);
void UseDefault();
/** end of d.h ***/
/** use.cpp **/
```

```
#include "d.h"
void UseDefault(){
    f();
}
/** end of use.cpp ***/
/// End of Program 3.4-1 ///
```

这个程序无法通过编译。因为在源文件use.cpp中，编译器只知道函数f()函数带有一个参数，它不知道函数f()的参数有默认值。如果在头文件d.h中，将函数f()的声明改为void f(int x=10);则仍然会有编译错误。原因是不能够同时在函数声明和函数定义中都指定参数的默认值。所以，较好的做法是，在函数声明中指定函数参数的默认值，而在函数定义时不要再次指定参数的默认值，用注释描述一下就可以了，这样可以保证在任何情况下都能够正常使用。

(2) 如果一个函数有多个参数有默认值，则任何一个不带默认值的参数都要出现在所有带默认值的参数的左边，只有这样才能保证在函数调用时不发生参数匹配时的二义性。例如，定义函数void func(int x=10,int y,int z=11)，那么函数调用func(5,6)中的实参5不知是应与x匹配，还是应与y匹配。

(3) 函数参数的默认值虽然给程序员带来了一定程度的便利，也可能造成函数重载时的二义性。考察下面的程序。

```
/// Program 3.4-2 ///
#include <iostream>
using namespace std;
void show(int x=10, double y=9.8){
    cout << x << " " << y << endl;
}
void show(int x){
    cout << x << endl;
}
int main(){
    show();
    show(3,4.5);
    show(6);      //是调用void show(int)还是调用void show(int,double)?
}
/// End of Program 3.4-2 ///
```

这个程序无法通过编译。在main()函数中，第一函数调用和第二次函数调用都没有问题，但是第三次函数调用（即show(6);）有二义性，不知是调用void show(int)还是调用void show(int,double)。因此，在定义和使用函数参数的默认值时要注意避免这种重载函数之间的冲突。

虽然函数参数的默认值有时会与函数重载发生冲突，但另一方面，却可以利用函数参数的默认值减少重载函数的个数或取消函数的重载。类的构造函数就是这样一种函数。

一般的函数重载，函数参数列表之间的差异有可能是很大的，没有太多的规律可寻。而重载的类的构造函数之间，其参数列表却是有规律可寻的。构造函数是为类对象的初始化准



备的，因此，从构造函数的语义出发，构造函数参数的最大个数应该就是类对象的成员变量的个数。所以，尽管重载的构造函数的参数列表可能不同，但这些参数应该都是最大可能参数集合的子集。所以，在很多情况下，用一个带有所有可能参数的默认值的构造函数，就可以统一不同的构造函数的定义。具体的例子可参见12.2节。

3.5 如何禁止传值调用

按照参数形式的不同，C++有两种函数调用方式：传值调用和引用调用。一般情况下，用某一种类型的变量做函数参数，既可以使用传值的方式，也可以使用传引用的方式，特别是对于基本数据类型，这两种方式在执行效率方面没有多少差别。但是，对于类类型来说，传值调用和引用调用之间的区别却很大，类对象的尺寸越大，这种差别也越大。在一些特殊的应用场合，就希望凡是某个类的对象做函数参数，就只能传引用，而不允许传值。

要实现这个目的，就必须了解传值调用和引用调用之间的区别。它们最大的区别在于：传值调用会在进入函数体之前，在堆栈上建立一个实参的副本，而引用调用则没有这个动作。建立副本的操作是利用拷贝构造函数进行的。因此，要禁止传值调用，就必须在类的拷贝构造函数上做文章。

可以直接在拷贝构造函数中抛出异常，这样就迫使程序员不能使用拷贝构造函数，否则程序总是出现运行时错误。但是，这不是一个好的办法。应该在编译阶段就告诉程序员，不能使用该类的拷贝构造函数。那么，在类的定义中，不定义拷贝构造函数，让编译器“没有”拷贝构造函数可用，这样能否达到目的呢？考察下面的程序。

```
//// Program 3.5-1 ////  
#include <iostream>  
using namespace std;  
class A{  
public:  
    int num;  
    A(){num=5;}  
};  
void show(A a){  
    cout << a.num << endl;  
}  
int main(){  
    A obj;  
    show(obj);  
}  
//// End of Program 3.5-1 ////
```

该程序能顺利地通过编译，并输出：5。看来，不显式定义拷贝构造函数并不能阻止对类的拷贝构造函数的调用，原因是编译器会自动为没有显式定义拷贝构造函数的类提供一个默认的拷贝构造函数。

一个真正可行的、简单的禁止使用拷贝构造函数的方法，是显式定义拷贝构造函数，并



将它的访问权限设定为private（或protected）。将上面的程序改造如下。

```
//// Program 3.5-2 /////
#include <iostream>
using namespace std;
class A{
    A(const A&){};
public:
    int num;
    A(){num=5;};
};

void show(A a){
    cout << a.num << endl;
}

int main(){
    A obj;
    show(obj);
}

//// End of Program 3.5-2 /////

```

这个程序在VC++ 2005环境下编译时，得到如下出错信息：

error C2248：“A::A”：无法访问private 成员(在“A”类中声明)

这样，就成功阻止了在函数调用时，类A的对象（实参）以传值的方式进入函数参数。要保持函数show()在main()中的调用形式不变，而且能正常工作，必须将其改成：

```
void show(const A &a){
    cout << a.num << endl;
}
```

实际上，C++标准库中的istream类和ostream类就是以这种方式阻止传值调用的。对于基本数据类型而言，由于没有拷贝构造函数可用，所以不能阻止这类参数的传值调用。当然，在实际应用中，也没有必要阻止基本数据类型参数的传值调用。

另外，禁止使用拷贝构造函数实际上在大多数情况下也阻止了类对象作为函数的返回值。换句话说，如果函数确实要将类对象的信息返回到函数外部，只能返回类对象的引用。

3.6 定义和使用可变参数函数

一般的C/C++函数，其参数个数在函数定义的时候就确定了。在函数调用的时候，传递给函数的实参要和函数的定义相一致，否则会发生编译错误。但是，在C语言中就允许定义参数个数可变的函数，典型的如printf()和scanf()。可变参数函数可以简称为va函数（variable argument function）。在C++中，由于函数重载的出现，使得可变参数的函数在大多数情况下变得没有必要，但C++中语言仍然允许沿用C语言中的方法定义和使用可变参数函数。下面是一个定义和使用可变参数函数的例子。函数PrintAll()将所有的整型实参依次输出到标准输出设备上。



```
/// Program 3.6-1 ///
#include <iostream>
using namespace std;
void PrintAll(int n1, ...){
    int *ptr;
    ptr = &n1;
    while(*ptr){
        cout << *ptr << endl;
        ptr++;
    }
}
int main(){
    PrintAll(3,4,5,0);
}

/// End of Program 3.6-1 ///
```

程序的执行结果是：

```
3
4
5
```

由此可以看出，要定义和使用可变参数的函数，要解决这样几个问题：一是可变参数函数的函数头的书写形式；二是如何依次取出各个参数；三是如何判断实际上有多少参数传递给了该函数。

由于在函数体内部，必须先获取某个参数的地址，然后根据其他参数与该参数的相对位置来获取其他参数。因此，定义可变参数的函数，该函数必须至少带有一个固定参数（超过一个也可以），然后才是可变参数。带有一个固定参数的可变参数函数头具有如下形式：

```
return_type func_name(para_type para1, ...)
```

如果函数带有多个固定参数，则在写完所有固定参数之后，再书写…。

确定函数的可变参数的实际个数和各个参数的实际类型，是一件困难的事情。原因是在定义函数的时候，无法预料实际调用时的情形。而且在函数参数入栈的过程中，也没有保留参数个数和参数类型的信息。因此，只能从逻辑上做某种约定，然后根据此约定进行判断。例如，在上面的程序中，约定所有传入函数的参数都是整型，以0作为最后一个参数（这样可以确定调用时参数的实际个数）。在系统提供的函数printf()中，第一个参数是一个字符串，该字符串的内容决定了参数的个数和每个参数的类型。

C编译器为了统一在不同的硬件平台上的实现，增加代码的可移植性，提供了一系列宏来屏蔽硬件环境不同带来的差异。ANSI C标准下，va的宏定义在stdarg.h中，它们有：va_list, va_start(), va_arg(), va_end()。下面的例子使用了va的相关宏定义来实现可变参数函数。

```
/// Program 3.6-2 ///
#include <iostream>
using namespace std;
#include <stdarg.h>
```



```

int SqSum1(int n1, ...){
    va_list arg_ptr;
    int nSqSum=0,n=n1;
    va_start(arg_ptr, n1);
    while (n > 0){
        nSqSum += (n * n);
        n = va_arg(arg_ptr, int);
    }
    va_end(arg_ptr);
    return nSqSum;
}

int SqSum2(int n1, ...){
    va_list arg_ptr;
    int nSqSum=0,n;
    va_start(arg_ptr,n1);
    while (n1 > 0){
        n = va_arg(arg_ptr, int);
        nSqSum += (n * n);
        n1--;
    }
    va_end(arg_ptr);
    return nSqSum;
}

int main(){
    int nSqSum = SqSum1(1, 1, 1, 1, 0);
    cout << nSqSum << endl;
    nSqSum = SqSum2(4, 1, 1, 1, 1);
    cout << nSqSum << endl;
}
/// End of Program 3.6-2 ///

```

在这个程序中实现了两个可变参数函数：SqSum1()和SqSum2()。它们的功能都是计算传递给函数的各个整型参数的平方和，只不过它们对函数参数的个数采用了不同的约定：SqSum1()约定最后一个参数为0；SqSum2()则利用第一个参数指明后继参数的个数。程序的输出结果是：

4

4

打开stdarg.h文件，找一下va相关的宏定义，发现不单单只有一组，但是在各组定义前都会有条件编译指令。条件编译指示的是不同硬件平台和编译器下用怎样的va宏定义。比较一下，不同之处主要在偏移量的计算上。在X86平台上的相关宏定义如下：

`typedef char * va_list;`

```
#define _INTSIZEOF(n) ((sizeof(n) + sizeof(int)-1) & ~(sizeof(int)-1))
#define va_start(ap,v) (ap = (va_list)&v + _INTSIZEOF(v))
#define va_arg(ap,t) (*t*)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t))
#define va_end(ap) (ap = (va_list)0)
```

下面解释一下这些代码的含义：

①把va_list定义成char*，是因为在X86平台上，字符指针类型可以用来指向任何一个内存单元的地址（因为sizeof(char)==1）。而在有些机器上va_list是被定义成void*的。

②定义_INTSIZEOF(n)主要是为了在某些需要内存对齐的系统中，获取某个参数的实际内存大小。从宏的名字可看出是同整型量对齐。一般sizeof(int)==4，则参数在内存中的地址都为4的倍数。通过计算可知，如果sizeof(n)在1~4之间，那么_INTSIZEOF(n)==4；如果sizeof(n)在5~8之间，那么_INTSIZEOF(n)==8。

③va_start的定义为 &v+_INTSIZEOF(v),这里&v是最后一个固定参数的起始地址，再加上其实际占用大小后，就得到了第一个可变参数的起始内存地址。所以我们运行va_start (ap, v)以后,ap指向第一个可变参数所在的内存地址,有了这个地址，就可以陆续得到其他参数的地址。

④va_arg()的作用是获取参数列表指针当前所指的参数的值。通过va_start取得了第一个可变参数的地址之后，va_arg()的任务就是根据指定的参数类型取得本参数的值，并且把指针调到下一个参数的起始地址。

⑤va_end宏的解释：x86平台定义为ap=(char*)0;使ap不再指向堆栈,而是变成空指针。

补充说明两点：

①在intel+windows的机器上，函数栈的方向是向下的，栈顶指针的内存地址低于栈底指针，所以先进栈的数据是存放在内存的高地址处。

②在VC等绝大多数C编译器中，默认情况下，参数进栈的顺序是由右向左的，因此，参数进栈以后，最后一个固定参数的地址位于第一个可变参数之下，并且是连续存储的。

3.7 关于函数指针

函数指针实际上代表一个函数在内存中的入口地址。当然，要正确地定义一个函数指针，还必须同时指明函数的返回值类型和函数的参数列表。

定义函数指针可采用如下的形式：

ReturnType (*pfunc)(var-list)

其中ReturnType是指函数的返回值类型，而pfunc则是函数指针，var-list是函数的参数列表。下面是一个定义和使用函数指针的例子。

```
/// Program 3.7-1 ///
#include <iostream>
using std::cout;
using std::endl;
int PrintVal(int i){
    cout << i << endl;
    return 0;
```

```

}

int Add(int i,int j){
    return i+j;
}

void Compare(int i,int j){
    if(i==j)
        cout << i << "==" << j << endl;
    else if(i>j)
        cout << i << ">" << j << endl;
    else
        cout << i << "<" << j << endl;
}

typedef int (*pFunc)(int);

int main(){
    pFunc p1=PrintVal;
    int (*p2)(int,int);
    p1(7);
    p1=&PrintVal;
    (*p1)(8);
    p2=Add;
    cout << p2(4,5) << endl;
//    p2=Compare;           //直接赋值会导致编译错误
//    p2=reinterpret_cast<int(*)(int,int)>(Compare);
//    p2(4,5);
}

/// End of Program 3.7-1 ///
程序的执行结果是:

```

7
8
9
4<5

定义和使用函数指针，要注意以下几个问题：

①由于定义函数指针的语法形式比较复杂，所以常常借助于**typedef**声明将函数指针类型简化。如在上面的程序中，**pFunc**就是一种函数指针类型，它代表**int (*)(int)**。利用**pFunc**定义函数指针**p1**，则在定义的同时为它赋初值就显得非常清晰。否则，初学者可能这样为函数指针赋初值：**int (*p1= PrintVal)(int)**，从而导致编译错误。正确的写法是：**int (*p1)(int) = PrintVal**，但其可读性较差。

②函数名本身就代表了函数在内存中的入口地址，所以在为函数指针赋值时，**p1=PrintVal**和**p1=&PrintVal**两种写法都是正确的。在利用函数指针实现函数调用的时候，**p1(8)**和**(*p)(8)**两种用法也都是正确的。



③为指针函数赋值时，函数的原型必须与定义函数指针时的原型一致，否则会导致编译错误。但是，在某些特殊情况下，可以利用`reinterpret_cast`运算在不同类型的函数指针之间进行转换。当然，这种转换必须非常谨慎，只有在有充足的理由下才可以偶尔为之。在上面的程序中，采用C风格的强制类型转换，即利用`p2=(int(*)(int,int))Compare;`，也可以实现将函数名`Compare`赋值给`p2`的目的。但是，应该尽量避免使用功能过于强大的C风格的强制类型转换。

④有一种函数叫做“回调函数”（callback function）。回调函数是一个定义了函数的原型，函数体则交由第三方来实现的一种动态应用模式。在实现调用时，先将回调函数的地址作为参数之一传递给主调函数，在主调函数内部通过函数指针调用回调函数。回调函数的机制打破了主调函数与被调函数静态绑定的限制，为用户提供了一种充分利用操作系统功能（或可重用代码）的方便手段。因此，可以将回调函数理解成通过函数指针调用的函数。

在上面的程序中，`p2=Add`这条语句从形式上看，与一般的赋值语句没有什么不同，如果不结合上下文，无法判断`p2`和`Add`到底是什么数据类型。在这里，`p2`和`Add`甚至算不上是“数据”，在概念上它们代表“代码”。在函数指针作为函数的参数进行传递的时候尤其要注意这一点。下面就是一例。

```
/// Program 3.7-2 ///
#include <iostream>
class DoubleValue{
public:
    double GetValue(){
        return 3.7;
    }
    static int RetInt(){
        return 5;
    }
};
typedef int (*pInt)();
double operator+(DoubleValue &d,pInt pfunc){
    return d.GetValue()+pfunc();
}
int main(){
    DoubleValue obj;
    pInt p1=obj.RetInt();
    std::cout << obj+p1 << std::endl;
}
/// End of Program 3.7-2 ///
```

程序的输出结果是8.7。表达式`obj+p1`的计算靠的是对全局的操作符函数`operator+`进行重载的结果，而函数指针类型`int(*)()`正是该操作符函数的参数之一。所以，函数指针使得函数可以像普通变量一样被传递并参与运算，大大提高了程序的计算能力。我们在进行标准输出时，常用到这样的语句：`cout << endl;`，其实，`endl`就是一个函数（模板函数）。也就是说，操作符`operator<<`接收一个函数指针作为它的参数。

当函数指针作为另一个函数的参数传递时，对函数指针的声明可以采用“显式”的方式进行，也可以采用“隐式”的方式进行。下面是一个具体的例子。

```
/// Program 3.7-3 ///
#include <iostream>
using namespace std;
int f(){
    return 1;
}
void invoke1(int (*func)()) { //显式声明函数指针
    cout << (*func)() << endl;
}
void invoke2(int func()){ //隐式声明函数指针
    cout << func() << endl;
}
int main(){
    invoke1(f);
    invoke2(f);
}
/// End of Program 3.7-3 ///
```

函数invoke1()和invoke2()的参数表面上不同，但实际上它们是完全一样的，都是一个类型为int(*)()的函数指针。只不过invoke1()的参数显式采用了函数指针的声明方式，而invoke2()则是将函数原型放在了参数应该出现的位置。由于出现在参数列表中的函数原型只可能用于描述某个参数，所以它描述的是一个函数指针。

函数指针在C语言中就已经存在，对外部函数而言，C++语言继续沿用C语言中对函数指针的定义和使用规范。在上面的程序中，由于类的静态成员函数可以理解成“作用域受限的外部函数”，所以DoubleValue::RetInt可以直接赋给函数指针p1。

在C++语言中，由于面向对象机制的引入，程序中不但有外部函数，还有类对象的成员函数。对于类的非静态成员函数而言，函数指针要以对象的“成员指针”的形式定义和赋值。考察下面的程序。

```
/// Program 3.7-4 ///
#include <iostream>
class A{
public:
    int RetInt() {return 100;}
};
class B{
public:
    int (A::*pFun)(void);
    void Print(A &obj){
        int val = (obj.*pFun)();
    }
}
```

```

    std::cout << "Val = " << val << std::endl;
}
};

int main(){
    A a;
    B b;
    b.pFun = &A::RetInt;
    b.Print(a);
}

/// End of Program 3.7-4 ///

```

程序的执行结果是：

Val = 100

在class B对象中，pFun是一个类型为函数指针的成员，它指向class A中的函数原型为int funcname(void)的成员函数。在为pFun赋值的时候，并不需要指定函数所操作的对象。但是在调用函数指针所代表的函数时，必须同时指明函数所操作的对象，这是利用对象的“成员指针”这种特殊的语法形式实现的。表达式(obj.*pFun)()的含义是：pFun是一个指向类B对象obj的成员函数的指针，通过该表达式调用该函数指针所代表的函数。需要注意的是：取某个类的成员函数的地址，必须显式使用取地址运算符&。这与外部函数取入口地址的要求不同，原因是避免与类的普通静态成员相混淆。

3.8 关于函数重载

所谓函数重载，是指在相同的作用域中定义的函数，其函数名相同，而参数列表不同的现象。参数列表不同是指：要么参数的个数不同，要么是对应位置的参数类型不同。重载的函数正是通过不同的参数列表来确定被调用函数的实际版本的。使用函数重载的一种典型情形是：同样的操作，需要针对不同的数据类型进行，这时就可以使用函数重载来达到统一操作名称的目的。例如，要从两个变量中返回其中较大的一个值，可以编写int Max(int,int)、double Max(double,double)等函数，这些函数的函数名相同，参数类型不同，执行类似的操作。

在C++语言中使用函数重载，要注意几个要点。

(1) 不同作用域中定义的函数可能形成隐藏(Hide)，但不会形成重载(Overload)

在C++语言中，严格规定只有在同一作用域中定义的函数才能形成重载。例如，同为外部函数，同为一个类的成员函数等。如果定义函数的作用域发生嵌套，则内部作用域中定义的函数将隐藏外围作用域中定义的函数。考察下面的程序。

```

/// Program 3.8-1 ///
#include <iostream>
using namespace std;
void func(char *s){
    cout << "This is a global function with name " << s << endl;
}
class A{

```

```

void func(){
    cout << "This is a member function of A" << endl;
}
public:
void useFunc(){
//    func("func");           //A::func()将外部函数func(char*)隐藏
    func();
    ::func("func");
}
};

class B: public A{
public:
void useFunc(int i){
    cout << "In B's useFunc(), i=" << i << endl;
}
};

int main(){
A a;
a.useFunc();
B b;
// b.useFunc();      //A::useFunc()被B::useFunc()隐藏
b.useFunc(6);
}

/// End of Program 3.8-1 ///

```

程序的运行结果是：

```

This is a member function of A
This is a global function with name func
In B's useFunc(), i=6

```

在类A的成员函数useFunc中，直接调用外部函数func(char*)将导致编译错误，原因是：尽管A::func()与外部函数func(char*)同名，且参数列表不同，但并不形成重载。在类域A中，类A的成员函数func()将外部函数func(char*)隐藏，要调用外部函数，必须显式使用作用域指示符::，如上例所示。

同样地，在类域B中，类B的成员函数useFunc(int)将其基类的成员函数useFunc()隐藏，如果要调用类A中成员函数useFunc()，也必须指明其作用域。在上例的main()函数中，可以这样调用：b.A::useFunc()。

(2) 类的静态成员函数与实例成员函数可以形成重载

由于类的实例成员函数可以自由调用类的静态成员函数，而当一个函数调用发生时，无法仅从形式上判断调用的到底是一个实例成员函数，还是一个静态成员函数，所以，类的静态成员函数与实例成员函数可以形成重载。见下面的例子。

/// Program 3.8-2 ///

```
#include <iostream>
using namespace std;
class A{
    static void func();
    void func();
}
/// End of Program 3.8-2 ///
```

此程序无法通过编译。一个类的静态成员函数与实例成员函数同名时，其参数列表一定要有所不同，否则在发生实际调用时，有时无法判断该调用这个函数的哪一个版本。

(3) 函数重载与函数覆盖的比较

函数重载 (Overload) 与函数覆盖 (Override) 是两个不同的概念。它们的不同体现在以下几个方面：

① 函数重载发生在同一作用域的两个函数之间；而函数覆盖则发生在基类的成员函数和派生类的成员函数之间。

② 形成重载的两个函数，要求其函数名必须相同，而参数列表一定不同，对返回值类型不作要求；而形成覆盖的两个函数，则要求函数名相同，参数列表和返回值类型也相同。

③ 类的静态成员函数可与非静态成员函数发生重载，而函数覆盖只可能发生在基类的非静态成员函数与派生类的非静态成员函数之间。

④ 发生函数调用时，重载函数的入口地址是在编译时确定的（静态联编），而发生虚调用（使用函数覆盖）时，函数的入口地址是在运行时决定的（动态联编）。

(4) 如何判断被调用函数的实际版本

C++编译器是如何判断调用的函数是重载函数中的哪个版本的呢？当然是通过函数的参数列表。但识别的过程并非想象中的那么容易，其中涉及到参数的等级划分、参数转换等方面的操作。假设有下面一组函数：

```
void S();
void S(int);
void S(double , double = 1.2);
void S(const char*,const char*);
void Max(int,int);
//.....
int main(){
    S(2.4);
    return 0;
}
```

函数调用 S(2.4); 与 S(); S(int); S(double , double = 1.2); S(const char* , const char*); 的声明在同一作用域，因此实际调用版本应该从这些函数中选择。

编译器判断函数调用版本的第一步，是确定该调用中所考虑的重载函数的集合，该函数集合被称为候选函数 (candidate function)。所谓候选函数就是与被调用函数同名的函数。上面的前四个函数都可以成为候选函数，而唯有 Max(int , int) 被排除在外。

编译器判断函数调用版本的第二步，就是从候选函数中找出可行函数 (viable function)。

可分为两个动作进行,第一个动作是找出函数参数个数与调用的函数参数个数相同的函数(如S(int)),或者可行函数的参数可以多一些,但是多出来的函数参数都要有相关的默认值(如S(double, double =1.2);)。第二个动作是根据参数类型的转换规则将被调用的函数实参转换成候选函数的实参。这里本着充分利用参数类型转换的原则,换句话说,尽可能地利用参数类型转换。上面的函数中只有两个是可行函数,它们分别是S(int); S(double, double)。

如果依照参数转换规则没有找到可行函数,则该调用就是错误的,即没有函数与调用匹配,属于无匹配情况(no match function)。

编译器判断函数调用版本的第三步,是从可行函数中选出最佳可行函数(best match situation)。在最佳可行函数的选择中,从函数实参类型到相应可行函数参数所用的转化都要划分等级,根据等级的划分(ranked),最后选出最佳可行函数。

最佳可行函数即编译器要调用的函数。如上面的S(double,double=1.2)就是函数调用S(2.4)的最佳可行函数。

3.9 关于操作符重载

操作符重载是C++语言为了提高程序的可读性而引入的一项重要机制。在传统的数学演算中,我们经常使用各种符号(如+, -, ×, /等)来表示某种具有特定含义的运算,这些符号被称为运算符或操作符。在C语言中,可以使用一些基本的操作符来实现某些功能,如可以用+, -, *, /来构成数学表达式,用[]取数组中的元素,用*来获取指针所指单元的内容等。但是,这些操作符只能作用于特定类型的操作数,否则会出现语法错误。如定义一个整型变量int i,表达式i[0]就是一个不能通过编译的错误表达式。

在C++中,由于用户可以自定义类类型,而这些类类型数据(对象)之间的运算有时也具有明确的含义,如定义复数类型,两个复数之间的加、减、乘、除运算都有着明确的数学含义。如果要使用操作符来表达用户自定义类型的对象之间的运算,就必须对操作符原有的功能进行扩展。操作符重载正是在这种背景下产生的。要正确地理解和使用操作符重载,必须注意以下几个要点。

①引入操作符重载的目的,就是为了提高程序的可读性。程序员可以用接近数学演算的方式书写程序,这样的程序便于理解和维护。

②只有C++预定义的操作符集中的操作符才可以被重载,程序员不可以“发明”新的操作符。

③操作符本质上等同于一个函数。所以,C++把操作符理解成“操作符函数”。操作符函数名是由operator关键字后跟操作符构成,如加法的操作符函数名是operator+。

④操作符所带操作数的个数不允许改变,操作符之间的优先级不允许改变。

⑤可以有两种形式进行操作符重载。一是把操作符函数视为一个外部函数,所有的操作数作为这个函数的参数。由于操作符函数往往需要访问类对象的私有成员,所以一般将这个外部函数声明为参数类型的友元函数。另一种是将操作符函数作为类的成员函数看待,这时操作符的第一操作数一定是该类的对象。例如,假设有一个加法表达式a+b,如果是按第一种方式重载,那么是将表达式重新解释成operator+(a,b);如果是按第二种方式重载,则是将表达式重新解释成a.operator+(b)。

⑥不允许为基本数据类型定义其他的操作。换句话说,对操作符进行重载,其操作数中



至少要有一个用户自定义的类类型或枚举类型。以下程序对操作符operator+进行了重载，它接收一个枚举类型的参数和一个整型参数，其功能是计算某一天的若干天之后是星期几。

```
//// Program 3.9-1 ////  
#include <iostream>  
using namespace std;  
enum Week{ Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saterday};  
Week operator+(Week day, int i){  
    int n = (int)day;  
    n += i;  
    n %= 7;  
    return (Week)n;  
}  
void PrintDay(Week day){  
    switch(day){  
        case Sunday: cout << "Sunday" << endl; break;  
        case Monday: cout << "Monday" << endl; break;  
        case Tuesday: cout << "Tuesday" << endl; break;  
        case Wednesday: cout << "Wednesday" << endl; break;  
        case Thursday: cout << "Thursday" << endl; break;  
        case Friday: cout << "Friday" << endl; break;  
        case Saterday: cout << "Saterday" << endl; break;  
    }  
}  
  
int main(){  
    PrintDay(Monday+1234);  
}  
/// End of Program 3.9-1 ///
```

程序的输出结果是：Wednesday。在本例中，由于枚举类型中不允许定义成员函数，所以只能以外部函数的形式对操作符operator+进行重载。

⑦有些操作符是不允许重载的，如.、.*、::、?:等。

⑧进行操作符重载之后，操作符函数可以用两种方式调用。一是隐式调用，如Program 3.9-1中的表达式Monday+1234；另一种是显式调用，将Program 3.9-1中的表达式Monday + 1234改写成operator+(Monday+1234)，程序的语义不发生变化，执行结果也是一样的。再下面的程序。

```
//// Program 3.9-2 ////  
#include <iostream>  
using namespace std;  
class Complex{  
    double real;
```

```

double image;
public:
Complex(double r=0.0,double i=0.0){
    real=r;
    image=i;
}
void Show(){
    cout << real << "+" << image << "i" << endl;
}
Complex operator+(const Complex&);
};

Complex Complex::operator+(const Complex& c){
    return Complex(real+c.real,image+c.image);
}

int main(){
    Complex c1(1.1,2.0);
    Complex c2(2.2,3.0);
    Complex c3 = c1+c2;
    c3.Show();
    Complex c4 = c1.operator+(c2);
    c4.Show();
}
/// End of Program 3.9-2 ///

```

此程序定义了一个复数类，并将operator+作为该类的成员函数进行了重载。在调用操作符函数时，表达式c1+c2和表达式c1.operator+(c2)都是合法的，结果都是3.3+5i。不过，由于操作符重载的目的是提高程序的可读性，所以一般情况下采用隐式的操作符函数调用。

⑨可以将类型（包括基本数据类型和用户自定义类型）作为特殊的操作符，即类型转换操作符，在某个类中对这个类型操作符进行重载，就可以达到类型转换的目的。如下面的程序。

```

/// Program 3.9-3 ///
#include <iostream>
using namespace std;
class A{
    int m;
public:
    A(int i){m=i;}
    void Show(){
        cout << m << endl;
    }
};

```

```

class B{
    int n;
public:
    operator A(){
        return A(n);
    }
    B(int i){n=i;}
};

int main(){
    B b(5);
    A a=b; // A a=A(b);或者A a=(A)b;是等价的表达式
    a.Show();
}

/// End of Program 3.9-3 ///

```

在类B中定义类型转换操作符函数operator A(), 就可以实现将类B的对象转换成类A的对象（实际上是根据类B的对象构造出一个新的类A对象）的目的。要注意的是，类型转换操作符函数既不需要参数，也不需要返回值类型（因为返回值必然是操作符本身）。如果要将类B的对象转换成类A的对象，除了可以在类B中定义类型转换操作符函数operator A()外，还可以在类A中定义构造函数A(B b)（参见2.6节中的Program 2.6-7）。不过，这两种手段只能采用其中一种，不然会产生类型转换的二义性。

⑩除了对()操作符外，对其他重载操作符提供参数的默认值都是非法的。

3.10 类的成员函数与外部函数（静态函数）的区别

在C语言中，所有的函数都是外部函数（静态函数）。而在C++中，从面向对象的角度来说，由于封装的作用，函数是“属于”某个类的。类的静态成员函数在调用特性上与外部函数相同，因为它不依赖于类的实例。

因此，类的非静态成员函数与外部函数（静态函数）的本质区别，就是前者要访问类的实例，因此在函数调用之前，必须将类对象的地址传递给该函数。从C++的语法上看，类对象地址的传递是“隐式”进行的，由编译器生成的低级代码来实现，C++程序员没有必要关心这个问题。在类的非静态成员函数中，可以使用this指针，该指针就是类的非静态成员函数所操作的当前对象的地址。

理解类的非静态成员函数与外部函数（静态函数）的区别，要注意以下几点。

(1) 外部函数（静态函数）的地址可以赋给void*型指针，但类的非静态成员函数不行。

类的非静态成员函数也是函数，也就是一段可执行代码，因此它必然有入口地址。它的特殊性在于，在函数代码中要修改类的对象的状态，因此在函数调用之前，必须先生成该类的对象，并且将对象的地址传递给该函数。所以，调用类的非静态成员函数有两个前提条件：已知函数的入口地址和已知类对象的首地址。如果可以将类的非静态成员函数的入口地址赋给void*型指针，那么就可以进一步将void*型指针转换成函数指针，从而抛开对象的地址而直接调用类的非静态成员函数。这样做破坏了面向对象的封装性，后果将是严重的。当函数

代码访问对象的数据成员时必然造成对内存的非法访问，从而出现运行时错误或逻辑错误。见下面的例子。

```
//// Program 3.10-1 /////
#include <iostream>
using namespace std;
void print(){
    cout << "External Function" << endl;
}
class A{
public:
    void print(){
        cout << "A's Member Function" << endl;
    }
};
typedef void (*fun)();
int main(){
    fun p;
    void *v;
    v = (void*)&print;
    p=(fun)v;
    p();
    v = (void*)&A::print; //这一句出现编译错误
    p=(fun)v;
    p();
}
/// End of Program 3.10-1 ///
```

此程序的本意，是通过函数指针来调用外部函数和类的非静态成员函数。由于类的非静态成员函数在调用时，要同时提供类对象的首地址信息，因此`v = (void*)&A::print;`在编译时出现错误，以防止对类的非静态成员函数的非法调用。

(2) 可通过内联汇编的方式获取类的非静态成员函数的入口地址

内联汇编（Inline Assembly）是C++提供的一种手段，它有两个作用：①程序的某些关键代码直接用汇编语言编写可提高代码的执行效率。②有些操作无法通过高级语言实现，或者实现起来很困难，必须借助于汇编语言达到目的。用内联汇编获取类的非静态成员函数的入口地址就属于第二种情况。把Program 3.10-1重新改写以后，生成如下程序。

```
//// Program 3.10-2 /////
#include <iostream>
using namespace std;
void print(){
    cout << "External Function" << endl;
}
```

```

class A{
public:
    void print(){
        cout << "A's Member Function" << endl;
    }
};

typedef void (*fun)();

int main(){
    fun p;
    void *v;
    v =(void*)print;
    p=(fun)v;
    p();
    _asm{
        lea eax,A::print
        mov v, eax
    }
    p=(fun)v;
    p();
}

```

//// End of Program 3.10-2 ////

程序中以`_asm`开头的语句块就是内联汇编代码，它将类的非静态成员函数的入口地址送入了变量`v`，而`v`进一步转换成函数指针`p`，通过`p`完成对类的非静态成员函数的调用。程序的输出结果是：

External Function

A's Member Function

程序中使用了`v =(void*)print;`来获取函数的入口地址。在C++中，函数名本身就代表了函数的入口地址。如果为了明确起见，在函数名前显式地使用`&`来获取函数的入口地址，也是允许的。

虽然获取了类的非静态成员函数的入口地址，并且成功地进行了调用。但是Program 3.10-2存在严重问题。如果函数`A::print()`需要访问`A`类对象的数据成员，就会引发运行错误。

在Visual C++中，类的成员函数在调用之前，编译器生成的汇编码将对象的首地址送入寄存器`ecx`。因此，如果通过内联汇编代码获取了类的非静态成员函数的入口地址，在调用该函数之前，还应该将类对象的首地址送入寄存器`ecx`，只有这样才能保证正确的调用。见下面的例子。

//// Program 3.10-3 ////

```

#include <iostream>
using namespace std;
typedef void (*fun)();

```

```

class A{
    int i;
public:
    A(){i=5;}
    void print(){
        cout << i << endl;
    }
};

void Invoke(A& a){
    fun p;
    _asm{
        lea eax,A::print
        mov p, eax
        mov ecx,a      //将对象地址送入ecx寄存器
    }
    p();
}

/*
void Invoke(A a){
    fun p;
    _asm{
        lea eax,A::print
        mov p, eax
        lea ecx,a      //将对象地址送入ecx寄存器
    }
    p();
}
*/
int main(){
    A a;
    Invoke(a);
}

/// End of Program 3.10-3 ///

```

程序的输出结果是5。在函数Invoke()当中，如果去掉mov ecx,a这条语句，在运行时会抛出异常。原因是在函数A::print()当中访问了A类对象的成员变量i。如果将Invoke()函数替换成被注释掉的版本，也可以顺利输出5。不过，由于传递的不是对象的引用，所以在汇编代码中必须使用lea ecx,a，将放在堆栈上的实参的副本的地址送入寄存器ecx。



3.11 关于内联函数

内联函数是为了提高程序的执行效率而引入的。普通的函数（非内联函数）在被调用的时候，要经历传递参数、传递返回地址、保存重要寄存器的值、函数调用、恢复重要寄存器的值、返回、清理堆栈等一系列步骤。在这些步骤中，只有函数调用本身是最重要的，其他步骤只是为了实现函数调用而不得不采取的配合行动。

在C语言中，可以定义一个宏来避免函数调用，这样可以省去参数入栈、返回地址入栈、返回、清理堆栈等操作，从而提高程序的执行效率。宏代码本身不是函数，但使用起来像函数。使用宏很容易出错，因为宏在展开时并不进行语法和语义的检查，只是进行机械的源代码替换。这样，在宏的参数中带有自增（自减）运算时，宏展开的意义和程序员的本意常常是背离的。也由于机械替换的原因，宏展开并不考虑使用宏时的上下文环境，常常造成一些意想不到的边界效应。在C++语言中，类的友元函数可以直接访问这个类的对象的私有成员。但是要将这样的友元函数改写成宏却是一件办不到的事情。

在定义和使用内联函数时，C++编译器要像对待其他非内联函数一样，检查内联函数是否存在语法错误。这时，内联函数与普通函数没有任何区别。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）。如果正确，内联函数的实现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++语言的内联函数机制既具备宏代码的效率，又具有普通函数的适应性和安全性。所以在C++程序中，应该尽可能使用内联函数取代宏代码，只有在特殊的情况下，才使用宏。例如，“断言”assert是仅在Debug版本起作用的宏，它的使用是为了不在程序的Debug版本和Release版本之间引起差异。又如，如果用到一些无法直接在高级语言中完成的操作，需要借助内联汇编来完成，那么这些内联汇编代码也只能用宏的形式来编写，而不能写成内联函数，如Program 8.10-2中的宏ShowFuncAddress。

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。如果一个成员函数是在一个类的内部定义的，那么它将自动地被当做内联函数。对于用户而言，不需要知道一个函数是否为内联函数。原因有两个：一是函数是否内联并不影响函数的使用方式；二是即使将一个函数定义为内联函数，编译器也不见得会真正将该函数处理成内联函数。所以，在声明函数时使用inline关键字是没有必要的。

内联函数的主要目的是提高程序的执行效率，它是通过省去调用开销而做到这一点的。但是这样做的代价是：一般而言，使用内联函数会增加执行代码的体积。因为每一次调用内联函数时，都会在函数调用处将函数代码复制一份，而函数代码的长度一般都大于实现函数调用所需的代码长度。

如果函数是一个复杂函数（需要较多执行时间或函数代码较长），那么函数调用的开销占整个函数执行时间的比例就显得微不足道，或者将它实现成内联函数会大大增加执行代码的长度。在这种情况下，就不宜将此函数实现为内联函数。所以，尽管可以将一个函数定义为一个内联函数，但编译器会根据某些准则决定是否真正将它实现为一个内联函数。一般情况下，这样的函数不能成为内联函数：



①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函数体内出现循环。由于一般情况下，循环的执行次数是不固定的，所以可认为执行函数体代码的时间要比函数调用的开销大得多。

③递归调用。函数的递归调用就是控制流返回到函数的入口地址处继续执行，而内联函数是没有固定的函数入口的，所以递归调用的函数不能是内联函数。

④若是将函数指针指向某个函数，那么这个函数也不能是内联函数。

在编程实践中，怎样知道一个定义成内联函数的函数是否最终实现了内联呢？这只能从编译产生的执行代码入手才能知道。考察下面的程序。

```
/// Program 3.11-1 ///
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){ cin >> num; }
    int InlineFunc(int i){ return i+num+5; }
};
int main(){
    int i;
    A a;
    cin >> i;
    cout << a.InlineFunc(i) << endl;
}
/// End of Program 3.11-1 ///
```

函数InlineFunc()被定义成一个内联函数（默认定义）。在Release版中考察汇编代码，发现语句cout << a.InlineFunc(i) << endl;所对应的汇编代码是：

```
mov edx, DWORD PTR _a$[esp+12]
mov eax, DWORD PTR _i$[esp+12]
mov ecx, DWORD PTR __imp_?endl@std@@YAAV?$basic_ostream@DU?$char_traits@D@std@@@1@AAV21@Z
push ecx
mov ecx, DWORD PTR __imp_?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A
lea  eax, DWORD PTR [eax+edx+5]
push eax
call DWORD PTR __imp_??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@H@Z
mov ecx, eax
call DWORD PTR __imp_??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QAEAAV01@P6AAV01@AAV01@@Z@Z
```



其中，对函数调用a.InlineFunc(i)只对应了一条汇编语句，即：

```
lea eax, DWORD PTR [eax+edx+5]
```

寄存器eax中存放的是变量i的值，寄存器edx中存放的是变量a.num的值，这条语句的执行结果是将表达式eax+edx+5的值存入eax，而这正是函数a.InlineFunc(i)的返回结果。这说明函数A::InlineFunc()是真正被编译器实现为内联函数。

需要特别注意的是，考察一个函数是否被实现为内联函数，只能在程序的release版中进行考察，程序的调试版不会实施内联函数代码的复制。另外，有些函数尽管没有被定义成内联函数，但由于程序优化的作用，也有可能省去函数调用的开销，也就是说，自动被编译器实现为内联函数，这也体现了现代编译器的优越性能。

3.12 函数的返回值放在哪里

调用函数时，函数的返回值到底放在什么地方呢？从概念上说，函数的返回值应该放在被调用函数的运行结束之后，主调函数可以有效访问的地方。因此，一个函数的返回值应该是存放在由主调函数开辟的栈空间上。

随着计算机硬件的发展，CPU的通用寄存器的字长在增加，个数也在增多。因此，像函数的参数的传递，函数的返回值的传递等工作，在很多情况下都可以通过寄存器来完成，并非一定要借助于计算机的存储器。考察下面的程序。

```
//// Program 3.12-1 ////
```

```
#include <stdio.h>
```

```
struct LittleStruct{
```

```
    int num1;
```

```
    int num2;
```

```
};
```

```
int gvar;
```

```
char ReturnChar(){
```

```
    char c='c';
```

```
    return c;
```

```
}
```

```
short ReturnShort(){
```

```
    short h=6;
```

```
    return h;
```

```
}
```

```
int ReturnInt(){
```

```
    int i=9;
```

```
    return i;
```

```
}
```

```
double ReturnDouble(){
```

```
    double d=5.7;
```

```
    return d;
```

```

}

int * ReturnPtr(){
    return &gvar;
}

LittleStruct ReturnLittle(){
    LittleStruct a={12,34};
    return a;
}

int main(){
    char c;
    short h;
    int i,*p;
    double d;
    LittleStruct s;
    c=ReturnChar();
    h=ReturnShort();
    i=ReturnInt();
    d=ReturnDouble();
    p=ReturnPtr();
    s=ReturnLittle();
}

/// End of Program 3.12-1 ///

```

此程序中，函数ReturnChar()、ReturnShort()、ReturnInt()、ReturnDouble()、ReturnPtr()、ReturnLittle()的返回值类型分别是char、short、int、double、int*、LittleStruct。编译此程序，在生成的汇编源文件中，6条return语句对应的汇编代码分别是：

```

mov al, BYTE PTR _c$[ebp]      ; return c;
mov ax, WORD PTR _h$[ebp]       ; return h;
mov eax, DWORD PTR _i$[ebp]     ; return i;
fld  QWORD PTR _d$[ebp]         ; return d;
mov eax, OFFSET ?gvar@@3HA     ; return &gvar;
mov eax, DWORD PTR _a$[ebp]     ; return a;
mov edx, DWORD PTR _a$[ebp+4]

```

以上最后两条汇编语句对应一条return a;，其他的几条return语句都对应一条汇编语句。从return语句对应的汇编语句来看，当函数的返回值所占空间不超过64bit时，编译器将返回值存在各种不同的寄存器中，并不借助于计算机的内存。例如，单字节字符（8bit）放在寄存器al中，短整数（16bit）放在寄存器ax中，整数（32bit）放在寄存器eax中，双精度数（64bit）放在协处理器堆栈中，各种指针（包括引用，占32bit）放在寄存器eax中，64位的结构对象被存放在寄存器eax和edx的组合中。

当函数返回一个对象，而对象的体积超过64位时，函数的返回值又是如何存放的呢？考察如下的程序。

```
//// Program 3.12-2 ////
```

```
#include <stdio.h>
```

```
class A{
```

```
    char c;
```

```
    int i;
```

```
    double d;
```

```
public:
```

```
    A(){
```

```
        c='0';
```

```
        i=1;
```

```
        d=2.3;
```

```
}
```

```
};
```

```
A ReturnObject(){
```

```
    A a;
```

```
    return a;
```

```
}
```

```
int main(){
```

```
    A a;
```

```
    a=ReturnObject();
```

```
}
```

```
//// End of Program 3.12-2 ////
```

语句a=ReturnObject();对应的汇编代码是：

```
lea  eax, DWORD PTR $T3724[ebp]
push eax
call ?ReturnObject@@YAAV@XZ          ; ReturnObject
add esp, 4
mov ecx, DWORD PTR [eax]
mov DWORD PTR _a$[ebp], ecx
mov edx, DWORD PTR [eax+4]
mov DWORD PTR _a$[ebp+4], edx
mov ecx, DWORD PTR [eax+8]
mov DWORD PTR _a$[ebp+8], ecx
mov edx, DWORD PTR [eax+12]
mov DWORD PTR _a$[ebp+12], edx
```

由于函数返回的对象体积较大（超过64bit），主调函数（这里是main()）会在栈上创建一个临时对象用于接收函数的返回值。在调用函数ReturnObject()之前，会先把临时对象的地址送入寄存器eax（通过lea eax, DWORD PTR \$T3724[ebp]实现），然后将eax压入堆栈，作为函数ReturnObject()的额外参数。在函数ReturnObject()内部，语句return a;对应的汇编代码是：

```
mov eax, DWORD PTR __$ReturnUdt$[ebp]
```



```

mov ecx, DWORD PTR _a$[ebp]
mov DWORD PTR [eax], ecx
mov edx, DWORD PTR _a$[ebp+4]
mov DWORD PTR [eax+4], edx
mov ecx, DWORD PTR _a$[ebp+8]
mov DWORD PTR [eax+8], ecx
mov edx, DWORD PTR _a$[ebp+12]
mov DWORD PTR [eax+12], edx
mov eax, DWORD PTR __$ReturnUdt$[ebp]

```

其中地址`__$ReturnUdt$[ebp]`就是接收返回值的临时对象的首地址，语句`return a;`在执行时就是利用拷贝构造函数把对象a中的各个数据成员拷贝到临时对象中去。所以，如果函数的返回值体积较大，在函数调用时会先将接收返回值的对象（一般是临时对象，也可能是经过代码优化之后确定的其他对象）的首地址（4字节）压入堆栈，在被调用函数内部，通过该地址将返回值写入到接收对象中去。

知道了函数的返回值放在什么地方，就可以避免某些错误的函数调用，特别是函数指针类型转换的时候，尤其要注意这一点。考察下面的程序。

```
//// Program 3.12-3 ////
```

```
#include <stdio.h>
```

```
class A{
    char c;
    int i;
    double d;
public:
    A(){
        c='0';
        i=1;
        d=2.3;
    }
};
```

```
A ReturnObject(){
```

```
    A a;
    return a;
}
```

```
int ReturnInt(){
```

```
    return 1;
}
```

```
typedef void (*PTR)();
```

```
int main(){
```

```
    PTR p;
```

```
    p=reinterpret_cast<PTR>(ReturnObject);
```



```
p();
p=reinterpret_cast<PTR>(ReturnInt);
p();
}
/// End of Program 3.12-3 ///
```

此程序在Visual Studio 2005下能够通过编译，但运行的时候会出现运行错误。原因在于p=reinterpret_cast<PTR>(ReturnObject);这条语句。由于函数ReturnObject()返回的是一个体积较大的对象，所以主调函数为接收这个对象会在栈上创建一个临时对象（或者经过优化之后利用一个已经存在的对象接收返回值）。但函数指针p所代表的函数，其返回值类型是void，因此主调函数main()在通过p实现函数调用时，并没有为函数调用准备接收返回值的空间。在函数ReturnObject()内部，当执行到return a;这条语句时，发生运行时错误，因为返回值被非法地写入到其他变量占据的空间，使程序运行处于未知状态。而另一条类型转换语句p=reinterpret_cast<PTR>(ReturnInt);则是安全的，因为ReturnInt()的返回值被写入寄存器eax中，在主调函数中不使用eax的值，对函数的运行不会造成任何影响。

3.13 extern “C”的作用

C++语言最初的定位是“a better C”，在后来的实现中C++也保持了对C语言的兼容。因此，可以在C++程序中定义不属于任何类的全局变量和函数。但是，这并不意味着C++中的全局变量和函数所采用的编译和连接方式与C语言完全相同。C++语言引入了很多新的机制，为了支持函数的重载，C++对全局变量和函数的处理方式与C有明显的不同。

在Visual Studio 2005环境下打开头文件math.h，可以发现这个头文件的结构是这样的：

```
#ifndef _INC_MATH
#define _INC_MATH
/*...*/
#ifndef __cplusplus
extern "C"{
#endif
/*...*/
#ifndef __cplusplus
}
#endif
#endif /* _INC_MATH */
```

其他的一些系统头文件也具有类似的结构。那么，C++标准头文件为什么要采用这样一种结构呢？

显然，头文件中的编译宏“#ifndef _INC_MATH、#define _INC_MATH、#endif”的作用是防止该头文件被重复包含。那么，以下的条件编译指令：

```
#ifndef __cplusplus
extern "C"{
#endif
```



```
#ifdef __cplusplus
```

```
}
```

```
#endif
```

它的作用又是什么呢？

`extern`是C/C++语言中表明函数和全局变量作用范围（可见性）的关键字，该关键字告诉编译器，其声明的函数和变量可以在“全局”范围内使用。例如`extern int a;`是对全局变量a的声明语句。它只是告诉编译器，a是一个可在全局范围内使用的整型变量，并未为a分配内存空间。变量a在所有模块中作为全局变量只能被定义一次，否则会出现连接错误。有关变量的定义和声明的详细讨论参见2.7节。

所以，`extern “C”`包含两重含义：首先，被它修饰的目标是“`extern`”的；其次，被它修饰的目标是“C”的。

通常，在模块的头文件中对本模块提供给其他模块引用的函数和全局变量以关键字`extern`声明。例如，如果模块B欲引用该模块A中定义的全局变量和函数时只需包含模块A的头文件即可。这样，在模块B中调用模块A中定义的函数，在编译阶段，模块B虽然找不到该函数，但是并不会报错，它会在连接阶段中从模块A编译生成的目标代码中找到此函数。

由于`extern`表达的是“全局”的含义，而`static`关键字则表明变量或函数只能局限在单个模块内使用，所以，一个函数或变量只能被本模块使用时，不能用`extern “C”`修饰。

那么，对于全局变量和函数，C语言与C++语言的编译、连接方式到底有什么不同呢？首先考察下面的程序。

```
//// Program 3.13-1 /////
#include <iostream>
using namespace std;
int gvar=5;
void testfunc(){
    gvar++;
}
int testfunc(int para){
    return para+1;
}
int main(){
    testfunc();
    cout << gvar << endl;
    cout << testfunc(gvar) << endl;
}
/// End of Program 3.13-1 ///
```

查看此程序编译之后生成的汇编代码，发现全局变量gvar的名字变为?gvar@@3HA，函数`void testfunc();`所对应的函数名是?testfunc@@YAXXZ，而函数`int testfunc(int);`所对应的函数名是?testfunc@@YAH@Z。

作为一种面向对象的语言，C++支持函数重载，而C语言则不支持。正是由于这个原因，全局变量或函数被C++编译后在符号库中的名字与C语言的不同。将Program 3.13-1改造如下。

```

//// Program 3.13-2 ////
#include <iostream>
using namespace std;
#ifndef __cplusplus
extern "C"{
#endif
int gvar=5;
void testfunc(){
    gvar++;
}
int main(){
    testfunc();
    cout << gvar << endl;
}
#endif __cplusplus
#endif
//// End of Program 3.13-2 ////

```

查看对应的汇编代码，全局变量gvar在汇编源文件中的名字是_gvar，而函数testfunc在汇编源文件中的名字为_testfunc。可见，在C语言中，由于不存在函数重载，编译器对全局变量或函数的名字进行的处理是非常简单的，只要在C源程序中的名字前加上一个下画线“_”，就可以生成编译时所需的符号。

在C++编译器中，则不能采用C编译器所使用的方案。因为形成重载的两个函数如果在编译时产生的符号名也相同，将无法正常进行连接。所以，C++编译器会设法根据函数名和函数参数类型为每个函数起一个新名字，这种方案叫做“名字改编”(name mangling)。

例如，假设某个函数的原型为void foo(int x, int y)；，则该函数被C编译器编译后在符号库中的名字为_foo，而C++编译器则会产生像_foo_int_int之类的名字（不同的编译器可能生成的名字不同，但思想是一样的，生成的新名字称为“mangled name”）。

像_foo_int_int这样的名字包含了函数名、函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。又比如，函数void foo(int x, float y)编译后生成的符号应该是_foo_int_float这样的标识符，它与_foo_int_int显然是不同的。

C++除了支持全局变量外，还支持类的成员变量和局部变量。在高级语言层面上，类的成员变量、局部变量、全局变量因为同名，可能形成“隐藏”的现象。而本质上，编译器在进行编译时，对局部变量是通过局部变量在栈上距帧指针ebp的偏移量来进行访问的，对类对象的成员变量是通过其相对于对象首地址的偏移量来进行访问的，这样无论如何它们都不会发生冲突。

理解了C语言编译器与C++编译器对全局变量及函数的不同处理方式，就可以了解何时应该使用extern “C”。由于历史上存在大量的C函数库可以继续供C++程序使用，也由于在某些情况下需要采用C（或其他语言）和C++的混合编程，所以，在这些情况下，都需要使用extern “C”，以完成对特定全局变量或函数的连接任务。具体地说，有如下两种情形：



(1) C++程序调用C函数

在C++源程序中调用C函数，要在源程序中声明该函数原型。但如果用C++方式解析函数名，将导致连接错误。如下面的程序。

```
/// End of Program 3.13-3 ///
```

```
/** invoke.cpp **/
```

```
#include <iostream>
```

```
using namespace std;
```

```
int value();
```

```
int main(){
```

```
    cout << value() << endl;
```

```
}
```

```
/** End of invoke.cpp **/
```

```
/** value.c **/
```

```
int value(){
```

```
    return 5;
```

```
}
```

```
/** End of value.c**/
```

```
/// End of Program 3.13-3 ///
```

源文件value.c经过编译之后，函数value()在目标文件中的名字为_value（采用C语言的规范），而在源文件invoke.cpp中，函数声明中的value被C++编译器解析成?value@@YAHXZ，因此在连接时发生错误。

为了解决这个问题，在源文件invoke.cpp中，要使用extern “C” 告诉编译器：按照C语言规范解析函数名value。即这样实现函数声明：

```
extern "C"{
    int value();
}
```

由于C函数通常是以库函数的形式提供的，而函数声明都放在头文件中，所以在C++源程序中应将包含头文件的语句至于extern “C” 块中，即：

```
extern "C"{
    #include "header.h"
}
```

一种典型的情况是，如果C++程序调用一个C语言编写的.DLL，当包含.DLL的头文件或声明接口函数时，应该使用extern “C”。

(2) C程序调用C++函数

如果C程序想调用C++源文件中定义的全局变量或函数，一般的做法是让C++源文件在编译时，采用C语言的对全局变量和函数的编译规范。也就是说，在C++源程序中使用extern “C” 对函数名或全局变量名进行修饰。考察下面的程序。

```
/// Program 3.13-4 ///
```

```
/** invokecpp.h **/
```

```

extern "C"{
    void show();
}
/** End of invokecpp.h **/

/** invokecpp.cpp **/
#include <cstdio>
#include "invokecpp.h"
void show(){
    printf("Hello");
}
/** End of invokecpp.cpp **/

/** main.c **/
void show();
int main(){
    show();
}
/** End of main.c **/
/// End of Program 3.13-4 ///

```

在C++源文件invokecpp.cpp中，为了让函数void show();能够被C语言程序调用，必须用extern “C”通知编译器采用C语言的规则解析函数名。这个工作可在头文件invokecpp.h中完成。但是，在C语言源文件main.c中，却不能直接包含invokecpp.h，因为C语言不支持extern “C”声明，在.c文件中包含了extern “C”时会出现编译错误。所以，在main.c中，应该直接进行函数原型的声明，即声明void show();，这样，编译器可在所有参与连接的目标文件中寻找名字为_show的函数完成函数调用。

第4章 | 类与对象

4.1 类与对象概述

面向对象的程序设计（OOP）是程序设计方法的一次革命，它的核心机制是封装、继承和多态。而OOP最重要的两个概念，就是“类”和“对象”。对象是一个运行时概念，而类则是对对象的描述。

类与对象的出现，改变了程序员对外部世界的建模方式。在传统的面向过程的编程语言（如C语言中），程序员为了解决一个实际问题，会首先分析这些问题涉及到哪些数据，然后设计一些算法来处理这些数据。所以，有一个流传很广的公式：程序=数据结构+算法。在面向过程的程序设计中，数据的定义和对数据的处理是分离的（或者说是松散耦合的）。在面向对象的程序设计中，程序员会分析要解决的问题会涉及到现实世界中的哪些事物，这些事物是怎样相互作用的，然后用类来描述这些事物，并在程序运行时创建这些事物的实例——对象，依靠对象的行为或对象之间的交互来解决问题。

下面用一个具体的例子：约瑟夫问题，来展示面向过程的程序设计和面向对象的程序设计在组织代码方面的差异。约瑟夫问题是这样的：设有n个人围坐在一个圆桌周围，现从第s个人开始报数，报到第m个人出列，然后从出列的下一个人重新开始报数，数到第m个人又出列，……，如此重复直到所有的人全部出列为止。写出n个人的出列顺序。

为了简化问题，在这里假设s为1。设计一个算法，要求n和m可变，写出所有人员出列的顺序。约瑟夫问题可以用数组结构解决，也可以用链表结构解决。在这里我们采用数组，开始将所有的人的序号存在数组中，然后开始循环遍历数组，每出列一个人，就将数组对应元素的值修改为0，直到所有人员都出列。用C语言实现的方案如下。

```
//// Program 4.1-1 /////
#include <stdio.h>
#include <assert.h>
int *JosephWithArray(int n){
    int *p;
    p = new int[n];
    for(int i=0;i<n;i++)
        p[i]=i+1;
    return p;
}
int NextPeople(int prev,int *WorkingArray,int number_of_people,int step){
    int count=0;
```



```
int travel=prev;
int emptyplace=0;
while(count<step){
    travel=(travel+1)%number_of_people;
    if(WorkingArray[travel]){
        count++;
        emptyplace=0;
    }
    else{
        emptyplace++;
        if(emptyplace==number_of_people)
            return -1;
    }
}
return travel;
}

void CreateOutput(int *WorkingArray,int number_of_people,int step){
    int i;
    int outnum=-1;
    for(i=0;i<number_of_people;i++){
        outnum = NextPeople(outnum,WorkingArray,number_of_people,step);
        assert(outnum>-1 && outnum<number_of_people);
        printf("%d ",WorkingArray[outnum]);
        WorkingArray[outnum]=0;
    }
    printf("\n");
}

void Dispose(int *WorkingArray){
    delete[] WorkingArray;
}

int main(){
    int n,m;
    int *WorkingArray;
    printf("please input number of people:");
    scanf("%d",&n);
    printf("please input step:");
    scanf("%d",&m);
    WorkingArray = JosephWithArray(n);
    CreateOutput(WorkingArray,n,m);
    Dispose(WorkingArray);
}
```

```

}

/// End of Program 4.1-1 ///

```

这个程序用了4个函数解决约瑟夫问题。函数JosephWithArray()负责动态申请数组空间并对数组初始化。函数NextPeople()负责计算下一个出列的人的序号。函数CreateOutput()负责输出所有的人的出列序号。函数Dispose()负责释放动态申请的数组空间。约瑟夫问题涉及的关键参数和数据结构则分别由main()函数的局部变量n,m和WorkingArray负责维护。

如果用面向对象的程序设计方法，则要先分析约瑟夫问题涉及哪些参数，需要怎样的存储结构，在这些数据之上可以定义怎样的操作，然后将数据和对数据的操作封装在一起形成类，根据具体的数据集合形成类对象，执行类对象的行为完成相应的操作。以下是约瑟夫问题的面向对象的解决方案。

```

/// Program 4.1-2 ///
#include <iostream>
#include <cassert>
using namespace std;
class Joseph{
protected:
    int number_of_people;
    int step;
public:
    virtual void CreateOutput()=0;
    Joseph(int n,int m){
        number_of_people=n;
        step=m;
    }
};

class JosephWithArray:public Joseph{
    int *WorkingArray;
    int NextPeople(int prev);
public:
    JosephWithArray(int n,int m);
    void CreateOutput();
    ~JosephWithArray(){delete[] WorkingArray;}
};

JosephWithArray::JosephWithArray(int n,int m):Joseph(n,m){
    WorkingArray = new int[n];
    for(int i=0;i<n;i++)
        WorkingArray[i]=i+1;
}

void JosephWithArray::CreateOutput(){
    int i;
}

```

```

int outnum=-1;
for(i=0;i<number_of_people;i++){
    outnum = NextPeople(outnum);
    assert(outnum>-1 && outnum<number_of_people);
    cout << WorkingArray[outnum] << " ";
    WorkingArray[outnum]=0;
}
cout << endl;
}

int JosephWithArray::NextPeople(int prev){
    int count=0;
    int travel=prev;
    int emptyplace=0;
    while(count<step){
        travel=(travel+1)%number_of_people;
        if(WorkingArray[travel]){
            count++;
            emptyplace=0;
        }
        else{
            emptyplace++;
            if(emptyplace==number_of_people)
                return -1;
        }
    }
    return travel;
}

int main(){
    int n,m;
    cout << "please input number of people:";
    cin >> n;
    cout << "please input step:";
    cin >> m;
    JosephWithArray obj(n,m);
    obj.CreateOutput();
}

/// End of Program 4.1-2 ///

```

虽然所用的算法相同，但面向对象的解决方案和面向过程的解决方案在代码的组织上却是不同的。利用类和对象来组织代码，比全部用过程来组织代码，有很多好处。直接的效果就是更便于代码的编写和维护。详细的讨论可参见12.1节。



在理解类和对象时，注意这样几个要点：

- ① 类是外部世界的概念在计算机程序中的内部表示。类是对同一类事物共性的描述。
- ② 对象是类的实例。同一个类的不同对象有着共同的内部结构，有着共同的行为方式，但却有不同的内部状态。
- ③ 类是面向对象的源程序的基本组成单位，而对象是程序运行时数据的基本组织单位。
- ④ 类是一种数据类型。从这个意义上说，用类定义的变量就是对象。在源程序中定义类类型时，并不为类分配内存空间，所以不能对类的数据成员进行初始化。类中的数据成员也不能用extern、auto和register限定其存储类型。
- ⑤ 类由成员数据和成员函数组成。因此，定义一个类时，一般就是说明类对象所具有的数据成员，并完成成员函数的定义。但是，为了更有效地方便程序的编写，还允许在类中定义一些数据类型（如类、枚举、typedef声明等），具体的例子可参见Program 2.4-2、Program 4.7-1等。
- ⑥ 类提供了一种代码复用的手段。使用一个类的其他类或函数，称为该类的客户。在一个类的所有客户中都能被直接访问的成员，称为这个类的接口（Interface）。接口是C++类设计中的一个概念，代表了一个类的所有公有成员的集合。它并不是一个具体的语言机制，interface也不是C++的一个关键字，这一点与其他的几个面向对象的程序设计语言（如Java、C#等）不同。
- ⑦ 从概念上说，同对象交互是通过向对象发送消息来实现的，每个对象根据所接收到的消息的性质来决定需要采取的行动，以响应这个消息。响应这些消息的是一系列的方法，方法是在类定义中使用函数来定义的。在实现上，C++使用函数调用的机制把消息发送到一个对象上。

4.2 类定义后面为什么一定要加分号

类定义结束后，一定要在“}”之后加分号“;”，否则编译器会报错。为什么一般的复合语句结束之后不需要加分号，而类定义结束之后一定要加分号呢？原因是，类在C++中被当做一种数据类型，而凡是直接跟在数据类型后面的标识符，不是变量就是函数。这样，类的定义结束后，如果不在其后加分号，就会将其后面的内容解释成一个变量或函数，从而导致错误。考察下面这个简单的程序。

```
//// Program 4.2-1 /////
#include <iostream>
using namespace std;
int g;
int main(){
    cout << g << endl;
}
//// End of Program 4.2-1 ///
```

程序的输出结果是：

0

这是非常短小的一个程序，其中定义了一个整型的全局变量g，它被初始化为0。然后，



在主函数中将它的值打印出来。现在，把这个程序改造一下。既然类是一种数据类型，那么我们将全局变量g的类型换成某一class。改造后的程序如下。

```
//// Program 4.2-2 /////
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){num=6;}
    friend ostream& operator<<(ostream&,const A&);
}g;
int main(){
    cout << g << endl;
}
ostream& operator<<(ostream& out,const A& a){
    out << a.num << endl;
    return out;
}
//// End of Program 4.2-2 /////

```

程序的输出结果是：

6

这个程序的结构与上一个程序的结构基本上是一样的。全局变量的名字仍然是g，但类型由原来的int变成了class A。所以，直接出现在类定义后面的标识符，被编译器解释成该类类型的一个变量（若标识符后跟括号，则该标识符代表函数）。

但在大多数的情况下，我们不会采取此程序中如此“紧凑”的写法，而是先完成类的定义，然后再利用类名（就是一种类型名）来定义该类的变量。这时如果忘记在类定义后加分号，会出现什么情况呢？考察下面的程序。

```
//// Program 4.2-3 /////
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){num=6;}
    friend ostream& operator<<(ostream&,const A&);
}
int main(){
    A g;
    cout << g << endl;
}
```

```
ostream& operator<<(ostream& out,const A& a){
    out << a.num << endl;
    return out;
}
```

/// End of Program 4.2-3 ///

此程序无法通过编译。由于在类定义后缺少分号，编译器会将紧跟在class A定义之后的int当做类型A的一个变量，这显然是错误的。在类定义后加上分号后，程序能够顺利通过编译并输出6。

所以，除非是在定义类的同时立即定义该类的对象，否则类定义结束后一定要加分号，这样就可以防止把类定义后面的单词解释成该类的对象或函数。

既然定义类时是这样规定的，那么在定义结构和联合时，也应该遵守同样的规定，因为在C++中，结构和联合也都是数据类型。

有趣的是，对于基本数据类型而言，直接在其后加分号，并不算一个语法错误。如下面的程序。

```
/// Program 4.2-4 ///
#include <iostream>
using namespace std;
int main(){
    int;
    cout << "Hello,World!" << endl;
}
/// End of Program 4.2-4 ///
```

对此程序进行编译，得到一条警告信息：没有声明变量时忽略“int”的左侧。程序仍能通过编译并输出：Hello,World!。

4.3 关于初始化列表

C++语言规定，不能在类体中直接指定数据成员的初值，所以对象的初始化工作只能通过调用对象的构造函数来完成。在构造函数中，初始化列表扮演了十分重要的角色。

对于普通的数据成员而言，使用初始化列表和在构造函数体内赋初值，效果是一样的。见下面的程序。

```
/// Program 4.3-1 ///
#include <iostream>
using namespace std;
class A{
    int i;
public:
    A(){
        i=1;
    }
}
```



```
void show(){
    cout<<i<<endl;
}
};

class B{
    int i;
public:
    B():i(1){}
    void show(){
        cout<<i<<endl;
    }
};

int main(){
    A().show();
    B().show();
}

/// End of Program 4.3-1 ///
```

程序的输出结果是：

```
1  
1
```

类A的构造函数和类B的构造函数实际上没有任何差别。

但是，在另外一些情况下，只能使用初始化列表对成员进行初始化，否则会发生编译错误。例如，数据成员是引用、常变量、类对象（该类没有提供不带参数的构造函数）等。见下面的程序。

```
/// Program 4.3-2 ///
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(int i){
        num=i;
    }
    void show(){
        cout<<num<<endl;
    }
};
class B{
    int &r;
    const double PI;
```

```

A a;
public:
B(int i);
void show(){
    cout<<r<<endl;
    cout<<PI<<endl;
    a.show();
}
};

int e=5;
B::B(int i):r(e),PI(3.1415926),a(i){}
int main(){
    B(1).show();
}
/// End of Program 4.3-2 ///

```

程序的输出结果为：

```

5
3.14159
1

```

在类B的构造函数中，将初始化列表中的任何一个成员的初始化工作移到函数体内，都会导致编译错误，见下面的说明：

```

B::B(int i){
    r=e;           //这是为引用赋值，并不是初始化
    PI=3.1415926; //常量的值不允许改变
    a=A(i);        //在类 B 的构造函数调用之前，会先调用类 A 的构造函数 A()，而此
                    //函数无定义
}

```

有时，程序员试图利用对象之间的赋值来代替初始化，而不是显式使用初始化列表。这样虽然不会发生编译错误，但却造成了程序运行效率的降低。如下面的程序。

```

/// Program 4.3-3 ///
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){
        cout<<"In default constructor"<<endl;
    }
    A(int i){
        cout<<"In user-defined constructor"<<endl;
    }
}

```

```

    num=i;
}
A & operator=(const A& a){
    cout<<"Using assignment"<<endl;
    num=a.num;
    return *this;
}
void show(){
    cout<<num<<endl;
}
};

class B{
    A a;
public:
    B(int i);
    void show(){
        a.show();
    }
};
B::B(int i){
    a=A(i);
}
int main(){
    B b(1);
    b.show();
}

/// End of Program 4.3-3 ///

```

程序的输出结果是：

In default constructor
In user-defined constructor
Using assignment

1

从中可以看出，如果没有显式地在初始化列表中对成员对象a进行初始化，则在进入类B的构造函数体之前，会先调用类A的默认构造函数。在类B的构造函数体内进行的是赋值操作，并且要调用类A的构造函数A(int)先生成一个无名对象。这种为成员对象初始化的方式不仅逻辑结构不清晰，而且效率低下。如果将类B的构造函数改写成：

B::B(int i):a(i){}

则程序的输出结果变成：

In user-defined constructor

1



这样不仅完成了想要的初始化工作，而且效率很高，避免了不必要的操作。

表面上看，有很多类的构造函数根本没有使用初始化列表。但实际上，有些内容不管有没有显式写进初始化列表，都是会被“强行”加进去的。这些内容包括：

①如果该类是某个类的派生类，那么它的直接基类的构造函数一定要出现在初始化列表中。要么是程序员显式地在初始化列表中调用直接基类的构造函数，否则编译器把直接基类的默认构造函数插入到初始化列表中。

②如果该类包含其他类的对象作为其数据成员，那么这些对象的初始化工作也一定要在初始化列表中完成。要么是程序员显式地在初始化列表中给出对象的构造形式，否则编译器会在初始化列表中调用这些对象的默认构造函数来完成初始化。

使用初始化列表还有一个问题要注意，初始化列表中各部分的执行顺序与它们的书写顺序无关，而是遵循这样的规则：基类的构造函数最先执行，其他的成员按照它们在类中声明的顺序依次被初始化。考察下面的程序。

```
/// Program 4.3-4 ///
#include <iostream>
using namespace std;
class A{
public:
    int num;
    A(int i){
        num=i;
        cout<<"In A constructor"<<endl;
    }
};
class B{
public:
    int num;
    B(int i){
        num=i;
        cout<<"In B constructor"<<endl;
    }
};
class C{
protected:
    int num;
public:
    C(int i){
        num=i;
        cout<<"In C constructor"<<endl;
    }
};
```



```
class D:public C{
    A a;
    int num;
    B b;
public:
    D(int i):num(i++),b(i++),a(i++),C(i++){}
    void show(){
        cout<<"C::num="<<C::num<<endl;
        cout<<"a.num="<<a.num<<endl;
        cout<<"num="<<num<<endl;
        cout<<"b.num="<<b.num<<endl;
    }
};

int main(){
    D d(1);
    d.show();
}

/// End of Program 4.3-4 ///
```

如果按照初始化列表的书写顺序执行，那么程序的输出结果应该是：

```
In B constructor
In A constructor
In C constructor
C::num=4
a.num=3
num=1
b.num=2
```

而实际上程序的运行结果是：

```
In C constructor
In A constructor
In B constructor
C::num=1
a.num=2
num=3
b.num=4
```

在初始化列表中，无法完成为对象的数组成员进行初始化的工作，原因是C++语言没有提供这样的机制，所以只能在构造函数体内分别为成员数组的每个元素分别赋初值。如果数组元素本身也是对象，那么这种赋值操作会造成较大的运行时开销。问题还不止这些。如果在类体中定义一个常量数组，如何进行初始化呢？例如，定义一个类：

```
class A{
    const int arr[3];
```

```
public:
```

```
    A();
```

```
};
```

然后定义下列构造函数：A::A():arr{1,2,3}{}

或者是：A::A():arr(1,2,3){}

或者是：A::A():arr[0](1), arr[1](2), arr[2](3){}

再或者是：A::A(){arr[0]=1;arr[1]=2;arr[2]=3;}

都无法通过编译。所以实际上是不能在一个类中定义一个常量数组的，因为无法为对象的常量数组成员初始化。

在实际应用中，通常只会在类中定义静态的常量数组，这样就可以完成数组的初始化工作。如果确实想在类中定义一个常量数组，一个变通的做法是定义一个指向常量的指针常量，然后在初始化列表中为它初始化。见下面的程序。

```
/// Program 4.3-5 ///
#include <iostream>
using namespace std;
int *CreateArr(int n){
    int *p;
    p = new int[n];
    for(int i=0;i<n;i++)
        p[i]=i+1;
    return p;
}
class A{
    int arrsize;
    const int * const arr;
public:
    A(int n):arr(CreateArr(n)){
        arrsize=n;
    }
    void show(){
        for(int i=0;i<arrsize;i++)
            cout<<arr[i]<<' ';
        cout<<endl;
    }
    ~A(){delete[] arr;}
};
int main(){
    A a(3);
    a.show();
}
```



/// End of Program 4.3-5 ///

程序的输出结果是：

1 2 3

由于arr是一个指向常量的指针常量，所以arr的初始化工作一旦完成，就不能改变arr数组元素的值，也不能将arr指向别处。客观上arr与一个常量数组的作用是相同的。在上面的程序中，将函数CreateArr()移到类A体内作为该类的一个成员函数，程序运行结果是一样的。由于实际上是在构造函数中为arr分配了空间，所以应该在类A的析构函数中释放这部分空间，即在类A中定义：

`-A(){delete[] arr;}`

这样就能保证不发生内存泄漏。

4.4 对象的生成方式

本节讨论两个问题：

- ①可以用哪些语句定义（生成）一个对象？
- ②为对象分配空间和执行构造函数之间的关系是怎样的？

可以生成一个全局对象，也可以生成一个局部对象，还可以用new操作在堆上生成一个动态对象。这些操作在具体的表现形式上有些不同的规定，需要引起注意。

假设定义一个类：

```
class SomeClass{  
    int num;  
public:  
    SomeClass(){num=0;}  
    SomeClass(int i){num=i;}  
};
```

并定义一个指向该类对象的指针：SomeClass *p;

则下述生成对象的操作是合法的：

```
SomeClass obj(5);  
SomeClass obj=SomeClass(5);  
Someclass obj(SomeClass(5));  
SomeClass obj;  
SomeClass obj=SomeClass();  
p = new SomeClass(5);  
p = new SomeClass();  
p = new SomeClass;  
p = new SomeClass[3];           //一次性产生三个对象
```

而下述生成对象的操作是非法的：

```
SomeClass obj();  
SomeClass obj=SomeClass;  
SomeClass obj(SomeClass);
```



```
SomeClass obj(SomeClass());
p = new SomeClass(5)[3];           //企图对每个对象调用指定的构造函数
```

SomeClass obj();被编译器解释为一个函数原型，函数名为obj，返回值类型为SomeClass，而且该函数不带任何参数。SomeClass obj(SomeClass);也被解释成一个函数原型，函数名为obj，返回值类型为SomeClass，带一个类型为SomeClass的参数。SomeClass obj(SomeClass());仍然被解释成函数原型，函数名为obj，返回值类型为SomeClass，带一个类型为SomeClass(*)() 的函数指针参数（参见Program 3.7-3）。它们都不能用来生成对象。

SomeClass obj=SomeClass;也是错误的，因为对象不能等于一种数据类型。

当用new操作申请一个对象的数组（如p = new SomeClass[3];）时，会产生一个有趣的问题。C++语言不允许在用new操作申请动态数组时为数组的每一个分量赋初值，这样当一个类没有定义默认构造函数时，就不能利用new操作生成该类对象的动态数组。

在上面的例子中，虽然产生一个局部对象的三种写法不同：

```
SomeClass obj(5);
SomeClass obj=SomeClass(5);
SomeClass obj(SomeClass(5));
```

但实际上它们是完全等价的，效果是只产生一个对象，只调用一次构造函数。SomeClass obj=SomeClass(5);并不是赋值操作，它是与SomeClass obj(SomeClass(5));等价的另一种写法。显式调用类的构造函数的本意是创建一个无名对象，然后用这个无名对象来初始化对象obj。一个无名对象的生成如果仅仅是为了创建另一个同类的对象，那么编译器并不会真的生成这个无名对象，而是直接调用构造函数创建另一个对象。所以，上述两条语句并不会调用复制构造函数，而是直接在对象obj上调用构造函数SomeClass(5)来完成初始化的工作。

显式调用类的构造函数产生的无名对象，可以在两种场合下使用。一是用于类型转换，把一种数据类型的变量显式转换成另一种类型的变量后参与运算。还有一种用途就是作为函数的返回值。如下面的程序。

```
/// Program 4.4-1 ///
#include <iostream>
using namespace std;
class Complex{
    double real;
    double image;
public:
    Complex(double r=0.0,double i=0.0){
        real=r;
        image=i;
    }
    void Show(){
        cout << real << "+" << image << "i" << endl;
    }
    Complex operator+(const Complex&);
};
```

```

Complex Complex::operator+(const Complex &c){
    return Complex(real+c.real,image+c.image); //调用构造函数生成无名对象返回运算结果
}
int main(){
    Complex c1(1.1,2.2),c2(2.3,4.5);
    Complex c3=c1+c2;
    c3.Show();
}
/// End of Program 4.4-1 ///

```

程序的运行结果是：3.4+6.7i。在运算符重载函数operator+()中，直接调用类Complex的构造函数生成一个无名对象，完成传递运算结果的任务。在语句Complex c3=c1+c2;的执行过程中，operator+()中显式调用了Complex类的构造函数。由于调用构造函数的目的就是为了初始化对象c3，所以，这个构造函数是直接在对象c3上调用的（也就是说，传递进构造函数的this指针是c3的首地址），而并没有真正生成一个无名对象。具体的实现过程请参阅4.5节。

需要注意的是，由于全局对象和静态对象必须是有名的，所以直接调用构造函数产生一个无名的全局对象或静态对象是非法的。在类中定义一个无名的成员对象也是非法的。

从概念上说，构造函数是用来“生成”对象的。换句话说，只有当构造函数完成之后，这个对象才构造好，才能交付使用。但这只是从高级语言的层面来看待构造函数的。而从底层实现来说，一个对象是否靠构造函数来为它分配空间呢？既然构造函数完成之前，对象还未“生成”，那么在构造函数中能不能调用该对象的其他成员函数呢？为对象分配空间的动作和调用构造函数的动作是否总是同时进行的呢？考察下面的程序。

```

/// Program 4.4-2 ///
#include <iostream>
using namespace std;
class B;
class A{
public:
    A(B &obj){cout << "Befor constructor, b's address is: " << &obj << endl;}
};
class B{
    int num;
public:
    B(){
        cout << "In constructor, b's address is: " << this << endl;
        show();
    }
    void show(){
        cout << "B's member function,num=" << num << endl;
    }
};

```

```

extern B b;
A a(b);           // 定义一个全局对象
B b;              // 定义另一个全局对象
int main(){
    cout << "After constructor, b's address is: " << &b << endl;
}
/// End of Program 4.4-2 ///

```

程序的输出结果是：

```

Befor constructor, b's address is: 0041A178
In constructor, b's address is: 0041A178
B's member function,num=0
After constructor, b's address is: 0041A178

```

从程序的输出结果来看，对象a的构造函数在对象b的构造函数之前执行，而此时对象b的地址已经确定了。在调用b的构造函数时，和在构造函数调用完成之后，b的地址都没有发生变化。因此，有如下结论：

①为对象分配空间和调用对象的构造函数是两个不同的工作。构造函数不是用来为对象分配空间的，而是对数据成员进行初始化，使它们首次获得有意义的值。进入对象的构造函数时，对象空间分配的工作已经完成。因此，在构造函数中可以自由地访问对象的数据成员和调用成员函数（如上例）。

②一般情况下，为对象分配空间和调用构造函数总是紧接着进行的，这是编译器精心组织代码的结果。但也有例外，如在函数体内定义的静态对象，其空间是在程序执行之初就已经分配（与全局变量同在一个存储区），而该对象的构造函数却是当控制流第一到达定义该对象之处才执行的。

4.5 关于临时对象

顾名思义，临时对象就是在程序中没有显式地给出创建对象的语句，但确实产生的对象。临时对象是在栈上产生的，C++利用它来完成一些重要的操作。理解临时对象产生的原因和产生的过程，对于理解程序的行为和提高程序的效率都是很重要的。

临时对象一般出现在两种情况下：接收函数的返回值、用于实现隐式的类型转换。在语句const A &ra=5;中，由于5是一个文字常量，无法寻址，所以必须先由5转换成一个A类的临时对象，再用该临时对象初始化引用ra。利用临时对象接收函数的返回值的例子如下。

```

/// Program 4.5-1 ///
#include <iostream>
using namespace std;
class X{
    static int order;
    char ch;
    int n;
public:

```

```
X(char c){  
    order++;  
    n=order;  
    ch=c;  
    cout << "Constructing object " << n << endl;  
}  
~X(){  
    cout << "Destructing object " << n << endl;  
}  
X(X&x){  
    order++;  
    n=order;  
    ch=x.ch;  
    cout << "Constructing object " << n << " by copying" << endl;  
}  
X& operator=(const X &x){  
    ch=x.ch;  
    return *this;  
};  
int X::order;  
X makewho(){  
    X B('B');  
    return B;  
}  
int main(){  
    X A('A');  
    A=makewho();  
}  
/// End of Program 4.5-1 ///
```

程序的执行结果是：

```
Constructing object 1  
Constructing object 2  
Constructing object 3 by copying  
Destructing object 2  
Destructing object 3  
Destructing object 1
```

在main()函数中，对象A是有名的；在函数makewho()中，对象B是有名的。但makewho()函数返回的对象位于何处呢？这个返回值是怎样送到对象A中去的呢？实际上，语句A=makewho(); 等同于以下两条语句的执行结果：



```
X C(makewho());
```

```
A=C;
```

也就是说，在makewho()函数内部，产生了一个局部对象B，函数的返回值就是根据B的值决定的。要把makewho()函数内部的对象B的值传递到主调函数中的对象A中，实际上是在主调函数中生成了一个临时对象，这个临时对象是由对象B作参数调用类X的复制构造函数产生的。然后，再将临时对象的值赋给对象A完成规定的操作。主调函数中的临时对象的产生过程是非常有趣的：对象的空间是主调函数（main()函数）在调用makewho()之前就分配了，对象的构造函数是在makewho()函数返回之前执行的（对应于输出信息 Constructing object 3 by copying）。在这之后，对象B被释放（对应于输出信息 Destructing object 2）。在将临时对象的值传递给对象A的赋值操作完成之后，临时对象的析构函数被调用（对应于输出信息 Destructing object 3），在main()函数结束之前，对象A的析构函数被调用（对应于输出信息 Destructing object 1）。

这就说明，临时对象除了“不显式产生”之外，它的其他方面跟普通对象一样：产生的时候要调用构造函数，消亡的时候要调用析构函数，可以访问临时对象的成员，对临时对象进行寻址操作等。

如果函数的被返回对象是一个无名对象，则根据主调函数的接收情况，会出现两种不同的情形：

①主调函数正好要利用函数的返回值构造一个新的有名对象，这时没有必要在被调函数中产生临时对象，而是直接在被调函数返回之前构造出主调函数中的那个接收对象即可。例如，对上面的程序做一个调整：

```
X makewho(){
    return X('B');           //直接返回无名对象
}
int main(){
    X A=makewho();         //利用函数返回值产生新对象
}
```

本来，在函数makewho()中要产生一个用于传递返回值的无名对象，在main()函数中要产生一个用于接收函数值的临时对象，然后再将接收之后的值传递给对象A。由于对象A、被调函数中的无名对象、主调函数中的临时对象三者完全相同，而且临时对象除了用来构造A之外并无其他用处，所以编译器做了优化：并不产生临时对象，而是直接调用makewho()函数中的类X的构造函数X('B')来构造main()函数中的对象A。所以程序的输出是：

```
Constructing object 1
```

```
Destructing object 1
```

②主调函数并不是利用函数的返回值构造一个新的有名对象，那么在主调函数中还是要构造一个临时对象来接收函数的返回值。由于被调函数中的无名对象与主调函数中的接收对象完全相同，而被调函数中的无名对象除了传递返回值之外并无其他用处，所以，只需在被调函数返回之前，利用被调函数中的构造函数来构造主调函数中的接收对象即可。换句话说，没有必要产生属于被调函数的无名对象，而是直接在被调函数中创建主调函数中的接受临时对象即可。例如，如果把程序修改为：

```
X makewho(){
```

```

    return X('B');
}
int main(){
    makewho();
}

```

那么程序的输出结果是：

```

Constructing object 1
Destructing object 1

```

这两条输出分别是在makewho()函数返回之前和main()函数返回之前执行的，对应于main()中的临时对象的产生和消亡。而并没有创建一个属于makewho()函数的无名对象。

综上两点可以看出，如果某被调函数返回一个类对象，且在return语句中直接调用类的构造函数，则传入该构造函数的this指针一定来自主调函数，即主调函数中预备接受返回值的那片空间的首地址。

4.6 关于点操作符

在C语言中，点操作符用来存取结构的成员。在C++中，这种用法继续保留。并且，由于C++语言中大量使用类对象，所以自然地将点操作符用于访问类对象的成员，包括成员变量和成员函数。

面向对象的程序设计要求信息隐藏，在C++程序设计中表现为尽量地使数据成员成为私有型(private)成员或保护(protected)型成员，而通过公有的成员函数来间接访问它们。学过C++的程序员大多会留下“非公有成员不能用点操作符访问”的印象。要注意的是，这种说法过于笼统，我们还是应该分几种情况区别对待。

①在类对象的成员函数中访问该对象自己的非公有成员变量或非公有成员函数，同样可以通过点操作符进行。一般情况下，我们可以直接写成员变量或成员函数的名字对它们进行访问，不必借助点操作符。但要注意的是：就算没有必要，使用点操作符访问当前对象成员是符合语法要求的，不会发生编译错误；在某些特殊情况下，必须借助于“->”操作符或点操作符。见下面的例子。

```

//// Program 4.6-1 /////
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(int num){
        (*this).num=num;      //必须借助->操作符或.操作符
    }
    void show(){
        cout << (*this).num << endl;    //虽然没有必要使用.操作符，但写了也不会错
    }
}

```



```

};

int main(){
    A a(5);
    a.show();
}

/// End of Program 4.6-1 ///

```

在类 A 的构造函数中，由于输入参数恰好和类的成员变量同名，所以需要借助于->操作符或点操作符进行区分。

②在类对象的成员函数中访问该类的别的对象的非公有成员，可通过点操作符进行。这也是一种特殊情况，在实践中也是经常用到的。见下面的例子。

```

/// Program 4.6-2 ///
#include <iostream>
using namespace std;
class Student{
    char *name;
public:
    Student(){
        name= new char[20];
    }
    Student(char *n){
        name= new char[20];
        strcpy(name,n);
    }
    ~Student(){
        delete[] name;
    }
    void show(){
        cout << name << endl;
    }
    Student &operator=(const Student&);

};

Student & Student::operator=(const Student &st){
    strcpy(name,st.name);           //对象st是Student类的另一个实例
    return *this;
}

```

```

int main(){
    Student s1,s2("张三");
    s1=s2;
    s1.show();
}

```



/// End of Program 4.6-2 ///

在赋值操作符重载函数中，对象 st 是类 Student 的另一个实例，而不是当前对象。但是，在当前对象的成员函数中仍然可以安全地使用点操作符对 st 对象的私有成员进行访问。

③在类的静态成员函数中，可通过点操作符访问该类对象的非公有成员。在全局函数中则是不允许的。见下面的例子。

/// Program 4.6-3 ///

```
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(int i){
        num=i;
    }
    static int square(const A& obj){
        return obj.num*obj.num;
    }
};
int main(){
    A a(5);
    cout << A::square(a) << endl;
}
```

/// End of Program 4.6-3 ///

④若函数 func()是类 A 的友元函数，则在函数 func()中可以自由地访问类 A 对象的任何成员（包括公有成员和非公有成员）。这是大家熟知的，这里不做过多解释。

所以，使用点操作符并不是在任何情况下都不能访问对象的非公有成员。如果一个函数既不是某个类的成员函数，也不是这个类的友元函数，那么在这个函数中不能用点操作符访问该类对象的非公有成员，在其他情况下则是允许的。

另外要注意的是，箭头操作符（->）与点操作符是可以相互替代的两个操作符。也就是说，访问对象成员时，凡使用“.”操作符的地方，都可以使用->操作符的相应形式替代，反之亦然。例如，(*this).num 与 this->num 是等价的，obj.num 与(&obj)->num 也是等价的。只不过，在通过指针访问对象成员的时候，使用->操作符会比较方便；而在使用对象名或对象的引用访问对象成员时，使用.操作符会更加方便一些。

点操作符与箭头操作符的区别是：.操作符不允许重载，而->操作符允许重载。下面是一个重载->操作符的例子。

/// Program 4.6-4 ///

```
#include <iostream>
using namespace std;
class A{
    struct MemStruct{
```

```

int i;
int j;
}mem;
public:
A(){
    mem.i=0;
    mem.j=0;
}
MemStruct * operator->(){
    return &mem;
}
};

int main(){
A a;
cout << a->i << " " << a->j << endl;
}

/// End of Program 4.6-4 ///

```

程序输出结果是：0 0。

在对->操作符进行重载时要注意：

①操作符函数 `operator->()` 只能作为类的成员函数进行重载，且参数列表必须为空。这就意味着箭头操作符的左边出现的是一个对象或对象的引用，而不是指向对象的指针。这也是判断是否重载了->操作符的一个显著标志。同时可以看出，表面上箭头操作符像一个二元操作符，但实际上由于对象成员的类型各不一样，在重载时-> 操作符被解释成一元操作符。这就是操作符函数 `operator->()` 的参数列表为空的原因。

②操作符函数 `operator->()` 的返回值必须是指向某个类对象（或结构）的指针，或者是另一个对->操作符进行了重载的类的对象或引用。实际上，上面程序中的 `a->i` 就被重新解释成 `(a.operator->())->i`。经常见到的一种情况是，返回指向当前对象的指针。

③由于重载->操作符可能会改变程序员使用->操作的惯例，所以重新解释->操作符一定要合理，不要使用怪异的操作符重载。

4.7 嵌套类与局部类

在一个类的类体内部还可以定义另外一个类，这个类就是一个嵌套类。拥有嵌套类的类相应地成为外围类。定义嵌套类的目的在于隐藏类名，减少全局的标识符，从而限制用户能否使用该类建立对象。这样可以提高类的抽象能力，并且强调了两个类(外围类和嵌套类)之间的主从关系。下面是一个嵌套类的例子。

```

/// Program 4.7-1 ///
#include <iostream>
using namespace std;
class A{

```

```

public:
    class B{
        public:
            B(char *name){cout << "Constructing B: " << name << endl;}
    };
    B b;
    A():b("In class A"){
        cout << "Constructing A" << endl;
    }
};

int main(){
    A a;
    A::B b("Outside class A");
}

//// End of Program 4.7-1 /////

```

程序的运行结果是：

```

Constructing B: In class A
Constructing A
Constructing B: Outside class A

```

设立嵌套类的初衷是建立仅供某个类的成员函数使用的类类型，在其他函数中，这个嵌套类不可见。但是，如同类的其他成员一样，嵌套类也有访问权限。在上面的程序中，嵌套类 B 的访问权限是 public，所以可在类 A 的成员函数之外使用该嵌套类，只不过在使用时要加上名字限定，如 main() 函数中的 A::B b。如果将嵌套类的访问权限设定为 private，那么就只能在该类的成员函数中使用该嵌套类。

除了作用域上的联系之外，其实外围类与嵌套类是两个完全独立的类，并没有其他特殊的关系。也就是说，嵌套类的成员和外围类的成员没有任何关系，它们不能够相互访问，也不存在友元的关系。

可以仅在外围类的类体中声明嵌套类，而将嵌套类的成员函数的定义放到外围类之外去。如上面的程序可重写如下。

```

//// Program 4.7-2 /////
#include <iostream>
using namespace std;
class A{
public:
    class B{
        public:
            B(char *);
    };
    B b;
    A():b("In class A"){

```



```

cout << "Constructing A" << endl;
}
};

A::B::B(char *name){cout << "Constructing B: " << name << endl;}
int main(){
    A a;
    A::B b("Outside class A");
}

/// End of Program 4.7-2 ///

```

Program 4.7-2 与 Program 4.7-1 完全等价，惟一的不同就是嵌套类的构造函数的定义是在外围类的体外定义的。需要注意的是，如果将类 B 的构造函数放到类 B 的类体之外、类 A 的类体之内来定义，就会出现编译错误，因为这与重写纯虚函数的语法相冲突，详细解释参见 4.11 节。

在一个函数体内定义的类称为局部类。局部类可以定义自己的数据成员和函数成员。它也是一种作用域受限的类。下面是一个具体的例子。

```

/// Program 4.7-3 ///
#include <iostream>
using namespace std;
void func(){
    static int s;
    class A{
        public:
            int num;
            void init(int i) { s = i; }
    };
    A a;
    a.init(8);
    cout << s << endl;
}

int main(){
    func();
}

/// End of Program 4.7-3 ///

```

程序的执行结果是：8。阅读以上程序，要注意使用局部类时的几个要点：

- ① 局部类只能在定义它的函数内部使用，在其他地方不能使用。
- ② 局部类的所有成员函数都必须定义在类体内，因此在结构上不是特别灵活。
- ③ 在局部类的成员函数中，除了可以访问函数自己的局部变量、类自己的成员变量、全局变量、全局静态变量之外，还可以访问在定义该局部类的函数中定义的静态变量，如上例中的变量 s。
- ④ 局部类中不能定义静态数据成员（主要是这种数据成员的初始化工作无法完成），因

此，虽然允许定义静态成员函数，但在这些静态成员函数中只能访问函数自己的局部变量、全局变量、全局静态变量和在定义该局部类的函数中定义的静态变量。

在实践中，局部类很少使用。

4.8 对象之间的比较

对于 C++ 内置数据类型的变量来说，相互之间比较大小只要直接使用`>`、`<`、`=`等关系运算符就可以达到目的。这些关系运算符的语义由编译器负责解释。但对于用户自定义的类类型来说，问题却不是这么简单。本来，按照通常的想象，比较两个对象是否相等，只要把它们各自对应的属性（成员变量）值一一进行比较，只要所有的属性值都相等，这两个对象就相等。那么，C++ 编译器是不是这样处理的呢？看下面的程序。

```
//// Program 4.8-1 ////
#include <iostream>
using namespace std;
class Student{
    double average_score;
    int age;
public:
    Student(double s,int a){
        average_score=s;
        age=a;
    }
};
int main(){
    Student s1(80.5,18);
    Student s2(83.3,19);
    if(s1==s2)
        cout << "Student one is the same as student two" << endl;
}
/// End of Program 4.8-1 ///
```

在这个程序中生成了两个 `Student` 类的对象 `s1` 和 `s2`，每个对象都有两个数据成员。按“常理”推测，可以将 `s1` 同 `s2` 的比较转换成 `s1.average_score` 与 `s2.average_score`、`s1.age` 与 `s2.age` 的比较，但结果却是编译器报错，无法对 `s1==s2` 的语义作出正确的解释。

从这个程序可以看出，类对象之间进行比较要对关系运算符重新作出解释，也就是说，要对关系运算符（`>`、`<`、`==`、`>=`、`<=`、`!=`）进行重载。把上面的程序改造一下，并使用重载之后的运算符，源代码如下。

```
//// Program 4.8-2 ////
#include <iostream>
using namespace std;
class Student{
```



```

double average_score;
int age;
public:
    Student(double s,int a){
        average_score=s;
        age=a;
    }
    bool operator<(const Student &obj){
        return (average_score<obj.average_score);
    }
    bool operator>(const Student &obj){
        return (average_score>obj.average_score);
    }
    bool operator==(const Student &obj){
        return !operator>(obj)&&!operator<(obj);
    }
    bool operator>=(const Student &obj){
        return !operator<(obj);
    }
    bool operator<=(const Student &obj){
        return !operator>(obj);
    }
    bool operator!=(const Student &obj){
        return operator>(obj)||operator<(obj);
    }
};

int main(){
    Student s1(80.5,18);
    Student s2(83.3,19);
    char *fig;
    fig=(s1>s2)?">":"not ";
    cout << "Student one is " << fig << ">" << " student two" << endl;
    fig=(s1<s2)?"<":"not ";
    cout << "Student one is " << fig << "<" << " student two" << endl;
    fig=(s1==s2)?"==":"not ";
    cout << "Student one is " << fig << "==" << " student two" << endl;
    fig=(s1>=s2)?">=":"not ";
    cout << "Student one is " << fig << ">=" << " student two" << endl;
    fig=(s1<=s2)?"<=":"not ";
    cout << "Student one is " << fig << "<=" << " student two" << endl;
}

```



```

fig=(s1!=s2)?":";
cout << "Student one is " << fig << "!=" << " student two" << endl;
}
/// End of Program 4.8-2 ///

```

程序的执行结果是：

```

Student one is not > student two
Student one is < student two
Student one is not == student two
Student one is not >= student two
Student one is <= student two
Student one is != student two

```

通过对 6 个关系运算符的重载，实现了对象之间大小的比较，这在某些“排序”的过程中是非常有用的。在研究这个程序时，要注意这样几点：

① 在上面的程序中，我们是按照学生的平均成绩进行排序，如果要按照学生的年龄进行排序，就必须改写关系运算符重载函数。

② 从逻辑上说，这 6 个关系运算符只要实现两个（在例子中我们是实现了`>`和`<`），其他 4 个就可以根据已经实现的两个关系运算符推导出来。但是，在程序实现上，必须显式地写出这 6 个运算符重载函数，否则一旦用到某个关系运算符而没有对应的重载函数，编译器就会报错。

③ 除了可以用成员函数的方式实现关系运算符的重载，还可以用友元函数的方式实现。例如可以定义一个关系运算符函数：

```

bool operator<(const Student &obj1, const Student &obj2){
    return (obj1.average_score < obj2.average_score);
}

```

然后将它声明为类 Student 的友元函数即可。其余的运算符重载可依此类推。

要实现对象之间大小的比较，用重载关系运算符的方法的确行之有效。但是，如同我们已经看到的那样，为了完备起见，我们要对 6 个关系运算符都进行重载，显得非常的麻烦。有没有更简单一些的方法呢？

实际上，我们可以利用“类型转换”。在 6.2 节我们谈到，当发生函数调用的时候，C++ 编译器会依据三个步骤将发生调用时的实参同函数定义中的形参进行类型匹配，如果直接匹配不成功，就会试图将实参类型自动转换成形参类型，从而完成函数调用。由于关系运算符本身就是函数，所以如果参与关系运算的两个操作数没有办法直接比较，编译器就会尝试将它们转换成别的类型，然后再进行比较。在上面的例子中，由于学生之间进行比较只是为了比较他们的平均成绩，所以可以定义一个类型转换（将 Student 类型转换成 double 类型），这样只要是不能直接接受 Student 类型的地方，编译器都会尝试将 Student 类型转换成 double 类型，然后再看是否能完成相应的操作。改造之后的程序如下。

```

/// Program 4.8-3 ///
#include <iostream>
using namespace std;
class Student{

```



```

double average_score;
int age;
public:
    Student(double s,int a){
        average_score=s;
        age=a;
    }
    operator double(){
        return average_score;
    }
};

int main(){
    Student s1(80.5,18);
    Student s2(83.3,19);
    char *fig;
    cout << "The score of student one is " << s1 << endl;
    cout << "The score of student two is " << s2 << endl;
    fig=(s1>s2)?":>":"not ";
    cout << "Student one is " << fig << ">" << " student two" << endl;
    fig=(s1<s2)?":<":"not ";
    cout << "Student one is " << fig << "<" << " student two" << endl;
    fig=(s1==s2)?":<=":"not ";
    cout << "Student one is " << fig << "==" << " student two" << endl;
    fig=(s1>=s2)?":>=":"not ";
    cout << "Student one is " << fig << ">=" << " student two" << endl;
    fig=(s1<=s2)?":<=":"not ";
    cout << "Student one is " << fig << "<=" << " student two" << endl;
    cout << "Student one is " << fig << "!=" << " student two" << endl;
}
/// End of Program 4.8-3 /////

```

程序的执行结果是：

```

The score of student one is 80.5
The score of student two is 83.3
Student one is not > student two
Student one is < student two
Student one is not == student two
Student one is not >= student two
Student one is <= student two
Student one is != student two

```

从程序中可以看出，直接将 Student 类对象 s1、s2 向标准输出对象 cout 进行输出时，操作符函数 operator<<并不接受 Student 类对象，所以编译器便将 Student 类对象转换成相应的 double 类型数据输出。在进行关系运算时采取的方法是一样的。利用这种方法实现对象之间大小的比较的时候，还要注意这样几点：

①类型转换函数 operator double()必须定义为类 Student 的成员函数，而不能定义为它的友元函数。

②类型转换函数 operator double()一定要定义成公有（public）类型，否则在函数调用时，无法完成自动类型转换。

③当 Student 类可以转换成多种其他的可以比较大小的类型的时候，自动的类型转换也无法完成，因为编译器无法确定调用哪一个转换函数。这时要进行对象之间的比较，要么就重载关系运算符，要么进行显式类型转换。

4.9 类的静态成员的定义和使用

类的静态数据成员不单独属于该类的任何一个对象，从概念上说，它属于整个类。从变量的存储位置和生命期来看，它类似于全局变量，因为一方面它存放在程序的全局/静态区，另一方面，只要程序的运行没有结束，类的静态变量就有效。如果将类的静态数据成员的访问权限设为 public，那么它就可以被任何函数所访问。

类的静态数据成员的初始化工作分两种情况进行：

（1）类的只读静态数据成员的初始化

这种静态数据成员的初始化可以在类体内部进行，也可以在类体外部进行。这是非常特殊的一种初始化方式。因为 C++的语法认为类体中的相关语句是对变量的声明而不是定义。所以，一般情况下，不允许在类体内部指明成员变量的初值。见下面的例子。

```
//// Program 4.9-1 /////
#include <iostream>
using namespace std;
class A{
public:
    static const int i;
    static const int j=6;
};
const int A::i=5;
int main(){
    cout << A::i << endl;
    cout << A::j << endl;
}
/// End of Program 4.9-1 ///
```

程序的输出结果是：

类 A 的静态常量成员变量 j 的初始值是直接写在类体内部的。这种语法是强调类的静态常量数据成员是紧密与类相关的属性。如果是在类体之外为静态常量成员赋初值，变量前面的 const 关键字不能省略，就如同定义一个全局的常变量一样。

要注意的是，在类体中为静态常量成员指定初值，只能使用编译时常量。如果是利用函数的返回值为类的静态常量成员提供初值，必须将初始化语句写在类体之外，否则会有编译错误。见下面的程序。

```
//// Program 4.9-2 /////
#include <iostream>
using namespace std;
int getint(){
    int tmp;
    cin >> tmp;
    return tmp;
}
class A{
public:
    static const int i;
    static const int j=getint();      //statement1
//    static const int j;           //statement2
};
const int A::i=5;
//const int A::j=getint();        //statement3
int main(){
    cout << A::i << endl;
    cout << A::j << endl;
}
/// End of Program 4.9-2 ///
```

此程序在 statement1 处发生编译错误，错误提示是：应输入常量表达式。而如果将 statement1 删除，并解除 statement2 和 statement3 的注释，也就是说，将静态常量成员数据 j 的初始化语句移到类体外面去，程序就可以通过编译并正常运行。所以，为了统一起见，将静态数据成员的初始化工作放到类体之外是一种较为安全的做法。

(2) 类的普通静态数据成员的初始化

这类静态数据成员的初始化工作一定要在类体之外进行。因为，类体中的相关语句只是静态变量的一个声明，要在类体之外定义它（为它分配空间，并同时指定初始值）。考察下面的程序。

```
//// Program 4.9-3 /////
#include <iostream>
using namespace std;
class Student{
    double weight;
```

```

static double total_weight;
static int total_number;

public:
    Student(double w){
        weight=w;
        total_weight+=w;
        total_number++;
    }
    void display(){
        cout<<"The student's weight is "<<weight<<" pounds.\n";
    }
    static void total_disp();
};

int Student::total_number=0;
double Student::total_weight=0.0;
void Student::total_disp(){
    cout<<"total number is "<< total_number<<endl;
    cout<<"total weight is "<< total_weight<<endl;
}

int main(){
    Student s1(18.8),s2(16.6),s3(15.5);
    s1.display();
    s2.display();
    s3.display();
    Student::total_disp();
    return 0;
}
/// End of Program 4.9-3 ///

```

程序的执行结果是：

```

The student's weight is 18.8 pounds.
The student's weight is 16.6 pounds.
The student's weight is 15.5 pounds.
total number is 3
total weight is 50.9

```

在阅读上面的程序时要注意这样几个要点：

①static 关键字用于在类体内部对静态数据成员或静态函数成员进行声明。当在类体之外对静态成员进行定义时，不要再使用关键字 static，否则会报编译错误。

②同全局变量或全局静态变量一样，在定义类的静态变量的时候如果不显式给出初值，则它的初值就为 0。

③类的静态成员函数不能直接访问类的普通成员变量，而只能访问类的静态成员变量。实



际上，类的公有的静态成员变量可以在任何函数中被访问，只不过在同一个类的静态成员函数中被访问时，不用显式给出类名。而类的非公有静态成员变量只可以在本类的成员函数中被访问，当然，也不用在静态变量前显式使用类名。

④虽然可以通过点操作符访问“对象的”静态成员，但本质上静态成员与对象无关，或者说，类的所有对象“共享”一份静态成员。因此，建议在使用类的静态成员时直接通过类名（而不是对象名）来访问，这样在访问成员变量（或函数）的同时就可以得知它的静态特性，从而使程序有更好的可读性。

在语法上，类的静态成员变量也可以“继承”，也就是说，通过派生类的类名来访问基类中声明的静态变量。这时，在基类中声明的静态成员变量，不管是通过基类来访问，还是通过派生类来访问，实际上指的是同一个变量。看下面的例子。

```
/// Program 4.9-4 ///
#include <iostream>
using namespace std;
class A{
public:
    static int num;
};
int A::num=10;
class B:public A{
public:
    static void change(){
        num=12;
    }
//int B::num=14; //错误！不能够多次初始化静态变量
int main(){
    cout << B::num << endl;
    B::change();
    cout << A::num << endl;
    cout << B::num << endl;
}
/// End of Program 4.9-4 ///
```

程序的输出结果是：

10
12
12

在这个程序中，`A::num` 和 `B::num` 是同一个变量。由于 `A::num` 和 `B::num` 的访问权限都是 `public`，所以我们可以自由地在任何函数中访问它们。

如果，在派生类中定义一个与基类中的静态变量同名的静态变量，那么就形成“同名隐藏”。这时通过基类的类名访问到的静态变量和通过派生类的类名访问到的静态变量就不是



同一变量，需要加以区别对待。如下面的程序。

```
/// Program 4.9-5 ///
#include <iostream>
using namespace std;
class A{
public:
    static int num;
};
int A::num=10;
class B:public A{
public:
    static double num;
    static void change(){
        num=12;
    }
};
double B::num=14; //必须初始化静态变量
int main(){
    cout << B::num << endl;
    B::change();
    cout << A::num << endl;
    cout << B::num << endl;
}
/// End of Program 4.9-5 ///
```

程序的输出结果是：

```
14
10
12
```

可见，A::num 和 B::num 是不同的变量，它们要分别初始化，在使用的时候也要严格区分。在类 B 的成员函数中，如果要访问类 A 的静态成员变量 num，必须显式地写明 A::num，只有这样才能突破同名隐藏的限制。

4.10 类的设计与实现规范

规范是一种规定，遵守这种规定能够带来长远的利益。而违反这种规定却不会立即受到惩罚。程序设计的规范是人们在长期的编程实践中总结出来的，深入理解这些规范需要认真的思考和大量的实践。不符合程序设计规范的代码也能够通过编译并“正常”运行。但从长远看，不合规范的程序会逐渐带来严重的后果：可读性差、代码安全性差、不易扩展、不易使用、不易维护，等等。有关 C++ 编码规范的详细讨论，请参阅 11.5 节。在这里，主要讨论在设计和实现类的时候要遵循的规范。

类是面向对象程序设计的最主要的元素。设计出“性能”优良的类，并以适当的方式实现它，是编写出高质量程序的关键。在设计和实现类的时候，要遵循的规范非常多，以下列出几条较为重要的规范。

(1) 尽量将数据成员声明为私有的

数据成员表征了类对象的状态，这些状态对外界应该是透明的（不可见的）。在设计一个类的时候，把它的数据成员的访问权限声明为 public 和 protected，都会使类的封装性遭到破坏。所以，应该尽量将所有的数据成员声明为私有（private）的。考察下面的程序。

```
/// Program 4.10-1 ///
#include <iostream>
using namespace std;
class TDate{
public:
    void show(){
        cout << month << "/" << day << "/" << year << endl;
    }
    TDate(){
        year=2000;
        month=1;
        day=1;
    }
    int year;
    int month;
    int day;
};
int main(){
    TDate t1;
    t1.show();
    t1.month=13;
    t1.show();
}
/// End of Program 4.10-1 ///
```

程序的输出结果是：

1/1/2000

13/1/2000

TDate 是一个表示日期的类，设计者一定希望这个类的客户利用这个类创建“正确”的日期对象。不幸的是，13/1/2000 不是一个正确的日期，因为不存在 13 月份。产生这个错误的原因，是 TDate 类的设计存在缺陷。将数据成员 year、month 和 day 的访问权限设置成 public，意味着有无数的函数可以不加限制地访问日期对象的这三个数据成员。这样，就无法保证每次对数据成员的设置都是正确的。如果增加一个公有的成员函数 void set(int m,int d,int y){...}，每次要修改数据成员都只能调用这个函数进行，那么能够修改数据成员的函数就只有一个，



只要精心编写这个函数，排除各种误用的可能性，就能保证正确地创建和设置 TDate 类对象。另外，从不同代码单元之间耦合度的角度来看，将数据成员设置为公有，意味着类的所有客户都直接依赖数据成员，一旦数据成员的定义发生改变，类的所有客户端的代码（即使用了该类的代码）都要重写。

同样地，将数据成员声明为 `protected`，也破坏了类的封装性。因为该类的所有派生类都可以访问这些数据成员，而一个类的潜在的派生类的数量是巨大的（可以说是无限的，只要这个类还继续供客户使用）。这就意味着，一旦数据成员的定义发生变化，所有的派生类的代码都要重写。

所以，只要有可能，就尽量将所有数据成员的访问权限设置成 `private`。

(2) 将成员函数放到类外去定义

这里的成员函数指的是类的非静态成员函数，在语法上，它们既可以在类体内定义，也可以放到类外去定义。但将成员函数放到类外去定义是一个更好的选择。主要原因如下。

①将类的成员函数写在类的内部影响可读性。一般来说，类的定义放在头文件中，类的客户包含这个头文件，就可以使用类的功能。对类的客户来说，关心的是怎样使用这个类，就是希望了解类对外的接口（所有公共成员的集合），而对类的成员函数的实现细节是不关心的。因此，类的定义中只需要提供对外接口的描述就可以了。将类的成员函数定义放在类内，会使类定义代码的体积增大，影响阅读，也不利于类的修改与维护。

②将类的成员函数写在类的内部会泄露类的实现细节。从软件开发的知识产权和商业利益的角度来看，将类的成员函数定义在类的内部，会泄露类的功能的实现细节，不利于保护类的设计者的合理权益。

③将类的成员函数写在类的内部会有潜在的危险性。C++编译器采用分离编译的模式，这种模式的好处是为大型程序的开发提供了便于管理的代码组织方式。例如，一个源文件在变化之后，只需要重新编译这个源文件就可以了，不需要重新编译其他的源文件，这样就节省了开发时间。但是，分离编译模式也存在问题：各个源文件之间互相是“不知情”的，有很多问题（例如函数重定义等）只有到了连接的时候才会被发现。如果连接器（linker）也不能发现问题，那么这个问题就会被遗留下来并在将来给程序带来危害。因此，在分离编译模式下，编译器有可能被“欺骗”，尽管这种欺骗有时是无意的。考察下面的程序。

```
//// Program 4.10-2 /////
/** redef.h ***/
class A{
    int num;
public:
    void show();
};

/** end of redef.h **/


/** usea.cpp ***/
#include <iostream>
#include "redef.h"
using std::cout;
```

```

using std::endl;
void A::show(){
    cout << num << endl;
}
/** end of usea.cpp **/

/** redef.cpp **/
#include <iostream>
#include "redef.h"
using std::cout;
using std::endl;
void A::show(){
    cout << num << endl;
}
int main(){
    A a;
    a.show();
}
/** end of redef.cpp **/
/// End of Program 4.10-2 ///

```

这个程序由两个 cpp 源文件构成，它们都包含头文件 redef.h，并且使用了在该头文件中定义的类 A。由于类 A 的成员函数 show() 是在类外定义的，而且在 usea.cpp 中和 redef.cpp 中各被定义了一次，尽管函数的两次定义是完全一样的，但编译器还是会报错：找到一个或多个多重定义的符号。这是一个典型的函数重定义错误，与 C 语言中函数的重定义基本相同。

在 C++ 中，允许将函数定义在类内部。如果有多个源文件包含了这个类的定义（即包含了定义该类的头文件），那么这些源文件各自生成的目标文件（obj 文件）中就都有一份类的成员函数的实现代码。如果以传统的 C 语言的编译方式来看待这些 obj 文件，必然导致类的成员函数的重定义错误。但是由于 C++ 的语法允许在类内部定义成员函数，C++ 的编译器必须容忍这种行为，并以适当的方式处理它们。

设想这样一种情况：有两个源文件中定义了两个同名的类，而在这两个类中又有函数原型完全相同的两个成员函数，且这两个成员函数都是在类的内部定义的。那么，在两个源文件编译连接之后，这两个成员函数的执行正常吗？考察下面的程序。

```

/// Program 4.10-3 ///
/** confuse1.h **/
#include <iostream>
using std::cout;
using std::endl;
class A{
    int num;
public:

```



```
A(){num=5;}
void show(){
    cout << num << endl;
}
};

/** end of confuse1.h **/

/** confuse1.cpp **/
#include "confuse1.h"

void useClass();

int main(){
    A a;
    a.show();
    useClass();
}

/** end of confuse1.cpp **/

/** confuse2.h **/
#include <iostream>
using std::cout;
using std::endl;
class A{
    int num;
public:
    A(){num=6;}
    void show(){
        cout << "another num=" << num << endl;
    }
};

/** end of confuse2.h **/

/** confuse2.cpp **/
#include "confuse2.h"
void useClass(){
    A a;
    a.show();
}

/** end of confuse2.cpp **/
```



//// End of Program 4.10-3 ////

在这个程序当中，存在两个源文件中定义的类 A。应该说，这两个类 A 是不同的。最理想的情况是编译器能够发现这种错误，并及时通知程序员。但在分离编译模式下，如果将类的成员函数的实现写在类的内部，编译器就会忽略这种重定义错误，从而错过发现这种错误的机会。程序的输出结果是：

```
5
5
```

如果将源文件 confuse1.cpp 和 confuse2.cpp 的内容互换，重新编译运行，程序的输出结果是：

```
another num=6
another num=6
```

可以看出，类的成员函数的定义是以第一个包含其定义的 obj 文件为准的（依据在连接的过程中 obj 文件的出现顺序），而忽略其他 obj 文件中对该成员函数的重复定义。这是 C++ 编译器对 C++ 语法做出的让步，它的代价是不能充分检查类定义的不一致性。

如果将成员函数的定义放在类的外部，则编译器很快能发现这种重定义错误，从而及早发现程序设计中深层次的问题。所以，在类的实现中，应该将类的成员函数的定义放到类定义的外部去，如果要定义内联函数，只要在成员函数的定义前显式地使用 inline 关键字就可以了，这并不是一件麻烦的事情。

4.11 抽象类与纯虚函数

抽象类和纯虚函数是两个紧密相连的概念。含有纯虚函数的类就是抽象类。而纯虚函数是一个只声明而不定义的函数。纯虚函数按如下方式声明：

```
class classname{
    virtual type funcname(param-list)=0;
    .....
}
```

也就是说，在普通的虚函数声明的末尾加上“=0”的标志，就构成了纯虚函数的声明。由于纯虚函数并没有在当前的类中实现，所以它只能留到派生类中去实现。也因为纯虚函数是一个没有定义的函数，所以包含该函数的类不能被实例化，这也是“抽象类”这个名称由来的原因。下面是一个使用抽象类和纯虚函数的例子。

```
//// Program 4.11-1 /////
#include <iostream>
#include <math.h>
using namespace std;
class Shape{
public:
    virtual double Area()=0;
};
class Triangle:public Shape{
```



```
double a,b,c;
public:
    Triangle(double e1,double e2, double e3){
        a=e1;
        b=e2;
        c=e3;
    }
    double Area(){
        double s=0.5*(a+b+c);
        return sqrt(s*(s-a)*(s-b)*(s-c));
    }
};

class Rectangle:public Shape{
    double a,b;
public:
    Rectangle(double e1,double e2){
        a=e1;
        b=e2;
    }
    double Area(){
        return a*b;
    }
};

int main(){
    Shape *p;
    Triangle t(3.0,4.0,5.0);
    Rectangle r(3.0,4.0);
    p=&t;
    cout << "The area of triangle is: " << p->Area() << endl;
    p=&r;
    cout << "The area of rectangle is: " << p->Area() << endl;
}

/// End of Program 4.11-1 ///
```

程序的执行结果是：

```
The area of triangle is: 6
The area of rectangle is: 12
```

程序中 `Shape` 是二维几何图形类，只定义了一个成员函数 `Area()`，用来计算几何图形的面积。由于各种二维几何图形的面积计算方法相去甚远，所以无法统一编写该函数，而把实现该函数的任务交给各个派生类。这样，`Area()` 就成了一个纯虚函数，而 `Shape` 就是一个抽象类。



Triangle（三角形）和 Rectangle（矩形）是 Shape 的派生类，在这两个类中分别定义了相应的面积计算函数 Area()，该函数是基类中纯虚函数 Area() 的覆盖版本。程序的测试数据表明，这两个类都能正常工作。

在阅读上面的程序时，要注意这样几点：

①只有虚函数才能够只声明而不实现。普通的成员函数加上“=0”的标记是非法的。

②如果在派生类中并没有将基类中的纯虚函数覆盖，那么这个派生类本身也是一个抽象类，相当于将纯虚函数继承过来。

③有些文献将抽象类定义成“不能直接用来创建对象的类”。从这个意义上说，包含纯虚函数的类只是抽象类的一种形式。如果将一个类的构造函数或析构函数的访问权限定义为 protected（或 private），那么这样的类也不能直接用来创建对象，因此也是抽象类。这样的抽象类可以作为其他类的基类，因为它的构造函数或析构函数可以被派生类所调用。不过，如果不作特别说明，C++ 中提到的抽象类都是指包含纯虚函数的类（本书也遵循这一规范）。

④抽象类的主要作用是将有关的操作作为一个统一的接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，而由派生类来具体实现在基类中声明的接口。所以，抽象类实际上刻画了一组子类的操作接口的通用语义，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。

⑤虽然抽象类不能被实例化，但是仍然可以定义抽象类的指针（或引用），该指针（或引用）指向派生类的对象。如本例中的指针 p。在很多场合下，可以定义一个基类（可以是抽象类）的指针数组，而数组中的每个元素分别指向不同派生类的对象，利用调用基类中的虚函数实现统一的操作。

如果在基类中有纯虚函数，那么在派生类中定义该虚函数是真正意义上的“覆盖”，所以，C++ 允许一种较为特殊的语法强调“显式重写”。下面是一个例子。

```
//// Program 4.11-2 /////
#include <iostream>
using namespace std;
class A{
public:
    virtual void func()=0;
};
class B: public A{
public:
    void A::func(){
        cout << "Implementing A::func()" << endl;
    }
};
int main(){
    B b;
    b.func();
}
//// End of Program 4.11-2 ////
```



程序的输出结果是：Implementing A::func()。

在类 B 的类体中，看到函数 func 的函数头 void A::func()，就可以肯定 func() 在基类中是一个纯虚函数，否则，它怎么能到别的类中去定义呢？

实际上，如果在类 A 中 func() 不是一个纯虚函数，那么在类 B 中用 void A::func() [...] 的形式定义它会导致编译错误。也正是由于 A::func() 是一个纯虚函数，所以在 main() 函数中，函数调用 b.A::func(); 会导致一个连接错误，编译器提示无法找到该函数的定义。

4.12 类对象的内存布局

对象的内存布局是指对象的各成员在内存中是如何存放的。由于对象中有成员数据和成员函数，而且编译器在创建对象时还会在对象中插入一些“看不见”的内容，所以我们要通过具体的分析了解对象到底是如何存放的。考察下面的程序。

```
//// Program 4.12-1 /////
#include <iostream>
using namespace std;
// 定义一个宏将某个对象的首4个字节当做整数输出
#define PrintFirstAsInt(obj) {\
    void *p = &(obj);\
    int *q = (int*)p;\
    cout << *q << endl;}

// 一个不定义任何成员的类，其大小为1
class Empty{};

class OneByte{
    char c;
};

// 定义成员函数并不增大对象的尺寸
class OneByteWithFunction{
    char c;
public:
    void show(){
        cout << c << endl;
    }
};

// 定义静态成员不增大对象的尺寸
class OneByteWithStatic{
    char c;
    static int i;
};

// 对象尺寸并不一定等于各成员变量尺寸之和
class WithTwoMembers{
```



```

int i;
char c;
public:
    WithTwoMembers(){
        i=106;
        c='A';
    }
    void show(){
        cout << i << endl;
    }
};

//定义虚函数会导致对象尺寸增加4字节
class WithVirtualFunction{
    int i;
    char c;
public:
    WithVirtualFunction(){
        i=106;
        c='A';
    }
    virtual void show(){
        cout << i << endl;
    }
};

//派生类对象的起始位置包含一个基类对象
class DerivedClass:public WithTwoMembers{
    int newmem;
public:
    DerivedClass(){
        newmem=57;
    }
};

//包含虚函数的派生类对象，在距起始地址偏移量为4的位置包含一个基类对象
class DerivedFromVirtual:public WithVirtualFunction{
    int newmem;
public:
    DerivedFromVirtual(){
        newmem=57;
    }
};

```



```
int main(){
    cout << "sizeof(Empty) is " << sizeof(Empty) << endl;
    cout << "sizeof(OneByte) is " << sizeof(OneByte) << endl;
    cout << "sizeof(OneByteWithFunction) is " << sizeof(OneByteWithFunction) << endl;
    cout << "sizeof(OneByteWithStatic) is " << sizeof(OneByteWithStatic) << endl;
    cout << "sizeof(WithTwoMembers) is " << sizeof(WithTwoMembers) << endl;
    cout << "sizeof(WithVirtualFunction) is " << sizeof(WithVirtualFunction) << endl;
    cout << "sizeof(DerivedClass) is " << sizeof(DerivedClass) << endl;
    cout << "sizeof(DerivedFromVirtual) is " << sizeof(DerivedFromVirtual) << endl;
    WithTwoMembers obj1;
    PrintFirstAsInt(obj1);
    WithVirtualFunction obj2;
    PrintFirstAsInt(obj2);
    DerivedClass obj3;
    PrintFirstAsInt(obj3);
    DerivedFromVirtual obj4;
    PrintFirstAsInt(obj4);
    obj1.show();
    obj2.show();
}
```

//// End of Program 4.12-1 ////

程序的输出结果是：

```
sizeof(Empty) is 1
sizeof(OneByte) is 1
sizeof(OneByteWithFunction) is 1
sizeof(OneByteWithStatic) is 1
sizeof(WithTwoMembers) is 8
sizeof(WithVirtualFunction) is 12
sizeof(DerivedClass) is 12
sizeof(DerivedFromVirtual) is 16
106
4290548
106
4290336
106
106
```

我们可以对这个程序做如下分析：

(1) 不定义任何成员的类，其大小为 1。

类 Empty 没有定义任何成员，但它毕竟是一个类，可以由它创建对象。而对象的尺寸为 0 就失去了存在的意义，所以，`sizeof(Empty)` 的值为 1。在 Visual Studio 2005 的 debug 模式下，



可以看到 `Empty` 类对象的值是 `0xcc`，这是一个占位符，由编译器对栈内存所做的初始化决定（具体细节参见 3.3 节）。在 `release` 模式下是个随机值。

(2) 只定义一个数据成员的类，其大小就是该数据成员的大小。

`sizeof(OneByte)` 的值为 1，原因是 `sizeof(c)` 的值为 1。如果在类中定义多个成员变量，这种情况就会发生变化。

(3) 定义成员函数并不会增加对象的尺寸。

`sizeof(OneByte)` 和 `sizeof(OneByteWithFunction)` 的值都是 1。可见在类中定义成员函数并不会增加对象的尺寸。实际上，每个对象的成员变量的值都会不同，但所有的对象的成员函数的代码却必然是相同的。所以，没有必要在每个对象中都包含成员函数的代码。实际上，成员函数的代码放在代码区，而不是数据区。类对象的尺寸是由其数据成员的尺寸决定的，与成员函数的定义无关。

(4) 定义静态成员不会增大对象的尺寸。

`sizeof(OneByteWithStatic)` 的值为 1，说明静态数据成员并不增加对象的尺寸。实际上，对象的静态数据成员并不是为单个对象所拥有，而是属于整个类的属性。所以，静态成员变量存放在全局/静态数据区，并不存放在任何一个对象内部。关于 C++ 程序的内存布局，请参见 7.1 节。

类的静态成员函数存放在代码区，也不会增加对象的尺寸。

(5) 对象的尺寸并不一定等于各成员变量尺寸之和。

`sizeof(WithTwoMembers)` 的值为 8，而 `sizeof(int)+sizeof(char)` 的值为 5。可见，对象的尺寸并不一定等于各成员变量尺寸之和。这是因为编译器为了便于访问成员变量采取的对齐措施，有关对齐的详细讨论参见 1.7 节。所以，除非采用紧凑的存储方案，否则类对象的尺寸一般会大于各成员变量尺寸之和。

(6) 定义虚函数会导致对象尺寸增加 4 字节。

虚函数调用的入口地址是在运行时决定的，也就是所谓的“动态绑定”。实际上，一个类的所有虚函数的入口地址会放在某个“虚函数表”中，在运行时对象必须能够找到这个虚函数表。因此，在对象中必须附加一个指针（称为虚指针），该指针指向虚函数表的入口地址。所以，定义虚函数会导致对象尺寸增加 4 字节。有关虚函数表和动态绑定的详细讨论，请参见 8.10 节。

(7) 任何一个派生类对象都包含一个基类对象在其中。

从程序的输出可以看出，`sizeof(DerivedClass)==sizeof(WithTwoMembers)+4`，可以看到派生类对象的尺寸是基类对象的尺寸加上派生类中新添加的数据成员的尺寸。输出 `DerivedClass` 对象的第一个成员的值，发现正好是其基类对象的第一个成员变量 `i` 的值。这说明任何派生类对象先包含一个基类对象，然后再存放其他的数据成员。

(8) 包含虚函数的类对象在计算成员变量的偏移量时，要考虑虚指针的影响。

由于虚函数表要通过虚指针才能访问，所以虚指针必须存放在类对象的内部。至于放在对象内部的什么位置，不同的编译器可能有不同的实现。在 VS 2005 下，虚指针存放在对象的起始地址处。在程序中，类 `WithVirtualFunction` 的对象 `obj2` 的首 4 个字节的值是 4290548，而类 `DerivedFromVirtual` 的对象 `obj4` 的首 4 个字节的值是 4290336，都说明了在含有虚函数的类对象中，对象的起始地址处存放的不是对象的数据成员，而是由编译器插入的虚指针。



查看程序的汇编代码，可以发现，在函数 WithTwoMembers::show()和函数 WithVirtualFunction::show()中都有一条语句 cout << i << endl；但是在两个函数中，对类的成员变量 i 的寻址却是不同的。在函数 WithTwoMembers::show()中，获取 i 值的语句是 mov edx,dword ptr [ecx]；而在函数 WithVirtualFunction::show()中，获取 i 值的语句是 mov edx,dword ptr [ecx+4]。这多出的 4 个字节正好是虚指针所占据的空间。

我们知道，在调用对象的成员函数时，主调函数会隐含地把对象的首地址作为 this 指针传递给函数。那么在派生类对象中调用基类的成员函数，和在派生类对象中调用派生类里定义的成员函数，在这两种情况下 this 指针的值相同吗？

另外，可以把一个指向派生类对象的指针无条件地转换成指向基类对象的指针，在这个转换过程中，指针的值是否保持不变呢？考察下面的程序。

```
/// Program 4.12-2 ///
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){
        num=5;
    }
    void show(){
        cout << this << endl;
        cout << num << endl;
    }
};
class B:public A{
    double dig;
public:
    virtual void f0(){}
    void showAddr(){
        cout << this << endl;
    }
};
int main(){
    B b;
    cout << (void*)&b << endl;
    b.showAddr();
    A *p=&b;
    cout << p << endl;
    p->show();
    b.show();
}
```



```
}
/// End of Program 4.12-2 ///
```

程序的输出结果是：

0012FF4C

0012FF4C

0012FF54

0012FF54

5

0012FF54

5

可以看出，在 b.showAddr() 和 b.show() 中显示的 this 指针的值是不一样的。对象 b 的首地址是 0012FF4C，将它转换成 A*类型的指针之后，得到的结果是 0012FF54，这个结果是在对象 b 中，它所包含的基类对象的真正的起始地址。由于对齐的原因，在类 B 的对象中的头 8 个字节用来存放虚指针，而后才是存放 A 类的对象。

4.13 为什么说最好将基类的析构函数定义为虚函数

C++的赋值兼容性原则是：一个公有派生类的对象在使用上可以被当作基类的对象，反之则禁止。具体表现在：派生类的对象可以被赋值给基类对象、派生类的对象可以初始化基类的引用、指向基类的指针也可以指向派生类。C++的赋值兼容性原则不仅符合人类认识客观世界的基本规律，也为程序开发带来了很多操作上的便利。

但是，由于赋值兼容性原则的使用，导致指针（引用）的声明类型和所指对象的实际类型之间可能存在差异。这种差异如果处理不当，就可能造成错误。考察下面的例子。

```
/// Program 4.13-1 ///
```

```
#include <iostream>
using namespace std;
class Student{
    char *StudentName;
public:
    Student(char *n){
        int len=strlen(n);
        len = (len>10)?len:10;
        StudentName=new char[len+1];
        if(StudentName) strcpy(StudentName,n);
    }
    ~Student(){
        delete[] StudentName;
        cout << "StudentName has been deleted" << endl;
    }
};
```

```

class CollegeStudent:public Student{
    char *CollegeName;
public:
    CollegeStudent(char *n1,char *n2):Student(n1){
        int len=strlen(n2);
        len = (len>10)?len:10;
        CollegeName=new char[len+1];
        if(CollegeName) strcpy(CollegeName,n2);
    }
    ~CollegeStudent(){
        delete[] CollegeName;
        cout << "CollegeName has been deleted" << endl;
    }
};

int main(){
    Student *p = new CollegeStudent("王明","武汉大学");
    delete p;
}

/// End of Program 4.13-1 ///

```

程序的运行结果是：StudentName has been deleted。

这个程序的 main 函数中，通过 new 操作生成了一个 CollegeStudent 类的对象，并将对象的地址赋给该类的基类指针 p。在 CollegeStudent 类的构造函数中，实际上为两个成员变量（即 StudentName 和 CollegeName）申请了空间，而由于指针 p 的类型是 Student，在执行语句 delete p 时，调用的是 Student 类的析构函数，这样就只会释放指针 StudentName 所指的空间，而未能释放由指针 CollegeName 所指的空间，从而造成内存泄露。

产生这个问题的原因，是因为指针 p 的类型是 Student，而它所指的对象的实际类型是 CollegeStudent，这种差异导致在释放由 p 所指的对象时，无法调用析构函数的正确版本。解决的办法很简单，就是在基类 Student 中，将析构函数~Student()声明为虚函数，这样该类的所有派生类的析构函数都是虚函数。这样，再碰到 delete p 这样的操作，就能根据 p 所指对象的虚函数表找到正确的析构函数版本。

所以，如果某个派生类的析构函数非常重要（如它要担负释放空间等重要的任务），那么就应该将它的基类的析构函数声明为虚函数，以保证在赋值兼容原则下仍然可以访问到析构函数的正确版本。反之，如果析构函数本身并没有做什么工作，则可以不必将基类的析构函数声明为虚函数。这一点完全由程序员自己把握。

将析构函数声明为虚函数，与将一般的函数声明为虚函数有一点不同。那就是，基类中虚函数的名字与派生类的虚函数的名字不相同。例如上例中，基类的析构函数名字为~Student，而派生类的析构函数的名字叫~CollegeStudent。但是，这并不妨碍实现虚函数调用。一般的虚函数要求派生类和基类中的函数声明完全相同，这是为了找到基类中的虚函数和派生类的虚函数的对应关系。而任何类的析构函数只能有一个，它没有返回值，也不带任何参

数，所以基类与派生类的析构函数很容易对应，只要将基类的析构函数声明为虚函数，自然就可以找到派生类中对应的虚函数。这也算是虚函数定义和使用的一种特殊情形吧。

4.14 对象数据成员的初始值

如果在定义类时，不显式定义任何一个构造函数，那么编译器会为该类提供一个默认的构造函数。但是，这个构造函数实际上什么也不做。那么，生成该类的对象时，它的数据成员会有怎样的初始值呢？我们可以编写一个程序试验一下。

```
/// Program 4.14-1 ///
#include <iostream>
using namespace std;
class A{
public:
    int i;
    double d;
};
A obj;
A *pa=new A;
int main(){
    cout << obj.i << endl;
    cout << obj.d << endl;
    cout << endl;
    cout << pa->i << endl;
    cout << pa->d << endl;
    cout << endl;
    A a;
    cout << a.i << endl;
    cout << a.d << endl;
    cout << endl;
    A *p = new A;
    cout << p->i << endl;
    cout << p->d << endl;
}
/// End of Program 4.14-1 ///
```

在编译该程序的时候，会得到警告信息：使用了未初始化的局部变量“a”。执行程序，会得到如下结果：

```
0
0
```

```
-6.27744e+066
```

```
-858993460
```

```
-9.25596e+061
```

```
-842150451
```

```
-6.27744e+066
```

而且，每次执行的结果都有部分不相同。为此，我们可以得出如下结论：

①由于 C++ 规定不显式初始化的静态变量和全局变量应该初始化为 0，所以，静态（全局）对象的所有数据成员会自动初始化为 0。

②由于局部变量（自动变量）是分配在栈上的，它的初始值与栈的当前状态有关，所以局部对象（如上述程序中的 a）的数据成员的值为随机值。

③用 new 操作生成的对象，其内存被分配在堆上，C++ 并没有规定 new 操作要为它们赋初，所以这些动态对象的数据成员初始值也是随机的。

④与 Java 和 C# 等编程语言不同，C++ 中的对象的数据成员不是在任何情况下都有默认值，所以，编写合适的构造函数，防止出现对象的数据成员在没有初始化的情况下就被使用，是一个好的编程习惯。

4.15 对象产生和销毁的顺序

如果对象是用 new 操作生成的，那么它的空间被分配在堆（Heap）上，只有显式地调用 delete（或 delete[]）才能调用对象的析构函数并释放对象的空间。那么，在程序的其他存储区（全局/静态区，stack 区）上的对象是依据什么样的顺序产生和销毁的呢？考察下面的程序。

```
//// Program 4.15-1 /////
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;
class A{
    string name;
public:
    A(string s){
        name=s;
        cout << "object " << name << " has been created" << endl;
    }
    ~A(){
        cout << "object " << name << " has been destroyed" << endl;
    }
}
```

```

};

void func1(){
    static A sa1("static_object_1");
}

void func2(){
    static A sa2("static_object_2");
}

int main(){
    A la1("local_object_1");
    A la2("local_object_2");
    func1();
    func2();
}

A a1("global_object_1");
A a2("global_object_2");
/// End of program 4.15-1 ///

```

程序的执行结果是：

```

object global_object_1 has been created
object global_object_2 has been created
object local_object_1 has been created
object local_object_2 has been created
object static_object_1 has been created
object static_object_2 has been created
object local_object_2 has been destroyed
object local_object_1 has been destroyed
object static_object_2 has been destroyed
object static_object_1 has been destroyed
object global_object_2 has been destroyed
object global_object_1 has been destroyed

```

在解读以上程序的时候，注意这样几个要点：

①全局对象或全局静态对象不管是在什么位置定义的，它的构造函数都在 main() 函数之前就执行。

②局部静态对象的构造函数是当程序执行到定义该对象的函数内部后才会被调用。

③所有在栈（stack）上的对象都比在全局/静态区存储的对象早销毁。

④不管是栈上的对象，还是全局/静态区的对象，都遵循这样的顺序：越是先产生的对象，越是后被销毁。例如，global_object_2 比 static_object_1 先产生，所以 static_object_1 比 global_object_2 先销毁。

4.16 关于拷贝构造函数

拷贝构造函数又译为复制构造函数（copy constructor），它是在利用已有对象产生一个新对象的时候用到的构造函数。其语义是产生一个内容（数据成员的值）与已有对象相关联（通常是完全相同）的新对象。在定义和使用拷贝构造函数的时候，要注意这样几个问题。

（1）拷贝构造函数只带一个参数，且这个参数只能是同类对象的引用。考察下面的程序。

```
//// Program 4.16-1 ////
#include <iostream>
using namespace std;
class A{
    int a;
    float b;
public:
    void show(){
        cout << a << "," << b << endl;
    }
    A(){a=10;b=9.5f;}
    A(const A &obj){
        cout << "Copy constructor is used" << endl;
        a=obj.a;
        b=obj.b;
    }
};
int main(){
    A obj;
    A dup=obj;
    cout << "obj is ";
    obj.show();
    cout << "dup is ";
    dup.show();
}
/// End of Program 4.16-1 ////
程序的运行结果是:
```

Copy constructor is used
obj is 10,9.5
dup is 10,9.5

在这个程序中，对象 dup 是根据 obj 的内容（数据成员的值）生成的。成员函数只带一个类型为 const A&的参数。如果将参数类型改为 A&，此程序也能够通过编译并顺利运行。



但是那样做是有潜在危险的。一是源对象的值在复制过程中是不会改变的，所以应该使用 const；还有一个更重要的原因是类型 A& 是不能绑定到一个非左值的参数上，如果调用拷贝构造函数时参数是一个特殊表达式，将引发编译错误。考察下面的程序。

```
//// Program 4.16-2 /////
#include <iostream>
using namespace std;
class Complex{
    double real;
    double image;
public:
    Complex(double r=0.0,double i=0.0){
        real=r;
        image=i;
    }
    void Show(){
        cout << real << "+" << image << "i" << endl;
    }
    Complex operator+(const Complex&);
    Complex(Complex &c);
};
Complex Complex::operator+(const Complex& c){
    Complex tempobj;
    tempobj.real=real+c.real;
    tempobj.image=image+c.image;
    return tempobj;
}
Complex::Complex(Complex &c){
    cout<<"copy constructor"<<endl;
    real=c.real;
    image=c.real;
}
int main(){
    Complex c1(1.1,2.0);
    Complex c2(2.2,3.0);
    Complex c3(c1+c2);
    c3.Show();
}
/// End of Program 4.16-2 ///
```

此程序在 32 位 djgpp 编译器下将会报告编译错误：no matching function call to 'Complex::Complex(Complex)'。原因是表达式 $c1+c2$ 的返回结果是一个临时变量（非左值），不能对非

左值建立普通的引用，而必须对它建立常引用。虽然在 Visual Studio 2005 环境下此程序可以通过编译，但这并不是标准 C++ 所允许的。所以，应该编写符合 C++ 标准的程序以获得最大的可移植性。

将拷贝构造函数的参数类型改为值类型，在编译的时候就会得到一条出错信息：非法的复制构造函数。原因是：如果拷贝构造函数的参数是本类的一个对象（传值），那么在拷贝构造函数运行之前，先要在栈上建立实参的副本，这样导致对拷贝构造函数的递归调用，而这样的递归调用是没有出口的。

(2) 一个类如果不显式定义拷贝构造函数，编译器会自动为该类定义一个默认的拷贝构造函数，并在需要的时候使用它。默认构造函数的访问级别是公有的（public），且执行新对象与原对象的数据成员之间的一一对应的简单拷贝（浅拷贝）。

(3) 定义了拷贝构造函数之后，如果一个新生成的对象依赖于某个已经存在的同类（或其派生类）的对象，就会调用拷贝构造函数。典型的情况例如：

```
A a1;
A a2=a1;
A a3(a2);
Func(a1);
```

假设 A 是一个类，由于对象 a2 是利用 a1 产生的，所以产生对象 a2 时会调用类 A 的拷贝构造函数，而产生 a3 时的情形也一样。如果在函数调用时，传递给函数的参数是采用传值的方式（而不是传引用），那么也会调用类的构造函数。因此，如果想禁止某个类的对象的传值调用，可将类的拷贝构造函数的访问级别定义为 private。具体用法参见 3.5 节。

(4) 对象间的赋值操作和拷贝构造函数具有相近的语义。它们的区别在于：赋值操作是将一个对象的值赋予一个已经存在的对象，而拷贝构造函数则是利用一个已经存在的对象来创建一个新对象。在多数情况下，如果重载了赋值操作符函数，那么显式定义拷贝构造函数时只需调用重载后的赋值操作符函数（operator=）就可以了。

(5) 并不是每一个类都必须显式定义拷贝构造函数。显式定义拷贝构造函数通常发生在以下两种情况下：一是要完成“深拷贝”，二是新产生的对象的内容与原对象并不完全相同。

“深拷贝”是相对于“浅拷贝”而言的，其概念和实现参见 8.14 节。

如果新产生的对象与原对象的内容并不完全相同，那么也必须显式定义拷贝构造函数。例如下面的程序。

```
/// Program 4.16-3 ///
#include <iostream>
#include <string>
using namespace std;
class BankAccount{
    string name;
    int number;
    float balance;
public:
    void show(){
        cout << name << "'s account number is " << number << " and balance is " << balance <<
```

```

endl;
}
BankAccount(string name,int number,float balance){
    this->name=name;
    this->number=number;
    this->balance=balance;
}
BankAccount(const BankAccount &obj){
    cout << "Copy constructor is used" << endl;
    name = obj.name;
    number = obj.number+1;
    balance = 0.0;
}
};

int main(){
    BankAccount obj("Bob",1111,100.6f);
    BankAccount dup=obj;
    obj.show();
    dup.show();
}

/// End of Program 4.16-3 ///

```

程序的运行结果是：

```

Copy constructor is used
Bob's account number is 1111 and balance is 100.6
Bob's account number is 1112 and balance is 0

```

类 BankAccount 的拷贝构造函数是用来创建某个银行用户的一个新账户，该账户的号码、余额都和原来的账户的不同，但用户名相同，所以必须显式定义拷贝构造函数来创建新账户。



第5章 | 数组与指针

5.1 数组名的意义

除非特别指明，本章所讨论的数组是指静态数组，即大小（元素个数）在编译时决定的数组。数组名的本质是数组最开始的那个元素的首地址。有时也说数组名代表数组的首地址，但要注意，如果将数组当做一个整体看待，数组的首地址与数组的第一个元素的首地址在概念上是不同的：它们的数据类型不同。由于数组名所代表的地址是在编译阶段确定的，因此，数组名是一个地址常量，不允许为数组名赋值。定义一个数组 A，则 A、&A[0]、A+0 是等价的。

那么数组名还有其他意义吗？在 `sizeof()` 运算中，数组名代表的是全体数组元素，而不是哪个单个元素。例如，定义 `int A[5];` 那么在 32 位平台上，`sizeof(A)` 的值就是 $5 \times 4 = 20$ 。除此之外，数组名就代表数组的首地址，而且是地址常量。由于静态数组的空间是在编译阶段决定的，所以数组名是一个编译时常量，其本身是不可寻址的。直接在数组名前使用取地址操作 &，实际上是把全体数组元素当做一个整体来看待，所获得的地址与数组名本身代表的地址常量在值上是相等的，但数据类型并不相同。见下面的程序。

```
//// Program 5.1-1 ////
#include <iostream>
using namespace std;
int main(){
    int A[4]={1,2,3,4};
    int B[4]={5,6,7,8};
    int (&rA)[4]=A;
    cout << A << endl;
    cout << &A << endl;
    cout << A+1 << endl;
    cout << &A+1 << endl;
    cout << B << endl;
    cout << rA << endl;
    cout << &rA << endl;
}
//// End of Program 5.1-1 ////
```

生成此程序的 release 版可执行文件，某次运行的输出结果是：

0012FF60



```
0012FF60
0012FF64
0012FF70
0012FF70
0012FF60
0012FF60
```

从程序的输出结果来看，`A` 与 `&A` 在数值上是相等的，但 `A` 的数据类型是 `int[4]`，接近于 `int * const`，而 `&A` 的数据类型则是 `int (*)[4]`，它们在概念上是不同的。这就直接导致了表达式 `A+1` 和 `&A+1` 的结果完全不同。假设数组的首地址为 `a`，则 `A+1` 在数值上等于 `a+sizeof(int)`，而 `&A+1` 在数值上等于 `a+sizeof(A)`。由于数组的引用是数组的别名，所以 `rA` 与 `&rA` 在数值上是相等的，而它们的数据类型不同。

5.2 什么是指针

指针就是 C/C++ 中用来存放地址值的变量，相应的数据类型称为指针类型。在 32 位平台上，指针变量由于存储的是内存中某处的地址，所以占用 4 个字节（32 位）。实际上，在 32 位平台上作个实验，会发现 `sizeof(int*)`、`sizeof(double*)`、`sizeof(void*)` 等的值都是 4。所以，在 C/C++ 语言中，任何类型的指针，它的长度都为 4。

常常用类似“指向一个整型量的指针”这样的方式来描述一个指针，其含义是：指针变量中存放的是一个地址，而内存中该地址处存储的是一个整数。所以，与指针变量密切相关的两个要素是：地址和地址处存储的数据的类型。

定义指针的形式是：`type *p;`，其中 `type` 是指针所指向的对象的数据类型，而 `*` 则是指针的标志，`p` 是指针变量的名字。由于 C++ 中允许定义复合数据类型，所以，指向复合数据类型对象的指针的说明可能变得较为复杂。理解指针，关键是理解指针的类型和指针所指向的数据的类型。例如：

```
int (*p)[5]; //指针 p 的类型是 int(*)[5]，指针所指向的数据类型是 int[5]
int *p[5]; //p 是一个有 5 个分量的数组，每个分量的数据类型是 int*（指向整数的指针）
int **p; //指针 p 的类型是 int **，指针所指向的数据类型是 int*。p 是指向指针的指针
```

定义指针变量之后，指针变量的值一般是随机值，这样的值对应的地址不是指针变量可以合法访问的地址。使指针变量的值合法化的途径通常有两个，一是让指针指向一个已经存在的变量，如：

```
int i;
int *p=&i;
```

另外一种途径就是动态申请空间，然后让指针指向这片空间的第一个字节。例如：

```
int *p= new int[10];
```

需要注意的是 `&` 运算符的使用。在 C 语言中，一元运算符 `&` 表示取某个变量（表达式）的地址，所以常和指针配合使用。在 C++ 中，一元运算符 `&` 仍然表示取地址运算。只不过，由于 C++ 中增加了“引用”（Reference）这种机制，而对引用的声明就是使用了 `&` 符号。所以，要注意区分 `&` 的出现到底是完成取地址的操作还是用来声明一个引用。如下面的语句：

```
int * &p;
```

由于出现在&符号前面的是一个数据类型(int*), 所以这里的&是用来定义引用p, 即p是一个指向整型量的指针的引用, &不是取地址的运算。

由于指针是一个变量, 所以指针可以参与一些运算。假设定义指针int *p;, 那么指针变量p能够参与的运算有:

①解引用运算, 即获取指针所指的内存地址处的数据, 表达式为*p, 其数据类型为int。更为详细的讨论参见5.9节。

②取地址运算, 即获取指针变量的地址, 表达式为&p, 其数据类型为int **。

③指针与整数常量相加(减), 表达式为p+i(或p-i), 实际上是让指针变量向递增(减)的方向移动i个int型量的距离, 也就是让指针p在数值上增加(减少)i*sizeof(int)。指针的自增(p++)或自减(p--)也属于这种情况。

④两个指针相减, 如p-q, 其结果是两个指针所存储的地址之间的int型数据的个数。

有一种指针的数据类型是void*, 可将其称为“通用指针”或“泛指针”, 原因是void*指针可以很容易地转换成其他数据类型的指针。具体用法参见2.3节。

如果指针指向的是一个结构或者类的对象, 那么解引用之后访问对象的成员可以采用两种形式:(*p).mem或p->mem。其中p代表指针, mem代表对象的成员。这两种形式是完全等价的。->操作符允许重载, 具体方法参见4.6节。

如果将指针变量赋值为0, 那么这个指针也称为“空指针”, 它代表指针不指向任何具体的地址单元。一般会将0或者(void*)0定义成NULL, 这样NULL就称为C/C++语言中用于表示空指针的常量。

使用指针的关键, 是让指针变量指向一个它可以合法访问的内存地址, 如果不知道将它指向何处, 就干脆将它的值设置成NULL。在某些情况下, 指针的值开始是合法的, 以后随着某些操作的进行, 它就变成“非法”了。这个过程或许会比较隐蔽, 所以在使用指针时要特别注意。考察下面的程序。

```
//// Program 5.2-1 /////
#include <stdio.h>
int *pPointer;
void SomeFunction(){
    int nNumber;
    nNumber = 25;
    pPointer = &nNumber; //将指针pPointer指向nNumber
}
void UseStack(){
    int arr[100]={};
}
int main(){
    SomeFunction(); // 设置pPointer的值
    UseStack(); //重新使用堆栈
    printf("Value of *pPointer: %d\n", *pPointer); //输出的结果是多少?
}
/// End of Program 5.2-1 ///
```



这个程序最后输出的整数值是 0，而不是想像当中的 25。原因是当函数 Somefuncion() 运行结束之后，局部变量 nNumber 已经“失效”（它所占据的空间已经归还系统），所以在函数运行结束之后还想通过指针 pPointer 访问 nNumber 的值，实际上是访问到这片空间被重新分配给其他变量之后的值。所以，要使用指针，必须保证指针所指单元的有效性。

5.3 数组与指针的关系

如前所述，数组名代表的是数组的首地址，而数组 A 的某个元素 A[i]可以解释成*(A+i)，所以数组名本身就可以理解为一个指针（地址），只不过，它是指针常量。所以，在很多情况下，数组和指针的用法是相同的。例如，对一个指针变量 p 来说，p[i]与*(p+i)这两种写法完全是等价的。对一个数组 A 来说，A[i]与*(A+i)这两种写法也是完全等价的。

但数组与指针还是有一些重要的差别，主要表现在这样几个方面：

(1) 数组的空间是静态分配的，也就是说，在编译时决定的。而指针变量在刚定义的时候，还没有属于自己可以合法访问的空间。换句话说，是“悬挂”的。这一点也可以从 sizeof() 的运算中看得出来，对指针 p 来说，sizeof(p) 的值永远是 4，代表的是指针变量 p 本身所占空间的大小。而对于数组 A 来说，sizeof(A) 则代表全体数组元素所占空间的大小。

(2) 数组名代表的是一个指针常量，企图改变数组名所代表的地址的操作是非法的。下面这段程序。

```
//// Program 5.3-1 /////
#include <iostream>
using namespace std;
class A{
public:
    int arr[5];
};
int main(){
    int d[5]={1,2,3,4,5};
    d++;
    cout << d[0] << endl;
    A a;
    a.arr++;
    cout << a.arr[0] << endl;
}
/// End of Program 5.3-1 ///
```

由于 d 和 a.arr 都是数组，代表的是一个地址常量，所以 d++ 和 a.arr++ 的操作都是非法的，会出现编译错误。

需要了解的是，数组名本身并不是一个变量，数组名所对应的地址是编译器能够“算”出来的，所以没有必要单独开辟一个变量存储数组名代表的地址，这样数组名本身是不能寻址的。假设定义数组：

```
int A[10];
```



那么再定义一个引用：

```
int * &r=A;
```

就是错误的，因为变量 A 并不存在。如果确实需要建立数组 A 的引用，应该这样定义：

```
int * const &r=A;
```

此时先在数据区开辟一个无名变量，将数组名 A 代表的地址常量拷贝到该变量中，再将常引用 r 与此变量进行绑定。

(3) 就算不用“*”而是用“[]”进行说明，函数形参中的“数组”仍然是指针。考察下面这段程序。

```
/// Program 5.3-2 ///
#include <iostream>
using namespace std;
void show(int A[]){
    A++;
    cout<<A[0]<<endl;
}
int main(){
    int d[5]={1,2,3,4,5};
    show(d);
}
/// End of Program 5.3-2 ///
```

如果把函数 show() 的形参 A 作为数组看待，那么语句 A++ 就是非法的。而实际上这段程序可以正常运行并输出结果 2，这说明 A 被编译器作为指针而不是数组看待。把上面程序中的函数说明做一个改动，变成如下的形式。

```
/// Program 5.3-3 ///
#include <iostream>
using namespace std;
void show(int A[5]){
    A++;
    cout<<A[0]<<endl;
}
int main(){
    int d[5]={1,2,3,4,5};
    show(d);
}
/// End of Program 5.3-3 ///
```

程序仍然可以正常运行。在 C++ 语言当中，像 void show(int A[5]) 这样的函数声明中的数组长度会被忽略，其效果等同于 void show(int A[])，这时的 A 仍然是被当做指针看待。之所以这样处理，有两个原因：C++ 语言不对数组的下标作越界检查，这是忽略函数形参数组长度的原因之一；将数组作为一个整体进行传递，会导致较大的运行时开销，为了提高程序的运行效率，只需要传递数组的首地址。

思考：把上面程序中的 A[5]改成 A[1]，看看程序运行情况怎样？为什么？

(4) 如果函数的形参是数组实参的引用，那么数组的长度被作为类型的一部分。实际上，对数组建立引用，就是对数组的首地址建立一个常引用。由于引用是 C++ 引入的新机制，所以在处理引用时使用了一些与传统 C 语言不同的规范。在传统的 C 语言中，对数组的下标是不作越界检查的，因此在函数的参数说明中，int [5] 和 int [6] 都被理解为 int []（也就是 int *），C++ 语言也沿用了这种处理方式。但是，int (&)[5] 和 int (&)[6] 被认为是不同的数据类型，在实参与形参的匹配过程中要严格匹配，否则会出现编译错误。请看下面的例子。

```
//// Program 5.3-4 /////
#include <iostream>
using namespace std;
void show( int (&A)[5]){
    cout << "The type is int(&)[5]" << endl;
}
void show( int (&A)[6]){
    cout << "The type is int(&)[6]" << endl;
}
int main(){
    int d[5]={1,2,3,4,5};
    show(d);
}
/// End of Program 5.3-4 ///
```

程序的输出结果是：

The type is int(&)[5]

在 main() 函数中，如果 d 的类型是 int[7]（也就是数组的长度为 7），那么函数调用 show(d) 就会报告编译错误，因为函数 show() 的两种重载形式所接受的参数类型分别为 int (&)[5] 和 int (&)[6]，而不是 int (&)[7]，导致实参与形参无法严格匹配或转换。可见 C++ 标准对数组引用类型的检查是非常严格的。将此程序改造如下。

```
//// Program 5.3-5 /////
#include <iostream>
using namespace std;
void show( int A[5]){
    cout << "The type is int[5]" << endl;
}
void show( int A[6]){
    cout << "The type is int[6]" << endl;
}
int main(){
    int d[5]={1,2,3,4,5};
    show(d);
}
```



/// End of Program 5.3-5 ///

则此程序根本无法通过编译。前面说过，在函数形参的声明中，int [5] 和 int [6]都等同于 int []，所以出现函数的重定义错误。

(5) 在概念上，指针同一维数组相对应。而将多维数组当做一维数组看待时，可以有多种不同的分解方式。考察下面的程序。

/// Program 5.3-6 ///

```
#include <iostream>
using namespace std;
void show1(int A[],int n){
    for(int i=0;i<n;i++)
        cout << A[i] << ' ';
}
void show2(int A[][5],int n){
    for(int i=0;i<n;i++)
        show1(A[i],5);
}
void show3(int A[][4][5],int n){
    for(int i=0;i<n;i++)
        show2(A[i],4);
}
int main(){
    int d[3][4][5];
    int i,j,k,m=0;
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            for(k=0;k<5;k++)
                d[i][j][k]=m++;
    show1((int *)d,3*4*5);cout << endl;
    show2((int (*)[5])d,3*4);cout << endl;
    show3(d,3);cout << endl;
}
```

/// End of Program 5.3-6 ///

从程序的运行结果可以看出，负责输出的三条语句：

```
show1((int *)d,3*4*5);
show2 ((int (*)[5])d,3*4);
show3(d,3);
```

它们的输出结果完全一样，即从 0 一直到 59。这说明把 3 维数组 d 当作一维数组看待，至少可以有 3 种不同的分解方式：

- ①数据类型为 int，元素个数为 $3 \times 4 \times 5 = 60$ ；
- ②数据类型为 int [5]，元素个数为 $3 \times 4 = 12$ ；

③数据类型为 int [4][5]，元素个数为 3。

所以，可以将多维数组看做“数组的数组”。在将多维数组转换成指针的时候，一定要注意多维数组的分解方式，以便进行正确的（强制）类型转换。

(6) 字符数组和字符指针虽然在形式上很接近，但在内存空间的分配和使用上还是有重大差别。如前所述，数组名并不是一个运行时实体，因此数组本身是有空间的，这个空间由编译器负责分配。而指针是一个变量（运行时实体），它所指向的空间是否合法要在运行时决定。错误地使用指针将导致对内存空间的非法访问。下面的程序清楚地显示了字符数组和字符指针之间的差别。

```
//// Program 5.3-7 /////
#include <iostream>
using std::cout;
using std::endl;
int main(){
    char s[]="abc";
    char *p="abc";
    s[0]='x';           //s是字符数组，其空间分配在栈上。对字符数组元素的修改是合法的
    cout << s << endl;
    p[0]='x';           //p是字符指针，指向存储在常量区的字符串。对字符串常量的修改是
    cout << p << endl;
}
//// End of Program 5.3-7 /////

```

这个程序可以顺利地通过编译，但在执行时会出问题。原因是，s 是一个字符数组，在程序运行时，会在栈上分配一块大小为 4 字节的空间给数组 s，对 s 的分量的修改是合法的，只要下标不越界，就不会引起运行时错误。而 p 是一个字符指针，它指向存储在常量区的字符串“abc”，对字符串常量进行修改是非法的，会引起运行错误。在 Windows 环境下，这类错误的典型提示是：“0x00436781”指令引用的“0x0049f090”内存。该内存不能为“written”。如果观察该程序产生的汇编代码（在 Visual Studio 2005 环境下生成），则可看到字符串常量是放在一个名为 CONST 的段内，该段的数据是不允许修改的。具体形式如下所示：

```
CONST SEGMENT
??_C@_02DPKJAMEF@?$_CFd?$AA@  DB  'abc', 00H      ; 'string'
CONST ENDS
```

5.4 数组的初始化

定义数组的时候，为数组元素赋初值，叫做数组的初始化。可以为一维数组指定初值，也可以为多维数组指定初值。例如：

```
int A[]={1,2,3};
```

定义一个长度为 3 的数组，数组元素 A[0]、A[1]、A[2]的值分别为 1、2、3。

```
int A[3]={};
```



定义一个长度为 5 的数组，所有数组元素的值都为 0。

```
int A[5]={1,2};
```

定义一个长度为 5 的数组，A[0]、A[1]的值分别为 1 和 2，A[2]、A[3]、A[4]的值都为 0。

```
int A[]={ {1,2}, {3,4}, {5,6} };
```

定义一个类型为 int [3][2]的二维数组。

```
int A[]={{1},{1},{1}};
```

定义一个类型为 int [3][2]的二维数组，A[0][0]、A[1][0]、A[2][0]三个元素的值为 1，其他元素的值均为 0。

而以下是几种错误的初始化方法：

```
int A[3]={1,2,3,4}; //初始化项的个数超过数组的长度
```

```
int A[3]={1,,3}; //不允许中间跳过某项
```

如果一个数组是某个类对象的数据成员，C++语言并没有提供特别的手段来初始化这个数组。只能在构造函数中分别指定数组元素的值，或者是用 memset() 函数等其他方式来初始化数组。特别是，不能在构造函数的初始化列表中为对象的数组成员进行初始化。

考察下面的程序。

```
//// Program 5.4-1 ////  
#include<iostream>  
using namespace std;  
class A{  
    int num;  
public:  
    A(int n){  
        num=n;  
    }  
    A(){  
        num=100;  
        cout<<"in A constuctor"<<endl;  
    }  
    void show(){  
        cout<<num<<' ';  
    }  
};  
int main(){  
    int a[3]={};  
    float b[3]={5.5};  
    char c[3]={'a'};  
    double d[3]={3.3};  
    cout<<a[0]<<',<<a[1]<<',<<a[2]<<endl;  
    cout<<b[0]<<',<<b[1]<<',<<b[2]<<endl;  
    cout<<c[0]<<',<<c[1]<<',<<c[2]<<endl;
```

```

cout<<d[0]<<','<<d[1]<<','<<d[2]<<endl;
int arr[2][3]={2,6};
cout<<arr[0][0]<<','<<arr[0][1]<<','<<arr[0][2]<<',';
cout<<arr[1][0]<<','<<arr[1][1]<<','<<arr[1][2]<<endl;
A obj[4]={A(1),A(5)};
for(int i=0;i<4;i++)
    obj[i].show();
cout << endl;
}

```

/// End of Program 5.4-1 ///

程序的输出结果为：

```

0,0,0
5,5,0,0
a, ,
3,3,0,0
2,6,0,0,0,0
in A constructor
in A constructor
1 5 100 100

```

如果数组的元素是对象，那么在初始化的时候会调用对象的构造函数。在上面的程序中，把 A obj[4]={A(1),A(5)};改写成：

A obj[4]={1,5};

运行结果是一样的，即把 1、5 作为构造函数的参数，以此对数组元素进行初始化。对于没有提供初始化项的数组元素，并不是将 0 传递给构造函数，而是调用该对象的不带任何参数的构造函数（默认构造函数）。也就是说，A obj[4]={A(1),A(5)};的执行效果与 A obj[4]={A(1),A(5),A(),A()};的执行效果是一样的。

5.5 多维数组与多重指针

多维数组可以理解成数组的数组，而多重指针则是指针的指针。在严格考察数组与指针的区别意义上，数组指的是由编译器分配空间的静态数组。由于数组和指针本身存在着差异，导致多维数组和多重指针在具体实现上表现出更大的差异。考察下面的程序。

```

/// Program 5.5-1 ///
#include <iostream>
using namespace std;
int main()
{
    int A[2][3]={1,2,3,4,5,6};
    int **p;
    p=(int**)A;
    cout << A[0][0] << endl;
}

```

```

    cout << p[0][0] << endl;
}

```

/// End of Program 5.5-1 ///

这个程序在很多环境下并不能正常执行(出现运行时错误),就算能够“正常运行”, $A[0][0]$ 的输出结果和 $p[0][0]$ 的输出结果也是完全不同的。这是因为, $A[0][0]$ 和 $p[0][0]$ 虽然在形式上很类似,但表达的含义是完全不同的。如果把二维数组 A 当作一个矩阵看待,那么 $A[0][0]$ 代表的是这个矩阵最左上角的元素(第一行第一列的元素);而 $p[0][0]$ 的求值过程则是这样的:先取 $p[0]$ 的值,然后再回到 $p[0]$ 所指的内存地址处取一个整数。

以二维数组 A 和二重指针 p 为例,概括地说,多维数组和多重指针的主要差异表现在:

(1) 概念不同。 A 代表数组在内存中的起始地址, A 本身并不是一个变量,它是一个编译时符号,在运行时,并没有一个名为 A 的变量存在。 p 是指针,本身是一个变量,在运行时要占据一定长度的内存空间(32位平台上是4字节)。

(2) 寻址方式不同。以上面的程序为例,数据元素 $A[i][j]$ 的等价表示形式是 $*(&A[0][0]+i*3+j)$,而数据元素 $p[i][j]$ 等价表示形式是 $*(*p+i)+j$,这是完全不同的两种寻址方式,不但地址表达式的结果会不同,而且计算过程也完全不同。 $A[i][j]$ 的寻址是在编译时完成的,而 $p[i][j]$ 的寻址只能在运行时完成。

(3) 多维数组所占据的空间是在编译时静态计算的,所有元素占据连续的存储空间。而多重指针的空间是用new操作分批申请的。同一批申请的数据元素占据连续的空间,而不同批次申请的空间则是相互分离的。

由指针指向的连续空间也称为“动态数组”,特指由new操作分配的动态空间。而平常所说的数组则是静态数组,它的大小是在编译时就能静态计算出来的。还是以上面的程序为例,怎样利用指针申请到“多维数组”呢?这里的关键是:多维数组各数据元素占的空间一定是连续的,而用动态内存分配的方式申请空间,只能在一次new操作中得到连续的空间。所以,利用多重指针永远不可能得到真正的多维数组,而只能用单重指针来实现动态多维数组。见下面的程序。

```

/// Program 5.5-2 ///
#include <iostream>
using namespace std;
int main(){
    int A[2][3]={1,2,3,4,5,6};
    int (*p)[3];
    int i,j;
    p=new int[2][3];
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            p[i][j]=A[i][j];
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            cout << (p[i][j]==A[i][j]);
    delete[] p;
}

```

```

}
/// End of Program 5.5-2 ///

```

程序的执行结果是：111111。这说明 p 是一个与 A 的结构完全相同的动态二维数组。在上面的程序中，如果把 p 的定义改成：

```
int p[][3];
```

则无法通过编译。原因是编译器认为 p 是一个静态的二维数组，而静态数组是必须指明每一维长度的。再者，如果把 p 的定义改成：

```
int *p[3];
```

则 p 本身就是一个静态数组，而不是一个指针。p 代表一个地址常量，用 new 操作改变 p 的值显然是非法的。

如果对 p=new int[2][3]; 这中语法形式感到比较陌生，可以先将类型 int [3] 定义为某种数据类型，如：typedef int intarr[3]；

然后再使用如下的形式申请空间：p=new intarr[2]；

这样就可以清楚地看出指针 p 一次性地申请了 2 个类型为 intarr 的单元。

动态数组 p 所占的空间是连续的，这与 A 占据空间的方式完全相同。这就是为什么说 p 是一个真正的二维数组的原因。如果用多重指针来模拟多维数组，则可以采用如下的方式。

```
/// Program 5.5-3 ///

```

```

#include <iostream>
using namespace std;
int main(){
    int A[2][3]={1,2,3,4,5,6};
    int **p;
    int i,j;
    p=new int*[2];
    for(i=0;i<2;i++)
        p[i]=new int[3];
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            p[i][j]=A[i][j];
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            cout<< (p[i][j]==A[i][j]);
    for(i=0;i<2;i++)
        delete[] p[i];
    delete[] p;
}
/// End of Program 5.5-3 ///

```

程序的输出结果也是：111111。从宏观上，可以把 p 理解成一个与 A 在数学结构上相同的二维数组。但在微观实现上，二者有着很大的差别。p 的空间的申请与释放都要小心组织以防止内存错误。另外，如果 p 的分量 p[i] 和 p[j] 所代表的一维数组的长度并不相同，则形成



所谓“非均匀数组”。这种数组在 Java、C#语言中是存在的，在 C++中要实现类似的功能，只能通过多重指针的形式实现。

5.6 成员数据指针

成员数据指针是 C++引入的一种新机制。它的声明方式和使用方式都与一般的指针有所不同。一个类对象生成之后，它的某个成员变量的地址实际上由两个因素决定：对象的首地址和该成员变量在对象之内的偏移量。成员数据指针就是用来保存类的某个成员数据在类对象内的偏移量的。它只能作用于类的非静态成员变量。通常情况下，使用一般的指针就能解决问题，但在某些情况下，使用成员数据指针可能更为方便。下面就是一个例子。

```
//// Program 5.6-1 ////
#include <iostream>
using namespace std;
class Student{
public:
    int age;
    int score;
};
double average(Student* objs, int Student::* pm, int count){
    int result = 0;
    for(int i = 0; i < count; ++i)
        result += objs[i].*pm;
    return double(result)/count;
}
int main(){
    Student my[5] = {{17,75},{19,85},{20,80},{21,78},{22,83}};
    double age_avg = average(my, &Student::age,5);
    //计算5个Student的age成员的总和，再求平均值：(17+19+20+21+22)/5=19.8
    double score_avg = average(my, &Student::score,5);
    //计算5个Student的score成员的总和，再求平均值：(75+85+80+78+83)/5=80.2
    cout << age_avg << endl;
    cout << score_avg << endl;
}
/// End of Program 5.6-1 ///
```

程序的输出结果是 19.8 和 80.2，分别代表学生的平均年龄和平均分数。而平均年龄和平均分数都是通过同一个函数 average() 来计算的。如果不使用成员数据指针，可能就要编写两个计算函数，或者采用降低程序可读性和可移植性的特殊手段。阅读这个程序，注意这样几个要点：

① 定义成员数据指针时，采用下述形式：type className::* ptr，其中 type 表示指针所指数据的类型，className 表示指针指向哪种类的对象的数据成员，ptr 是指针的名字；为成员



数据指针赋值时，采用下述形式：ptr=&className::member，其中ptr表示成员数据指针名，className表示指针指向哪类对象的数据成员，member是类的某个成员；使用成员数据指针取得对象成员的值时，采用下述形式：obj.*ptr或pobj->*ptr，其中obj是对象名，pobj是指向对象的指针名，ptr是成员数据指针名。

②作为一个变量，成员数据指针在底层实现上，存放的是对象的数据成员相对于对象首地址的偏移量。从这个意义上说，成员的数据指针并不是一个真正的指针。通过该指针获取对象的数据成员时，实际上是将该指针表示的偏移量加到对象首地址上，然后再获取数据成员。可通过如下程序考察成员数据指针变量的实际内容。

```
//// Program 5.6-2 /////
#include <iostream>
using namespace std;
class Student{
public:
    int age;
    double score;
    Student(int a,double s){
        age=a;
        score=s;
    }
};
void PrintContent(void *addr,int n){
    char *p = (char*)addr;
    for(int i=0;i<n;i++)
        cout << (int)p[i] << " ";
    cout << endl;
}
int main(){
    Student me(17,75.5);
    int Student::* pi;
    double Student::*pd;
    cout << sizeof(pi) << endl;
    cout << sizeof(pd) << endl;
    pi = &Student::age;
    pd = &Student::score;
    PrintContent(&pi,4);
    PrintContent(&pd,4);
}
/// End of Program 5.6-2 ///
```

程序的运行结果是：

```

4
0000
8000

```

pi 和 pd 是两个成员数据指针，这两个变量的大小（size）都是 4。通过将这两个变量的 4 个字节的值打印出来，可以看出它们存放的是 Student 类中数据成员 age 的偏移量（0）和数据成员 score 的偏移量（8）。

③对象数据成员指针可以通过常规指针来模拟，例如在上面的程序中，可以将 pm 声明为一个整型量，原来的 obj.*pm 等同于*((int*)((char*)(&obj)+pm))，但是这样编写出来的程序可读性差，可移植性也不高，一不小心容易出错。

④使用成员数据指针，还需要注意一个问题，那就是被访问的成员往往是类的公有成员。如果用成员数据指针访问类的私有成员，使用不当会引发编译错误。如下面的程序。

```

/// Program 5.6-3 ///
#include <iostream>
using namespace std;
class HaveTwoArray{
    int arr1[5];
    int arr2[5];
public:
    HaveTwoArray(){
        for(int i=0;i<5;i++){
            arr1[i]=i+1;
            arr2[i]=i+5;
        }
    }
};

void PrintArray(HaveTwoArray &s,int (HaveTwoArray::*pm)[5]){
    int *p;
    p=s.*pm;
    for(int i=0;i<5;i++)
        cout << p[i] << " ";
}

int main(){
    HaveTwoArray me;
    PrintArray(me,&HaveTwoArray::arr2);
}
/// End of Program 5.6-3 ///

```

· 类 HaveTwoArray 有两个数组成员，因此可编写一个函数打印数组成员的值。但由于 arr2 都是私有的，所以编译时出错。

要解决这个问题，将函数 PrintArray() 设置成类 HaveTwoArray 的友元函数是不行的，因为是在调用该函数的时候访问了类 HaveTwoArray 的私有成员。所以，可定义一个调用了

PrintArray()的友元函数，该函数的参数中并不需要传递类 HaveTwoArray 的私有成员。修改后的程序如下所示。

```
//// Program 5.6-4 /////
#include <iostream>
using namespace std;
class HaveTwoArray{
    int arr1[5];
    int arr2[5];
public:
    HaveTwoArray(){
        for(int i=0;i<5;i++){
            arr1[i]=i+1;
            arr2[i]=i+5;
        }
    }
    friend void PrintArray(HaveTwoArray &s,int i);
};

void PrintArray(HaveTwoArray &s,int (HaveTwoArray::*pm)[5]){
    int *p;
    p=s.*pm;
    for(int i=0;i<5;i++)
        cout << p[i] << " ";
}

void PrintArray(HaveTwoArray &s,int i){
    if(i==1)
        PrintArray(s,&HaveTwoArray::arr1);
    else if(i==2)
        PrintArray(s,&HaveTwoArray::arr2);
}
int main(){
    HaveTwoArray me;
    PrintArray(me,2);
}
/// End of Program 5.6-4 ///
```

5.7 关于 this 指针

this 是 C++中的一个关键字，用在类的非静态成员函数内部，代表当前对象的首地址。然而 this 并不是一个常规变量，不能获取 this 的地址或者给它赋值。一般情况下，没有必要显式地使用 this 指针。但在特定的场合，this 指针是必不可少的。例如，如果一个类的成员

函数返回的是当前类对象的地址(或是当前对象的引用),必须使用 `return this;`(或 `return *this;`)来实现。具体的例子可参见程序 Program 4.3-3 和 Program 4.6-2。再者,如果类的成员函数的参数正好和类成员变量同名,为了将它们区分开来,也要使用 `this` 指针,具体的例子可参见 Program 4.6-1。

实际上,对于类的非静态成员函数而言,在实际调用的过程中, `this` 指针都被编译器隐含地传入到成员函数中,任何成员变量 `v` 实际上被解释成 `this->v`。对于静态成员函数而言,由于没有 `this` 指针可用,所以它不能访问类中的非静态成员。

`this` 指针实际上在对象的构造函数中就可以使用了。构造函数是在对象获得内存空间后,对成员变量进行初始化的过程。下面的程序演示了怎样用一个全局指针管理新产生的某个类(或其派生类)的对象。

```
//// Program 5.7-1 /////
#include <iostream>
using namespace std;
extern class A *pA;
class A{
public:
    A(){pA=this;}
    virtual void show(){cout<<"in A"<<endl;}
};

class B: public A{
    int i;
public:
    B(){i=5;}
    void show(){
        cout<<"in B"<< endl;
        cout<<"i="<<i<<endl;
    }
};

A *pA;
int main(){
    B b;
    pA->show();
}

//// End of Program 5.7-1 /////

```

程序的执行结果是:

in B
i=5

由于在 `class A` 的构造函数中,将新产生的对象的首地址 `this` 赋给了指针 `pA`,所以,程序中任意产生一个类 `A`(或其派生类)的对象, `pA` 的值都会发生变化,指向一个新产生的对象。在这个例子中,类 `A` 和其派生类中都有一个虚函数 `show()`,通过该函数的执行,就能准



确地得知新产生的对象的确切类型。

一般来说，通过指向基类对象的指针，不能访问那些在派生类中新加人的成员变量。但是，通过虚函数调用，却可以突破这种限制。如在上面的程序中，通过指向 class A 对象的指针 pA，间接访问到了 class B 中的数据成员 i。这种结构在设计一个可重用的类库时很有用，例如 MFC 中的 CwinApp 类就用到了这种结构。

5.8 什么是悬挂指针

指针变量代表内存中的一个地址，指针的类型还决定了怎样对内存地址单元的内容作出解释。例如，定义 int *p，已知 p 的值为 0x00436781，那么指针的类型说明了从该地址单元处开始，每次取出 4 字节的内容，将这 4 个字节解释成一个整数。

由于指针对内存单元的访问是一种间接访问，那么这种访问的合法性就是一个问题，而这个问题是 C++ 程序中最容易出现的问题，也是很多人批评 C++ 的一个主要原因。

指针所指向的内存地址，如果不能被当前的程序合法地访问，那么这个指针就是一个“悬挂指针”。悬挂指针有时也称为“野指针”，取“无法正常使用”之意。出现悬挂指针的最典型情形就是在定义指针变量之后没有对它进行初始化。见下面的程序。

```
/// Program 5.8-1 ///
#include <iostream>
using namespace std;
int main(){
    int *p;
    cout << *p << endl;
}
/// End of Program 5.8-1 ///
```

程序中的指针 p 是一个典型的悬挂指针。在定义指针 p 之后，并没有任何语句为它赋初值，然后就用它访问内存单元。这个程序不能正常运行。

如果使用了悬挂指针，虽然编译时往往只是给出一个警告，但程序在运行时通常都会发生严重的运行时错误。如果是往悬挂指针所指的单元内写数据，这种错误的性质要更严重一些，因为它有可能破坏一些操作系统或其他程序所依赖的关键数据或代码。

并不是所有的悬挂指针都是可以由编译器发现的。如果类对象的某个成员是指针，那么更有可能出现悬挂指针的现象，而且往往更隐蔽。见下面的例子。

```
/// Program 5.8-2 ///
#include <iostream>
using namespace std;
class A{
    int num;
    int *p;
public:
    void show(){
        cout << *p << endl;
    }
}
```

```

    }
};

int main(){
    A a;
    a.show();
}

/// End of Program 5.8-2 ///

```

在类 A 中，成员 p 是一个指针，但在构造函数中并没有语句对它进行初始化。对象 a 生成之后，a 的成员 p 是一个悬挂指针，然后在 a.show() 中又通过该悬挂指针对内存单元进行访问，所以出现了运行时错误。

以上程序在编译时，甚至没有出现警告信息，这说明这种类型的悬挂指针比较隐蔽，不是编译器能够发现的。为了防止悬挂指针带来的危害，应注意以下几点：

① C++ 引入了“引用”机制，如果使用引用可以达到编程目的，就可以不必使用指针。由于引用在定义的同时就必须初始化，所以可以避免悬挂指针的出现。

② 如果一定要使用指针，那么要养成在定义指针变量的同时对它进行初始化的习惯。这时有三种选择：一是将它初始化为 NULL，这样便于以后进行判断；二是将它指向一个有名变量，这可根据需要进行；第三是用 new 运算为其申请空间，不过同时要规划好何时使用 delete 运算释放空间。如果某个类的成员是指针，那么应该要在类的构造函数中对它进行初始化，防止它变成一个悬挂指针。

5.9 什么是解引用

解引用（dereference）是与指针相关的一个概念，指的是从指针所代表的内存地址处获取指针所指对象的操作。假设 p 是一个指针，那么 *p 就是解引用操作。解引用操作的结果是获得一个与指针类型相匹配的数据实体。

在对指针解引用时，要注意这样几点：

① 在对指针解引用之前，必须保证指针不是一个空指针或悬挂指针，否则会引起运行时错误。在定义指针的同时将它初始化为 NULL，并在对指针进行解引用操作之前判断它是否为空指针，是一种好的编程习惯。

② 取地址操作和解引用操作是两个互逆的操作，因此，对于任何一个变量 v 而言， $\ast(\&v) \equiv v$ ，对于任何指针 p 而言， $\&(\ast p) \equiv p$ 。

③ 当将一个引用变量与指针解引用之后的数据实体相绑定时，引用变量自身所保存的地址就是指针变量的值。考察下面的程序。

```

/// Program 5.9-1 ///
#include <iostream>
using namespace std;
void increase(int &r){
    r++;
}
int main(){

```

```

int i=5;
int *p=&i;
int &ri=*p;           //在编译之后的汇编语言代码中，变量ri和变量p具有相同的值
ri++;                //ri++等同于(*p)++
cout << *p << endl;
int j=*p;
increase(j);          //在函数中改变的是j的值，与*p无关
cout << *p << endl;
increase(*p);         //对指针解引用并不会产生一个新的无名对象
cout << *p << endl;
}
//// End of Program 5.9-1 /////

```

程序的执行结果是：

6
6
7

要注意的是，函数调用 increase(*p);并不等同于 j=*p;increase(j)。也就是说，对指针解引用并不会产生一个新的无名对象，因为指针所指的内存地址处已经存在一个数据实体。在底层实现上，向一个函数传递某个变量的引用，实际上就是将该变量的地址传进了该函数，所以，传递*p 的引用，实际上就是把 p 本身的数据传进了函数。

如果某个类的成员是指针，那么应该要在类的构造函数中对它进行初始化，防止它变成一个悬挂指针。

5.10 指针与句柄

句柄 (handle) 是 C++ 程序设计中经常提及的一个术语。它并不是一种具体的、固定不变的数据类型或实体，而是代表了程序设计中的一个广义的概念。句柄一般是指获取另一个对象的方法——一个广义的指针，它的具体形式可能是一个整数、一个对象或就是一个真实的指针，而它的目的就是建立起与被访问对象之间的惟一的联系。

在 C++ 中，要访问一个对象，通常可以建立一个指向对象的指针。但是在很多具体的应用中，直接用指针代表对象并不是一个好的解决方案。具体地说，会产生如下几种问题：

(1) 对指针的不当操作会引发一些运行时错误。

例如，我们可能需要建立一个哈希表 (hash-table)，以实现对一组对象的快速定位。但通常我们不会把对象直接存到哈希表中，因为那样做的存取开销太大，特别是涉及到元素的动态插入和删除的时候。一个可行的解决方案是把指向对象的指针存在哈希表中，然后通过指针访问对象的键 (key) 值或进行其他操作。但是，这样做存在一定的风险：是否能保证这些对象只有一个指针指向它？如果还有其他的指针指向它，那么如何能保证对象的键值被修改之后，存在哈希表中的指针能感知得到，从而重新计算在哈希表中的存储位置？应该在什么时候释放对象的空间？

指针是 C/C++ 语言有别于其他语言的最重要的特征之一，也是最容易引起争议的特征之



一。由于 C++ 允许程序员利用指针进行空间的动态申请和释放，并且同一块内存空间可能被多个指针所指向，所以何时释放空间成了一个需要小心对待的问题：如果释放得太早，那么其他的指针仍然指向这片内存，如果再使用它会造成未定义行为；如果一直不释放，可能会使最后一个指向这片内存的指针退出生命期，导致内存无法被释放（即造成内存泄露 Memory Leak）。

（2）直接使用指针可能造成类的实现细节的暴露。

当直接使用指针对类对象的成员进行访问时，必须提供类的部分实现信息。这些信息通常是写在头文件中，供使用它的 cpp 文件包含。但是，在一些特殊的场合下，可能不希望暴露某个特定类的任何实现细节。例如我们编写了一个加密算法，自然不希望在文件中暴露任何线索，以防有人破译我们的代码。或许我们在一个竞争极强的领域编写了一套关键性的代码，而竞争对手会不顾一切地用指针和类型转换存取我们的私有成员。在所有这些情况下，就有必要把一个编译好的实际结构放在实现文件中，而不是让其暴露在头文件中。

随之而来的问题是，如果我们把类的定义放在头文件中，那么当这个类的定义改变时，所有包含这个头文件的 cpp 源文件都要重新编译。对于一个大型项目而言，特别在开发初期，这个问题可能非常难以处理，因为实现部分可能需要经常改动；如果这个项目非常大，用于编译的时间过多就可能妨碍项目的完成。

从设计的角度来说，过早地确定实现的细节有时是不明智的。例如，我们要设计一个哈希表，如果一开始就确定在哈希表中存放指向对象的指针，那么就不得不考虑对象所属类的实现细节。而如果我们只是决定要把对象的键（key）值存放在哈希表中，对象的其他成员可以暂时不用考虑，将使我们的注意力集中到哈希表本身的相关问题上。而且，哈希表所操作的对象的类设计发生改变时，并不会影响哈希表的设计，甚至不需要重新编译实现哈希表的源文件，只要重新连接（link）一下就可以了。这样，我们可以忽略句柄的具体实现，而将注意力集中在利用句柄实现的业务上。

综上所述，如果需要一些惟一标识的东西来获取对象，那么这些东西就被称为句柄。通过句柄，我们可以快速、有效、安全地访问我们想要访问的对象。在 C++ 中，句柄通常是通过类来实现的。这种类被称为“句柄类”（handle classes）。这是因为类除了可以封装数据之外，还可以封装对数据的操作。句柄类的对象就是它要访问的对象的句柄。句柄类的对象一般都包含一个指向被管理对象的指针，由句柄类负责为该对象申请空间和释放空间。使用句柄类所能获得的好处体现在两个层次上：

①可以用一种更加安全的方式使用指针。既然句柄的效果象指针，那么可以使对它的访问看起来也象指针。我们可以重载句柄类的 operator-> 和 operator* 运算符来实现这一点。由于句柄类对象一般都保留了它包含的指针的种种特性，所以也称这种类对象为智能指针（smart pointer）。

②当句柄的实现或句柄所操作的对象发生改变时，维护工作将变得更为简单。可以将句柄的实现和句柄所操作的类的实现放在独立的源文件中，这样可以隐藏句柄所操作的类的实现细节。并且，只要保持句柄类对外的接口不发生变化，就可以使得由句柄的实现细节变化所导致的重新编译达到最少。

下面通过一个具体的例子，来考察一下句柄类可能的实现方式。我们要达到两个目的：一是被管理对象的空间申请和释放工作是由句柄类妥善处理的，二是隐藏句柄所操作的类的实现细节，当句柄类的实现发生改变（特别是句柄所操作的类发生改变）时，使用句柄的类



的代码不用重新编译。

```
/// Program 5.10-1 ///
/** handle.h ***/
#ifndef HANDLE_H
#define HANDLE_H
class handle{
    class wrapped;
    wrapped *working;
public:
    handle();
    void create();
    int read();
    void change(int);
    void destroy();
    ~handle();
};
#endif
/** end of handle.h **/
```

```
/** handle.cpp ***/
#include "handle.h"
class handle::wrapped{
public:
    int i;
};

void handle::create(){
    working = new wrapped;
    working->i=5;
}

int handle::read(){
    return working->i;
}

void handle::change(int n){
    working->i=n;
}

void handle::destroy(){
    delete working;
    working=0;
}

handle::handle(){
```

```

    create();
}
handle::~handle(){
    destroy();
}
/** end of handle.cpp **/

/** usehandle.cpp **/
#include "handle.h"
#include <iostream>
int main(){
    handle u;
    std::cout << u.read() << std::endl;
    u.destroy();
    u.create();
    u.change(9);
    std::cout << u.read() << std::endl;
}
/** end of usehandle.cpp **/
/// End of Program 5.10-1 ///

```

程序的输出结果是：

5
9

在头文件 handle.h 中，用户只能在句柄类 handle 的定义中看到一个指向句柄所操作的类 wrapped 对象的指针。类 wrapped 的类的全部实现细节在 handle.cpp 中，没有任何一个头文件提供任何 wrapped 类的任何信息。handle 类的成员函数 create() 用于为 wrapped 类对象分配空间，而 destroy() 则释放 wrapped 类对象的空间。其他几个成员函数负责对 wrapped 类对象的访问。由于句柄类所做的工作大致上是可以固定下来的，所以 handle.h 可以不发生变化。这样，当 wrapped 类的实现发生变化时，只需要重新编译 handle.cpp，其他任何使用到 handle 类（也就是包含了头文件 handle.h）的源文件都不需要重新编译。

第6章 | 模板与标准模板库

6.1 关于模板参数

模板参数可分为类型参数和非类型参数两种。类型参数代表的是一个基本类型或用户自定义类型，而非类型参数代表一个常量。如下面的例子。

```
/// Program 6.1-1 ///
#include <iostream>
using namespace std;
template <typename T, int Size> class Average{
    T Elements[Size];
    T average_value;
public:
    Average(T *in){
        average_value=(T)0;
        for(int i=0;i<Size;i++){
            Elements[i]=in[i];
            average_value += in[i];
        }
        average_value /= Size;
    }
    void show(){
        cout<<"The elements are: ";
        for(int i=0;i<Size;i++)
            cout<< Elements[i]<<" ";
        cout<<endl<<"The average is: "<<average_value<<endl;
    }
};
int main(){
    int arr_int[3]={3,6,9};
    double arr_dbl[4]={1.1,2.2,4.4,8.8};
    Average<int,3> ia(arr_int);
    Average<double,4> da(arr_dbl);
    ia.show();
}
```



```
    da.show();
}

//// End of Program 6.1-1 ////
```

程序的输出结果是：

```
The elements are: 3 6 9
The average is: 6
The elements are: 1.1 2.2 4.4 8.8
The average is: 4.125
```

以上是一个使用类模板的例子。在类模板的两次实例化过程中，类模板的类型参数 T 分别被实例化为 int 和 double，而非类型参数 Size 则被实例化为 3 和 4。在大多数情况下，以上的使用范例稍做修改就可以解决大部分编程问题。

但模板的非类型参数并不是只能实例化为文字常量，它也可以被实例化为在编译阶段能够静态计算的任何表达式。在上面的例子中，将 Average<double,4> da(arr_dbl); 改写成 Average<double,3+1> da(arr_dbl); 是完全一样的。而下面这个例子更能说明问题。

```
//// Program 6.1-2 ////
```

```
#include <iostream>
using namespace std;
template <typename T, T *Total,int *Number>class Single{
    T single_value;
public:
    Single(T s){
        single_value=s;
        (*Total) +=s;
        (*Number)++;
    }
};

int Int_Total;
int Int_Number;
double Dbl_Total;
int Dbl_Number;

int main(){
    Single<int,&Int_Total,&Int_Number> arr_int[3]={3,6,9};
    Single<double,&Dbl_Total,&Dbl_Number> arr_dbl[4]={1.1,2.2,4.4,8.8};
    cout << "The number of integers is: " << Int_Number << endl;
    cout << "The total value of integers is: " << Int_Total << endl;
    cout << "The number of doubles is: " << Dbl_Number << endl;
    cout << "The total value of doubles is: " << Dbl_Total << endl;
}

//// End of Program 6.1-2 ////
```

程序的执行结果是：

```
The number of integers is: 3
The total value of integers is: 18
The number of doubles is: 4
The total value of doubles is: 16.5
```

因为 Int_Total、Int_Number、Dbl_Total、Dbl_Number 这几个变量是全局变量，它们的地址可以看作常量，所以可以用来实例化模板的非类型参数。在上面的例子中，对数组的初始化过程值得关注。arr_int[3]={3,6,9};的实际意义是用三个参数(3,6,9)来初始化一个对象数组，每个对象的构造函数接收花括号列表中对应位置的实参。这是一种简洁的写法，否则语句会变得冗长。

在实际应用中，我们常用基本数据类型，或自定义类类型来实例化模板的类型参数。如果用一个模板类来实例化某个模板的类型参数，结果会怎样呢？在下面的例子中，将构造一个数组类模板 Array<T,Length>，该模板可用来生成具有特定数据类型和指定长度的数组，并且通过对输出操作符“<<” 的重载，实现对数组元素的打印输出。考察下面的程序。

```
/// Program 6.1-3 ///
#include <iostream>
using namespace std;
template <typename T,int Length> class Array{
    int size;
    T Elements[Length];
public:
    int Size(){return Length;}
    Array():size(Length){}
    T &operator[](int i){
        return Elements[i];
    }
    template <typename T> friend ostream& operator<<(ostream&,const Array<T,Length> &);
};
template <typename T,int Length> ostream& operator<<(ostream &out,const Array<T,Length> &a){
    for(int i=0;i<a.size;i++){
        out << a.Elements[i];
        if(typeid(a.Elements[i])==typeid(int) || typeid(a.Elements[i])==typeid(double))
            out << " ";
    }
    return out;
}
int main(){
    int i,j,k=0;
    Array<double,3> a1d;           //用基本数据类型double来实例化模板
    a1d[0]=1.1;
```



```
a1d[1]=2.2;
a1d[2]=3.3;
cout << a1d << endl;
Array<Array<int,3>,4> a2d; //用模板类类型Array<int,3>来实例化模板
for(i=0;i<a2d.Size();i++)
    for(j=0;j<a2d[i].Size();j++)
        a2d[i][j] = ++k; //像使用普通2维数组一样使用a2d
cout << a2d << endl;
}
/// End of Program 6.1-3 ///
```

程序的执行结果是：

```
1.1 2.2 3.3
1 2 3 4 5 6 7 8 9 10 11 12
```

在上面的程序中，可通过将数组元素的类型指定为 1 维数组，从而产生一个 2 维数组。依此类推，还可以将数组元素的类型指定为 2 维数组，从而产生一个 3 维数组，等等。数组元素的类型和长度可以灵活地根据实际需要加以控制。这种使用模板类来实例化类模板的机制，可以构造出复杂的数据结构，从而为程序员解决复杂的问题提供有效的手段。

在阅读上面的程序的时候，要注意这样几个问题：

①在类模板中声明友元的时候，如果该友元本身是一个函数模板，则应该在 `friend` 前使用 `template` 关键字，以防止编译器将 `operator<<()` 当做一个普通函数看待，详细讨论请参见 6.3 节。并且，在函数模板的模板参数列表中，只需要列出类型参数，而不要列出非类型参数，否则在 Visual Studio 2005 中会发生编译错误。这可能算是 Visual C++ 对模板支持不够好的一个特例。

②`typeid` 是在运行时确定变量的数据类型的操作符，具体用法参见 8.6 节。在这个程序中，使用 `typeid` 来判断当前输出的变量是否已经是基本数据类型（这里只考虑了 `int` 和 `double` 两种情况），如果是，则在两个数据之间加空格符。

③对`[]`操作符的重载，一定要返回对数组元素的引用。因为`[]`操作的结果必须能够作为左值。左值的概念参见 1.9 节。

④在某些编译器中，或者在 VC++ 的较低版本中，像 `Array<Array<int,3>,4> a2d;` 这种将模板实例化的方法可能会引发编译错误。这时可用如下方法试一试，即将实例化的工作分两步进行：

```
typedef Array<int,3> OneArray;
Array<OneArray,4> a2d;
```

如果还是存在编译错误，则可能是编译器不支持用模板类来实例化类模板。

关于模板的类型参数，还有一个重要的话题是：模板的“模板参数”。也就是说，模板的某个类型参数，其本身是另一个类模板，将它显式地写在模板参数列表中。考察下面的程序。

```
/// Program 6.1-4 ///
template<class T, int a> class Array{
    int size;
```

```

public:
    Array():size(a){}
    T val[a];
    T &operator[](int i){return val[i];}
    void show(){
        cout << "The elements of the array is: ";
        for(int i=0;i<size;i++)
            cout << val[i] << " ";
        cout << endl;
    }
};

template<class T, int a> class Student{
    int age;
public:
    Student():age(a){}
    double score;
    void show(){
        cout << "The student's age is: " << age << " and its score is: " << score << endl;
    }
};

template<class T,int a,template <class T, int a> class A> class Container{
public:
    A<T, a> entity;
    void show(){entity.show();}
};

#include<iostream>
using namespace std;
int main(){
    Container<double,3,Array> obj1; //用Array实例化参数A
    obj1.entity[0]=1.1;
    obj1.entity[1]=2.2;
    obj1.entity[2]=3.3;
    obj1.show();
    Container<double,18,Student> obj2; //用Student实例化参数A
    obj2.entity.score=80;
    obj2.show();
}

/// End of Program 6.1-4 ///
程序的输出是:

```



The elements of the array is: 1.1 2.2 3.3

The student's age is: 18 and its score is: 80

在类模板 Container 中，有一个成员 entity，它的类型是在模板被实例化的时候从外部传递进来的。在上面的程序中，类模板头是 template<class T,int a>template <class T, int a> class A> class Container，如果将它改写成 template<class T,int a>template <class T1, int a1> class A> class Container，并且将 entity 声明为 A<T1, a1> entity，则编译器会报告 T1 和 a1 是未知的标识符。这说明 A 作为 Container 的模板参数，它在模板参数列表中仅仅是一个声明，它的实例化必须借助 Container 的类型参数（或者其他可用的类型）来完成。实际上，在声明 A 时只需要告诉编译器 A 是一个模板参数就行了，A 自身的参数的名字是不重要的。把 Container 的说明改成 template<class T,int a>template <class , int> class A> class Container，并且将 entity 声明为 A<T, a> entity，则程序照常编译运行。

模板的模板参数是一种非常灵活的机制，它本身代表一个类模板，但又不是一个一般的类模板。一般的类模板在实例化的时候，它的类型参数和非类型参数将被替换，而类名本身不发生变化。而模板的模板参数在被实例化的时候，不但其类型参数和非类型参数要被替换，类名也必须被替换。在上面的程序中，A<T,a> 就被实例化为 Array<double,3> 和 Student<double,18>。如果在一个类模板的内部要用到另一个类模板，但在实例化的时候，另一个类模板的名字是不变的，那么就没有必要使用模板的模板参数，只需要在类模板的内部直接使用另一个类模板就可以了。例如下面的程序。

```
/// Program 6.1-5 ///
template<class T, int a> class X{
public:
    T valX[a];
};

template<class T,int a> class Y{
    X<T, a> valY;
public:
    void Set(T t){valY.valX[0]=t;}
    void ShowFirst(){cout<<valY.valX[0]<<endl;}
};

#include<iostream>
using namespace std;
int main(){
    Y<double,3> y;
    y.Set(0.8);
    y.ShowFirst();
}

/// End of Program 6.1-5 ///
```

程序的输出结果是：0.8。在此程序中，在类模板 Y 中有一个成员对象 valY，其类型为 X<T,a>，由于 X 本身就是一个类模板，不管 Y 被如何实例化，类模板名 X 不会发生变化，所以，不必使用模板的模板参数。而在 Program 6.1-4 中，A 并不是一个实际的类模板，而是

模板的模板参数，在 Container 被实例化的时候，A 也要被替换成真实的类模板名。请仔细比较这两个程序的差别。

模板的模板参数同时又是一种较为复杂的机制，初学者不太容易掌握。在实际应用中，除非确有必要，否则应该使用更为简单的表示方法，这样更便于程序员之间的交流和程序的维护。

6.2 关于模板实例化

简单地说，模板的实例化就是由函数模板（类模板）生成模板函数（模板类）的过程。对于函数模板而言，模板实例化之后，就会产生一个“真正”的函数（该函数经过编译之后产生可执行的二进制代码）。而类模板经过实例化之后，只是完成了类类型的定义，模板类的成员函数要到调用时才会被实例化。模板的实例化有三种具体的情形：

(1) 隐式实例化。这是针对函数模板而言的。在发生函数调用的时候，如果没有发现相匹配的函数存在，编译器就会寻找同名的函数模板，如果可以成功地进行参数类型推演，就对函数模板进行实例化。

函数模板的隐式实例化一般是发生在直接调用函数的时候。还有一种间接调用函数的情况要引起注意，这时也可以完成函数模板的实例化。所谓函数的间接调用，是指将函数入口地址传递给一个函数指针，通过函数指针完成函数调用。如果传递给函数指针的不是一个“真正”的函数，那么编译器就会寻找同名的函数模板并进行参数推演，进而完成函数模板的实例化。见下面的程序。

```
//// Program 6.2-1 ////
#include <iostream>
using namespace std;
template <typename T>
void func(T t){
    cout << t << endl;
}
template <typename T>
class A{
    T num;
public:
    A(){
        num = T(6.8);
    }
    void invoke(void (*p)(T)){
        p(num);
    }
};
int main(){
    A<int> a1;
```

```
a1.invoke(func);
A<double> a2;
a2.invoke(func);
}

/// End of Program 6.2-1 ///
```

程序的输出是：

```
6
6.8
```

类模板 $A<T>$ 在实例化为 $A<\text{int}>$ 之后，其成员函数 $\text{invoke}()$ 被调用时将被实例化为 $\text{void } A<\text{int}>::\text{invoke}(\text{void } (\ast p)(\text{int}))$ 。也就是说，这个函数接收一个类型为 $\text{void } (\ast)(\text{int})$ 的函数指针。发生函数调用 a1.invoke(func) 的时候，由于并不存在一个名为 func 的函数，所以编译器会将函数模板 $\text{func}<T>$ 与 $\text{a1.invoke}()$ 函数所接收的参数类型进行匹配。由于 $\text{func}<\text{int}>$ 正好可以被 $\text{a1.invoke}()$ 所接收，因而将函数模板 $\text{func}<T>$ 实例化为 $\text{func}<\text{int}>$ 。当 $A<T>$ 被实例化为 $A<\text{double}>$ 之后，在调用 a2.invoke(func) 时， $\text{func}<T>$ 会被实例化为 $\text{func}<\text{double}>$ 。

(2) 显式模板实参。对函数模板而言，在发生函数调用的时候，可以显式给出模板参数，而不需要经过参数推演，如 $\text{func}<\text{int}>()$ 等；而对于类模板而言，在生成一个模板类对象时，一定要指明用什么类型参数实例化该类模板，如 $\text{theClass}<\text{int}> \text{obj}$ 等，要注意的是这时类模板中的成员函数并没有被实例化。它们一直要到被调用的时候才会被实例化。

对函数模板而言，显式模板实参在参数推演不成功的情况下是有必要的。请看下面的例子。

```
/// Program 6.2-2 /////
#include <iostream>
using namespace std;
template <typename T> T Max(const T &t1,const T &t2){
    return (t1>t2)?t1:t2;
}
int main(){
    int i=5;
//    cout << Max(i,'a') << endl;           //这样的调用无法通过编译
    cout << Max<int>(i,'a') << endl;     //这样就OK
}
/// End of Program 6.2-2 ///
```

直接采用函数调用 $\text{Max}(i, 'a')$ 会产生编译错误，因为 i 和 ' a ' 具有不同的数据类型，无法从这两个参数中进行类型推演。而采用 $\text{Max}<\text{int}>(i, 'a')$ 调用后，函数模板的实例化不需要经过参数推演，而函数的第二个实参也可以由 char 型转换为 int 型，从而成功完成函数调用。

(3) 显式实例化。有时也称外部实例化，其含义是在不发生函数调用的时候将函数模板实例化，或者是不使用模板类的时候将类模板实例化。对函数模板而言，不管是否发生函数调用，都可以直接通过显式实例化声明将函数模板实例化，格式为：template 函数返回类型 函数模板名<实际类型列表>(函数参数列表)，如 template void func<int>(const int&);；而对于类模板而言，不管是否生成一个模板类对象，都可以直接通过显式实例化声明将类模板

实例化，格式为：template class 类模板名<实际类型列表>，如 template class theclass<int>;等，类模板中的所有成员函数模板会同时被实例化（只要在同一个源文件中提供其定义）。要注意的是，相同的显式实例化声明在一个源文件中只能出现一次，否则会造成重定义错误。具体的例子参见 6.5 节。

6.3 函数声明对函数模板实例化的屏蔽

C++语言引入模板机制后，函数调用的情形显得比 C 语言要复杂。当发生一次函数调用时，如果存在多个同名函数，则 C++ 编译器将按照如下的顺序寻找对应的函数定义：

- ① 寻找一个参数完全匹配的函数，如果找到了就调用它；
- ② 寻找一个函数模板，并根据调用情况进行参数推演，如果推演成功则将其实例化，并调用相应的模板函数；
- ③ 如果前面两种努力都失败，则试一试低一级的函数匹配方法，如通过类型转换能否达到参数匹配，如果可以，则调用它。

但是，如果使用了函数声明，则可能造成不希望的结果。考察下面的程序。

```
//// Program 6.3-1 /////
#include <iostream>
using namespace std;
int square(const int&);
template<class T> T square(const T &i){
    return i*i;
}
int main(){
    cout << square(5) << endl;
}
/// End of Program 6.3-1 ///
```

在这个程序中，如果没有函数声明 int square(const int&)，则函数调用 square(5) 一定会找到函数模板 square<T> 并将其实例化。但是由于前面那个函数声明的存在，使得编译器认为一定有一个 int square(const int&) 存在（编译器会认为该函数是在别的 cpp 文件中定义的，在连接阶段就能找到），函数调用 square(5) 应该跟这个函数声明相匹配，而不应该启用函数模板的实例化。实际上，函数 int square(const int&) 根本没有定义，这样就造成了连接错误。这种现象，可以把它叫做函数声明对函数模板实例化的屏蔽。其本质是，在发生函数调用的时候，编译器总是优先调用普通函数而不是函数模板。要解决这个问题，可以采用三种办法：

- ① 去掉函数声明；
- ② 显式指明函数模板的类型参数，即将函数调用写成：square<int>(5)；
- ③ 将函数声明改为模板声明，即声明 template <class T> T square(const T &); 这样就会启用函数模板的实例化。

6.4 将模板声明为友元

严格地说，函数模板（或类模板）是不能作为一个类的友元的，就像类模板之间不能发生继承关系一样。只有当函数模板（或类模板）被实例化之后生成模板函数（或模板类），该函数（或类）才能作为其他的类的友元。为了叙述的方便，我们也称一个函数模板（或类模板）是一个类或类模板的友元，其真实的含义是函数模板（或类模板）被实例化后生成的模板函数（或模板类）作为类（或模板类）的友元。

用函数模板作为类模板的友元，在声明友元的时候有两种不同的写法。它们的含义是不一样的，要根据不同的情况选用。先考察下面的程序。

```
/// Program 6.4-1 ///
#include <iostream>
using namespace std;
template <typename T> class A{
    int i;
    friend void show(const A<T>&); //在对A进行实例化操作时，替换友元声明中的类型参数
public:
    A(){i=5;}
    template <class T> void show(const A<T> &a){
        cout << a.i;
    }
    int main(){
        A<int> a;
        show(a); //由于该处的调用与友元声明相匹配，所以省略了将函数模板实例化的操作，即造成屏蔽
    }
/// End of Program 6.4-1 ///
```

在 Visual Studio 2005 平台下，此程序无法通过编译，出错信息是：无法解析的外部符号 "void __cdecl show(class A<int> const &)" (?show@@YAXABV?\$A@H@@@Z)，该符号在函数_main 中被引用。实际上，如果我们把友元函数的定义改写一下，即把原来的友元函数定义：

```
template <class T> void show(const A<T> &a)
```

```
{
```

```
    cout << a.i;
```

```
}
```

改写成：

```
void show(const A<int> &a)
```

```
{
```

```
    cout << a.i;
```

}

那么程序就能通过编译且正确地输出：5。这种结果要从类模板实例化的过程谈起。在类模板 `A<T>` 被实例化为 `A<int>` 时，模板类 `A<int>` 中的友元声明变为：

```
friend void show(const A<int>&);
```

这种形式的友元声明假定函数 `show()` 是一个“已经存在”的函数，即一个使用了模板类的普通函数，而不是一个函数模板。当发生函数调用 `show(a)` 时，编译器会认为这个调用与“已经存在”的函数 `void show(const A<int>&)` 相匹配，从而不启用函数模板的实例化。这就是函数声明对函数模板实例化的屏蔽作用（参见 6.3 节）。

为了避免这种错误的发生，至少有三种可行的解决办法。其一是将友元函数的声明和友元函数的定义合二为一，将友元函数的定义直接放在类模板体内，这样就不会出现友元函数的声明和友元函数的定义之间的不一致性。把上述程序改造如下。

```
/// Program 6.4-2 ///
#include <iostream>
using namespace std;
template <typename T> class A{
    int i;
    friend void show(const A<T>&a){
        cout << a.i;
    }
public:
    A(){i=5;}
};
void main(){
    A<int> a;
    show(a);
}
/// End of Program 6.4-2 ///
```

程序就可以正常编译并运行。当然，将友元函数的定义改成：

```
template<typename T> friend void show(const A<T>&a){
    cout << a.i;
}
```

也是完全可以的。由于无论是将友元函数声明为一个使用了模板类的普通函数，还是一个函数模板，由于将友元函数直接定义在类模板体内，所以不会出现声明和定义之间的不一致。

第二种办法是将友元声明为一个模板，而不是一个使用了模板类的普通函数。改造之后的程序如下。

```
/// Program 6.4-3 ///
#include <iostream>
using namespace std;
template <typename T> class A{
```



```
int i;
template <typename T> friend void show(const A<T>&); //将T换成T1也是对的
public:
    A(){i=5;}
};
template <class T> void show(const A<T> &a)
{
    cout << a.i;
}
int main()
{
    A<int> a;
    show(a); //此处会进行参数推演，从而将函数模板实例化
}
/// End of Program 6.4-3 ///
```

改造之后的程序能够正常编译运行。原因是在类模板 A<T>中，友元 show()被声明为一个函数模板，这就导致调用 show(a)时发生参数推演，并产生模板函数 show<int>(), 该函数是模板类 A<int>的友元函数。

第三种办法是提前将 show() 声明为一个函数模板，然后在类模板 A<T>中使用显式模板参数的语法对友元进行声明。改造之后的程序如下。

```
/// Program 6.4-4 ///
#include <iostream>
using namespace std;
template <class T> class A;
template <class T> void show(const A<T>&);
template <typename T> class A{
    T i;
    friend void show<T>(const A<T>&a);
public:
    A(){i=5;}
};
template <class T> void show(const A<T> &a){
    cout << a.i << endl;
}
void main()
{
    A<int> a;
    show(a);
}
/// End of Program 6.4-4 ///
```

这样，在类模板 A<T>实例化为 A<int>的过程中，编译器会认为由函数模板 show<T>生成的模板函数 show<int>() 是模板类 A<int>的友元函数，从而在发生函数调用 show(a) 的时候，



启动函数模板 `show<T>` 的实例化。

由于友元函数常用来做操作符重载，所以在利用函数模板进行操作符重载时，一定要注意友元声明对函数模板实例化的屏蔽问题。下面是一个例子。

```
//// Program 6.4-5 /////
#include <iostream>
using namespace std;
template <typename T> class UseOutputOverload{
    T var;
public:
    UseOutputOverload(T t){var=t;}
    template <typename T> friend ostream& operator<< (ostream& ,const
    UseOutputOverload<T> &);
};

template<class T> ostream& operator<< (ostream& out, const UseOutputOverload<T>& obj){
    out << obj.var;
    return out;
}
int main(){
    UseOutputOverload<int> o(3);
    cout << o << endl;           //此处不但进行参数推演，而且将函数模板实例化
    UseOutputOverload<double> s(7.8);
    cout << s << endl;           //此处不但进行参数推演，而且将函数模板实例化
}
/// End of Program 6.4-5 ///
```

程序的执行结果是：

```
3
7.8
```

在类模板 `UseOutputOverload<T>` 中，如果将友元的声明改写成：

```
friend ostream& operator<< (ostream& ,const UseOutputOverload<T> &);
```

就会发生友元函数的声明对操作符重载函数模板实例化的屏蔽，从而出现编译错误。如果将友元的声明改写成：

```
friend ostream& operator<<<T> ostream& ,const UseOutputOverload<T> &);
```

也能够达到操作符重载的目的。在函数名 `operator<<` 之后加上 `<T>`，表示用类型参数 `T` 显式实例化函数模板 `operator<<`，从而实现操作符重载。要注意的是，如果是在一般函数模板名后加 `<T>`，该函数模板必须事先声明，否则会出现编译错误。

如果是将一个类模板 `B` 声明为另一个类模板 `A` 的友元，也要注意类模板 `B` 是否已经声明或定义，否则可能引发编译错误。考察下面的程序。

```
//// Program 6.4-6 /////
#include <iostream>
using namespace std;
```

```
template <class T> class A{
    T i;
public:
    A(){i=5;}
    friend class B<T>;
};

template <class T> class B{
public:
    static void show(const A<T> &a){
        cout << a.i << endl;
    }
};

int main(){
    A<int> a;
    B<int>::show(a);
}

/// End of Program 6.4-6 ///
```

编译器在处理 `friend class B<T>;` 这条声明语句时报错，原因是无法确认 `B` 是一个类模板。所以，在此之前应该对类模板 `B` 进行说明。程序改造如下。

```
/// Program 6.4-7 /////
#include <iostream>
using namespace std;
template <class T> class B;
template <class T> class A{
    T i;
public:
    A(){i=5;}
    friend class B<T>;
};

template <class T> class B{
public:
    static void show(const A<T> &a){
        cout << a.i << endl;
    }
};

int main(){
    A<int> a;
    B<int>::show(a);
}

/// End of Program 6.4-7 ///
```



由于提前对类模板 B 进行声明，所以避免了编译错误。如果不想提前声明，还有一种解决方案是在类模板 A 类将 B 声明为：template<class T> friend class B;，同样可以将类模板 B 声明为类模板 A 的友元。

不过，这两种方式在概念上还是有一点差异。第一种方式中，类模板 B 的实例化依赖于类模板 A 的参数 T。也就是说，对于一个特定的模板类 A<T>来说，只有一个 B 的实例 B<T>是它的友元类。而在第二种方式中，对于一个特定的类模板 A<T>类说，B 的任何实例 B<u>都是它的友元类。

6.5 模板与分离编译模式

在 C++ 程序设计中，在一个源文件中定义某个函数，然后在另一个源文件中使用该函数，是一种非常普遍的做法。但是，如果定义和调用一个函数模板时也采用这种方式，会发生编译错误。下面的程序由三个文件组成：func.h 用来对函数模板进行声明；func.cpp 用来定义函数模板；main.cpp 包含 func.h 头文件并调用相应的模板函数。程序代码如下。

```
/// Program 6.5-1 ///
/* func.cpp */
#include <iostream>
using std::cout;
using std::endl;
template <class T> void func(const T& t){
    cout << t << endl;
}
/* end of func.cpp */

/* func.h */
template <class T> void func(const T&);

/* end of func.h */

/* main.cpp */
#include "func.h"
int main(){
    func(3);
}
/* end of main.cpp */
/// End of Program 6.5-1 ///
```

这是一个结构非常清晰的程序，但它不能通过编译。在 Visual Studio 2005 下的出错信息是：无法解析的外部符号“void __cdecl func<int>(int const &)”。这是怎么回事呢？

原因出现在分离编译模式上。在分离编译模式下，func.cpp 会生成一个目标文件 func.obj，由于在 func.cpp 文件中，并没有发生函数模板被调用，所以不会将函数模板 func<T> 实例化为模板函数 func<int>，也就是说，在 func.obj 中无法找到关于模板函数 func<int> 的实现代码。

在源文件 main.cpp 中，虽然函数模板被调用，但由于没有模板代码，也不能将其实例化。也就是说，在 main.obj 中也找不到模板函数 func<int> 的实现代码。这样，在连接的时候当然要报告函数 func<int> 没有定义。

怎样解决这个问题呢？一个最简单的解决方法是：把函数模板 func<T> 的定义写到头文件 func.h 中去。这样，只要包含了该头文件，就会把函数模板的代码包含进来，一旦发生函数调用，就可以依据函数模板代码将其实例化。这个方法虽然简单可行，但对于习惯了分离编译模式的 C++ 程序员来说，却不是一个令人满意的解决方案。原因是函数模板代码写进了头文件，所有使用头文件的人都可以轻易了解函数模板的实现细节。如果能将函数模板的代码写在 cpp 文件中，则程序会有更好的组织结构。

那么，如果仍然采用分离编译模式，有什么办法让函数模板实例化时能够找到相应的模板代码呢？一个可能的解决方案是：使用关键字 export。也就是说，在 func.cpp 里定义函数模板的时候，将函数模板头写成：

```
export template <class T> void func(const T& t)
```

这样做的目的是告诉编译器，这个函数模板可能在其他源文件中被实例化。这是一个对程序员来说负担最轻的解决方案。可惜的是，目前几乎所有的编译器都不支持关键字 export，包括 Visual Studio 2005。

所以，应该考虑用其他的办法解决函数模板的实例化问题。一个可行的方案是，将函数模板 func<T> 显式实例化。既然在 main.obj 和 func.obj 中都找不到模板函数 func<int> 的实现代码，那么就再构造一个源文件（例如叫 force.cpp），在这个源文件中将函数模板 func<T> 显式实例化为模板函数 func<int>。下面是 force.cpp 的源程序：

```
/** force.cpp */
#include "func.cpp"
template void func<int>(const int&);
```

这个源文件没有别的目的，就是将函数模板显式实例化，从而在 force.obj 文件中能找到模板函数 func<int> 的实现代码，这样整个程序就能正常编译运行了。

类模板在分离编译模式下会遇到同样的问题，这个问题也可以利用显式实例化的方法解决。下面是一个具体的例子。

```
/// Program 6.5-2 ///
/** classtemp.cpp */
#include <iostream>
#include "classtemp.h"
using namespace std;
template <class T> void theclass<T>::show(){
    cout << a;
}
/** end of classtemp.cpp */

/** classtemp.h */
template <class T> class theclass{
```

```

public:
T a;
void show();
};

/** end of classtemp.h **/

/** main.cpp **/
#include <iostream>
using namespace std;
#include "classtemp.h"
int main(){
    theclass<int> tobj;
    tobj.a=5;
    tobj.show();
}
/** end of main.cpp **/


```

```

/** force.cpp **/
#include "classtemp.cpp"
template class theclass<int>;
/** end of force.cpp **/
/// End of Program 6.5-2 ///

```

在这个程序中，类模板 `theclass<T>` 的声明是在头文件 `classtemp.h` 中完成的，在源文件 `classtemp.cpp` 中完成对类模板成员函数模板的定义，在源文件 `force.cpp` 中，利用显式实例化由类模板 `theclass<T>` 生成模板类 `theclass<int>`。这样可以解决 `theclass<int>` 类对象的成员函数 `show()` 没有定义的问题。但是这种做法破坏了通常情况下的“惰性实例化”准则，使得目标文件的体积变大，降低了编译速度。所谓惰性实例化，是指调用了模板类中的哪个成员函数，才将其实例化，否则不产生该函数的可执行代码。这样做的好处是可以经济高效地完成编译。

在上面的程序中，用到了模板显式实例化。详细介绍参见 6.2 节。

6.6 关于模板特化

模板特化不同于模板实例化，它指明了函数模板在特殊情况（模板参数为某种特定类型）下的实现版本。有时，一个统一的函数模板并不能在所有类型实例下正常工作。这时，需要定义类型参数在实例化为特定类型时函数模板的特定实现版本。看下面的例子。

```

/// Program 6.6-1 ///
#include <iostream>
using namespace std;
template <class T> T Max(T t1,T t2){
    return (t1>t2)?t1:t2;
}


```

```

}

// const char* 显式特化：覆盖了来自通用模板定义的实例
typedef const char *PCC;
template<> PCC Max<PCC>(PCC s1, PCC s2){
    return (strcmp(s1,s2)>0)?s1:s2;
}
int main(){
    // 调用实例： int Max< int >( int, int );
    int i = Max( 10, 5 );
    // 调用显式特化： const char* Max< const char* >( const char*, const char* );
    const char *p = Max( "very", "good" );
    cout << "i: " << i << endl << "p: " << p << endl;
}
/// End of Program 6.6-1 ///

```

程序的执行结果是：

i: 10

p: very

在模板显式特化定义 (Explicit Specialization Definition) 中，先是关键字 `template` 和一对尖括号`<>`，然后是函数模板特化的定义。该定义指出了模板名、被用来特化模板的模板实参，以及函数参数表和函数体。在上面的程序中，如果不给出函数模板 `Max<T>` 在 `T` 为 `const char*` 时的特化版本，那么在比较两个字符串的大小时，比较的是字符串的起始地址的大小，而不是字符串的内容在字典序中的先后次序。

思考：在上面的程序中，去掉函数模板 `Max<T>` 在 `T` 为 `const char *` 时的特化版本，看看程序的输出结果是什么？

除了定义函数模板的特化版本外，还可以直接给出模板函数在特定类型下的重载形式（普通函数），这样也可以避免函数模板的特定实例的失效。如，把上面的程序改成：

```

/// Program 6.6-2 ///
#include <iostream>
using namespace std;
template <class T> T Max(T t1,T t2){
    return (t1>t2)?t1:t2;
}
// 参数类型为const char* 的重载函数
typedef const char *PCC;
PCC Max(PCC s1, PCC s2){
    return (strcmp(s1,s2)>0)?s1:s2;
}
int main(){
    // 调用实例： int Max< int >( int, int );
    int i = Max( 10, 5 );

```

```

    // 调用重载函数: const char* Max( const char*, const char* );
    const char *p = Max( "very", "good" );
    cout << "i: " << i << endl << "p: " << p << endl;
}

/// End of Program 6.6-2 ///

```

程序 Program 6.6-2 与 Program 6.6-1 的执行结果完全一样。但是，使用普通重载函数和使用模板特化还是有不同之处，主要表现在以下两个方面：

(1) 如果使用普通重载函数，那么不管是否发生实际的函数调用，都会在目标文件中生成该函数的二进制代码。而如果使用模板的特化版本，除非发生函数调用，否则不会在目标文件中包含特化模板函数的二进制代码。这符合函数模板的“惰性实例化”准则。

(2) 如果使用普通重载函数，那么在分离编译模式下，应该在各个源文件中包含重载函数的声明，否则在某些源文件中就会使用模板实例化，而不是重载函数。

有时，一个函数模板有多个特化版本，而一次具体的函数调用必须只能与一个特化版本相匹配，否则会出现无法判断使用哪一个特化版本的错误。考察下面的程序。

```

/// Program 6.6-3 ///
#include <iostream>
using namespace std;
template<typename T>
void func (const T &v1, const T &v2){
    cout<<"template"<<endl;
}

```

//template1

template <>

```
void func<char*> (char* const &v1, char* const &v2){
```

```
    cout<<"Specialization: char*"<<endl;
```

}

//template2

template <>

```
void func<const char*> (const char* const &v1, const char* const &v2){
```

```
    cout<<"Specialization: const char*"<<endl;
```

}

//template3

template <>

```
void func<const char [6]> ( char const (&v1)[6], char const (&v2)[6]) {
```

```
    cout<<"Specialization: const char[6]"<<endl;
```

}

//template4

template <>

```
void func<char [6]> ( char const (&v1)[6], char const (&v2)[6]) {
```

```
    cout<<"Specialization: char[6]"<<endl;
```

```

    }

int main(){
    char *p1 = "hello";
    char *p2 = "world";
    func(p1, p2);
    char * const p3 = "hello";
    char * const p4 = "world";
    func(p3, p4);
    const char *p5 = "hello";
    const char *p6 = "world";
    func(p5, p6);
    const char * const p7 = "hello";
    const char * const p8 = "world";
    func(p7, p8);
    char a1[6] = "hello";
    char a2[6] = "world";
    func(a1, a2);
    const char a3[6] = "hello";
    const char a4[6] = "world";
    func(a3, a4);
    return 0;
}

```

/// End of Program 6.6-3 ///

程序的输出结果是：

```

Specialization: char*
Specialization: char*
Specialization: const char*
Specialization: const char*
Specialization: char[6]
Specialization: const char[6]

```

在此程序中要注意这样几点：

(1) 在声明或定义常变量时，关键字 `const` 写在数据类型的前面和后面是等价的（参见 1.3 节），所以，模板参数说明 `const T &v1` 和 `T const &v1` 的写法是完全等价的。这样，在上面的程序中，函数模板的第一个特化版本的声明可以按照这样的次序产生：`template<typename T> void func (const T &v1, const T &v2)` 等价于 `template<typename T> void func (T const &v1, T const &v2)` `T=char*` `template<char*> void func (char* const &v1, char* const &v2)`。

(2) 在上面的程序中，函数模板的第三个特化版本比较难以理解。实际上，它的实例化过程是这样的：`template<typename T> void func (const T &v1, const T &v2)` `T=const char[6]` `template< const char[6]> void func (const const char(&v1)[6],const const char(&v2)[6])` 等价于 `template<const char[6]> void func(const char (&v1)[6],const char (&v2)[6])` 等价于 `template<`

```
const char[6]> void func (char const (&v1)[6],char const (&v2)[6]).
```

(3) 函数模板的第四个特化版本(template4)与第三个特化版本(template3)在函数参数上完全相同。在发生函数调用 func(a1,a2)或 func(a3,a4)时，仅仅根据函数实参的类型是无法在函数模板的特化版本 func<char[6]>和 func<const char[6]>之间做出选择的。因此，此程序在 djgpp 编译器中和 vs2005 编译器中的输出结果并不一样。在 djgpp 中，函数调用 func(a1,a2) 和 func(a3,a4) 的输出都是：

Specialization: char[6]

在 vs2005 环境下，函数调用 func(a1,a2)和 func(a3,a4)的输出结果不同：应该是将函数调用的实参 a1,a2 (以及 a3,a4) 同函数模板特化时的类型参数 char[6]和 const char[6]进行比较之后的结果。

6.7 输入/输出迭代子的用法

可以将标准输入对象 cin 组装成输入流迭代子 (istream iterator)，也可以将标准输出对象 cout 组装成输出流迭代子 (ostream iterator)，以此来完成输入输出的操作。考察下面的程序。

```
/// Program 6.7-1 ///
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
#include <iostream> // ostream_iterator and istream_iterator
int main(){
    cout << "Enter two integers: ";
    // create istream_iterator for reading int values from cin
    std::istream_iterator< int > inputInt( cin );
    int number1 = *inputInt; //read int from standard input
    ++inputInt; // move iterator to the next input value
    int number2 = *inputInt; // read int from standard input
    // create ostream_iterator for writing int values to cout
    std::ostream_iterator< int > outputInt( cout );
    cout << "The sum is: ";
    *outputInt = number1 + number2; // output result to cout
    cout << endl;
    return 0;
}
/// End of Program 6.7-1 ///
```

依次输入两个整数 3 和 5，则输出加法运算的结果 8。在阅读以上程序时要注意以下几个要点：

①要能构造 istream_iterator 对象和 ostream_iterator 对象，必须包含头文件 iterator。

②将 cin 组装成 std::istream_iterator<int>类型的对象后，通过该对象只能读取 int 类型的

数据，如果输入其他类型的数据，就会在运行时抛出异常。

③可通过*操作符读取标准输入流中的内容，这是因为 `istream_iterator` 模板对*运算进行了重载。而且，从当前位置读取数据的操作可以多次进行。关于*操作符重载的有关内容请参阅 4.10 节。

④可通过++运算将迭代子移动到下一个输入数据处。

⑤将 `cout` 组装成 `std::ostream_iterator<int>` 类型的对象后，通过该对象只能输出 `int` 类型的数据，如果输出其他类型的数据，会出现编译错误。

⑥可通过*操作符向输出迭代子送入输出数据。输出过程是以赋值语句的方式进行的。

可见，使用输入/输出迭代子具有很浓厚的 STL 风格，在某些操作上显得更为紧凑、方便。

6.8 bitset 的简单用法

`bitset` 是 C++ 标准库提供的用于处理位集的类模板。所谓“位集”，是指由 0 和 1 组成的有限长度的有序集，习惯上也称为“标志位”集合。

要使用 `bitset` 类，就必须包含相关的头文件。一般可以使用如下两条语句：

```
#include <bitset>
using std::bitset
```

`bitset` 有几种形式的构造函数。由于 `bitset` 是一个类模板，所以在使用 `bitset` 的时候，必须对其实例化。与一般类模板不同的是，`bitset` 类模板只带一个非类型参数（整型常量），表示 `bitset` 类对象中封装的二进制位的个数。下面通过一个具体的例子来介绍 `bitset` 的简单用法：

```
//// Program 6.8-1 /////
#include <iostream>
#include <bitset>
using std::bitset;
using std::cout;
using std::endl;
int main(){
    std::string bitval;
    bitset<8> bs1;          //建立一个全的bitset
    bitset<8> bs2(012);     //用unsigned long初始化bitset
    bitval="1111";
    bitset<8> bs3(bitval);  //用string对象初始化bitset
    bitval="0000101011";
    bitset<8> bs4(bitval,5,4); //取string对象的一部分初始化bitset
    bitset<8> bs5(bitval,5);  //从某处一直取到字符串末尾，所得的子串用来初始化bitset
    cout << bs1 << endl;
    cout << bs2 << endl;
    cout << bs3 << endl;
    cout << bs4 << endl;
```

```

cout << bs5 << endl;
bs1[0]=1;                                //将bitset作为数组处理
cout << bs1 << endl;
cout << bs2.count() << endl;      //计算bitset中的个数
bs3.flip(0);                            //在某位取反
cout << bs3 << endl;
cout << bs4.to_ulong() << endl;    //转换为对应的无符号长整数输出
cout << bs5.size() << endl;        //取bitset的长度
}

/// End of Program 6.8-1 ///

```

程序的运行结果是：

```

00000000
00001010
00001111
00000101
00001011
00000001
2
00001110
5
8

```

阅读上面的程序，注意这样几个要点：

①bitset 的长度是由模板参数指定的，可以指定任意长度的位集。最简单的是先生成一个全 0 的位集，然后再通过后继的操作将某些位置 1。

②用 string 对象初始化 bitset 时，要注意在 string 中位的序号是从左向右编排的，而在 bitset 中位的序号是从右向左的，在计算的时候不要混淆。当然，它们的序号都是从 0 开始编号。

③用 string 字符串或它的某个子串初始化 bitset 时，如果长度不够，则会自动在最左边添加若干个 0；若长度大于 bitset 的长度，则最左边的部分将被截去。例如，用 `bitset<8> bs2(012);` 创建一个 bitset 对象，012 是一个八进制数，展开以后有 6 位二进制数，那么构造函数会在最左边添加两个 0。

④将 bitset 中某一位置 1，有两种办法。例如，`bs1` 是一个 bitset 对象，那么将它的第 0 位置 1，可写：`bs1[0]=1;`或 `bs1.set(0);`

⑤将 bitset 中某一位置 0，也有两种办法。例如，`bs1` 是一个 bitset 对象，那么将它的第 2 位置 0，可写：`bs1[2]=0;`或 `bs1.reset(2);`

⑥bitset 类有很多成员函数，分别实现计算 1 的个数、返回位集长度、对某一位取反、转换成对应的无符号长整数等功能，在上面的程序中列举了一些。其他的成员函数可查阅相关资料（如 MSDN 等）。

6.9 typename 的用法

typename 是 C++为了实现模板而引入的关键字。一般认为它是在模板参数说明中 class 关键字的替代物，实际上它有两种主要的用途。

要定义模板，就必须使用模板参数。有两种模板参数：类型参数和非类型参数。类型参数是主要的模板参数，一般的模板中都带有类型参数。下面是一个典型的函数模板的定义：

```
template <class T> T Max(T x, T y){  
    return (x>y)?x:y;  
}
```

而带一个类型参数的类模板可以这样定义：

```
template <class T> class ClassName{...};
```

在上面的定义中，类型参数 T 前面都使用了关键字 class，用来说明 T 是模板的类型参数。由于 C++中定义一个类也使用关键字 class，而这时的 class 与模板没有任何关系。为了避免混淆，C++引入了另一个关键字 typename，凡是在模板中说明一个标识符为类型参数，都可以使用 typename 来修饰。如上面的函数模板可改写成：

```
template <typename T> T Max(T x, T y){  
    return (x>y)?x:y;  
}
```

而带一个类型参数的类模板也可以定义成：

```
template <typename T> class ClassName{...};
```

在将标识符说明为模板的类型参数这一功能上，class 与 typename 是完全等价的。

typename 还有另一个用途：将某个带作用域的标识符显式地说明为类型。这在类型之间具有依赖关系的情况下是经常要用到的。考察下面的程序。

```
//// Program 6.9-1 ////  
#include <iostream>  
using namespace std;  
template <class T> class TypeClass{  
public:  
    typedef T Integer;  
};  
template <class T> class TestClass{  
public:  
    T::Integer i;  
};  
int main(){  
    typedef TypeClass<int> IntegerType;  
    typedef TestClass<IntegerType> UseInteger;  
    UseInteger u;  
    u.i=5;
```



```

cout << u.i << endl;
}

//// End of Program 6.9-1 /////

```

在类模板 TestClass 中, T::Integer 必须是一个类型, 变量 i 才有定义。此时, 类型 Integer 是依赖于类型 T 的。但是, 只有在类型 T 被实例化的时候, 才能知道 Integer 是否为一个类型。在某些类当中, 其成员 Integer 可能是一个成员变量或成员函数。C++ 编译器在默认情况下, 并不将依赖于别的类而存在的标识符认定为类型, 所以上面的程序无法通过编译。为了使编译器能够在模板实例化之前对类模板 TestClass 进行语法检查, 就必须有一种机制通知编译器: T::Integer 是一个类型, 而不是其他的成员。解决的办法是: 在必须为类型的标识符前面, 显式地使用 typename 关键字进行说明。

在 Program 6.9-1 中, 将语句 T::Integer i; 改写成 typename T::Integer i;, 程序就可以顺利通过编译并运行。

6.10 什么是仿函数

仿函数 (functor) 也称为函数对象 (function object), 就是把对象当函数使用。生成一个对象 a 后, 在类似 y=a(); 这样的语句中, 对象 a 无论从语法形式还是从实际功能来看都像是函数, 而与一般的对象不同。要实现仿函数的功能, 就必须在类中重载 operator() 操作符。这样, 生成该类的一个对象后, 这个对象的内部就封装了一个函数, 这个函数可通过对象的名字来调用。仿函数是继 C 语言的函数指针之后, 在 C++ 中引进的另一种传递和调用函数的方式。同函数指针相比较, 仿函数更为灵活。因为仿函数是一个对象, 而对象可以保留大量的状态, 而使用一般的函数指针却很难实现这一点。仿函数在 STL 中被大量使用, 是 STL 实现范型算法的一个有力工具。考察下面的程序。

```

//// Program 6.10-1 /////
#include <iostream>
using std::cout;
using std::endl;
template <class T> class Less{
public:
    bool operator()(const T &t1, const T &t2){
        return (t1<t2);
    }
};

template <class T> class Bigger{
public:
    bool operator()(const T &t1, const T &t2){
        return (t1>t2);
    }
};

```



```
template <class T, class T1> void PrintValue(const T &t1, const T &t2, T1 cmp){  
    if(cmp(t1,t2))  
        cout << t1 << endl;  
    else  
        cout << t2 << endl;  
}  
int main(){  
    PrintValue(1,7,Bigger<int>());  
    PrintValue(4.5,8.6,Less<double>());  
}  
/// End of Program 6.10-1 ///
```

在函数模板 PrintValue 中，对象 cmp 是当做一个比较函数使用的，它就是一个仿函数。在类模板 Bigger 和 Less 中，都重载了操作符 operator()，这样当类模板被实例化时，就可以利用函数对象实现对各种不同数据类型的对象的比较。程序的输出结果是：7 和 4.5。

6.11 什么是引用计数

引用计数（Reference Counting）从字面上看同两个要素相关：引用和计数。这里的引用同 C++ 当中的引用变量在概念上有稍许差别，它代表了访问对象的机制。引用一个对象就是指通过对象名、指针变量或引用变量来访问对象的成员。一个建立在堆（heap）上的对象，如果没有一个引用同它相关联，那么这个对象就会变成垃圾。

引用计数指的是这样一种技术：通过为某个数据对象建立多个引用，实现多个对象的数据共享，从而达到节省空间、提高内存管理效率、减少内存错误的目的。也就是说，使用引用计数，主要是为了解决如下两个问题：

(1) 内容相同的多个对象只建立一个内容实体。对象的内容，是指对象的关键数据成员的值。如果两个对象的内容相同，那么可以只创建一个内容实体，而建立该实体的两个引用。这样，既节省了内存空间，也不影响对对象内容的读取。

(2) 帮助实现对象空间的释放。在 C++ 中，内存管理一直是一个需要程序员小心对待的问题。用 delete（或 delete[]）释放对象空间时，可能出现两种错误：重复释放和提前释放。重复释放是指一个对象已经被释放了，却在之后又被释放一次。提前释放则是该对象还在被使用，却由于该对象的某个引用已完成使命而导致被释放。出现这些错误的原因是对象的所有权很难被跟踪，要确定一个对象的所有引用都完成了使命是一件非常困难的事情。引用计数则提供一个直观的方案：当一个内容实体的引用数达到 0 的时候，它便可以被释放了。一个内容实体的引用数达到 0，意味着它已经不再被需要，也再没有被访问到的可能性，所以这时必须将它释放掉。

要实现引用计数，必须考虑两个关键性问题：

- (1) 什么时候需要生成新的内容实体（或释放它），而什么时候只需要改变其引用数？
- (2) 内容实体的引用数存放在什么地方？

第一个问题实际上包含了相互关联的几个子问题：什么时候只需要改变内容实体的引用数（从而并不需要生成新的实体或释放其空间）？什么时候需要生成新的内容实体？什么时



候释放它？需要明确的一点是：不能通过搜索的方式确认两个对象的内容是否相同。如果将同类的所有对象全部找到，然后再一一同新产生的对象进行比较，不仅实现起来很困难，而且开销太大。在 C++ 中，可以考虑在对象的拷贝构造函数中使用引用计数，因为拷贝构造函数的本意是产生一个与原对象内容相同的“新”对象。还有对象间的赋值操作、以及有可能改变对象内容的操作（如 operator[] 操作符重载等），都要使用引用计数。

如果一个内容实体被多次引用（其引用数大于 1），而某一次操作有可能改变内容的值，这时就必须产生新的内容实体，而将原来的内容实体的引用数减 1。这一技术又被称为“写时复制（copy on write）”技术。当销毁一个对象时，首先要将其内容实体的引用数减 1，如果该内容实体的引用数已经到达 0，就可以将其空间释放掉。

另一个关键性的问题是，引用数应该放在什么地方。可不可以用全局变量或类的静态变量的方式来存放引用数呢？答案是否定的。对象内容的值千变万化，而全局变量或类的静态变量的个数一定是有限的，它们之间无法建立起对应关系。所以，应该将内容实体的引用数和引用实体本身“绑定”在一起，这样，内容实体本身就可以决定它的引用次数，而不必到外部的某个地方去寻找。

C++ 标准库中的 string 类，在不同的 C++ 实现中就采用了不同的方案。有些就采用了引用计数类实现 string 类对象的创建和销毁。下面的程序给出了利用引用计数思想来编写 string 类的一种可行的解决方案。有兴趣的读者可以自行给出自己的解决方案，并与本方案进行比较，找出它们的不同并分析其优劣。

```
//// Program 6.11-1 /////
#include <iostream>
using std::cout;
using std::endl;
class String{
    static int nCharArray;
    char* str;
    size_t _len;
public:
    String();
    String(char *p);
    String(const String &s);
    char& operator[](unsigned int idx);
    String& operator=(const String &s);
    ~String();
    const char* c_str(){
        return str+1;
    }
};
int String::nCharArray;
String::String(){
    str=NULL;
```

```

_len=0;
cout << "in constructor,nCharArray=" << nCharArray << endl;
}

String::String(char *p){
    _len = strlen(p);
    str = new char[_len+1+1];
    strcpy(str+1,p);
    str[0]=1;
    nCharArray++;
    cout << "in constructor,nCharArray=" << nCharArray << endl;
}

String::String(const String &s){
    str = s.str;
    _len = s._len;
    str[0]++;
    cout << "in constructor,nCharArray=" << nCharArray << endl;
}

char& String::operator[](unsigned int idx){
    if(idx<0 || idx>_len || str==NULL){
        static char nullchar = 0;
        return nullchar;
    }
    if(str[0]>1){
        char *buf=new char[_len+1+1];
        strcpy(buf+1,str+1);
        str[0]--;
        str=buf;
        str[0]=1;
        nCharArray++;
    }
    cout << "in operator[],nCharArray=" << nCharArray << endl;
    return str[1+idx];
}

String& String::operator=(const String &s){
    if(!str){
        _len=s._len;
        str=s.str;
        s.str[0]++;
    }
    else{

```



```

        str[0]--;
        if(!str[0]){
            delete[] str;
            nCharArray--;
        }
        _len=s._len;
        str=s.str;
        str[0]++;
    }

    cout << "in operator=,nCharArray=" << nCharArray << endl;
    return *this;
}

String::~String(){
    if(str){
        str[0]--;
        if(!str[0]){
            delete[] str;
            nCharArray--;
        }
    }

    cout << "in destructor,nCharArray=" << nCharArray << endl;
}

int main(){
    String s1;
    String s2="abc";
    String s3=s2;
    s3[0]='p';
    cout << s2.c_str() << endl;
    cout << s3.c_str() << endl;
    s1=s3;
    s2=s3;
    cout << s1.c_str() << endl;
    cout << s2.c_str() << endl;
}

/// End of Program 6.11-1 ///
程序的输出结果是:
in constructor,nCharArray=0
in constructor,nCharArray=1
in constructor,nCharArray=1

```

```

in operator[],nCharArray=2
abc
pbc
in operator=,nCharArray=2
in operator=,nCharArray=1
pbc
pbc
in destructor,nCharArray=1
in destructor,nCharArray=1
in destructor,nCharArray=0

```

以下是对该程序的编写及运行结果的几点说明：

①类的名称是 `String`，为的是与系统提供的 `string` 类相区别。其设计目标仍然是 C++ 标准库中的字符串类，只不过使用了引用计数的技术，并且只实现了该类的部分功能。

②`String` 类对象的内容，指的就是其成员 `str` 所指向的字符数组，该字符数组就是字符串对象的“内容实体”。

③`String::nCharArray` 是一个静态整型变量，它不是实现 `String` 类所必需的。程序中只是用它来统计生成字符串对象的“内容实体”的个数。`String::nCharArray` 的初始值为 0，每当新产生一个存放字符串的数组时，它的值就加 1；每当释放一个字符数组时，它的值就减 1。

④当生成对象 `s1` 时，由于并没有给出字符串的内容，所以 `String::nCharArray` 的值保持为 0。而当生成对象 `s2` 时，其字符串内容为“abc”，因此产生了一个字符数组（内容实体），`String::nCharArray` 的值变为 1。利用 `s2` 生成对象 `s3`，由于这两个字符串对象的内容相同，所以只需要改变内容实体的引用数，没有必要生成新的内容实体，所以 `String::nCharArray` 的值继续保持为 1。

⑤字符数组被引用的次数其实和该数组连成一体。当字符串对象的内容为长度为 `n` 的字符串时，实际上是用 `new` 操作申请了长度为 `n+2` 的一个字符数组，`str[0]` 代表该字符数组被引用的次数，字符串存放在 `str[1]~str[n]` 中，`str[n+1]` 为字符串结束标志 '`\0`'。在这种实现方案中，一个字符数组（字符串对象的内容实体）最多能够被引用 256 次。

⑥在使用引用计数时，字符串对象本身的个数并没有减少，真正减少的是字符串对象的“内容实体”的个数。在以上程序中，`s3` 对象刚产生时，`s3.str` 与 `s2.str` 的值相等，它们同时指向内容为“abc”的字符数组。也就是说，字符数组被引用了两次。

⑦对字符串对象实施 `operator[]` 操作时，由于返回的是字符数组的某个元素的引用，可以通过该引用改变字符串的内容，所以一旦对字符串对象实施了 `operator[]` 操作，且字符数组被引用的次数超过 1，就要另外产生一个字符数组（内容实体），将原来字符串的内容拷贝过来。

⑧将一个字符串对象赋值给另一个字符串对象时，会导致目的字符串对象（位于赋值符号左边）的内容实体的引用次数减 1。销毁一个对象时，在该对象的析构函数中也要将其内容实体的引用次数减 1。当一个内容实体的引用次数达到 0 时，就应该将其空间释放掉。

6.12 什么是 ADL

ADL 是 Argument Dependent Lookup 的缩写，即“参数相关查找”，也称作为 Koenig 查



找（以这种技术的发明人 Andrew Koenig 的名字命名，因此 ADL 有时也简称为 KL）。ADL 是指在编译器对无限定域的函数调用进行名字查找时所应用的一种查找规则。

由于 C++ 引入了类域、名称空间域等作用域范围，在调用一个函数时可以采用两种形式，如：

```
f(x, y, z); // unqualified
N::f(x, y, z); // qualified
```

第一个 f 是无限定域的函数调用，第二个调用将 f 限定在名字空间 N 的范围内，这是使用了完全限定名所产生的结果。

C++ 程序中的函数有三种作用域范围：类域（函数作为某个类的成员函数（静态或非静态））、名字空间域、全局域。ADL 的规则就是当编译器对无限定域的函数调用进行名字查找时，除了当前名字空间域以外，也会把函数参数类型所处的名字空间加入查找的范围。考察下面的程序。

```
/// Program 6.12-1 ///
#include <iostream>
using namespace std;
namespace Koenig{
    class OneClass{
        public:
            ostream& print(ostream& out) const{
                out<<member<<endl;
                return out;
            }
            OneClass(int mem = 5) : member(mem){}
        private:
            int member;
    };
    inline ostream& operator<<(ostream& out, const OneClass& one){
        return one.print(out);
    }
}
int main(){
    Koenig::OneClass one(10);
    cout << one;
    return 0;
}
/// End of Program 6.12-1 ///
```

在此程序中，`ostream& operator<<(ostream& out, const OneClass& one)` 的定义是处于名字空间 Koenig，为什么编译器在解析 main 函数（全局域）里面的 `operator<<` 调用时，它能够正确定位到 Koenig 名字空间里面的 `operator<<`？这是因为根据 Koenig 查找规则，编译器需要把参数类型 OneClass 所在的名字空间 Koenig 也加入对 `operator<<` 调用的名字查找范围中。



如果没有 Koenig 查找规则，我们就无法直接写 `cout<<one;`，而是需要写类似 `Koenig::operator<<(std::cout, one);` 这样的代码（使用完全限定名）。这种形式的代码很不直观，书写起来也不方便。

更重要的是，如果写的是模板代码，在模板参数还没有实例化之前，根本就不知道参数所处的名字空间，比如：

```
template<typename T> void print(const T& value){  
    std::cout<<value;  
}
```

很显然，在模板被实例化之前，该函数模板代码根本无法确认 `T` 是来自哪个名字空间，也就无法显式指明 `operator<<` 所处的作用域。

在 ADL 为我们带来便利的同时，也会引发一些新的问题。对 Koenig 查找规则的一个异议是，由于 Koenig 查找规则的存在，处于某个名字空间的函数调用的重载决议会受到另外一个名字空间的自由函数所影响，仅仅是由于它使用了另外一个名字空间的类型作为参数。在这样的规则下，名字空间看起来不像我们一般所想象的那样是完全封闭和独立的。

当 Koenig 查找规则和 C++ 原来的 Ordinal Lookup (OL, 顺序查找规则) 混合在一起的时候，它们之间的组合会产生许多新的复杂的状况。所谓顺序查找，就是从函数调用所处的域开始（如果函数调用处于一个成员函数中，初始域就是类域，如果处于自由函数中，初始域就是名字空间域或者全局域），依次由内到外到各个域进行名字查找，如果在某个域找到该名字的函数，就停止查找，将所有找到的重载函数进行重载决议，如果没有合适的候选者或者有多个合适的候选者而导致歧义，编译器则报错。如果一直找到全局域也没有找到任何该名字函数，编译器也报错。例如：

```
//// Program 6.12-2 ////  
#include <iostream>  
using namespace std;  
void func(){  
    cout << "function in global namespace" << endl;  
}  
namespace NS1{  
    void func(){  
        cout << "function in namespace1" << endl;  
    }  
    namespace NS2{  
        /*      void func(char *s){  
            cout << s << endl;  
        }*/  
        class KoenigLookup{  
            public:  
                void koenigLookup(){  
                    func();  
                }  
        };  
    }  
}
```

```

    };
}

}

int main(){
    NS1::NS2::KoenigLookup kl;
    kl.koenigLookup();
}

/// End of Program 6.12-2 ///

```

类 KoenigLookup 的成员函数 koenigLookup() 调用了函数 func(), func 的查找顺序依次是类 KoenigLookup, 名字空间 NS1::NS2, 名字空间 NS1, 最后是全局域。由于在名字空间 NS1 下找到了合适的重载函数版本, 所以程序的输出结果是:

function in namespace1

如果在上面的程序中, 将注释掉的函数 void func(char *s) 解除注释, 则会导致一个编译错误。原因是顺序查找在找到名为 func 的函数之后, 就不会向更外围的作用域查找重载函数。如果在找到的重载函数中没有合适的版本, 就会报告编译错误。实际上, 在 3.8 节曾经介绍过, 不同作用域内的同名函数只能形成隐藏, 不能形成重载。只不过, 当函数的参数用到了其他名称空间中定义的类型时, 这种重载决议的形式可能会受到影响。

应该说, OL 是名字查找的主要规则。只有当引入名称空间、模板等机制之后, 在 OL 应用的某些阶段中 KL 才起作用, 并将其作用附加在 OL 之上。下面分几种情况讨论顺序查找规则和 Koenig 查找规则相组合的结果。

(1) 类的成员函数与类外的自由函数同名

必须明确的是, 类域比名字空间域(包括全局域)有更高的优先级, KL 规则的作用范围是名字空间域里的自由函数, 当 OL 应用于类域的成员函数的时候, KL 是不起作用的。当进行名字查找的时候, 成员函数绝对不会跟非成员函数一起进行重载决议。如下面的例子。

```

/// Program 6.12-3 ///
#include <iostream>
using namespace std;
namespace KLSpace2{
    class KLArg2;
}
namespace KLSpace1{
    class KLArg1{
    };
    //重载函数1
    void KoenigLookupMethod(KLSpace1::KLArg1&, KLSpace2::KLArg2&){
        cout<<"Namespace KLSpace1::KoenigLookupMethod";
    }
}
namespace KLSpace2{

```

```

class KLArg2{
};

//重载函数2
void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
    cout<<"Namespace KLSpace2::KoenigLookupMethod";
}

}

//全局重载函数
void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
    cout<<"Global KoenigLookupMethod";
}

namespace KL{
    //重载函数3
    void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
        cout<<"Namespace KL::KoenigLookupMethod";
    }

    namespace KL_Inside{
        //重载函数4
        void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
            cout<<"Namespace KL::KL_Inside::KoenigLookupMethod";
        }
    }

    class KoenigLookup{
        public:
            //调用类KoenigLookup中的函数，该函数是一个非静态成员函数
            void koenigLookup(){
                KLSpace1::KLArg1 klArg1;
                KLSpace2::KLArg2 klArg2;
                KoenigLookupMethod(klArg1,klArg2);
            }

        private:
            //类的非静态成员函数
            void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
                cout<<"Non-Static Member KL::KL_Inside::KoenigLookup::";
                cout<<"KoenigLookupMethod";
            }
    };
}

int main(){
    //调用的是同一个类的非静态成员函数
}

```

```

KL::KL_Inside::KoenigLookup kl;
kl.koenigLookup();
return 0;
}

```

//// End of Program 6.12-3 ////

程序的输出结果是：

Non-Static Member KL::KL_Inside::KoenigLookup::KoenigLookupMethod

在函数调用 `kl.koenigLookup();` 中，类 `KoenigLookup` 的成员函数 `KoenigLookupMethod()` 被调用。整个过程中 `KL` 规则并没有起作用，因为 `OL` 开始作用于类域，找到符合名字的成员函数之后就停止了查找，经过重载决议后得到最后调用的版本，类域中 `KL` 是不起作用的。把上面程序的调用函数和重载函数换作类的静态函数，即做如下改造：

```

class KoenigLookup{
public:
    //调用类KoenigLookup中的函数，该函数是一个静态成员函数
    static void koenigLookup(){
        KLSpace1::KLArg1 klArg1;
        KLSpace2::KLArg2 klArg2;
        KoenigLookupMethod(klArg1,klArg2);
    }
private:
    //类的静态成员函数
    static void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
        cout<<"Static Member KL::KL_Inside::KoenigLookup::";
        cout<<"KoenigLookupMethod";
    }
};

```

程序的输出结果是：

Static Member KL::KL_Inside::KoenigLookup::KoenigLookupMethod

如果类的成员函数 `KoenigLookupMethod()` 的定义并不符合函数 `koenigLookup()` 中的调用形式，则编译器直接报告错误，并不会到类域外去查找其他重载函数。这就说明了，如果类的成员函数与自由函数同名，不管函数的参数是否用到了其他名称空间中的类型，它们之间都不会进行重载决议。也就是说，在这种情况下，`KL` 规则不起作用。

(2) 调用类外的自由函数

在某个类的成员函数中，如果被调用函数不是在本类中定义的，那么它就是一个外部函数。这时，编译器会根据顺序查找规则，从类所在的作用域向外一直到全局作用域查找重载函数，直到确定重载函数的集合时停止查找。这时，除了 `OL` 规则发生作用外，`KL` 规则也起作用。见下面的例子。

//// Program 6.12-4 ////

```
#include <iostream>
using namespace std;
```

```

namespace KLSpace2{
    class KLArg2;
}

namespace KLSpace1{
    class KLArg1{
    };
    //重载函数
    void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
        cout<<"Namespace KLSpace1::KoenigLookupMethod";
    }
}

namespace KLSpace2{
    class KLArg2{
    };
    //重载函数
    void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
        cout<<"Namespace KLSpace2::KoenigLookupMethod";
    }
}

//全局重载函数
void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
    cout<<"Global KoenigLookupMethod";
}

namespace KL{
    //重载函数
    void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
        cout<<"Namespace KL::KoenigLookupMethod";
    }

    namespace KL_Inside{
        //重载函数
        void KoenigLookupMethod(KLSpace1::KLArg1&,KLSpace2::KLArg2&){
            cout<<"Namespace KL::KL_Inside::KoenigLookupMethod";
        }
    }

    class KoenigLookup{
        public:
            //被调用函数在类KoenigLookup中没有定义，必须到类域之外去查找
            static void staticKoenigLookup(){
                KLSpace1::KLArg1 klArg1;
                KLSpace2::KLArg2 klArg2;
                KoenigLookupMethod(klArg1, klArg2);
            }
    }
}

```

```

    }
};

}

int main(){
    //被调用函数KoenigLookup()是自由函数
    KL::KL_Inside::KoenigLookup::staticKoenigLookup();
    return 0;
}

/// End of Program 6.12-4 ///

```

这个程序在编译时会报错。在 VS 2005 下，出错信息是：“KL::KL_Inside::KoenigLookupMethod”。对重载函数的调用不明确，可能是“void KL::KL_Inside::KoenigLookupMethod(KLSpace1::KLArg1 &, KLSpace2::KLArg2 &)”，或“void KLSpace2:: KoenigLookupMethod(KLSpace1::KLArg1 &, KLSpace2::KLArg2 &)”[使用参数相关的查找找到]，或“void KLSpace1::KoenigLookupMethod(KLSpace1::KLArg1 &, KLSpace2::KLArg2 &)”[使用参数相关的查找找到]。

按照顺序查找规则，编译器在找到 void KL::KL_Inside:: KoenigLookupMethod()之后就应该停止查找，同时，根据 KL 规则，还应该在第一个参数的名字空间域 KLSpace1 和第二个参数的名字空间域 KLSpace2 中进行查找，所以重载函数集合中包含三个候选函数，显然无法从中挑选最合适的调用版本，编译器因此报错。

值得注意的是，全局域和名字空间 KL 的重载函数 KoenigLookupMethod 并没有被编译器列入导致歧义的版本，因为这两个域此时根本不在查找的范围内。

假设在名字空间 KL_Inside 里面找不到，那么编译器会继续到外层名字空间域 KL 中进行查找，如果还是找不到，就会到全局域查找。编译器会将根据顺序查找规则找到的候选函数，同根据 KL 规则从参数相关域 KLSpace1 和 KLSpace2 中查找到的更多候选函数一起参与重载决议。

如果根据 OL 规则和 KL 规则都无法找到候选函数，编译器就会报告没有 KoenigLookupMethod 这个标识符。

如果函数 KoenigLookupMethod 不是在类的成员函数里被调用，那么除了不会在类域中进行查找之外，其他的查找过程与上述情况相同。

需要注意的是，与参数相关的名字空间域，有时会多于一个。这样，在进行 KL 查找时，同样可能引起麻烦。请看下面的例子。

```

/// Program 6.12-5 ///
#include <iostream>
using namespace std;
namespace NS2{
    class Derived;
}
namespace NS1{
    class Base{};

```

```
void func(NS2::Derived &obj){  
    cout << "in namespace1" << endl;  
}  
}  
namespace NS2{  
    class Derived:public NS1::Base{};  
    void func(Derived &obj){  
        cout << "in namespace2" << endl;  
    }  
}  
int main(){  
    NS2::Derived obj;  
    func(obj);  
}
```

/// Program 6.12-5 ///

在函数 main() 中调用函数 func() 时，与实参相关的名字空间域有两个，一个是类 Derived 所在的名字空间 NS2，另一个是其基类 Base 所在的名字空间域 NS1。而恰好在这两个类中都定义有函数 void func(NS2::Derived &obj)，导致编译器无法形成有效的重载决议，从而报错。在实践中，由于一个参数可能同多个名字空间域相关，从而导致编译器进行大量的无效查找，所以应该避免这种情况的发生。

对于 KL 查找规则的引入可能造成的副作用，程序员应该能够找到问题发生的原因，并找到合适的解决办法。看下面的程序。

```
/// Program 6.12-6 ///  
#include <iostream>  
using namespace std;  
namespace boost{  
    class noncopyable{  
    protected:  
        noncopyable() {}  
        ~noncopyable() {}  
    private: // emphasize the following members are private  
        noncopyable( const noncopyable& );  
        const noncopyable& operator=( const noncopyable& );  
    };  
    template<typename T>  
    void foobar(const T&){  
        cout << __FUNCTION__ << ":" << __LINE__ << endl;  
    }  
}  
namespace base{  
    class Base : private boost::noncopyable{
```



```

};

void foobar(const Base&){
    cout<<_FUNCTION_<<" : "<<_LINE_<<endl;
}

namespace derived{
    class Derived : public base::Base{
    };
    void foobar(const Derived&){
        cout<<_FUNCTION_<<" : "<<_LINE_<<endl;
    }
}
int main(){
    derived::Derived d;
    foobar(d);
    return 0;
}

/// End of Program 6.12-6 ///

```

在调用函数 foobar() 时，由于全局域并没有名为 foobar 的函数存在，所以编译器在参数类型所在的名称空间 derived 中找到重载函数 void foobar(const Derived&)，同时，由于类 Derived 的基类是 base::Base，所以名称空间 base 中的函数 void foobar(const Base&) 也成为候选函数。由于名称空间 derived 中的重载函数 void foobar(const Derived&) 更加匹配，所以它成为最终被调用的函数。程序的输出结果是：

derived::foobar : 32

如果将名称空间 derived 中的函数 foobar() 注释掉，输出的结果又会怎样呢？也许，程序员的本意是让名称空间 base 下的函数 void foobar(const Base&) 来处理所有的 Base 及其派生类的对象，那么在上述程序中能达到目的吗？

通过重新编译运行，可以发现，在将名称空间 derived 中的函数 foobar() 注释掉之后，程序的输出结果是：boost::foobar : 15。也就是说，名称空间 base 中的函数 void foobar(const Base&) 并没有按照期望的那样被调用。问题的根源还在于 KL 查找规则的副作用。编译器在查找 foobar() 的候选函数时，先找到了名称空间 base 下的函数 void foobar(const Base&)，由于类 Base 的基类 noncopyable 在名称空间 boost 当中，所以，编译器又将 boost 当中的函数模板实例化，生成模板函数 void foobar<derived::Derived>(const derived::Derived&)，这个函数比名称空间 base 中的函数更加匹配，从而成为最终被调用的函数。

为了避免不必要的名字查找，也为了最有效地避免 KL 副作用，可以将某个类型，特别是有很多派生类的基类，定义在一个封闭的名称空间中，在这个名称空间中除了这个类之外，再没有其他可能作为候选重载函数的自由函数或模板存在。上面的程序可做如下修改。

```

/// Program 6.12-7 ///
#include <iostream>
using namespace std;
namespace boost{

```

```

namespace noncopyable_{
    class noncopyable{
        protected:
            noncopyable() {}
            ~noncopyable(){}
        private: // emphasize the following members are private
            noncopyable( const noncopyable& );
            const noncopyable& operator=( const noncopyable& );
    };
}

typedef noncopyable_::noncopyable noncopyable;
template<typename T>
void foobar(const T&){
    cout<<_FUNCTION_<<" : "<<_LINE_<<endl;
}

namespace base{
    class Base : private boost::noncopyable{
    };
    void foobar(const Base&){
        cout<<_FUNCTION_<<" : "<<_LINE_<<endl;
    }
}

namespace derived{
    class Derived : public base::Base{
    };
}

int main(){
    derived::Derived d;
    foobar(d);
    return 0;
}

/// End of Program 6.12-7 ///

```

由于名称空间 noncopyable_里面只有一个类 noncopyable，而没有其他的任何自由函数或模板存在，所以，不管将类 noncopyable 移到哪个名称空间中去，只要还是将名称空间 noncopyable_作为一个整体移动，就不会因为 KL 规则带来任何的副作用。在上面的程序中，为了方便类 noncopyable 的使用，加上了一个类型定义 `typedef noncopyable_::noncopyable noncopyable;`，这样在名称空间 boost 中就可以直接使用 noncopyable 这个类。这一次，程序的输出结果是：

base::foobar : 26



第7章 | 内 存 管 理

7.1 C++程序的内存布局

一个C/C++源程序经过编译之后，其应用程序使用的内存可划分为以下几个部分：

①代码区。在低级语言的层次上，程序是以函数的集合的方式存在的。代码区存放函数体的二进制代码。

②栈区（stack）。是程序中用来存放函数的参数值、局部变量、临时变量的值的地方。由编译器生成的代码自动完成内存的分配与释放，其操作方式类似于数据结构中的栈。

③堆区（heap）。供程序员动态申请和释放空间的内存区。这个区域由程序员自己负责维护。堆区内存管理是C/C++程序设计中容易出错的一项工作。

④全局/静态区。是用来存放全局变量和静态变量的内存区。程序在创建这块存储区时会自动将所有字节清0。所以，如果没有显式地为全局（静态）变量赋初始值，那么它的初始值就是0。程序结束后由系统释放。

⑤常量区。用来存放字符串常量的内存区。程序运行结束后由系统释放。

以下是一个示例程序。程序中的每个变量、字符串常量和动态申请的空间所处的位置都由注释做了说明。

```
/// Program 7.1-1 ///
#include <iostream>
using namespace std;
int a = 0;           //显式初始化区的全局变量
char *p1;           //隐式初始化区的全局变量
int main(){
    int b;           //局部变量，分配在栈上
    char s[] = "abc"; //局部字符数组，数组元素分配在栈上
    char *p2;         //局部字符指针变量，分配在栈上
    //p3为局部字符指针，分配在栈上；字符串常量"123456"存放在常量区
    char *p3 = "123456";
    static int c=0;   //局部静态变量，分配在全局/静态区
    p1 = new char[10];
    p2 = new char[20]; //动态申请的10和20字节的空间就在堆区
    //"123456"放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方
    strcpy(p1, "123456");
}
```



/// End of Program 7.1-1 ///

下面就 C++ 程序所使用的几个重要内存区域做进一步的解释。

(1) 代码区

可执行文件加载之后，就存放在进程的代码区。这部分区域是只读的，如果试图直接修改内存中的程序代码，将导致运行时错误。一般来说，程序的代码区存放的是程序的可执行代码。在某些特殊情况下，一些重要的数据也可以放入代码区，以防止误修改。

在 1.2 节曾经介绍过，程序中的文字常量，经过编译之后直接写入了代码区。下面的例子通过打印函数的机器码，找到了文字常量存放的具体地址。

```
/// Program 7.1-2 ///
#include <iostream>
#include <iomanip>
using namespace std;
void func(int i){
    if(i>5)                                //通过输出机器码查看文字常量5存放的具体位置
        cout << "more" << endl;
    else
        cout << "less" << endl;
}
void showcode(unsigned char *pi){
    unsigned char *keep=pi;
    int tmp=*pi;
    unsigned long exp;
    if(tmp==233){                         //跳转指令，后面紧跟4字节的跳转偏移量
        unsigned long addr=0L;
        exp=1;
        for(int i=0;i<4;i++){
            pi++;
            tmp = int(*pi);
            addr += tmp*exp;
            exp *= 256;
        }
        addr += int(keep)+5;      //当前指令的地址，加上跳转偏移量，再加5，就是目的地址
        showcode((unsigned char*)addr);
    }
    else{
        for(int i=0;i<4;i++){
            cout << setfill('0');
            cout << "0X" << setw(8) << hex << int(pi) << " : ";
            for(int j=0;j<10;j++){
                tmp = *(pi++);
            }
        }
    }
}
```

```

        cout << setw(2) << hex << tmp << " ";
    }
    cout << endl;
}
}

int main(){
    void (*pf1)(int);
    pf1=func;
    unsigned char *pi = (unsigned char*)pf1;
    showcode(pi);
}

/// End of Program 7.1-2 ///

```

在此程序中，通过函数指针 pf1 获取了函数 func() 的入口地址，然后，就可以打印函数 func() 在内存中的机器码。由于由函数指针 pf1 所指明的地址可能只是一条跳转语句，所以要经过适当的跟踪，找到函数 func() 真正开始执行的第一条代码，然后开始打印。函数 showcode() 输出了指定地址的头 40 个字节的内容。程序执行结果如下：

```

0X004114e0 : 55 8b ec 81 ec c0 00 00 00 53
0X004114ea : 56 57 8d bd 40 ff ff ff b9 30
0X004114f4 : 00 00 00 b8 cc cc cc cc f3 ab
0X004114fe : 83 7d 08 05 7e 2d 8b f4 a1 58

```

其中，机器码 83 7d 08 05 7e 2d 所对应的汇编语言代码是：

```

cmp  DWORD PTR _i$[ebp], 5
jle  SHORT $LN2@func

```

也就是函数 func() 中 if(i>5) 这条语句所对应的比较和跳转指令。地址 0X00411501 处的字节的值 05 就是源程序中的文字常量。你可以试着去修改这个常量的值，但会导致程序运行错误。也就是说，代码区是不允许动态修改的。

(2) 栈区

栈空间是由编译器产生的代码来管理的，不用程序员操心堆上的空间的申请和释放的问题。在 Windows 下，栈是一块由高地址向低地址方向生长的空间，栈指针 esp 总是指向最后一个入栈的字节。随着新的元素入栈，esp 的值将减小。在程序运行时，只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常，提示栈溢出。

除非特别指定，C++ 程序运行时，栈的最大容量是系统预先规定好的，在 Windows 下栈的大小是 1M。因此，能从栈获得的空间较小。以下是一个测试 Windows 下栈空间大小的程序。

```

/// Program 7.1-3 ///
#include <iostream>
#include <iomanip>
using namespace std;
int *addr;

```



```
int count;
void func(int i){
    cout << count << ":" << (int)addr-int(&i) << endl;
    count++;
    addr = &i;
    func(i);
}

int main(){
    int i;
    addr = &i;
    func(2);
}
/// End of Program 7.1-3 ///
```

该程序启动了一个没有出口的递归调用序列，每向下递归一层，都要在栈上申请空间，每次申请空间的大小可以从参变量 i 的相邻两次地址变化得到。程序一直运行到堆栈溢出时退出。程序的运行结果是：

```
0 : 212
1 : 216
2 : 216
...
4771 : 216
```

所以，栈的大小约等于 $4771 \times 216 / 1024 = 1006\text{K}$ 。也就是说，栈的大小约为 1M。

C++程序的函数调用是通过栈来实现的。函数返回地址、参变量、局部变量、临时变量、函数的返回值等都是分配在栈上，具体的工作细节请参阅 3.2、3.3 和 3.12 节。由于在栈上分配空间由编译器管理，而且栈的结构相对简单，所以栈空间的使用效率很高。但栈上变量的生命周期相对较短，栈空间有限，这时堆的作用就凸显出来。

(3) 堆区

堆的具体实现和管理方式与操作系统相关，不同操作系统的实现细节是不同的。一般的原理是：在操作系统中设置一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。而且，一般会在这块内存空间中的首地址处记录本次分配的大小，以便代码中的 delete 语句能够正确地释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

由于堆内存空间是由链表负责维护的，所以是不连续的内存区域。堆是向高地址扩展的数据结构，链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

在 C 语言当中，动态内存的申请和释放使用的是 malloc() 和 free() 这两个函数。而 C++ 则是使用 new 和 delete 操作。在对象的分配和释放上，new 和 delete 会自动调用对象的构造函数和析构函数，而 malloc 和 free 则不会。所以，在 C++ 中提倡使用 new 和 delete。堆的弱



点是速度相对较慢，而且容易产生内存碎片；优点是使用方便，便于构造复杂的数据结构。

(4) 全局/静态区

全局/静态区中存放的是全局变量或静态变量，这些变量的生命期与整个程序的运行期相同。因此，从某种意义上说，全局/静态变量是一种很便于访问的变量。但同时，由于全局变量会增加模块之间的耦合性，不利于降低程序的复杂度，所以，C++程序要谨慎使用全局变量。

全局/静态中的变量一定具有确定的初始值，要么在程序中显式指定，要么就保持为0（或相当于0的值，如false、NULL等）。在C++中，允许用别的变量或函数来为全局变量赋初值，这时，这些变量的初始化工作实际上分为两步进行：先是被初始化为0，然后再利用别的变量的值或利用函数的返回值为它赋值。考察下面的程序。

```
/// Program 7.1-4 ///
#include <iostream>
using namespace std;
extern int getVal();
extern int j;
int i=j;
int j=getVal();
int getVal(){
    return i+j+5;
}
int main(){
    cout << i << endl;
    cout << j << endl;
}
/// End of Program 7.1-4 ///
```

程序的输出结果是：

0

5

要正确解释该程序的行为，一定要弄清全局变量i和j被初始化的过程。它们被初始化的顺序是这样的：首先，变量i和j都被初始化为0；然后，将j的值（0）赋值给i，所以i的值仍然保持为0；最后，执行函数getVal()，在执行这个函数的时候，i和j的值都是0，所以它返回5，该返回值被传递给j，所以j的最终值为5。

(5) 常量区

常量区既不是存放文字常量（存放在代码区），也不是存放常变量（存放在栈区或堆区）的存储区。常量区里存放的是字符串常量，以及一些在程序运行中不能修改的重要数据（如类的虚函数表等）。常量区的性质和全局/静态区很接近，只不过这一区域中的数据是只读的，如果试图修改，将导致运行时错误（参见Program 5.3-7），而且，常量区的数据的存放和使用是由编译器负责完成的，程序员一般没有必要关心常量区的实现细节。

7.2 理解 new 操作的实现过程

new 操作是 C++ 用来动态申请空间的操作，它返回分配空间的首地址。编译器在处理 new 操作时，实际上是将它分两步实现的：第一步是分配空间，第二步是调用相关的构造函数。因此，要注意区别两个不同的概念：new 操作符 (new operator) 和操作符 new (operator new)。

当 C++ 程序中出现这样的代码：

```
string *ps = new string("A new object");
```

这是一个使用 new 操作创建新的对象的语句，这里的 new 是 new 操作符。这个操作符就像 sizeof 一样是语言内置的，程序员不能改变它的含义，它的功能总是固定不变的。它要完成的功能分成两部分。第一部分是分配足够的内存以便容纳所需类型的对象。第二部分是调用构造函数初始化内存中的对象。new 操作符总是做这两件事情，C++ 语言不允许以任何方式改变它的行为。

程序员所能改变的是如何为对象分配内存。new 操作符调用一个函数来完成必需的内存分配，因此能够重写或重载这个函数来改变分配内存的方式。new 操作符为分配内存所调用函数的名字是 operator new。函数 operator new 通常这样声明：

```
void * operator new(size_t size);
```

返回值类型是 void*，因为这个函数返回一个未经处理 (raw) 的指针，代表一段未初始化内存的首地址。operator new 的缺省动作是系统定义好的，没有特别的需要是不必改动的。但是，程序员确实可以重载 operator new，甚至做一些较为特殊或“怪异”的工作，如在返回一个指针之前就初始化内存以存储一些数值等。函数 operator new 的参数 size_t 确定分配多少内存。程序员能增加额外的参数重载函数 operator new，但是第一个参数类型必须是 size_t。

一般不直接调用 operator new，但这么做却是合法的，可以像调用其他函数一样调用它：

```
void *rawMemory = operator new(sizeof(string));
```

操作符 operator new 将返回一个指针，指向一块足够容纳一个 string 类型对象的内存。

就像 malloc 一样，operator new 的职责只是分配内存。它并不知道构造函数的存在。operator new 所做的只是内存分配，把 operator new 返回的未经处理的指针传递给一个对象是 new 操作符的工作。

当编译器遇见这样的语句：

```
string *ps = new string("Memory Management");
```

它生成的代码或多或少与下面的代码相似：

```
void *memory = operator new(sizeof(string)); //得到未经处理的内存  
call string::string("Memory Management") on *memory; //为 string 对象初始化  
string *ps = static_cast<string*>(memory); //ps 指针指向新的对象
```

第二步包含了构造函数的调用，这是编译器解释 new 操作符时所做的工作之一。程序员无法直接调用构造函数来初始化对象，C++ 没有提供这样的机制。如果想建立一个堆对象，就必须使用 new 操作符。

要改变 new 操作分配空间的默认方式，就必须重载 operator new() 函数。有两种形式的 operator new()。一种是全局 operator new 函数，另一种作为类的成员函数存在。如果一个类没有定义 operator new 成员函数，那么在动态分配该类对象的空间时调用的就是全局 operator



new 函数。下面是一个重载 operator new 的例子。

```
/// Program 7.2-1 ///
#include <iostream>
using namespace std;
void *operator new(size_t size){
    cout << "global operator new" << endl;
    return malloc(size);
}
class SomeClass{
public:
    void *operator new(size_t size){
        cout << "local operator new" << endl;
        return malloc(size);
    }
};
int main(){
    string *ps;
    SomeClass *pc;
    ps = new string("A new string");
    delete ps;
    pc = new SomeClass;
    delete pc;
}
/// End of Program 7.2-1 ///
```

为了方便说明，在 operator new 中直接调用了 malloc 来分配空间。这并非出自实际编程的需要，而仅仅是为了演示如何实现重载 operator new 函数。程序的输出结果是：

```
global operator new
local operator new
```

为了避免内存泄漏，每个动态内存分配必须与一个等同相反的 deallocation 对应。函数 operator delete 与 delete 操作符的关系与 operator new 与 new 操作符的关系一样。delete 操作符的典型用法如下所示：

```
string *ps;...
delete ps; //使用 delete 操作符
```

编译器会生成代码来析构对象并释放对象占有的内存。operator delete 用来释放内存，它被这样声明：

```
void operator delete(void *memoryToBeDeallocated);
```

因此，delete ps; 导致编译器生成类似于这样的代码：

```
ps->~string(); //call the object's dtor
operator delete(ps); // deallocate the memory the object occupied
```

从另一方面说，如果只想处理未被初始化的内存，则应该绕过 new 和 delete 操作符，而



调用 operator new 获得内存和 operator delete 释放内存给系统。例如：

```
void *buffer = operator new(50 * sizeof(char)); // 分配足够的内存以容纳 50 个 char  
//没有调用构造函数  
operator delete(buffer); // 释放内存没有调用析构函数
```

这与在 C 中调用 malloc 和 free 等同。

如果程序员不加限制地申请动态空间，则可能“耗尽”堆内存资源。这时，new 操作可能失败。在 C 语言中，如果内存申请失败，malloc() 函数会返回一个空指针 (NULL)。在 C++ 中，如果 new 操作申请空间失败，编译器可能会采取如下的两种做法之一：返回一个空指针，或者抛出一个 std::bad_alloc 类型的异常。在早期的 C++ 语言中，采用的就是前一种方式，与 C 语言中的申请空间失败时的做法相同。后来，C++ 编译器采用的则是抛出异常的方式。以下的一个小程序演示了申请空间失败时应如何发现并做出适当的处理。

```
/// Program 7.2-2 ////  
#include <iostream>  
#include <iomanip>  
using namespace std;  
struct Large{  
    int arr[20000000];  
};  
void Bad_Allocation(){  
    int i;  
    Large *p;  
    for(i=1;i<1000000;i++){  
        cout << i << endl;  
        p = new Large;  
        if(!p){  
            cout << "break";  
            break;  
        }  
    }  
}  
int main(){  
    try{  
        Bad_Allocation();  
    }  
    catch(bad_alloc){  
        cout << "No enough memory!" << endl;  
    }  
}/// End of Program 7.2-2 ///
```

此程序由于不断地申请大容量的动态空间，导致整个计算机系统运行速度越来越慢，在



Windows 操作系统下还有可能提示虚拟内存不够。最后在“耗尽”堆空间的时候，发生 std::bad_alloc 异常。所以，程序的运行结果看起来应该是这样的：

```

1
2
3
...
No enough memory!

```

以上程序也演示了发生大量的内存泄漏可能给系统造成危害：可利用的堆空间越来越少，new 操作的效率越来越低，系统的运行速度越来越慢，等等。所以，在动态空间不再使用之后立即通过 delete 操作归还给系统，是一个良好的编程习惯。

7.3 怎样禁止在堆（或栈）上创建对象

有时，程序员可能不希望对象在堆上产生。比如，某类对象由于封装了其他的资源，因此它的析构函数的调用显得非常重要。如果该对象只能在栈中产生，这样就能在异常的情况下自动释放封装的资源。而如果对象被允许在堆上产生，那么对象的析构函数不能被正常调用的可能性会增大。

怎样禁止产生堆对象呢？这要从动态对象的产生过程入手。我们知道，产生堆对象的惟一方法是使用 new 操作，因此应该禁止使用 new 操作产生某类对象。由于 new 操作的执行实际上分为两步（第一步是调用 operator new，第二步是调用构造函数），而 operator new 是可以重载的，所以，可以使 new operator 变为 private，这样就禁止了 new 操作的进行。为了对称，最好将 operator delete 也重载为 private。见下面的例子。

```

/// Program 7.3-1 ///
#include <stdlib.h> //需要在C中内存分配函数
class Resource{}; //代表需要被封装的资源类
class NonHeapObject{
private:
    Resource* ptr;//指向被封装的资源
    //其他数据成员
    void* operator new(size_t size){ //非严格实现，仅作示意之用
        return malloc(size);
    }
    void operator delete(void* pp){ //非严格实现，仅作示意之用
        free(pp);
    }
public:
    NonHeapObject(){
        //此处可以获得需要封装的资源，并让ptr指针指向该资源
        ptr = new Resource();
    }
}

```

```

~NonHeapObject(){
    delete ptr; //释放封装的资源
}
};

//NonHeapObject现在就是一个禁止产生堆对象的类了
int main(){
    NonHeapObject* np = new NonHeapObject(); //编译错误!
    delete np;
}

//// End of Program 7.3-1 /////

```

上面代码会产生编译错误，出错信息是：无法访问 private 成员(在“NonHeapObject”类中声明)。

当然，这种禁止产生堆对象的办法只是“提醒”程序员不要在堆上产生某类的对象。如果程序员想“强行”在堆上创建对象，一样可以通过别的途径实现。比如，先在堆上申请一块空间，然后利用定位放置 new (placement new) 操作产生对象，或者，“手工”创建一个堆对象等等。C++语言功能强大到能做成程序员想做的一切事情。只不过，不滥用 C++的强大功能，用软件工程的原则进行程序开发，是广大 C++程序员必须面对的一个重要问题。

相对于禁止在堆上创建对象，要实现禁止在栈上创建对象却要麻烦得多。因为栈空间的分配是不受程序员控制的，如果想要在编译阶段就禁止程序员创建栈对象，只能在类的构造函数和析构函数上做文章。如果将类的构造函数设置成私有，一方面构造函数有多个（多种重载形式，包括拷贝构造函数），另一方面也会影响 new 操作，所以，可将类的析构函数设置成私有（private）。考察下面的程序。

```

//// Program 7.3-2 /////
#include <iostream>
using namespace std;
class NonStack{
private:
    int num;
    ~NonStack(){}
public:
    NonStack(){num=1;}
    void show(){
        cout << num << endl;
    }
    void Delete(){
        delete this;
    }
};
int main(){
    // NonStack obj;

```

```

NonStack *p= new NonStack;
p->show();
p->Delete(); //代替使用 delete p;
}

/// End of Program 7.3-2 ///

```

此程序能通过编译，能正常运行并输出：1。如果把 main()函数中创建栈对象的语句 NonStack obj; 从注释中释放出来，将导致编译错误。因为对象 obj 在生命期结束时会调用类 NonStack 的析构函数，而此析构函数是私有的。不过，这种禁止产生栈对象的方案是不完美的。首先，它在禁止产生栈对象的同时，连全局对象也一同禁止了。另外，由于类的析构函数是私有的，所以在销毁堆上的对象的时候，不能使用平时程序员惯用的 delete p; 这种形式，而不得不采用一种比较“别扭”的类似 p->Delete() 的形式。

7.4 new 和 delete 的使用规范

C++的动态内存管理是通过 new 和 delete 两个操作来完成的，即用 new 来申请空间，用 delete 来释放空间。在使用 new 和 delete 时，要注意以下原则：

(1) 程序运行时，new 操作和 delete 操作必须一一对应。

用 new 操作申请空间，如果申请成功，必须在以后的某个时刻用 delete 释放该空间，既不能忘记释放，也不能多次释放。前者会引起内存泄漏，后者会引起运行时错误。如下面的程序。

```

/// Program 7.4-1 ///
#include <iostream>
using namespace std;
int main(){
    int *p;
    p=new int(3);
    if(p)
        delete p;
    delete p;
}

```

/// End of Program 7.4-1 ///

在这个程序中，如果用 new 为指针 p 申请空间成功，那么后继的语句会两次释放 p 所指向的空间。这种内存错误对 C/C++ 程序的危害极大，也是很多人对 C++ 语言望而却步甚至对其提出批评的一个重要原因。多次释放同一块内存空间，并不一定立即引起程序运行错误，也不一定会导致程序运行的崩溃，这跟具体的编译器实现有关。但是，多次释放同一块内存空间绝对是一个编程错误，这个错误可能会在其后的某个时刻导致其他逻辑错误的发生，从而给程序的调试和纠错带来困难。考察下面的程序。

```

/// Program 7.4-2 ///
#include <iostream>
using namespace std;

```

```

int main(){
    int *p,*q,*one;
    one=new int;
    if(one) cout << one << endl;
    delete one;
    p=new int(3);
    if(p) cout << p << endl;
    delete one;           //假设这条语句是程序员不小心加上的
    q=new int(5);
    if(q) cout << q << endl;
    cout << (*p)+(*q) << endl; //程序的原意是输出8，即3+5
    delete p;
    delete q;
}

```

//// End of Program 7.4-2 ////

此程序在 VC++ 2005 环境下能顺利通过编译，生成 release 版本后，运行时也看不到运行时错误。程序的执行结果是：

003A4898

003A4898

003A4898

10

从程序的输出可以看出，在将指针 one 所指向的空间释放之后，为指针 p 申请的空间就是原来 one 所指向的单元。由于不小心在为 p 分配空间之后再次使用了 delete one，导致 q 申请到的空间就是原来 p 所申请的空间，这样赋给*q 的值就改写了原来 p 所指向的单元的值，导致最后的输出结果是 10，而程序员本来想看到的结果是 8。由此可知，多次释放同一块内存空间，即使不导致程序运行中断，也会破坏堆环境，使指针与所对应空间的隶属关系出现混乱，从而导致逻辑错误。在大型程序设计中，这种逻辑错误的查找会变得十分费时。

当指针 p 的值为 NULL 时，多次使用 delete p 并不会带来麻烦，因为释放空指针的空间实际上不会导致任何操作。所以，将“不用”的指针设置成 NULL 是一个好的编程习惯。

(2) 当类的数据成员中有指针变量时，在构造函数中用 new 申请空间，并且在析构函数中用 delete 释放空间是一种“标准的”、安全的做法。例如下面的程序。

```

/// Program 7.4-3 ///
#include <iostream>
using namespace std;
class Student{
    char *name;
public:
    Student(char*);
    ~Student();
};

```

```

Student::Student(char *s){
    cout << "In constructor, allocating space" << endl;
    name = new char[strlen(s)+1];
    strcpy(name,s);
    cout << name << endl;
}
Student::~Student(){
    cout << "In destructor, free space" << endl;
    delete name;
}
int main(){
    Student s1("张三");
}
/// End of Program 7.4-3 ///

```

程序的运行结果是：

In constructor, allocating space

张三

In destructor, free space

由于任何一个对象，其构造函数只调用一次，其析构函数也只调用一次，这样就能够保证运行时 `new` 和 `delete` 操作是一一对应的，也就是保证了内存管理的安全性。

在 C++ 语言中，一个构造函数不能调用本类的另一个构造函数，其原因就是为了防止构造函数的相互调用打破了内存申请与释放之间的这种对应关系。

7.5 `delete` 和 `delete[]` 的区别

`delete` 是用来释放空间的操作。在回收用 `new` 分配的单个对象的内存空间的时候用 `delete`，在回收用 `new[]` 分配的一组对象的内存空间的时候用 `delete[]`。那么，如果在回收用 `new` 分配的单个对象的内存空间的时候用 `delete[]`，而在回收用 `new[]` 分配的一组对象的内存空间的时候用 `delete`，会出现什么情况呢？

实际上，要回答上面的问题，就要搞清 `new` 和 `new[]` 在申请空间时的实现细节。由于 `new`（或 `new[]`）操作的实现细节并不属于 C++ 标准的一部分，各个不同的编译器在实现 `new`（或 `new[]`）操作时，其底层机制很可能是不同的。其中，有一种比较直观的做法是，在用 `new[]` 申请空间时，例如用下述语句 `A *p= new A[n];` 申请空间，则实际申请的空间的大小为 `n*sizeof(A)+m`，这多申请的 `m` 字节的空间存放诸如申请的对象的个数、申请空间的长度等信息，在它之后才是一个连续存放的对象数组。VC++ 就是采用这种方式实现 `new[]` 操作的。如果是用 `new` 申请空间，例如用下述语句 `A *p= new A;` 申请空间，由于只申请了一个对象，所以在额外申请的空间中保存的信息与申请多个对象时并不相同。下面的示例程序说明了这一点。

/// Program 7.5-1 ///

#include <iostream>

```
using namespace std;
void PrintContent(void *addr,int offset){
    char *s = (char*)addr;
    int *p = (int*)(s-offset*4);
    for(int i=0;i<offset;i++){
        cout << p[i] << " ";
    }
    cout << endl;
}
class A{
    char c[10];
};
class B{
    char c[10];
public:
    ~B(){}
};
int main(){
    int *p= new int[5];
    PrintContent(p,5);
    delete[] p;
    char *s= new char[100];
    PrintContent(s,5);
    delete[] s;
    double *q= new double;
    PrintContent(q,5);
    delete q;
    A *a1= new A;
    PrintContent(a1,5);
    delete a1;
    A *a2= new A[5];
    PrintContent(a2,5);
    delete[] a2;
    B *b1= new B;
    PrintContent(b1,5);
    delete b1;
    B *b2= new B[5];
    PrintContent(b2,5);
    delete[] b2;
}
```

/// End of Program 7.5-1 ///

程序的输出结果是：

```
0 20 1 111 -33686019
0 100 1 132 -33686019
0 8 1 133 -33686019
0 10 1 134 -33686019
0 50 1 135 -33686019
0 10 1 136 -33686019
54 1 137 -33686019 5
```

现在把该程序的运行情况分析一下。要注意这样几点。

①在使用 new 和 new[] 操作申请空间时，系统会在要求申请的空间大小之外额外申请一块空间，放在返回给用户的空间的前面，存放诸如给用户申请空间的字节数、申请对象的个数等信息。

②函数 PrintContent() 用来在给定地址空间处，倒退若干个整数（int）变量的空间，然后将这几个整数的值输出。

③本程序在每次用 new 或 new[] 操作申请空间后，在返回给用户的地址处倒退 5 个整数的空间，并打印这 5 个整数的值，从中发现关键信息在这些额外空间中存放的位置。

④当用 new 操作申请空间时，由于对应地只需要用 delete 来释放空间就可以了，并不涉及多次调用被释放的对象的析构函数的问题，所以，不用存放申请元素的个数。观察输出可以发现，在返回地址处倒退第 4 整数处存放的用户申请空间的字节数。

⑤当用 new[] 操作申请空间时，根据被申请对象的数据类型的具体情况，有两种安排。如果申请的是基本数据类型的数据，或者虽然申请的是类类型数据，但在释放空间时并不需要调用析构函数（如程序中的 class A），那么额外信息的排列顺序与用 new 操作申请时相同。如果申请的是类类型数据，并且在释放空间时需要调用析构函数（如程序中的 class B），那么会在返回地址处倒退第 5 整数处存放的用户申请空间的字节数（再加 4），在倒退第 1 整数处存放动态数组元素的个数。

以下的程序通过构造函数和析构函数的调用情况考察 new 和 new[] 操作的结果。

```
/// Program 7.5-2 ///
#include <iostream>
using namespace std;
class A{
    int i;
public:
    A(){cout<<"A"<<endl;}
    ~A(){cout<<"~A"<<endl;}
};
int main(){
    A* pA=new A[3];
    int* p=(int*)pA;
    cout<<*(p-1)<<endl;
```

```
delete []pA;
pA = new A;
p=(int*)pA;
cout<<*(p-1)<<endl;
delete pA;
}
/// End of Program 7.5-2 ///
```

程序的执行结果是：

```
A
A
A
3
~A
~A
~A
A
-33686019
~A
```

从程序运行结果可以看出，用 `new[]` 申请空间时，指针 `pA` 的地址是对象数组的首地址，而 `pA` 的前面（低地址）的 4 个字节，则正好存放了数组中对象的个数（在本例中是 3）。而用 `new` 申请空间时，指针 `pA` 指向申请到的对象的首地址，而 `pA` 的前面（低地址）的 4 个字节中的整数值并不为 1，这是因为用 `new` 操作只会申请一个对象，不用保存对象个数的信息。

知道了 `new` 和 `new[]` 的实现细节，就不难明白为什么用 `new` 申请的空间要用 `delete` 释放，而用 `new[]` 申请的空间要用 `delete[]` 释放了。在上例中，如果用 `new` 申请的空间用 `delete[]` 来释放，那么程序就会在给定地址的前面寻找数组元素个数（得到的是一个很大的无符号整数），运行结果是无法预料的。而用 `new[]` 申请的空间用 `delete` 来释放，则 `delete` 操作需要知道的内存块的大小信息无法正确获取（位置不对），也不会多次调用对象的析构函数，这样运行结果同样是无法预料的。

下面的程序演示了当申请空间时，额外信息遭到破坏会发生什么情况。

```
/// Program 7.5-3 ///
#include <iostream>
using namespace std;
class A{
    int i;
public:
    A(){cout<<"A"<<endl;}
    ~A(){cout<<"~A"<<endl;}
};
int main()
```

```

A* pA=new A[3];
int* p=(int*)pA;
*(p-1)=1;
delete []pA;
}

/// End of Program 7.5-3 ///

```

程序的运行结果是：

```

A
A
A
~A

```

由于把指针 pA 所指地址前面的那个整数的值修改为 1（修改之前的值是 3），导致最终在释放空间时，类 A 的析构函数只被调用了 1 次。

7.6 什么是定位放置 new

一般说来，使用 new 申请空间时，是从系统的“堆”（heap）中分配空间。申请所得的空间的位置是根据当时的内存的实际使用情况决定的。但是，在某些特殊的情况下，可能需要在程序员指定的特定内存处创建对象，这就是所谓“定位放置 new”（placement new）操作。定位放置 new 操作的语法形式不同于普通的 new 操作。例如，一般都用如下语句 A *p = new A; 申请空间，而定位放置 new 操作则使用如下语句 A *p = new (ptr) A; 申请空间，其中 ptr 就是程序员指定的内存首地址。如下面的程序。

/// Program 7.6-1 ///

```

#include <new>           //必须包含这个头文件，才能使用placement new
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){cout<<"A"<<endl;}
    ~A(){cout<<"~A"<<endl;}
    void show(){
        cout << num << endl;
    }
};
int main(){
    char memory[100];
    memory[0]='a';
    memory[1]=0;
    memory[2]=0;
}

```



```
memory[3]=0;
cout << (void*)memory << endl;
A* p = new(memory) A;
cout << p << endl;
p->show();
p->~A();
}
```

//// End of Program 7.6-1 ////

程序的执行结果是：

0012FEF0

A

0012FEF0

97

-A

阅读上面的程序，要注意这样几个要点：

- ①要使用定位放置 new 操作，必须包含头文件“new”。
- ②用定位放置 new 操作，既可以在堆栈（stack）上生成对象，也可以在堆（heap）上生成对象。如本例就是在栈上生成一个对象。
- ③使用语句 A *p = new (memory) A; 申请空间时，指针 p 和指针 memory 指向同一片存储区。所以，与其说定位放置 new 操作是申请空间，还不如说是利用空间，真正的申请空间的工作是在此之前完成的。而且事先申请的空间必须足够大，以满足在其上创建对象的要求，否则会引起运行时错误。
- ④使用语句 A *p = new (memory) A; 定位生成对象时，会自动调用类 A 的构造函数，但是由于对象的空间不会自动释放（对象实际上是“借用”别人的空间），所以必须显式地调用类的析构函数，如本例中的 p->~A()。
- ⑤万不得已时才使用“placement new”语法。只有当你真的在意对象在内存中的特定位时才使用它。例如，你的硬件有一个内存映象的 I/O 计时器设备，并且你想放置一个 Clock 对象在那个内存位置。

7.7 在函数中创建动态对象

所谓动态对象，是指用 new 生成的对象。由于动态的内存分配要求程序员自己管理申请的内存，所以必须遵循一定的规则以避免出错。

除了要对 new 操作的返回值作非空检查之外，还要注意“一次 new，一次 delete”原则的实现。如果在一个函数中，用 new 操作生成了一个对象，而 delete 操作（即对象空间的释放）不是在该函数内完成的，就必须设法以某种方式保存动态对象的地址，以便在函数运行结束之后，在某个合适的时刻释放动态对象的内存空间。在这个过程中，初学者容易采用某些不好的甚至是错误的编程方式。

(1) 将动态对象以值的形式返回

在 C++ 语言中，当从某个函数返回一个值时，编译器会直接将函数的某个变量的值（或



函数可访问的堆上的对象的值)传送到调用者函数的某个变量(有名变量或临时无名变量)中去。这是一个值的传递过程,其结果是值被保留下来,而源头的地址信息则丢失了。如果在函数体内申请了一个动态对象,而将动态对象以值的形式返回到函数外部,则会丢失动态对象的地址,从而造成内存泄漏。考察下面的程序。

```
/// Program 7.7-1 ///
#include <iostream>
using namespace std;
int ReturnValue(){
    int *p = new int(3);
    if(p) return *p;
}
int main(){
    int i = ReturnValue();
    cout << i << endl;
    //delete ????
}
/// End of Program 7.7-1 ///
```

此程序在函数 `ReturnValue()` 中动态生成了一个 `int` 型的变量,由于函数的返回值并不是指针,而只是将动态变量的值传递到函数外部以供使用,所以在将来想要释放动态变量的空间时,却发现早已丢失了动态变量的地址。所以,在一个函数内部如果不释放动态对象的空间,就一定要想办法把动态对象的地址传递出去。简单地将动态对象以值的形式返回是一种典型的错误用法。

(2) 通过引用访问动态对象

引用是 C++ 语言引入的机制,虽然引用本质上是通过指针实现的,但是它在书写形式上更为简洁,安全性能也要好一些。但是,如果将引用与动态对象相绑定,却可能给程序员带来麻烦。

假设 `TClass` 是一个用户定义的类类型,那么这条语句 `TClass &r= *new TClass();` 会让程序员很难判断 `r` 的实际情况。因为,如果内存申请失败,那么 `new` 操作会返回空指针,而对空指针解引用会导致运行时错误。所以,通过引用访问动态对象只能按类似下述方式初始化:

```
TClass *p = new TClass();
if(!p) Error( "Can not create TClass object" );
else TClass &r = *p;
```

也就是说,即使最终是通过引用来访问动态对象,中间还是要通过指针来传递动态对象的地址。另一种比较糟糕的做法是,在一个函数中创建一个动态对象,然后返回这个对象的引用。如定义这样一个函数:

```
TClass &Func(){
    TClass *p = new TClass();
    if(!p) Error( "Can not create TClass object" );
    else return *p;
}
```



然后在调用者函数中使用这条语句：TClass &r = Func()。

然而，就算成功地实现了通过引用来访问动态对象，还有一件重要的事情必须做。那就是，在引用变量结束生命周期之前完成对动态对象的释放。也就是编写这样的语句：delete &r。

由于r并不显式地表现为一个指针，所以程序员往往容易忽略对r所引用的动态对象的释放。这也是造成错误的一个潜在因素。

所以，通过引用访问动态对象的较好的做法是：将引用作为函数参数，而将动态对象空间的申请和释放交给指针去处理。如下面的程序。

```
//// Program 7.7-2 ////  
#include <iostream>  
using namespace std;  
class TClass{  
    int num;  
public:  
    TClass(int i){num=i;}  
    friend void show(const TClass &);  
};  
void show(const TClass &t){  
    cout << t.num << endl;  
}  
int main(){  
    TClass t1(3);  
    TClass *p = new TClass(5);  
    show(t1);  
    if(p) show(*p);  
    delete p;  
}  
//// End of Program 7.7-2 ////
```

程序的运行结果是：

```
3  
5
```

由于函数show()接收一个TClass的引用作为参数，而不考虑被引用对象到底是栈上的对象还是堆上的对象，所以能够采用统一的方式处理，充分发挥了引用带来的避免构造参数副本、书写形式简洁等好处。同时，由于对象空间的申请与释放与引用无关，也避免了引用的书写形式可能对内存管理造成的干扰。

7.8 什么是内存池技术

内存池是用来改善标准的new/delete内存管理机制可能造成的运行效率低下的问题的一种技术。在C++程序中，如果频繁地申请大小不同的对象，可能在堆空间造成很多碎片，从而影响动态内存的申请效率。同时，用于通用的内存管理机制要考虑很多复杂的具体情况（如

应付多线程环境等), 难以对算法做有效的优化, 所以, 在一些特殊场合, 利用一些简化条件, 就能够在一定程度上提高内存管理的效率。

经典的内存池 (Memory Pool) 技术, 是一种用于分配大量大小相同的小对象的技术。通过该技术可以极大加快内存分配/释放过程。既然是针对特定对象的内存池, 所以内存池一般设置为类模板, 根据不同的对象的大小进行实例化。

经典的内存池的内部结构是这样的: 先申请一块连续的内存空间 (Memory Block), 该空间能容纳一定数量的对象。每个对象连同一个指向下一个对象的指针一起构成一个内存结点 (Memory Node)。各内存结点之间通过指针形成一个链表, 链表当中的每一个结点都是一块可供分配的内存空间。某个内存结点一旦分配出去, 就将从链表中去除; 而一旦释放了某个内存结点的空间, 又会将该结点重新加入自由内存结点链表。如果一个内存块中的所有内存结点已经分配完毕, 而程序继续申请新的对象空间, 则会再次申请一个内存块, 以便从内存块中分割出新的内存结点。所以, 内存池类模板有这样几个数据成员: 两个常量, MemBlockSize(内存块的大小)、ItemSize(对象的大小), 以及两个指针变量 pMemBlockHeader (指向内存块链表的第一个结点)、pFreeNodeHeader (指向内存结点链表的第一个结点)。开始, 这两个指针均为空。以下的程序提供了经典内存池的一种实现方案。

```
/// Program 7.8-1 ///
/** mempool.h **/
//模板参数ObjectSize代表实际申请的对象的大小, NumofObjects代表每个内存块所能容纳的对象个数
template <int ObjectSize,int NumofObjects=20>
class MemPool{
private:
    const int MemBlockSize; //每个内存块的大小
    const int ItemSize; //每个内存结点的大小
    struct _FreeNode{
        _FreeNode* pNext;
        char data[ObjectSize];
    };
    struct _MemBlock{
        _MemBlock* pNext;
        _FreeNode data[NumofObjects];
    };
    _MemBlock* pMemBlockHeader;
    _FreeNode* pFreeNodeHeader;
public:
    MemPool():ItemSize(ObjectSize+sizeof(_FreeNode*)),
    MemBlockSize(sizeof(_MemBlock*)+NumofObjects*(ObjectSize+sizeof(_FreeNode*))) {
        pMemBlockHeader=NULL;
        pFreeNodeHeader=NULL;
    }
}
```



```
~MemPool(){
    _MemBlock *ptr;
    while(pMemBlockHeader){
        ptr = pMemBlockHeader->pNext;
        delete pMemBlockHeader;
        pMemBlockHeader = ptr;
    }
}
void * malloc();
void free(void*);
};

template <int ObjectSize,int NumofObjects>
void* MemPool<ObjectSize,NumofObjects>::malloc(){ // 没有参数
if (pFreeNodeHeader == NULL){
    _MemBlock* pNewBlock = new _MemBlock;
    pNewBlock->data[0].pNext = NULL;
    for (int i = 1; i < NumofObjects; ++i)
        pNewBlock->data[i].pNext = &pNewBlock->data[i-1];
    pFreeNodeHeader = &pNewBlock->data[NumofObjects-1];
    pNewBlock->pNext = pMemBlockHeader;
    pMemBlockHeader = pNewBlock;
}
void* pFreeNode = pFreeNodeHeader;
pFreeNodeHeader = pFreeNodeHeader->pNext;
return pFreeNode;
}
template <int ObjectSize,int NumofObjects>
void MemPool<ObjectSize,NumofObjects>::free(void* p){
    _FreeNode* pNode = (_FreeNode*)p;
    pNode->pNext = pFreeNodeHeader;
    pFreeNodeHeader = pNode;
}
/** end of mempool.h**/

/** main.cpp ***/
#include <iostream>
#include "mempool.h"
using namespace std;
class ActualClass{
    static int count;
```



```

int num[10];
public:
    ActualClass(){
        count++;
        for(int i=0;i<10;i++)
            num[i]=count+i;
    }
    void show(){
        cout << this << " : ";
        for(int i=0;i<10;i++){
            cout << num[i] << " ";
        }
        cout << endl;
    }
    void * operator new(size_t size);
    void operator delete(void *p);
};

class theMemoryPool{
    static MemPool<sizeof(ActualClass),2> mp;
    friend class ActualClass;
};

void * ActualClass::operator new(size_t size){
    return theMemoryPool::mp.malloc();
}
void ActualClass::operator delete(void *p){
    theMemoryPool::mp.free(p);
}

MemPool<sizeof(ActualClass),2> theMemoryPool::mp;
int ActualClass::count;
int main(){
    ActualClass *p1 = new ActualClass;
    p1->show();
    ActualClass *p2 = new ActualClass;
    p2->show();
    delete p1;
    p1 = new ActualClass;
    p1->show();
    ActualClass *p3 = new ActualClass;
    p3->show();
    delete p1;
}

```



```
delete p2;  
delete p3;  
}  
/** end of main.cpp ***/  
/// End of Program 7.8-1 ///
```

程序的运行结果是：

```
003A4B80 : 1 2 3 4 5 6 7 8 9 10  
003A4B54 : 2 3 4 5 6 7 8 9 10 11  
003A4B80 : 3 4 5 6 7 8 9 10 11 12  
003A5C78 : 4 5 6 7 8 9 10 11 12 13
```

阅读以上程序，注意这样几个要点：

①对一种特定的类对象而言，内存池中内存块的大小是固定的，内存结点的大小也是固定的。内存块在申请之初就被划分为多个内存结点，每个 Node 大小为 ItemSize（对象的大小加上一个指针的空间）。刚开始时所有的内存结点都是自由的，它们被串成链表。

②指针变量 pMemBlockHeader 是把所有申请的内存块（Memory Block）串成一个链表，以便通过它可以释放所有申请的内存。pFreeNodeHeader 变量则是把所有自由内存结点（Free Node）串成一个链。pFreeNodeHeader 为空表明已经没有可用的自由内存结点，必须申请新的内存块。

③申请空间的过程如下。在自由内存结点链表（pFreeNodeHeader）非空的情况下，malloc 过程只是从链表中摘下自由内存结点链表的头一个结点，然后把链表头指针移动到下一个结点上去。否则，意味着需要一个新的内存块（Memory Block）。这个过程需要将新申请的内存块切割成多个内存结点，并把它们串起来，内存池技术的主要开销就在这里。

④释放对象的过程极其简单，只要把被释放的结点重新插入到自由内存结点链表的开头即可。最后被释放的结点就是下一个即将被分配的结点。

⑤内存池技术申请/释放内存的速度很快。其内存分配过程多数情况下复杂度为 O(1)，主要开销在 pFreeNodeHeader 为空时需要生成新的内存块。内存释放过程复杂度为 O(1)。

⑥在上面的程序中，指针 p1 和 p2 连续两次申请空间，它们代表的地址之间的差值为 44，正好为一个内存结点的大小（sizeof(ActualClass)+4）。指针 p1 所指向的对象被释放后，再次申请空间，得到的地址与刚刚释放的地址正好相同（都是 0X003A4B80）。指针 p3 代表的地址与前两个对象的地址相距很远，原因是第一内存块中的内存结点已经用完（为了便于验证，此程序中每个内存块只容纳两个内存结点），p3 指向的对象位于第二内存块中。

⑦以上的内存池方案并不完美，比如：只能单个单个申请对象空间，不能申请对象数组；内存池中内存块的个数只能越来越大，不能减小等。现在已有很多新的改进方案，有兴趣的读者可以查阅相关资料。



第8章 | 继承与多态

8.1 私有成员会被继承吗

在类的继承中，基类的私有成员在派生类中是“不可见”的。这种“不可见”，是指在派生类的成员函数中，或者通过派生类的对象（指针、引用）不能直接访问它们。但是，不能直接访问并不代表不能访问。在派生类中还是能够通过调用基类的公有函数的方式间接地访问基类的私有成员，包括私有成员变量和私有成员函数。考察下面的程序。

```
/// Program 8.1-1 ///
#include <iostream>
using namespace std;
class A{
    int i;
public:
    A(){i=5;}
    int GetI(){
        return i;
    }
    void UsePrivateFunc(){
        PrivateFunc();
    }
};
class B:public A{
public:
    void PrintBaseI(){
        cout << GetI() << endl;
    }
    void UsePrivateFunction(){
        UsePrivateFunc();
    }
};
int main(){
    B b;
```



```
b.PrintBaseI();
b.UsePrivateFunction()
}
```

```
/// End of Program 8.1-1 ///
```

在类 B 中，由于其基类 A 的成员变量 i 和成员函数 PrivateFunc()都是私有的，所以在类 B 的成员函数中无法直接访问到它们。但是，由于类 A 的公有成员函数 GetI()可以访问到私有成员变量 i，而 UsePrivateFunc()可以访问私有成员函数 PrivateFunc()，所以在类 B 中通过调用函数 GetI()和 UsePrivateFunc()就可以间接地访问基类 A 中的私有成员。程序的输出结果是：

```
5
```

```
This is a private function of base class
```

那么，如果基类中并没有提供访问私有成员的公有函数，那么其私有成员是否“存在”呢？还会不会被继承呢？其实，这些私有成员的确是存在的，而且会被继承，只不过程序员无法通过“正常”的渠道访问到它们。下面的程序以一种特殊的方式访问了类的私有成员。

```
/// Program 8.1-2 ///
```

```
#include <iostream>
using namespace std;
class A{
    int i;
    void PrivateFunc(){ cout << "This is a private function of base class" << endl;}
public:
    A(){i=5;}
};

class B:public A{
public:
    void PrintBaseI(){
        int *p=reinterpret_cast<int*>(this);           //获取当前对象的首地址
        cout << *p << endl;
    }
    void UsePrivateFunction(){
        void (*fun)();          //定义函数指针，用来存放基类私有成员函数的入口地址
        _asm{
            mov eax,A::PrivateFunc
            mov fun,eax
        }
        fun();
    }
};

int main(){
    B b;
```

```

    b.PrintBaseI();
    b.UsePrivateFunction();
}

/// end of Program 8.1-2 ///

```

虽然类 A 没有提供访问私有成员变量 i 的公有方法，但是在类 A（以及类 A 的派生类）对象中，都包含有变量 i 的数据实体，所以只要将一个 int* 指针指向对象的首地址，就可以顺利“取出”成员变量 i。同样，虽然类 A 并没有提供访问私有成员函数 PrivateFunc() 的公有函数，但在程序的代码区依然存有函数 PrivateFunc() 的代码，通过内联汇编获取该函数的入口地址，仍然可以顺利地访问到该函数。Program 8.1-2 的运行结果与 Program 8.1-1 相同。

综上所述，类的私有成员一定存在，也一定被继承到派生类中。只不过受到 C++ 语法的限制，在派生类中访问基类的私有成员只能通过间接的方式进行。

8.2 怎样理解构造函数不能被继承

在很多 C++ 的教材中，都谈到类的构造函数、析构函数、赋值操作符函数是不能被继承的。而在实际编程的过程中，却可以看到在派生类的成员函数中显式调用基类的构造函数或者赋值操作符函数，程序照样通过编译并正常运行。如下面的程序。

```

/// Program 8.2-1 ///
#include <iostream>
using namespace std;
class A{
protected:
    int num;
public:
    A(int i){num=i;}
    A& operator=(const A &a){
        num=a.num;
        return *this;
    }
};

class B:public A{
    double dig;
public:
    B(double d):A(int(d)+1){dig=d;}
    B& operator=(const B&);
    void show(){
        cout << num << "," << dig << endl;
    }
};

B& B::operator =(const B &b){

```

```

A::operator=(b);
dig=b.dig;
return *this;
}
int main(){
    B obj(8.7);
    obj.show();
}
/// End of Program 8.1-1 ///

```

在定义派生类的赋值操作符函数 `B::operator=()` 时，显式调用了基类的赋值操作符函数 `A::operator()`，程序能够通过编译并正常运行，输出结果为：9, 8.7。

既然基类的赋值操作符函数不能被派生类继承，为什么可以在派生类的成员函数中调用基类的赋值操作符函数呢？应该如何理解类的构造函数、析构函数和赋值操作符函数不能被继承呢？

继承是面向对象的程序设计中一个重要的概念。从程序设计方法学的角度来看，对象是现实世界的事物在计算机程序中的表示，而类是同一类对象的描述。同类对象之间具有相同的属性和行为。当从一个类派生出一个新类的时候，派生类对象和基类对象之间是一种“is-a”关系，也就是说，派生类对象可以当做一个基类对象看待。基类对象的所有属性，派生类对象都有，基类对象的所有行为，派生类对象也都有。这就是继承的本质，也是谈论继承关系是否成立的最主要的依据。

从程序实现的角度来看，在一个类中定义的属性和行为，如果被派生类继承，派生类就不用再次定义这些属性和行为，从而派生类对象可以表现出与基类对象相同的属性和行为。对于类的构造函数来说，它的语义是为类对象进行初始化，如果派生类继承基类的构造函数，就意味着可以利用基类的构造函数为派生类对象初始化，而这是不可能的。同样，也不能利用派生类的析构函数来销毁一个派生类对象，不能利用基类的赋值操作符函数完成派生类对象之间的赋值操作。

因此，不能继承构造函数、析构函数和赋值操作符函数是从继承的概念出发得出的结论，并不是说在派生类中不能调用基类的构造函数、析构函数和赋值操作符函数。不从基类继承构造函数、析构函数和赋值操作符函数，意味着派生类必须定义自己的构造函数、析构函数和赋值操作符函数，不管是由编译器默认提供，还是由程序员显式定义。

同样，基类中声明的友员函数或友员类，永远只能是基类的友员，而不会变成派生类的友员。因此，基类的友员也是不会被继承的。

8.3 什么是虚拟继承

虚拟继承一般发生在多重继承的情况下。C++允许一个类有多个父类，这样就形成多重继承。多重继承使得派生类与基类的关系变得更为复杂，其中一个容易出现的问题是某个基类沿着不同的路径被派生类继承（即形成所谓“菱形继承”，如图 8-1 所示），从而在一个派生类对象中存在同一个基类对象的多个拷贝。例如下面的例子：

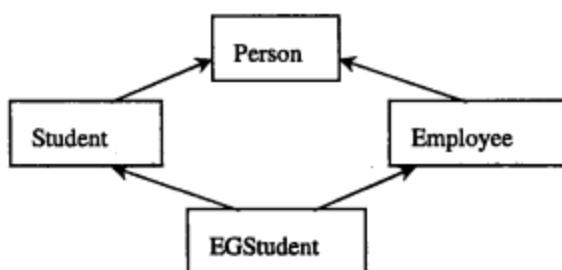


图 8-1 菱形继承示意图

```

/// Program 8.3-1 ///
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;
//以下是一个表示“人员”的类
class Person{
protected:
    string IdPerson;          //身份证号
    string Name;              //姓名
public:
    Person(string s1,string s2){
        IdPerson=s1;
        Name=s2;
    }
};

//以下是一个表示“学生”的类
class Student:public Person{
    int No;
public:
    Student(string s1,string s2, int n):Person(s1,s2),No(n){}
};

//以下是一个表示“高校职工”的类
class Employee:public Person{
    int No;
public:
    Employee(string s1,string s2, int n):Person(s1,s2),No(n){}
};

//以下是一个表示“在职研究生”的类
class EGStudent:public Student, public Employee{
};
  
```



```
class EGStudent:public Employee,public Student{
    int No;
public:
    EGStudent(string s1,string s2, int n):Employee(s1,s2,n),Student(s1,s2,n),No(n){}
    void show(){
        cout << Employee::IdPerson << "," << Employee::Name << "," << No << endl;
    }
};

int main(){
    EGStudent one("330106197304210456","张三",123);
    one.show();
    cout << "sizeof(string)=" << sizeof(string) << endl;
    cout << "sizeof(Person)=" << sizeof(Person) << endl;
    cout << "sizeof(Employee)=" << sizeof(Employee) << endl;
    cout << "sizeof(Student)=" << sizeof(Student) << endl;
    cout << "sizeof(EGStudent)=" << sizeof(EGStudent) << endl;
}
```

//// End of Program 8.3-1 ////

程序的执行结果是：

```
330106197304210456,张三,123
sizeof(string)=32
sizeof(Person)=64
sizeof(Employee)=68
sizeof(Student)=68
sizeof(EGStudent)=140
```

在这个程序中，EGStudent 类有两个父类：Employee 和 Student，而 Employee 和 Student 都是 Person 类的派生类。通过观察这几个类的大小可以发现这样一组等式：

```
sizeof(EGStudent)=sizeof(Employee)+sizeof(Student)+sizeof(int)
sizeof(Employee)=sizeof(Person)+sizeof(int)
sizeof(Student)=sizeof(Person)+sizeof(int)
```

也就是说，在一个 EGStudent 类对象中包含了两个 Person 类对象，一个来自 Employee 类对象，一个来自 Student 类对象。在 EGStudent 类的成员函数 show() 中，直接访问 IdPerson 或 Name 都会引发编译错误，因为编译器不知道它们指的是哪个 Person 对象中的成员。所以在上面的程序中，在 show() 中显示的是 Employee 中的成员（IdPerson 和 Name）。实际上，在 EGStudent 类对象中还有来自 Student 类的成员（IdPerson 和 Name）。

从逻辑上说，一个在职研究生只可能有一个名字和一个身份证号码，所以在一个 EGStudent 类对象中有 IdPerson 和 Name 字段的两个拷贝是不合理的，只需要一个拷贝就可以了。虚拟继承就是解决这个问题的，通过把继承关系定义为虚拟继承，在构造 EGStudent 类对象的时候，EGStudent 类的祖先类 Person 的对象只会被构造一次，这样就可以避免存在多个 IdPerson 和 Name 的拷贝的问题。将程序 Program 8.3-1 修改之后的代码如下。

```

/// Program 8.3-2 ///
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;
class Person{
protected:
    string IdPerson;
    string Name;
public:
    Person(string s1,string s2){
        IdPerson=s1;
        Name=s2;
    }
    Person(){}
};

class Student:public virtual Person{
    int No;
public:
    Student(string s1,string s2, int n):Person(s1,s2),No(n){}
};

class Employee:public virtual Person{
    int No;
public:
    Employee(string s1,string s2, int n):Person(s1,s2),No(n){}
};

class EGStudent:public virtual Employee,public virtual Student{
    int No;
public:
    EGStudent(string s1,string s2, int n):Employee(s1,s2,n),Student(s1,s2,n),No(n){}
    void show(){
        cout << IdPerson << "," << Name << "," << No << endl;
    }
};

int main(){
    EGStudent one("330106197304210456","张三",123);
    one.show();
    cout << "sizeof(string)=" << sizeof(string) << endl;
    cout << "sizeof(Person)=" << sizeof(Person) << endl;
}

```

```

cout << "sizeof(Employee)=" << sizeof(Employee) << endl;
cout << "sizeof(Student)=" << sizeof(Student) << endl;
cout << "sizeof(EGStudent)=" << sizeof(EGStudent) << endl;
}

/// End of Program 8.3-2 ///

```

程序的执行结果是：

```

,,123
sizeof(string)=32
sizeof(Person)=64
sizeof(Employee)=72
sizeof(Student)=72
sizeof(EGStudent)=88

```

在解读上面这段程序的时候，注意这样几个要点：

①一个类（如程序中的 Person 类）的对象如果想在派生类对象中只存在一个拷贝，那么必须在它的派生类中把它声明为虚基类，就像 Student 和 Employee 中的声明那样。

②为了实现虚拟继承，派生类对象的大小会增加 4。所以，在上面的程序中，
 $\text{sizeof}(EGStudent) = \text{sizeof}(Employee) + \text{sizeof}(Student) - \text{sizeof}(Person) + \text{sizeof}(\text{int}) + 4$ 。

③由于在 EGStudent 类对象中，只存在一份 Person 类对象的拷贝，所以 Student 和 Employee 的构造函数实际上并未对 Person 类对象进行初始化。也就是说，在 EGStudent 类的构造函数中，Person 类对象必须单独构造。在这例中，由于在初始化列表中没有显式调用 Person 类的构造函数，所以 IdPerson 和 Name 字段是空字符串。

④在上面的程序中，如果将类 EGStudent 的声明改为 class EGStudent:public Employee,public Student，那么除了 $\text{sizeof}(EGStudent)$ 会变成 84 外，其他的不会发生变化。因为虚拟继承只是表明某个基类的对象在派生类对象中只被构造一次，而在本例中类 Student 和 Employee 对象在 EGStudent 对象中本来就不会被构造多次，所以不将它们声明虚基类也是完全可以的。

8.4 怎样编写一个不能被继承的类

在 Java 语言中，一个类用 final 修饰，那么它就是一个“终结类”，即不能由该类派生出其他的类。在 C# 语言中，使用 sealed 关键字可以达到相同的目的。那么，在 C++ 语言中，怎样定义一个类使之成为一个“终结类”呢？

由于 C++ 并没有直接提供这种支持，所以我们必须自己实现。基本思路是这样的：由于任何派生类的对象在创建的时候，都必然会调用其所有祖先类的构造函数，所以，只要这些构造函数中任何一个无法访问，就可以阻止该派生类对象的构造。

如果将一个类的构造函数定义成私有 (private)，当然可以阻止它进一步派生。但是这样这个类本身也无法实例化了。如果在该类中定义一个公有的静态成员函数，由这个函数来完成对象的初始化工作，也会带来很多问题。最主要的就是破坏了程序员使用类的惯例：只能调用一个具有特殊名字的静态函数才能创建对象。如果是在堆上创建对象，还存在忘记使用 delete 的危险，毕竟 new 操作是在那个特殊的静态成员函数内部使用的。



实际上，我们可以使用“虚拟继承”来实现终结类。一个基类如果被虚拟继承，那么在创建它的任何后辈类（派生类的派生类）的对象时，它的构造函数都是独立被调用的。这一点和一般的继承关系不同。例如，由类 A 派生出类 B，再由类 B 派生出类 C，那么在一般的继承关系中，类 C 的构造函数只是调用了类 B 的构造函数，而由类 B 的构造函数去调用类 A 的构造函数。而如果类 A 是被虚拟继承的，那么类 C 的构造函数会先调用类 A 的构造函数，然后再调用类 B 的构造函数，并且由于类 A 的构造函数已经被调用过了，所以在类 B 的构造函数中，类 A 的构造函数不会再次被调用。

利用虚拟继承的这种特性，我们可以设计这样一个基类 FinalParent，它不定义任何数据成员，这样任何类从它派生并不会增加空间上的开销。将它的默认构造函数的访问权限设定为 protected，这样它自身不能生成任何实例，只能用作基类。一个使用基类 FinalParent 的例子如下所示。

```
/// Program 8.4-1 ///
#include <iostream>
using namespace std;
class FinalParent{
protected:
    FinalParent(){}
};

class FinalClass:private virtual FinalParent{
    int num;
public:
    FinalClass(){num=5;}
    void show(){
        cout << num << endl;
    }
};

int main(){
    FinalClass obj;
    obj.show();
}

/// End of Program 8.4-1 ///
```

此程序能够顺利通过编译，并产生输出结果：5。

在此程序中，类 FinalParent 被虚拟继承，并且是通过私有继承产生派生类 FinalClass。由于 FinalParent::FinalParent() 的访问权限是 protected，所以可以在类 FinalClass 的构造函数中调用它。这样，类 FinalClass 可以被实例化，在该类中定义任何成员的访问不受父类的影响。

但是，由于采用的是私有继承的方式，所以 FinalParent::FinalParent() 在类 FinalClass 中的访问权限就变成了 private。也就是说，任何 FinalClass 的派生类都无法调用类 FinalParent 的构造函数。而正是由于类 FinalParent 是被类 FinalClass 虚拟继承的，所以在 FinalClass 的派生类的构造函数中必须直接调用类 FinalParent 的构造函数，这样就导致了编译错误。考察下面的程序。

```

/// Program 8.4-2 ///
#include <iostream>
using namespace std;
class FinalParent{
protected:
    FinalParent(){}
};

class FinalClass:private virtual FinalParent{
    int num;
public:
    FinalClass(){num=5;}
    void show(){
        cout << num << endl;
    }
};

class FinalChild: public FinalClass{
};

int main(){
    FinalClass a;
    a.show();
    FinalChild b;
    a.show();
}

/// End of Program 8.4-2 ///

```

此程序无法通过编译。原因是在创建类 FinalChild 的对象时，由于虚拟继承的缘故，在 FinalChild::FinalChild() 中必须独立调用类 FinalParent 的构造函数，而此构造函数在类 FinalClass 中已经变成私有的（private），不能被类 FinalChild 的任何成员函数所访问，因此导致编译错误。

所以，任何一个类，只要有私有虚拟继承类 FinalParent，就不能被继续继承，从而以一种简洁的方式实现了“终结类”。

8.5 关于隐藏

隐藏是“同名隐藏”的简称。同名隐藏是指具有相同名称的标识符，其作用域发生重叠，在此重叠的作用域范围内，只能够直接访问到其中一个标识符的现象。隐藏是一个编译时概念，它与代码中标识符的解析相关。在类的设计中，要注意派生类成员对基类成员的隐藏现象。当派生类中定义的成员与基类中某个成员同名时，在派生类的成员函数中，或者通过派生类的对象，不能直接访问到基类中的同名成员，即发生了隐藏。

在 C++ 中，有些隐藏是“真”隐藏，也就是说，被隐藏的标识符无法通过其他途径访问到，C++ 语法阻止了这种访问的可能性（如 Program 2.8-1 中 main() 函数的局部变量 i）。而在

大多数情况下，同名隐藏只是一种“假”隐藏。虽然不能通过标识符的名称直接访问到它，但通过指明标识符的作用域，仍然可以访问到被隐藏的标识符。在某个上下文环境中，如果一个标识符与别的标识符因同名而无法区分，就应显式指明标识符的作用域。C++程序的作用域有全局作用域、名称空间作用域、类作用域、块作用域等，只有块作用域无法显式指明，其他作用域都可以通过作用域指示符“::”来指定。

在类的设计中，处于不同的类中的标识符实际上具有不同的作用域（处于不同的类体中）。在派生类中定义的标识符在基类的类域中无效。所以，即使在派生类中定义了与基类中同名的标识符，只要显式地指明了标识符的作用域（也就是使用完全限定名），仍然可以有效地将它们区分开来。考察下面的程序。

```
//// Program 8.5-1 /////
#include <iostream>
using namespace std;
class Base{
protected:
    int sameint;
public:
    void SetValue(int i){
        sameint=i;
    }
    void ShowValue(){
        cout << sameint << endl;
    }
};
class Derived:public Base{
    int sameint;
public:
    void SetValue(int i){
        sameint=i;
    }
    void SetBaseValue(int i){
        Base::sameint=i;
    }
    void ShowValue(){
        cout << sameint << endl;
    }
    void ShowBaseValue(){
        Base::ShowValue();
    }
};
int main()
```

```

Derived obj;
obj.SetValue(5);
obj.ShowValue();
obj.SetBaseValue(8);
obj.ShowBaseValue();
obj.Base::ShowValue();
}

```

//// End of Program 8.5-1 ////

程序的输出结果是：

```

5
8
8.

```

可以看到，在派生类 Derived 中，如果简单地使用 sameint 或 ShowValue()，那么因为在基类和派生类中都有这样的标识符，所以只能把它们当做是派生类中的标识符。但如果在标识符前显式地加上作用域（类名），就可以清楚地分辨出它们是属于哪个类的标识符。如 Base::sameint 指的是基类中的成员变量，Base::ShowValue()指的是基类中的成员函数。如果你愿意，可以把派生类中的成员变量写成 Derived::sameint，虽然没有必要，但绝对是正确的。

所以，这种同名隐藏实际上是“假”隐藏。在概念上，应将基类和派生类中的成员理解成完全不同的实体，即使它们同名也没有关系。在实际编程中，通常发生的是派生类中的成员函数对基类的成员函数的隐藏。如下面的程序。

```

//// Program 8.5-2 /////
#include <iostream>
#include <string>
using std::string;
using std::cout;
using std::endl;
class A{
public:
    void f(string s){
        cout << s << endl;
    }
};
class B:public A{
    int f(){
        return 0;
    }
};
int main(){
    B b;
    b.f("ok");
}

```



```
}
```

/// End of Program 8.5-2 ///

此程序不能通过编译。编译器给出的出错信息是，“B::f”：函数不接受 1 个参数。这就是因为类 B 当中的函数 f()隐藏了类 A 中的函数 f(string s)，尽管这两个函数的参数列表并不相同。把语句 b.f("ok")改写成 b.A::f("ok")，程序就可以通过编译并正常运行了。

但是，并不是只有派生类的成员变量隐藏基类的成员变量，或者是派生类的成员函数隐藏基类的成员函数，只要派生类的成员中有一个与基类中的成员同名，就形成对基类所有同名成员的隐藏，而不管该成员是数据成员还是成员函数。如下面的例子。

```
/// Program 8.5-3 /////
#include <iostream>
class A{
public:
    void show(){
        std::cout << "ok" << std::endl;
    }
};
class B:public A{
public:
    int show;
};
int main(){
    B b;
    b.show=3;
    b.show();
}
/// End of Program 8.5-3 ///
```

该程序无法通过编译。问题出在 b.show(); 这条语句上。派生类中的成员变量 show 对基类中的成员函数 show() 形成隐藏，从而导致不能通过 B 类对象直接调用定义在其基类 A 中的成员函数 show()。要达成此目的，必须写成 b.A::show()。

可以这样理解发生在类的继承关系中的隐藏：基类的所有成员都被继承到派生类中，只是由于派生类中本身又有同名的成员，所以在访问那些被隐藏的继承自基类的成员时，必须显式指明它们的作用域。由于标识符的作用域可能重叠，所以在访问一个标识符时可能需要多次使用作用域指示符::。在下面的程序中，在派生类中实现了对比直接基类更高级别的基类的同名成员的访问。之所以在类 Grand 和类 Father 中将变量 sameint 的存取级别设定为 protected，是为了在它们的派生类中有访问该成员变量的权限。

```
/// Program 8.5-4 /////
#include <iostream>
using namespace std;
class Grand{
protected:
```

```

int sameint;
public:
    Grand(int i){
        sameint=i;
    }
};

class Father:public Grand{
protected:
    int sameint;
public:
    Father(int i):Grand(i+1){
        sameint=i;
    }
};

class Son:public Father{
    int sameint;
public:
    Son(int i):Father(i+1){
        sameint=i;
    }
    void ShowGrand(){
        cout << Son::Father::Grand::sameint << endl;
    }
};

int main(){
    Son obj(5);
    obj.ShowGrand();
}

/// End of Program 8.5-4 ///

```

程序的输出结果是 7。如果把 `cout << Son::Father::Grand::sameint << endl;` 改写成：

`cout << Grand::sameint << endl;`

程序仍然可以正确运行。只不过前者更加清楚地显示了各个类之间的派生关系。

如果在基类中定义了虚函数，在派生类中重新定义一个与基类中的虚函数的函数原型相同的函数，就形成对基类中虚函数的覆盖。对基类的虚函数进行覆盖时，如果函数名和函数参数列表是相同的，而函数的返回值类型不同，就会引发一个编译错误。而一般的同名隐藏不会理会函数的返回值类型。如果派生类中定义了一个与基类中的虚函数同名、但参数列表不同的函数，则此函数是一个普通成员函数（非虚函数），并形成对基类中同名虚函数的隐藏。

函数的隐藏与覆盖是两个不同的概念。隐藏是一个静态概念，它代表了标识符之间的一种屏蔽现象，而覆盖则是为了实现动态联编，是一个动态概念。但隐藏和覆盖也有联系：形



成覆盖的两个函数之间一定形成隐藏。例如，可以对虚函数采用“实调用”（也就是说，尽管被调用的是虚函数，但被调用函数的地址是在编译阶段静态确定的），那么派生类中的虚函数仍然形成对基类中虚函数的同名隐藏。另外，在通过指针（或引用）调用虚函数时，如果指明虚函数的作用域，那么实际上发生的还是实调用。见下面的例子。

```
//// Program 8.5-5 /////
#include <iostream>
using namespace std;
class Base{
public:
    virtual void show(){
        cout << "In Base" << endl;
    }
};

class Derived: public Base{
public:
    void show(){
        cout << "In Derived" << endl;
    }
};

int main(){
    Base b;
    b.show();
    Derived d;
    d.show();           //对函数show()的实调用
    d.Base::show();     //对函数show()的实调用
    Base *pb;
    pb = &d;
    pb->show();        //对函数show()的虚调用
    pb->Base::show();   //对函数show()的实调用
}

//// End of Program 8.5-5 /////

```

程序的输出结果是：

```
In Base
In Derived
In Base
In Derived
In Base
```

语句 `pb->show()` 是通过指向基类 `Base` 的指针 `pb`，访问了在派生类 `Derived` 中实现的函数 `show()`，是一种典型的虚调用。而将它改成 `pb->Base::show()` 之后，由于显式地指明了要调用哪一个类中定义的函数，因而变成了一种实调用，与普通的同名隐藏的情形相同。



在设计类的时候，要避免在派生类中重新定义基类的非虚函数。如果重新定义继承而来的非虚函数，那么使用指针来调用函数时，对函数的调用结果是由指向对象的指针的静态数据类型决定的，而不是由对象本身的类型来决定。另外，从概念上说，派生类对象可以被看作一个基类对象，因此它们的行为应该是一致的。在派生类中重新定义基类的非虚函数，说明类的继承关系设计得不合理，所以应该避免这种做法。

8.6 什么是 RTTI

RTTI 是 Runtime Type Identification 的缩写，是“运行时类型识别”的意思。RTTI 特性是 C++ 语言加入较晚的特性之一，这与 C++ 对性能的要求有关。动态的类型识别会在一定程度上造成运行效率的降低。所以，除非确有必要，否则应尽量减少 RTTI 的使用。

RTTI 通过两个运算符实现：`typeid` 和 `dynamic_cast`。其中 `typeid` 不是“纯”的运行时运算符，它也可以用于静态地确定变量或表达式的类型，这取决于 `typeid` 的操作数本身是否拥有虚函数。下面分别介绍这两个运算符。

(1) `typeid` 的用法

`typeid` 可以静态地确定操作数的类型，也可以动态地确定操作数的类型，这取决于操作数本身是否拥有虚函数。当 `typeid` 的操作数是一个基本数据类型的变量，或者是一个不带虚函数的对象时，`typeid` 的运算结果是在编译阶段决定的，所以是一种静态的类型判断。见下面的例子。

```
//// Program 8.6-1 ////
#include <iostream>
using namespace std;
template <typename T> void func(T a){
    if(typeid(T)==typeid(int))
        cout << "Instance with int" << endl;
    else if(typeid(T)==typeid(double))
        cout << "Instance with double" << endl;
}
int main(){
    func(1);
    func(0.5);
}
/// Program 8.6-1 ///
```

程序的输出结果是：

```
Instance with int
Instance with double
```

在此程序中，函数模板 `func()` 被实例化的时候，`T` 被实例化为 `int` 和 `double`，`typeid(T)` 是在编译阶段静态确定的。在函数模板内部，可以通过 `typeid` 操作决定在模板参数被实例化为不同数据类型时要采取的不同行动。

`typeid` 更多的时候是在运行时用来动态地确定指针或引用所指向对象的类型，这时要求



typeid 所操作的对象一定要拥有虚函数。见下面的例子。

```
/// Program 8.6-2 ///
#include <iostream>
using namespace std;
class A{
public:
    virtual void func(){}
};

class B:public A{
};

void reportA(A *pa){
    if(typeid(*pa)==typeid(A))
        cout<<"Type of *pa is: A"<<endl;
    else if(typeid(*pa)==typeid(B))
        cout<<"Type of *pa is: B"<<endl;
}

void reportB(B *pb){
    if(typeid(*pb)==typeid(A))
        cout<<"Type of *pb is: A"<<endl;
    else if(typeid(*pb)==typeid(B))
        cout<<"Type of *pb is: B"<<endl;
}

int main(){
    A a,*pa;
    B b,*pb;
    pa=&a;
    reportA(pa);
    pa=&b;
    reportA(pa);
    pb=static_cast<B*>(&a);
    reportB(pb);
    pb=&b;
    reportB(pb);
}

/// End of Program 8.6-2 ///
```

程序的运行结果是：

```
Type of *pa is: A
Type of *pa is: B
Type of *pb is: A
Type of *pb is: B
```

从程序的运行结果可以看出，使用 typeid 确实能够在程序运行期间动态地判断指针所指对象的实际类型。使用引用可以达到同样的效果。

要注意的是，如果在 class A 的定义中，将函数 func() 定义为普通函数（即将前面的 virtual 关键字去掉），那么 typeid(*pa) 的结果永远是 typeid(A)，而 typeid(*pb) 的结果永远是 typeid(B)。也就是说，由于 pa 和 pb 所指的对象中没有虚函数，该对象就没有虚函数表存放运行时信息，typeid 实际上变成了一种静态运算符。这一点一定要清楚。

C++ 中的一切“动态”机制，包括虚函数、RTTI 等，都必须通过指针或引用实现。换句话说，指针所指的对象或引用所绑定的对象，在运行阶段可能与声明指针或引用时的类型不一致。如果不使用指针或引用，而是直接通过对象名访问对象，那么即使对象拥有动态信息（虚函数表），对象的动态信息与静态声明对象时的信息必然一致，就没有必要访问虚函数表；而如果对象不拥有虚函数，就没有虚函数表存放动态信息，也就无法在运行时动态判断指针所指向对象（或引用所绑定对象）的实际类型。

(2) dynamic_cast 的用法

dynamic_cast 是一个“纯”动态操作符，因此，它只能用于指针或引用间的转换。而且，dynamic_cast 运算符所操作的指针（或引用），指针所指的对象（或引用所绑定的对象）必须拥有虚函数成员，否则会出现编译错误。

具体地说，dynamic_cast 可以进行如下的类型转换：在指向基类的指针（引用）与指向派生类的指针（引用）之间进行转换；在多重继承的情况下，在派生类的多个基类之间进行转换（称为交叉转换：crosscast）。

在某些 C++ 编译器中，认为将指向派生类的指针转换为指向基类的指针总是安全的。这时根本没有必要使用 dynamic_cast，就算用了，它也不会“认真工作”。见下面的例子。

```
//// Program 8.6-3 /////
#include <iostream>
using namespace std;
class A{
public:
    int i;
    virtual void show(){
        cout << "class A" << endl;
    }
    A(){i=1;}
};

class B:public A{
public:
    int j;
    void show(){
        cout << "class B" << endl;
    }
    B(){j=2;}
};
```

```

class C: public B{
public:
    int k;
    void show(){
        cout << "class C" << endl;
    }
    C(){k=3;}
};

int main(){
    A a;
    B *pb;
    C *pc;
    pc=static_cast<C*>(&a);
    pb = dynamic_cast<B*>(pc);
    if(pb){
        pb->show();
        cout << pb->j << endl;
    }
    else
        cout << "Conversion failed" << endl;
}
/// End of Program 8.6-3 ///

```

在 VC++ 2005 平台下编译运行，程序的输出结果是：

```

class A
-858993460

```

本来，语句 `pb = dynamic_cast<B*>(pc);` 的目的是检查指针所指的对象是否能安全地转换为 class B 类型，但由于 C++ 编译器认为从派生类指针转换为基类指针总是安全的，所以实际上这里的 `dynamic_cast` 并没有发挥动态检查的作用，最后导致输出的结果没有意义。所以，一定要明确：`dynamic_cast` 用来作向下转换（由基类指针转换为派生类指针）的安全性检查时总是有效的，作向上转换（upcast）时是否有效要看具体的编译器的实现。因此，做向上转换时可考虑采用其他方式，而不使用 `dynamic_cast`。

`dynamic_cast` 在向下转换（downcast）时，会“认真”检查指针所指的对象的实际类型，从而决定转换是否成功。下面是一个例子。

```

/// Program 8.6-4 ///
#include <iostream>
using namespace std;
class A{
public:
    int i;
    virtual void show(){

```

```

        cout << "class A" << endl;
    }
    A(){i=1;}
};

class B:public A{
public:
    int j;
    void show(){
        cout << "class B" << endl;
    }
    B(){j=2;}
};

class C: public B{
public:
    int k;
    void show(){
        cout << "class C" << endl;
    }
    C(){k=3;}
};

int main(){
    A *pa=NULL;
    B b,*pb;
    C *pc;
    pb = dynamic_cast<B*>(pa);
    if(!pb)
        cout << "Result is NULL" << endl;
    pa = &b;
    pb = dynamic_cast<B*>(pa);
    if(pb){
        pb->show();
        cout << pb->j << endl;
    }
    else
        cout << "Conversion failed" << endl;
    pc = dynamic_cast<C*>(pa);
    if(pc){
        pc->show();
        cout << pc->k << endl;
    }
}

```

```

else
    cout << "Conversion failed" << endl;
}

```

//// End of Program 8.6-4 ////

程序的输出结果是：

Result is NULL

class B

2

Conversion failed

由于指针 pa 所指的对象的实际类型是 class B，所以将 pa 转换成 B* 类型时没有问题，而将 pa 转换成 C* 类型时则失败。当指针转换失败时，返回 NULL。如果被转换指针的值为 NULL，则 dynamic_cast 也返回 NULL。指针转换失败与否，是由返回值是否为空来进行判断的。由于 C++ 中不存在空引用，所以引用的转换失败要借助异常来表示。见下面的例子。

//// Program 8.6-5 ////

```

#include <iostream>
using namespace std;
class Other{
    virtual void func(){}
};

class A{
public:
    int i;
    virtual void show(){
        cout << "class A" << endl;
    }
    A(){i=1;}
};

class B:public A{
public:
    int j;
    void show(){
        cout << "class B" << endl;
    }
    B(){j=2;}
};

class C: public B{
public:
    int k;
    void show(){
        cout << "class C" << endl;
    }
};

```

```

    }
    C() {k=3;}
};

int main(){
    A a,*pa;
    C c;
    Other o;
    bool bad;
    //try to set reference from Other& to B&
    try{
        bad=false;
        B &rb=dynamic_cast<B&>(o);
    }
    catch(bad_cast){
        cout << "Reference failed from Other& to B&" << endl;
        bad=true;
    }
    if(!bad){
        B &rb=dynamic_cast<B&>(o);
        rb.show();
    }
    //try to set reference from A& to B&
    pa = &a;
    try{
        bad=false;
        B &rb=dynamic_cast<B&>(*pa);
    }
    catch(bad_cast){
        cout << "Reference failed from A& to B&" << endl;
        bad=true;
    }
    if(!bad){
        B &rb=dynamic_cast<B&>(*pa);
        rb.show();
    }
    //try to set reference from C& to B&
    try{
        bad=false;
        B &rb=dynamic_cast<B&>(c);
    }
}

```



```

catch(bad_cast){
    cout << "Reference failed from C& to B&" << endl;
    bad=true;
}
if(!bad){
    B &rb=dynamic_cast<B&>(c);
    rb.show();
}
}
/// End of Program 8.6-5 ///

```

程序的运行结果是：

```

Reference failed from Other& to B&
Reference failed from A& to B&
class C

```

将 Other 类对象的引用转换为 B 类对象的引用肯定是要失败的，因为它们之间不存在继承关系。如果引用的动态转换过程失败，则会抛出 bad_cast 异常，捕捉该异常就可以知道在程序运行时转换是否成功。

交叉转换 (crosscast) 是在两个“平行”的类对象之间进行。本来它们没有什么关系，从其中的一种转换为另一种是不可行的。但是，如果类 A 和类 B 都是某个派生类 C 的基类，而指针所指的对象本身就是一个类 C 的对象，那么该对象既可以被视为类 A 的对象，也可以被视为类 B 的对象，类型 A* (A&) 和 B* (B&) 之间的转换就成为可能。见下面的例子。

/// Program 8.6-6 ///

```

#include <iostream>
using namespace std;
class A{
public:
    int num;
    A(){num=4;}
    virtual void funcA(){}
};
class B{
public:
    int num;
    B(){num=5;}
    virtual void funcB(){}
};
class C:public A,public B{};
int main(){
    C c;
    A *pa;

```

```

B *pb;
pa = &c;
cout << pa->num << endl;
pb = dynamic_cast<B*>(pa);
cout << "pa= " << pa << endl;
if(pb){
    cout << "pb= " << pb << endl;
    cout << "Conversion succeeded" << endl;
    cout << pb->num << endl;
}
else
    cout << "Conversion failed" << endl;
}
/// End of Program 8.6-6 ///

```

程序的执行结果是：

```

4
pa= 0012FF54
pb= 0012FF5C
Conversion succeeded
5

```

可以看到，pa 转换成 pb 之后，其值由 0012FF54 变成了 0012FF5C。也就是说，在类 C 的对象中，基类 A 的成员和基类 B 的成员所占的位置（距对象首地址的偏移量）是不同的，类 B 的成员要靠后一些，所以将 A*转换成 B*时，要对指针的位置进行调整。如果将程序中的 dynamic_cast 替换成 static_cast，则程序无法通过编译，因为这时编译程序会认为类 A 和类 B 是两个“无关”的类。

8.7 虚调用的几种具体情形

虚调用是相对于实调用而言的，它的本质是动态联编。在发生函数调用的时候，如果函数的入口地址是在编译阶段静态确定的，就是实调用。反之，函数的入口地址要在运行时通过查询虚函数表的方式获得，就是虚调用。

虚调用不能简单地理解成“对虚函数的调用”，因为对虚函数的调用很有可能是实调用。考察下面的程序。

```

/// Program 8.7-1 ///
#include <iostream>
using namespace std;
class A{
public:
    virtual void show(){
        cout << "in A" << endl;
    }
}
```



```

    }
};

class B:public A{
public:
    void show(){
        cout << "in B" << endl;
    }
};

void func(A a){
    a.show();
}

int main(){
    B b;
    func(b);
}

/// End of Program 8.7-1 ///

```

此程序的运行结果是：in A。在函数 func() 中，虽然在 class A 中函数 show() 被定义为虚函数，但是由于 a 是类 A 的一个实例，而不是指向类 A 对象的指针或引用，所以函数调用 a.show() 是实调用，函数的入口地址是在编译阶段静态决定的。函数调用 func(b) 的执行过程是这样的：先由对象 b 通过调用类 A 的复制构造函数，产生一个类 A 的对象作为 a 进入函数 func() 的函数体。在函数体内，a 是一个“纯粹”的类 A 对象，与类型 B 毫无关系，所以 a.show() 是实调用。

在构造函数中调用虚函数，对虚函数的调用实际上是实调用。这是虚函数被“实调用”的另一个例子。由于从概念上说，在一个对象的构造函数运行完毕之前，这个对象还没有完全“诞生”，所以在构造函数中调用虚函数，实际上都是实调用。见下面的例子。

/// Program 8.7-2 ///

```

#include <iostream>
using namespace std;
class A{
public:
    virtual void show(){
        cout << "in A" << endl;
    }
    A(){show();}
};

class B:public A{
public:
    void show(){
        cout << "in B" << endl;
    }
}

```

```
};

int main(){
    A a;
    B b;
}

/// End of Program 8.7-2 ///
```

程序的执行结果是：

```
in A
in A
```

在构造类 B 的对象 b 时，会先调用基类 A 的构造函数，如果在构造函数中对 show() 的调用是虚调用，那么应该打印出“in B”。而实际的情况是，由于在执行类 A 的构造函数的时候，类 B 的构造函数还未执行完毕，所以对象 b 还没有构造好，因此无法完成对 b 的成员函数的调用。所以，在构造函数中对虚函数的调用实际上都是实调用。一般情况下，应该避免在构造函数中调用虚函数，这一点是完全可以做到的。

设立虚函数的初衷，是想在设计基类的时候，就对该基类的派生类实施一定程度的控制。笼统地说，就是“通过基类访问派生类成员”。因此，虚调用最常见的形式是：通过指向基类对象的指针访问派生类对象的虚函数，或通过基类对象的引用调用派生类对象的虚函数。这种情形在绝大多数教科书中都有详细介绍。不过，由于虚调用是通过查询虚函数表来实现的，而拥有虚函数的对象都可以访问到所属类的虚函数表，所以，尽管不常见，但的确可以实现这种形式的虚调用：通过指向派生类对象的指针调用基类对象的虚函数，或通过派生类对象的引用调用基类对象的虚函数。见下面的程序。

```
/// Program 8.7-3 /////
#include <iostream>
using namespace std;
class A{
public:
    virtual void show(){cout<<"in A"<<endl;}
};
class B:public A{
public:
    void show(){cout<<"in B"<<endl;}
}
int main(){
    A a;
    B &rB = static_cast<B&>(a);
    rB.show();
}

/// End of Program 8.7-3 ///
```

程序的执行结果是：in A。通过派生类对象的引用 rB 实现了对基类中虚函数 show() 的调



用。如果在 class A 的定义中，将函数 show()前面的关键字 virtual 去掉，那么程序的执行结果就是：in B。tB.show()就变成了实调用。

那么，实现虚调用是否一定要显式借助于指针或引用才能实现呢？答案是：在实际应用中，绝大多数的虚调用的确是显式借助于指针或引用实现的。但是，这并不说明其他方式就一定不能实现虚调用。以下就是一个例子。

```
/// Program 8.7-4 ///
#include <iostream>
using namespace std;
class A{
public:
    virtual void show(){cout<<"in A "<<endl;}
    void callfunc(){show();}
};
class B:public A{
public:
    void show(){cout<<"in B "<<endl;}
};
int main(){
    B b;
    b.callfunc();
}
/// End of Program 8.7-4 ///
```

程序的执行结果是：in B。在这个程序中，看不到一个指针或引用，却发生了虚调用。函数调用 b.callfunc()执行的实际上是 A::callfunc()，如果在 class A 中去掉函数 show()前面的关键字 virtual，那么程序的输出结果应该是：in A。也就是说，在函数 callfunc()中，函数调用 show(); 是一个虚调用，它是在运行时才决定使用派生类中的虚函数还是使用基类中的虚函数的。实际上，如果一个虚函数 funca 被该类的另一个成员函数 funcb 调用，那么该调用一定是虚调用。原因是：函数 funcb 在被调用时，调用它的对象的实际类型是不确定的，因此导致对 funca 的调用在编译阶段不能静态确定，所以一定要采用虚调用的形式。

8.8 不要在构造函数或析构函数中调用虚函数

虽然可以对虚函数进行实调用，但程序员编写虚函数的本意应该是实现动态联编。在构造函数中调用虚函数，函数的入口地址是在编译时静态确定的，并未实现虚调用，具体的例子参见 Program 8.7-2。那么，为什么在构造函数中调用虚函数，实际上并未发生动态联编呢？原因有两个：

在概念上，构造函数的工作是为对象进行初始化。在构造函数完成之前，被构造的对象应该被认为“未完全生成”。当创建某个派生类的对象时，如果在它的基类的构造函数中调用虚函数，那么此时派生类的构造函数并未执行，所调用的函数可能操作还没有被初始化的成员，这将导致灾难的发生。



第二个原因是，即使想在构造函数中实现动态联编，在实现上也会遇到困难。这涉及到对象虚指针（vptr）的建立问题。在 Visual C++ 中，包含虚函数的类对象的虚指针被安排在对象的起始地址处，并且虚函数表（vtable）的地址是由构造函数写入虚指针的。所以，一个类的构造函数在执行时，并不能保证该函数所能访问到的虚指针就是当前被构造对象最后所拥有的虚指针，因此无法完成动态联编。考察下面的程序，可以了解一个派生类对象的虚指针的建立过程。

```
//// Program 8.8-1 ////
#include <iostream>
using namespace std;
class A{
    int num;
public:
    virtual void show(){
        cout << "In A" << endl;
    };
};
class B:public A{
public:
    void show(){
        cout << "In B" << endl;
    }
};
int main(){
    B b;
    A *p=&b;
    p->show();
}
//// End of Program 8.8-1 ////
```

在这个程序中，基类 A 和派生类 B 都包含虚函数。建立派生类对象 b 时，该对象起始地址处的虚指针（vptr）的值不是一次性决定的。创建对象 b 的语句 B b; 所对应的汇编代码是：

```
lea    ecx, DWORD PTR _b$[ebp]
call   ??0B@@QAE@XZ
即先把 b 的首地址送入寄存器 ecx，然后调用类 B 的构造函数。
类 B 的构造函数的汇编代码是：
??0B@@QAE@XZ PROC           ; B::B, COMDAT
:_this$ = ecx
    push   ebp
    mov    ebp, esp
    sub    esp, 204          ; 000000ccH
    push   ebx
```



```

push    esi
push    edi
push    ecx
lea    edi, DWORD PTR [ebp-204]
mov    ecx, 51          ; 00000033H
mov    eax, -858993460   ; ccccccccH
rep    stosd
pop    ecx
mov    DWORD PTR _this$[ebp], ecx
mov    ecx, DWORD PTR _this$[ebp]
call    ??0A@@@QAE@XZ
mov    eax, DWORD PTR _this$[ebp]
mov    DWORD PTR [eax], OFFSET ??_7B@@@6B@
mov    eax, DWORD PTR _this$[ebp]
pop    edi
pop    esi
pop    ebx
add    esp, 204          ; 000000ccH
cmp    ebp, esp
call    __RTC_CheckEsp
mov    esp, ebp
pop    ebp
ret    0
??0B@@@QAE@XZ ENDP      ; B::B

```

其中，这两条语句：`mov ecx, DWORD PTR _this$[ebp]`和`call ??0A@@@QAE@XZ`，是在类 B 的构造函数内部调用其基类 A 的构造函数，传进 A::A() 的 this 指针仍然是派生类对象 b 的首地址。在类 A 的构造函数调用完成之后，有这样两条语句：

```

mov    eax, DWORD PTR _this$[ebp]
mov    DWORD PTR [eax], OFFSET ??_7B@@@6B@

```

它们的功能就是将类 B 的虚函数表的地址写入当前对象 (b) 的首地址，也就是为 b 所包含的虚指针赋值。实际上，在类 A 的构造函数也有同样的过程。考察类 A 的构造函数的汇编代码，会发现有这样两条语句：

```

mov    eax, DWORD PTR _this$[ebp]
mov    DWORD PTR [eax], OFFSET ??_7A@@@6B@

```

它们的功能是将类 A 的虚函数表地址写入 this 指针指向的内存处。所以，在执行基类的构造函数 A::A() 时，对象 b 的虚指针实际上指向类 A 的虚函数表，只有当类 B 的构造函数执行之后，对象 b 的虚指针才真正指向类 B 的虚函数表。所以，在类 A 的构造函数中调用虚函数，由于无法访问到其派生类的虚函数表，所以只能实现静态联编。

同样地，在析构函数中调用虚函数，函数的入口地址也是在编译时静态决定的。也就是说，实现的是实调用而非虚调用。请考察下面的例子。



```
//// Program 8.8-2 ////  
#include <iostream>  
using namespace std;  
class A{  
public:  
    virtual void show(){  
        cout<<"in A"<<endl;  
    }  
    virtual ~A(){show();}  
};  
class B:public A{  
public:  
    void show(){  
        cout<<"in B"<<endl;  
    }  
};  
int main(){  
    A a;  
    B b;  
}  
/// End of Program 8.8-2 ///
```

程序的执行结果是：

in A

in A

在类 B 的对象 b 退出作用域时，会先调用类 B 的析构函数，然后调用类 A 的析构函数。在析构函数~A()中，调用了虚函数 show()。如果在析构函数中对 show()的调用是虚调用，那么应该打印出“in B”。而实际的情况是打印出“in A”，这说明并没有发生虚调用。从概念上说，析构函数是用来“销毁”一个对象的。在销毁一个对象时，先调用该对象所属类的析构函数，然后再调用其基类的析构函数。所以，在调用基类的析构函数时，派生类对象的“善后”工作已经完成了，这个时候再调用在派生类中定义的函数版本已经没有意义了，因为派生类对象的数据已经“失效”。所以，不要在构造函数和析构函数中调用虚函数。如果一定要这样做，至少程序员必须清楚，这时对虚函数的调用其实是实调用。

8.9 虚函数可以是私有的吗

虚函数一般被声明为公有的（public）。这样实现虚函数调用会比较方便。但 C++语言并没有要求虚函数必须是公有的。将虚函数设置成私有的（private）和受保护的（protected）并不妨碍虚函数之间的覆盖（override）和虚函数调用。见下面的例子。

```
/// Program 8.9-1 ////  
#include <iostream>
```



```

using namespace std;
class A{
    virtual void f1(){
        cout << "A's f1()" << endl;
    }
public:
    void func(){
        f1();
    }
};

class B:public A{
public:
    void f1(){
        cout << "B's f1()" << endl;
    }
};

int main(){
    A a;
    B b;
    A *pA;
    B *pB;
    pA = &a;
    pA->func();
    pA = &b;
    pA->func();
    pB = &b;
    pB->f1();
}

/// end of program 8.9-1 ///

```

在类 A 中，函数 f1() 被声明为私有的，这就意味着在类 A 的外部，不能够直接调用函数 f1()。如在 main() 函数中，pA->f1() 这样的调用就是非法的。但是，不能“直接”调用并不意味着“不能”调用，否则这个函数就没有存在的必要了。在函数 A::func() 中实现了对 A::f1() 的调用，而且是真正的动态联编。所以，程序的输出结果是：

```

A's f1()
B's f1()
B's f1()

```

尽管在类 A 中，虚函数 f1() 是私有的，但在派生类 B 中，仍然实现了对 f1() 的改写，这是通过动态联编技术实现的。所以，虚函数是否声明为公有，对虚函数表（vtable）的生成过程不造成任何影响，它只是通知编译器直接通过指针（或引用）调用虚函数是否合法。在上面的程序中，如果将 B::f1() 声明为私有，语句 pB->f1(); 将不能通过编译，而其他语句都能够



通过编译，执行结果与原来完全相同。

8.10 动态联编是怎样实现的

所谓动态联编，是指被调函数的入口地址是在运行时、而不是在编译时决定的。C++语言利用动态联编来完成虚函数调用。C++标准并没有规定如何实现动态联编，但大多数的C++编译器都是通过虚指针(vptr)和虚函数表(vtable)来实现动态联编的。基本的思路是：为每一个包含虚函数的类建立一个虚函数表，虚函数表的各个表项存放的是各虚函数在内存中的入口地址；在该类的每个对象中设置一个指向虚函数表的指针，在调用虚函数时，先利用虚指针找到虚函数表，确定虚函数的入口地址在表中的位置，获取入口地址完成调用。我们将从以下几个方面来考察动态联编的实现细节。

(1) 虚指针(vptr)放在哪里？

虚指针是作为对象的一部分存放在对象的空间中的。一个类只有一个虚函数表，因此该类的所有对象中的虚指针都指向同一个地方。在不同的编译器中，虚指针在对象中的位置是不同的。两种典型的做法是：在Visual C++中，虚指针位于对象的起始位置；在GNU C++中，虚指针位于对象的尾部而不是头部。可通过下面的程序考察在Visual C++中，虚指针在对象中的位置。

```
/// Program 8.10-1 ///
#include <iostream>
#include <iomanip>
using std::cout;
using std::endl;
using std::hex;
int globalv=0;
class NoVirtual{
    int i;
public:
    void func(){
        cout << "No Virtual Function" << endl;
    }
    NoVirtual(){
        i = ++globalv;
    }
};
class HaveVirtual:public NoVirtual{
public:
    virtual void func(){
        cout << "Virtual Function" << endl;
    }
};
```

```

int main(){
    NoVirtual n1,n2;
    HaveVirtual h1,h2;
    unsigned long *p;
    cout << "sizeof(NoVirtual) is " << sizeof(NoVirtual) << endl;
    cout << "sizeof(HaveVirtual) is " << sizeof(HaveVirtual) << endl;
    p = reinterpret_cast<unsigned long*>(&n1);
    cout << "the content of first 4 bytes of n1 is " << p[0] << endl;
    p = reinterpret_cast<unsigned long*>(&n2);
    cout << "the content of first 4 bytes of n2 is " << p[0] << endl;
    p = reinterpret_cast<unsigned long*>(&h1);
    cout << "the content of first 4 bytes of h1 is 0x" << hex << p[0] << endl;
    p = reinterpret_cast<unsigned long*>(&h2);
    cout << "the content of first 4 bytes of h2 is 0x" << hex << p[0] << endl;
}

```

/// End of Program 8.10-1 ///

以上程序在 Visual Studio 2005 下通过编译，执行后的输出结果是：

```

sizeof(NoVirtual) is 4
sizeof(HaveVirtual) is 8
the content of first 4 bytes of n1 is 1
the content of first 4 bytes of n2 is 2
the content of first 4 bytes of h1 is 0x417800
the content of first 4 bytes of h2 is 0x417800

```

从程序的输出结果中，可以清楚地看到虚指针对类对象大小的影响。类 NoVirtual 不包含虚函数，因此类 NoVirtual 的对象中只包含数据成员 i，所以 sizeof(NoVirtual) 为 4。类 HaveVirtual 包含虚函数，因此类 HaveVirtual 的对象不仅要包含数据成员 i，还要包含一个指向虚函数表的指针（大小为 4），所以 sizeof(HaveVirtual) 为 8。

如果虚指针不是放在对象的头部，那么对象 h1 和对象 h2 的头 4 个字节（代表整型成员变量 i）的值应该是 3 和 4。而程序打印出来的结果显示，类 HaveVirtual 的两个对象对象 h1 和对象 h2 的头 4 个字节的内容相同，都等于 0x417800，其实这个值就是类 HaveVirtual 的虚函数表所在的地址。

(2) 虚函数表 (vtable) 的内部结构

虚函数表是为拥有虚函数的类准备的。虚函数表中存放的是类的各个虚函数的入口地址。那么，虚函数的入口地址是按照什么顺序存放在虚函数表中呢？不同的类（比如说父类和子类）是否可以共享同一张虚函数表呢？看下面的程序。

```

/// Program 8.10-2 ///
#include <iostream>
using namespace std;
#define ShowFuncAddress(function) _asm{\n    mov eax,function}\n

```

```

    _asm{ mov p,eax}\n
    cout << "Address of "#function" is 0x" << p << endl;\n
void ShowVTableContent(char *ClassName,void *pObj,int index){\n
    unsigned long *pAddr;\n
    pAddr = reinterpret_cast<unsigned long*>(pObj);\n
    pAddr = (unsigned long*)*pAddr;          //获取虚函数表地址\n
    cout << "the content of " << ClassName << "'s vtable[" << index << "];\n
    cout << " is 0x" << (void*)pAddr[index] << endl;\n
}\n
class Base{\n
    int i;\n
public:\n
    virtual void f1(){\n
        cout << "Base's f1()" << endl;\n
    }\n
    virtual void f2(){\n
        cout << "Base's f2()" << endl;\n
    }\n
    virtual void f3(){\n
        cout << "Base's f3()" << endl;\n
    }\n
};\n
class Derived: public Base{\n
public:\n
    virtual void f4(){\n
        cout << "Derived's f4()" << endl;\n
    }\n
    void f3(){\n
        cout << "Derived's f3()" << endl;\n
    }\n
    void f1(){\n
        cout << "Derived's f1()" << endl;\n
    }\n
};\n
int main(){\n
    Base b1;\n
    Derived d1;\n
    void *p;\n
    unsigned long *pAddr;\n
    pAddr = reinterpret_cast<unsigned long*>(&b1);\n
}

```



```

cout << "Address of vtable of Base is 0x" << (void*)*pAddr << endl;
pAddr = reinterpret_cast<unsigned long*>(&d1);
cout << "Address of vtable of Derived is 0x" << (void*)*pAddr << endl;
ShowFuncAddress(Base::f1);
ShowVTableContent("Base",&b1,0);
ShowFuncAddress(Base::f2);
ShowVTableContent("Base",&b1,1);
ShowFuncAddress(Base::f3);
ShowVTableContent("Base",&b1,2);
ShowFuncAddress(Derived::f1);
ShowVTableContent("Derived",&d1,0);
ShowFuncAddress(Derived::f2);
ShowVTableContent("Derived",&d1,1);
ShowFuncAddress(Derived::f3);
ShowVTableContent("Derived",&d1,2);
ShowFuncAddress(Derived::f4);
ShowVTableContent("Derived",&d1,3);
}
/// End of Program 8.10-2 /////

```

此程序的输出结果是：

```

Address of vtable of Base is 0x00417C1C
Address of vtable of Derived is 0x00417C2C
Address of Base::f1 is 0x004110CD
the content of Base's vtable[0] is 0x004110CD
Address of Base::f2 is 0x0041111D
the content of Base's vtable[1] is 0x0041111D
Address of Base::f3 is 0x0041103C
the content of Base's vtable[2] is 0x0041103C
Address of Derived::f1 is 0x0041110E
the content of Derived's vtable[0] is 0x0041110E
Address of Derived::f2 is 0x0041111D
the content of Derived's vtable[1] is 0x0041111D
Address of Derived::f3 is 0x004112A3
the content of Derived's vtable[2] is 0x004112A3
Address of Derived::f4 is 0x00411271
the content of Derived's vtable[3] is 0x00411271

```

在程序中使用了宏ShowFuncAddress，利用内联汇编来获取类的非静态成员函数的入口地址。这是一个带参数的宏，并且对宏的参数做了一些特殊处理。关于复杂的宏的定义请参见1.16节。

从程序的输出结果可以看出，类Base的虚函数表的地址是0x00417C1C，而类Derived的虚

函数表的地址是0x00417C2C。尽管类Base是类Derived的父类，但它们却各自使用不同的虚函数表。可见，所有的类都不会和其他的类共享同一张虚函数表。

对于任意的包含虚函数的类，将虚函数的入口地址写入虚函数表，按照如下的步骤进行：

①确定当前类所包含的虚函数的个数。一个类的虚函数有两个来源，一是继承自父类（在当前类中可能被改写），其他的是在当前类中新声明的虚函数。

②为所有的虚函数排序。继承自父类的所有虚函数，排在当前类新声明的虚函数之前。新声明的虚函数，按照在当前类中声明的顺序排列。

③确定虚函数的入口地址。继承自父类的虚函数，如果在当前类中被改写，则虚函数的入口地址是改写之后的函数的地址，否则保留父类中的虚函数的入口地址。新声明的虚函数，其入口地址就是在当前类中的函数的入口地址。

④将所有虚函数的入口地址按照排定的次序依次写入虚函数表。

考察Program 8.5-2，尽管在类Derived中，函数f4()在函数f3()之前定义，但由于f3()是继承自父类的虚函数，而f4()是新声明的虚函数，所以f3()的次序排在f4()之前。另外，尽管函数f3()定义在函数f1()之前，由于它们都继承自父类，而在父类中f1()是排在f3()之前的，所以在类Derived中，f1()仍然排在f3()之前。再者，在类Derived中，并没有其父类的虚函数f2()的改写版本，所以函数Derived::f2的入口地址仍然保留其父类的虚函数Base::f2的入口地址（在程序的输出中是0x0041111D）。

(3) 虚函数表放在哪里？

虚函数表放在应用程序的常量区。将Program 8.5-2编译之后生成汇编语言代码，从中可以发现这样两段内容：

CONST SEGMENT

??_TBase@@6B@ DD FLAT:??_R4Base@@6B@ ; Base::'vftable'

DD FLAT:?f1@Base@@UAEXXZ

DD FLAT:?f2@Base@@UAEXXZ

DD FLAT:?f3@Base@@UAEXXZ

; Function compile flags: /Odtp /RTCsu /ZI

CONST ENDS

CONST SEGMENT

??_TDerived@@6B@ DD FLAT:??_R4Derived@@6B@ ; Derived::'vftable'

DD FLAT:?f1@Derived@@UAEXXZ

DD FLAT:?f2@Base@@UAEXXZ

DD FLAT:?f3@Derived@@UAEXXZ

DD FLAT:?f4@Derived@@UAEXXZ

; Function compile flags: /Odtp /RTCsu /ZI

CONST ENDS

这是两个常量段，其中分别存放了Base类的虚函数表和Derived类的虚函数表。从中可以清楚地了解到，虚函数表中的每一项代表了一个函数的入口地址，类型是Double Word。类中每个虚函数的入口地址在虚函数表中的排放顺序，也可以从相应的标识符中看出。如FLAT:?f1@Derived@@UAEXXZ代表的就是Derived::f1的入口地址。

(4) 通过访问虚函数表手动调用虚函数

既然知道了虚函数表的位置和结构，那么就可以通过访问虚函数表，手动调用虚函数。虽然在利用C++编写应用程序时完全没有必要这样做，但如果想了解动态联编的实现机理，编写这样的程序却很有帮助。下面就是一个例子。

```
/// Program 8.10-3 ///
#include <iostream>
using namespace std;
typedef void (*fun)();
void ExecuteVirtualFunc(void *pObj,int index){
    fun p;
    unsigned long *pAddr;
    pAddr = reinterpret_cast<unsigned long*>(pObj);
    pAddr = (unsigned long*)*pAddr;      //获取虚函数表地址
    p = (fun)pAddr[index];              //获取虚函数入口地址
    _asm mov ecx,pObj
    p();                                //实施函数调用
}
class Base{
    int i;
public:
    Base(){i=0;}
    virtual void f1(){
        cout << "Base's f1()" << endl;
    }
    virtual void f2(){
        cout << "Base's f2(),i=" << i << endl;
    }
    virtual void f3(){
        cout << "Base's f3()" << endl;
    }
};
class Derived: public Base{
    int j;
public:
    Derived(){j=1;}
    virtual void f4(){
        cout << "Derived's f4(),j=" << j << endl;
    }
    void f3(){
        cout << "Derived's f3()" << endl;
    }
}
```

```

void f1(){
    cout << "Derived's f1()" << endl;
}
};

int main(){
    Base b1;
    Derived d1;
    ExecuteVirtualFunc(&b1,1);
    ExecuteVirtualFunc(&d1,3);
}

```

/// End of Program 8.10-3 ///

执行ExecuteVirtualFunc(&b1,1);就是调用对象b1的第二个虚函数(b1.f2())，执行ExecuteVirtualFunc(&d1,3);就是调用对象b1的第四个虚函数(d1.f4())。程序的输出结果是：

Base's f2(),i=0

Derived's f4(),j=1

结果表明成功地对不同类对象上的不同虚函数实现了调用。这些调用是通过访问每个对象的虚函数表实现的。由于在调用类对象的非静态成员函数时，必须同时给出对象的首地址，所以在程序中使用了内联汇编代码`_asm mov ecx,pObj`来达到这个目的。在Visual C++中，在调用类的非静态成员函数之前，对象的首地址都是送往寄存器ecx的。

8.11 !操作符重载

“!”是一个一元操作符，它通常作用于布尔量，表示逻辑取反。当在某个类中对“!”操作符进行重载时，通常表明该类的对象出现了“异常”状况。例如，可以对!操作符进行重载，以表明用new生成的对象是否构造成功。考察下面的程序。

```

/// Program 8.11-1 ///
#include <iostream>
using namespace std;
class A{
    int m;
    char *str1;
    char *str2;
public:
    A(int n1,int n2,int n3){
        str1=new char[n1];
        str2=new char[n2];
        m=n3;
    }
    ~A(){
        delete str1;
    }
}

```



```

    delete str2;
}
void show(){
    cout << m << endl;
}
};

int main(){
    A *p=new A(4,5,6);
    p->show();
    delete p;
}

/// End of Program 8.11-1 ///

```

虽然在大多数情况下，程序可以正常编译运行，但此程序是存在缺陷的。因为用new操作在堆上申请空间，有可能会失败。因此，更为稳健的做法是对所有的new操作结果进行测试，以确保对象被成功地创建。在上面的程序中，一共出现了三处new操作，每一处操作失败都将导致对象无法正确创建，因此要对这些new操作的结果进行检验。修改后的程序如下。

```

/// Program 8.11-2 ///
#include <iostream>
using namespace std;
class A{
    int m;
    char *str1;
    char *str2;
public:
    A(int n1,int n2,int n3){
        str1=new char[n1];
        str2=new char[n2];
        m=n3;
    }
    ~A(){
        delete str1;
        delete str2;
    }
    void show(){
        cout << m << endl;
    }
    bool ObjOK(){
        if(str1 && str2) return true;
        return false;
    }
}

```



```
};  
int main(){  
    A *p=new A(4,5,6);  
    if(p && p->ObjOK())  
        p->show();  
    delete p;  
}
```

//// End of Program 8.11-2 ////

经过修改的程序考察了所有new操作可能出现的失败，因而是一个健壮的程序。不过，像p->ObjOK()这样的函数调用带有浓厚的程序员的个人风格，通过重载该类的operator!操作符，就可以将对象的检测工作统一表示。例如下面的程序。

```
//// Program 8.11-3 ////  
#include <iostream>  
using namespace std;  
class A{  
    int m;  
    char *str1;  
    char *str2;  
public:  
    A(int n1,int n2,int n3){  
        str1=new char[n1];  
        str2=new char[n2];  
        m=n3;  
    }  
    ~A(){  
        delete str1;  
        delete str2;  
    }  
    void show(){  
        cout << m << endl;  
    }  
    // friend bool operator!(const A &a);  
    bool operator!(){  
        if(str1 && str2) return false;  
        return true;  
    }  
};  
/*bool operator!(const A &a){  
    if(a.str1 && a.str2) return false;
```





```

        return true;
    }*/
int main(){
    A *p=new A(4,5,6);
    if(!p || !(*p))
        cerr << "new operation failed" << endl;
    else
        p->show();
    delete p;
}
/// End of Program 8.11-3 ///

```

通过对operator !操作符进行重载，可以判断一个A类对象的数据成员是否存在问题（空间是否成功分配）。这样，普通的指针、文件指针、对象都可以统一地用operator !来判断是否出现异常，便于程序的编写与排错，也便于程序员之间的交流与代码复用。

对operator !进行重载可以用两种方式实现。程序中采用的是将operator !()作为类A的成员函数。还有一种方式是将operator !()作为类A的友元函数，具体实现方式见代码中被注释的部分。

8.12 []操作符重载

“[]”是C++中的下标运算符，对于数组或指针来说，下标运算符的语义是确定的，不能够进行重载。C++语言规定：“[]”只能作为类的成员函数进行重载。因此，如果看到一个运算结果不是数组（指针）的表达式后跟“[]”运算符，一定是对“[]”进行了重载。下面是一个例子。

```

/// Program 8.12-1 ///
#include <iostream>
using namespace std;
class A{
    int num[3];
public:
    A();
    Int& operator[](int);
};
A::A(){
    num[0]=1;
    num[1]=2;
    num[2]=3;
}
int& A::operator [](int sub){

```

```
if(sub<0 || sub>2)
    throw sub;
else
    return num[sub];
}
int main(){
    A a;
    A *p=&a;
    try{
        for(int i=0;i<=4;i++)
            cout << p[0][i] << endl;
    }
    catch(int sub){
        cout << "Subscript out of range: " << sub << endl;
    }
}
/// End of Program 8.12-1 ///
```

程序的运行结果是：

```
1
2
3
Subscript out of range: 4
```

在表达式`p[0][i]`中，第一个`[]`运算符是下标运算符的原意，`p[0]`相当于`*p`，代表指针所指的对象，而第二个`[]`运算符使用的是重载后的语义，表示从对象内部的成员数组中取数据。当下标为4时，程序会抛出异常，这是对下标越界进行判断的结果。

对`[]`运算符进行重载，一般会将操作符函数的返回值类型定义为引用类型。这是与传统的数组下标运算保持一致的做法。通过下标找到特定的数组元素后，应该既可以对它进行读操作，也能对它进行写操作。而这一点只能通过返回引用才能做到。

一般说来，下标的数据类型是整型，这是C/C++语言中下标的基本用法。但从语法的角度来说，对`operator []`操作符函数进行重载，并没有限定下标的数据类型，这样在某些特殊情况下，可以采用特殊数据类型的下标。下面是一个具体的例子。

```
/// Program 8.12-2 ///
#include <iostream>
#include <string>
using namespace std;
class Employee{
    string name;
    string position;
public:
    Employee(string,string);
```



```

        string &operator[](string);
};

Employee::Employee(string n,string p){
    name=n;
    position=p;
}

string &Employee::operator[](string s){
    if(s=="name")
        return name;
    else if(s=="position")
        return position;
    throw s;
}

int main(){
    Employee a("张三","职员"),b("李四","经理");
    try{
        cout << "A's name is: " << a["name"] << endl;
        cout << "A's position is: " << a["position"] << endl;
        cout << "B's name is: " << b["name"] << endl;
        cout << "B's position is: " << b["position"] << endl;
        cout << "B's address is: " << b["address"] << endl;
    }
    catch(string &s){
        cout << "Unknown attribute: " << s << endl;
    }
}
/// End of Program 8.12-2 ///

```

程序的运行结果是：

```

A's name is: 张三
A's position is: 职员
B's name is: 李四
B's position is: 经理
Unknown attribute: address

```

在这个例子中，下标的数据类型是字符串类string，直接在下标中指明对象的成员的名字，然后对其进行读写操作，是一种较为新颖的做法，也具有较好的可读性。特别是，下标的值可以由用户以字符串的形式从程序外部输入，增强了程序的灵活性。

最后，总结一下重载operator []要注意的几项内容：

①操作符函数operator []()只接受一个参数，没有参数或者多于一个参数都会造成编译错误。参数的类型可以是包括int在内的任意类型。

②操作符函数operator []()的返回值类型应该是引用类型，这是为了与传统的数组下标运

算的语义保持一致。

③当a为一个数组时， $a[i]$ 和*i[a]*都是合法的，都表示数组a的下标为i的元素。如果a是一个重载了operator []的类的对象，那么*i[a]*的表示方法将引发编译错误。所以，为了显式地表明a是一个数组，在使用下标运算时可采用*i[a]*的表示形式。

8.13 *操作符重载

“*”是一个一元操作符，它作用于指针，表示取指针所指单元的内容。当在某个类中对*操作符进行重载时，是将该类的对象当做一个指针看待，而用*操作符提取指针所指的内容。考察下面的程序。

```
//// Program 8.13-1 /////
#include <iostream>
using namespace std;
template <typename T> class Data_Container{
    T *p;
public:
    Data_Container(T *inp){
        p = inp;
    }
    ~Data_Container(){
        delete p;
    }
    template <typename T> friend T operator*(const Data_Container<T>&);
};
template <typename T> T operator*(const Data_Container<T> &d){
    return *(d.p);
}
int main(){
    Data_Container<int> IntData(new int(5));
    Data_Container<double> DoubleData(new double(7.8));
    cout << *IntData << endl;
    cout << *DoubleData << endl;
}
/// End of Program 8.13-1 /////
此程序的输出结果是:
```

5
7.8

在阅读上面的程序时要注意以下几个要点：

①*操作符既可以友元函数的形式重载，也可以成员函数的形式重载。如果是后者，应在类体中这样定义*操作符号函数：

```
T operator*(){
    return *p;
}
```

可见这样定义的操作符函数更加简洁。一般情况下，重载*操作符都是以成员函数的方式进行的。

②一般来说，对*操作符进行重载的类都含有一个指针，*操作符通过类对象取数据，实际上就是从该指针所指的单元取数据。如上例中的p。

③为了防止内存泄露，应该妥善处理new和delete运算。如果在对象的构造函数中使用了new运算申请空间，则应该在对象的析构函数中释放空间。反之，将指针所指空间的申请和释放的工作放到外部去处理。

8.14 赋值操作符重载

赋值操作是一个使用频率最高的操作之一，通常情况下它的意义十分明确：就是将两个同类型的变量的值从一端（右端）传到另一端（左端）。但在两种情况下，需要对赋值操作符进行重载：一是赋值号两边的表达式类型不一样（且无法进行转换），一是需要进行“深拷贝”。

首先要注意的是，赋值操作符只能通过类的成员函数的形式重载。这就说明了，如果要将用户自定义类型的值传递给基本数据类型的变量，只能使用类型转换机制，而不能利用重载来实现。

当赋值号两边的表达式类型不一致的时候，可能需要对赋值操作符进行重载。但是，如果可以成功地进行类型转换，那么通常不需要对赋值操作符进行重载。见下面的例子。

```
/// Program 8.14-1 ///
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){num=0;}
    A(int i){num=i;}
    void show(){
        cout << num << endl;
    }
};
int main(){
    A a=5;           //赋值符号两边的数据类型不一样，这里表示创建新对象
    a.show();
    A a1;
    a1=1;           //赋值符号两边的数据类型不一样，这是真正的赋值运算
    a1.show();
}
```

//// End of Program 8.14-1 ////

程序的输出结果是：

5

1

在语句A a=5中，虽然用到了“=”，但它的语义是构造一个类A的对象a，它等价于语句A a(5)，所以该语句与赋值无关。而语句a1=1是一个真正的赋值语句，变量a1的类型是A，而常量1的类型是int，由于可以通过类A的构造函数A(int)将类型int转换成类型A（实际上是以int为参数构造了一个类A的临时对象），然后再完成赋值操作，所以不必再对赋值操作符进行重载。

当然，在定义了可以用于类型转换的构造函数之后，仍然可以重载赋值操作符。这在语法上是没有问题的。考察下面这个例子。

//// Program 8.14-2 ////

```
#include <iostream>
using namespace std;
class A{
    int num;
public:
    A(){num=0;}
    A(int i){num=i;}
    void show(){
        cout << num << endl;
    }
    A& operator=(int i){
        cout << "Parameter is " << i << endl;
        return *this;
    }
};
```

int main()

A a=5;

a.show();

A a1;

a1=1;

a1.show();

}

//// End of Program 8.14-2 ////

程序的运行结果是：

5

Parameter is 1

0

虽然程序的编译、运行都没有问题，但是A a=5;和a1=1;这两条语句具有完全不同的含义，



在逻辑上容易给程序员造成混乱。这一点要引起注意。

深拷贝是要对赋值操作符进行重载的又一个因素。那么什么是深拷贝呢？简单地说，深拷贝就是在把一个类对象a拷贝到另一个对象b中去时，如果对象a中包含非悬挂指针（有关悬挂指针的概念请参阅5.7节），那么要将a的指针所指区域的内容拷贝到b的相应指针所指的区域中去。进行深拷贝时，一般对象a和b有相同的数据类型。如果在进行赋值时发生深拷贝，就一定要对赋值操作符进行重载，否则赋值运算就会按照赋值的常规语义进行（成员变量之间传递数据），而不会发生深拷贝。下面是一个具体的例子。

```
//// Program 8.14-3 /////
#include <iostream>
using namespace std;
class Student{
    char *name;
    int age;
public:
    Student(){
        name = new char[20];
    }
    Student(char *n,int a){
        name = new char[20];
        if(name) strcpy(name,n);
        age=a;
    }
    Student(const Student& s){
        name = new char[20];
        *this = s;
    }
    void show(){
        cout << "The student's name is " << name;
        cout << " and of age " << age << endl;
    }
    ~Student(){
        delete[] name;
    }
    Student& operator=(const Student &s){
        if(name) strcpy(name,s.name);
        age=s.age;
        return *this;
    }
};

int main(){
```



```

Student s1("张三",18),s4("李四",20);
Student s2;
s1.show();
s2 = s4;
s2.show();
Student s3=s1;
s3.show();
}

```

//// End of Program 8.14-3 ////

程序的输出结果是：

```

The student's name is 张三 and of age 18
The student's name is 李四 and of age 20
The student's name is 张三 and of age 18

```

由于在类Student中，存在指针成员name，所以，当两个Student类成员之间赋值时，必须使用深拷贝。执行s2 = s4;语句，就是将s4对象赋值给s2，其中将s4.name字符串的内容拷入s2.name就是对深拷贝的具体体现。类的拷贝构造函数虽然与赋值操作符并不是一回事，但通常可以在拷贝构造函数中利用赋值操作符重载，以避免对两个对象之间传递数据的重复解释。

在上面的程序中，直接使用 strcpy(name,s.name);实现了两个对象的字符串成员的数据传递。这是一种简化了的做法，如果源字符串的长度超过 20 个字符，此程序将会出现运行错误。更为稳健的做法是每次根据源字符串的长度，重新分配目的字符串的长度（在此之前还要先释放目的字符串的空间）。不过，如果是将一个对象赋值给自己，也会出现错误。解决的办法是：要么进行源对象和目的对象地址的比较，要么先将源对象的字符串内容另外保存起来。

由于深拷贝会涉及到内存的动态分配和释放等一些较为复杂的操作，所以，程序员在编写自定义类时要尽量避免深拷贝的出现。例如，在上例中，将成员变量name定义成string name，就可以避免自己编写实现深拷贝的代码。实际的深拷贝工作是由string类来完成的，而string类是C++标准库提供的，我们有理由放心使用。

另外，对赋值操作符进行重载时，通常将操作符函数的返回值定义为赋值左操作数类型的引用。这是为了实现对赋值表达式的求值，还有一个目的是为了实现“链式操作”。有关链式操作的内容请参见1.14节。

8.15 输入、输出操作符重载

输入操作符是>>，输出操作符是<<，又叫做流对象的“插入操作符”和“提取操作符”。其实这两个操作符最初是在C语言中用于整数的移位运算，到了C++中才利用操作符重载的技术将它们应用于输入、输出操作。

应用于基本数据类型的输入、输出操作都已经在C++标准库中定义好，没有必要重新定义，也不允许重新定义。而对于用户自定义类来说，如果想利用输入、输出操作符进行本类对象的输入、输出操作，就需要对>>和<<操作符进行重载。

对输出操作符<<进行重载，只能采用友元函数的形式进行，而不能将operator <<()声明为ostream类的成员函数。这是因为，ostream是在C++标准库中定义的类，既然是标准库，就不



允许用户随意修改。所以，要将类someClass的对象输出到标准输出对象，只能采用将operator<<()声明为someClass类的友元的形式进行。而且，这时的输出操作符函数原型是下述三种形式之一：

```
ostream& operator <<(ostream&,const someClass&);  
ostream& operator <<(ostream&,someClass&);  
ostream& operator <<(ostream&,someClass);
```

其中，第一种形式最好，也最常用。这种函数重载，既安全又高效。

对输入操作符>>进行重载，也只能采用友元函数的形式进行，而不能将operator>>()声明为istream类的成员函数。这是因为，istream也是C++标准库中的类，也不允许用户随意修改。所以，要从标准输入对象将数据读入类someClass的对象中，只能采用将operator>>()声明为someClass类的友元的形式进行。而且，这时的输入操作符函数原型一定是：

```
istream& operator >>(istream&,someClass&);
```

下面是一个输入、输出操作符重载的例子。

```
/// Program 8.15-1 ////  
#include <iostream>  
using namespace std;  
class Complex{  
    double real;  
    double image;  
public:  
    Complex(double r=0.0,double i=0.0){  
        real=r;  
        image=i;  
    }  
    friend ostream& operator<<(ostream&,const Complex&);  
    friend istream& operator>>(istream&,Complex&);  
};  
ostream& operator<<(ostream &o,const Complex &c){  
    o << c.real << "+" << c.image << "i" ;  
    return o;  
}  
istream& operator>>(istream &i,Complex &c){  
    bool success=false;  
    char ch;  
    while(!success){  
        cout << "Please input a complex: ";  
        i >> c.real;  
        i >> ch;  
        if(ch!='+')  
            continue;
```

```

    i >> c.image;
    i >> ch;
    if(ch!=i)
        continue;
    else
        success=true;
}
return i;
}
int main(){
    Complex c;
    cin >> c;
    cout << c;
}
/// End of Program 8.15-1 ///

```

从键盘输入 $3.40+5.60i$ ，则程序的运行结果是：

Please input a complex: $3.40+5.60i$

$3.4+5.6i$

阅读以上程序，要注意这样几个要点：

①对输入、输出操作符进行重载，只能采用友元函数的形式，而不能采用成员函数的形式，原因前面已经说过。

②如果将输出操作符函数声明为：

`ostream operator <<(ostream,const Complex &);`

或者将输入操作符函数声明为：

`istream operator >>(istream,Complex &);`

都会产生编译错误。原因是`istream`类和`ostream`类的拷贝构造函数被声明为私有(`private`)成员，这样实际上就阻止了`istream`类型和`ostream`类型的参数的传值行为，也阻止了它们成为函数的返回值。详细描述可参见3.5节。

③格式化的输出操作比较容易实现，因为输出的内容已经准备好，如何输出完全由程序员来安排。而格式化的输入操作则要复杂一些，因为输入的内容事先是未知的，用户在输入数据的过程中可能会存在违反约定的行为。所以，在格式化输入函数中通常还要加入一些容错的处理。用户在编写自己的输入函数时要注意这一点。在上面的程序中，对用户输入内容的错误性判断还不是特别完善，有兴趣的读者可自行改进。有关提高输入操作健壮性方面的内容，请参见9.5节。



第9章 | 流类库与输入/输出

9.1 什么是 IO 流

输入输出（IO）是指计算机同任何外部设备之间的数据传递。常见的输入输出设备有文件、键盘、打印机、屏幕等。数据可以按“记录”（或称“数据块”）的方式传递，也可以按“流”的方式传递。

所谓记录，是指有着内部结构的数据块。记录内部除了有需要处理的实际数据之外，还可能包含附加信息，这些附加信息通常是对本记录数据的描述。

“流”是一种抽象概念，它代表了数据的“无结构化”传递。按流的方式进行输入输出，数据被当成无结构的字节序列或字符序列。从流中取得数据的操作称为提取操作，而向流中添加数据的操作则称为插入操作。用来进行输入输出操作的流就称为 IO 流。换句话说，IO 流就是以流的方式进行输入输出。

C++ 的 IO 流，特指以流的方式进行输入输出的 ISO/ANSI 标准 C++ 库的输入输出类库，也就是专门负责处理 I/O 操作的一套系统。任何需要传递的数据，都要经过这套系统的处理。而且，数据在被处理前后是不同的。这样，我们可以把数据的表示分为两种：内部表示和外部表示。

数据的内部表示便于程序进行数据处理。典型的内部表示有：整型数的二进制表示、浮点数的 IEEE 表示、字符的 ASCII 或 Unicode 编码表示。数据的外部表示则根据不同外部设备的需要有具体不同的表现形式。如果外部数据表示是可读的字符序列，则称为文本 I/O，否则为二进制 I/O。标准 IO 流的主要目的是支持文本 I/O，不直接支持二进制 I/O。

虽然 IO 流是以流的方式进行数据传递，但这并不表明传递的数据不能有任何结构，而是指 IO 流的概念是以流的方式进行输入输出，所传递数据的内部结构隐藏在对流数据的解释中。对于 C++ IO 流的用户，与外部设备间的输入输出就是字符流（而不是位流或字节流），这是因为 IO 流主要是为文本 I/O 而设计的。

综上所述，标准 IO 流是针对文本的输入输出而设计的。IO 流主要进行字符序列的外部数据表示与内部数据表示之间的转换，以及在程序和外部设备之间以流的方式传递字符序列。

在 IO 流里，输入输出包括 4 步：格式化/解析，缓冲，编码转换，传递。

格式化/解析在内部数据表示（以字节为单位）与外部数据表示（以字符为单位）之间进行双向转换。例如一个 2 字节表示的整数，可能转换成由 5 个字符组成的序列。

缓冲用于在格式化/解析与传递之间缓存字符序列。对于输出，较短的字符序列格式化后并不马上输出，而是保存在缓冲区里，待累积到一定规模之后再传送到外部设备。相反，从外部设备读入的大量数据也是先放入缓冲区，然后逐步取出完成输入。默认时，IO 流的输入输出操作都是经过缓冲的。也可以让 IO 流工作在无缓冲模式下。



编码转换是将一种字符表达式转换成另一种字符表达式。如果格式化产生的字符表达式与外部字符表达式不同（输出时），或者外部表达式与 IO 流能解析的表达式不同（输入时），就必须进行编码转换。如多字节编码与宽字符编码之间的转换等。多数情况下并不需要进行编码转换。

传递主要是与外部设备进行通信。输出时，传递负责将经过格式化、缓冲及编码转换后的字符序列发送到外部设备；输入时，则负责从外部设备抽取数据，为其后进行的编码转换、缓冲及解析提供字符序列。

IO 流类库随着具体开发平台的发展，其具体实现细节会有所变化。从总体设计上看，I/O 流结构中，ios 是基类，它包含如下两个派生类：

(1) istream 类

istream 类主要供处理数据输入之用，它所包含的成员函数可以处理格式化或非格式化的输入动作，例如 get()、getline()、peek()、putback()、read() 及 ">>" 等。

(2) ostream 类

ostream 类主要供处理数据输出之用，它所包含的成员函数可以处理格式化或非格式化的输出动作，例如 put()、write() 及 "<<" 等。

iostream 类派生于 istream 和 ostream 类，因此，它继承了 istream 和 ostream 类的特性。ifstream 和 ofstream 这两个类则分别派生于 istream 和 ostream 类，因此它们也分别继承了其基类的特性。ifstream 负责对文件进行提取操作，而 ofstream 则负责对文件进行插入操作。

程序员利用 IO 流类库进行输入输出的编程，大多数情况下就是要同 istream、ostream、ifstream 和 ofstream 这四个类打交道。有时，可以将字符串看成字符流，因此，可以利用istrstream 和 ostrstream 类来完成串流的输入输出操作。

实际上，随着 C++语言引入模板机制，以上提到的各个类都是通过模板机制实现的模板类。它们的具体定义如下：

```
typedef basic_ios<char, char_traits<char>> ios;
typedef basic_istream<char, char_traits<char>> istream;
typedef basic_ostream<char, char_traits<char>> ostream;
typedef basic_iostream<char, char_traits<char>> iostream;
typedef basic_ifstream<char, char_traits<char>> ifstream;
typedef basic_ofstream<char, char_traits<char>> ofstream;
typedef basic_fstream<char, char_traits<char>> fstream;
```

它们所依赖的 IO 流类模板的结构体系如图 9-1 所示：

basic_streambuf 不是 basic_ios 的派生类，而是一个独立的类，只是 basic_ios 有一个保护访问限制的指针指向它。类 basic_streambuf 的作用是管理一个流的缓冲区。

basic_ios 类模板提供了对流进行格式化输入输出和错误处理的成员函数。所有派生都是公有派生。basic_istream 类模板提供完成提取（输入）操作的成员函数，而 basic_ostream 类模板提供完成插入（输出）操作的成员函数。basic_iostream 类是前两者的聚合，并没有增加成员。派生全部为公有派生。

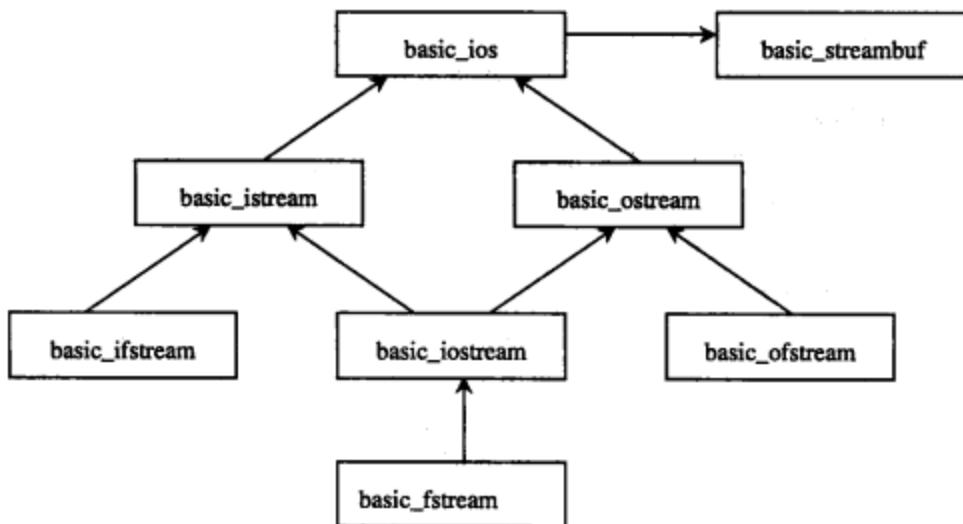


图 9-1 IO 流类模板体系结构

9.2 IO 流类库的优点

C++语言开发了自己的I/O流类库，用以取代C语言的基本输入输出函数族。对于有经验的C程序员来说，C语言提供的I/O函数库是有效和方便的。但是，C语言的I/O函数库有其自身的缺点，特别是在C++这种面向对象的程序设计语言中，C语言函数库无法直接支持面向对象的程序设计。因此，C++语言开发自己的I/O流类库是必然的。具体地说，IO流类库具有如下优点。

(1) 简明与可读性

IO流类库用I/O运算符（提取运算符`>>`和插入运算符`<<`）代替了不同的输入输出函数名（如`printf`, `scanf`等）。从直观上来看，这种改变使得I/O语句更为简明，增加了可读性。另一方面，也减轻了程序员在记忆函数名和书写程序上的一些负担。例如，考察下面的两个输出语句：

```
printf("n=%d,a=%f\n",n,a);
cout<<"n=<<n<<",a="<<a<<endl;
```

虽然两条语句的输出结果是一样的，但在编写程序语句和阅读它们时，感觉却是不同的。后者简明，直观，易写，易读。C程序员转而使用C++语言后，接受这种新的形式并不会有什么困难，而且会逐渐欣赏这种新的输入输出方式给编程带来的便利。

(2) 类型安全 (type safe)

所谓类型安全，是指编译器所理解的数据实体（如变量、指针所指向的数据等）的类型，与数据实体的实际类型或对该数据所进行的操作之间保持一致性。进行I/O操作时，如果程序员所希望输入（或输出）的数据，与实际输入（或输出）的数据在类型上不一致，就发生了类型不安全的情况。以最简单的输出语句为例，下面是一个显示彩色值color和尺寸size的一个简单函数：

```
void show(int color,float size){
    cout<<"color="<<color<<,size="<<size<<endl;
```

} 在这个函数的调用过程中，编译器将自动按参数的类型定义检查实参的表达式。在显示的结果中，color按整型变量输出，而size按浮点型变量输出。这个过程是编译器按照链式操作和函数重载的相关规则自动完成的。

如果采用printf()函数，由于参数的数据类型必须由程序员以参数格式“%d”，“%f”，“%c”，“%s”等形式给出，同样实现上述函数show()，就可能产生编译器无法解决的问题：

```
void show(int color, float size){
```

```
    printf("color=%f, size=%d\n", color, size);
```

```
}
```

由于程序员的疏忽，在格式描述字符串中将color的类型描述成为float型，而将size的类型描述为int型，这些描述与两个变量的实际类型是不符的，但编译器无法检查出这种错误，最后导致输出结果出错。因此说，C语言的I/O函数是类型不安全的。而C++的I/O系统不会出现这种情形。

(3) 易于扩充

C++语言的I/O流类库，是建立在类的继承关系和模板、操作符重载等机制的基础之上的。它把原来C语言中的左、右移位运算符“<<”和“>>”，通过操作符重载的方法，定义为插入（输出）和提取（输入）运算符。既然可以在系统定义的类库上采用这种方法，同样也可以在用户自定义的类型上采用相同的方式，实现输入输出功能对于各种用户定义的类型数据的扩充。

在I/O流类库中，运算符<<的重载函数是作为输出流类ostream的成员函数来定义的，分别对字符串,char, short, int, long, float, double, const void *（指针）等类型作了说明。同样，运算符>>的重载函数是作为输入流类istream的成员函数来定义的，对于上面提到的那些数据类型也分别有定义。在此基础上，对于用户自定义的新的数据类型，其输入输出操作可以友元函数的形式来进行定义。例如，用于复数类Complex的输出操作符重载函数可以定义为：

```
friend ostream &operator<<(ostream &s, const Complex &c){  
    s<<('{'<<c.re<<',''<<c.im<<'}');  
    return s;  
}
```

输入输出操作符重载函数有着固定的格式，具体请参见8.15节。由于C语言并不支持函数重载，也不直接支持面向对象的程序设计，所以，想扩充C语言的输入输出函数使它们支持用户自定义的新数据类型，是一件非常困难的事情。

9.3 endl是什么

自从在C语言的教科书中利用Hello world程序作为学习的起点之后，很多程序设计语言的教科书都沿用了这个做法。我们写过的第一个C++程序可能就是这样的：

```
/// Program 9.3-1 ///  
#include <iostream>  
using namespace std;
```



```
int main(){
    cout << "Hello, world!" << endl;
}
/// End of Program 9.3-1 ///
```

学过C语言的程序员自然会把输出语句与C语言中的输出语句联系起来，也就是说：cout << "Hello, world!" << endl;相当于printf("Hello, world!\n")。由于endl会导致输出的文字换行，自然地我们会想到endl可能就是字符'\n'。

但是，如果我们定义char c = endl;会得到一个编译错误，这说明endl并不是一个字符。所以，应该到系统头文件中去查找endl的定义。在Program 9.3-1中，只包含了一个头文件（即iostream）。在VS 2005环境下，找到头文件iostream并打开，发现其中并没有endl的定义。在头文件iostream中，包含了头文件istream，在istream中也没有endl的定义。继续查找，发现istream包含了头文件ostream，而endl正是在ostream中定义的。定义如下：

```
template<class _Elem,
         class _Traits> inline
basic_ostream<_Elem, _Traits>&
    __CLRCALL_OR_CDECL endl(basic_ostream<_Elem, _Traits>& _Ostr)
{
    // insert newline and flush stream
    _Ostr.put(_Ostr.widen('\n'));
    _Ostr.flush();
    return (_Ostr);
}
```

从这个定义中可以看出，endl是一个函数模板，它实例化之后变成一个模板函数，其作用如这个函数模板的注释所示：插入换行符并刷新输出流。但是，函数调用应该使用一对圆括号，也就是应该写成endl()这样的形式，而在语句cout << "Hello, world!" << endl;中并没有这样，这又是怎么回事呢？

在头文件iostream中，有这样一条声明语句：extern ostream &cout;，这说明cout是一个ostream类的对象。而<<原本是用于移位运算的操作符，在这里用于输出，说明它是一个经过重载的操作符函数。如果把endl当做一个模板函数，那么cout<<endl可以解释成cout.operator<<(endl)，由于一个函数名代表的是一个函数的入口地址，所以在cout的所属类ostream中应该有一个operator<<()函数的重载形式接受一个函数指针做参数。

查找ostream类的定义，发现它是另一个类模板实例化之后生成的模板类，即：

```
typedef basic_ostream<char, char_traits<char> > ostream;
```

所以，实际上应该在类模板basic_ostream中查找函数operator<<()的重载版本。在头文件ostream中查找basic_ostream的定义，发现其中operator<<作为成员函数被重载了17次，其中的一种重载形式是：

```
basic_ostream <_Elem, _Traits>& operator << ( basic_ostream <_Elem, _Traits>& (*_Pfn)(basic_ostream <_Elem, _Traits>&));
```

这个重载正好与endl函数的声明相匹配，所以<<后面是可以跟着endl的。也就是说，cout对象的<<操作符接受到endl函数的地址后会在重载的操作符函数内部调用endl函数，而endl函数会结束当前行并冲洗缓冲区。



为了证明endl是一个函数模板，或者说endl是一个经过隐式实例化（有关模板隐式实例化的详细讨论和例子请参见6.2节）之后的一个模板函数，我们把程序Program 9.3-1改造如下。

```
/// Program 9.3-2 ///
#include <iostream>
using namespace std;
int main(){
    cout << "Hello, world!" << &endl;
}
/// End of Program 9.3-2 ///
```

这个程序的运行结果与Program 9.3-1完全相同，原因是对于函数而言，函数名本身就代表函数的入口地址，而函数名前加&也代表函数的入口地址。

实际上，endl被称为I/O操纵符，也有翻译成I/O算子的。I/O操作符的本质是自由函数，它们并不封装在某个类的内部，使用时也不采用显式的函数调用的形式。在<iostream>头文件中定义的操纵符有：

endl	输出时插入换行符并刷新流
ends	输出时在字符后面插入 NULL 作为尾符
flush	刷新，把流从缓冲区输出到目标设备
ws	输入时略去空白字符
dec	令 I/O 数据按十进制格式
hex	令 I/O 数据按十六进制格式
oct	令 I/O 数据按八进制格式

在<iomanip>头文件中定义的操纵符有：

```
setbase(int)
resetiosflags(long)
setiosflags(long)
setfill(char)
setprecision(int)
setw(int)
```

这些格式控制符大致可以代替ios的格式函数成员的功能，且使用比较方便。例如，为了把整数457按16进制输入，可有两种方式：

```
int i=457;
cout.setf(ios::hex,ios::basefield);
cout<<i<<endl;
或者：
int i=457;
cout<<hex<<i<<endl;
```

可以看出采用格式操纵符比较方便。二者的区别主要在于：格式成员函数是标准输出对象cout的成员函数，因此在使用时必须和cout同时出现，而操纵符是自由函数，可以独立出现；使用格式成员函数时要显式采用函数调用的形式，不能用I/O运算符“<<”和“>>”形成链式操作。



实际上，用户可以编写自己的格式操纵符，只要该操纵符函数满足特定的函数原型的要求即可。如下面的程序。

```
//// Program 9.3-3 /////
#include <iostream>
#include <iomanip>
using namespace std;
ostream& Tab(ostream &outs){
    for(int i=0;i<4;i++)
        outs << '-';
    return outs;
}
ostream& Money(ostream &outs){
    outs.setf(ios::left,ios::adjustfield);
    outs<<'$'<<setw(12)<<setfill('#');
    return outs;
};
int main(){
    double amount=4532.64;
    cout << "The amount is" << Tab << Money << amount << endl;
}
/// End of Program 9.3-3 ///
```

程序的运行结果是：

The amount is---\$4532.64#####

程序中的Tab和Money就是用户自定义的格式操纵符。由于cout的成员函数operator<<()实际上有这样一种重载形式：

ostream & ostream::operator << (ostream& (*_Pfn)(ostream&));

所以只要编写一个返回值为std::ostream&，接收一个类型为std::ostream&参数的函数，就可以把该函数的入口地址传递给cout.operator<<()，完成格式操纵符的功能。

9.4 实现不带缓冲的输入

IO流类库的输入输出操作默认是带缓冲的。在某些情况下，带缓冲的输入（输出）会与程序员期望的程序行为有差异。例如，编写一个控制台应用程序，让用户输入5位密码，然后根据密码正确与否决定下一步的动作。如果采用带缓冲的输入，那么，在用户输入回车之前，不管用户是否已经输完5位密码，程序都不会将输入缓冲区中的内容读入，这样与程序员希望的即时判断是相违背的。

使用库函数_getch和_getche，可以实现不带缓冲的输入。因为不带缓冲，所以也来不及对输入进行任何格式化的操作，所以不带缓冲的输入总是针对单个字符进行的。下面是一个利用_getch()模拟密码输入的程序。

```
/// Program 9.4-1 ///
```



```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main(){
    char c[6];
    cout << "Please input password:";
    for(int i=0;i<5;i++){
        c[i]=_getch();
        cout << '*';
    }
    cout << endl;
    c[5]='\0';
    if(string(c)==="hello")
        cout << "Your password are right";
    else
        cout << "Your password are wrong";
}
/// End of Program 9.4-1 ///
```

通过_getch()读取用户的输入，用户每输入一个字符，都立即被读入内存，并且，在标准输出设备（屏幕）上不会有回显字符。所以，程序中利用cout<<'*'向屏幕输出星号表示用户已经输入一位密码。函数_getche()也是不带缓冲的输入，只不过它会把输入字符回显在屏幕上。

注意，在上面的程序中，输出仍然是带缓冲的。那么，为什么每输入一位密码，都会立即在屏幕上显示出一个星号呢？这是因为系统默认地将输入输出流进行“捆绑”的结果。也就是说，在下一次输入之前，输出缓冲区内的内容必须全部清空。在利用cout对象进行输出时，如果想实现输出缓冲区内的内容强制清空，可以利用flush算子和endl算子。前者只是强制清空输出缓冲区，并不附加一个换行；后者不但清空输出缓冲区，同时还附加一个换行符。所以，如下两条语句是等价的。

```
cout << flush << "sentence\n";
cout << "sentence\n" << endl;
```

9.5 提高输入输出操作的稳健性

输入输出操作有时会出错，C++程序必须能够发现这些错误，并且按照一种对用户友好的方式处理这些错误，以提高程序的稳健性。一种典型的情况是，用户没有清楚地了解输入数据的格式，从而输入了错误的数据。

在C++的IO流类库中，有一个表示输入输出状态的整型状态字state，状态字的各位在ios中说明，其中几个关键性的状态常量是：

```
goodbit=0x00; //流正常
```

```

eofbit=0x01;      //输入流结束，忽略后继提取操作；或者文件结束，已无数据可取
failbit=0x02;     //最近的IO操作失败，流可恢复
badbit=0x04;      //最近的IO操作非法，流可恢复
hardfail=0x08;    //IO出现致命错误，流不可恢复

```

有两种方法可以获得输入/输出的状态信息。一种方法是通过调用rdstate()函数，它将返回当前的状态字。根据状态字中各位的位置情况，可以了解当前IO操作的状态。例如，没有发生任何错误会返回goodbit。

另一种方法则是使用下面任何一个函数来检测相应的输入/输出状态：

```

bool bad();       //返回非法操作位
bool eof();       //返回流（文件）结束位
bool fail();      //返回操作非法和操作失败这两位，也可以用重载后的！代替
bool good();      //正常则返回1，否则返回0

```

假如badbit标志被标设，则bad()函数返回true；假如failbit标志被标设，则fail()函数返回true；假如没有错误发生（goodbit标志被标设），则good()函数返回true；假如操作已经到达了文件末尾（eofbit被标设），则eof()函数返回true。

下面是一个简单的利用直接读取状态字的办法来判断当前IO操作状态的例子程序。

```

/// Program 9.5-1 ///
#include <iostream>
#include <fstream>
using namespace std;
int main(){
    ifstream filein;
    filein.open("1.txt");
    if(filein.rdstate() == ios::eofbit)
        cout << "End of file!\n";
    if(filein.rdstate() == ios::badbit)
        cout << "Fatal I/O error!\n";
    if(filein.rdstate() == ios::failbit)
        cout << "Non-fatal I/O error!\n";
    if(filein.rdstate() == ios::goodbit)
        cout << "No errors!\n";
    filein.close();
}
/// End of Program 9.5-1 ///

```

如果要打开的文件不存在，则failbit标志会被设置，程序输出结果为：

Non-fatal I/O error!

也可以使用相关的状态函数来进行判断。在文件处理中，经常使用eof()函数判断是否已经到达文件的末尾。以下是一个统计某文本文件中有多少个空格的小程序。

```

/// Program 9.5-2 ///
#include <iostream>

```

```
#include <fstream>
using namespace std;
int main(){
    ifstream fin("1.txt");
    char ch; int counter=0;
    while (!fin.eof()){
        ch = fin.get();
        if (ch == '\n') counter++;
        cout << int(ch) << endl;
    }
    cout << counter << endl;
}
/// End of Program 9.5-2 ///
```

使用提取操作(>>操作)无法从输入流读取空格，所以要使用get()函数。理解上述程序的行为要注意两点：首先，文本文件中的回车实际上在文件中占用了两个字节，即0XA0D，但用函数get()读取时，只会返回一个0XA；其次，只有用get()函数读到文件末尾，函数eof()才会返回1，所以，get()读到的最后一个字符是-1，这实际上是文件结束的标志。

对于流，只要出错，相应的状态标志就会置1，此后忽略所有对该流对象的操作，必须用clear()函数清0，然后才能正常运行。

标准输入流比较容易出问题，因此要加强对相应状态的检测，并做出恰当的处理，以提高程序的稳健性。一般说来，要注意这样几个问题。

①由于标准输入流缺省的输入方式是带缓冲的输入，所以从键盘输入的数据是放在输入缓冲区中的，直到用户按下回车键，程序才会真正从输入缓冲区中读取数据。所以，用户输入的数据既不能多，也不能少，否则都有可能引起程序运行的异常。

②输入数据类型与要提取的数据类型必须一致，否则会出错。一旦出错，后继的操作都无法正常运行，所以，必须手动将状态字清0，然后改正错误，并进行后继的操作。

③在格式化输入中，空格和回车都可以作为数据的分隔符，因此，多个数据可以在一行中输入，也可以在多行中输入。如果输入的是字符或字符串，则不能用提取操作符>>完成空格的输入，字符串中也不能有空格，不能读取回车。

④输入数字后再输入字符或字符串，如果数后有回车，则应用cin.get()提取回车，如果有空格，则应读空缓冲区。

以下的程序演示了在输入整数之后，再输入字符串，用检测IO状态字的方式可以提高程序的稳健性。

```
/// Program 9.5-3 ///
#include <iostream>
using namespace std;
int main(){
    char str[256];
    int i;
    cout << "请输入整数: ";
    cin >> i;
    cout << "请输入字符串: ";
    cin.get(str, 256);
    cout << str << endl;
}
```

```

    cin >> i;
    while(cin.fail()){
        cout << "状态字为: " << cin.rdstate() << endl;
        cin.clear(0);
        cin.getline(str,255);
        cout << "输入错误, 请重新输入整数: ";
        cin >> i;
    }
    cin.getline(str,255);
    cout << "请输入字符串: ";
    cin.getline(str,255);
    cout << "输入的整数为: " << i << endl;
    cout << "输入的字符串为: " << str << endl;
}
/// End of Program 9.5-3 ///

```

在输入整数时, 有可能输入了非数字字符, 这样会导致输入出错, 并阻塞后继的输入操作。所以必须对IO状态字进行检测。并且, 由于错误的输入数据放在缓冲区中会对将来的输入操作造成影响, 所以必须将输入缓冲区读空。这里采用的是`cin.getline()`函数, 该函数带两个参数, 第一个参数是存放字符串的内存首地址, 第二个参数表示读取字符串的最大允许字符数。由于一般情况下, 用户不会在一行中输入超过255个字符, 所以可以认为`CIN.GETLINE (STR,255);`会从输入流的当前位置直到当前行的末尾(包括空格、回车键)全部从输入缓冲区中读出。以下是该程序某次运行的结果:

```

请输入整数: t5t
状态字为: 2
输入错误, 请重新输入整数: 78
请输入字符串: How are you?
输入的整数为: 78
输入的字符串为: How are you?

```

9.6 为什么要设定 locale

`locale`翻译成中文是“地区、地域”的意思。它是国际化与本土化过程中的一个非常重要的概念。`locale`是根据计算机用户所使用的语言, 所在国家或者地区, 以及当地的文化传统所定义的一个软件运行时的语言环境。

对于中文用户而言, 与国际化或者本土化相关的任务通常有三种: 看中文, 写中文, 与中文系统通信。`Windows`的用户通常并不会感觉到`locale`的存在。就算是在纯英文的`Windows`操作系统中浏览网页, 只要系统中有相应的字符集(字库)和合适的字体, 并不需要设定`locale`(即使把`locale`设置成`en_US.ISO-8859-1`这样一个标准的英文`locale`也不会有影响), 就可以浏览中文。具体的过程是: 浏览器在接收到网页内容后, 会判断相应的编码的字符集, 根据网页采用的字符集, 去字体库里面找合适的字体, 然后由文字渲染工具把相应的文字在屏幕上

显示出来。所以，尽管我们看到的都是中文网页，但它们的字符编码很可能是不一样的。例如，著名的www.google.cn的网页采用的是UTF-8编码，www.sina.com.cn网页采用的是gb2312编码，而有些网页甚至采用的是ISO-8859-15编码。

有时，网页显示乱码，原因是什么呢？显示乱码通常有两个原因：一是设定的字符集不对，二是使用的字库有问题。如果网页是用UTF-8编码的，而浏览器却要把它当做GB2312编码的网页去显示，那么根据GB2312编码找到的字符肯定不对，在屏幕上显示的当然是一堆乱码。另外一些时候，虽然浏览器已经正确判断出网页使用的字符编码，但由于字库不全，有一部分汉字在特定的字体下根本找不到有效的字模信息，导致浏览器只能显示一部分汉字，而其他地方是方框。找一个比较全的字库可以解决这个问题。

那么，什么时候需要设定locale呢？一种典型的情况是：当程序员需要输出特定语言的文字的时候。对计算机而言，输出文字的时候必须知道使用的是哪一种字符集，否则，就算能够成功写入，也未必能够成功地读出。

为了适应不同的文化习俗，标准C++库提供了许多方法，使得C++程序可以十分方便地进行国际化处理，这些方法被封装在facet类中。一个facet将一套关于不同文化和语言的数据以及一套相关的国际化方法封装在一起。facet类是在标准库中被预定义的，或者是用户自定义的。预定义的facet类称为标准facet类。标准facet类提供有关基本文化差异的方法和信息。这些文化差异涉及语言、字母表以及数据、货币、日期和时间的格式。用户自定义的facet类则涉及更深层次的文化差异，远远超出标准facet类所提供的范围。用户自定义的facet类只有当它们被明确地添加到locale对象中之后，locale对象才包含用户自定义的facet。相反，标准facet类自动包含在每一个locale对象中，它的方法在每个locale对象中都能使用。

要设定locale，必须在程序中创建locale对象。操作系统中为特定的locale提供了相应的名称，利用locale类的构造函数或者静态成员函数，再指明locale的名称，就可以建立locale对象。有效的locale名称是“C”，“”和任何预定义的locale名称。“C”表示美国英语的ASCII码locale，是在非国际化程序中使用的默认方式。通过语句locale(“C”)或调用函数locale::classic()可以创建相应的locale对象。用空字符串表示的locale名称“”则表示系统选择的本国习俗，它与当前的系统设定有关。其他预定义的locale名称没有统一的标准，在不同的操作系统下可能有不同的规范。例如，在X/Open系统中的“De_DE”，与Microsoft的软件平台中的“Germany_Germany.1252”表示的是相同的本地化环境。以下的程序演示了怎样建立具有特定名称的locale对象。

```
/// Program 9.6-1 ///
#include <iostream>
#include <locale>
using namespace std;
int main(){
    locale native("");
    locale usa("American_USA.1252");
    locale Holland("Dutch");
    locale global;
    cout << "native: " << native.name() << endl;
    cout << "classic: " << locale::classic().name() << endl;
```



```

cout << "global: " << global.name() << endl;
cout << "Holland: " << Holland.name() << endl;
cout << "usa: " << usa.name() << endl;
}
/// End of Program 9.6-1 ///

```

程序的运行结果如下：

```

native: Chinese_People's Republic of China.936
classic: C
global: C
Holland: Dutch_Netherlands.1252
usa: English_United States.1252

```

输出结果表明，程序是运行在一个简体中文的平台上。每个C++程序，只有单个全局的locale对象存在。locale的默认构造函数返回当前全局locale的副本。这个副本建立之后，即使全局locale对象已经发生变化，该副本也不会随之发生变化。默认的情况下，全局locale对象是C locale，可以使用locale::global(locale& loc)来代替它。当流被创建时，它们使用这个全局的locale，这就意味着，cin、cout、cerr、wcin和wcerr都是使用C locale。如果需要使用遵循某个特定的locale传统的格式，就必须显式地改变它。Locale的名字是非标准化的，在Windows操作系统下，它们看上去与下面的相同：

<language>_<country>.<codepage>

这里语言或许是一个全名，如西班牙语“Spanish”，或者是两个字符的代码，如“sp”；国家或者是国家的全名，如“Colombia”；或者是两个字符的国家代码，如“CO”，代码页代表特定的字符集，如1252。语言是唯一的必须部分。如果你使用的locale名字是不合法的，程序将抛出一个运行时错误。

在创建locale对象时，所能够合法使用的语言名称和国家名称，在Visual C++中，是在系统源文件getqloc.c中定义的，如下所示：

```

// non-NLS language string table
__declspec(selectany) const LOCALETAB __rg_language[] =
{
    {"american",                  "ENU"}, 
    {"american english",          "ENU"}, 
    {"american-english",          "ENU"}, 
    {"australian",                 "ENA"}, 
    {"belgian",                    "NLB"}, 
    {"canadian",                   "ENC"}, 
    {"chh",                        "ZHH"}, 
    {"chi",                        "ZHI"}, 
    {"chinese",                     "CHS"}, 
    {"chinese-hongkong",           "ZHH"}, 
    {"chinese-simplified",         "CHS"}, 
    {"chinese-singapore",          "ZHI"}, 
}

```

```

    {"chinese-traditional", "CHT"},  

    {"dutch-belgian", "NLB"},  

    {"english-american", "ENU"},  

    {"english-aus", "ENA"},  

    {"english-belize", "ENL"},  

    {"english-can", "ENC"},  

    {"english-caribbean", "ENB"},  

    {"english-ire", "ENI"},  

    {"english-jamaica", "ENJ"},  

    {"english-nz", "ENZ"},  

    {"english-south africa", "ENS"},  

    {"english-trinidad y tobago", "ENT"},  

    {"english-uk", "ENG"},  

    {"english-us", "ENU"},  

    {"english-usa", "ENU"},  

    {"french-belgian", "FRB"},  

    {"french-canadian", "FRC"},  

    {"french-luxembourg", "FRL"},  

    {"french-swiss", "FRS"},  

    {"german-austrian", "DEA"},  

    {"german-lichtenstein", "DEC"},  

    {"german-luxembourg", "DEL"},  

    {"german-swiss", "DES"},  

    {"irish-english", "ENI"},  

    {"italian-swiss", "ITS"},  

    {"norwegian", "NOR"},  

    {"norwegian-bokmal", "NOR"},  

    {"norwegian-nynorsk", "NON"},  

    {"portuguese-brazilian", "PTB"},  

    {"spanish-argentina", "ESS"},  

    {"spanish-bolivia", "ESB"},  

    {"spanish-chile", "ESL"},  

    {"spanish-colombia", "ESO"},  

    {"spanish-costa rica", "ESC"},  

    {"spanish-dominican republic", "ESD"},  

    {"spanish-ecuador", "ESF"},  

    {"spanish-el salvador", "ESE"},  

    {"spanish-guatemala", "ESG"},  

    {"spanish-honduras", "ESH"},  

    {"spanish-mexican", "ESM"},  


```





```

    {"spanish-modern",           "ESN"},  

    {"spanish-nicaragua",        "ESI"},  

    {"spanish-panama",          "ESA"},  

    {"spanish-paraguay",         "ESZ"},  

    {"spanish-peru",             "ESR"},  

    {"spanish-puerto rico",      "ESU"},  

    {"spanish-uruguay",          "ESY"},  

    {"spanish-venezuela",        "ESV"},  

    {"swedish-finland",          "SVF"},  

    {"swiss",                     "DES"},  

    {"uk",                        "ENG"},  

    {"us",                        "ENU"},  

    {"usa",                       "ENU"}  

};  
  

// non-NLS country/region string table  

_declspec( selectany ) const LOCALETAB __rg_country[] =  

{
    {"america",                  "USA"},  

    {"britain",                   "GBR"},  

    {"china",                     "CHN"},  

    {"czech",                      "CZE"},  

    {"england",                   "GBR"},  

    {"great britain",             "GBR"},  

    {"holland",                   "NLD"},  

    {"hong-kong",                 "HKG"},  

    {"new-zealand",                "NZL"},  

    {"nz",                         "NZL"},  

    {"pr china",                  "CHN"},  

    {"pr-china",                   "CHN"},  

    {"puerto-rico",                "PRI"},  

    {"slovak",                     "SVK"},  

    {"south africa",               "ZAF"},  

    {"south korea",                 "KOR"},  

    {"south-africa",                "ZAF"},  

    {"south-korea",                  "KOR"},  

    {"trinidad & tobago",          "TTO"},  

    {"uk",                          "GBR"},  

    {"united-kingdom",              "GBR"},  

    {"united-states",                "USA"},  

}

```

```

    {"us",
     "USA"},

};

```

在输出特定编码的字符时，要特别注意相应的locale的设置，否则会导致输出流不能正常工作。考察下面的程序。

```

/// Program 9.6-2 ///
#include <iostream>
using namespace std;
int main(){
    char str1[10] = "ABC我们";
    wchar_t str2[10] = L"ABC我们";
    cout << "cout << str1: " << str1 << endl;
    cout << "cout << str2: " << str2 << endl;
    wcout << "wcout << str1: " << str1 << endl;
    wcout << "wcout << str2: " << str2 << endl;
    cout << "*****" << endl;
    wcout << "wcout << str1: " << str1 << endl;
}

```

```
/// End of Program 9.6-2 ///
```

程序的运行结果是：

```

cout << str1: ABC我们
cout << str2: 0012FF38
wcout << str1: ABC我们
wcout << str2: ABC*****

```

从程序的运行结果可以看出，cout对象负责多字节字符串的输出，而wcout主要负责宽字符串的输出。如果将wchar_t*类型的字符串输出到cout，由于并没有进行相应的操作符重载，所以wchar_t*类型的参数只是被当做一个指针进行输出。多字节字符串向wcout对象输出，可以得到正确的输出结果，这是相应的操作符重载的结果。缺省情况下，全局locale是"C" locale，wcout并不能输出宽字符的汉字字符。因此，在上面的程序中，语句wcout << str2的执行失败，导致后继的所有利用wcout对象进行的输出全部处于阻塞状态。也就是说，wcout << endl;并没有执行，并且其后的wcout << "wcout << str1: " << str1 << endl;也没有执行。

所以为了正确地输出宽字符汉字，必须设置wcout的locale为本地语言（简体中文）。替换全局locale对象并不是一个好的办法，因为有多个输入/输出对象会依赖全局locale对象。可以利用wcout.imbue()函数来更改wcout的locale设置，这样就可以正确地输出宽字符汉字了。见下面的程序。

```

/// Program 9.6-3 ///
#include <iostream>
#include <locale>
using namespace std;
int main(){
    char str1[10] = "ABC我们";

```

```
wchar_t str2[10]= L"ABC我们";
cout << "cout << str1: " << str1 << endl;
cout << "cout << str2: " << str2 << endl;
wcout << "wcout << str1: " << str1 << endl;
cout << "*****" << endl;
wcout.imbue(locale("CHS"));
wcout << "wcout << str1: " << str1 << endl;
wcout << "wcout << str2: " << str2 << endl;
}
```

//// End of Program 9.6-3 ////

程序的执行结果是：

```
cout << str1: ABC 我们
cout << str2: 0012FF2C
wcout << str1: ABC 我们
*****
wcout << str1: ABC
wcout << str2: ABC 我们
```

将wcout的locale设置成简体中文之后，多字节字符串中的汉字就不能够正常通过wcout显示出来，因为此时的char*已经被解释成单字节编码的字符串。而wchar_t*类型的字符串就可以正确显示了。这个例子说明在输出时设置locale的必要性。

如果要将宽字符串输出到文件中，简单地设置输出流对象的locale并不能达到目的。考察下面的程序。

```
//// Program 9.6-4 /////
#include <iostream>
#include <fstream>
using namespace std;
int main(){
    char str1[10] = "ABC我们";
    wchar_t str2[10] = L"ABC我们";
    ofstream file1("file1.txt",ios_base::out);
    file1 << str1;
    file1.close();
    wofstream file2("file2.txt",ios_base::out);
    file2.imbue(locale("CHS"));
    file2 << str2;
    file2.close();
    ofstream file3("file3.txt",ios_base::out);
    short flag = 0xFEFF;
    file3.write((char*)&flag,sizeof(short));
    file3.write((char*)str2,wcslen(str2)*2);
```



```

    file3.close();
}

/// End of Program 9.6-4 ///

```

在这个程序中，一共生成了三个文件。第一个文件是将char*类型的字符串写入文本文件，文件的大小为7字节，这正是多字节字符串的特点。第二个文件虽然设置了输出文件流对象的locale，但那只是保证了wchar_t*类型的字符串（中的中文字符）能够被输出到文件中，但仍然是按照多字节字符串的编码输出的。所以，file1.txt和file2.txt的内容是完全一样的。实际上，要真正在文件中使用Unicode编码，可以直接用二进制的方式将wchar_t*类型的字符串的内容写入文件中，只不过在文件的最开头写入一个Unicode编码的标志：0xFEFF。函数wcslen(wchar_t*)用来计算宽字符串的长度（字符个数），由于每个字符占2个字节，所以宽字符个数乘以2就是输出数据所占字节数。文件file3.txt的长度为12，正好是5个宽字符加上一个Unicode编码标记的长度。

9.7 char*和 wchar_t*之间的转换

char字符与wchar_t字符由于编码不同，所以在char*和wchar_t*之间使用强制类型转换达不到正确转换字符串的目的。考察下面的程序。

```

/// Program 9.7-1 ///
#include <iostream>
using namespace std;
int main(){
    wchar_t *str = L"ABC我们";
    char *s = (char*)str;
    cout << s << endl;
}
/// End of Program 9.7-1 ///

```

经过强制类型转换，s指向了宽字符编码字符串，字符串数据没有发生任何变化，只是用多字节字符编码重新对它进行解释，自然输出的结果是错误的（只会输出一个A）。

所以，要将宽字符编码字符串转换成多字节编码字符串（或者反过来），必须先“读懂”原来的字符串，然后再重新对它进行编码。只有这样才能达到转换的目的。由于宽字符可以表示多国的文字，因此，以下的讨论限于中文的宽字符串与多字节字符串之间的相互转换。在不同的操作系统上，有一些特殊的库函数可以用来进行字符编码之间的转换。我们先讨论如何利用标准C++库来实现字符编码转换，其中关键性的几个函数是setlocale()、wcstombs_s()和mbstowcs_s()。

setlocale的函数原型是：char *setlocale(int category,const char *locale);

如果第二个参数为空指针，函数返回当前的locale设置，否则依据输入的两个参数设置新的locale。如果设置成功，返回一个描述新locale的字符串，否则返回一个空指针。

wcstombs_s的函数原型是：errno_t wcrtomb_s(size_t *pReturnValue,char *mbchar,size_t sizeOfmbchar,wchar_t *wchar,mbstate_t *mbstate);

wcstombs_s用于将宽字符编码字符串转换成多字节编码字符串。参数pReturnValue指向转

换后的字符串的长度，mbchar是转换后的字符串的首地址，sizeOfmbchar是多字节字符串所可能拥有的最大长度，wchar是源宽字符串首地址，mbstate是一个指向状态字的指针。

函数mbstowcs_s与wcstombs_s的用法基本相同，只是转换的方向正好相反。以下是一个利用C++标准库函数进行字符编码转换的例子。

```
/// Program 9.7-2 ///
#include <iostream>
#include <string>
#include <locale.h>
using namespace std;
string ws2s(const wstring& ws){
    size_t convertedChars=0;
    string curLocale = setlocale(LC_ALL, NULL);           // curLocale = "C";
    setlocale(LC_ALL, "chs");
    const wchar_t* _Source = ws.c_str();
    size_t _Dsize = 2 * ws.size() + 1;
    char* _Dest = new char[_Dsize];
    wcstombs_s(&convertedChars, _Dest, _Dsize, _Source, _TRUNCATE);
    string result = _Dest;
    delete []_Dest;
    setlocale(LC_ALL, curLocale.c_str());
    return result;
}
wstring s2ws(const string& s){
    size_t convertedChars=0;
    setlocale(LC_ALL, "chs");
    const char* _Source = s.c_str();
    size_t _Dsize = s.size() + 1;
    wchar_t* _Dest = new wchar_t[_Dsize];
    mbstowcs_s(&convertedChars, _Dest, _Dsize, _Source, _TRUNCATE);
    wstring result = _Dest;
    delete []_Dest;
    setlocale(LC_ALL, "C");
    return result;
}
int main(){
    wchar_t*wstr = L"ABC我们";
    string obj(ws2s(wstr));
    cout << obj << endl;
    char*str = "ABC我们";
    wstring wobj(s2ws(str));
}
```

```

    std::wcout.imbue(std::locale("chs"));
    wcout << wobj << endl;
}

```

/// End of Program 9.7-2 ///

程序的输出结果是：

ABC我们

ABC我们

程序运行结果表明，由char*到wchar_t*的双向转换都是成功的。要注意的是，执行转换的函数mbstowcs_s和wcstombs_s的运行是依赖于当前的locale设置的。在程序中去除相关的setlocale()函数调用，就得不到正确的结果。

除了可以利用C/C++库函数解决字符编码的转换问题，还可以利用特定操作系统下提供的函数。例如，可以利用Windows API实现字符编码的转换。下面就是一个例子。

/// Program 9.7-3 ///

```

#include <iostream>
#include <windows.h>
using namespace std;
int main(){
    wchar_t *ws=L"测试字符串";
    char *ss="ABC我们";
    int BufSize;
    BufSize = WideCharToMultiByte(CP_ACP,NULL,ws,-1,NULL,0,NULL, FALSE);
    cout << BufSize << endl;
    char *sp = new char[BufSize];
    WideCharToMultiByte (CP_ACP,NULL,ws,-1,sp,BufSize,NULL, FALSE);
    cout << sp << endl;
    delete[] sp;
    BufSize=MultiByteToWideChar(CP_ACP,0,ss,-1,NULL,0);
    cout << BufSize << endl;
    wchar_t *wp= new wchar_t[BufSize];
    MultiByteToWideChar(CP_ACP,0,ss,-1,wp,BufSize);
    std::wcout.imbue(std::locale("chs"));
    std::wcout << wp << std::endl;
    delete[] wp;
}

```

/// End of Program 9.7-3 ///

程序的输出结果是：

11

测试字符串

6

ABC我们



其中函数调用BufSize=WideCharToMultiByte(CP_ACP,NULL,ws,-1,NULL,0,NULL, FALSE);是用来获取宽字符串转换成多字节字符串后所占据空间的大小(以字节计),这是将第4个参数设置成NULL之后达到的效果。同样,函数调用BufSize=MultiByteToWideChar(CP_ACP,0,ss,-1,NULL,0);是用来获取多字节字符串转换成宽字符串后所占空间的大小(以宽字符个数计),这是将第5个参数设置成NULL之后达到的效果。

WideCharToMultiByte是宽字符到多字节字符的转换函数,函数原型如下:

```
int WideCharToMultiByte(
    UINT   CodePage,
    DWORD  dwFlags,
    LPCWSTR lpWideCharStr,
    int    cchWideChar,
    LPSTR  lpMultiByteStr,
    int    cbMultiByte,
    LPCSTR lpDefaultChar,
    LPBOOL lpUsedDefaultChar
);
```

此函数把宽字符串转换成指定的新字符串,如ANSI,UTF8等,新字符串不必是多字节字符集。各参数的含义如下。

CodePage指定要转换成的字符集代码页,它可以是任何已经安装的或系统自带的字符集,也可以使用如下所示代码页之一:CP_ACP,CP_MACCP,CP_OEMCP,CP_SYMBOL,CP_THREAD_ACP,CP_UTF7,CP_UTF8。通常用得最多的是CP_ACP和CP_UTF8,前者将宽字符转换为ANSI,后者转换为UTF8。此参数设置为UTF8时lpDefaultChar和lpUsedDefaultChar都必须为NULL。

dwFlags指定如何处理没有转换的字符。也可以不设此参数(设置为0),函数会运行的更快一些。对于UTF8,dwFlags必须为0或WC_ERR_INVALID_CHARS,否则函数都将失败返回并设置错误码ERROR_INVALID_FLAGS,可以调用GetLastError获得。

lpWideCharStr是待转换的宽字符串。

cchWideChar是待转换宽字符串的长度,-1表示转换到字符串结尾。

lpMultiByteStr是接收转换后输出新串的缓冲区。

cbMultiByte是输出缓冲区大小,如果为0,lpMultiByteStr将被忽略,函数将返回所需缓冲区大小而不使用lpMultiByteStr。

lpDefaultChar是指向字符的指针,在指定编码里找不到相应字符时使用此字符作为默认字符代替。如果为NULL则使用系统默认字符。对于要求此参数为NULL的dwFlags而使用此参数,函数将失败返回并设置错误码ERROR_INVALID_PARAMETER。

lpUsedDefaultChar是开关变量的指针,用以表明是否使用过默认字符。对于要求此参数为NULL的dwFlags而使用此参数,函数将失败返回,并设置错误码ERROR_INVALID_PARAMETER。lpDefaultChar和lpUsedDefaultChar都设为NULL,函数会更快一些。

如果函数成功,且cbMultiByte非0,则返回写入lpMultiByteStr的字节数(包括字符串结尾的null);cbMultiByte为0,则返回转换所需字节数。函数失败,返回0。

MultiByteToWideChar是多字节字符到宽字符转换函数,函数原型如下:



```
int MultiByteToWideChar(
    UINT   CodePage,
    DWORD  dwFlags,
    LPCSTR  lpMultiByteStr,
    int    cbMultiByte,
    LPWSTR  lpWideCharStr,
    int    cchWideChar
);
```

此函数把多字节字符串转换成宽字符串(Unicode)，待转换的字符串并不一定是多字节的。此函数的参数，返回值及注意事项参见上面函数WideCharToMultiByte的说明，这里只对dwFlags做简单解释。dwFlags指定是否转换成预制字符或合成的宽字符，对控制字符是否使用像形文字，以及怎样处理无效字符。对于UTF8，dwFlags必须为0或MB_ERR_INVALID_CHARS，否则函数都将失败，并返回错误码ERROR_INVALID_FLAGS。

9.8 获取文件信息

在对一个文件进行处理之前，往往希望了解一下文件的基本信息（或者说，文件的元数据），比如文件的大小、文件的创建时间、上一次修改的时间等。C++标准库提供了对文件内容处理的支持，但并不直接支持对文件基本信息的获取和修改。所以，获取文件的基本信息可以借助于C库中定义的结构和函数。

在<sys/stat.h>头文件中，定义了一个结构stat，它是用来描述文件元数据结构的。其定义如下所示：

```
struct stat {
    _dev_t      st_dev;
    _ino_t      st_ino;
    unsigned short st_mode;
    short       st_nlink;
    short       st_uid;
    short       st_gid;
    _dev_t      st_rdev;
    _off_t      st_size;
    time_t      st_atime;
    time_t      st_mtime;
    time_t      st_ctime;
};
```

该结构的某些数据成员在不同的操作系统中可能具有不同的含义，如st_uid和st_gid在Windows操作系统下是没有意义的。而另外一些数据成员则具有固定的含义，如st_mtime表示文件上一次被修改的时间，st_ctime是文件的创建时间，st_dev表示设备号（与文件所在的磁盘号相关）等等。与该结构同名的一个系统调用函数stat()可以用来获取文件信息，它的功能是直接由操作系统的服务提供的。以下是一个简单的显示文件信息的例子程序。

```

/// Program 9.8-1 ///
#include <iostream>
#include <ctime>
#include <sys/types.h>
#include <sys/stat.h>
#include <cerrno>
#include <cstring>
using std::cout;
int main(int argc, char** argv ){
    struct stat fileInfo;
    char message[30];
    if (argc < 2) {
        cout << "Usage: fileinfo <file name>\n";
        return(EXIT_FAILURE);
    }
    if (stat(argv[1], &fileInfo) != 0) { // Use stat( ) to get the info
        strerror_s(message,29,errno);
        std::cerr << "Error: " << message << '\n';
        return(EXIT_FAILURE);
    }
    cout << "Type:           : ";
    if ((fileInfo.st_mode & S_IFMT) == S_IFDIR) // From sys/types.h
        cout << "Directory\n";
    else
        cout << "File\n";
    cout << "Size           : " << fileInfo.st_size << '\n'; // Size in bytes
    cout << "Device         : " << (char)(fileInfo.st_dev + 'A') << '\n'; // Device number
    ctime_s(message,29,&fileInfo.st_ctime);
    cout << "Created        : " << message; // Creation time
    ctime_s(message,29,&fileInfo.st_mtime);
    cout << "Modified       : " << message; // Last mod time
}
/// End of Program 9.8-1 ///

```

程序的运行结果类似如下的形式：

```

Type:           : File
Size            : 377516
Device          : D
Created         : Wed Jul 09 15:35:32 2008
Modified        : Wed Jul 09 16:09:52 2008

```

time_t数据类型表示的时间（日历时间）是从一个时间点（例如：1970年1月1日0时0分0



秒) 到此时的秒数。在time.h中, 可以看到time_t是一个长整型数:

```
#ifndef _TIME_T_DEFINED  
typedef long time_t; /*时间值 */  
#define _TIME_T_DEFINED /*避免重复定义 time_t */  
#endif
```

既然time_t实际上是长整型, 到未来的某一天, 从一个时间点(一般是1970年1月1日0时0分0秒)到那时的秒数(即日历时间)超出了长整型所能表示的数的范围怎么办? 对time_t数据类型的值来说, 它所表示的时间不能晚于2038年1月18日19时14分07秒。为了能够表示更久远的时间, 一些编译器厂商引入了64位甚至更长的整型数来保存日历时间。比如微软在Visual C++中采用了_time64_t数据类型来保存日历时间, 并通过_time64()函数来获得日历时间(而不是通过使用32位字的time()函数), 这样就可以通过该数据类型保存3001年1月1日0时0分0秒(不包括该时间点)之前的时间。

time_t类型的数据便于计算机的处理, 但是不便于用户阅读。因此, 可用函数ctime_s()将time_t类型的数据转换成字符串形式, 然后再输出给用户。

9.9 管理文件和目录的相关操作

在处理文件时, 除了对文件的内容进行读写之外, 还有获取文件所在目录、对文件进行重命名、删除文件、创建和删除目录、获取文件列表等等重要操作。下面就几种重要操作分别加以讨论。

(1) 获取程序的当前工作目录

有时在程序中要打开一个文件, 却因为文件的路径不对而失败。所以, 有必要了解程序的当前工作目录。C++标准库并没有提供直接的方法获取该信息, 可以利用C语言运行库中提供的函数来解决这个问题。以下是一个示例。

```
//// Program 9.9-1 ////  
#include <iostream>  
#include <direct.h>  
#include <errno.h>  
int main(){  
    char filepath[100];  
    char *forcestr;  
    _getcwd(filepath,99);  
    std::cout << filepath << std::endl;  
    forcestr = _getcwd(NULL,0);  
    std::cout << forcestr << std::endl;  
    if(forcestr)  
        free(forcestr);  
    if(_chdir("..")){  
        switch (errno){  
            case ENOENT:  
                std::cout << "No such directory." << std::endl;
```



```

        std::cout << "Unable to locate the directory" << std::endl;
        break;
    case EINVAL:
        std::cout << "Invalid buffer" << std::endl;
        break;
    default:
        std::cout << "Unknown error.\n" << std::endl;
    }
}

else{
    forcestr = _getcwd(NULL,0);
    std::cout << forcestr << std::endl;
    if(forcestr)
        free(forcestr);
}
}

```

//// End of Program 9.9-1 ////

函数_getcwd()带两个参数，第一个参数是存放路径的内存首地址，第二个参数是路径字符串所允许的最大长度。要调用这个函数，要包含direct.h头文件。如果传入该函数的参数是NULL或0，则在函数内部会根据路径长度调用malloc()申请空间，并将路径存入动态申请的空间中。不过，一定要在此后用free()释放空间，否则会造成内存泄漏。

如果要改变程序的当前工作目录，可以调用函数_chdir()，该函数带一个char*类型的参数，代表目标目录的绝对路径或相对路径。函数返回0表示操作成功，返回非0值表示操作失败。可以根据errno的值判断操作失败的原因。以下是在某台机器上用Visual Studio 2005调试程序的结果。

```

D:\teaching\C++\study\c++137\filepath
D:\teaching\C++\study\c++137\filepath
D:\teaching\C++\study\c++137

```

在这次调试中，可执行文件所在的目录为D:\teaching\C++\study\c++137\debug，而程序运行结果却显示当前工作目录为D:\teaching\C++\study\c++137\filepath。实际考察可知，在Visual Studio IDE中进行调试时，程序的工作目录被设置为项目文件所在的目录。如果在命令行下启动程序，则当前工作目录为exe文件所在的目录。

(2) 删除或重命名文件

利用C语言库函数，可以实现文件的重命名和删除操作。并且，这些操作不是调用操作系统的API实现的。这样在一定程度上保证可程序的可移植性。

```

//// Program 9.9-2 ////
#include <iostream>
#include <cstdio>
#include <cerrno>
using namespace std;

```



```
int main(int argc, char** argv){  
    char message[50];  
    if(argc==3){  
        if(rename(argv[1], argv[2])){  
            strerror_s(message,49,errno);  
            cerr << "Error: " << message << endl;  
            return(EXIT_FAILURE);  
        }  
        else  
            cout << argv[1] << " has been renamed to " << argv[2] << endl;  
    }  
    else if(argc == 2){  
        if(remove(argv[1]) == -1) { // remove( ) returns -1 on error  
            strerror_s(message,49,errno);  
            cerr << "Error: " << message << endl;  
            return(EXIT_FAILURE);  
        }  
        else  
            cout << "File " << argv[1] << " removed." << endl;  
    }  
    else{  
        cerr << "You must supply a file name to remove." << endl;  
        return(EXIT_FAILURE);  
    }  
}  
/// End of Program 9.9-2 ///
```

此程序通过命令行参数来指明要重命名或删除的文件。当命令行中包括执行文件在内一共有3个参数时，表示重命名；有2个参数时，表示删除文件；只有一个参数时，提示必须输入文件名。重命名是通过函数rename()来实现的，函数返回非0值表示重命名失败。删除文件是通过函数remove()实现的，返回值为-1表示删除操作失败。

(3) 创建和删除目录

可用函数mkdir()创建目录，用_rmdir()删除目录，它们都是C运行库中的函数，使用时要包含头文件direct.h。下面是使用这两个函数的例子。

```
/// Program 9.9-3 ///  
#include <iostream>  
#include <direct.h>  
using namespace std;  
int main(int argc, char** argv) {  
    char message[50];  
    if (argc < 2){
```



```

    std::cerr << "Usage: " << argv[0] << " [new dir name]\n";
    return(EXIT_FAILURE);
}

if (_mkdir(argv[1]) == -1) { // Create the directory
    strerror_s(message, 49, errno);
    std::cerr << "Error: " << message;
    return(EXIT_FAILURE);
}

}

/// End of Program 9.9-3 ///

```

`_mkdir()`带一个`char*`类型的参数，表示要创建的目录的路径，可以使用相对路径，也可以使用绝对路径。每调用一次该函数，只能产生一个新目录。如果创建目录失败（如目录已经存在），则返回非0值，否则返回0。

/// Program 9.9-4 ///

```

#include <iostream>
#include <direct.h>
using namespace std;
int main(int argc, char** argv){
    char message[50];
    if (argc < 2){
        cerr << "Usage: " << argv[0] << " [dir name]" << endl;
        return(EXIT_FAILURE);
    }

    if (_rmdir(argv[1]) == -1) { // Remove the directory
        strerror_s(message, 49, errno);
        cerr << "Error: " << message << endl;;
        return(EXIT_FAILURE);
    }

}

```

/// End of Program 9.9-4 ///

`_rmdir()`带一个`char*`类型的参数，表示要删除的目录的路径，可以使用相对路径，也可以使用绝对路径。每调用一次该函数，只能删除一个目录。如果某个目录非空，则必须先清空该目录下的所有文件和子目录，然后才能删除。删除目录不成功返回非0值，否则返回0。

9.10 二进制文件的IO操作

习惯上，把非文本文件叫做二进制文件。实际上，所有的文件都是按照二进制存储的，所以这种叫法只能从习惯的角度加以解释。文本文件中存放的大部分是可见字符，加上少许控制字符。而二进制文件则可以包含任何字符和数据。二进制文件的输入输出不能利用插入和提取操作符（`<<`和`>>`操作符）来完成，因为它们是为文本文件准备的。必须使用`read()`和

`write()`函数读取和写入二进制文件。创建一个二进制文件可用下述语句实现：

```
ofstream fout("file.dat", ios::binary);
```

这样会以二进制方式打开文件，而不是默认的ASCII方式。写入文件用函数`write()`实现。该函数带有两个参数，第一个是指向对象的`char`类型的指针，第二个是对象的大小（字节数）。例如，把一个整数写入二进制文件：

```
int number = 30; fout.write((char *)(&number), sizeof(number));
```

第一个参数是取出整数`number`的首地址，然后把它转换成`char *`指针。第二个参数表明要将多少字节的内容写入文件。在32位操作系统上，`sizeof(int)`的值为4。

使用二进制文件的好处是可以一次把一个结构写入文件。而如果采用文本文件存储结构，不得不分别将结构的数据成员一个一个地写入文件。看下面的例子。

```
struct OBJECT { int number; char letter; } obj;
obj.number = 15;
obj.letter = 'M';
fout.write((char *)(&obj), sizeof(obj));
```

这样就写入了整个结构。使用二进制文件能够大大提高数据的存取效率。它的缺点是不利于人的阅读，也不利于进行检索。

从二进制文件中读取数据，使用函数`read()`，它是函数`write()`的对等函数，使用方法也相同。例如，要读取上面刚刚存入文件的对象`obj`，可以使用下述语句。

```
ifstream fin("file.dat", ios::binary); fin.read((char *)(&obj), sizeof(obj));
```

当使用这样的语法打开二进制文件：

```
ofstream fout("file.dat", ios::binary);
```

“`ios::binary`”是打开选项的额外标志。默认地，文件以ASCII方式打开，不存在则创建，存在就覆盖。这里有些额外的标志用来改变默认设置。

`ios::app`：添加到文件尾。

`ios::ate`：把文件标志放在末尾而非起始。

`ios::trunc`：默认，截断并覆写文件。

`ios::nocreate`：文件不存在也不创建。

`ios::noreplace`：文件存在则失败。

实际上，文本文件也可以用二进制方式打开，而且在某些处理上更为方便。以下是一个利用C运行库函数实现文件拷贝的程序。该程序采用二进制文件读写的方式实现文件的快速拷贝。

```
//// Program 9.10-1 /////
#include <iostream>
#include <fstream>
using std::ios_base;
int main(int argc, char** argv){
    const static int BUF_SIZE = 4096;
    if(argc!=3){
        std::cout << "Usage: copyf sourcefile targetfile" << std::endl;
        return 1;
    }
    ifstream fin(argv[1], ios::binary);
    ofstream fout(argv[2], ios::binary);
    char buffer[BUF_SIZE];
    while(fin.read(buffer, BUF_SIZE))
        fout.write(buffer, BUF_SIZE);
}
```

```

}

std::ifstream in(argv[1],
    ios_base::in | ios_base::binary);           // Use binary mode so we can
std::ofstream out(argv[2],                  // handle all kinds of file
    ios_base::out | ios_base::binary);          // content.

if(!in || !out){
    std::cout << "opening or creating file failed" << std::endl;
    return 1;
}

char buf[BUF_SIZE];
do{
    in.read(&buf[0], BUF_SIZE);             // Read at most n bytes into
    out.write(&buf[0], in.gcount());        // buf, then write the buf to
}while(in.gcount() > 0);                   // the output.

in.close();
out.close();
}

```

/// End of Program 9.10-1 ///

在此程序中，利用了重载的操作符!来判断打开或创建文件的操作是否成功，in.gcount()返回上一次读操作所实际读取数据的字节数。该程序不论是对文本文件还是二进制文件都能够正常工作。

由于二进制文件可以字节为单位精确地进行读写，所以，可以程序控制文件指针的移动，从而实现文件的随机访问。istream类中提供了如下3个成员函数：

istream& istream::seekg(streampos);	//指针直接定位
istream& istream::seekg(streamoff,ios::seek_dir);	//指针相对定位
long istream::tellg();	//返回当前指针位置

流的指针位置类型streampos和流的指针偏移类型streamoff定义为长整数，也就是可访问文件的最大长度为 2^{32} B，即4GB。

其中，seek_dir是ios中定义的一个枚举类型，其中定义了3个枚举常量：beg、cur和end，分别表示文件的开头、当前位置和结尾。

ostream类也提供了3个成员函数来管理文件定位指针，它们是：

ostream& ostream::seekp(streampos);
ostream& ostream::seekp(streamoff,ios::seek_dir);
long ostream::tellp();

这三个函数的作用与用法与前面介绍的三个函数相同。为了便于记忆，函数名中g代表get，p代表put，也就是分别代表读与写。

以下的程序演示了如何在某个已经存在的文件中改写部分内容，其中文件指针的定位是实现程序功能的关键。

/// Program 9.10-2 ///

```
#include <iostream>
```

```

#include <fstream>
using namespace std;
int main(){
    char c;
    ios::streampos position;
    fstream fout("1.bin",ios::in|ios::out|ios::binary);
    fout.seekp(0L,ios::end);
    position = fout.tellp();
    fout.seekp(0L,ios::beg);
    cout << position << endl;
    while(fout.tellp()!=position){
        fout.read(&c,1);
        cout << fout.tellp() << endl;
        if(c=='A'){
            fout.seekp(-1L,ios::cur);
            c='0';
            fout.write(&c,1);
        }
    }
    fout.close();
}

```

//// End of Program 9.10-2 ////

假设某文件共有8个字符：ABCDABCD，那么文件指针的初始位置（也就是文件刚打开时，`tellp()`函数的返回值）为0，而文件末尾（`ios::end`）对应的值为8。此程序依次从文件开头读取每个字符，如果发现某个字符为‘A’，就倒退一个字符的位置（向文件头将文件指针移动一个字节），然后将‘A’改写成‘0’，所以文件执行结束之后，文件的内容变成了：0BCD0BCD。

如果将程序中的循环条件改写成`while(!fout.eof())`，则此程序可能陷入死循环。这是要特别引起注意的地方。因为，程序必须在文件的末尾（也就是偏移量为8）的地方执行一次读取操作，才能确定是否到达文件尾（`eof()`返回`true`），而此时有可能读取的字符本身就是‘A’，这样导致新的写入动作。所以，应该对纯输入文件使用`eof()`函数，而对于同时可读写的文件，应该采用其他的方式确定文件是否结束。

第10章 | 异常处理

10.1 C++为什么要引入异常处理机制

在程序设计中，错误是不可避免的。及时有效地发现错误，并做出适当的处理，无论是在软件的开发阶段还是在维护阶段都是至关重要的。错误修复技术的改进是提高代码健壮性的最有效方法之一。

而程序员往往忽视出错处理。这并不是因为程序员真的认为自己的程序不会出错，而是因为出错处理实在不是一件轻松的事情。编写出错处理的代码，一方面可能会分散处理“主要”问题的精力，另一方面会引起代码的膨胀，给阅读和维护带来困难。而且，尽可能详尽地考虑出错的情形也是一件费时费力的事情。

在C语言中，有一些处理错误的常用方法。例如，使用C标准库的assert()宏作为出错处理的方法。在开发过程中，使用这个宏进行必要的条件检测，项目完成后用#define NDEBUG使之失效，以便推出产品。随着程序规模的扩大，使用宏来进行出错处理的复杂性也在增加。

如果在当前的上下文环境中，程序员可以确切地掌握每一个具体步骤的运行结果，出错处理就变得十分明确和容易了。若错误问题发生时在一定的上下文环境中得不到足够的信息，则需要从更大的上下文环境中提取出错处理信息。C语言处理这类情况通常有三种典型的方法。

①出错信息可通过函数的返回值获得。如果函数返回值不能用，则可设置一全局错误判断标志（标准C语言中errno()和 perror()函数支持这一方法）。由于对每个函数调用都进行错误检查十分繁琐，并增加了程序的混乱度，程序设计者可能简单地忽略这些出错信息。另外，来自偶然出现异常的函数的返回值可能并不能提供什么有价值的信息。

②可使用C标准库中一般不太常用的信号处理系统，利用signal()函数（判断事件发生的类型）和raise()函数（产生事件）。由于信号产生库的使用者必须理解和安装合适的信号处理系统，所以使用这两个函数进行出错处理时应紧密结合各信号产生库。对于大型项目而言，不同库之间的信号可能会产生冲突。

③使用C标准库中非局部的跳转函数：setjmp()和longjmp()。setjmp()函数可在程序中存储一典型的正常状态，如果程序发生错误，longjmp()可恢复setjmp()函数的设定状态，从而实现goto语句无法实现的“长跳转”。事先被存储的地点在恢复时，可以得知是从哪里跳转过来，也就是说，可以确定错误发生的地点。下面是一个使用setjmp()和longjmp()实现“长跳转”的例子。

```
/// Program 10.1-1 ///
#include <iostream>
#include <setjmp.h>
```

```

using namespace std;
class game{
public:
    game(){cout << "game()" << endl;}
    ~game(){cout << "~game()" << endl;}
};

jmp_buf zhashan;
void TZ(){
    game GM;
    for(int i=0;i<3;i++)
        cout << "There is no interesting game." << endl;
    longjmp(zhashan,47);
}
int main(){
    if(setjmp(zhashan)==0){
        cout << "one,two, three ..." << endl;
        TZ();
    }
    else
        cout << "It's fantastic!" << endl;
}
/// End of Program 10.1-1 /////

```

用djgpp进行编译，程序的执行结果是：

```

one,two, three ...
game()
There is no interesting game.
There is no interesting game.
There is no interesting game.
It's fantastic!

```

`setjmp()`是一个特别的函数，如果直接调用它，它就把当前进程状态的所有相关信息存放在`jmp_buf`类型的变量中，并返回零。这样，它的行为像通常的函数。然而，如果使用同一个`jmp_buf`调用`longjmp()`，控制流就会回到`setjmp()`执行的地方，并且将`longjmp()`的第二个参数作为调用`setjmp()`的返回值。如果程序中有多个不同的`jmp_buf`，就可以弹出程序的多个不同位置的信息。

在上面的程序中，可以看到，控制流从函数TZ()内部跳转到main()函数内部时，函数TZ()内的局部对象GM的析构函数没有被调用。这正是在C++程序中使用`setjmp()`和`longjmp()`所必须认真对待的问题。由于C语言的信号处理技术和`setjmp/longjmp`函数不能调用析构函数，所以对象不能被正确地清除。这可能导致更为严重的问题。

所以，采用上述的第一种方法，存在繁琐和破坏程序结构的问题，采用上述第二、第三种方法，实际上不可能有效正确地从异常情况中恢复出来。因此，必须考虑新的出错处理机

制, C++中的“异常处理”就是在这个背景下产生的。

注意: 在VS 2005下编译并运行Program 10.1-1, 可以发现对象GM的析构函数被正确调用了。这是Visual C++编译器为了使setjmp()和longjmp()能够适用于C++程序而做的改进, 并不能保证在其他编译器中都有这样的结果。

10.2 抛出异常和传递参数的不同

从语法上看, 在C++的异常处理机制中, 在catch子句中声明参数与在函数里声明参数几乎没有差别。例如, 定义了一个名为stuff的类, 那么可以有如下的函数声明:

```
void f1(stuff w);
void f2(stuff& w);
void f3(const stuff& w);
void f4(stuff *pw);
void f5(const stuff *pw);
```

同样地, 在特定的上下文环境中, 可以利用如下的catch语句来捕获异常对象:

```
catch (stuff w)
catch (stuff& w)
catch (const stuff& w)
catch (stuff *pw)
catch (const stuff *pw)
```

因此, 初学者很容易认为用throw抛出一个异常到catch子句中与通过函数调用传递一个参数两者基本相同。这里面确有一些相同点, 但是他们也存在着巨大的差异。

首先谈相同点。传递函数参数与异常的途径可以是传值、传引用或传指针, 这是相同的。但是在传递参数和异常的过程中, 系统所要完成的操作则是完全不同的。产生这个差异的原因是: 调用函数时, 程序的控制权最终还会返回到函数的调用处, 但是当抛出一个异常时, 控制权永远不会回到抛出异常的地方。考察下面的程序。

```
/// Program 10.2-1 ///
#include <iostream>
using namespace std;
class stuff{
    int n;
    char c;
public:
    void Addr(){
        cout << this << endl;
    }
    friend istream& operator>>(istream&, stuff&);
};
istream& operator>>(istream& s, stuff& w){
    w.Addr();
}
```

```

    cin >> w.n;
    cin >> w.c;
    cin.get();
    return s;
}

void passAndThrow(){
    stuff localStuff;
    localStuff.Addr();
    cin >> localStuff;      //传递localStuff到operator>>
    throw localStuff;       //抛出localStuff异常
}

int main(){
    try{
        passAndThrow();
    }
    catch(stuff &w){
        w.Addr();
    }
}

/// End of Program 10.2-1 ///

```

在执行输入操作时，实参localStuff是以传引用的方式进入函数operator >>，形参变量w接受的是localStuff的地址，任何对w的操作实际上都施加到localStuff上。在随后的抛出异常的操作中，尽管catch子句捕捉的是异常对象的引用，但是捕捉到的异常对象已经不是localStuff，而是它的一个拷贝。原因是，throw语句一旦执行，函数passAndThrow()的执行也将结束，localStuff对象将被析构从而结束其生命期。如果把localStuff本身(而不是它的拷贝)传递给catch子句，这个子句接收到的只是一个被析构了的对象，一个localStuff的“尸体”。这是无法使用的。因此C++规范要求被作为异常抛出的对象必须被复制。

程序的执行结果是：

0012FE68

0012FE68

2 a

0012FD98

可以看到，在operator >>()函数中接收输入的对象w与localStuff有相同的地址（是同一个对象），而在catch子句中捕捉到的异常对象的地址与localStuff不同，它只是localStuff的一个拷贝。

即使被抛出的对象不会被释放，也会进行拷贝操作。例如，如果passAndThrow函数声明localStuff为静态变量(static)，即：

```

void passAndThrow(){
    static stuff localStuff;
    localStuff.Addr();
}

```

```

    cin >> localStuff;
    throw localStuff;
}

```

当抛出异常时仍将复制出localStuff的一个拷贝。这表示即使通过引用来捕获异常，也不能在catch块中修改localStuff，仅仅能修改localStuff的拷贝。对异常对象进行强制复制拷贝，这个限制有助于我们理解参数传递与抛出异常的第二个差异：抛出异常运行速度比参数传递要慢。

当异常对象被拷贝时，拷贝操作是由对象的拷贝构造函数完成的。该拷贝构造函数是对象的静态类型(static type)所对应类的拷贝构造函数，而不是对象的动态类型(dynamic type)对应类的拷贝构造函数。

```

/// Program 10.2-2 ///
#include <iostream>
using namespace std;
class stuff{
    int n;
    char c;
public:
    stuff(){
        n=c=0;
    }
    stuff(stuff&){
        cout << "stuff's copy constructor invoked" << endl;
        cout << this << endl;
    }
};
class SpecialStuff:public stuff{
    double d;
public:
    SpecialStuff(){
        d=0.0;
    }
    SpecialStuff(SpecialStuff&){
        cout << "SpecialStuff's copy constructor invoked" << endl;
        Addr();
    }
    void Addr(){
        cout << this << endl;
    }
};
void passAndThrow(){

```

```

SpecialStuff localStuff;
localStuff.Addr();
stuff &sf=localStuff;
cout << &sf << endl;
throw sf; //抛出stuff类型的异常
}
int main(){
try{
    passAndThrow();
}
catch(stuff &w){
    cout << "caught" << endl;
    cout << &w << endl;
}
}
/// End of Program 10.2-2 ///

```

这里抛出的是stuff类型的异常，也就是说，调用的是类stuff的拷贝构造函数，即使sf引用的是一个SpecialStuff类型的对象。编译器会根据被抛出对象（在这里是sf）的静态类型(static type)stuff，而不是SpecialStuff来创建相应的异常对象。这与其他情况下C++中拷贝构造函数的行为是一致的。

以上程序的运行结果是：

```

0012FE60
0012FE60
stuff's copy constructor invoked
0012FD84
caught
0012FD84

```

可以看到，sf和localStuff的地址是一样的，这体现了引用的作用：把一个SpecialStuff类型的对象当做stuff类型的对象使用。localStuff被拷贝生成被抛出的异常对象，catch语句中获取的是异常对象的引用，所以通过类stuff的拷贝构造函数产生的对象（异常对象）与在catch块中使用的对象w是同一个对象，因为它们具有相同的地址0012FD84。

在上面的程序中，将catch子句做一个小的修改，变成：

```
catch(stuff w){...}
```

程序的输出结果就变成：

```

0012FE5C
0012FE5C
stuff's copy constructor invoked
0012FD80
stuff's copy constructor invoked
0012FF4C

```

caught
0012FF4C

可以看到，类stuff的拷贝构造函数被调用了2次。这是因为localStuff通过拷贝构造函数传递给异常对象，而异常对象又通过拷贝构造函数传递给catch子句中的对象w。实际上，抛出异常时生成的异常对象是一个临时对象，它以一种程序员不可见的方式在发挥作用。异常对象在被重新抛出时，有两种不同的选择。请看下面的程序。

```
/// Program 10.2-3 ///
#include <iostream>
using namespace std;
class stuff{
    int n;
    char c;
public:
    stuff(){
        n=c=0;
    }
    stuff(stuff&){
        cout << "stuff's copy constructor invoked" << endl;
    }
};
```

void passAndThrow(){
 stuff localStuff;
 throw localStuff; //抛出stuff类型的异常
}

```
void rethrow1(){
    try{
        passAndThrow();
    }
    catch(stuff &w){
        cout << &w << endl;
        throw w; //将异常对象重新抛出
    }
}
```

void rethrow2(){
 try{
 passAndThrow();
 }
 catch(stuff &w){
 cout << &w << endl;
 throw; //将异常对象重新抛出
 }
}

```

    }
}

int main(){
    try{
        rethrow1();
    }
    catch(stuff &w){
        cout << "caught rethrow1" << endl;
        cout << &w << endl;
    }
    try{
        rethrow2();
    }
    catch(stuff &w){
        cout << "caught rethrow2" << endl;
        cout << &w << endl;
    }
}

```

/// End of Program 10.2-3 ///

函数rethrow1()和函数rethrow2()几乎没有什么不同，只是在函数rethrow2()中，throw后面多了一个w。但从程序的运行结果来看，它们的执行过程却有很大差别。程序的输出是：

```

stuff's copy constructor invoked
0012FC98
caught rethrow1
0012FC98
stuff's copy constructor invoked
0012FC88
stuff's copy constructor invoked
caught rethrow2
0012FD80

```

在函数rethrow1()中，被重新抛出的是当前捕获的异常对象，因而没有发生拷贝构造函数的调用。而在rethrow2()中，在catch块中重新抛出的是当前捕获异常的一个新的拷贝。如果是将当前异常(current exception)重新抛出，那么无论该异常是什么类型，新抛出的异常都保持与原来的一致（它们在物理上就是同一个对象）。如果抛出的是原异常的一个拷贝，则对象的动态数据类型可能发生变化。一般来说，应该用throw来重新抛出当前的异常，因为这样不会改变被传递出去的异常类型，而且更有效率，因为不用生成一个新拷贝。

抛出一个异常对象之后，可以用三种不同的方式来捕捉它。例如，抛出一个stuff类型的对象，那么可以使用：

catch (stuff w)	//通过传值捕获异常
catch (stuff& w)	//通过传递引用捕获异常

```
catch (const stuff& w) //通过传递常引用捕获异常
```

在这里又可以发现传递参数与传递异常的另一个差异。一个被异常抛出的对象(总是一个临时对象)可以通过普通的引用捕获，它不需要通过指向const对象的引用(reference-to-const)捕获。在函数调用中不允许传递一个临时对象到一个非const引用类型的参数里，但是在异常中却被允许。

当用传值的方式传递函数的参数时，会制造被传递对象的一个拷贝，并把这个拷贝存储到函数的参数里。当通过传值的方式传递一个异常时，也是这么做的。例如下面的语句：

```
catch (stuff w) ... //通过传值捕获异常
```

会建立两个被抛出对象的拷贝，一个是所有异常都必须建立的临时对象，第二个是把临时对象拷贝进w中。同样，当通过引用捕获异常时，如使用下面的语句：

```
catch (stuff& w) //通过引用捕获
```

```
catch (const stuff& w) //也通过引用捕获
```

这仍旧会建立一个被抛出对象的拷贝：拷贝是一个临时对象。相反当通过引用传递函数参数时，没有进行对象拷贝。当抛出一个异常时，系统构造的(以后会析构掉)被抛出对象的拷贝数比以相同对象作为参数传递给函数时构造的拷贝数要多一个。

对象从函数的调用处传递到函数参数里与从异常抛出点传递到catch子句里所采用的方法不同，这只是参数传递与异常传递的区别的一个方面，第二个差异是在函数调用者或抛出异常者与被调用者或异常捕获者之间的类型匹配的过程不同。考察下面的程序。

```
//// Program 10.2-4 ////
```

```
#include <iostream>
#include <math.h>
using namespace std;
void throwint(){
    int i=5;
    throw i;
}
double Sqrt(double d){
    return sqrt(d);
}
int main(){
    int i=5;
    cout << "sqrt(5)=" << Sqrt(i) << endl;
    try{
        throwint();
    }
    catch(double){
        cout << "caught" << endl;
    }
    catch(...){
        cout << "not caught" << endl;
    }
}
```

```

    }
}

/// End of Program 10.2-4 ///

```

C++允许进行从int到double的隐式类型转换，所以函数调用Sqrt(i)中，i被悄悄地转变为double类型，并且其返回值也是double。一般来说，catch子句匹配异常类型时不会进行这样的转换。所以程序的输出结果是：

```

sqrt(5)=2.23607

```

从上面的输出结果可以看出，异常类型没有被正确地捕获，而是直接输出了。这说明在try块中抛出的int异常不会被处理double异常的catch子句捕获。该子句只能捕获真正为double类型的异常，不进行类型转换。因此如果要想捕获int异常，必须使用带有int或int&参数的catch子句。

不过，在catch子句中进行异常匹配时可以进行两种类型转换：第一种是继承类与基类间的转换。一个用来捕获基类的catch子句也可以处理派生类类型的异常。这种派生类与基类间的异常类型转换可以作用于数值、引用以及指针上。第二种是允许从一个类型化指针(typed pointer)转变成无类型指针(untyped pointer)，所以带有const void* 指针的catch子句能捕获任何类型的指针类型异常。下面是一个在catch子句中允许类型转换的例子。

```

/// Program 10.2-5 ///
#include <iostream>
using namespace std;

class stuff{
    int n;
    char c;
public:
    stuff(){
        n=c=0;
    }
    stuff(stuff&){
        cout << "stuff's copy constructor invoked" << endl;
        cout << this << endl;
    }
};

class SpecialStuff:public stuff{
    double d;
public:
    SpecialStuff(){
        d=0.0;
    }
    SpecialStuff(SpecialStuff&){
        cout << "SpecialStuff's copy constructor invoked" << endl;
        Addr();
    }
};

void Addr(){
    cout << "Address of SpecialStuff object: " << this << endl;
}

```



```

}

void Addr(){
    cout << this << endl;
}

void passAndThrow(){
    SpecialStuff localStuff;
    localStuff.Addr();
    throw localStuff; //抛出localStuff异常
}

void throwPointer(){
    static int i=4;
    throw &i;
}

int main(){
    try{
        passAndThrow();
    }
    catch(stuff &w){
        cout << "catched" << endl;
        cout << &w << endl;
    }
    try{
        throwPointer();
    }
    catch(void* p){
        cout << "caught pointer is " << p << endl;
    }
}
/// End of Program 10.2-5 ///

```

程序的运行结果是：

0012FE54
SpecialStuff's copy constructor invoked

0012FD7C
catched

0012FD7C
caught pointer is 0041A058

在第一个try块中，抛出的对象为SpecialStuff类型，而在catch子句中是用stuff类型的引用来捕捉的，这种转换在C++中是允许的。输出结果显示在catch子句中捕捉到的异常对象就是通过类SpecialStuff的拷贝构造函数创建的临时对象。在第三个try块中，抛出的是int*类型的

指针，它被catch (void*)子句捕捉到。

传递参数和传递异常间最后一点差别是：catch子句匹配顺序总是取决于它们在程序中出现的顺序。因此一个派生类异常可能被处理其基类异常的catch子句捕获，即使同时存在有能力处理该派生类异常的catch子句与相同的try块相对应。考察下面的程序。

/// Program 10.2-6 ///

```
#include <iostream>
using namespace std;
class stuff{
    int n;
    char c;
public:
    stuff():n(0),c('0){}
};

class SpecialStuff:public stuff{
    double d;
public:
    SpecialStuff():d(0.0){}
};

int main(){
    SpecialStuff localStuff;
    try{
        throw localStuff; //抛出SpecialStuff类型的异常
    }
    catch(stuff &){
        cout << "stuff caught" << endl;
    }
    catch(SpecialStuff&){
        cout << "SpecialStuff caught" << endl;
    }
}
/// End of Program 10.2-6 ///
```

这个程序的输出结果是：

stuff caught

程序中被抛出的对象是SpecialStuff类型的，本应由catch(SpecialStuff&)子句所捕获，但由于前面有一个catch(stuff &)，而在类型匹配时是允许在派生类和基类之间进行转换的，所以最终是由前面的catch子句将异常捕获。不过，这个程序在逻辑上多少存在一些问题，因为在前面的catch子句实际上阻止了后面的catch子句捕获异常。所以，当有多个catch子句对应同一个try块时，应该把捕获派生类对象的catch子句放在前面，而把捕获基类对象的catch子句放在后面。否则，代码在逻辑上是错误的，而且编译器会发出警告。

与上面这种行为相反，当调用一个虚拟函数时，被调用的函数是由发出函数调用的对象



的动态类型(dynamic type)决定的。所以说，虚拟函数采用最优适合法，而异常处理采用的是最先适合法。

综上所述，把一个对象传递给函数（或一个对象调用虚拟函数）与把一个对象作为异常抛出，这之间有三个主要区别。第一，把一个对象作为异常抛出时，总会建立该对象的副本。当通过传值方式捕获时，对象被拷贝了两次。对象作为参数传递给函数时，不需要进行额外的拷贝；第二，对象作为异常被抛出与作为参数传递给函数相比，前者允许的类型转换比后者要少（前者只有两种转换形式）；最后一点，catch子句进行异常类型匹配的顺序是它们在源代码中出现的顺序，第一个类型匹配成功的catch将被用来执行。当一个对象调用一个虚拟函数时，被选择的函数位于与对象的动态类型匹配最佳的类里，与该类在源代码中出现的位置无关。

10.3 抛出和接收异常的顺序

异常(exception)是C++语言引入的错误处理机制。它采用统一的方式对程序的运行时错误进行处理，具有标准化、安全和高效的特点。C++为了实现异常处理，引入了三个关键字：try、throw、catch。异常由throw抛出，格式为throw [expression]，由catch捕捉。try语句块是可能抛出异常的语句块，它通常和一个或多个catch语句块连续出现。try语句块和catch语句块必须相互配合，以下三种情况都会导致编译错误：

- ①只有try语句块而没有catch语句块；或者只有catch语句块而没有try语句块；
- ②在try语句块和catch语句块之间夹杂有其他语句；
- ③当try语句块后跟有多个catch语句块时，catch语句块之间夹杂有其他语句。

通常情况下，一个try语句块中抛出的异常，应该由紧随其后的几个catch语句块中的某个所捕捉并处理。例如下面的程序。

```
//// Program 10.3-1 ////
#include <iostream>
using namespace std;
class ExClass{
    int num;
public:
    ExClass(int i){
        cout << "Constructing exception object with num=" << i << endl;
        num=i;
    }
    ExClass(ExClass &e){
        cout << "After copy constructor num=" << e.num+1 << endl;
        num=e.num+1;
    }
    ~ExClass(){
        cout << "Destructing exception object with num=" << num << endl;
    }
}
```

```

void show(){
    cout << "The number is " << num << endl;
}
};

void NormalCatch(){
    ExClass obj1(99);
    ExClass obj2(199);
    try{
        throw obj1; //导致输出：After copy constructor num=100
    }
    catch(double f){
        cout << "Exception caught" << endl;
    }
    catch(ExClass e){ //导致输出：After copy constructor num=101
        e.show();
        cout << "Exiting NormalCatch()" << endl;
    }
}

int main(){
    NormalCatch();
    cout << "After NormalCatch()" << endl;
}
/// End of Program 10.3-1 ///

```

程序的输出结果是：

```

Constructing exception object with num=99
Constructing exception object with num=199
After copy constructor num=100
After copy constructor num=101
The number is 101
Destructing exception object with num=101
Destructing exception object with num=100
Exiting NormalCatch()
Destructing exception object with num=199
Destructing exception object with num=99
After NormalCatch()

```

用throw语句抛出一个对象时，会构造一个新的对象，这个对象就是异常对象。该对象的生命期从被抛出时开始计算，一直到被某个catch语句块捕捉，就会在该catch语句块执行完毕后被销毁。在上面的程序中，异常对象的num值为100，“Destructing exception object with num=100”这句话在“Exiting NormalCatch()”之前输出，正好说明异常对象的销毁时间是在它被捕获的catch块执行之后。

所有的catch分支在执行时类似一次函数调用，catch后面的参数相当于函数的形参，而被抛出的异常对象相当于函数调用时的实参。当形参与实参成功匹配时，就说异常被某个catch分支所捕捉。catch后面的参数只能采用传值和传引用两种形式，如果采用传值方式，则会生成实参的一个副本，如果实参是一个对象，就会导致构造函数被调用。在上面的程序中，执行catch(ExClass e)语句就是利用异常对象构造一个对象e，因此会调用复制构造函数。要注意的是：同一种数据类型的传值catch分支和传引用catch分支不能同时出现。

在某些情况下，可能所有的catch分支都无法捕捉到抛出的异常，这将导致当前函数执行的结束，并返回到主调函数中。在主调函数中，将继续以上的捕捉异常的过程，直到异常被捕获或最终结束整个程序的运行。考察下面的程序。

/// Program 10.3-2 ///

```
#include <iostream>
using namespace std;
class ExClass{
    int num;
public:
    ExClass(int i){
        cout << "Constructing exception object with num=" << i << endl;
        num=i;
    }
    ExClass(ExClass &e){
        cout << "After copy constructor num=" << e.num+1 << endl;
        num=e.num+1;
    }
    ~ExClass(){
        cout << "Destructing exception object with num=" << num << endl;
    }
    void show(){
        cout << "The number is " << num << endl;
    }
};
void ThrowExFunc()
{
    try{
        throw ExClass(199);
    }
    catch(double f){
        cout << "Exception caught!" << endl;
    }
    cout << "After catch statement" << endl;
}
```

```

int main(){
    try{
        ThrowExFunc();
    }
    catch(ExClass e){
        e.show();
    }
    catch(...){
        cout<<" All will fall in" << endl;
    }
    cout<<"Continue to execute" << endl;
}
/// End of Program 10.3-2 /////

```

程序的输出结果是：

```

Constructing exception object with num=199
After copy constructor num=200
The number is 200
Destructing exception object with num=200
Destructing exception object with num=199
Continue to execute

```

从程序的输出结果来看，被抛出的异常对象的num值为199，由于它没有在函数ThrowExFunc()中被捕捉，所以它导致了ThrowExFunc()的执行结束（否则会输出：After catch statement）。在main()函数中，catch(ExClass e)捕获了异常对象，通过复制构造函数产生对象e，e的num值为200，catch语句块运行结束后，对象e首先被销毁，紧接着销毁异常对象。这之后，程序继续运行，输出：Continue to execute。

catch(...)的意思是可以捕获所有类型的异常。不提倡随意地使用catch(...)，因为这会导致程序员对异常类型的不精确处理，并降低程序的运行效率。但在程序开发阶段，catch(...)还是有用的：如果在精心安排异常捕捉之后，还是进入了catch(...)语句块，说明前面的代码存在缺陷，需要进一步改正。

在捕捉异常对象时，还可以采用传引用的方式，例如把catch语句写成catch(ExClass &e)，这样可以不必产生异常对象的副本，减少程序的运行开销，提高执行效率。

在抛出异常时，还可以抛出一个指针。当然这种做法并不总是安全的。如果要确保安全，应该将指向全局（静态）对象的指针或指向动态申请的空间的指针抛出，或者被抛出的指针在本函数内被捕获。否则，利用一个被抛出的指向已经被“销毁”的对象的指针，要格外注意。最好是不要用，如果实在要用，首先，必须保证对象的析构函数不能对对象的内容作损伤性的修改，其次，对象的空间没有被其他新产生的变量覆盖。也就是说，尽管对象被释放，但它的有效内容依然保留在栈中。

10.4 在构造函数中抛出异常

在构造函数中抛出异常，在概念上将被视为该对象没有被构造完成，因此当前对象的析构函数不会被调用。同时，由于构造函数本身也是一个函数，在函数体内抛出异常将导致当前函数运行的结束，并释放已经构造的局部对象。由于构造一个对象时，先要执行其直接基类的构造函数和它的成员对象的构造函数，所以，在构造函数中抛出异常将导致销毁当前对象的动作，也就是说，要执行其直接基类和成员对象的析构函数。考察下面的程序。

```
//// Program 10.4-1 /////
#include <iostream>
using namespace std;
class C{
    int m;
public:
    C(){cout<<"in C constructor"<<endl;}
    ~C(){cout<<"in C destructor"<<endl;}
};
class A{
public:
    A(){cout<<"in A constructor"<<endl;}
    ~A(){cout<<"in A destructor"<<endl;}
};
class B: public A{
public:
    C c;
    B(){cout<<"in B constructor"<<endl;throw -1;}
    ~B(){cout<<"in B destructor"<<endl;}
};
int main(){
    try{
        B b;
    }
    catch(int){
        cout<<"catched"<<endl;
    }
}
/// End of Program 10.4-1 ///
```

该程序的执行结果是：

in A constructor

```
in C constructor  
in B constructor  
in C destructor  
in A destructor  
caught
```

在构造B类的对象b的时候，先要执行其直接基类A的构造函数，再执行其成员对象c的构造函数，然后才进入类B的构造函数。由于在类B的构造函数中抛出了异常，而此异常并未在构造函数中被捕获，所以导致类B的构造函数的执行中断，对象b并未构造完成。在类B的构造函数“回滚”的过程中，c的析构函数和类A的析构函数相继被调用。最后，由于b并没有被成功构造，所以main()函数结束时，并不会调用b的析构函数。也就是说，对象b是中途“夭折”的。

10.5 用传引用的方式捕捉异常

前面介绍过，用catch语句捕捉异常相当于一次函数调用。被抛出的异常对象与catch后的参变量之间存在一个匹配的问题。当采用传引用的方式捕捉异常时，还有可能发生虚函数调用。下面是一个例子。

```
//// Program 10.5-1 ////  
#include <iostream>  
using namespace std;  
class base{  
public:  
    virtual void what(){  
        cout << "base" << endl;  
    }  
};  
class derived: public base{  
public:  
    void what(){  
        cout << "derived" << endl;  
    }  
};  
void f(){  
    throw derived();  
}  
int main(){  
    try{  
        f();  
    }  
    catch(base b){
```

```

    b.what();
}
try{
    f();
}
catch(base& b){
    b.what();
}
}

/// End of Program 10.5-1 ///

```

此程序的运行结果是：

```
base
derived
```

函数f()抛出了一个derived类型的对象，该对象在被catch(base b)捕捉的时候采用的是传值方式，即将derived类型的异常对象通过复制构造函数产生base类型的对象b，这一过程在某些资料中称为“切片”。这时b是一个有名对象，b.what()是实调用，函数调用是在编译阶段决定的。如果用catch(base& b)来捕捉异常，则会将引用b与异常对象进行绑定，由于class base中有一个名为what()的虚函数，所以b.what()是一个虚调用，它的执行版本要到运行时才能决定。

10.6 在堆栈展开时如何防止内存泄漏

堆栈展开（Stack Unwinding）是指，如果在一个函数内部抛出异常，而此异常并未在该函数内部被捕获，就将导致该函数的运行在抛出异常处结束；所有已经分配在栈上的局部变量都要被释放。如果被释放的变量中有指针，而该指针此前已经用new运算申请了空间，就有可能导致内存泄漏。因为堆栈展开的时候并不会自动对指针变量执行delete（或delete[]）操作。

因此，在有可能发生异常的函数中，可以利用“智能指针”auto_ptr来防止内存泄漏。见下面的例子。

```

/// Program 10.6-1 ///
#include <iostream>
#include <memory>
using std::cout;
using std::endl;
using std::auto_ptr;
class A{
    int num;
public:
    A(int i):num(i){
        cout << "this is A's Constructor,num=" << num << endl;
    }
}
```

```

~A(){
    cout << "this is A's Destructor,num=" << num << endl;
}
void show(){
    cout << num << endl;
}
};

void autoptrtest10{
    A *pa = new A(1);
    throw 1;
    delete pa;
}

void autoptrtest20{
    auto_ptr<A> pa(new A(2));
    pa->show();
    throw 2;
}

int main(){
    try{
        autoptrtest1();
    }
    catch(int){
        cout << "there is no destructor invoked" << endl;
    }
    cout << endl;
    try{
        autoptrtest2();
    }
    catch(int){
        cout << "A's destructor does be invoked" << endl;
    }
}

```

//// End of Program 10.6-1 ////

程序的运行结果是：

this is A's Constructor,num=1
there is no destructor invoked

this is A's Constructor,num=2

2

this is A's Destructor,num=2

A's destructor does be invoked

在解读上面这段程序的时候，注意这样几个要点：

- ①在函数autoptest1()中，由于异常的发生，导致delete pa;无法执行，从而导致内存泄漏。
- ②auto_ptr实际上是一个类模板，在名称空间std中定义。要使用该类模板，必须包含头文件memory。auto_ptr的构造函数可以接收任何类型的指针，实际上是利用指针类型将该类模板实例化，并将传入的指针保存在auto_ptr<T>对象中。

③在堆栈展开的过程中，auto_ptr<T>对象会被释放，从而导致auto_ptr<T>对象的析构函数被调用。在该析构函数中，将使用delete运算符保存在该对象内的指针所指向的动态对象销毁。这样，就不会发生内存泄漏了。

④由于已经对*和->操作符进行了重载，所以可以像使用普通的指针变量那样使用auto_ptr<T>对象，如上面程序中的pa->show()。这样可以保留使用指针的编程习惯，方便程序员编写和维护程序。

从上面的分析可以看出，通过使用auto_ptr，可以有效地避免内存泄漏，同时也可以很好地解决指针的管理问题。

第11章 程序开发环境与实践

11.1 关于开发环境

要编写C++程序，就一定要和开发环境打交道。开发环境最重要的是提供编译器和标准库，以便程序员能够顺利地完成编辑、编译、连接、调试等任务。

最常用的开发环境是GNU开发环境和微软的Visual Studio开发环境，它们的典型代表是32位的DJGPP和Visual Studio 2005，两款能够在Windows操作系统下运行的C++开发环境。

(1) DJGPP介绍

DJGPP提供了一套在Windows环境下的程序开发环境，包含了gcc, gpp等编译器，同时还包含了RHIDE集成开发工具。

DJGPP是一套移植自UNIX操作系统下赫赫有名的GNU C/C++与GNU development tools的32-bit保护模式程序开发环境，主要的平台是针对在Intel 32-bit CPU (386以上，不含286)下的MS-DOS或其他相容OS (如OS2, Windows等)。DJGPP整个移植计划都是由DJ Delorie<dj@delorie.com>及其他志愿者负责统筹。

自2.0版之后，DJGPP环境全面采用DPMI，摆脱了以往需加挂extender(go32.exe)的方式(Watcom C/C++就得加挂DOS4GW.EXE)。所以，只要有DPMI Server(如OS/2, Win31, Win95中的DOS BOX 及MS-DOS下的QEMM)就可以运行。另外，DJGPP也内附一个DPMI Server (cwsdpmi)，可以提供32-bit, 4GByte的平滑模式寻址空间和最高达256 MB的虚拟内存，以便在没有DPMI Server的情况下使用。

DJGPP最初是作为UNIX上的开发环境移植到MS-DOS操作系统下来，但至今已经发展成独立的MS-DOS程序的强力开发环境。而且，它是免费的！

可从以下地址下载DJGPP：<http://www.delorie.com/djgpp/>。由于DJGPP现在已经发展成一个比较大的系统，所以，对于初学者或者是希望使用其基本功能的用户来说，可以选择基本安装。先将以下文件拷到同一个目录下：

```
unzip32.exe
v2/copying.dj
v2/djdev203.zip
v2/faq230b.zip
v2/readme.1st
v2/apps/rhid15ab.zip
v2gnu/bnu217b.zip
v2gnu/gcc422b.zip
v2gnu/gdb611b.zip
```

v2gnu/gpp422b.zip

v2gnu/mak3791b.zip

v2gnu/txi411b.zip

然后将所有压缩文件解压到安装目录（如C:\djgpp），这样就完成了基本安装。在安装目录下创建一个djgpp.bat，将以下内容置于其中：

```
@echo off
set PATH=c:\djgpp\bin;%PATH%
set DJGPP=c:\djgpp\djgpp.env
```

每次使用DJGPP之前，先运行djgpp.bat，就可以利用它进行程序开发了。对于单个C++源文件（如test.cpp），可以用如下命令完成编译和连接：

gxx test.cpp

这时可生成可执行文件a.exe。如果在编译时，想指定输出文件的名字，可以使用如下命令：

gxx -o test.exe test.cpp

当然，也可以先生成OBJ文件，然后再连接。即连续使用如下两条命令：

gcc -c test.cpp

gxx -o test.obj test.o

如果整个项目由多个源文件组成（假设为test1.cpp, ..., testn.cpp），那么可以直接将它们编译成可执行文件：

gxx test1.cpp test2.cpp ... testn.cpp

也可以分别将每个源文件编译成目标文件（OBJ文件），然后再用gxx连接，即：

gxx -o test.exe test1.o test2.o ... testn.o

有关DJGPP编译器的更为详细的用法，请参阅DJGPP系统的帮助文档。

(2) Visual Studio 2005介绍

Visual Studio 2005是一套完整的开发工具集，用于生成 ASP.NET Web 应用程序、XML Web Services、桌面应用程序和移动应用程序。Visual Basic、Visual C++、Visual C# 和 Visual J# 全都使用相同的集成开发环境 (IDE)，利用此 IDE 可以共享工具且有助于创建混合语言解决方案。另外，这些语言利用了 .NET Framework 的功能，通过此框架可使用简化 ASP Web 应用程序和 XML Web Services 开发的关键技术。

Visual Studio 2005环境下的C++分为托管C++和非托管C++，前者的运行环境是公共语言运行时 (CLR)，而后者则是直接生成本地代码。

本书当中的例子都是使用非托管C++。开发一个具体项目的步骤如下：

- ①启动Microsoft Visual Studio 2005。
- ②创建一个新的项目，项目类型为：Visual C++→Win32→Win32控制台应用程序。
- ③打开项目属性，在配置属性→C/C++→预编译头→创建/使用预编译头中，选择“不使用预编译头”，并将stdafx.h和stdafx.cpp从项目中移除。
- ④如果操作系统使用的文件系统格式为FAT32，则在配置属性→清单工具→常规→使用FAT32解决办法中，选择“是”。
- ⑤编辑源文件，选择生成调试版 (Debug) 执行文件，进行调试。
- ⑥确认程序功能已经实现且没有其他错误，生成发行版 (Release) 执行文件。



(3) 为什么要使用两套编译器

本书的例子程序都在VS 2005环境下调试通过，但在讲解的过程中经常要提及DJGPP。在学习标准C++的过程当中，使用这样两套编译器是有好处的。主要原因是：

①VS 2005开发环境功能虽然强大，但要全面掌握它却需要一个过程。而且，C/C++开发环境最初都只有命令行方式的，即使是像VS 2005这样的开发环境，其底层基础仍然是命令行方式的编译器。所以，在全面掌握VS 2005之前，先使用DJGPP，可以很快进入编程实践，而且可以了解一些被IDE工具包裹起来的开发细节。

②虽然C++有国际标准，但这个标准并没有规定实现C++编译器的所有细节，而且每个软件厂商生产出于各种考虑，在对C++标准的支持上会有不同程度的扩展。因此，使用多个编译器对于了解程序的“可移植性”的概念大有好处。应该鼓励程序员编写可移植性好的程序，避免过分依赖一种具体的编译器和在一些技术细节上过于“钻牛角尖”。这对于掌握像C++这样以技术和效率著称的语言尤其重要。

11.2 在 IDE 中调试程序时查看输出结果

IDE是Integrated Development Environment的首字母缩写，代表“集成开发环境”的意思。本书中的示例程序使用的IDE是Visual Studio 2005，是微软公司提供的一套开发工具。在实践中，程序的最终发行(release)版没有生成之前，都需要进行反复的调试。在VS 2005环境中，按下F5键、点击工具栏中的▲按钮或点选菜单项“调试(D)→启动调试(S)”，都可以开始调试程序。

调试程序的目的在于发现程序中的错误。而判断程序是否有错的第一步，是观察程序的输出结果是否为程序员所预期，然后再采用设置断点、观察中间变量等调试手段。在控制台程序中，可用printf()函数(传统C方式)或用cout(标准C++方式)进行输出。例如，我们调试下面的一个经典的入门程序。

```
/// Program 11.2-1 ///
#include <iostream>
using namespace std;
int main(){
    cout<<"Hello, world!"<<endl;
}
/// End of Program 11.2-1 ///
```

这个程序是很多C++教科书中第一个完整的程序，演示在标准输出设备(屏幕)上打印一句话。但我们在IDE中调试时，屏幕上的输出信息一闪就消失了，根本来不及看清楚程序的运行结果。所以，我们需要某种方法使程序“暂停”，然后在适当的时候再结束运行。最简单的办法是增加一条从标准输入设备(键盘)上读取一个字符的语句。把上述程序修改如下。

```
/// Program 11.2-2 ///
#include <iostream>
using namespace std;
int main(){
    cout<<"Hello, world!"<<endl;
```

```

    getchar();
}

/// End of Program 11.2-2 ///

```

这个程序在输出“Hello, world!”这句话后，就会等待用户输入一个字符。getchar()函数在C语言中就存在了。一般的用法如char c=getchar()，把从键盘输入的字符读入某个变量中。由于该函数在执行时是采用带缓冲的方式读入字符（包括读入回车），所以一直要等到按下回车之后才可以真正执行getchar()函数，并导致程序结束运行。一般情况下是直接按下回车键。

如果觉得使用getchar()不够“C++”，那么可用使用输入语句cin.get()，其作用和表现方式同getchar()完全一样。

以上两种方式，都只有在输入回车之后才能结束程序运行。如果想按任意键都退出程序，可使用system(“pause”);语句。system()是一个系统调用函数，它可以调用控制台（console）命令，而pause正是控制台中的暂停命令。

11.3 使用汇编语言

对于想深入学习C++语言的程序员来说，掌握一些汇编语言的知识是很有好处的。这并非是鼓励大家直接用汇编语言进行程序开发（毕竟那是一项难度很大的工作），而是为了如下两个目的：

①充分发挥汇编语言和C++语言各自的优势，在特定的场合配合工作，以达到特定的效果或最佳的效果。当然，即使是使用汇编语言，也大多采用内联汇编的形式。

②对于一些难以理解的语言机制，如C++的对象模型、引用的实现等等，如果能直接观察C++程序所对应的汇编代码，将给我们提供极大的帮助，使我们对各种语言现象有更深入的理解。

下面我们讨论几个与使用汇编语言相关的问题。

(1) 在VS 2005中查看汇编代码

有两种方式在VS 2005 IDE中查看汇编代码。一是在调试的同时查看，二是在编译时，除了生成可执行文件外，同时生成C++程序所对应的汇编代码，然后再打开asm文件进行查看。

在调试阶段查看汇编代码，通常需要两个步骤。首先是在需要查看汇编代码的C++语句处暂停下来，然后切换到汇编语言窗口进行查看。对于一个具体的项目来说，可以用单步调试的方式，一步一步运行到需要查看的语句，然后再切换到汇编语言窗口。也可以在需要查看的语句处设置断点，然后让程序直接运行到断点处暂停。

启动单步调试的操作步骤是：在解决方案资源管理器中，用鼠标右键点击项目名称，在弹出菜单中选择“调试→进入并单步执行新实例”。或者在主菜单中选择“调试→逐过程”或“调试→逐语句”，都可以进入单步调试状态。

在某条语句处暂停后，可以在主菜单上选择“调试→窗口→反汇编”，这样就可以看到C++程序所对应的汇编代码了。

如果想在编译的同时产生汇编源文件，可以在主菜单上选择“项目→属性→C/C++→输出文件→汇编输出→带源代码的程序集(/FAs)”，这样对程序进行编译后，除了得到可执行文件，还可以得到带源代码的asm文件（汇编语言源文件）。

需要注意的是，在两种方式下查看到的汇编代码并不是完全相同的。例如下面的程序。

```
/// Program 11.3-1 ///

```

```
#include <stdio.h>
```

```
void func(int i,int j){
```

```
    printf("%d %d\n",i,j);
```

```
}
```

```
int main(){
```

```
    int a=1,b=2;
```

```
    func(a,b);
```

```
}
```

```
/// End of Program 11.3-1 ///

```

函数调用func(a,b)在反汇编窗口中对应的汇编语句是：

```
mov     eax,dword ptr [b]
push    eax
mov     ecx,dword ptr [a]
push    ecx
call    func (4110E6h)
add    esp,8
```

而在编译输出的汇编语言文件中所对应的汇编代码是：

```
mov eax,DWORD PTR _b$[ebp]
push eax
mov ecx,DWORD PTR _a$[ebp]
push ecx
call ?func@@YAXHH@Z           ; func
add esp,8
```

这种差异主要体现在变量和函数的调用方式上。在调试阶段，汇编代码为了调试的需要做了一定的处理。在两种方式下查看到的汇编代码本质是一样的。

(2) 在VS 2005中使用内联汇编

使用内联汇编可以在C/C++代码中嵌入汇编语言指令，而且不需要额外的汇编和连接步骤。在Visual C++中，内联汇编直接由编译器支持，不需要配置诸如MASM一类的独立汇编工具。

内联汇编代码可以使用C/C++变量和函数，因此能够非常容易地整合到C/C++代码中。它能做一些对于单独使用C/C++来说非常笨重或不可能完成的任务。

使用内联汇编要用到`_asm`关键字，它可以出现在任何允许C/C++语句出现的地方。在书写形式上，`_asm`关键字有两种用法，如下面的例子所示：

```
_asm{
    MOV AL, 2
    MOV DX, 0xD007
    OUT AL, DX
}
```

或者：



```
_asm MOV AL, 2
_asm MOV DX, 0xD007
_asm OUT AL, DX
```

因为`_asm`关键字是语句分隔符，所以可以把多条汇编指令放在同一行：

```
_asm MOV AL, 2 _asm MOV DX, 0xD007 _asm OUT AL, DX
```

但不允许这样写：

```
_asm MOV AL, 2 MOV DX, 0xD007 OUT AL, DX
```

两条汇编语句之间必须换行或有`_asm`分隔符。

不像在C/C++中的“{}”，`_asm`块的“{}”不会影响C/C++变量的作用范围。同时，`_asm`块可以嵌套，而且嵌套也不会影响变量的作用范围。

为了与低版本的Visual C++兼容，`_asm`和`_asm`具有相同的意义。另外，Visual C++ 2005并不支持标准C++的`asm`关键字。要使用内联汇编，必须使用`_asm`而不是`asm`关键字。

虽然`_asm`块中允许使用C/C++的数据类型和对象，但它不能使用MASM指示符和操作符来定义数据对象。特别是，`_asm`块中不允许MASM中的定义指示符(DB、DW、DD、DQ、DT和DF)，也不允许使用DUP和THIS操作符。MASM中的结构和记录也不再有效，内联汇编不接受STRUC、RECORD、WIDTH或者MASK。

11.4 怎样调试C++程序

在讨论调试程序的方法之前，有一点必须强调：程序是设计出来的，而不是调试出来的。这是所有的程序员必须牢记的一条准则。一个没有设计或者设计得很糟糕的程序，无论怎样调试，也不会成为一个合格的程序。

在程序有着良好设计的前提下，发现编码当中的错误便成了调试的主要目的。一个C++程序可能出现的错误有两类：语法错误和逻辑错误。调试通常是指在消除了语法错误之后，发现程序中的逻辑错误的过程。对C++程序进行调试，有这样几种常用的手段。它们既可以单独使用，也可以配合使用。

(1) 使用打印输出语句

这是最朴素，也是最有效的方法。程序的运行可以看成是一组变量（状态）不断变化的过程，这个过程就是数据的处理过程。如果程序的最终结果不对，那么我们必须考察这一组状态什么时候出现了问题，而查看中间结果就成了一种最有效的手段。

因此，不要过分迷信功能强大的调试工具。在大部分情况下，程序的问题都是一些“小问题”。而正是这些小问题，却造成了大麻烦。一开始就希望依赖复杂的调试工具，反而有可能干扰程序员调试程序的思路。实际上，程序员对自己编写的不同代码的“信任”程度是有差异的，在认为有可能出错的代码附近，往往只要通过简单的`printf()`语句或`cout << ...`语句就可以发现出现异常的变量，继而找到问题的根源所在。

(2) 使用调试标记

在调试程序的时候使用相应的辅助代码（如输出中间结果等），在调试完成之后隐藏这些代码，是一种常用的调试策略。这种策略可以借助于`#define`、`#ifdef`和`#endif`这三个预编译指令实现。具体地说，在调试程序时，利用编译器的命令行参数定义调试标记（相当于在程序中用`#define`定义的宏），然后在`#ifdef`和`#endif`之间包含相应的调试代码就可以了。当程序最

经调试完成后，在生成发行版时，只要在编译器的命令行参数中不再提供调试标记，程序中的调试代码就会消失。常用的调试标记为_DEBUG（在VS 2005 C++中，编译调试版的程序时会缺省定义宏_DEBUG），见下面的例子程序。

```
/// Program 11.4-1 ///
#include <iostream>
using namespace std;
int main(){
    int i=5;
#ifdef _DEBUG
    cout << i << endl;
#endif
    cout << "Hello,World!" << endl;
}
/// End of Program 11.4-1 ///
```

在调试程序的时候，会执行#ifndef和#endif之间的语句。当调试完成之后，由于调试标记_DEBUG失去定义，从而隐藏调试代码。

（3）使用调试变量

与上面提到的方法类似，可以在运行时设置一个供调试用的bool型变量，它的值决定了特定调试代码的开放和关闭。并且，可以通过程序的命令行参数来控制该变量的开关。程序Program 11.4-1经过这种方式改造之后，可得到如下的程序。

```
/// Program 11.4-2 ///
#include <iostream>
#include <string>
using namespace std;
bool debug;
int main(int argc,char *argv[]){
    int i=5;
    for(int j=0;j<argc;j++){
        if(string(argv[j])=="debug=on")
            debug=true;
    }
    if(debug){
        cout << i << endl;
    }
    cout << "Hello,World!" << endl;
}
/// End of Program 11.4-2 ///
```

在程序运行时，只要在命令行参数中指明debug=on，就可以输出调试信息。否则，只是输出程序“正常”运行的部分。这样就具有较高的灵活性。

（4）几个用于调试的宏

在调试程序的过程中，经常希望知道当前运行的是哪个模块下的哪个函数，在源文件中是第几行等等。如果手工添加这些信息，无疑会给程序员带来很大的负担。因此，C++提供了几个宏，它们分别是`_FILE_`、`_FUNCTION_`和`_LINE_`，可以利用它们“自动”获取有关模块、函数和行的信息。见下面的程序。

```
/// Program 11.4-3 ///
#include <iostream>
using namespace std;
void func1(){
    cout << _FILE_ << endl;
}
void func2(){
    cout << _FUNCTION_ << endl;
}
void func3(){
    cout << _LINE_ << endl;
}
int main(){
    func1();
    func2();
    func3();
}
/// End of Program 11.4-3 ///
```

在作者的机器上，程序的输出是：

```
d:\teaching\c++\study\c++141\c++141\c++141.cpp
func2
13
```

分别将源程序的带完整路径的文件名，函数名和语句的行号显示出来。这些信息在调试程序时是非常有用的。

另外，可以用`assert()`进行条件判断。`assert()`是一个仅在调试版本下起作用的宏，前面已经多次介绍，这里不再赘述。与此类似的还有一个宏`verify`，它可以在程序中对应该满足的条件进行判断。所不同的是，在Release版本中，`assert`不计算输入的表达式的值，而`verify`计算表达式的值。

用户也可以定义自己的宏辅助来完成调试任务。例如下面的宏可以用来显示变量的值，而且变量的名字会一同显示出来。

```
#define PR(x) cout<<"#x"="#"<<x
```

这是利用#对宏的参数进行处理的结果。具体用法可参见1.16节。

(5) 利用工具进行调试

利用集成开发环境进行调试也是一种选择。可以在IDE中设置断点、单步跟踪、查看变量和内存的值、动态修改变量的值以改变程序的执行路径等等。每一种具体的调试工具，其调试命令和方法都有差异，使用时要参阅相应的文档（如MSDN等）。要说明的一点是，使用



工具进行调试与基于打印输出的调试除了在使用的方便程度上有差别之外，在某些特殊的情况下，不能或者很难用工具进行某些程序的调试。如在Windows程序设计中，要调试与窗口重绘的有关代码时，就不适合用IDE进行调试。原因是焦点从IDE窗口转到应用程序窗口时，会引发新的重绘动作，导致程序运行陷入“死循环”。

使用各种调试的手段或工具，其目的是尽可能早、尽可能多地发现已经存在于程序中的错误。与此相关联的问题是，如何较少地引入错误、如何有策略地使用调试手段。给出几条具体的建议：

①采用良好的编程风格。比如，用统一的规范为变量、函数和类型命名，程序的基本单位（如函数）的规模控制在一定范围以内（如不超过100行），锯齿形编码，合理的注释等等。

②进行代码复查。这是Watts S Humphrey领导的研究小组指定的PSP（Personal Software Process，即个体软件过程）规范中提倡的做法。在编译之前就进行代码复查，比直接进行编译更能有效地发现程序缺陷。

③对历史数据进行统计和跟踪。每个程序员的知识背景和工作习惯各不相同，通过统计历史上个人最容易出现哪些类型的编程错误，以便在将来有针对性地进行排查，是一种有效的提高程序质量的做法。

11.5 关于编码规范

应该说，凡符合C/C++语法的代码就是合格的代码，但合格的代码不一定是优秀的代码。编写优秀的代码是每一个有进取心的程序员所追求的目标。编码规范就是用来解决如何编写优秀的代码这个问题的。就好比一个人在社会上生存，只要不触犯法律，就不会受到法律的制裁，但是要成为一个高尚的人，还必须遵循某些道德规范。C++的语法就是C++程序的“法律”，违背了它就无法通过编译。而编码规范则是C++程序开发的“道德”，违背了它就不可能成为一名优秀的程序员。

编码规范本身应该是一些规定，这些规定对程序员个人或在一个团体内部有约束力。制定编码规范的目的主要有两个：其一，有助于程序员个人一致地产生清晰、易维护的程序；其二，有利于程序员之间的交流和团队合作。

软件产业几十年的发展表明，软件设计更多地是一种工程，而不是一种个人艺术。由于大型产品的开发通常由很多人协作完成，如果不统一编码规范，最终由各个不同的部分合到一起的程序，其可读性将不可避免地变得较差。这不仅给代码的理解带来障碍，增加维护阶段的工作量，同时不规范的代码隐含错误的可能性也比较大。

BELL实验室的研究资料表明，软件错误中18%左右产生于概要设计阶段，15%左右产生于详细设计阶段，而编码阶段产生的错误占的比例则接近50%。分析表明，编码阶段产生的错误当中，语法错误大概占20%左右，而由于未严格检查软件逻辑导致的错误、函数（模块）之间接口错误及由于代码可理解度低导致优化维护阶段对代码的错误修改引起的错误则占了一半以上。

可见，制定详细的软件编码规范，并让参与项目开发的每一位程序员共同遵守，就可以大大降低编码阶段的出错率。从大的方面说，代码的可维护性（可读、可理解性、可修改性）、代码逻辑与效率、函数（模块）接口、可测试性是四个最重要的方面，对此应制定行之有效的编码规范。



必须说明的是，并不是所有的编码规范都是唯一确定的，不同的人或团队可以建立不同的规范。因为编码规范本身并没有非此不可的科学依据，它本身是具备一定程度的“柔性”的。例如，下述三种放置括号的方式，在可读性上没有任何差异。

```
void using_k_and_r_style() {
    // K&R 风格
}

void putting_each_brace_on_its_own_line()
{
    // 括号独占一行
}

void or_putting_each_brace_on_its_own_lineIndented()
{
    // 括号独占一行并缩进
}
```

任何专业程序员都可以毫不费力地阅读上面所列的任何一种风格的代码。只不过，程序员应该以团队共同认可的风格书写程序，这样更便于大家的交流与合作。这也说明了，规范的制定要有尽可能充分的依据，特别是应该得到实践的检验。

C++的编码规范涉及到程序设计的方方面面，不是三言两语就可以描述清楚的。下面给出一些具体的编码规范，仅供参考，它们说明了编码规范所可能拥有的形式。

(1) 函数

- <规则1>一定要做到先定义后使用。
- <规则2>函数原型声明放在一个头文件中。
- <规则3>函数无参数一定要用void 标注。
- <规则4>对于内置类型参数应传值(除非函数内部要对其进行修改)。
- <规则5>对于非内置类型参数应传递引用(首选)或指针。
- <规则6>关于何时用指针传递参数。
- <规则7>避免使用参数不确定的函数。
- <规则8>若不得不使用参数不确定的函数，用<stdarg.h> 提供的方法。
- <规则9>避免函数的参数过多。
- <规则10>尽量保持函数只有唯一出口。
- <规则11>显式定义返回类型。
- <规则12>(非void)任何情况都要有返回值。
- <规则13>若函数返回状态，尝试用枚举作类型。
- <规则14>返回指针类型的函数应该用NULL 表示失败。
- <规则15>函数尽量返回引用(而不是值)。
- <规则16>若必须返回值，不要强行返回引用。
- <规则17>当函数返回引用或指针时，用文字描述其有效期。
- <规则18>禁止成员函数返回成员(可读写)的引用或指针。



<规则19>重复使用的代码用函数替代。

(2) 类的设计和声明

<规则1>类应是描述一组对象的集合。

<规则2>类成员应是私有的(private)。

<规则3>保持对象状态信息的持续性。

<规则4>提高类内聚合度。

<规则5>降低类间的耦合度。

<规则6>努力使类的接口少而完备。

<规则7>保持类的不同接口在实现原则上的一致性。

<规则8>保持不同类的接口在实现原则上的一致性。

<规则9>避免为每个类成员提供访问函数。

<规则10>不要在类定义时提供成员函数体。

<规则11>函数声明(而不是实现)时定义参数的缺省值。

<规则12>恰当选择成员函数、全局函数和友元函数。

<规则13>防范、杜绝潜在的二义性。

<规则14>显式禁止编译器自动生成不需要的函数。

<规则15>当遇到错误时对象应该应对有度。

<规则16>用嵌套类的方法减少匿名命名空间类的数量。

(3) 表达式和控制流程

<规则1>让表达式直观。

<规则2>避免在表达式中用赋值语句。

<规则3>不能将枚举类型进行运算后再赋给枚举变量。

<规则4>避免对浮点类型做等于或不等于判断。

<规则5>尝试用范围比较代替精确比较。

<规则6>范围用包含下限不包含上限方式表示。

<规则7>尽量不使用goto语句。

<规则8>在循环过程中不要修改循环计数器。

(4) 编译

<规则1>关注编译时的警告(warning)错误。

<规则2>把问题尽量暴露在编译时而不是运行时。

<规则3>减少文件的依赖程度。

<规则4>减少编译时间。

<规则5>透彻研究编译器。

要想写出优秀的C/C++代码有很多注意点，远远不是上面这些规则所能完全囊括的。并且，这里罗列的编码规范可能和你曾经见到过的一些编码规范有所抵触，这不足为奇。比如，很多编码规范规定了函数体的最大行数。这是因为，在实践中，过多的行数大部分情况下源自功能结构化分不清，不利于阅读，但却不一定如此。所以，如果你确信你的代码是清晰的、内聚性强的，那么不必拘泥于这条特定的编码规范。编码规范的灵魂是隐藏在规范之后所要达到的真正目的，这是理解编码规范的关键。



11.6 正确使用注释

注释是提高程序可读性的一个重要手段。C++的注释有两种形式：一是沿用C语言中的注释方法，将注释内容放在/*和*/之间；还有一种就是单行注释，在一行当中凡是出现在//之后的内容都被当做注释。注释是写给程序员看的，编译器会简单地忽略注释，就好像注释不存在一样。但对于程序员而言，注释却是程序中必须认真对待的一个问题。下面从注释的语法、注释的内容、注释的用途等方面谈谈使用注释要注意的事项。

(1) 多行注释不允许嵌套

使用/*和*/进行注释时，在这段注释内部不允许再出现用/*和*/标明的注释，否则会引发编译错误。如下面的注释：

```
/*
    This is a sample program.
/*
    This is a nested comment.
*/
    This becomes a normal statement.
*/
```

在上面这段注释中，第一个/*会跟第一个*/匹配，导致注释内容“`This becomes a normal statement.`”被编译器当做一条正常的C++语句，从而导致编译错误。

(2) 程序中不能没有注释

除非是编写一些用于学习的、非常初级的小程序，否则一个没有一行注释的程序不会被认为是一个“好”程序。程序设计必然涉及到人类复杂的智力活动，越是大型的程序，其复杂程度就越高。为了便于程序员之间的交流与合作，为了程序的可读性和可维护性，在程序中使用注释是每个程序员必不可少的工作。

(3) 不可过度使用注释

注释并不是越多越好。对理解程序起不到帮助作用的注释，不但不会增强程序的可读性，反而会分散程序员的注意力，甚至妨碍程序员对程序的理解。例如下面的注释：

```
a=b; //assign b to a
```

这条注释纯粹是多余的，因为任何程序员都明白`a=b;`的含义，这样的注释并不能提供任何有价值的信息。

不可过度使用注释的更为重要的原因是：注释同样是需要维护的。而且，从某种意义上说，注释的维护或许比代码的维护更加困难。因为程序员往往忽略了对注释的及时维护，等过了一段时间之后，对注释的维护会变得更加麻烦。例如上面的例子，如果代码发生了变化而没有及时更新注释，变成如下的样子：

```
c=b; //assign b to a
```

那么，这条注释会对程序员造成困扰。程序员读到这条注释的第一反应，不会认为注释是错误的，而是会分析注释的“真正意图”。或许，`c`是`a`的一个引用，于是`c=b;`就完成了`a=b;`的工作？

因此，注释应该能为读懂程序提供有价值的信息，而不是作为一种装点门面的饰品。如

果有可能，应当让程序本身尽量具有可读性，而让注释起到一个“锦上添花”的作用。如果程序本身的可读性较差，指望依靠注释来提高程序的可读性的愿望往往会落空。考察如下的注释：

```
for(int qz=0;qz<8;qz++) //qz stands for counter, 8 is the number of students
```

在这段程序中，程序员使用一个较为费解的变量名qz，然后又使用了一个代表学生数量的字面常量8。为了增强程序的可读性，程序员在后面加了一条注释，对qz和8的含义做了解释。但是，这并不是一个很好的做法。应该让程序本身尽量地“说明问题”，然后再考虑使用注释。将这段代码改成如下形式：

```
int number_of_students = 8;
for(int counter=0;counter<number_of_students;counter++)
```

就比先前的代码要好得多。代码本身的可读性很强，不需要再使用注释，这样就省去了对注释的维护，同时能让程序员“直接”读懂代码，而不是通过后面的注释来“间接”地理解代码。

(4) 注释的用途

一般说来，注释可用来达到如下目的：提供程序本身所无法提供的额外信息，如程序的作者、开发时间、背景资料等；提供程序本身难以提供的帮助信息，如程序的设计思路、应该注意的要点等；对难以理解的算法或语句做出说明；等等。另外，在程序的开发调试阶段，还可以利用注释测试不同版本的运行效果。例如下面的程序。

```
//// Program 11.6-1 /////
#include <iostream>
using namespace std;
void show(){
    cout << "this is first version" << endl;
}
/*
void show(){
    cout << "this is second version" << endl;
}
int main(){
    show();
}
//// End of Program 11.6-1 ////
```

在这个程序中，函数show()有两个版本，一个是被编译的版本，一个是被注释掉的版本。如果要考察被注释掉的版本的运行情况，解除其注释，并将现行版本注释掉。在实际开发的过程中，有时一时难以确定两种方案的优劣，可以暂时使用这种方法使两种解决方案“并存”，等最终比较结果出来之后，再从程序中删除被淘汰的方案的代码。

注意，这种方式只是一个权宜之计，在正式交付的程序中不应该出现一个问题的多个版本的解决方案的代码。

11.7 静态库与动态库

面向对象程序设计的一个重要初衷就是实现代码复用。而程序的具体开发方式则走过了第一条从“无库→静态库→动态库”的发展道路。将已经编写并经过充分测试的代码存放在库中，并在合适的时候加以重用，是提高程序开发效率、提高程序质量的一条重要途径。

(1) 创建和使用静态库

静态库就是Static Library，之所以称之为静态，与库中代码的重用方式有关。当应用程序复用了静态库中的代码（类、函数、变量等），这部分代码会加入最终的目标代码（如可执行exe文件等）中。换句话说，静态库只在程序的链接阶段有用，程序运行时，并不需要静态库的支持，因为程序运行时所需要的那部分代码已经从静态库中提取出来并合并进目标文件中去了。

下面通过一个具体的实例，演示如何创建和使用一个静态链接库，并阐明最终生成的可执行文件与静态库之间的关系。

首先，设计两个可以重用的类，类名分别为ReUse1和ReUse2，它们的实现代码如下。

```
/// Program 11.7-1 ///
/** staticlib.h ***/
class ReUse1{
    int member;
public:
    ReUse1();
    void Effect();
};

class ReUse2{
    char Name[10];
public:
    ReUse2();
    void Output(char*);
};
/** end of staticlib.h **/

/** staticlib.cpp ***/
#include <iostream>
#include "staticlib.h"
ReUse1::ReUse1(){
    member=1;
}
void ReUse1::Effect(){
    std::cout << "class" << member << std::endl;
```

```

}

ReUse2::ReUse2(){
    strcpy(Name,"张三");
}

void ReUse2::Output(char *s){
    std::cout << s << " " << Name << "!" << std::endl;
}

/** end of staticlib.cpp ***/
/// End of Program 11.7-1 ///

```

可重用的类ReUse1和ReUse2是在头文件staticlib.h中说明，在staticlib.cpp中实现的。在分离编译模式下，要使用这两个类，可以在C++源文件中包含头文件staticlib.h，然后在链接阶段，将目标文件staticlib.obj链接进来即可。因此，静态链接库可以理解成多个目标文件（obj文件）的集合。在本例中，在命令行下输入：

cl /c /EHsc staticlib.cpp

就可以生成staticlib.obj，可以利用一个或多个obj文件生成一个静态链接库。在本例中，通过如下命令：

lib staticlib.obj /out:StaticLibrary.lib

生成了一个名为StaticLibrary.lib的静态链接库。lib是静态库文件的后缀名。

有关Visual C++的命令行工具的设置和使用，请参阅MSDN文档。

有了静态库，就可以重用静态库中的代码。下面的程序重用了ReUse2这个类。

```

/// Program 11.7-2 ///
#include "staticlib.h"
int main(){
    ReUse2 obj;
    obj.Output("Hello");
}
/// End of Program 11.7-1 ///

```

程序的输出结果是：

Hello 张三！

要产生这个程序的exe文件（假设源文件是test.cpp），可以有两种办法。第一种是使用如下的命令：

cl test.cpp /EHsc /link StaticLibrary.lib

第二种方法是连续使用如下两条命令：

cl /c /EHsc test.cpp

link test.obj StaticLibrary.lib

通过以上两种方法，都可以产生名为test.exe的可执行文件。

以上是在命令行方式下生成和使用静态库的过程。在Visual Studio IDE中，生成静态库的步骤是这样的：

①选择菜单项“文件→新建→项目”，在弹出的对话框中选择“Visual C++→Win32→Win32控制台应用程序”。在“Win32应用程序向导”中，在“应用程序类型”中选择“静态库”，

在“附加选项”中排除“预编译头”，再点击“完成”按钮，就可以生成一个静态库项目。

②添加相应的头文件，相应的cpp源文件，然后编译，就可以产生以项目名称加上.lib后缀构成的静态库文件。

要使用静态库，关键要解决两个问题：一是必须将静态库对应的头文件放在编译器可以找得到的位置（将头文件所在的目录加入include环境变量，或者将头文件拷贝到cpp源文件所在的目录）；二是要让编译器能够找到所使用的静态库，采用的方法是：

首先，将静态库文件拷贝到当前工程文件目录中来，然后选择菜单项“项目→属性”，在弹出的对话框左边选择“配置属性→链接器→输入”，在右边的“附加依赖项”中输入静态库文件的名字，然后点击“完成”按钮。再编译、链接就可以生成可执行文件。

如果可以找到生成静态链接库的项目（Project）文件，也可以选择菜单项“项目→属性”，在弹出的对话框左边选择“通用属性→引用”，在右边点击“添加新引用(N)...”按钮，将需要使用的静态库项目添加进来即可。

在上面的例子中，静态库StaticLibrary拥有两个类（ReUse1和ReUse2）的实现代码，而程序test.cpp只重用了类ReUse2，所以，编译器在链接时，只将test.cpp需要的有关ReUse2的实现代码加入了最终的exe文件中，而不会理会ReUse1的实现代码。

（2）创建和使用动态库

动态链接（Dynamic Linking）是相对于静态链接（Static Linking）而言的。早期的程序设计只采用静态链接。如果用户程序调用到了静态库文件（在Windows操作系统中是以lib为后缀的文件）中的函数，则在程序编译时，编译器会自动将相关函数的二进制代码从静态库文件中抽取出来，与目标程序一起编译成可执行文件。这样做的确在编码阶段实现了代码的重用，减轻了程序设计者的负担，但并未在执行期实现重用。如一个程序a.exe使用了静态库中的f()函数，那么当a.exe有多个实例运行时，内存中实际上存在了多份f()的拷贝，造成了内存的浪费。

动态链接的处理方式与静态链接很相似，同样是将可重用代码放在一个单独的库文件中（在Windows操作系统中是以dll为后缀的文件，Linux下也有动态链接库，被称为Shared Object的so文件），所不同的是：如果用户程序调用了动态链接库中的函数，编译器并不将库文件中的函数执行体复制到可执行文件中，而是只在可执行文件中保留一个函数调用的标记。当程序运行时，才由操作系统将动态链接库文件一并加载入内存，并映射到程序的地址空间中，这样就保证了程序能够正常调用到库文件中的函数。同时，操作系统保证当程序有多个实例运行时，动态链接库也只有一份拷贝在内存中，也就是说动态链接库是在运行期共享的。

使用动态链接方式带来了几大好处：首先是动态链接库和用户程序可以分开编写，这里的分开既可以指时间和空间的分开，也可以指开发语言的分开，这样就降低了程序的耦合度；其次由于动态链接独特的编译方式和运行方式，使得目标程序本身体积比静态链接时小，同时运行期又是共享动态链库，所以节省了磁盘存储空间和运行内存空间；最后一个增加程序的灵活性，可以实现诸如插件机制等功能。用过winamp的人都知道，它的很多功能都是以插件的形式提供的，这些插件就是一些动态链接库，主程序事先规定好了调用接口，只要是按照规定的调用接口写的插件，都能被winamp调用。

程序Program 11.7-1也可以用动态链接库的形式实现。具体步骤如下。

①选择菜单项“文件→新建→项目”，在弹出的对话框中选择“Visual C++→Win32→Win32控制台应用程序”。在“Win32应用程序向导”中，在“应用程序类型”中选择“DLL(D)”，

再点击“完成”按钮，就可以生成一个动态链接库项目。

②为了与Program 11.7-1生成的库在结构上保持一致，选择菜单项“项目→属性”，在弹出的对话框左边选择“配置属性→C/C++→预编译头”，在右边将“创建/使用预编译头”设置为“不使用预编译头”。并在项目中移除stdafx.h和stdafx.cpp这两个文件。

③在项目中添加头文件dynamiclib.h和源文件dynamiclib.cpp，代码如下所示。

//// Program 11.7-3 ////

/** dynamiclib.h **/

class ReUse1 {

 int member;

public:

 __declspec(dllexport) ReUse1();

 __declspec(dllexport) void Effect();

};

class ReUse2 {

 char Name[10];

public:

 __declspec(dllexport) ReUse2();

 __declspec(dllexport) void Output(char*);

};

/** end of dynamiclib.h **/

/** dynamiclib.cpp **/

#include <iostream>

#include "dynamiclib.h"

ReUse1::ReUse1()

 member=1;

}

void ReUse1::Effect()

 std::cout << "class" << member << std::endl;

}

ReUse2::ReUse2()

 strcpy(Name,"张三");

}

void ReUse2::Output(char *s)

 std::cout << s << " " << Name << "!" << std::endl;

}

/** end of dynamiclib.cpp **/

//// End of Program 11.7-3 ////

④编译该程序，就可以在debug目录下看到产生了两个文件：dynamiclib.dll和dynamiclib.lib。之所以会产生两个文件，是因为如果想分离出一部分代码到一个dll文件中，仍然需要一



个lib文件。这个lib文件将被连接到程序告诉操作系统在运行的时候你想用到什么dll文件，一般情况下，lib文件里有相应的dll文件的名字和一个指明dll输出函数入口的顺序表。如果不想用lib文件或者是没有lib文件，可以用WIN32 API函数LoadLibrary、GetProcAddress。事实上，我们可以在Visual C++ IDE中以二进制形式打开lib文件，大多数情况下会看到ASCII码格式的C++函数或一些重载操作的函数名字。

在头文件dynamiclib.h中，在类ReUse1和类ReUse2的某些成员函数的声明前，加上了描述语句__declspec(dllexport)，这个语句的作用就是向编译器指明需要在生成的动态链接库中导出的函数，没有导出的函数是不能被其他程序调用的。实际上，可以通过Visual Studio提供的实用工具dumpbin.exe来查看动态链接库文件中有哪些导出函数。执行下述命令：

```
dumpbin /exports dynamiclib.dll
```

可以得到如下输出信息：

```
Microsoft (R) COFF/PE Dumper Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.
```

Dump of file dynamiclib.dll

File Type: DLL

Section contains the following exports for dynamiclib.dll

00000000 characteristics

48675A02 time date stamp Sun Jun 29 17:46:42 2008

0.00 version

1 ordinal base

4 number of functions

4 number of names

ordinal	hint	RVA	name
1	0	0000111D6 ??0ReUse1@@QAE@XZ	
2	1	0000111D ??0ReUse2@@QAE@XZ	
3	2	0000111F4 ?Effect@ReUse1@@QAEXXZ	
4	3	000011041 ?Output@ReUse2@@QAEXPAD@Z	

Summary

1000 .data

1000 .idata

2000 .rdata

1000 .reloc

```
1000 .rsrc
```

```
6000 .text
```

```
10000 .textbss
```

可以看到在此动态链接库中有4个导出函数：??0ReUse1@@QAE@XZ、??0ReUse2@@QAE@XZ、?Effect@ReUse1@@QAEXXXZ和?Output@ReUse2@@QAEXPAD@Z。这些函数的名字与源文件dynamiclib.cpp中定义函数的名字并不一样，这是C++编译器实施了name mangling的结果（参阅3.13节）。

有了动态链接库，接下来可以写一个测试程序重用动态链接库中的代码。步骤如下。

①选择菜单项“文件→新建→项目”，在弹出的对话框中选择“Visual C++→Win32→Win32控制台应用程序”。在“Win32应用程序向导”中，在“附加选项”中排除“预编译头”，再点击“完成”按钮，就可以生成一个测试程序项目。

②在项目中移除stdafx.h和stdafx.cpp这两个文件。然后添加test.cpp源文件，代码如下。

```
//// Program 11.7-4 /////
#include "dynamiclib.h"
int main(){
    ReUse2 obj;
    obj.Output("Hello");
}
//// End of Program 11.7-4 ////
```

③将动态库文件dynamiclib.dll和相应的dynamiclib.lib拷贝到当前工程文件目录中来，然后选择菜单项“项目→属性”，在弹出的对话框左边选择“配置属性→链接器→输入”，在右边的“附加依赖项”中输入文件名dynamiclib.lib，然后点击“完成”按钮。再编译、链接就可以生成可执行文件。这里的dynamiclib.lib并不是静态库文件，而是dynamiclib.dll的导入库文件，它包含了dynamiclib.dll动态链接库导出的函数信息，只有在工程链接设置里添加了该文件，才能够正确调用动态链接库中的函数。

如果可以找到产生动态链接库的项目，也可以选择菜单项“项目→属性”，在弹出的对话框左边选择“通用属性→引用”，在右边点击“添加新引用(N)...”按钮，将需要使用的动态链接库项目添加进来。

④生成可执行文件。运行之，输出：Hello 张三！。测试成功。

有了动态链接库.dll文件、对应的.lib文件和有关库内容的头文件(.h文件)，就可以按照上述方法使用动态链接库中的函数。这种方式叫做动态链接库的静态加载。如果只有.dll文件，而没有对应的.lib文件，或者想在程序运行的过程当中自主选择调用哪个动态链接库中的函数，就要用到动态链接库的动态加载。

比起静态加载，动态加载更加灵活，而且在编译链接时不需要.lib导入库文件，也不需要提前声明函数。通过windows提供的API函数来动态加载动态链接库并调用其中的函数，用完后可以马上释放内存中的动态链接库，十分方便。下面是一个动态加载动态链接库的例子。

生成动态链接库的源文件如下所示。

```
//// Program 11.7-5 /////
_declspec(dllexport) int myAdd(int x, int y);
int myAdd(int x, int y){
```

```

    return x+y;
}

/// End of Program 11.7-5 ///

```

编译之后生成动态链接库mylib.dll。执行命令：

dumpbin /exports mylib.dll

可以得到如下信息：

```

Microsoft (R) COFF/PE Dumper Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.

```

Dump of file mylib.dll

File Type: DLL

Section contains the following exports for c++132.dll

00000000 characteristics

48687127 time date stamp Mon Jun 30 13:37:43 2008

0.00 version

1 ordinal base

1 number of functions

1 number of names

ordinal hint RVA	name
1 0 00011005	?myAdd@@YAHHH@Z = @ILT+0(?myAdd@@YAHHH@Z)

Summary

- 1000 .data
- 1000 .idata
- 2000 .rdata
- 1000 .reloc
- 1000 .rsrc
- 4000 .text
- 10000 .textbss

可知函数myAdd经过名字改编（Name Mangling）之后，在动态链接库mylib.dll中的名字是?myAdd@@YAHHH@Z。动态加载该函数的源程序例子如下。

/// Program 11.7-6 ///

```

#include <iostream>
#include <windows.h>

```



```
using namespace std;
int main(){
    HINSTANCE hInstance=LoadLibrary("mylib.dll");
    if(!hInstance){
        cout << "找不到库" << endl;
        return 1;
    }
    typedef int (*AddProc)(int,int);
    AddProc Add=(AddProc)GetProcAddress(hInstance,"?myAdd@@YAHHH@Z");
    if(!Add){
        cout<<"动态链接库函数未找到"<<endl;
        return 1;
    }
    cout<<"3+5="<<Add(3,5)<<endl;
    FreeLibrary(hInstance);
    return 0;
}
/// End of Program 11.7-6 ///
```

因为加载动态链接库的过程是与操作系统相关的，所以这里包含了头文件windows.h，并且使用了Windows API函数。动态调用动态链接库中的函数的过程分三个步骤进行：首先是通过LoadLibrary()函数加载动态链接库。LoadLibrary函数把指定的可执行模块映射进调用进程的地址空间并返回一个可以被GetProcAddress函数使用的句柄，以得到DLL函数的地址；第二步是通过GetProcAddress()函数获取动态链接库中被调用函数的入口地址，将它传递给一个函数指针，通过执行函数指针完成函数调用；最后，通过FreeLibrary()函数释放内存中的动态链接库。

此程序的执行结果是：

3+5=8

有一个细节要注意：VS 2005中开发程序时，默认的字符集是Unicode。在这种情况下，很多函数只接收宽字符串作为它的参数，只有将字符串常量显式转换为Unicode型才能正确调用这些函数，否则编译器报错。解决这个问题有两种办法：

①将项目的默认字符集改变成“使用多字节字符集”。具体操作步骤是：选择菜单项“项目→属性”，在弹出的对话框左边选择“配置属性→常规”，在右边将“字符集”设置成“使用多字节字符集”。Program 11.7-6就进行了这样的改造，否则编译时出错。

②保持项目的默认字符集不变，将字符串常量显式转换为Unicode型字符串。在Program 11.7-6中，使用如下的语句：

```
HINSTANCE hInstance=LoadLibrary(TEXT("mylib.dll"));
```

这样，也可以动态加载动态链接库。



第12章 编程思想与方法

12.1 C与C++最大的区别

将C语言与C++语言进行比较，是一个非常大的话题，不是三言两语能够说得清的。同时，没有非常深厚的程序设计功底和丰富的经验，也无法进行全面的比较。C++虽然是在C语言的基础之上发展起来的，但它们毕竟是两种不同的语言。C和C++各自的标准委员会是独立的，最新的C++标准是C++98，最新的C标准是C99。在这里，想从源头探讨一下C语言与C++语言本质的区别，这对于我们学习C++语言本身有帮助。

有很多初学C++的程序员知道C++是从C发展而来的，自然就面临一个问题：是先学会C再学C++，还是直接学习C++呢？还有一些已经学过C语言的程序员，认为C++就是C语言的升级版，那么只要按C语言的编程习惯使用C++语言就可以了。之所以会出现这些问题，就是因为对C语言与C++语言的主要差别认识不清。我们先罗列一些C语言与C++语言的区别：

①C语言是面向过程的，而C++是面向对象的。

②C语言的标准函数库是松散的，只是把功能相关联的函数组织在一起；而C++的标准库以类库（class library）的形式出现，大多数的函数都通过类紧密地结合在一起。例如，虽然可以单独调用Windows API函数，但是C++中的API是对Windows系统的大多数API的有机组合，是一个整体，使用起来会更加方便、安全。

③C和C++中都有结构（struct）的概念，但C语言的结构只有成员变量，没成员函数，而C++的结构可以同时拥有成员变量和成员函数。C语言结构的成员是公有的（public），任何时候都可以访问；而C++结构的成员可以有公有的（public）、私有的（private）和受保护的（protected）几种选择。

④C++语言的函数允许重载，而C语言没有函数重载的概念。

⑤C语言中调用函数时，函数参数的传递方式只有传值和传地址（指针）两种，而C++语言中还可以传引用。

⑥C++中new和delete是对内存分配的运算符，取代了C中的malloc和free。

⑦C++中用iostream类库替代了标准C中的stdio函数库，来进行输入和输出的操作。

⑧C++允许定义函数参数的默认值，而C语言不允许。

⑨C++实现了异常处理机制，而C语言的出错处理代码（如setjmp）是和业务逻辑代码混合在一起的。

.....

仔细比较，可以列出很多条C语言与C++语言的差别。在这众多的差别中，有一条是最根本的：C语言是结构化的面向过程的语言，C++是面向对象的程序设计语言。这体现了编程思想的本质不同。C++语言存在的根本原因，或者说一个程序员选择用C++语言而不是C语言进

行开发的根本原因，是用C++语言能够更快、更好地开发大型系统。

C语言是基于过程的，强调的是程序的功能，以函数（功能）为中心。用C语言进行程序开发，就是编写许多实现具体功能的函数，这些函数之间相互调用完成业务逻辑。这对于小型的、设计稳定的系统来说是可行的。但是，如果系统设计要发生变化，要将原有的C语言程序改造成符合新的系统设计要求的程序，却不是一件轻松的事情。

新的系统设计必然涉及到函数定义的修改，也就是函数的实现会发生一定程度的变化。由于C语言的函数无法自身维护一组状态，所以当函数要处理新的状态时，要么需要增加新的参数，要么需要增加全局变量。但这都会带来问题：增加新的函数参数意味着函数调用接口的变化，程序中所有的函数调用语句都要修改，这对于一个大型程序来说是一个不小的工作量，况且有谁能保证不进行第二次这样的修改呢？如果增加全局变量，则会增加模块之间的耦合度，增加程序的复杂度，不利于程序的维护并增加出错的可能性。

C++语言强调的是用面向对象的方式思考和编程。程序是以类的定义为基本单位进行的，程序的运行则是一组对象之间的交互作用。类的成员函数要处理的状态存放在一个个具体的对象中，当函数要处理新的状态的时候，不必修改函数的接口，也不必定义新的全局变量，只需在类中加入新的状态信息即可，因为对象是“天然的”存放状态信息的场所。

可以看出，C语言在开发大型程序时，面临一个现实的问题：当系统设计发生变化时，很难找到一个合适的地方存放新的状态信息。这给程序的修改和维护都带来了很大麻烦。而C++语言的面向对象的程序设计思想，却可以很自然地通过修改类的定义来解决这个问题。C++语言中的类将数据和对数据的操作显式地“捆绑”在一起（在C语言中它们是松散地结合的），这些对于明确地表达程序员的意图、进行程序的修改和维护、提供代码复用都是有帮助的。这种思想还派生出其他很多具体的语言机制。例如，如果想将函数作为参数传递，除了可以使用C语言中已存在的函数指针之外，还可以使用函数对象（仿函数，参见6.10节）。但使用仿函数更具有优越性，因为仿函数是一个对象，而对象可以维护大量的状态，使用一般的函数指针却很难实现这一点。

所以，虽然C++语言是C语言的超集，但它们的编程思想却是完全不同的。这也是它们之间最本质的区别。在使用C++进行程序设计，特别是在开发大型项目的时候，都应用面向对象的分析和设计技术。

12.2 一个代码重构的例子

代码重构（Refactoring）就是在不改变软件现有功能的基础上，通过调整程序代码改善软件的质量、性能，使其程序的设计和架构更趋合理，提高软件的可扩展性和可维护性。

那么，为什么不在项目开始时多花些时间把设计做好，而要以后花时间来重构呢？要知道一个完美得可以预见未来任何变化的设计，或一个灵活得可以容纳任何扩展的设计是不存在的。系统设计人员对即将着手的项目往往只能从大方向予以把控，而无法知道每个细枝末节，其中永远不变的就是变化，提出需求的用户往往要在软件成型后，才开始“品头论足”，系统设计人员毕竟不是先知先觉的神仙，功能的变化导致设计的调整在所难免。所以“测试为先，持续重构”作为良好开发习惯被越来越多的人所采纳。在实践中，测试和重构成为维持软件生存期和持续提高软件质量的法宝。

实施代码重构有一些先决条件、常用方法和重要准则。例如：一个设计得非常糟糕的项



目不值得做代码重构，不如重新设计、重新开发，因为毕竟重构也是存在一定工作量的；先做代码复审（Code Review），发现程序设计中存在的问题，然后做代码重构（Refactoring）；可以为不够明确或不够合理的标识符重新命名、可以将一个过长（内聚性差）的函数拆成两个函数实现、可以把经常重复的代码抽取出来形成一个使用频率很高的函数（提供service），这些都是代码重构时常用的手段；代码重构，特别是手工进行的代码重构，一定要配合以单元测试，才能保证不至于引入新的错误；等等。

代码重构的话题很多，需要系统研究的人可参阅 Martin Fowler 所著的《重构—改善既有代码的设计》一书。在这里，试图通过一个例子展示一个具体的手工进行代码重构的过程。

设计一个日期类，要求在创建该类对象的时候，如果不提供任何参数，就将日期的默认值设置为“1/1/2000”。另外，还提供参数个数分别为1、2、3的构造函数，可以单独设置月份、月份和日子和同时设置年月日。该类还要提供一个显示日期的成员函数，它能够以“月/日/年”的格式显示类对象代表的日期。该类的初始设计和测试如下。

```
//// Program 12.2-1 /////
#include <iostream>
using namespace std;
class TDate{
    int month;
    int day;
    int year;
public:
    bool isleapyear(){
        return (year%4==0 && year%100!=0 )||(year%400==0 );
    }
    TDate(){
        month=1;
        day=1;
        year=2000;
    }
    TDate(int m){
        if(m>0 && m<13)
            month=m;
        else
            month=1;
        year=2000;
        day=1;
    }
    TDate(int m,int d){
        int limit[12]={31,29,31,30,31,30,31,31,30,31,30,31};
        if(m>0 && m<13 && d>0 && d<32){
            month=m;
            day=d;
        }
    }
}
```

```

    day=d;
}
else{
    month=1;
    day=1;
}
if(day>limit[month-1]){
    month=1;
    day=1;
}
year=2000;
}

TDate(int m,int d,int y){
int limit[12]={31,29,31,30,31,30,31,31,30,31,30,31};
if(m>0 && m<13 && d>0 && d<32 && y>0){
    month=m;
    day=d;
    year=y;
}
else{
    month=1;
    day=1;
    year=2000;
}
if(!isleapyear()&&month==2&&day==29){
    month=1;
    day=1;
    year=2000;
    return;
}
if(day>limit[month-1]){
    month=1;
    day=1;
    year=2000;
}
}

void show(){
    cout << month << "/" << day << "/" << year << endl;
}
};


```



```

int main(){
    TDate t1(5);
    t1.show();
    TDate t2(5,32);
    t2.show();
    TDate t3(8,38,2008);
    t3.show();
    TDate t4(8,8,2008);
    t4.show();
}

```

//// End of Program 12.2-1 ////

此程序设计了一个日期类TDate。它有4个重载的构造函数，参数个数分别为0、1、2、3。参数个数虽然不同，但每一个函数的参数集合都是最大可能参数集合的子集。也就是说，最多只需要使用3个参数就可以完成日期类对象的初始化工作。当用错误的参数初始化TDate类对象时，TDate的值自动变为“1/1/2000”。程序的输出结果是：

```

5/1/2000
1/1/2000
1/1/2000
8/8/2008

```

我们对这个设计进行代码重审（Code Review）时，可以发现，这4个构造函数要表达的语义是相同的：如果提供了合适的初值，就用这些初值来初始化month、day和year这三个成员变量，如果参数非法，就将日期设置为“1/1/2000”。因此，可以将这4个构造函数的工作交给一个“服务”函数来完成，该函数的功能就是判断初值的合法性，并完成对象数据成员的修改工作。这个函数的实现代码可以从有3个参数的构造函数代码中抽取，并且它的访问权限可以设置成public，从而使该类具有动态安全修改类对象内容的功能。改造后的程序如下。

```

//// Program 12.2-2 ////
#include <iostream>
using namespace std;
class TDate{
    int month;
    int day;
    int year;
    bool valid(int m,int d,int y) const{
        if(m<1 || m>12 || d<1 || d>31 || y<1)
            return false;
        if(!isleapyear(y)&&m==2&&d==29)
            return false;
        int limit[12]={31,29,31,30,31,30,31,31,30,31,30,31};
        if(d>limit[m-1])
            return false;
    }
}
```

```

    return true;
}

public:
    bool isleapyear(int y=0) const{
        int t=(y==0)?year:y;
        return (t%4==0 && t%100!=0 )||( t%400==0 );
    }
    void setdate(int m,int d,int y){
        if(valid(m,d,y)){
            month=m;
            day=d;
            year=y;
        }
        else{
            month=1;
            day=1;
            year=2000;
        }
    }
    TDate(){
        setdate(1,1,2000);
    }
    TDate(int m){
        setdate(m,1,2000);
    }
    TDate(int m,int d){
        setdate(m,d,2000);
    }
    TDate(int m,int d,int y){
        setdate(m,d,y);
    }
    void show(){
        cout << month << "/" << day << "/" << year << endl;
    }
};

int main(){
    TDate t1(5);
    t1.show();
    TDate t2(5,32);
    t2.show();
}

```

```

TDate t3(8,38,2008); // 类的构造函数重载，参数为单月开始同公牛尊的盛大一个千枚。但是
t3.show(); // 又是类的成员函数，输出类的成员变量来表示日期，输出结果非常非
TDate t4(8,8,2008); // 相同。向式将该类的成员函数的参数设置为“年-月-日”，则输出结果将
t4.show(); // 一样。
}
/// End of Program 12.2-2 ///

```

这个程序的功能与上一个完全相同，输出结果是一样的，但程序结构已经发生了很大的变化。每个构造函数的函数体都只有一条语句，它们共同调用一个设置日期的函数setdate()。而setdate()又将最为复杂的初值正确性判断的任务交给了私有函数valid()。这样就完成了一次代码重构。程序的功能没有变化，类所能提供的服务没有减少（甚至还增加了新的服务），类的成员函数的调用方式没有发生变化，甚至利用函数参数的默认值和const约束增强了成员函数isleapyear()的功能。

代码还可以再次重构，这种“持续重构”的做法也是软件维护的一项重要内容。继续对Program 12.2-2进行代码重审（Code Review），发现重构的4个构造函数除了参数的默认值不同外，其他方面完全相同。于是，利用C++的函数参数默认值的机制，可以将这4个重载的构造函数统一成一个。也就是定义这样一个构造函数：

```

TDate(int m=1,int d=1,int y=2000){ // 一旦需要重写，代码就不容易被修改，所以整个重构
    setdate(m,d,y); // 也不太容易被修改，更不容易被修改，所以整个重构
}

```

这样，就不用定义4个构造函数，只写一个就够了。程序的结构变得更为清晰。

考察以上代码重构的过程，我们要注意这样几个细节：

①代码重构是在不改变程序功能的前提下，调整程序的结构。所以，对于类的设计来说，最为安全的重构，是改变类的成员函数的实现方式，而保持调用接口不变。

②代码重构有时需要用到一些具体的技术或技巧。在本例中，就是利用了函数参数的默认值来取消函数的重载。这种技巧的使用是有条件的，那就是重载的几个函数的参数列表必须位于最大可能参数序列的最左端。在TDate类中，如果单参数的构造函数是为成员变量day（或year）赋初值的，就无法统一这4个构造函数了。另外，如果存在拷贝构造函数，也只能单独定义。可见，代码重构的持续进行必须依靠各种可能的条件，并没有完全固定的模式。

③代码重构与软件设计是软件开发的两个重要环节。有了代码重构的经验，对于设计出高可扩展性的程序绝对是有帮助的，并且可以防止“过度设计”。而良好的设计则是代码重构的前提，代码重构只能改善设计，而不能代替设计。

12.3 实现代码重用需要考虑的问题

程序设计语言和方法的发展，一个重要的动因就是代码的重用。今天的软件项目规模越来越大，如果一切代码都要从头编写，那么软件开发的效率必然是低下的。其实，我们可能在学习程序开发的第一天起，就已经在做着“代码重用”的事情了，只是自己没有明确地意识到而已。C语言当中的输入输出函数（scanf()、printf()等），C++中的标准输入输出对象（cin、cout等），字符串类string等，我们在利用它们实现自己程序的功能时，已经在利用系统给我们提供的可重用代码了，它们是以标准库的方式提供给我们的。不过，这只是最低程度的代码

重用。对于一个大型的软件公司或开发单位而言，针对特定的业务领域建立起一套完善的可重用代码库，以提高将来开发类似系统的效率，提高市场竞争力，是一件极有意义同时又是一件非常困难的事情。

代码重用首先是一种思想，它关乎我们进行软件开发的目标和方向。同时，它又是一种技术（或者说，是多种技术的结合体），它是为了实现我们的目标而采取的手段。

C++是一种面向对象的程序设计语言，而面向对象的思想，其重要的目标之一就是实现代码重用。但是，并不是说只要使用了C++这种面向对象的编程语言，就一定能够编写出可重用的代码，这里面涉及到很多问题。最根本的原则只有一个：保持类或模块的简单和纯粹。一个类或模块越是简单、越是较少地依赖于其他的类或模块，它的可重用性就越高。对于程序员而言，编写出可重用性高的代码，需要对问题进行细致的分析、对代码进行合理的分解和组合。下面，结合具体的例子讨论几个提高代码可重用性的具体原则或手段。

(1) 模块功能单一化

模块的功能要单一，这不仅是代码重用的原则，同时也是提高代码的可理解性、可维护性的重要原则。虽然大家都认同这一点，但是在设计编码过程中，并不是谁都能小心地处理这个问题。

举一个具体的例子。假设我们有一些文本文件，文件中每一行都存储着固定数量的整数，但是每个整数所占据的宽度是各不相同的。现在要编写一个小应用程序，要求将文本文件中的每一个整数调整为固定宽度，同时保持文本文件的结构不变。例如，将包含如下数据的文本文件：

```
11 22 3 414 5
6 70 18 29 55
```

转换成如下格式文件：

```
11      22      3      414      5
       6      70      18      29      55
```

也就是说，每个整数占据8个字符的宽度。可以编写如下程序解决这个问题。

```
/// Program 12.3-1 ///
/** filename.h ***/
#include <string>
void GetFileName(std::string &, std::string &);

/** End of filename.h **/


/** filename.cpp ***/
#include <iostream>
#include <string>
using std::string;
using std::cout;
using std::cin;
void GetFileName(string &infilestr, string &outfilestr){
    cout << "Please input source file: ";
    cin >> infilestr;
```

```

cout << "Please input target file: ";
cin >> outfilestr;
}

/** End of getfilename.cpp **/

/** aligncopy.h **/
void AlignCopy(int);
/** End of aligncopy.h **/

/** aligncopy.cpp **/
#include "getfilename.h"
#include <iostream>
#include <string>
#include <fstream>
#include <strstream>
#include <iomanip>
using namespace std;
void AlignCopy(int n){
    char line[256];
    int num;
    string infilestr,outfilestr;
    GetFileName(infilestr,outfilestr);
    ifstream infile(infilestr.c_str());
    if(!infile){
        cout << "Can not open source file" << endl;
        return;
    }
    ofstream outfile(outfilestr.c_str());
    if(!outfile){
        cout << "Can not open target file" << endl;
        return;
    }
    while(!infile.eof()){
        infile.getline(line,255);
        if(!line[0]&&infile.eof()) break;
        line[strlen(line)+1]='\0';
        line[strlen(line)]=';
        istrstream input(line);
        input >> num;
        while(!input.eof()){

```

```

        outfile << setw(n) << num;
        input >> num;
    }
    outfile << endl;
}
infile.close();
outfile.close();
}
/** End of aligncopy.cpp **/

/** main.cpp **/
#include "aligncopy.h"
int main(){
    AlignCopy(8);
}
/** End of main.cpp */
/// End of Program 12.3-1 ///

```

这个程序可以正常运行，只要输入的文件中存放的全部都是整数，就可以得到全部整数都占8个字符的输出文件。

这个程序的最大问题在于，如果我们想重用其中的代码，比如，在另一个项目中重用函数AlignCopy()来对某个文件进行重新排版并输出，就会发现很大的问题。首先，要重用AlignCopy()，就要包含头文件aligncopy.h，同时将aligncopy.cpp拷过来。但是，在aligncopy.cpp中又用到了函数GetFileName()，于是，又要包含头文件getfilename.h，同时将getfilename.cpp拷贝过来。若是该项目很大，getfilename.cpp还要依赖其他的源文件，这种链式依赖关系将使重用者吃尽苦头，几乎将整个项目合并进来，并且AlignCopy()还是没法使用，因为在新的项目中，要转换的输入文件和输出文件的名称并不是通过键盘输入的！

问题的根源在于，aligncopy.cpp这个模块的功能不够单一。作为一个文件处理模块，打开文件并重新输出是其功能目标，而获取输入输出文件的名称，看似和AlignCopy()相关，但实质上并不是它的目标功能。AlignCopy应该可以处理任何全部由整数组成的文本文件。所以我们修改aligncopy.cpp中的AlignCopy()这个函数，给它传入输入输出文件的名称作为参数，而不是调用GetFileName()去获取名称。

```

void AlignCopy(char* infilestr, char* outfilestr, int n){
    char line[256];
    int num;

    ifstream infile(infilestr);
    ...
    ofstream outfile(outfilestr);
    ...
}

```



这个修改看似简单，但是实际上，却使aligncopy.cpp解除了对getfilename.cpp的依赖。也就是说，aligncopy.cpp模块已经相对独立，可以抽取出来加以重用了。甚至，我们可以对AlignCopy()再进行改造，让它带一个istream&和一个ostream&类型的参数，从而实现更大范围的复用（不仅仅针对文件）。

所以，在设计编码阶段的任何一点疏忽，都会给后继开发带来巨大的麻烦。切记：模块的功能要单一。在模块中要使用某些由别的模块的代码产生的数据时，不要直接调用其他模块中定义的函数，而是利用参数的形式将数据从外部传递进来，从而使模块尽量减少对其他模块的依赖，保持功能单一。

（2）头文件尽量减少包含其他头文件

与模块功能单一化相关的一个问题是，一个头文件最好尽量少地包含其他的头文件。在开发阶段，为了方便起见，程序员往往把所有的头文件、宏定义、或者函数声明都包含在一个头文件中。那么，在编写.cpp文件的时候会非常方便，一般只要包含这个“总”的头文件就可以了，不用担心任何缺少头文件之类的问题。但是，在代码重用的时候，这样的头文件却会给程序员带来麻烦。因为，打开这样的头文件之后，常常看到如下的情况：

```
#include "a.h"
#include "b.h"
#include "c.h"

...
```

换句话说，要在工程中使用这个头文件，就必须拷贝“a.h”，“b.h”，“c.h”这三个文件，而且在这几个文件中，必须没有再度包含别的头文件。目前的一般情况是，很容易在头文件中包含其他的头文件。最终的结果，发现包含某个头文件是不可能的，因为需要拷贝的文件太多了。

所以，在头文件中包含其他的头文件往往是不必要的，是应该禁止的。只有万不得已的情况，才能这样做。

这条原则有些违反我们的直觉。在编程的实践中，一个模块用到其他模块中定义的东西是天经地义的事情，因此，一个头文件也应该包含其他的头文件。但是，在很多情况下，我们都可以利用特定的技巧或方法，避免在头文件中包含其他的头文件。考察下面的程序。

```
//// Program 12.3-2 /////
/** classb.h ***/
class MyClassB{
public:
    void Show();
};

/** End of classb.h **/


/** classb.cpp ***/
#include <iostream>
#include "classb.h"

void MyClassB::Show(){
```

```
    std::cout << "Hello" << std::endl;
}
/** End of classb.cpp **/
```

```
/** classa.h ***/
#include "classb.h"
class MyClassA{
private:
    MyClassB mobj;
public:
    void ShowObject();
};
/** End of classa.h **/
```

```
/** classa.cpp ***/
#include "classa.h"
void MyClassA::ShowObject(){
    mobj.Show();
}
/** End of classa.cpp **/
```

```
/** main.cpp ***/
#include "classa.h"
int main(){
    MyClassA a;
    a.ShowObject();
}
/** End of main.cpp **/
```

/// End of Program 12.3-2 ///

在这个例子中，由于MyClassB的对象作为MyClassA的一个成员而存在，所以，似乎#include "classb.h"是惟一的选择，否则MyClassB mobj这条语句无法通过编译。但是实际上，在这里定义MyClassB的对象作为MyClassA的一个成员不是一个好的设计，用对象的指针会更好一些。上面的程序改造之后如下。

```
/// Program 12.3-3 ///
/** classb.h ***/
class MyClassB{
public:
    void Show();
};
/** End of classb.h **/
```



```

/** classb.cpp ***/
#include <iostream>
#include "classb.h"
void MyClassB::Show(){
    std::cout << "Hello" << std::endl;
}
/** End of classb.cpp **/


/** classa.h ***/
class MyClassB;
class MyClassA{
private:
    MyClassB *mobj;
public:
    void ShowObject();
    MyClassA();
    ~MyClassA();
};

/** End of classa.h **/


/** classa.cpp ***/
#include "classa.h"
#include "classb.h"
MyClassA::MyClassA(){
    mobj = new MyClassB();
}
MyClassA::~MyClassA(){
    delete mobj;
}
void MyClassA::ShowObject(){
    mobj->Show();
}
/** End of classa.cpp **/


/** main.cpp ***/
#include "classa.h"
int main(){
    MyClassA a;
    a.ShowObject();
}

```

```

}
/** End of main.cpp ***/
/// End of Program 12.3-3 ///

```

改造后的程序与原来的程序功能完全一样，甚至有可能更节约内存（可以在MyClassA的构造函数中将mobj置为NULL，等需要用到的时候再分配空间）。关键是，此时classa.h中不需要包含classb.h，从而摆脱了对classb.h的依赖，使得代码重用变得简单。所以，将有可能包含的头文件从.h文件中转移到.cpp文件中去（例如本例中，将#include "classb.h"从classa.h中移入了classa.cpp中），是在头文件中减少对其他头文件的包含的一条有效思路。另外，在类的数据成员中使用其他的类，或者函数的参数是用户自定义类类型，要注意这样一条原则：尽量使用指向对象的指针或引用，而不要直接使用对象。这样做有两个好处：一是提高了程序的执行效率（比如函数调用时避免了对实参对象的拷贝），二是有利于代码的复用（解除头文件之间的依赖关系）。

在继承机制下，包含定义基类的头文件是不可避免的，比如：

```

class MyClassA : public MyClassB{
};

```

此时，必须使用#include "classb.h"。不过，软件工程的实践表明，过多地使用继承，特别是使用多层次的复杂继承，不是一个很好的主意。它会给软件的扩展和维护带来一系列的麻烦。专家主张：优先使用组合（Composition）而不是继承。继承仅仅用在基类是一个抽象类的情况下。

那么，是不是在任何情况下，头文件都不能包含其他的头文件呢？其实，这不仅不合理，而且也做不到。一个普遍为大家采用而且是安全的做法是：包含系统给我们提供的头文件。例如#include <iostream>以使用cin和cout，#include <string>以使用类string等。这些头文件是由开发环境提供给程序员的，没有人找不到它们，而且由系统的标准库加以支持，高效而安全。在头文件中包含用户自己编写的头文件时，则必须小心谨慎地加以避免！

（3）隐藏实现细节

在头文件中包含别的头文件，一个重要的原因是用到了别的类或模块中提供的服务。直觉上，必须包含提供服务的头文件，才能使用该项服务。这种认识并没有错，问题的关键是：包含头文件的操作一定要在头文件中进行吗？

假设模块service.cpp的作用是对外提供某些服务，重用代码时只要包含service.h，然后利用类Service中提供的服务DoSomething()就可以了。如果服务DoSomething()并不是由service.cpp独立完成的，它借助了在helper.cpp中实现的类Helper。一般的做法是：在service.h中包含头文件helper.h，然后利用类Helper的成员函数帮助完成服务DoSomething()。如果是这样设计，就会给代码重用带来麻烦。因为，service.h中包含了头文件helper.h，而头文件helper.h又暴露了类Helper的全部细节。我们要使用服务DoSomething，不仅要拷贝头文件service.h，还要拷贝头文件helper.h。假设helper.h还包含其他头文件，问题就变得更加复杂。

因此，正确的做法是：虽然service.cpp要用到类Helper，但并不在service.h中包含头文件helper.h，而是在service.cpp中包含它。service.h只需要定义提供给用户的接口就可以了，就像Program 12.3-3中那样。这个例子引出了设计可重用代码的一条重要原则：头文件只提供给使用者最低限度的必要信息，而将任何不需要使用者关心的实现细节隐藏起来。

（4）合理设计接口



从软件设计的角度说，接口是一个与实现相对的概念。这与编程语言有无interface关键字并无关系。例如，Java语言中有interface关键字，而C++中并无interface关键字，但这并不说明用C++编程就不需要设计接口。

接口应与实现相分离，接口本身反映了系统设计人员对系统的抽象理解。例如，在一个面向对象的系统中，系统的各种功能是由许多不同对象协作完成的。在这种情况下，各个对象内部的实现并不是最重要的；而各个对象之间的协作关系则成为系统设计的关键。小到不同类之间的通信，大到各模块之间的交互，在系统设计之初都需要重点考虑，这也是系统设计的主要工作内容。接口的设计一旦完成，就应该尽可能保持稳定。越是到程序开发的后期，接口的变化所带来的损失（即由此引发的代码的修改量）就越大。这也是“面向接口编程”所体现的中心思想。

接口总是相对于实现而言的。对于类而言，所谓接口，就是指该类能够提供给外部调用的公有成员函数。对于模块而言，接口就是该模块可以提供给外部调用的外部函数。从这个意义上说，接口好像是单向的。但实际上，接口是一个类（或模块）对外交换信息的窗口，这个窗口既可以实现外部对类（或模块）的调用，同时，也可以实现类（或模块）对外部功能的调用。因此，接口有这样的特性：需要使用一个功能，但是并不确定这个功能是如何实现的，就可以通过接口来调用这个功能。接口可以表现为一个函数指针，一个虚函数，或者一个抽象类等。

对于代码重用而言，合理设计接口有两个方面的意义：一是可重用代码使用起来更加方便；二是通过接口设计解除模块间不必要的依赖关系。

在进行软件开发的时候，程序员要同时考虑两件事情：一是如何成功地完成当前的开发任务；二是如何为将来积累可重用资源（代码）。其实，这两者之间应该是相通的，一个高度可重用的代码结构往往有着更好的可扩展性和可维护性，也就表明当前的开发非常成功。不过，出于进度的压力，程序员往往会把精力集中在完成当前的任务上。我们在设计一个类的时候，会定义一些公有的成员函数，它们就是该类对外所能提供的服务（接口）。如果仅仅是为了满足当前项目的需要，这些类的设计可能存在缺陷。也就是说，没有将复杂的实现隐藏在简单的接口之下，这样，将来重用这些代码时，用户就不得不直接跟复杂的实现打交道，增加了重用的难度。因此，合理设计接口，将复杂的实现隐藏起来，对代码的重用十分重要。

依赖关系往往是代码复用最大的羁绊。因此，尽可能解除模块间的依赖关系，是程序设计的努力方向之一。从技术的角度来说，巧妙地设计接口，就可以最大限度地解除依赖。由于接口并不关心实现细节，所以接口是依赖的终点，接口不需要依赖任何东西。依赖接口是安全的，不会带来更多的依赖关系。当一个模块的确需要依赖外部时，必须尽量做到依赖接口，而不是实际的东西。

下面是一个计算函数积分的例子。被积函数f1()用外部函数的形式表示。

```
/// Program 12.3-4 ///
#include <iostream>
#include <math.h>
using namespace std;
double f1(double x){
    return x;
}
```

```

double f2(double x){
    return x*sin(x);
}

//将积分区间分成n段
double onepass(double x1,double x2,int n){
    double x;
    double deltax;
    double result=0.0;
    deltax=(x2-x1)/n;
    x = x1+0.5*deltax;
    for(int i=0;i<n;i++){
        result += f1(x)*deltax;
        x += deltax;
    }
    return result;
}

double integration(double x1, double x2){
    double result,newresult;
    int n=2;
    result = onepass(x1,x2,n);
    n *=2;
    newresult = onepass(x1,x2,n);
    while(fabs(newresult-result)>1E-5){
        result = newresult;
        n *=2;
        newresult = onepass(x1,x2,n);
    }
    return newresult;
}

int main(){
    cout << "Integration of f1 is " << integration(0,1.0) << endl;
}

```

/// Program 12.3-4 ///

程序的运行结果是：

Integration of f1 is 0.5

计算积分的函数是Integration(), 它依赖于函数onepass(). 函数onepass()以一定的精度（将积分区间分成n等份）计算在特定区间[x1,x2]上函数f1()的积分。Integration()对onepass()的依赖是合理的、“健康的”。因为onepass()不需要被其他函数调用，它是“专门”为Integration提供服务的。当积分的计算精度不够时，会将积分区间等分2n等份，以获得更高的计算精度。直到连续两次积分的结果没有显著变化为止。

但是，在上面的程序中，函数onepass()对函数f1的依赖却是不合理的。如果被积函数发生变化（例如要对f2()积分），那么只能手动修改函数onepass()的源代码。这样设计出来的代码是没有可重用性的。

解决这个问题的方法，就是让函数onepass()依赖于一个接口，而不是依赖于一个具体的函数。在这里，就是解除onepass()对f1()的依赖。改造后的程序如下。

```
/// Program 12.3-5 ///
#include <iostream>
#include <math.h>
using namespace std;
double f1(double x){
    return x;
}
double f2(double x){
    return x*sin(x);
}
typedef double (*PFunc)(double x);
//将积分区间分成n段
double onepass(double x1,double x2,int n,PFunc f){
    double x;
    double deltax;
    double result=0.0;
    deltax=(x2-x1)/n;
    x = x1+0.5*deltax;
    for(int i=0;i<n;i++){
        result += f(x)*deltax;
        x += deltax;
    }
    return result;
}
double integration(double x1, double x2, PFunc f){
    double result,newresult;
    int n=2;
    result = onepass(x1,x2,n,f);
    n *=2;
    newresult = onepass(x1,x2,n,f);
    while(fabs(newresult-result)>1E-5){
        result = newresult;
        n *=2;
        newresult = onepass(x1,x2,n,f);
    }
}
```

```

    return newresult;
}
int main(){
    cout << "Intergration of f1 is " << integration(0,1.0,f1) << endl;
    cout << "Intergration of f2 is " << integration(0,1.0,f2) << endl;
}
/// End of Program 12.3-5 ///

```

程序的运行结果是：

```

Integration of f1 is 0.5
Integration of f2 is 0.301168

```

修改代码之后，函数onepass()不再直接依赖被积函数，而是依赖一个函数指针f。f的类型是double (*) (double)，代表一个合法的一元函数。f就是代表一元被积函数的接口，在onepass()实际被调用的过程中，函数f1()和f2()等都是f的具体实现。

函数Integration()也要做相应的修改，它会多带一个函数指针参数，以便将指向被积函数的指针传递给函数onepass()。这样，对于任意的一元可积函数，都可以调用函数Integration()进行有限区间上的数值积分。函数Integration()就成了真正的可重用代码。

除了利用函数指针进行接口设计，在C++中，虚函数和抽象类也是接口的常见表现形式。例如下面的程序。

```

/// Program 12.3-6 ///
#include <iostream>
using namespace std;
class Person{
public:
    virtual void Happy() const =0;
    virtual char *Name() const =0;
};
class likePeter: public Person{
public:
    void Happy() const{
        cout << "sing a song." << endl;
    }
    char *Name() const{
        return "Peter";
    }
};
class likeJohn: public Person{
public:
    void Happy() const{
        cout << "have a dance." << endl;
    }
}

```



```

char *Name() const{
    return "John";
}
};

void whenHappy(const Person &p){
    cout << "When someone like " << p.Name() << " is happy, they ";
    p.Happy();
}

int main(){
    likePeter p;
    likeJohn j;
    whenHappy(p);
    whenHappy(j);
}

/// End of Program 12.3-6 /////

```

程序的输出结果是：

When someone like Peter is happy, they sing a song.
When someone like John is happy, they have a dance.

函数whenHappy()描述了当一个人很快乐时所可能有的行为。如果让它的参数固定为某个具体的类，那么这个函数就只能描述某一个（类）人的行为，而不能描述具有不同性格的人的行为。为此，可以让函数whenHappy()依赖一个抽象类Person，在实际调用该函数时，可以向它传递实现抽象类的具体类的对象，以描述不同的人的行为。在这里，抽象类Person就是一个接口，它解除了函数whenHappy()对具体类的依赖。

设计接口，就是站在概念的层次上（而不是实现的层次上）来分析问题。良好的接口设计来自于对用户需求的充分理解，而非程序员苦思冥想的结果。

(5) 如何复用代码

如果手头已经有了可重用代码，应该怎样复用？用C++编写而成的可重用代码，有两种表现形式：源代码和目标代码（通常以库的形式或以obj文件的形式提供）。因此，在复用代码时，也有两种可能：拷贝源代码、链接目标代码。

除非万不得已，永远也不要拷贝源代码。

原因是：如果不需对源代码做出修改，那么为什么不直接使用目标代码呢？为什么还要经过编辑、编译等不必要的过程呢？

如果在同一个工程内部拷贝源代码，那么在同一个工程内，相同的代码就会重复出现，这不仅仅是对源文件资源、编译时间资源的一种浪费；更重要的是给将来的维护带来麻烦：如果其中的一份代码发生改变，如何保证其他相同的代码同步变化呢？

如果把源代码从一个工程拷贝到另外一个工程，那么已经说明了这部分源代码是可重用的，并且经过了原来的工程的检验。既然如此，为什么不直接使用目标文件呢？如果这份源代码是没有bug的，那么在拷贝的过程中，可能会出现bug。如果这份代码是有bug的，那么这些bug就会被复制到当前的工程中来。

有些程序员把其他的源代码拷贝到自己的工程中来，经过改造之后形成了自己的程序开



发。这似乎也是一种代码重用。但是，可以从两种情况否定这种重用代码的方式。

①如果修改的幅度不大，说明原来的代码设计上有缺陷。可以对原来的代码进行小幅度的修改，使其更加通用。

②如果修改的幅度很大，那么可能只是原来的代码在某一方面有某些参考作用而已。需要大幅度修改的代码已经没有重用的意义了。

12.4 为什么需要设计模式

设计模式（Design Pattern）是一个被反复谈论的关于软件架构的主题，它为特定上下文中的常见设计问题提供了解决方案，并描述了这种解决方案的结果。在长期的软件开发实践中，软件工程师们积累了大量有着重要价值的设计经验。这些经验被不同的人在不同的场合以不同的方式在自觉地或自发地应用着。而设计模式的出现改变了这种现状。将这些成功的设计经验及相关技术记录下来，并以一种规范的形式加以表述，以便进行交流和复用，就形成了各种不同的设计模式。设计模式使系统设计人员之间的交流有了“官方语言”。

从实践的角度看，设计模式有两个重要的属性。首先，它们描述了经过验证的、成功的设计技术，这些技术可以按上下文相关的方式进行定制，以便满足新的设计场合的要求。其次，并且可能更重要的是，在提及某个特定模式的应用时，不仅包括其中用到的技术，还包括应用该模式的动因，以及应用后所达到的效果。

设计模式是和面向对象紧密联系的。使用模式的目的就是为了使各个类之间相互分离，每个类不必关心其他类的内部实现（低耦合，有利于扩展），同时还要方便类与类之间的信息交互。因此，设计模式是实践中优良的设计经验的总结，它能帮助程序员设计出结构良好的可重用代码，而不是让程序员一切都从头做起。同时，它也提供了一种标准的描述方式，以方便程序员讨论和交流他们的设计。以往，当程序员向他人描述自己的设计时，由于缺乏统一的标准，每个人的描述方式都是繁琐而各不相同的。这种方式低效且不够精确，不利于面向对象设计的文档化、程序员之间的交流以及有效地对设计方案进行评议。这并不是说，用于复杂的面向对象的设计技术尚不存在，而是这些技术尚未以一个共享的、通用术语的方式为整个编程社群所使用。设计模式解决了这个问题。我们现在可以像描述算法设计一样高效、毫无歧义地描述面向对象的设计。

就像标准的算法都有一个公认的名字（如“折半查找”、“快速排序”等）一样，每一种设计模式也有一个标准的名称。例如，当我们看到Bridge模式被应用于系统设计时，我们立即可以知道这个模式所包含的各项内容。这个模式被人们反复讨论过，人们了解这个模式的应用场景以及它所涉及的方方面面。

设计一个类时，如果该类有两个（或两个以上）的维度可以发生变化，若是以派生的方式实现该类，它所派生出的子类的数目将是庞大的。例如，我们设计一个盒子Box类，假设它有大小（Size）和颜色（Color）两个维度可以发生变化，第一个维度有2种选择（big和small），第二个维度有3种选择（red、green、blue），那么我们必须实现6（即 2×3 ）个派生类，它们是BigRedBox、SmallRedBox、BigGreenBox、SmallGreenBox、BigBlueBox、SmallBlueBox。显然，这种状况可以得到改善，因为这两个维度是独立变化的。可以在设计类时将抽象部分与它的实现部分相分离，使它们都可以独立地变化。还是以2个维度为例，假设第一个维度有n1个值，第二个维度有n2个值，那么以多层次派生的方式产生的子类的数目是n1×n2，而以Bridge



模式产生的类的数目是n1+n2，并且一个类与其他类之间的耦合关系减弱，有利于代码的扩展和复用。

在Bridge模式中，抽象数据类型的实现被分离成一个接口类和一个实现类。这两个类可以各自独立地变化，同时又可以动态组合。不但减少了重复代码，而且使程序有了更清晰的组织结构。Bridge模式的类图描述如下：

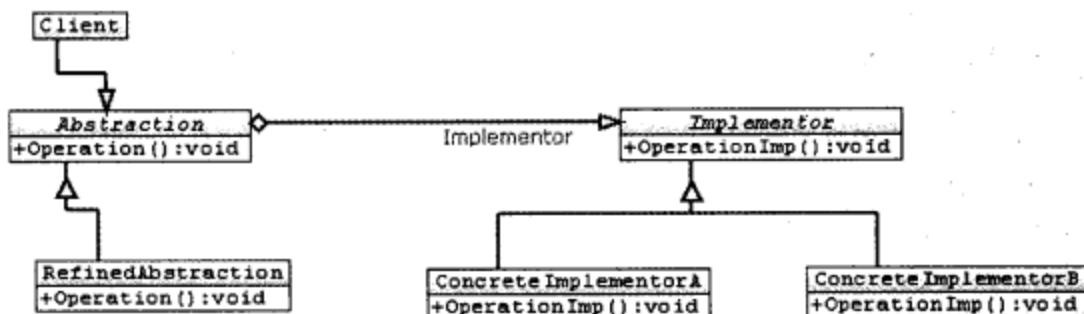


图 12-1 Bridge 模式类图

其中Client是Bridge模式的使用者（类的客户）；Abstraction是抽象类接口（代表第一个维度），维护对行为实现（Implementor）的引用；RefinedAbstraction是Abstraction的子类（代表了第一个维度的一种选择）；Implementor是行为实现类接口（代表了第二个维度）；Concrete ImplementorA是Implementor的子类（代表了第二个维度的一种选择）；ConcreteImplementorB是Implementor的子类（代表了第二个维度的另一种选择）。Abstraction的派生类与Implementor的派生类的动态组合，是依靠传递Implementor的引用（或指针）来实现的。下面的程序演示了实现Bridge模式的一个代码模板，在实际开发过程中要使用Bridge模式，可以通过对该程序代码做少量修改而实现。

```

/// Program 12.4-1 ///
/** abstraction.h ***/
#ifndef _ABSTRACTION_H_
#define _ABSTRACTION_H_
class AbstractionImp;
class Abstraction{
public:
    virtual ~Abstraction();
    virtual void Operation() = 0;
protected:
    Abstraction();
private:
};
class RefinedAbstraction:public Abstraction{
public:
    RefinedAbstraction(AbstractionImp* imp);

```

```

● ~RefinedAbstraction();
void Operation();

protected:
private:
    AbstractionImp* _imp;
};

#endif
/** end of abstraction.h **/

/** abstraction.cpp **/
#include "Abstraction.h"
#include "AbstractionImp.h"
#include <iostream>
using namespace std;
Abstraction::Abstraction(){
}
Abstraction::~Abstraction(){
}
RefinedAbstraction::RefinedAbstraction(AbstractionImp* imp){
    _imp = imp;
}
RefinedAbstraction::~RefinedAbstraction(){
}
void RefinedAbstraction::Operation(){
    _imp->Operation();
}
/** end of abstraction.cpp **/

/** abstractionimp.h **/
#ifndef _ABSTRACTIONIMP_H_
#define _ABSTRACTIONIMP_H_
class AbstractionImp{
public:
    virtual ~AbstractionImp();
    virtual void Operation() = 0;
protected:
    AbstractionImp();
private:
};
class ConcreteAbstractionImpA:public AbstractionImp{

```

```

public:
    ConcreteAbstractionImpA();
    ~ConcreteAbstractionImpA();
    virtual void Operation();
protected:
private:
};

class ConcreteAbstractionImpB:public AbstractionImp{
public:
    ConcreteAbstractionImpB();
    ~ConcreteAbstractionImpB();
    virtual void Operation();
protected:
private:
};

#endif

/** end of abstractionimp.h **/

/** abstractionimp.cpp **/
#include "AbstractionImp.h"
#include <iostream>
using namespace std;
AbstractionImp::AbstractionImp(){
}
AbstractionImp::~AbstractionImp(){
}
ConcreteAbstractionImpA::ConcreteAbstractionImpA(){
}
ConcreteAbstractionImpA::~ConcreteAbstractionImpA(){
}
void ConcreteAbstractionImpA::Operation(){
    cout<<"ConcreteAbstractionImpA...."<<endl;
}
ConcreteAbstractionImpB::ConcreteAbstractionImpB(){
}
ConcreteAbstractionImpB::~ConcreteAbstractionImpB(){
}
void ConcreteAbstractionImpB::Operation(){
    cout<<"ConcreteAbstractionImpB...."<<endl;
}

```

```

/** end of abstractionimp.cpp **/


/** main.cpp **/
#include "Abstraction.h"
#include "AbstractionImp.h"
#include <iostream>
using namespace std;
int main(){
    AbstractionImp* imp = new ConcreteAbstractionImpA();
    Abstraction* abs = new RefinedAbstraction(imp);
    abs->Operation();
    getchar();
    return 0;
}
/** end of main.cpp **/


/// End of Program 12.4-1 ///

```

从上面的例子可以看出，模式的名字是关于某项特定技术的诸多信息和经验的高效且无歧义的“句柄”，在设计和文档中小心并正确地使用模式和模式术语，可以使代码和设计更加明晰。

在介绍设计模式的书籍或文献中，设计模式以某种正式的结构被描述。尽管这些结构可能不尽相同，但不管哪种描述方式，均包含以下4个必不可少的部分。

首先，设计模式必须有一个毫无歧义的名字。这个名字最好能精确地概括该设计模式的特点，避免使用一些早已广泛使用的术语。例如，使用术语“包装器（Wrapper）”来命名一个设计模式就不是一个好的选择，因为它早已被广泛使用并且具有许多不同的含义。而像“Bridge”、“Strategy”、“Facade”、“Object Adapter”这些名称都可以用来代替“Wrapper”，而且更为精确。使用精确的模式名字比使用不那么精确的名字具有明显的优势，就像术语“二分查找”比“查找”更精确、更有意义一样。

其次，模式描述必须定义该模式所能解决的问题。这种描述可以相对宽泛，也可以相对狭窄。

再次，模式描述要记述该问题的解决方案。根据陈述的问题，该解决方案可以相对高级，也可以相对低级。但无论如何，它应该具有足够的通用性，以便可以根据问题可能出现的不同上下文进行定制。

最后，模式描述要记述将该模式应用于某个上下文的后果。在应用该模式后，该上下文是如何发生改变的？不管是变好，还是变坏。

设计模式的出现也改变了程序员设计系统的方式。在对各种设计模式有了深入的理解之后，程序员就有了构造系统的“半成品”，而不是一切都要从头做起。设计模式常常被描述为“微架构（micro-architecture）”，它们可以与其他的模式进行组合从而生成一个新的架构。程序员的工作就是选择适当的模式并有效地对其进行组合。虽然这项工作同样是需要设计方面的经验和细致思考的，但毕竟对高效地完成设计有利。设计一旦完成，别人对设计的理解也变得相对容易，只要他具备一些必需的模式方面的知识即可。



12.5 再论 C++的复杂性

C++语言已经有了20多年的历史。作为一门影响广泛的编程语言，它所受到的关注和争论恐怕是任何一门其他的语言所不能比拟的。十几年前，Java等新生语言的出现曾导致“C++信任危机”，但最终C++以自身非凡的品质屹立于主流编程语言的行列。在有着众多编程语言可以选择的今天，到底还有没有必要学习C++？怎样学习C++？怎样使用C++？对于广大的程序员，特别是对于刚刚接触编程的初学者，这些问题都是至关重要的。

C++语言遭受批评最多的是它的复杂性。对于这个话题，已经有很多的文献讨论过。在这里，不想提出什么新的观点（其实恐怕也提不出什么新观点），只是想把作者支持的观点罗列一下，以期对希望深入了解C++的读者做一番指引。

（1）C++真的很复杂吗

这个问题的答案是肯定的，这一点应该确信不疑。

从C++语言本身的发展和组成来看，C++语言并不是一种单一、“纯粹”的编程语言，它有着复杂的内部结构。

最初，C++仅仅是在C的基础上附加了一些object-oriented（面向对象）的特性。C++最初的名字——“C with Classes”就非常直观地表现了这一点。

以后，C++语言经历了一个大胆而冒险的成长过程，它不断地吸收新的思想、特性，今天的C++已经成为一个 multiparadigm programming language（多范式的编程语言），一个囊括了procedural（过程化）、object-oriented（面向对象）、functional（函数化）、generic（泛型）以及metaprogramming（元编程）特性的联合体。这些能力和弹性使C++成为无可匹敌的工具，但也使它变得越来越复杂。

因此，不要将C++视为一门单一的语言，而是一门由几个相互关联的子语言(sublanguage)组成的联合体。在每一个特定的子语言中，它的特性趋于直截了当，简单易记。但你从一个子语言转到另外一个，它的规则也许会发生变化。概括地说，C++主要的子语言有4个：

C——归根结底，C++依然是基于C的。blocks（模块）、statements（语句）、preprocessor（预处理器）、built-in data types（内建数据类型）、arrays（数组）、pointers（指针）等，全都来自于C。在很多方面，C++提出了比相应的C版本更高级的解决问题的方法，例如内联函数、引用、函数和操作符重载等。这些特性能够和传统的C很好地结合在一起，可以视为对C的扩展，体现了C++的“A better C”的初衷。

Object-Oriented C++——C++的这部分就是C with Classes涉及到的全部：classes（类）（包括构造函数和析构函数）、encapsulation（封装）、inheritance（继承）、polymorphism（多态）、virtual functions（dynamic binding）（虚拟函数（动态绑定））等。C++的这一部分直接适用于object-oriented design（面向对象设计）的经典规则。

Template C++——这是C++的 generic programming（泛型编程）部分，大多数程序员对此都缺乏经验。template（模板）的考虑已遍及C++，而且好的编程规则中包含特殊的template-only（模板专用）条款已不再不同寻常。实际上，templates（模板）功能极为强大，它提供了一种全新的 programming paradigm（编程范式）——template metaprogramming（TMP）（模板元编程）。

STL——STL是一个template library（模板库），但它是一个非常特殊的template library（模



板库)。它将containers(容器)、iterators(迭代器)、algorithms(算法)和function objects(函数对象)非常优雅地整合在一起。但是,templates(模板)和libraries(库)也可以围绕其他的想法建立起来。STL有很多独特的处理方法,使用STL编程时,需要遵循它的规则。

C++的四种子语言(sublanguages)紧密地结合在一起,但它们的确又有各自鲜明的风格。当从一种子语言转到另一种时,为了高效编程有时需要改变编程的策略,这是C++程序员可能遇到的情形,对此必须有心理准备。例如,使用built-in(内建)(也就是说,C-like(类C的))类型时,pass-by-value(传值)通常比pass-by-reference(传引用)更高效,但是当你从C++的C部分转到Object-Oriented C++(面向对象C++),由于传值调用会导致建立参数的副本(调用用户自定义的构造函数和析构函数),所以更好的做法是传const引用。在Template C++中工作时,这一点更加重要,因为,在这种情况下,你甚至不知道你的操作涉及到的对象的类型。然而,当你进入STL,由于iterators(迭代器)和function objects(函数对象)以C的pointers(指针)为原型,对于STL中的iterators(迭代器)和function objects(函数对象),古老的C中的pass-by-value(传值)规则又重新生效。

C++不是使用一套规则的单一语言,而是federation of four sublanguages(四种子语言的联合体),每一种都有各自的规则。有了这样的理解,就能更清楚地了解C++的内部结构,并能根据不同的应用需求使用不同的子语言,充分发挥C++语言的长处。

从学习和使用C++语言的角度看,C++的复杂性体现在以下三个方面:①学习周期长。谁都想使用易学易用的东西,更何况在IT技术日新月异的今天,但C++偏偏在这方面不给任何人“面子”。业界的规律是,成为一个合格的C++程序员一般需要3-5年的时间;②开发效率低。主要是历史原因,C++的诸多库都停留在“很低”的层次上,使用便利性无法与RAD工具相提并论。有人提议将C++库的层次“提高”,但这是一项非常困难的工作,原因是这与C++语言的设计理念是有冲突的,C++希望最大限度地保持通用性、底层性;③容易犯错,维护难度大。C++是一种功能强大且自由度极大的语言,使用C++的过程中一不小心就容易犯错误,特别是对于初学者。要能够自如地使用C++语言需要长时间的“修炼”。

(2) C++语言复杂的原因

C++复杂的真正原因是什么?对此,大家有不同的看法。因为是学院派的东西?不,学院派出来的东西就一定复杂吗?这个理由站不住脚;是“体积”过于庞大吗?经历了近三十年的发展,C++的触角已经遍及当今世界学术、工业界的方方面面,“体积”是够大的,但“体积”虽大,结构却很清晰的C++,并不会因此而复杂;是C++支持的编程范式太多了吗?也不是,新生的语言几乎都在走C++的成功范式,Python和Ruby等新型动态语言的范式甚至多于C++,然而它们却以简单和开发效率高著称。其实C++真正复杂的原因在于,坚守“三大原则”,决不妥协:①对C的完全兼容;②静态类型检查;③最高性能。而其中“最高性能”又是这三大原则中的重点。既要发展新的特性,同时又要保持最高的性能,这是C++语言复杂性的根本原因。C++没有采用一些可能会降低程序性能的做法(如采用垃圾回收机制等),而这些做法是有可能降低C++的复杂性的。Stroustrup博士在多种场合下都表示,对C++的设计没有大的后悔之处,原因在于对三大原则的坚持首先是正确的,然后,若坚守三大原则,即使重新设计一遍C++,结果也与今日相差不远。

(3) 需要学习和使用C++吗

既然C++十分复杂,那么有必要学习和使用C++吗?

对于这个问题,无法给出强制性的回答。在这个世界上,一定存在从来不用C++编程而



仍然能够胜任特定编程工作的程序员，也许它们所使用的语言就是Java、C#或者是其他编程语言。但是，我仍然强烈建议：即使是不用（或者不主要使用）C++进行编程开发，仍然应该学习和实践C++。这主要是基于如下的几点理由。

①C++是一门贯通低级和高级特性的编程语言。C++语言向下兼容C语言，能够直接同计算机的硬件和底层打交道，甚至能够直接使用内联汇编；向上，C++语言支持几乎所有其他高级语言所支持的高级特性。正如前面所介绍的，C++语言是4种子语言的结合体，它所能支持的特性的丰富程度也是其他语言所难以企及的。如果要挑选一门“无所不能”的语言，那么这门语言一定是C++无疑。对于一个能够静下心来，能够持续不断努力提高自己对计算机系统理解程度（计算机体系结构、硬件、操作系统、应用开发、软件项目和过程管理）的程序员来说，C++语言是一个绝佳的选择。

②C++是一种高效的语言。C++程序的执行效率与C程序相当，同时又提供了诸多的高级特性。这样，C++语言为程序员创造了这样一种可能：在利用各种高级特性（面向对象方法、范型编程等）充分表达设计思想、解决各种复杂问题的同时，保持应用程序的高效运行。这也是其他的编程语言难以做到的。

③C++是一门复杂的语言。这个观点听起来有些怪异。C++语言的复杂性往往是造成人们放弃C++的原因，但同时，C++语言的复杂性也有可能成为人们选择C++语言的原因。C++的先驱和大师Andy Koenig在他的《C++沉思录》里回击了对C++复杂的攻击。他认为，选择什么样的编程语言，取决于要解决的问题。世上没有万灵药，要解决复杂的问题，必然要依赖于复杂的工具。C++程序员是实用主义者，他们首先要保证能解决问题，其次才能谈得上其他。实际上，要解决的问题是复杂的，计算机系统是不完美的，人类的自然语言体系和表达习惯就更是不完美和复杂的，而一门成熟的通用编程语言，要在这三极之间保持平衡，谈何容易！Java语言通过削减矛盾（用虚拟机代替真实机器），削减表达能力来获得简单性，这也同时限制了它在实时性高计算密集的领域里得到应用。无论是调度仿真实时控制还是媒体编辑，一旦触及重量型的关键性应用，除了C++你别无选择。C++的复杂性源于其对高效解决问题的承诺。这就好比，在现实生活中，思想简单的人不能委以重任。

④C++是一门成熟的编程语言。这并不是说其他的编程语言就不成熟。成熟是一种相对的概念。C++语言在其20多年的发展和使用的过程中，开发了无数成功的软件系统，积累了丰富的成功案例和可重用资源。其数量之大，影响之广，也是首屈一指的。有兴趣的读者可以光临Bjarne Stroustrup博士的主页，了解一下C++语言在业界创造的辉煌战绩。

（4）如何应对C++的复杂性

尽管C++的复杂性有其产生的深刻背景，但复杂性确实是一个问题。在实践上最突出的表现就是开发效率的降低，毕竟简单易用的工具能带来生产率的提高，这已是毋庸置疑的事实。但是，C++的复杂性导致开发效率的降低只是一种表象，它是没有对复杂性进行有效控制而产生的后果。换句话说，问题不在于C++的复杂性，而在于使用C++的人没有有效地控制这种复杂性。

那么，应该如何应对C++的复杂性能？下面给出几条具体的建议。

①用沉稳的心态去学习C++。

学习编程语言，掌握语法，能上手实践，不过是万里长征迈出了第一步。更何况像C++这样的语言，要做到掌握各个子语言的基本内容，都不是一件很容易的事情。所以，心态一定要平稳，着急不来，更不可轻狂。真正要掌握语言，非得集中精力学习实践一两年，将该



语言所擅长领域的应用问题熟悉过一遍，才有可能。若论精通，那是一个没有止境的过程。Henry Spencer用了30年C，仍乐此不疲；Pragmatic Programmer中评价Ruby说，学上四个小时就可以用它解决实际问题，但是10年之后还为它层出不穷的新意感到惊讶。真正掌握C++语言，然后再熟悉一两门层次不同、思维不同的语言，那就是更高层次的追求了。

要特别注意的是，这也是一个心理学的问题：C++语言中总是存在着一些新奇的特性，它会引起你强烈的兴趣，将你的注意力从真正有用的事情中分离出来，更不用说那些有可能对真正问题带来（在某种程度上）漂亮的解决方案的语言特性了。这些被称之为“奇技淫巧”的东西即使能短暂地给你带来自豪的感觉，也不应该成为你学习C++的主流。

②采用科学的学习方法。

全面掌握C++固然重要，但是，那不等于说，只有掌握了C++的全部你才能用它来解决问题。你可以把你对C++的理解限制在一个相对简单的程度，只要你需要解决的问题的复杂度不超过你所掌握的工具的复杂度。初学者要把C++分解为逻辑层次上、难度上比较独立的部分，根据自己的需要循序渐进地学习，利用每一部分的所学解决能够解决的问题。只有这样，才能够学得扎实，同时培养信心和兴趣。随着学习的深入，逐步将各个部分衔接贯通起来，把自己逐步培养成真正的“高手”。

③正确地使用C++。

C++被错误地使用是一种很普遍的现象，这也是C++遭致“过于复杂”的抱怨的真正原因。C++语言由4种子语言组成，C++语言提供了如此丰富的特性和自由度。如何选择这些特性体现了C++程序员的真正“功力”和成熟度。

首先，要小心选择你所使用的子语言。例如，C++是向下兼容C的。那么，是不是在任何场合下，都要使用C++的面向对象的特性呢？或者无论在什么情况下，都选择C，因为C更简单？这种一刀切的思维是不可取的。显然，有些领域C是更好的选择。例如设备驱动开发就不需要那些OOP/GP技巧，而只是简单的处理数据，真正重要的是程序员确切地知道系统是如何运转的，以及他们正在做什么。而像写操作系统这样的工作，很大一部分也不需要OOP/GP。然而，在更多的领域，抽象与效率是并重的，这些正是C++的面向对象的特性适用的场合。

其次，要充分利用现有的、经过实践检验的资源。代码重用是现代软件工程提倡的一种做法，不仅因为它可以提高开发的效率，还因为它可以降低程序的复杂程度。如果一个高效的容器（或智能指针）能把你从无聊的手动内存管理中解放出来，为啥还要用那原始的malloc/free呢？如果一个好的string类或正则表达式类能把你从繁琐的字符串处理中解脱出来，那么为啥还要手动去做这些事呢？如果一个“transform”（或“for_each”）能够用一行代码把事情漂亮搞定，为啥还要手写一个for循环呢？

再次，控制你的代码的复杂程度。C++语言不是为了复杂而复杂，而是因为要解决负责的问题而引入了复杂的机制。问题的关键在于，程序员有时是自己把问题搞复杂了。例如，在C++中，一个普通程序员很可能会写出一堆高度耦合的类，很快情况就变得一团糟。但这不是C++的问题，这种事情很可能发生在任何一门面向对象语言中，因为总是有程序员在还没弄懂什么是HAS-A和IS-A之前，就敢于在类上再写类，就这样一层一层堆砌上去。他们学会了在一门特定的语言中如何定义类，如何继承类的语法，然后就认为自己已经掌握了OOP的精髓了。由于C++是如此灵活，很多问题在C++中都有好几种解决办法，于是在这些选择中进行权衡本身就成了一个困难。这也使得程序员犯错误的可能性增加了。所以，掌握



优秀的设计思想（比如说优先使用组合、而不是继承），或者遵循C++社群这些年积攒下来的智慧，或者干脆只使用C++语言中的C with class部分以规避复杂性的风险，都是程序员需要不断学习和不断实践的。

总之，正确使用C++所应遵循的原则是：了解C++的高级特性，用简单的方法解决简单的问题，用简单的形式解决复杂的问题（将复杂的解决方案包装在简单的形式之下，重用前人的劳动成果，遵循最佳的实践）。

基础
入门
进阶
实践
进阶
进阶

PDF

参 考 文 献

- [1] Stanley B Lippman, Josee Lajoie, Barbara E Moo. C++ Primer 中文版. 第 4 版. 李师贤, 蒋爱军等译. 北京: 人民邮电出版社, 2006.
- [2] Stephen C Dewhurst. C++必知必会. 荣耀译. 北京: 人民邮电出版社, 2006.
- [3] Andrew Koenig, Barbara Moo. C++沉思录. 黄晓春译. 北京: 人民邮电出版社, 2002.
- [4] Bruce Eckel. C++编程思想. 刘宗田等译. 北京: 机械工业出版社, 2003.
- [5] Scott Meyers. Effective C++: 改善程序技术与设计思维的 55 个有效做法. 第 3 版. 侯捷译. 北京: 电子工业出版社, 2006.
- [6] 林锐, 韩永泉. 高质量程序设计指南——C++/C 语言. 第 3 版. 北京: 电子工业出版社, 2007.