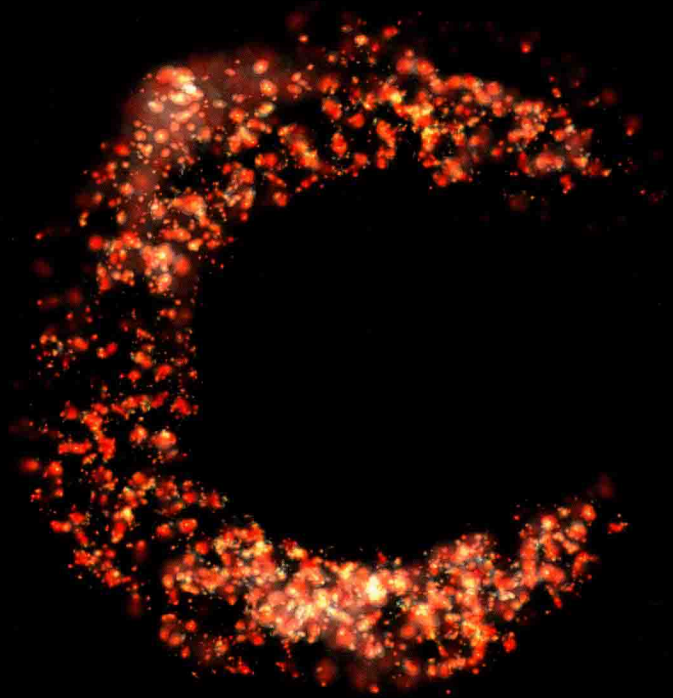
Broadview
www.broadview.com.cn



C++语法详解

本书全方位阐述了C++语法知识点,语法示例短小精悍，案头必备，方便速查
细致的图解分析，让读者更易理解语法原理，确保读者知其然更知其所以然

黄勇 / 编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

C++语法详解

本书主要内容如下：

- ◎整型、字符型和浮点型专题
- ◎运算符、表达式和左值专题
- ◎指针和数组专题
- ◎类基础、类作用域及相关运算符专题
- ◎名称空间专题
- ◎运算符（操作符）重载专题
- ◎对象模型与虚函数表专题
- ◎I/O专题
- ◎预处理器、typeid和强制类型转换专题
- ◎声明、定义、复杂声明和typedef专题
- ◎选择语句和循环语句专题
- ◎函数和标识符的作用域专题
- ◎构造函数、复制构造函数和析构函数专题
- ◎类中的成员专题
- ◎继承、虚函数与多态性专题
- ◎模板专题
- ◎异常专题
- ◎string类专题

上架建议：编程语言 / C++

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：安　娜
责任编辑：葛　娜
封面设计：李　玲

ISBN 978-7-121-31655-5

定价：89.00元

C++语法详解

黄勇 / 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书对C++的语法进行了全面介绍和深入讲解，内容包括：C++整型、字符型、浮点型、声明、定义、typedef、运算符、表达式、左值、选择语句、循环语句、指针、数组、函数和标识符的作用域、类基础、类作用域及相关运算符、构造函数、复制构造函数、析构函数、名称空间、类中的成员、运算符（操作符）重载、继承、虚函数、多态性、对象模型、虚函数表、模板、I/O、异常、预处理器、typeid、强制类型转换和string类等。本书层次分明，由浅入深，各章节相对独立，语法示例短小精悍，方便对有疑惑的语法进行速查。学习完本书，读者不会再对C++的各种语法感到困惑。

本书适合有一定C++基础、对C++的语法有疑惑、想深入了解C++语法细节的人员阅读。本书同时也可以作为解决C++语法问题的参考书；对于学习过C++或已精通C++的人员，也是一本不错的资料查阅手册。

图书在版编目（CIP）数据

C++语法详解 / 黄勇编著. —北京：电子工业出版社，2017.7
ISBN 978-7-121-31655-5

Ⅰ.①C… Ⅱ.①黄…Ⅲ. ①C 语言－程序设计 Ⅳ.①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 119321 号

策划编辑：安　娜
责任编辑：葛　娜
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：31.25　字数：697 千字
版　　次：2017 年 7 月第 1 版
印　　次：2017 年 7 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

本书具有如下特点：

（1）本书是专门讲解 C++语法规则的书籍，因此书中不会介绍任何有关程序设计的内容（比如编写一个计算规形面积的程序等）。本书将语法问题分离出来，避免既不像写语法的书，也不像写程序设计的书。

（2）书中的示例程序都使用简短的名字，比如 a, b, A 等，以便于记忆，而不会使用很长的名字。

（3）一个知识点能用一段话讲解清楚的，尽量不使用两段话。一个知识点一个标号，方便查阅和增补。

（4）一个知识点列举一个单独的简短易懂的程序作为示例。大多数教材都喜欢在第 1 章开头定义一个变量，然后一直到章尾都在使用那个变量作为示例。本书打破传统，一个知识点就是一个单独的示例，不与上一个知识点的示例拉上关系，更不会与上一章的内容拉上关系，让读者能够随时独立复习每个知识点，而不用再去复习不必要的章节内容。

（5）本书的示例程序主要是针对语法问题的，示例程序每行都有注释，尽量做到把每个语法问题都反映出来。

（6）本书引用了大多数教材上没有提到的一些概念，并对这些概念做了深入介绍。

（7）本书对某些难点内容做了细致的图解分析，让读者更容易明白难点的原理。书中的图是专门针对语法问题的，尽量做到让读者看图就能明白其原理。

（8）本书对指针和数组的理解有独到的见解，学完数组和指针章节会给读者耳目一新的感觉。

（9）本书尽量做到用最少的文字、最少的篇幅描述清楚知识点，是一本真正的含金量高的图书。

由于能力有限，书中难免有错漏之处，望广大读者指出更正，不胜感激。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源：**本书提供的附录 A 和附录 B 文件，可在 *下载资源* 处下载。
- **提交勘误：**您对书中内容的修改意见可在 *提交勘误* 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 *读者评论* 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31655

目 录

第1章

C++快速入门

（1）C++是大小写敏感的语言，即在程序中大写字母C和小写字母c是不同的。

（2）关键字（保留字）：关键字是C++使用的名字，任何自定义的名字都不能与关键字相同。

（3）注意：C++中的所有符号都应在英文状态下输入，比如分号应使用“;”，而不是“；”。

示例如下：

```
/*此符号是C++的多行
注释符号*/
#include<iostream>
using namespace std;    //双斜杠符号是C++的单行注释符号
int main()
{cout << "hy" <<endl;   //将右边的表达式输出到由<<指向的左边的设备cout上。cout一般表示控制台
   cin.get();           //暂停等待输入，或按下回车键继续执行，此句可以省略
   return 0;
}
```

程序说明如下。

1. 解决执行后的结果一闪而过

有些编译器在运行C++程序时，会在独立的窗口中运行，程序执行完之后窗口就关闭了（程序执行得很快，一般情况下，窗口都是一闪而过），这样就无法看到程序的执行结果。解决方法就是在程序中想要暂停的地方加上 cin.get();语句，这样程序就会停在这里等按下回车键。一般情况下，只需将 cin.get();放在 main 函数的结尾或者 return 语句之前就行。

2. 注释

（1）注释：注释的内容是不会被编译器编译的，注释就是程序中对代码的说明性文字，这些说明性文字是为提高程序的可读性而写的，可以在注释中写入任何内容，当然也可以没有注释。

（2）单行注释使用“//”符号：C++使用“//”作为程序注释的开头，“//”后面的内容为程序的注释。但使用“//”只能注释单行的程序，若注释超过一行，到第二行又必须以“//”作为

注释的开头。

（3）多行注释使用/*...*/符号：这是 C 风格的注释，但 C++同样支持，/*和*/之间的内容为注释，以/*开头，以*/结束，这种注释方法可以跨越多行。

3．预处理器和#include<iostream>

（1）#include 是预处理器指令，预处理器就是编译器在把代码编译为机器指令之前执行的一个过程，它会在编译程序的时候自动运行，所有的预处理器都是以“#”开头的，其中 include 是预处理器的一个指令。

（2）#include<iostream>的作用：将 iostream 文件的内容添加到程序中。在将代码编译为机器指令之前，会将 iostream 文件的内容添加到程序中，这时 iostream 文件的内容将取代代码行#include<iostream>。

（3）iostream 文件的作用：这里的 io 指的是输入（input）/输出（output），使用 cout 进行输出或使用 cin 进行输入的程序必须包含 iostream 文件，因为 cout、cin 这些内容是在 iostream 文件中定义的。

4．头文件（或包含文件）和 iostream

（1）头文件（或包含文件）：像#include<iostream>中的 iostream 文件，被包含在其他文件起始处的文件被称为头文件或包含文件。

（2）可以将任何合法的 C++文件包含进其他文件，只需像上面那样使用#include<...>即可。

（3）库文件：C++编译器本身自带了很多头文件，这些头文件都有一定的功能，例如 math.h、iostream.h，其中 math.h 文件有很多关于数学计算方面的函数（函数见后文解释）。C++自带的这些标准头文件被称为库文件。

（4）在 C 语言中头文件使用扩展名.h，而 C++则不再使用扩展名了，因此老式的 C++编译器应使用#include<iostream.h>并将后面的 using namespace std;去掉，若还有更老的编译器，则应使用#include<stream.h>。因此，建议使用比较新的编译器。

（5）C++转换了 C 中的一些头文件，有些头文件还做了一些修改，并将这些文件重新命名，新命名的文件去掉了扩展名.h，并在文件名前面加上了前缀 c，比如 math.h 的 C++版本为 cmath，转换后的文件可以使用 C 所不具有的 C++特性。

5．名称空间

（1）名称空间：就是提供一个声明名称的区域，比如在某个区域声明了一些名称，那么拥有这些名称的这个区域就是名称空间。当然，你可以为这个区域取一个名字，有了名称空间就能更好地控制名称的作用范围，同时一个名称空间中的名称不会与另一个名称空间中声明的相同名称相冲突。比如重庆市有一个合川市，广西也有一个合川市，在这里“重庆市”和“广西”

就是两个名称空间。若直接说合川市并不能知道它的具体位置，但说广西合川市就能知道它是广西的，这时并不会与重庆市的合川市相冲突。因此，在使用名称空间之前应先告诉程序，将使用哪一个名称空间中的名称，这样当以后出现这个名称时就知道它来自哪个名称空间了。

（2）名称空间的创建：名称空间使用 namespace 来创建，比如 namespace sss{...}就创建了一个名称空间，该名称空间的名称为“sss”，在大括号{}中可以声明名称。

（3）使用名称空间中的名称。

- 可以使用::（作用域解析运算符）来完全限定名称空间中的名称，比如 std::cout 就表示使用的是 std 名称空间中的名称 cout。
- 使用 using 编译指令：其语法格式为“using namespace 名称空间的名称;”，表示指定名称空间中的所有名称都可用。比如使用 using namespace std;语句，就表示位于 std 名称空间中的所有名称都可直接使用。

6．std 名称空间和 C++标准库文件的关系

（1）std 是 C++的标准名称空间，所有 C++标准库文件中的名称都是在 std 名称空间中声明的。

（2）凡是需要使用 C++标准库文件（需要使用#include 将库文件包含进来）的地方都必须使用 std 名称空间，也就是应使用 using namespace std;语句。当然，也可以通过上面讲的作用域解析运算符“::”来使用 std 名称空间中的名称。

7．函数

（1）函数基础。

① 函数：函数就是一种操作或一种功能。比如使用某函数实现对一些数据进行排序，再用另一个函数进行求和运算等。

② 函数的形式为：返回类型 函数名(形参列表){函数体}。比如 int f(int a, int b){...}，表示定义了一个名称为 f，返回类型为 int（整型），且带两个 int 类型的形参变量 a 和 b 的函数。

③ 函数若没有返回值，则只需使用 void 表示函数的返回类型即可。比如 void f(){...}，关键字 void 表示没有返回类型，也就是函数没有返回值。

④ 函数的形参列表也可以为空，比如 int f(){...}。注意，即使函数的形参列表为空，也不能省略函数名后面的小括号。

⑤ 函数由两部分组成：函数头和函数体。比如 int main(){...}，函数头是 int main()，函数体则是大括号{}中的内容，在函数体中可以编写程序以使函数实现一种操作或一种功能。

（2）函数的调用。

① 在一个函数内部可以调用其他函数。

② 在一个函数中调用另一个函数只需使用另一个函数的函数名即可，当程序执行到被调用

的函数时，程序会转去执行被调用函数的函数体的内容，当执行完被调用函数的代码后，再返回源程序执行被调用函数之后的语句。比如：

```
void f(){…}
void g()
{
    f();   /*当程序执行到这里时，会转去执行函数 f()中的代码，执行完函数 f()中的代码后再返回程序执行 f()之后的语句*/
}
```

（3）函数的返回类型、返回值和 return 0。

① 函数的返回值必须与返回的类型相匹配。也就是说，如果函数返回的类型是 int 整型，而却返回一个数组，这显然是不正确的做法。

② 关键字 return 后面的数据就是函数的返回值，比如示例中的 return 0;表示 main 函数返回一个整型值 0。

③ 当函数执行到 return 语句时就标志着此函数结束，而不会再执行 return 之后的语句。

（4）函数的形参。

函数的形参是向被调用函数传递信息的一种方法，比如 void f(){...}表示在调用函数 f 时不需要向它传递任何信息，也就是可以直接调用，如 f();表示调用函数 f；void g(int a){...}表示在调用函数 g 时必须给它传递一个整型数据，如 g(3);。若直接调用则会出错，g();就错误了。

8．main 函数讲解

（1）在 C++中，只能有唯一的一个 main 函数，它是程序与操作系统交互的接口，C++中一个完整的程序是从 main 函数开始执行的，同时 main 函数执行完毕，整个程序也就执行完毕。若没有 main 函数，程序将无法执行。比如 void f(){...} int main(){... return 0; }程序中有两个函数，在 C++中是从 main 函数开始执行的，而不会因为 f 函数写在 main 函数的前面而先执行它，当执行到 main 函数的 return 0 语句时程序结束执行。对于 f 函数，在程序中有可能执行，也有可能没有执行，若在 main 函数中直接或间接地调用了 f 函数，则会转去执行 f 函数；若没有调用 f 函数，那么它不会被执行。

（2）在 C++标准中，必须要求 main 函数返回 int 类型值，也就是 main 函数必须这样 int main(...){...}，对于 main 函数中的返回值，C++规定可以在 main 中不使用 return 语句，这时默认返回值为 0，比如 int main(){...}将返回值 0，它与 int main(){... return 0;}等同，但这一规定只适用于 main 函数，其他函数不适用。

9．分号与语句

（1）一条语句就是一条计算机指令，语句是 C++程序中的最小单位，函数的函数体一般由多条语句组成。

（2）在 C++中，一条语句是以分号作为结束的，当然复合语句后面不必加分号。

（3）注意，分号应在英文状态下输入，比如分号“;”是正确的，而分号“；”是错误的。

（4）复合语句：可以使用大括号{}，将多条语句写在其中，这样组成的语句就是复合语句。在大括号后不必加上分号，比如{int a; int b;}就是一条复合语句。

（5）在 C++中可以将一条语句放在一行或者多行上，即在 C++中可以使用空格或制表符的地方都可以使用回车。

（6）空白：空格、回车、制表符被统称为空白。空白的作用是用于将两个标记分开，这样我们才能很方便地阅读程序。比如 int main(){};，在标记 int 和 main 之间就有空白。

（7）标记：在 C++中把不可分割的元素称为标记。标记是不能被分开的，也就是在这些元素中间不能加空格、制表符或者回车，比如函数名、变量名、关键字等。

（8）要将标记分开写在多行上，则需要在本行的末尾加上“\”字符即可，在“\”字符后面不能有任何其他字符（包括注释），在下一行的开头也不能有任何其他字符（包括空白）。

10．输入/输出（I/O）和 cout<<"hy"<<endl;

（1）C++使用 cout 和<<运算符进行输出，使用 cin 和>>进行输入。

（2）<<和>>是一种运算符，C++将<<称为插入运算符，将>>称为提取运算符。

（3）cout 和 cin：cout 是一个对象，它知道如何显示数据，比如 cout<<"hy"；表示将字符串输出到 cout 指定的设备上（一般是显示屏），而 cin>>a 表示将指定设备上（一般指键盘）的数据输入到变量 a 中。一般情况下，cout 和<<是配合使用的，而 cin 和>>是配合使用的。

（4）可以使用 cout 和<<进行连续输出，比如 cout<<"hy"<<"kkk"；表示先输出字符串“hy”，再接着输出字符串“kkk”。同理，也可以使用 cin 和>>进行连续输入，比如 cin>>a>>b。

11．控制符和 endl

（1）endl 是 C++中的控制符之一，表示让输出的内容另起一行。使用 endl 后将使光标在屏幕上移到下一行，使用 cout 直接输出的内容是不会自动换行的。也就是说，cout<<"hy"<<"kkk"；输出的 hy 和 kkk 在同一行，而 cout<<"hy"<<endl<<"kkk"；输出的 hy 和 kkk 在不同的行上。

（2）endl 同样需要使用头文件 iostream，同时是 std 名称空间中的。

（3）当然，也可以使用“\n”进行换行，比如 cout<<"hy\n"<<"kkk"，其中的符号“\”是转义字符。

（4）C++将字符串放在双引号之内。也就是说，在双引号之内的才叫字符串，若没有双引号，则表示是自己命名的变量（当然变量名不能与关键字冲突）。

第 2 章

整型、字符型和浮点型专题

2.1 基础

1. 基本概念

（1）变量：就是可以变化的量，这和数学上的变量是一个意思，其值是可以改变的。当然每个变量都必须有自己的名字，就像数学中的自变量 x 一样，只不过在程序中可以为变量取一个不止一个字符的变量名。

（2）常量：就是不能改变的量，如整数 2，它就是整数 2，不能改变。

（3）常变量：使用关键字 const 把变量设置为值不可改变的常量，称为常变量。

（4）字面值：只能使用它本身的值来表示其名称的数被称为字面值，比如像 22 这样的整数值，只能使用 22 来称呼它，还有如浮点数 2.2、字母 a 等。注意，若 a 表示的是变量就不是字面值，因为 a 不是变量 a 的值，因此变量 a 不是使用它的值来表示其名称的。

（5）字面值常量：表示这个数既是字面值又是常量，如整数 3、浮点数 3.3、字符字面值 a 等。

（6）符号常量：是指使用#define 定义的常量，比如#define PI 3.14，当程序中使用到 PI 这一名字时就表示常量 3.14。

（7）常量、常变量、字面值、字面值常量、符号常量这几个概念没有明显的区分，很多时候都统称它们为常量。

（8）标识符：就是一个名字，可以用来标识变量、函数、数组等的名称。

（9）关键字：即 C++语言本身需要使用的名称，如 int、float、main、void 等。

（10）标识符应遵守 C++的命名规则（包括变量名、函数名）：

- 只能使用字母、数字和下画线（_）。
- 名称的第一个字符不能是数字，即标识符只能以字符和下画线开头，如 abc、_abc 是正确的，而 3abc 是错误的。

- 不能将 C++中的关键字作为变量的名称。
- 变量名的长度没有限制（但有一些系统会作限制）。
- C++是大小写敏感的语言，因此标识符 abc 和 Abc 是两个不同的标识符。
- 一个标识符中不应有空格、制表符或者回车，比如 abc def 是两个标识符，即 abc 和 def。
- 以下画线开始的名字是为编译器或系统所保留的，因为编译器生成的名字一般是以下画线开始的，所以不要使用以下画线开始的标识符，这样可以避免与编译器生成的名字相冲突。

2. 对数据类型的简单理解

（1）数据类型就是数据的类型，在数学中见过很多，比如数学有复数、整数、小数、分数之分，每一种其实就是不同的数据类型，当然 C++划分数据类型的方法与数学不同。

（2）不管是变量还是常量，都有自己的数据类型。

（3）C++为每种数据类型都提供了一个修饰符来标识，比如整型使用 int、浮点型使用 float 等，这和数学将复数使用 Z、自然数使用 N 来表示是一个道理。

（4）内置类型与用户自定义类型：像 int、float、double、char 等类型是 C++语言本身提供的类型，这些类型被称为内置类型，与之相对应的类（见后文）、枚举等就是用户自定义类型。

（5）C++中的数据类型分为基本类型和复合类型等，具体见表 2.1。

表 2.1　C++的数据类型

<table>
<tr><td rowspan="6">基本类型</td><td rowspan="4">整型</td><td>短整型 short int，简写为 short</td><td rowspan="3">有符号整型使用 signed，无符号整型使用 unsigned 关键字，比如 signed short int 表示有符号短整型，C++默认为 signed，即如果省略 signed 或者 unsigned 则表示有符号整型</td></tr>
<tr><td>整型 int</td></tr>
<tr><td>长整型 long int，简写为 long</td></tr>
<tr><td colspan="2">long long 和 unsigned long long（这是新标准，编译器不一定支持）</td></tr>
<tr><td colspan="3">字符型 char</td></tr>
<tr><td rowspan="3">浮点型</td><td colspan="2">单精度型 float</td></tr>
<tr><td colspan="2">双精度型 double</td></tr>
<tr><td colspan="2">长双精度型 long double</td></tr>
<tr><td colspan="3">布尔型 bool</td></tr>
<tr><td>复合类型</td><td colspan="3">枚举类型 enum、数组类型、结构体类型、struct 共用体类型、union 类、类型 class</td></tr>
<tr><td>其他类型</td><td colspan="3">指针类型、引用类型、空类型 void</td></tr>
</table>

2.2 整型和 sizeof 操作符

1. 基本概念

（1）本文所说的整型是指整数类型，有些书上将整数类型、字符型和布尔型统称为整型。

（2）整型表示的数据是一个没有小数的数字，比如 3、44、–33、0 等。

（3）C++有 3 种整型：int、short int 和 long int，即整型、短整型和长整型，这些类型的区别在于它们所能表示的最大值不同。

（4）C++中的整数类型可简写，如 long int 简写为 long，short int 简写为 short。

（5）整型还分为有符号整型和无符号整型，即正/负数的区别，使用 signed 表示有符号整型，使用 unsigned 表示无符号整型，比如 signed short int 或 unsigned long int 等。

（6）整型默认为有符号整型，即 signed 修饰符可省略，比如 long int 表示有符号长整型。

（7）有符号整型既可以表示正数，也可以表示负数，而无符号整型只能表示正数（包括 0）。

2. 整型数据的存储及数值范围

（1）计算机使用二进制形式来存储数据，计算机内的存储单元是位（bit），每一位都可以表示 1 位二进制数(即 0 或者 1)，比如 8 位存储单元，可以表示的二进制数的范围是 0000 0000~1111 1111，即十进制数 0~255。

（2）计算机的内存使用 8 位表示一个字节（Byte，简写 B。注意：是大写 B，小写 b 表示的是位，即 1B=8b），1GB=1024MB，1MB=1024KB，1KB=1024B。

（3）C++对字节的定义有一些不同，有可能是 16 位表示一个字节，有可能是 8 位表示一个字节，C++字节的多少是依据字符集的位数来确定的，本文使用标准的位数来表示字节，即 8 位表示一个字节。

（4）C++使用不同的内存量来存储整型数据，使用的内存量越大，能存储的值就越大。比如对于 int 类型数据，假如使用 16 位来存储，若是无符号整型，则能表示的值的范围是 0~65535（即 1111 1111 1111 1111）；若是有符号型，则能表示的值的范围是–32768~32767。对于有符号整型，最高位（就是第 1 位）表示符号位，0 表示正数，1 表示负数，因此除去符号位，有效位只有 15 位，其最大正值是 32767，即 0111 1111 1111 1111，而最大负值是–32768，即 1000 0000 0000 0000。从以上示例可以看到，无符号整型可以表示更大的整数，但不能表示负数。

（5）C++规定的各种整型的最小位数（注意：只规定了最小位数，因此各系统和编译器的长度可能会不一样）。

- short 至少 16 位。
- int 至少要与 short 一样长。
- long 至少 32 位，且至少要与 int 一样长。

- long long 至少 64 位，且至少要与 long 一样长。注意：long long 是新标准，编译器不一定支持。

3．sizeof 操作符

（1）sizeof 操作符的作用：返回一个对象或类型名的长度，返回值的类型为 size_t（一般被定义为整型），长度的单位是字节。sizeof 表达式的结果是一个编译时常量，可以用在任何使用常量的地方，比如用作函数的参数等。

（2）sizeof 有 3 种语法形式：sizeof(typename)、sizeof(expr)和 sizeof expr；其中，typename 表示类型名，expr 表示表达式。也就是说，使用 sizeof 不但可以求出类型的长度，还可以求出表达式的长度，比如 sizeof 3+3.0、sizeof a+b+c、sizeof(int)。

（3）当 sizeof 操作符用于内置类型名时必须要加括号，比如 sizeof(int)，而 sizeof int 是错误的。

（4）对 char 类型及其值执行 sizeof 操作将恒为 1，因为 char 占据 1 个字节。

（5）对引用类型执行 sizeof 操作，将返回存放该引用类型对象所需内存空间的大小。

（6）对指针执行 sizeof 操作，将返回存放指针所需内存空间的大小。要返回指针所指对象的大小，则必须对指针进行解引用操作。

（7）对数组名执行 sizeof 操作，将返回其元素类型的大小乘以数组元素的个数，即返回整个数组在内存中的字节长度，所以对数组名执行 sizeof 操作除以数组类型的 sizeof 操作，就可以求出数组的实际大小。比如对于数组 int a[11]，可以这样求数组的大小：sizeof(a)/sizeof(int)，其中 sizeof(a)返回的大小是 11×4=44，即整个数组所占的字节数，sizeof(int)返回的大小是 4。

（8）对于 string 类型的对象使用 sizeof 操作符时，长度是固定的，不因字符串的多少而不同。

（9）sizeof 操作符不能用在函数类型或不完整类型的表达式中，比如 sizeof(void)是错误的，因为 void 是不完整类型；void f(){}; sizeof f;或 sizeof(f)是错误的，因为标识符 f 是函数类型。

4．使用 climits 头文件判断数据类型的最大值（老版本 C++为 limits.h 文件）

（1）C++没对使用多少位来表示数据类型做出严格规定，而只规定了最小位数，因此各系统所使用的位数可能并不一致。比如 DOS 操作系统可能使用 16 位来存储 int 类型数据，而 Windows XP 则可能使用 32 位来存储 int 类型数据。

（2）climits 头文件中定义了很多符号常量（暂时理解为整型数据）对整型和字符型数据类型所能表示的最大值、最小值进行了限制，这些符号常量都可以使用 cout 进行直接输出。比如符号常量 INT_MAX 表示 int 类型所能表示的最大值，若使用 16 位表示 int 类型，则会输出 32767；若使用 32 位，则会输出 2147483647。

（3）要使用 climits 中的符号常量，应使用#include<climits>语句，然后就可以直接使用此文

件中定义的符号常量了。

（4）判断当前 C++系统使用的字节是多少位的方法。

- 方法一：使用 UCHAR_MAX 或者 SCHAR_MAX 等符号常量，根据其所表示的最大值或最小值来判断位数。
- 方法二：对 char 类型使用 sizeof 返回的值恒为 1，即 sizeof(char)返回值 1，说明 char 类型占据 1 个字节，因此只需直接 cout<<CHAR_BIT 就能知道系统中 1 个字节到底是 8 位还是 16 位。CHAR_BIT 是 climits 中定义的符号常量，表示 char 类型所占据的位数。

（5）符号常量就是使用#define 定义的常量（有关#define 的内容，请参见预处理器章节）。climits 头文件中定义的符号常量如表 2.2 所示。

表 2.2　climits 头文件中定义的符号常量

符号常量名	说　明	符号常量名	说　明
SHRT_MAX	short 短整型的最大值	ULONG_MAX	unsigned long 的最大值
SHRT_MIX	short 短整型的最小值	CHAR_BIT	char 的位数
USHRT_MAX	unsigned short 的最大值	CHAR_MAX	char 的最大值
INT_MAX	int 的最大值	CHAR_MIN	char 的最小值
INT_MIN	int 的最小值	SCHAR_MAX	signed char 的最大值
UINT_MAX	unsigned int 的最大值	SCHAR_MIN	signed char 的最小值
LONG_MAX	long int 的最大值	UCHAR_MAX	unsigned char 的最大值
LONG_MIN	long int 的最小值		

在 Windows XP 操作系统下 Visual Studio 2010 编译器所得各整型的取值范围及长度如表 2.3 所示。

表 2.3　在 Windows XP 操作系统下 Visual Studio 2010 编译器所得各整型的取值范围及长度

类　型	长度（位）	取值范围	类　型	取值范围
short int	16	-32768~32767	unsigned long	0~4294967295（32 位）
unsigned short	16	0~65535	CHAR_BIT	8（表示 1 个字节为 8 位）
int	32	-2147483648～2147483647	char	-128~127（默认为 signed char）
unsigned int	32	0~4294967295	singed char	-128~127
long int	32	-2147483648~2147483647	unsigned char	0~255

5. 整型常量

（1）整型常量就是整数，比如 3、33、333 等。

（2）整型常量可以使用八进制、十进制、十六进制表示。注意：不能使用二进制表示。

（3）整型数默认为十进制，使用 0 开始书写的数字是八进制，比如 045、033 等；使用 0x 开始书写的数字是十六进制，比如 0x33、0xaf、0xA8 等，对于十六进制形式的 a~f 6 个字母，既可大写也可小写。

（4）注意：八进制表示的数最大值为 7，若超过了 7 则会出错，比如 099 就是错误的，系统不会认为这个数是十进制形式的 99。

（5）在整型常量后面使用后缀可以强制表示某一类型。

- 字母 l 或 L 表示整型常量为 long，比如 33L。
- 字母 u 或 U 表示整型常量为 unsigned int，比如 33U。
- 字母 ul、UL、LU 或 lu（字母大小和顺序无关紧要）表示 unsigned long，比如 33UL、33LU、33uL、33Lu 都表示将整型常量 33 存储为 unsigned long 类型。
- 注意：没有有关 short 类型的后缀。

6. 整型常量超出默认类型时的自动扩展规则

注：以下规则可使用 cout<<typeid(数值).name()<<endl;进行测试。

（1）C++默认的整型常量类型为 int，但如果数值超过了 int 类型所能表示的范围，则 C++会将整型数值存储为 long 类型，若 long 类型还无法表示该数值，则会存储为 unsigned long 类型。

（2）对于八进制或十六进制而言，默认也为 int 类型，若数值过大，则会被转换为 unsigned int、long 类型或 unsigned long 类型。因为八进制和十六进制经常用来表示存储内存的地址，而地址不可能为负，因此这里才会存储为 unsigned 整型。

（3）若整型常量有后缀 u 或 U，则默认为 unsigned int 类型或 unsigned long 类型。

（4）若整型常量有后缀 l 或 L，则类型 long、unsigned long 哪个可最先表示此常量，那么就使用哪个类型。

（5）若整型常量有后缀 ul、uL、Ul、UL、LU、Lu、lU、lu，则类型为 unsigned long。

2.3　char（字符型）

1. 基础

（1）字符，就是我们平时看到的单个英文字母，包括大小写。C++中的字符应使用单引号括起来，比如'a'、'b'、'A'，数字也是字符，比如'1'、'2'等。

（2）字符型使用标识符 char 来表示，比如 char a;表示变量 a 是字符型，可以将一个字符值赋给变量 a。

（3）char 占据的长度始终是 1 个字节（一般都是 8 位。注意：C++标准没有对 1 个字节是多少位作出规定）。

（4）char 占据的位数可以使用 climits 头文件中的符号常量 CHAR_BIT 或根据其最大值 UCHAR_MAX 来判断。

2. 字符字面值

（1）C++中单个字符字面值使用单引号括起来，比如字符 a 应书写为'a'；若直接写成 a，则表示 a 是变量名。

（2）字符串是用双引号括起来的，字符串由 0 个或多个字符组合成，这里要加以区别，比如"a"表示字符串 a，而'a'表示字符 a。

（3）单引号中只能有一个字符，超过 1 个字符 C++未作规定，因此不建议使用。超过 1 个字符应使用双引号，表示这是一个字符串。比如'ab'应写成"ab"。VC++ 2010 在单引号中使用多个字符不会出错。

3. signed char 和 unsigned char

（1）在 C++中可以将 char 类型当作整型来使用。也就是说，将一个整数赋值给 char 类型的变量，同样可以表示一个字符；比如 char c=97; cout<<c<<endl;会输出字符 a（数值 97 是 ASCII 字符集中的字符 a 的编码）。同理，也可以将 char 类型的值赋给 int 类型的变量。比如 int c='a'; cout<<c<<endl;会输出 97，因为字符 a 的 ASCII 编码为 97。

（2）在默认情况下，C++没有规定 char 的符号性质。也就是说，在默认情况下，char 既不是没有符号的，也不是有符号的，char 的符号由实现来决定，一般编译器都会把 char 当作 signed 来处理。

（3）可以使用 unsigned 和 signed 来明确指定 char 是否有符号，比如 unsigned char。

（4）因为 char 类型占据 1 个字节（一般为 8 位），所以有符号的 char 表示的值的范围是 -128~127，无符号的 char 表示的值的范围是 0~255。

（5）判断编译器中 char 类型是否有符号的方法如下。

- 方法一：CHAR_MIN（表示 char 类型的最小值）的值应与 SCHAR_MIN 的值相同，CHAR_MAX 的值应与 SCHAR_MAX 的值相同；若 char 是无符号类型，则 CHAR_MIN 的值应为 0，而 CHAR_MAX 的值应与 UCHAR_MAX 的值相同。
- 方法二：假设一个字节为 8 位，首先对 char 变量赋一个大于 128 小于 255 的数，然后将此变量赋给一个整型变量，再输出这个整型变量，若输出的值为原值，则表示是无符号型 char；若不是原值（有符号型会溢出，因此不可能输出原值），则表示是有符号

型 char。比如 char c=222; int a=c; cout<<a<<endl;，若输出的 a 是原值 222，则表示 char 是无符号型的；若输出的不是原值，则表示 char 是有符号型的。注意：在有些编译器中 char c=222;可能会出错（数据溢出），这时就表示 char 是有符号型的。

（6）注意：因为 char 可以当作整型来使用，所以在 C++ 2.0 版本以前使用 cout 不能直接输出字符，比如 cout<<'a';会输出整数值 97。而现在使用的 C++基本上都是 2.0 以后版本，因此使用 cout 能直接输出字符。

4．字符的编码

（1）在计算机中存储数字是很容易的，但是计算机不能直接存储字符，必须将字符转换为计算机能识别的二进制数字才能进行存储。

（2）怎样将字符转换为二进制形式存在一个编码的问题，一般都要进行两次编码。

（3）字符集：字符的第一次编码，是将字符编码为与一个数值（如一个整数）相对应，比如将字符 A 编码为十进制整数 65，B 编码为 66 等。将每一个字符都编码为与一个数值对应，这样就组成了一个字符集，比如常用的 ASCII 字符集，就是将 128 个常用的字符编码为 128 个整数。除此之外，还有 Unicode 字符集、GB2312 字符集等。

（4）编码字符集：字符的第二次编码，就是把第一次编码好的整数值再编码为相应的二进制形式，这样计算机就可以直接进行存储了。比如对于 Unicode 字符集，就有 3 种不同的方式进行编码，分别是 UTF-8、UTF-16 和 UTF-32，它们分别使用变长位、16 位、32 位来表示 Unicode 字符集中编码好的数值（每个数值都与一个字符相对应）。

（5）字符集中字符所对应的数值是唯一的，但其在计算机中所表示的二进制形式却不一定相同，因此使用字符集中编码好的数值能唯一确定一个字符。

（6）C++系统使用的字符集是根据机器而定的，在大多数机器上使用的都是 ASCII 字符集（本文将对使用 ASCII 字符集进行讲解）。

（7）为了满足国际需要，C++支持宽字符类型 wchar_t，宽字符类型可以存储更多的值，比如 Unicode 字符集。本文不会具体介绍宽字符类型。

5．多字符和宽字符简介

（1）对于像 ASCII 码这样的字符集，在将其编码为计算机所存储的格式时（即第二次编码），只使用 1 个字节（8 位）来存储字符就足够了，但对于汉字来讲，使用 1 个字节来存储显然是不行的。一般汉字使用 2 个字节来存储这个整数值，即汉字“中”被存储为 1101 0110 1101 0000（0xD6D0）。

（2）多字节字符和宽字符：多字节字符指的是存储长度（即第二次编码时二进制位的长度）为 1 至多个字节的字符；如中国汉字的 GB2312 字符集编码，它就是多字节字符，但是它并不是宽字符。而宽字符一般指的是 Unicode 编码的字符，目前用得较多的是使用 UTF-8 存储的

Unicode 字符集，宽字符一般占据 2 个字节的长度。注意：Unicode 字符集也是使用 1 至多个字节来存储其字符的，因此 Unicode 字符也是一种多字节字符。Unicode 字符集编码的字符范围很广，其中也有对汉字的编码。

（3）将多字节字符转换为宽字符，即把 GB2312 中的字符编码转换为 Unicode 中的字符编码。

6. 宽字符类型在 C++中的表示

（1）C++使用 wchar_t 来表示宽字符类型，比如 wchar_t c=L'a';。

（2）宽字符在 C++中的表示，就是在字符的前面加上大写字母 L（中间不能有空格），比如 L'a';就表示字符 a 是宽字符；再比如 L'\x65';等。

（3）宽字符应使用 wcout 进行输出，比如 wcout<<L'a'<<endl;。

7. 转义字符

（1）很多字符无法从键盘直接输入，比如回车符、退格符等，还有一些字符在 C++中有特殊的意义，也不能被直接显示，比如双引号、单引号等，这些字符就需要使用转义字符的形式来表示。

（2）转义字符以反斜杠符号“\”开始，后面紧接数字或字符，同时需要使用单引号括起来，因为其表示的是一个字符，比如'\n'表示回车换行，'\b'表示退格，'\'表示单引号，'\\'表示符号“\”。

（3）注意：转义字符虽然是由几部分组成的，但它表示的是一个字符。比如'\n'表示一个字符，而不是两个字符'\'和'n'的组合。

（4）转义字符通用格式。

- 八进制格式：\ooo，其中 ooo 表示 3 个八进制数，可以是 1~3 个八进制数。系统默认使用八进制数，书写时不必书写前缀符数字 0
- 十六进制格式：\xhh 或\xh，其中 x 表示使用十六进制数，十六进制的位数没有限制，最好使用 1 位或 2 位。在使用前缀符表示十六进制数时，最前面的数字 0 不必书写，也就是说，\0xhh 是不正确的书写方法。
- 注意：无法在反斜杠“\”后面使用十进制数。
- 八进制和十六进制格式的转义字符以第一个不是八进制或十六进制数字的字符结束。比如 cout<<"\70f";将输出 8 和 f 两个字符，因为字符 f 已不是八进制数字，转义字符到此结束。其中八进制的 70 转换为十进制后为 56，对应的 ASCII 码表示的就是数值 8。
- 可以将任意字符都表示为转义字符的通用格式。比如'\x61'表示字符 a，也可以写成八进制形式，即'\141'，十六进制的 61 与 ASCII 表中的小写字符 a 相对应。

（5）通用字符名：形式为\uhhhh，其后是相应的十六进制数，数必须是 4 位，使用“\u”表示的是 ISO 10646 编码的字符。比如\u00E2、\u00F6 等，若系统不支持 ISO 10646 编码，则无法显示这些字符。

C++定义的转义字符如表 2.4 所示。

表 2.4　C++定义的转义字符

字符形式	含　义	十进制 ASCII 码	对应的 ASCII 符号
\n	换行，将当前位置移到下一行的开头	10	NL(LF)
\r	回车，将当前位置移到本行开头	13	CR
\t	水平制表符	9	HT
\v	垂直制表符	11	VT
\a	响铃	7	BEL
\b	退格	8	BS
\f	换页，将当前位置移至下页开头	12	
\\	反斜杠符“\”	92	\
\’	单引号	39	‘
\”	双引号	34	“
\0	空字符	0	

8．转义字符的限制

（1）以八进制或十六进制格式表示的转义字符，其取值范围不能超过编译器默认的 r 类型。假设 1 个字节为 8 位，则 signed char 的取值范围是–128~127，unsigned char 的取值范围是 0~255。

（2）不管是有符号 char 还是无符号 char，在使用八进制或十六进制格式的转义制符时，其取值范围都不能超过十进制数 255（假设 1 个字节为 8 位），即八进制的 377，十六进制的 0xFF，否则会出错。

（3）不管是有符号 char 还是无符号 char，在使用八进制或十六进制格式的转义字符时，都不能使用负数，比如'\–22'将会被解释为 3 个字符，即'–'、'2'、'2'。

（4）在使用八进制格式的转义字符时，最多只使用 3 位来表示转义字符，超过 3 位八进制数时就表示转义字符结束，若前 3 位八进制数未超过十进制的 255，但八进制的位数超过 3 位时，就相当于单引号中有多个字符的情况，在某些编译器上可能会出错。但 VC++ 2005 不会出错，它会把其解释为单引号中有多个字符的情况，比如以八进制格式表示的'\14133'，因为'\141'在 ASCII 码中表示的是字符 a，所以'\14133'与'a33'是相同的。

（5）使用十六进制格式的转义字符时，位数是没有限制的，比如'\x000033'仍然只表示一个字符，且 sizeof('\x000033')的值为 1。再比如 cout<<"\x00001112"将不能输出，因为表示的字符太大，字符串"\x00001112"不会被解释为'\x00'、'0'、'1'等多个字符，而是作为一个字符处理。

（6）使用少于 3 位的八进制格式和少于 2 位的十六进制格式的转义字符是允许的，比如'\12'、'\f'等是正确的。

9. 转义字符的书写

（1）以八进制格式使用转义字符时应省略前缀符数字 0，系统默认是八进制格式的，若出现前缀 0，则系统会忽略这个数字 0，但位数不会忽略。比如'\0141'并不能输出字符 a，它相当于单引号中有两个字符，即'\014'和'1'。

（2）以十六进制格式使用转义字符时也应省略前缀符数字 0，但前缀 x 不能省略，若出现前缀 0，则系统不会忽略这个数字 0。比如'\0x61'，有些编译器会出错，VC++2005 会把它当作 4 个字符来处理，即'\0'、'x'、'6'、'1'。注意，空字符的值为 0。可以使用字符串的形式来验证，比如 cout<<"\0x61"<<endl;只会输出一个空行，因为字符串遇到空字符'\0'就结束输出，后面的字符是输出不了的。

（3）若“\u”之后的十六进制数不足 4 位，则会把 u 和后面的数字当作多个字符来处理。比如\u3、\u33 或\u333，因为“\u”后的十六进制位数不足 4 位，将不会使用通用字符名，若它们在字符串中出现，则会被分别当作 2、3、4 个字符来处理。比如"\u3"将会输出两个字符'u'和'3'；同理，"\u33"将输出 3 个字符'u'、'3'、'3'。若使用的是单引号，比如'\u33'则相当于使用单引号括起了 3 个字符，即与'u33'相同，反斜杠被忽略了。

（4）若单引号中只有一个反斜杠，而反斜杠之后什么都没有，则是错误的，比如'\'。

（5）若反斜杠之后的字符没有在 C++中作相应的定义，则程序会忽略反斜杠，而使用反斜杠后面的字符，这时同样只表示一个字符。比如'\g'，因为在 C++中没有作相应的定义，所以'\g'就和'g'是相同的，反斜杠“\”会被忽略。

2.4 bool（布尔型）

（1）布尔型在 C++中使用关键字 bool 表示。

（2）布尔型变量占用 1 个字节（一般为 8 位）的长度。

（3）布尔型只有 true（真）或 false（假）两个值， true 和 false 是关键字，比如 bool a=true。

（4）在 C++中所有的非零值都被解释为 true，只有值 0 才被解释为 false。

（5）true 和 false 都可以被提升转换为 int 类型，其中 true 被转换为 1，而 false 被转换为 0。

（6）在默认情况下，使用 cout 输出 bool 类型值时，若值为 true 则输出 1，若值为 false 则输出整数 0。

（7）对 bool 类型变量输出时的格式控制，详见 I/O 章节。

2.5　浮点型

1．基础

（1）浮点型表示的数据是带有小数点的，比如 3.2、5.11、4.0、0.23 等。

（2）浮点型有 3 种，即：float（单精度）、double（双精度）和 long double。

（3）浮点数可以使用小数和科学计数法（指数表示法）表示，小数表示比如 3.3 等。

（4）浮点数的指数表示法：指数表示法分为 3 部分，即：符号位+尾数+指数，比如-2.3×10^3，其中前面的负号表示符号位（即表示此数是正数还是负数），2.3 表示尾数，10^3 中的 3 表示浮点数的指数，10 表示基数，若是十进制则以 10 为基数，二进制则以 2 为基数，八进制则以 8 为基数。

2．C++中浮点数的书写

（1）在 C++中，指数表示法使用大写字母 E 或小写字母 e 来表示，在数字中间不能有空白。比如 2.3E5 表示 2.3×10^5，$-2.3E^{-5}$ 表示-2.3×10^{-5}；而 3.4 E5 是错误的，在 3.4 和 E 之间不能有空格。

（2）在 C++中不能以八进制、十六进制或其他进制形式来书写浮点数。比如以八进制形式来表示浮点数 02.2E2，结果将是十进制的 2.2E2；同理，0x2E2 则会将字母 E 解释为十六进制的 E，即十进制的 14，再比如 0x2.2E3 是错误的表示方法；同理，02.22 的结果将是十进制数，而 0x2.22 是错误的。

（3）在 C++中书写浮点数时必须有小数点（用指数表示法书写时可以省略小数点），小数点两边的数可以省略其中一个，但不能两个都省略。比如 2.、2.e2、22、22E2 或.0E2 是正确的，而.或.e2 是错误的。

3．浮点数在计算机中的存储

（1）所有的浮点数在计算机中都是以指数表示法存储的，不管是以小数形式还是指数表示法书写的，在计算机中都会被转换为二进制形式的指数表示法，其基数为 2。

（2）C++规定 float 至少 32 位（二进制的位数），double 至少 48 位且不少于 float，long double 至少要和 double 一样长。通常 float 为 32 位，double 为 64 位，long double 为 80、96 或 128 位。3 种类型的指数范围至少是–37~37，因此在不同的编译器和操作系统上可能会有不同的数值。

（3）计算机中存储的浮点数被分为两部分存储，即：尾数部分和指数部分，当然这些数都是二进制形式。

4．浮点型常量

（1）C++默认浮点型常量的类型是 double。

（2）浮点型常量必须有小数点（指数表示法除外），若没有小数点，系统就会把这个数当作整数处理。比如 cout<<1;中的 1 是整数，cout<<1.0;中的 1.0 则是浮点数。

（3）浮点型常量也可以用指数形式表示，比如 1.0E2、2.2e2、2E2 等。

（4）浮点型常量小数点两边的数可以省略其中一个，但不能两个都省略，比如 1.、.1 等是正确的，而.E2 则是错误的。

（5）使用后缀可以指定浮点型常量的类型，其中 F 或 f 表示 float 浮点型常量，L 或 l 表示 long double 浮点型常量，对于 double 浮点型常量不需要指定后缀。比如 1.0f、1.1E2F 等。

（6）浮点型常量的后缀 F 或 L 不能用在整数的后面，比如 2F、3F 等是错误的表示，而 3L、4L 等会被编译器解释为 long int 类型，而不会被解释为 long double 类型。

5. 以指定的格式输出浮点数

详见 I/O 章节。

6. 浮点数的精度

注： *若不能理解下面的内容，请参阅本书下载资源中的附录 A。*

（1）IEEE 754 标准规定，float 浮点型的总长度是 32 位，double 浮点型的总长度是 64 位。

（2）精度的概念：精度指的是最多允许有多少位十进制有效数字能保证转换为二进制数后，再准确地还原为以前的十进制数。这种转换也被称为 D-B-D 转换，即十进制—二进制—十进制转换。其中有效数字的位数包括整数和小数部分，比如 3333.33 是 6 位而不是 2 位。

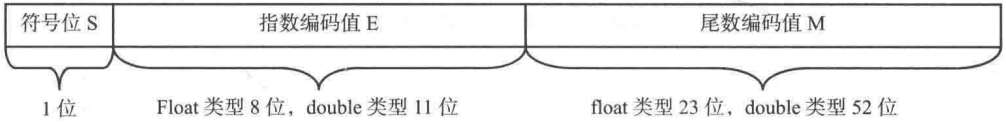
（3）float 类型的精度是 7 位，double 类型的精度是 15 位，long double 类型的精度是 18 位。

（4）float 类型的精度为 7，并不代表所有 8 位十进制数都会产生溢出，而是能保证所有精度为 7 的数都不会产生尾数溢出。比如小于 16777215 的 8 位整数，以整数 10 开始的小数 10.xxx 总共可以表示 8 位数，但它们都不会溢出。关于为什么不会溢出，详见 IEEE 754 标准。

（5）注意：并不是所有的十进制小数都能经过有限次转换变成二进制小数。这种情况产生的溢出与我们要讨论的精度无关，这种精度损失属于转换误差。比如十进制数 0.1 就不能经过有限次转换变成二进制数。

（6）浮点数的表示形式为：$\pm d_0.d_1d_2\cdots d_{p-1}\times R^e$，其中 $d_0.d_1d_2\cdots d_{p-1}$ 表示尾数，有效位数为 p，e 表示浮点数的指数，R 表示浮点数的基数，对于十进制数则为 10，二进制数则为 2，八进制数则为 8，计算机系统使用的都是二进制形式，因此其基数 R 为 2。

（7）IEEE 754 标准规定单精度浮点数总共使用 32 位来存储，其中用 1 位来表示浮点数的符号，用 8 位表示指数，用 23 位表示尾数；双精度浮点数则使用 64 位来存储，符号位 1 位，指数位 11 位，尾数位 52 位，其格式如下：



（9）符号位 S：1 表示负数，0 表示正数。

（10）浮点数的精度公式：精度 $n<(p-1)\times\lg b$；其中 p 表示将十进制数转换为以 b 为基数的浮点数的位数。比如对于 float 类型的二进制数，其 p 应为 24，而 b 应为 2，因此其精度就为 $23\times0.3=6.9$，取下限应为 6，但大多数教材都是取 7，所以以 7 为准。

7．浮点数的分类

详见本书下载资源中的附录 A。

（1）IEEE 754 标准把浮点数分为 5 类：规约浮点数、非规约浮点数、0、无穷大、NaN（非法浮点数）。

（2）规约浮点数：是指指数编码值 E 为 $0<E<2^n-1$，尾数编码值 M 是非 0 的浮点数。

（3）非规约浮点数：是指指数编码值为 0，尾数编码值 M 为非 0 值的浮点数。

（4）若指数编码值 E 全为 0，且尾数编码值也为 0，则这个数是 0（有正负 0 之分，与符号位 S 有关）。

（5）若指数编码值 E 为 2^n-1（即全为 1），且尾数编码值为 0，则这个数是无穷大（有正负无穷大之分，与符号位 S 有关）。

（6）若指数编码值 E 为 2^n-1（即全为 1），且尾数编码值为非 0 值，则这个数是 NaN，即不是一个数。这个数是超过规约浮点数最大值的值，这种类型的值有很多，只要是比 $2^{128}\approx3.4\times10^{38}$ 更大的数，就都是 NaN。

8．浮点数的取值范围

浮点数的最大值、最小值的计算如表 2.5 所示。

表 2.5　浮点数的最大值、最小值的计算（IEEE 754 标准）

	指数位数	尾数位数	符号位位数	总长度	正负值取值范围
float 类型最大规约数	8	23 位	1	32	$\pm(2-2^{-23})\times2^{127}\approx\pm3.4\times10^{38}$
float 类型最小规约数	8	23 位	1	32	$\pm1\times2^{-126}\approx\pm1.18\times10^{-38}$
float 类型最大非规约数	8	23 位	1	32	$\pm(1-2^{-23})\times2^{-126}\approx\pm1.18\times10^{-38}$
float 类型最小非规约数	8	23 位	1	32	$\pm2^{-23}\times2^{-126}=2^{-149}\approx\pm1.4\times10^{-45}$
double 类型最大规约数	11	52 位	1	64	$\pm(2-2^{-52})\times2^{1023}\approx\pm1.8\times10^{308}$
double 类型最小规约数	11	52 位	1	64	$\pm1\times2^{-1022}\approx\pm2.2\times10^{-308}$

续表

	指数位数	尾数位数	符号位位数	总长度	正负值取值范围
double 类型最大非规约数	11	52 位	1	64	$\pm(1-2^{-52})\times2^{-1022}\approx\pm2.2\times10^{-308}$
double 类型最小非规约数	11	52 位	1	64	$\pm2^{-52}\times2^{-1022}=2^{-1074}\approx\pm4.9\times10^{-324}$
补充：当指数编码值和尾数编码值均为 0 时就表示浮点数 0。					

C++浮点数的取值范围如表 2.6 所示。

表 2.6 C++浮点数的取值范围

	最大正数范围	最大负数范围
float 类型规约数	$1.18\times10^{-38}\sim3.4\times10^{38}$	$-3.4\times10^{38}\sim-1.18\times10^{-38}$
float 类型非规约数	$1.4\times10^{-45}\sim1.18\times10^{-38}$	$-1.18\times10^{-38}\sim-1.4\times10^{-45}$
double 类型规约数	$2.2\times10^{-308}\sim1.8\times10^{308}$	$-1.8\times10^{308}\sim-2.2\times10^{-308}$
double 类型非规约数	$4.9\times10^{-324}\sim2.2\times10^{-308}$	$-2.2\times10^{-308}\sim-4.9\times10^{-324}$

从上表可以得出以下结论。

（1）规约浮点数与非规约浮点数的区别在于它们与 0 的接近程度，使用非规约浮点数的作用是避免下溢而损失精度，当比规约浮点数更小时，则使用非规约浮点数来表示，这样损失的精度比直接下溢为 0 更小；而未使用非规约浮点数时，则凡是比规约数更小的数都被直接处理为 0，这样损失的精度比较大。因此，IEEE 754 标准允许使用比最小规约数还要小的非规约数来表示更接近于 0 的数。

（2）使用浮点数可以表示 0，但并不能表示 $0\sim1.18\times10^{-38}$ 之间的数（对于 float 类型而言），即使使用非规约数，最小也只能表示到与 0 更接近的数 1.4×10^{-45}。所以 float 类型并不能表示 $-3.4\times10^{38}\sim3.4\times10^{38}$ 之间的所有数。对于 double 类型是同样的道理。

（3）现在很多计算机都支持 IEEE 754 标准，C++也不例外，因此使用编译器不能输出比非规约数还要小的数。比如对于 float 类型，不能输出比 1.4×10^{-45} 还要小的数。也就是说，比 1.4×10^{-45} 还要小的数要么是 0，要么是 1.4×10^{-45} 本身，因此比 1.4×10^{-45} 小，但差距不大的数会当作 1.4×10^{-45} 处理（主要是舍入的误差引起的），但差距比较大的数会当作 0 处理。比如 1.2E–45 比 1.4×10^{-45} 小，但差距不大，会被当作 1.4×10^{-45} 处理；而 2E–47 因差距较大会被当作 0 处理；同理，比 -1.4×10^{-45} 更大的数也会做如此处理。比如 float a=1.3E–46; cout<<a<<endl;将输出 0。

9. C++中的 cfloat 头文件（C++老版本是 float.h）

（1）头文件 cfloat 中定义了一些符号常量，可以通过这些符号常量看到各编译器关于浮点数各标准的取值，其中各符号常量的取值和含义如表 2.7 所示。

表 2.7　cfloat 头文件中各符号常量的取值和含义

注意：C++标准的指数值比 IEEE 754 标准规定的数要大 1，因此对于 float 类型指数取值−125~128，对于 double 类型指数取值−1021~1024。			
符号常量	IEEE 754 标准取值	VC++ 2010 取值	含　义
FLT_ROUNDS	−1，0，1，2，3	1（就近舍入）	系统默认的舍入方式，−1 代表不确定，0 表示向 0 舍入，1 表示就近舍入，2 表示向正无穷大舍入，3 表示向负无穷大舍入
FLT_RADIX	2，10，16	2	浮点数的基数
FLT_MANT_DIG	24	24	float 类型的二进制有效数字位数
DBL_MANT_DIG	53	53	double 类型的二进制有效数字位数
FLT_DIG	6（有的书中为 7 位）	6	float 类型的精度，即准确地将十进制转换为二进制再转换为十进制（D-B-D）时的十进制数的最大有效位数（包括整数和小数部分）
DBL_DIG	15	15	double 类型的精度，即准确地将十进制转换为二进制再转换为十进制（D-B-D）时的十进制数的最大有效位数（包括整数和小数部分）
FLT_MIN_EXP	−126	−125	float 类型二进制浮点数的最小指数
FLT_MAX_EXP	127	128	float 类型二进制浮点数的最大指数
DBL_MIN_EXP	−1022	−1021	double 类型二进制浮点数的最小指数
DBL_MAX_EXP	1023	1024	double 类型二进制浮点数的最大指数
FLT_MIN_10_EXP	−38	−37	float 类型十进制浮点数的最小指数
FLT_MAX_10_EXP	38	38	float 类型十进制浮点数的最大指数
DBL_MIN_10_EXP	−308	−307	double 类型十进制浮点数的最小指数
DBL_MAX_10_EXP	308	308	double 类型十进制浮点数的最大指数
FLT_MIN	1.18×10^{-38}	1.18×10^{-38}	float 类型的最小规约正数
FLT_MAX	3.4×10^{38}	3.4×10^{38}	float 类型的最大规约正数
DBL_MIN	2.2×10^{-308}	2.2×10^{-308}	double 类型的最小规约正数
DBL_MAX	1.8×10^{308}	1.8×10^{308}	double 类型的最大规约正数

（2）注意：以上值是在操作系统为 Windows XP、编译器为 VC++2010、x86 机器上得到的，不同系统、不同编译器、不同机器可能会得到不同的值。

2.6 符号常量和#define 预处理指令简介

（1）符号常量是使用预处理器指令#define 定义的。

（2）预处理器：预处理器是编译器把 C++代码编译为机器指令之前执行的一个过程，所有的预处理器都以“#”开头，以便与 C++语句区分开。

（3）#define 指令格式：#define 标识符 字符序列。比如#define N 3。

① 表示以后凡是使用到“标识符”的地方都会被后面的“字符序列”替换。注意，该语句不以分号结束。

② 编译器不会对“字符序列”进行类型检查。也就是说，字符序列可以是任意字符。

③ 语句中的字符序列可以是任意字符，而不仅仅是数字。比如#define PI HYONG，在使用 PI 时就会用 HYONG 来替换，替换之后 HYONG 可能会是一个未定义的标识符。

④ 示例：#define N 3

- 表示以后使用到符号 N 时都会被替换为 3，比如 int a[N];定义大小为 3 的数组。
- 虽然 N 看起来和变量一样，但它们没有任何关系，N 只是一个符号或标志，在程序代码编译前该符号会用一组指定的字符来替换。
- 可以看到，N 被 3 替换后，相当于拥有常量的性质，但在 C++中最好使用 const 来声明常量，比如 const long double PI=3.1416;。

第 3 章

声明、定义、复杂声明和 typedef 专题

3.1 声明与定义

（1）标识符：标识符就是一个名字，使用标识符主要是要与 C++中的关键字相区别，本文所讲的名字和标识符都是指标识符。

（2）变量、类型：请参考“整型、字符型和浮点型专题”相关内容。

1. 变量、对象、实体

（1）注意：本文中的对象与类的对象是两个概念，应区别对待。

（2）对象：指的是某种类型所占据的一块连续的内存单元。注意：对象不仅是指一片连续的内存单元，而且这块内存区域已经指定了某种类型。

（3）变量：其实就是命名后的对象。也就是说，变量是为指定类型的连续的内存单元（即对象）取的一个名字。若使用内存的地址编号来访问一块连续的内存单元，这样的程序很难让人理解，因此就有必要为一块连续的保存特定类型的内存单元（即对象）取一个名字，这个名字就是变量。

（4）实体：本文中所说的实体就是变量的另一个名字。一般情况下，变量指的是像整型、浮点型等这些类型的对象，按照变量的概念，变量还可以是函数名、指针名、数组名等、为了描述方便，有时会使用“实体”一词。

（5）从以上概念可以看出，变量、对象和实体三者没有本质的区别。

2. 类型

（1）类型与内存。

内存中比特值的含义完全取决于这块内存单元所表示的类型，保存在计算机中的值是一些二进制比特，计算机并不知道它们有什么意义，只有当我们决定如何解释这些比特值时才有意义。比如 65 和字符'a'在内存中的比特值是相同的，若将比特值解释为 int 类型，那么它是一个十进制数；若解释为 char 类型，则是字符 a。因此内存单元中的数据应具有一个类型，当类型确定后就能对其中的数据做出正确的解释了。

（2）类型的作用。

- 类型决定了可以把什么数据赋给对象（比如整数 3 可以赋给 int 类型的变量等）。
- 类型决定了可以对该对象进行什么样的操作（比如可以对 int 类型的变量或常量进行加、减、乘、除等操作，不能对指针变量进行乘、除等操作）。
- 类型还决定了对象的内存大小、布局和取值范围。

（3）每个名字（或变量，变量就是命名后的对象，因此一个变量就是一个名字）、表达式都应有一个类型，这个类型决定了可以对这个名字进行什么样的操作，因此类型决定了这个名字或表达式的使用方式。

（4）不能给变量赋予一个类型错误的值。编译器会记录每个变量的类型，并确认对它进行的操作是否与类型相一致。

3．声明的语法形式

声明的语法形式之一：[[存储类区分符] | [类型限定词]]<类型区分符>声明符[, 声明符[, 声明符[, ...]]];

声明的语法形式之二：声明区分符 声明符[, 声明符[, 声明符[, ...]]];

比如 int a, b;，表示声明两个 int 类型的变量 a 和 b，其中 int 是类型区分符，a 和 b 都是声明符。

说明：

（1）方括号“[]”中的内容是可选项，以短竖线“|”隔开的内容可以只选一项，尖括号“<>”中的内容是必选项。

（2）存储类区分符有 extern、static、auto、register、typedef。其中 extern 表示外部的，static 表示静态的，auto 表示自动的，register 表示寄存器。关于存储类区分符请参阅函数章节内容。

（3）注意：typedef 是声明语法中存储类区分符的一种，因此 typedef 语句是一个简单的声明，可以按照声明的原理来解释 typedef。

（4）类型限定词有两个：const 和 volatile。其中 const 一般被理解为只读的，而 volatile 是易变的。

（5）类型区分符有基本类型（如 int、float、double、unsigned long int 等）、void 类型、枚举型、结构或联合类型、用户自定义类型。注意，声明时的类型区分符是不可省略的。

（6）声明区分符由存储类区分符、类型限定词、类型区分符组成，其中类型区分符在 C++ 中是不可省略的。比如 extern const int *p;，其中 extern const int 共同组成声明区分符，extern 是存储类区分符，const 是类型限定词，int 是类型区分符，*p 是声明符。再比如 int p;，它的声明区分符只有一个，即类型区分符 int，后面的 p 是声明符。

（7）声明符。

- 声明符可以有多个，多个声明符之间使用逗号隔开，这样就可以声明多个变量了。
- 有 5 种不同的声明符，即标识符、函数声明符()、数组声明符[]、指针声明符*、引用声明符&。这 5 种声明符可以相互进行嵌套，嵌套之后仍是一个声明符且是一个整体。比如*和标识符结合是一个声明符，表明该标识符是指针。再比如*a[2]也是一个声明符，表明该标识符是一个数组，这个数组中存储的是一个指针变量，根据声明符的不同，我们可以声明不同的对象。
- 注意：在声明符中也可以出现类型限定词，该类型限定词一般出现在指针声明符之后或函数声明符的小括号内（即用于限定形参）。比如 int *const p;，其中的类型限定词应属于声明符的内容，而不是类型区分符，即声明符是*const p;，声明区分符是 int。
- 若声明符是一个标识符，则标识符会被声明为一个变量。
- 函数声明符()：若标识符后跟一对圆括号，圆括号内可能有形参表，那么这时标识符将被声明为一个函数。比如 int f(float);就表示标识符 f 是一个返回类型为 int 且带有一个 float 形参的函数。
- 数组声明符[]：若标识符后跟一对方括号，方括号内是任意的常量表达式，那么这时标识符将被声明为一个数组。比如 int a[11];表示标识符 a 是一个 int 类型数组。
- 指针声明符*：若在标识符的前面有一个星号“*”，在星号和标识符之间可能会有类型限定词，那么这时标识符将被声明为一个指针。比如 int *p;表示标识符 p 是一个 int 类型指针；再比如 int *const p; float *p;等。
- 引用声明符&：若在标识符的前面有一个符号“&”，那么这个标识符将被声明为一个引用。比如 int &b;表示标识符 b 被声明为一个类型为 int 的引用。

（8）理解声明符和声明区分符的作用。在声明多个变量时，就能体现出声明符的作用了，声明符说明了所声明的标识符是普通变量、函数、指针、数组还是引用。在声明多个标识符时，各标识符之间是以逗号分隔的，但每个声明都应是“声明区分符+声明符”的形式。因此，在使用逗号声明多个标识符时，应把声明区分符和声明符分别找出来，然后才能判断出所声明的标识符究竟是普通变量还是指针、数组、函数、引用。比如 const int *const f1(),*const p1, p2, *f2();，这个声明看起来很复杂，其实将声明区分符和声明符区别开就很简单了，这里声明了 4 个标识符 f1、p1、p2、f2，其中 const int 是声明区分符，*const f1()、*const p1、p2 和*f2()是声明符。因此，这里声明了一个函数 f1，该函数返回一个指针，这个指针是常量，指向 const int 类型的

变量；同理，指针 p1 是一个常量，指向 const int 类型的变量；p2 是 const int 类型的变量，而不是 const int * const 类型；f2 是一个函数，这个函数返回一个指针，这个指针指向 const int 类型的变量。其实 p1、p2 和 f2 分别相当于 const int *const p1;、const int p2;和 const int *f();，因此理解声明符和声明区分符是相当重要的。

4．声明的规则和限制

（1）注意：声明语句后面有一个分号，表示此声明语句到此结束。

（2）在一个声明中最多只能有一个存储类区分符，即 extern、static、auto、register、typedef 只能有其中一个。

（3）一个标识符必须且只能指定一种类型。也就是说，不能让一个变量既是 int 类型又是 float 类型。在 C++中，必须为标识符指定类型，C++没有默认的类型。

（4）可以使用多个相同或不同的类型限定词（即 const 和 volatile），顺序无关紧要，编译器会忽略相同的多余的限定词。

（5）存储类区分符、类型限定词、类型区分符必须在标识符的前面，三者的顺序无关紧要。比如 int const a 与 const int a 是相同的，int const *p 与 const int *p 是相同的，但要注意 int * const p 与 const int *p 是不同的，具体内容请参看指针章节。

（6）若函数是类的一个成员，那么类型限定词 const 和 volatile 可以出现在函数名的后面。注意：类的成员变量不能将 const 放在变量名的后面。

（7）声明符中的()、[]应放在标识符的后面，*、&、*const 应放在标识符的前面，且()与[]优先级相同，()和[]比*和&的优先级高。

（8）注意：*可以放在类型限定词 const 和 volatile 的前面，但不能放在类型区分符（如 int、float 等）和存储类区分符（如 auto、extern 等）的前面。

（9）函数的返回值不能是一个函数，但可以是一个函数指针。比如 int f()();是错误的，int (*f())()是正确的。

（10）函数的返回值不能是一个数组，但可以是一个指向数组的指针。比如 int f()[2]是错误的，int (*f())[];是正确的。

（11）数组里面的元素不能是函数，但可以是函数指针。比如 int f[2]()是不正确的，int (*f[2])();是正确的。

（12）在数组里面可以有其他数组，二维数组 int f[3][4];就属于此情形。

（13）使用声明语法可以产生很复杂的声明，比如 int *const volatile *(*f)();等。关于复杂声明的类型的判断方法，本文后面会做详细介绍。

（14）在理解声明时不必知道函数、指数、数组是什么，只需知道它们的形式即可，即在标识符后有方括号的是数组，比如 a[]；在标识符后有圆括号的是函数，比如 f()；在标识符前有星

号的是指针，比如*p 等。

5. 声明与定义

（1）声明是一条语句，声明为对象起了一个名字，同时为这个名字确定了一种类型。

（2）声明的作用。

- C++使用声明语句告诉编译器一个对象的名字。比如 int x;语句表示 x 是一块类型为 int 的内存区域的名字。
- C++使用声明语句告诉编译器一个对象是什么类型。比如 int x;语句就能告诉编译器变量 x 的类型为 int。一个命名的对象必须有一个类型，有了具体的类型我们才能确定对对象做出什么样的操作，对象能够接收什么样的值等，C++使用声明语句来实现此目的。

（3）定义：是一个声明，定义对声明的实体给出了一个完整的描述。也就是说，一个定义明确地指明了一个名字代表什么。特别地，变量的定义会为其分配内存空间。

6. 声明与定义的区别及注意事项

（1）C++中的名字（标识符）必须先声明后使用。也就是说，在使用前必须先确定其类型，以通知编译器这个名字所引用的是什么类型的实体。

（2）对同一个名字只能定义一次，因为定义为命名对象（比如变量、函数名等）分配了内存空间，同一个名字的对象只能分配一个内存位置，所以只能定义一次。

（3）可以对同一个名字声明多次。

（4）对同一个名字的所有声明必须具有相同的类型。

（5）进行了初始化操作的任何语句都是定义。

（6）每个定义都是一个声明，但声明未必是定义。

（7）非定义的声明仅仅告诉编译器程序中有这么一个具有指定类型的名字，因此声明不会为对象分配内存空间（比如为变量分配内存空间，为函数指定函数体）。

（8）对于变量来说，其声明只说明了类型，而定义则会为该变量分配存储空间。

（9）对于函数来说，声明也提供了类型（即参数类型和返回类型），而定义才会提供函数体（即由大括号{}括起来的部分）。

（10）注意：函数体是作为程序的一部分而被保存在内存中的，因此函数和变量的定义都消耗了内存，而声明则没有。

7. 变量的声明、定义及初始化、赋值

（1）注意：这部分所讲的变量不包括函数、指针、数组，关于函数、指针、数组的声明与定义详见相关章节。

（2）对于变量来说，未使用 extern 关键字的声明是一个定义，因此会为变量分配内存空间。

（3）初始化：是指在定义变量时给出变量的初始值。比如 int a=1;表示将变量 a 的值初始化为 1，这时变量 a 拥有值 1。

（4）两种初始化变量的形式：复制初始化和直接初始化。

（5）复制初始化：就是在定义时使用赋值运算符“=”进行的初始化。比如 int a=1; float b=2.0; 就是复制初始化。

（6）复制初始化有两种形式：第一种是使用表达式，比如 int a=1; int b=2+3;等；第二种是使用初始化列表（即使用大括号）。初始化列表就是将各个表达式放在大括号中，表达式之间使用逗号隔开，这种形式一般用于初始化数组，比如 int a={1}; int b={2+3}; int c[2]={1,2+3};。有关数组的初始化请参看数组相关内容。

（7）注意：使用初始化列表初始化变量时，大括号中的表达式在 C++标准中是以逗号结尾的（也可省略），因此 int c[]={1,3,};是正确的初始化语法。

（8）直接初始化：是指在定义时将值放在变量名后的一对小括号中。比如 int a(1); 将变量 a 的值初始化为 1。

（9）注意：在直接初始化时，变量名后的括号不能为空值，若为空值，则表示声明一个返回指定类型的无参函数（这并不是错误，只是声明了一个函数）。比如 int a(); 表示声明的是一个返回 int 类型没有形参的函数，而不是在定义一个变量 a。

（10）复制初始化与赋值的区别：复制初始化使用赋值运算符“=”对变量进行初始化，会让人以为初始化时就是在赋值，其实初始化与赋值是有区别的，初始化是在创建变量时给它一个初始值；而赋值则发生在变量已经创建之后，擦除旧值写入新值的操作。比如 int a; a=2; 就是赋值，int a;语句定义了一个变量，并为变量分配了存储空间，这时变量已经拥有一个值，若为局部变量（有关局部变量见相关内容），那么这个值是随机的，然后 a=2; 将变量 a 的随机值擦除，并将新值 2 写入到变量 a 中；而 int a=2; 则是在创建变量 a 时（这时变量 a 还没有值）就将其值初始化为 2 了。

（11）在函数体外定义的或使用 static 定义的内置类型变量都会自动初始化为 0，若是在函数体内定义的内置类型变量，则不会对其进行初始化，这时变量虽然没有初始化，但不代表就没有值，编译器一般都会为变量生成一个随机值，若在程序中使用了这种变量，将会导致一个难以发现的错误，因此建议对内置类型变量最好都要初始化，然后再使用。

（12）变量的非定义声明：若使用 extern 来声明变量，同时没有初始化变量，则是对变量的非定义声明，extern 表明此变量在程序中的其他地方或者其他文件中有定义。比如 extern int i; 表示声明一个名为 i、类型为 int 的变量，此变量在其他地方有定义。

（13）非定义声明变量的条件：① 使用 extern 关键字；② 变量没有初始化。比如 extern int x;表示声明一个变量 x，而 int y;因为没有使用 extern 关键字，因此不是非定义声明；同理，extern

int z=1; 表示对变量进行了初始化，因此同样不是一个非定义声明。

（14）注意：使用 extern 关键字声明变量时不能在函数内对其进行初始化，比如 void f(){extern int a=1;}将会得到一个错误。

（15）非定义声明变量与定义的区别：使用 extern 对变量进行的是非定义声明，因为变量没有定义，所以可以对变量进行多次声明，而定义则只能进行一次。

（16）声明（或定义）和初始化多个变量：要声明多个变量，只需在各变量名之间使用逗号隔开即可，比如 int a,b,c;、extern int a,b,c;等。要对所声明的多个变量进行初始化，只需对要初始化的变量使用初始化方法即可，比如 int a, b=2, c;、int a=1,b=2,c=3;等。

8. 使用类型限定词（const 和 volatile 关键字）声明或定义变量

（1）可以使用多个相同或不同的类型限定词（即 const 和 volatile），顺序无关紧要，编译器会忽略相同的多余的限定词。

（2）若变量使用 const 限定词，则该变量就是常量或只读的，定义该变量后，将无法修改其值。

（3）volatile 限定词，表示变量可能会被其他程序或事件所修改。由于其他程序或事件可能修改变量的值，因此 volatile 关键字还告诉编译器，在每次使用该变量时都要进行重新读取。

（4）编译器可以将非 volatile 类型的 const 对象放在只读存储区，若从不使用它，则编译器有可能不为该变量分配内存空间。

（5）若同时使用 const 和 volatile 限定变量，则此变量无法被程序本身修改，但可以被其他程序或硬件修改。

（6）使用 const 声明常变量时，const 关键字与类型区分符的顺序无关紧要，比如 int const a=1;与 const int a=1;是等价的。

（7）使用 const 声明的变量必须进行初始化，因为使用 const 后变成常量，其值无法在以后使用赋值运算符进行修改。

（8）注意：若 const 的左右两边都没有类型区分符，且左边也没有指针运算符“*”时，这种声明是错误的。比如 int (const p)=0;和 int (const p[3]); int *(const p)=0; int *(const p[3])={0};等都是错误的。

（9）volatile 限定词很少使用，因此这里不重点介绍了。

示例 3.1：声明的规则和限制、变量的声明和定义、复杂声明

```
#include <iostream>
using namespace std;
 //a1=2;          //错误，标识符必须先声明后使用
 int a2;          //未使用 extern 的变量声明同时是一个定义，因此此语句会为变量分配内存空间
//int a2;                  //错误，同一个变量只能定义一次
extern int a3;              //这是对变量的非定义声明，使用 extern 关键字，而且未初始化
```

```
extern int a3;                  //正确，可以对同一个标识符声明多次
//extern long int a3;           //错误，对同一个标识符的多次声明必须要有相同的类型
extern int a5=2;                //进行了初始化的任何语句都是一个定义
//extern auto int a6;//错误，只能指定一个存储类区分符，即 auto,extern,static,register 中的一个
//const a7;                     //错误，在 C++中必须为标识符指定类型区分符（比如 int、float 等）
const const volatile int a8=8;  //正确，在声明中可以使用多个相同或不同的类型限定词，编译器会忽略
                                //多余的相同类型限定词。若同时使用 const 和 volatile 限定变量，
                                //则此变量无法被程序本身修改，但可以被其他程序或硬件修改。注意：
                                //const 常量必须初始化
//int a9 auto;                  //错误，存储类区分符应在标识符的前面
//int a10 const;                //错误，类型限定词应出现在标识符的前面，但在类中声明成员函数时除外
class A{void f1() const;};      //在类中声明的成员函数可以将类型限定词放在函数名的后面。注意：只有函数
                                //才是正确的，若是变量将会出错
//int []a11;                    //错误，()、[]运算符在声明时必须放在标识符的后面，此语句在 Java 中是允许的
//int a12*;                     //错误，声明时*、&、*const 必须放在标识符的前面
//int *static a13;              //错误，*不能放在类型区分符（如 int 等）和存储类区分符（如 auto 等）的前面
int *const a14=0;               //正确，*可以放在类型限定词的前面，这时类型限定词限定的是左边的*（即指针）
//int f2()();                   //错误，函数不能返回函数。
int (*f3())(); //正确，函数的返回值可以是一个指向函数的指针，函数 f3 返回一个指向函数的指针
//int f4()[];                   //错误，函数的返回值不能是一个数组
int (*f5())[];                  //正确，函数的返回值可以是一个指向数组的指针
//int a15[]();                  //错误，数组中的元素不能是函数
int (*a16[])();                 //正确，数组中的元素可以是指向函数的指针
int a17[3][11];                 //正确，数组中的元素可以是数组，这就是普通的二维数组
int a18=18;                     //变量的复制初始化语法之一
int a19={19};                   //使用大括号{}对变量进行复制初始化，此方法一般用于声明有多个元素的数组
int a20[]={1,2,};               //使用大括号{}初始化有多个元素的数组，各值之间使用逗号隔开，且以逗号结
                                //束，当然也可以省略最后的那个逗号
int a21(21);                    //变量的直接初始化方法，就是在变量名后紧跟小括号并在其中指定初始值
int a22();                      //注意：直接初始化时，变量名后的括号不能为空值，若为空值，则表示声明一个返回指定
                                //类型的无参函数，因此这里表示声明一个返回 int 类型的无参函数，不是使用直接初始化
                                //方式初始化变量，此语句没错
int a23;                        //在函数体外定义的变量会被自动初始化为值 0
int main()
{   int b1,b2,b3=33,b4,b5=3; //同时声明或初始化多个变量时使用逗号隔开
    //extern int a24=3;         //错误，在函数内部不能对使用 extern 声明的变量进行初始化
    extern int a23;             //extern 表明此变量在程序中的其他地方或者其他文件中有定义，这里 a23
                                //在函数体外被定义
int a25;                        //在函数体内定义的内置类型变量不会对其进行初始化，编译器一般都会为变量生成一个随机值

                                //复制初始化与赋值区别：初始化是在创建变量时给其一个初始值
                                //而赋值则发生在变量已经创建之后，擦除旧值写入新值的操作
    int a26=26;                 //这是复制初始化，在创建变量 a26 时（这时变量 a26 还没有值）就将其值初始化为 26
    a26=261;                    //这是赋值，表示将变量 a26 的旧值擦除，并将新值写入到变量 a26 中
    //a8=8;                     //错误，不能对常量 a8 的值进行修改
    //int *(const p)=0;         //错误，const 的左右两边必须有类型区分符或者左边必须有指针运算符*。这
                                //里 const 的左边是小括号“(”
}
```

3.2　复杂声明和 typedef 的使用

3.2.1　复杂声明

1．运算符的基本规则

（1）声明之所以复杂，主要是因为*、[]、( )这三种运算符的作用，其中*表示指向×××的指针，()表示返回类型为×××的函数，[]表示×××的数组。

（2）*是前缀运算符，()和[]是后缀运算符，前缀运算符只能出现在标识符的前面，而后缀运算符则只能出现在标识符的后面。比如 int [22]a; int b*; int *[]c;和 int *[] f();都是错误的，因为后缀运算符[]只能出现在标识符的后面。

2．复杂声明的分析方法

（1）优先级规则。

① 首先应从未声明的标识符（就是名字）开始分析，应注意区分函数的形参名和被声明的标识符名。

② 声明中被小括号“()”括起来的内容，优先级最高，若小括号有多层，则最里层的优先级最高，小括号不会与函数运算符的小括号相混淆，这从语句中可以明显地看出来。

③ 后缀运算符比前缀运算符优先级高。

④ 后缀运算符具有相同的优先级，当有多个后缀运算符时，应按从左到右的顺序进行分析。

⑤ 当有多个前缀运算符时，应按从右到左的顺序进行分析。

⑥ 当有 const 和 volatile 限定词时，若 const 或 volatile 后面紧接着的是类型区分符（如 int、float 等），则限定词将作用于类型区分符；否则，限定词将作用于其左边紧邻的指针运算符（*）。这里要注意，若 const 左边没有指针运算符，则 const 应作用于类型区分符，比如 int const *p;和 const int *p;是等同的，都表示指针指向的是 const int 类型的变量；同理，const int a[1];和 int const a[1];也是等同的，都表示数组 a 存储的数据是 const int 类型的。

⑦ 若声明中有多个标识符（主要是函数的形参名），则从左向右看，最左边的标识符是未声明的。因为函数形参名只能出现在小括号中（区别于优先级的小括号），而表示函数的小括号“()”是后缀运算符，它只能出现在标识符之后，而未声明的标识符也只能有一个，因此从左向右看第一个出现的标识符一定是未声明的，再后面的标识符只能是函数形参名。

（2）右左法则。

① 从标识符开始，先向右看，然后再向左看，若遇小括号，则进入小括号中使用上述规则；当有 const 和 volatile 限定词时，规则同优先级规则法第 6 条；当有多个标识符时，则优先级规

则法的第 7 条同样适用于右左法则。

② 右左法则原理：由优先级规则法的第 1~5 条可以明显地看出右左法则的原理。因为后缀运算符的优先级高于前缀运算符，且后缀运算符只能出现在标识符之后，所以分析标识符类型时，总要先向右边看有没有比前缀运算符优先级更高的后缀运算符，然后再看左边。

③ 注意：右左法则不是指右边看一下左边看一下，而是要先向右看完所有的运算符，直到遇到反小括号或没有运算符时再向左看。比如 int **(*p[4][5])[6];，先向右看，p 先与[4]结合，再向右看，接着与[5]结合，再向右看，遇到反小括号，向左看，与*结合，跳出小括号，向右看，与[6]结合，右边已经没有东西了，再向左看，与*结合，再向左看，接着与*结合，最后再向左看，与 int 结合。

（3）复杂声明时标识符究竟是什么（数组、指针、函数或变量）。

① 若与标识符结合的第一个运算符是指针符号“*”，则该标识符就是指向×××的指针。也就是说，标识符就是一个指针，只是这个指针指向的内容可能很复杂。比如 int (*b)[];，标识符 b 先与*结合，因此 b 是一个指针，这个指针指向一个数组；再比如 int *(*b)();，表示 b 是一个指针，这个指针指向一个返回类型为 int *的函数。

② 若与标识符第一个结合的是数组下标运算符“[]”，则该标识符是×××的数组。也就是说，这个标识符一定是一个数组，只是数组里存储的内容可能很复杂。比如 int *a[];，因为[]优先级更高，所以标识符 a 先与[]结合，故标识符 a 是一个数组，这个数组里的内容是 int *的指针。

③ 若与标识符第一个结合的是函数符()，则该标识符是返回×××的函数。也就是说，这个标识符就是一个函数，只是这个函数的返回类型可能很复杂。比如 int (*f())[];，表示 f 是一个函数，这个函数返回一个指针，这个指针指向一个数组。

示例 3.2：const int *const (*f())[];

① 从标识符 f 开始分析，标识符首先与()结合，说明 f 是一个函数。

② 按优先级规则法分析小括号内的内容，与*结合，因此函数 f 返回的是一个指针。

③ 接着与[]结合，说明函数 f 返回的指针指向一个数组。

④ 再分析 const 限定词，因为 const 后不是类型区分符，所以 const 与左边紧邻的*结合，说明函数 f 返回的指针指向一个数组，这个数组里存储的是常量指针。

⑤ 最左边的 const 后紧接一个类型区分符“int”，因此 const 与 int 结合，最后说明函数 f 返回的指针指向一个数组，该数组里存储的是常量指针，且这个指针指向 cont int 类型的常量。

示例 3.3：int (*(*p(int a))(int a, int b))(int a);

① 程序中有多个标识符，从左向右看第一个标识符 p 是未声明的。

② p 与(int a)结合，说明 p 是一个函数，这个函数有一个名称为 a、类型为 int 的形参。

③ 接着与*结合，说明函数 p 返回一个指针。

④ 跳出小括号，与后面的(int a, int b)结合，说明函数 p 返回的指针指向一个带有两个形参的函数，这两个形参类型为 int，名称分别为 a 和 b。

⑤ 再与*结合，说明函数 p 返回的指针指向的函数返回的又是一个指针。

⑥ 跳出小括号，与最右边的(int a)结合，说明函数 p 返回的指针指向的函数返回的指针指向的又是一个函数，这个函数带有一个 int 类型的形参，名称为 a。

⑦ 最后与最左边的 int 结合，说明函数 p 返回的指针指向的函数返回的指针再指向的函数返回一个 int 类型的值。

示例 3.4：int *(*p)[4][5][6];

① p 与*结合，说明 p 是一个指针。

② 跳出小括号，与后面的[4]结合，说明此指针指向一个有 4 个元素的一维数组。

③ 接着与后面的[5]结合，说明这个一维数组存储的是一个有 5 个元素的一维数组，其实就相当于指针指向 4×5 的二维数组。

④ 再与后面的[6]结合，说明 4×5 的二维数组存储的是一个有 6 个元素的一维数组，即相当于一个 4×5×6 的三维数组，因此指针 p 指向的是 4×5×6 的三维数组。

⑤ 最后与左边的*结合，说明这个三维数组存储的是一个指针，因此指针 p 指向 4×5×6 的三维数组，此三维数组存储的是 int*的指针。

3．复杂声明的类型

（1）声明时的类型可用于强制类型转换，也可用于 typedef 重定义类型；复杂声明时，声明的类型只需把声明时的标识符去掉即可。比如 const int *const (*f())[];，其类型就是 const int *const (*())[];，表明这是一个函数类型。

（2）若要知道函数返回的类型，则应将代表函数的小括号去掉。比如 const int *const (*f())[];，其函数返回的类型就是 const int *const (*)[];，表明函数返回一个指针。

（3）若要知道指针指向的类型，同样应把代表指针的“*”去掉。比如 int (*(*p)[4])();，其指针指向的类型是 int (*[4])();。

4．怎样分析一个无标识符的复杂类型

（1）判断一个复杂的无标识符的类型时，最好先把标识符(名称)填写在相应的位置，然后再根据复杂声明的分析方法来判断这个复杂类型。

（2）标出相应名称所在位置的方法：

① 对于一个复杂的类型，其名称应在前缀运算符和后缀运算符之间，对于有多个前缀运算符和后缀运算符的情况（一般出现在函数形参中），则从左向右看，第一个出现的前缀运算符和

后缀运算符对必是标识符所在位置（其原理同复杂声明时有多个标识符的情况）。

② 若只有前缀运算符，其名称应在所有前缀运算符的最右边。

③ 若只有后缀运算符，其名称应在所有后缀运算符的最左边。

④ 若前缀运算符出现在后缀运算符的右边或后缀运算符出现在前缀运算符的左边，则一定是错误的语法。因为前缀运算符只能出现在标识符的前面，后缀运算符只能出现在标识符的后面。

示例 3.5：const int *const (*())[]

首先找到前缀运算符和后缀运算符之间的结合点在“*()”，因此其名称应在这两个运算符之间。假设名称为 f，则完整的语句是 const int *const (*f())[]，然后再根据复杂声明的分析方法来分析这是什么类型。

示例 3.6：int *(*[2])(int *[3]);

从左向右看，找到第一个前缀运算符和后缀运算符对*[2]，其名称应在这两个运算符之间。假设名称为 f，则完整的语句是 int *(*f[2])(int *[3])。

示例 3.7：int **[](*(*)[])()。

语法错误，不管声明多复杂，只要前缀运算符出现在后缀运算符的右边或后缀运算符出现在前缀运算符的左边，都一定是错误的语法。

3.2.2 typedef

1．typedef 基本规则

（1）typedef 可以给指定的类型取一个新名字，这个新取的名字也被称为自定义类型名。

（2）typedef 并不会产生一个新的类型，而只是为指定的类型取了另一个名字。

（3）typedef 声明的语法形式与普通声明基本相同。比如 typedef int c;，表示给 int 类型取一个新名字 c，以后可以使用 c 来声明变量，就像使用 int 声明一样。比如 c a;，表示将变量 a 声明为 int 类型。

（4）使用 typedef 可以将复杂的很长的声明变得简短。比如 typedef int *p[2];，把存储两个指向 int 类型指针元素的数组的类型使用名称 p 来代替，以后就可以直接使用新名字 p 来声明这种类型了。比如 p c;，c 的类型就是存储两个指向 int 类型指针元素的数组的类型。

（5）注意：typedef 是存储类区分符中的一种，在声明时只能有一个存储类区分符，因此 typedef static int a; 是错误的，这里使用了两个存储类区分符。

2. typedef 要点（typedef 的核心）

（1）注意：若 typedef 声明的新名字所代表的类型很复杂，则可以使用分析复杂声明的方法分析该类型。比如 typedef int p;，新名字 p 代表的类型是 int；typedef int p();，新名字 p 代表的类型是一个无形参、返回 int 类型值的函数；typedef int p[3];，名字 p 代表的类型是有 3 个 int 元素的一维数组。

（2）使用 typedef 声明的新名字声明变量时，若在声明变量时有类型限定词（比如 const）和其他声明符（比如指针运算符*、下标运算符等）时，则标识符首先与这些声明符组合成一个新的声明符，然后再使用这个新的声明符与 typedef 重命名的新类型结合。也就是说，typedef 新命名的类型具有低优先级。

示例 3.8：typedef int *T; const T p1;，则 p1 与 int *cont p1;相同，p1 不等于 cont int *p1;

首先 p1 与 const 结合，说明 p1 是一个常量，然后再与 p 所表示的类型 int*结合，说明这个常量是一个指针，这个指针指向 int 类型。

示例 3.9：typedef int p[3]; p *p1;，则 p1 与 int (*p1)[3]; 相同，p1 不等于 int *p1[3];

首先 p1 与*结合，说明 p1 是一个指针，然后再与重命名的代表 int [3]的 p 结合，说明这个指针指向有 3 个 int 元素的数组。

示例 3.10：typedef void p(); p *p1[4];，则 p1 与 void (*p1[4])();相同，p1 不等于 void *p1[4]();

首先 p1 与下标运算符结合，表明这是一个有 4 个元素的数组，这个数组返回一个指针，这个指针指向 p 所表示的类型，即无形参、无返回值的函数。

示例 3.11：typedef int *p; const p *pp;，则 pp 与 int *const *pp; 相同

首先 pp 与*结合，说明 pp 是一个指针，然后再与 const 结合，说明 pp 指向常量，接下来再与 p 所代表的类型 int *结合，说明这个常量是一个指针，这个指针指向 int 类型，最后 pp 的类型就是 int *const *;。

3. 使用 typedef 简化复杂声明

（1）使用 typedef 简化复杂声明的基本原理：使用从左到右、从外到里、层层剥茧法简化复杂声明，也可叫作左右法则。

（2）原理分析。

① 类型区分符决定应从左边开始分析：使用 typedef 简化声明时，在没有类型区分符的情况下是无法重命名类型的，类型区分符在声明时又是唯一的（函数形参除外），而且类型区分符一般出现在左边，因此应先从左边的类型区分符开始化简。只有在有了第一个使用 typedef 命名的新类型之后，才能继续向后化简，而且还是要从类型区分符开始入手，而这时的类型区分符

必定是使用 typedef 重命名的新类型，这个新类型一般也在左边，因此仍从左边的类型区分符开始化简。一直重复下去。

② typedef 的特点决定应从左边的运算符开始化简，然后才是右边的运算符。

- 因为使用 typedef 新命名的类型具有低优先级，所以使用 typedef 简化后的类型中的运算符的优先级不能高于剩下的未化简部分组成的新声明符中运算符的优先级；否则不能还原为以前的类型。比如 int *p[4];，这样简化将不能还原为以前的类型；typedef int T[4]; T *p;，因为 typedef 声明的新类型中的下标运算符[]的优先级高于剩下的未化简部分组成的新声明符*p 中的*运算符的优先级，所以不能还原到以前的 int *p[4];。因为新组成的声明符具有更高的优先级，而被还原为 int (*p)[4];，这样化简 typedef int *T1; T1 p[4]; 便与 int *p[4]的类型相同。
- 因为左边的运算符具有较低的优先级，所以从左边开始化简，可以保证化简后类型中的运算符的优先级低于未化简部分组成的新声明符中运算符的优先级；若从右边的运算符开始化简，如果标识符的左边还有优先级更低的*运算符时，则化简后类型中的运算符的优先级就存在高于剩下的未化简部分组成的新声明符中的运算符的优先级情况，所以化简后不能还原为以前的类型。

（3）使用 typedef 简化复杂声明的具体步骤（左右法则）。

① 首先找到最左边的类型区分符。

② 然后从类型区分符这里从左向右看，若没有遇到小括号则继续向右看，直到遇到小括号为止。

③ 若遇到小括号，则表示左边的运算符分析完毕，现在从最右边开始向左看，分析右边的运算符，直到右边的运算符分析完毕，或遇到小括号。若遇到的小括号是函数运算符，则转进函数进行上述分析，然后跳出函数继续向左看，直至右边分析完毕。若小括号不是函数运算符，而是用来改变运算符优先级的，则转进小括号进行上述分析。注意：小括号是函数运算符还是用来改变运算符优先级的，可以明显地看出来。

④ 若复杂声明有函数，函数有形参，形参也是复杂声明，则可以按以上法则先使用 typedef 简化函数的形参，然后再简化其他部分内容。

⑤ 运用以上法则，使用 typedef 简化多少个运算符，根据需要来决定。既可以使用 typedef 简化一个运算符，也可以一次简化多个运算符，但都得根据以上规则进行简化。

⑥ 若使用 typedef 简化，则根据简化后的类型重写出复杂声明，再运用以上法则继续简化。

⑦ 整理简化后的声明，将相同类型的声明使用一个名称代替。

示例 3.12：int (*f(int (*p)(int *, int *)))(int *, int *);

① 首先找到最左边的类型区分符 int。

② 然后从左向右看，找到小括号，表示左边分析完毕。

③ 再从最右边开始向左看，遇到小括号，这个小括号是函数运算符，且函数的形参不是复杂声明，则可以使用 typedef 进行一次简化，比如 typedef int T1(int *, int *); 。

④ 写出简化后的复杂声明为 T1 (*f(int (*p)(int *, int *)))，去掉多余的小括号后的结果是 T1 *f(int (*p)(int *, int *))。

⑤ 接下来简化 T1 *f(int (*p)(int *, int *))，从左向右看，最先遇到类型 T1，然后遇到*，说明是一个指针，这个指针指向 T1。

⑥ 再向右看，遇到标识符，左边分析完毕。从最右边向左看，遇到小括号，这个小括号是函数运算符，且函数的形参是复杂声明，则先简化函数的形参 int (*p)(int *, int *)，这个形参可以使用 typedef 直接简化，比如 typedef int (*T2)(int *, int *);。

⑦ 写出简化后的声明：T1 *f(T2 p);。

⑧ 现在只有指针和函数运算符没有简化了，再从左向右将其简化：typedef T1 * T3;。

⑨ 写出简化后的声明：T3 f(T2 p);。

⑩ 简化后 T3 的类型为 int (*)(int *, int *);，与 T2 的类型相同：因此再将重复的类型 T3 使用 T1 代替，最终的简化结果是 T2 f(T2 p)。

⑪ 经过整理，int (*f(int (*p)(int *, int *)))(int *, int *);只需使用 typedef int (*T2)(int *, int *); 简化一次即可，简化后的结果是 T2 f(T2 p);。

示例 3.13：int (*(*b[3])(void (*p)()))[4];

① 经过类型分析，此声明有函数，且函数的形参 void (*p)()声明复杂，先简化函数的形参：typedef void (*T1)()。

② 简化后的声明：int (*(*b[3])(T1 p))[4];。

③ 从左向右看，除了类型区分符 int，遇到小括号，转至最右边向左看，遇到数组[4]，说明数组存储的是 int 类型数据。

④ 遇到小括号，此小括号是用来改变优先级的，则进入小括号，然后从最左边向右看，遇到指针，说明指针指向数组[4]，这个数组存储的是 int 类型数据，简化一次 typedef int (*T2)[4];。

⑤ 简化后的声明：T2 (*b[3])(T1 p);。

⑥ 以上声明已经很简单了，但注意不能这样简化以上声明：typedef T2(*T3)[3];，结果 T3 b(T1 p);是错误的，因为此结果中的 b 是函数，而 b 实际上是一个数组。虽然 T2(*b[3])(T1 p); 已经很简单了，若要再简化，也要按左右法则进行，即只能这样简化：typedef T2 T3(T1 p); T3 *b[3]; 或 typedef T2 *T3(T1 p); T3 b[3];。

4. 还原 typedef 简化后的复杂声明

（1）还原有两种方法：一种是分析法；一种是替换法。

（2）分析法比较简单，只需按照优先级规则分析即可，但还原过程很复杂，推荐使用替换法。

（3）替换法：就是把剩下的未化简部分重新组成一个新的声明符，然后使用这个声明符直接替换掉 typedef 中声明时的标识符即可。因为 typedef 新命名的类型具有低优先级，也就意味着新组成的声明符具有高优先级，当新组成的声明符替换掉 typedef 中的标识符后，应能保证 typedef 中的运算符不会改变其优先级，所以替换时应使用小括号把新组成的声明符括起来，以确保正确性。

（4）注意：使用的是剩下的未化简部分重新组成的新的声明符，去替换 typedef 中的标识符。声明符包括标识符、函数声明符“()”、数组声明符“[]”、指针声明符“*”、引用声明符“&”，还有类型限定词“const”。

示例 3.14：typedef void (*T1)(); typedef int (*T2)[4];

typedef T2 (*T3)[3]; T3 *b(T1 p);

使用替换法还原：

① 首先将未化简部分组成新的声明符*b(T1 p)，因为它具有高优先级，所以使用小括号括起来——(*b(T1 p))。这里必须使用小括号括起来，否则还原时可能会改变优先级。

② 然后使用(*b(T1 p))替换 typedef T2 (*T3)[3];中的 T3，结果为 T2 (*(*b(T1 p)))[3];。

③ 同理，未化简部分组成新的声明符，并使用小括号括起来——((*(*b(T1 p)))[3])。这里也可以不用加小括号，因为(*(*b(T1 p)))[3]都是最高优先级的运算符，在还原时不会影响剩下未简化部分组成的新的声明符。但为了安全起见，请加上小括号。

④ 再用((*(*b(T1 p)))[3])替换 typedef int(*T2)[4]; 中的 T2，结果为 int (*((*(*b(T1 p)))[3]))[4]。

⑤ 最后使用标识符 p 替换 typedef void (*T1)();，得到 void (*p)()。

⑥ 还原后的类型为 int (*((*(*b(void (*p)())))[3]))[4]，其中加黑的两对小括号是多余的。

使用分析法还原：

① 从最左边的标识符开始分析，b 是一个函数，它有一个形参，形参类型为 T1。

② 再分析 T1，形参 p 是一个指针，它指向一个返回 void 的函数。

③ 再分析 T3，T3 是一个指针，它指向数组[3]，数组存储的元素类型为 T2。

④ 再分析 T2，T2 是一个指针，它指向数组[4]，这个数组存储的元素类型为 int。

⑤ T3 的类型：T3 是一个指针，它指向数组[3]，这个数组存储的元素又是指针，该指针指向数组[4]，这个数组存储的元素类型为 int。

⑥ 转回标识符 b，b 是拥有一个形参的函数，这个函数返回一个指针，该指针指向另一个指针，这个指针指向数组[3]，此数组存储的元素又是指针，这个指针指向数组[4]，该数组存储的元素类型为 int。函数 b 的形参类型是指向一个返回 void 的函数的指针。

⑦ 还原后 b 的类型为 int (*(* *b(void (*p)()))[3])[4];。

第4章

运算符、表达式和左值专题

注意：本章所说的整型包括整数类型、字符型、布尔型及枚举类型。

本章使用“运算符”，不使用“操作符”，因为操作符与操作数只相差一个字，易产生混乱。

依实现、取决于实现：指的是取决于编译器（因为C++语法规则是由编译器来实现的）。

（1）对象类型：指的是可以确定内存单元大小的类型，包括整型、浮点型、字符型、布尔型等。注意，函数类型和不完全类型都是无法确定内存单元大小的，因此都不是对象类型。

（2）函数类型：用于描述函数。函数类型由其返回值类型、形参数目和形参类型表征，函数类型一般被表述为“返回结果为某类型的函数”。

（3）不完整类型：指的是大小（或成员）不确定的类型。元素数不确定的数组、成员没有定义的结构与联合类型、void类型都是不完整类型，而且void永远都是不完整类型。比如int a[];struct b;等都是不完整类型。可以在其他位置使用完整类型说明使不完整类型成为完整类型。比如对于struct b;，可以在其他位置使用struct b{...};将其说明为完整类型。

（4）算术类型：包括所有整型与浮点型，即包括所有整数类型（short, int, long等）、字符型、布尔型、枚举型及所有浮点型。

（5）标量类型：是不可再分为其他类型的类型，包括算术类型与指针类型两类，也可认为标量类型是除聚集类型之外的所有类型。

（6）聚集（合）类型：是由标量类型、聚集类型聚合而成的。数组、结构、类类型都属于聚集类型。注意，类中的成员不一定是聚集类型。比如class A{public: int b;}; A ma;，则ma.b的类型不是聚集类型，而是int类型，因为b是对象ma的成员，它的类型是int，不是聚集类型。

4.1　赋值、左值和右值

1．赋值

（1）在 C++中使用等于符号“=”表示对某个实体进行赋值。对某个实体赋值后，这个实体就拥有了该值，比如 a=2;表示将整数 2 赋给对象 a。

（2）注意：在 C++中“=”表示赋值，使用“==”判断两个实体是否相等，比如 a==b;表示判断 a 和 b 是否相等，若相等则为真，否则为假。

（3）在 C++中“=”被称为赋值运算符。

2．左值和右值

（1）左值和右值。

① 左值：原来是指位于赋值语句左边的东西。

② 左值：指的是变量所指存储数据的内存单元，因此左值是一个可被寻址的对象（即可以使用&运算符提取左值的内存地址），也可以从左值中读取一个值，若左值是非 const 的还能修改它的值。

③ 右值：指的是变量所指内存单元中的数据值。也就是说，右值表示一个值，因此不能改变这个值，也不能对其进行寻址（即不可以使用&运算符提取右值的内存地址）。最简单的右值就是字面值，比如 3, 4, 5.5, 'a'等就是右值。注意：临时对象也是右值，比如 int a=f();，其中函数 f()返回的值是临时对象，它是一个右值。

④ 左值可以出现在赋值语句的左边或者右边。

⑤ 右值只能出现在赋值语句的右边，不能出现在赋值语句的左边。

⑥ 左值和右值主要是为了区分是否可以对变量进行赋值（当然并不全是这样），若变量是不可修改的左值，则不可以对其进行赋值。

（2）C 标准对左值的定义。

① 左值在 C89 标准文件中的定义：左值是指示对象的表达式，它是一种对象类型或除 void 外的不完整类型。当说某对象具有一种特定的类型时，该类型由用于指示该对象的左值来规定。

② 理解标准定义的左值。

- 从定义可以看出，左值是一个表达式。也就是说，只要这个表达式表示一个对象（内存单元），它就是左值。注意：因为对象是指某种类型所占据的一片连续的内存单元，命名的对象就是变量，所以变量指示了一个对象；还因为一个单独的变量也是表达式，所以它就是左值。当然，也可以反过来理解，左值在某种程度上是一个变量或一个对象或一片连续的内存单元，因此有时左值与对象具有相同的意思，使用左值的概念只是为了说明某个变量可以位于赋值运算符的左边（即可以对其赋值）。因为字符串在 C

中占据了一片连续的内存单元，同时字符串也是一个表达式，所以它也是左值，但这是一个不可修改的左值。

- 比如 const int a=2; cout<<a;，这里的常变量 a 是左值，因为它指示一个对象的表达式；但是不能对其赋值，因为赋值表达式有一条规则是“不能对有 const 限定的左值赋值”。

（3）左值到右值的转换。

① C89 标准规定：除作为 sizeof、&、++、--、.运算符的操作数或赋值运算符的左操作数外，不是数组类型的左值总是被转换为存储在所指示对象中的值，于是不再成为左值。若左值是限定类型的，则该值的类型是左值的非限定形式；否则该值的类型就取左值的类型；若左值是不完整类型且不是数组类型，则行为是未定义的。

② 左值到右值转换的实际应用：左值和右值是相对于变量（或对象）而言的，因为在 C++ 中一个变量既可以出现在赋值语句的左边，也可以出现在赋值语句的右边，在这种情况下就存在一个左值到右值的转换问题。比如 a=a+1;，变量在左边还是右边所代表的意义就不一样，因此每个变量都有两个值，即变量所指内存中的数据值和变量所指存储数据的那块内存单元。变量所指内存单元中的数据值被称为右值，而变量所指存储数据的内存单元被称为左值。在 a=a+1; 语句中，左边的变量 a 表示它所指的内存单元，而右边的变量 a 指的是它所指内存中的数据值，在执行加法前，会先把 a 的值抽取出来，因此变量 a 就从左值转换成右值。所以结果就是，将右边的变量 a 所指内存单元中的数据值加 1 之后，再写入左边的变量 a 所指的内存单元中。

③ 比如 cout<<a;，会输出其值而不是地址。因为按照 C89 标准规定，cout<<a 应输出左值 a（非数组）所指示对象（相当于内存）中的值。注意，这里输出的值的类型是左值 a 的非限定形式，也就是 int 类型，而不是 const int 类型；若 a 是数组类型的左值，即 a 是数组，则按照标准规定是不会输出其所指示对象中的值的，程序会输出一个地址（具体见相关章节）。

（4）可修改的左值：是指不是数组类型，不是不完整类型，也不是 const 限定的类型，若类型是结构或联合类型时，其任何成员都不是 const 限定的类型。比如 int a;，变量 a 是可修改的左值；const int b;，变量 b 是不可修改的左值；int c[22];，数组变量 c 是不可修改的左值。

4.2 表达式和运算符

4.2.1 基础

1. 运算符基本概念

（1）运算符（或称为操作符）：C++使用运算符对数据进行操作。C++提供了多种运算符，同时还支持对运算符进行重载操作。本文将介绍使用内置类型作为操作数进行计算的运算符。

（2）C++提供的运算符有：算术运算符（如+、-、*，/）、关系运算符（如>、<、>=等），逻辑运算符（如&&、||等）、赋值运算符（如=、+=）、递增和递减运算符（如++、--）、条件操作符（即?:）、逗号操作符、位操作符（如~、&等）。

（3）操作数：即运算符所操作的对象。比如 a+2，其中 a 和 2 都是运算符+的操作数。

（4）运算符对操作数的类型是有要求的，并不是任何类型的操作数对运算符都适合，比如求余运算符“%”只能对整型数进行操作。

（5）根据操作数的数量对运算符进行分类。

- 一元运算符：作用在一个操作数上的运算符，比如取址运算符“&”。
- 二元运算符：作用在两个操作数上的运算符，比如加“+”和减“-”运算符。
- 三元运算符：作用在三个操作数上的运算符，比如?:。

（6）有些符号可能是一元运算符，也可能是二元运算符。比如指针运算符“*”和乘法运算符“*”使用的是同一个符号，需要根据上下文来确定它具体代表什么运算符。

（7）对于二元运算符，在进行计算时，通常会要求两个操作数具有相同的类型，或者可以转换为同一种类型。比如 7.0+5 在进行计算时，会将整型数值 5 转换为浮点数再进行计算。

2．表达式基本概念

（1）表达式是由运算符（比如+、-、*、/等）、操作数与括号按一定规则组成的。比如表达式 a+2，其中 a 和 2 是操作数，符号+表示“加”。操作数既可以是数值，也可以是变量。

（2）能对表达式进行求值、函数调用和产生副作用，或者这几种的组合。函数调用请参见函数章节。

（3）表达式语句：在表达式后面加上一个分号（分号是 C++中语句的结束标志），就成为表达式语句。

（4）注意：表达式不一定是一条语句，比如 while(a+2){}中的 a+2 就是表达式。

（5）最简单的表达式就是一个操作符，如常量、变量等。

（6）每个表达式都会产生一个结果，若表达式中没有运算符，则结果就是操作数本身的值。

（7）若表达式中有运算符，则表达式的值就是通过运算符运算后获得的。

（8）除特殊情况外，表达式的结果都是右值。也就是说，可以读取表达式的结果，但不能对表达式赋值。

（9）若不知道操作数的类型，则无法确定一个表达式的含义。比如 a+b;，如何计算该表达式完全取决于操作数的类型。

（10）表达式都有类型，其类型是经过计算后所得结果的类型。详见类型转换章节。

（11）若表达式没有得到值，则其类型为 void。

4.2.2 表达式的副作用和顺序点

（1）副作用：是指对执行环境状态的改变（比如对变量值的改变）。如下操作会产生副作用。

- 存取 volatile 对象。
- 修改对象的值。因为变量是命名的对象，因此修改变量的值也会产生副作用。
- 修改文件的内容。
- 调用上述运算或操作的函数。也就是说，只要在函数内没有发生上述三种情况的操作，函数就不会产生副作用。

（2）表达式的副作用：一般可以理解为对作为操作数的变量的值的改变，因此如下一些运算符会产生副作用：自增“++”、自减“--”，以及各种赋值运算符（包括复合赋值运算符）等。比如(b=2)+(b=3)+(b=4);有三个副作用，因为存在三个赋值运算符，它们能改变操作数 b 的值；再比如(b++)+(b++)+(b++);，存在三个副作用，其中三个“++”运算符会改变操作数 b 的值。

（3）序点（顺序点）：序点就是程序中的一个点，当程序执行到这个点时，必须保证对这个点之前的所有副作用都已经完成了。以下情况都属于顺序点。

- 逻辑与“&&”、逻辑或“||”、条件运算符“?:”、逗号运算符的左（或第一）操作数结束处。
- 初始化之后。
- 每个完整的声明处。比如 int a,b;在逗号和分号之前都有一个顺序点。
- 表达式语句中的表达式，说得简单一点就是在分号之前。表达式语句就是在表达式之后加一个分号，比如 a=b+c;是表达式语句，那么表达式语句中的表达式就是“a=b+c”。
- if 或 switch 语句中的控制表达式。
- while 或 do-while 语句中的控制表达式。
- for 语句的三个表达式中的任意一个表达式之后。
- return 语句中的表达式。
- 函数调用时，实参赋值给形参之后，在被调用函数的第一条语句执行之前。
- 函数调用时，返回值之后，在函数以外其他语句执行之前。
- 比如 a=(b+c+d) || (f &&g ? h,i : j);表达式中的顺序点有四个，分别是逻辑或“||”运算符的左操作数(b+c+d)之后；逻辑与“&&”运算符的左操作数 f 之后；条件运算符“?:”的第一个操作数 f&&g 之后；逗号运算符的左操作数 h 之后。在进行计算时，所有的副作用必须在到达顺序点之前完成。

（4）注意：标准只规定了程序必须在顺序点之前已经完成所有的副作用（即对变量的修改），但并未规定在顺序点之前什么时候完成副作用。比如(a++)+(a++)+(a++);只有一个顺序点（即在分号之前），程序可以在每执行一个 a++后就完成一次副作用（即对 a 的修改），也可以在

执行第二个 a++后完成副作用，还可以在第三个 a++处完成副作用，程序只需保证在顺序点之前完成所有的副作用即可，即保证对变量 a 进行三次递增。这时不同的编译器会得到不同的结果。

（5）顺序点和副作用示例（有关++、=运算符，请参阅后文）。

示例 14.1：b=a++;

这里有两个副作用、一个顺序点；顺序点在分号处，而副作用则一个来自赋值运算符，一个来自“++”运算符。这两种运算符都会改变操作数的值。

示例 14.2：int a=1,b; b=(a++)+(a++)+(a++);

这里有四个副作用和一个顺序点；顺序点在分号处，而副作用来自于赋值运算符和“++”运算符。在这个表达式中，在同一个顺序点之前对同一个变量进行了三次修改，即++运算符对变量 a 的修改，因此不同的编译器会得到不同的结果。因为没有规定在顺序点之前什么时候完成副作用（即对 a 的修改），所以编译器只需在顺序点（分号处）之前完成所有的副作用即可。在 VC++ 2010 中，得到值 3，即副作用在分号之前才产生。其他编译器有可能得到值 6，即每执行一次 a++就产生一次副作用（改变 a 的值）。还可能得到值 5，即执行一次(a++)+(a++)后产生两次副作用。同理，对于 b=(++a)+(++a)+(++a);，在 VC++ 2010 中得到值 12，即副作用产生于分号之前。其他编译器可能得到值 9，即每执行一次++a 就产生一次副作用。

示例 14.3：int a, b;　b=(a=2)+(a=3)+(a=4);

这里有四个副作用和一个顺序点，顺序点在分号处，而副作用来自于赋值运算符。此表达式在同一个顺序点（或两个顺序点之间）存在对同一变量修改两次及以上的情况，所以不同的编译器会得到不同的结果。编译器可以在每一次赋值后就完成副作用（即对变量 a 进行赋值），也可以在分号之前完成所有的副作用。在 VC++ 2010 中，得到值 12，即编译器在分号之前才产生副作用。其他编译器可能会得到值 9 或其他值。

（6）综上所述，在两个顺序点之间，不应对同一个变量进行两次以上修改，或者同时有读取和修改的操作，否则行为是未定义的（即不同的编译器会有不同的结果）。也就是说，为了兼容性，程序中不要出现在两个顺序点之间对同一个变量修改两次以上的表达式，比如 a=(b++)+b(++);、i=i++;之类的语句。

4.2.3　运算符的优先级、结合性和操作数的求值顺序

（1）运算符的优先级规定：优先级高的运算符在优先级低的运算符之前先进行计算。比如 a+b*c，因为“*”的优先级高于“+”，因此先计算 b*c，然后再把 b*c 的结果与 a 相加。

（2）小括号可以改变运算符的优先级，当有小括号时，应先计算小括号中的表达式；当有

多个小括号嵌套时，则从内到外进行计算。注意：C++中的方括号“[]”不是用来改变运算符优先级的。比如(a*(b+c))*d，首先计算最内层小括号中的表达式 b+c，然后将 b+c 的结果与 a 相乘，最后再将结果与 d 相乘。

（3）运算符的结合性：由多个相同的运算符组成的表达式，其运算符按照从左到右或从右到左的顺序进行计算。比如 a+b+c 表达式中的“+”运算符，具有左结合性，应从左向右进行计算，因此先计算 a+b，即 a+b+c 与(a+b)+c 是相等的；再比如 a=b=c，因为赋值运算符具有右结合性，应从右向左进行计算，所以 a=b=c 与 a=(b=c)相等。

（4）操作数的求值顺序：优先级和结合性规定了运算符的求值顺序，但没有规定运算符所需操作数的求值顺序。比如对于 a*b+c*d 表达式，因为没有对+、-、*、/等运算符的操作数（即其中的 a, b, c, d）规定求值顺序，所以既可以先计算 a*b，也可以先计算 c*d。当然，对于这个表达式先计算左操作数还是右操作数对结果没有影响，但对于像函数调用或自增、自减、赋值等这样的表达式就可能会产生不同的结果。比如 f()+g()，对于这个表达式，是先调用函数 f()还是 g()对结果就可能会产生不同的影响；再比如(a=b)+(b=c)，很明显，这个表达式先计算左操作数 a=b 还是右操作数 b=c 产生的结果是不同的，大多数编译器都是先计算左边的操作数再计算右边的操作数。针对这种情况，不同的编译器可能会产生不同的值，在编程时应避免出现这样的程序。

4.2.4 运算符性质总结

对运算符我们需要了解其如下性质。

- 对操作数的类型要求。
- 对操作数的数目要求，即一元、二元或三元运算符。
- 操作数的求值顺序，大部分运算符没有对操作数的求值顺序做出规定。
- 运算符计算后的结果类型。
- 运算符计算后的结果是否是左值，大部分为右值。
- 运算符的优先级，查阅优先级表（见表 4.2）即可。
- 运算符的结合性。
- 运算符是否产生副作用。
- 运算符是否有顺序点。大部分运算符都没有顺序点。
- 运算符的运算规则。

4.3　运算符

4.3.1　二元算术运算符

二元算术运算符有：+（加）、-（减）、*（乘）、/（除）、%（求余或求模）。

1. 操作数要求

（1）二元算术运算符要求的操作数为右值。

（2）“乘”“除”运算符的操作数应是算术类型。

（3）“加”运算符的操作数应是算术类型，或者一个操作数是指针，而另一个操作数是整型（包括整数类型、字符型、布尔型）。注意：这一规定说明了两个指针不能相加。

（4）“减”运算符的操作数应是算术类型，或者两个操作数均为相容类型的限定或非限定形式的指针，或者左操作数是指针，右操作数是整型。由此可知，两个指针可以相减。

（5）“%”运算符的操作数应是整型，否则会发生错误。

（6）注意：不能对两个指针进行加、乘、除运算，但可以进行减运算。详见指针章节。

2. 结果及其类型

（1）二元算术运算符返回的结果为右值。

（2）二元算术运算符的结果的数据类型依操作数而定，具体规则详见类型转换章节。

（3）“%”运算符的结果是两数相除后的余数而不是商，比如 7%4 的结果为 3。

（4）若 a/b 两个操作数都是整数，则 (a/b)*b+a%b 表达式的结果应等于 a。

（5）“/”运算符的两个操作数都是整数的话，则结果是整数；若商不是整数，则小数部分被截掉。比如 5/2 的结果虽然为 2.5，但 5 和 2 都是整数，所以小数部分被截掉，结果为 2。

（6）“/”运算符的两个操作数只要有一个是浮点数，则结果就为浮点数。比如 5.0/2 的结果为 2.5。

（7）“/”和“%”，当第二个操作数（被除数）是 0 时，其结果是未定义的，比如 7%0、7/0 等。

（8）使用算术运算符时，产生的结果若超出了其类型的表示范围也是未定义的。

（9）在编写程序时，使用“/”和“%”运算符时，应避免操作数中出现负数，一旦出现负数，就会出现程序的可移植性问题，详见表 4.1。

表 4.1 “/”和“%”运算符的结果

运算符	左操作数	右操作数	结果	舍入方式
/（除）	正数	正数	正数	直接截掉小数部分。比如 5/2=2
	负数	负数	正数	依实现（可向 0 或无穷大取整）
	正数	负数	负数	VC++ 2010 始终向 0 取整 比如–5/–2，因为结果为 2.5，向 0 取整则为 2；若向正无穷大取整，则为 3。同理，–5/2=–2 向 0 取整为–2，向负无穷大取整则为–3
	负数	正数		
%（求模）	正数	正数	正数	
	负数	负数	负数	
	正数	负数	符号依实现（编译器可依分子或分母而定）	
	负数	正数		VC++ 2010 的符号依分母而定 比如–5%2，若符号依分母而定，则结果为-1；若依分子而定，则结果为 1

说明：当操作数有一个为负数时，有些编译器把“/”的舍入方式与“%”的符号取向进行相关联。比如，若求模结果的符号与分子符号相同，则除出来的值向 0 一侧取整；若求模结果的符号与分母符号匹配，则除出来的值向负无穷大一侧取整。

3. 求值顺序及结合性

（1）二元算术运算符未规定操作数的求值顺序。即对于 f()+g()，是先调用 f 还是 g 未做出规定。

（2）二元算术运算符的结合性按照从左到右的顺序执行（即左结合性），即 a+b+c 等于(a+b)+c。

示例 14.4：二元算术运算符

```
#include <iostream>
using namespace std;
int main()
{    int a1=1,a2=2; int *p1=&a1,*p2=&a2;
     //p1+p2;                  //错误，不能对两个指针进行相加
     p1+2;                     //正确，可以将一个指针和一个整数相加
     p1+a1;                    //正确，可以将一个指针和一个整型数值相加
     //p1*p2; p1/p2; p1%p2;    //错误，不能对指针进行乘、除、求模运算
     //2%3.2;                  //错误，求余运算符要求操作数必须是整型的
     cout<<"3%2 ="<<3%2<<endl;       //输出 1，求余的结果是相除后的余数而不是商
     cout<<"5/2 ="<<5/2<<endl;       //输出 2 而不是 2.5，因为两数都是整型的，相除后的结果也是整型的
     cout<<"5.0/2 ="<<5.0/2<<endl;   //输出 2.5，因为 5.0 是浮点数，相除后的结果也是浮点数
     cout<<"-5/-2 ="<<-5/-2<<endl;   //输出 2，某些编译器可能输出 3，若两数都为负的，则除运算的结果
                                     //为正，VC++ 2010 是向 0 取整的
     cout<<"-5%-2 ="<<-5%-2<<endl;   //输出-1，若两数都为负的，则求余运算的结果为负的
```

```
                //除或求余运算符的两个操作数若有一个为负的，则求余结果的符号和除结果的舍入方式
                //取决于实现。编译器求模操作结果的符号可依分子或分母的符号来确定
                //在 VC++ 2010 中，除操作结果的舍入方式始终是向 0 取整的。求模结果的符号是依分母而定的
    cout<<"-5/2 ="<<-5/2<<endl;  //输出-2，在 VC++ 2010 中，除操作的结果始终是向 0 取整的
    cout<<"5/-2 ="<<5/-2<<endl;  //输出-2，在 VC++ 2010 中，除操作的结果始终是向 0 取整的
    cout<<"-5%2 ="<<-5%2<<endl; //输出-1，当有一个操作数为负数时，VC++ 2010 的结果符号依分母而定
    cout<<"5%-2 ="<<5%-2<<endl; }//输出 1，当有一个操作数为负数时，VC++ 2010 的结果符号依分母而定
```

4.3.2　关系运算符

关系运算符有 6 个：>（大于）、<（小于）、>=（大于或等于）、<=（小于或等于）、==（等于）和!=（不等于）。

1. 操作数要求

（1）关系运算符都是二元运算符，即要求有两个操作数。

（2）关系运算符的两个操作数均应为算术类型，或者是指向相容的对象类型的限定或非限定形式的指针类型，或都是指向相容的不完整类型的限定或非限定形式的指针。

（3）注意：关系运算符的操作数条件说明不能将指针和算术类型进行比较，比如 int a=1, b=2; int *p=&b;，则 p>a;将是错误的。

（4）关系运算符要求的操作数为右值。

2. 结果及其类型

（1）关系运算符产生的结果是布尔型。结果若为真，则为 1；结果若为假，则为 0。

（2）关系运算符返回的结果为右值。

3. 结合性及求值顺序

（1）关系运算符的结合性是按照从左到右的顺序执行的，即左结合性。

（2）关系运算符没有规定操作数的求值顺序。

4. 注意事项

（1）C++用于比较两个对象是否相等的运算符是“==”，而不是“=”运算符。在 C++中“=”运算符表示赋值。

（2）C++中的关系运算符最好不要连接使用，比如 a>b>c，此语句会先执行 a>b，其结果值为 0 或者 1，然后再将 0 或者 1 与 c 进行比较，因此 a>b>c 并不是数学意义上的 b 的大小位于 a 和 c 之间。

（3）使用 cout 进行输出时注意：因为在 cout 之后使用的“<<”符号是移位运算符，所以凡是比移位运算符优先级低的运算符，在使用 cout 进行输出时都应加上括号。因为关系运算符的

优先级比“<<”运算符低，所以在使用 cout 输出时，应加上括号，比如 cout<<(a>b)<<endl;，否则将出错；但是 cout<<a>b;不会出错，此语句与(cout<<a)>b 相同。

4.3.3 逻辑运算符

（1）在 C++中所有的非零值均被解释为 true（真），只有 0 才被解释为 false（假）。

（2）逻辑运算符有三个：！（逻辑非）、&&（逻辑与）和|（逻辑或）。

1. 操作数要求

（1）逻辑与“&&”和逻辑或“||”运算符是二元运算符。

（2）逻辑非“!”运算符是一元运算符。

（3）逻辑运算符要求的操作数为右值。

（4）逻辑运算符的操作数是标量类型，这意味着逻辑运算符可以将指针和算术类型进行混合运算。

2. 结果及其类型

（1）逻辑与“&&”表示当且仅当两个操作数均为真时，其结果为真，否则为假。比如 3&&1 的结果为真，3&&0 的结果为假。

（2）逻辑“||”或表示当且仅当两个操作数的值均为假时，其结果为假，否则为真。比如 3||0 的结果为真，0||0 的结果为假。

（3）逻辑“!”非表示若操作数的值为真，则其结果为假；若操作数的值为假，则其结果为真。

（4）逻辑运算符返回的结果为布尔型。

（5）逻辑运算符返回的结果一般为右值。

3. 结合性及求值顺序

（1）逻辑与和逻辑或运算符的结合性是按照从左到右的顺序执行的，即左结合性。

（2）逻辑非运算符的结合性是按照从右到左的顺序执行的，即右结合性。

（3）“&&”和“||”运算符会控制操作数的求值顺序，即后面要讲的短路求值所规定的先左操作数再右操作数；同时在运算符前面有顺序点，在顺序点之前所有的副作用必须完成。比如 int a=0; (a++)||a;，结果为 1。分析：一开始表达式 a++的结果为 0，然后将 0 与右操作数 a 进行逻辑或运算，即 0||a，因为在“||”之前有顺序点，所以 a++的副作用必须在“||”之前完成，也就是 a 的值必须在“||”之前更新。在“||”之前 a 的值更新为 1，然后再与右操作数执行“||”运算，这时右操作数 a 的值为 1，因此(a++)||a 的结果为 1。

（4）短路求值：即逻辑与和逻辑或操作总是先计算左操作数，若左操作数能确定该逻辑表

达式的结果，则不再计算右操作数，只有当左操作数的值不能确定该逻辑表达式的结果时，才计算右操作数。逻辑与“&&”运算符，当左操作数的值为 0（假）时，则不管右操作数为什么值，该逻辑表达式的结果一定为 0，因此不必再计算右操作数，只有当左操作数值为非 0 时才计算右操作数；同理，对于逻辑或“||”运算符，若左操作数的值为非 0（真），则该逻辑表达式的结果一定为 1，因此不必再计算右操作数，只有当左操作数的值为 0 时才计算右操作数。

（5）逻辑非运算符没有规定操作数的求值顺序。

4. 注意事项

使用 cout 进行输出时注意：逻辑运算符“&&”和“||”的优先级比“<<”运算符低，因此在使用 cout 输出时应加上括号，比如 cout<<(a&&b)<<endl;，否则将出错；但是 cout<<a&&b; 不会出错，此语句与(cout<<a)&&b;等同。

示例 14.5：逻辑运算符

```
#include <iostream>
using namespace std;
int main()
{   int a1=1,a2=2;          int *p1=&a1,*p2=&a2
    cout<<(p1&&2)<<endl; //输出 1，逻辑与表示若两个操作数为真，则结果为真。逻辑运算符可以将指针与
                         //算术类型混合运算，同时逻辑运算符在输出时也应加上括号
    a1=0;
    cout<<(a1++||a1)<<endl;//输出 1，顺序点：表达式 a1++的结果为 0，然后将 0 与 a1 进行逻辑或运算，
                           //即 0||a1；但因为在“||”之前有顺序点，因此 a++的副作用必须在“||”
                           //之前完成，也就是 a1 的值必须在“||”之前更新。在“||”之前 a1 的值更
                           //新为 1，然后再与右操作数执行“||”运算，这时 a1 的值为 1，所以
                           //(a1++)||a1 的结果为 1
    a1=1;
    cout<<(a1||(a1=3))<<endl;  //输出 1，短路求值，对于逻辑或“||”运算符，只要左操作数的值为真
                               // （非 0 值），则整个逻辑表达式的值为真，因此不必再计算右操作数
    cout<<"a1 ="<<a1<<endl; } //输出 1，可以看到上句中的 a1=3 没有被执行
```

4.3.4　赋值运算符

赋值运算符即“=”，它不同于数学意义上的等于，而是表示将右操作数的值存储在左操作数所指定的位置，比如 a=2;表示将值 2 赋给左操作数 a 所指定的位置。

1. 操作数要求

（1）赋值运算符是二元运算符。

（2）赋值运算符的左操作数必须是可修改的左值，比如 4=a、a+b=3 都是错误的，a+b 返回的结果是右值；同理，const int c; c=3; 也是错误的，变量 c 是不可修改的左值；int a[3], b=1; a=&b; 错误，数组名是不可修改的左值。

（3）赋值运算符的两个操作数都是指向相容类型的限定或非限定形式的指针，则左操作数所指向的类型应具有右操作数所指向的类型的全部限定词。比如对于 int a=1; const int *p=&a; ，则 int *p1=p; 是错误的，因为 p1 指向的类型没有包括指针 p 指向的类型限定词 const；但是对于 int *const p=&a;，int *p1=p;是正确的，因为 int *const p 指向的类型是 int，而 int 没有被限定词限定，const 限定的是指针 p，表示指针变量 p 是常变量。此条规则应注意理解“指针指向的类型”这句话。

2．结果及其类型

（1）赋值运算符返回的结果就是赋值后左操作数的新值。

（2）赋值运算符返回的结果类型为左操作数的类型。

（3）赋值运算符返回的结果是左值。

（4）当两个操作数的类型不相同时，赋值运算符可能会进行隐式类型转换，以将右操作数的类型转换为左操作数的类型，从而可能会修改被赋的值。比如 int a; a=3.1;，因为 3.1 是浮点型，所以会产生截断操作，这样变量 a 的值其实为 3。

（5）赋值运算符会产生副作用，也就是会修改操作数的值。

3．结合性及求值顺序

（1）赋值运算符的结合性是按照从右到左的顺序执行的，即具有右结合性。比如 a=b=3;，首先将值 3 赋给 b，然后再将 b=3 的结果赋给 a。

（2）赋值运算符没有规定操作数的求值顺序。

4．注意事项

（1）初始化与赋值的区别：初始化是在创建变量的时候给它一个初始值，而赋值则是发生在变量已经创建之后进行的擦除旧值写入新值的操作。

（2）赋值运算符的优先级比“<<”运算符低，在使用 cout 输出时应加上括号。

4.3.5 复合赋值运算符

复合赋值运算符共有 10 个，分别是：+=、-=、*=、/=、%=、<<=、>>=、&=、^=、|=，其中“<<”表示左移，“>>”表示右移，“&”表示按位与，“^”表示按位异或，“|”表示按位或。

1．操作数要求

（1）复合赋值运算符是二元运算符。

（2）复合赋值运算符的操作数类型应与所对应的二元运算符所允许的操作数类型一致。

（3）复合赋值运算符的左操作数必须为可修改的左值。

2．结果及其类型

（1）复合赋值运算符的运算规则，以“+=”为例，则 E1+=E2 与 E1=E1+(E2)等效，其中 E1 表示左操作数，E2 表示右操作数（注意 E2 被括起来了），注意操作数不一定是一个变量，有可能是一个表达式。

（2）复合赋值运算符返回的结果为左值。

（3）复合赋值运算符返回的结果类型是左操作数的类型，返回的结果就是进行相应的计算再赋值后的左操作数的新值。

（4）复合赋值运算符会产生副作用，也就是会修改操作数的值。

3．结合性及求值顺序

（1）复合赋值运算符是按照从右至左的顺序执行的，也就是具有右结合性。

（2）复合赋值运算符没有规定操作数的求值顺序。

4．注意事项

（1）复合赋值运算符与赋值运算符的区别：复合赋值运算符只对左操作数求一次值。比如 a[f()]+=5，假设 f()函数返回 int 类型，而且无副作用，表达式 a[f()]+=5;将只调用一次 f()函数，而 a[f()]=a[f()]+5;将会调用两次 f()函数。

（2）复合赋值运算符总是保证优先计算右操作数，比如 E1+=E2 与 E1=E1+(E2)是等效的，而与 E1=E1+E2 不等效（注意，E2 被小括号括起来）；再比如 a+=5>6 与 a=a+(5>6)等效，而不与 a=a+5>6 等效，因此两个表达式的意义是完全不一样的。

（3）复合赋值运算符的优先级比“<<”运算符低，在使用 cout 输出时应加上括号。

示例 14.6：赋值运算符和复合赋值运算符

```
#include <iostream>
using namespace std;
int main()
{  int a1=1,a2=2,a4; const int a3=3, b[3]={0};
   //规则：赋值运算符和复合赋值运算符的左操作数必须是可修改的左值
   //a3=4;            //错误，a3 是不可修改的左值
   //b=&a3;           //错误，数组名 b 是不可修改的左值
   //a3+a1=3;         //错误，a3+a1 返回的是右值，而不是左值
   //3=4;             //错误，不能对右值进行赋值
   a4=3.2;            //当右操作数的类型与左操作数不相同时，右操作数转换为左操作数的类型，因此 a4
                      //的值为 3，但这并不改变右操作数的值和类型
   (a4=2)=4;          //赋值运算符返回的是左值
   a1=a2=a4=3;        //赋值运算符具有右结合性，即计算此表达式时，首先从 a4=3 开始，然后将 a4=3 的结果
                      //赋值给 a2，最后再赋值给 a1，因此 a1、a2、a4 具有相同的值 3
   cout<<"a4 ="<<a4<<endl;
   //E1+=E2 与 E1=E1+E2 的区别：  E1+=E2 与 E1=E1+(E2)等效，而不等效于 E1=E1+E2，注意括号
```

```
a4=4;
cout<<(a4+=5>6)<<endl;   //输出 4，a4+=5>6 等效于 a4=a4+(5>6)；而不等效于 a4=a4+5>6，因此
                         //结果为 4
a4=4;
cout<<(a4=a4+5>6)<<endl; } //输出 1，首先计算 a4+5 得到 9，然后再执行 9>6 结果为真，值为 1
```

4.3.6 递增和递减运算符

递增运算符是++，递减运算符是--，递增运算符有前缀递增、递减和后缀递增、递减之分，递增或递减运算符位于操作数之前的被称为前缀递增或递减运算符，比如++a；同理，递增或递减运算符位于操作数之后的被称为后缀递增或递减运算符，比如 a++。

1. 操作数要求

（1）递增运算符是一元运算符，只需要一个操作数。

（2）递增、递减运算符（不论前缀还是后缀形式）的操作数必须是可修改的左值。

（3）递增、递减运算符（不论前缀还是后缀形式）的操作数是限定或非限定的标量类型，这意味着可以对指针类型进行运算。

2. 结果及其类型

（1）递增、递减运算符会产生副作用，也就是会修改操作数的值。

（2）后缀递增运算符表达式的结果是其操作数的值，在获得结果后，操作数的值加 1，也可以理解为“先使用后修改”；后缀递减运算符与递增相同，只是值减 1。比如 int a=1, b; b=a++; cout<<a<<b; 输出 21，因为 a++返回的结果是其操作数 a 的值未递增时的 1，然后将 1 赋值给变量 b， b 的值为 1，然后变量 a 的值递增 1，所以 a 的值为 2。

（3）前缀递增运算符是先将操作数的值加 1，表达式的结果是增加后的新值，也可以理解为“先修改后使用”；前缀递减运算符与递增相同，只是值减 1。比如 int a=1, b; b=++a; cout<<a<<b; 输出 22，因为++a 的结果是增加后的新值，即 2，然后再将 2 赋值给 b，所以输出 22。

（4）前缀递增运算符++E 与表达式 E+=1 或 E=E+1 等价，条件是：操作数 E 本身不会产生副作用。递减运算符与递增运算符类似。

（5）前缀递增递减运算符返回的结果是左值，所以前缀操作返回对象本身，其结果是左值。

（6）后缀递增递减运算符返回的结果是右值。比如++(i++); 是错误的，因为后缀递增返回的是右值，而前缀递增要求左值；但是(++i)++是正确的，因为前缀递增返回的是左值；同理，i++++是错误的。注意，++i++与++(i++)相同，因为后缀递增的优先级高于前缀递增。

3. 结合性及求值顺序

前缀递增、递减运算符没有规定操作数的求值顺序。

4．递增、递减运算符的副作用及注意事项

（1）后（前）缀递增、递减运算符更新其操作数存储值的副作用必须在前一个顺序点和下一个顺序点之间发生。示例如下：

int b=0; cout<<(b++||b);，结果将输出 1。此例在逻辑运算符章节中已经讲过。

int a=1, int b; b=(a++)+(a++)+(a++);，此表达式在不同的编译器中将得到不同的值。此例在 4.2.2 节中已经讲过，此处不再重述。

（2）因为副作用只规定必须在顺序点之前完成，没有规定在顺序点之前什么时候完成，而 C++中的运算符大部分都没有顺序点，所以在表达式中应避免出现两个以上产生副作用的表达式，比如(b=2)+(b=3)+(b=4); (++i)+(++i)+(++i);应避免出现。

（3）使用 cout 进行输出时注意：比如 int i=2; cout<<i++<<i++;，很多编译器都输出 32，而不是 23，因为大多数编译器在使用 cout 进行输出时都是按照从右到左进行计算，然后从左到右输出的，所以输出值为 32。

5．对 i+++++j 的理解及词法元素与贪心算法

（1）词法元素也称为单词，是由一个个字符组成的，包括运算符，因此编译器的一个重要任务就是对源程序进行扫描分析，然后识别出一个个词法元素，称为词法分析。这就好比英语单词是由一个个字母组成的，词法元素相当于英语单词。

（2）词法元素一般被分为标识符、关键字、常量、运算符、分隔符。词法元素还有其他分类方法，这里不深讲。

（3）在程序中有些类别的词法元素可以连在一起，中间不需要使用空白符（空格、横向制表符、纵向制表符、新行、换行符等）进行分隔；但两个相邻的标识符之间、标识符与关键字之间、两个相邻的关键字之间、标识符或关键字与其后的常量之间必须使用空白符进行分隔，若不分隔编译器会认为是一个词法元素。

（4）贪心算法：词法元素是可构成词法元素的最长的字符序列（即贪心算法）。比如对于 i+++++j 表达式编译器将把其分析为 i++ ++ +j，按照贪心算法首先读入 i，然后读入+，因为第一个+之后又是一个+，两个+能组成++运算符（即++是一个词法元素），所以第一步分析成 i++；同理，第 3 个和第 4 个+也能组成++运算符，因此它们被分析成++，结果就是 i++ ++ +j；若要让 i+++++j 成为(i++)+(++j)，则应在相应的位置加入空格符，比如 i++ + ++j 与(i++)+(++j)等同。本书不具体讲解编译器怎样进行词法分析。

示例 14.7：递增和递减运算符

```
#include <iostream>
using namespace std;
int main()
{   int a1=1,a2=2; int *p1=&a1,*p2=&a2
```

```
//3++;                                  //错误，递增运算符要求操作数必须是可修改的左值
const int a5=5; int b1[3]={0};
//a5++;                                 //错误，递增运算符要求操作数必须是可修改的左值
//b1++;                                 //错误，递增运算符要求操作数必须是可修改的左值
int a6=1;       int a7;
a7=a6++;     //后缀递增是先使用后修改，因此先将 a6 的值赋给 a7，然后 a6 自增，结果是 a7=1，a6=2
cout<<"a6="<<a6<<",a7="<<a7<<endl; //输出 21
a6=1; a7=++a6;                  //前缀递增是先修改后使用，因此先执行递增，即把 a6 的值增加为 2，
                                //然后将增加后的值赋给 a7，结果是 a7=2，a6=2
cout<<"a6="<<a6<<",a7="<<a7<<endl;  //输出 22
//a7++++;  ++(a7++);                //错误，后缀递增运算符返回的结果是右值
(++a7)++; ++++a7;  (++a7)=1;   //正确，前缀递增运算符返回的结果是左值
//规则：C++标准规定，后（前）缀递增运算符更新操作数的值的副作用在下一个顺序点之前完成
a7=(2+a6++)+(a6++)+(a6++);      //运算后的结果不确定，不同的编译器将得到不同的值
//贪心算法：词法元素是可构成词法元素的最长的字符序列
//a7+++++a7;  a7++ +++a7;//错误
 a7+++ ++a7; a7++ + ++a7;  //正确，只需在适当的位置加上空格即可
```

4.3.7 位运算符

位运算符有：~、&、^、|、<<、>>，分别表示按位求反、按位与、按位异或、按位或、左移、右移。

1. 操作数要求

（1）按位求反“~”运算符是一元运算符。

（2）按位与“&”、按位或“|”、按位异或“^”、移位运算符是二元运算符。

（3）位运算符的操作数应为整型（包括整数类型、字符型、布尔型）。

（4）移位运算的右操作数不能是负数，且该数必须小于（注意：不能等于）左操作数位数的值，否则是未定义的。

（5）若位运算符的操作数是有符号的负数，则怎样进行处理就取决于机器了，因此在不同的计算机上可能会有不同的结果。

2. 结果及其类型

（1）位运算符是对整型值的二进制位进行的操作。比如整型值 8，在计算时会先转换为二进制数 1000，然后再对其中的每一个二进制位进行位运算。

（2）按位求反“~”运算符，对操作数的每一个二进制位进行取反操作，即将 1 置为 0，将 0 置为 1，然后产生一个新值。

（3）按位与“&”运算符，在进行计算时，对两个操作数对应的二进制位逐位进行计算，若两个操作数对应的位都为 1，则操作结果为 1，否则为 0。比如 4&7，假设二进制位为 8 位，则 4 的二进制形式为 0000 0100，7 的二进制形式为 0000 0111，对两个数对应的二进制位逐位

进行与运算，结果为 0000 0100，其十进制值为 4，即 4&7=4。

（4）按位或“|”运算符，在进行计算时，对两个操作数对应的二进制位逐位进行计算，若两个操作数对应的位都为 0，则为 0，否则为 1。比如 4|7，假设二进制位为 8 位，则 4 的二进制形式为 0000 0100，7 的二进制形式为 0000 0111，对两个数对应的二进制位逐位进行或运算，结果为 0000 0111，其十进制值为 7。

（5）按位异或“^”运算符，在进行计算时，对两个操作数对应的二进制位逐位进行计算，若两个操作数对应的位只有一个为 1，则结果为 1，否则为 0。注意：只有一个为 1 时结果才为 1，两个都为 1 或 0 时则结果为 0。比如 4^7，假设二进制位为 8 位，则 4 的二进制形式为 0000 0100，7 的二进制形式为 0000 0111，对两个数对应的二进制位逐位进行异或运算，结果为 0000 0011，其十进制值为 3。

（6）移位运算符，需要两个操作数，右操作数表示需要移动的位数，移位运算符的结果是将左操作数的各个位向右或向左移动右操作数指定的位数，移出去的位被丢弃，从而产生新值。

（7）左移运算符“<<”在右边插入 0 以补充空位，多出的位舍弃，从而产生新值。

（8）右移运算符“>>”，若操作数是无符号数，则从左边开始插入 0；如果操作数是有符号数，则从左边插入 0 还是 1 依实现而定，一般的机器都是插入与符号位相同的值。也就是说，若符号位是 1，则插入 1；若是 0，则插入 0。

（9）位运算符的操作数若是负数，则不同的机器会得到不同的结果，但大多数计算机都是按补码存储负数的，因此在进行位运算时需要先将负数转换为补码。补码就是符号位不变，其他位按位取反，然后再加 1。正数的补码是原码；补码还原为原码时只需再求一次补码即可。

示例 14.8：–4&–7

–4 的原码为：1000 0100；补码为：1111 1100

–7 的原码为：1000 0111；补码为：1111 1001

对补码进行按位与，结果为：1111 1000

结果为负数，再将其还原为原码，结果为 1000 1000，所以–4&–7 的结果为–8。

示例 14.9：–3&7

-3 的原码为：1000 0011；补码为：1111 1101

7 的原码为：0000 0111；补码为：0000 0111（即原码）

对补码进行按位与，结果为：0000 0101

结果为正数，正数不需要转换，因此–3&7 的结果为 5。

3. 结合性及求值顺序

（1）位运算符没有规定操作数的求值顺序。

（2）按位求反“~”运算符是从右向左执行的，即右结合性。

（3）按位与“&”、按位或“|”、按位异或“^”、移位运算符是从左向右执行的，即左结合性。

（4）输出时注意：位运算符“&”、“|”、“^”的优先级低于“<<”运算符，输出时应加括号。

4.3.8 条件运算符

条件运算符“?:”是C++中唯一的一个三元运算符（需要三个操作数）。

1. 操作数要求

（1）条件运算符的第一个操作数应为标量类型（意味着第一个操作数可以是指针）。

（2）条件表达式的操作数都应为右值。

（3）对条件表达式的第二个和第三个操作数的要求如下。

- 二者均为算术类型。
- 二者均为相容的结构或联合类型或类类型。
- 二者均为 void 类型。
- 二者均为指向相容类型的限定或非限定形式的指针。
- 其中一个是指针，另一个是空指针常量。
- 其中一个是指向一个对象或不完整类型的指针，另一个是指向 void 的限定或非限定形式的指针。
- 总之：条件运算符的第二个和第三个操作数必须是相容的类型。
- 比如 int a=1; int *p=&a;，表达式 1?a:p;是错误的，因为第二个和第三个操作数类型不合要求。但是 1?a:*p 是正确的（有关指针的内容请参阅相关章节）。

2. 结果及其类型

（1）条件运算符的运算规则：exp1?exp2:exp3，若 exp1 的结果为真（即非零值），则计算 exp2，整个表达式的值为 exp2 的结果；若 exp1 的结果为假，则计算 exp3，整个表达式的值为 exp3 的结果。注意：不论怎样 exp1 都会被计算，但 exp2 和 exp3 只有一个会被计算。

（2）条件表达式的第二个和第三个操作数若是相同类型的左值，则返回左值，否则为右值。比如 int a,b;，1?a:b;将返回左值，但对于 int a; short b;，1?a:b;将返回右值，因为 a 与 b 的类型不同。

（3）条件表达式的第二个操作数可以是比条件运算符优先级更低的运算符，比如逗号“,”运算符，1?2,3:4;将得到 3。

（4）条件表达式的第三个操作数不应使用比条件运算符优先级更低的逗号运算符，比如

0?2:3,4;，无论怎样这个表达式的结果都为 4，因为首先计算 0?2:3，然后再计算逗号运算符，结果就是逗号运算符计算的值，是最右边的值，所以得到值 4。

（5）注意：条件表达式的第三个操作数若是比条件运算符优先级更低的赋值运算符时，与逗号运算符并不相同，比如 0?2:a=4 不等同于(0?2:a)=4。

（6）条件表达式返回值的类型依第二个和第三个操作数的不同而遵守如下规则。

- 若第二个和第三个操作数都为算术类型，则执行一般算术转换，以使它们拥有公共类型，结果也是该公共类型。
- 若第二个和第三个操作数均为结构或联合类型或类类型，则结果为该结构或联合类型或类类型。
- 若第二个和第三个操作数均为 void 类型，则结果也为 void 类型。
- 若第二个和第三个操作数均为指针，或其中一个是空指针常量，另一个是指针，则结果的类型是指针，该指针指向的类型被两个操作数所指向类型的所有限定词限定。
- 若第二个和第三个操作数都是指向相容类型或相容类型的不同限定形式的指针，则结果是其复合类型。
- 若第二个和第三个操作数均为指针，且其中有一个是空指针常量，则结果是另一个操作数的类型。
- 若第二个和第三个操作数均为指针，且其中有一个是 void 的指针或 void 的某种限定形式的指针，则另一个操作数将转换为 void 的指针类型，且结果也是指向 void 的指针类型。

示例如下（假设均已初始化）：

```
void *p1; int *p2;                1?p1:p2;      //结果类型为: void *
void *p1; const int *p2;          1?p1:p2;      //结果类型为: const void *
int *p1; const int *p2;           1?p1:p2;      //结果类型为: const int *
const void *p1; int *p2;          1?p1:p2;      //结果类型为 const void *
const int *p1;                    1?p1:0;       //结果类型为 const int *
char *p1; int *p2;                1?p1:p2;      //错误，int *和 char *无法相互转换
1?2:3.3                                         //结果类型为 2 和 3.3 的公共类型 double
```

3. 结合性及求值顺序

（1）条件表达式会控制操作数的求值顺序，即在第一个操作数的结尾处（也就是“?”前面）有顺序点。

（2）条件表达式的求值顺序是从右向左执行的。

4. 嵌套条件表达式的分解方法

（1）因为条件运算符“?:”具有右结合性，所以应从最右边的“:”开始分析，找出与其匹

配的“?”。

（2）条件运算符“?:”总是成对出现，因此在分析时若“:”之前是符号“?”，则它们应是一对，是一个表达式，使用括号将表达式括起来；否则，“:”应与最前面的未与“:”匹配的“?”成对，同样也要使用括号括起来，但要注意括号的位置，括号应括住位于“:”与“?”之间的部分，即在“:”之前、“?”之后的部分。因为位于“:”与“?”之间的式子是一个复合表达式，所以应将整个复合表达式括起来。

示例 14.10：0?1?2:3:4?5:6 等价于 0?(1?2:3):(4?5:6)

首先从最右边的“:”开始分析，因为最右边的“:”之前是“?”，所以它们应是一对，然后将它们括起来就成为 0?1?2:3:(4?5:6)。然后再分析数字 3 之后的“:”，因为在这个“:”之前是一个“:”，因此应与这个“:”匹配的“?”应是最前面的未与“:”匹配的“?”成对，因此与 0 之后的“?”匹配，再把位于“:”与“?”之间的部分括起来，就成为 0?(1?2:3):(4?5:6)。

示例 14.11：1?2?3?4:5?6:7:8?9:10:11 等价于 1?(2?(3?4:(5?6:7)):(8?9:10)):11

① 首先从最右边的“:”开始分析，也就是 10 之后的“:”，因为在其之前也有一个“:”，所以应与最前面的未与“:”匹配的“?”成对。这里是数字 1 之后的“?”，因此将它们之间的部分括起来，成为 1?(2?3?4:5?6:7:8?9:10):11。

② 然后再分析 9 之后的“:”，因为 9 之后的“:”的前面就是一个“?”，因此它们应成对，是一个完整的表达式，使用括号将表达式括起来，成为 1?(2?3?4:5?6:7:(8?9:10)):11。

③ 接下来再分析 7 之后的“:”，因为之前是一个“:”，因此应与最前面的未与“:”匹配的“?”成对，所以应与数字 2 之后的“?”匹配，再使用括号将它们之间的部分括起来，成为 1?(2?(3?4:5?6:7):(8?9:10)):11。

④ 6 之后的“:”与其前的“?”匹配，最后成为 1?(2?(3?4:(5?6:7)):(8?9:10)):11。

（3）输出时注意：条件运算符的优先级比“<<”运算符低，在输出时应加上括号。

4.3.9 逗号运算符

（1）逗号表达式是由一组逗号所分隔的表达式，比如 1,2; a,b;。

（2）逗号表达式的结果是最右边表达式的值，其类型也是最右边操作数的类型，比如 1,2;的结果是 2，其类型是 int；1,2.2;的结果是 2.2，类型是浮点型。

（3）若逗号表达式最右边的操作数是左值，则逗号表达式的结果也是左值。

（4）逗号表达式的左操作数求值之后有一个顺序点。比如 a,b;在左操作数 a 之后有一个顺序点。

（5）逗号运算符是所有运算符中优先级最低的。

（6）表达式从左向右进行计算。比如 a,b,c;，首先计算 a，然后是 b，最后是 c。

（7）因为逗号运算符的优先级比“<<”运算符低，因此在输出时应加上括号，比如 cout<<(1,2,3)<<endl。

4.3.10　sizeof 运算符

sizeof 运算符详见第 2 章。

4.4　类型转换

关于类型的省略：一般 short int、long int 等会省略 int；signed 也会省略，比如 short 表示 signed short int；同理，short int 表示 signed short int；关于 long 也是同样的道理。

4.4.1　基础

（1）类型转换：把某一对象、函数或常量的类型显式或隐式地转换成其他类型。

（2）显式类型转换：使用强制转换运算符“()”将某一对象或函数的类型强制转换为另一种类型。关于 C++新增的 4 种强制类型转换方法（如 static_cast<>等）详见相关章节。

（3）隐式（或自动）类型转换：在运算过程中不使用强制转换运算符，而由系统自动改变某一对象或函数的类型。

（4）一般算术转换：需要算术类型操作数的二元运算符在求值过程中，会遇到两个操作数的类型不相同的情况，这时就要求转换操作数的类型，以得到两个操作数的公共类型，该公共类型一般也是结果的类型（关系运算符除外）。

（5）把一个对象或函数转换为相容类型后，其值或其表示并不会发生改变。比如 float a=3.3; int b=a; ，在初始化变量 b 时，会将变量 a 转换为 int 类型，因此 3.3 小数点后的值被丢掉，b 的值为 3，但 a 的值仍为 3.3，其类型仍为 float。

（6）转换的原则是要保证计算值的精度，一般是将较小的类型转换为较大的类型。

4.4.2　各种类型转换

1. 隐式类型转换

以下情况会发生隐式类型转换。

- 在混合类型的表达式中操作数将被转换为相同的类型。

- 在使用表达式初始化某个变量，或将表达式赋值给某个变量时，该表达式将被转换为这个变量的类型。
- 条件运算符“?:”的第二个和第三个操作数（请参阅条件运算符部分内容）。
- 函数调用时也会发生隐式类型转换（请参阅函数章节）。
- 用作 if、while、for、do-while 语句的条件表达式将被转换为 bool 类型，另外，条件运算符“?:”的第一个操作数、逻辑运算符的操作数是条件表达式，会被转换为 bool 类型。

2．赋值时的类型转换

因为赋值运算符或初始化的左操作数类型是不可更改的，因此总是将右操作数转换为左操作数的类型，转换后不影响右操作数的类型和值。比如 float a=3.1; int b; b=a;，将 a 的值赋给 b 时会将 a 小数点后的小数丢掉，但并不改变 a 的值和类型，即 a 仍为 float 类型，其值为 3.1。

3．一般算术转换规则

以下规则只适用于算术类型，对于指针等其他类型不适用，如图 4.1 所示。

（1）若一个操作数的类型是 long double，则另一个操作数将转换为 long double。

（2）若一个操作数的类型是 double，则另一个操作数将转换为 double。

（3）若一个操作数的类型是 float，则另一个操作数将转换为 float。

（4）否则，说明两个操作数都是整型，对两个操作数进行整型提升（见下文），然后使用如下规则。

- 若一个操作数的类型是 unsigned long int，则另一个操作数也转换为 unsigned long int。
- 若一个操作数的类型是 long int，另一个操作数的类型是 unsigned int；如果 long int 可以表示 unsigned int 的所有值，则将 unsigned int 的操作数转换为 long int；如果 long int 不能表示 unsigned int 的所有值，则两个操作数都转换为 unsigned long int。在 32 位机器上 long 和 int 一般都是 32 位长度，因此其操作数都应转换为 unsigned long 。

（5）否则，说明两个操作数只有一个为 long int 或 unsigned int 或两种类型都不是，因此执行以下操作。

- 若一个操作数的类型是 long int，则另一个操作数将转换为 long int。
- 若一个操作数的类型是 unsigned int，则另一个操作数将转换为 unsigned int。
- 否则，两个操作数的类型都为 int。

（6）可见，如果表达式的操作数分别为整型和浮点型的话，那么整型的操作数会被转换为浮点型。

（7）bool 类型的 false 将被转换为 0，true 被转换为 1；相反，任何非零算术类型的值都被转换为 true，只有 0 值才被转换为 false。

（8）注意：即使两个操作数都是 float，传统 C 语言也总是将 float 提升为 double。

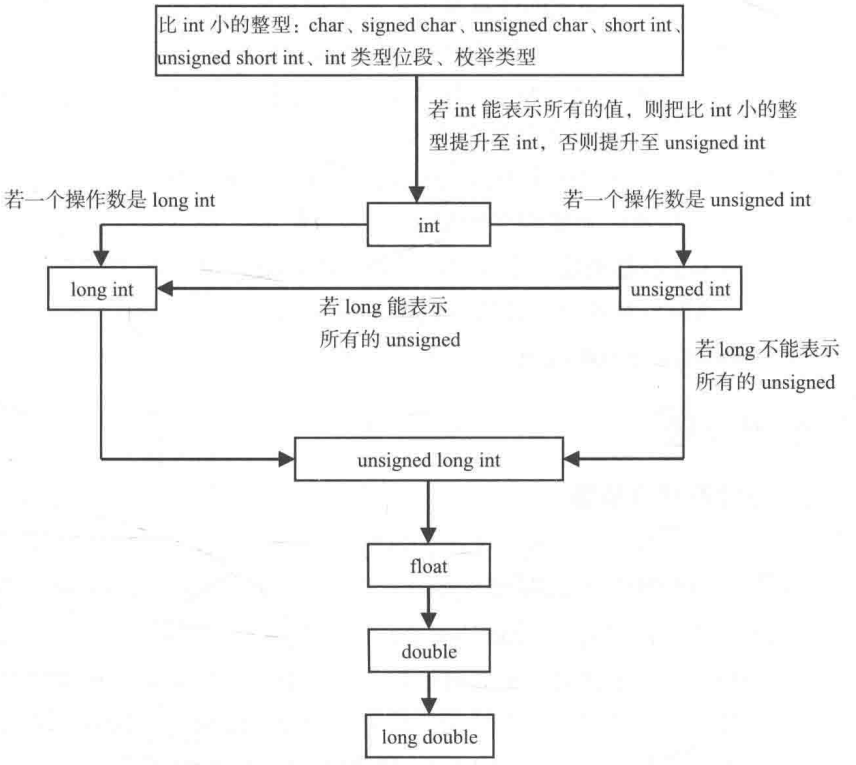


图 4.1　一般算术转换规则图解

4．整型提升

（1）所有比 int 小的整型，包括 char、signed char、unsigned char、short int、unsigned short int、int 类型位段、枚举类型，若这些类型的所有原始值都能被 int 类型所表示，则将它们提升为 int 类型；若 int 不能表示这些类型的值，则被提升为 unsigned int。比如 short 类型比 int 类型短，则 unsigned short int 将被转换为 int；若两种类型的长度相同（指占据的二进制位数相同），则 unsigned short int 将被转换为 unsigned int，因为 int 不能表示所有的 unsigned short int 的值。这种转换保持了待提升类型的值（包括其符号在内），也被称为保值提升。C++使用的是保值提升模式。

（2）注意：只要比 int 小的整型都会自动执行整型提升，但与 int 相等或更大的整型不会执行整型提升。

（3）保无符号提升模式，即总是把 unsigned char 和 unsigned short 提升为 unsigned int。这种提升模式与保值提升有一点差别，但有可能出现一些可移植性问题。某些编译器可能会使用这

种保无符号提升模式。

（4）保值提升和保无符号提升模式的注意事项：在采用保值提升模式进行整型提升后，会得到正确的负数。假设 int 为 32 位，short 为 16 位，则 unsigned short int 在进行计算时将会被整型提升为 int；比如 unsigned short int a=1,b=2;，那么 a–b;将得到正确的值–1，因为 unsigned short int 类型的 a 和 b 在进行相减运算时都被转换为有符号的 int 类型，所以结果是正确的值。若采用保无符号提升模式，则 a 和 b 将被转换为 unsigned int，这样 a–b 将得不到正确的值–1，而是一个溢出后的很大的正数。注意：如果两个数都是 unsigned int，则不会进行整型提升。比如 unsigned int a=1,b=2;，则 a–b;的结果是一个溢出后的很大的正数。

（5）按照所占据的二进制位数和有无符号，整型从最小到最大的次序是：bool、char、signed char、unsigned char、short、unsigned short、int、unsigned int、long、unsigned long。

4.4.3 转换溢出处理

1. 转换为无符号或有符号整型

（1）当将一个负整数值转换为相同或更大的无符号整数时，转换之后的结果应是原始值一次或多次加上该无符号类型的最大值加 1，直到结果在无符号类型的范围内。比如 unsigned long 和 long，若 long 是负数，则 long 转换为 unsigned long 时会遵守以上规则。

（2）若将某个值降格为较小的无符号整数时，如果被转换的数超过无符号类型的范围，则转换之后的结果应是原始值一次或多次减去该无符号类型的最大值加 1，直到结果在无符号类型的范围内。比如将一个更大范围的 unsigned long 赋值给 unsigned int 类型时，会做出以上情况处理。

（3）当一个整数值降格为较小的有符号整数，或将无符号整数转换为相应的有符号整数时，若不能表示其结果，则结果是未定义的。大多数编译器采用的是丢弃原始二进制表示时的最高位，新的类型使用低位。

2. 浮点型与整型之间的转换

（1）浮点型转换为整型时，小数部分将被忽略（注意不会进行四舍五入处理，而是直接丢弃小数部分），若整数部分不能用该整型表示，则行为是未定义的。

（2）浮点型转换为无符号整型时，不一定需要执行整型转换为无符号整型时所需的求模运算，因此，浮点型转换为整型的范围是 0 至无符号类型的最大值。

（3）整型转换为浮点型时，若要转换的值在浮点型所能表示的范围内而又不能确切表示时，结果将是最接近的或者稍大或者稍小的值，由实现定义。比如 float 的精度只有 6 位，若把超过这个精度的整数赋值给 float 类型的变量时，就只能使用近似值，但是怎样取近似值，则是由实现定义的。

（4）将较大的浮点型转换为较小的浮点型时（例如 double 转换为 float），若值超出了目标类型的取值范围，则结果是不确定的。

4.4.4　强制类型转换运算符

这里只介绍 C 语言中的强制类型转换运算符，即使用“()”进行强制类型转换，C++中新增的 4 种强制类型转换方法（如 static_cast<>等）请参阅相关内容。

1．强制类型转换的两种格式

（1）（类型名）表达式。

这是 C 风格的强制类型转换格式，表示把表达式的计算结果强制转换为“()”中指定的类型。比如(int)3.2;、(int)b;、(int)(3+2.5);等，表示把表达式结果强制转换为 int 类型。

（2）类型名（表达式）。

① 这是 C++特有的强制类型转换格式，表示将表达式强制转换为类型名所指定的类型。

② 注意：对于 int(a);这样的单独语句，会被 C++理解为在声明一个变量，从而达不到强制转换的目的；但 int(a)+1;是正确的。像 int(a)这样的单独语句最好不要出现在程序中。

③ 该转换方法只适用于类型简单的名字（比如 float、int 等），比如 float*(p1)+1;试图将指针 p1 强制转换为 float *类型，是非法的。

2．强制类型转换的注意事项

（1）类型名应是限定或非限定形式的标量类型，操作数也应为标量类型。因此像类类型、结构类型这样的复合类型不能作为类型名来转换，也就是说，不能将一种结构或类类型转换为另一种结构或类类型。比如 class A{}; class B{}; A ma; B mb; (A)mb; 是错误的，不能将类型 B 转换为类型 A。

（2）强制类型转换不会修改表达式或变量本身的值和类型。比如 float a=3.2; ，(int)a;不会改变 a 的值和类型，a 的值仍为 3.2，类型仍是 float，但(int)a;的结果是 int 类型。

（3）强制类型转换运算符的结果不是左值，比如 float a=1.1;，则((int)a)=1; ((int)a)++;是错误的。

（4）强制转换为限定类型与强制转换为非限定形式的类型具有相同的效果，对于这种情况不同的编译器可能会有不同的处理，详见第 5 章中的 switch 语句。

运算符及优先级总结如表 4.2 所示。

表 4.2　运算符及优先级总结

等　级	运算符	名　称	操作数	示　例	操作数类型	左/右值	结合性	顺序点
1	::							

续表

等　级	运算符	名　称	操作数	示　例	操作数类型	左/右值	结合性	顺序点
2	.	成员选择		ob.member			从左至右	
	->	成员选择		p->member			从左至右	
	[]	下标		a[expr]			从左至右	
	()	函数调用		f(expr_list)			从左至右	
	()	类型构造		type(expr_list)				
	++	后缀递增	一元	lexp++	标量类型	右值	从左至右	否
	--	后缀递减	一元	Lexp--	标量类型	右值	从左至右	否
3	sizeof					右值	从右至左	否
	()	类型转换		（类型）rexp		右值	从右至左	否
	new	创建对象		new type				
	delete	释放对象		delete exp				
	delete []	释放数组		delete []exp				
	++	前缀递增	一元	++lexp	标量类型	左值	从右至左	否
	--	前缀递减	一元	--lexp	标量类型	左值	从右至左	否
	~	按位求反	一元	~rexp	整型	右值	从右至左	否
	!	逻辑非	一元	!rexp	标量类型	右值	从右至左	否
	-	一元负号	一元	-rexp	算术类型	右值	从右至左	否
	+	一元正号	一元	+rexp	算术类型	右值	从右至左	否
	*	解引用	一元	*rexp	指针类型	左值	从右至左	否
	&	取地址	一元	&lexp	见后文	右值	从右至左	否
4	->*、.*				见后文			
5	*、/	乘、除	二元		算术类型	右值	从左至右	否
	%	求模（余）	二元	rexp%rexp	整型	右值	从左至右	否
6	+	加法	二元	rexp+rexp	参见正文	右值	从左至右	否
	-	减法	二元	Rexp-rexp	参见正文	右值	从左至右	否
7	<<	左移	二元	rexp<<rexp	整型	右值	从左至右	否
	>>	右移	二元	rexp>>rexp	整型	右值	从左至右	否
8	<	小于	二元	rexp<rexp	参见正文	右值	从左至右	否
	<=	小于或等于	二元	rexp<=rexp		右值	从左至右	否
	>	大于	二元	rexp>rexp		右值	从左至右	否

续表

等　级	运算符	名　称	操作数	示　例	操作数类型	左/右值	结合性	顺序点
8	>=	大于或等于	二元	rexp>=rexp		右值	从左至右	否
9	==	等于	二元	rexp==rexp	参见正文	右值	从左至右	否
	!=	不等于	二元	rexp!=rexp		右值	从左至右	否
10	&	按位与	二元	rexp&rexp	整型	右值	从左至右	否
11	^	按位异或	二元	rexp^rexp	整型	右值	从左至右	否
12	|	按位或	二元	rexp | rexp	整型	右值	从左至右	否
13	&&	逻辑与	二元	rexp&&rexp	标量类型	右值	从左至右	是
14	||	逻辑或	二元	rexp || rexp	标量类型	右值	从左至右	是
15	?:	条件运算符	三元	rexp?rexp : rexp	参见正文	依情况	从右至左	是
16	=、+=、-=、*=、/=、%=、<<=、>>=、&=、|=、^=	赋值和复合赋值运算符	二元	lexp = rexp lexp+=rexp lexp<<=rexp	参见正文	左值	从右至左	否
17	throw	抛出异常						
18	,	逗号运算符		exp,exp	参见正文	依情况	从左至右	是

第 5 章

选择语句和循环语句专题

5.1 语句概念

1. 重要概念

（1）常量表达式：就是可以在编译（翻译）时求值而不用等到运行时求值的表达式。常量表达式中不应含有赋值运算符、递增运算符、递减运算符、函数调用、逗号运算符。从概念上可以看出，若表达式中有变量，则不是常量表达式。比如 int a=1;，则 a+2 不是常量表达式。

（2）整型常量表达式：要求其操作数只能是整数常量、枚举常量、字符常量、将浮点型常量（不包括浮点型常变量）强制转换为整型的表达式、sizeof 表达式。若表达式中有浮点数、变量，则都不是整型常量表达式。比如 3+5、int(3.3)+5;是整型常量表达式；3.3+5 不是整型常量表达式；float i=1.1;，则 int(i+1);、int(i)+1;都不是整型常量表达式，因为 i 是浮点型变量，并不是浮点型常量。注意：sizeof 运算符的结果是整型常量，因此 sizeof 表达式可以用作整型常量表达式的操作数。

（3）整型常量表达式的注意事项。

① 使用常量表达式初始化的整型 const 变量可作为整型常量表达式使用，但使用非常量表达式初始化的整型 const 变量，则不能作为整型常量表达式使用。比如 const int a=1;，a 是整型常量表达式；int b=1;const int c=b;，c 不是整型常量表达式。

② 因为整型常量表达式同时也是常量表达式，所以整型常量表达式在编译时就能求出其值，而不用等到运行时才能求出其值。比如 const int a=f();，a 不是整型常量表达式，因为 a 的值必须在调用 f 之后才能确定，而调用 f 是在运行时而不是在编译时。

（4）标量类型、聚合类型、算术类型、不完整类型、对象的概念等见第 4 章。

2. 语句基本概念

（1）一条语句就是一条计算机指令，语句是 C++程序中最小的单位，函数的函数体一般是

由多条语句组成的。

（2）在 C++中一条语句一般以分号作为结束标志，但复合语句是个例外。同时语句表示指定一个行为且不是#include 指令的部分。

（3）注意：语句的分号应在英文状态下输入，比如分号“;”是正确的，而分号“；”是错误的。

（4）除复合语句外，在 C++中没有分号的语句是错误的。比如 int f(){return 0}是错误的，应在数字 0 后加上分号，如 return 0;才表示一条完整的语句。

（5）除特别说明外，程序是按照语句书写的顺序来执行的。

（6）在 C++中可以将一条语句放在一行或者多行上，即在 C++中可以使用空格或制表符的地方都能使用回车。注意：这里是说一条语句，而不是说一个标识符或关键字可以写在多行上，如果要将一个标识符或关键字分开写在多行上，则应在上一行的行尾使用“\”符号，且在“\”之后不能有任何其他字符（包括注释），在下一行的开头也不能有任何其他字符（包括空白符），而语句则不同，语句分开写在多行上的时候在上一行的行尾不必使用“\”符号，而且在其后还可以有注释，在下一行也可以使用空白符。示例如下：

```
in\   //将关键字 int 写在多行上，“\”后面不能有任何其他字符，下一行的开头也不能有任何其他字符
t a=  //把初始化语句 a=1 写在两行上，在行尾不必使用“\”符号，且两行间可以有空白符和注释
1;
```

（7）C++中的语句有：空语句、带标号语句、复合语句、表达式语句、选择语句、循环语句、跳转语句。

（8）空语句：即只有一个分号的语句。空语句不执行任何操作。空语句一般用在循环语句、条件语句的后面，比如 while(...);表示循环语句什么也不执行。

（9）复合语句：也称为块，使用大括号{}，将零条或多条语句写在其中。在大括号后不必加上分号，比如{int a; int b;}就是一条复合语句，其中有两条语句，分别是 int a;和 int b;。当然，大括号中也可以是空的、没有任何语句的块，被称为空块。

（10）复合语句一般用在如 if、while 等需要执行多条语句的地方。

（11）表达式语句：就是以分号结束的表达式。关于表达式请参见第 4 章。

（12）注意：复合语句不以分号结束，而以右大括号“}”结束。

（13）注意：在复合语句中声明的标识符或名字，只在该复合语句中可访问。详见函数章节。

5.2　if 语句

（1）if 语句的作用是根据控制表达式（或称为条件表达式）的值有选择地执行语句或语句块。

（2）if-else 结构一般被视为一条语句，本文为了方便都称为 if 语句。

1. if 语句形式一

```
if(条件表达式) 语句1; 语句2;
```

运行原理：若条件表达式的结果为真，则执行语句 1，然后继续执行语句 2；否则跳过语句 1，直接执行语句 2。比如 if(a>1)cout<<"A"; cout<<"B";，若 a>1 为真则输出 A，否则输出 B。

2. if 语句形式二（if-else 结构）

```
if(条件表达式) 语句1; else 语句2; 语句3;
```

运行原理：若条件表达式的结果为真，则执行语句 1，然后跳过语句 2，执行语句 3；否则执行 else 后的语句 2，然后接着执行语句 3。比如 if(a>1) cout<<"A"; else cout<<"B"; cout<<"C"; ，若 a>1 为真，则输出 AC；若 a>1 为假，则输出 BC。

3. if 语句形式三（嵌套形式）

```
if（条件表达式1） 语句1;
else if（条件表达式2）语句2;
else if（条件表达式3）语句3;
...
else 语句n;   //else语句可省略
```

（1）形式三其实就是嵌套的 if-else 结构，只是 else 后面又是一个 if-else 语句而已。若加上大括号或者将新的 if 语句写在另一行就很明显。

（2）形式三加上大括号后的形式如下：

```
if(条件表达式1) 语句1;
else {if(条件表达式2)语句2;
else {if(条件表达式3)语句3;
else 语句4; }}
```

（3）形式三将新的 if 语句写在另一行的形式如下：

```
if(条件表达式1) 语句1;
else
if(条件表达式2)语句2;
else
if(条件表达式3)语句3;
...
else 语句n;
```

运行原理：

- 若条件表达式 1 的结果为真，则执行语句 1，然后执行整个 if 语句后的下一条语句，即直接跳过语句 2、语句 3、…、语句 *n*。
- 若条件表达式 1 的结果为假，条件表达式 2 的结果为真，则执行语句 2，然后执行 if 语句后的下一条语句。
- 若条件表达式 1 和条件表达式 2 的结果都为假，条件表达式 3 的结果为真，则执行语

句 3，然后执行 if 语句后的下一条语句。

- 若所有条件表达式的结果都为假，则执行 else 之后的语句 *n*，然后执行 if 语句后的下一条语句；若没有 else 语句，则 if 语句什么也不做。
- 若条件表达式 1、条件表达式 2、条件表达式 3、…、条件表达式 *n* 同时有多个为真，则只执行最先为真的表达式后的语句，而不会再执行后面的条件表达式及其后的语句。

示例 5.1：if(a++) cout<<"A"; else if(a++) cout<<"B"; else if(a++) cout<<"C"; cout<<"D";

- 若第一个 a++为真，则输出 AD。
- 若第一个 a++为假，第二个 a++为真，则输出 BD。
- 若三个 a++的结果都为真，则输出 AD，同时 a 只自增一次，第二个和第三个 a++不会被执行；假设自增前 a 的值为 1，则执行 if 语句后 a 的值为 2。
- 若第一个 a++的结果为假，第二个和第三个 a++的值为真，则输出 BD，同时 a++自增两次，即第一个和第二个 a++，但第三个 a++不会执行；假设自增前 a 的值为 0，则执行 if 语句后 a 的值为 2。

4．if 语句注意事项

（1）条件表达式就是指普通表达式。

（2）条件表达式应为标量类型。类类型等聚合类型一般不能用于控制表达式（I/O 类型是个例外）。注意：类对象的成员不一定是聚合类型。比如 class A{}; A ma; if(ma)...;是错误的，ma 是聚合类型。

（3）可以在条件表达式中定义变量，但变量必须初始化，变量应为标量类型，然后将初始化后的变量的值转换为 bool 值，并将该值作为条件表达式的值，以判断结果是否为真。比如 if(int a)...;是错误的，定义的变量必须初始化；if(int a=1) ...; 是正确的。

（4）if 语句形式中的语句可以是复合语句、独立的语句、空语句（空语句表示什么也不做）。若是复合语句，则执行复合语句中的所有语句。比如 if(a>1){语句 1; 语句 2;} else ; {语句 5; }，若表达式 a>1 的结果为真，则执行语句 1 和语句 2，然后执行语句 5；若 a>1 的结果为假，则执行 else 后的空语句，然后执行语句 5。

（5）当程序中出现多余的 else 时，该 else 被称为悬垂的 else。这是一种错误，为了避免出现这种情况，在书写时最好使用缩进形式。

（6）if 语句只会执行满足条件后的第一条语句，而不会执行多条语句，若要让 if 执行多条语句，则应使用大括号将这些语句括起来组成一个语句块。

示例 5.2：if(a>1) 语句 1;语句 2; else 语句 3; 语句 4;

这是错误的，因为 else 找不到匹配的 if，程序将“if(a>1) 语句 1”看成一条 if 语句，然后

执行语句 2，接下来就发现一个独立的 else，从而出现错误。若要使 a>1 为真时执行语句 1 和语句 2，为假时执行语句 3 和语句 4，则应使用大括号将它们括起来，比如 if(a>1){语句 1; 语句 2} else {语句 3; 语句 4;}。

示例 5.3：if (a>1) 语句 1; 语句 2; 与 if(a>1){语句 1; 语句 2;}

若 a>1 为真，则第一个和第二个 if 语句都会执行语句 1 和语句 2；若 a>1 为假，则第一个 if 语句会跳过语句 1，但会执行语句 2，而第二个 if 语句的语句 1 和语句 2 都不会被执行。

（7）当多个 if 语句嵌套时，else 与最近的 if 相匹配。

示例 5.4：if(a>1) 语句 1; if(a>2) 语句 2; else 语句 3; 语句 4;

else 与最近的 if(a>2)相匹配，而不是与 if(a>1)相匹配。若 a>1 的结果为假，则会直接跳过第二个 if-else 结构，而执行语句 4；要执行语句 3，只有当 a>1 为真，且 a>2 为假时。

5.3 switch 语句

1. switch 语句形式

```
switch (控制表达式)
{case 整型常量表达式 1:语句 1;[break;]   //大括号[]中的内容是可选项
case 整型常量表达式 2:语句 2;[break;]
...
case 整型常量表达式 n:语句 n;[break;]
[default:缺省处理语句;  break;]}
```

（1）switch 工作原理：将 switch 中控制表达式的值与 case 后的整型常量表达式的值相比较，如果相等则执行后面的语句，直到遇到 break，如果没有 break，则会一直向下执行，直到 switch 语句结束；如果控制表达式的值与所有 case 后的整型常量表达式的值都不相等，则执行 default 后的语句，若没有 default，则程序什么也不做。比如，控制表达式的值与 case 后的整型常量表达式 2 的值相匹配，则执行语句 2，若没有 break，则继续向下执行语句 3、语句 4、…、语句 n、default 后的语句，直到 switch 语句结束。注意：在没遇到 break 时，switch 会一直向下执行。

（2）case 标号、case 常量：case 标号指 case 和其后的整型常量表达式的组合。case 后的整型常量表达式被称为 case 常量。

2. switch 基本语法规则

（1）switch 语句一般都写在一对大括号之内。本文不讨论不写在大括号内的情况，因为容易出错，所以程序中应避免出现这样的语句。

（2）可以把 default 标号放到任何位置（但不建议这样做，本文以后的内容都不讨论 default 标号不出现在所有 case 标号之后的情况），此时 default 同样是当所有的 case 都不匹配时就会执

行其后的语句，同时 default 的位置并不影响 switch 语句的执行规则，因此程序同样是在没遇到 break 时会继续向下执行，直到整个 switch 结束。比如 switch(4){case 2: 语句 2; default: 语句 3; case 3: 语句 4;}”，因为 4 与 case 后的 2 和 3 都不匹配，所以执行 default 后的语句 3；因为在 default 后没有 break;，所以执行语句 3 之后会继续执行语句 4，直到 switch 语句结束。

（3）一般都需要在 case 后面使用 break;语句来达到结束 switch 的目的，若不加上 break;语句，容易得到意料之外的结果。其语法没有错，但结果并不是想要的，因此最好在每一个 case 标号后都加上 break;语句。

（4）case 整型常量表达式后的冒号“:”不能省略，default 之后的冒号“:”也不能省略。

（5）case 后必须要有一个整型常量表达式，也就是说，“case :”这样的语句是错误的。

（6）注意：case 和其后的整形常量表达式之间有一个空格，不然会出错。

（7）case 标号后可以是空语句，比如 case 1: ;，注意冒号后有个分号。

（8）每个 case 分支都能包含任意条语句，在 case 分支内可以不使用大括号。但为了避免出错，case 后的语句最好是使用大括号。

（9）case 标号后也可以没有任何语句，但是若没有 default 的话，则最后一个 case 后面必须要有一条语句；若有 default（注意，default 标号必须在所有的 case 之后出现），这时所有的 case 后都可以没有语句，但 default 后面不能没有语句。比如 switch(1){case 0: case 1: 语句 1; case 2: case 3: 语句 2;}，最后的 case 3 之后必须有一条语句，否则出错；再比如 switch (1){case 0: case 1: default: 语句 1;}，default 之后必须有一条语句，但所有的 case 后都可以没有语句。

（10）default 是可选的。一条 switch 语句最多只能有一个 default 标号。

（11）同一条 switch 语句中的两个 case 整型常量表达式在转换之后不能具有相同的值。比如 switch(4){case 2: cout<<"A"; case 2: cout<<"B";} 是错误的，case 后的两个表达式的值相同。

（12）case 常量必须是单个的值，不能为 case 标号指定多个值，也不能为它定义一个范围值。若要为 case 标号定义一个范围值，则可以在 case 标号后不使用任何语句来达到这一目的。比如 switch (控制表达式){case 0: case 1: case 2: case 3: 语句 3; break; case4:语句 4;}只要控制表达式的值小于等于 3 都会执行语句 3。

示例 5.5：switch 基本语法规则

```
switch(2)                    //此段程序输出 ABC
{case 4:;                    //正确，case 之后可以是空语句
//case 5                     //错误，省略了 case 标号后的冒号“:”。
//case6:                     //错误，在 case 和其后的整型常量表达式之间需要有空格
//case :                     //错误，case 后必须要有一个整型常量表达式
//case 2+2: ;                //错误，switch 语句中的两个 case 整型常量表达式不能有相同的值
//case 9,10:                 //错误，不能为 case 标号指定多个值
case 0: cout<<"a"<<endl; cout<<"b"<<endl;  //正确，case 分支可包含任意条语句
case 2: cout<<"A"<<endl;  //控制表达式与 case 2 匹配，首先输出 A，因为无 break;，因此向下执行
```

```
default: cout<<"B"<<endl; //输出B，可以把default标号放到任何位置，但不建议这样做
//default:;              //错误，一条switch语句最多只能有一个default标号
case 1: {cout<<"C"<<endl; break;} //输出C，因为有break;，因此程序跳出当前switch语句
case 7:case 8:          //正确，case标号后可以没有任何语句
case 3: cout<<"D"<<endl; //未输出D，因为在case 1就遇到break;而跳出当前switch语句了
//case 12: //错误，最后一个case后面必须要有一条语句
}
```

3. 控制表达式和 case 后的整型常量表达式

（1）控制表达式应是一个能产生整型结果的表达式，控制表达式不会将结果转换为整型。也就是说，控制表达式的结果不能是浮点数等非整型。比如 3.3+5 是错误的；int(3.3+5)是正确的；a+2;，若 a 是 int 类型则是正确的。注意：整型包括整数类型、字符型、布尔型、枚举等。

（2）case 后的表达式必须是整型常量表达式。比如 3+5;是正确的，而 3.3+5 则是错误的。

（3）case 后的整型常量表达式与控制表达式的主要区别：控制表达式只要求结果类型是整型即可，因此控制表达式可以有整型变量，可以在控制表达式中定义变量；而 case 后要求是整型常量表达式，因此 case 后的表达式中不能有变量存在，不能在 case 后的表达式中出现赋值、自增、自减、逗号运算符等，也不能有函数调用。

示例 5.6：switch 控制表达式的语法要求

```
int i=1; float k=3;
//switch(2.2+5){case 1:;}              //错误，控制表达式2.2+5不能产生整型结果
//switch(k+1){case 1:;}                //错误，控制表达式的结果不是整型
switch(int(2.2+5)) {case 1:;}          //正确，把控制表达式的结果强制类型转换为整型
switch(i+1){case 3:;}                  //正确，变量i是整型，控制表达式的结果是整型
switch(int(k+1)){case 2:;}             //正确，将表达式的结果强制类型转换为整型
switch(int i=4){case 4:;}              //正确，可以在控制表达式中定义变量
```

示例 5.7：case 后的整型常量表达式

```
int i=1; const int j=2; float k=3; const float m=4; const int n=i;
switch(2)
{    //case k:; case k+1:;          //错误，浮点型变量k及表达式k+1不是整型常量表达式
     //case i:;  case i+1: ;        //错误，表达式中有变量i，因此不是整型常量表达式
     //case m:;                     //错误，浮点型常变量不是整型常量表达式
     //case 3.3+3:;                 //错误，浮点数不能用作整型常量表达式
     //case n:;//错误，使用非常量表达式初始化的整型const变量，不能作为整型常量表达式使用
     //case int a=4:;               //错误，不可在case后定义变量
     //case int(k+5):;//错误，将浮点型常量（不包括浮点型常变量）强制类型转换为整型表达式是
                                    //整型常量表达式，k是浮点型变量，不是浮点型常量
     //case (int)k+5:;              //错误，原因同上
     //case (int)m+1:;              //错误，原因同上，m是浮点型常变量
     case int(3.3+5):cout<<"K"<<endl;      //正确，原因同上
     case (int)3.3+7:cout<<"L"<<endl;      //正确，原因同上
     case j:;        //正确，使用常量表达式初始化的整型const变量可作为整型常量表达式使用
     case j+1:cout<<"J"<<endl; }           //正确，原因同上
```

4. switch 语句的跳跃

switch 语句会直接跳跃到与控制表达式相对应的 case 语句处，即使 case 位于 if 或循环语句内，然后从相应的 case 处执行其后的语句（嵌套的 switch 语句除外），而不管 if 或循环语句的条件表达式是否为真。如果 if 之后有 else 语句，程序会跳过包含此 case 的所有与 if 相关的 else 语句；如果 case 在 for 循环语句头之后，则程序执行完 case 语句后，会继续按 for 的规则执行它的条件表达式 3 和条件表达式 2，但不会执行 for 的条件表达式 1。示例如下：

```
switch(2){case 1: {  //程序输出 AB
                  if(0){case 2: //不管 if 的条件是真还是假都会跳跃到 case 2 处开始执行程序
                              cout<<"A"<<endl; }
              cout<<"B"<<endl;} }

switch(2){ case 1: {  //程序输出 AC
                  if(0){case 2: //不管 if 的条件是真还是假都会跳跃到 case 2 处开始执行程序
                       cout<<"A"<<endl; }
                  else if(2) {cout<<"B"<<endl;}//包含 case 2 的 if 所关联的 else 语句会被跳过
              cout<<"C"<<endl;}}

int i=2;   //i 应定义于此处
switch(2) { case 1: {   //程序输出 ABAC
     for(i=0;i;i--)
         {cout<<"B"<<endl;
case 2:   //不管 for 如何都会跳跃到 case 2 处开始执行程序，然后继续按 for 的规则执行
         //它的条件表达式 3（即 i--）和条件表达式 2（即 i）
         //但不会执行它的条件表达式 1（即 i=0）
         cout<<"A"<<endl; }
    cout<<"C"<<endl;}}

switch(2){case 1: {  //无输出
         switch(0){case 2:   //嵌套在 switch(0) 中的 case 2 与内层的 switch(0) 相匹配，case 2
                             //不会与外层的 switch(2) 匹配，因此程序不会从外层的 switch 跳跃到
                             //内层的 switch 中的 case 2 处
                             cout<<"A"<<endl; }  cout<<"B"<<endl;}}
```

5. 从一个作用域跳跃到另一个作用域

（1）C++标准。

① 一个程序从不在自动存储期局部变量的作用域范围内的一个点跳跃到在作用域范围内的另一个点是不规范的，除非这个变量是 POD 类型，以及是没有初始化声明的变量（从 switch 的条件表达式到 case 标号的转移，被认为是遵守这种跳跃的）。

② POD 类型：即 C 语言的 struct 类型，或者是未自定义构造函数的类类型或 struct 类型。

（2）出现作用域跳跃情况的主要原因是变量的作用域问题。

（3）对标准的解释：switch 语句在 switch 的控制表达式这个点，程序就会产生跳跃（根据

控制表达式的结果选择相应的 case)。如果在 switch 内部定义有自动存储期的局部变量，那么在 switch 的控制表达式这个点，是不拥有在 switch 内部定义的局部变量的作用域范围的，因此跳进 switch 之后，也就跳进了声明该变量的作用域范围内（即从一个作用域跳跃到了另一个作用域)。所以说从 switch 的条件表达式跳跃到 case，即满足标准中所说的从不在自动存储期局部变量的作用域范围内的一个点跳跃到在作用域范围内的另一个点，这种跳跃是不规范的。除非在 switch 内部定义的局部变量是没有初始化的，或者它是一个 POD 类型的对象。也就是说，POD 类型的对象和只定义但不初始化的变量是允许这种跳跃的。

示例 5.8：从一个作用域跳跃到另一个作用域

```
switch(0)//程序从此处开始跳跃到局部变量a、b、c所在的作用域内，很明显，此处不在局部变量a、b、c的作用域内
{   //局部变量a、b、c的作用域从此处开始
    case 0: int a;    //正确，只定义未初始化的变量允许被跳跃绕过
    case 1: int b=2;  //错误，已经初始化的变量不允许被跳跃绕过
    case 2: int c;  }
```

（4）从一个点跳跃到另一个点，若没有绕过初始化，则是正确的；如果绕过了初始化，则是错误的。具体分为以下几种情形。

- 在 switch 内部定义变量但不初始化，就不存在绕过初始化的问题。
- 若类或结构体没有显式定义构造函数（即 POD 类型），则可以在 switch 内部创建类或结构体对象；若类或结构体显式定义了构造函数，则不能在 switch 内部创建对象。
- 可以在 switch 结构的最后一个 case 标号处（没有 default 标号时）或者 default 标号后面（default 应在所有 case 之后），对变量进行定义并初始化。因为这样 switch 是绕不过变量初始化的。

（5）解决作用域跳跃绕过初始化变量问题的最好方法就是在声明并初始化变量的地方使用大括号，将变量的作用域范围限制在大括号之内，这也是 case 之后的语句最好加上大括号的原因。比如 switch(3){case 0:; case 1: ; case 2:{int i=2; } case 3::}，switch 无法跳跃到变量 i 所在的作用域范围内，因为 i 的作用域范围在它所在的块（大括号）之内。

示例 5.9：跳进作用域绕过初始化

```
class A{};                  //类A没有显式定义构造函数，此时类A的类型相当于POD类型
class B{public: B(){}}; //类B显式定义了构造函数
switch(0)                   //程序从switch的条件表达式之处开始进行跳跃
    { int a;
    //case 2: int j=2;    //错误，j的初始化操作会被case 2及其后的case标号和default标号绕过
    //case 1: int i=1;  //错误，原因同上
    case 0: ;
    case 3: int b;  b=1;   //正确，因为变量b只是定义而没有初始化
    case 4: b=2;a=3;
    case 5: cout<<a<<b<<endl;   //若程序跳过case 4而直接执行此句，则变量a和b是没有初始值的
    case 6:{ int c=1;}    //正确，加上大括号之后，变量c的作用域只在大括号之内可见，switch无法直
```

```
                                    //接跳跃到变量 c 所在的作用域范围内
    case 7: A a2;                   //正确，因为类 A 未显式定义构造函数，类 A 的类型相当于 POD 类型
    //case 8: B b1;                 //错误，因为类 B 已显式定义了构造函数
    default:int k=3;}               //正确，不管 switch 的条件表达式为何值都不能绕过初始化变量 k 的行为
```

6. 有关强制转换为限定或非限定形式的规则（此规则对于数组下标同样有效）

强制转换为限定或非限定形式的类型具有相同的效果（标准规定），这种情况有可能在不同的编译器中有不同的处理，因此不推荐出现类似的语句。VC++ 2010 的规则如下。

（1）若将限定或非限定形式的整数类型（浮点型）强制转换为浮点型（整型）时，结果都将是非限定形式的相应类型。比如 float a=1;，则(const int)a; 结果为非限定形式的 int 类型；同理，const float a=1;，则(const int)a; 结果仍为非限定形式的 int 类型。

（2）若将限定或非限定形式的整数类型强制转换为非限定或限定形式的更长或更短的整数类型，则转换后不会改变整数类型的限定或非限定形式。比如 const int a=1;，则(long)a;的结果为 const long 类型；同理，int b=2;，(const long)b;的结果为 long 类型，因此不能用作数组的下标，如 int c[(const long)b];是错误的。

（3）若只是将同一类型的限定（非限定）形式转换为相应的非限定（限定）形式，则强制转换不起作用。比如 int a=1;，则(const int)a;对变量 a 没有影响，这时如果 a 以前是左值则仍可当作左值使用；((const int)a)=2; 是正确的，因为(const int)a 对变量没有产生作用。

5.4　while 和 do-while 语句

注意： while 或 do-while 语句指的是 while(控制表达式)和循环体所组成的一个整体。

1. while 和 do-while 语句的结构

```
while(控制表达式) 循环体
do 循环体 while(控制表达式);    //注意，后面有个分号
```

（1）while 语句的执行原则：首先判断控制表达式的结果是否为真，若为真则执行一次循环体，然后再次判断控制表达式的结果是否为真，若为真则再执行一次循环体，一直到控制表达式的结果为假为止。若控制表达式的值一直为真，则程序陷入死循环；若控制表达式的结果为假，则跳过循环体，继续执行 while 语句之后的语句。do-while 语句首先执行循环体（这里不用先判断控制表达式），然后判断控制表达式的结果是否为真，其余步骤与 while 相同。

（2）do-while 与 while 的区别：do-while 语句不管控制表达式的结果一开始是否为真，都会执行一次循环体；而 while 语句若控制表达式的结果一开始为假，则循环体一次都不会执行。示例如下：

```
while(1) cout<<"A";
```

说明：控制表达式恒为真，陷入死循环，程序不断输出 A。

```
while(0) 语句1; 语句2;
```

说明：控制表达式恒为假，直接跳过循环体（即语句 1），执行语句 2。

```
int i=2; while(i){语句1; i--;} 语句2;
```

说明：此语句执行两次循环，第一次控制表达式 i 的值为 2，结果为真，依次执行语句 1 和 i--；再次判断控制表达式，这时 i 的值为 1，同样为真继续执行，之后 i 的值为 0，结束循环，然后执行循环体后的语句 2。

```
do cout<<"A"; while(0);
```

说明：虽然控制表达式的结果为假，但仍然执行了一次循环体。do-while 语句不管控制表达式的值是真还是假，都要先执行一次循环体，然后再执行控制表达式，接下来再根据结果判断是否继续循环。这是与 while 语句的区别。

2. while 和 do-while 语句基本语法规则

（1）控制表达式应为标量类型。类类型等聚合类型一般不能用于控制表达式（I/O 类型是个例外）。注意：类对象的成员不一定是聚合类型。比如 class A{}; A ma;，则 while(ma){}是错误的。

（2）控制表达式可以是一个带有初始化的变量定义，若未进行初始化则是错误的。比如 while(int i=1){}是正确的，while(int i){}是错误的。

（3）不能在控制表达式中初始化并定义多个变量。比如 while(int i=1,j=2){}是错误的。

（4）循环体可以是一个单独的语句，也可以是一个语句块，循环体是 while(控制表达式)之后的第一条语句。一般情况下，循环体是一个语句块（即复合语句）；若要让循环语句执行一个语句序列，也应使用语句块。示例如下：

```
int i=-2;
while(i++)
cout<<"C"<<endl; //是 while 的循环体，会参与循环，C 被输出两次
cout<<"D"<<endl; //不是 while 的循环体，不参与循环，在执行完 while 循环之后输出 D
```

（5）根据 while 和 do-while 语句的执行原则，编写程序时在循环体中应具有影响控制表达式的结果的语句（如递增、递减等），否则程序会陷入死循环或无法控制循环的次数。比如 int i=2; while(i){}，如果在循环体中没有改变变量 i 的值的语句，则程序会陷入死循环，因此应在循环体中加上 i--之类的语句以改变控制表达式的结果。再比如 int i=2; while(i){其他语句; i--;}，循环两次，然后结束循环，这样程序就不会陷入死循环。

（6）在控制表达式中定义的变量，以及在循环体内定义的变量，每次循环时都要经历创建、撤销的过程。比如 while(int i=1){i--;}，程序是死循环，虽然循环体中有 i--;以改变变量 i 的值，但在每次循环时都会重新创建变量 i 并初始化为 1，因此控制表达式的结果始终为真，程序会一直循环。

5.5　for 语句

本文的 for 语句指的是“for(表达式 1;表达式 2;表达式 3)循环体”这一整体。

1. for 语句的结构

```
for(表达式 1;表达式 2;表达式 3) 循环体
```

for 语句执行顺序如图 5.1 所示。

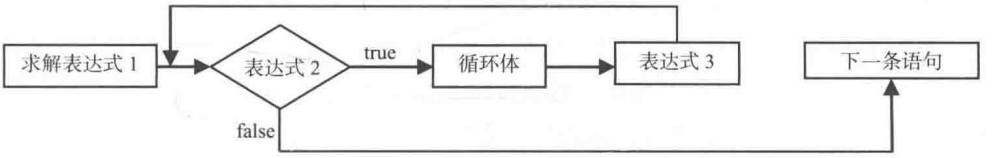


图 5.1　for 语句执行顺序

（1）for 语句的执行原则：首先执行表达式 1（只会执行一次），然后执行表达式 2（就是控制表达式），若表达式 2 的结果为真，则执行循环体，接下来执行表达式 3；然后再执行表达式 2，若表达式 2 的结果仍为真，则再执行循环体，再表达式 3，如此循环，直到表达式 2 的结果为假，退出循环，执行 for 语句之后的语句。

（2）比如 for (int i=2; i; i--) {cout<<"A"<<endl;} cout<<"B"<<endl;，输出 AAB。

执行顺序为：首先执行表达式 1（即 int i=2，只会执行一次），然后执行表达式 2（即 i），i 的值为 2，结果为真，输出 A，接下来执行表达式 3（即 i--）；然后再执行表达式 2，这时 i 的结果为 1，同样为真，再次输出 A，接下来再执行表达式 3（即 i--）；再执行一次表达式 2，这时 i 的值为 0，程序结束循环，执行循环体之后的下一条语句，输出 B。

2. for 语句基本语法规则

（1）for 语句可以用以下 while 语句代替：

```
表达式 1;while(表达式 2){for 的循环体部分;表达式 3;}
```

（2）表达式 1、表达式 2 和表达式 3 都可以省略。比如 for(;;)...、for(int i=1;;i++)...、for(;i;i++)...、for(int i=1;i;)...等。

（3）当省略表达式 2 时，其 for 语句的控制表达式恒为真，程序会陷入死循环。也就是说，省略表达式 2 与 for(...; true; ...)是等同的。

（4）表达式 1、表达式 2 和表达式 3 互不相关，不一定非要使用三个表达式来控制同一个变量。比如 for(int i=1; j; k++)...;是正确的。

（5）在表达式 1、表达式 2 和表达式 3 中可以使用逗号运算符，表达式遵守逗号运算符的规则，即整个逗号表达式应从左向右求值，以最右边的表达式的结果作为整个表达式的结果。使用逗号运算符之后，主要会对 for 的表达式 2（控制表达式）的结果产生影响，因为表达式 2

是被用来判断是否继续循环的依据。比如 for(int i=1,j=2; j+1,i+1; i--,j--)...，表达式 2 为逗号表达式“j+1,i+1”，因此表达式 2 的结果应是 i+1 的值，最后以 i+1 的结果为依据来判断程序是否继续循环。

（6）循环体可以是一个单独的语句，也可以是一个语句块，循环体是 for 头之后的第一条语句。一般情况下，循环体是一个语句块（即复合语句）；若要让循环语句执行一个语句序列，也应使用语句块。比如 for(...){语句 1;语句 2;}，当控制表达式为真时会依次执行语句 1 和语句 2；但是 for(...)语句 1;语句 2;则不同，当控制表达式为真时只会执行语句 1，不会执行语句 2。

（7）表达式 1 可以使用声明来代替，比如 for(int i,int j;;)、for(int i=1;;)等。

（8）表达式 1 只在 for 语句开始执行时执行一次。

（9）表达式 2 就是控制表达式，应为标量类型。类类型等聚合类型一般不能用于表达式 2（I/O 类型是个例外）。注意：类对象的成员不一定是聚合类型，比如 class A {}; A ma;，for(; ma ;) 是错误的。

（10）表达式 2 可以使用带初始化的变量定义语句代替，但变量没有初始化则是错误的，且不能在控制表达式中初始化并定义多个变量。比如 for(; int i=1;)是正确的，而 for(; int i;)、for(, int i=1,j=2;)都是错误的。

（11）表达式 2 在每次循环前都会求一次值，若其结果为假，则结束循环；若为真，则继续循环。

（12）表达式 3 在每次循环后执行一次，表达式 3 一般应为影响表达式 2 结果的表达式（如递增、递减等），以此来控制循环次数和防止死循环。当然也可以把影响表达式 2 结果的表达式（如递增、递减等）移至循环体内。

（13）同 while 语句一样，在控制表达式（即表达式 2）中定义的变量，以及在循环体内定义的变量，在每次循环时都要经历创建、撤销的过程。比如 for(int i;int j=2; j--){cout<<"A"; }，程序陷入死循环，虽然表达式 3 中有 j--;以改变变量 j 的值，但表达式 2 在每次输出 A 之后都会重新创建变量 j，因此变量 j 的值恒为真。

示例 5.10：for 语句综合示例

```
#include <iostream>
using namespace std;
int main(){
for(int i=2;i;i--)
cout<<"C"<<endl; //是 for 的循环体，参与循环，C 被输出两次，循环体是 for 头之后的第一条语句
cout<<"D"<<endl; //不是 for 的循环体，不参与循环，在执行完 for 循环之后输出 D
for(int i=2;i;i--)
{cout<<"E"<<endl; cout<<"D"<<endl;} // for 的循环体就是整个语句块，
for(int i=1,j=3,k;j+2,i+1;i--)//表达式 2 为逗号表达式“j+2,i+1”，以最右边的表达式的结果作为整个
                              //表达式的结果，因此以 i+1 的结果作为控制表达式是否为真的判断依据，
                              //所以程序循环两次，输出两次 F。
```

```
{cout<<"F"<<endl;}
//for(int i=2;int j;i--);      //错误，表达式 2 中的变量 j 没有初始化
for(int i=2;int j=1;i--)       //正确，表达式 2 可以使用带初始化的变量定义语句代替
{cout<<"G"<<endl;break;}   //break 用于跳出循环
//for(int i=2;int j=1,m=2;i--);  //错误，不能在控制表达式中初始化并定义多个变量
//for(int i=2,j=2; j,int k=1; i--);  //错误，原因同上
for(;;){cout<<"G"<<endl;break;}  //正确，表达式 1、表达式 2 和表达式 3 都可以省略，
                                 //当省略表达式 2 时，其 for 语句的控制表达式恒为真
for(int i=2;int j=1;j--){}  }//陷入死循环，int j=1 在每次循环后都会被重新创建，因此表达式 2 恒为真
```

5.6　continue 和 break 语句

（1）continue 和 break 语句的语法形式，就是在 continue 和 break 后面加上分号即可。

（2）break 语句只能出现在循环结构或 switch 结构中。比如 if(1){break;}是错误的，而 while(1){if(1){break;}}是正确的。

（3）continue 语句只能出现在循环结构中（注意：continue 不能出现在 switch 结构中）。比如 switch(1){case1: continue;}是错误的，而 while(1){if(1){continue;}}是正确的。

（4）break 语句的作用是跳出当前循环或 switch 语句，并执行紧接其后的语句。如果是嵌套的循环或 switch 语句，则只能跳出包含该 break 的循环或 switch 语句。示例如下：

```
for(int i=3;i;i--){语句 1; break; 语句 2;}语句 3;
```

说明：程序开始时为真，执行语句 1，然后遇到 break;，直接跳出循环，执行语句 3。这里语句 2 和表达式 3 都没有被执行到。

```
for(int i=3;i;i--)for(int j=2;j;j--){语句 1; break; 语句 2;}语句 3;
```

说明：包含 break 的循环语句是第二个 for 语句，而不是第一个，因为 break;在第二个 for 语句的循环体中，所以遇到 break;只会跳出第二个 for 循环，而不会跳出第一个 for 循环。本例把第二个 for 语句作为第一个 for 语句的循环体。

执行步骤：开始时 i 的值为真，执行循环体（即第二个 for 语句），这时 j 的值为真，执行语句 1，遇到 break;跳出第二个 for 循环，执行外围 for 语句的表达式 3（即 i--），再执行表达式 2（即 i），这时 i 的值为 2，仍为真，再执行第二个 for 语句，执行表达式 1（即 int j=2），初始化 j 为 2，表达式 2（即 j）的结果为真，执行语句 1，遇到 break;跳出第二个 for 循环，继续执行第一个 for 循环的表达式 3（即 i--），再判断第一个 for 循环的表达式 2（即 i），如此循环。故本例中语句 1 被执行 3 次，语句 2 未执行。若没有 break;语句，则语句 1 和语句 2 都会被执行 6 次。

```
for(int i=3;i;i--) {for(int j=2;j;j--){语句 1; 语句 2;}break; 语句 3; }语句 4;
```

说明：本例将 break;移到第二个 for 语句的循环体之外，这样包含 break;语句的是第一个 for 循环，因此遇到 break;会直接跳出第一个 for 循环。

执行步骤：第一个 for 循环最初为真，执行循环体中的语句，循环体中第一条语句是 for 语

句，for 语句循环两次，因此执行两次语句 1 和语句 2，然后执行后面的 break;跳出第一个 for 循环，执行语句 4。故本例中语句 3 没有被执行。若没有 break;语句，则程序会执行 6 次语句 1 和语句 2，执行 3 次语句 3。

（5）continue 语句的作用是结束包含该 continue 在内的循环语句的本次循环。继续下一次循环。对于嵌套循环，continue 同样只能结束包含该 continue 在内的循环语句的本次循环；对于 for 语句，遇到 continue 就直接转到表达式 3;对于 while 和 do-while 语句，程序直接转到控制表达式。示例如下：

```
for(int i=3; i; i--){语句1; continue; 语句2;}
```

说明：表达式 2 最初为真，执行循环体中的语句 1，然后遇到 continue;语句结束本次循环，继续执行下一次循环，因而跳过语句 2，而执行表达式 3（即 i--），接下来再执行表达式 2（即 i），这时 i 的值为 2，结果为真，继续执行语句 1，遇到 continue;，再次跳过语句 2，执行表达式 3;，如此循环，最后执行 3 次语句 1，语句 2 未执行。

```
int i=3; while(i){i--;continue; 语句2;}
```

说明：最初 i 的值为真，执行 i--，然后遇到 continue;结束本次循环，因此跳过语句 2，直接转到控制表达式（即 i），这时 i 的值为 2，继续执行 i--;，遇到 continue;，再次跳过语句 2，如此循环，最后 i--;被执行 3 次，语句 2 未执行。

（6）continue 与 break 的区别：循环语句一般会执行多次，continue 语句只是结束当前循环，若循环没有结束，则会继续执行下一次循环；而 break 则是直接跳出循环，即使循环未结束也不再执行循环语句。比如 int i=3; while(i){i--;continue; 语句 2;}语句 3;，i--;会被执行 3 次，然后才执行语句 3。若将 continue;换成 break;，则 i--;只执行一次就跳出循环执行语句 3 了。对于 for 语句也是同样的道理。

（7）注意：使用 continue 无法跳出死循环，要跳出死循环应使用 break。比如 while(1){cout<<"A"<<endl; continue;}，程序会无限输出 A，因为 continue 只能结束一次循环，要跳出死循环应将 continue 换成 break。

5.7 循环语句头定义的变量的作用域

有关作用域的内容请参阅第 7 章。

（1）对于在 for、while、dow-while、if 等语句头中声明的变量，其作用域一般只在与其相关的块语句中可见。比如 if(int i=1){...} i=3;，其中变量 i 在 for 语句头之后的块中是可见的，但是块外的 i=3;则是错误的，因为在这里 i 是不可见的；再比如 for(int i=1;...; ...){} i=3;，其中变量 i 的作用域在其后的块语句中是可见的，同样 i=3;是错误的。

（2）在早期的 C++版本中，可能会将 for 语句头中定义的变量视为在 for 语句之前定义的，

这样就允许在 for 语句之外访问该变量了。

5.8　goto 跳转语句简介

（1）使用 goto 语句将使程序变得很难理解和阅读，因此不推荐使用。

（2）goto 的作用是无条件跳转到同一函数内 goto 指定的标号后的第一条语句处。注意：goto 只能在同一函数内跳转。

（3）goto 语法形式为：goto 标号;。

（4）标号：就是后面带有冒号的标识符，比如 assf:、ddf:、rrr:等都是标号。

比如 void f(){int i=1; int j=2; goto ddd; j=3; i=4; ddd: j=5;}，程序执行到 goto ddd;时，将直接跳转到标号为 ddd 后的第一条语句，即 j=5;处。

第 6 章

指针和数组专题

（1）本章需要具有较强的复杂类型分析知识，请认真阅读第 3 章。

（2）重要概念：对象、变量、类型与内存，请参阅第 3 章；左值、右值，请参阅第 4 章。

（3）指针两要素：指针指向的类型和指针指向何处。大多数读者只关心指针指向何处，而忽略了指针指向的类型的重要性，本章将对指针这两方面的内容加以分析。

（4）本章讲解范围：对象类型指的是可以根据类型确定内存单元大小的类型，包括整型、浮点型、字符型、布尔型等。注意，函数类型和不完整类型都是无法确定内存单元大小的，因此都不是对象类型。为简单起见，本章只针对对象进行讲解，有关函数和不完整类型不作讲解。

6.1 指针

1. 地址值、地址类型、地址变量、地址常量的引入

（1）地址值和内存单元。

① 计算机中的内存，一般都是以字节为单位进行划分的。也就是说，计算机内存的最小单位是 1 个字节（8 位）。

② 内存单元：一个内存单元占据 1 个字节。

③ 地址值：每个内存单元使用一个值来标识，这个值就是地址值。地址值使用整数来表示，但它的类型不是整型。可见，一个地址值对应一个内存单元，可以使用一个地址值来指示一个内存单元。

④ 地址值的大小：在 32 位机器上，要表示所有的内存地址，需要使用 4 个字节（32 位）的二进制整数才能标识出所有的内存单元，因此标识一个内存单元的地址值应是 4 个字节的二进制整数（一般使用十六进制形式书写）。

（2）地址类型：地址值应该拥有一种数据类型，就像浮点数拥有浮点型一样，本文把地址值拥有的类型称为“地址类型”。因此，当内存中存储的数据为地址值时，它是地址类型，就像

内存中存储的数据是浮点数，为浮点型一样。在 32 位机器上地址类型占据的内存大小一般为 4 个字节。

（3）读者可能认为地址类型的值是一个整数不好理解，其实这就好比一个整数值既可以是浮点型也可以是整型，程序能区分出一个整数值按什么类型进行存储。比如将一个整数值赋给浮点型变量，则这个整数值会被按照浮点型的格式存储在内存中。

（4）地址类型变量（地址变量）：拥有地址类型之后，就可以创建地址类型的变量，就好比整型是一种类型，整型变量是表示整型的变量，整型变量存储的值是整数值。

（5）地址常量：十进制的整数常量就是地址常量，就好比 2.3 是浮点数常量一样。因此，一个整数值既是整数常量又是地址常量，但在 C++中对地址常量的使用是有限制的。

2. 指针类型、指针变量、指针值、指针常量的引入

在 C++中没有地址类型变量、地址类型、地址值、地址常量的说法，取而代之的是指针类型变量（指针变量）、指针类型、指针值、指针常量。也就是说，在 C++中指针类型就是指地址类型，指针变量就是指地址变量，指针值就是指地址值，指针常量就是指地址常量。

3. 指针变量如何存储地址值

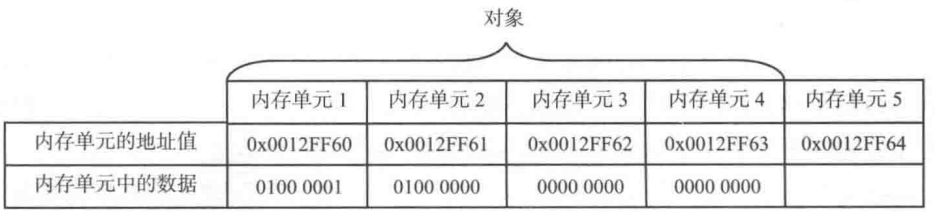
（1）指针类型的大小：在 32 位机器上指针类型的大小一般是 4 个字节（当然也可以更大，依系统而不同）。

（2）指针变量用于存储变量的地址值，而变量一般不止占据一个内存单元的大小，但指针变量只能保存一个内存单元的地址值，因此指针变量只能选择存储变量的首地址值或尾地址值（依机器而定）。一般机器存储的都是变量的首地址值，然后根据指针指向的变量类型，就能确定该变量应占据多少个内存单元。指针变量相当于一个指向某个对象的箭头。

（3）地址值是一个整数值，这就意味着可以直接将一个整数值赋给指针变量，但 C++禁止直接将整数值赋给指针变量（0 值除外），因此在 C++中这种操作不会成功。虽然现在禁止直接将整数值赋给指针变量，但是可以通过将整数值强制转换为地址类型（指针类型）后再使用。比如(int *)100=1;，表示把值 1 存储在内存中地址值为 100 的位置；int *p=(int *)100;，表示把内存中地址值为 100 的位置赋给指针 p。

4. 图解对象、变量、类型、左值、内存、内存的划分、指针的引入

图解对象、变量、类型、左值、内存、内存的划分、指针的引入，如图 6.1 所示。



	内存单元 1	内存单元 2	内存单元 3	内存单元 4	内存单元 5
内存单元的地址值	0x0012FF60	0x0012FF61	0x0012FF62	0x0012FF63	0x0012FF64
内存单元中的数据	0100 0001	0100 0000	0000 0000	0000 0000	

图 6.1　图解对象、变量、类型、左值、内存、内存的划分、指针的引入

说明：

（1）一个内存单元占据 1 个字节（8 位），其中每一位都可以存储一个二进制数值，每个内存单元都使用一个地址值进行标识，在 32 位机器上，一般使用 4 个字节的地址值来标识一个内存单元。比如图 6.1 中的内存单元 1 使用地址值 0x0012FF60 进行标识，这样更方便我们访问内存。0x0012FF60 是以十六进制形式表示的整数值，是地址（或指针）类型。

（2）数据类型决定了分配多少个连续的内存单元，可以存储什么样的数据，以及进行什么样的运算。说简单一点，就是数据类型决定了怎样解释内存单元中的数据。比如 int 类型一般占据 4 个字节，则 int 就决定了应为数据分配 4 个字节的内存单元，即要分配 4 个内存单元。

（3）C++使用“对象”这一概念来表示多个连续的内存单元（注意，并不仅仅指一个内存单元）。在图 6.1 中，假设某数据为 int 类型，且为该数据分配的内存单元分别是内存单元 1 至 4，则对象指的就是内存单元 1 至 4 总共 4 个内存单元。

（4）变量是命名的对象，左值则为指示一个对象的表达式。假设图中的内存单元 1 至 4 为一个对象，并为该对象取一个名字为 a，则 a 就是变量，表示的是 1 至 4 这 4 个字节的内存单元。左值是一个表达式，这个表达式应能表明指示的是一个对象，比如 a=1;表达式，a 指示了对象表示的内存单元 1 至 4，因此 a 是左值。因为单独的变量就是一个表达式，所以 a 就是一个左值。因此，变量名既可以代表该对象本身，同时也是左值。

（5）内存中存储的数据依声明时的类型而被解释，若图 6.1 中的内存单元 1 至 4 的数据为整型，则该数据被解释为整数 1094713344；若为浮点型，则该数据是浮点数 12（按 IEEE 754 标准进行转换）；若此数据被解释为一个内存地址值，则该数据为地址值 0x41400000（十六进制）。

（6）指针变量存储的值应是地址类型的值（地址值）。假设有一个指针变量 p，则存储的值应是内存单元的地址值，比如图 6.1 中的 0x0012FF60、0x0012FF61 等。假设 int 类型被分配到内存单元 1 至 4，并将这个对象命名为 a（即变量），且把变量 a 的地址赋给指针 p，则指针 p 存储的值就是变量 a 所代表的内存单元的首地址（依机器而定），即内存单元 1 的地址值 0x0012FF60。

5. 易混概念

以下概念虽然被混合使用，但从上下文中应能很容易区分开来。

（1）地址、地址值、地址类型：单独说“地址”一词时既可以表示地址值，也可以表示地址类型。这就好比 3 是一个整数，这个整数既可以代表整数值，也可以代表整数类型。

（2）由于习惯的原因，指针类型、指针变量值、指针值常被简单地称为指针，比如 int *p;，把 p 说成是指向 int 的指针，这里既可以强调 p 是一个指针类型的变量，也可以强调 p 的类型是指针；还可以说 p 是一个 int 指针，同样既可以强调 p 是指针类型，也可以强调 p 是指针类型的变量。

（3）“地址”和“指针”两个概念也常被混合使用，比如&a;既可以说成是返回 a 的地址，也可以说成是返回 a 的指针。

（4）指针的类型与指针指向的类型：经常把指针指向的类型说成是指针的类型，比如 int *p;，表示指针 p 的类型为“指向 int 的指针”。本书会经常使用“指针类型”这个概念。

6.1.1　指针的概念

1. 标准对指针的定义

（1）指针的定义（K&C）：指针是一种保存变量地址的变量。这里的指针指的是指针类型变量，简称为指针变量或指针。平时所讲的指针就是指指针类型变量。

（2）指针类型的定义（ANSI C89）：指针类型可由函数类型、对象类型或不完整类型派生，派生指针类型的类型称为引用类型。指针类型描述一个对象，这个对象的值提供对该引用类型实体的引用。由引用类型 T 派生的指针类型有时称为“（指向）T 的指针”，从引用类型构造指针类型的过程称为“指针类型的派生”。

（3）注意，标准中把指针的类型称为“（指向）T 的指针”。

2. 理解 ANSI C89 标准对指针类型的描述

（1）指针类型可由函数类型、对象类型或不完整类型派生，说明了以下两点内容。

① 指明指针类型是一种类型，所以就决定了如下一些性质。

- 因为有了类型，所以就有了有关这一类型的变量和值（比如有了整型，就有了整型变量和整型值），因此就有了关于指针类型的变量（简称指针变量或指针）和指针类型的值（简称指针值）。
- 指针类型决定了分配内存的长度（或称为大小），一般在 32 位机器上指针类型占据内存的大小为 4 个字节，在 64 位机器上可能为 8 个字节。
- 指针类型决定了所分配的内存中应保存何种类型的数据，在指针类型的内存中保存的数据是地址类型的值（简称地址值），这个值可能是以十进制、二进制或十六进制的整

数形式表示的，在计算机中一般以十六进制形式表示地址值。注意：虽然地址值是一个整数值，但是它的类型是地址类型，而不是整型，这就好比整数“2”在内存中保存为浮点型，则这个数并不是整数类型。

② 指明指针类型是一种派生类型。也就是说，指针类型应在某一类型的基础上产生（或派生），不存在单独的指针类型。指针类型是一种复合类型，必须配合函数类型、对象类型或不完整类型才能产生。比如 int *p;是一个指向 int 类型的指针类型，它是由对象类型（int）派生的。注意：对象类型能根据其类型确定内存的大小，而函数类型和不完整类型则不能。

（2）派生指针类型的类型称为引用类型；从引用类型构造指针类型的过程称为指针类型的派生。

首先要把引用类型和引用区别开来，这里的引用类型指的是能派生出指针类型的类型，因此函数类型、对象类型、不完整类型都可以被称为引用类型。

（3）指针类型描述一个对象，这个对象的值提供对该引用类型实体的引用。

① 指针类型描述一个对象，而对象指的是某种类型所占据的一片连续的内存单元，因此指针类型可以用来描述一片连续的内存单元。简单点说，指针类型其实就是类型。还说明指针类型是对象类型，能确定其内存大小（比如在 32 位机器上一般为 4 个字节，具体依系统而不同）。

② 这个对象的值提供对该引用类型实体的引用，这里的引用类型指的是派生指针类型的类型（比如整型、浮点型等），对象指的是指针类型所描述的内存单元，所以对象的值指的就是内存单元中存储的值（其实就是指指针类型的值）。我们知道指针类型的值应是地址类型的值，但标准中却是指“对该引用类型实体的引用”，这就说明指针类型的地址值应是派生出指针类型的引用类型（比如整型、浮点型等）所描述的对象（或内存单元）的引用，引用的方式一般是使用该对象（或内存单元）的首地址或尾地址（依机器而不同）表示这个连续的地址。这也就是为什么说指针类型一般被解释为指向派生指针类型的引用类型的箭头的原因。比如整型的地址有 4 个字节，其地址分别是 0x0012FF60、0x0012FF61、0x0012FF62、0x0012FF63，则对该整型的引用一般使用首地址 0x0012FF60 来表示这 4 个字节的内存，一般不会将 4 个字节的内存单元都作为指针类型的值。

（4）注意“指针类型”的表述：由引用类型 T 派生的指针类型有时称为“（指向）T 的指针”，就是表示平时所说的“指针类型”有时指的是“指向 T 的指针”，而不是地址类型。

6.1.2 &与*运算符

注：读取内存地址时，本章假设读取的是首地址，而不是尾地址。

1. 理解&（取址）与 *（解引用）运算符

注意：一个内存单元占据 1 个字节的大小。

（1）对&（取址）运算符的简单理解：就是读取对象所示连续内存单元的地址（值），但只能读取首地址，即第一个内存单元的地址值。比如&a，表示读取操作数 a 的地址值。当然，&运算符并不会把操作数所表示对象的所有内存单元地址值都读取出来，一般只读取第一个内存单元的地址（即首地址），然后根据操作数的类型就能知道共占据多少个内存单元。若 a 是 int 类型，占 4 个字节内存单元，地址分别是 0x0000 0002~0x0000 0005，则&a 将只会读取操作数 a 所在的连续内存单元的首地址 0x000 0002，而不会全部读出。

（2）&运算符的操作数应是表示一个对象的左值，该对象不能是位段，且声明时无 register（寄存器）存储类区分符（寄存器是没有地址可取的）。因为左值是指示一个对象的表达式，也就是说，左值本身就是表示的对象，所以说&运算符的操作数应是一个左值。比如&3 是错误的，因为整数 3 是右值；再比如 register int a=1;，则&a;是错误的，因为操作数是 register 变量。

（3）注意：register 关键字只是 C++标准对编译器的一个建议，编译器不一定会执行该操作，因此在某些编译器上对 register 变量使用&运算符不一定会出错。为了使兼容性最好，不要对 register 变量使用&运算符。

（4）ANSI C 中对&运算符描述的原话：“&（地址）运算符的结果是指向由操作数所指示的对象或函数的指针，其结果类型是指向 type 的指针”。

理解标准：虽然&运算符的结果是指针，但并没有为该指针分配内存地址，因此&运算符的结果不能作为左值，只能作为右值。因为指针指向的是操作数所指示的对象或函数的指针，所以这个右值就是操作数所指示对象或函数的地址，在这里也可将&运算符的结果作为指针常量（即地址值）来理解。比如 int a=1;，则&a 的结果是指向操作数 a 所指示对象的指针，这个指针并没有分配内存地址，只能作为右值使用，这时该指针的值就是操作数 a 所代表的一片连续的内存单元（4 个字节）的首地址或尾地址，该指针的类型是“指向 int 的指针”。此处混用了指针和地址的概念。

（5）*（解引用）运算符，其操作数应是指针类型，若操作数指向一个对象（即操作数是指针），则结果是一个指示该对象的左值。若操作数的类型是指向某类型的指针，则结果类型就是该类型。*运算符的操作数应是右值，但其类型必须是指针类型。

① 理解 1：*运算符的结果是操作数指向的对象的左值或指针所指对象的左值。意思是说使用*运算符之后得到的是一个左值，这个左值表示的是操作数所指向的对象，在左值不是很重要的情况下可以忽略左值，因此*运算符的结果就是操作数所指向的对象。这就意味着，使用*运算符之后，可以获取操作数（或指针）所指向的对象。比如 int a=1; int *p=&a;（注意，此语句只是声明指针 p 的一种语法形式），若以后在表达式中出现*p，则结果是操作数 p（p 是指针）指向的对象的左值，操作数 p 指向的是被命名为 a 的对象，*p 和 a 都是表示相同对象的左值，所以对*p 的改变会影响到变量 a 的值；对于*p=2，则变量 a 的值也变为 2 了。注意：变量名 a 既可以表示被命名为 a 的对象，也可以表示指向对象 a 的左值，因为在某些情况下左值就是所

表示的对象。

为什么强调左值？主要是为了说明*运算符的结果不是右值，因此可以使用*运算符间接改变指针所指向的对象的值。若*运算符的结果是右值，则不能间接改变指针所指向的对象的值。比如 int a=1; int *p=&a;，则*p=2;会间接改变 a 的值。

② 理解 2：*运算符能得到操作数所指对象处存储的值。比如 int b=1; int *p=&b;，则*p 在需要将*p 作为右值的地方（会经过从左值到右值的转换），确实得到了 p 所指对象处存储的值 1。但*运算符的结果是左值，还可以对*p 进行赋值，而所指地址处存储的值一般都是右值，不能对右值进行赋值操作。当然，左值可以作为右值使用。

（6）比如 int a=1;，则*&a;的操作数&a 是一个指向对象 a 的指针。也就是说，操作数指向对象 a，因此*&a 的结果就是操作数&a 指向的对象 a 的左值，即*&a 是名称为 a 的对象的左值。而 a 本身也是名称为 a 的对象的左值。也就是说，*&a 和 a 都是指名称为 a 的对象的内存单元，因此*&a 与 a 等价，即*&a 的值是 1，同时*&a 是左值，可以对其赋值，比如*&a=3;会间接改变 a 的值。因为&a 是指向 int 的指针，所以*&a 的结果类型是 int。

再比如 int b=1; int *p=&b;，则*p 的结果是操作数 p 指向的对象 b 的左值，即*p 是名称为 b 的对象的左值，而 b 本身就是对象 b 的左值。也就是说，*p 与 b 表示的都是名称为 b 的对象的内存单元，因此*p 与 b 是等价的，即*p 的值是 1，同时*p 的结果是左值，可以对其赋值，比如*p=3;会间接改变 b 的值。因为 p 是指向 int 的指针，所以*p 的结果类型是 int。

（7）注意：*（解引用）运算符与*（乘法）运算符是相同的，&（取址）运算符与&（按位与）运算符是相同的，这些运算符都能根据程序的上下文区分开来。

2. &运算符总结要点

（1）&运算符能读取对象所示连续内存单元的地址（值），一般为首地址或尾地址。

（2）&运算符的操作数是左值。

（3）&运算符的结果是右值，不能作为左值使用。

（4）&运算符的结果类型是“指向某类型的指针”。

3. *运算符总结要点

（1）使用*运算符之后，可以获取操作数（或指针）所指向的对象。比如 int a=1; int *p=&a;，则*p 就是表示操作数 p 所指向的对象 a。

（2）*运算符的操作数应是右值，且是指针类型。

（3）*运算符的结果是左值，因此可以对使用*运算符之后的结果进行赋值。

（4）*运算符的结果类型就是操作数指向的对象的类型。

（5）*运算符会间接改变操作数所指向的对象的值。

4．图解指针的概念、地址、&与*运算符

图解指针的概念、地址、&与*运算符，如图 6.2 所示。

图 6.2　图解指针的概念、地址、&与*运算符

说明：

（1）假设有这样的声明：int a=128; int *p=&a;（这是声明指针 p 的语法形式）。

（2）在声明语句 int *p=&a;中，&a 表示读取变量 a 所代表的一片连续的内存单元的首地址（即内存单元 1 的地址），然后将该首地址用于初始化指针变量 p，这时 p 的值应是变量 a 所指示的内存单元 1 的地址值，也就是 0x0012FF60。因此，指针 p 相当于一个箭头，这个箭头指向了变量 a 所表示的一片内存单元的首个内存单元处（即内存单元 1）。

（3）从图 6.2 中可见，指针变量 p 的值就是变量 a 的首地址值，p 的值虽然是一个整数，但其类型是地址类型（指针类型），而不是整数类型。

（4）若在以后的表达式或语句中使用形如*p 的形式，比如*p=3;，则*是解引用运算符，而不是在声明时表示所声明的对象是一个指针的声明符，这时*运算符的结果是指针 p 所指对象 a 的左值，因此*p 和 a 是等价的，即*p 和 a 都表示名称为 a 的对象所代表的内存单元 1 至 4。所以，*p=3 就相当于对 a 进行赋值，*p 间接改变了变量 a 存储在内存单元中的值；cout<<*p;相当于输出 a 的值。

6.1.3 指针（变量）的声明、初始化

为便于描述，下文将不对指针的声明和定义进行详细的区分，相关内容请参阅第 3 章。

（1）要声明一个指针，应在标识符的前面加上*（解引用）运算符；声明多个指针时，应在每个标识符的前面都加上*运算符，声明时*运算符只用来表明该标识符是一个指针，而不会产生解引用运算符的作用。比如 int *p1, p2, *p3;，这里声明了 p1 和 p3 两个指针且都指向 int 类型，因为 p2 前面没有*运算符，所以 p2 不是指针，而是一个 int 类型变量。

（2）声明指针时的两种书写风格：在声明指针时*可以靠近类型一侧，也可以靠近标识符一侧。比如 int* p;，*靠近类型一侧；int *p，*靠近标识符一侧。在声明多个指针时第一种风格容易产生误解，比如 int* p1, p2;容易误认为声明了 p1 和 p2 两个指针，但 p2 并不是指针；而第二种风格 int *p1, p2;不容易产生误解，本文声明指针时使用第二种风格。有人认为 int*是一种类型，*应靠近类型一侧，但实际不然，*与标识符组合之后仍是声明符，声明符应归类放在一起用于声明变量，因此将*靠近标识符一侧更合理，更不易出错，也更便于理解所声明的指针。

（3）对声明多个指针的理解：在*与标识符之间可能还会有类型限定词，在声明多个指针时，可以在*和标识符之间加入不同的限定词以声明具有不同限定类型的指针。比如 int *const p1, p2;，p1 是指针常量，而 p2 却是 int 类型变量，因为*const p1 是一个声明符，声明时表明 p1 指向的是 int 类型；同理，p2 也是声明符，其中声明区分符是 int，而*const 并不是声明区分符的内容，因此 p2 是 int 类型的。

（4）指针的初始化：每个指针都有相应的类型，类型决定了指针应指向什么类型的对象，以及指针能存储什么类型对象的地址，指针不能接受非地址类型的值，还决定了进行指针运算时的偏移量。比如 int *p; int a; float b; int *p1=&a;，则 p=&a;、p=p1;都是正确的；而 p=&b;、p=32;、p=a; 都是错误的，其中 b 的类型与 p 不相同，整数值 32 和 a 都是非地址类型的值。

（5）对于 int *p=&a;，初始化的是指针 p 本身，而不是*p，此语句与 int *p; p=&a;等同。

（6）与普通变量一样，若使用了未初始化的指针，则其行为是未定义的，可能导致程序崩溃。与变量相同，未初始化的指针未定义，并不代表指针没有值，而是拥有一个随机值，因为指针保存的是地址，所以这个随机值会被解释为内存地址。当使用*运算符访问这个指针中保存的随机地址值处的内容时，就极有可能导致程序崩溃，因为所访问的地方是未知的。

（7）指针存储的是地址值，因为指针变量存储的值是对象的地址，即指针的值是地址值，所以指针名是一个地址值。比如 int a=1; int *p=&a;，则 cout<<p;将输出 p 的值，因为 p 的值是变量 a 的地址（一般为首地址），所以将输出变量 a 的地址。要使用指针输出存储在指针所指变量 a 处的值，则应在指针名前使用*运算符，比如 cout<<*p;，则会输出 a 的值 1。

6.1.4　各种指针

1．二级指针（指向指针的指针）及多级指针

（1）指针本身也是一个变量，因此也会占据内存空间，所以就可以使用另一个指针存储该指针所占内存空间的地址值，这样就出现了指向指针的指针及多级指针。

（2）使用两个“*”号就能声明指向指针的指针(二级指针)，使用三个“*”号就能声明三级指针，依此类推。比如 int **p;就表示指向指针的指针，指针 p 存储的值是 int *类型指针的地址；而 int ***p;表示这是一个三级指针，存储的是一个 int **类型的指针的地址。

（3）对于二级指针，应将一个指针的地址赋给它；对于三级指针，应将一个二级指针的地址赋给它，依此类推。比如 int a=1; int *p=&a; int **p1;，则 p1=&p;是正确的，而 p1=p; p1=&a;是错误的。

（4）对二级指针进行一次解引用操作将得到它所指向的一级指针，要想访问到真正存储值的对象，则应对其进行两次解引用。同理，对于三级指针，应进行 3 次解引用操作才能访问到真正存储值的对象。比如 int a=1; int *p=&a; int **p1=&p;，则 cout<<*p1;输出的是 p1 所指向的对象 p 的值，即 p 指向的对象的地址&a;，而 cout<<**p1;输出的是 p1 指向的对象 p 指向的对象 a 的值 1。

（5）对二级或多级指针初始化时应注意，在未对二级或多级指针进行有效初始化时，就对其使用解引用是错误的。比如 int **p; int a=1; int b=2; int *p1=&a;，则*p=&a;是错误的，这里并没有达到对二级指针 p 的初始化的目的，因为二级指针 p 未初始化，而对其进行解引用显然是错误的。再比如 p=&p1;，则*p=&b; 是正确的，因为 p 指向是 p1，因此*p 就是指 p1 本身，这样就间接改变了指针 p1 的值，即让 p1 指向对象 b，而不是以前的 a。同理，对于 p=0;，则*p=&b;同样是错误的，因为 p 没有指向任何对象，而对其进行解引用显然会得到意想不到的结果。

（6）若声明的是 *N* 级指针，则进行一次解引用将得到 *N*–1 级指针，两次解引用将得到 *N*-2 级指针。比如 int *****p;，则 p 是五级指针，*p 是四级指针，**p 是三级指针，依此类推。

图解二级指针，如图 6.3 所示。

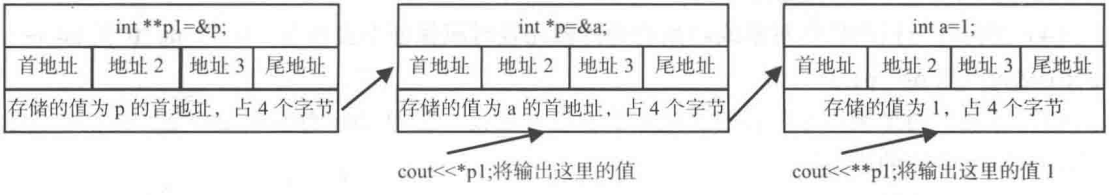


图 6.3　图解二级指针

示例 6.1：二级指针

```
#include <iostream>
using namespace std;
int main(){
    int a1=1;int a2=2; int **p1;int *p2;
    p2=&a1;
    //p1=&a1;     //错误，二级指针需要一个指针的地址
    //*p1=&a1;    /*错误，这里并没有达到对二级指针 p1 初始化的目的，在未对二级或多级指针进行有效初始
                    化时，就对其进行解引用将是错误的*/
    p1=&p2;       //正确，将指针 p2 的地址赋给二级指针 p1
    *p1=&a2;      /*正确，对二级指针进行一次解引用得到的是 p1 所指向的一级指针 p2，这里间接改变了 p2
                    的指向，这时 p2 与*p1 都指向了变量 a2 的地址。同时**p1 得到的也是 a2 的值*/
    cout<<*p1<<endl;    cout<<p2<<endl;  //输出相同的地址
    **p1=55;      //正确，对二级指针进行两次解引用，将访问到真正存储值的对象 a2
    cout<<**p1<<endl;   cout<<a2<<endl; }//输出相同的值 55
```

2. 空指针、空指针常量

（1）空指针常量：值为 0 的整型常量表达式，或强制（转换）为 void*类型的此类表达式。可见，在特殊情况下整数值 0 可以作为一个地址类型的值赋给指针变量，除整数值 0 之外的其他整数类型是不能作为地址类型的值赋给指针变量的。比如整型常量 0、4–4; '\0'、0*5;(void*)0 是空指针常量，但是 int *p; (void *)p、0.0 不是空指针常量，因为 p 是变量，0.0 是浮点型而不是整型常量表达式；同理，(char *)3; (char*)0 也不是空指针常量，因为不是 void*类型；再比如 int a=0;，其中的 a 不是空指针常量，因为 a 是变量，不是值为 0 的整型常量表达式。

（2）有两个值可作为空指针常量，即值为 0 的整型常量表达式和(void*)0，选择哪种值作为空指针常量是由系统决定的。在大多数系统中都使用值为 0 的整型常量表达式，下面就以 0 作为空指针常量，而不计算(void*)0 为空指针常量的情况。

（3）空指针：将空指针常量赋予一个有类型的指针变量时，这个指针被称为空指针。空指针不指向任何对象或函数。比如 int *p=0; char *p1=4–4;，其中 p 和 p1 都是空指针，整数值 0 和 4–4 是空指针常量；但对于 int a=0;，则 int *p=a;是错误的，这里的 a 不是空指针常量，因为 a 不是值为 0 的整型常量表达式。

（4）空指针与指向任何对象或函数的指针做比较时应保证不会相等。比如 int *p=0; int a=1; int *p1=&a;，则 p= =p1 应为假。

（5）注意：两个类型不兼容的空指针不能进行比较。比如 char *p1=0; int *p2=0; p1= =p2; 会出错，因为它们的类型不兼容。

（6）空指针常量一般使用宏 NULL 来定义，标准中并没有规定空指针应指向内存的什么位置。也就是说，空指针指向的位置是由系统实现的。一般情况下空指针都指向全 0 的地址，但也有一些系统不一定使用 0 来表示空指针的位置，这样的系统 NULL 的值就不一定是 0。因此

在程序中若要使用空指针常量，最好是使用 NULL，而不是 0，以便获得更好的兼容性。

（7）检查一个指针是否为空指针的办法最好是将指针和 NULL 进行对比，比如 p= =NULL。当然，若知道系统使用的是值为 0 的整型常量表达式作为空指针常量，也可以将指针与值为 0 的整型常量表达式进行比较，以判断该指针是否为空指针。

（8）不能对空指针进行解引用操作，比如 int *p=0;，则*p=2;是非法的。但有些系统允许进行这种操作。

示例 6.2：空指针、空指针常量

```
int main() {
    int *p3;   int a3=3;
    //p3=(void*)0;     //这里不保证所有编译器都正确，因为空指针常量可使用两个值表示，即值
                       //为 0 的整型常量表达式和(void *)0，具体依系统而定。本系统使用值
                       //为 0 的整型常量表达式表示空指针常量
    p3=0;              //正确，0 是空指针常量
    p3=4-4;            //正确，4-4 也是空指针常量，4-4 是值为 0 的整型常量表达式
    //p3=a3-3;         //错误，虽然 a3-3 的值为 0，但 a3-3 不是整型常量表达式
    //p3=0.0;          //错误，0.0 是浮点数，不是整型常量表达式
    //p3=4-4.0;        //错误，4-4.0 不是整型常量表达式
    p3='\0';           //正确
    char *p4=0;
    //p3==p4;         //错误，两个类型不兼容的空指针不能进行比较
    //cout<<*p4<<endl; //错误，不能对空指针进行解引用操作
    p4=NULL;   }//在程序中若要使用空指针常量，最好是使用 NULL，而不是 0
               //以便获得更好的兼容性
```

3. void 指针

（1）void 指针：即类型为 void 的指针。void 表示无类型，因此 void 指针也称为无类型指针。这种指针可以保存任何类型对象的地址，其中包括不完整类型，void 指针表明这是一个指针，但不清楚该指针指向何种类型的对象。

（2）指向 void 的指针可以转换为指向任何不完整类型或对象类型的指针；反之，任何不完整类型或对象类型的指针也可以转换为指向 void 的指针。这里的转换是指强制类型转换，而不是隐式或自动转换。

（3）因为 void 指针可以保存任何类型对象的地址，所以可以将其他任何类型的对象的地址赋值给 void 指针，但反过来却不可以。比如 int a=1; int *p=&a; void *p1;，则 p1=p;、p1=&a;都是正确的，但 p=p1;则是错误的。

（4）C++不允许 void 指针操作它所指向的对象，即不允许对 void 指针使用*（解引用）运算符对其指向的对象进行直接操作。比如 int a=1; void *p=&a;，则 cout<<*p;是错误的，正确形式为 cout<<*(int*)p;。

（5）不能对 void 指针进行算法操作，原因很简单，因为 void 没有类型，所以无法确定 void 指针所指向的对象的大小，也就无法对 void 指针的算法进行计算。比如 int a=1; void *p=&a;，则 p+1;、p++;、++p;等都是错误的操作。

4. 常量指针与指针常量

（1）常量指针（const int *p）：指的是指向 const（常量）的指针，指向常量的指针不能通过指针来修改它所指对象的值（即*p 是常量），但可以改变指针本身的值（即 p 不是常量），即可以改变指针所指向的地址。比如 int a;=1; const int *p=&a;，则*p=3;是错误的，但可以改变指针本身的值；再比如 int b=1;，则 p=&b;是正确的，因为 p 不是常量。

（2）指针常量（int *const p）：指的是指针本身是常量（即 p 是常量），指针常量不可以改变指针本身的值（即不能改变指针所指向的地址），但可以修改它所指对象的值（即*p 不是常量）。因为指针常量是常量，所以若局部声明指针常量，则必须初始化。比如 int a=1; int *const p=&a; int b=1;，则 p=&b;是错误的，但*p 不是常量，可以改变*p 的值，即*p=3;是正确的。

（3）常量指针与指针常量两个概念很容易搞混，因此本书不推荐使用这两个概念。本书后面的内容将常量指针称为指向常量的指针，将指针常量称为 const 指针，这样更通俗易懂。

（4）const int *p 与 int *const p 的声明分析：对于 const int *p，这里 p 首先与*结合，说明 p 是一个指针，指向的类型是 const int，此处 const 限定的是 int，而不是指针；而对于 int *const p，这里 p 首先与 const 结合，说明 p 是常量，然后与*结合，说明这个常量是一个指针，这个指针指向 int，此处 const 限定的是指针 p 本身，而不是 int。

（5）指向 const 对象的 const 指针（const int *const p）：这种指针既不能修改指针本身的值，也不能通过指针修改它所指向的对象的值。比如 int b=1; int a=2;const int *const p=&b;，则 p=&a;是错误的，*p=2;也是错误的。

5. 含 const 的指针间的赋值

（1）若赋值运算符的两个操作数都是指向相容类型的限定或非限定形式的指针，则左操作数所指向的类型应具有右操作数所指向的类型的全部限定词。此规则要注意理解指针指向的类型和指针的类型。对规则的理解如下。

① 不可以使用指向非 const 对象的指针（比如 int *p）指向 const 对象。因为若允许这种操作，则很明显可以通过指针来间接修改 const 限定类型的对象的值，这违背了 const 的常量状态。比如 const int a=1;，若 int *p=&a;被允许，则*p=3;就改变了 a 的值，从而违背了 a 的 const 常量状态，因此不允许这种赋值。

② 可以使用指向 const 对象的指针（比如 const int*p）指向非 const 或 const 对象。示例如下：

- const int a=1;，则 int *p=&a;错误。

- int a=1; const int *p1=&a;，则 int *p2=p1;错误。
- int a=1; const int b=2; const int *p;，则 p=&a;、p=&b;都是正确的。
- const int a=1;，则 int *const p=&a; 正确，因为 p 指向的是非 const 类型的。
- const int b=3;，则 int *const p=&b;错误，因为指针 p 指向的是非 const 类型的。

（2）虽然指向 const 的指针不能通过指针来修改它所指向的对象的值，但不能保证指向 const 的指针所指向的对象的值不能被修改，因为指向 const 的指针可以指向非 const 对象。比如 int a=1; const int *p=&a;，则可以通过改变变量 a 的值来间接修改指针 p 所指向的对象的值。比如 a=2; 是正确的，但*p=2;是错误的。

（3）void *指针同样不能用于保存 const 对象的地址，而应使用 const void *类型的指针保存 const 对象的地址。

（4）若试图通过使用非 const 限定类型的左值去修改定义为 const 限定类型的对象，则是未定义的，比如 const int a=2;，则对于 int*p=(int *)&a; *p=3; 或*(int *)&a=3;，这时常量 a 的值是否改变是未定义的。注意：int *p=&a;语句是错误的，(int)a=3;语句也是错误的。

6. 多级指针与 const

（1）对于二级指针，在 const 与非 const 之间进行赋值将不再安全。与一级指针的情况有些不同，对于二级指针，不同的编译器可能会有不同的处理方法。

（2）下面是 VC++ 2010 对多级指针含有 const 的一些处理方法。

- int **p1; const int **p2;，则 p1=p2; 或 p2=p1;都是错误的。
- int *p1; const int **p2;，则 p2=&p1;是错误的。
- const int *p1; int **p2;，则 p2=&p1;是错误的。
- const int *p1; const int **p2;，则 p2=&p1; 是正确的。
- int *p1; const int *const *p2;，则 p2=&p1; 是正确的。
- const int *p1; const int *const *p2;，则 p2=&p1;是正确的。
- const int *p1; int *const *p2;，则 p2=&p1; 是错误的。

7. 多级指针与 const 的结合

（1）在声明多级指针时，const 始终与左边的*相结合。

（2）若声明了一个 *N* 级指针，const 与从标识符的位置开始从右向左数的第 *M* 个“*”号相结合（注意：const 与左边的“*”相结合），则 const 限制的是 *N*–*M*+1 级指针，或者说从 const 的位置向左数，其中间有多少个“*”，则 const 限制就的是多少级指针。比如 int ****const**p;，这里是六级指针，const 与第三个“*”相结合，即从右边向左数的第三个星号，因此 const 限定的是四级指针，即限制的是**p。这里**p 是四级指针。

6.1.5 指针的简单运算

1. 指针的运算

（1）指针存储的是地址，虽然计算机常把地址作为整数处理，但是它们的类型并不相同，整数可进行加、减、乘、除等运算，而地址描述的是内存的位置，显然将两个地址进行相乘是没有意义的。

（2）可以对指针进行算术和关系运算。

（3）标准中对指针的运算仅限于指向数组中某个元素的指针。

2. 指针的算术运算

（1）指针的算术运算只限于两种形式：一是指针与整数相加减；二是指针与指针相减。注意：两个指针不能相加。

（2）指针加上或减去整数值 *n*，将使指针向前或向后偏移“n*指针指向类型所占据的字节数”这么多个字节，可见指针类型决定了指针加上或减去一个整数的偏移量。

① 指针加上或减去一个整数值 *n*，并不是将指针的地址值简单地加上或减去这个整数值 *n*，而是加上或减去指针所指向类型的大小再乘以整数值 *n*，这样可以确保指针指向的一定是下一个对象，而不是对象内部（对象一般不止占据一个内存单元）的某个位置。

② 若指针的算术运算不是在数组内部进行的或结果超出数组的范围，则结果在标准中是未定义的，而且也会产生意想不到的后果，因为并不知道进行算术运算后指针所指向的下一个位置究竟是什么内容。

③ 示例：int a=1; int *p=&a;，则 p+2; 会把指针的地址值加上指针所指向类型的大小（int 类型一般为 4 个字节）乘以数值 2，因此结果是把指针向后偏移 8 个字节，而不是 2 个字节。因为 p 不位于数组内，因此执行 p+2 后，并不知道指针指向的位置究竟是什么内容，即对 p+2 解引用后*(p+2)，会得到意想不到的值。

（3）当指针加上或减去一个整数值 *n* 后，将产生一个新指针，并且可以直接使用这个新指针，这个新指针将指向原指针指向的元素之后或之前的第 *n* 个元素；若新指针指向位于同一个数组中的元素，或者指向数组中最后一个元素后面的那个位置（即最后一个元素的下一个位置）则是合法的，否则是未定义的。也就是说，若执行算术运算之后指针指向了数组第一个元素前面或最后一个元素后面的那个位置的后面，则结果是未知的，但大多数编译器允许这种操作。注意：虽然进行算术运算后允许指针指向数组中最后一个元素后面的那个位置，但对其进行解引用则可能会产生意想不到的结果，因为并不清楚该位置以前存储的是什么内容。比如 int a[11]={0}; int *p=&a[0]; *(p+1)=1;，指针 p 指向数组的第一个元素，p+1 会产生一个新指针，这个新指针指向的是原指针指向的元素之后的第一个元素，即 p+1 指向的是数组的第二个元素，因此*(p+1)=1;将把数组中第二个元素 a[1]的值修改为数值 1。再比如 int a[3]={0}; int *p=&a[0];，

则 p+3;将指向数组 a 中最后一个元素之后的那个位置，标准中规定这是合法的，但对其进行解引用则可能会产生意想不到的后果。对于*(p+3);，则结果是无法预料的；而 p–1 将使指针指向数组 a 的第一个元素的前面，这在标准中是不合法的，但大多数编译器允许这样做。

（4）指针加上或减去一个整数值 *n*，虽然会产生一个新指针并可以使用这个新指针，但算术运算的结果是右值，不是左值，不能对该指针进行赋值，但可以使用解引用操作访问新指针所指向的内容，解引用运算符的结果是左值。比如 int a[33]={0}; int *p1=a; int b=3;，则(p1+3)=&b;是错误的，p1+3 的结果是右值，但*(p1+3)=3;是正确的。

（5）若两个指针指向同一个数组中的元素，或者指向数组中最后一个元素的下一个位置，则 C++还支持这两个指针做相减运算。若两个指针指向的不是同一个数组中的元素或数组中最后一个元素的下一个位置，则其行为是未定义的。

（6）若两个指针指向同一个数组中的元素，那么两个指针相减的结果是两个数组元素的下标之差，结果可能是负数，结果类型是<cstddef.h>中所定义的 ptrdiff_t 类型，其实 ptrdiff_t 类型就是 signed 整型。两个指针之差的结果值应保证在 ptrdiff_t 所规定的类型的取值范围之内。比如 p1 指向 a[i]，p2 指向 a[j]，则 p1–p2 的值就等于 i–j 的值。

（7）注意：两个指针之差是两个数组元素的下标之差，其单位并不是字节，而是以数组元素的长度为单位，结果类型是整型。这个差值并不是指两个数组元素（或两个指针）之间相差的宽度，两个数组元素之间相差的宽度是这个差值乘以数组类型的长度。比如 p1 指向 a[1]，地址值为 0x0000ff61；p2 指向 a[3]，地址值为 0x0000ff69，假设其类型为 int（占 4 个字节），则 p2–p1 的结果是(0x0000ff69–0x0000ff61)/4=2，这个“2”指的是整数值 2，而不是指 p2 和 p1 相差 2 个字节，而是相差 2×4=8 个字节的内存宽度。

（8）多级指针加上或减去一个整数值 *n*，同样会使指针向前或向后偏移“*n**指针指向类型所占据的字节数”这么多个字节。比如 double ***p;，假设 p 已经正确赋值，指针在内存中占据 4 个字节的长度，double 类型占据 8 个字节。则 p+1;将使指针向后偏移 4 个字节（一个指针类型的大小），因为三级指针 p 所指向的类型是一个“指针（地址）类型”，指针类型占据 4 个字节的大小，因此 p+1 只向后偏移 4 个字节，而不是 double 类型的 8 个字节。同理，*p+1;也将向后偏移 4 个字节，因为*p 指向的类型仍然是“指针类型”。但是**p+1;将向后偏移 double 类型的 8 个字节，因为**p 指向的类型是 double 类型。

（9）注意：不同类型的指针之间不能相减，比如 int *p1; short int *p2;，则 p2–p1 是错误的。

3．指针的关系运算

（1）当指针都指向同一个数组或数组中最后一个元素的下一个位置，或者聚集类型中的元素时，可以对两个指针进行关系运算（即>、 <、 >=、 <=）。对于其他类型指针间的关系运算标准中没有定义，尽管现在大多数编译器可以对其他类型指针进行关系运算，但并不保证所有编译器都可以进行运算。

（2）可以在任意指针间进行相等或不相等运算，因为指针要么指向同一个地址，要么指向不同的地址。

示例 6.3：指针的算术及关系运算

```
#include <iostream>
using namespace std;
int main()
{    int a1=1;     int a2=2;     int *p1=&a1;       int *p2=&a2;
     //p1*p2;                  //错误，指针的算术运算只限于指针与整数相加减、指针与指针相减
     //p1+p2;                  //错误，两个指针不能相加
     //cout<<p1-p2<<endl;      //标准中未定义的行为。若两个指针所指向的不是同一个数组中的元素或数组中
                               //最后一个元素的下一个位置，则其行为是未定义的
                               //指针加上或减去一个整数值 n，是加上或减去指针所指向类型的大小再乘以整
                               //数值 n，这样可以确保指针指向的一定是下一个对象，而不是对象内部（对象
                               //一般不止占据一个内存单元）的某个位置

     cout<<p1<<endl;
     cout<<p1+2<<endl          //该行为是未定义的，因为指针的算术运算不是在数组内部进行的
                               //这里只是为了说明指针的运算，可以看到指针偏移了 4×2=8 个字节
                               //而不是简单地将地址值加上 2
     cout<<(p1>p2)<<endl;      //该行为是未定义的，进行关系运算时 p1 和 p2 应指向同一个
                               //数组或数组中最后一个元素的下一个位置，或者聚集类型中的元素
     cout<<(p1==p2)<<endl;     //正确，可以在任意指针间进行相等与不相等运算
     int a3[33]={1,2,3,4};int a4=4;  int *p3; p3=a3;
     int *p4=&a3[2];           //指针 p4 指向数组第 3 个元素的地址
     //(p3+3)=&a4;             //错误，p3+3 的结果虽然是指针，但这个指针是右值
     *(p3+3)=4;                //正确，可以通过解引用操作访问算术运算后产生的新指针所指向的内容
                               //解引用运算符的结果是左值
     cout<<p3+33<<endl;        //正确，算术运算后的新指针指向数组中最后一个元素后面
                               //的那个位置则是合法的
     //*(p3+33)=4;             //错误，虽然进行算术运算后允许指针指向数组中最后一个元素后面
                               //的那个位置，但对其进行解引用操作则可能会产生意想不到的结果
                               //因为并不清楚该位置以前存储的是什么内容
     cout<<*(p3+2)<<endl;      //输出 3，p3+2 新产生的指针向后偏移两个位置，新指针指向数组的第三个元素
     cout<<p4-p3<<endl;  }     //输出 2，若两个指针指向同一个数组中的元素，则两个指针相减的
                               //结果是两个数组素元的下标之差，结果可能是负数，其单位不是字节，
                               //而是数组元素的长度，因此 p4-p3 的结果为 2，但并不是指 p4 和 p3 之间
                               //只相差 2 个字节的内存宽度，而是相差 8 个字节
```

6.2 数组

6.2.1 一维数组

（1）数组是一个存储单一数据类型对象的集合。

（2）数组的元素：数组中存储的每个对象被称为数组的元素，数组的元素都具有相同的数据类型。

（3）数组由数组名（标识符）、类型名和数组的维数组成。数组的维数确定了数组的元素个数。

1．数组的声明

（1）声明数组的简单格式为：

```
类型区分符 数组名[整型常量表达式][整型常量表达式]...;
```

（2）以上各部分都是必选项，包括其中的中括号“[]”。声明时的“[]”数量指定了数组的维数，若声明时有两个“[]”则是二维数组，有三个“[]”则是三维数组，有 N 个“[]”则是 N 维数组。比如 int b[2][3];，表示声明一个拥有 2 行 3 列的二维数组，数组中的每个元素都是 int 类型的。

（3）与声明变量一样，声明数组时也必须指定类型区分符，类型区分符指明了数组中元素的类型。当然，声明数组时还可以指定类型限定词和存储类区分符，它们都与声明变量时相同，这里就不重复讲解了。

（4）数组名之后的中括号“[]”是声明数组的标志，该中括号不能省略。

（5）数组的大小：“[]”中的整型常量表达式指定了数组存储元素的个数，一般被称为数组的大小。比如 int a[11];，表示声明了一个数组，这个数组拥有 11 个元素，每个元素的类型为 int。

2．声明数组的限制

（1）在声明数组时，中括号中的数值必须是整型常量表达式。比如 int a[2.2]是错误的，因为 2.2 是浮点数；int b=1; int a[b];也是错误的，因为 b 是变量，不是整型常量表达式。

（2）在声明未显式初始化的数组时，必须指定数组的大小，且数组的大小必须大于或等于 1。比如 int a[];、int a[0];都是错误的，但 int a[]={1,2};是正确的，因为进行了显式初始化。

（3）数组元素的类型不能是引用，也不能是函数，但可以是另一个数组（即二维数组）。比如 int &a[3];、int a[3]();错误；int a[3][3];、 int (*a[3])();正确。

（4）示例如下：

```
int [2]s;    //错误，声明数组时，数组声明符“[]”应在数组名之后
int s[3];    //正确，声明一个存储 3 个 int 类型元素的数组，其数组名为 s
int s[0];    //错误，声明数组时，数组的大小必须大于或等于 1
int s[];     //错误，必须指定数组的大小
int &a[3];  //错误，数组元素的类型不能是引用、函数
```

3．使用下标运算符访问数组中的元素

（1）使用下标运算符访问数组中的元素，其形式为：

```
数组名[下标值][下标值]...
```

（2）以上各项均为必选项，包括其中的方括号“[]”。

（3）数组中的元素可以通过下标运算符“[]”进行访问，比如a[0]表示数组a中第一个元素的值，a[1]表示数组a中第二个元素的值等，下标值标明了该元素所在数组中的位置。

（4）注意：数组的下标值是从0开始的，而不是从1开始的。比如要访问数组a中的第一个元素可以使用a[0]，访问第二个元素可以使用a[1]。

（5）与声明数组时不同，访问数组中的元素时其下标值不必是整型常量，但必须是整型表达式。比如int a=1; float b=2;int s[7]={0};，则b=s[2]+s[2+3]+s[a+1];中的s[2]、s[2+3]、s[a+1]都是正确的访问数组元素的方法，但s[b]、s[1+b]则是错误的，因为b不是整型，同样1+b的结果也不是整型。

（6）注意：编译器不会检查数组的下标值是否在数组的有效范围之内，若下标值超过数组的有效范围，则可能产生意想不到的结果，因为并不清楚在该内存位置存储的是什么值。比如数组a只拥有11个元素，但在程序中使用了a[15]来访问数组范围之外的元素，则可能会产生意想不到的结果。

（7）数组的长度是固定的，一旦确定了数组的大小，就无法改变。

4．初始化数组的基本规则

（1）初始化列表是由大括号“{}”括起来的一组初始值，每个初始值之间使用逗号隔开，其形式有两种：{初始值1,初始值2,…,初始值N}和{初始值1,初始值2,...,初始值N,}。第二种形式在最后多了一个逗号，这个逗号在初始化列表中是没有多少实际意义的，因此在程序中使用第一种形式居多。

（2）初始化列表还可以嵌套，比如{{初始值1,初始值2},{初始值11,初始值22,…},…}。

（3）数组应使用初始化列表进行初始化，初始化列表中的初始值分别对应于数组中的各个元素。注意，初始化列表之后应有一个分号。比如int a[3]={1,2,3};，则数组中第一个元素a[0]的值被初始化为1，a[1]的值为2，a[2]的值为3。再比如int a[1]=2;是错误的，因为数组应使用初始化列表进行初始化，正确形式是int a[1]={2};。

（4）在初始化列表中初始值的数目不能大于需要初始化的数组元素的数目。比如int a[2]={1,2,3};是错误的。

（5）若初始化未知大小的一维数组，则数组的大小由初始化列表所提供的初始值的数目确定，数组初始化之后，该数组不再是不完整类型的。比如int a[]={1,2,3};，则数组a拥有3个元素，初始化之后数组a不再是不完整类型的。

（6）若初始化列表中的初始值数目少于数组的元素个数，则数组中其他元素的值被初始化为0。比如int a[11]={1,2};，则a[0]的值为1，a[1]的值为2，a[2]到a[10]的值均为0。

（7）int a[11]={0};或者int a[11]={};都会把数组a中的所有元素初始化为0，但是int a[11]={1};并不是把数组中的所有元素都初始化为1，而是只有数组的第一个元素a[0]的值被初

始化为 1，其余元素的值为 0。

（8）当对数组进行赋值时，不能使用初始化列表，初始化列表只能用于对数组的初始化，因此只能对数组元素逐个进行赋值，一般采用循环语句进行赋值；同理，当需要输出数组中的所有元素值时，也应逐个元素进行输出。比如 int a[3];，则 a={1,2,3}或者 a[2]={1}; a[3]={1,2,3}; 等都是错误的语法。

（9）不能使用一个数组直接初始化另一个数组，也不能将一个数组赋给另一个数组。要把一个数组赋给另一个数组，必须对其每个元素逐一进行赋值。比如 int a[3]={1,2,3};，则 int b[3]=a; 是错误的，而 b[0]=a[0]; b[1]=a[1]; b[2]=a[2];是正确的。

（10）若数组是静态存储期的，则初始化只进行一次；若未对其进行显式初始化，则自动将数组中的各个元素初始化为 0。若数组是自动存储期的，且未显式初始化，则值是不确定的，值是由系统随机分配的。

（11）在初始化数组时，不能使用添加逗号的方式跳过要初始化的元素，以使其自动初始化为 0，而应显式初始化为 0。比如 int a[3]={1, , 3}; 是错误的。

（12）若声明数组时没有对其进行初始化，则在以后只能对数组元素逐个进行赋值。

（13）对于字符数组，可以使用初始化列表进行初始化，也可以使用字符串的形式进行初始化，使用字符串的形式进行初始化时，应注意字符串末尾有个空字符。比如 int a[3]={'c','+','+','\0'}; 和 int a[4]="c++";相同，后者的长度为 4，因为字符串是以空字符'\0'结尾的。

示例 6.4：初始化数组的基本规则

```
#include <iostream>
using namespace std;
int main()
{    int b1=1;
     //int c1[1]=4;          //错误，虽然数组只有一个元素，也应使用初始化列表进行初始化
     //int c2[3]={1,2,3,4}; //错误，初始化列表中初始值的数目不能大于需要初始化的数组元素的数目
     int c3[]={1,2,3};       //正确，初始化未知大小的数组时，其大小由初始化列表提供的初始值的数目确定
     cout<<c3[0]<<c3[1]<<c3[2]<<endl;  //输出 123，注意：数组的下标值是从 0 开始计算的
     cout<<c3[b1]<<endl;     //正确，使用下标值访问数组元素时，只要求是整型表达式即可
     int c4[5]={1,2,3};      //正确，初始化列表中的初始值数目少于数组的元素个数时，数组中的其他元
                             //素都被初始化为 0
     cout<<c4[0]<<c4[1]<<c4[2]<<c4[3]<<c4[4]<<endl;      //输出 12300，初始化列表中的初始值分
                                                         //别对应于数组中的各个元素
     //c4[3]={1};            //错误，当对数组进行赋值时，不能使用初始化列表
     //int c5[5]=c4;         //错误，不能使用一个数组直接初始化另一个数组
     int c6[5];
     //c6=c4;                //错误，不能将一个数组赋给另一个数组
     //int c7[6]={1,,3};     //错误，不能使用添加逗号的方式跳过要初始化的元素，以使其自动初始化为 0
     int c8[11]={},c9[4]={0}; //把数组中的所有元素初始化为 0
     int c10[4]={1};         //只有数组的第一个元素 c10[0]的值被初始化为 1，其余元素的值为 0
     int c11[]={};       }   //错误，此语句会认为数组的大小为 0
```

6.2.2 多维数组

1. 对多维数组的理解

（1）多维数组是数组中的数组：C++中没有多维数组类型，对于二维及多维数组，其实就是数组中的数组，多维数组的元素就是所有第一维的元素，只是这一维的元素又是数组（或多维数组）而已，因此除第一维外的其他维的长度都是多维数组中元素的一部分。在理解多维数组时，可按一维数组的方式进行拆分理解。比如 a[2][3][4]，这个数组只有第一维指定的 2 个元素即 a[0] 和 a[1]，每个元素又是一个二维数组，这个二维数组有 3 个元素，每个元素又是一个有 4 个元素的一维数组；再比如 a[4][3][5][6]，这个数组共有 4 个元素，每个元素又是一个三维数组，该三维数组也可以按这种方法理解，即该三维数组有 3 个元素，每个元素又是一个二维数组；同理，这个二维数组共有 5 个元素，每个元素又是一个有 6 个元素的数组。

（2）二维数组的简单理解：对于二维数组，可认为是一个由 *N* 行 *M* 列组成的表格，数组的第一维指定行数，第二维指定列数，比如 int a[3][4];可以看成是拥有 3 行 4 列的一个表格。

（3）多维数组元素的存储次序：多维数组是按照行主序的原则进行存储的，即先按照最右边的下标先进行存储。比如 int a[2][3][4];，在内存中存储的顺序依次是 a[0][0][0], a[0][0][1], a[0][0][2], a[0][0][3], a[0][1][0], a[0][1][1], a[0][1][2], a[0][1][3]……对于二维数组，其在内存中是按照先行的顺序进行存储的。比如 int a[3][4];，则数据在内存中存储的顺序是先存储第一行的 4 个元素即 a[0][0], a[0][1], a[0][2], a[0][3],然后是第二行的 a[1][0], a[1][1], a[1][2], a[1] [3];。

（4）使用下标运算符访问 *N* 维数组中的具体元素时应使用 *N* 个下标运算符，若使用的下标运算符少于 *N* 个，则表示的是指向某处的指针，即某处的地址。比如 int a[2][3][4];假设已初始化，应使用 3 个下标运算符访问具体的元素，如 a[0][1][1], a[1][2][3]等都能访问到具体的元素，而 a[1][2], a[1], a[0][1]等都是指向某处的指针。

2. 初始化多维数组的规则

（1）二维及多维数组的初始化与一维数组一样，都需要使用初始化列表进行初始化。

（2）对于多维数组，除第一维外的其他维都是其元素类型的一部分，因此在初始化多维数组时，除第一维外的其他维是不能省略的，第一维的长度是可以省略的，省略的维数通过初始化列表的内容确定。比如 a[][2]={1,2,3};，第一维的长度应该是 2；同理，对于 a[][2]={{1},{2,3},{4}};，第一维的长度是 3；但 a[2][]={1,2}是错误的，因为除第一维外的其他维是不能省略的。

（3）多维数组的初始化。

① 顺序初始化：即像初始化一维数组一样，将要初始化的值逐个填在初始化列表中，程序会按照最右边的下标先初始化（行主序）的原则进行初始化，若初始值的数目少于元素的个数，

则剩余元素的值被初始化为 0。比如 int a[2][3][4]={1,2,3,4,5,6,7,8,9};，依次先初始化最右边的下标，即 a[0][0][0]=1; a[0][0][1]=2; a[0][0][2]=3; a[0][0][3]=4; a[0][1][0]=5; a[0][1][1]=6; a[0][1][2]=7; a[0][1][3]=8; a[0][2][0]=9; 其余元素被自动初始化为 0。

② 按行初始化：即使用嵌套的初始化列表对多维数组进行初始化，初始化二维数组需要两层初始化列表，初始化三维数组则需要三层初始化列表，依此类推。嵌套的初始化列表之间以逗号分隔，在本文中内层的初始化列表称作子初始化列表。比如 int a[2][3]={{1,2,3},{4,5,6}};，初始化后的结果是 a[0][0]=1; a[0][1]=2; a[0][2]=3; a[1][0]=4; a[1][1]=5; a[1][2]=6;。

③ 在按行初始化时，程序同样是按照最右边的下标先初始化（行主序）的原则进行初始化的，若将多维数组理解为数组的数组，则把按行初始化的原则理解为先初始化多维数组的第一个元素，然后是第二个元素……更为贴切，只不过多维数组中的元素仍是数组。

④ 子初始化列表中初始值的数目不能大于需要初始化的子数组元素的数目，子初始化列表的数目不能大于需要初始化的数组的相应维数的长度。若子初始化列表的数目少于需要初始化的数组的相应维数的长度，则其余值为 0。比如 int a[2][3]={{1,2},{3},{4}};错误，因为子初始化列表的数目（共有 3 个子初始化列表）大于相应的维数数目，即第一维的长度 2；int a[2][3]={{1,2,3,4},{5}}; 错误，因为第一个子初始化列表中共有 4 个初始值，数目大于需要初始化的子数组第二维的长度 3。

⑤ 在初始化数组时，不推荐既使用顺序初始化又使用按行初始化方法，因为这样做程序既容易出错又不方便理解。比如 int a[2][3]={1, {2,3}};是错误的。

⑥ 示例：int a[2][3]={{1},{2,3}};

- 子初始化列表为{1}和{2,3}，且子初始化列表的数目与需要初始化的二维数组的第一维的长度 2 相等，所以使用相应的子初始化列表初始化二维数组对应的元素。
- 使用子初始化列表{1}初始化二维数组第一维的第一个元素即 a[0]，这个元素又是一个有 3 个元素的数组，这 3 个元素分别是 a[0][0], a[0][1], a[0][2]。因为子初始化列表中初始值的数目少于二维数组第一维第一个元素数组所拥有的元素个数，所以 a[0][1]和 a[0][2]初始化为 0，初始化后的结果是 a[0][0]=1; a[0][1]=0; a[0][2]=0;。
- 同理，使用{2,3}初始化二维数组第一维的第二个元素即 a[1]，它同样是有 3 个元素的数组，其结果是 a[1][0]=2; a[1][1]=3;，其余项为 0。
- a 的最终结果是：a[0][0]=1; a[0][1]=0; a[0][2]=0; a[1][0]=2; a[1][1]=3; a[1][2]=0;。

⑦ 示例：int b[3][2]={{1}, {2,3}};

- 子初始化列表为{1}和{2,3}，且子初始化列表的数目少于需要初始化的二维数组第一维的长度 3，所以使用相应的子初始化列表初始化二维数组对应的元素，其余的值为 0。
- 使用子初始化列表{1}初始化二维数组的第一个元素 b[0]，这个元素又是一个有两个元素的数组，因此其结果是 b[0][0]=1; b[0][1]=0;。

- 使用子初始化列表{2,3}初始化二维数组的第二个元素 b[1]，这个元素也是一个有两个元素的数组，其结果是 b[1][0]=2; b[1][1]=3;。
- 因为子初始化列表的数目少于需要初始化的二维数组第一维的长度 3，因此二维数组的第三个元素 b[2]全部被初始化为 0，即 b[2][0]=0; b[2][1]=0;。
- b 的最终结果是：b[0][0]=1; b[0][1]=0; b[1][0]=2; b[1][1]=3; b[2][0]=0; b[2][1]=0;。

示例 6.5：多维数组的初始化

```
#include <iostream>
using namespace std;
int main()
{    //int a1[][5], a2[3][];   //未初始化的声明，必须指定数组的大小
     //int a3[3][]={0};        //错误，初始化多维数组时，除第一维外的其他维是不能省略的
     int a4[][2]={1,2,3};  //正确，初始化多维数组时，第一维是可以省略的，结果是 a4 的第一维长度为 2
     cout<<a4[0][0]<<a4[0][1]<<a4[1][0]<<a4[1][1]<<endl; //输出 1230
     int a5[][2]={{1},{2,3},{4}}; //正确，初始化多维数组时，第一维可以省略，结果是第一维长度为 3
     cout<<a5[0][0]<<a5[0][1]<<a5[1][0]<<a5[1][1]<<a5[2][0]<<a5[2][1]<<endl;
                                                                    //输出 102340
     //int a6[2][2]={1,2,3,4,5};   //错误，初始值数目过多
     //int a7[2][2]={{1,2,3},{4,5}}; //错误，第一个嵌套的初始化列表的初始值数目过多
     //int a8[2][2]={{1},{2},{3}};   //错误，子初始化列表的数目过多
}
```

6.3 指针与数组

要掌握好指针，需要清楚指针指向的类型及指针指向何处。复杂的指针一般与数组脱不了关系，其次就是二级及多级指针。本节重点讲解指针与数组的关系。

6.3.1 理解数组名

注：为方便讲解，本节所说的指针的类型，指的是指针指向的类型。

1. 对数组名的简单理解

数组名表示的是数组的首地址。比如一维数组 int a[11]的数组名 a 就表示数组的首地址。这里的首地址让人产生误解，认为数组名表示的是整个一维数组的地址，其实数组名指向的是数组中第一个元素地址的指针（即 a 指向的地址是&a[0]），表示整个数组的地址是&a，&a 表示的是一个包含有 4 个元素的一维数组的地址。虽然 a 和&a 的地址值是相同的，但是它们的类型不一样，a 的类型是“指向 int 的指针”，而&a 的类型是“指向一个有 11 个元素的一维数组的指针”。

2．对数组名的理解

（1）对数组名的理解，关键是要明白数组名所表示的指针的类型。

（2）重要规则：数组名是指向第一个元素地址的指针，这个指针的类型是“指向某类型的指针”，这个指针是个常量。比如 int a[4]={0};，数组名 a 是一个指向第一个元素 a[0]地址的指针（即 a 指向的地址是&a[0]），它的类型是“指向 int 的指针”，这个指针是个常量，即不能改变它的值；再比如 int b=1;，则 a=&b;是错误的。

（3）对多维数组的数组名的理解：对于多维数组，重点是要理解它的第一个元素是什么。通常讲的多维数组其实就是数组中的数组，即元素为数组的数组。因为数组名是指向数组中第一个元素地址的指针，所以多维数组的数组名指向的也是第一个元素地址的指针，但这个元素又是一个数组或多维数组。因此对于 N 维数组，数组名是指向第一个元素地址的指针，这个指针的类型是“指向 N–1 维数组的指针”，指向的数组有 N–1 维，每维有相对应的 X 个元素，每个元素都是某类型的（如 int）。而一维数组的数组名的类型一般是“指向某类型的指针”。

示例如下：

- int a[3];，数组名 a 指向的是第一个元素地址&a[0]的指针，其类型是“指向 int 的指针”。
- a[2][3];，数组名 a 指向的是第一个元素 a[0]地址的指针（即 a 指向的地址是&a[0]），这个元素是一个有 3 个元素的一维数组，因此数组名是一个指向有 3 个元素的数组的指针，即数组名与指针(*p)[3]的类型相同，它的类型是“指向一维数组的指针”。这个一维数组有 3 个元素，即数组名代表的地址是第一个元素的地址&a[0]，这与一维数组的数组名是相同的。
- b[3][4][5]={0};，数组名 b 是指向第一个元素 b[0]地址的指针，即 b 指向的地址是&b[0]。这个元素存储的是 4×5 的二维数组，因此这个指针的类型是“指向二维数组的指针”，指向的数组有两维，第一维有 4 个元素，第二维有 5 个元素。注意：多维数组是数组的数组，三维数组 b 的第一个元素是 b[0]，而不是 b[0][0][0]。

（4）从以上概念可以看出，多维数组的数组名所表示指针的类型与多维数组的第一维的大小是没有关系的，只与除第一维外的其他维的大小有关系，但这并不意味着第一维的大小就没有作用了。

（5）对数组名取址的运算：若对数组名进行取址运算，则产生的结果是指向整个数组的指针，即对于一维数组名，取址后的类型是“指向一维数组的指针”，这个数组有 X 个元素；对 N 维数组名取址后的类型是“指向 N 维数组的指针”，这个 N 维数组的每一维有 X 个元素。比如 int a[2][3][4]={0};，则&a;的类型是“指向三维数组的指针”，这个数组的第一维有 2 个元素，第二维有 3 个元素，第三维有 4 个元素。

（6）注意：早期 C 语言没有“数组的地址”这一概念，所以对数组名取址要么非法，要么

就等于数组名本身。

（7）数组名的类型相同，应满足以下条件。

① 对于 N 维数组，声明时的元素类型要相同。

② 数组名的类型应指向相同维数的数组。

③ 指向的数组的每一维的元素个数必须相同。

④ 示例：int a[2][3][4]; int b[5][3][4]; int c[2][3][5]; float d[2][3][4]

- 数组名 a 和 b：所表示的指针具有相同的类型，a 和 b 的元素的类型相同，都是 int 类型。且 a 和 b 的类型都是“指向二维数组的指针”，指向的数组的第一维都有 3 个元素，第二维都有 4 个元素。
- 数组名 a 和 c：所表示的指针具有不相同的类型，虽然它们都是指向二维数组的指针且元素的类型相同，但数组名 a 指向的二维数组的第二维有 4 个元素，而 c 指向的二维数组的第二维有 5 个元素，即它们所指向的数组的每一维的元素个数不相同。
- 数组名 a 和 d：具有不相同的类型，虽然它们都是指向二维数组的指针，且每一维相对应的元素个数也相等，但 a 和 d 的元素类型不相同，a 的元素类型是 int，而 d 的元素类型是 float。

3. 数组名与指针的关系

（1）对于 N 维数组 a,其数组名 a 与&a[0],&a[0][0]，&a[0][0]...[N]的地址值是相同的，也就是说，指向的是相同地址的指针。但要注意，这些指针的类型并不相同，虽然数组名 a 和&a[0]类型相同，都是“指向 N–1 维的指针”，&a[0][0]和 a[0]类型相同都是“指向 N–2 维的指针”，但是&a[0]和&a[0][0]类型并不一样，因为对指针进行运算时，指针偏移多少个字节是由其类型决定的，所以在进行运算时它们的偏移量是不一样的。比如有二维数组 int a[2][3]，假设 int 类型占据 4 个字节的内存空间，则&a[0]所指向对象的长度占据 12 个字节的内存空间（因为它指向的对象是具有 3 个元素的数组），而&a[0][0]所指向对象的长度只占据 4 个字节的内存空间，对其进行算术运算时，&a[0]+1 将移动 12 个字节，而&a[0][0]+1 则只移动 4 个字节。对于 N 维数组也是同样的道理。

（2）对 N 维数组的数组名做加法运算，将使其指向 N 维数组中第 X 个元素的地址。比如对于 N 维数组 a，可以使用&a[0]+1 或 a+1 来表示指向第二个元素的地址&a[1]。

（3）对于 N 维数组，直接书写成 M（$0<M<N$）维数组，当对其进行指针偏移运算时，并不能像数组名那样直接进行偏移，而应将其替换为指向的实际地址再进行偏移，否则容易出错。比如 int a[3][4][5][6]={0};，则 a[1]指向的是&a[1][0]，当进行 a[1]+1 偏移运算时，并不是指向 a[2]，而是指向&a[1][1]。也就是说，a[1]+1 并没有指向地址&a[2][0]，因为 a[1]+1=&a[1][0]+1=&a[1][1]（具体请参阅后文中的“指针与数组的混合运算”）。

4．判断指针指向的是几维数组的指针

（1）对于 N 维数组，若直接书写成 M（$0<M<N$）维数组，则指针是“指向 $N–M–1$ 维的指针”。指向的数组有 $N–M–1$ 维，每一维有相对应的 X 个元素，指向的地址是 M 维之后第一个元素的地址。比如 int a[2][3][4][5][6][7]={0};，则 a[1][2][3]的指针指向的是二维数组（6–3–1）的指针，这里 a 是六维数组，即 N=6，书写成三维数组，即 M=3。再比如 int a[3][4][5][6]={0};，则 a[2]是指向&a[2][0]的指针，它的类型是“指向二维数组(4–1–1)的指针”，这个二维数组的第一维有 5 个元素，第二维有 6 个元素；a[1]指向的是&a[1][0]，类型是“指向二维数组的指针”，这个二维数组的第一维有 5 个元素，第二维有 6 个元素，可以看到 a[2]与 a[1]的类型是相同的，但指向的地址值不相同。

（2）对于 N 维数组，若书写成带有&（取址）运算符的 M 维数组形式时，则指针指向的是“$N–M$ 维的指针”，或者说 M 维之后还有多少维就是指向多少维的指针。比如 int a[2][3][4][5][6][7]={0};，则&a[1][2]指向的是四维数组（6–2）的指针，这里 a 是六维数组，即 N=6，书写成带取址运算符的二维数组，因此 M=2；或者说 a 是六维数组，而&a[1][2]只写了二维，二维之后还有四维，因此&a[1][2]是指向四维数组的指针。

5．数组名与指针的不同

（1）当数组名作为 sizeof 的操作数时，返回的是整个数组的长度，而不是指向数组的指针的长度（在 32 位机器上指针的长度一般为 4 个字节）。比如 int a[5]={0};，则 sizeof(a);的结果是 20（假设 int 类型占 4 个字节），而不是指针的长度 4，因为 a 有 5 个元素，每个元素占 4 个字节；再比如 int a[3][4]={0};，则 sizeof(a);的结果是 48，即 3×4×4=48。

（2）当数组名作为&的操作数时，所产生的将是一个“指向一维或多维数组的指针”，而不会返回指针本身的地址。比如 int a[4];，则&a;将产生一个“指向一维数组的指针”，这个一维数组有 4 个元素。

（3）注意：数组名虽然是一个指针，但它与指针并不完全相同，比如数组名作为 sizeof 和&的操作数时。

6.3.2　指针与数组的混合运算

注意：指针运算符“*”的优先级低于下标运算符“[]”。

1．下标运算符与指针的关系

（1）数组名既然是指向数组中第一个元素的指针，那么就意味着可以使用指针的方式访问数组中的元素。比如 int a[3]={1,2,3};，则*(a+1)同样是表示访问数组的第二个元素，即*(a+1)与 a[1]相等，因为 a 是指向数组中第一个元素的指针，对指针加 1 将使指针指向数组的下一个

元素，然后对其进行解引用，就访问到了数组的第二个元素。

（2）下标运算符的定义（ANSI C89）：E1[E2]与(*(E1+(E2)))是完全相同的，若 E2 是整数，则 E1[E2]表示的是 E1 的第 E2 个元素（下标从 0 开始）。比如 a[3]与(*(a+(3)))完全相同，a[2+3]与(*(a+(2+3)))相同。可以看到，对于数组和指针的运算，可以简单地利用公式E1[E2]=(*(E1+(E2)))进行替换。

（3）由下标运算符的定义可以得出若 a 是一维数组，则 2[a]等价于(*(2+(a)))。但不推荐这种写法，因为会降低程序的可读性。

2．指针运算符与下标运算符的原理

（1）指针运算符“*”的作用是求出“*”后面所指地址中的值，因此只要“*”后面的变量表示的是一个地址，就可以使用“*”运算符求出这个地址中的值，而不用管这个地址的表示形式是怎样的。

（2）“[]”运算符的运算法则：把左侧的地址加上“[]”内的偏移量，然后再进行指针运算，即 E1[E2]与(*(E1+(E2)))是完全相同的。注意：有“[]”运算符的地方就有隐含的指针，比如 x[2]表示将指针 x 偏移 2 个单位量后再进行指针运算，即 x[2]与*(x+2)是相等的。

（3）指针与数组混合运算时，也可以直接套用公式 E1[E2]=(*(E1+(E2)))。但直接套用公式容易出错，特别是将(*(E1+(E2)))换算为 E1[E2]时更易出错。

3．指针与数组混合运算的规则

（1）对于 *N* 维数组，应把使用下标表示的指针替换为指针指向的实际地址后再进行运算，这样才方便使用下标运算符进行指针偏移的直接运算。因为数组名指向的是数组中第一个元素地址的指针，所以对于 *N* 维数组 a，a[]...[M]（假设有 *M* 个下标且 *M*<*N*）都应转换为地址 &a[]...[M][0]后再进行运算。注意：若下标有 *N* 个，则表示的是实际元素值，而不是一个指针了，这时只需将实际的元素值与相加的值相加即可。比如对于二维数组 int b[3][4]，则 b[1]应表示为 &b[1][0]后再进行偏移运算，如 b[1]+1=&b[1][0]+1=&b[1][1]，b[1]+1 不等于 b[2]，也就是 b[1]+1 不是直接对 b[1]加 1，因为 b[1]表示的是第二行第一个元素的地址，对其加 1，应该表示的是第二行第二个元素的地址，也就是&b[1][1]，而 b[2]表示的是第三行第一个元素的地址，因此错误。所以在计算时应把 b[1]转换为&b[1][0]之后，才能直接进行地址的偏移，即 b[1]+1=&b[1][0]+1=&b[1][1]，这样才能得到正确的结果，并且不会出错。

（2）对于有小括号的地方，一定不要省略小括号，且应严格按照公式 E1[E2]=(*(E1+(E2)))进行换算。比如有二维数组 int b[3][4]，则(&b[1])[1]与&b[1][1]表示的是不同的结果，因为(&b[1])[1]=*((&b[1])+1)=*(&b[1]+1)=*(&b[2])=b[2]，表示的是第三行第一个元素的地址。因此对于(b+1)[1]这样的运算，不能省略小括号，即(b+1)[1]=(&b[1])[1]=*((&b[1])+1)=*(&b[1]+1)=*(&b[2])=b[2]，如果省略了小括号，则是(b+1)[1]=&b[1][1]，将产生不易发现的错误。

（3）注意：对于有指针运算符“*”和下标运算符“[]”的混合运算，最好使用公式E1[E2]=(*(E1+(E2)))采用数学方式来得出正确的结论。若用语言来描述这类综合运算，则很容易出错；如果使用数学方式，结合运算符优先级，弄懂指针的首地址，然后从优先级高的运算符开始运算，这样得到的结果是不会出错的。比如对于(*(b+1))[1]，假设 b 是二维数组 b[3][4]，只要弄懂数组 b 表示的类型（或地址），也就是 b=&b[0]，然后再运算，就很容易得到答案了。

（4）示例（假设 b 为二维数组 b[3][4])如下：

① 方法 1：b[1][2]=*(b[1]+2)=*(*(b+1)+2)

方法 2：b[1][2]=(*(b+1))[2]=*(*(b+1)+2)

② (&b[1])[1]=*((&b[1])+1)=*(&b[1]+1)=*(&b[2])=b[2]

③ *b = *(b+0)=*(&b[0])=b[0]=&b[0][0]

④ *b[1]=*(b[1]+0)=*(&b[1][0])=b[1][0]

⑤ *(b+1)=*(&b[0]+1)=*(&b[1])=b[1]

⑥ *(*(b+1)+1)=*(*(&b[0]+1)+1)=*(*(&b[1])+1)=*(b[1]+1)=*(&b[1][0]+1)=*(&b[1][1])=b[1][1]

⑦ (*b)[1]=(*(b+0))[1] = (*(&b[0]+0))=(*(&b[0])[1]= (b[0][1]= b[0][1]

⑧ (*(b+1))[1]=(*(&b[0]+1))[1]=(*(&b[1]))[1]=(b[1])[1]=*((b[1])+1)=*(&b[1][0]+1)=*(&b[1][1])=b[1][1]

⑨ (b+1)[1]=(&b[0]+1)[1]=(&b[1])[1]=*(&b[1]+1)=*(&b[2])=b[2]

⑩ 方法 1：*(b+1)[1]=*(&b[0]+1)[1]=*(&b[1])[1]=*(*(&b[1]+1))=*(*(&b[2]))=*(b[2])=*(&b[2][0])= b[2][0]

方法 2：*(b+1)[1]=*(*((b+1)+1))=*(*(b+2))=*(*(&b[0]+2))=*b[2]=b[2][0]

⑪ (*b+1)[2]=(b[0]+1)[2]=(&b[0][0]+1)[2]=(&b[0][1])[2]=*(&b[0][1]+2)=*(&b[0][3])= b[0] [3]

示例 6.6：对以上 11 个示例的验证

```
#include <iostream>
using namespace std;
int main()
{    int b[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
     cout<<"例 1: "<<b[1][2]<<"="<<*(*(b+1)+2)<<endl;     //输出 7=7
     cout<<"例 2: "<<(&b[1])[1]<<"="<<b[2]<<endl;         //输出两个相等的地址
     cout<<"例 3: "<<*b <<"="<<b[0]<<endl;                //输出两个相等的地址
     cout<<"例 4: "<<*b[1]<<"="<<b[1][0]<<endl;           //输出 5=5
     cout<<"例 5: "<<*(*(b+1)+1)<<"="<<b[1][1]<<endl;     //输出 6=6
     cout<<"例 6: "<<*(b+1)<<"="<<b[1]<<endl;             //输出两个相等的地址
     cout<<"例 7: "<<(*b)[1]<<"="<<b[0][1]<<endl;         //输出 2=2
     cout<<"例 8: "<<(*(b+1))[1]<<"="<<b[1][1]<<endl;     //输出 6=6
     cout<<"例 9: "<<(b+1)[1]<<"="<<b[2]<<endl;           //输出两个相等的地址
     cout<<"例 10: "<<*(b+1)[1]<<"="<<b[2][0]<<endl;      //输出 9=9
     cout<<"例 11: "<<(*b+1)[2]<<"="<<b[0][3]<<endl;}     //输出 4=4
```

6.3.3 数组指针(*p)[]和指针数组*p[]

数组指针强调的是指针，即(*p)[]中的 p 是指针，指向数组；而指针数组强调的是数组，即*p[]中的 p 是数组，存储的是指针。同理，数组引用强调的是引用，即(&p)[]中的 p 是引用，引用的是数组。

1. 数组指针

（1）数组指针的形式为：int (*p)[N]，表示定义了一个指向一维数组的指针，即指针 p 指向的是一个一维数组，其类型为“指向一维数组的指针”，这个一维数组有 *N* 个元素，每个元素都是 int 类型的。这里 p 先与“*”结合，表示 p 是一个指针，然后再与下标运算符结合，表示这个指针指向一个有 *N* 个元素的数组。注意：小括号不能省略，因为指针运算符“*”的优先级低于下标运算符。

（2）定义指向 *n* 维数组指针的形式为：int(*p)[x1][x2]...[xn]，表示定义了一个指向 *n* 维数组的指针，其类型为“指向 *n* 维数组的指针”，这个 *n* 维数组的第一维有 x1 个元素，第二维有 x2 个元素……第 *n* 维有 xn 个元素。

2. 对数组指针进行赋值

（1）对数组指针进行赋值时，应注意赋值的类型应与数组指针的类型相同，即都是指向 *N* 维数组的指针，且每一维相应的元素个数应分别相等，元素的类型应相同。比如 int (*p)[3][4]; int a[5][3][4]={0};，则 p=a;是正确的，这里三维数组的数组名 a 的类型是“指向二维数组的指针”，这个数组的第一维有 3 个元素，第二维有 4 个元素，数组名 a 的类型与数组指针的类型相同，可以进行赋值。

（2）示例如下：

```
int (*p)[4], int a[4]={0}; int b[33][4]={o}; int c[5]={0}; float d[4]={0}; int e[2][3][4]={0};
```

① p=&a;，正确，&a 的类型是“指向一维数组的指针”，这个一维数组有 4 个元素，与 p 的类型相同。

② p=b;，正确，二维数组名 b 的类型是“指向一维数组的指针”，这个数组有 4 个元素。

③ p=&b[0];，正确，&b[0]的结果是“指向一维数组的指针”，这个数组有 4 个元素。

④ p=&b[1];，正确，&b[1]的结果是“指向一维数组的指针”，这个数组有 4 个元素。&b[1]指向的是二维数组中第二个元素的地址，而&b[0]指向的是二维数组中第一个元素的地址，它们的类型是相同的，只是指向的地址值不同。

⑤ p=e[1]; p=&e[1][0]; p=&e[1][2];，都是正确的，其中 e[1]指向的地址是&e[1][0]，它们都是“指向一维数组的指针”，且这个一维数组都拥有 4 个元素，只是它们指向的地址值不同，但类型是相同的。

⑥ p=e[1]+1;，正确，e[1]指向的地址是&e[1][0]，对其加 1 指向的是&e[1][1]，其类型是“指向一维数组的指针”，且这个一维数组有 4 个元素，与指针 p 的类型是相同的。

⑦ p=a;，错误，一维数组名 a 指向的地址是&a[0]，它的类型是“指向 int 类型的指针”，与 p 的类型不同。

⑧ p=&b;，错误，&b 的类型是“指向二维数组的指针”，&b 与 p 的类型不相同。

⑨ p=&c;，错误，虽然&c 的结果类型是“指向一维数组的指针”，但这个一维数组有 5 个元素，与指针 p 要求的有 4 个元素不相同，因此 p 和&c 的结果不是相同的类型。

⑩ p=&d;，错误，虽然&d 的结果类型是“指向一维数组的指针”，且这个数组有 4 个元素，但元素的类型与指针所//指向的元素的类型不相同。

3．使用数组指针访问成员

（1）若数组指针的类型与数组名所表示的指针的类型相同，则可以把数组指针 p 当成数组名来理解，即可以像使用数组名一样使用指针。比如 int a[2][3]={0}; int (*p)[3]=a;，则可以像使用二维数组名那样使用数组指针 p。

（2）若数组指针指向的类型与数组名所表示的指针的类型不相同，则在使用数组指针时，应将其替换为指向的地址之后，再利用指针与数组的混合运算法则，计算出数组指针实际指向何处。

（3）虽然可以将数组指针当成数组名来理解，但并不代表数组指针与数组名完全相同，它们只是在一般情况下可以等同使用。比如用作 sizeof 的操作数时，数组指针返回的是指针的长度，在大多数机器上固定为 4；而数组名返回的是数组中所有元素占据的字节数。比如 int a[3][5]; int (*p)[5]=a;，则 sizeof a; 将得到值 60，而 sizeof p 将得到值 4。还有&p 与&a 也是不相同的，&p 表示 p 是一个二级指针，即&p 与 int (**p)[5]相同，而&a 与 int (*a)[3][5]相同。所以说数组名与数组指针并不完全等同。数组指针始终是指针，拥有指针的性质。

示例 6.7：数组指针的运算

```
int main()
{ int x[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
  int y[4]={13,14,15,16};
  int (*p)[4]=&y;  //必须要有“&”运算符,&y 表示指向的是有 4 个元素的一维数组的指针，与 p 的类型相同
  //p=y;           //错误，数组名 y 的类型是指向 int 类型的指针，与 p 的类型不同
  cout<<p[0][1];   //输出 14，p[0][1]=(*(p+0))[1]=(*(&y+0))[1]=(*(&y))[1]=y[1]
  cout<<(*p)[1];   //输出 14，(*p)[1]=(*(&y))[1]=y[1]
  cout<<*p[0];     //输出 13，*p[0]=*(*(p+0))=*(*(&y+0))=*(*(&y))=*y=*(&y[0])=y[0]
  cout<<(*p)[5];   //输出随机值，(*p)[5]=(*(&y))[5]=y[5]
  cout<<*p[1]; //输出随机值，*p[1]=*(*(p+1))=*(*(&y+1))，这里&y+1 超出一维数组 y 所示地址的范围
  cout<<p[0];      //输出 y[0]的地址，p[0]=*(p+0)=*(&y+0)=y=&y[0];
  p=x;             //把数组 x 的地址赋给指针 p
  cout<<p[1][1];   //输出 6，数组名 x 的类型与指针 p 的类型相同，可以将数组指针 p 作为数组名 x 来使用
```

```
cout<<(*p)[5];    //输出 6，(*p)[5]=(*(x))[5]=(*(&x[0]))[5]=(x[0])[5]=x[0][5]
cout<<*p[1];      //输出 5，*p[1]=*(*(p+1))=*(*(x+1))=*(*(&x[0]+1))=*(*(&x[1]))=
                  // *(x[1])= *(&x[1][0])=x[1][0]
p=x+1;//或者 p=&x[1];
cout<<p[1][1];    //输出 10
cout<<(*p)[5];}   //输出 10
```

4. 指针数组

（1）指针数组的形式为：int *p[N];，表示声明了一个有 *N* 个元素的数组，其中每个元素都是一个 int 类型的指针。这里 p 先与下标运算符结合，表示 p 是数组，然后再与“*”运算符结合，表示数组中的内容是一个指针。

（2）初始化指针数组时，应像普通数组一样，使用初始化列表进行初始化，但必须对每个元素使用指针进行初始化。比如 int a=0, b=1; int *p1=&a; int *p[3]={&a, &b, p1}。

（3）指针数组一般用作字符串，比如 char *p[5]={"1", "22","333"};。

（4）指针数组的数组名同样是指向数组中第一个元素地址的指针，但需要利用一些声明时的类型分析方法，分析指针数组所表示数组名的类型。

（5）理解指针数组的数组名，看如下示例。

```
int *p[4];
```

① 数组名 p：int **p1=p;或 int **p1=p+1。

分析：数组名 p 是指向第一个元素 p[0]地址的指针，即 p 指向&p[0]，因为 p[0]存储的元素是一个 int 类型的指针，所以数组名 p 就是一个指向指针的指针，因此可以这样赋值。

② &p 的类型：int *(*p1)[4]=&p;。

分析：&p 是指向一维数组的指针，这个数组有 4 个元素，其类型是 int*，因此&p 的类型是*(*)[4]。

```
int *p[4][5];
```

① 数组名 p：int *(*p1)[5]=p;。

分析：数组名 p 是指向第一个元素 p[0]地址的指针，即 p 指向&p[0]，但 p[0]存储的元素又是一个数组，这个数组有 5 个元素，其类型是 int*，因此 p 的类型是*(*)[5]。

② &p 的类型：int *(*p1)[4][5]=&p;。

分析：&p 是指向二维数组的指针，这个二维数组的第一维有 4 个元素，第二维有 5 个元素，且每个元素的类型是 int*，因此&p 的类型是*(*)[4][5]。

（6）对指针数组进行运算时，应将指针数组替换为指针所指向的实际地址后再进行运算。

5. 数组引用

（1）对于引用，请记住一个关键点，即引用是某变量的别名。也就是说，凡是出现引用的地方，只需简单地将引用替换为被引用的变量的名字即可，关键要明白到底引用了什么名字。

（2）引用和指针有一些差别，因为对引用赋值必须是左值。比如，int y[4];int (&p)[4]=y;是正确的，而用&y 将得到一个错误的结果；同样 int x[22][4]; int (&p)[4]=x[1];是正确的，不能用 x+1 或者&x[1]来初始化引用（因为它们的结果是右值）；int (&p)[4]=x;也将得到一个错误的结果，只能是 int (&p)[4]=x[0];。

（3）使用数组引用访问数组的成员时，应将其所引用的变量的名字替换过来，只要替换之后是正确的，结果就是正确的。比如 int x[22][4]; int (&p)[4]=x[1];，在使用引用时，只需将引用 p 替换为 x[1]即可；p[0][1]是错误的，因为 p[0][1]=x[1][0][1]；p[0], p[1]是正确的，因为 p[0]=x[1][0], p[1]=x[1][1]。

6.4　动态分配内存 new 关键字

注：本节所讲的动态内存的分配只针对内置类型。

6.4.1 内存管理基础

1. 内存管理概述

（1）内存管理器：内存是由内存管理器根据相应的内存分配策略进行分配的。这里只需知道内存是由内存管理器分配的这一事实就行了，有关内存管理器存在于何处，以及是怎样实现的，则没必要对其进行深入了解。

（2）内存管理策略：一般有 3 种策略，即任意大小的分配策略、固定大小的分配策略、垃圾收集的分配策略。任意大小的分配策略，即可以分配任意大小的内存块，这种方法非常灵活，但性能差且易产生碎片（即许多小的、不连续的未分配的内存区域）。固定大小的分配策略，即总是分配固定大小的内存块。

（3）内存管理的分层：内存管理一般存在多种管理层面，比如操作系统内核提供最基础的内存分配服务，而编译器也会建立起自己的内存分配服务， C++中的 new 或 C 中的 malloc 建立在操作系统的本地分配服务基础上，有可能是编译器从操作系统那里首先预定一大块内存，然后再根据需要分配出去，以此来覆盖操作系统的本地分配服务。至于编译器是怎样实现的，不是 C++语法的重点。

2. C++对内存的划分

（1）C++将内存划分为 5 个区，分别是堆、栈、自由存储区、全局/静态存储区、常量存储区。

（2）栈的特点。

- 栈存储的是自动变量。
- 栈内存的分配与释放是由编译器自动完成的，不需要手工控制。
- 栈是先进后出的存储结构。
- 栈内存的分配只涉及一个指针的递增，而且栈内存的分配运算内置于处理器的指令集中，因此效率很高，速度很快。
- 使用栈分配的内存容量有限，在编译器中一般只有几 MB 的大小。
- 注意：也可以使用 alloca 函数对栈内存进行动态分配，但使用 alloca 对栈进行动态分配、内存释放是由编译器完成的，无须程序员手工操作。

（3）堆和自由存储区的特点。

- C++对堆和自由存储区的概念没有做出详细的说明，它们都是 C++的动态存储区域，其性质很相似，只不过一个使用 malloc 和 free 函数，而另一个使用 new 和 delete 对内存进行分配和释放。
- 堆是动态分配的，即在运行时才分配内存。
- 堆的分配和释放编译器不用管，而由程序员控制，一般使用 new 分配动态内存，使用 delete 释放动态分配的内存。也就是说，程序员可以控制动态分配的存储区内存的生命期。
- 可以在堆中分配任意数量的内存块。
- 堆内存的释放是由程序员控制的，因此容易发生内存泄漏，同时易产生内存碎片。
- 堆的实现很复杂，因此堆的效率比栈要低很多。
- 堆的空间大（一般 32 位系统有 4GB 大小），而且灵活（即在运行时才分配内存）。

（4）全局/静态存储区：全局变量和静态变量被分配到这个内存区域中。

（5）常量存储区：这个区域存放常量，不允许修改。

（6）区分堆与栈：int *p=new int;，表示在栈内存 p 中存放一个指向由 new 分配的堆内存的指针，程序先使用 new 在堆中分配一块合适大小的内存，然后返回这块内存的首地址，并将该地址放入栈中。

6.4.2 使用 new 动态分配单个对象

1. new 运算符的使用

（1）动态内存分配：在运行时从自由存储区（或堆）中分配内存。

（2）C++使用 new 运算符来动态分配内存，使用 delete 释放由 new 分配的内存。使用 new 运算符系统将从自由存储区为对象分配内存，并返回一个指向该对象的指针，即该对象的地址。

（3）使用 new 分配单个对象的方法：在 new 之后加上一个类型指示符（比如 int 类型），类

型指示符可以是内置类型，也可以是用户自定义类型（比如 class 类型），这时 new 会根据指定的类型找到一个长度合适的内存块，然后返回该内存块的地址。程序员需要做的是创建一个类型匹配的指针，指向使用 new 分配的内存。比如 int *p=new int;，表示使用 new 从空闲存储区分配了一个 int 类型的对象，并返回该对象的地址，然后让指针 p 指向这个内存地址。再比如 const int *p; p=new int;，而 float *p=new int;是错误的，两者类型不匹配。

（4）new 运算符的特点是，使用 new 运算符分配的对象没有名字，对该对象的操作都要通过指针间接地完成。比如 new int，就是从空闲存储区分配了一个 int 类型对象，但这个对象没有名字，因此没法对这个对象进行操作，而需要通过指针来间接完成操作。比如 int i;int*p=&i; 和 int *p=new int 都是将 int 对象的地址赋给指针，但不同的是前者可以用名称 i 和*p 来访问该 int 类型对象，而后者则只能用*p 来访问该对象，即 p 指向的内存没有名字。

2．使用 delete 释放内存空间

（1）使用 new 分配的对象将一直存在，直到使用 delete 结束该对象的生命期，或者说释放该对象的内存，将内存归还给自由存储区。也就是说，若没有使用 delete 释放由 new 分配的内存，则该资源会一直被占用。

（2）使用 delete 的方法是，在 delete 关键字之后加上指向以前使用 new 动态分配的内存的指针名即可。比如 int *p=new int;，则 delete p;就表示释放了 p 指向的内存。

（3）注意：使用 delete 释放的是指针所指向的内存，而没有将指针消亡。比如 int *p=new int;，则 delete p; 表示释放 p 所指向的动态分配的内存，这时由 new 创建的动态对象的生命期也就结束了，但是指针 p 仍然存在，并没有消亡，这种指针称为悬垂指针。悬垂指针往往是错误的根源。

（4）解决悬垂指针的方法是让指针指向 NULL。

3．使用 new 动态创建对象的初始化

（1）使用 new 动态创建对象的初始化方法：在类型名之后使用一对小括号来指定初始值，小括号内的值用于初始化该动态分配的对象；若小括号内没有值，则将被初始化为各类型的默认值，对于 int 和 float 类型为值 0，对于 char 类型一般是一个空字符'\0'，对于类类型则将调用默认的构造函数对其进行初始化。比如 int *p=new int(102);，使用 new 从空闲存储区分配一个 int 类型的对象，并用 102 初始化这个对象，然后使用这个对象的地址初始化指针 p。

（2）若没有为动态创建的对象进行显式初始化，那么不管该语句在全局还是局部范围内，此对象的初始化方式都与在函数内定义的变量相同，即对于内置类型没有初始化，访问该对象中的值时将得到一个随机值；对于类类型则使用默认的构造函数进行初始化。比如 int *p=new int;，这里由 new 分配的对象没有初始值，不管此语句在全局还是局部范围内，使用 new 创建的对象都与在函数内创建的变量相同（局部变量），因此 cout<<*p<<endl;将得到一个意想不到的值。

示例 6.8：使用 new 分配单个对象

```
#include <iostream>
using namespace std;
int *p1=new int;          //不管是在全局还是局部范围内，使用 new 分配的对象没有初始化，访问该对象中的
                          //值时将得到一个随机值
int main()
{    int a=3;
     int *p2=new int();       //将由 new 分配的对象初始化为默认值 0
     int *p3=new int(3);      //将由 new 分配的对象初始化为值 3
     cout<<*p1<<endl;         //输出随机值，因为 p1 指向的对象没有初始化
     *p1=4;                   //我们无法直接访问由 new 分配的内存，只能通过指针来间接访问
     delete p1,p2,p3;   //释放由 new 分配的内存资源。但指针 p1,p2,p3 并没有消亡，它们成了悬垂指针
     p1=NULL;  p3=NULL;       //解决悬垂指针的方法是让指针指向 NULL
     p2=&a;     }             //虽然使用 delete 释放掉了由 p2 指向的内存资源，但指针 p2 并没有消亡
```

6.4.3 使用 new 动态创建数组

1. 静态联编与动态联编

（1）静态联编：声明的数组是在编译阶段为它分配内存空间的，不管数组是否被使用，它都占用了内存空间。在编译阶段分配内存的方式被称为静态联编。

（2）动态联编：在运行阶段需要数组时就创建，不需要时就不创建。这种在运行阶段才分配内存的方式被称为动态联编，这种数组叫作动态数组。

（3）在编译阶段使用静态联编创建的数组变量有 3 个限制，即数组的大小不能改变，在编译时必须指定数组的大小，数组变量的作用域只限于创建它的语句块内。

（4）在运行阶段使用动态联编创建的数组不需要在编译时就知道其长度，一般在运行时才确定数组的长度。也就是说，数组的大小可以使用变量来指定，动态分配的数组将一直存在，直到显式释放它为止。

2. 使用 new 运算符创建动态数组

（1）创建动态数组时应指定数组元素的类型，且必须有方括号“[]”，方括号中要有创建的长度。比如 int *p=new int[3];、int a=3; int *p=new int[a];。

（2）使用 new 运算符创建的动态数组，返回的将是指向数组中第一个元素地址的指针。比如 int *p=new int[4];，其中 new int[4]返回的是指向第一个元素地址的指针，这个指针的类型是 int，因此只需要一个 int 类型的指针指向 new 分配的内存即可。

（3）使用 new 创建的 *N* 维数组，返回的同样是第一个元素地址的指针，因为多维数组是数组的数组，因此对于 *N* 维数组，它的第一个元素是 a[0]，这个元素的地址是&a[0]而不是 a[0]，所以第一个元素的地址是指向 *N*–1 维数组的指针。也就是说，new 返回的是一个数组指针，因此程序员只需声明一个类型相同的数组指针指向 new 返回的指针即可，这也是使用 new 创建多

维数组的方法。比如 new int[3][4][5];返回的是一个指向二维数组的指针，其中第一维有 4 个元素，第二维有 5 个元素，因此应使用 int (*p)[4][5]这样的指针来指向 new 返回的地址，如 int (*p)[4][5]=new int[3][4][5];。

（4）注意：int ***p=new int[3][4][5];是错误的，因为它们的类型不相同，new int[3][4][5]并不是分配的一个三级指针。多级指针与数组指针是两个不同的概念，请不要搞混。

（5）使用 new 运算符创建的动态数组，其第一维可以不是整型常量表达式，但除第一维外的其他维在编译时必须是已知的常量值，即第一维的长度可以在运行时确定。比如 int a=4; int (*p)[4]=new int[3][a];错误，因为第二维的 a 不是常量。

（6）访问动态分配的数组：使用 new 创建的动态数组返回的其实是指向数组中第一个元素地址的指针，对于一维数组，可以像使用数组名一样来使用指针；而对于多维数组，返回的则是一个数组指针，因此同样可以像使用数组名一样来使用由 new 创建的多维数组。一般情况下，数组指针可以被当作数组名来使用（详见数组指针部分内容）。

（7）比如 int *p=new int[30];与 int *p1=new int;，不要认为它们都只是 int 指针没有区别，其实两者有很大的差别，new int[30]分配了存储 30 个 int 类型的空间，因此可以保证在以 p 为基础的地址上向后在 29 个单位内偏移而不会出错；但 new int 只分配了存储 1 个 int 类型的空间，因此在以 p1 为基础的地址上进行偏移，访问到的将是不确定的内存空间（有可能是空闲的内存，也可能不是空闲的内存。比如 p1[29]=3;，程序就有可能因为内存不是空闲的而崩溃；但 p[29]=3;就不会有错。

（8）C++允许动态分配长度为 0 的空数组。使用 new 创建的长度为 0 的空数组，new 返回的将是一个有效的非零指针，因为这种指针没有指向任何元素，所以在程序中不能对该指针进行解引用操作，否则可能会出错。比如 int *p=new int[0];，则语句*p=2;就可能会出错。长度为 0 的数组变量是非法的，但使用 new 创建的长度为 0 的动态数组是合法的。

3. 动态数组内存的释放

（1）使用 new 创建的动态数组，同样需要使用 delete 释放其内存，方法是在 delete 关键字和指针名之间加上一对空的方括号，空的方括号的作用是告诉编译器，后面的指针指向的是自由存储区中的数组，而不是单个对象。比如 delete [] p;。

（2）若释放动态数组内存时忘记加上空的方括号，则编译器不会报错，但不保证程序能正确运行。

（3）若释放动态数组内存时漏写了方括号对，则至少可以表示编译器少释放了由 new 分配的内存空间，这样会发生内存泄漏。

（4）注意：释放 new 分配的内存，不管使用 new 创建的是几维数组，方法都是 delete [];，并不是使用 new 创建了 *N* 维数组就需要在 delete 和指针名之间加 *N* 对空的方括号，而是只需加

上一对空的方括号即可。比如 int (*p)[3][4][5]=new int[2][3][4][5];，则释放方式为 delete []p;。

4. 动态数组的初始化

（1）值初始化：可以在方括号后面使用一对空的圆括号来初始化数组，这种初始化方式被称为值初始化。比如 int *p=new int[4]();，数组中各元素将被初始化为其类型的默认值（比如 int 类型为 0），若是类类型，则将调用默认的构造函数对数组中的每个元素依次进行初始化。

（2）对于动态分配的数组，只能使用值初始化的方式将各元素初始化为其类型的默认值，而不能像数组变量那样，可以使用初始化列表为数组中的各元素提供不同的初始值。

（3）注意：int *p=new int[4](0);这种语法是错误的，这并不是值初始化，值初始化要求是一对空的圆括号。

（4）对于类类型，不能使用除默认的构造函数之外的构造函数初始化动态分配的数组。比如 A *p=new A[4](1);是错误的，这里假设 A 是一种类类型，正确的形式是 A *p=new A[4]();或 A *p=new A[4];。

（5）若没有对动态分配的数组进行值初始化，那么不管是在全局还是函数内动态创建的数组，对于内置类型的初始化方式与在函数内定义的变量相同，即对于内置类型没有初始化，访问该对象中的值时将得到一个随机值；对于类类型则使用默认的构造函数对数组中的每个元素依次进行初始化。

示例 6.9：使用 new 动态创建数组

```
#include <iostream>
using namespace std;
int main()
{    int a=3;
     int (*p)[4][5]=new int[3][4][5]; //使用 new 创建的 N 维数组返回的是第一个元素地址的指针，这
                                      //里返回的是指向二维数组的指针
     //int ***p1=new int[3][4][5];     //错误，new 返回的类型与指针 p1 的类型不相同
     int (*p2)[4][5]=new int[a][4][5]; //正确，由 new 创建的动态数组的第一维可以不是整型常量表
                                       //达式
     //int (*p3)[4][5]=new int[3][4][a];  //错误，由 new 创建的动态数组除第一维外其他维在编译时
                                      //必须是已知的常量值
     int (*p5)[5]=new int[3][5]();    //动态分配的数组，只能使用值初始化的方式将各元素初始化为
                                      //其类型的默认值
     //int (*p6)[5]=new int[3][6](0);    //错误，对于动态分配的数组，其元素只能使用值初始化方式，
                                       //值初始化应使用空的圆括号，此语句试图将各元素的值初始化
                                       //为 0，但这不是值初始化
     p2[2][3][4]=5;                    //可以像使用数组名一样来使用指向动态分配的数组的指针
     int *p4=new int[0];  //正确，C++允许动态分配长度为 0 的空数组，new 返回的是一个有效的非零指针
     //*p4=3;             //可能发生程序崩溃。p4 是指向长度为 0 的空数组指针，这种指针没有指向任何
                          //元素，不能对该指针进行解引用操作
     int *p7=new int; int *p8=new int[33];
     *(p8+22)=3;          // p8 指向的内存已经由 new 分配了 33 个大小的内存空间，因此可以保证在以 p8
```

```
                        //为基础的地址上，向后在 33 个单位内偏移而不会出错
//*(p7+22)=32;          //有可能引发程序崩溃
delete []p;            // 释放由 new 分配的内存，不管使用 new 创建的是几维数组，方法都是 delete [];
//delete [][][]p2;  //语法错误
delete []p4; delete p7;     delete []p8;
delete p5; }        //发生内存泄漏，释放动态数组内存时漏写了方括号编译器不会报错，并且不保证程序
                    //能正确运行，但至少可以表示编译器少释放了由 new 分配的内存空间
```

6.4.4　使用 new 动态分配内存的类型分析

1. 使用 new 创建 const 对象及多级指针

（1）使用 new 创建 const 对象的语法为：const int *p= new const int;。

（2）使用 new 创建的 const 对象的回收方法为：delete p;。虽然不能修改 const 对象的值，但仍能回收 const 对象的内存。

（3）使用 new 创建 const 对象时的注意事项：与普通常量一样，动态创建 const 对象时必须在创建时初始化，且一旦初始化，其值就不能被修改。

（4）使用 new 创建 const 对象的数组时，只能使用值初始化方式将数组初始化为 0，比如 const int *p=new const int[4]();。

（5）使用 new 创建二级指针的方法为（*N* 级依此类推）：int **p=new int *;。

（6）注意：对于二级指针 int **p=new int *;，只表示指针 p 是动态分配的内存，可以使用 delete p;进行释放；但对于*p 指向的内存是否是动态分配的，这就要视程序而定，若不是动态分配的，则使用 delete *p;释放程序可能会出错（释放了不是由 new 分配的内存）。

2. 对 new 动态分配内存的类型分析

（1）关键原理：new 返回的是动态分配的内存空间的指针，对于数组返回的是动态分配的内存空间中第一个元素的指针。

（2）类型分析的方法：与标识符声明时的类型分析方法相同，既可以使用右左法则，也可以使用运算符的优先级法则进行分析。

（3）进行类型分析时，应从前缀和后缀运算符之间的位置开始分析。

（4）示例如下：

```
int *(*p)[5]=new int *[4][5];
```

分析：从前缀与后缀运算符之间的位置开始分析，new 分配的是一个存储 4 个元素的内存空间，且返回第一个元素的内存地址；这 4 个元素的类型是一个有 5 个元素的数组，且每个元素都是一个 int 类型的指针，因此只需保证所声明的指针的类型与这 4 个元素中存储的元素的类型相同即可。也就是需要一个存储 int 类型指针且有 5 个元素的数组，即指针指向的类型应是 int *[5]，因此我们这样声明指针——int *(*p)[5]。

```
int ***p=new int **[4];
```

分析：从前缀与后缀运算符之间的位置开始分析，new 分配的是一个存储 4 个元素的内存空间，且返回第一个元素的内存地址；这 4 个元素的类型是一个二级指针，因此我们需要一个三级指针来指向这段内存。

```
int *const(*p)[5]=new int *const [4][5];
```

分析：从前缀与后缀运算符之间的位置开始分析，new 分配的是一个存储 4 个元素的内存空间，且返回第一个元素的内存地址；这 4 个元素的类型是一个有 5 个元素的数组，每个元素都是一个 const 常量，且这个常量是一个指针，这个指针是 int 类型的，因此 new 返回的类型应是 *const (*)[5];。所以我们这样声明指针——int *const (*p)[5]。

6.4.5 使用多级指针动态创建多维数组

注意：int ***p=new int**;，这是分配的一个三级指针，而不是三维数组，使用类似于 p[1][2] 的语法会出现与 int *p=new int 和 int *p1=new int[30];类似的错误，详见 6.4.3 节。

1. 使用多级指针创建多维数组的基本原理

（1）前面介绍的创建动态数组的方法，只能保证在运行时确定第一维的长度，而从第二维以后却需要在编译时就确定其长度。比如 int (*p)[4][5]=new int[n][4][5];，只有第一维可以动态确定，第二维和第三维必须在编译时就确定其长度。

（2）要实现多维数组的每一维都真正地能动态分配，可以使用链表（也叫作索引）的形式，其方法是使用多级指针。

（3）使用多级指针动态创建多维数组的原理：让指针指向数组，且数组中的每个元素又是指针，这些指针又指向数组，依次逐级指向数组，从而实现多维数组。其原理如图 6.4 所示。

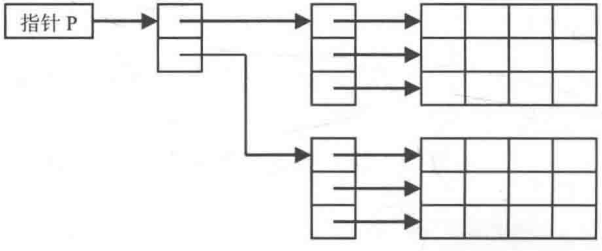


图 6.4 使用多级指针实现多维数组原理图

（4）原理具体分析：要实现 N 维数组，就需要使用 N 级指针。

示例：对于 $N \times M \times L$ 的三维数组就需要三级指针。

① 首先让指针指向一个有 N 个元素的数组，该数组中的 N 个元素都是指针，这时需要一个二级指针指向这个数组中的元素。

② 然后使这 N 个元素中的每一个元素分别指向有 M 个元素的数组，这 M 个元素又是一个指针，这时需要一个三级指针指向这个数组中的元素。

③ 接下来让这 M 个元素中的每一个元素分别指向有 L 个元素的数组，若这个数组的内容不是指针，则只需创建一个三级指针即可；若仍是指针，则需要一个四级指针指向它。

④ 可以看到，创建 N 维数组时，应使用 new 按第 1 维到第 N 维的顺序分配内存空间。

⑤ 注意：C++没有多维数组，多维数组就是指数组的数组。

2. 使用多级指针动态分配多维数组的具体方法

动态分配形如 a[x1][x2]...[x_{n-1}][xn];的 n 维数组，其步骤如下。

（1）首先创建一个 n 级指针，指针的级数与数组的维数相同，即 int **...**p; 共有 n 个“*”号。

（2）然后逐级分配数组每一维的元素个数。

① 动态分配第一维：

```
//p=new int **...**[x1];   其中有 n-1 个“*”号。
```

② 动态分配第二维：

```
for(int k=0;k<x2; i++)  p[k]=new int **...**[x2];  //其中有 n-2 个“*”号
```

③ 动态分配第三维：

```
for(int k1=0; k1<x2; k1++) for(int k2=0;k2<x3;k2++)
p[k1][k2]=new int **...**[x3];              //其中有 n-3 个“*”号
```

④ 动态分配第四维：

```
for(int k1=0; k1<x2; k1++)
for(int k2=0;k2<x3;k2++)
for(int k3=0;k3<x4;k3++)
p[k1][k2][k3]=new int **...**[x4];  //其中有 n-4 个“*”号
```

⑤ 其他维的分配类似。

⑥ 动态分配第 n 维：

```
for(int k1=0; k1<x2; k1++)  for(int k2=0; k2<x3; k2++)...for(int k_{n-1}=0; k3<xn; k_{n-1}++)
p[k1][k2]....[k_{n-1}]=new int [xn];      //其中有 n-1 个嵌套循环
```

（3）示例如下：

① 动态分配 $n \times m$ 的二维数组的方法。

```
int **p;
p=new int*[n];                    //分配二维数组第一维的长度 n
for(int i=0; i<n; i++) p[i]=new int[m];    //分配二维数组第二维的长度 m
```

② 动态分配 $N \times M \times L$ 的三维数组的方法。

```
int ***p;
p=new int **[N];                          //分配三维数组第一维的长度 N
for(int i=0; i<N; i++)  p[i]=new int *[M];      //分配三维数组第二维的长度 M
for(int i=0; i<N; i++) for(int j=0; j<M; j++) p[i][j]=new int[L]    //分配第 3 维的长度 L
```

③ 动态分配 $N\times M\times L\times R$ 的四维数组的方法。

```
int ****p;
p=new int ***[N];                    //分配四维数组第一维的长度 N
for(int i=0; i<N; i++)  p[i]=new int **[M];      //分配四维数组第二维的长度 M
for(int i=0; i<N; i++) for(int j=0; j<M; j++) p[i][j]=new int*[L]    //分配第三维的长度 L
for(int i=0; i<N; i++) for(int j=0; j<M; j++) for(int k=0;k<L; k++)
p[i][j][k]=new int[R]                //分配四维数组第四维的长度 R
```

3. 使用 delete 释放由多级指针动态分配的多维数组

（1）注意：应按照与 new 分配的内存空间相反的顺序逐层释放内存空间，new 是按照第一维到第 n 维的顺序进行内存空间分配的，而使用 delete 释放内存空间时，应按照第 n 维到第一维的顺序进行内存空间释放，否则会出错。

（2）具体步骤如下。

① 释放第 n 维分配的空间：

```
for(int k1=0; k1<x2; k1++)  for(int k2=0; k2<x3; k2++)...for(int kn-1=0; k3<xn; kn-1++)
{ delete[] p[k1][k2]....[kn-1];  p[k1][k2]...[kn-1]=NULL; }  //其中有 n-1 个循环
```

② 释放第 $n-1$ 维分配的空间：

```
for(int k1=0; k1<x2; k1++)  for(int k2=0; k2<x3; k2++)...for(int kn-2=0; k3<xn; kn-2++)
{ delete[] p[k1][k2]...[kn-2];  p[k1][k2]...[kn-2]=NULL; }   //其中有 n-2 个循环
```

③ 其他维空间的释放省略。

④ 释放第二维分配的空间：

```
for(int k=0;k<x2; i++)  {delete [] p[k]; p[k]=NULL}
```

④ 释放第一维分配的空间：

```
 delete [] p; p=NULL
```

（3）示例：释放 $N\times M\times L\times R$ 的四维数组的方法（假设已为四级指针 p 成功分配内存）。

```
for(int i=0; i<N; i++) for(int j=0; j<M; j++) for(int k=0;k<L; k++)
{delete[] p[i][j][k];  p[i][j][k]=NULL;}      //释放第四维的内存空间
for(int i=0; i<N; i++) for(int j=0; j<M; j++)
{ delete[] p[i][j];  p[i][j]=NULL; }   //释放第三维的内存空间
for(int i=0; i<N; i++)  {delete[] p[i];    p[i]=NULL; }      //释放第二维的内存空间
delete [] p;  p=NULL;              //释放第一维的内存空间
```

示例 6.10：使用多级指针实现多维数组

```
#include <iostream>
using namespace std;
int main()
{    //使用多级指针创建一个 2×3×4×5 的四维数组示例
     int ****p; int a=1;
     //分配第一维的空间
     p=new int***[2];
     //分配第二维的空间
     for(int i=0;i<2;i++) p[i]=new int**[3];
```

```
    //分配第三维的空间
    for(int i=0;i<2;i++)   for(int j=0;j<3;j++)   p[i][j]=new int*[4];
    //分配第四维的空间
    for(int i=0;i<2;i++)for(int j=0;j<3;j++)for(int k=0;k<4;k++)p[i][j][k]=new int[5];
    //为创建的p[2][3][4][5]的四维数组分配值
    for(int i=0;i<2;i++) for(int j=0;j<3;j++)for(int k=0;k<4;k++) for(int
        m=0;m<5;m++){p[i][j][k][m]=a++;   }
    //输出四维数组的值
    for(int i=0;i<2;i++) for(int j=0;j<3;j++) for(int k=0;k<4;k++)  for(int m=0;m<5;m++)

       cout<<p[i][j][k][m]<<endl;
    //delete[] p;       //错误，应按照与new分配的内存空间相反的顺序逐层释放内存空间
    //delete[] p[1][2]; //错误，原因同上
    //释放第四维的内存空间
    for(int i=0;i<2;i++)for(int j=0;j<3;j++)for(int k=0;k<4;k++)
       {delete[] p[i][j][k]; p[i][j][k]=NULL;}
    //释放第三维的内存空间
    for(int i=0;i<2;i++)for(int j=0;j<3;j++){delete[] p[i][j]; p[i][j]=NULL;}
    //释放第二维的内存空间
    for(int i=0;i<2;i++){delete[] p[i]; p[i]=NULL;}
    //释放第一维的内存空间
delete[] p;  }
```

6.4.6　delete 常见错误及内存错误

1. 使用 delete 释放内存的基本原理

（1）delete 关键：delete 只能释放由 new 分配的内存，而且 delete 释放内存之后指针本身并不会消亡。

（2）注意：delete 释放的是指针所指向的内存。

（3）区别指针与指针所指向的对象：对于动态分配的内存，若指针消亡了，并不能代表指针所指向的内存被释放了；若指针指向的内存被释放了，也并不能代表指针就会跟着消亡或者成了 NULL 指针（指向 0 的指针）。示例如下：

```
int *p=new int;  int *p1=p; delete p1;  delete p;   //错误
```

分析：指针 p 和 p1 都指向由 new 分配的动态内存，它们指向相同的对象，而语句 delete p1;释放的是 p1 所指向的内存，即由 new 分配的内存，也就间接地释放了 p 指向的内存。本例对使用同一个 new 分配的内存释放了两次，因此是错误的。

```
int *p= new int; int a=1; delete p; p=&a;
```

分析：语句 delete p;只是将 p 指向的内存释放了，但指针 p 并没有消亡，还可以使用指针 p 指向其他地方，比如 p=&a;这种指针就是悬垂指针。

```
void f(){int *p=new int; }   //内存泄漏，但程序不会出错
```

分析：函数调用结束后 p 的生命期也就结束了，即函数调用结束后 p 就消亡了，但 p 所指

向的内存并没有被释放，这块内存仍然占据着堆空间，函数 f 每调用一次就会多占用一块堆内存空间，若多次调用 f 则可能会使内存耗尽。这种情形被称为内存泄漏。

2. 使用 delete 注意事项及内存错误

（1）不要使用 delete 释放已经释放过的内存块，否则结果是不确定的。这种情况一般发生在多个指针同时指向同一个动态创建的对象的时候，这种错误比较难被发现。比如 int *p=new int; int *p1=p; int *p2=p;，这时指针 p、p1、p2 都是指向使用 new 创建的同一块内存，释放这块内存，只需对 3 个指针中的任何一个使用 delete 即可，如 delete p1;、delete p;或者 delete p2;都可释放由 new 分配的内存，若同时对这 3 个指针使用 delete，则会发生释放相同内存两次的错误。

（2）不要使用 delete 释放不是由 new 分配的内存，否则程序可能会在运行期间出现未定义的行为，但使用 delete 释放值为 0 的指针不会有错误，是安全的。比如 int *p=0; delete p; 是正确的，但是对于 int a=9; int *p=&a; delete p;，程序可能会在运行阶段出现未定义的错误。

（3）应使用 delete []释放动态数组。释放动态数组必须要有[]，若忘记加上空的方括号，虽然编译器不会报错，但是不保证程序能正确运行。若漏写了方括号，则至少可以表示编译器少释放了由 new 分配的内存空间，这样会产生内存泄漏。

（4）delete 只能释放前次使用 new 分配的空间。比如 int *p=new int; p=new int; delete p; delete p;，其中 p=new int;将改变指针 p 的指向，使其指向第二个 new 分配的内存；第一个 delete 并没有释放掉第一个 new 分配的内存，delete 只能释放前次使用 new 分配的空间，因此后面的两个 delete 都是在释放第二个 new 分配的空间，这里就存在使用 delete 对相同的内存释放两次的错误;而使用第一个 new 分配的内存，在此没有得到释放。

（5）若用 new 分配了资源，而没用 delete 释放该资源，则该资源将一直被占用。

（6）悬垂指针（或称为野指针）：执行 delete p;后只是把 p 所指向的内存释放了，并没有删掉指针 p 本身，还可以将 p 重新指向另一个新的内存块，这样的指针被称为悬垂指针。这样的指针很难被检测出来。

（7）解决悬垂指针的方法：执行 delete 后，应立即将该指针设置为空指针，以使指针不指向任何对象，比如 delete p; p=0;。

（8）悬垂指针的危害：在大多数机器上，悬垂指针还是保存着指向原来对象的地址，虽然还能输出这个地址，但不可以继续使用这个地址，因为这个地址已经被释放了，即这个地址没有被分配，若继续使用这个地址，则可能会出现意想不到的错误。比如 int *p=new int; delete p;，则在大多数机器上 cout<<p<<endl;能输出这个地址，但这个地址被释放了，若继续使用可能会出错，如*p=2;。

（9）总结：使用 new 分配的动态内存易发生 3 种类型的内存错误，第一种是使用 delete 失败(或忘记使用 delete 释放内存)产生的内存泄漏；第二种是对同一个内存区使用了两次 delete；

第三种是使用 delete 释放后的内存（一般指悬垂指针）。

（10）内存耗尽：自由存储区的空间是有限的，虽然现在内存空间很大，但也有可能被耗尽。若 new 表达式无法获得足够大的内存空间，则 new 将返回一个 NULL 指针，系统可能还会抛出 bad_alloc 异常，这时可以采用判断指针是否为 NULL 来进行处理，比如 int *p=new int; if(p==NULL)...。

3. 本章有关 new/delete 的未尽内容

（1）new 有 3 种用法，即使用 new 分配单个对象、使用 new 分配数组、定位 new 表达式。

（2）重载 new 和 delete、定位 new 表达式、内存对齐的内容详见后文。

示例 6.11：悬垂指针

```
#include <iostream>
using namespace std;
int main(){   int a=3;
    int *p7=new int;
    delete p7;  //悬垂指针，执行 delete p7;后只是把 p7 所指向的内存释放了，并没有删掉指针 p7 本身
    cout<<p7<<endl;  //在大多数机器上，悬垂指针还是保存着指向原来对象的地址，还能输出这个地址
    //*p7=3;//悬垂指针虽然有地址，但这个地址没有被分配，若继续使用这个地址，可能会出现意想不到的错误
    p7=NULL;     //正确，解决悬垂指针的方法是立即将释放了内存的指针设置为空指针
    p7=&a;       //p7 指向的内存虽然被释放了，但指针 p7 仍然存在，仍能继续使用这个指针，但必须给指针
                 //赋予正确的地址
    *p7=3;     cout<<*p7<<endl;}  //正确
```

6.4.7　使用 malloc/free 动态创建和释放内存简介

（1）malloc/free 是 C 中的内容，这里只需知道怎样使用这两个库函数即可。

（2）C++使用 new/delete 的原因： malloc/free 是库函数，无法满足对类类型动态分配对象的要求。因为对于类类型在创建时要自动执行构造函数，在消亡时要自动执行析构函数，而 malloc/free 不能满足这些要求，因此在 C++中引入了 new/delete 进行动态内存分配。

（3）C++经常需要调用 C 函数，为了与 C 兼容，而保留了 malloc/free。

（4）malloc 是库函数，其原型为：void *malloc(unsigned int);。

（5）使用 malloc 时需要计算分配的内存空间的长度，还必须将返回的指针类型强制转换为所需要的类型。

（6）动态分配内存的方法为：int *p=(int*)malloc(sizof(int));。

（7）动态分配存储 5 个元素的一维数组的方法为：int *p=(int *)malloc(sizeof(int)*5);。

（8）分配 4×5 的二维数组的方法为：int **p; p=(int **)malloc(sizeof(int *)*4); for(int i=0;i<4; i++) p[i]=(int *)malloc(sizeof(int)*5);。释放内存时使用 free，按照相反的顺序释放内存，使用 free 只需直接把指针作为参数即可，比如 for(int i=0;i<4; i++) free(p[i]); free(p);。

（9）分配 N 维数组的方法与使用 new 相似，这里不再介绍。

6.5 C 风格字符串

6.5.1 C 风格字符串

1. C 风格字符串简介

（1）字符串是一种数据类型，但在 C++中没有直接能表示字符串的内置类型。

（2）C++有两种处理字符串的方法：一种是 C 风格字符串；另一种是使用<string>头文件中定义的 string 类来创建字符串类型，比如 string s;中的 s 就是一个字符串。

（3）C 风格字符串一般以字符串常量形式出现，或者将字符串存储在字符数组中（即 char 数组）。

（4）C 风格字符串的特点是：以空字符（'\0'）结尾，空字符在 ASCII 码中的值为 0。比如 char c[]={'a','b','c'};和 char d[]={'a','b','c','\0'};，在这两个字符数组中，只有 d 才是字符串，因为 d 是以空字符结尾的，而 c 却不是。

（5）空字符在 C 风格字符串中具有重要作用，因为很多处理字符串的函数，包括 cout 都是逐个处理字符串中的字符的，直到遇到空字符为止。

（6）当使用 cout 输出 C 风格字符串时，如果 cout 遇到的是一个字符的地址，则从该字符开始输出，直到遇到空字符（'\0'）为止。比如 char c[]={'a','b','c'}; char d[]={'a','b','c','\0'};，则 cout<<c<<endl;输出字符 abc 之后，会接着输出内存中其他位置的值，直到遇到空字符为止，因此会有一堆乱码；而 cout<<d<<endl;只输出字符 abc 后就结束输出。

2. 字符串常量

（1）字符串常量是使用双引号括起来的 0 个或多个字符，比如"a"或"abcd"。

（2）字符串常量中的每个字符都占据与 char 类型相同的字符空间，即 1 个字节。

（3）所有的字符串常量末尾都被系统自动地添加了一个空字符'\0'（注意，不是'\n'），作为字符串的结束标记。

（4）因为每个字符串的末尾都有一个空字符，所以字符串占据的内存空间比实际的字符数要多 1 个字节，比如"abc"就占据了 4 个字节的空间，'a'、'b'、'c'、'\0'各占 1 个字节。再比如"a\n"占据 3 个字节的空间，'a'、'\n'、'\0'各占一个字节。

（5）空字符（即'\0'在 ASCII 中的符号为 NUL，其值为 0）被作为字符串的结束标记，空字符不是字符串的组成部分，因此 cout<<"abc";并不会输出末尾的空字符。但空字符仍会占据 1 个字节的存储空间，因为它始终是一个字符。

（6）字符串常量的连接：两个相邻的且由空白（即空格、制表符或换行符）分开的字符串常量，会被自动连接成一个字符串常量。在连接时两个相连的字符串之间不会添加空格，第二个字符串的第一个字符将代替第一个字符串中的'\0'字符。比如：

```
cout<<"aa"
"bb"   "cc"<<endl;
```

输出的结果是：aabbcc。

（7）可以使用"\"符号将一个字符串写在两行上，但“\”符号的后面不能有任何内容（包括空格和注释），且第二行开头的任何内容都会被认为是字符串内容的一部分。比如 cout<<"aa\ee"<<endl; 将输出 aaee。注意：这里假设 ee 之前没有其他符号。

3. 字符串常量的类型及其使用

（1）字符串常量是 const char 类型的数组，即相当于字符串常量被存储在一个字符数组中，字符串常量本身表示的是该字符串所在内存空间中第一个元素的地址，但不能修改这个地址。

- "abc"，其类型与 const char c[4];中的变量 c 的类型相同。
- char c; "abc"=&c;，错误，试图修改字符串的地址。
- char *p; p="abc";，表示把字符串"abc"所在的第一个元素的地址赋给指针 p。
- char c[4]="abc";，则 c= ="def";是在比较两个对象的地址是否相等，而不是在比较字符串的内容是否相等。
- "abc"=="def";，同样是在比较两个字符串的地址，而不是在比较两个字符串的内容。

（2）正确使用字符串常量。因为字符串常量本身表示的是该字符串所在内存空间中第一个元素的地址，字符串常量是 const char 类型的数组，所以我们可以对字符串常量本身进行解引用、取址、下标运算等操作。

- 可以像使用数组名一样使用字符串常量。比如"abc"[1];。
- 可以对字符串常量进行解引用操作。比如 cout<<"abc";将输出字符 a，cout<<*("abc"+1); 将输出字符 b。
- 可以使用下标的方式访问字符串中的单个字符。比如 cout<<"abcd"[1]; 将输出字符 b。
- 可以将字符串常量赋给一个字符指针。比如 const char *p="abc"，表示使用字符串常量"abc"所在的地址初始化指针 p，指针 p 指向字符串第一个元素的地址。
- 可以对字符串常量进行取址操作，取址的结果是一个指向整个字符串数组的指针。比如 const char (*p)[4]=&"abc";，表示一个指向拥有 4 个元素的一维数组的指针，且数组中元素的类型是 const char。
- 注意：字符串常量的地址是不可修改的。比如 char a; "abc"=&a;错误。

（3）字符串常量和字符是两个不同的概念，字符的类型是 char，在 ASCII 系统中字符是整数的另一种写法；而字符串常量是 const char 类型的数组，它实际上代表的是字符串所在内存的

首地址。比如 char c="a"; 是错误的，因为"a"表示的是字符串常量 a 所在内存的首地址，正确形式是 char *p="a";再比如'\0'和空字符串常量""，虽然都是相同的值，但是'\0'的类型是 char，表示一个字符，而""则表示的是空字符所在内存的首地址。

4．使用 C 风格字符串

（1）使用 C 风格字符串有两种方法：一种是使用指针指向字符串常量；另一种是将字符串存储在字符数组中(即 char 数组)。比如 const char *p="abc"; char a[]="abc"; char b[]={'a','b','c','\0'};，但 char c[]={'a','b','c'}不是 C 风格字符串，因为末尾没有空字符'0'。

（2）使用字符串常量初始化字符数组时，其形式等价于使用字符串常量中的每个字符初始化数组中相应的各个元素比如 char c[4]="abc"; 等价于 char c[4]={'a','b','c','\0'};。

（3）只有字符数组才允许使用字符串常量以地址的形式进行直接初始化。

```
char a[]="abc";//正确；等价于 char c[4]={'a', 'b', 'c','\0'};
char a[]="abc";   char b[4]=a;//错误，不能用数组初始化另一个数组
char *p="abc";  char b[4]=p;  //错误，不能使用地址初始化一个数组
char c[]='a';  //错误，a 是字符而不是字符串
```

（4）若开始时没有对字符数组进行初始化，则以后要么使用库函数 strcpy()或 strncpy()进行赋值，要么就对字符数组元素逐个进行赋值。比如 char a[4];，则 a="abc"; 是错误的，此语句在试图改变数组 a 的地址。

（5）在 C++中字符串常量被视为常量，但并不一定所有的编译器都严格按照这种方式来实现。

（6）对于同一个字符串常量，若被程序多次使用，则编译器可能会存储该字符串常量的多个副本，也可能只存储其中的一个副本。可以使用不同的指针指向相同的字符串常量，然后输出指针所指向的地址加以验证是否指向相同的副本。比如 const char *p="abc"; const char *p1="abc"; cout<<(int*)p<<endl<<(int*)p1<<endl;，注意 cout<<p<<endl;将输出字符串，而不能得到其地址。

（7）使用字符数组时，改变字符数组的内容并不会影响到原始的字符串常量，因为数组与字符串所在的内存地址没有关系，数组是栈上分配的空间，而字符串常量则被存储在常量存储区中，这时数组的各个元素与字符串常量是没有关系的。比如 char c[4]="abc"，则 c[1]='e';是正确的，这里的改变并不会影响到字符串常量"abc"。

（8）从理论上讲，char *p="abc";是错误的，因为不能将 const 常量的地址赋给非 const 的指针，但 C++为了与现有的 C 兼容，而允许这种赋值方法，所以编译器不会报告错误。因为支持这种操作，就可能存在试图通过指针修改字符串常量的行为，这种行为在 VC++2010 中被禁止。比如 char *p="abc";，则 p[1]='e';这种行为是被禁止的。

示例 6.12：C 风格字符串

```
#include <iostream>
using namespace std;
int main()
{    //char c1[3]="abc";    //错误，字符串的末尾都有一个空字符，因此占据的内存空间比实际的字符数要
                           //多 1 个字节
     char *p;
     p="abc";     //正确，把字符串"abc"所在的第一个元素的地址赋给指针 p，C++为了与现有的 C 兼容，才
                  //允许将 const 常量的字符串地址赋给非 const 的指针
     cout<<("abc"=="def")<<endl;//此语句用于比较两个对象的地址是否相等，而不是比较字符串的内容
                                //字符串常量本身表示的是它所在内存的首地址，其是 const char 类型
                                //的数组，因此可进行解引用、取址、下标运算等操作
     cout<<*"abc"<<endl;         //输出字符 a
     cout<<*("abc"+1)<<endl;     //输出字符 b
     cout<<"abcd"[1]<<endl;      //输出字符 b
     const char (*p1)[4]=&"abc";
     const char c2[]="abc"; //只有字符数组才允许使用字符串常量以地址的形式进行直接初始化
                            //c2 有 4 个元素，等价于 char c2[4]={'a','b','c','\0'}
                            //末尾有个空字符'\0'
     //"abc"=c2;            //错误，试图改变字符串的地址
     //char c3[]='a';       //错误，'a'是字符，而不是字符串
     char c4[]="";          //正确，表示一个空字符串
     //char c5[1]=c4;       //错误，不能用数组初始化另一个数组
     //char c6[4]=p;        //错误，不能使用地址初始化一个数组
     char c7[4];
     //c7="abc";         //错误，不能改变数组的地址。要初始化数组 c7，只能逐个元素初始化或使用库函数
     char *p2="def";
     //p2[1]='a';        //错误，使用字符指针改变指针指向的字符串的值，VC++ 2010 禁止这种行为
     char c8[]="def";
     c8[1]='b';             //正确，使用字符数组时，改变字符数组的内容并不会影响到原始的字符串常量，
                            //因为数组与字符串它们所在的内存地址没有关系
     char c9[]={'a','b','c'};
     cout<<c9<<endl;    } //输出 abc 之后会有一堆乱码，因为当使用 cout 输出 C 风格字符串时，如果 cout
                          //遇到的是一个字符的地址，则将从该字符开始输出，直到遇到空字符（'\0'）为止
```

6.5.2　C 风格字符串的标准库函数

（1）使用 C 风格字符串的库函数时需要包含头文件<cstring>。

（2）传递给 C 风格字符串的库函数的参数都必须是指针或地址，可直接调用这些函数。

（3）使用以下的库函数时一定要确定数组 s1 有足够的空间，还要注意最后的空字符'\0'要占一个位置。

- strlen(s);返回 s 的长度，不包括字符串结尾的空字符'\0'。注意：该函数处理的对象一定是要以空字符'\0'结束的字符数组。比如 char a[ ]={'c','b'}; strlen(a);错误，因为当调用 strlen 函数时，系统将从实参 a 指向的地址开始搜索，直到遇到空字符为止，strlen()

返回这一段空间的长度，在该例中由于数组 a 没有空字符，所以最后的结果是无法预料的。

- strcmp(s1,s2)，比较字符串 s1 和 s2，如果 s1 大于 s2 则返回正数，如果 s1 小于 s2 则返回负数，如果 s1 等于 s2 则返回 0。
- strcat(s1,s2)，将字符串 s2 连接到 s1 后，并返回 s1。
- strcpy(s1,s2)，将 s2 复制给 s1，并返回 s1。
- strncat(s1,s2,n)，将 s2 的前 *n* 个字符连接到 s1 后面，并返回 s1。
- strncpy(s1,s2,n)，将 s2 的前 *n* 个字符复制给 s1，并返回 s1。

6.5.3 C 风格字符串的输入/输出

可以使用 cin、get、getline 函数对 C 风格字符串进行输入/输出，详见第 16 章。

第 7 章

函数和标识符的作用域专题

（1）学习本章需要具有较强的复杂类型分析知识，请认真阅读第 3 章。

（2）学习本章需要较强的指针和数组理论知识，请阅读第 6 章。

（3）函数类型是用于描述函数的，函数类型由其返回值类型、形参个数和形参类型表征。函数类型一般被称为“返回结果为某类型的函数”。注意：形参的名称并不是函数类型的一部分。比如 void f(int i, int j)，其中的形参名 i、j 不是函数类型的一部分。

（4）函数可以实现一种操作或者一种功能，在需要使用这种操作或者功能的地方只需直接调用函数即可，这样可以减少重复的程序内容。比如使用某函数实现计算矩形的面积，只需在出现计算矩形面积的地方直接调用该函数即可。

（5）函数是一种被动的实体，它是不会自动执行的（main 函数除外，main 函数是通过特殊方式进行调用的），只有被其他函数调用时才会执行。当函数被调用时，程序便转去执行被调用的函数，直到被调用的函数执行完毕，再转回来执行其后的程序。

（6）存储类区分符有 4 个：extern、static、auto、register；类型限定词有 2 个：const、volatile。

7.1 函数基本语法规则

7.1.1 函数声明、定义及形参的语法规则

函数声明的语法形式：<存储类区分符> 返回类型 函数名(形参 1,形参 2,形参 3,...);（注意：有个分号）。

函数定义的语法形式：存储类区分符 返回类型 函数名(形参 1,形参 2,形参 3,...){...}。

1. 基本要求

（1）存储类区分符是可选的，若有存储类区分符，则只能是 extern 或 static，不能是其他存储类区分符（如 register），默认是 extern（与变量默认是 auto 不一样）。

（2）函数的返回类型不能是函数或数组类型。

（3）函数必须有返回类型，省略返回类型是错误的；若函数没有返回值，其返回类型应是void。比如 f(){}是错误的，正确的形式应是 void f(){}；而 f(); 则表示调用一个函数。

（4）函数名后面的小括号不能省略，即使没有形参也应使用空的小括号。函数名后面的小括号是程序识别函数的标志，比如 int f{}是错误的；int f;表示声明一个变量 f，而不是声明一个函数 f。

（5）函数的形参是可选的。也就是说函数可以没有形参。若函数没有形参，可以使用 void 或空的小括号来说明。比如 int f(){};和 int f(void){} 都表示定义了一个没有形参的函数 f。

（6）函数体应是使用大括号“{}”括起来的复合语句，而不能是一条单独的语句，即使是空语句也应使用空的大括号。比如 void f(); 在语法上没有错误，但这只是函数声明，这个函数没有函数体； void f() cout<<"A"<<endl; 是错误的，函数体应使用大括号括起来； void f(){} 正确；void f(){cout<<"A"<<endl;}正确。

（7）函数体是一个语句块，这就形成了一个新的作用域，在函数体中定义的变量只能在该函数内可见，这种变量相对于定义它的函数而言是局部的，因此也被称为局部变量。

（8）示例：int f(int a, int b){...}表示定义了一个函数，其名称为 f，函数返回值的类型为 int，且带两个 int 类型的形参变量 a 和 b。假设函数 f 具有计算矩形面积的功能，其返回值就是矩形面积值，两个形参分别表示矩形的长和宽，那么当需要使用这个函数完成计算矩形面积的功能时，只需向函数提供两个参数（这时被称为实参），再调用函数，并将计算结果作为面积值即可；如 int c; c=f(5,4); 表示计算长为 5、宽为 4 的矩形面积，并将计算结果赋给变量 c，这里的 f(5,4)是函数调用的表达式，程序执行到 f(5,4)时，将转到被调用的 f 函数处（即定义 f 函数的地方）执行 f 函数，当 f 函数执行完之后，再返回被调用处，继续向下执行。

2. 区分函数声明与定义

（1）函数声明：与变量声明相同，函数声明也只提供了类型（形参类型和函数的返回类型）。函数声明明确了函数名、函数类型及形参类型、个数和顺序，这样在调用该函数时就可以对函数名是否正确，实参与形参的类型和个数是否一致进行检查，但不包括函数体，这些检查都是在编译时完成的。

（2）函数声明只包括函数定义除函数体外的部分，即函数头，然后在后面加个分号。

（3）函数定义：若为函数提供了函数体，那么这就是函数定义。函数体是函数的可执行语句，作为程序的一部分被保存在内存中，因此也可以说函数定义消耗了内存。函数定义是一个完整、独立的函数单位。

（4）函数定义也是函数声明，这与变量声明和定义相同。

（5）函数原型：包括函数的返回类型、函数的形参个数及形参类型，这在本质上与函数声

明是一样的。使用函数原型主要是为了代替老式 C 语言中的函数说明语法。

3．声明和定义函数的规则

（1）在函数定义之前，应先对函数进行声明，然后才能调用该函数。

（2）和变量一样，同一个函数只能定义一次，可以声明多次，但每次声明都必须相同，即每次声明都应有相同的返回类型和形参表。注意：若形参表不同，则是函数重载，这时声明的是两个函数。比如 void f(); 和 int f(); 是错误的；void f();和 void f(int i); 声明的是重载的两个函数，这不是错误。

（3）在一个函数内不可以定义另一个函数，即函数定义不可以嵌套，但可以在函数内调用或者声明另一个函数。比如 int f(){ int g(){} }是错误的，不能在 f 函数内定义函数 g；int f(){ int g(); } 正确，可以在函数内声明另一个函数。

（4）在声明函数时可以同时声明或定义变量，但在定义函数时不允许这样做。注意：变量不能是 void 类型的。比如 int f(), a, b; 和 int a=1,f();是正确的；int f(){}, a;和 int a, f(){};是错误的；void f(), a;也是错误的。

4．函数形参的规则

（1）形参：在定义函数时，把用于接收传递值的变量或对象称为形参，每个形参都有相应的类型和名称。比如 int f(int i, float a, int *p){};，其中的变量 i、a、p 都是形参。

（2）函数的形参和实参主要用于数据的传送。

（3）形参表中形参的名字不能相同。比如 void f(int i, float i); 是错误的

（4）若有两个相同类型的形参，则形参必须声明两次，而不能像声明变量一样使用逗号声明两个变量。比如 int f(int i, j){} 错误，形参 i 和 j 必须声明两次，正确形式为 int f(int i, int j){}。

（5）声明函数时形参名字可以省略，因为函数声明只提供类型，而形参名字不是类型的一部分，所以声明和定义函数时形参名字也可以不同。比如 int f(int ,int, double); 与 int f(int i, int j, double k);是等价的。

（6）在定义函数时若形参仍没有名字，则会导致在程序中无法使用该形参。比如 void f (int i, int){...};，在函数体中无法使用第二个形参，因为它没有名字。

（7）声明和定义函数时必须要有相同的形参类型和函数名。当然，若函数的形参类型不同，在 C++中并不一定是错误，程序会把它当作函数的重载处理。

（8）注意：除(void)外，不能指定 void 类型的形参。比如 void f(void i);和 void f(int i, void); 错误，void f(void)正确，表示函数无形参。

（9）多个形参之间使用逗号分隔，比如 void f(int i, int j, int k){};。

（10）若形参中出现存储类区分符（即 auto, static, register, extern），则将被忽略。形参的存储类区分符只能是 register 和 auto。所有形参都具有自动存储期（即 auto）。

（11）函数形参的作用域是从形参声明处开始，一直到函数体结束处，因此在函数体内部不能定义与形参具有相同名字的变量。

（12）若形参为非引用的 const 类型，编译器会将其视为非 const 类型的形参（指针除外），因为在实参传递给形参时，并不考虑形参是否为 const；但在函数体中仍把形参当作 const 类型处理，编译器的这种处理方法可能是为了与 C 语言兼容。比如 void f(const int i);，编译器会认为该函数的声明形式是 void f(int i);但是在函数 f 中，形参 i 仍是 const 类型的；void f(const int i); 和 void f(int i);，这两个声明是相同的；int f(const int i); 和 void f(int i);，第二个声明是错误的，这并不是函数重载，因为它们的形参表是一样的，仅仅是返回类型不同。若形参为引用或为指向 const 对象的指针（比如 const int *p），则有无 const 限定，并不遵守这条规则；若 const 仅仅是修饰指针（比如 int *const p），则遵守这条规则，具体见后文。

示例 7.1：函数声明、定义、形参的基本语法规则

```
#include <iostream>
using namespace std;
void f1(auto int i){}  //形参的存储类区分符只能是 register 和 auto。所有形参都具有自动存储期。若
                       //形参中出现其他存储类区分符（即 static、extern），将被忽略
void f2(void){cout<<"A"<<endl;}  //若函数没有形参，可以使用 void 来说明，也可以使用空的
                                 //小括号来表示。此语句与 void f2(){}相同
//void f3{}            //错误，函数名后必须有小括号；要声明函数，则应在函数名后面加上小括号
//f4(){}               //错误，函数必须有返回类型，若没有返回值，则应在函数前使用 void 关键字
//void f5() cout<<"A"<<endl;  //错误，函数体应是使用大括号“{}”括起来的复合语句，即使是空语句
                              //也应使用空的大括号
//register void f6(){} //错误，函数的存储类区分符只能是 extern 或 static，默认是 extern
void f7(){}
//void f7(){}          //错误，同一个函数只能定义一次
void f8();
//int f8();            //错误，声明同一个函数应有相同的返回类型和形参表，仅返回类型不相同是错误的
int f8(int i);         //正确，若形参表不同，则是函数重载，这时声明的是两个函数
void f9(){void f10(); f10(); }  //在函数定义之前，应对该函数先进行声明，然后才能调用该函数
void f10(){}
//void f11(){ void f12(){} }     //错误，在一个函数内不可以定义另一个函数
int f13(), a, b=1;               //在声明函数时可以同时声明或定义变量
//int f14(){}, a1;               //错误，在定义函数时不能同时声明或定义变量
//void f15(), a2;                //错误，因为变量不能是 void 类型的
void f16(int, int);              //声明函数时形参名字可以省略，声明和定义函数时形参名字也可以不同
//void f17(void i);     //错误，不能指定 void 类型的形参，若函数没有形参，则应是 void f17(void);
 //int f(int i, j){}    //错误，若有两个相同类型的形参，则形参必须声明两次，而不能像声明变量一
                        //样，使用逗号声明两个变量。正确形式为 int f(int i, int j){}
//void f18(int i){ int i;} //错误，函数形参的作用域是从形参声明处开始，一直到函数体结束处，因
                           //此在函数体内部不能定义与形参具有相同名字的变量
int main(){
  f2(); } //函数是一种被动的实体，不会自动执行，只有被其他函数调用时才会执行
```

7.1.2　函数调用、实参、返回值、return 语句语法规则

1．函数调用和实参语法规则

（1）实参：调用函数时，把传递给被调用函数的值称为实参。实参既可以是变量，也可以是常量，还可以是包含多个运算符的表达式。比如函数 void f(int a, int b){}，则函数调用 f(1,2+3); 中的数值 1 和表达式 2+3 都是实参，其中数值 1 会被传递给形参 a，表达式 2+3 的计算结果被传递给形参 b。

（2）函数调用时使用实参初始化对应的形参，并转移到被调函数，主调函数的执行被挂起，被调函数被执行。被调函数的运行是以形参的定义和初始化开始的，也就是说，在被调函数运行时，第一件事就是创建与形参相对应的变量，并将该变量初始化为调用此函数时传递过来的实参的值。

（3）函数调用：C++使用一对小圆括号（即调用运算符）来实现函数的调用，在调用函数时需要指定函数的名字，并在小括号中填上相应的实参。调用函数不需要指定函数的返回类型。若调用时有多个实参，则各实参之间使用逗号隔开。比如 f(2,3)是一个函数调用，这里提供了两个整型实参；f()也是一个函数调用，这里函数没有实参；int f(2,3);是错误的，函数调用不需要指定返回类型。

（4）函数调用使用调用运算符来实现，因此函数调用其实就是一个表达式，函数调用可以出现在需要表达式的地方。

（5）调用运算符也是运算符，因此需要一个操作数并且有一个结果值。调用运算符的操作数就是函数名和函数名后小括号中以逗号分隔的表达式（有可能为空），该表达式被称为实参。调用运算符的结果类型就是函数的返回类型，结果值就是函数的返回值（return 后的表达式就是函数的返回值）。比如 f(2, 2+3); 是一个调用函数 f 的函数调用表达式，其操作数是函数名 f 和括号中的表达式 2 和 2+3，括号中的两个表达式也就是调用函数时的实参，实参将被传递给函数 f 相应的形参，函数 f 的返回值和类型依函数的原型或者定义而定。

（6）函数调用返回的值可用在相应的表达式中进行运算，但若函数没有返回值，则不能用在表达式中。比如 int f();，在调用函数 f 之后返回的值可以用在表达式中，如 2+f()+3; 再比如 void g(); ，因为函数没有返回值，所以调用函数 g 不能用在表达式中，如 2+g()+3;是错误的。

（7）注意：调用运算符与强制类型转换符都是小括号，但它们在程序中是很容易区分的。

（8）函数调用时实参的个数与形参的个数必须相同，比如 void g(int i){...}，则调用 g(1,2) 是错误的，而调用 g(3)是正确的。

（9）函数调用时实参的类型与形参的类型必须相同，或者能隐式转换为形参的数据类型。比如 void g(int i){} void f(int *p){}; int a=2; float b=2.2; int c[2]={0}; int *p=&a;，则调用 g(2);、g(a);、g(b);、g(*p);、f(c);、f(p);、f(&a);都是正确的；而调用 g(c);、g(p);、f(2);、f(a);、f(b);、f(&b)

都是错误的，因为实参的类型和形参的类型不同，或者不能被隐式地转换为形参的类型。

（10）在一个函数中不可以定义另一个函数，但可以调用另一个函数，即函数调用可以嵌套。比如 void f(){ f1(); f2();}，在函数 f 中就调用了函数 f1 和 f2，但 void f(){void f1(){}}则是错误的，因为不能在一个函数中定义另一个函数。

（11）函数调用时，对函数的实参及实参中子表达式的求值顺序没有规定。比如 f1(f2(), f3(), f4()+f5())，其中的 f2, f3, f4, f5 的调用次序是不确定的；再比如 f(i, i++);，其中的 i 和 i++也并未规定究竟是先执行 i 还是 i++，因此不同编译器可能会有不同的结果。为了编写可移植的程序，在调用函数时，最好不要在实参中出现改变实参值的表达式。

2. 函数的返回类型与 return 语句语法规则

（1）return 语句有两个作用：①终止当前函数的执行；②返回函数的计算结果。无论哪种情况，只要遇到 return 语句，程序就会把控制权返回到调用该函数的地方继续向下执行。

（2）return 语句：也被称为返回语句，return 语句就是在关键字 return 后面接一表达式并以分号结束，或直接接一个分号的语句。比如 return 3;，使用常量表达式；return a+b;，假设 a 和 b 为 int 类型，则表示使用一般的表达式；return ;，表示直接后接一个分号。

（3）不带表达式的 return 语句的作用仅是终止当前函数的执行，并返回到调用该函数的地方继续执行，不会返回值。这种语句用于函数返回类型为 void 的情况，比如 void f(){return;}。

（4）带表达式的 return 语句的作用是返回函数的计算结果，并终止当前函数的执行，回到调用该函数的地方继续执行。这种 return 语句常用于函数要求返回值的情况。

（5）注意：return 0; 与 return; 是不同的，return 0;表示返回值 0，并不表示 return 不返回值；return;才表示不返回值。比如 void f(){return 0;}是错误的，但 void f(){return;}是正确的。

（6）在一个函数中可以包含多条 return 语句，return 语句可以出现在函数体中的任何位置，在没有返回值的情况下也可以没有 return 语句。

（7）被执行的函数，在函数体中遇到 return 语句时，则函数结束执行。若函数体中有多条 return 语句，则函数在遇到第一条 return 语句时就结束执行；若函数体中没有 return 语句，则函数在作为函数体最末的“}”处（即右花括号处）结束执行，这时也相当于执行了一条不带表达式的 return 语句。若函数的返回类型为 void，则应使用不带表达式的 return 语句或者不使用 return 语句。例如：

- void f(){}，表示函数 f 没有返回值，其函数在右花括号处结束。
- void g(){if (...) ...return; else ...return;}，正确，函数遇到 return 则结束。
- void g(){ return 2;}，错误，使用 return 语句返回了值，但该函数不要求返回值。

（8）若函数有返回值，则函数的返回类型应是除 void 之外的其他类型，且函数体必须使用带表达式的 return 语句返回一个值；若 return 语句的表达式的类型与函数的返回类型不同时，

则将表达式的类型隐式转换为函数的返回类型之后再将其值返回，否则是错误的。这时的效果等同于将表达式的值赋给与函数的返回类型相同的对象（或变量）。例如：

- int f(){return 3+2;}，表示函数 f 返回一个 int 类型的值，并且在函数体中使用 return 3+2; 返回值 5。
- int f(){return ;}，错误，函数 f 要求返回一个 int 类型的值，但函数体没有返回值，因为 return 之后没有任何值。
- int f(){int *p; return p;}，错误，返回的值的类型（这里为整型指针）不能隐式转换为函数所要求的返回类型（这里为 int 类型），
- int f(){return 2.2;}，正确，浮点数 2.2 将被隐式转换为 int 类型之后返回。

（9）函数的返回类型不能是数组或函数，但可以是指向数组或函数的指针。

```
int f()[2];          //错误，试图返回一个 int 类型的数组
int [] f();          //语法错误，[]是后缀运算符，必须在标识符的后面
int f()();           //错误，试图返回一个函数
int (*f())[2];       //正确，返回一个指向带两个 int 类型元素的数组的指针。注意，小括号必须要，否则就是
                     //返回数组了
int (*f())(int i); //正确，返回指向带一个 int 类型的形参，返回类型为 int 的函数的指针
```

（10）注意：在早期的 C 语言中，不带表达式的 return 语句返回的是一个不确定的值，因此若函数有返回值，但却使用了不带表达式的 return 语句或没有遇到 return 语句，则函数将返回一个不确定的值。这种行为在 C++中是不允许的。

（11）当函数返回的是引用时，这时返回的值是一个左值，则可以对函数的返回值进行赋值，就是对函数返回时所引用的对象进行赋值。比如 int a=3; int &f(){return a;}，则 f()=4;是正确的，此时 a 的值被修改为 4。

（12）当函数返回的是指针时，由于指针是指向某个地址的，因此可以对返回的地址解引用后进行赋值，但不能对返回的指针直接进行赋值，因为它并不是左值。比如 int a=1; int *f(){return &a;}，则*f()=3; 是正确的，但 int b=2; f()=&b; 则是错误的。

（13）不能返回指向局部变量（局部变量存储在栈内存中)的指针或引用，因为局部变量在函数结束时会自动消亡。若返回指向局部变量的指针，那么这个指针将会指向一个被释放的内存空间，因此指针就指向了不确定的内存。例如：

```
int *f(){int a=2; int *p=&a; return p;}
int main(){int *pp=f(); cout<<*pp<<endl; cout<<*pp<<endl;}
```

指针 p 指向栈内存的地址，函数 f 结束执行后该内存消失。在 main 函数中使用此地址，若地址未被占用，则程序可能会输出正确的结果，但不保证一直不会出错。

```
char *g(){int *p="abc"; return p; }
```

这是正确的，因为字符串常量"abc"不是存储在栈内存中的，因此指针指向的并不是栈内存的地址。字符串存储在常量存储区中。

```
char *h(){int p[]="abc"; return p; }
```

返回的是栈内存的地址，这里的 p 是数组的名称，这个数组的内存空间是在栈内存中开辟的，然后将字符串"abc"存储在该栈内存中，因此 p 指向的并不是字符串常量的指针，而是一个栈内存的地址。

（14）注意：在 C++中 main 函数必须有返回类型，若 main 函数没有返回值，则是 void 类型，main 函数的返回类型不能是 double 类型。

（15）注意：main 函数是个特例，若 main 函数有返回值（即非 void 返回类型），但在 main 函数中没有 return 语句，则等价于返回 0，意味着程序成功执行完成。若 main 函数有返回值，但在 main 函数中有 return 语句，则 return 语句之后必须返回一个值，而不能使用空语句，否则会出错。

（16）函数的返回类型若是非引用类型，则在求解表达式时，需要在某处存储运算的结果，则编译器会创建一个没有命名的临时对象，然后使用函数的返回值初始化这个临时对象。比如 int f(){return 1; }，则函数返回的是一个未命名的类型为 int 的临时对象，然后使用返回值 1 来初始化这个临时对象。此处主要是为了理解临时对象的概念，在此不进行深讲。

（17）函数的返回类型若是引用类型，则在函数调用和返回结果时，不会复制函数的返回值，也没有临时对象，函数返回的是引用的对象本身。比如 int i=1; int &f(){return i; }，则函数不会复制返回值，也没有临时对象，而是返回引用的对象本身 i。

示例 7.2：函数调用、实参、返回类型、return 语句的基本语法规则

```
#include <iostream>
using namespace std;
int i=1;
void f2(){int a=1; return;//f2 在此处结束，有多条 return 语句时，遇到第一条 return 语句就结束
if(a<2) {a++; return; } else return; } //if 语句不会被执行
void f3(){}                          //正确，不使用 return 语句时，函数在右大括号“}”处结束
//void f4(){ return 0;}              //错误，函数要求不返回值，应使用不带表达式的 return 语句
//int f5(){return; }                 //错误，函数要求返回一个 int 类型的值，但函数体没有返回值
//int f6(){}                         //错误，同上，函数没有返回值
//int f7(){int *p=0; return p;}      //错误，返回值的类型不能隐式转换为函数所要求的返回类型
int f8(){return 2.2;}                //正确，浮点数 2.2 将被隐式转换为 int 类型之后作为函数的返回值
int f9(){return 3+2;}                //正确，函数返回值 5
//int f10()[2];                      //错误，试图返回一个 int 类型的数组
//int [] f11();                      //语法错误，[]是后缀运算符，必须在标识符的后面
//int f12()();                       //错误，试图返回一个函数
int (*f13())[2];                  //正确，返回一个指向带两个 int 类型元素的数组的指针，注意小括号不能省略
int (*f14())(int i);                //正确，返回一个指向带一个 int 类型形参，且返回类型为 int 的函数的指针
//int &f15(int a){return a;}        //不推荐返回局部变量的引用或指针
int *f16(){return &i;}
int f17(int a, int b){ cout<<"A"<<endl;  return a+b;  }
//main 函数开始
int main(){
```

```
 int a=1;        int b=2;        int c=3;        int *p=&a;
f17(2,3);  //这是一个函数调用，跳至函数 f17，转去执行 f17 中的内容，执行完毕后再返回此处执行
           //后面的语句。函数是一种被动的实体，不会自动执行，只有被其他函数调用时才会执行
f17(3,4);  //函数调用时使用实参初始化对应的形参，这里的数值 3 和 4 是实参，函数 f17 的形参 a
           //和 b 分别被初始化为 3 和 4
//int f17(3,4); //错误，函数调用时不需要指定函数的返回类型
f17(a+5,4+5);   //正确，函数调用时的实参可以是任意的表达式
b=f17(3,3)+4;   //正确，函数若有返回值，则函数调用可以用在相应的表达式中
//b=f2()+2;     //错误，函数没有返回值时不能用在表达式中，这里的函数 f2 没有返回值
//f17(3);       //错误，函数调用时的实参个数必须与形参的个数相同
//f17(2,p);     //错误，函数调用时的实参的类型必须与形参的类型相同，或者能隐式转换为形参
                //的数据类型。这里的指针 p 不能被转换为 int 类型。
f17(2.2, 'a'); //正确，数值 2.2 和字符'a'都能被隐式转换为 int 类型
c=f17(a,a++);   //不推荐这样做，在函数调用时，对函数的实参及实参中子表达式的求值顺序没有规
                //定。这里的 a 和 a++也并未规定究竟是先执行 a 还是 a++，为了编写可移植的程序，
                //在调用函数时，最好不要在实参中出现改变实参值的表达式
//f16()=&b;     //错误，函数返回的地址并不是一个左值，因此不能对其进行赋值
*f16()=4;       //正确，当函数返回的是指针时，可以对返回的地址解引用后对其进行赋值。这里间接修改
                //了函数返回时所指向的全局变量 i 的值
cout<<i<<endl; }
```

7.2　函数参数传递

7.2.1　指针形参和引用形参

1. 按值和按引用传递参数

（1）按值传递参数：若形参是非引用类型的，在调用函数时，则复制实参的值初始化形参，这种方式被称为按值传递参数。即非引用类型的形参都是使用按值传递参数的。比如 void f(int a); int b=1; f(b);，因为函数的形参 a 是非引用类型的，所以在调用函数时传递的是实参 b 的一个副本，即使用复制的实参的值初始化形参。

（2）按引用传递参数：若形参是引用类型的，在调用函数时，那么形参就是传递过来的实参的一个别名，这种方式被称为按引用传递参数。比如 void f(int &i); int a=1; f(a);，其中形参 i 是引用形参，实参是按引用传递的，因此形参 i 是实参 a 的一个别名。

（3）按值传递与按引用传递的区别：按引用传递时形参是实参的别名，因此在函数中对形参的值的改变会影响实参的值；而按值传递，因为形参使用的是复制的实参的值（也可以认为是实参的副本）进行初始化的，因此在函数中对形参的值的改变不会影响实参的值。

（4）只有形参的类型是引用类型时，才按引用传递参数。即使形参是指针，也是按值的方式传递地址的，而非按引用传递。详见下一部分“指针形参”。

（5）按值传递较大的对象时，会付出较多的时间和存储空间，而且无法改变实参值。当遇

到这种情况时，可以使用指针或引用来解决。虽然使用指针也是按值传递，但是复制的数据相对来说小得多（复制的是实参指针指向的地址），而且指针形参也能间接改变实参所指向的对象的值。

（6）按值传递时，会创建一个临时变量，然后将实参的值传递给该临时变量，并使用临时变量初始化形参。

（7）按值传递与按引用传递形参示例如下：

```
void f(int a, int b){a=1; b=2;} int main(){int c=3,d=3; f(c,d); }
```

因为形参是非引用类型的，所以按值传递参数。形参 a 和 b 接收到的是实参 c 和 d 的副本，因此对形参 a 和 b 的改变不会影响实参的值，实参 c 和 d 的值仍为 3。

```
void f(int &a, int &b){a=1; b=2} int main(){int c=3,d=3; f(c,d);}
```

因为形参是引用类型的，所以按引用传递参数。形参 a 和 b 是实参 c 和 d 的别名，因此对形参 a 和 b 的改变会影响实参的值，调用函数 f 后，c 的值变为 1，d 的值变为 2。

2. 指针形参

（1）当形参为指针时，其实参也是按值传递的（因为指针是非引用类型的）。即在调用函数时，复制的将是指针实参的值（即实参指针所指向的地址），但是指针形参可以改变指针实参所指向的地址中的值。例如：

```
void f(int *p){*p=3; int b=2; p=&b; *p=4;} int a=1; int *p1=&a; f(p1);
```

① 形参 p 得到的将是实参 p1 的值。注意：实参 p1 的值是变量 a 的地址，而不是变量 a 的值 1，因此形参 p 和实参 p1 指向的是相同的地址。

② 函数 f 中的语句*p=3;将改变实参 p1 所指向的对象的值。也就是说，*p=3;改变了实参 p1 指向的变量 a 的值，这样就间接修改了变量 a 的值。

③ 这里 p=&b;语句改变了形参 p 所指向的地址，但因为指针是按值传递的，所以形参 p 的改变不会对实参 p1 产生影响，实参 p1 指向的仍是变量 a 的地址。因此，后面的*p=4; 语句不会改变实参 p1 所指向的变量 a 的值（因为 p 和 p1 指向的是不同的地址），它改变的是变量 b 的值。

④ 调用函数 f 后 a 的值为 3 而不是 4。

（2）因为形参指针会改变实参所指对象的值，所以当形参为指针时可以用来交换两个实参的值。比如 void f(int *p1, int *p2){int temp=*p1; *p1=*p2; *p2=temp; } int a=1; b=2;，则 f(&a,&b); 就交换了 a 和 b 的值。本例中，*p1 表示的是指针 p1 所指向的地址中的值，即变量 a 的值，因此改变*p1 就是间接改变变量 a 的值。注意：以下函数不能交换实参的值。void g(int *p1, int *p2){int *temp=p1; p1=p2; p2=temp; }，这里只交换了 p1 和 p2 两个指针所指向的地址。同理，void h(int p1, int p2){int temp=p1; p1=p2; p2=temp; }也没有交换实参的值，只是将实参的副本的值进行了交换，并没有影响实参的值。

3．指针形参与 const

（1）若形参为非引用非 const 类型的（指针除外），则实参既可以是 const 类型的，也可以是非 const 限定的。比如 void f(int i){}; const int a=1; int b=2;，则 f(a)和 f(b)都是正确的。

（2）若形参为非引用 const 类型的（指针除外），编译器会将其视为非 const 类型的形参，因为在实参传递给形参时，并不考虑形参是否为 const 类型的；但在函数体中仍把形参当作 const 类型处理。这种使用方式可能是为了对 C 语言的兼容。比如 void f(const int i);，编译器会认为该函数的声明形式是 void f(int i);，但是在函数 f 中，形参 i 仍是 const 类型的；再比如 void f(const int i); void f(int i);，这两个声明是相同的；int f(const int i); void f(int i);，第二个声明是错误的，因为它们的形参表是一样的，仅仅是返回类型不同，这并不是函数重载。若形参为引用的或指向 const 对象的指针（比如 const int *p），则有无 const 限定，并不遵守这条规则；若 const 仅仅修饰的是指针（比如 int *const p），则遵守此规则。

（3）基本原则：赋值运算符的两个操作数都是指向相容类型的限定或非限定形式的指针，则左操作数所指向的类型应具有右操作数所指向的类型的全部限定词。

（4）若形参为指针且指向非 const 类型的对象（比如 void f(int *p)），则实参不能是 const 类型对象的地址。比如 void f(int *p){} const int a=1;，则 f(&a);是错误的。注意：int *const p 这种类型的指针是指向非 const 类型对象的。

（5）若形参为指针且指向 const 类型的对象（比如 void f(const int *p)），则实参可以是 const 类型或非 const 类型对象的地址。比如 void f(const int *p){} const int a=1; int b=2;，则 f(&a)和 f(&b)都是正确的。

（6）对于二级及多级指针，在 const 与非 const 之间进行赋值将不再安全。比如 int *p1;，则 const int **p2=&p1;可能不再安全，这与一级指针的情况有些不同。对于二级及多级指针，不同的编译器可能会有不同的处理方法。

4．引用形参与 const

（1）引用就是某个对象的另一个名字（即别名），因此对引用形参的操作其实就是对实参进行操作。比如 void f(int &p1, int &p2){int temp=p1; p1=p2; p2=temp; } int a=1; b=2;，则 f(a,b);交换了实参 a 和 b 的值，即调用函数 f 后 a=2; b=1。形参 p1 和 p2 都是引用的，因此 p1 是实参 a 的别名，p2 是实参 b 的别名。对引用的操作就是对实参的操作，因此函数 f 中有关形参的操作都会影响实参。

（2）若形参为非 const 引用的（比如 void f(int &p)），则实参应遵守以下规则。

① 不允许使用需要转换为形参类型的实参。也就是说，在参数传递时，不允许存在类型转换（包括隐式类型转换）。比如 void f(int &p){}; short a=1;，则 f(a);是错误的。

② 实参不能是一个右值。比如 int a=1,b=2; void f(int &p){};，则 f(2); f(a+b)都是错误的，因

为 a+b 的结果是右值。

③ 实参不能是 const 类型的对象，这与指针原理相同。比如 void f(int &p){} const int a=1;，则 f(a);是错误的。

（3）若形参为 const 引用类型的对象(比如 void f(const int &a))，则实参可以是以下类型。

① 实参可以是 const 类型或非 const 类型的。

② 实参可以是一个右值、临时对象或表达式。

③ 原理：若函数调用时的实参不是左值，或者与引用类型形参的类型不匹配时，则 C++会创建正确的临时对象（注意：若实参是匹配的左值，则不会创建临时对象），并将实参的值传递给该临时对象，让形参引用该对象。因此，对于形参为 const 引用类型的对象，可以接受 const 类型或非 const 类型的实参，实参也可以是一个右值、临时对象或表达式。例如：

- void f(const int &a){}，则 f(2.2);是正确的，首先将 double 类型的 2.2 转换为 int 类型，然后保存为一个 int 类型的临时对象，最后将这个临时对象传递给形参 a;。
- void f(const int &a){} float b=3.3;，则 f(b);正确，首先把 float 类型的 b 转换为 int 类型，然后保存为一个 int 类型的临时对象，最后使用形参 a 引用这个临时对象。

（4）注意：在早期版本的 C++中，若引用是非 const 的（比如 int &a），则可以接受类型不匹配的对象、右值、临时对象，比如 int &a=3.3;，在早期版本的 C++中是正确的，但在新版本中是错误的。原因在于传递参数时引用形参并没有改变实参的值。比如 void f(int &a);，若有调用 float b=2.2; f(b);，则程序将创建临时对象，然后形参 a 引用该临时对象，这样在函数 f 中对 a 的改变，将不能影响实参 b 的值，而在新版本的 C++中将不允许创建这种临时变量。

7.2.2 数组形参

1. 重要规则

（1）不能复制数组，在传递数组时，传递的是指向实参数组的第一个元素的地址。

（2）函数是不能直接传递数组的，不存在数组类型的形参，当函数的形参为数组类型时，形参数组名会自动转换为指向第一个元素的指针，对于多维数组，同样会转换为指向第一个元素的指针，这时形参是一个指针，不再是数组名，形参具有指针的性质，即可以改变指针指向的地址。由规则可见，转换后的指针类型与数组名的类型是相同的。比如 void f(int a[2][3][4]);，则形参 a 会转换为指向第一个元素的指针，因为数组名就是指向数组第一个元素的指针，所以转换后的指针类型与数组名的类型相同，其类型为(*a)[3][4]。

（3）凡是返回 type 的函数的形参都将调整为指向返回 type 的函数的指针，这时形参是一个指针，不再是函数名，形参具有指针的性质，即可以改变指针指向的地址。比如 void f(int p());，其中的形参是返回 int 类型的无参函数 p，这时将转换为 int (*p)();，即指向返回 int 类型的无参

函数的指针。注意：这时 p 是指针，不是函数。

（4）数组形参的转换原理：因为转换后的指针类型与数组名的类型相同，所以只需分析出数组形参中存储的元素的类型，然后使用指针指向该类型即可。下面通过示例进行讲解。

- int a[3];转换为 int *a;，数组存储的元素为 int 类型，应转换为指向 int 类型的指针。
- int a[2][3];转换为 int (*a)[3];，数组存储的元素为 int [3]，应转换为指向 int [3]类型对象的指针，即 int(*a)[3];。
- int a[2][3][4]; 转换为 int (*a)[3][4];，规则同上。
- int *a[3];转换为 int **a;，a 首先与[3]结合，说明 a 是数组，此数组中存储的元素是 int*类型的对象，因此形参应转换为指向 int*类型对象的指针，即 int **a;。
- int *const a[3];转换为 int *const *a;，a 首先与[3]结合，说明 a 是数组，此数组中存储的元素是 int*const 类型的对象，因此形参应转换为指向 int*const 类型对象的指针，即 int *const *a;。
- int *const a[3][4]; 转换为 int *const (*a)[4];，a 首先与[3]结合，说明 a 是数组，此数组中存储的元素是 int*const[4]类型的对象，因此形参应转换为指向 int*const[4]类型对象的指针，即 int *const (*a)[4]。
- const int *a[3][4]; 转换为 const int* (*a)[4];，a 首先与[3]结合，说明 a 是数组，此数组中存储的元素是 const int* [4]类型的对象，因此形参应转换为指向 const int* [4]类型对象的指针，即 const int * (*a)[4];。

2．一维数组形参

（1）当函数的形参为数组类型时，数组形参会被转换为指向第一个元素的指针。

① void f(int *a); void g(int a[]); void h(int a[11]); 这 3 个函数的形参是等价的，其形参的类型都是 int *，即指向 int 的指针。

② 在函数的形参中为数组指定大小是没有意义的，即 void h(int a[11]); 中的数组形参的长度 11 是没有意义的；它与 void h(int a[]); void h(int *a); 是相同的。

③ 可以改变数组形参指向的地址，因为数组作为形参时会自动转换为指针，具有指针的性质，可以改变指针的地址。这是与数组名的不同之处，数组名的地址是不能改变的。比如 void f(int a[]){ int b=1; a=&b;}，其中 a=&b;是正确的，因为形参 a 是一个指针，而非数组名。注意：这条规则只适用于数组是形参时，当数组不是形参时，数组名与指针并不完全等同。

（2）注意：int *; 和 int []只有在函数头和函数原型中才是相同的，在其他地方并不是相同的。

（3）因为 int *; int []; int [N];在函数形参中是等价的，因此对于拥有这 3 种形参的函数声明是相同的，它们不是函数重载。比如 void f(int *p); void f(int p[]); void f(int p[3]); 是 3 个相同的

声明，因此只能对其中一个进行定义，而不能重复定义两个以上，如 void f(int *p){} void f(int p[]){}就是重定义错误。

（4）若没有改变数组形参所指向的地址，则对数组形参的操作将与指针相同，它会间接改变传递进来的数组的值。比如 void f(int a[]){a[0]=2;} int b[4]={0};，则调用 f(b); 之后，将把数组 b 中第一个元素的值改为 2。

（5）越界问题：因为数组形参会被自动转换为指针，所以编译器会忽略在形参中为数组指定的长度，当传递实参时，编译器是不会检查数组的长度的。比如 void f(int a[11]){}; int b[3],c[22];，则 f(b); f(c);是正确的，在调用 f(b)时有可能会造成数组越界，因为传递进来的实参数组只有 3 个元素，而形参却误认为有 11 个元素。若在函数 f 中编写有超过 3 个元素的代码，则程序就可能造成数组越界，这时程序可能会崩溃。

（6）在编写接收数组的形参时，应使用一些编程技巧来解决数组实参的越界问题。常见的办法是定义第二个形参表示数组的大小，在调用函数时，使用数组名和数组的长度作为实参调用此函数。比如要形参接收 int a[4]这样的实参，则形参形式为 void f(int *p, int i);或 void f(int p[], int i);，因此应这样调用：f(a,4)。

3．多维数组形参

为方便讲解，以下假设下标中的字符 N、M、x1、x2 等都是合法的数组下标值。

（1）多维数组形参：函数的形参是多维数组时，只有第一维的长度是可以省略的，其他维是不能省略的。因为除第一维外的其他维的长度都是多维数组中元素的一部分，详见第 6 章。所以：

① void f(int (*a)[x2]x[3]...[xn]); void g(int a[][x2][x3]...[xn]); void h(int a[x1][x2][x3]...[xn]); 这 3 个函数的形参是等价的，其形参类型都是 int (*a)[x2][x3]...[xn]，即是“指向 *N*–1 维的指针”，每一维都具有相应大小的元素，且存储的元素是 int 类型。

② 在函数的形参中为多维数组指定第一维的大小是没有意义的。比如 void h(int a[11][12][13]); 中的数组形参的长度 11 是没有意义的。

③ 可以改变多维数组形参指向的地址，因为多维数组作为形参，会将其自动转换为指针，所以可以改变指针的地址，这与数组名是不同的，数组名的地址是不能改变的。

④ 除第一维外的其他维是不能省略的。比如 void f(int a[][][4]); void f(int a[3][][4]); void f(int a[2][3][]);等省略除第一维外的其他维的行为是错误的。

（2）注意：int (*)[N]; int [][N]只有在函数头和函数原型中才是相同的，在其他地方并不是相同的。同理，int (*)[x2][x3]...[xn]与 int [x1][x2]...[xn]也只在函数头和函数原型中才是相同的，在其他地方并不是相同的。

（3）与一维数组相同，int (*)[N]; int [][N]; int [M][N]在函数形参中是等价的，对于拥有这 3

种形参的函数声明是相同的，而不是函数重载。

（4）与一维数组相同，若没有改变数组形参所指向的地址，则对数组形参的操作将与指针相同，它会间接改变传递进来的数组的值。

（5）若多维数组形参转换之后是一个一级指针，则应遵守一级指针的规则——不能将 const 限定的数组实参传递给非 const 限定的数组形参，但是可以将非 const 限定的数组实参传递给 const 限定的数组形参。比如 void f(int i[][5]){...} const int a[4][5]={0};，则 f(a); 是错误的，因为形参 i 被转换为 int (*i)[5]类型，它是一个指向非 const 限定的一级指针。

（6）若数组形参转换之后不是一级指针，则不遵守一级指针的规则——数组形参的元素类型是指针时，转换之后就不是一个一级指针。比如 void f(int *a[]);，则形参转换之后是一个二级指针，即 int *a[]与 int **a 等同。

（7）越界问题：与一维数组相同，多维数组形参都会被自动转换为指针，因此编译器会忽略在形参中为多维数组指定的第一维的长度，当传递实参时，编译器不会检查多维数组的第一维的长度。在编写接收数组的形参时，应使用一些编程技巧来解决数组实参的越界问题，最常用的方法就是使用第二个形参来指定多维数组第一维的大小。比如 void f(int a[][3][4], int i);，使用第二个形参 i 来指定多维数组形参第一维的大小。

（8）当函数的数组形参需要依赖实参数组的大小时，可以使用引用数组形参来传递数组，这时实参数组的大小就必须与形参数组的大小相同。比如 void f(int (&a)[10]){}，则这时传递给函数 f 的实参数组的大小就必须与形参数组的大小相同；再比如 int b[10]; int c[11];，则 f(b); 正确，但 f(c);是错误的。注意：int (&a)[10];中的小括号不能省略。

（9）传递给多维数组的实参，除第一维的大小可以与形参不同之外，其余维的大小必须与形参的大小相同。比如 void f(int a[3][5]){}; int a[4][5]={0}; int b[3[6]={0};，则 f(a);正确，f(b); 错误。

4．接收 C 风格字符串的形参

（1）当字符串作为参数传递时，实际上传递的是字符串的第一个字符的地址，因此用于接收字符串的形参的类型只要是 char 类型的指针即可，所以形参可以是 char *或 char[]类型。比如 void f(char *i); void f(char i[]);，这两种类型的形参都能接收 C 风格的字符串，其中 char []会被自动转换为 char*，这与数组形参是相同的。

（2）函数无法返回字符串，当函数需要返回字符串时，应返回指向字符串的首地址的指针。因此，返回 C 风格字符串的函数应具有形如 char * f(...){...}这样的形式。

示例 7.3：一维数组和多维数组形参

```
#include <iostream>
using namespace std;
void f1(int a[11])  //对于一维数组形参，在函数的形参中为数组指定大小是没有意义的，这里的形参与
```

```
                    //void f(int a[]);是相同的，都会被转换为int *a
    {int b=1;   a=&b;}  //可以改变数组形参指向的地址，这是与数组名不同的地方，这时的形参a并不是数
                        //组名，而是一个指针；这条规则同样适用于多维数组形参。注意，这条规则只适用
                        //于数组是形参时，当数组不是形参时，数组名与指针并不完全等同
void f3(int p[3]){p[0]=1;   p[1]=2;}
void f3(int *p);
//void f3(int p[]){}          //错误，f3被重定义，f(int[])与f3(int[3])、f3(int*)是等价的
//void f4(int a[3][][5]);     //错误，除第一维之外的其他维不能省略
//void f4(int a[3][4][]);     //错误，同上
//void f4(int a[3][][]);      //错误，同上
void f5(int a[][4][5]);       //形参int a[][4][5]会被自动转换为int(*a)[4][5]
void f5(int a[33][4][5]);     //形参int a[33][4][5]会被自动转换为int(*a)[4][5]
void f5(int (*a)[4][5]){    a[0][0][1]=1; a[0][0][2]=2; a[0][1][0]=6; }
    void f10(int *a[3][4])      //形参被转换为int *(*a)[4]
            {cout<<"F10"<<endl;}
int main(){int a=1;
    int s[13]={0}; int s1[13][6][7]={0}; int s2[13][4][5]={0};  int s3[3][4][5]={0};
    f1(s);          //一维数组形参对形参数组指定大小是没有意义的，实参s有13个元素，形参有11个元素
    f3(s);          //函数对数组形参的操作会间接改变实参数组的值
    cout<<s[0]<<s[1]<<endl; //输出1,2
    //f5(s1);       //错误，传递给多维数组的实参，除第一维的大小可以不同之外，其余维的大小必须与形参
                    //的大小相同
    f5(s2);         //正确，原因同上
    int *s4[4]={&a,s,0,0};int *p[6][4]={0};int *(*p1)[4]=&s4;
    f10(p); f10(&s4); f10(p1);  } //对f10的调用都是正确的
```

7.2.3 函数指针

函数指针即指向函数的指针。

（1）声明函数指针的形式为：返回类型（*标识符）(形参列表);。比如 int (*p)(int i, int j);，声明一个带有两个 int 形参，且返回类型为 int 的函数指针 p；再比如 void (*p)();，声明一个无形参且无返回值的函数指针 p。注意：小括号不能省略。

（2）函数类型：由函数的返回类型及形参的个数及类型确定。注意，形参名不是函数类型的一部分，比如 void f(int a, int b);，函数类型是由返回类型 void、形参个数和形参类型描述的，形参名 a、b 并不是函数类型的一部分，因此相同类型的函数类型可以拥有不相同的形参名，如 void f(int a, int b)与 void h(int i, int j);是同一类型的函数类型。

（3）函数名单独使用时表示指向该函数的指针，或者表示该函数的入口地址。函数名与数组名一样，也是一个常量指针或常量地址，不能对其进行赋值操作。

（4）函数指针的初始化。

① 同数组一样，函数名就是函数的地址，所以在给函数指针赋值时只需给出函数名即可。函数名后不能有小括号，也不能有形参，如果函数名后带有小括号，则是在调用函数或者是错

误的语法。比如 void f(); void (*pf)();，则初始化函数指针的方法是 pf=f;。注意：p=f();不是正确的初始化方法，在使用函数的地址时只需使用函数名即可，这里是在调用函数 f。

② 与数组名不同的是，在函数名前使用&运算符，同样表示的是函数的地址，在较老的编译器上可能是非法的，原因是函数名本身就是一个地址，在地址前再取地址就是错误的。但 ANSI C 支持这种格式。比如 void f(); void (*pf)();，则 pf=f; 和 pf=&f;都是正确的。

③ 赋给函数指针的函数必须和函数指针有相同的返回类型、相同的形参列表（即形参的类型和个数要相同），但并不要求形参的名称要一致。比如 void f(int a); void (*pf)(int i); void (*pf1)(int);，则 pf =f; pf1 =f; 都是正确的；int (*pf2)(int)=f; 是错误的，返回类型不同；void (*pf3)(long)=f; 是错误的，形参类型不一致；void (*pf4)(int a, int b)=f; 是错误的，形参个数不一致。

（5）函数指针的使用：可以把函数指针当作函数名一样来调用函数，也可以把函数指针解引用后来调用函数。前一种是新式的调用方法，后一种是旧式的调用方法，这两种调用方法在逻辑上是矛盾的，但 C++对这两种格式都支持，这源自于 C++的两种解释：一是由于函数名是指向函数的指针，因此指向函数的指针的行为应与函数名相似，所以函数指针应作为函数名来使用；二是因为函数指针是指针，而对指针解引用后才是所指的函数，所以应使用解引用后的函数指针来调用函数。比如 void g(){} void (*pf)()=g;，则 pf()与(*pf)(); 都是在调用函数 f 且都是正确的，其中(*pf)()是旧式的调用方法，pf()是新式的调用方法。

（6）不能对函数指针进行算术运算，即不能进行+、-、++、--等运算，因此也不能对函数指针进行数组下标运算。

（7）函数类型形参：在声明函数类型形参时，与声明普通形参相同，可以省略函数的名称（即标识符）。比如 int f(void (*)(int ,int));和 int f(void (int, int));，加上标识符之后的形式为 int f(void (*p)(int i, int j));和 int f(void g(int i, int j));，它们都表示函数 f 拥有一个无返回值且带两个 int 类型形参的函数作为形参。

（8）当函数的形参为函数类型时，将被自动转换为相应的指向返回某类型的函数指针。

（9）对于与函数类型形参所对应的实参，在传递参数时，实参将被转换为指向相应函数类型的指针。

（10）因为 C++支持把函数指针当作函数名来使用，同时又支持把解引用后的函数指针当作函数名来使用，因此就延伸出了以下一些使用方法相同的情形。

示例：void f(){}void (*p)=f;

① 以下调用的情形是相同的，且都是正确的。

f()、(&f)()、(*f)()、(**f)()、(***f)()、(***f)()、…、p()、(*p)()、(**p)()、(***p)()、(****p)()、…

② 注意：对函数指针取址之后再调用函数的这种调用方法是错误的，比如(&p)(); 错误。

③ 注意：对函数名取址两次及以上是错误的，比如&(&f)(); 错误，因为&f 的结果是右值，

而&要求操作数是左值。

④ 下面对函数指针赋值时的情形都是正确的。

p=*f;、p=**f;、p=***f;、p=***f;

⑤ 注意：在声明函数指针时，使用多个解引用符号时，并不与以上情形相同。比如 void (**p2)();，则 p2=f; 是错误的，因为 p2 是一个二级指针，这个指针指向的又是一个指针，第二个指针指向的是一个无返回值、无形参的函数，所以 p2 不是一个指向函数的指针，而是一个指向指针的指针。

7.2.4 默认参数与可变形参

1. 可变形参

（1）可变形参就是指函数的形参个数可以改变。

（2）可变形参使用“...”（即省略符）表示（注意：省略符只能出现在尾部）。

（3）可变形参有 3 种形式，分别如下：

返回类型 函数名(形参列表, ...);，比如 void f(int i, ...);。

返回类型 函数名(形参列表...);，比如 void f(int i...);，此处在“...”前没有逗号。

返回类型 函数名(...);，比如 void f(...);。

（4）比如 void f(int i, ...);，表示调用 f 的实参至少应有一个，且第一个实参应是 int 类型或与其相容的类型，其他实参的类型不清楚。因此，调用 f(2,3,4); f(2,3); f(2); f(2, 3.3,4.4);都是正确的。

（5）对于 C++，只能将简单的数据类型传递给含有省略符形参的函数，大多数类类型对象都是不能正确进行传递的。

（6）当调用函数时，如果遇到与省略符对应的实参，则暂停类型检查机制，并告知编译器，可以有 0 或多个实参，但实参的类型不清楚。

2. 默认参数（默认实参）

（1）默认参数：即函数在参数表中使用初始化语法为形参指定默认值，有默认值的形参被称为默认实参或默认参数。比如 void f (int x=0, int y=90);，其中形参 x 和 y 都是默认参数。

（2）调用包含默认实参的函数，若调用时提供了实参，则实参会覆盖形参的默认值，否则使用形参的默认值。比如 void f(intx=10, int y=20);，则调用 f (1,2)表示将 1 传给 x，将 2 传给 y，这里实参 1 和 2 分别覆盖默认实参的值 10 和 20；调用 f (11)表示将值 11 赋给 x，y 使用默认值 20；调用 f()表示参数 x 和 y 都使用默认值。

（3）调用含有默认实参的函数，若调用时使用默认实参，则默认实参只能替换调用函数尾部的实参。比如 f(int x, int y=10, int z=12);，则 f (10,,15)就是错误的，这里试图让第二个参数使

用默认实参，而第一个和第三个参数使用调用函数的实参。

（4）接收默认值的所有形参都必须出现在不接收默认值的形参的右侧，或者说一旦开始定义接收默认值的形参，后面就不能指定不带默认值的形参了。比如 f (int x ,int y=10);是正确的，但是 f(int x=10, int y);就是错误的。

（5）对于同一个形参，其默认值只能指定一次，即使对两个函数的声明是相同的，也都只能指定一次。注意：函数形参的名字对形参并不产生影响。比如 void f(int a=1); void f(int c=2); 错误，void f(int a, int b=1); void f(int a, int=1); 错误，第二个函数 f 的第二个形参即使没有形参名，也是错误的。

（6）若函数有多个形参，则可以在后继的函数声明中指定前面声明没有指定的默认实参，但是同一形参仍然只能指定一次默认实参，而且接收默认值的所有形参都必须出现在不接收默认值的形参的右侧。比如 void f(int i,int a, int b, int c=1);，则后继声明 void f(int i,int a, int b=1, int c);是正确的； void f(int i,int a, int b=1, int c=1); 错误，同一形参只能指定一次默认实参； void f(int i=1,int a, int b; int c);错误，因为接收默认值的形参 i 出现在不接收默认值的形参 a 的左侧了。

（7）若在函数声明的文件中没有指定默认实参，而在函数定义的文件中指定了默认实参，则该默认实参只能在定义该函数的文件中使用。比如在 a.h 中有声明 void f(int a);，在 b.c 文件中有#include "a.h"void f(int a=1){...}语句，则默认实参的值只能在 b.c 文件中使用。

（8）默认实参可以是任意适当类型的表达式，若默认实参是表达式，且默认值被用作实参，则该表达式在调用函数时会被求解；若默认值没有被用作实参，则在调用函数时不会执行该表达式。比如 int g(){cout<<"A"<<endl; return 5;} void f(int a, int b=g()){}，则 f(2); 会计算默认实参的表达式，因此会调用函数 g；但是 f(2,3); 则不会调用函数 g，因为函数 f 没有使用默认实参。

（9）默认参数并没有改变形参的个数。比如 void f(int a, int b=1);，虽然可以使用一个实参来调用，但这里的形参仍然是两个，而不是一个。再比如 void f(int i, int j =1){} void f(int i){}，这是对函数 f 的重载，而不是对函数 f 的重定义，因为第一个和第二个函数定义中的形参个数并不相同。

7.2.5　内联函数、main 函数、extern"C"链接指示符

1. 内联函数（inline 关键字）

（1）内联函数：就是在声明函数或定义函数的开头加上 inline 关键字。比如 inline void f(); 就是一个内联函数 f。

（2）使用内联函数的目的：可以实现重复利用，不必为相同的操作重写相同的代码。但是直接的函数调用比直接计算某些操作（比如计算某个表达式）要慢很多，因为在函数调用时需要执行复制实参、保存机器的寄存器等多种操作。内联函数就是为解决这一问题而产生的，若

一个函数被定义为内联函数，则该函数会在程序的调用点上被“内联地”展开，即编译器会使用相应的函数代码替换函数调用，从而使代码执行得更快。但是内联函数却需要占用更多的内存，若程序在多个不同的地方调用同一个内联函数，则该程序就会包含多个该函数的拷贝。比如 inline int f(int a, int b){return a>b?a:b;}，则调用 int c=f(i,j); 在编译时会被展开为 int c=a>b?a:b;，因此在调用内联函数时的执行速度会比调用普通函数时的执行速度快得多，但会占用更多的内存。

（3）inline 指示符对于编译器只是一个建议，编译器可以不遵循该建议执行，因此某些编译器可能并不支持内联函数。如果函数中包含有循环、switch 或 goto 语句、递归函数，以及含有 static 的函数时，则会阻止编译器遵循建议。

（4）内联函数一般用来优化比较小的、只有几行、经常被调用的函数，因为对于较大的函数，执行函数代码的时间相对于节省的函数调用的时间来讲，只是很小的一部分，所以使用内联函数没有多少优势且较大的函数不一定在调用点上适合展开；对于较小的函数，虽然节省了函数调用的时间，但由于现在计算机的运算速度相当快，所节省的时间的绝对值并不是很大，除非该函数被经常调用。

（5）内联函数并不是在所有的调用点上都适合展开，比如某些递归函数、一些很大的函数（如超过几千行的函数），都不大可能在调用点上完全展开。

（6）与非内联函数不同的是，内联函数必须在调用该函数的每个文本文件中都进行定义（即内联函数可以定义多次），以便能在调用点上展开该函数；并且在不同的文件中其内联函数的定义必须相同，否则其行为就是未定义的，编译器最终会选择哪个函数是不确定的，当然程序不一定会出错。比如在 aa.cpp 和 bb.cpp 文件中都对内联函数 f 进行了定义，且两个定义不相同，则程序会有未定义的行为。注意：编译器不一定支持内联函数。

（7）建议把内联函数的定义放在头文件中，在每个需要使用该内联函数的文件中包含该头文件即可，这样就能保证对每个内联函数的定义只有一个且是相同的，而且避免了在每个需要使用内联函数的文件中复制相同代码的麻烦。

（8）内联函数与宏的区别：内联函数在调用时使用的仍然是按值传递方式，而宏在使用时则是简单的替换。比如 inline int f(int b){return b*b;} int a=1; f(a++);，调用函数 f 后 a 的值增加为 2，虽然在调用 f(a++)时，程序被展开为 b*b;，但函数仍是按值进行传递的，并不是简单的替换，因此实参 a 只被使用了一次。而宏则不同，比如#define F(X) X*X，则 int a=1; F(a++); 之后 a 的值为 3，因为在执行 F(a++);语句时被替换为 a++*a++; ，程序执行后 a 被自加两次，所以 a 的值变为 3。

2. main 函数

（1）main 函数在 C++中只能有唯一的一个，它是程序与操作系统交互的接口。在 C++中一

个完整的程序是从 main 函数开始执行的，同时 main 函数执行完毕，整个程序也就执行完毕了。若没有 main 函数程序将无法执行，比如 void f(){...}int main(){... return 0; }程序中有两个函数，在 C++中是从 main 函数开始执行的，而不会因为函数 f 写在 main 函数的前面而先执行它，当执行到 main 函数的 return 0 语句时程序结束执行。对于函数 f，在程序中有可能被执行，也有可能不被执行，若在 main 函数中直接或间接调用了函数 f，则会转去执行函数 f；若没有调用函数 f，则它不会被执行。

（2）在 C++标准中，必须要求 main 函数返回 int 类型值，也就是 main 函数必须这样 int main(...){...}，但很多编译器没有遵循这一规定，在程序中也可以写成 void main(...){...}的形式。为了在某些编译器上运行正确，最好写成 int main(...){...}的形式。

（3）关于 main 函数中的返回值，C++规定可以在 main 函数中不使用 return 语句，这时默认返回值为 0。比如 int main(){...}将返回值 0，它与 int main(){....return 0;}等同，但这一规定只适用于 main 函数，不适用于其他函数。

（4）使用 main 函数处理命令行选项：一般使用的 main 函数的形参表都是空的，但实际上主函数 main 中定义有两个形参，即 int main(int argc, char *argv[]);，第一个形参 argc 表示接收的命令行选项的 C 风格字符串的个数，第二个形参 argv 是一个 C 风格字符串数组，用于接收命令行选项中的所有字符串。比如有命令行选项 xxx -a -h yyy，则第一个参数表示的是命令行中 C 风格字符串的个数，这里共有 4 个字符串，因此 argc 的值为 4；而 argv 会依次保存这 4 个字符串为 argv[0]="xxx"、argv[1]="-a"、argv[0]="-b"、argv[0]="yyy"，其中 argv 的第一个字符串通常是调用程序的名字。

3. extern "C"链接指示符

（1）extern 的功能之一是告诉编译器，某函数是使用其他程序设计语言编写的（比如 C、FORTRAN 等语言）。

（2）extern 链接指示符的形式如下。

- extern 后跟一个字符串常量及一个普通函数声明，字符串指明了声明的函数所使用的编写语言。比如 extern "C" void f(int);，表示在程序中遇到的函数 f 是使用 C 语言编写的。
- extern 后跟一个字符串常量及包括多个函数声明的复合语句，字符串指明了声明的函数所使用的编写语言，比如 extern "C"{void f(int); void g(float);}。注意：在使用 extern 链接指示符的花括号内声明的函数名对外是可见的，这里的花括号只是一种分隔符，并不表示花括号内的成员具有局部作用域。

（3）若链接指示符的复合语句中包含#include 时，则在头文件中声明的函数都被假定是用链接指示符所指示的语言编写的。比如 extern "C"{#include <cmath>} 表示头文件 cmath 中声明

的函数都是使用 C 语言编写的。

（4）链接指示符不能出现在函数或类定义的内部。

（5）也可以使用 extern 来指定其他语言，比如 extern"Ada"表示声明的函数是用 Ada 语言编写的，extern"FORTRAN"表示声明的函数是用 FORTRAN 语言编写的。因为 extern 支持的其他语言因编译器的不同而不同，因此应查看编译器的相关说明，以获得更多的信息。

7.3 函数重载

注意：在下文中所指的形参表都不包括形参名；形参表不相同，不是仅仅指形参名不相同。形参名不是形参类型的一部分，因此形参名并不能修改函数的形参表。比如 void f(int a)、void f(int b)和 void f(int)这 3 个函数的形参表是相同的，形参名不影响函数的形参表。

1. 基本规则

（1）在相同作用域中的两个函数（此前提在不同的名称空间中时比较重要），若函数名相同、形参表不同，则称为函数重载，重载的每个函数都应有一个单独的定义。比如在相同作用域中的函数 void f(int i)、void f(int i, int j); 和 void f(float a); 就是函数重载。

（2）函数重载时的条件："在相同作用域中"这一条件，当在不同作用域中时比较重要（比如名称空间作用域、类作用域等）。为便于说明，以下内容若不特别指出，声明的函数都是指在相同作用域中。

（3）调用重载的函数时，会执行形参与实参相匹配的那个函数。比如 void f(int i){} void f(float i){} void f(int i, int j){}，则 f(2)将调用 f(int i)函数，f(3.3)将调用 f(float i)函数，f(2, 3) 将调用 f(int i, int j)函数。

（4）函数重载时的形参表不同指的是每个重载函数的形参类型、形参数量或形参顺序必须不同，函数的返回类型或形参名不同不能实现函数重载。示例如下：

- void f(int i){} int f(int i){}，错误，仅返回类型不同，不是函数重载。
- void f(int a){} void f(int b){}，重定义错误，仅形参名不同，不是函数重载。
- void f(int a){} int f(int b){}，重定义错误，返回类型和形参名都不同，也不是函数重载。
- void f(int a){} void f(int a, int b){}，正确，形参的数目不同，是函数重载。
- void f(int a){} void f(float a){}，正确，形参的类型不同，是函数重载。
- void f(int a, float b){} void f(float a, int b){}，正确，形参顺序不同，是函数重载。
- void f(int a, int b){} void f(int b, int a){}，重定义错误，形参名不同，不是函数重载。这里并不是形参的顺序不同，两个函数都是 void f(int ,int)类型的。
- void f(long a){} void f(int a){}，正确，形参的类型不同，是函数重载。

- void f(int p){} void f(int *p){}，正确，形参的类型不同，是函数重载。
- void f(int p){} void f(int &p){}，正确，形参的类型不同，是函数重载。

（5）若两个函数的形参表只是默认参数不同，则不是函数重载。比如 void f(int i, int j){} void f(int i, int j=2){}，错误，函数 f 被重定义。

（6）若两个函数的形参表的区别只是一个使用了 typedef 定义的替代名，而另一个却使用了与 typedef 相对应的类型，则这不是函数重载。比如 typedef int T;，则 void f(T i){} void f(int i){} 错误，函数 f 被重定义，这不是函数重载。

2. const 与重载

（1）若声明的两个函数的形参为非引用类型的（指针除外），且只是有无 const 限定的区别，则编译器会将 const 形参视为非 const 类型的，因此它们的形参是相同的，这不是函数重载。比如 void f(const int i){} void f(int i){}，错误，函数 f 被重定义，这不是函数重载。若声明的两个函数的形参为指针类型的，且 const 修饰的是指针本身，则遵守此规则。比如 void f(int *const p){} void (int *p){}，则产生重定义错误，这不是函数重载。

（2）若声明的两个函数的形参为引用类型的（指针除外），且只是有无 const 限定的区别，则它们的形参是不相同的，它们是两个重载的函数。比如 void f(const int &p){} void f(int &p){}，正确，这是函数重载。

（3）若声明的两个函数的形参为指针类型的，且一个指向 const 限定的对象，一个指向非 const 限定的对象，那么它们是两个重载的函数。比如 void f(const int *p){} void f(int *p){}，正确，这是函数重载。

7.4　函数匹配（或函数重载解析）

7.4.1　函数匹配的过程

本节主要讲解单形参时的函数匹配规则。

（1）函数匹配：调用函数时，究竟调用哪个重载函数。

（2）二义性：这是一种错误的行为，指的是在调用重载的函数时，出现编译器不知道调用哪个函数的情况。因此，在编程时，应避免出现调用函数时的二义性行为。

（3）函数匹配的过程如下。

① 确定候选函数。候选函数指的是与被调函数同名的函数，并且在调用时，这些函数的声明是可见的。也就是说，只要与被调函数同名的函数都是候选函数。

② 选择可行函数。这一步相对比较简单。可行函数是从候选函数中选出的一个或多个函数，

这些函数都可以被调用函数正确调用。

可行函数必须满足两个条件。

- 函数的形参的个数与调用函数的实参的个数必须相同，若有默认参数时，形参的个数会多于实参的个数，但多出的形参必须都有默认参数。
- 每个实参的类型必须与对应的形参的类型相匹配，或者实参的类型能被隐式转换为对应的形参的类型。

③ 确定是否有最佳匹配的可行函数，若有则调用它，否则出错。在确定最佳匹配的函数时，编译器会将实参的类型转换划分等级。此步最复杂，具体见后文。

7.4.2 候选函数的确定方法

（1）若局部声明一个函数，则该函数会屏蔽外层作用域中的同名函数。也就是说，在调用这个函数时，只有局部声明的函数是可见的，外层作用域中的同名函数是不可见的。因此，候选函数将只有局部声明的这个函数，而不会包括外层作用域中的同名函数。

（2）示例如下：

```
void f(int){} void f(float){} void f(long){}
void g(){ void f(int); void f(long); f(2.2);}
```

调用 f(2.2);的候选函数只有局部声明的 f(int)和 f(long) 两个函数，而 f(float)将被屏蔽掉而不在候选函数之列，所以调用 f(2.2)将出现二义性错误。

```
void f(int ){} void f(float){} void g(){void f(double); f(2.2);}
```

调用 f(2.2)将发生错误，因为这时的候选函数只有局部声明的 void f(double)，但在外层作用域中却找不到有关 f(double)函数的定义。其外围的 void f(int)和 void f(float)两个函数被局部声明的 void f(double)屏蔽（或隐藏）掉了，并不在候选函数之列。

（3）其他情形的名称可见性规则分散在后续的章节中介绍，本章节重点讲解确定最佳匹配函数的方法。

7.4.3 确定最佳匹配函数的方法

（1）为便于讲解，本节中介绍的函数只写出函数原型，而不给出函数的完整定义，读者在测试时应把函数原型定义完整。函数的形参名在本节中也会被省略。

（2）注意：这里所讲的最佳匹配，并不一定是完全匹配或最完美的匹配。调用重载函数时，会从众多候选函数中调用最佳匹配的函数，该函数只是相对于当前其他的候选函数而言的，其匹配比其他候选函数更好。比如 void f(int); void f(float)，则 f('a')将调用 void f(int)，对于字符型 f(int)不是匹配得最好的，但在 f 的两个重载函数中，f(int)匹配得更好，因此 f(int)是最佳匹配。

1. 最佳匹配的等级划分

（1）最佳匹配由最佳到最差的次序为：完全匹配→通过提升转换实现的匹配→经过标准转换实现的匹配→经过类类型转换实现的匹配。详见图 7.1。

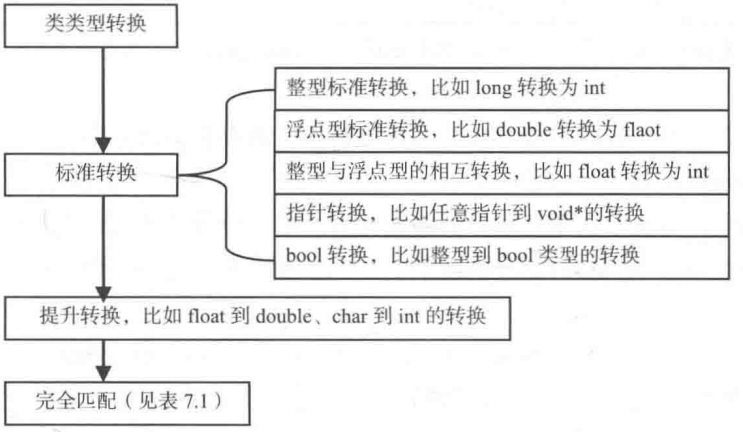


图 7.1　最佳匹配的等级划分(由差到佳)

表 7.1　完全匹配所允许的所有转换

转　换	说　明
从左值到右值的转换	① 若函数参数是按值传递的，且实参是左值时，则会发生从左值到右值的转换 ② 若形参为 const type &类型时，则可能会（也可能不会）发生从左值到右值的转换
数组到指针的转换	type[] → type*
函数到指针的转换	type（参数列表）→ type(*)（参数列表）
非引用间的转换	type → const type const type → type type* → const type* const type* → type*　//错误，指向非 const 的指针不能指向 const 对象
引用间的转换	type → type& type → const type& type& → type type& → const type& const type → const type& const type& → type const type& → const type type → type&　const type&　优先选择此形参 type* → type*　const type*　优先选择此形参 type1 → const type2 &　这种转换之后的转换序列不一定是完全匹配的

续表

转　换	说　明
引用间的转换	const type → type&　　//错误，非 const 对象的引用不能引用 const 对象 const type& → type&　　//错误，同上

说明：以上情形都是指从实参到形参的转换，比如 type → const type，则实参类型为 type，形参类型为 const type。

（2）注意：在转换过程中，提升转换和标准转换一般只会使用其中一种，不会既出现提升转换又出现标准转换。

（3）完全匹配：即实参类型与形参类型相同的匹配，但普通函数优于模板函数及其特化版本（关于模板请参阅第 15 章）。完全匹配的实参与形参并不一定是完全一致的，而是允许进行一些转换。

（4）提升转换：比如 char、short 提升为 int；float 提升为 double；bool 提升为 int 等。

（5）经过标准转换实现的匹配：所有的标准转换都具有相同等级，不存在谁比谁更接近的优势。有 5 种标准转换。

① 整型标准转换，即整型或枚举类型到其他整型的转换，不包括提升转换。比如 long 到 int 的转换，注意 short 到 int 是提升转换。

② 浮点类型转换，即浮点类型与浮点类型之间的转换（提升转换除外）。比如 double 到 float 的转换，注意 float 到 double 是提升转换。

③ 浮点类型到整型或整型到浮点类型的转换，比如 char 到 float 的转换等。

④ 指针转换，比如数值 0 到任何指针类型的转换、任何指针类型到 void*的转换。

⑤ bool 类型转换，比如整型、浮点类型到 bool 类型的转换。

（6）注意：任何指针类型间的转换都优于指针类型到 bool 类型的转换。比如 void f(void *){} void f(bool){} int a=1; float *p=0;，则 f(p); f(&a);会调用 f(void*);。

（7）标准转换示例：void f(long){} void f(float){}。

对于 f(long)和 f(float)，调用 f(2); f(2.2);都是标准转换，不存在谁比谁更有优势，因此出现二义性，其中数值 2 是 int 类型，int 到 long 是标准转换，同样 int 到 float 也是标准转换；数值 2.2 是 double 类型，double 到 float 与 double 到 long 都是标准转换。

2．理解转换序列

（1）转换序列：实参到形参的匹配往往不是经过一次转换就能完成的，一般都需要经过多次转换，这样的转换称为转换序列。

（2）转换序列一般由以下顺序构成。

① 左值转换（包括左值到右值的转换、数组到指针的转换和函数到指针的转换）。

② 提升转换或标准转换。

③ 限定修饰转换。

（3）转换序列的步骤如下。

① 0 次或一次左值转换。

② 0 次或一次提升转换，或者 0 次或一次标准转换。一般提升转换和标准转换只会有一种被执行。

③ 0 次或一次限定修饰转换。

（4）除了上面所讲的标准转换序列，还有用户自定义的转换序列。

（5）转换序列的等级：指构成转换序列时最坏转换的等级。比如一次转换经过左值转换、标准转换后才与形参匹配，那么这个转换序列的等级就是标准转换。

（6）示例：void f(const int a){} float b=2.2; int c=3;。

关于 f(b)调用，因为 b 的 float 类型与形参 a 的 const int 类型不匹配，所以首先进行左值到右值的转换（完全匹配），即提取 b 的值创建临时变量，然后进行一次标准转换，即 flaot 到 int 的转换，最后是 const 限定转换（完全匹配）。整个转换分为 3 步，即左值转换（左值到右值的转换）—标准转换—const 限定转换，每一步转换的等级为完全匹配—标准转换—完全匹配，因此转换序列的等级为标准转换。

7.4.4　完全匹配详解

以下讲解的都是在完全匹配情形下的等级划分规则。

下面以实参到形参的转换为例进行说明。

1. 完全匹配所允许的无关紧要的转换

（1）完全匹配时实参与形参并不一定是完全一致的，其允许存在一些不必要（或无关紧要）的转换。也就是说，虽然进行了一些转换，但仍然是完全匹配的。

（2）完全匹配所允许的无关紧要的转换如下。

① 左值到右值的转换。

② 数组到指针的转换。比如 void f(int *p){} int a[3]={0};，则调用 f(a)与 f(int *)是完全匹配的。

③ 函数到指针的转换。比如 void f(void (*p)()){} void h(){};，则调用 f(h);与 f((*p)());是完全匹配的。

④ 限定修饰转换，即 const 与非 const、volatile 与非 volatile 之间的转换，具体见表 7.2。

表 7.2 完全匹配所允许的限定修饰转换

允许的限定修饰转换	示　例
type 到 const type 的转换	void f(const int); int a=1;，则 f(a)是完全匹配的
const type 到 type 的转换	void f(int); const int b=1;，则 f(b)是完全匹配的
type*到 const type*的转换	void f(const int *); int a=1;，则 f(&a)是完全匹配的
type&到 const type&的转换	void f(const int&); int a=1; int &s=a;，则 f(s)是完全匹配的
const type*到 type*的转换（错误）	不允许指向非 const 的指针指向 const 对象
const type&到 type&的转换（错误）	非 const 对象的引用不能引用 const 对象

说明：以上情形都是指实参到形参的转换。此表只包括了 const 的转换，未包括引用间的转换，比如 type 到 type&的转换等。

2. 左值到右值的转换

（1）左值到右值的转换：表示从左值表达式所表示的对象中抽取值的动作。比如 int a=1, b=2; int c=a+b;，在执行加法之前，会将 a 和 b 所指内存单元中的值抽取出来，这里就存在左值到右值的转换。

（2）若函数参数是按值传递的，且实参是左值时，都会进行左值到右值的转换，因为按值传递时会创建一个临时变量，然后将实参的值传递给该临时变量，并使用临时变量初始化形参，所以会提取实参的值。比如 void f(int a); int b=1;，则 f(b)中的实参 b 就会进行左值到右值的转换。

（3）若函数的形参是引用时，则不会进行左值到右值的转换，因为引用表示的就是一个左值，所以对于引用形参应接收一个左值的实参，而不能接收右值的实参。注意：若形参引用的是一个常量（即 const 引用），则实参可以是右值。比如 void f(int &a); int b=1;，则 f(b)中的实参 b 不会进行左值到右值的转换，而 f(3)是错误的，因为 3 是右值；若形参是 const 引用，比如 void f(const int &a);，则 f(3)的调用是正确的。

（4）若形参为 const 引用类型的对象，则可能会/不会进行左值到右值的转换，因为若函数调用时的实参不是左值或与引用类型形参的类型不匹配时，则 C++会创建正确的临时对象，并将实参的值传递给该临时对象，让形参引用该对象，所以可能会提取实参的值。因此，若实参是与 const 引用形参的类型匹配的左值时，就不会进行左值到右值的转换；反之，若不匹配，则会进行左值到右值的转换。注意：若实参为右值，则不存在左值到右值的转换。

示例如下：

```
void f(const int &a); float b=2.2; int c=3;
```

① f(b)调用，因为 b 的类型 float 与形参 a 的类型 const int 不匹配，所以首先进行左值到右值的转换（即提取 b 的值创建临时变量），然后进行一次标准转换，即从 float 到 int 的转换，最后是 const 限定转换。

② f(c)调用，因为 c 的类型与形参 a 的类型相匹配，所以不会进行左值到右值的转换。

③ f(2.2)调用，因为 2.2 是右值，所以不存在左值到右值的转换，首先使用值 2.2 创建一个临时对象，然后进行一次标准转换，即从 float 到 int 的转换，最后是 const 限定转换。

（5）因为从左值到右值这种转换仍是完全匹配的，因此对于 void f(int &a); void f(int a); int b=1;，调用 f(b)会出现二义性错误。对于 f(int);，调用 f(b)存在从左值到右值的转换，这是完全匹配的；对于 f(int &);，调用 f(b)不存在转换，也是完全匹配的，这两种完全匹配没有等级差别，因此会出现二义性错误。

3．const 限定转换对指针或引用的影响

（1）若实参是非 const 类型的对象，则从实参到 const 引用的转换和从实参到非 const 引用的转换都是完全匹配的。比如 void f(int &);void f(const int &);int b=1; f(b);，则实参 b 与 f(int &)和 f(const int &)都是完全匹配的，但究竟与哪一个匹配得更好，下面就来介绍怎样从完全匹配中选出最佳匹配。

（2）引用或指针的 const 限定转换原则：对于两个形参都是引用或指针的情况，若两个转换序列的前两个步骤是相同的，且只有一个转换序列在尾部有限定转换，则另一个无限定转换的转换序列更好。注意：只要有一个重载函数的形参不是指针或引用，此规则就不适用，同时该规则只适用于把 const 或 volatile 限定词使用到引用或指针指向的类型上的转换。也就是说，若是非 const 引用或指针指向的是非 const 对象时，此规则不适用。

示例如下：

```
void f(int *a){} void f(const int *a){} int b=2; int *p=&b; f(p);
```

f(p)调用 f(int *)，对于形参 int*和 const int *，实参 p 都是首先进行从左值到右值的转换，即提取 p 中的值，然后进行 0 次提升转换或标准转换，这两个转换序列的前面都是相同的，因为两个函数的形参都是指针，所以最后没有 const 限定的转换序列优于有 const 限定的转换序列，因此 f(p)会调用 f(int *)。

```
void f(int& a){}  void f(const int& a){}  int b=1;  f(b);
```

f(b)调用 f(int&)，对于两个形参都是引用的情况，若转换序列的前两个步骤相同，则没有 const 的引用形参优于有 const 的引用形参，因此调用 f(int&)。实参 b 对于 int &和 const int &来讲，都是完全匹配的，不需要转换，因此没有 const 限定的转换序列优于有 const 限定的转换序列。

```
void f(int& a){} void f(const float& a){} char c='a'; f(c); f(3);
```

f(c)、f(3)都会调用 f(const float&)，因为 C++禁止在非 const 引用之间进行类型转换，也禁止非 const 引用引用临时对象，非 const 引用也不能引用右值。因此，c 和 3 对于 int&是不正确的类型匹配，详见 7.2.1 节“指针形参和引用形参”。

```
void f(long a){}void f(const float& a){} char c='a'; f(c);
```

f(c)调用出现二义性错误，分析步骤如下。

① 对于形参 long a，首先进行的是从左值到右值的转换，然后进行了一次标准转换，即从 char 到 long 的转换，转换序列的等级为标准转换。

② 对于形参 const float &a，首先进行的是从左值到右值的转换（即提取 c 的值创建临时变量），然后也进行了一次标准转换，即从 char 到 float 的转换，最后是 const 限定转换。由于两个函数的形参并不都是引用，因此 const 限定转换是完全匹配中无关紧要的转换，转换序列的等级仍为标准转换。所以对于形参 long a 和 const float &a 都匹配得一样好，f(c)调用出现二义性错误。

```
void f(int a){} void f(const int &a){} int b=1; short s=2; f(b); f(2); f(s);
```

① f(b)、f(2)和 f(s)调用均出现二义性错误。

② 对于 f(b)调用，实参 b 到形参 a 只进行了从左值到右值的转换，这是完全匹配的；对于形参 const int &a，这里只有 const 限定转换，这也是完全匹配的，且两个函数的形参中有一个不是引用，因此两个转换序列都是完全匹配且都匹配得一样好，所以这两个调用出现了二义性错误。

③ 对于 f(2)调用，因为都不存在转换，所以都是完全匹配的且匹配得一样好。

④ 对于 f(s)调用，实参 s 到形参 int a 首先进行的是从左值到右值的转换，然后是一次提升转换，转换序列的等级为提升转换。实参 s 到形参 const int &a 首先进行的是从左值到右值的转换（因为 s 的类型与 int 类型不相同，因此需要提取形参 s 的值创建临时变量），然后是一次提升转换，最后是 const 限定转换。由于两个形参并不都是引用，因此 const 限定转换是完全匹配中无关紧要的转换，转换序列的等级仍为提升转换，所以调用出现二义性错误。

```
void f(int *a){} void f(const int *a){} int b=2; const int *p=&b; f(p);
```

f(p)会调用 f(const int *a)，这里 f(p)与 f(int *a)是不匹配的，即 f(int *a)不是 f(p)的可行函数，因为指向非 const 的指针不能指向 const 对象。

```
void f(int *p){} void f(int *const p){}
```

错误，这是对函数 f 的重定义。

4. 其他重载解析规则

（1）普通函数优先于模板函数及特化的模板函数。

（2）当完全匹配的函数都是模板函数时，则更具体的模板函数优先。更具体指的是编译器推断使用哪种类型时进行的转换更少。

（3）若函数有多个形参，则最佳匹配的函数必须满足其所有参数的匹配程度都不能低于其他候选函数所有参数的匹配程度。

5. 函数重载解析综合示例

```
#1 void f(int a){}          #2 void f(float a, float b=3){}  #3 void f(char a){}
#4 void f(const char* a){}    #5 void f(const char & a){} char c='b';
```

调用 f(c)发生二义性错误，分析如下。

① 确定候选函数。上面函数都与调用函数的名字相同，因此都是候选函数。

② 确定可行函数。对于 f(c)调用#4 不可行，因为 char 不能被隐式转换为指针类型，因此#4 不是可行函数，其余函数都是可行函数。

③ 按类型转换等级确定最佳匹配的函数。

- #1 调用优先于#2。因为 char 到 int 是整型提升转换，而 char 到 float 是标准转换，提升转换优先于标准转换，所以#1 调用优先于#2。
- #3、#5 调用都优先于#1 和#2。因为#3、#5 都是完全匹配的，且两个函数只有一个是引用形参，所以#3 和#5 都是最佳匹配。因此，调用 f(c)出现二义性错误。

```
#1 void f(int a){}  #2 void f(const int a){}    #3 void f(int &a){}
#4 void f(const int &a){} int b=1;
```

① 如果只有#1 和#2，则将发生对函数 f 的重复定义错误。

② 如果只有 1#和#3，则 f(2)将调用#1，而 f(b)调用会出错。因为形参 int &a 是左值，而实参值 2 是右值，不能传递给引用形参 int &a，所以对于 f(2)调用，f(int &a)不是可行函数，但对于 f(b)，b 到 int 只进行了一次从左值到右值的转换，这是完全匹配的，b 到 int &则是完全匹配中 type 到 type&的无关紧要的转换（其实 type 到 type&不需要转换），因此也是完全匹配的。从左值到右值的转换与 type 到 type&的转换没有等级差别，所以调用出错。

③ 如果只有#1 和#4，则 f(3)和 f(b)都将出错。对于 f(3)，因为两个调用都没有进行转换，所以都是完全匹配的，它们不存在谁更优于谁的情况，所以出错。对于 f(b)，实参 b 对形参 int a 的转换只有从左值到右值的转换，对于 const int &a 则只有 const 限定转换，但其中有一个形参不是引用，因此 const 限定转换是无关紧要的转换。它们的转换都是完全匹配的且具有相同的等级，所以没有最佳匹配的函数，结果出现二义性错误。

④ 如果只有#2 和#3，则 f(2)将调用#2，而 f(b)将出错。对于 f(2)调用#2，因为 2 是右值，形参 int &a 要求的是左值，所以 f(int &a)不是可行函数。对于形参 b 到 const int a，首先进行从左值到右值的转换，然后是 0 次标准转换或提升转换，最后就是 const 限定转换。对于 int &，则属于完全匹配中 type 到 type&的无关紧要的转换（或者说没有转换），因为从左值到右值的转换与 type 到 type&的转换是无等级差别的，且其中有一个形参是非引用，因此 const 限定转换是完全匹配中无关紧要的转换。两个转换匹配得一样好，所以出现二义性错误。

⑤ 如果只有#2 和#4，则 f(3)和 f(b)都将出错。对于 f(3)，不存在转换，所以两个函数都是最佳匹配，最后出错。但对于 f(b)，type 到 const type&的转换只有 const 限定转换，而 type 到 const type 的转换也只有 const 限定转换，因此没有等级差别，所以调用出错。

⑥ 如果只有#3 和#4，则 f(3)将调用#4，因为对于实参 3 来讲 f(int &a)不是可行函数；f(b)将调用#3，因为两个形参都是引用，若转换序列的前两个步骤相同，则没有 const 的引用形参优于有 const 的引用形参。实参 b 对于 int &和 const int &来讲，转换序列的前两个步骤相同，即都有 0 次左值转换、0 次提升转换或标准转换，只有一个转换序列在尾部有限定转换，因此没有 const 限定的转换序列优于有 const 限定的转换序列。

7.5 作用域、存储持续期、链接性和存储类区分符

（1）注意：下文中所提到的标识符、名字、名称在本质上没有区别，指的都是相同的概念。

（2）变量或函数的称呼：具体称呼视其作用域、存储持续期、链接性而定。比如在局部作用域中声明的变量被称为局部变量；在全局作用域中声明的变量被称为全局变量；在局部作用域中使用 auto 声明的变量被称为局部自动变量；在局部作用域中使用 static 声明的变量被称为局部静态变量；在全局使用 static 声明的变量被称为全局静态变量；在全局使用 static 声明的函数被称为内部函数；在全局使用 extern 声明的函数被称为全局函数或外部函数。

（3）链接性分为内部链接、外部链接和无链接三种。链接性为外部的标识符可以在多个文件中使用或共享；链接性为内部的标识符只能在定义它的文件中使用。无链接的标识符只能在某个函数或局部作用域中使用。

7.5.1 作用域

1. 基本概念

（1）文件及程序：一个程序可以写在多个文件中，即一个程序可以由多个文件组成。因此，本节所讲的程序的范围都比文件的范围大。

（2）作用域：其作用是为了解决名称冲突的问题，因此作用域描述的是标识符（或名称）的可见性问题，即描述知道该标识符的区域。每一个函数都具有不同的作用域，比如在 void f(){int a=1;}中声明的标识符 a 在该函数中是可见的，即在该函数中程序知道有这个名字；而在另一个函数中就不知道有这个名字（或者说该名字在这个函数中不存在），因此在这个函数中就不能使用函数 f 中声明的名字。比如 void g(){cout<<a<<endl;}错误，因为函数 g 中的标识符 a 是未声明的；再比如函数 void f(){int a;}中的标识符 a 与函数 void g(){int a;}中的同名标识符 a 互不影响，因为它们没有在同一个作用域中，所以可以同时出现且不会出现名称冲突。

2. 作用域的分类

（1）作用域可以被细分为全局作用域、局部作用域、函数作用域、函数原型作用域、文件

作用域、名称空间作用域、类作用域。

① 全局作用域：程序最外层的名称空间作用域被称为全局作用域或全局名称空间作用域，全局作用域是名称空间作用域中的一个特例。全局作用域中的标识符可以全局使用，即可以在整个程序中使用，它在整个程序中都是可见的。全局作用域一般指的是在所有函数、类、名称空间之外声明的标识符，该标识符可能表示变量、函数或类。比如 int a=1; void f(){} void main(){}，其中标识符 a 和 f 都是在全局作用域范围内声明的。

② 局部作用域：指的是存在于代码块中的程序文本部分。代码块是由一对花括号括起来的一系列语句，函数体是一个代码块，因此在函数体中声明的标识符具有局部作用域。在函数体中还可以嵌入其他代码块，但不能在全局作用域范围内嵌入代码块（在全局作用域范围内应使用名称空间定义作用域）。在定义函数时声明的形参具有局部作用域，它在该函数的代码块内是可见的，因此不能在代码块内定义与形参名相同的标识符。示例：

- void f(){int a=1; {int b=2}}，标识符 a 和 b 都在局部作用域内，其中标识符 a 位于函数体内，标识符 b 所在的作用域嵌套于 a 所在的作用域内，因此标识符 a 在 b 所在的作用域中是可见的，但标识符 b 在 a 所在的作用域中却是不可见的。
- {int a=1; } void main{}，错误，不能在全局作用域范围内直接嵌入代码块。
- void f(int a){int a=1;}，错误，不能在代码块内定义与形参名相同的标识符。

③ 函数原型作用域：在函数原型（声明）中使用的名称具有函数原型作用域，该名称只在包含参数列表的括号内是可见的，因此函数原型中的形参名称是不重要的，其可以在其他任何地方被重新使用，而不会出现名称冲突。比如 void f(int a); void f(int i){a=1;}错误，名称 a 具有函数原型作用域，在后面的函数 f 的定义中是不可见的。

④ 函数作用域：只有用作 goto 语句的标号才具有函数作用域。因为不推荐使用 goto 语句，故本文不对函数作用域进行介绍。

⑤ 文件作用域：在当前文件中可见的标识符具有文件作用域。比如使用 static 声明的函数或全局变量就具有文件作用域。

⑥ 名称空间作用域：使用名称空间定义来声明的标识符具有名称空间作用域，每个名称空间都是一个不同的作用域，在名称空间中可以包含对象、函数、类和模板的声明及定义，以及嵌套在其中的名称空间。关于名称空间详见第 10 章。

⑦ 类作用域：在定义类时都会引入一个独立的类作用域。

（2）作用域的范围由大到小依次是：全局作用域、文件作用域、局部作用域（名称空间作用域、类作用域）。也就是说，具有全局作用域的标识符在整个程序中都是可见的，具有文件作用域的标识符在其本文件中的局部作用域、名称空间作用域、类作用域中都是可见的。

（3）注意：函数的作用域不能是局部的，即不能在局部作用域中定义函数，因为在局部作用域中定义的函数只对它自己可见，而不能被其他函数调用，这样的函数将无法运行。虽然不

能在局部作用域中定义函数，但是可以在局部作用域中声明函数，用于调用在此作用域之后才定义的函数。注意：名称空间作用域并不是局部作用域，因此可以在名称空间作用域中定义函数。

（4）在相同作用域中定义相同名称的变量会出现名称冲突（函数重载是个例外），但在不同的作用域中定义相同名称的变量不会出现名称冲突，因为在不同的作用域中是看不到其他作用域中定义的名称的。注意：相同的声明可以进行多次。

（5）通过声明引入的名称，从声明点开始一直到声明它的作用域（包括其中的嵌套域）结束都可见。

（6）函数形参、for、if、while 头中定义的名称都只在其所属的代码块中是可见的，因此具有局部作用域。其名称的可见性，从声明此名称处开始，一直到相应的代码块结束，因此在该代码块之外，其名称是不可见的。

示例 7.4：作用域

```
#include <iostream>
using namespace std;
int a=5;                    //名称 a 具有全局作用域，名称 a 在整个程序中都是可见的
void h(int c);              //名称 c 具有函数原型作用域，其可见性仅限于函数 h 的形参列表
void h(){
   //c=1;                   //错误，函数 h 原型中的形参 c 在此处是不可见的
   int c=1;                 //正确，函数 h 原型中的形参 c 在此处是不可见的，因此可以在函数内部定义相同名
                            //称的标识符,而不会出现名称冲突
   //void h4{}              //错误，不能在局部作用域内定义函数
   void h3();  h3(); }      //正确，可以在局部作用域内声明函数，在调用 h3 之前应先声明
//{int f=3;}                //错误，在全局作用域范围内不能使用语句块
void h3(){int m=1;}
void h5(){int m=2; }        //在 h3 和 h5 中定义的名称 m 互不影响，因为它们是在不同的作用域中定义的
void h1(int x){             //形参 x 具有局部作用域，其可见性从声明之处起一直到函数体结束
   //int x;                 //错误，在相同作用域中定义了相同的名称
   //int x[2];              //错误，同上
    extern int y;           //使用 extern 声明一个变量，extern 表明此变量在其他地方被定义
    //int y;                //错误，这是对 y 的重复定义错误
    void h6();  void h6(); } //正确，指向同一个实体的名称可以被声明多次
void main(){
    int j=1;                    //名称 j 具有局部作用域
    {int k=2;                   //注意：大括号在此处开始，这里产生了一个新的作用域
    cout<<j<<endl; }            //正确，名称 j 在嵌套的作用域中是可见的
    //cout<<k<<endl;            //错误，名称 k 是在嵌套的作用域中声明的，其可见范围只限于声明它所在的
                                //作用域之内，在其外围的作用域中是不可见的
}
```

3. 名称解析

（1）名称解析：把名称与声明相关联的过程，即给出该名称意义的过程。名称解析在不同

的环境中有不同的解析方法。

（2）局部作用域内的名称解析过程：首先查找使用此名称的作用域，若找到一个声明，则表明该名称被解析，此时名称查找结束，不再向外围作用域进行查找。若未找到，则向上查找包含该作用域的作用域，直到找到一个声明或已经找完整个全局域。若最后没有找到该名称的声明，则这个名称被标记为错误。

（3）因为名称解析时的顺序是由内向外的，在外围作用域中声明的名称会被嵌套作用域中声明的同名名称所隐藏，所以在嵌套作用域中，可以声明或定义与外围作用域中相同的名称。比如 int a=1; void g(){} void f(){float a=2; int g=1;}正确，在函数 f 中声明的 a 和 g 两个名称分别隐藏了全局作用域中的名称 a 和函数名称 g，此处不会产生名称冲突。

（4）可以使用作用域解析运算符“::”引用被隐藏的全局作用域的名称。注意：只能使用“::”访问被隐藏的全局变量，而不能引用外围作用域中的对象。比如 int a=1; void f(){int a=2; cout<<::a;}，则 f()将输出 1，此处的“::a”指的是全局作用域中声明的名称 a，而非局部作用域中的 a=2。

（5）重载与作用域：函数重载的前提是“重载的函数应在相同的作用域中（带类类型形参的函数除外）”。可见，函数重载与名称解析（查找）是完全不同的概念。因此，若在局部作用域中声明一个函数，则该函数会隐藏掉外围作用域中与该函数形参表不相同的其他重载函数，它们不会形成函数重载。注意：形参表不相同，指的是形参的类型、个数或顺序不相同。

示例如下：

```
void f(int i){cout<<"A";} void f(float i){cout<<"B";}
int main(){void f(int i); f(3.3); }
```

其中 f(3.3)调用的是 f(int i)函数，在 main 函数中声明的 void f(int i)屏蔽了外层作用域中与该函数形参表不同的重载函数 f(float i)，这时将把 3.3 转换为 int 类型。

```
void f(int i){cout<<"A"<<endl;} void f(float i){cout<<"B"<<endl;}
void main(){void f(double i); f(3.3);}
```

这时调用 f(3.3)将发生错误，因为在外层作用域中没有找到有关 f(double)函数的定义。在 main 函数中声明的 void f(double i)屏蔽了外层作用域中的 f(int i)和 f(float i)两个重载函数。

7.5.2　存储持续期、链接性与作用域

本节将对 auto、register、static 和 extern 四个存储类区分符进行讨论。

1. 基本概念

（1）存储持续期（也叫生命期）：指的是对象在程序执行过程中所存在的时间，或者说某一对象占有所分配的内存空间的期限。注意：生命期描述的是已经分配了内存空间的对象。

（2）存储持续期分为静态存储持续期、自动存储持续期和动态存储持续期。

（3）使用 new 操作符分配的内存具有动态存储持续期，请参阅第 6 章。

（4）链接性：用于描述标识符（名称）在代码块之间、源代码文件之间、编译目标模块之间、库之间的共享情况。

（5）链接性的真实意义：现代的编译器一般都是按文件进行编译的，因此在编译时各文件中所定义的全局变量互不可见，即在编译阶段，全局变量的作用域仅限于本文件内部。也就是说，在编译阶段，在不同文件中定义相同名称的全局变量不会出错；但在链接阶段，就要将各文件的内容“合为一体”，这时若在不同文件中定义了相同名称的全局变量，就会出现重定义错误。由此可见，在链接阶段扩大了全局变量的可见范围，将其扩大到了整个程序。

（6）链接性分为内部链接、外部链接和无链接三种。

① 链接性为外部的标识符可以在所有文件中使用或共享，即作用域为全局的。

② 链接性为内部的标识符只能在定义它的文件中使用或共享，即作用域为本文件。

③ 无链接的标识符只能在某个函数或局部作用域中使用或共享。

（7）标识符的链接性需要根据标识符所引用的实体（如变量、函数等）、标识符的位置（全局的还是局部）和存储类区分符（如 static、extern）三方面来确定。

（8）作用域、生命期、链接性是三个不同的概念，其中作用域描述的是标识符的可见范围；生命期描述的是对象占有所分配的内存空间的期限；链接性描述的是标识符的共享情况。因此具有不同作用域的标识符，可以具有不同的存储持续性，同时还具有不同的链接性。

2. 自动存储持续期、自动变量及 auto 关键字

（1）自动存储持续期与自动变量：具有自动存储持续期的变量被称为自动变量。自动变量一般是在函数（包括函数参数）中声明的，自动变量从程序开始执行其所属的函数或代码块时被创建，在退出函数或代码块时被销毁，这时与自动变量相关联的存储区也被释放。这一过程是自动进行的，因此自动存储持续期在程序执行期间自动分配和释放内存单元。

（2）以下情况的局部变量具有自动存储持续期。

① 函数的形参。

② 在声明时未使用存储类区分符的局部变量。在声明局部变量时，若未使用任何存储类区分符，则被默认为使用了存储类区分符 auto，因此该对象具有自动存储持续期。因为局部变量默认具有自动存储持续期，所以 auto 关键字一般很少看到。

③ 使用存储类区分符 auto 或 register 修饰的局部变量。

（3）存储类区分符 auto 用于将变量声明为自动的，即使用 auto 声明的变量具有自动存储持续期。

（4）auto 只能用于声明局部变量，不能把全局变量声明为自动的。

（5）auto 不能用于声明函数，因为函数名不是局部的。

（6）自动变量具有局部作用域，但不是所有具有局部作用域的变量（即局部变量）都具有自动存储持续期，比如局部静态变量就不具有自动存储持续期。有些书上把局部变量和自动变量等同对待，本文中局部变量并不等同于自动变量，而只有具有自动存储持续期的变量才是自动变量。

（7）注意：VC++ 2010 对 auto 关键字实现了新的功能，即在默认情况下 auto 关键字不能和类型区分符联合使用，比如 auto int a;在 VC++ 2010 中是错误的。可以通过“编译器选项”来改变 auto 的默认行为，步骤为：选择“项目”→“属性”，在对话框左侧选择“配置属性”→“C/C++”→“命令行”，然后在右侧的“其他选项”中输入“Zc:auto-”。若输入“Zc:auto”，则表示使用 VC++ 2010 的默认设置。

3．自动变量的特点

（1）自动变量具有局部作用域、自动存储持续期、无链接性。

（2）自动变量每次进入或退出相应的作用域时，存储单元都会重新进行分配和释放，因此无法保证其值与上次退出相应的作用域时的值相同。注意：静态局部变量会保持上次退出相应的作用域时的值。

（3）自动变量的生命期在相应的代码块结束时结束，这时它的值被抛弃，相应的存储区被释放，这也是自动变量的地址不能用作函数的返回值的原因，因为函数一旦结束，该地址就指向了一个无效的存储区。

（4）未初始化的自动变量将有一个随机值，使用未初始化的自动变量是不可预测的。

（5）每进入一次相应的作用域，其存储单元都会被重新分配，因此自动变量都会被重新初始化一次。注意：静态局部变量只会初始化一次。

（6）自动变量的数目会随函数的开始和结束而发生相应的变化，因为进入函数时会创建新的变量，退出函数时会销毁变量。因此程序在运行时，必须对自动变量进行管理，常用的方法就是留出一块内存，并将其当作栈来处理，以管理自动变量的增减。

4．寄存器变量及 register 关键字

（1）寄存器变量：指使用 register 声明的变量，以请求编译器尽量通过 CPU 的寄存器访问该变量，从而提高程序运行速度。

（2）编译器不一定会满足使用寄存器的要求。

（3）寄存器变量是自动变量的另一种形式，因此寄存器变量具有局部作用域、自动存储持续期、无链接性。

（4）寄存器变量与自动变量的不同是，寄存器变量是通过 CPU 的寄存器来使用的，而不是自动变量使用的堆栈，因此对寄存器变量的访问速度比自动变量快。

（5）寄存器变量没有内存地址，因为变量被存储在寄存器中，而寄存器是不可寻址的。因

此不能将取址操作符（&）用于寄存器变量，或者说在程序中不得使用寄存器变量的地址。即使实际上编译器没有使用寄存器来存储该变量，也不能对寄存器变量进行取址操作。比如 register int x=9;，则 int *p=&x; 是错误的。

（6）register 不能用于修饰全局对象，比如不能使用 register 来声明函数、全局变量。

（7）register 可以用于声明函数形参中的标识符。

（8）需要频繁使用的变量可以被声明为寄存器变量。

（9）一般计算机的寄存器数量是有限的，因此不要声明过多的寄存器变量。

（10）一般寄存器的长度只有 2~4 个字节，因此应将整型（如 char、int 等）变量声明为寄存器变量。

（11）注意：VC++ 2010 是把寄存器变量当作普通自动变量进行处理的，而且还会为其分配内存地址，因此在 VC++ 2010 中使用寄存器变量的地址不会出错，但要注意移植性的问题。

示例 7.5：自动变量、寄存器变量

```
#include <iostream>
using namespace std;
int a1=1;                          //全局变量的存储持续期不是自动的
//auto int a2;                     //错误，auto 关键字不能用于声明全局变量
//register int a3;                 //错误，register 关键字不能用于声明全局变量
//register void f1();              //错误，register 关键字不能用于声明函数
//auto void f2();                  //错误，auto 关键字不能用于声明函数
int * f(int a4)                    //形参 a4 默认是自动变量，具有自动存储持续期、局部作用域
    {                              //形参 a4 的内存在函数体开始处被自动分配
    a4=4;
        {                          //函数内部的代码块开始处
        int a5;                    //变量 a5 位于函数 f 中的代码块内，且未使用存储类区分符，因此变量 a5 默
                                   //认是自动变量，具有自动存储持续期。当程序执行到此步时为变量 a5 分配内
                                   //存单元，同时 a5 没有进行初始化，这时它将拥有一个随机值
        a5=5;
    auto int a6=6;                 //自动变量在每次进入相应的作用域时都会被初始化一次
    static int a7=7;               //a7 具有静态存储持续期。并不是所有的局部变量都具有自动存储期
    }                              //自动变量 a5 的作用域结束处，在此处 a5 的内存被释放，a5 的值也被抛弃
    //return &a4;                  //这是危险的，不一定出错。因为自动变量的生命期在函数结束时结束，它的值
                                   //被抛弃，相应的存储区也被释放，这时自动变量的地址就指向了一个无效的存储区
    return &a1;                    //a1 是全局变量
}                                  //函数 f 结束处，形参 a4 的内存空间将在这里被释放，其值也将被抛弃
void main(){
     register int a8=8;            //将 a8 声明为寄存器变量
    //int *p=&a8;//错误，不能使用寄存器变量的地址，因为寄存器是不可寻址的。但在 VC++ 2010 中是正确的
}
```

5. 静态存储持续期的概念及其创建

（1）静态存储持续期的特点：具有静态存储持续期的对象，在整个程序的运行过程中都是

存在的，其存储单元只会分配一次——在每次进入相应的作用域时存储单元不会重新进行分配，在退出相应的作用域时也不会被释放。具有静态存储持续期的对象的存储单元是在编译时进行静态分配的，而不是在运行时进行动态分配的。

（2）创建具有静态存储持续期的变量（static 用法一）：使用 static 存储类区分符修饰的变量就具有静态存储持续期，这种变量称为静态变量。

（3）使用 static 关键字定义的变量和在全局作用域中定义的对象（即全局变量、函数）都具有静态存储持续期。

6. 链接性的创建及 extern、static 关键字的使用

（1）链接性的创建。

① 外部链接性的创建（extern 用法一）：具有外部链接性的对象应在全局作用域中使用 extern 加以说明，并进行初始化或定义。注意：若没有初始化或定义，则表示这是 extern 引用声明（见后文）。比如 extern int a=1;，则 a 具有外部链接性；extern void f(){}，则 f 具有外部链接性；但 extern int a、extern void f();，则表示是使用 extern 引用声明。

② 内部链接性的创建（static 用法二）：具有内部链接性的对象应在全局作用域中使用 static 或 const 加以说明。

③ 无链接性的创建（static 用法三）：要创建无链接性的静态存储持续期变量，应在代码块内声明，并使用 static 加以说明。

（2）各种变量的默认链接性。

① 全局变量，默认具有外部链接性。

② 自动存储期的对象，只能是无链接性的。

③ 使用 const 声明的全局变量默认其链接性为内部的。

④ 要让 const 定义的变量的链接性是外部的，则必须使用 extern 关键字加以说明，并对其进行初始化；若不对其进行初始化，则表示这是 extern 引用声明。比如 extern const int a=1;，表明变量 a 的链接性是外部的，此语句不能出现在局部作用域内。

7. 静态存储持续期的特点及其与自动存储持续期的比较

下面以静态变量为例进行说明。

（1）静态变量的作用域范围可以是全文件或全程序，也可以是局部作用域；而自动变量只能是局部作用域。

（2）静态变量只会分配一次存储单元，所以静态变量在程序执行期间只会初始化一次；而自动变量在每次进入相应的作用域时都会初始化。

（3）静态变量在整个程序执行期间是一直存在的，因此其生命期在函数结束时并不结束，这时它的值也不会被抛弃，相应的存储区也未被释放；而自动变量在函数结束时就结束其生命

期，其值被抛弃，相应的存储区也被释放了。

（4）静态变量在退出并重新进入相同的作用域后，其值会保持不变；而自动变量则是每次进入相应的作用域时都会重新分配存储单元，每次退出时也会释放存储单元，无法保持自动变量的值不变。比如 void f(){static int b=1; b++; cout<<b<<endl;} void main(){f(); f(); }，第一次调用 f 输出 2，第二次调用输出 3，静态变量只被初始化了一次（即只分配了一次存储单元），且保持值不变。而对于自动变量，每次调用都会输出 2（即无法保持值不变），且每次调用都会进行初始化（即分配存储单元）。读者可将 static 关键字去掉之后自行验证。

（5）若静态变量在声明时没有进行显式初始化，则编译器会将其初始化为 0。在默认情况下，静态的数组和结构会把每个元素或成员都初始化为 0；而自动变量不会对其进行初始化，其得到的将是一个随机值。

（6）静态变量在整个程序执行期间是一直存在的，因此静态变量的数目不会变，编译器会为静态变量分配固定的内存块来存储所有的静态变量；而自动变量的数目会随着进入相应的作用域和退出而发生相应的变化。

（7）静态变量可以具有 3 种链接性，即外部链接性、内部链接性和无链接性；而自动变量只具有无链接性。

（8）具有静态存储持续期的对象在整个程序中一直存在，其内存单元是独占的，无法实现共享，因此若滥用该对象不利于内存单元的有效利用。

（9）具有静态存储持续期的对象的特点是：只会初始化一次；若未初始化，则默认初始化为 0；每次进入和退出相应的作用域时，其值会保持不变；退出相应的作用域时，静态变量并不会销毁，除非整个程序结束。

8. 全局变量、全局静态变量、局部静态变量

（1）全局变量的定义一：应在全局作用域中直接定义，或者在定义时使用带初始化的 extern 语句。

（2）外部（全局）变量的定义二：链接性是外部的变量被称为外部变量，作用域为整个程序。因为外部变量对整个程序是可见的，因此也被称为全局变量。

（3）全局静态变量：在全局作用域内使用 static 声明的变量。这里的 static 关键字限制了全局静态变量的链接性是内部的。

（4）局部静态变量：在局部作用域内使用 static 存储类区分符修饰的变量。因为局部静态变量的作用域为局部可见，因此其具有无链接性。

（5）全局变量、全局静态变量和局部静态变量的区别。

① 全局变量：具有静态存储持续期、外部链接性，其作用域为整个程序。全局变量在各文件中只能定义一次，而且可以使用 extern 语句引用其他文件中定义的全局变量。

② 全局静态变量：具有静态存储持续期、内部链接性，其作用域为本文件可见。全局静态变量不可被其他文件共享，即使使用 extern 也不能引用其他文件定义的全局静态变量。在不同文件中可以使用相同名称的全局静态变量。

③ 局部静态变量：具有静态存储持续期、局部作用域、无链接性。

（6）全局变量、全局静态变量和局部静态变量都具有静态存储持续期，因此它们都具有静态存储持续期的特点（见上文）。

（7）若在文件中定义了一个与另一个文件中的全局变量同名的全局静态变量，则该变量会隐藏掉同名的全局变量。这与局部变量会隐藏掉其外围作用域中的同名变量相同，通过作用域的名字解析过程很容易理解。

（8）注意：在类作用域和自定义的名称空间作用域中的函数外定义的变量不是全局变量，这种变量具有类作用域或名称空间作用域。本文不对类作用域和名称空间作用域进行讲解。

9. 函数与链接性

（1）所有函数都有静态存储持续期，所以它们都具有静态存储持续期的特点，即在整个程序执行期间一直存在。

（2）在默认情况下，函数的链接性是外部的，在声明或定义函数时默认使用 extern 存储类区分符。当然，使用 extern 的函数声明还有另一个作用，就是表示此函数是在其他地方被定义的。不过，这也是可选项，比如 extern void f();与 void f();具有相同的效果。

（3）若定义函数时使用的存储类区分符为 static，则此函数具有内部链接性，其作用域仅限于本文件。这种函数也被称为内部函数。因为内部函数名只在本文件中是可见的，在其他文件中是不可见的，因此就可以在不同的文件中定义相同名字的内部函数。

（4）若定义函数时使用的存储类区分符是 extern，或者没有使用存储类区分符，则该函数具有外部链接性，其作用域是整个程序。这种函数也被称为全局函数或外部函数。因为外部函数的函数名在整个程序中是可见的，所以在同一个程序内的不同文件中不能定义相同名字的外部函数。

10. 存储类区分符的使用总结

（1）static 关键字有三种用法：一是将对象声明为具有静态存储持续期；二是在全局作用域中将对象声明为具有内部链接性；三是在局部作用域中把对象声明为具有无链接性。

（2）extern 关键字有四种用法：一是将变量声明为非定义的；二是用作引用声明，表示该对象是在其他地方定义的；三是作为语言链接性使用（即 extern "C"）；四是将某一对象声明为具有外部链接性。

（3）局部变量默认具有自动存储持续期、无链接性。全局变量默认具有静态存储持续期、外部链接性。函数默认具有静态存储持续期、外部链接性。

（4）注意：声明变量时存储类区分符只能使用一个，即 auto、 register、 static、 extern、 typedef 之一。

11. extern 引用声明

（1）extern 引用声明（extern 用法二）：不带初始化的 extern 声明。比如 extern int a;就是一个引用声明。

（2）使用 extern 引用声明的原因。

虽然在链接阶段全局变量的作用域为整个程序，但并不是说在一个文件中定义的全局变量就直接能在另一个文件中使用，因为编译器在编译阶段并不知道某个变量可能会在其他文件中定义。为此，在需要使用其他文件中定义的全局变量时，使用 extern 引用声明告诉编译器，现在编译的文件中使用 extern 引用声明的变量可能是在其他文件中定义的。在某些情况下，也可以理解为使用 extern 引用声明将其他文件中定义的变量引入到该文件中。

（3）extern 引用声明的作用。

① 表示该变量在程序的其他地方或其他文件中有定义。

② 若要在一个文件中使用在其他文件中定义的全局变量，则应使用 extern 引用声明，将该变量引入到该文件中。

（4）使用 extern 引用声明的变量的作用域：若 extern 引用声明出现在全局作用域中，则该变量在全局均可见；若 extern 引用声明出现在局部作用域中，则该变量只在局部可见，但仍具有静态存储持续期。

（5）全局使用 extern 引用声明的变量，若在程序的其他地方没有定义，则此时只是对该变量的一个非定义声明，程序不会对其分配内存空间，在未定义之前该变量不能使用，应该先定义后使用（注意是定义而不是赋值）。定义时又分为以下两种情形。

① 若定义该变量时是在局部作用域中进行的，则该变量只在此局部作域中可见。若未对其进行初始化，则该变量将会得到一个随机值。

② 若定义该变量时是在全局作用域中进行的，则该变量在全局可见。若未对其进行初始化，则该变量将会得到一个默认值 0，且仍然具有静态存储持续期的性质。

（6）若在局部作用域内使用了 extern 引用声明，则不能在局部作用域内再对该变量进行定义，也不能在局部作用域内使用带初始化的 extern 语句。比如 void main(){extern int a; int a=1;}，其中 int a=1;是错误的。

示例 7.6：extern 引用声明及链接性

```
//文件 xx.cpp 中的内容
int a1=1;           //外部链接性
static int a2=2;    //内部链接性
int a3=3;           //外部链接性
```

```
void f(){}          //函数默认具有外部链接性
//主文件的内容
#include<iostream>
using namespace std;
extern int a1;      //使用 extern 引用声明，把名称 a1 引入该文件中，全局引入的名称具有全局作用域
//int a1=11;        //重定义错误，a1 与 extern 引入的名称 a1 冲突
extern int a2;      //无法引入名称 a2，因为文件 xx 中的名称 a2 具有内部链接性，此处是对 a2 的非定义声明
extern int b;       //使用 extern 引用声明的变量未在其他文件中定义，因此这是一个非定义声明
void f();           //引入其他文件中具有外部链接性的函数名称时可以不使用 extern 关键字
void g(){
     int b=1;       //在局部作用域中定义在全局使用 extern 进行非定义声明的变量，因此 b 具有局部作用域
     extern int a3;     //局部引入的名称 a3 具有局部作用域
     //extern int c=3; //错误，在局部作用域内不能使用带初始化的 extern 声明
     extern int c1;     //变量 c1 的非定义声明
     //int c1=1;        //错误，不能在局部作用域中既使用 extern 对变量进行非定义声明，又对其进行定义
     }
void g1(){              //验证静态存储持续期和自动存储持续期的特点
     static int s=1;   int i=4;
     cout<<"static s="<<s<<endl;s++;
     cout<<"i="<<i<<endl;i++;          }
     void main(){
     //a3=3;           //错误，文件 xx 中的名称 a3 未被引入到该作用域中（a3 被引入到了函数 g 内部）
     //b=2;            //错误，在此处 b 的全局非定义声明的定义不可见（b 定义于函数 g 内部）
     g1(); g1(); }     //连续两次调用 g1，可见静态变量 s 只会初始化一次，且在函数调用结束后会保持其值
```

7.5.3　将程序写在多个文件中

（1）一个程序一般会被写在多个文件中，C++的文件分为源文件和头文件，头文件一般包含一些类、函数、变量等的声明；源文件内容一般是对这些类、函数、变量的定义。当然，在头文件中也可以定义函数或变量，但不推荐这样做。

（2）在 C++中头文件的后缀一般为.h，而源文件的后缀一般为.cpp。

（3）C++的每个文件都可以单独进行编译，然后使用链接程序把它们链接成可执行程序。现在的编译器一般既有编译又有链接功能。

（4）在编译阶段，标识符的可见性一般只作用于当前文件；而在链接阶段时，要将各文件的内容“合为一体”，这时就扩大了标识符的可见范围，标识符的可见性被扩大至整个程序。也就是说，在编译阶段，各个文件中可以出现相同名称的全局对象而不会引起冲突；而在链接阶段，各个文件中的全局对象则不能相同，否则就有可能引起名称冲突。当然，若是链接性为内部的标识符，在各文件中的定义相同或者指向不同的实体是不会出错的。

例如，文件 1 的内容为：int a=1; static int b=2; 文件 2 的内容为：int a=2; int b=3;。

在编译阶段，这两个文件的变量 a 不会产生名称冲突；但在链接阶段，也就是把两个文件链接成一个完整的程序时，变量 a 就会出现名称冲突。因为在链接阶段扩大了标识符的可见范

围，这时就会检测到变量 a 的重定义。而在文件 1 中变量 b 的链接性是内部的（即此变量只在本文件中可见），所以在编译和链接阶段都不会出现名称冲突。

（5）头文件的作用是提供函数或类等的声明。在一个大型的程序中，可能会把程序分别写在多个源文件中，而这些源文件中都包含有很多名称，在使用其他源文件中的名称时，应先对其进行声明，然后才能使用，这样就显得很烦琐。而头文件则将这些源文件中的名称声明集中写在一起，在需要使用其他文件中的名称时，只需将该头文件使用#include 包含进来即可，这时就相当于把所有名称都进行了一次声明，然后在该文件中就可以直接使用这些名称了，这样就减少了不少的工作量。

（6）要使用头文件中的内容，则应使用#include 将其包含进来；否则无法使用头文件，即编译器不会执行头文件中的内容。

（7）在#include 后面的文件名最好使用双引号引起来，而不要使用尖括号，比如#include "eee.h"。使用双引号和尖括号的区别是，使用尖括号时编译器将首先在标准头文件中查找；而使用双引号则首先在当前工作目录或源代码目录中查找，若未找到则在标准头文件中查找，这样查找速度更快。

（8）注意：源文件不能使用#include 语句进行包含。

（9）在头文件中最好不要包含有非 inline 函数（内联函数）或对象的定义，否则当多个源文件中都包含有相同的头文件时，就会出现重复定义错误。比如在头文件 aa.h 中有一个定义 int a=1;，在文件 1 和文件 2 中都分别包含了该头文件，这时运行程序就会出现重复定义错误。注意：编译时不会出错。

（10）头文件的作用主要是用于声明，因此在头文件中最好只包含对象的声明，而不应有定义。

（11）在头文件中一般应包含的内容有函数原型、使用#define 或 const 定义的符号常量、结构声明、类声明、模板声明、内联函数的定义。注意：全局 const 常量默认的链接性是内部的且常量一般不需要改变，因此各源文件中包含有 const 常量的头文件并不会出现名称冲突且也符合实际需要。内联函数的要求是，在每个源文件中都要进行一次相同的定义，因此推荐在头文件中包含内联函数的定义。

（12）不应在同一个文件中包含头文件多次，因为这样做容易出现错误——因为同一个头文件被包含多次，就意味着头文件中的对象被声明了多次，若头文件中有对某一对象的定义（比如头文件中定义了某个 const 常量），就会出现重复定义错误了。

（13）可以使用预处理器指令#ifndef 来解决包含头文件多次的问题。关于预处理器详见第 18 章。

第 8 章

类基础、类作用域及相关运算符专题

8.1 面向对象程序设计基本概念

1. 类、对象、抽象、实例基础

初学者必须理解这些概念。

（1）对象：指的是任何需要设计的东西，比如人、飞机、动物、三角形、学校、项目计划、一个整数等都可以是对象。

（2）对象都具有某些特性或行为，对象的特性也被称为对象的属性。

（3）对象的特性（属性）一般用于描述对象所拥有的特征，所以一般都是静态的。比如三角形对象，它的特性包括三角形的边长、每个角的角度、三角形的高等；再比如电视机对象的特性有颜色、重量、长、宽、高、屏幕大小等。

（4）对象的行为用于描述对象所拥有的功能或对对象的操作，所以对象的行为一般是动态的。比如三角形对象，它的行为包括计算三角形的周长、面积等；再比如电视机对象的行为有调整电视机的亮度、对比度、颜色及对电视机进行频道切换等。

（5）对象的特性（属性）在程序设计语言中一般使用变量来实现；对象的行为（或功能）在程序设计语言中一般使用函数来实现。

（6）类：是具有相同特性和行为的对象的一个模板（或者框架）。因此，类是对具有共同特性和行为的对象进行的一次归纳（或归类），从而形成一个模板或框架。比如等腰三角形、等边三角形等所有三角形对象，因为这些对象都具有三角形的共同特性（如边长）和行为（如计算面积），因此可以将这些对象归为“三角形”，三角形就成为一个类，可以称其为“三角形类”，从而“三角形类”就成为所有三角形的一个模板或框架；再比如美国人、中国人、法国人这些

对象都可以被归为“人”这一类，因此“人类”就成为这些对象的一个模板或框架。可见，一个类包括多个对象，单独说一个类时并不知道具体代表什么对象，但知道这些对象拥有什么特点，比如单独说“人类”，并不会知道指的是美国人、中国人还是日本人，但却知道“人类”拥有的特点。

（7）抽象：就是将对象的特性或行为进行归类的一个过程。比如将美国人、中国人、法国人这些对象归为“人类”的过程被称为抽象，也可以这样说，把美国人、中国人、法国人抽象为“人”这样一个类。同理，可以把等腰三角形、等边三角形抽象为“三角形类”。

（8）实例概念及类与对象的区别：对实例最简单的理解就是指“例子”，从对象和类的概念可以看到，对象是具体存在的，是实实在在指的某一事物；而类则只相当于具有共同特性和行为的对象的一个模板或框架，因此可以说对象是具体的，对象是类的一个实例或特例或例子。而产生类的过程是对具有共同特性和行为的对象的归类，这个归类的过程被称为抽象，因此也可以说类是对象的抽象。比如等腰三角形对象就是三角形类的一个实例，等腰三角形只是众多三角形类中的一个具体的三角形而已，它是所有三角形当中的一个实例或一个例子或一个特例。而三角形类是三角形对象的抽象，是三角形对象的一个模板。一个类拥有多个对象。

（9）类与类型：其实类就是一种类型，只是这种类型相对于简单的内置类型而言看起来更复杂而已，在程序设计语言中被称为“自定义类型”或“抽象数据类型”，而程序设计语言中的类型被称为“内置类型”（比如整型、浮点型等）。在 C++声明章节中已经讲过，类型决定了对象的内存大小、布局和取值范围，决定了可以对该对象进行什么样的操作（比如进行加法、减法运算等），决定了可以将什么数据赋给对象，而 C++中的类都具有这些功能。

2. C++类和对象的实现

（1）C++类的实现：C++使用关键字 class 来实现类。比如 class A{int a; int b,c ; void f();};，表示定义一个类 A，这个类有三个数据成员变量 a、b、c，还有一个成员函数 f()，若使用变量 a、b、c 表示三角形的三条边的长度，使用函数 f()实现计算三角形的面积，那么就可以说类 A 实现了三角形类。

（2）C++对象的实现：使用由 C++定义的类就可以创建对象了，其方法与创建一个变量相同，在使用 C++的类时，就和使用 C++的“内置类型”相同，只是 C++的类是“用户自定义类型”而已。比如 class A{...};，则实现对象的方法为 A ma;，其中 ma 是类 A 的一个对象；再比如 A mb, mc;，其中 mb、mc 是类 A 的对象。

（3）创建对象时需要为对象指定属性值：因为对象是具体指的某一事物，因此在创建对象时必须为对象的属性指定具体的值。比如 A ma;表示使用默认值来指定对象 ma 的各属性值，其实就是为类 A 中的所有变量指定初始值。当然，对象 ma 的属性值是其专有的，并不属于整个类，也不会影响到类 A 创建的其他对象的属性值。也可以使用非默认值来初始化对象的各属性

值，其具体方法详见后文。

（4）C++创建的对象是各自独立互不影响的。比如 A ma, mb;，其中对象 ma 和 mb 是互不影响的，虽然它们使用的都是相同框架类 A 中的变量，但创建对象后，该对象就拥有这些变量的值，且各自独立，改变对象 ma 的属性并不会影响到 mb。假如 ma 指的是等腰三角形，mb 指的是等边三角形，改变等腰三角形 ma 各边的长度，肯定不会因此而改变等边三角形 mb 各边的长度。

（5）在 C++中，类其实就是一种类型，只是这种类型比一般的内置类型（比如 int 类型）更复杂。而类的对象其实就是变量，只是该变量的类型是一个类，是“自定义类型”而已。

3. 设计类的方法简介

（1）设计类的关键要点是能从所有对象中找出这些对象的共同特性和行为，然后进行抽取。特性一般描述的是某个对象所需的数据（比如三角形的边长，汽车的长、宽、高、颜色等），行为一般描述的是对象所能提供的功能或操作（比如汽车的舒适性、易操作性等可能就要根据汽车的某些数据进行综合估算后才能确定，计算三角形的面积或周长）。

（2）使用 C++语言实现类时，一般使用变量来表示对象的特性（比如三角形的边长、人的高度等），使用函数来表示对象的行为（比如计算三角形的面积、周长等）。

（3）下面以所有的三角形对象为例进行介绍。

① 类：将这些对象所拥有的共同特性和行为使用一个类来表示，比如使用三角形类。

② 抽取共同的特性和行为：三角形类应包含所有三角形对象的基本共同特性和行为，比如包含的特性有边长、高，包含的行为有计算三角形的面积、周长等。

③ 对象和类的差别：三角形类的具体对象具有明确的边的长短和高的数据值，还包括具体的行为，类只是对象的一个框架或模板，类没有三角形边长的具体数据和高的具体值。这里三角形类是所有具体的三角形对象的一个整体模板或框架，而三角形对象则具体指明了是一个什么样的三角形（比如等边三角形、等腰三角形等）。这里可以看到三角形类相对于具体的三角形对象而言是抽象的、不具体的（因为使用类无法表示一个具体的三角形），而每个三角形对象都是三角形类的一个具体的例子（或实例），都能明确地确定或指明一个三角形。

④ 使用 C++语言实现类的大概过程如下。

- 创建一个三角形类，比如其名称为 A，其语法为 class A;。
- 使用变量表示所有三角形对象的特性。假设只指定三角形的三个边长，因此可以使用三个浮点型变量来表示三角形的三个边长，比如使用 float 类型的变量 a、b、c 分别表示三边的长度。
- 使用函数表示所有三角形对象的行为。假设使用函数 f 和 g 分别来计算三角形的面积和周长。

- 因此，三角形类 A 应包含三个浮点型变量 a、b、c 和两个函数 f 和 g，其完整形式为 class A{float a,b,c; float f(); float g();};。此处可以看到，类只是具有共同特性和行为的一个模板或框架，类无法明确地指定一个事物。
- 使用三角形类创建对象时，必须为对象指定具体的属性值，以确定是一个什么样的对象。比如 A ma;，创建三角形类 A 的一个三角形对象 ma，这里表示使用默认值初始化对象的三个属性（变量）a、b、c，以后可以修改这些属性的值，当然也可以在创建对象时指定初始值。

4. 继承与多态性

（1）继承：是在现有类的基础上进行的扩展。也就是说，继承是在之前的类的基础上，增加了对象的一些更具体的共同特性和行为，继承之后的类将拥有之前的类的特性和行为。比如可以把人类细分为中国人类、美国人类等，中国人类、美国人类都具有人类的特性和行为，如身高、体重等，但其又具有自己的特性，如国籍、肤色等，因此中国人类、美国人类可以继承人类，这样可减少一些重复的代码。

（2）父类和子类：被继承的类称为父类或基类，继承之后的新类称为子类或派生类。

（3）多态性：此处的多态是相对于具有继承关系的类而言的，指的是对于这些由继承而来的不同的类所产生的对象，执行相同的操作或功能时，其反应各不相同的情况。比如 A、B、C 是同一年级而不同班级的三个学生，当他们听到上课铃声时，会进入不同的教室去上课。

（4）静态多态性和动态多态性：静态多态性在编译时就能确定所引用的对象；动态多态性需要运行时才能确定所引用的对象。函数重载就是最简单的静态多态性。

（5）C++继承与多态的实现方法：C++使用冒号“ : ”实现继承，使用 virtual（虚拟的）关键字实现多态。

5. 封装与信息隐藏

（1）封装：其实就是把某些东西封装起来，只留一些必要的接口从外部进行操作，而具体内部的实现细节则隐藏起来。当然，也可以完全不给外部留下可操作的接口，但这样的封装不起任何作用，就没有意义了。比如在看电视时，我们只需通过电视机上的按键对其进行换台、调音量等操作，而无须关心电视机内部是如何实现这些功能的，假如不给电视机提供任何可对其操作的接口，那么这台电视机将是毫无用处的。

（2）函数是一种封装形式，因为函数的操作细节被封装在函数体中，函数体隐藏了函数的实现细节。类也是一种封装形式，类把它的所有成员都对外进行了隐藏。

（3）面向对象程序设计中的封装有两方面含义：一是相对于对象本身而言的，对象本身就是被封装后的结果，因为对象本身就拥有有关的数据和操作，且各个对象相对独立。比如对于等边三角形对象，可以使用该对象相应的接口来计算其面积，而无须知道该接口在等边三角形

对象中是怎样实现的。二是相对于对象内部而言的，对于面向对象程序设计，可以将对象内部的部分数据或操作对外进行隐蔽，这些被隐藏的数据或操作只能通过该对象提供的接口进行访问。比如将等边三角形对象的三个边的长度对外进行隐藏，只留下一个接口用于设置该对象的三个边的长度等。

（4）信息隐藏：指对象内部的数据隐藏。

（5）C++实现信息隐藏的方法：C++使用 public、private、protected 三个关键字来实现信息隐藏，这三个关键字被称为访问控制符。public 表示公有的，private 表示私有的，protected 表示受保护的。

8.2　类的声明/定义、类成员简介及相关运算符

8.2.1　类和对象的声明、定义

1. 类和对象的声明、定义基本语法

（1）声明类的语法：class 类名;。其中 class 是关键字，比如 class A; class B;声明了类 A 和类 B。

（2）定义类的语法：class 类名{类体};。其中 class 是关键字。

（3）定义类的注意事项。

① 大括号之后必须有一个分号或者老式 C 语言类型的声明。

② 大括号里的内容被称为类的成员，是类的一部分。若成员为变量，则称为成员变量；若是函数，则称为成员函数。

③ 类体可以为空。

④ 类的定义在右大括号（而不是分号）结束处结束。

⑤ 例如 class A{};class B{int a;};，表示分别定义了类 A 和类 B，其中类 A 是一个空类，而类 B 有一个数据成员变量 a。

（4）对象的定义：定义对象与定义变量相同，只是使用的类型不是内置类型，而是已定义的类类型。比如 A ma,mb;表示定义类 A 的两个对象 ma 和 mb，这里假设类 A 已经被定义过。

（5）类头与类体：定义类时由 class 关键字及其后的类名组成的部分称为类头，由大括号括起来的部分称为类体，类体可以为空。比如 class A{int a; int b;};，其中类头是 class A，类体是{int a; int b;}。

（6）类与类型：类一旦被定义，就确定了一种新的类型，新类型的名字就是定义类时的名称。因此类就是一种类型，只是这种类型比较复杂，其与内置类型是相对的，一般把类称为“自

定义类型”，对于内置类型 int、float 等是类型的名称，而自定义类型的名称就是类名。

（7）即使两个类具有完全相同的成员列表，它们也仍是两个不同的类型，而不是相同的类型。比如 class A{int a;}; class B{int a; };，类 A 和类 B 的成员完全相同都只有 int 类型变量 a，但它们仍是两个不同的类型。

2. 类的声明、定义及对象定义的区别

（1）类的声明只为程序引入了名字，但不知道这个声明的类具体的大小及其类型占据多少存储空间，这时就无法使用此类定义对象。这与不知道 int 类型占据多少存储空间就无法定义变量相似。

（2）类定义之后就会生成新的类型，这时类的所有成员就确定了，类的存储大小也可以确定，就可以使用该类定义其类型的对象了。

（3）定义类时不会分配存储空间，这时我们只知道这个新定义的类型会占据多少存储空间。这就与知道 int 类型占据存储空间的大小，但在定义 int 类型的变量之前，系统并不会为内置类型 int 分配存储空间一样。

（4）定义对象时系统会为其分配存储空间。其实定义对象与定义变量差不多，只是对象和变量的类型不同，一个是自定义类型，一个是内置类型。

（5）类只能被定义一次，若在多个文件中定义了相同的类，那么对该类的定义都必须完全相同。

（6）示例如下：

```
class A;              //类的声明，只为程序引入了名字，无法确定该类占据多少存储空间
A ma;                 //错误，因为类 A 未定义，还不知道 A 占据多少存储空间，所以无法使用类 A 来创建对象
class B{int a; };     //类的定义
B mb;                 //正确，定义对象，系统会为其分配存储空间
```

8.2.2 类成员简介、成员运算符、作用域解析运算符、访问控制符

1. 类成员简介

（1）类成员：类的成员在定义类的大括号中（类体）声明，其声明方式与声明普通变量和函数相同，它是类的一部分，在类体中声明的标识符都是类成员。比如 class A{int a ; int b; void f();};，类 A 的成员有三个，即整型变量 a、b 和函数 f。注意：不是在类体中声明的标识符不是类成员。比如 class A{void f(){int a;}};，这里 a 不是类 A 的成员，因为 a 是在函数体内进行声明的；f 是类 A 的成员，因为 f 是在类体中声明的。

（2）类成员通常被分为数据成员和成员函数两大类，其中数据成员是在类中声明的变量、对象等。

（3）类中各种成员的称呼：若成员是普通变量（比如 int、float 类型变量等），则被称为数

据成员；若成员是函数，则被称为函数成员或成员函数；若成员是类的对象，则被称为对象成员；若成员是静态变量，则被称为静态数据成员。通常把除成员函数外的成员统称为数据成员。其余名称根据成员的类型可以很容易理解。

（4）类成员不能是自身类型的数据成员。比如 class B{B mb; }; 错误，因为 B mb;的类类型 B 的大小是未知的、不确定的，必须在类 B 定义之后才能定义它的对象。注意：class A{void f(){A ma;}};是正确的，因为在函数体内定义的 ma 不是类 A 的成员，而且在使用函数 f 时，类 A 的对象已经创建了。

（5）类成员可以是指向自身类型的指针，也可以是指向被声明但未被定义的类类型的指针或引用。因为指针和引用的大小是固定的，与指针所指向的对象的大小无关，所以可以声明该类指针，但是必须等到定义了该类之后才能使用这种指针。比如 class A{ A *p;};正确，其中 A *p 中的指针 p 的大小是固定的（32 位机一般是 4），与指针本身的类型无关，而名称 A 在 A *p 之前的类头 class A 中已经看到了，程序认为 A 已被声明，因此 A *p 是正确的。

（6）不能从类外直接访问类成员，而是必须通过对象进行访问（静态成员除外），这是因为定义类时形成了一个类作用域，在类中定义的成员具有类作用域，在类外是不知道类中的成员的，所以类成员不能从外部直接进行访问。

2. 数据成员的初始化及对象的数据成员的独立性

（1）在定义类时类的数据成员是不能进行初始化的，数据成员声明的其他形式与普通变量相同。因为定义类时系统并没有为类分配存储空间，所以系统也就无法存储这些数据成员的值。比如 class A{int a=1;}; 是错误的，因为数据成员 a 在定义类时不能进行初始化。再比如 class A{int a,b;};，连续声明 a 和 b 两个数据成员。

（2）创建对象之后，对象就拥有了定义该类时的数据成员的副本。比如 class A{int a; int b; void f();};A ma, mb;，则对象 ma 和 mb 都拥有类的数据成员 a、b。

（3）由类创建的每个对象是相互独立互不影响的，每个对象都拥有该类的数据成员的副本，修改一个对象的数据成员不会改变另一个对象的数据成员。比如 class A{public: int a; }; A ma, mb; ma.a=1; mb.a=2;，则 cout<<ma.a;输出 1，其中 public 是访问控制符；这里对象 ma 的数据成员 a 与对象 mb 的数据成员 a 是相互独立的，它们互不影响。

（4）每个类对象的成员函数只有一个副本。其原理详见第 11 章中的 this 指针部分。

（5）对象的初始化：因为对象是客观存在的实体，在创建对象时会为对象分配存储空间，因此对象应该有确定的值，这时对象所拥有的数据成员就应该有确定的值，可以对对象进行初始化来达到初始化对象的数据成员的值的目的，初始化之后，也可以改变该对象的数据成员的值。对象的初始化使用构造函数进行，有关构造函数的内容详见后文。

（6）类的对象的生命期和作用域与普通变量相同，详见第 7 章。

3. 类访问控制符及 struct 结构简介

（1）C++使用访问控制符来实现信息隐藏。

（2）访问控制符有三个，分别是 public（公有的）、private（私有的）和 protected（受保护的）。

（3）访问控制符出现在类体中且后面应有一个冒号“:”。

（4）private：表示该成员只能被成员函数和类的友元访问，在类外或该类的对象都不能访问。比如 class A{private: int a; void f(){a=1;}}; A ma;，则 ma.a=1;和 ma.f();都是错误的，因为成员 a 和 f 是私有的，不能被类的对象访问；而成员函数 f 中的 a=1;是正确的。

（5）public：表示该成员在任何地方都可被访问。比如 class A{public: int a; void f(){}}; A ma;，则 ma.a=1;、cout<<ma.a; 和 ma.f();都是正确的。

（6）protected：对该类表现为像 private，对子类则依情况，详见第 13 章。

（7）若未指定访问控制符，则默认是 private。

（8）一个类可以有多个访问控制符。

（9）访问控制符一旦出现就一直有效，直到遇到另一个访问控制符或结束的右括号为止。

（10）C++中的 struct 结构类型，与 C++中的关键字 class 的用处是一样的，唯一的区别在于 class 定义的类默认是 private，而 struct 定义的类默认是 public。

4. 成员函数基础

（1）在类中声明的函数被称为成员函数。比如 class A{public: void f();}，则 f 是类 A 的成员函数。

（2）成员函数只在其所声明的类中可见，在类外是不可见的。

（3）成员函数可以直接访问类中的成员，而不需要点或箭头运算符。比如 class A{public: int a; void f(){a=1; } };，其中 a=1 只需直接访问，不需要点或箭头运算符。

（4）成员函数可以访问类中的公有的、私有的、受保护的成员而不会破坏这些成员的访问权限。比如 class A{private: int a; void g(){} public: void f(){a=1; cout<<a<<endl; g(); }};，函数 f 访问了类 A 的私有成员 a 和函数 g，但 a 和 g 仍是私有的。

（5）成员函数也可以被重载，但范围限制在当前类的作用域中。在类中重载的成员函数与类外的函数不相关。

（6）注意：每个类对象都会拥有自己的数据成员的副本，但每个类对象的成员函数的副本只有一份。比如 class A{public: void f(){static int a=1; a++; cout<<a<<endl;}; A ma; A ma1; ma.f();ma1.f();，两次调用函数 f 依次输出 2 和 3，可见 ma 和 ma1 是共用的同一个成员函数。

5. 成员函数定义

（1）成员函数既可在类中定义，也可在类外进行定义。比如 class A{public: void f(){}};，在类 A 中定义函数 f。

（2）在类外定义成员函数的方法：使用作用域解析运算符“::”来表示所定义的函数是某个类的成员。比如 class A{ void f();};，类外定义函数 f 的方法为 void A::f(){};。

（3）在类外定义成员函数时，被定义的成员函数必须先在类体内被声明，也就是说，带有成员函数声明的类体必须出现在成员函数的定义之前。比如 void A::f(){}class A{void f();}; 错误；class A; void A::f(){}class A{void f();}; 同样是错误的。

6. inline 内联与非内联成员函数

（1）在类中定义的函数，会被自动作为 inline 内联函数处理。当然，也可以加上 inline 关键字。

（2）在类体中使用 inline 关键字声明的成员函数是内联函数，在类体外定义函数时使用 inline 关键字的成员函数也是内联函数。

（3）注意：内联函数必须在调用它的每个文件中都被定义，因此在将程序写在多个文件中时，需要注意内联成员函数的定义问题。若类体中的内联成员函数未定义，则应将其定义放在类定义出现的头文件中。

示例 8.1：数据成员、成员函数及访问控制符

```
#include <iostream>
using namespace std;
class B1;              //声明一个未定义的类
class B2{public:int a;};
//void A::f2(){}    //错误，带有成员函数声明的类体必须出现在成员函数的定义之前
class A1{
    //C++访问控制符默认是 private
    //A1 ma1;         //错误，类成员不能是自身类型的数据成员
    A1 *p;            //正确，类成员可以是指向自身类型的指针，因为指针和引用的大小是固定的
    B1 *p1;           //正确，类成员也可以是指向被声明但未被定义的类类型的指针或引用
    //int a1=1;       //错误，类的数据成员在定义类时是不能进行初始化的
    int a2;
    public:           //在访问控制符之后应有一个冒号
      int a3;
      void f(){ a2=1; }   //成员函数可以访问类中的公有的、私有的、受保护的成员
                          //而不会破坏这些成员的访问权限。成员函数访问类中的成员时，不需要点或箭头
                          //运算符，可直接进行访问
      void f1();  void f2();
      protected:          //一个类可以有多个访问控制符，访问控制符一旦出现就一直有效，
                          //直到遇到另一个访问控制符或结束的右括号为止
          int a4;  }; //类 A1 结束
```

```
void A1::f1(){}    //在类外定义成员函数的方法，注意作用域解析运算符“::”的应用
//void A1::k(){}        //错误，函数 k 在类 A1 中未被声明
void main(){
    //由类创建的每个对象是相互独立互不影响的，每个对象都拥有该类的数据成员的副本，修改一个对象的数
    //据成员不会改变另一个对象的数据成员
    A1 ma1;   A1 ma2;
    ma1.a3=1;     ma2.a3=2;  //使用点运算符访问成员
    cout<<ma1.a3<<endl;//可见 ma1 的数据成员 a3 的值并未被 ma2 修改。ma1 和 ma2 中的 a3 是互不影响的
    //ma1.a2;                 //错误，对象无法访问 private 成员
    //ma1.a4;                 //错误，对象无法访问 protected 成员
}
```

7. 成员运算符和作用域解析运算符

（1）成员运算符：.（点）运算符和->（箭头）运算符被称为成员运算符。

（2）通过对象访问成员时，需要使用点运算符或箭头运算符。

（3）使用点运算符访问对象的成员，其方法为“对象名.成员名”。比如 class A{ int b; void f(){}}; A ma;，则 ma.b; ma.f();表示分别访问对象的数据成员 b 和成员函数 f。

（4）使用箭头运算符访问对象的成员，其方法为“指向对象的指针名->成员名”。比如 class A{ int b; void f(){}}; A ma; A *p=&ma;，则 p->b; p->f();表示分别访问指针所指向的对象的数据成员 b 和成员函数 f。

（5）点运算符和箭头运算符的左操作数都不能是类名，点运算符的左操作数是类的对象，而箭头运算符的左操作数必须是指向对象的指针。这两种运算符也可用于结构、枚举、联合类型的变量访问其成员，其原理相同。

（6）::（作用域解析运算符）的作用是位于运算符右边的名称应在运算符左边的作用域中进行查找，因此作用域解析运算符的左操作数应该是作用域的名称。

（7）作用域解析运算符的应用。

① 在类外定义成员函数时就需要使用作用域解析运算符。

② 访问类定义的静态成员：每个类都定义了一个作用域，因此理论上使用作用域解析运算符可以直接访问类中的成员，但对于类而言，除静态成员外，其他成员是不能直接通过类名进行访问的。比如 class A{public: static void f(){}};，则 A::f();表示使用类直接访问类的静态成员函数，若 f 是非静态的则会出错。

③ 作用域解析运算符用于名称空间中。

④ 作用域解析运算符用于访问全局变量。比如 int a; void main(){int a=1;::a=2;}，其中::a 表示使用的是全局定义的变量 a，而不是 main 函数中定义的变量 a。

（8）作用域解析运算符和成员访问运算符的区别：作用域解析运算符一般用于指明某个名称来自于哪个作用域；而成员访问运算符则表明某个成员属于某个对象。作用域解析运算符还用于在类外定义成员。

（9）综合示例：

```
class A{public: int a;}; A ma; A* p=&ma;
```

① ma.a=2;，正确，表示访问对象 ma 的成员 a。

② A.a=2;，错误，因为 A 不是对象，它是一个类作用域的名称。

③ p->a;，正确，使用指针访问指针所指向的对象 ma 的成员 a。

④ ma->a;，错误，ma 不是指向对象的指针。

⑤ p.a;，错误，p 不是对象。

⑥ A::a=2;，对于类作用域是错误的。对于类作用域只有静态成员才能这样访问 。

⑦ ma::a=2;，错误，ma 不是作用域名称。

⑧ ma.A::a;，正确，注意“::”运算符的优先级高于点运算符，这里表示来自于类作用域 A 中的名称 a 是对象 ma 的成员。

⑨ A::ma.a;，错误，因为“::”运算符的优先级更高，因此先计算 A::ma，这里 ma 不是类作用域中的名称，所以错误。注意：无法使用小括号改变作用域解析运算符和点运算符的优先级，比如 A::(ma.a);是错误的。

```
class B{public: int a; }; class A{public: static B mb; }; B A::mb;
```

① B A::mb;，不能省略，这是在类外定义静态成员的方法，详见第 11 章。

② A::mb.a=1;，正确，“::”运算符的优先级高于点运算符，因此先计算 A::mb，表示来自于作用域 A 中的名称 mb，其中 mb 是一个对象。

```
namespace N{class A{public: void f();};}
```

① 在名称空间 N 外定义函数 f 的方法为：void N::A::f(){}。

② 创建类 A 的对象的方法为：N::A ma;，创建类 A 的对象 ma。

③ 可以看到，作用域解析运算符的作用主要是为了指明某个名称来自于哪个作用域。

8.3　类作用域

8.3.1　类作用域中的名称

1. 简介

（1）每个类都定义了一个新的作用域，不同的类具有各不相同的类作用域。

（2）不能从类作用域外直接访问类中的成员，因为定义类时形成了一个类作用域，在类中定义的成员具有类作用域，在类外是不知道类中的成员的。

（3）一般规则：在类作用域中用到的名称必须先声明后使用。注意：成员函数有一些例外。

2. 名称是否处于类作用域中（是否使用作用域解析运算符）

（1）在类作用域中可以直接访问类中的成员，而不需要使用作用域解析运算符。

（2）当在类体（类作用域）外定义类成员时，其定义是否在类作用域中，就决定了访问类作用域中的其他成员时是否需要使用作用域解析运算符。

（3）重要规则：若类成员的定义在类体（类作用域）外，则一旦出现被定义成员的完全限定名，直到该成员定义结束，该成员的定义就都是在类体（类作用域）中。也就是说，相当于此成员是在类体（类作用域）中定义的，所以直到该成员定义结束，可以直接使用类体中的其他成员，而无须使用作用域解析运算符；但在被定义成员的完全限定名之前的部分，则不是在类体（类作用域）中，因此这部分若要访问类体中的其他成员，就需要使用作用域解析运算符以指明成员来自于哪个作用域。此规则不仅适用于类作用域，还适用于名称空间作用域、嵌套类等。

（4）示例如下。

成员函数：

```
class A{public: int a; typedef int IN; IN f(IN b); };
A::IN A::f(IN b) {a=1; return 2;}
```

① 在类体外定义的成员函数 f，在完全限定名 A::f 出现时（注意：完全限定名不包括后面的形参），直到该定义结束的部分，都被认为是在类体中进行的，因此在这部分之后直到该定义结束，可直接访问类中的其他成员。所以形参对成员 IN 和在函数体中对成员 a 的访问不需要使用作用域解析运算符。

② 在被定义成员函数的完全限定名 A::f 之前的部分（即函数的返回类型），则不是在类体中定义的，因此这部分要访问类体中的成员应使用作用域解析运算符，以指定所使用的名称来自于哪个作用域。所以函数 f 的返回类型使用了作用域解析运算符。

③ 注意：应从被定义的成员处开始分析，比如本例应从 f 处开始分析，而不是从 A::IN 中的 IN 处开始分析，因为 IN 不是被定义的成员。

静态数据成员：

```
class A{public: typedef int IN; static IN a; static int f(){return 2;} } ;
A::IN A::a=f();
```

① 在类体外被定义的静态数据成员 a，在完全限定名 A::a 出现时，直到该定义结束的部分，都被认为是在类体中进行的，因此在这部分之后可直接访问类中的其他成员，直到该定义结束。所以对函数 f 的访问不需要使用作用域解析运算符。

② 在被定义成员的完全限定名 A::a 之前的部分（静态成员变量的类型），不是在类体中定义的，因此这部分要访问类中的成员应使用作用域解析运算符，以指定所使用的名称来自于哪个作用域。所以静态成员变量 a 的返回类型应使用作用域解析运算符。

③ 注意：静态成员变量必须在类外进行定义。

8.3.2　类作用域中的名称解析

类作用域中的名称解析被分为两部分，其中一部分是类体中出现的名称解析；另一部分是类成员函数的函数体中的名称解析。类成员函数的函数体中的名称解析与常规的作用域中的名称解析的表现有点不相同，其可以不必先声明后使用。

1. 类体中出现的名称解析过程

以下规则也适用于函数形参，但成员函数的函数体中的名称和默认实参除外。

（1）查找位于该名称之前的类成员的声明，若找到，则使用该名称。

（2）若没找到，则在包含该名称的类体（或作用域）之前的名称空间（或作用域）中查找。一直按此步骤进行下去，直到找到该名称的声明或者出错为止。

（3）这就是常规的名称解析过程，即名称应先声明后使用。

示例 8.2：在类体中出现的名称解析过程

```
typedef long T6;
namespace D{      //namespace 用来创建名称空间，名称空间相当于一个作用域，只是作用域的名称为 D
   typedef int T1;
   typedef int T3;
   class A { public:
     typedef int T2;
     //typeid(a).name()的作用是输出 a 的类型。假如 a 为 int 类型，则输出 int
     void f1(T1 a1){cout<<typeid(a1).name(); }  //正确，使用在名称空间中声明的 T1
     void f2(T2 a2){cout<<typeid(a2).name(); }  //正确，使用类 A 中声明的名称 T2
     void f3(T3 a3){cout<<typeid(a3).name(); }  //正确，使用名称空间 D 中的 T3
     //void f4(T4 a4){ } //错误，类 A 中声明的 T4 在 f4 的定义之后，因此 T4 不可见
     //void f5(T5 a5){ }                        //错误，同上
     void f6(T6 a6){cout<<typeid(a6).name(); }  //正确，全局作用域中的名称 T6
    //void f7(T7 a7){ }   //错误，全局作用域中的名称 T7，在 f7 的定义之后，T7 不可见
    typedef double T3;  //此名称未被 f3 的形参 T3 查找到
    typedef int T4;  };        //类 A 定义结束
    typedef short T5; }        //名称空间 D 结束
typedef char T7;
void main(){}
```

2. 成员函数的函数体中出现的名称解析过程

以下规则同样适用于默认实参、静态变量。

（1）在成员函数的函数体的局部作用域中查找，若在此名称之前找到，则使用该名称。

（2）若第一步未找到，则在包含该成员函数的类体（或类作用域）中查找所有的类成员声明，若找到则使用该名称。注意：查找的范围是包含该成员函数的类的整个类作用域，因此有

可能会在该成员函数之后的类作用域中找到该名称。此步不遵守名称先声明后使用的规则。

（3）若第二步未找到，则在此成员函数定义之前的作用域中进行查找，若找到则使用该名称。此步主要是指成员函数在类外定义的情形，要考虑在类外定义之前的作用域中查找，而不仅仅指包含该成员函数的类之前的作用域。

（4）对于在类中定义的成员函数，若第二步未找到，则在包含该成员函数的类体（或类作用域）之前的名称空间（或作用域）中查找。

（5）对于带默认实参的成员函数，若默认实参在类的内部指定，则按照第二步和第四步进行；若在类外指定，则按照第二步和第三步进行。注意：对于 typedef int T; void f(T a=b);，这里只有 b 才是默认实参，只有 b 才遵守这一规则，而 a 是形参，T 是形参类型，都不遵守这一规则。若默认实参是类的成员，则默认实参应是静态的，默认实参不能是非静态的数据成员。

（6）静态成员变量一般在类外进行定义，按照第二步和第三步进行。

示例 8.3：成员函数中的名称解析过程

```
int a5;
class A {public:
  void f(){
    //a1=1;  //错误，在成员函数的函数体的局部作用域中该名称之前未找到其声明
    int a1;  //正确
    a2=2;    //正确，使用类 A 中 a2 的声明（规则 2）
    //a3=3;  //错误，全局作用域中定义的 a3 出现在类 A 的定义之后（规则 3）
    a5=5; }  //正确，使用全局作用域中的名称 a5，a5 在类 A 的定义之前（规则 3）
  void g();  //准备在类 A 外定义的成员函数 g
  int a2; }; //类 A 结束
int a3;      //在类外定义的函数 g 之前定义的名称 a3
void A::g(){
  a2=2;      //正确，同函数 f
  a3=3;   //正确，使用全局作用域中的名称 a3，a3 的定义在 g 的定义之前。注意：在函数 f 中 a3 是不可见的
  //a4=4;    //错误，全局变量 a4 在函数 g 的定义之后（规则 4）
  }
int a4;      //全局变量 a4
void main(){A ma;  ma.f();  ma.g();}
```

示例 8.4：默认实参名称解析

```
#include <iostream>
using namespace std;
class A {public:
   void f(int a);
   void g(int a=c){}   //默认实参 c 在类的内部指定。注意：默认实参 c 是类 A 的成员，因此应是静态的
   //void h(int a=b);  //错误，无法找到全局名称 b
   //void g1(T a=c);   //错误，此处的 T 无法被识别，这里默认实参指的是 c，而 T 不是默认实参
   typedef int T;
   static int c;  };
```

```
int b=1;
int A::c=b;                    //静态成员变量的定义
void A::f(int a=b){            //默认实参 b 在类外部指定，全局名称 b 位于函数 A::f(...)的定义之前，
                               //所以能找到，而类体中的函数 h 无法找到名称 b
   cout<<a<<endl; }
void main(){A ma; ma.f();}
```

3. 名称隐藏及作用域解析运算符的再次使用

（1）先解析的名称会隐藏后解析的名称。

（2）作用域解析运算符的再次运用：虽然先解析的名称会隐藏后解析的名称，但仍可使用作用域解析运算符来访问被隐藏的名称。

重点理解成员函数与普通函数的区别，下面通过示例进行说明。

示例 8.5：对成员函数与普通函数的理解

```
void f(){cout<<"::f"<<endl;}
void f(int){cout<<"fi"<<endl;}
void g(int){cout<<"gi"<<endl;}
class A{public:
   void f();  //声明成员函数（未被定义），此名称会隐藏掉外围类中的同名函数 f()和 f(int)
   void g(){cout<<"gA"<<endl;}
   void x();  //声明成员函数 x
   void h(){void x();  //正确，对普通函数的声明（此函数未被定义），这不是对成员函数 x 的重新声明
         //void A::x();//错误，不能对成员函数 x 进行重新声明
         //f();  /*错误，查找名称时，最先找到成员函数 f，因此调用它，但未被定义，此处不会调用全
                 局函数。这里要重点理解成员函数与全局函数是不同的*/
         //f(3);  /*错误，查找名称时，最先找到成员函数 f，此时结束查找，但 f 不带任何形参。此处
                  全局的 f(int)被成员函数的 f()的声明隐藏了*/
         g();      //正确，调用成员函数 g()
         //g(3);   //错误。全局函数 g(int)被成员函数 g()隐藏了
         ::f();    //正确，调用全局函数 f()
         ::f(3);   ::g(4);} }; //同上
void main(){A ma; ma.h();}
```

第 9 章

构造函数、复制构造函数和析构函数专题

有关合成的默认构造函数、合成的默认析构函数、合成的默认复制构造函数、合成的赋值运算符的更深入讲解，请参阅后续章节。

9.1 构造函数与析构函数简介

9.1.1 构造函数、默认构造函数、单形参构造函数、explicit 关键字

1. 构造函数简介

（1）构造函数的形式为：class A{A（形参 1,形参 2...){...} };，类体中的函数 A 为构造函数。

（2）构造函数的主要作用是用于初始化对象的数据成员。

（3）构造函数就是一个函数，只不过它是类的一个特殊的成员函数。

（4）构造函数的名称必须与类名相同。

（5）构造函数不能返回任何值。也就是说，不能为构造函数指定返回类型（包括 void 类型）。比如 class A{ public: void A(){}};是错误的。

（6）构造函数可以有形参，也可以没有形参，还可以重载多个构造函数的版本。

（7）构造函数不能有 const 或 volatile 限定词，比如 class A{public: A() const{}}; 错误。

（8）只要创建对象，就会自动执行一个构造函数，且只执行一次；反过来理解，就是只要调用构造函数就会创建一个对象。比如 class A{public: A(){}; } void main(){A();}，在 main 函数中表示使用默认构造函数创建一个对象，不过这个对象没有名字。

（9）不能用类的对象来直接调用构造函数，但并不代表构造函数不可被显式调用。比如 class A{public: A(){} };A ma;，则 ma.A();是错误的，但 A::A()、A()、ma.A::A()都是正确的。

2. 默认构造函数

（1）默认构造函数是不需要实参就能被调用的构造函数，因此没有任何参数的构造函数就是默认构造函数，所有的形参都有默认值的构造函数也是默认构造函数。比如 class A{ public : A(){};，其中 A()是默认构造函数；再比如 class B{public: B(int i=1, int j=2){};，其中 B(int i, int j)也是默认构造函数。

（2）如果没有定义任何构造函数，则 C++会自动提供一个默认构造函数，该函数被称为合成的默认构造函数。如果用户自定义了一个构造函数，则 C ++就不自动提供默认构造函数了；这时如果用户也没提供默认构造函数，则此类就没有默认构造函数。比如 class A{};，则 A ma;将使用合成的默认构造函数创建对象 ma；再比如 class A{public: A(int i, int j){};，则 A ma;是错误的，因为没有默认构造函数可调用。

（3）默认构造函数只能有一个。比如 class A{public: A(){}A(int i=1){};是错误的，这里提供了两个默认构造函数。

示例 9.1：构造函数和默认构造函数

```
#include <iostream>
using namespace std;
class A{public:
   //void A(){}                          //错误，构造函数不能返回任何值（包括 void 类型）
   //A()const{}                          //错误，构造函数不能有 const 或 volatile 限定词
   //A1(){}                              //错误，构造函数的名称必须与类名相同
   A(){cout<<"A"<<endl;}                 //正确，构造函数可以有形参，也可以没有形参
   A(int i){cout<<"Ai"<<endl;}           //正确，可以重载多个构造函数的版本
   };
class A1{};                              //A1 使用合成的默认构造函数
class A2{public:
   A2(int i,int j){} };                  //若用户自定义了一个构造函数，则 C ++就不会自动提供默认构造函数；
                                         //这时如果用户也没提供默认构造函数，则此类就没有默认构造函数
class A3{public:
   A3(){}                                //这是默认构造函数
   //A3(int i=2,int j=3){}               //错误，这也是默认构造函数，因为默认构造函数只能有一个。所有的形
                                         //参都有默认值的构造函数也是默认构造函数
};
void main(){
   A ma;          //调用构造函数 A()创建对象
   A ma1(3);      //调用构造函数 A(int)创建对象
   A1 ma2;        //使用 A1 合成的默认构造函数创建对象
   //A2 ma3;      //错误，因为类 A2 没有默认构造函数可调用
   //ma.A();      //错误，不能用类的对象来直接调用构造函数
   A(3);}         //直接调用 A 的构造函数 A(int)创建一个临时对象，然后临时对象被撤销
```

3. 单形参构造函数、隐式转换与 explicit 关键字

（1）单形参构造函数也被称为转换构造函数，它执行的是一种类型转换，可以把某一类型转换为该类的类类型。比如，可以把 int 类型转换为类类型 A，把类型 B 转换为类型 A 等。如 class B{};class A{public:A(int i){} A(B mb){}}; A ma(8); B mb(9);，则 ma=3; 是正确的，此时会调用单形参构造函数把 int 类型的“3”转换为类型 A，再把转换后的结果赋给 ma；同理，ma=mb; 也是正确的，此时会调用 A(B mb)的单形参构造函数，把 mb 转换为类型 A，再进行赋值。语句 ma=3;和 ma=mb;相当于 ma=A(3); 和 ma=A(mb);，其中都使用了相应的单形参构造函数创建了一个临时对象，再使用临时对象进行赋值。

（2）有单形参的构造函数，同样支持显式的（强制）和隐式的类型转换。比如 class A{public:A(int i){} }; A ma(9);，则 ma=3;执行的是隐式类型转换；而 ma=(A)3;执行的是显式类型转换。

（3）注意：单形参构造函数执行的是把某类型转换为类类型；反过来，把类类型转换为其他类型则是不可以的，这种转换需要使用转换函数。比如 class A{public:A(int i){} }; A ma(7); int a=3;，则 ma=a;是正确的；但反过来，a=ma;就是错误的。

（4）单形参构造函数隐式转换规则：使用单形参构造函数存在一个隐式转换，其规则是，在需要使用该类类型对象的地方，就会将这个单形参构造函数的形参类型转换为该类类型。在转换的过程中可能会生成不必要的临时对象，因此以下情况会出现隐式转换

① 将该类的对象初始化为单形参构造函数的形参类型的值时。比如 class A{public: A(int i){} };，则 A ma1=3;与 A ma1(3)等同，都会调用单形参构造函数来初始化 ma1 对象。

② 将单形参构造函数的形参类型的值赋给该类的对象时。比如 class A{public: A(int i){} }; A ma(2);，则 ma=3;就会把整数值“3”作为实参调用单形参构造函数 A(int i)生成一个临时对象，然后把这个临时对象赋给 ma。此语句等同于 ma=A(3);。

③ 把单形参构造函数的形参类型的值作为实参传递给需要该类类型形参的函数时。比如 class A{public: A(int i){} };void f(A ma){};，则 f(3)会执行单形参构造函数的隐式转换，首先把“3”作为实参调用 A(int i)生成一个临时对象，然后把这个临时对象作为实参传递给函数 f。

④ 函数返回值为该类类型，但该函数却返回一个单形参构造函数的形参类型的值时。比如 class A{public: A(int i){} }; A f(){return 3;}，则在函数返回整数值“3”时，会把“3”作为实参调用 A(int i)并生成一个临时对象作为函数的返回值。

⑤ 使用隐式的内置类型转换（如标准转换、提升转换等），转换为单形参构造函数的形参类型时。比如 class A{public: A(int i){} };void f(A ma){}; 。则 f(1.2)会执行隐式类型转换，首先对 float 类型的“1.2”进行标准转换，转换为单形参构造函数的 int 类型，然后使用单形参构造函数创建一个临时对象，再把这个临时对象传递给函数 f 的形参。再比如 class A{public: A(int

i){}}; void f(const A &ma){};，则 f(2.3)同样会调用单形参构造函数进行隐式转换，首先把“2. 3”转换为 int 类型，然后使用转换后的值作为实参调用 A(int i)创建临时对象，再把该临时对象传递给函数 f 作为实参。注意：若形参为非 const 引用，则是不正确的。也就是说，对于 void f(A& ma){}，则 f(2.2)的调用是错误的，因为不能把右值转换为非 const 引用。有关引用详见第 7 章。

（5）限制单形参构造函数的隐式转换：将构造函数声明为 explicit 就可以禁止这种隐式转换。explicit 关键字只能作用于构造函数的声明上，在类外定义构造函数时不需要指定 explicit 关键字。比如 class A{public: explicit A(int i);}; explicit A::A(int i){} 错误，在类外定义构造函数时无须指定 explicit 关键字；再比如 class A{public: A(int i);}; explicit A::A(int){}也是错误的。

（6）explicit 只能限制单形参构造函数的隐式转换，但显式（强制）转换或明确调用单形参构造函数仍然是允许的。比如 class A{public: explicit A(int i){} }; A ma(9);，则 ma=3;是错误的，但 ma=A(3)；和 ma=(A)4 ;是正确的，A(3)是明确调用单形参构造函数，(A)4 是强制转换。

（7）若单形参构造函数被 explicit 限制，则不能再使用 A ma=3;形式的语法创建对象。比如 class A{public: explicit A(int i){} };，则 A ma=3;错误，正确形式为 A ma(9);。

（8）为避免不必要的转换和产生临时对象，建议单形参构造函数都使用 explicit 进行限制。

9.1.2　析构函数

1．析构函数的作用及形式

（1）析构函数用来释放或返还生命期即将结束的类对象所占据的资源。析构函数与构造函数是相对立或互补的，构造函数获取资源，析构函数释放资源。

（2）可以在析构函数中执行一些对象结束时需要执行的操作。

（3）默认析构函数：若没有显式定义析构函数，系统会自动生成一个默认的析构函数。

（4）析构函数的形式：析构函数是类的一个成员函数，析构函数需要在类名前加上“~”符号，它不能有返回值和形参，而且只能定义一个析构函数。比如 class A{public: ~A(){}};，其中 ~A()是析构函数。

（5）即使我们编写了自己的析构函数，默认析构函数也仍然存在，这是与默认构造函数和复制构造函数不同的地方。因此，对于拥有类对象成员的类，当该对象结束时除了会调用自定义的析构函数之外，还要使用默认析构函数按照声明的相反顺序来释放类对象成员。若类成员是内置类型或复合类型的，则默认析构函数不会对这些成员产生影响。比如 class B{}; class A{public: B mb; ~A(){ cout<<"A"<<endl; }};void f(){A ma;} f(); 在函数 f 调用结束后，程序首先调用类 A 中自定义的析构函数输出字符 A，然后使用类 A 的默认析构函数调用类 B 的析构函数释放类对象成员 mb。注意：此例在类 A 的显式定义的析构函数中没有释放类成员 mb 的语句，类成员 mb 是由类 A 的默认析构函数调用类 B 的析构函数释放的。

2. 何时调用析构函数

（1）当对象的生命期结束时或类对象被撤销时会自动调用析构函数。

（2）对于局部对象，当该对象离开它的作用域时（即生命期结束时）会自动调用析构函数。

（3）若只有指向该对象的指针离开了作用域，是不会调用析构函数的。因为指针只是指向了该对象（指针也可以不指向任何对象），只有当指向的对象离开相应的作用域时才会自动调用析构函数。对于引用也是同样的道理，只有当所引用的实际对象离开作用域时才会调用相应的析构函数。比如 void f(){ A ma; {A *p=&ma; } cout<<"A"<<endl; } f();，则析构函数不会在指针 p 所在的块作用域结束时被调用，而是在 cout 语句之后离开函数 f 之前被调用，这时指针指向的对象 ma 离开了其作用域。

（4）当 delete 作用于指向该类对象的指针上时会自动调用析构函数。

（5）若是使用 new 动态分配的对象，则只有在指向该对象的指针被删除时才会调用析构函数；若程序没有删除该指针，则不会运行该对象的析构函数，从而对象会一直存在，导致内存泄漏。

（6）对于全局对象，当程序结束时调用析构函数。

（7）对于静态局部对象，若对象创建于函数内部，则构造函数只在第一次调用函数时被调用一次，即使函数被调用多次，构造函数也只会被调用一次。在函数调用结束后对象不会被释放，因此不会调用析构函数，只有程序结束时才会调用析构函数。

3. 析构函数的调用顺序

（1）基本原则：析构函数是按照与构造函数相反的顺序调用的，即析构函数是按创建对象时的相反顺序调用的，这和栈队列的后进先出一个道理。简单理解就是先构造的后析构，后构造的先析构。比如 class A{}; void f(){A ma, ma1, ma2;} f();，则调用函数 f 时先依次执行 ma、ma1、ma2 对应的构造函数创建对象，调用结束后再按照相反的顺序调用 ma2、ma1、ma 的析构函数释放对象所占的资源。

（2）在所有函数之外定义的对象，在 main 执行之前被构造，其析构函数在退出 main 之后才会被调用。比如 A ma; void main(){B mb;}，则先创建对象 ma，然后是 mb；在 main 函数结束时，首先调用 mb 的析构函数，然后调用 ma 的析构函数。

（3）若程序存在于多个文件中，且不同文件中都定义了全局对象，那么这些对象构造函数的调用顺序将是不明确的，相应的析构函数的调用顺序也是不明确的。

4. 何时需要自定义析构函数

（1）一般不需要显式定义析构函数。若需要显式定义析构函数，一般也需要显式定义赋值操作符和复制构造函数。这是一个很有用的经验。

（2）当构造函数中使用 new 来分配指针时，应按以下操作进行处理。

① 当在构造函数中使用 new 操作符来分配类成员指针的内存时，需要自定义析构函数，以便在析构函数中使用 delete 来释放其所分配的动态资源。

② 构造函数中的 new 应与析构函数中的 delete 相对应，即 new 对应于 delete，new[]与 delete[]对应。

③ 因为类的析构函数只有一个，所以所有的构造函数都要与这个唯一的析构函数相兼容。因此，若有多个构造函数，则所有的构造函数要么都使用 new，要么都不使用 new，以便与析构函数相兼容，且所有的构造函数使用 new 的方式应相同。也就是说，要么都使用 new，要么都使用 new []，不能一些构造函数使用 new，而另一些使用 new []；否则会出现析构函数中的 delete 错误释放由 new 分配的内存的情况。比如 class A{public: int *p; int a; A(){p=new int;} A(int i){a=1; p=&a;} ~A(){delete p;} }; A ma; A ma1(3);，其中 A ma;没有问题，正确释放了内存；而 A ma1(3);则是错误的，因为析构函数使用 delete 释放了不是由 new 分配的内存。

④ 因为有指针成员，因此还应自定义复制构造函数和赋值操作符。

5. 显式调用析构函数

（1）析构函数一般是不需要显式调用的，只有在少数情况下需要显示调用，比如对于使用布局 new 运算符（详见第 12 章）时应显式调用析构函数。

（2）在显式调用析构函数时，只是简单地执行析构函数内的语句，它不会释放内存和销毁对象。

（3）若程序使用 new 动态分配了类成员指针的内存，且自定义的析构函数内又使用 delete 释放了内存，这时若显式调用析构函数，则会引起动态内存被重复释放错误。比如：

```
class A{public: int *p; A(){p=new int(3);} ~A(){delete p;} };
void main(){ A ma; ma.~A();}
```

在 ma.~A();时会调用自定义的析构函数释放指针 p 的内存，然后在 main 函数结束时，还会再调用一次自定义的析构函数，再次释放指针 p 的内存，从而造成内存重复释放错误。

（4）若类中有对象成员，在显式调用析构函数时，则会使用默认析构函数调用相应类的析构函数释放对象成员的资源，当该类结束时，还会再释放一次对象成员的资源，导致出现两次释放资源错误。比如：

```
class B{public: ~B(){cout<<"~B"<<endl;}};
class A{public: B mb; ~A(){cout<<"~A"<<endl;}};
void main(){ A ma; ma.~A(); };
```

在 ma.~A();时程序调用类 A 的自定义析构函数输出~A，然后使用类 A 的默认析构函数调用对象成员 mb 所属类 B 的自定义析构函数输出~B，释放对象成员 mb 的资源；当 main 函数执行结束之后，还会调用一次类 A 的自定义析构函数输出~A，然后同样会使用类 A 的默认析构函数释放对象成员 mb 的资源，这时 mb 被释放两次。

9.2 对象初始化

类只是对象的一个框架（或模板），因此编译器不会为类分配存储空间。这就意味着，对于类中的数据成员，在创建类时是不能进行初始化的，因为没有空间来存储这些数据。比如 class A{ int a=1;};，其中的初始化语句 int a=1;是错误的。

9.2.1 使用构造函数、默认构造函数初始化对象

1．使用构造函数初始化对象（调用构造函数的方法）

（1）调用构造函数（使用构造函数初始化对象）的几种方法如下。

① 显式调用构造函数初始化对象的方法：类名 对象名=类名(实参表)。比如 class A{public: A(){}A(int i, int j){}}; A ma=A();A mb=A(1,2);，其中 ma 调用默认构造函数初始化 ma 的数据成员，而 mb 则调用带两个形参的构造函数初始化 mb 的数据成员。

② 隐式调用构造函数初始化对象的方法：类名 对象名(实参表)。比如 class A{public: A(int i, int j){}}; A ma(1,2);，则 ma 调用带两个形参的构造函数初始化 ma 的数据成员。

③ 使用 new 动态创建对象。比如 class A {public: A(int i, int j){}}; A *ma=new A(1,2);，则调用带两个形参的构造函数初始化对象（注意：这个对象没有名称），然后将该对象的地址赋给指针变量 ma，该对象可使用指针来访问。

④ 单形参构造函数可以像初始化普通变量那样初始化对象，即形式为：类名 对象名=初始值。注意：此方法只对单形参构造函数有效，若没有定义单形参构造函数，则是错误的。比如 class A{public: A(int i){}}; A ma=5; 正确；再比如 class A{public: A(int i,int j){}};，则 A ma=2;、A ma=2,3; 和 A ma={2,3}; 都是错误的。

（2）调用构造函数时编译器的内部展开样式：比如 class A{public: A(int i){}}; void main(){ A ma(1);}，则在 main 中内部展开的样式为 void main(){A ma; ma.A::A(1);}。

（3）特别注意：对于显式调用构造函数初始化对象的方式（即形如 A ma=A(...);的初始化方法），C++有两种实现，其中一种是使用与隐式调用构造函数初始化对象相同的方法来实现；另一种是允许调用相应的构造函数创建一个临时对象，然后将该临时对象复制到要初始化的对象中，最后丢弃该临时对象（不一定会立即丢弃临时对象，有可能会晚点丢弃）。这取决于编译器采用哪种方式，若使用临时对象的方式，则程序会多调用一次构造函数和析构函数，但该析构函数不一定会立即调用，有可能会晚点调用，也就是临时对象不一定会立即删除。对于第二种方式还要注意赋值与初始化的区别，即 A ma=A(...); 与 A ma; ma=A(...)是不同的，ma=A(...);是对已存在的对象赋值，在这里会产生临时对象；而 A ma=A(...);是初始化，不一定会产生临时对象。

2. 使用默认构造函数初始化对象

（1）C++合成的默认构造函数不具有任何初始化数据成员的功能，其函数体相当于是空的，因此不能用于将数据成员的值初始化为 0。比如 class A{public:int a;}; A ma;，则 ma 中的数据成员 a 未被初始化，其值是一个随机值。

（2）合成的默认构造函数的初始化规则：按照与变量初始化相同的规则来初始化数据成员，其成员的值根据创建对象时的作用域来确定，对于内置类型和复合类型的成员（如指针、数组），若对象是在全局创建的，则成员初始化为 0；若对象是在局部创建的，则成员拥有随机值；若是对象成员，则使用该对象所属类的默认构造函数来初始化。注意：内置类型成员并不是真正地被初始化。

（3）创建对象时若没有显式调用其他构造函数，则将调用默认构造函数初始化该对象。比如 class A{public: A(){}A(int i, int j){} };A ma; A mb=A(); A *mc=new A(); 都将调用默认构造函数初始化对象

（4）不要用形如 A ma();的形式调用默认构造函数来初始化对象。此语句没有错，但编译器会把 ma 理解成一个返回类型为 A 的函数声明，而不会使用默认构造函数来创建对象 ma。正确的形式是 A ma 或 A ma=A();，A ma 使得默认构造函数创建类的对象时就像创建普通变量一样

（5）注意：默认构造函数与合成的默认构造函数是两个不同的概念，合成的默认构造函数是由 C++编译器（或系统）自动生成的。

示例 9.2：使用构造函数和默认构造函数初始化对象

```
#include <iostream>
using namespace std;
class C{public:
    C(int i,int j){}
    ~C(){cout<<"~C"<<endl;}};
class A{public:
    A(){}
    A(int i){}
    A(int i,int j){} };
class B{ public: int a;};
B mb;                //使用 C++合成的默认构造函数在全局创建的对象，将使数据成员初始化为 0
void main(){
    A ma1(2);        //隐式调用构造函数 A(int)初始化对象
    A ma2=3;         //单形参构造函数特有的初始化对象的形式，可以使用与普通变量初始化相同的语法
    C mc1=C(2,3);  //显式调用构造函数 A(int,int)初始化对象，此方法有可能创建临时对象
    A *ma3=new A(2,3);  //使用 new 动态创建对象
    A ma4;               //使用默认构造函数创建对象
    A ma5();  //此语句没错，但编译器会把 ma5 理解成一个返回类型为 A 的函数声明
    B mb1;    //在局部使用合成的默认构造函数创建的对象是没有进行初始化的，mb1 的数据成员拥有随机值
    cout<<mb.a<<endl;   //输出 0
cout<<mb1.a<<endl;  }   //输出随机值
```

9.2.2 使用成员初始化表初始化数据成员

1. 使用成员初始化表的基本规则和语法要求

（1）成员初始化表：位于构造函数的函数体之前、构造函数的形参表之后（即形参表的小括号之后），并由冒号开始，其后是成员名和括在括号中的初始值（类似于函数调用的语法），各个成员名之间使用逗号隔开，其中的成员是在类中进行声明的。比如 class A{public: int a;int b; A():a(2),b(3){} };，其中:a(2),b(3)就是成员初始化表。

（2）成员初始化表的语法要求。

① 成员初始化表只能在定义构造函数时指定，在声明时不能指定。比如 class A{public: int a, b;A():a(1),b(2);};是错误的；正确的形式为 class A{public:int a,b; A();}; A::A():a(1),b(1){}。

② 成员初始化表中的每个成员只能被指定一次。比如 class A{public: int a,b; A():a(2),b(3),a(4){}}错误。

（3）使用成员初始化表初始化数据成员的基本规则。

① 使用成员名之后括号中的初始值来初始化相应的成员。比如 class A{public: int a;int b; A():a(2),b(3){} };，表示把成员变量 a 的值初始化为 2，b 的值初始化为 3。

② 若是对象成员，则调用该类对象的相应构造函数进行初始化，只需为构造函数提供相应的实参即可。比如 class B{public: B(){}B(int i, int j){}}; class A{public: B mb1, mb2; A():mb1(), mb2(1,2){}};，在类 A 中，A():mb1(), mb2(1,2)表示使用类 B 中的默认构造函数初始化 mb1，使用带两个形参的构造函数初始化 mb2。

2. 理解数据成员的初始化

（1）构造函数的执行过程一般分为两步：第一步为隐式或显式初始化阶段；第二步为计算阶段。

① 计算阶段指的就是构造函数体中所执行的语句。在构造函数体中执行的初始化其实是赋值，并不是真正的初始化。

② 显式初始化指的是使用成员初始化表显式初始化成员。

③ 隐式初始化指的是成员初始化表中未出现的成员使用隐式规则进行初始化，即对象成员使用该对象所属类的默认构造函数进行隐式初始化。对于内置类型成员不会进行隐式初始化。比如 class B{public:B(){} };class A{const int a; const B mb; A(){}};是错误的，其中 a 是 const 常量，必须被初始化，但因为是内置类型且未在成员初始化表中提及，所以 a 是未被初始化的，程序出错； mb 虽然也是 const 常量且未在成员初始化表中提及，但程序会使用类 B 的默认构造函数来初始化。由此例可见，内置类型成员没有被初始化，在全局创建对象时把内置类型成员初始化为 0 的行为，并非真正的初始化。

（2）使用成员初始化表与在构造函数体内初始化成员的区别如下。

① 使用成员初始化表才是真正地初始化成员，而在构造函数体内对数据成员的初始化其实是赋值。比如 class A{public:const int a; A(){a=3;}}; 错误，不能对常量 a 赋值，同时常量 a 未被初始化，可见构造函数体内的语句 a=3;并不是初始化而是赋值。正确的形式为：class A{public:const int a; A():a(3){}};，把常量 a 初始化为“3”，可见使用成员初始化列表才是真正的初始化。

② 对于非类类型（const 和引用除外）的成员，使用成员初始化表初始化它们后的结果和在构造函数体内进行赋值，在性能上是等同的。

③ 对于对象成员，总是在初始化阶段进行初始化（以显式或隐式方式），因此对象成员的初始化总是先于构造函数体内的语句，或者说对象成员不是在构造函数体内被赋值的。从此处也可以看到，在创建包含对象成员的对象时，对象成员先于所创建的对象而创建，也就是调用对象成员的构造函数先于包含该对象成员的对象的构造函数，释放成员时的析构函数的调用按照相反的顺序进行。

示例如下：

```
class B{public: B(){cout<<"B"<<endl;}}; B mb1;
class A{public: int a; B mb; A(int i):a(1),mb(){}
   A(int i, int j):a(1){ cout<<"A"<<endl; mb=mb1; } };
A ma(3); A ma1(3,3);
```

① 创建 ma 时，对象成员 mb 是在 A(int i)的成员初始化表中被显式初始化的。

② 创建 ma1 时，对象成员 mb 会在初始化阶段被隐式初始化（成员初始化表中未出现成员 mb 的名称），初始化步骤先于在函数体内进行的赋值 mb=mb1;，因此创建 ma1 时程序输出 BBA，而非 BAB。

③ 对于内置类型成员 a，在初始化阶段初始化还是在构造函数体内进行赋值，在结果和性能上是没有差别的。

3．必须使用和不能使用成员初始化表的情况

（1）必须使用成员初始化表初始化的成员有，没有默认构造函数的对象成员、const 成员、引用类型成员。这三种类型的成员必须在成员初始化表中进行初始化。

① 对于没有默认构造函数的对象成员，因为所有的对象成员都是在初始化阶段先于构造函数体内的语句进行初始化的，因此必须在成员初始化表中以显式方式初始化该成员，否则程序会因为找不到默认构造函数而出错。比如 class B{public:B(int i){}}; class A{public: B mb; A(){mb=B(3);}}; 是错误的，因为 mb 的初始化先于 mb=B(3)就已经进行了，因此程序会出现无法使用默认构造函数初始化类成员 mb 的错误。正确的语法是 A():mb(3){}。

② const 成员和引用类型成员：在声明时必须进行初始化，但因为内置类型成员不会进行隐式初始化，因此若在成员初始化表中未对 const 成员和引用类型成员进行初始化，则这些成员不会被初始化，而在构造函数体内进行的初始化实际上是赋值，对于 const 对象是无法进行赋值操作的，对于没有初始化的引用类型也是无法赋值的，因此这两种类型的成员都必须在成员初始化表中进行显式初始化。比如 class A{public: const int a; A(){a=1;} };错误，在构造函数体内不能对常量 a 进行赋值，正确的语法是 A():a(1){}。同理，class A{public: const int a; A(){} }; 也是错误的，因为常量 a 没有初始化。

（2）在类中声明 const 类型变量、没有默认构造函数的对象成员、引用成员时，所有的构造函数都必须使用初始化列表的形式来初始化这些成员。记住是所有的构造函数，也就是每定义一个构造函数都必须初始化这些成员。比如 class A{public: const int a; A():a(3){} A(int i){}}; 错误，因为 A(int i)没有在成员初始化表中初始化成员 a。

（3）数组成员是无法在成员初始化表中进行初始化的，若非要在成员初始化表中进行初始化，则可以将要初始化的数组封装成一个对象。比如 class A{public: int a[2]; A();};，则 A::A():a[0](1),a[1](2){}是错误的；同理，A::A():a(1,2){};也是错误的。

（4）不能使用成员初始化表初始化静态（static）成员。比如 class A{public: static int a; A():a(9){}};错误。

4．使用成员初始化表初始化的顺序

使用成员初始化表初始化的顺序不是按照成员在初始化表中出现的顺序执行的，而是按照成员在类中被声明的顺序执行的。比如 int a,b,c; A():c(1), b(2), a(c){}; 是错误的，初始化的顺序是，先使用 c 初始化 a，再将 b 初始化为“2”，然后把 c 初始化为“1”，因为初始化 a 的值时，c 还未被初始化，所以出错。

9.2.3 使用复制构造函数初始化对象及临时对象

1．复制构造函数、直接初始化、复制初始化

（1）使用复制构造函数应弄清的几个问题：何时调用复制构造函数、复制构造函数有何功能、为什么要定义自己的复制构造函数。

（2）复制构造函数的形式：A（const A& ma){...};。复制构造函数就是一个比较特殊的构造函数，其拥有一个形参，这个形参必须是对该类类型的引用，一般都会将该引用声明为 const，当然也可以不声明为 const。

（3）复制构造函数的作用：对对象初始化。一般在把一个已存在的对象复制到新创建的对象中时会使用复制构造函数。

（4）对新创建的对象和已存在的对象的理解，比如 A ma; A ma1(ma);会调用复制构造函数，因为 ma 是已存在的对象，而 ma1 是新创建的对象，　A ma1(ma);是初始化语句。

（5）复制初始化与直接初始化：复制初始化指的是在创建一个新对象时，其初值来自于另一个已存在的对象，复制初始化总是调用复制构造函数来初始化；直接初始化是把初始化式放在圆括号中，对于类类型来说，直接初始化总是调用与实参匹配的构造函数来初始化。比如 A ma; A ma1(ma); 就是复制初始化，这里使用一个已存在的对象 ma 来初始化所创建的新对象 ma1，这时会调用复制构造函数；而 A ma(实参表);是直接初始化，若实参只有一个，则假设实参是非 A 类型的。

（6）合成的默认复制构造函数：若没有显式提供复制构造函数，系统会自动生成一个默认的复制构造函数，该函数就被称为合成的默认复制构造函数。

（7）使用合成的默认复制构造函数初始化对象的原理：按照逐个成员初始化的原则来初始化新创建对象的成员。逐个成员初始化的原则指的是把现有对象的所有非静态成员逐个复制到新创建的对象中。对于类类型成员，则调用该类的复制构造函数进行相同的复制；对于数组成员，则复制数组的每一个元素；对于静态成员则不会被复制，因为静态成员属于整个类。关于静态成员详见第 11 章。对合成的默认复制构造函数的模型应理解为，使用复制构造函数初始化数据成员是在成员初始化表中进行的。比如 class A{public: int a, b, c; };，其合成的默认复制构造函数的模型可能就是 A(const A& ma):a(ma.a),b(ma.b),c(ma.c){}。

（8）调用复制构造函数：复制构造函数其实就是带一个形参的构造函数，拥有这种构造函数的性质，可以像调用带一个参数的构造函数那样对其进行调用，也可以使用 explicit 对其进行限制，以使形如 A ma=ma1;的语法不可使用。因此，可以有 3 种调用复制构造函数的方法，即 A ma(ma1);、A ma=A(ma1); 和 A ma=ma1;，假设 ma1 是类 A 声明的。

2. 赋值运算符及默认赋值运算符

（1）为什么需要定义赋值运算符？若类没有对其相应的运算符进行定义，则不能对其对象进行任何运算，比如不能将两个对象相加，如 A ma,ma1;，则 ma+ma1;是错误的。同样，若没有对赋值运算符进行定义，则不能在两个对象之间赋值。关于运算符重载详见第 12 章。

（2）合成的默认赋值运算符：若用户未自己定义赋值运算符，则 C++会自动生成一个默认的赋值运算符，该运算符被称为合成的默认赋值运算符。因此，即使不对赋值运算符进行定义，也可以在两个对象之间进行赋值，只不过使用的是合成的默认赋值运算符，比如 A ma; A ma1;，则 ma=ma1;是正确的。

（3）合成的默认赋值运算符的形式为：class A{public: A& operator=(const A&){...}}。

（4）合成的默认赋值运算符的工作原理：其工作原理与默认复制构造函数相似，都是逐个成员赋值，对于非数组成员，使用的是该类型的常规方式进行赋值；对于数组，则对每个数组

元素进行赋值；对于静态成员则不会被赋值，因为静态成员属于整个类；对于类类型成员，则调用该类的赋值运算符对其对象进行赋值。

（5）理解赋值与复制初始化的区别：赋值是在两个已存在的对象之间进行的，也就是用一个已存在的对象去改变另一个已存在的对象的值，赋值将调用赋值运算符对对象进行操作；而复制初始化是使用一个已存在的对象去初始化一个新创建的对象，调用的是复制构造函数，其行为属于初始化。比如 A ma; A ma1;，则 ma=ma1;是赋值，因为对象 ma 和 ma1 都是已经存在的对象；而 A ma1=ma;则是复制初始化，因为是用一个已存在的对象 ma 去创建一个新对象 ma1，将调用复制构造函数。

（6）复制初始化、赋值与直接初始化的区别：复制初始化和赋值都是在两个对象之间进行的操作，而直接初始化则不是。

（7）注意：在自动调用合成的默认赋值运算符和合成的默认复制构造函数时，应注意初始化类中的数据成员。比如 class A{public: int a; }; A ma, ma1;，则 ma=ma1;会得到意想不到的结果，因为这里调用默认的赋值运算符复制成员的值，但类 A 的成员 a 并没有初始化，因此复制的值是一个未知的值。

3. 何时调用复制构造函数及何时生成临时对象

这里不讨论容器等标准模板库。

（1）注意：只要产生了临时对象，就会调用一次析构函数以销毁该对象。

（2）下面通过示例讲解何时调用复制构造函数及何时生成临时对象。

```
class A{ public : A(){}   A(int i){}   A(int i, int j){}   ~A(){cout<<"~A"<<endl;}};
```

① 对于 VC++ 2010，在以下有可能生成临时对象的地方都不会生成；而其他编译器有可能生成也有可能不会生成

② 有没有生成临时对象，读者只需查看有没有多调用一次析构函数输出~A 即可验证。

```
A ma;
A ma1(ma);          //调用复制构造函数，ma 是已存在的对象，ma1 是新对象，且是初始化语句
A ma2=ma;           //调用复制构造函数，有可能生成临时对象根据编译器而定
A ma3=A(ma);        //调用复制构造函数，有可能生成临时对象。此语句与 A ma3=ma 是等价的
A *ma4=new A(ma);   //调用复制构造函数
A ma3=A(1,2);       //有可能生成临时对象，若生成临时对象，则会调用复制构造函数
                    //否则不调用复制构造函数。生成临时对象的步骤为，使用构造函数 A(1,2)
                    //创建一个临时对象，然后调用复制构造函数把 ma3 初始化为临时对象的副本
A ma4=3;            //可能生成临时对象，若生成临时对象，则会调用复制构造函数，否则不调用复制构造函数
                    //生成临时对象的步骤为，调用单形参构造函数 A(int)生成一个临时对象，然后调用复制
                    //构造函数把 ma4 初始化为这个临时对象的副本
A ma6;
ma6=ma;             //这是赋值，而不是初始化，不调用复制构造函数，不生成临时对象
A ma5(1)            //这是直接初始化，不调用复制构造函数，不生成临时对象
ma5=A(1,2);         //这是赋值，但 A(1,2)会生成临时对象，不会调用复制构造函数
```

（3）函数按值传递对象时会调用复制构造函数且会生成临时对象。比如 class A{}; void f(A ma){} A ma1; f(ma1);，调用函数 f 时，会调用类 A 的默认复制构造函数创建一个 ma1 的副本，并把该临时对象赋值给形参 ma。

（4）从函数返回一个该类型的对象时会调用复制构造函数且会生成临时对象。

4．自定义复制构造函数

（1）复制构造函数就是带一个该类类型引用形参的构造函数，因此完全可以自定义复制构造函数。

（2）可以在类中定义两个复制构造函数，但形参表必须不相同，调用时遵守形参为引用时的规则，详见 7.4.4 节的第 3 点。比如 class A{public: A(A& ma){} A(const A& ma){}};，其中 A(A &ma)和 A(const A &ma)是可以同时存在的。

（3）不可以在类中定义该类类型的非引用单形参构造函数。比如 class A{public: A(A ma){}};，其中 A(A ma)是错误的。

（4）与默认构造函数一样，自定义复制构造函数后，系统就不会自动生成默认的复制构造函数了，因此一般都要再定义自己的构造函数；否则创建对象时，会无可用的构造函数。

（5）自定义复制构造函数时通常应使用带 const 的引用形参，且不应设置为 explicit 的。因为复制构造函数一般用于传递和返回对象，所以不适合设置为 explicit 的。

（6）注意：对于复制构造函数，A ma=A(ma1)与 A ma=ma1;是等价的，若使用 explicit 对复制构造函数进行限制，则这两条语句都无法使用。

（7）若自定义的复制构造函数什么都不做，则复制初始化时被初始化的对象的数据成员没有初始化。比如 class A{public:int a; A(){a=1;} A(const A& ma){}}; A ma; A ma1(ma);，则 ma1.a 是未被初始化的，将得到一个随机值。

（8）何时需要自定义复制构造函数：类的数据成员有指针或者有分配的其他资源时，一般需要重新定义复制构造函数。因为这时默认复制构造函数在复制初始化对象时，只是将指针的地址简单地赋给新对象里的指针成员，新对象和旧对象的指针成员都指向相同的内存，改变其中一个就会对另一个产生影响，若其中一个指针被析构函数删除了，另一个指针依然指向被删除了的内存，从而产生悬垂指针。比如 class A{public: int *p; int b; A(){b=1; p=&b;} }; A ma; A ma1(ma);，这时 ma 和 ma1 中的指针成员 p 指向同一个对象，当其中一个发生改变时，比如 *ma.p=2;，这时 ma1 中的指针成员所指向的值也相应地变为 2。

（9）自定义复制构造函数时，若类中有指针成员，则可以使用 C++自带的智能指针类，也可以自定义智能指针类，当然最简单的办法就是使用复制构造函数复制指针指向的值，而不是复制指针的地址。比如 class A{public: int *p; int b; A(){b=1; p=&b;} A(const A& ma);};，复制构造函数的实现方法为 A::A(const A& ma):p(&b){*p=*ma.p; b=ma.b; }，其中*p=*ma.p 复制的是

指针所指向的值。注意：应先在成员初始化表中使用 p(&b)初始化指针成员，否则指针是未初始化的。

5. 自定义赋值运算符

（1）一般来讲，需要重定义复制构造函数时，就需要自定义赋值运算符；相反，需要自定义赋值运算符的地方，一般也需要自定义复制构造函数。

（2）若自定义的赋值运算符不是对默认赋值运算符的重定义，则默认赋值运算符依然是存在的。比如 class A{public: void operator =(int i){}}; A ma, ma1;，则 ma=ma1;仍会使用默认赋值运算符进行赋值；再比如 class A{public: void operator =(A&){}};A ma, ma1;，则 ma=ma1;会调用自定义的赋值运算符，因为自定义的赋值运算符是对默认赋值运算符的重定义。

（3）若类中拥有指针成员，则自定义赋值运算符的解决方法与自定义复制构造函数相同。

（4）自定义复制构造函数和赋值操作符的难点在于，需要知道何时自定义它们。

（5）禁止使用复制构造函数和赋值运算符的方法为：将其设置为私有的，此时友元和成员函数仍可访问。要彻底禁止使用的方法就是将其设置为私有的，同时只声明而不定义函数。在类中只声明而不定义一个函数是合法的，但在使用该函数时就会出现错误，即当友元或成员函数使用到未定义的函数时会产生链接错误。比如 class A{private: A(const A&); }，这时类 A 的复制构造函数就彻底无法被使用。

（6）C++自动提供了默认构造函数、默认复制构造函数、默认析构函数、默认赋值操作符、地址操作符即 this 指针，这 5 种实体如果用户没有定义，则系统会自动创建。

第 10 章

名称空间专题

名称空间又称为名字空间、命名空间。

关于链接性、作用域、源文件、头文件、局部作用域内的名称解析的内容，请参阅第 7 章。

10.1 名称空间基础

1. 名称空间基本规则

（1）名称空间的含义：即提供一个声明名称的区域。一个名称空间中的名称不会与另一个名称空间中相同的名称发生冲突。

（2）创建名称空间的语法形式：namespace 名称空间名{...}。

示例如下：

```
namespace N{int a; double b;}//声明了一个名为“N”的名称空间，该名称空间内声明了两个成员
```

① 名称空间使用关键字 namespace 后接名称空间名字创建。

② 注意：在创建名称空间的大括号后不需要分号。

③ 名称空间成员：名称空间中声明的名称。

（3）可以在名称空间中放入任何在全局作用域中出现的声明和定义。注意：类似于 a+2;这样的表达式语句不能出现在名称空间中(全局作用域中也不能出现这样的语句)。比如 namespace N{int a; a=1;}，其中 a=1;是错误的。

（4）名称空间不能定义在函数或类的内部，但可以在全局作用域或其他作用域中定义（比如一对大括号中），也可以在另一个名称空间中定义（即嵌套的名称空间）。比如 class A{namespace N{int a; } }; 错误，因为名称空间定义在类的内部了。

（5）名称空间不可以只声明而不定义。比如 namespace N; 错误。

（6）名称空间中的名称必须是唯一的（函数重载除外），但不同的命名空间中可以有同名的名称，这与常规作用域中名称的规则相同。比如 namespace N{int a; void a(){}}错误，因为名称 a 重定义了。

（7）一个名称空间其实就是一个命名的作用域，因此也把名称空间称为名称空间作用域。在名称空间中声明的名称被隐藏在该名称空间作用域中，因此名称空间中的名称不会与全局作用域中声明的相同名称发生冲突。比如 int a; namespace N{int a;} 正确，因为名称空间 N 中的名称被隐藏在该空间内部，与全局作用域中的名称 a 不会发生冲突。

（8）重要规则：名称空间可以追加定义。也就是说，可以把名称空间分为多个部分，它可以是不连续的，这时后一个同名的名称空间的内容将累（追）加到前一个名称空间中。比如 namespace N{int a=1;} namespace N{int b=a;} 相当于 namespace N{int a=1; int b=a;}。注意：namespace N{int a=b;} namespace N{int b=1;}是错误的，因为 b 的声明位于 a 之后。

（9）在名称空间中声明的成员是具有外部链接性的、静态存储持续期的全局名称（未命名的名称空间除外）。因此和全局变量一样，若只声明而不定义一个变量，则应使用 extern 关键字进行说明。比如 namespace N{int a; void f(){cout<<a; a++;}}，则 N::f(); N::f();输出 01，可见名称空间中的成员 a 是具有静态存储持续期的，若未初始化则默认值为 0。

（10）最常使用的名称空间是 std，即标准名称空间，C++将所有的名称都包含在该名称空间中，包括 cout、cin、string 等名称。这也是为什么在程序中会有 using namespace std;的原因。关于 using namespace 指令见后文。

2．名称空间位于多个文件中

（1）因为名称空间可以追加，所以同一个名称空间可以位于多个不同的文件中，此时名称空间由分离的各部分组成，要访问其他文件中相同名称空间内的名称则应对该名称进行声明，即对于变量应使用不带初始化的 extern 语句。比如在文件 1 中有 namespace N{int a; int b; void g(){}}，在文件 2 中有 namespace N{extern int a; void g(); void f(){a=1; g(); b=2}}，其中 b=2 是错误的，因为 b 未声明。注意：不存在 extern int N::a;这样的声明语句。

（2）因为名称空间具有外部链接性，所以不能在多个源文件的相同名称空间中定义相同的成员（未命名的名称空间除外），否则在链接阶段会出现重定义错误。比如在 文件 1 中有 namespace N{void g(){}}，在文件 2 中有 namespace N{void g(){ }}，则将出现重定义错误。

（3）最好不要把变量的声明或定义及函数的定义（内联函数除外）放在头文件的名称空间中，否则在多个文件中使用#include 把头文件包含进来时就会出现同一个变量多次定义的错误。

（4）可以在头文件的名称空间中包含函数的声明，而在其他文件中定义该函数。

3．在名称空间外定义名称空间中的成员

（1）在名称空间外定义名称空间中的成员，其方法与在类外定义成员函数相同，都是使用作用域解析运算符来限定其名称的。比如 namespace N{void g(); class A;}，则在外面定义函数的方法为 void N::g(){}；定义类的方法为 class N::A{}，如果不使用作用域解析运算符，则表示在名称空间外定义了一个与名称空间中同名的函数或类。

（2）重要规则：8.3.1 节中介绍的规则同样适用于名称空间成员，即一旦被定义成员的完全限定名出现，则直到该成员定义结束，该定义都位于类体（名称空间作用域）中，也就是相当于此成员是在类体中定义的。所以直到该成员定义结束，都可以直接使用类体中的其他成员，而无须使用作用域解析运算符。

（3）不能在不相关的作用域中对名称空间的成员进行定义，即名称空间中成员的定义只能出现在该名称空间的外围作用域之中（包括全局名称空间）。比如 namespace N{void f();} class A{void N::f(){}}; namespace M{void N::f(){}}，其中在类作用域 A 和名称空间作用域 M 中对 N 中的成员 f 的定义都是错误的。

10.2　名称空间的分类

1. 嵌套的名称空间

（1）嵌套的名称空间：即在名称空间中定义的名称空间。比如 namespace N{ namespace kk{int a;}…}，其中 kk 就是嵌套的名称空间。

（2）因为一个名称空间就是一个作用域，因此嵌套的名称空间会形成自己的作用域，其中的名称被隐藏在嵌套的名称空间中，外围的名称空间看不到嵌套名称空间中的名称，同名的嵌套名称空间中的名称会隐藏掉外围作用域中的同名名称。

（3）在嵌套的名称空间外对成员的定义方式与常规名称空间相同，只是多了几层名称限定。

（4）不能在嵌套的名称空间外对嵌套的名称空间进行定义或追加。比如 namespace N{ namespace N1{}}，则 namespace N::N1{}是错误的。

2. 未命名（无名）的名称空间

（1）名称空间可以是无名的，定义该名称空间时只要不指定名称即可。比如 namespace {int a;}就是未命名的名称空间。

（2）未命名的名称空间中的名称具有内部链接性、静态存储持续期，因此具有如下特点。

① 未命名名称空间中的名称只在当前文件中可见，不能跨越多个文件。即使对该名称进行了声明，也不能引入其他文件中未命名名称空间中的名称。比如在文件 1 中有 namespace {int a;}在文件 2 中有 namespace{extern int a; void f(){a=1;}}，其中 a=1;是错误的，extern int a;并不能把文件 1 中具有内部链接性的名称 a 包含进来。

② 不同文件中未命名名称空间中的相同名称不会存在重定义错误。比如在文件 1 中有 namespace {int a=1;}，在文件 2 中有 namespace{int a=2; void f(){cout<<a<<endl; }，这是正确的，函数 f 将输出文件 2 中变量 a 的值。

③ 未命名名称空间中的名称在程序结束之前一直存在，因为具有静态存储持续期。

（3）未命名的名称空间同样可以追加定义，但仍然不能跨越文件。

（4）未命名名称空间中的名称可以直接被使用，因为名称空间没有名称，所以无法使用作用域解析运算符，使用 using 等对其进行限定。比如 namespace {int a=1;} void f(){cout<< a<<end1;}，在函数 f 中对 a 的访问是正确的。

（5）未命名名称空间中的名称可以在定义该名称空间所在的作用域中找到，所以在该作用域中定义与未命名名称空间中相同的名称会出现名称冲突。这种冲突不是错误，错误只发生在对该名称的使用具有二义性时，若使用作用域解析运算符来明确调用不是未命名名称空间中的相同名称，则不会出错。比如 namespace M{ namespace {int b;} int b; void f(){b=1; M::b=2;}}，则在函数 f 中的 b=1 是错误的，但 M::b=2 是正确的。

（6）未命名的名称空间可以代替全局静态变量，因此 C++推荐不在文件中使用全局静态变量，取而代之的是使用未命名的名称空间。

3．名称空间的别名

（1）名称空间的别名主要用于简化名称很长的名称空间。

（2）创建名称空间别名的语法（namespace 赋值语法）：namespace 别名=原名称空间名。比如 namespace N=hyong 表示把名称空间 hyong 重命名为名称 N。

（3）使用 namespace 赋值语法只能对名称空间进行重命名，不能对名称空间中的成员进行重命名。比如 namespace N{int a;}，则 namespace m1=N::a;错误。

（4）名称空间的别名可用于简化嵌套的名称空间，比如 namespace N=hh::jj::m。

（5）可以在使用名称空间的地方使用名称空间的别名。比如 namespace N{int a;} namespace M=N;，则 void f(){M::a=1;}中的 M::a=1 与 N::a=1 等同。

（6）不能使用别名对名称空间进行追加定义。比如 namespace N{} namespace M=N;，则 namespace M{int b;}是错误的。

（7）名称空间可以有多个别名。比如对于 namespace N{}，namespace m1=N;、 namespace m2=N; 和 namespace m3=m2;都是正确的。

（8）使用名称空间的别名时，应将其代替为名称空间的名称后进行分析。比如 namespace N{ int a; namespace M{int b;}} namespace m1=N; namespace m2=N::M; namespace m3=N;，则 m1::M::b=N::M::b;正确，m1::m2::a=N::N::M::a;错误，m1::m3::b=N::N::b;错误，m1::m3::a=N::N::a;错误。

10.3　访问名称空间中的名称

在名称空间内部，可以直接使用该名称空间中声明的名称。

1．使用作用域解析运算符访问名称空间中的成员

其方法就是在要访问的名称前使用名称空间名和作用域解析运算符进行限定修饰。比如有名称空间 namespace N{int a;}，则 N::a=3;表示访问名称空间 N 中的成员 a。

2．使用 using 声明访问名称空间中的成员

（1）使用 using 声明的形式：using N::a，表示把名称空间 N 中的名称 a 引入到当前作用域中。

（2）使用 using 声明可以把名称空间中的名称引入到当前作用域中，在当前作用域中，就不需再使用作用域解析运算符对该名称进行限定了（当然也可继续使用）。比如 namespace N{int a;} void f(){using N::a; a=2; N::a=4; }，其中 a=2 是正确的，指的是 N::a；同理，N::a=4;也是正确的。

（3）使用 using 声明一次只能引入名称空间中的一个名称，要引入多个名称就需要使用 using 多次声明。比如 namespace N{int a; int b; int c; } void f(){using N::a; using N::b; a=1; b=2; c=3;}，其中 c=3;是错误的。

（4）使用 using 声明引入的名称从 using 声明处开始，直至 using 声明所在的作用域结束都是可见的。比如 namespace N{int a;} void f(){ a=2; using N::a; a=3;} int b=a;，其中 a=2;是错误的，因为在使用 using 声明之前，名称空间 N 中的名称 a 是不可见的；int b=a;也是错误的，因为使用 using 声明引入的名称在该声明所在作用域外是不可见的。

（5）使用 using 声明引入的名称的作用域：使用 using 声明把名称空间中的名称引入到当前局部作用域中，就像所引入的名称在当前作用域中声明的一样。使用 using 声明引入的名称遵守“在外围作用域中的声明会被嵌套作用域中的同名声明所隐藏”规则，因此使用 using 声明引入的名称具有以下特点。

① 在相同作用域中定义的名称不能与使用 using 声明引入的名称相同，即使用 using 声明引入的名称必须是唯一的。比如 namespace N{int a;} void f(){int a=1; using N::a; }错误，因为重定义了名称 a。

②使用 using 声明引入的名称会隐藏外围作用域中的相同名称。比如 namespace N{int a=1;} int a=2; void f(){using N::a; cout<<a<<endl;}，则函数 f 输出 1，即输出的是 N::a 的值。

③ 在嵌套的作用域中声明的相同名称会隐藏使用 using 声明引入的名称。比如 namespace N{int a=1;} using N::a; void f(){int a=2; cout<<a;}，则函数 f; 输出 2，即输出的是该函数中名称 a 的值。

3. 使用 using 指令访问名称空间中的成员

（1）using 指令语法：using namespace 名称空间名;。比如 using namespace N;表示把名称空间 N 中的所有名称引入到当前作用域中。

（2）使用 using 指令可以把名称空间中的所有名称都引入到当前作用域中，在当前作用域中，就不需要再使用作用域解析运算符对名称进行限定了（当然也可继续使用）。比如 namespace N{int a; int b;} void f(){using namespace N; a=1; b=2; N::a=3;}，其中 a=1; b=2;都是正确的，此时无须再使用作用域解析运算符了；N::a=3;也是正确的。

（3）using 指令作用域：使用 using 指令引入的名称从 using 指令处开始，直至该指令所在的作用域结束都是可见的。比如 namespace N{int a;} void f(){ a=2; using namespace N; a=3;} int b=a;，其中 a=2;错误，因为在 using 指令之前，名称空间 N 中的名称 a 不可见；int b=a;也是错误的，因为 using 指令引入的名称在该指令所在作用域之外不可见。

（4）使用 using 指令引入的名称的作用域（引入作用域）。

① 使用 using 指令把名称引入到定义该名称空间及该 using 指令的外围作用域（为方便讲解以下简称“引入作用域”）中。比如 namespace N{int a;} void f(){using namespace N;} ，则 using 指令的引入作用域为定义名称空间 N 所在的作用域（本例为全局作用域）和 using 指令的外围作用域（本例为函数 f 的作用域，即全局作用域），因此 using 指令的引入作用域是全局作用域。

② 在 using 指令的作用域之外的其他作用域中无法引用引入作用域中的名称，即引入作用域只对 using 指令的作用域有效，对其他作用域没有影响。比如 namespace N{int a;} void f(){ a=2; using namespace N; a=3;} int b=a;，该引入作用域为全局作用域，但对 a=2 和 b=a 的引用都是错误的，因为它们都不在 using 指令的作用域内，因此只有 a=3;是正确的。

③ 若在位于引入作用域内的相同作用域中定义的名称与使用 using 指令引入的名称相同，则在 using 指令的作用域内使用该名称会产生二义性错误，若使用不具有二义性则不会产生错误（比如使用作用域解析运算符进行调用）。在其他作用域中使用该名称不受影响。

④ 若在引入作用域的嵌套作用域中定义的名称与使用 using 指令引入的名称相同，则该名称会隐藏引入作用域中的名称。

⑤ 在 using 指令的作用域中定义的名称会隐藏使用 using 指令引入的名称，因为 using 指令的作用域嵌套于引入作用域内。

示例 10.1：using 指令的作用域

```
//using 指令的引入作用域是全局的，using 指令把名称 a,b,c 引入到全局作用域中
namespace N{int a;int b; int c;}
int a=1;//此名称会与使用 using 指令引入到引入作用域中的相同名称产生冲突（因为它们位于同一个作用域中）
void f(){int b=11;    //此名称会隐藏使用 using 指令引入到引入作用域中的相同名称（因为此处的作用域嵌
                      //套于引入作用域内）
    cout<<a<<endl; //正确，使用全局名称 a，引入作用域只对 using 指令的作用域有效
```

```
    using namespace N; //using 指令的作用域开始
    //a=3;             //二义性错误，引入作用域中的名称 a 和全局作用域中的名称 a 产生冲突，引入作用
                       //域对 using 指令的作用域有效
    ::a=3;             //正确，使用作用域解析运算符明确地调用全局作用域中的名称
    cout<<b;           //输出函数 f 中的名称 b
    int c=3;           //在 using 指令的作用域内定义的名称会隐藏使用 using 指令引入到
                       //引入作用域中的相同名称（因为 using 指令的作用域嵌套于引入作用域内）
    }                  //using 指令的作用域结束
void main(){  a=4; //正确，使用全局名称 a，引入作用域只对 using 指令的作用域有效
    f();}
```

4．using 声明和 using 指令的共同特点及相互关系

（1）在相同作用域中，若使用 using 声明和 using 指令引入了相同的名称，则使用 using 声明引入的名称将隐藏使用 using 指令引入的名称，因为它们的作用域互为嵌套关系。比如 namespace X{int a=1;} namespace Y{int a=2;} void f(){using namespace X; using Y::a; cout<<a;};，则函数 f 会输出 2，其中的 a 是 Y 中的 a，即与 Y::a=2 相同。

（2）using 指令和 using 声明对重定义的处理：使用 using 指令产生的二义性错误需要等到使用该名称时才会被检测出来，使用 using 声明产生的错误在该声明处会被立即检测出来。比如 namespace N{int a=1; } void f(){using N::a; int a;}，则程序会立即产生重定义错误；再比如 namespace N{int a=1;} int a=2; void f(){using namespace N;}，程序不会出现错误，因为该程序没有使用名称 a。

（3）using 声明和 using 指令可以出现在全局作用域、名称空间作用域和局部作用域中。但在类作用域中不能使用 using 指令引入名称空间中的名称，只能使用 using 声明对父类中的名称进行引入（详见第 13 章）。比如 namespace N{int a;}，则 class A{using N::a; using namespace N;}; 中的两条 using 语句都是错误的。

（4）using 语句和作用域解析运算符混合使用时的注意事项（用示例进行说明）。

```
namespace Y{int a;} namespace X{int b;}
namespace N{using X::b; using namespace Y;}
void f(){ X::b; Y::a;        //正确
   N::a=1; N::b=2;           //正确，因为 a 和 b 在名称空间 N 中可见
   //N::X::b; N::Y::a;       //错误，因为 X 和 Y 不是 N 的成员
}
```

5．使用 using 声明和 using 指令引入函数名称

（1）注意：函数重载的前提是“重载的函数应在相同的作用域中（带类类型形参的函数除外）”。

（2）不能使用 using 声明引入重载函数的某个特定版本，因为它会把该函数的所有重载版本都引入进来。也就是说，在使用 using 声明引入函数时，函数名后不能有小括号。比如 using N::g();是错误的，正确的为 using N::g，表示把函数 g()的所有重载版本都引入到当前作用域中。

（3）若使用 using 声明引入的函数的形参表与当前作用域中同名的函数不相同，那么这些函数与当前作用域中的函数会形成重载。比如 namespace N{void g(){} void g(int i){}} void g(int,int){} using N::g;，其中 using N::g;会把函数 g()和其重载版本 g(int)都引入进来，并与全局的 g(int,int)形成重载。

（4）若使用 using 声明引入的函数的形参表与该作用域中同名的函数相同，则产生重定义错误。比如 namespace N{void f(){} void f(int){}} using N::f; void f(){};错误，因为函数 f 重定义了。

（5）若 using 声明在局部作用域中，则名称空间中的函数会屏蔽全局（或外层）作用域中的同名函数，因此候选函数集中只有名称空间中的函数。

（6）使用 using 指令引入函数名称：

① 使用 using 指令会把函数的所有重载版本都引入到引入作用域中，因此引入的函数会与引入作用域中的函数形成重载。若在引入作用域中定义了相同的函数，则会产生二义性错误，而且这种二义性错误同样可以使用作用域解析运算符进行明确的限定来消除。

② using 指令作用域中的同名函数会隐藏使用 using 指令引入进来的所有函数重载版本(因为它们的作用域为嵌套关系)。

示例 10.2：使用 using 指令引入函数名称

```
namespace N{void f(){} void f(int){} void g(){} void g(int){}}
//使用 using 指令把名称 f 和 g 引入到全局作用域中
void g(){}   //使用 using 指令时，会与 N 中的名称 g 产生冲突，同时会与 N 中的 g(int) 形成重载
namespace M{
    using namespace N;
    void f(){} //using 指令作用域中的同名函数隐藏了使用 using 指令从 N 中引入的 f() 和 f(int)
               //因为 using 指令作用域嵌套于引入作用域中
    void h(){//f(3);    //错误，原因见上
    N::f(3);            //正确，可以使用作用域解析运算符调用被隐藏的名称
    N::f();             //正确，原因同上。
    g(3);               //正确，使用 using 指令引入的名称与全局函数 g 是重载关系
    //g();              //产生二义性错误
    N::g();                  //正确，使用作用域解析运算符明确调用 N 中的函数 g，可以消除二义性错误
    ::g(); } }               //正确，调用全局函数 g，原因同上
void main(){  M::h();}
```

10.4 名称空间中的名称解析

（1）名称解析分散于各个章节中，本节只讲解与名称空间有关的名称解析。

（2）对名称空间中的名称解析（查询）遵守常规的名称查找（见 7.5.1 节）规则，即从当前声明点逐层向外围作用域中查找名称，在查找时只考虑成员定义之前的名称声明。虽然名称空

间是不连续的，但仍遵守这条规则。

(3)成员定义在名称空间外的名称查找规则与类作用域中的名称查找类似(详见 8.3.2 节)，即首先在包含该成员函数的名称空间中查找所有的成员声明（当名称空间不连续时，只查找使用该名称之前的名称)，若未找到，则在此成员函数定义之前的作用域中进行查找。

示例 10.3：名称空间中的名称解析

```
#include<iostream>
using namespace std;
namespace M{int a3=3; int a2=22;
   namespace N{int a1=1;    void g();  int a2=2; }
   tint a4=4; }
int a5=5;
namespace M{int a5=55;}
void M::N::g(){
   cout<<a1;              //正确
   cout<<a2<<endl;        //正确，因为 g 定义于名称空间之外，因此从名称 g 所在的名称空间 N 开始查找找到
   //N::a2，此处不会找到 M::a2
   cout<<a3<<endl;        //正确，从名称空间 N 开始查找，若未找到 a3，则查找外围名称空间 M，找到 a3
   cout<<a4<<endl;        //正确，同上
   cout<<a5<<endl;        //首先在 N 中查找，若未找到，则在定义于 g 之前的第二个追加的名称空间 M 中查找，
                          //找到 a5，调用 M::a5 而非全局::a5
   //cout<<a6<<endl;      //错误，a6 在成员函数 g 的定义之后
   //cout<<a7<<endl;      //错误，同上
   cout<<::a5<<endl; } //显式调用全局变量 a5
int a6=6;
namespace M{int a7=7;}
void main(){M::N::g(); }
```

示例 10.4：名称空间成员、前置声明和名称隐藏

```
class A{};
void f(int){}
namespace N{
   class B{};
   void f(int,int){}
   namespace M{
      void f();     //声明名称空间 M 的成员函数 f，而非对全局函数或 N 中的函数 f 的前置声明，此名称会隐
                    //藏全局名称 f(int)和 N 中的 f(int,int)，它们不会形成函数重载
      class A;      //声明名称空间 M 的成员类 A，这不是对全局类 A 的前置声明
      class B;      //声明名称空间 M 的成员类 B，这不是对名称空间 N 中的类 B 的前置声明
      class C;      //声明 M 的成员类 C，该类定义于名称空间 N 中
      void g(){//f(3); //错误，全局的 f(int)被隐藏了
         //f(3,3);      //错误，N 中的 f(int,int)被隐藏了
         //B mb;        //错误，使用在 M 中还未被定义的 B 创建对象，此处不会使用 N 中的 B 创建对象
         //A ma;        //错误，同上，A 并不是指全局的类 A
         }}             //名称空间 M 结束
      class C{};  //这不是对 M 中的类 C 进行的定义，而是定义了一个名称空间 N 的成员类 C
```

```
        class M::C{};  //定义名称空间M中的类C成员
        class M::B{}; //在N中对M中的类B进行定义
}
void main(){N::M::g();}
```

第 11 章

类中的成员专题

本章所提到的数据成员、成员变量、成员函数等含有“成员”一词的概念都是指在类中声明的成员。为简便起见，有时会把类的数据成员直接说成数据成员或成员。

11.1 静态成员

11.1.1 静态数据成员

1. 静态数据成员基础及特点

（1）静态数据成员的声明：要声明一个静态数据成员，只需在数据成员的前面加上 static 关键字即可。比如 class A{static int b;};，其中 b 就是一个静态数据成员。

（2）静态数据成员属于整个类，而不属于某个对象，因此其具有以下特点。

① 静态数据成员被类的所有对象共享，每个类对象都只有静态数据成员一份共同的副本，因此该类及其所有对象使用的都是同一个静态数据成员。若静态数据成员的值发生了改变，则对所有该类的对象而言，该值都被改变了。比如 class A{public:static int b;}; int A::a=3; A ma, mb;，若 ma.b=1;，则 ma.b;的值也会改变为 1。静态变量 b 属于整个类，被类的所有对象共享，因此 ma.b 和 mb.b 都是指相同的静态变量 b，使用 ma.b 改变 b 的值，会使 ma.b 的值也发生改变。int A::a 是对静态数据成员的定义。

② 即使不创建任何类的对象，但是只要定义了静态数据成员，程序就可以使用静态数据成员，因为静态数据成员属于类而不是对象。比如 class A{public: static int a; }; int A::a=3;，则 A::a=4; 对静态数据成员 a 重新赋值是正确的。其中 int A::a=3;是对静态数据成员的定义，A::a 是使用类名引用静态数据成员的方法。

③ 创建对象只会为对象的非静态数据成员分配内存，而不会为静态数据成员分配内存，因此在类中只是对静态数据成员的声明而非定义，若未定义静态数据成员就直接使用，将产生错

误。比如 class A{public: static int a; int b;}; A ma;，则 ma.a=3;是错误的，因为创建 ma 后只会为非静态数据成员 b 分配内存，而不会为静态数据成员 a 分配内存。

④ 静态数据成员的类型可以是静态数据成员所属的类类型，而非静态数据成员则不可以。比如 class A{public: static A a; A b;};，其中 static A a;是正确的，而 A b;则是错误的。

⑤ 静态数据成员可以作为默认实参，而非静态数据成员则不可以，因为非静态数据成员的值在被使用之前必须先绑定到该类的某个对象（或对象的指针）上，因此非静态数据成员无法提供对象，从而得到该成员的值。所以默认实参不能是类的非静态数据成员。比如 class A{public: static int a; int b; void f(int i=a){} void g(int i=b){}};，其中函数 f 是正确的，函数 g 是错误的，因为使用了非静态数据成员 b 作为默认实参。

（3）静态数据成员可以是 public（公有的）、private（私有的）、protected（受保护的）。

（4）静态数据成员也可以是类类型对象、数组、const 常量、引用等。

（5）在成员函数中可以像使用普通数据成员一样使用静态数据成员，只不过要注意静态数据成员属于整个类这个特性。

2. 访问静态数据成员

（1）可以使用类名和作用域解析运算符直接访问静态数据成员。比如 class A{public: static int b;};，则 A::b 是正确的访问方式。

（2）可以像普通数据成员那样使用类的对象，或者指向该类对象的指针、引用来访问静态数据成员。比如 class A{public: static int b;}; A int::b=3; A ma; A*p=&ma;，则 ma.b; 和 p->b;都正确。

3. 静态数据成员的定义

（1）静态数据成员必须在类外进行定义且只能定义一次，在类内只是对静态数据成员的声明。静态数据成员的定义方式为使用作用域解析运算符指明它来自哪个类，但在定义时不能再次指定 static 关键字。比如 class A{public:static int a; };，其定义方式为 int A::a=2;或 int A::a;，前者表示定义 a 并初始化为 2；后者表示只是定义 a 但不初始化（会自动初始化为 0）。

（2）若未对静态数据成员进行定义而直接使用，将发生错误。比如 class A{public: static int a; A(){} A(int i){a=3;} void f(){a=2;}}; A ma; A ma1(2); ma.f(); 其中 A ma1(2);和 ma.f();是错误的，因为在这两个地方都使用了未定义的静态数据成员 a；而 A ma;是正确的，因为此时未使用未定义的静态数据成员。

（3）静态数据成员必须在该类及其嵌套父类外进行定义，这里的类外是指该类所在的外围作用域，在类外的非父作用域内进行定义是错误的。例如：

```
class A{public:static int a; }; void f(){int A::a=2;}//错误，函数 f 内不是类 A 的父作用域
class A{public:static int a; }; namespace N{int A::a=3;}//错误，N 不是类 A 的父作用域
namespace N{class A{public:static int a; }; int A::a=3;}//正确
```

```
namespace N{class A{public:static int a; };} int N::A::a=3;// 正确
class A{public: class B{public: static int a;}; int B::a=3; };//错误，不能在嵌套父类中定义
```

（4）定义静态数据成员时的作用域规则：详见 8.3.1 节，比如 class A{public: typedef int IN; static IN a; static int f(){return 2;} };，此规则主要是为了解决在定义 a 时，即 A::IN A::a=f();，基中 A::IN 和 f()函数形参及函数体内是否需要使用作用域解析运算符的问题。

4．静态数据成员的初始化

（1）静态数据成员是在定义时进行初始化的，若定义时未对静态数据成员赋初值，则自动初始化为 0。比如 class A{public:static int b; static int c;}; int A::b; int A::c=2;，其中定义 b 时未对其进行初始化，其值自动初始化为 0，而 c 在定义时被初始化为 2。

（2）静态数据成员不是使用构造函数进行初始化的，在构造函数内部进行的是赋值而不是初始化，也不是定义。虽然在构造函数内进行了赋值，但静态数据成员还是必须在类外进行定义。比如 class A{public: static int a; A(){a=3;}};，则 A ma;错误，因为未对静态数据成员 a 进行定义就在构造函数内对 a 进行了赋值，因此语句 a=3;是错误的。

（3）不能在成员初始化表中对静态数据成员进行初始化。比如 class A{public: static int a; A():a(1){}};，则 A():a(1){}是错误的。

（4）同其他数据成员一样，在声明时不能对静态数据成员进行初始化。比如 class A{public:static int a=3;};错误。

示例 11.1：静态数据成员的声明、定义和初始化

```
#include <iostream>
using namespace std;
class A{public:
   static int a;          //声明一个静态数据成员 a，此处只是对 a 进行了声明，并未定义
   static A a1;           //正确，静态数据成员的类型可以是所属的类类型
   //A a2;                //错误，非静态数据成员的类型不可以是所属的类类型
   static int a3;
   //static int a4=4;     //错误，声明时不能对静态数据成员进行初始化
   void f(int i=a) {}     //正确，静态数据成员 a 可以作为函数的默认实参
   //A():a(3){}           //错误，不能在成员初始化表中对静态数据成员进行初始化
   A(){ } };              //类 A 结束
//以下内容为静态数据成员的定义及初始化
int A::a;                 //静态数据成员必须在类外进行定义，此处自动初始化为 0
//int A::a=2;             //错误，静态数据成员只能在类外进行一次定义
//static int A::a3;       //错误，定义静态数据成员时无须再次指定 static 关键字
int A::a3=3;              //正确，a3 被初始化为 3
class B{public: static int a; B(){}
   B(int i){a=3;}         //若 a 未在类外进行定义，则可能发生错误，此处并非是对 a 的初始化，也并
                          //非定义，而只是赋值。
   void f(){a=2;} };      //若 a 未在类外进行定义，则可能发生错误。类 B 结束
void f(){ //int B::a=3; //错误，在类外的非父作用域内对静态数据成员进行定义将发生错误
```

```
}
namespace N{
    class A{public: static int a; static int b; };
    int A::a=3;}          //对a的定义正确
int N::A::b=3;            //对b的定义正确
//main函数开始
void main(){
    cout<<A::a<<endl;     //输出0，静态数据成员可以通过类名和作用域解析运算符直接进行访问
    A ma;                 //调用默认构造函数
    cout<<ma.a<<endl;     //输出1，静态数据成员也可通过类的对象或指向对象的指针进行访问
    A ma1; ma1.a=3;
    cout<<ma.a<<endl;     //输出3，静态数据成员被类的所有对象共享，每个类对象都只有该类静态数据
                          //成员的一份共同的副本，若静态数据成员的值发生改变，则对所有该类的对象
                          //而言，该值都被改变了
    cout<<ma1.a<<endl;    //输出3
    B mb;                 //正确。注意：类B中的静态数据成员a未被定义
    //B mb1(2);           //错误，在构造函数内使用了未定义的静态数据成员a
    //mb.f();             //错误，函数f使用了未定义的静态数据成员a
    //mb.a=3;         //错误，使用了未定义的静态数据成员a，虽然创建了对象，但静态数据成员a并未定义
}
```

5. 静态常量数据成员

（1）静态常量整型数据成员可以在声明时进行初始化。此条规则只适合于 int、char、bool 这种具有整型性质的类型，对于 float、double 类型的静态常量是不能在声明时进行初始化的。比如 class A{public: const static int a=2; const static float b=1;};，其中 const static int a=2; 是正确的；const static float b=1;是错误的。

（2）虽然静态常量整型数据成员在类体中声明时进行了初始化，但仍必须在类体外对其再进行一次定义，且只能进行一次定义，定义时不需要再次指定初始值，若指定了初始值则将发生错误，定义时也不能再次指定 static 关键字。比如 class A{public: static const int a=3;}，在类体外定义的方式为 const int A::a;，而 const int A::a=3;是错误的，不能再次指定初始值。

（3）对于具有非整型性质的静态常量数据成员，其初始化的唯一机会是在类体外进行定义时（因为在构造函数内是赋值，而静态成员又不能在成员初始化表中初始化）。对于 const 常量必须进行初始化，因此即使静态常量数据成员的值为 0，也必须显式进行初始化。比如 class A{public: const static float a;};，初始化 a 的方法为 const float A::a=0;，即使值为 0 也必须显式进行初始化，而 const float A::a;试图把 a 自动初始化为 0 是错误的。

（4）静态常量数据成员可以作为编译时常量使用（编译器有可能不支持），这意味着静态常量整型数据成员可以用来指定数组的大小，但对常量的初始化工作必须在定义该数组之前。比如：

```
class A{public: static const int a=3; int b[a]};//正确
class A{public: static const int a; int b[a];}; const int A::a=3; //错误
```

```
class A{public: static const int a; }; const int A::a=3; int b[a]; //正确
```

6. 其他静态数据成员

（1）静态对象成员：必须在类外进行定义，定义时与静态数据成员相同。类外是指该类所在的外围父作用域。初始化该成员的方式与初始化普通类对象相同，但在类中声明静态对象成员并试图初始化时将发生错误，同理，只声明不定义静态对象成员也不能为该成员分配内存。比如 class B{public: B(){} B(int i){}} ; class A{public: static B mb;};，其定义方式为 B A::mb;、B A::mb(3);、B A::mb=B(3);、B mb1; B A::mb(mb1); 或 B A::mb=mb1;。注意：class A{public: static B mb(9);}; 是错误的，在类中只能声明静态对象成员，而不能初始化。

（2）静态数组成员：也必须在类外进行定义及初始化，因为定义时必须在全局作用域内，因此初始化静态数组成员时就无法使用循环进行初始化了。

11.1.2　静态成员函数

1. 基础

（1）与静态数据成员一样，静态成员函数也是属于整个类而不是某个对象。

（2）静态成员函数的调用与静态数据成员相同，可以通过类名、对象以及指向对象的指针、引用进行调用。

（3）声明静态成员函数只需在函数声明前加上 static 关键字即可，比如 class A{static void f();};。

（4）在类外定义静态成员函数与定义普通成员函数相同，但定义时不能再次指定 static 关键字。比如 class A{static void f();}; static void A::f(){}是错误的。

2. 静态成员函数的特点

（1）静态成员函数没有 this 指针，这是静态成员函数与非静态成员函数的主要区别。

（2）静态成员函数只能访问这个类的其他静态成员（即静态数据成员和静态成员函数），不能访问除静态成员之外的成员。注意：此条规则并不表示静态成员函数不能访问对象的非静态成员，只是因为静态成员函数没有 this 指针，无法知道某个非静态成员来自于哪个对象，若在使用时指明了某个成员来自于哪个对象，则是可以访问的。比如 class A{public:int a; static void f(A ma){ma.a=2; this->a=4; a=3;}}; 其中 ma.a=2；是正确的；a=3;是错误的，因为 a 是非静态成员；this->a=4;也是错误的，因为静态成员函数没有 this 指针。

（3）静态成员函数不能用 const 和 volatile 修饰。因为 const 成员函数意味着该函数不能修改其所属的对象，而静态成员函数又不属于任何对象，这里相互矛盾，所以静态成员函数不能是 const 的。注意：此处是指 const 成员函数，而不是指返回类型为 const 的函数，即 static int f()

const {...} 这样的静态成员函数是不允许的，但 static const int g(){...}是可以的，g 只是返回类型为 const int 的函数。

（4）同一个函数不可以有静态和非静态两种版本，它们并不是函数重载。

（5）静态成员函数不可以是虚函数。

11.2 const 成员、mutable 关键字、this 指针

1. const 数据成员

const 数据成员的值不能被改变，声明 const 数据成员时必须初始化，const 数据成员必须使用成员初始化表进行初始化。还应注意若有多个构造函数，则所有的构造函数都必须使用成员初始化列表的形式初始化 const 数据成员。

2. const 成员函数

（1）const 成员函数把 const 关键字放在形参表和函数体之间。比如 class A{ void f() const{} const int g(){return 2;}};，其中函数 f 是 const 成员函数，函数 g 只是一个返回类型为 const int 的函数。

（2）在类体外定义 const 成员函数时也必须同时指定 const 关键字。比如 class A{void f() const; };，在类体外定义函数 f 的形式为 void A::f() const {...};。

（3）在 const 成员函数体内，数据成员相当于被 const 修饰，即 const 成员函数不能修改类的数据成员，但非类的数据成员则可以修改。比如 class A{int a; void f(int b) const {int c; b=1; c=2; a=1; }};，其中 a=1;是错误的，其他都是正确的，因为 b 和 c 不是类的数据成员。再比如 class A{int a; void f()const {int *p=&a; const int *p1=&a;}};，其中 const int *p1=&a;是正确的，int *p=&a;是错误的，因为指向非 const 的指针不能指向 const 常量，a 在 const 成员函数内部，相当于被 const 修饰。

（4）在 const 成员函数体内，成员指针也相当于被 const 修饰，但 const 是修饰的指针本身，也就是指针成员相当于* const p。因此在 const 成员函数体内能修改指针成员所指向的对象的值，但不能改变指针成员本身的值，或者说指针成员是常量，但指向的值不是常量。比如 class A{public: int a; int *p; void f() const {int b=1; *p=3; p=&b; }};，其中 p=&b;是错误的。注意：需要加一个构造函数初始化指针 p。

（5）const 成员函数可以与具有相同参数表的非 const 成员函数形成重载，在调用时，非 const 对象将调用非 const 成员函数，而 const 对象则只能调用 const 成员函数。比如 class A{void f(){}; void f()const {}};，其中的两个函数 f 是重载关系，而非重定义；再比如 const A ma; A mb;，则 ma.f()将调用 const 成员函数，mb.f()将调用非 const 成员函数。

（6）const 成员函数不能调用非 const 成员函数。因为在 const 成员函数内，调用类的成员时需要通过隐藏的 this 指针，而 const 成员函数的 this 指针类型为“const 类名 *const”，即既不能改变 this 指针本身的值，也不能改变 this 指针所指向的对象，因为无法保证非 const 成员函数不改变数据成员的值，所以 const 成员函数不能调用非 const 成员函数。

（7）注意：若使用强制转换去掉 this 的 const 性质，const 成员函数就可以调用非 const 成员函数了；同理，也可以在 const 成员函数内改变数据成员的值。比如 class A{public: int a; void g(){} void f()const { ((A* const)this)->g(); ((A* const)this)->a=4; } }; 正确，其中((A* const)this)表示把 this 指针强制转换为 A* const，即强制转换去掉了最前面的 const。

（8）注意：非成员函数不能把 const 关键字放在小括号之后，比如 void f()const {};是错误的。

3. const（类）对象

（1）const 对象指的是使用 const 修饰创建的对象。比如 class A{}; const A ma;，则 ma 就是 const（类）对象

（2）不能使用 const 对象修改其数据成员。比如 class A{public: int a; }; A ma;，则 ma.a=1; 错误。

（3）静态数据成员的值可以被 const 对象改变，因为静态数据成员不属于某个对象，而属于整个类。比如 class A{public:static int a;}; int A::a=1; const A ma;，则 ma.a=3;是正确的。

（4）const 对象只能调用 const 成员函数，但构造函数和析构函数例外，因为不是 const 的函数有可能改变数据成员的值。比如 class A{public: void f(){} void g() const {} }; const A ma;，则 ma.f();是错误的，ma.g();是正确的。

（5）使用 const 对象可以调用静态成员函数。因为静态成员函数属于整个类，且只能调用静态成员，因此它不会改变不属于该对象的数据成员的值。比如 class A{public:static void f(){} }; const A ma;，则 ma.f()是正确的。

（6）对于指向 const 对象的指针，同样不能用来修改指针所指向对象的数据成员的值。比如 A {public:int a;}; A ma; ma.a=2; const A *p=&ma;，则 p->a=3;是错误的。

（7）因为 const 对象的成员的常量性，因此不能使用指向非 const 对象的指针来指向 const 类对象的成员。比如 class A{public:int a;}; const A ma;，则 int *p=&ma.a;是错误的，因为 ma.a 是常量的地址。

（8）对于指向 const 对象的指针和指向非 const 对象的指针之间的赋值规则与常规指针相同，即指向 const 对象的指针可指向非 const 对象，指向非 const 对象的指针不能指向 const 对象。比如 const A ma;，则 A *p=&ma;错误；再比如 A ma;，则 const A *p=&ma;正确。

（9）volatile（可变的）成员函数与 const 成员函数类同，这里不多讲。

4. mutable 关键字

（1）mutable（易变的）数据成员总是可以被更改，就算是 const 对象和在 const 成员函数中，mutable 数据成员也可以被更改，声明 mutable 数据成员的形式与声明 const 变量相同。比如 class A{public:mutable int a; void f() const {a=1;}}; const A ma;，则 ma.a=2;和函数内的 a=1;都是正确的。

（2）mutable 关键字只能用于类的数据成员，在其他形式下使用是错误的。比如 class A{void f(){mutable int a;}};，mutable int a;是错误的。

（3）mutable 关键字不能与 const 同时使用，也就是不可能既是 const 数据成员又是 mutable 数据成员。比如 class A{ mutable const int a;}; 是错误的。

5. this 指针与成员函数

（1）每个成员函数都有一个指向调用该函数的类对象的隐藏指针，这个指针的名称固定为 this，我们称为 this 指针。this 指针就是一个指针，拥有指针的性质，只是这个指针是系统隐藏的，且名称是固定的。

（2）this 指针工作原理：this 指针在成员函数中是被隐式使用的，编译器会自动实现 this 指针。其编译器实现原理是给成员函数增加一个额外的形参，这个形参就是 this 指针，然后成员函数使用数据成员时都隐式加上 this 指针，在调用成员函数时，编译器会自动对被调用的成员函数额外增加一个实参，这个实参就是调用此函数的对象的地址。比如 class A{int a; void f(){a=2;}}; A ma; ma.f();，编译器会自动把成员函数转换为 void f(A *const this){ this->a=2;}，其调用会被自动转换为 ma.f(&ma);。

（3）可以在成员函数中显式使用 this 指针，但不能在成员函数的形参中明确地指定 this 指针。比如 class A{public: void f(A* this){}}; 是错误的；再比如 class A{public:int a;void f(){ this->a=1; (*this).a=2;}};，其中的 this->a=1; (*this).a=2;是正确的，这是显式使用 this 指针的方法。注意：(*this).a 的小括号不能省略，因为“.”运算符的优先级更高。

（4）this 指针的作用一：每个对象都有一份自己的数据成员的副本，但对于成员函数，则只会有一份副本，每个对象调用的都是相同成员函数的副本，那么当使用对象调用成员函数时，怎样保证成员函数中使用的数据成员是调用该函数的对象的副本呢？this 指针的作用之一就是把成员函数所使用的数据成员绑定到调用该成员函数的对象上，即 this 指针指向调用此函数的对象。比如 class A{public: int a; void f(){cout<<a<<endl;}}; A ma,mb; ma.a=1; mb.a=2; ma.f(); mb.f()，若没有 this 指针，则因为 ma.f()和 mb.f()调用的是相同成员函数 f 的副本，f 中使用的数据成员 a 究竟是 ma 的副本还是 mb 的副本不可确定，这时可能会输出 1 或 2；但有 this 指针时，则能保证 ma.f()调用的成员函数中使用的数据成员是对象 ma 的副本，而 mb.f()调用的成员函数中使用的数据成员是对象 mb 的副本，因此 ma.f()输出 1，而 mb.f()则输出 2。为了更清楚地明白其原理，这里写出编译器的实现方式，只重写函数 f，void f(A *const this){cout<<this->a<<endl;}；

而函数调用，编译器会自动实现为 ma.f(&ma);。这里可以很明显地看到函数 f 中使用的数据成员是对象 ma 的副本，而不是 mb 的，因为 this 指针指向的是 ma。

（5）this 指针示例：class A{public: int a; void f(A ma){this->a=ma.a; }}; A ma1, ma2; ma2.a=3; ma1.f(ma2);，语句 this->a=ma.a;表示将实参传递过来的对象 ma2 所在的成员 a 的值 3，赋给调用该成员函数的对象 ma1 的成员 a，调用函数 f 之后 ma1.a 的值为 3。this->a=ma.a 中的 this 也可以省略。

（6）this 指针的类型：对于非 const 成员函数，this 指针的类型是指向该类类型的 const 指针（即类型为“类名 *const this”），可以改变 this 指针所指向的对象的值，但不能改变 this 指针本身的值（即 this 指针保存的地址）；对于 const 成员函数，this 指针的类型是指向 const 类类型的 const 指针（即类型为“const 类名 *const this”），即既不能改变 this 指针所指向的对象的值，也不能改变 this 指针保存的地址。

（7）this 指针的作用二：返回调用该函数的对象的引用，这种情况在操作符重载时用得较多（详见第 12 章），也可以用于对成员函数的连接调用。比如 class A{public: A& f(){cout<<"H"<<endl; return *this;}}; A ma;，则可以这样连续调用函数 f——ma.f().f().f();。

（8）this 指针的作用三：用于复制对象。

11.3　对象数组、对象成员、数组成员和对象数组成员

11.3.1　对象数组

（1）对象数组就是每个元素都是由对象组成的数组，比如 A ma[3]。

（2）对象数组的初始化：与普通数组一样，都是使用初始化表进行初始化，使用花括号中相应的构造函数初始化对应的对象元素，这也是把对象数组的各个对象都初始化为不同值的唯一方法。比如 A ma[3]={A(2), A(2,3), A(3)};，其中 ma[0]被 A(2)初始化，ma[1]被 A(2,3)初始化，ma[2]被 A(3)初始化。这里假设 A 是一个已定义的类，且定义了相应的 3 个构造函数。

（3）在创建对象数组时，若未进行初始化，则该对象所属的类必须要有默认构造函数，否则将发生错误。因为此时创建的对象数组，相当于创建了多个未显式初始化的类对象，需要使用默认构造函数对其进行初始化。比如 class A{}; A ma[3];正确，表示使用 A 的默认构造函数初始化 ma 的 3 个对象元素；而 class A{public:A(int){}}; A ma[3];错误，因为类 A 没有默认构造函数。

（4）若只使用相应的构造函数初始化了对象数组的部分对象元素时，则余下的部分对象元

素将使用默认构造函数进行初始化，若该对象所属的类没有默认构造函数，就会出错。比如 A ma[5]={A(2), A(2,3)};，则 ma[0]被 A(2)初始化，ma[1]被 A(2,3)初始化，余下的 ma[2], ma[3], ma[4]则被默认构造函数初始化，若类 A 没有默认构造函数，则会出错。

（5）若对象数组的所有对象元素都是使用带一个参数的构造函数进行初始化的，则可以像初始化普通数组那样使用初始化列表，但对象元素需要使用带一个以上参数的构造函数进行初始化时，则必须显式提供构造函数的调用。

比如 class A{public:A(int i){};，则 A ma[3]={1,2,3};与 A ma[3]={A(1), A(2), A(3)};相同。

再比如 class A{public:A(int i=1, int j =2, int k=3){};，则：

```
A ma[2]={4,5,6};  //错误，等同于A ma[2]={A(4),A(5),A(6)};。可见初始化值过多，试图调用带三个或
                  //两个形参的构造函数初始化ma，但此处只会调用三个带一个参数的构造函数来初始化ma
A mb[5]={4,5,6, A(7,8),9 } ;//正确，等同于A mb[5]={ A(4),A(5),A(6),A(7,8),A(9) };
```

11.3.2 对象成员、数组成员和对象数组成员

将对象作为另一个类的成员（对象成员）也被称为类类型成员。

1. 对象成员的初始化

（1）注意：构造函数内部的语句是赋值，而不是初始化，初始化只会发生在成员初始化表中。

（2）没有默认构造函数的对象成员必须使用成员初始化表进行初始化。

（3）对于没有默认构造函数的对象成员，所有的构造函数都必须使用初始化表的形式初始化这些成员。

（4）若对象成员使用成员初始化表进行初始化，则调用该类对象的相应构造函数进行初始化，只需为构造函数提供相应的实参即可。比如 class B{public: B(int i){} B(int i, int j){}; class A{public: B mb1, mb2; A():mb1(1), mb2(1,2){};，则 mb1 使用 B(int)初始化，mb2 使用 B(int,int)初始化。

（5）若对象成员没有在成员初始化表中出现或没有成员初始化表，则使用该对象成员所属类型的默认构造函数初始化，若该对象成员没有默认构造函数则出错。比如 class A{public:A(int i){}}; class B{public: A ma; B(){ma=A(9);}}; 错误，ma 没有默认构造函数，类 B 的构造函数中的语句 ma=A(9);是赋值，不是初始化。

（6）在创建包含对象成员的类对象时，对象成员的创建总是先于所创建的对象，即先调用对象成员的构造函数，然后再调用包含该对象成员的对象的构造函数；释放成员时的析构函数的调用顺序按相反的次序进行。

示例如下：

```
class B{public: B(){cout<<"B"<<endl;}}; B mb1;
```

```
class A{public: int a;  B mb; A(int i):a(1),mb(){}
   A(int i, int j):a(1){ cout<<"A"<<endl; mb=mb1; } };
A ma(3);  A ma1(3,3);
```

① 创建 ma 时，对象成员 mb 是在 A(int i)的成员初始化表中被显式初始化的。

② 创建 ma1 时，对象成员 mb 会在初始化阶段被隐式初始化（成员初始化表中未出现成员 mb 的名称），初始化的步骤先于在函数体内进行的赋值 mb=mb1;，因此创建 ma1 时程序输出 BBA，而非 BAB。

2．对象成员的析构函数

（1）即使我们编写了自己的析构函数，默认析构函数也仍然存在。因此对于拥有对象成员的类，当该对象结束时除了会调用自定义的析构函数之外，还要使用默认析构函数按照声明的相反顺序来释放对象成员。

示例如下：

```
class B{}; class A{public: B mb; ~A(){ cout<<"A"<<endl; }}; void f(){A ma;} f();
```

在函数 f 调用结束后，程序首先调用类 A 中自定义的析构函数输出字符 A，然后再使用类 A 的默认析构函数调用类 B 的析构函数释放类对象成员 mb。注意：此例在类 A 的显式析构函数中没有释放类成员 mb 的语句，类成员 mb 是由类 A 的默认析构函数调用类 B 的析构函数释放的。

（2）若类中有对象成员，则显式调用析构函数时，会使用默认析构函数调用对象成员相应类的析构函数释放对象成员，当该类结束时，还会再释放一次对象成员的资源，导致对类对象成员释放资源两次。比如：

```
class B{public: ~B(){cout<<"~B"<<endl;}};
class A{public: B mb; ~A(){cout<<"~A"<<endl;}};
void main(){ A ma; ma.~A(); };
```

在 ma.~A();时程序调用类 A 的自定义析构函数输出~A，然后使用类 A 的默认析构函数调用对象成员 mb 所属类 B 的自定义析构函数输出~B，释放了对象成员 mb 的资源；当 main 函数执行结束之后，还会调用一次类 A 的自定义析构函数输出~A，然后同样会使用类 A 的默认析构函数释放对象成员 mb 的资源，这时 mb 被释放两次。

3．对象成员的复制

（1）对类对象成员的处理：与析构函数不同，当自定义了复制构造函数之后，默认的复制构造函数就不存在了，因此当类中有对象成员时，若自定义了复制构造函数，且没有使用复制构造函数初始化该对象成员时，则应注意以下两点。

① 自定义的复制构造函数拥有与其他构造函数相同的性质，会使用默认构造函数初始化该对象成员，而不会调用复制构造函数初始化。比如 class B{}; class A{public:B mb; A(A& ma);};，则定义 A::A(A&ma){}只会使用类成员 mb 所属类的默认构造函数初始化类成员 mb，而不会调

用类对象 mb 所属的默认复制构造函数或自定义的复制构造函数初始化该类成员对象。

② 要使用所属类的复制构造函数初始化该对象成员，必须在自定义的复制构造函数中显式进行，而这种调用方法只能在成员初始化表中进行操作，因为在构造函数内部是赋值，而不是初始化。比如 class B{}; class A{public:B mb; A(A& ma);};，其正确形式是 A::A(A& ma):mb(ma.mb){}，而 A::A(A& ma){mb=B(ma.mb);}不正确，因为 mb 首先在构造函数体之前使用类 B 的默认构造函数进行创建，然后在类体中进行复制。虽然此处的结果是正确的，但若类 B 无默认构造函数则会出错。这是在构造函数内部和在成员初始化表中使用复制构造函数复制对象的差别。在构造函数内部是赋值操作，初始化发生在成员初始化表阶段。

4．数组成员和对象数组成员

（1）数组成员是无法在成员初始化表中进行初始化的，若非要在成员初始化表中进行初始化，则可以将要初始化的数组封装成一个对象。比如 class A{public:int a[2];A();};，则 A::A():a[0](1),a[1](2){}是错误的；同理，A::A():a(1,2){};也是错误的。

（2）对象数组成员只能通过该对象所属类的默认构造函数进行初始化，无法通过成员初始化表初始化，因为对于数组是无法使用成员初始化表进行初始化的，所以对象数组成员所属的类必须要有默认构造函数，否则会出错；若非要在成员初始化表中进行初始化，则同数组成员一样，需要将其进行封装。比如 class B{public:B(int i){}}; class A{public: B mb[3];};，则 A ma;错误，因为类 B 没有默认构造函数。

11.4 嵌套类、局部类、友元

11.4.1 嵌套类

1．嵌套类声明、定义和嵌套类对象的创建

（1）嵌套类就是指类中的类。其声明形式如 class A{public: class B;}，即在类 A 中声明了一个嵌套类 B。

（2）定义嵌套类：嵌套类既可在外围类的作用域内进行定义，也可在外围类外的作用域中进行定义，其定义方法与成员函数相同。比如 class A{public: class B{};};，表示嵌套类 B 在外围类的作用域内进行定义；再比如 class A{class B; class B{};};，在外围类外定义的方法为 class A::B{};。

（3）在同一类作用域中，嵌套类可先声明再定义。注意：成员函数不能这样进行定义。比如 class A{class B; class B{}; };正确，而 class A{void f(); void f(){} };错误。

（4）定义嵌套类的成员函数（静态成员等其他成员也是以相同的方式进行定义的）：因为嵌

套类的成员不是外围类的成员，因此不能在外围类的作用域内定义嵌套类的成员，而必须在定义最外围类所在的作用域或其父作用域中进行定义，且必须使用作用域解析运算符指明其作用域，当然嵌套类的成员函数也可以在嵌套类中进行定义。比如 class A{class B{class C{C(); void f();};};};,则函数 f 和类 C 的构造函数不能在外围类 B 和 A 的作用域中进行定义，即 class A{ class B{ class C{ void f();}; void C::f(){}}; B::C::C(){} };是错误的，函数 f 和类 C 的构造函数必须在最外围类 A 所在的作用域内进行定义，其定义方式为 void A::B::C::f(){}，构造函数形式为 A::B::C::C(){}。

（5）创建嵌套类的对象的方法，就是使用作用域解析运算符。比如 class A{public: class B{}; };，则创建类 B 的对象的方法为 A::B mab;，这时只会调用嵌套类中的构造函数，而不会调用外围类 A 的构造函数，因为这里只创建了嵌套类 B 的对象。

（6）嵌套类在被定义之前，不能声明其对象作为成员，只能声明其指针和引用作为成员。比如 class A{ B mb; B *p; class B; B mb1; B*p1; }; class A::B{};，其中 B *p1 的声明是正确的，而 B mb1;的声明是错误的（因为类 B 未定义）；B mb; B *p 也是错误的，因为名称 B 在类 A 中无法找到。

2. 嵌套类和外围类的成员互访

（1）每定义一个类，就会生成一个相应的类作用域，其成员只在该作用域内可见，因此嵌套类的成员位于其作用域内，外围类不知道这个成员，当然在嵌套类的作用域内，是知道外围类的作用域中的成员的。

（2）嵌套类是否能访问外围类的私有和受保护成员，根据编译器而定。比如 class A{ private: static int a; class B{void f(){a=3;} };}; int A::a;，在 VC++ 2010 中嵌套类不能访问外围类的私有成员。这种错误发生在创建嵌套类对象时，比如 A::B mb;会发生错误，若不创建嵌套类对象，则不会出错。

（3）嵌套类的成员函数不能直接访问外围类的非静态数据成员和非静态成员函数，要访问它们需要使用外围类类型的对象、指针或引用。因为每个成员函数都拥有所属该对象类型的 this 指针，要直接访问外围类的成员则需要 this 指针指向外围类类型对象，但嵌套类的成员函数的 this 指针是指向嵌套类类型对象的。比如 class A{public: int a; void g(){} class B{void f(){a=3; g(); }}; };，对 a 和函数 g 的访问是错误的。

（4）在嵌套类中可以直接访问外围类的静态成员、类型名和枚举值（注意：这些成员必须是公有的）。因为静态成员、枚举值这些成员是属于整个类的，因此在嵌套类中可以访问。在外围类外使用作用域解析运算符可以直接访问这些成员，而不需要使用类的对象进行访问（此规则见下面示例对 A::T b 的使用）。其中类型名指的是 typedef 定义的名字、枚举类型名、类名。注意：枚举类型名和枚举值是不同的，如 enum M{MM, NN};，枚举类型名是 M，枚举值是 MM、NN。

示例 11.2：嵌套类可访问的外围类的成员

```
class A{public:typedef int T;  //T、M、C、a 的声明应在类 B 之前
    enum M{MM,NN};
    class C{};
    static int a;
    class B{public:
        T b;              //正确，T 是类型名
        M mm;             //正确，M 是枚举类型名、也属于类型名
        C mc;             //正确，C 是类名，也属于类型名
        void f(){a=MM;} //正确，a 是类 A 中的静态成员，MM 是枚举值
    }; };
A::T b;     //在类外引用这些成员时，只需使用作用域解析运算符直接访问即可，不需要使用类的对象进行访问
int A::a=2;       //对静态成员变量 a 的定义
void main(){A ma;  A::B mb;  mb.f();}
```

3. 外围类和嵌套类的对象

（1）外围类的对象与嵌套类的对象之间没有关联。

（2）嵌套类的成员不是外围类的成员；相反，外围类的成员也不是嵌套类的成员。因此嵌套类的对象不能访问外围类的成员；同理，外围类的对象也不能访问嵌套类的成员。比如 class A{public: int a; void f(){} class B{public: int b; void g(){} }; }; A ma; A::B mab;，则 mab.a=3;、mab.f();、ma.b=3; 和 ma.g();都是错误的。

4. 嵌套类类体中的名称解析

以下规则包括函数形参，但成员函数体中的名称和默认实参除外。

（1）查找在该名称之前的嵌套类成员的声明，若找到，则使用该名称。

（2）若第一步不成功，且嵌套类是在外围类作用域中定义的，则查找该名称使用点之前的外围类成员的声明；若仍未找到，则查找该名称使用点之前的名称空间（或作用域）中的声明。

（3）若第一步不成功，且嵌套类是在外围类作用域之外进行定义的，则查找整个外围类成员的声明，因为这时外围类的定义对于嵌套类而言是完全可见的；若仍不成功，则在定义嵌套类之前的名称空间（或作用域）中进行查找。

（4）虽然外围类的非静态成员、非类型名、非枚举值不能被嵌套类访问，但它们对于嵌套类仍是可见的，这就是外围类的静态成员可被嵌套类直接使用的原因。比如 float a; class A{int a; class B{public: void f(){ a=1;}};};，其中 a=1;是错误的，此处的 a 指的是外围类中的名称 int a，而不是全局作用域中的名称 float a。

（5）这是常规的名称解析过程，即名称应先声明再使用。

示例 11.3：嵌套类类体中的名称解析

```
typedef int T;
class A{ public: typedef float T;
```

```
    class B{ public:     //嵌套类在外围类的作用域内进行定义
        T b;             //使用外围类 A 中的 float T（规则 2）
        //T1 b1;         //错误，类 A 和全局类中的 T1 对类 B 不可见，因为 T1 在类 B 的定义之后
        };               //嵌套类 B 结束
    class C;             //声明而不定义嵌套类 C
    typedef int T1;  };  //外围类 A 结束
typedef float T1;
class A::C{              //在外围类的作用域外定义嵌套类 C
    T1 c;}; //使用外围类 A 中的 int T1，若外围类中没有 T1 的定义，则使用全局作用域中的 float T1（规则 3）
```

5. 嵌套类的成员函数体中的名称的解析

以下规则同样适用于默认实参、静态变量。

（1）首先在成员函数体的局部作用域中查找，若在该名称之前找到，则使用该名称。

（2）若第一步未找到，则查找整个嵌套类作用域内的成员声明。

（3）若第二步未找到，则查找整个外围类作用域内的成员声明。

（4）若第三步未找到，且成员函数是在嵌套类类体中进行定义的，则查找该名称使用点之前的名称空间（或作用域）中的声明。

（5）若第三步未找到，且成员函数是在外围类类体外进行定义的，则查找该成员函数定义之前的名称空间（或作用域）中出现的声明。

（6）对于带默认实参的成员函数，若默认实参在嵌套类的内部指定，则遵守第三步和第四步的规则；若在类外指定，则遵守第三步和第五步的规则。注意：对于 typedef int T; void f(T a=b);，这里只有 b 才是默认实参，只有 b 才遵守这一规则，而 a 是形参，T 是形参类型，都不遵守这一规则。若默认实参是类的成员，则默认实参必须是静态的，不能是非静态数据成员。

（7）静态成员变量一般在类外进行定义，遵守第三步和第五步的规则。

（8）虽然外围类的非静态成员、非类型名、非枚举值不能被嵌套类访问，但它们对于嵌套类仍是可见的。

（9）注意：上面的名称解析过程不遵守名称先声明后使用的规则。

示例 11.4：嵌套类的成员函数体中的名称解析

```
int a;  int b;  int c;   //全局名称 a,b,c
class A{public:int a;
    class B{public:
        void f(){
        //a=1;//错误，并不使用全局作用域中的名称 a，而是使用外围类中的名称 a，但它是非静态的，不能访问
              //由此可见，外围类中的名称即使不能被嵌套类访问，也仍是可见的
        b=2;          //正确，使用嵌套类类体中的名称 b（规则 2）
        //c=3;        //错误，使用外围类中的名称 c，并不使用全局名称 c，虽然外围类中的名称 c 在函数 f
                      //的定义之后声明，但仍会使用它，它是非静态的，不能访问（规则 3）
        //d=4; //错误，全局作用域中的名称 d 在函数 f 的定义之后不可见（规则 4）
        }
```

```
        void g();   //声明而不定义函数g
        int b; };   //嵌套类B定义结束
    int c; };     //外围类A定义结束
int d;            //全局名称d
void A::B::g(){     //在外围类类体外定义嵌套类的成员函数g
   d=4; }         //正确，使用全局作用域中的名称d，因为d出现在g的定义之前（规则5）
```

11.4.2 局部类

（1）局部类指的是函数体内定义的类。比如 void f(){class A{}; }，其中类 A 就是局部类。

（2）因为是在函数体内定义的，因此局部类的可见性只限于该局部作用域内。在该局部作用域外没有其他语法可以引用局部类的成员。

（3）由于局部类的可见性只限于该局部作用域内，因此局部类拥有以下一些性质。

① 因为局部类只在该局部作用域内可见，在函数内又不能定义另一个函数，所以局部类的成员函数只能被定义在类定义中。比如 void f(){class A{void g();}; void A::g(){}}，其中对函数 g 的定义是错误的，正确的方式是 void f(){class A{void g(){}};}，把函数 g 定义在类定义中。

② 在局部类中不能有静态数据成员（注意：静态成员函数是允许的），因为没有语法能够定义局部类的静态数据成员。比如 void f(){class A{static int a;};}是错误的，因为无法对 a 进行定义。

（4）局部类只能访问外围局部作用域中的类型名、静态变量及枚举值。其中类型名指的是 typedef 定义的名字、枚举类型名、类名。

（5）在函数内可以定义局部类，因此在局部类中嵌套的类也可以被定义在函数体内，但嵌套类的定义必须在包含外围局部类定义的函数体内，当然也可以在外围类中进行定义。比如 void f(){ class A{ class B;}; class A::B{} ; }，其中 class A::B{}就是对局部类中的嵌套类在函数体内进行的定义。

（6）局部类中的名称解析过程与嵌套类相似，具体详见 11.4.1 节。

（7）局部类的使用很有限，这里不再举例说明。

11.4.3 友元

1. 友元的声明和定义

（1）声明友元的目的是让类的非成员函数或者类能访问其私有的和受保护的成员，一旦把函数或者类声明为友元，就可以访问该类的私有的和受保护的成员。比如 class A{private: int a;}; void f(){A ma; ma.a=2;};，其中 ma.a=2;是错误的，因为 a 是私有的，若函数 f 是类 A 的友元，则是正确的。

（2）可以把函数、类的成员函数、类声明为友元，分别称为友元函数、友元成员函数、友

元类。

（3）声明友元只需使用 friend 关键字即可。比如 class A{friend void f(); friend class B;};，其中 f 是类 A 的友元函数，类 B 是类 A 的友元类。

（4）友元关系是单向且不可传递的。因此，若类 A 是类 B 的友元，不等于类 B 是类 A 的友元；若类 B 是类 A 的友元，类 C 是类 B 的友元，不等于类 C 是类 A 的友元。

（5）声明友元类时，若这个类在之前已被声明或定义过，则可以省略 class 关键字。比如 class B; class A{friend B;};正确；class A{friend B;}; class B{};错误，类 B 未在类 A 前进行声明，因此把类 B 声明为类 A 的友元时需要加上 class 关键字，即正确形式为 class A{friend class B;}; class B{};。

（6）在声明友元函数时可同时进行定义，但在声明友元类时不能同时进行定义。比如 class A{friend class B{}; };错误；class A{friend void f(){}};正确。

（7）一个函数或类可以同时是一个或多个类的友元。比如 void f(){}class A{friend void f();}; class B{friend void f();};，函数 f 既是类 A 又是类 B 的友元。

（8）友元函数不能使用存储类标识符，即不能把友元函数定义为 static 或 extern。此规则某些编译器不一定支持。

（9）友元函数不会被继承。

（10）重载函数需要集中把每一个要设为友元的版本都声明为友元，否则就不是该类的友元。比如 class A{friend void f();}; void f(){} void f(int i){}，则函数 f(int i)就不是类 A 的友元。

（11）友元函数不是声明这个友元的类的成员函数，但却可以访问这个类的非公有成员，因此友元具有以下特点（友元类具有相同的性质）。

① 友元可以在类外进行定义，这时不需要使用 friend 关键字，也不需要使用类名进行限定，只需像定义普通函数一样定义即可。比如 class A{friend void f();};，则 friend void f(){}和 void A::f(){}的定义都是错误的，因为友元函数不是类 A 的成员，正确形式是 void f(){}。

② 友元函数只需像普通函数一样调用，不能使用成员运算符或作用域解析运算符调用友元函数。比如 class A{friend void f();}; void f(){} A ma;，则 ma.f(); 和 A::f();都是错误的，因为 f 不是类 A 的成员，调用 f 的正确形式是 f()。

③ 友元的声明可以在类定义中的任何位置，不受该类访问权限的影响，因为友元函数不是类的成员函数。比如 class A{public: friend void f(); private: friend void g();}，则函数 f 和 g 都是类 A 的友元函数，声明友元的位置是在 public 还是 private 后没有影响。

④ 在友元函数内访问该类的成员时，需要使用该类的对象或者指向该类对象的指针或引用，即使友元函数定义在类中，也不能直接对类中的成员进行访问，因为友元函数不是类的成员函数。比如 class A { int a; friend void f(){a=3; A ma; ma.a=4;}};，其中 a=3;是错误的，ma.a=4;是正确的。

2. 友元声明是否会给外围作用域引进一个名称

经过测试，各编译器的行为不一致。

（1）规则 1：友元声明不会给外围作用域引进一个名称。比如 class A{friend void f(){} }; void main(){ f();}，其中对函数 f 的调用是错误的。

（2）规则 2（VC++ 2010 遵守此规则）：

① 若友元函数在类的内部定义，则该函数的作用域扩展到包围该类定义的非类作用域。也就是说，若友元函数在类的内部进行了定义，则不能在该类外进行重复定义。比如 class A{friend void f(){}}; void f(){}错误，函数 f 在类 A 外被重复定义。

② 若友元函数在类体中定义，则该友元函数需要在某个外围作用域内显式声明，或者以它的类或由该类派生的类作为一个参数，否则无法调用这个友元函数。

- 外围作用域，是指声明友元函数所在的外围最近的作用域，即授权类所在的作用域。
- “以它的类或由该类派生的类作为一个参数”，指的是友元函数的形参为授权类或该类的子类类型。比如 class A{void f(A ma, int){}}，形参 ma 就是属于这种情形。关于此情形的详细解释请参阅 12.3.3 节和第 15 章的 ADL 部分。

示例如下：

```
class B{class A{friend void f(){}};}; void f(){}
```

错误，函数 f 被重定义。友元函数的作用域被扩展到包围授权类所在的非类作用域。

```
class A{friend void f(){}}; void main(){void f(); f();}
```

正确，在调用函数 f 之前对其进行声明。注意：虽然友元声明是在 main 函数内部进行，但 main 的作用域是授权类所在作用域的嵌套作用域，因此仍是在授权类所在的作用域范围内。

```
namespace N{class A{friend void f(){} }; void f();} void main(){N::f();}
```

正确，友元函数在外围作用域内显式声明。这里要注意：若函数 f 不是在名称空间 N 中进行声明的，则是错误的。比如 namespace N{class A{friend void f(){}};} void N::f(); void main(){N::f();} 错误，因为对友元函数的声明不在授权类所在的作用域内。

```
class A{friend void f(A ma){} }; void main(){A ma1; f(ma1);}
```

正确，使用形参找到友元函数。注意：若函数 f 的形参不是类 A 或类 A 的派生类的形参，则无法找到该函数。

```
class A{friend class B; friend void f(){}};
```

B *p;是正确的，但 B mb;是错误的（因为类 B 未定义）。

（3）应注意区分友元函数、成员函数和名称空间成员函数，这三种函数是不同的，下面通过示例进行说明。

示例 11.5：区分友元函数、成员函数和名称空间函数

```
class A{public:friend void f(){cout<<"f"<<endl;}
    friend void f1(){cout<<"f1"<<endl;}
    friend void f2(){cout<<"f2"<<endl;}
```

```
    void f();//声明一个成员函数 f（未被定义），这不是对友元函数的声明
    void f1(){cout<<"Af1"<<endl;}
    void g(){
        f2();  //正确，这不是在授权类外使用友元
        //f();  //错误，调用最先找到的未被定义的成员函数 f，此处并不会调用友元函数
        void f();  //对友元函数 f 的声明，这不是对成员函数 f 的声明
        //void A::f();  //错误，不能对成员函数 f 进行重新声明
        f();  //正确，调用最先找到的对友元声明的名称 f
        f1();  //调用成员函数 f1
        void f1();
       f1();}}; //调用友元函数 f1
//将友元函数 f,f1,f2 的作用域扩展至全局作用域
void f1(int i){cout<<"::f1i"<<endl;};
namespace N{
   //f1(3);   //错误，不能在名称空间中调用函数
    void f();  //声明一个名称空间成员（未被定义），这不是对友元函数的声明
    void g(){
        //f2();  //错误，因为未对友元进行声明就直接使用
        //f();  //错误，调用最先找到的未被定义的名称空间中的成员函数 f，此处并不会调用友元函数
        f1(3);  //正确，调用全局名称 f1(int)
        void f1();  //正确，友元函数声明，此声明会隐藏全局名称 f1(int)
        f1();    //正确，调用最先在名称空间中找到的对友元声明的名称 f1()
        //f1(3);  //错误，f1(int)被友元声明隐藏
    }
}
void main(){A ma; ma.g(); N::g();}
```

3. 判断哪些是友元（友元查找）

（1）在外围作用域中先进行过声明的是友元。比如 void f(); class A{friend void f(); }; ，其中函数 f 是类 A 的友元函数。

（2）在授权类所在的非类作用域外直接（或最内层）作用域中定义的是友元。

（3）下面通过示例对以上两点内容进行理解。

示例 11.6：对友元查找之授权类所在的非类作用域外直接（或最内层）作用域的理解

```
class E1;
namespace N{class E2;
   class A{ public:
        class C1;
        class E3;
        class B{int b;
          friend class C;     //非友元，因为 C 的定义是在授权类 B 所在的类作用域内
                              //（即类 A 的作用域）进行定义的
          friend class C1;    //是友元，C 与 C1 是在同一个作用域中定义的，但 C1 进行了前向声明
          friend class D;     //是友元，因为 D 的定义是在授权类 B 所在的非类作用域外直接
                              //（最内层）作用域 N 中进行定义的
          friend class E;     //非友元，因为类 E 是在授权类 B 所在的非类作用域外非最内层作用域
```

```
                                    //（全局作用域）中进行定义的
        friend class E1;    //是友元，E1 与 E 都定义于全局作用域内，但 E1 进行了前向声明
        };  //嵌套类 B 定义结束
      class C{//void f(){B mb; mb.b=3;}        //C 不是类 B 的友元，不能访问类 B 的私有成员
       };
      class C1{ void f(){B mb; mb.b=3;}};      //C1 是类 B 的友元，可以访问类 B 的私有成员
      };  //类 A 定义结束
   class D{ void f(){A::B mb; mb.b=3;}};       //D 是类 B 的友元，可以访问类 B 的私有成员
   }      //名称空间 N 定义结束
   class E{//void f(){N::A::B mb; mb.b=3;}     //E 不是类 B 的友元，不能访问类 B 的私有成员
   };
class E1{ void f(){N::A::B mb; mb.b=3;} };     //E1 是类 B 的友元，可以访问类 B 的私有成员
void main(){}
```

示例 11.7：友元查找之作用域解析运算符的使用

```
class E;
namespace N{  class E;
   class A{public: class E;
      class B{int b;
         class E{};
         friend class E;          //此处指的是 N::A::B 中的 E，因为首先找到的是该名称
         friend class A::E;       //把类 A 中的 E 声明为友元
         friend class N::E;       //把名称空间 N 中的 E 声明为友元
         friend class ::E;        //把全局作用域中的 E 声明为友元
      };                          //类 B 定义结束
   class E{void f(){B mab; mab.b=3;} };
   };                             //类 A 定义结束
   class E{void f(){A::B mab; mab.b=3;}};
   }                              //名称空间 N 定义结束
class E{void f(){N::A::B mab; mab.b=3;}};
void main(){}
```

4. 友元成员函数与前向声明

（1）当把成员函数声明为友元时，成员函数必须使用其所属的类名加以限定，且该类必须先已定义。比如 class A{friend void B::f();};，类 B 必须先被定义，否则无法知道 B 是一个类而产生错误。同理，class B; class A{friend void B::f();}; class B{public: void f(){}}; 是错误的，虽然前向声明了类 B，但是程序无法知道函数 f 是否是类 B 的成员。其正确做法有两种，一种是在类 A 之前定义类 B 并定义函数 f；一种是在类 B 的定义中只声明而不定义函数 f，把函数 f 的定义放到后面进行，其形式为 class B{public: void f();};class A{friend void B::f();};void B::f(){}。

（2）示例如下：

```
class A;                                //前向声明
class B{public: void f(A ma);};         //此处只能声明而不能再定义函数 f，因为类 A 未被定义
class A{int a;  friend void B::f(A ma);};
void B::f(A ma){ ma.a=1;  }             //在定义类 A 之后，对类 B 的成员函数 f 进行了定义
```

本例把类 B 的成员函数 f 声明为类 A 的友元，就要求类 B 的定义必须在类 A 的定义之前，同时类 B 的成员函数 f 又要用到类 A，因此要求类 A 的定义必须出现在类 B 之前。这就出现了互相嵌套的问题，解决此问题的办法就是对类 A 进行前向声明，同时在类 B 中对成员函数 f 只声明而不定义，把对成员函数 f 的定义放在类 A 的定义之后，这样程序就不需要在类 B 之前对类 A 进行定义了，只要求对类 A 进行声明即可，否则将出错

11.5　指向类成员的指针

1. 指向数据成员的指针

（1）指向数据成员的指针，就是指指针指向的是类的数据成员。

（2）其声明形式为：类型名 类名::*指针名; 比如 int A::*p;。

（3）指向数据成员的指针类型为“类型名 类名::”，即要求指针指向的是来自于某个类的成员，因此初始化此指针时，要求数据成员的类型名和所属的类类型相匹配。

（4）注意区分指向数据成员的指针与指向对象成员的指针，指向对象成员的指针只需一个普通指针即可。比如 class A{public:int a;}; A ma; int A::*p1; int *p2;， 则 p2=&ma.a;正确，p1=&A::a 正确。但 p1=&ma.a;错误，p1 是指向数据成员的指针；p2=&A::a;错误，p2 不是指向类数据成员的指针。

（5）不能直接引用类的成员，以试图把类成员的地址直接赋给指向类成员的指针。比如 class A{public:int *a;}; int A::*p;，则 p=A::a;是错误的，这里试图使用 A::a 对指针赋值，但不能直接引用类的成员 a，即 A::a 是错误的，其正确形式为 int *(A::*p)=&A::a;。详见后文。

（6）类的静态成员不属于某个类的对象，因此指向类静态成员的指针，与普通指针相同，而不能使用指向类成员指针的形式。比如 class A{public:static int a;}; int A::a=3;，则 int A::*p=&A::a;是错误的，其正确形式是 int *p=&A::a;。

示例如下：

```
class A{public:int a;}; class B{public:int a;}; A ma;
int A::*p1=&A::a;        //正确
float A::*p2=&A::a;      //错误，a 的类型 int 与 p2 的 float 类型不匹配
int B::*p3=&A::a;        //错误，a 的类类型 A 与 p3 的类类型 B 不匹配
int B::*p4=&B::a;        //正确
int *p5=&A::a;           //错误，p5 是普通指针，不是指向数据成员的指针
int A::*p1=&ma.a;        //错误，&ma.a 的地址只需一个 int 类型的普通指针即可。
```

2. 指向成员函数的指针

（1）指向成员函数的指针，就是指指针指向的是类的成员函数。

（2）其声明形式为：返回类型(类名::*指针名)(参数表);。注意：小括号不能省略。比如 void

(A::*p)(int);，表示声明一个指向来自于类 A 且带一个 int 形参返回类型为 void 的指针 p。

（3）指向类成员函数的指针类型为“返回类型(类名::)(参数表)”，即要求指针指向的是来自于某个类的成员，因此初始化此指针时，要求所属的类类型、函数的返回类型和参数表相匹配。

（4）初始化普通的指向函数的指针时，为了与 C 兼容可以直接使用函数名进行初始化，但初始化指向类成员函数的指针时，最好是使用“&”运算符，否则有可能会出错。比如 class A{public: void f(){}};void g(){}void (*p)(); void (A::*p1)();，则 p=g; p1=&A::f;是正确的，但 p1=A::f;不一定正确，此语句可以表示调用类 A 的静态成员函数 f。

（5）不能通过对象名引用成员函数的地址。与数据成员不同，成员函数是属于整个类的，其所有对象共用一个副本，而数据成员可以根据所创建的对象而创建相应的副本。因此成员函数的入口地址不能通过对象进行引用，只能通过所有对象的公用函数代码段的入口地址，它是通过类名进行引用的。比如 class A{public:void f(){}};A ma; void (A::*p)(); void (*p1)();，则 p=&ma.f;或 p1=&ma.f;是错误的，其正确形式是 p=&A::f;。

示例如下：

```
class A{public: void f( int ){}}; class B{public:void f(int){}}; A ma;
void (*p)()=&A::f;                  //错误，p 不是指向成员函数的指针
void (A::*p1)()=&A::f;              //错误，p1 的形参表与函数 f 的形参表不匹配
void (A::*p2)(float)=&A::f;         //错误，p2 的形参类型为 float 与 A::f 的形参类型 int 不匹配
void (A::*p3)(int,int)=&A::f;       //错误，形参表不匹配，p3 和 A::f 的形参数目不相同
int (A::*p4)(int)=&A::f;            //错误，p4 与 A::f 的返回类型不匹配
void (A::*p5)(int)=&B::f;           //错误，p5 的类类型 A 和 B::f 所属的类类型不匹配
void (A::*p6)(int)=&A::f;           //正确
void (*p7)()=&ma.f;                 //错误，不能通过对象引用成员函数的地址
void (A::*p8)()=&ma.f;              //错误，原因同上
```

3. 使用指向类成员的指针

（1）指向类成员的指针必须通过该类的对象或指向该类类型对象的指针来访问，也就是不能直接访问，通过对象访问要使用“.*”运算符，通过指针访问要使用“->*”运算符。比如 class A{public:int a;}; int A::*p=&A::a; A ma; A *p1=&ma;，则 ma.*p=3;和 p1->*p=4;都是正确的；但 *p=3;、A::*p=3;、*ma.p=3;等都是错误的。

（2）指向类成员的指针的地址值是该成员相对于类起始地址的偏移量。注意，得到的并不是地址，而是一个偏移量，这意味着指向第一个成员的指针的偏移量是 0。比如 class A{public:int a; double b; char c;};，则 char A::*pa=&A::c;，cout<<pa<<endl;将得到 12，即数据成员 c 相对于类起始地址的偏移量为 12（VC++ 2010 始终输出 1）。

（3）指向数据成员的指针得到的偏移量比实际的值总是多 1，其原因在于与空指针相区别。比如 class A{public: int a;};，则 int A::*pa=&A::a;，pa 的值为 1，若不对 pa 的值加 1，pa 将与空指针的值相同，都是 0。

（4）注意：使用 VC++ 2010 对指向数据成员的指针输出值始终为 1，这可能与编译器实现该功能或做了优化有关。另外，在 VC++ 2010 中对指向数据成员的指针进行算术运算是错误的。比如 int A::*pa=&A::a;，则&A::a+1;、pa+1;、pa−1;、&A::a−&A::b;等都是错误的。

（5）获取一个非虚拟成员函数的地址，将得到该函数在内存中的实际地址（VC++ 2010 仍然为值 1），这与获取数据成员时的地址不同。但要使用该成员函数，还是必须要绑定于某个类对象的地址上才行。比如 void (A::*p)()=&A::f; A ma;，则必须这样调用 f 函数(ma.*p)();，直接使用指针 p 进行调用是错误的，如(*p)()错误。

4．指向类成员的指针的类型分析

（1）对有关声明时形如 int **…**A::**…**p 的类型理解：首先需要从标识符 p 开始分析，p 的左边至“::”之间有多少个“*”，p 就是几级指针，且指针 p 最终所指向的一级指针指向的是类型为 A 的成员，这个成员是一个拥有类名 A 和类型 int 之间所包含的“*”数目的指针。比如 int ****A::*****p;，表明 p 是一个五级指针，指向类型为“int ****A::****”的对象，这个五级指针最终所指向的一级指针指向的类型是 A 的成员，且 A 的成员必须是类型为 int 的四级指针。比如 class A{public: int *****a;};，因为五级指针需要一个四级指针的地址，因此初始化五级指针 int ****A::*****p 的具体步骤如下：

```
int *****A::*p1=&A::a;      //初始化一级指针 p1
int *****A::**p2=&p1;       //初始化二级指针 p2
int *****A::***p3=&p2;      //初始化三级指针 p3
int *****A::****p4=&p3;     //初始化四级指针 p4
int *****A::*****p=&p2;     //初始化五级指针 p
```

（2）对有关赋值时&A::a 的类型理解：&A::a 表示获取类成员 a 的地址，取址后的类型相当于 A::*p，若再对 p 取址即&p，则得到二级指针，其类型与 A::**p1 中的 p1 相同，若对 p1 再取址即&p1，则得到三级指针，其类型与 A::***p2 中的 p2 相同，依此类推。&A::a 的类型与类 A 中成员的指针级别无关，而只与声明指针时位于（内置）类型与类名之间的指针级别有关。比如 class A{public: int ****a;};应这样声明指针：int ****A::*p=&A::a; ，可以看到&A::a 相当于类型 A::*p;是一个一级指针。类成员 a 的指针级别与&A::a 无关，而只与位于 int 和类名 A 之间的指针级别有关。注意：不能对变量连续使用取址运算符，比如&(&a)是错误的，因为&的结果值是右值，但&运算符要求的是左值。

（3）对有关“对象名.**…**”的理解：只要右边的所有*解引用之后是对象的成员，则结果就是正确的，否则就是错误的。比如 class A{public:int a;}; A ma; int A::*p1=&A::a; int A::**p2=&p1;，则 ma.**p2=3;是正确的，其中 p2 解引用两次之后得到的是类 A 的成员 a，它是对象 ma 的成员，因此 ma.**p2 相当于 ma.a 就是指对象 ma 的成员 a。注意 ma.*p2 是错误的，因为对 p2 解引用一次之后得到的是 p1，p1 不是 ma 的成员。

（4）对有关“**…**（对象名.**…**）”的理解：注意小括号不能省略。只要右边的所有*解引用之后得到的是对象的成员，且这个成员能解左边所有*的次数，就是正确的。比如 class A{public: int ***a; }; A ma; int ***A::*p=&A::a;，则 ma.*p;正确，p 解引用一次后得到的是类 A 的成员 a，它是对象 ma 的成员，因此 ma.*p 相当于 ma.a 是正确的；同理，*(ma.*p)相当于*ma.a 也是正确的；但****(ma.*p);是错误的，因为****(ma.*p)相当于****ma.a，错在解引用次数过多。

示例 11.8：使用指向类成员的指针

```
class A{public:
    int a; int ***a1; int *a2; int **a3;
    A(){a2=&a; a3=&a2; a1=&a3;} };
void main(){
    A ma; ma.a=1;
    int A::*p=&A::a;
    A *p1=&ma;
    //对声明时***A::****和&A::a 的类型理解
    int ***A::*p2=&A::a1;    //正确，对&A::a1 和***A::*p2 的理解
    int ***A::**p3=&p2;      //正确，p2 是一个二级指针
    //对有关“对象名.**…**”的理解
    cout<<ma.*p2<<endl;      //正确,对 p2 解引用一次之后得到 a1,而 a1 是类 A 的成员,因此相当于 ma.a1
    cout<<ma.a1<<endl;
    //ma.**p2;           //错误，p2 是一级指针，却解引用两次。
    //ma.*p3;            //错误，p3 是二级指针，解引用一次后得到 p2 并不是类 A 的成员
    //ma.p3;             //错误，p3 不是 ma 的成员
    cout<<ma.**p3<<endl;          //正确，原理同 ma.*p2，这时相当于 ma.a1
    //对有关“*…*(对象名.**…**)”的理解
    cout<<***(ma.**p3)<<endl;     //输出 1，正确，ma.**p3 之后相当于 ma.a1，然后对 a1 解引用三次，
                                  //因此最后得到 a1 所指地址中的值，相当于***ma.a1
    cout<<***ma.a1<<endl;
    cout<<*(ma.**p3)<<endl;       //输出 a2 的地址，即相当于 ma.a3、&ma.a2 或*ma.a1，原理同上
    cout<<ma.a3<<endl;
    cout<<&ma.a2<<endl;
    cout<<*ma.a1<<endl;  }
```

11.6 枚举、联合（共用体）、位段（域）

11.6.1 枚举类型

（1）本节应注意区分枚举变量（对象）、枚举常量（成员）、枚举类型三者的区别。枚举常量是一个编译时常量，可用于定义数组的下标等；枚举变量就不是常量了，不能进行数组下标的定义；而枚举类型是用于区分变量的类型和定义枚举变量的。例如 enum E{aaa,bbb}; E me;，其中 E 是枚举类型名，可用于定义枚举变量及区分类型，aaa 和 bbb 是枚举常量，me 是枚举变

量。因此 int a[bbb];是正确的，而 int a[me];是错误的，因为 me 是变量；注意 int a[aaa];是错误的，因为 aaa 的值在此处为 0。

（2）枚举常量、枚举元素、枚举成员、枚举值是同一概念，指的就是枚举成员。

1．枚举声明

（1）枚举其实就是一个有名字的整型常量的集合。

（2）枚举声明的形式为：enum 枚举类型名{枚举成员列表};。

枚举使用 enum 关键字进行声明；枚举类型名就是一个自定义的标识符，使用枚举类型名来定义枚举变量；枚举成员也是一个自定义的标识符，它们之间使用逗号隔开；在声明的最后还应以分号结束。例如 enum E{abc, de, fr}; E me;，其中 enum 是关键字，E 是枚举类型名，abc, de, fr 是枚举成员，me 是枚举变量。

（3）枚举成员与枚举常量：枚举成员是一个有名字的整型常量，它是一个编译时常量。正因为枚举成员是常量，所以常被称为枚举常量。

（4）枚举的作用：若编程时需要在一组有限的值当中进行选择时，枚举就很有用。比如要判断当前的颜色是否是 red（红）、green（绿）、blue（蓝）、这三种颜色当中的一种时，就可以使用枚举，其程序为 if(me= =red){...}，这里的 me 只能在 red、green、blue 这三个值当中进行选择，这时就可以把 me 声明为枚举类型，而把 red、green、 blue 声明为枚举常量，即 enum E{red, green, blue}; E me; if(me= =red){...}。

2．枚举类型变量（对象）

（1）声明枚举类型之后，枚举类型名就成为一个新类型（即枚举类型）的名字，然后可以使用枚举类型名定义该类型的变量（对象）。比如 enum E{aaa, bbb};，则 E me;中的 me 就是新定义的枚举类型变量。

（2）枚举变量有三种定义方法。

① 使用枚举类型名直接定义，就像定义一个普通变量一样。比如 enum E{aaa, bbb};E me;。

以下两种方法是从 C 语言继承而来的。

② 定义时使用枚举类型名和 enum 关键字，比如 enum E{aaa}; enum E me;。

③ 在声明枚举类型时进行定义，比如 enum {aaa, bbb} me; 或 enum E{aaa, bbb} me;。

（3）枚举类型变量的值只能是枚举成员（常量）或同一枚举类型的其他变量。注意：非法赋值某些编译器不一定会报错。比如 enum E{aaa, bbb};enum EE{ddd}; E me;，则 me=aaa; me=bbb; E me1=me;是正确的，但 me=3;错误，因为 3 是整型，与 me 的枚举类型不兼容； me=xx;错误，因为 xx 不是 me 的成员；EE mee=me;错误，因为 mee 的类型 EE 与 me 的类型 E 不相同。

3. 访问枚举常量（成员）及其作用域

（1）调用枚举常量：只需在可见的作用域中直接使用枚举常量的名字即可，而不能通过枚举变量使用点运算符进行访问。比如 enum E{aaa,bbb};，则 int a=aaa;表示把枚举常量 aaa 赋给 a；再比如 E me;，则 int a=me.aaa;是错误的，因为枚举常量不能使用点运算符进行访问。

（2）枚举常量的作用域与定义枚举时的作用域相同。比如 void f(){enum E{aaa=3, bbb=2}; int c=aaa;} 正确，枚举常量 aaa、bbb 和 c 在同一个作用域中；再比如 class A{enum E{aaa=1,bbb}; int a[aaa];};，其中 int a[aaa]是正确的，因为枚举常量是编译时常量，可用于定义数组下标。

（3）枚举常量的名字在其作用域内必须是唯一的。比如 enum E{a,bbb,a};错误，enum E{a,bbb}; int a;也错误，因为枚举常量名 a 与在同一个作用域内的整型变量 a 同名了。

4. 枚举类型名及枚举常量的作用域

枚举类型名与结构类型名、联合类型名、类类型名共用同一个名称空间（或作用域），枚举常量与一般标识符共用同一个名称空间。

示例如下（不建议使用相同的名称，这里只是为了说明相关规则）：

```
enum E{aa, bb}; int E=2;     //正确，枚举类型 E 与整型变量 E 不在同一名称空间
enum E{aa,bb}; class E{};    //错误，枚举类型 E 与类类型 E 在同一名称空间
enum E{E, aa};               //正确，枚举类型 E 与枚举常量不在同一名称空间
enum E{a, bb}; int a=3;      //错误，枚举常量与整型变量 a 在同一名称空间
enum E{A, bb}; class A{};    //正确，枚举常量 A 与类类型 A 不在同一名称空间
const int a=1; enum a{aa=1, bb=a};  //正确，枚举类型 a 与变量名 a 不在同一名称空间；
                                    //bb=a 也是正确的，这里的 a 指的是 const int a
```

5. 枚举常量的值及匿名枚举

（1）枚举常量的类型就是定义时的枚举类型。注意：虽然它的值是一个整数，但枚举常量的类型不是 int。

（2）枚举常量的值是按定义时的顺序进行赋值的，默认第一个枚举常量的值为 0，其后的枚举常量的值就在前一个值的基础上加 1。比如 enum E{aaa, bbb,ccc};，则枚举常量 aaa 的值为 0，bbb 的值为 1，ccc 的值为 2。

（3）初始化枚举常量：可以为枚举常量提供初始值，这个初始值必须是一个整型常量表达式。比如 enum E{aaa=3, bbb, ccc=3+4};，则枚举常量 aaa 的值为 3，bbb 的值为 4，ccc 的值为 7；enum E{aaa=3.3};错误，因为 3.3 不是整型值；int a=3; enum E{aaa=a};错误，a 是变量；const int a=3; enum E{aaa=a};正确，int b=3; const int a=b; enum E{aaa=a};错误，因为不能在编译阶段求出 b 的值。

（4）枚举常量的值可以不是唯一的，也可以是负数。良好的编程风格是，枚举常量的值尽量不要取负数，也不要让同一枚举类型中有两个相同值的枚举常量。比如 enum E{a=–4, b, c=3, d, e=3};，则枚举常量 a 的值为–4，b 的值为–3，c 的值为 3，d 的值为 4，e 的值为 3。

（5）不能把整数值直接赋给枚举变量。注意：此行为某些编译器不一定会报错。比如 enum E{aaa, bbb}; E me;，则 me=3;是错误的，因为类型不兼容。

（6）注意：C 语言允许把与枚举常量相同的整数值赋给枚举变量，但 C++不允许。比如 enum E{aaa=1, bbb=2};，则 E me=2;在 C 语言中正确，但 C++会产生类型不兼容错误。

（7）匿名枚举：使用匿名枚举只能定义枚举常量，这时枚举常量就相当于符号常量，可以在表达式中直接使用这些枚举常量。比如 enum {aaa,bbb,ccc};，则可以这样使用：int a=3+aaa;。

（8）枚举常量的名字不能直接进行输入输出，其实枚举常量也很少用于输入输出。比如 enum E{aaa, bbb};，则 cout<<aaa;将输出值 0，而不会输出 aaa。

6. 枚举常量的算术运算及取值范围

（1）在算术表达式中，枚举常量（变量）能自动转换为 int 类型，但 int 类型不能自动转换为枚举类型，这也意味着不能把整数值直接赋给枚举变量。比如 enum E{aaa,bbb,ccc};，则 E me=1;错误，因为类型不兼容；int a=3+aaa;正确，这里会把 aaa 转换为 int 类型之后再与 3 进行计算；E me=aaa+bbb;错误，因为在对 aaa+bbb 进行算术运算时，会把 aaa 和 bbb 转换为 int 类型，计算之后的结果是 int 类型，但在赋值时并不能把 int 类型再转换为枚举类型，因此出现类型不兼容错误。

（2）严格地说，只为枚举类型定义了赋值操作符，没有为枚举定义算术运算操作，这意味着不能对枚举变量进行除了赋值以外的其他算术运算（注意：此规则只适用于枚举变量，不适用于枚举常量），但有些编译器没有这种限制，在 C++中枚举属于用户自定义类型，因此用户可以为枚举自定义自身的操作。比如 enum E{aaa,bbb,ccc}; E me=bbb; E me1=ccc;，则++me 和 me+me1 都是错误的表达式。

（3）枚举是按照枚举常量所拥有的值来比较其大小的，而不是按照声明时的顺序。比如 enum E{a, b=4}，则 a>b 为假；enum E{a=5, b=4};，则 a>b 为真。

（4）可以把整型强制转换为枚举类型，但转换之后的值必须位于枚举成员的取值范围之内，否则结果是未定义的（未定义意味着并不一定会出错，但结果并不可靠）。比如 enum E{aaa,bbb,ccc}; E me;，则 me=(E)1; 正确；而 me=(E)111;中的 111 超出了枚举成员的取值范围，因此结果是未定义的。

（5）枚举类型的取值范围：若枚举类型的所有枚举常量均为非负数，则取值范围为[0, N–1]，其中 N 是能使所有的枚举常量都位于此范围内的最小的 2 的幂；若枚举常量有负数，则取值范围为[–N, N–1]。比如 enum E{aaa=–1,bbb,ccc=8};，查找最大值，首先找到枚举常量中的最大值为 8，然后查找能容纳 8 的最小的 2 的幂的值，这个值是 16，减去 1，其最大值为 15，最小值为–16。

（6）枚举类型的大小由编译器决定，一般采用能够容纳其取值范围的整型的大小，但不会

大于 int 类型，即不大于 sizeof(int)。假设 int 类型在机器上占 4 字节，如 enum E{aaa,bbb};，则 sizeof(E)可能是 1 也可能是 4，但绝不可能大于 4。

11.6.2 联合（共用体）类型

为便于讲解，以下内容不严格区分声明与定义。

本节应注意区分共用体变量（对象）、共用体成员、共用体类型三者的区别。

1. 共用体的声明及特点

（1）共用体声明的形式为：union 共用体类型名{共用体成员列表};。

（2）共用体使用 union 关键字进行声明；共用体类型名就是一个自定义的标识符，使用共用体类型名来定义共用体变量；共用体成员是在大括号中声明的各种变量，各共用体成员之间使用逗号隔开；在声明的最后还应以分号结束。比如 union U{int a,b; float c; };U mu;，其中 union 是关键字，U 是共用体类型名，a, b, c 是共用体成员，mu 是共用体变量。

（3）共用体与类类型相似，但共用体成员共用同一个内存单元，即所有的共用体成员都位于同一个内存空间中，因此共用体具有以下特点。

① 共用体所占内存的长度为最长成员的长度。

② 共用体变量及共用体中各成员的起始地址都是相同的。

③ 同一时刻只能存放一个共用体成员的值。

2. 访问共用体成员

（1）调用共用体成员：共用体成员需要通过共用体对象使用成员访问符（即点或箭头运算符）进行访问，而不能直接对成员进行访问。比如 union U{int a; int b;}; U mu, *p;，则访问成员的方式分别为 mu.a=2; 和 p->b=3;，而 a=3;或 U.a=2; U->p=3;等都是错误的。

（2）每次访问到的值都是共用体变量中最后一次被赋值的成员的值。对新的成员赋值之后，原有的成员的值就失去作用了。因此，程序员必须清楚当前是哪个共用体成员在起作用。比如 union U{int a; int b;}; U mu; mu.a=2; cout<<mu.a<<endl; mu.b=3;，则 cout<<mu.a<<endl; 并不会得到以前的值 2。

（3）若使用非当前成员读取了不适当的当前成员的值，则其结果是不确定的，有可能产生错误。比如 union U{int a; int *p;}; U mu; mu.a=1;，则对于 cout<<mu.p<<endl;，此时指针 p 本身的值（即 p 存储的地址）为 1，因此输出的地址是 0x0000 0001（即值为 1 的地址，地址的位数根据机器而定）；这个地址中存储的是什么并不确定，也就是说，cout<<*mu.p;将可能得到意想不到的结果。注意：指针所指向的值和指针的值是不同的。

3．共用体成员的规则

（1）与类相同，共用体也可以使用访问控制符 public（公有的）、 protected（受保护的）、private（私有的），默认是 public（公有的）

（2）共用体可以有自己的构造函数和析构函数，比如 union U{U(){}~U(){}};。

（3）共用体的成员可以是函数，比如 union U{void f(){} };。

（4）共用体不能具有静态数据成员、引用成员。比如 union U{static int a; int &b;}; 错误。

（5）共用体不能具有定义了构造函数或析构函数、赋值操作符的类类型的成员，因为编译器无法保证共用体中的类对象不会被破坏，也不能保证在共用体离开作用域时可以正确地调用析构函数。比如 class A{A(){}};class B{};union U{A ma; B mb;};，其中的共用体成员 ma 是错误的，mb 是正确的。

（6）共用体不能是基类，也不能有虚成员函数。

4．共用体变量（对象）及初始化

（1）声明共用体类型之后，共用体类型名就成为一个新类型（即共用体类型）的名字，然后可以使用共用体类型名定义该类型的变量（对象）。比如 union U{int a; int b;};，则 U mu;中的 mu 就是新定义的共用体类型变量。

（2）共用体变量有 3 种定义方法。

① 使用共用体类型名直接定义，就像定义一个普通变量一样，比如 union U{int a; int b;}; E mu;。

以下两种方法是从 C 语言继承而来的。

② 定义时使用共用体类型名和 union 关键字进行定义，比如 union U{}; union U mu。

③ 在声明共用体类型时进行定义，比如 union {} mu; 或 union U{} mu;。

（3）共用体对象可以在声明时进行初始化，但只能初始化第一个成员的值，且必须用初始化表（即用花括号）初始化。比如 union U{int *p; int b;}; int a=1;，则 U mu={&a};正确；U mu={&a, 3};错误，因为初始值太多；U mu={3};错误，因为类型不符；U mu=&a;错误，应使用花括号。

（4）在共用体内声明成员时，不能对成员进行初始化。比如 union U{int a=3; int b;}; 错误。

（5）若创建共用体对象时未对共用体成员初始化，则共用体成员的值根据创建共用体对象时的作用域决定。也就是说，若是在局部作用域中创建的共用体对象，则共用体成员拥有随机值；若是在全局作用域中创建的共用体对象，则共用体成员的值为 0。

（6）共用体变量的运算：除赋值操作外，没有为共用体变量定义算术运算操作，这意味着不能对共用体变量进行除了赋值以外的其他算术运算。比如 union U{int a; int b;}; U mu, mu1;，则++mu 和 mu+mu1 都是错误的表达式。

5. 匿名共用体

（1）匿名共用体就是没有共用体类型名，同时在声明时也未定义共用体变量的共用体。比如 union {int a; int b;}; 是匿名共用体；union {int a; int b;} mu; 不是匿名共用体，这里定义了共用体变量 mu；同理，union U {int a; int b;}; 也不是匿名共用体。

（2）访问匿名共用体成员：若共用体是匿名的，因为没有其他途径访问成员，因此可直接访问共用体成员，而无须使用成员运算符（即点或箭头运算符）。

（3）匿名共用体成员的作用域与定义该匿名共用体时的作用域相同，这也意味着不能在该作用域中拥有与共用体成员同名的名字，但与类名、结构名、共用体名、枚举名是可以相同的。比如 void f(){union {int a; int b;}; a=2; int b=1; }，其中 int b=1;是重定义错误，a=2 指的是匿名共用体中的 a。

（4）匿名共用体不能有私有的或受保护的成员，也不能有成员函数。

（5）在全局作用域中定义的匿名共用体都应被声明为 static。比如 static union {int a;}; void main(){}正确；union {int a}; void main(){}错误；namespace N{union {int a}; } void main(){}错误。注意：名称空间中的名称是具有外部链接性的。

（6）若匿名共用体出现在类体之中，则共用体成员就像是类的成员一样，其使用方式就与使用类的成员相同。比如 class A{public: static union {int a;}; }; A ma;，则 ma.a=1; 是正确的，而 A::a=3;是错误的，因为 static 只是说明共用体类型是静态的，并没有说明共用体成员 a 是静态的。注意：匿名共用体成员不能是静态的。

11.6.3 位段（域）

1. 位段基础及声明

（1）位段是一种特殊的类、结构或者联合的数据成员。这种数据成员的长度可以被设置为以位为单位。

（2）声明位段：声明方式与声明普通数据成员类似，但需要在标识符之后加上一个冒号“:”，并在冒号后面指定一个整型常量表达式的值，这个值指出了该数据成员所占据的字节位数。比如 class A{unsigned int a:4;}; 就声明了一个位段 a，共占据 4 位的大小，因此 a 的最大值是 15，即二进制数 1111。

（3）位段的数据类型必须是整型或枚举。比如 class A{float a:4;}; 错误，因为类型不能是 float。

（4）冒号后的数值必须是整型常量表达式。比如 int b=2;、class A{int a:4.2;};和 class {int a:b;}; 都是错误的；class A{int a:2+2;};正确。

（5）位段也可进行连续声明，比如 class A{public: int a:4, b:3,c;};。

（6）在同一个类中，可以混合使用位段和其他数据成员。比如 class A{ int a:4; float c;};

正确。

（7）位段的访问方式、初始化和赋值与类的其他数据成员相同，即可以通过点或箭头运算符进行访问，而且同样受限于访问控制符（public, private, protected）。

2．位段的限制

（1）位段的长度不能大于 int 类型所占字节的位数。比如，假设 int 类型占 4 字节（即 32 位），则 class A{public: int a:34;}; 是错误的。

（2）不能对位段进行取址操作，因此不能声明指向类的位段的指针。比如 class A{public: int a:4;}; A ma;，则 int *p=&ma.a;和 int A::*p=&A::a;都是错误的。

（3）位段不能是静态的。比如 class A{public: static int a:4;}; 错误。

3．位段的存储和匿名位段

（1）当编译器遇到一个字节中剩余的位不能用于存储下一个位段时，不同的编译器会有不同的做法，有些编译器可能会跨字节存储，即把下一个位段部分存储在剩余的位中，也可能把下一个位段直接存储在下一个字节中。注意：非位段的数据成员是不允许跨字节存储的。比如 class A{public: int a:4; int b:3; int c:3;};，这时前两个位段 a 和 b 占据了 7 位，这样一个字节中就剩余了一位，那么对于位段 c，编译器可能会存储一位在 a 和 b 剩余的那个字节中，其余两位存储于下一个字节中，也可能把 c 的 3 位直接存储在下一个字节中。若 c 不是位段，则会直接存储在下一个字节中。

（2）匿名位段：指的是在声明时不指定名字而只指定类型和长度的位段。匿名位段常用于字节对齐。比如 class A{public: int a:4; int : 3; int b:1; };，其中 int :3 就是匿名位段。

（3）匿名位段的长度可以为 0（注意：有名位段的长度不能为 0），长度为 0 的匿名位段表示把前后两个位段或成员分别存储在两个字节中。也就是说，位于长度为 0 的匿名位段后面的位段从下一个字节开始存储，这种用法避免了跨字节存储的问题。比如 class A{int a:7; int :0; int b:3;};，则位于长度为 0 的匿名位段之后的位段 b 从下一个字节开始存储。

4．位段注意事项

（1）最好把位段的类型设为 unsigned（无符号），因为 signed（有符号）类型的位段行为与机器有关，比如 signed 类型的最高位是否表示符号位就根据机器而定。注意：int 类型默认是 signed。

（2）若位段的长度为 1，则最好将其类型声明为 unsigned（无符号），因为 signed（有符号）类型需要使用 1 位来表示符号位，因此若是 signed 类型 1 位长度的位段，则可能会得不到想要的值。比如 class A{public: signed int a:1;}; A ma; ma.a=1;，则对于 cout<<ma.a<<endl;，有的编译器可能会输出–1，而不是想要的 1。

（3）位段的实现因编译器和机器不同而不同，所以应尽量不要使用位段。

（4）虽然位段节约了空间，但却增加了处理位段的时间。

（5）在通常情况下，可以使用整型和按位操作符来代替位段。

第 12 章

运算符（操作符）重载专题

运算符重载函数通常被简称为运算符函数。

12.1 运算符重载基本概念

1. 理解运算符重载

（1）运算符重载就是给运算符赋予新的意义。

（2）运算符重载的目的：就是让两个对象能像内置类型（如 int、float 类型等）变量那样直接使用运算符进行运算。当然，对对象进行运算的结果是什么，完全由程序员自己决定。比如对类 A 的对象 ma+mb，如果没有重载+运算符就不能进行相加，否则会出错；若对“+”运算符进行了重载，则是正确的，且更容易让人理解。当然，也可以在类中定义一个对象间相加的函数来代替运算符重载的功能。比如 ma.add(mb)表示调用函数 add()实现 ma 和 mb 两个对象相加，但此语句没有 ma+mb 容易理解。

（3）运算符重载函数的形式：返回类型 operator 运算符（形参列表){};，其中 operator 是关键字。比如“+”运算符重载函数的形式为：A operator +(A ma, A mb){...}，其返回类型为类类型 A，并有两个类型为 A 的形参 ma 和 mb。

（4）理解运算符重载函数。

① 运算符重载函数就是一个函数，拥有普通函数的性质，可以像普通函数一样进行调用，只不过其函数名为 operator 加上后面要重载的运算符而已，并有一些其他限制及其特有的调用方式。比如 A operator +(A ma, A ma1){}，该运算符重载函数的函数名为 operator +，其调用方式为 operator +(ma1,ma2);，当然也可以使用特有的方式进行调用，即 ma1+ma2;。

② 虽然运算符重载函数的功能完全可以使用一个普通函数来代替，但是使用运算符重载函数更直观易懂。比如 A operator +(A ma, A ma1){}的功能完全可以使用 add(A ma, Ama1)函数来代替，但它只能这样调用 add(ma1, ma2)，而运算符重载函数可这样调用 ma1+ma2。

③ 理论上讲，可以重载运算符实现任意功能，但不推荐这样做。比如重载“+”运算符实现比较两个对象的大小，会让人无法理解。

2. 运算符重载函数的调用及与友元的使用

（1）运算符重载函数既可以是类的成员，也可以是独立的一个函数，还可以是类的友元。

（2）注意：相同版本的运算符重载函数，只能选择作为类的成员或者独立函数其中之一，否则在调用时会出现二义性错误（这种错误可以通过明确调用消除）。比如 class A{ public: void operator +(A ma){} };void operator +(A ma1, Ama2){} A ma1, ma2;，则 ma1+ma2 会出现二义性错误，但 ma1.operator+(ma2)会调用运算符成员函数。

（3）运算符重载函数作为类的成员时的调用方式：当使用普通运算符的形式（即 *m*+*n* 的形式）调用类中定义的运算符重载函数时，最左边的对象是调用运算符重载函数的对象，这时隐含的 this 指针指向调用该函数的第一个对象，因此最左边的操作数必须是重载了运算符函数的类的对象，否则将出错。比如 class A{public: A operator +(A ma){}}; A ma1, ma2;，则 ma1+ma2 和 ma1.operator +(ma2)都是正确的调用方式，其中 ma1+ma2 表达式中最左边的对象 ma1 是调用运算符重载函数的对象，这时 this 指针指向的对象是 ma1，而最右边的对象 ma2 将被作为参数传递，即 ma1+ma2 会转换成 ma1.operator +(ma2)。而对于 1+ma1 和 2.2+ma1，因为最左边的操作数不是类 A 的对象，它们会被转换成 1.operator+(ma1)和 2.2.operator+(ma1)，可以明显看出是错误的。

（4）友元运算符重载函数或独立的运算符重载函数的调用方式：当调用类的友元运算符重载函数或独立的运算符函数时，ma1+ma2 会转换为显式的调用方式。比如有友元运算符重载函数或独立的运算符重载函数 A operator +(A a, A b){}，则 ma1+ma2 与 operator +(ma1,ma2)相等，且前者会被转换为后者，前者的第一个对象传递给第一个参数，第二个对象传递给第二个参数。

（5）运算符重载函数作为类的成员、友元或者独立于类的区别：当运算符重载函数作为类的成员时，其参数会比作为友元或者独立于类的运算符重载函数少一个，若是需要两个操作数的运算符，则只需一个参数；若是需要一个操作数的运算符，则没有参数。

（6）何时需要运算符重载函数作为独立的函数或者类的友元：此情况通常出现在对一个内置类型和对象进行操作的时候。比如 class A{ public: void operator +(A ma){} }; A ma1;，则 1+ma1 错误，因为数值 1 不是类 A 的对象。此时运算符重载函数就应定义为一个独立的函数，而是否需要声明为友元，则视是否需要访问类的私有成员来定，若要访问则声明为友元，否则不声明为友元也可以。比如 class A{}; void operator +(int i, A ma2){}，则 1+ma1 就是正确的，这时会转换成 operator+(1, ma1)。注意：交换律在此处不起作用，也就是说，ma1+1 是错误的，要想让 ma1+1 正确，就必须再次重载运算符，且将两个形参对调，即 void operator +(A ma1, int i){}。

示例 12.1：理解运算符重载函数

```
class A{public:
   int a,b;
   A(){a=0;b=0;}
   A(int i,int j){a=i;b=j;}
   A operator+(A ma){  //当运算符重载函数作为类的成员时，其形参比作为普通函数时要少一个
      A t;
      t.a=this->a+ma.a; //对于 m+n 形式的调用，this 指针指向的是调用该函数的第一个对象
      t.b=b+ma.b;
      return t;}
   friend void operator+(A ma,int i);  //是否需要声明为友元，视该函数是否需要访问该类的私有
                        //成员来定，当运算符重载函数作为类的友元或普通函数时要比作为成员多一个形参
   };    //类 A 结束
//A operator+(A ma1,A ma2){...}   //此函数与运算符成员函数 operator+(A ma) 是相同的版本，当调用
                                  //该函数时会出现二义性错误，因此选择此版本与成员函数版本其中之一
void operator+(A ma,int i)    //定义友元运算符重载函数
   {i++; }     //可重载运算符实现任意功能，但不推荐这样做
void operator+(int i, A ma){ }  // 此运算符重载函数可实现 3+m 的调用形式
void main(){ A m(1,2),n(3,4),k;
   k=m.operator+(n);  //此语句与 k=m+n 等价。调用类成员运算符重载函数 A operator +(A ma)
   k=m+n;             //此处 m+n 会转换成 m.operator+(n)。this 指针指向的是调用该函数的第一个对象 m
   m+3;               //此语句会转换成 operator(m,3)，调用的是友元运算符重载函数，调用时第一个对象 m
                      //传递给第一个形参，第二个对象会传递给第二个形参
   3+m;}          //正确，调用全局运算符函数 void operator+(int i, A ma)
```

3．运算符重载函数的特点

（1）用于内置类型的运算符的含义不能被改变，因此它不能被重定义。比如 int operator+(int , int);错误，这里试图对整型的“+”运算符进行重定义。

（2）运算符重载函数必须至少有一个类或者枚举类型的形参，这个规定可以防止程序员运用运算符重载改变内置类型的含义。注意：当运算符重载函数作为类的成员时可以没有类或者类类型的形参，因为这时有一个隐形的形参，就是调用该函数时最左边的对象。比如 void operator+();int operator +(int ,float)错误；class A{public: void operator+(int i){}}; 正确。

（3）被重载的运算符的优先级、结合性、操作数的个数不会改变。比如“+”运算符要求有两个操作数，则重载的“+”也必须要有两个操作数，因此 A operator+(A ma, A ma1, A ma2);是错误的，操作数的个数过多。

（4）运算符重载函数不能使用默认实参，但函数调用运算符 operator()是个例外，因为默认实参会改变形参的个数，这就意味着间接改变了操作数的个数。比如 int operator+(A ma, int i=1){};错误。

（5）被重载的运算符将不再具有短路求值的特性，并且也不会保证操作数的求值顺序。比如重载“&&”“||”等运算符时，运算符两边的操作数都会进行求值，而且不会对求值的顺序做

规定，因此不建议重载“&&”、“||”等逻辑运算符和逗号运算符。

（6）不能重载自创的运算符，比如 void operator **(A ma){}是错误的。

4．运算符重载函数对运算符的限制

（1）只能重载下面列出的运算符，其中“+”“-”“*”“&”既可以是一元运算符，也可以是二元运算符。

+ - * / % ^ & | ~ ! , = < > <= >= ++ -- << >> == != && || += -= /= %= ^= &= |= *= <<= >>= [] () -> ->* new new[] delete delete[]

（2）下面的运算符不能被重载。

.（成员运算符 .*（成员指针运算符） ::（作用域解析运算符） ?:（条件运算符） sizeof typeid

（3）必须作为类成员的运算符如下。

=（赋值运算符） []（下标运算符） ()（函数调用运算符） ->（成员访问运算符）

12.2 运算符重载示例

下面会大量使用引用，引用是被引用对象的别名。比如 int i=1; int &p=i;，则引用 p 就是变量 i 的别名。

1．重载运算符时的分析

（1）重载运算符时应分析运算符所要求的操作数的特点（即分析运算符重载函数的形参），比如操作数的个数、操作数本身是否会被修改等。

（2）还应分析运算符返回的值。比如返回左值，就应返回某个对象的引用；返回右值，则应返回一个该类类型的对象或其他值；返回 const 类型的值，则意味着该值不能被修改；返回该类类型的对象可实现连续操作。

2．重载二元运算符“+”

（1）重载二元运算符时有“内置类型+对象”这种形式的表达式，这时应把该运算符重载函数定义为非类的成员。因为类中的成员运算符重载函数要求最左边的操作数必须是调用该函数的对象

（2）当把二元运算符重载函数作为成员时，其形参应比作为普通函数时少一个，即只需一个形参便可。

（3）“+”（加）运算符并不会改变两个操作数的值，因此加运算符重载函数的形参应是 const 类型。形参是引用还是非引用，对于重载的加运算符没有多大影响。

（4）进行加法操作后的结果是右值（即不能对加法结果进行赋值）。为了保持加法的特性，建议不要返回引用，因为返回引用意味着能对结果值进行赋值或更改。

（5）加运算符的结果会产生一个新值，且这个值应被返回，建议使用一个局部对象存储这个结果，并返回这个局部对象。注意：不能返回局部对象的引用，否则是错误的。

（6）加运算符返回该类类型的对象，则意味着可以对其进行连续相加。比如 A operator+(A ma1, A ma2){return ma1;} A ma1,ma2,ma3;，则 ma1+ma2+ma3 的形式是正确的。首先 ma1+ma2 的结果是类 A 类型的对象，然后使用该对象调用重载的加运算符，与 ma3 进行相加；若加运算符返回的不是该类类型的对象，则不能进行此种形式的连续相加。

（7）若不需要对类的私有成员进行访问，则不必把运算符重载函数声明为类的友元。

（8）其他二元运算符与此类似，这里不多做讲解。

示例 12.2：重载二元运算符“+”

```
#include <iostream>
using namespace std;
class A{public: int a,b; A(){a=0;b=0;} A(int i,int j){a=i;b=j;}
    //1. 把“对象+对象”的运算符+重载为内置类型
    const A operator +(const A &j)  //形参为 const 类型，以防止重载的“+”运算符修改操作数
                                    //返回类型不能是引用，因为加操作后的结果是右值
       {A t;                        //定义一个临时对象 t，以防止修改操作数的值
       t.b=this->b+j.b;             //this 指针指向调用该函数的第一个对象
       t.a=a+j.a;
       return t;}                   //将相加的结果存储在局部变量中，并返回
    friend const A operator +(const A &j,const int i);//把“对象+内置类型”的运算符声明为友元
    //const A  operator +(const A &j)
       //{b=b+j.b;return *this;}   //对于“+”运算符不应改变被加的操作数的值，不推荐这样做
    };  //类 A 定义结束
    const A operator +(const A &j, const int i){   //2. 实现“对象+内置类型”的友元函数的定义
       A t;    t.b=j.b+i; t.a=j.a+i; return t;}
//3. 将“内置类型+对象”的运算符重载为非成员函数。此种情形不能被定义为类的成员运算符重载函数，因为成员
//运算符重载函数要求第一个操作数必须是对象，但这里第一个操作数要求是内置类型而不是对象
const A operator +(const int i,const A &j) { A t; t.b=j.b+i; t.a=j.a+i;  return t;}
void main()
{   A m(1,2),n(3,4),k;
    //1.“对象+内置类型”，调用友元函数 operator +(const A &j,const int i)
    k=m+3;                  //该语句会转换成 operator +(m,3)
    cout<<k.a<<k.b<<endl;  //输出 45
    k=operator +(m,3);     //此语句与 k=m+3 等价。这是显式调用运算符重载函数的形式
    cout<<k.a<<k.b<<endl; // 输出 45
    //2.“对象+对象”，调用类成员运算符重载函数 operator +(const A &j)
    k=m+n;      //该语句会转换成 m.operator +(n)。最左边的对象 m 是调用成员运算符重载函数的对象
    cout<<k.a<<k.b<<endl; //输出 46
    k=n.operator +(m);     //此语句与 k=m+n 等价
    cout<<k.a<<k.b<<endl;  //输出 46
```

```
//3. “内置类型+对象”，调用全局普通函数 operator +(const int i,const A &j)
k=4+m;                  //此语句和 operator+(4,m) 等价
cout<<k.a<<k.b<<endl; //输出 56
k=operator +(4,m);    //此语句与 k=4+m 等价
cout<<k.a<<k.b<<endl;  //输出 56
//4. 实现连续相加
k=m+n+3; //首先执行 m+n，然后返回类 A 的临时对象，
         //再用该对象调用普通函数 operator +(const int i,const A &j) 与整数值 3 相加
cout<<k.a<<k.b<<endl; } //输出 79
```

3. 重载一元运算符“++”和“--”

（1）“++”（自增）运算符会改变自身操作数的值，因此其形参应是该类型对象的引用且不能为 const 类型。若前缀自增运算符是成员函数，则不需要形参，this 指针指向的就是该对象。

（2）前缀与后缀自增运算符的区分：重载前缀自增运算符时，是按照常规的形式进行重载的。但重载后缀自增运算符时，会比重载前缀自增运算符时多出一个无用的 int 类型形参，该 int 参数不会使用，应忽略它，该参数只是为了让编译器区分自增运算符的前缀和后缀形式。在默认情况下，编译器提供值 0 作为这个形参的实参。当然也可以使用这个额外的形参，但不推荐这样做。比如重载后缀运算符“++”的友元函数形式为 operator ++(A &a, int i){}，后面的参数 int i 没有实际意义，应忽略它。

（3）前缀自增运算符的结果是左值（即可对结果值进行赋值），因此其返回类型应是修改后的自身对象的引用（引用意味着可以更改返回的对象），且不能是 const 类型。

（4）后缀自增运算符的结果是右值，且是自增之前的值，因此重载的后缀运算符应返回自增之前的自身对象的值，且应以值的形式返回，而不能是引用。

（5）若显式调用后缀运算符重载函数时，需要向该函数传递一个 int 类型的实参值，这个值是必不可少的，但它没有实际用处，只是为了告诉编译器调用的是后缀自增运算符重载函数。比如 class A{public: void operator++(int){}}; A ma;，则应这样调用 ma.operator++(9)，这时的数值 9 没有实际意义，但在调用时是必需的。

示例 12.3：重载一元运算符“++”和“--”

```
#include <iostream>
using namespace std;
class A{public: int a,b;   A(){a=0;b=0;}
   A & operator ++()  //定义前缀运算符“++”，其计算结果是左值，因此返回类型应是修改后的自身
                      //对象的引用
       {++a; ++b;      //对每个成员进行自增运算
       return *this;} //返回自增后自身类型的对象
   const A operator--(int i){ //定义后缀形式的自减运算符，后缀形式比前缀形式多了一个无用的 int 类
                              //型的形参。后缀自增运算符的结果是右值，因此应以值的形式返回，
                              //而不能是引用，返回 const 类型是为了防止返回的值被修改
       A t(*this);                  //使用默认复制构造函数创建一个对象，以保存自减之前该对象的值
```

```
        --a; --b;
        cout<<"i="<<i<<endl;        //无用的 int 类型形参的值可以使用，但不推荐这样做
        return t;}                  //返回自减之前的对象
    };  //类 A 结束。
const A operator ++(A &j, int i)  //定义非成员形式的后缀自增运算符函数，因为该运算符会改变原来
                                  //对象的值，因此必须把第一个形参声明为引用。同样，后缀形式
                                  //会比前缀形式多一个无用的 int 类型的形参
   {A t(j);                       //使用默认复制构造函数创建一个对象，以保存自增之前该对象的值
   ++j.a; ++j.b;                  //对操作数 j 的成员进行自增计算
   return t;}                     //返回自增之前的对象
void main(){A n,k;
   ++k;      //调用前缀形式的类成员自增运算符函数，该语句等价于 m=k.operator++()，但
             //operator ++(k) 是错误的，因为没有接受一个实参的自增运算符函数
   cout<<k.a<<k.b<<endl;          //输出 11，可以看到前缀自增运算成功
   k.operator ++ ();              //显式调用前缀形式的类成员自增运算符函数。同++k
   cout<<k.a<<k.b<<endl;          //输出 22
   n=k++;              //调用后缀形式的非成员运算符函数，该语句等价于 n=operator ++(k,1)，其
                       //中第二个实参是没有意义的，只是为了让编译器区别是前缀还是后缀形式
   cout<<n.a<<n.b<<k.a<<k.b<<endl; //输出 2233，可以看到成功实现后缀自增
   n= operator ++(k,1);            //显式调用后缀形式的非成员自增运算符函数。同 n=k++
   cout<<n.a<<n.b<<k.a<<k.b<<endl;  //输出 3344
   k--;}//输出 i=0，在默认情况下，编译器会给后缀自减（增）运算符多余的形参提供整型值 0 作为实参
```

4．重载赋值运算符“=”

（1）注意：重载赋值运算符和[]、()、->运算符时必须定义为类的成员函数。

（2）注意：如果程序不提供显式的赋值运算符，则系统会提供一个默认的赋值运算符。

（3）什么时候重载赋值运算符：当类中含有指针成员时，一般都要重定义类的赋值运算符。

（4）重载赋值运算符时，应有处理对自身赋值（即 m=m）的情况，可使用 this 指针进行处理。比如 const A & operator=(A &j){if(this==&j) return *this;}，即比较当前调用赋值运算符函数的对象的地址和被赋值的对象的地址，如果地址相等则说明是同一个对象。

（5）赋值运算符的结果是左值，也就是可以对其进行赋值，且结果值是赋值后的左操作数的新值，因此重载赋值运算符应返回左操作数对象的引用。注意：因为返回的左值是可被修改的，所以返回类型不能是 const。

示例 12.4：重载赋值运算符“=”

```
class A {public: int b; A(){b=1;} A(int i){ b=i;}
   A & operator =(const A & j) //重载赋值运算符必须是类的成员，且应返回一个类的对象的引用
     {if( this==&j) return *this; //使用 this 和 j 的地址来检查是否是对自身赋值，如果调用赋值运
                                  //算符函数的地址和被赋值的对象的地址相等，则说明是同一个
                                  //对象，就返回当前对象
      b=j.b;
      return *this;} };           //返回左操作数对象的引用，类 A 结束
//const A & operator =(const A & j) {}    //错误，重载的赋值运算符函数必须是类的成员函数
```

```
int main()
{    A m(2); A n; n=m;cout<<n.b;
     n=n;  //对象对自己赋值的情况
     cout<<n.b;   }
```

5. 重载 I/O 运算符

（1）若类没有为该对象重载 I/O 运算符，则不能直接对其进行输入输出。比如 A ma;，则 cout<<ma;是错误的。

（2）I/O 运算符应为非类成员，若 I/O 运算符被定义为成员（此行为并非错误），则意味着在使用该运算符时，其左操作数应是类对象，而不是 I/O 对象，这与使用习惯相反，即形如 ma<<cout 的形式，而不是常规的 cout<<ma 的形式。比如 class A{public: ostream& operator<<(ostream& out){...}};，则表示在调用该 I/O 运算符时，要使用 ma<<cout 的形式，因此应把 I/O 运算符定义为非类成员，即形如 ostream& operator<<(ostream& out, const A& ma){...} 的形式。当然，不能把该运算符作为 ostream 类的成员，因为 ostream 是标准库中的类，不能修改标准库。

（3）重载输出运算符的形式：ostream& operator<<(ostream& out, const A& ma){...}。

① 为了与使用习惯相同，I/O 运算符函数的第一个形参应是 I/O 类型，输出应是 ostream 类型，输入应是 istream 类型；第二个形参应是该类的类类型。

② 重载输出运算符时，第一个形参应是 ostream 类型的非 const 引用。因为输出流会改变流的状态，因此应是非 const 的；还因为不能复制 ostream 对象，因此必须是引用。

③ 重载输出运算符的第二个形参可以是 const；类型的引用，使用引用可避免复制实参产生开销，使用 const 是因为输出操作不会改变该对象。

④ 重载输出运算符的返回类型应是一个 ostream 引用。首先，不能复制 ostream 对象，因此只能返回引用；其次，返回 ostream 类型的引用还可以进行连续输出，即形如 cout<<ma<<ma1 的输出。

（4）重载输入运算符的形式：istream& operator>>(istream& in, A& ma){...}。

① 重载输入运算符的第一个形参应是 istream 类型的非 const 引用，其原理同输出运算符。

② 重载输入运算符的第二个形参应是该类类型对象的非 const 引用，此处使用引用是因为输入运算符要改变该对象的值。

③ 重载输入运算符的返回类型应该是一个 istream 引用。

④ 对于重载输入运算符要注意，在程序中应有处理输入错误的语句，还应有处理流状态的语句。

示例 12.5：重载 I/O 运算符

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

```
class A {public: int a,b; A(){a=0;b=0;} A(int i,int j){a=i;b=j;} };   //类A结束
//I/O运算符应为非类成员，以与使用习惯相同
ostream&   //返回类型应是一个ostream引用，这样可进行连续输出。注意：不能复制ostream对象
operator<<(ostream& out, const A& ma){//第一个形参应是非const引用，因为输出流会改变流的状态
   out<<ma.a<<ma.b; return out; }
istream& operator>>(istream& in, A& ma){   //第二个形参是非const引用，因为输入运算符要改变
                                           //该对象的值
   cin>>ma.a>>ma.b; return in;}
void main(){A ma(2,3),ma1;
   cout<<ma<<endl;
   cin>>ma1;     //此处应输入正确的值，因为本例重载的输入运算符未作输入错误处理
   cout<<ma1<<endl;}
```

6. 重载下标运算符“[]”

（1）重载下标运算符的难点：下标运算符的结果可以出现在赋值运算符的左、右两边，这种情况可通过返回引用来实现。比如 string s="abc", d;，则 d[2]=s[2];是正确的。

（2）重载下标运算符需要一个形参，该形参用作下标运算符中的下标值，因此应为 int 类型。当然，使用其他类型也可以。

示例 12.6：重载下标算符“[]”

```
#include <iostream>
using namespace std;
class A
{public: char *p; int s;          A(){p=0;s=0;}
   A(char *i){
      s=strlen(i);              //判断传递进来的C风格字符串的长度
      p=new char[s];            //为指针动态分配内存
      strcpy(p,i); }            //将传递进来的字符串的值复制到指针p中。注意：不能使用p=i进行赋值
   char& operator[](int i){ //重载下标运算符必须是类的成员，返回引用可以达到下标运算符
                             //的结果既可是左值又可是右值的目的
      if(i>=0&&i<=s)         //判断是否越界
      return p[i];
      else {cout<<"越界"<<endl;  //越界后应抛出异常，本例只对越界作简单处理
   return p[0];} } };            //越界时返回数组的第一个字符。类A结束
void main()
{ char c[4]={'f','g','h'};
   A ma1(c);                     //此操作可能会出错，因为c不是以'\0'结束的
   //A ma2('a');                 //错误，本例不支持单个字符的操作
   A ma("abcde"); cout<<ma[0]<<endl;
   ma[0]='e';          cout<<ma[0]<<endl; //验证下标运算符可作为左值
   ma[0]=ma[1];        cout<<ma[0]<<endl; //验证下标运算符的结果既可是左值又可是右值
   cout<<ma[7]<<endl; }     //验证对越界情况的处理
```

7. 重载成员访问运算符“->”和解引用运算符“*”

（1）注意，.（点成员运算符）、.*（成员指针运算符）、::（作用域解析运算符）不能被重载，

但->（箭头运算符）可以被重载。

（2）“->”运算符必须是类的成员。

（3）解引用运算符不需要作为类的成员，只需一个类类型的形参即可。同时为了保持指针的特性，对该运算符应返回引用。

（4）重载解引用运算符比较简单，只需返回某种类型的引用即可。比如 class A{public: int a; int &operator*(){return a;}}; A ma;，则*ma=3;就相当于 ma.a=3;，因为*ma 调用重载的解引用运算符之后返回的就是该类的成员 a。同理，解引用运算符也可以返回类类型的引用。

（5）重载“->”运算符的目的是为了让某个类的对象看起来像类类型的指针一样。重载“*”和“->”运算符一般用来实现智能指针。比如 A ma;，若重载了“->”运算符，则可以像定义了一个指针一样来使用 ma，如 ma->a;，但要注意 a 并不一定是类 A 的成员。

（6）重载“->”运算符的规则。

① 基本规则。

- “->”运算符只需要一个操作数，因为必须作为类成员，所以不接受形参。
- “->”运算符是一元运算符，虽然它看起来像二元运算符，即左操作数是对象，右操作数是成员名，但其右操作数并不是表达式，只是一个对应的类成员的标识符，而标识符是不能传递给函数的。
- 重载的“->”运算符必须返回类类型的指针，或重载了“->”运算符的类类型的对象，否则使用该运算符时会发生错误。

② 调用重载的“->”运算符，假设调用形式为：mb->f();。

- 常规调用：若 mb 是一个类 B 类型的指针，指向类 B 的类对象，且 B 具有成员函数 f，则编译器会使用内置的“->”运算符来调用类 B 的成员函数 f()。
- 若 mb 是一个类 B 类型的对象，且定义了 operator->，则调用这个重载的“->”运算符函数。若类 B 未定义“->”运算符函数，则程序出错，因为对象访问类的成员应使用“.”运算符，而不是“->”运算符。

③ 调用“->”运算符函数的规则（递归循环调用）：重复检测“->”的左操作数是指针还是对象，若是指针，则检测结束；若是类的对象，则调用该对象所属类中重载的“->”运算符函数，然后再检测该函数返回的结果是指针还是对象，若是指针则结束，若是对象则继续重复以上步骤，直到重载的“->”运算符函数返回的是指针或没有重载的“->”运算符函数而出错。注意：其他运算符重载函数不会执行递归循环调用，这是重载的“->”运算符函数的显著特点。比如 class A{public:int a; A operator +(A ma){return ma; } A operator ->(){A ma; return ma;}}; A ma;，则 ma->a;会因无限循环调用类 A 重载的“->”运算符函数而出错，而 ma+ma+ma+ma;不会出错。

（7）示例如下：

```
class C{public:void f(){}  };
```

```
class B{public:C mc; void f(){} C* operator->(){return &mc; } };
class A{public: B mb; void f(){}B operator->(){return mb; }};
```

① B mb; B*pb=&mb;

- pb->f()，首先检测到左操作数 pb 为指针，结束检测，使用内置的“->”运算符调用类 B 的成员函数 f。
- 调用 mb->f()，首先检测到 mb 的类型为对象，执行类 B 中的 operator->()重载函数，该函数返回类 C 类型对象的指针，因为“->”运算符的左操作数是指针，因此结束检测。然后使用返回的指针调用内置的“->”运算符访问类 C 的成员函数 f 因此 mb->f()调用的是类 C 的成员函数 f，可见重载“->”运算符之后，mb 对象看起来就像一个类类型对象的指针。
- mb.operator->()->f()是 mb->f()的显式调用形式，它们都是调用类 C 的成员函数 f。
- 此处的显式调用与其他运算符有些不同，其他运算符会把右操作数 f 作为运算符的参数传递给运算符函数，但重载的“->”运算符并没有把右操作数 f 作为函数的参数，这里 f 只是一个对应的类成员的标识符。

② A ma;

- 调用 ma->f()，首先检测到左操作数 ma 的类型为对象，因此调用类 A 中重载的“->”运算符，即变为 ma.operator->()->f()。因为类 A 中的 operator->()函数返回类类型 B 的对象，因此语句变为 mb->f()，其中假设 mb 为临时对象的名称。然后再次检测到“->”运算符的左操作数是对象，因此又接着调用类 B 中重载的“->”运算符。因为类 B 中的 operator->()返回的是指向类 C 类型的指针，因此语句变为 pc->f()，其中假设 pc 为类 C 返回的临时指针对象的名称。再次检测“->”运算符的左操作数，此时为指针，结束检测，使用内置的“->”运算符调用类 C 的成员函数 f，因此 ma->f()实际上调用的是类 C 的成员函数 f。若 mb.operator->()返回的仍是一个类类型的对象，则重复以上步骤，直到返回一个类类型的指针，或此类没有重载“->”运算符而出错。
- ma.operator ->().operator ->()->f()是 ma->f()展开后的显式调用形式，最终调用的是类 C 的成员函数 f。
- 可见，若重载的“->”运算符返回的是类类型的对象，则该对象所属的类必须定义了“->”运算符，否则会出错。

8. 重载函数调用运算符“()”与函数对象（仿函数）

（1）重载函数调用运算符的目的就是让通过类类型的对象调用函数调用运算符时，使该对象看起来就像一个普通的函数调用一样。比如 A ma; ma(3,4);，其中 ma 虽然是类类型的对象，但是其形式却类似普通函数的函数调用。

（2）一般把定义了函数调用运算符的类称为函数对象或仿函数。因为该对象虽然是类类型

的，但其行为却类似于普通函数，具有函数和对象双重性质。

（3）函数调用运算符“()”必须是类的成员。

（4）函数调用运算符可以有任意数目和类型的形参。可以重载多个版本的函数调用运算符，只要它们的形参个数或类型不同即可。

（5）重载的函数调用运算符的返回值可以是任意类型。

（6）调用函数调用运算符和创建该类对象不会产生冲突。比如 A ma(3); ma(4);，前者是使用构造函数创建对象 ma，后者是调用重载的函数调用运算符。

（7）一般情况下，若需要封装某个类以实现某种功能时，就会重载函数调用运算符。比如封装类 A 实现两个整数相加，这时就可以重载函数调用运算符。比如 class A{public: int operator()(int a, int b){return a+b;}}; A ma，则 ma(3,4)便可实现两个整数相加。

（8）在大多数情况下，普通函数可以代替重载的函数调用运算符的功能，函数调用运算符一般用在泛型算法或通用容器之中作为实参使用。有关泛型算法或通用容器本书不进行讨论。

（9）读者以后看到类似于“对象（参数表)”的函数调用，就知道该对象所属的类重载了函数调用运算符。

12.3 转换函数和重载解析

12.3.1 转换函数

1. 转换函数基础及形式

（1）转换函数的作用是把类类型转换为其他类型，比如可以把自定义类类型 A 转换为 int、double 类型等，也可以把类型 A 转换为类型 B。

（2）转换函数和单形参构造函数都具有类型转换的功能，它们的转换是互补的，其中单形参构造函数只能把内置类型转换为类类型，而转换函数则是把类类型转换为内置类型。关于单形参构造函数详见 9.1.1 节。

（3）转换构造函数：即单形参构造函数，因为它也具有类型转换的功能，因此也可称为转换函数。但为了与本章的转换函数相区别，把单形参构造函数称为“转换构造函数”。

（4）转换函数一般使用类来实现某一内置类型的功能，此时可以避免重载不必要的运算符。比如由于内置类型无法检测整数溢出的情况，所以就可以使用类 A 来代替内置的整型并增加检测溢出的功能。这时由于类 A 代替了整型，因此就必须支持整型的所有运算，包括算术、逻辑、关系等运算。若没有转换函数，类 A 就需要重载很多运算符；若在类 A 中使用转换函数，就不必重载这些运算符了，此时只需把类 A 类型的对象转换为整型后再进行运算即可。比如 ma+3，

若 ma 有转换函数，则只需把 ma 转换为 int 类型，然后再与 3 相加即可；若无转换函数，则必须调用重载的“+”运算符。同理，ma>3 也只需调用转换函数即可，若无转换函数，则要重载“>”运算符。

（5）转换函数的形式：operator 目标类型(){};。

① operator 是关键字。

② 转换函数必须是类的成员函数。

③ 目标类型：表示该类类型转换后的类型。目标类型可以是内置类型、类类型或使用 typedef 命名后的新类型，但不能是数组或函数类型，指向函数或数组的指针可作为目标类型。

④ 若“目标类型”为复杂类型，则建议使用 typedef 重命名该类型。

⑤ 转换函数不能有任何形参。比如 class A{public: operator int(int i);};错误。

⑥ 转换函数不能有返回类型，但转换函数必须返回一个指定类型的值（否则会出错），这个值作为整个函数的值被返回；返回的值的类型就是“目标类型”，当然也可返回任何值，但其类型最终都会被转换为“目标类型”。比如 class A{public: operator int(){return 2.2;}}; A ma;，则 ma+3 首先调用转换函数把 ma 转换为 int 类型，但因为转换函数返回的值是 double 类型的，因此还应进行一次标准转换，把 2.2 转换为转换函数所要求的 int 类型，然后把转换后的值作为转换函数的结果值，最后结果为 2+3。

⑦ 为方便说明，以下示例只有转换函数的声明，没有完全的定义。

```
class A{public: operator int();}; //正确，把类 A 类型的对象转换为 int 类型
class B{}; class A{public: operator B();}; //正确，把类型 A 转换为类型 B
typedef int T[3]; class A{public: operator T();}; //错误，不能转换为数组类型
typedef void F(); class A{public: operator F();};//错误，不能转换为函数类型
typedef int *P;  class A{public: operator P();}; //正确，转换为指向 int 类型的指针
typedef void (*p)(); class A{public: operator p();}; //正确，转换为指向函数的指针
class A{public: operator const int *();}; //正确，转换为指向 const int 类型的指针
typedef int (*T)[4]; class A{public: operator T();}; //正确，转换为指向数组类型的指针
class A{public:operator int[3]();}; //错误，此处会把 int[3]()看作一个类型，而不是
                                    //返回 int[3] 类型的转换函数，这也是建议使用
                                    //typedef 重命名复杂类型的原因
```

（6）记住：转换函数就是一个函数，拥有普通函数的性质，但也有其特殊性。

（7）转换函数的函数名为“operator 目标类型”，与重载的其他运算符相同，只不过它重载的是“类型”，在显式调用转换函数时应使用此处的名称。比如 class A{public: operator int(){return 2;}}; A ma;，则显式调用方式为 ma.operator int()。

（8）转换函数的别名：因为转换函数是以 operator 开始的，因此又被称为“类型转换运算符函数”；又因为转换函数是重载函数，因此也被称为“类型转换重载函数”；还因为转换函数用于类类型的转换，因此又被称为“类类型转换函数”或“类型转换函数”。

（9）用户定义的转换：因为转换函数或转换构造函数是由用户自己定义的，因此这两种转

换通常被称为用户定义的转换。

2. 转换函数和转换构造函数的调用

（1）转换函数和转换构造函数既可被显式调用，也可被隐式调用，分别被称为显式（强制）类型转换和隐式类型转换。

（2）转换函数的隐式类型转换：与内置类型相同，只要有需要转换的地方，就会进行隐式类型转换，因此在表达式中，在 if、while 等语句的条件表达式中，以及函数参数传递或返回值时等会发生隐式类型转换。比如 class A{public: operator int(){return 2;}}; void f(int i){} A ma;，则 ma+2 将对 ma 调用转换函数执行隐式转换，把 ma 转换为 int 类型值；同理，if(ma)、f(ma)等操作也会对 ma 进行隐式类型转换。

（3）转换函数和转换构造函数的显式调用：比如 class A{ public: A(){}A(int i){} operator int(){return 2;}}; A ma;，则 ma=A(3)中的 A(3)就表示显式调用转换构造函数，而 int a=ma.operator int(); 则表示显式调用转换函数。

（4）转换函数和转换构造函数还可以使用传统的显式（强制）转换形式进行调用，即形如“(类型)对象”或“类型（对象)”的语法，其中“类型（对象)”的语法与显式调用转换构造函数的语法相同。比如 class A{ public: A(){}A(int i){} operator int(){return 2;}}; A ma;，则对于 ma=A(3); int a=int(ma);或 ma=(A)3; int a=(int)ma;，其中对转换构造函数 A(3)的强制转换形式与调用转换构造函数的形式相同。

（5）因为转换函数存在隐式类型转换，因此在使用转换函数时有可能会产生意外的结果。

（6）与转换构造函数不同，转换函数不能使用 explicit 关键字进行限制。

3. 调用转换函数时的一些转换规则

（1）在使用转换函数或转换构造函数时，被转换的类型不一定要与所需的类型完全匹配，其间可允许存在一些其他类型的转换（比如标准转换或提升转换等）。比如 class A{public: operator int(){return 2;}}; A ma; double d;，则对于 d=ma，首先使用转换函数把 ma 转换为 int 类型，然后再对 int 类型进行一次标准转换，转换为 double 类型。再比如 class A{public: A(int i){}}; A ma; double d;，则对于 ma=d，首先使用标准转换将 d 转换为转换构造函数所需的 int 类型，然后再使用转换构造函数转换为类类型 A。

（2）在进行了一次类类型转换（即用户定义的转换）之后，不能再进行另一次类类型转换，否则将是错误的。注意本规则的前提条件是进行了一次类类型转换之后，因此这并不表示在完成这唯一的一次类类型转换的过程中不能使用两次用户定义的转换。

示例 12.7：理解在类类型转换之后不能再进行类类型转换

```
class B{public:  operator int(){return 2;} };
class A{public: B mb; operator B(){return mb;}}; A ma; int a;
```

分析：语句 a=ma; 是错误的，此处试图进行两次类类型转换，这里类 A 没有直接从类型 A 转换为 int 类型的转换函数，因此首先需要使用类 A 中的 operator B()把 ma 转换为类型 B，然后再调用类 B 中的 operator int()把 ma 转换为 int 类型。但这种行为是错误的，因为在使用 operator B()进行转换之后，不能再进行第二次类类型转换。

示例 12.8：理解在完成一次类类型转换的过程中可以使用两次用户定义的转换

```
class B{public: operator int(){return 2;} };
class A{public: B mb; operator int(){return mb;}}; A ma; int a;
```

分析：语句 a=ma;是正确的，因为在类 A 中有直接转换为 int 类型的转换函数，因此首先调用类 A 中的 operator int()转换函数；但该转换函数返回的是类 B 类型的对象，因为类 B 类型的对象不是 int 类型的，因此转去调用类 B 中的 operator int()转换函数，把类 B 类型的对象转换为 int 类型，最后把该值作为类 A 中的 operator int()转换函数的结果。在这里要注意，虽然调用了两次转换函数，但并不是在调用完一次转换函数之后再调用一次转换函数，而是直到类 B 中的转换函数返回值时类 A 中的转换函数才算调用完成，在这之前只是调用类 A 中的转换函数的一些中间步骤而已。此处可以看到并不只进行一次用户定义的转换。

12.3.2　有转换函数时的函数重载解析

1．函数重载解析回顾

（1）函数重载解析就是指在众多的重载函数中，究竟调用哪一个重载函数。关于函数重载解析请参阅 7.4 节。

（2）最佳匹配的等级划分（最佳到最差）：完全匹配、提升转换、标准转换、用户定义的转换（类类型转换）。可见，用户定义的转换是最佳匹配当中转换级别最差的，且只要是用户定义的转换，其转换等级就是相同的。本节专门讲解关于用户定义的转换问题。

2．有转换函数时的几种匹配情形（1 对 *N*、*N* 对 1 和 *N* 对 *N*）

（1）为了便于讲解，本文把有转换函数时的匹配分为 1 对 *N*、*N* 对 1 和 *N* 对 *N* 三种情形。

① 1 对 *N*：即一个调用面对多个转换函数的情形。这种情形出现在选择哪个转换函数时比如 class A{public:operator int(){return 2;} operator float(){return 2.2;}}; void f(int a); A ma;，则 f(ma)就属于这种情形，即同一个调用面对两个转换函数。

② *N* 对 1：即多个调用面对同一个转换函数的情形。这种情形一般出现在多个重载函数面对同一个转换函数时。比如 class A{public:operator int(){return 2;}}; void f(int a); void f(float f); A ma;，则 f(ma)就属于这种情形，两个重载的函数 f 面对的是同一个转换函数。

③ *N* 对 *N*：即多个调用面对不同的转换函数的情形。这种情形发生在有多个转换函数和多个重载函数时。比如 class A{public:operator int(){return 2;} operator float(){return 2.2;}}; void f(int

a); void f(float f); A ma;，则 f(ma)就属于这种情形，其比较复杂，需要用到函数重载解析的三个步骤对其进行分析。

（2）对于 1 对 N 即一个调用面对多个转换函数的情形，其属于如何选择转换函数的问题，而不属于函数重载的情况，因此匹配的转换函数规则与函数重载是不同的。但对于 N 对 N 的情形则属于函数重载的问题，应使用函数重载解析的规则进行分析。对于 N 对 1 即多个调用面对同一个转换函数的情形，虽然其属于函数重载的问题，但它使用的是 1 对 N 时的规则。

3．如何选择转换函数

以下规则对于转换构造函数同样适用，本文以转换函数为例进行讲解。

（1）对于一个调用面对多个转换函数的情形，其主要解决如何把对象转换为目标类型，即如何选择匹配的转换函数的问题。

（2）对于 1 对 N 或 N 对 1 的情形，为了选择匹配的转换函数，一般需要经过几个转换步骤，这就形成了一个转换序列。因为转换函数是自定义的，因此本文把这个转换序列称为“用户定义的转换序列”。

（3）对于 1 对 N 或 N 对 1 的情形用户定义的转换序列的一般形式为：标准转换—用户定义的转换—标准转换。这里的标准转换指的是所有非用户定义的转换，而不能是另一个用户定义的转换，因为用户定义的转换只能进行一次。

（4）最佳转换函数的确定规则（注意理解第 3 点）：若所有转换都使用了用户定义的转换，则在转换函数之后的标准转换将是选择最佳匹配函数的依据。若是转换构造函数，则在执行转换构造函数之前的标准转换将是选择最佳匹配函数的依据。此规则与函数重载解析时的普通规则有些不同，它不是按照整个转换序列的等级作为选择最佳匹配函数的依据的。

（5）同时存在转换函数和转换构造函数时可能会出现二义性错误，因此在程序中要尽量避免出现这种情况。

示例 12.9：一个调用面对多个转换函数（1 对 N）时最佳函数的选择

```
class A{public: operator short(){return 2;} operator float(){return 2.2;}};
int s; A ma; s=ma;   //调用 short 版本
```

分析：

① s=ma;调用 operator short()的转换序列为：用户定义的转换—提升转换（即 short 转换为 int）。

② s=ma;调用 operator float()的转换序列为：用户定义的转换—标准转换（即 float 转换为 int）。

③ 因为 operator short()在调用转换函数之后的标准转换中执行的是提升转换，比 operator float()执行的标准转换等级高，因此最后 s=ma;调用 operator short()。

示例 12.10：多个调用面对同一个转换函数（*N* 对 1）时最佳函数的选择

```
class A{public: operator int(){return 2;} };
void f(int i){}void f(float f){}A ma;f(ma);  //调用 f(int)
```

分析：

① 调用 f(int)的转换序列为：用户定义的转换—完全匹配。

② 调用 f(float)的转换序列为：用户定义的转换—标准转换。

③ f(int)的完全匹配优于 f(float)的标准转换，因此调用 f(int)。

示例 12.11：同时存在转换函数和转换构造函数时的二义性

```
class A;
class B{public: B(){}B(A& ma){}};
class A{public: B mb;operator B(){return mb; } };
void f(B mb1){}; A ma; f(ma); //错误，对函数 f 的调用不明确
```

分析：

① 调用 f(ma)时会出现二义性错误。因为在把 ma 转换为类 B 类型时，既可使用类 B 的转换构造函数（本例为复制构造函数），也可使用类 A 的转换函数，而这两种用户定义的转换没有谁更优于谁，因此调用出错。

② 可以通过显式调用类 A 的转换函数进行修正。比如 f(ma.operator B())，则会调用类 A 的转换函数。

③ 注意：使用显式强制类型转换进行调用是否仍存在二义性，这就要看编译器了。理论上讲，f(B(ma))或 f((B)ma)的调用是存在二义性的，因为对于转换函数和转换构造函数这种强制类型转换都是正确的。但 VC++ 2010 能成功调用转换构造函数的版本。

示例 12.12：多个调用面对不同的转换函数（*N* 对 *N*）时的匹配

```
class A{public: operator int(){return 2;} operator float(){return 3;}};
void f(int i){} void f(float i){}
void main(){A ma; f(ma); } //错误，对函数 f 的调用不明确。
```

分析：

① 对 f(int)调用的转换序列为：用户定义的转换（调用 operator int()）—完全匹配。

② 对 f(float)调用的转换序列为：用户定义的转换（调用 operator float()）—完全匹配。

③ 可见，两个函数匹配得一样好，所以出错。

示例 12.13：多个调用面对不同的转换函数（*N* 对 *N*）时的匹配

```
class A{public: operator short(){return 2;}    operator double(){return 3;}};
void f(int i){} void f(double i){}
void main(){A ma; f(ma); }//调用 f(double)
```

分析：

① 转换函数的选择。

- f(int) 调用 operator short()的转换序列为：用户定义的转换—提升转换（short 转换为 int）。
- f(int)调用 operator double()的转换序列为：用户定义的转换—标准转换（double 转换为 int）。
- operator short()的提升转换优于 operator double()的标准转换，因此调用 operator short()。
- 同理，f(double)调用 operator double()。

② 重载函数的匹配。

- 对 f(int)调用的转换序列为：用户定义的转换（调用 operator short()）—提升转换。
- 对 f(double)调用的转换序列为：用户定义的转换（调用 operator float()）—完全匹配。
- f(double)的完全匹配优于 f(int)的提升转换，因此调用 f(double)。

12.3.3 带有类类型实参和在类作用域中调用函数时函数重载解析

1. 带有类类型实参和在类作用域中调用函数时候选函数的确定方法

（1）当调用的函数具有类类型实参或位于类作用域中时，其候选函数的确定方法与普通函数有些不同，选择可行函数与确定最佳匹配函数的规则与普通函数相同。注意：确定候选函数可以知道哪些函数可组成函数重载，这是与名称解析不同的地方。

（2）带有类类型实参的重载函数候选名的确定方法：若函数调用时的实参是一个类类型对象、指向类类型的指针、类类型的引用或指向类成员的指针时，其候选函数需在以下范围内进行查找。

① 调用点可见的函数。

② 定义实参所属类的名称空间中的函数，这里要注意类实际来自于哪个名称空间。

③ 在实参所属类中声明的友元函数。

示例如下：

```
namespace N{
   class A{friend void f(A ma, int i){}; void f(N::A ma, float s){}} N::A ma;
```

f(ma,3.3)将发生二义性错误，其候选函数有：定义实参 ma 所属类的名称空间中的函数 f(N::A, float)和实参所属类中声明的友元函数 void f(A ma, int)，两个函数匹配得一样好，所以出错。

示例 12.14：理解名称空间中的成员

```
#include<iostream>
using namespace std;
namespace N{
   namespace M{
      class A;    //声明名称空间成员 A，并在名称空间 M 外进行定义
      class B;    //声明名称空间成员 B，此成员未被定义
      void f(A &ma){cout<<"A"<<endl;}    //形参必须是类 A 的引用，因为类 A 未定义
      void g(B &mb){cout<<"B"<<endl;} }  //同上，名称空间 M 结束
```

```
    class M::A{};   //在名称空间 N 中定义成员 A。
    class B{};      //定义名称空间 N 中的成员 B，此成员与名称空间 M 中的成员 B 是两个不同的成员
    void f(int i, M::A ma){cout<<"A"<<endl;}
    void g(int i,B mb){cout<<"Bi"<<endl;} }     //名称空间 N 结束
void main(){N::M::A ma; N::B mb;
    //N::M::B mb1;  //错误，名称空间 M 中的成员 B 未定义
    f(ma);        //正确，调用名称空间 M 中的函数 f。因为 ma 所属的类 A 是名称空间 M 中的成员
    //f(3,ma);    //错误，ma 所属的类 A 是名称空间 M 中的成员，不是名称空间 N 中的成员，因此名称
                  //空间 N 中的函数 f 不是候选函数
    //g(mb);      //错误，mb 所属的类 B 是名称空间 N 中的成员，因此名称空间 M 中的函数 g 不是候选函数
    g(3,mb); }    //正确，输出 Bi，原因同上
```

（3）在类作用域中调用函数时候选函数的确定方法：首先按照类作用域中的名称解析方法对调用函数的名称进行查找，一旦找到了函数声明，对候选函数的查找就立即结束，而不会再考虑其他作用域中的名称，候选函数也就是该名称空间中的函数。对于有名称空间时的名称查找，则按照与限定修饰符相反的顺序进行。比如 N::M::D::，则先查找名称空间 D，然后是 M，最后是 N。有关类作用域中的名称解析方法，请参阅第 8 章。

示例 12.15：在类作用域中调用函数时候选函数的确定方法

```
namespace N{class A {public:     void h(float c){cout<<"A"<<endl;}
        void h(int c){cout<<"B"<<endl;}
        void g(); };    //类 A 结束
    void f(int c){cout<<"C"<<endl;}
    void f(float c){cout<<"D"<<endl;} }  //名称空间 N 结束
void f(double i){cout<<"E"<<endl;}
void h(double i){cout<<"F"<<endl;}
void N::A::g(){
    h(3.3);     //发生二义性错误，候选函数是类作用域 A 中的两个函数 h，不包括全局的函数 h
    f(3.3); }  //发生二义性错误，候选函数是名称空间 N 中的两个函数 f，不包括全局的函数 f
void main(){N::A ma; ma.g();}
```

分析：

① 对于 N::A::g()中的名称查找过程是，首先查找函数 g 中的名称，然后在类 A 的作用域中查找，然后是名称空间 N 中的名称。除类作用域外，其余两个名称都应遵守先声明后使用的规则。

② h(3.3)调用时的名称查找过程是，先查找函数 g 中的名称，没找到，然后查找类作用域 A 中的名称，找到两个重载的 h 函数，至此名称查找结束，不再在外围的名称空间 N 和全局作用域中进行查找。这是在类作用域中调用成员函数时名称查找的特殊规则。因此，最后只有类 A 中定义的两个 h 函数作为 h(3.3)调用的候选函数。因为对于 h(3.3)来讲两个重载的 h 函数匹配得一样好，所以调用出现二义性错误。对于 f(3.3)调用，其名称查找过程与之类似，首先在 g 中查找，未找到，然后在类 A 中查找，未找到，继续在定义点之前向外围作用域 N 中查找，找到 N 中的两个 f 函数，名称查找结束，不再在全局作用域中查找名称。

2. 使用对象调用重载成员函数时的重载解析

（1）最佳匹配函数的确定规则：当使用对象对重载成员函数进行调用时，首先应确定候选函数，然后选择可行函数，最后按等级划分确定最佳匹配的函数。

（2）使用对象调用重载成员函数的候选函数的确定方法：因为类作用域是一个单独的作用域，因此重载成员函数的查找范围仅限于当前的类（或基类）作用域中，其他名称空间中的名称不会被考虑在内。比如 namespace N{class A{public: void f(int); void f(float);}; void f(double);} N::A ma;，则调用 ma.f(3.3) 的候选函数只包括类 A 中声明的 f(int)和 f(float)，很明显名称空间中的函数 f 不是类 A 的对象 ma 的成员函数，而 f(int)和 f(float)对于 f(3.3)来讲匹配得一样好，因此调用出现二义性错误。

（3）当调用重载成员函数时需要注意静态成员函数和 const 类对象的特殊性。

（4）当重载成员函数中有静态成员，使用类名调用函数时，其候选函数并不只有静态成员函数，还包括非静态成员函数。比如 class A{public: void f(int){} static void f(float){}};，则调用 A::f(3.3)的候选函数有 f(int)和 f(float)，因为这两个函数对 f(3.3)的调用匹配得一样好，所以 A::f(3.3)的调用具有二义性。注意：虽然不能使用 A::f(3.3)调用非静态的 f(int)函数，但其仍是候选函数，且是可行的。

（5）当使用 const 类对象调用成员函数时，需要考虑成员函数的可行性问题，也就是 const 类对象只能调用 const 成员函数和静态成员函数，其他的成员函数都是不可行的。因为静态成员函数属于整个类，不属于某个对象，所以 const 类对象可调用静态成员函数。比如 class A{public: void f(int) const{}static void f(float){} void f(double){} }; const A ma;，则调用 ma.f(3.3)的候选函数有 f(int)、f(float) 和 f(double)，但可行函数只有 f(int)和 f(float)，又因为匹配得一样好，因此调用具有二义性。再比如 class A{public: void f(int) const{} void f(float){} }; const A ma;，则调用 ma.f(3.3)的候选函数有 f(int)和 f(float)，因为 f(float)不是可行函数，因此最后调用 f(int)。注意：虽然 f(float)不是可行函数，但其是候选函数。

（6）注意：成员函数不能既是 const 的又是 static 的。比如 class A{public:static void f() const;}; 是错误的。

12.3.4 重载运算符函数时的重载解析

（1）本节讨论的规则只适用于使用运算符语法调用的情形，即形如 ma+3 这样的形式。若使用函数语法进行调用，即形如 operator+(ma,3)的形式，则使用与普通函数重载相同的规则；若使用对象调用成员函数的语法，即形如 ma.operator(3)的形式，则使用重载成员函数的重载解析规则。

（2）最佳匹配函数的确定规则：调用多个重载的运算符函数时，其重载解析规则同普通函

数重载解析规则。

（3）调用多个重载的运算符函数时，其候选函数的确定规则与普通重载函数有些不同，重载运算符的候选函数范围更大。其他如转换序列、最佳匹配等级划分、可行函数的规则与普通重载函数相同。

（4）重载运算符函数的候选函数的确定应在以下范围内查找候选函数。

① 调用点可见的运算符函数。

② 定义实参所属类的名称空间中的运算符函数。

③ 在实参所属类中声明的友元运算符函数。

④ 在左操作数所属类中声明的成员运算符函数。

⑤ 内置运算符函数。

其中的内置运算符函数形如 int operator+(int ,int); double operator+(double, double); T* operator+(T* , i); T* operator +(i, T*); 等形式，最后两个内置运算符函数的形式用于指针与整数相加。对于任何符合要求的运算，内置运算符函数都会被考虑。

（5）注意：对于运算符重载的函数匹配，是在两个形参与两个实参之间的匹配，这种匹配情形比单形参函数重载要复杂，其判定规则为最佳匹配的函数必须满足该函数的所有参数的匹配程度都不能低于其他候选函数所有参数的匹配程度。

示例 12.16：验证内置运算符函数的存在

```
class A{public:A(){} operator int(){cout<<"B"<<endl;return 23;}};
class B{public:B(){} B(int i){}};
int operator+(B ma,A ma1){cout<<"A"<<endl;return 2;}  //程序唯一重载的“+”运算符函数
void main(){A ma; 3+ma;}   //3+ma 调用出现二义性错误
```

分析：

① 在 main 函数中 3+ma 的调用具有二义性。

② 确定候选函数：唯一的全局函数 operator+(B,A)是候选函数，内置类型的 operator+(int ,int)也是候选函数。

③ 对于 operator+(B,A)，第一个参数的转换序列为：用户定义的转换（调用类 B 的转换构造函数把值 3 转换为类型 B）—完全匹配，转换序列等级为用户定义的转换。第二个参数是完全匹配的。

④ 对于内置的 operator+(int ,int)，第一个参数是完全匹配的，第二个参数的转换序列为：用户定义的转换（调用类 A 的转换函数把 ma 转换为 int 类型）—完全匹配，转换序列等级为用户定义的转换。

⑤ 可以看到，对于 operator+(B,A)的匹配情况是“用户定义的转换<>完全匹配”，而 operator+(int ,int)的匹配情况是“完全匹配<>用户定义的转换”。对于 operator+(B,A)，第二个参

数匹配得更好；对于 operator+(int,int)，则是第一个参数匹配得更好。二者没有谁比谁更有优势，所以对 3+ma 的调用具有二义性。

⑥ 虽然本例只重载了一个“+”运算符函数，但调用仍具有二义性，这说明内置的 operator+(int, int)是存在的。

示例 12.17：重载运算符函数时候选函数的确定

```
namespace N{ class A{public: int operator+(int i){cout<<"A"<<endl;return 23;}
   friend int operator+(A ma,float i){cout<<"B"<<endl;return 2;} };//类A结束
    int operator+(A ma, char i){cout<<"C"<<endl;return 22;} } //名称空间N结束
    int operator+(N::A ma,double i){cout<<"D"<<endl;return 2;}
    void main(){N::A ma; int a=1; float b=2; char c='4';    double d=5;
       ma+a;   //调用成员函数 operator +(int);
       ma+b;   //调用友无函数 operator+(A,float)
       ma+c;  //调用名称空间中的函数 operator+(A,char)
       ma+d;  } //调用全局函数 operator+(A,double)
```

分析：对于 ma+a; ma+b; ma+c; ma+d;，它们的候选函数都有 5 个，即：类 A 的成员函数 operator+(int)、类 A 的友元函数 operator+(A , float)、名称空间 N 中的函数 operator+(A, char)、全局函数 operator+(A, double)、内置运算符函数 operator+(int,int)。

12.3.5 仿函数与重载解析

（1）仿函数的难点在于与代理函数（重载的转换为函数类型的运算符函数）之间的重载解析及其所隐藏的形参。比如 typedef void F(int); class A{public:void operator ()(int){} operator F*(){}}; A ma;，则调用 ma(3)的难点是对仿函数 operator()(int)和代理函数 operator F*()之间的重载解析。

（2）仿函数与代理函数所隐藏的形参：仿函数所隐藏的形参是 this 指针，而代理函数所隐藏的形参就是函数本身。比如 typedef void F(int); class A{public:void operator ()(int){} operator F*(){}};，其中仿函数 operator()(int)有两个形参，即(*this, int)；代理函数 operator F*()也有两个形参，即(F*, int)，其中 F*表示指向 F 类型（即函数）的指针。

示例 12.18：仿函数与代理函数的重载解析

```
#include <iostream>
using namespace std;
typedef void P(int);
void f(int){cout<<"F"<<endl;}
class A{public:
     void operator()(float){cout<<"A"<<endl;}   //仿函数
     operator P*()  //代理函数，此处必须返回指针，否则会出现函数返回函数的错误。因为operator P()
                    //要求返回类型P，这个P类型是函数，而重载的运算符函数本身也是一个函数，函数不
                    //能返回函数，因此必须使返回的类型是一个指向P的指针
```

```
    {cout<<"B"<<endl;} return f; };
void main(){A ma;
    //ma(3); //错误。1. 对于operator()(float)而言，此处有两个形参(this&,float)；对于operator
             //P*()而言，也有两个形参(P*,int)
    //2. 第一个实参ma与(this&,float)中的第一个形参this&是完全匹配的，与(P*,int)中的第一个形参
    //P*需要通过自定义类型转换才能匹配（即调用operator P*()把ma转换为P*），很明显ma和this&匹配
    //得更好；而第二个整型实参3明显与(P*,int)的第二个形参int匹配得更好，因此调用存在二义性错误
}
```

12.4　重载 new/delete 运算符和定位 new/delete

（1）本章有必要讲解一下类型与内存的关系。一块连续内存的地址是没有实际意义的，其意义完全取决于类型，类型决定了怎样解释和处理这块内存。比如一块拥有 16 字节的内存，若被解释为 int 类型，假设 int 类型占 4 字节，则这 16 字节的内存可存放 4 个 int 类型数据；若被解释为 char 类型，假设 char 类型占 1 字节，则这 16 字节的内存可存放 16 个 char 类型数据。

（2）在使用重载的自定义 new 运算符函数分配内存时，返回的往往是 void*的指针，即无类型。其实返回的类型无关紧要，我们需要关心的是返回的内存大小能否足够存储所需的对象。因为类型只决定怎样解释这块内存，因此类型并不重要，可把 void*强制转换为任何类型。比如 int p[44]; class A{public: void* operator new(size_t){return p;}}; A* pa=new A;，这时 new A 调用类 A 中的 operator new 分配内存，而 operator new 分配的内存是 p 指定的，对于所分配的这块内存只需关心 p 的大小能否容纳类 A 的对象即可，而不用去关心 p 的类型是 int 还是 char，因为这无关紧要。假设 int 类型占 4 字节，则 int p[44]共分配了 4*44=176 字节的空间，显然在本例中能容纳类 A 的对象，因此没有问题。至于 operator new 返回的是 void *类型，这无关紧要，因为 void*最后被 new A 强制转换为 A*，那么就意味着在 new A 进行分配之前，这 176 字节的内存空间按照 int 类型进行解释；但使用 new A 之后，这 176 字节的内存空间就按照类型 A*进行解释，可见类型只影响程序怎样解释这 176 字节的内存，因此对使用 new 分配内存影响不大，当然在赋值时类型必须兼容。可见，使用 new 分配内存时主要应关心此内存空间能否容纳下所需的对象。对于定位 new 也是同样的道理。

12.4.1　重载 new/delete 运算符

1. 理解 new 运算符和 new 表达式

（1）与常规运算符类似，new 和 delete 被 C++实现为运算符，它们具体怎样实现我们不用去关心。记住 new/delete 就是运算符。

（2）可以对 new 和 delete 运算符进行重载，其重载方式与常规运算符相同，即重载时的函

数名为 operator new 和 operator delete，当然也会有一些其他特殊限制，详见后文。

（3）new 表达式：new 运算符也有与其相关的表达式，就像“+”是运算符，而 a+b 是使用“+”运算符的表达式一样，最常见的 new 表达式就是使用常规的 new 分配内存的方法。比如 int *p=new int;，其中 new int 就是 new 表达式。

（4）new 表达式需要完成两个功能，第一个功能是为对象分配内存；第二个功能是使用构造函数初始化内存中的对象。这两个功能是无法改变的。比如 A *p=new A（9);，首先使用 new 为类型 A 的对象分配内存，然后调用单形参构造函数初始化该对象，若无后面的括号，则使用默认构造函数初始化该对象。

（5）delete 表达式也需要完成两个功能，在使用 delete 释放内存之前，调用该类的析构函数销毁对象，然后释放内存资源。比如 A *p=new A(); delete p;，则在 delete p 之前会首先调用类 A 的析构函数销毁对象，然后释放内存资源。

（6）注意理解 new 表达式和 delete 表达式的功能，在使用 new 表达式时会自动实现该功能，但在不使用 new 表达式时有可能并不会自动实现该功能，需要我们手动实现。

2. 重载 new 运算符

（1）与常规运算符一样，虽然不能改变 new 表达式的含义，但可重载 new 运算符以控制如何分配内存。

（2）与重载常规运算符相同，重载 new 运算符也需要使用 operator 关键字，重载后的 new 运算符函数名为 operator new。

（3）既可以使用函数名直接调用 new 运算符函数，也可以使用 new 表达式调用 new 运算符函数。

（4）全局默认的内置 new 运算符函数的形式为：void*operator new(size_t){...}，它带一个 size_t 类型形参，且返回类型为 void*。用户可以在全局重定义这个内置的 new 运算符函数，也可以在类的内部重载内置的 new 运算符函数。

（5）可以直接调用全局默认的内置 new 运算符函数，比如 int *p=(int*)operator new(sizeof(int));。

（6）在全局重定义内置的 new 运算符会覆盖内置 new 运算符的功能，内置 new 运算符能为所有的类型分配内存，其功能相当强大，因此不建议重定义全局内置的 new 运算符。

（7）重载的 new 运算符函数形式为：void *operator new(size_t, ...){...};，其形参可以有多个，但第一个形参的类型必须是 size_t，返回类型必须是 void*。

（8）重载 new 运算符函数的基本规则。

① new 运算符函数的返回类型必须是 void*，即 new 运算符返回的是没有类型化的内存（即不知道内存中存储的数据是什么类型）。

② new 运算符函数的第一个形参的类型必须是 size_t，size_t 是在头文件<cstddef>中使用 typedef 重定义的类型名，其类型一般是 unsigned int。记住，size_t 只是一个类型的别名。

③ new 运算符函数既可重载为全局函数，也可重载为类的成员函数。

④ 重载为类成员的 new 和 delete 运算符函数被自动处理为静态成员函数，因此它们必须遵守静态成员函数的限制，即没有 this 指针，只能访问类的静态数据成员。new 和 delete 运算符函数作为类成员时是静态的原因很简单，在调用这些运算符时要么是在类的对象被创建之前，这时对象成员就不会拥有值；要么是在对象被销毁之后，这时对象成员的值已不存在，因此没有必要把 new 和 delete 运算符函数声明为非静态的类成员。

⑤ 可以重载多个版本的 new 运算符函数，只要它们遵守函数重载的特性（即形参个数和类型不相同）即可。但第一个形参必须是 size_t 类型的。

3. 重载 delete 运算符

（1）全局默认的内置 delete 运算符函数的形式为：void operator delete(void*){...}，它带一个 void*类型形参，且返回类型为 void。

（2）可以直接调用全局默认的内置 delete 运算符函数，比如 operator delete(p)。

（3）与 new 运算符相同，可以在全局重定义内置的 delete 运算符，但这会覆盖内置 delete 运算符的功能，内置 delete 运算符能为所有的类型释放内存，其功能相当强大，因此不建议重定义全局内置的 delete 运算符。

（4）重载的 delete 运算符函数形式为：void operator delete(void*, ...){...};，其形参可以有多个，但第一个形参的类型必须是 void*，返回类型必须是 void。

（5）重载 delete 运算符函数的基本规则：

① delete 运算符函数的返回类型必须是 void。

② delete 运算符函数的第一个形参的类型必须是 void*。

③ delete 运算符函数既可重载为全局函数，也可重载为类的成员函数。

④ 可以重载多个版本的 delete 运算符函数，只要它们遵守函数重载的特性（即形参个数和类型不相同）即可。但第一个形参必须是 void*类型的。

⑤ 重载为类成员的 new 和 delete 运算符函数被自动处理为静态成员函数。

（6）注意：若重定义了 new 运算符函数，也应重定义相应的 delete 运算符函数。

示例 12.19：重载 new/delete 运算符函数的规则

```
#include <iostream>
using namespace std;
char c[44];
class A{public: int a;     //非静态成员 a
   //重载 new 运算符函数
   void* operator new(size_t ){cout<<"A"<<endl;return c;}//重载全局默认的内置 new 运算符函数
```

```
    //char* operator new(size_t, char d){return c;}  //错误，返回类型必须是 void*
    //void * operator new(int i){return c;}          //错误，第一个形参的类型必须是 size_t
    void *operator new (size_t t, int i){return c;} //正确，可以重载多个版本的 new 运算符
    //void *operator new (float i, int j){return c;}//错误，第一个形参的类型必须是 size_t
    void *operator new (size_t t, float i)          //重载形式正确
     {//a=3;    //错误，a 是非静态成员。重载为类成员的 new 运算符函数被自动处理为静态成员函数
              //只能访问静态成员
      return c;}
//重载 delete 运算符函数
void operator delete(void* p){}              //重载全局默认的内置 delete 运算符函数
//int operator delete(void* p,char d){}      //错误，返回类型必须是 void
//void operator delete(int* p){}             //错误，第一个形参的类型必须是 void*
void operator delete(void*, int i){}         //正确，可以重载多个版本的 delete 运算符
//void operator delete(int i, int j){}       //错误，第一个形参必须是 void*
void operator delete(void*, float j){        //重载形式正确
    //a=1;              //错误，a 是非静态成员，原因与 new 相同
}};
//全局重载 new/delete 运算符函数
void *operator new (size_t t, float i) {return c;}//正确，new 运算符函数可重载为全局函数
void operator delete(void*, float j){ }  //正确，delete 运算符函数可重载为全局函数
void main(){ }
```

4. 调用重载的 new 运算符函数的方式

（1）可以使用 new 表达式和通过显式调用的方式调用 new 运算符函数。

（2）使用 new 表达式调用重载的 new 运算符函数，其重载形式必须为 void *operator new(size_t){...}且应作为类的成员函数。若 operator new 函数有多个形参，则无法使用 new 表达式进行调用，而应使用其他形式（详见定位 new 运算符部分）。

（3）使用 new 表达式调用 new 运算符函数时，new 表达式会自动处理以下事情。

① 分配内存。

② 把 new 运算符函数的形参类型 size_t 自动初始化为操作数类型的字节大小。

③ 调用相应的构造函数初始化对象。

④ 将 new 运算符函数返回的 void*类型强制转换为所需的类型。

⑤ 例如 A *p=new A;，相当于以下 3 条语句。

```
void *p=operator new(sizeof(A)); //调用 operator new 函数分配内存（对象），但不知道内存的类型
A::A();                  //调用默认构造函数初始化内存（对象）
static_cast<A*>（分配的内存）//将所分配的内存强制转换为 A*类型的指针（类型化内存），并返回该指针
```

（4）像调用普通函数一样显式调用 new 运算符函数时，不会对对象进行初始化，也不会对返回的类型和传递的参数等进行处理，因为 new 运算符函数的职责只是负责分配内存。比如 A *p=(A*)operator new(sizeof(A));，此处只负责分配内存，因此需要处理 operator new 函数返回的 void*类型（此处进行强制转换），还要处理传递给形参的字节大小（使用 sizeof(A)）。同时要注意 operator new 只是为类型 A 分配了内存，并没有对其进行初始化。这是显式调用 new 运算符

函数与使用 new 表达式调用 new 运算符函数的不同点。

5. 调用重载的 delete 运算符函数的方式

（1）可以使用 delete 表达式和显式调用的方式调用 delete 运算符函数。

（2）使用 delete 表达式调用重载的 delete 运算符函数时，其重载形式有两种，即 void operator delete(void*){...}和 void operator delete(void*, size_t){...}，其中第二种形式的第二个形参必须是 size_t 类型的且还应作为类的成员函数，第二个形参在有继承关系时会有用处。operator delete 的其他重载形式无法使用 delete 表达式进行调用，而需要使用显式调用的形式。

（3）使用 delete 表达式调用 delete 运算符函数时，delete 表达式会自动处理以下事情。

①调用指针所指向对象的析构函数销毁对象。

② 调用该对象的 delete 运算符函数释放其内存。

③ 第一个形参 void*会使用 delete 表达式之后的指针自动初始化。

④ 若 delete 运算符函数是类的成员函数，且有两个参数，则自动把第二个形参类型 size_t 初始化为第一个形参类型（即 void*）所指向对象的字节大小。

（4）像调用普通函数一样显式调用 delete 运算符函数时，其只负责释放内存，对其他事情不会进行处理，也就是不会调用析构函数销毁对象。比如 operator delete(p)，假设 p 是指向类 A 对象的指针，则此处它只负责释放内存，其他事情一概不处理，因此类 A 的析构函数不会被调用。这是显式调用 delete 运算符函数与使用 delete 表达式调用 delete 运算符函数的不同点。

（5）注意：若使用 new 表达式调用全局的 new 运算符函数，则应使用相对应的 delete 表达式调用全局的 delete 运算符函数。

示例 12.20：调用重载的 new/delete 运算符函数

```
#include <iostream>
using namespace std;
char c[44];
class A{public: A(){cout<<"A"<<endl;}     ~A(){cout<<"~A"<<endl;}
   void *operator new(size_t t){ cout<<t<<endl; return c;  }
   void operator delete(void* p){cout<<"D"<<endl;}};     //类A结束
class B{public: B(){cout<<"B"<<endl;} void *operator new(size_t, int i){return c;}};
class C{public:  ~C(){cout<<"C"<<endl;}};
void operator delete(void* p,size_t){cout<<"E"<<endl;}    //全局delete运算符函数
void main(){A ma; B mb;
    //使用new表达式调用重载的new运算符函数，其重载形式必须为void *operator new(size_t)
     cout<<sizeof(A)<<endl;
     A *pa=new A;  //使用new表达式调用类A中重载的operator new(size_t)运算符函数，
                   //其自动执行以下内容
                   //首先分配内存
                   //然后输出1，可以看到new表达式自动把形参类型size_t初始化为类类型A的字节大小
                   //接着输出A，可以看到new表达式自动调用类A的默认构造函数初始化对象
```

```
            //最后将 new 运算符函数返回的 void*类型强制转换为所需的类型 A*
A* pa1=(A*)ma.operator new(sizeof(A));//显式调用重载的 new 运算符函数。此处并没有输出 A,
//可见并没有自动使用构造函数创建对象；同时传递给形参的字节大小也需要通过 sizeof(A)的形式手动
//分配，new 运算符的返回类型也需要手动进行强制类型转换处理。这是显式调用与使用 new 表达式调
//用的区别
//B *pb=new B; //错误，new 表达式无法调用重载形式不是 void *operator new(size_t)的版本
//使用 delete 表达式调用重载形式为 void operator delete(void*)的运算符函数
delete pa;   //表达式 delete pa 自动执行以下内容
             //输出~A，可以看到此处自动调用了指针 pa 所指向对象 A 的析构函数销毁对象
             //输出 D，可以看到此处调用该对象的 delete 运算符函数释放其内存。本例中的 delete
             //函数什么也没做
             C *pc=new C;
operator delete(pc,3); //正确，显式调用全局的 delete 运算符函数，未输出~C。可见没有自动调用
              //pc 所指向对象 C 的析构函数销毁对象。这是显式调用与使用 delete 表达式调用的区别
   C *pc1=new C;
delete pc1;} //调用全局默认的 delete 运算符函数，此处不会调用全局的重载了两个形参的 delete 运
             //算符函数，若此函数是类 C 的成员函数，则会使用 delete 表达式调用该成员函数
```

6. 调用作为类成员的 new/delete 运算符成员函数

（1）若 new/delete 运算符函数被重载为类的成员函数，则在使用 new/delete 表达式时会优先选择作为成员函数的 new/delete 运算符函数进行内存分配和释放。比如 class A{public: void *operator new(size_t){ int *p=new int; return p; }};，则 A *pa=new A;会使用类 A 中重载的 new 运算符成员函数进行内存分配，而不会调用全局的 new 运算符函数。

（2）若 new/delete 运算符函数被重载为类的成员函数，这时可使用作用域解析运算符调用全局的 new/delete 运算符函数。比如 class A{public: void *operator new(size_t){int *p=new int; return p;}};，则 A *pa=::new A();表示使用全局的 new 运算符函数为类 A 的对象分配内存。

（3）若类内部重载了 new 运算符函数，则会隐藏全局的同名 new 运算符函数；若类内部的 new 运算符函数不止一个形参，而又使用了 new 表达式，这时可能会造成错误，解决办法是使用作用域解析运算符明确调用全局的 new 运算符函数。比如 class A{public: void *operator new(size_t,int){int *p=new int; return p; }};，则 A *pa=new A;错误，因为类内部的 new 运算符函数需要两个形参，类内部的 new 运算符函数隐藏了全局的 new 运算符函数。其正确形式为 A*pa=::new A;。

（4）显式调用 new/delete 运算符成员函数：使用类名和作用域解析运算符进行调用，其形式为 A::operator new(N);（注意 new/delete 运算符成员函数默认是静态的），不推荐显式调用作为类成员的 new/delete 运算符成员函数，因为在显式调用时，new 运算符函数只负责分配内存，而不负责初始化对象；delete 运算符函数只负责释放内存，而不会调用相应的析构函数销毁对象。比如 class A{public: void *operator new(size_t){int *p=new int; return p; }void operator delete(void*){...}};，则显式调用 new 运算符成员函数的方法为 A* pa=(A*)A::operator new

(sizeof(A));，但此方法无法初始化所分配的内存，对类型 A 的字节大小转换、对返回类型的处理都需要程序员自己进行。显式调用 delete 运算符函数的方法为 A::operator delete(pa)。

7．重载 new[]和 delete[]运算符

（1）重载 new[]运算符函数的形式为 void *operator new[](size_t, ...){...};，其形参可以有多个，但第一个形参的类型必须是 size_t。返回类型必须是 void*。

（2）new[]运算符函数的 size_t 参数会被自动初始化为存放 *N* 个指定对象的数组所需内存的字节大小。比如 A *pa=new A[4];，则 size_t 被自动初始化为 4 个 A 类型对象的数组所需内存的字节大小。

（3）使用 new 表达式调用 new[]运算符函数时，会自动调用默认构造函数依次初始化每一个数组元素，若指定的类没有默认构造函数，则 new 表达式出错。注意：若显式调用 new[]运算符函数，并不会自动调用构造函数初始化对象，因此没有默认构造函数不会出错。比如 A *pa=new A[4];，则会自动使用类 A 的默认构造函数创建 4 个类 A 类型的对象；而 A *pa=(A*)operator new[](4);，则不会调用构造函数创建对象。

（4）重载 delete[]运算符函数的形式为 void operator delete[](void*, ...){...};，其形参可以有多个，但第一个形参的类型必须是 void*。返回类型必须是 void。

（5）delete[]运算符函数的 void*形参会被自动初始化为被删除数组存储区的起始位置。比如 delete [] pa;，则 void*被自动初始化为 pa 所指数组存储区的起始位置。

（6）使用 delete 表达式调用 delete[]运算符函数时，会依次调用该类的析构函数销毁数组的每一个对象元素，然后再调用 delete[]运算符函数释放内存。

（7）注意：在使用 delete 表达式调用 delete[]运算符函数时，应使用数组语法，即 delete[] p;，而 delete p 则是错误的。

（8）与 delete 运算符函数相同，若 delete[]是类的成员函数，则使用 delete 表达式也会调用两个形参的 delete[]版本，但第二个形参必须是 size_t 类型的，即 void operator delete[](void*,size_t){...}形式。在使用 delete 表达式调用时 size_t 会被自动初始化为数组所需内存的字节大小。

（9）new[]和 delete[]的其他调用方式及规则与 new 和 delete 相同。

12.4.2　定位（布局）new 和 delete

1．定位 new 表达式与定位 new 运算符

（1）要使用定位 new 表达式和运算符，需要包含头文件<new>。

（2）定位 new/delete 运算符：其实就是重载的内置 new/delete 运算符函数的另一种形式。只不过这两个版本的程序实现代码位于头文件<new>中。

（3）默认的内置定位 new 运算符函数的形式为 void *operator new(size_t, void * p){return p;}，可以看到内置的定位 new 运算符函数很简单，只返回一个指针。用户也可以把定位 new 运算符函数重载为类的成员函数。

（4）若把定位 new 运算符函数重载为类的成员函数，那么定位 new 运算符函数的第二个形参可以不是 void*，可以是其他类型的指针，但第一个形参类型必须是 size_t，返回类型必须是 void*。比如 class A{public: void *operator new(size_t, A* p){return p;}};。

（5）定位 new 表达式：用于调用内置的定位 new 运算符函数。

（6）定位 new 表达式的形式：new(指针或地址) 类型名 (初始化列表);。

例如 new (p) A(2);：

① 该表达式的作用是把“指定类型的对象”创建在已分配好的内存中。也就是说，定位 new 表达式把对象创建（定位）在指定的已分配好的内存中。

② 指定类型的对象由“类型名(初始化列表)”，通过初始化列表提供的参数调用相应类型的构造函数进行创建，若没有初始化列表，则调用默认构造函数创建对象。

③ 已分配好的内存通过括号中的“指针或地址”指定，“指针或地址”也被称为定位实参。

④ 比如 int a[4];，则 A *pa=new (a) A; 表示把使用默认构造函数创建的类类型 A 的对象放置在已分配好的内存 a 中，当然 a 应有足够大的空间存储该对象。

⑤ 比如 int *p=new int;，则 A *pa=new (p) A(2,3); 表示把使用带两个形参的构造函数创建的类类型 A 的对象放置在已分配好的内存 p 中。

2. 使用定位 new 表达式调用定位 new 运算符函数分配内存

（1）首先调用定位 new 运算符函数，然后调用类的相应的构造函数创建对象，最后返回该对象的地址。比如 new (pa) A(2,4);，首先调用定位 new 运算符函数，然后调用类 A 的带两个参数的构造函数创建对象，最后返回该对象的地址。

（2）定位 new 表达式的参数传递规则。

① 定位 new 运算符函数的第二个形参被初始化为 new 之后括号中的“指针或地址”。

② 定位 new 运算符函数的第一个形参被初始化为“类型名”所指类型的字节大小。

③ 所创建的对象使用类型名之后与初始化列表对应的构造函数进行构造。

④ 这里要注意的是，使用定位 new 表达式会自动调用相应的构造函数创建对象。

⑤ 比如 int a[4]; A *pa=new (a) A(9);，则 a 被传递给定位 new 运算符函数的第二个形参，类型 A 的字节大小被传递给定位 new 运算符函数的第一个形参，类类型 A 的对象使用带一个形参的构造函数进行创建。

（3）定位 new 表达式还可用来调用带两个及以上形参的重载 new 运算符函数，其规则就是 new 运算符函数的第一个形参被指定类型的字节大小初始化，第二个形参被定位 new 表达式中

new 之后括号中的第一个参数初始化，第三个形参被括号中的第二个参数初始化，依此类推。因为没有与定位 new 表达式对应的 delete 表达式，因此无法使用定位 delete 表达式调用有多个形参的 delete 运算符函数。比如 void *operator new(size_t, int i, int j){return 0;}，则定位 new 表达式的形式应为 int *p=new (2,3)int;，其中 size_t 被 int 类型的字节大小初始化，i 被 2 初始化，j 被 3 初始化。

3．使用定位 new 表达式分配内存的原理及方法

（1）原理：其实定位 new 表达式自身并没有分配内存，它只是把对象放置在已分配好的内存中。比如 int a[4];，则对于 A *pa=new (a) A;，若使用 delete pa;将会发生错误，因为内存是分配在 int a[4]上的，new(a) A;并未分配内存，pa 指向的是 int a[4]分配的地址，但 int a[4]的内存不是使用 new 动态分配的，所以不能使用 delete 进行释放。

（2）在使用定位 new 表达式分配内存时，已分配的内存应足够大，以便能容纳下待分配的类类型对象。比如 char c[1]; A *pa=new (c) A();，很明显已分配的内存 c 不一定能容纳下待分配的类类型 A 的对象。

（3）内存区域重叠问题：定位 new 运算符函数不会对已分配的内存情况进行检测，因此在使用定位 new 表达式分配内存时应注意内存区域重叠的问题。比如 char c[44]; A *p1=new (c) A();A *p2=new (c) A();，这时 p1 和 p2 都被分配在相同的内存区域 c 中，它们的起始地址都与 c 相同。分配 p1 之后，正确分配 p2 的方式为 A *p2=new (c+sizeof(A)) A();。

（4）定位 new 表达式不支持多态。比如 class A{public:virtual void f(){} };class B:public A{void f(){} };，则对于 char *mc=new char[sizeof(A)]; A *pa=new (mc) B;，若 mc 不能容纳 B 对象，则会对内存产生严重的破坏；若 mc 能容纳子类 B 对象，则 pa->f();的行为是未定义的，也就是说，可能调用类 A 中的函数 f，也可能调用类 B 中的函数 f。

示例 12.21：定位 new 运算符函数和定位 new 表达式

```
#include <iostream>
#include <new>          //使用定位 new 表达式和运算符函数需要包含此头文件
using namespace std;
class A{public: int a; A(){cout<<"A"<<endl;} A(int i, int j){cout<<"AA"<<endl;}
   void *operator new(size_t t, char* p)//定位 new 运算符函数的第二个形参可以不是 void*类型
      {cout<<"N"<<endl; cout<<"size_t="<<t<<endl; cout<<p[1]<<endl; return p;}
   void *operator new(size_t t,int i,int j) {int *p=new int;cout<<"NN";return p;}};
                                                                    //类 A 结束
void main(){
   char c[44]={'a','b','c','d','e','f','g'};
   cout<<"类型 A 的字节大小="<<sizeof(A)<<endl; //输出“类型 A 的字节大小=4”
   A *pa=new(c) A; //定位 new 表达式，表示把使用默认构造函数创建的类类型 A 的对象放置在
                   //已分配好的内存 c 中。定位 new 表达式按以下步骤执行
                  //1. 调用定位 new 运算符函数，输出以下内容
```

```
                    //A. 首先输出 N
                    //B. 然后输出 size_t=4。可见，定位 new 运算符函数的第一个
                    //形参类型 size_t 的值就是类类型 A 的字节大小
                    //C. 最后输出 b。可见，定位 new 运算符函数的第二个形参
                    //被 new 之后小括号中的指针 c 所初始化
                    //2. 调用类 A 的相应的构造函数创建对象（此处为默认构造函数），因此输出 A
                    //3. 返回所创建的对象的地址
A *pa1=new(2,3) A; //使用定位 new 表达式调用重载的带有两个及以上形参的 void* operator new(size_t
                     //t,int i,int j)运算符函数，其中 size_t 被类型 A 的字节大小初始化
                     //i 被 2 初始化，j 被 3 初始化
//delete pa;     //错误，pa 是分配在 char c[44]上的，而 new(c) A 并未分配内存，pa 指向的是地址 c
                 //但 c 的内存不是使用 new 动态分配的，所以不能使用 delete 释放
//A* pa2=new(c) A; //潜在的危险的错误，此处出现内存区域重叠问题，pa2 分配的内存与 pa 分配的内
                   //存位于同一位置
A *pa3=new(c+sizeof(A)) A; //解决内存区域重叠问题的方法
A *pa4=new(c) A(4,5);}//表示把使用 A(int,int)创建的类类型 A 的对象放置在已分配好的内存 c 中
```

4. 显式调用析构函数及定位 delete 运算符函数

（1）不存在定位 delete 表达式（但有定位 delete 运算符函数），即不存在 delete (p1) p 之类的表达式。

（2）delete 不能释放由定位 new 表达式分配的内存：因为定位 new 表达式并不分配内存，所以 delete 无法释放由定位 new 表达式分配的内存，至于如何处理由定位 new 表达式创建的对象所占用的内存就要看该内存的来源了（即位于 new 之后小括号中的指针或地址）。若内存是使用常规 new 动态分配的，则可使用对应的 delete 释放内存，否则就是错误的。比如 int a[4]; A* pa=new(a) A;，则 delete pa 是错误的，这里的内存不是使用 new 动态分配的；再比如 int *p=new int; A* pa=new (p) A;，则 delete pa 是正确的，但这里并不是在释放使用定位 new 表达式分配的内存 pa 的地址，而是在释放由 pa 指向的使用 new 动态分配的内存地址 p。

（3）显式调用析构函数：在使用定位 new 表达式时，会自动调用类的相应的构造函数创建对象，在使用 delete 释放内存时，需要看定位 new 表达式的内存来源，否则可能会出现错误，无法销毁由定位 new 表达式创建的对象，这时就需要显式调用析构函数来销毁对象。在正常情况下，析构函数并不需要显式调用。

示例如下：

```
int a[4]; A* pa=new(a) A;
```

使用 delete 无法销毁由定位 new 表达式创建的对象，因为内存不是由 new 动态分配的。由定位 new 表达式创建的对象需要通过显式调用析构函数进行销毁，其形式为 pa->~A();。

```
int *p=new int[44]; A* pa=new(p) A;
```

① 无法使用 delete 销毁由定位 new 表达式创建的对象。

② 因为内存来源 p 是由 new 动态创建的数组，因此 p 应由 delete []p 进行释放。

③ 假设使用 delete []pa;释放内存，则全局的 delete 会调用 pa 指向的类类型 A 的析构函数，

但由于释放的是一个 new 数组的内存，因此这里就存在 delete 多次调用析构函数销毁对象的问题，而 new(p) A;只创建了一个对象，所以出错。

④ 正确处理方式是显式调用析构函数，并且不能把 delete 作用于 pa 上，而应作用于 p 上。其正确形式是 pa->~A(); delete []p;。

```
int *p=new int; A* pa=new(p) A;
```

可以使用 delete pa 销毁由定位 new 表达式创建的对象，因为在执行 delete pa 时，全局的 delete 会调用析构函数，同时执行 delete pa 实际释放的是内存来源 p 指向的内存，而 p 是由 new 动态分配的，所以正确。注意：执行 delete pa 并没有释放 pa 的内存，而是 p 的内存。

（4）定位 delete 运算符函数及调用时机：该函数应与定位 new 表达式配套使用，在使用定位 new 表达式调用构造函数初始化对象失败而抛出异常时，定位 delete 运算符函数会被程序自动调用。若未抛出异常，则不会调用定位 delete 运算符函数。若程序中没有重载定位 delete 运算符函数，则发生异常时不会调用任何 delete 运算符函数。

（5）定位 delete 运算符函数的形式：第一个形参的类型必须是 void*，第二个形参的类型必须与定位 new 运算符函数的第二个形参相同，否则就不是与定位 new 运算符函数配对的定位 delete 运算符函数。比如 void* operator new(size_t, A* p){return p;}，则与之配对的定位 delete 运算符函数是 void operator delete(void*, A* p){...};。

（6）是否应提供定位 delete 运算符函数：若定位 new 运算符函数内部本身分配了内存，则应提供定位 delete 运算符函数，以便当定位 new 表达式初始化对象抛出异常时能正确释放内存；否则无须提供配对的定位 delete 运算符函数。

（7）当然，也有相应的定位 new[]表达式和定位 new[]运算符函数，它们的原理与定位 new 表达式和运算函数基本相同，差别只在于定位 new[]表达式使用的是默认构造函数初始化对象，比如 A* pa=new (p) A[3];，将调用类 A 的默认构造函数初始化对象。

示例 12.22：显式调用析构函数及定位 delete 运算符函数

```
#include <iostream>
#include <new>          //使用定位 new 表达式和运算符函数需要包含此头文件
using namespace std;
class A{public:    int a; A(){cout<<"A"<<endl;}     ~A(){cout<<"~A"<<endl; }
   void* operator new(size_t t, void *p){return p;}    //重载的定位 new 运算符函数
   void operator delete(void*, void* p){}   //与定位 new 运算符函数配对的定位 delete 运算符函数
                //该函数只在 new 表达式调用构造函数初始化对象失败而抛出异常时被自动调用
                //定位 delete 运算符函数的第二个形参类型必须与定位 new 运算符函数的第二个形参相同
                //否则就不是与定位 new 运算符函数配对的定位 delete 运算符函数
};     //类 A 结束
void main(){
   /int c[444]; int *pc=new int;int *pc1=new int[44]; //定位 new 表达式的不同内存来源
   A *pa1=new (c)A; A *pa2=new (pc) A; A *pa3=new (pc1) A;  //各种定位 new 表达式
   //delete pa1;     //错误，内存来源不是由 new 动态分配的内存
```

```
    ::delete pa2;   //正确，全局的 delete 会调用析构函数，delete 实际释放的内存来源是 pc，而 pc 是使
                    //用 new 动态分配的。注意，类 A 中重载的 delete 运算符隐藏了全局的 delete 运算符
    //::delete pa3;//错误，内存泄漏
    //delete[] pa3;//错误，因为 delete 会多次调用析构函数来销毁对象，而 new(pc1)A;只
                   //创建了一个对象，所以出错
    pa3->~A();     //正确形式，显式调用析构函数销毁对象，并且不能把 delete 作用于 pa3 上
                   //而应作用于 pc1 上，以正确销毁对象并释放内存
    delete[]pc1;}
```

12.4.3 new 表达式和 new 运算符函数总结

1. 全局 new 的 6 种默认形式

```
void *operator new(size_t) throw(bad_alloc);              //C++常用版本
void *operator new(size_t, const nothrow_t&) throw(); //兼容早期的版本，不抛出异常
void *operator new(size_t,void *) throw ();               //定位 new 运算符
void *operator new[](size_t) throw(bad_alloc);
void *operator new[](size_t, const nothrow_t&) throw();
void *operator new[](size_t,void *) throw ();
```

2. 全部的 new 表达式

说明：new 表达式的含义是无法改变的，用户只能改变 new 运算符函数的内容。

（1）new T(参数);

常用的 new 表达式，调用重载或全局的 new 运算符函数分配内存，并用后面的参数调用构造函数创建对象；若无小括号和参数，则用默认构造函数创建对象。使用 delete 进行销毁。

（2）new T[];

动态创建对象数组，使用默认构造函数初始化，调用重载或全局的 new[]运算符函数分配内存。使用 delete[]销毁。

（3）new(参数 1) T(参数 2); //定位 new 表达式

① 定位 new 表达式，调用重载或全局的定位 new 运算符函数，并使用参数 2 调用 T 的构造函数创建对象；若无小括号和参数 2，则使用 T 的默认构造函数创建对象。参数 1 的类型应与定位 new 运算符函数的第二个形参的类型兼容。

② 此表达式还可调用重载的带多个形参的 new 运算符函数，这时参数 1 若是一系列形参，其相应的类型和个数应依次与 new 运算符函数的第二、三、四、…个形参相对应，new 运算符函数的第一个形参被类型 T 所占字节大小自动初始化。比如 new (2,3,4) T(5,6);，表示调用 void *operator new(size_t, int i, int j, int k){…};形式的 new 运算符函数，其第一个形参类型 size_t 被类型 T 的字节大小初始化，第二个形参 i 对应于 new 后面小括号中的整数 2，j 对应于 3，k 对应于 4。T 创建的对象调用类型 T 的带两个形参的构造函数初始化。

③ 定位 new 表达式的对象不能直接使用 delete 销毁，由定位 new 表达式创建的对象所占的内存如何处理，需要看内存的具体来源。

（4）new(参数 1) T[];

定位 new[]表达式，其原理同定位 new 表达式，只是创建一个对象数组，且创建对象时使用默认构造函数初始化对象。

第 13 章

继承、虚函数与多态性专题

（1）多级继承：指的是类 A 继承自类 B、类 B 继承自类 C、类 C 继承自类 D 的情形。

（2）多重继承：指的是类 A 同时继承自类 B、类 C、类 D 的情形。

13.1 继承

13.1.1 继承基础及继承后的访问级别

1. 继承基础及语法

（1）继承：是在现有类的基础上进行的扩展。也就是说，继承是在以前的类的基础上，增加了属于自己成员的类。继承之后的新类将拥有之前类的成员，同时拥有新增加的属于自己的成员。由此可知，继承可以减少不必要的重复代码，新类拥有比被继承的旧类更具体的内容。假设有类 class A{public: int a; void f();};和 class B:A{public: int b; void g();};，若类 B 继承自类 A，则类 B 不但拥有自己的成员 b 和 g，还拥有从类 A 继承的成员 a 和 f。若类 B 不是继承自类 A，而又要同时拥有成员 a 和 f，则应这样声明：class B{public:int a; int b; void g(); void f(); };。

（2）父类（超类、基类）和子类（派生类）：被继承的类称为父类、基类或超类，继承之后的新类称为子类或派生类。

（3）继承的语法。

```
class 子类名:访问控制符 父类名{...}; //示例：class B:public A{}
```

① 比如 class B: public A{};，表示子类 B 以公有方式继承自父类 A。

② C++使用冒号“:”表示继承。

③ 访问控制符可省略，这时若子类是使用 class 声明的，则默认为 private（私有的）；若子类是使用 struct 声明的，则默认是 public（公有的）。比如 struct A:B{...};，则子类 A 以默认公有的方式继承自父类 B。

④ 父类必须是已经定义过的类，只声明而未定义的类不能作为父类，因为子类包含父类的

成员且可访问。比如 class B;，则 class A: public B{}; 错误。

⑤ 只声明而不定义子类时，子类不能包含有父类部分，只需按照声明普通类的形式声明子类即可。比如 class A:public B;或 class A:B;都是错误的，正确形式为 class A;。

⑥ 在C++中派生类可以同时从多个父类继承，当继承多个父类时，使用逗号将父类隔开。

2. 从父类继承的成员

（1）继承之后，子类的成员包括两部分：一部分是从父类继承的成员（非静态的）；一部分是自己新增加的成员（非静态的）。比如 class A{public: int a; void f(){}}; class B:public A{public:int b; void g(){}}; ，则子类 B 的成员包括 a, f, b, g，其中 a 和 f 是从父类继承的。

（2）继承之后，子类从父类继承的成员成为自己的成员，访问它们就像访问自己的成员一样，因此成员函数可直接使用，不需要添加任何操作符。但是需要注意所继承的成员的访问级别。比如 class A{public: int a;}; class B:public A{void f(){a=4;} };，从父类继承的成员 a，在子类 B 中就像自己的成员一样。

（3）子类从父类继承的成员，与父类原有的成员是相互独立不相关的。比如 class A{public: int a;}; class B:public A{}; A ma; B mb; ma.a=3; mb.a=4;，则 cout<<ma.a;将输出 3，可以看到子类从父类继承的成员 a 与父类中的成员 a 是相互独立不相关的。

（4）注意：父类不能访问子类的成员。原因很简单，因为父类不知道子类有哪些成员。比如 class A{}; class B:public A{public:int b;}; A ma;，则 ma.b=2;错误，不能访问子类的成员。

3. 从父类继承的成员的访问级别

（1）访问类的成员分两种情况：一种是在成员函数内；一种是在该类之外。比如私有成员可以在成员函数内被访问，但在类外不能访问。比如 class {private: int a; void f(){a=3;}; A ma; ma.a=4;，在成员函数内对私有成员 a 的访问 a=3 正确，而在类外对 a 的访问 ma.a=4 错误。

（2）子类从父类继承的成员的访问级别（见表 13.1）：由父类的成员访问级别和子类对父类的继承方式共同决定，具体规则如下。

表 13.1　子类从父类继承的成员的访问级别

	父类的公有成员	父类的受保护成员	父类的私有成员
子类公有继承	公有的	受保护的	不可访问（未被继承）
子类私有继承	私有的	私有的	不可访问（未被继承）
子类受保护继承	受保护的	受保护的	不可访问（未被继承）

① 若是公有（public）继承，则父类的公有成员成为子类的公有成员；父类的受保护成员成为子类的受保护成员；父类的私有成员仍为父类的私有成员。

② 若是私有（private）继承，则父类的公有成员和受保护成员都成为子类的私有成员；父

类的私有成员仍为父类的私有成员。注意：默认继承为私有继承。

③ 若是受保护（protected）继承，则父类的公有成员和受保护成员都成为子类的受保护成员；父类的私有成员仍为父类的私有成员。

④ 私有成员不会被子类继承：不管父类以何种方式被继承，父类的私有成员仍然属于父类且是私有的，其不属于子类，即子类不会继承父类的私有成员，因此子类不能访问父类的私有成员。比如 class A{int a; public:int b;}; class B:private A{ public: void f(){a=1;b=2; }}; ，在成员函数 f 中对父类的私有成员 a 的访问是错误的，因为 a 并不是子类 B 的成员；b=2;是正确的，因为 b 被继承为子类 B 的私有成员。

⑤ 受保护继承与私有继承的区别：主要在于多级继承。私有继承之后，所有父类成员都成为子类的私有成员（父类的私有成员除外），而且私有成员不能再被继承，因此私有继承只能使成员进行一级继承；而受保护继承之后，所有父类成员都成为子类的受保护成员（父类的私有成员除外），因此受保护继承可以使成员进行多级继承。比如 class A{public: int a;}; class B:private A{void f(){a=1;}}; class C:public B{void g(){a=2;} };，则类 C 中的 a=2;是错误的，因为 a 未被继承到类 C 中，若类 B 是以 protected 继承的，则类 C 中的 a=2;是正确的。

⑥ 可见，protected 就表示该成员不能被外界访问，但可被子类成员访问（一般指成员函数）。因此 protected 只在继承时才有实质性用处，在其他情况下并无多大用处，所以在无继承时，对于本类的成员而言，只需要 private 和 public 两种访问权限即可。

（3）对于多级继承，只需按照上面的规则分析即可。比如 class A{public: int a;}; class B:protected A{};class C:private B{}; class D:public C{};，类 B 继承类 A 之后，a 成为类 B 的受保护成员；类 C 继承类 B 之后，a 成为类 C 的私有成员；类 D 继承类 C 之后，在类 D 中 a 未被继承，不可访问，因为 a 是类 C 的私有成员。

（4）恢复从父类继承的成员的访问级别：只能把所继承的成员恢复为父类中原来的访问级别，而不能比父类中原来的访问级别高或低。恢复访问级别有如下两种方法。

① 使用 using 声明恢复从父类继承的成员的访问级别。其具体方法是在待恢复的访问级别后面，对需要恢复的成员使用 using 声明语句进行声明。比如 class A{protected: int a; void f(){}};class B:private A{protected: using A::a; using A::f;};，此处把子类 B 从父类 A 私有继承的成为子类的私有成员 a 和 f 重新声明为原来父类的 protected 级别。

② 使用旧式的类名和作用域解析运算符来恢复。此方法不推荐使用，因为有可能会被 C++ 淘汰。其具体方法与使 using 声明相同，比如 class A{protected: int a; void f(){}};class B:private A{protected: A::a; A::f;};。

示例 13.1：从父类继承的成员的访问级别、多级继承以及私有继承与受保护继承的区别

```
class A{private:int a1; protected: int a2; public: int a3; A(){a1=0;a2=0;a3=0;}};
//公有继承：父类中的 a1 不会被继承，a2 成为受保护的，a3 成为公有的
class B:public A{public:
   void f(){A ma; B mb;   //注意：应在成员函数内创建正在被定义的类对象，否则是错误的
   //mb.a1=1; a1=1;      //错误，不可访问，因为父类 A 的私有成员 a1 不会被继承，因此不是子类 B 的成员
   mb.a2=2; a2=22;       //正确，a2 被继承为子类 B 的受保护成员，可被该类的成员函数访问。注意：在成
                         //员函数内可对非公有成员进行访问
   mb.a3=3; a3=33;       //正确，a3 被继承为子类 B 的公有成员
   //ma.a2=2;            //错误，因为 ma 是父类 A 的对象，因此 ma.a2 访问的是父类 A 的成员 a2，
                         //而不是子类从父类继承的成员 a2，而且父类的 a2 是受保护的，不能在该类外访问
   //ma.a1=1;            //错误，a1 是父类 A 的私有成员，私有成员不能在该类外进行访问
   ma.a3=32; }};         //正确，此处访问的是类 A 的 a3，它是公有的
                         //私有继承：父类的 a1 不会被继承，a2 成为私有的，a3 成为私有的
class C:private A{public:
   void f(){  C mc;
   //mc.a1=1; a1=1;      //错误，不可访问，类 A 的私有成员 a1 未被继承
   mc.a2=2; a2=22;       //正确，a2 被继承为子类 C 的私有成员，可被该类的成员函数访问
   mc.a3=3; a3=33; }}; //正确，父类的公有成员 a3 也被继承为子类 C 的私有成员
                         //受保护继承：父类的 a1 不会被继承，a2 成为受保护的，a3 成为受保护的
class D:protected A{public:
   void f(){D md;
   //md.a1=1; a1=1;      //错误，不可访问，类 A 的私有成员 a1 未被继承
   md.a2=2; a2=22;       //正确，a2 被继承为子类 D 的受保护成员，可被该类的成员函数访问
   md.a3=3; a3=33; } };//正确，父类的公有成员 a3 也被继承为子类 D 的受保护成员
//多级继承：验证私有继承与受保护继承的区别
class E:public C{public: //直接基类 C 私有继承自类 A
   void f(){//a1=1;//错误，不可访问。间接父类 A 的私有成员 a1 未被继承
        //a2=2;    //错误，不可访问。间接父类 A 的受保护成员 a2 被直接父类 C 以私有方式继承为类 C 的
                   //私有成员，因为 a2 在类 C 中已是私有成员，因此不会被类 E 继承
        //a3=3;    //错误，不可访问。间接父类 A 的公有成员 a3 被直接父类 C 以私有方式继承为类 C 的私有
                   //成员，因为 a3 在类 C 中已是私有成员，因此不会被类 E 继承
};
class F:public D{public://直接基类 D 受保护继承自类 A，受保护继承可使父类的成员进行多级继承
                        //而私有继承则不可以
   void f(){//a1=1;     //错误，不可访问。间接父类 A 的私有成员 a1 未被继承
        a2=2;           //正确，可访问。间接父类 A 的受保护成员 a2 被直接父类 D 以受保护方式继承为类 D
                        //的受保护成员，因此可被类 F 继承，继承之后 a2 成为类 F 的受保护成员
        a3=3; } };  //正确，可访问。间接父类 A 的公有成员 a3 被直接父类 D 以受保护方式继承为类 D 的受保
                    //护成员，因此可被类 F 继承，继承之后 a3 成为类 F 的受保护成员 main 函数开始
int main(){A ma1; B mb1; C mc1; D md1; F mf;
   //mb1.a1=1; //错误，a1 不是类 B 的成员，它是父类 A 的私有成员，未被子类 B 继承
   //mc1.a1=1; //错误，同上
   //md1.a1=1; //错误，同上
   //mb1.a2=2;  //错误，父类的受保护成员 a2 被类 B 公有继承后仍是受保护成员
   //mc1.a2=2;  //错误。父类的受保护成员 a2 被类 C 私有继承后成为私有成员
   //md1.a2=2;  //错误，父类的受保护成员 a2 被类 D 受保护继承后成为受保护成员
```

```
    mb1.a3=3;    //正确，输出 0，父类的公有成员 a3 被类 B 公有继承后仍是公有成员
    //mc1.a3=3;  //错误，父类的公有成员 a3 被类 C 私有继承后成为私有成员
     //md1.a3=3;  //错误，父类的公有成员 a3 被类 D 受保护继承后成为受保护成员
    //mf.a2=2; mf.a3=3; //错误，间接父类 A 的受保护成员 a2 和公有成员 a3 最后被类 F 继承为受保护成员
}
```

4. 继承与友元、静态数据成员

（1）不管静态成员被继承多少次，静态成员也只会有一个（因为静态成员是属于整个类的），继承之后可以使用子类的类名直接对静态成员进行调用，也可以用子类的类对象进行调用，但要注意静态成员仍然遵守继承时的访问级别规则，还要注意静态成员不能使用派生类进行初始化（对于此种情形，VC++ 2010 只会发出一个警告）。比如 class A{public:static int a;}; class B:private A{};int A::a=1;，则 cout<<B::a;错误，因为 a 被继承之后在类 B 中是私有的；cout<<A::a;正确。再比如 class A{public:static int a;}; class B:private A{};，则 int B::a=1;初始化是错误的，但 VC++ 2010 只会发出一个警告。

（2）友元在继承关系中不会被传递，父类中的友元只对父类有特殊访问权限。注意：子类中的友元对从父类继承的成员具有特殊访问权限，因为其已经是子类的成员了。

（3）父类和子类是互不相关的，若父类是某个类的友元，则只有父类拥有对该类的特殊访问权限；同理，若子类是某个类的友元，则只有该子类对此类拥有特殊访问权限。

示例 13.2：友元在继承关系中不会被传递

```
class A{private: int a; public: int a1; friend void f();};
class B:private A{private: int b; friend void g();};
void f(){A ma;  B mb;
   //mb.b=4;     //错误，函数 f 不是类 B 的友元
   ma.a=4;}    //正确，函数 f 是类 A 的友元
void g(){ A ma;   B mb;
   //ma.a=4;    //错误，g 不是类 A 的友元，不能访问类 A 的私有成员
   //mb.a=4;    //错误，子类 B 未继承父类 A 的私有成员 a，因此 a 不是子类 B 的成员
mb.a1=4;} //正确，a1 被继承为子类 B 的私有成员，而 g 是类 B 的友元，所以正确
int main(){}
```

示例 13.3：友元父类和友元子类是互不相关的

```
class A; class B;                  //前向声明
class E{int e; friend class A;};
class F{int f; friend class B;};
class A{public:
   void g(){  E me; F mf;
        //mf.f=4;    //错误，类 A 不是 F 的友元
        me.e=3;}}; //正确，类 A 是 E 的友元
class B:public A{public:
   void g(){  E me; F mf;
         //me.e=3;     //错误，类 B 不是 E 的友元
```

```
        mf.f=4;  }};  //正确，类B是F的友元
int main(){}
```

13.1.2　继承下的构造函数与复制控制

父类的构造函数、复制构造函数、析构函数、赋值操作符函数不会被子类继承。比如 class A{public: A(){}A(int i,int j){}}; class B: public A{};，则 B mb(4,9);错误，此处试图使用父类的带两个形参的构造函数创建对象，但父类的构造函数并不会被子类继承。

1．继承下的构造函数和析构函数

（1）继承时初始化成员的基本规则：继承时，子类的构造函数不能直接初始化父类的成员，只能初始化子类的成员，而父类的成员需要使用父类的构造函数初始化。比如 class A{public: int a;}; class B:public A{public: B():a(3){}};错误，子类不能对父类的成员 a 直接进行初始化。注意：在构造函数内的语句是赋值不是初始化。

（2）调用父类的构造函数的基本规则：父类的构造函数是由子类的构造函数调用的，也就是说，当创建子类对象时，先用子类的构造函数调用父类的构造函数初始化父类的成员，然后再执行子类的构造函数初始化子类的成员。即先构造父类再构造子类的顺序。执行析构函数的顺序与此相反。

（3）子类的构造函数调用父类的构造函数的方法：一种是显式调用；一种是隐式调用。

① 显式调用父类的构造函数的方法：在子类的构造函数中，可以使用成员初始化表的形式显式调用父类的构造函数初始化父类的成员。比如 class A{public: A(int i){}}; class B:public A{ public: int b; B(int j):A(j),b(j){} };，其中 B(int j):A(j),b(j){}表示调用父类的构造函数 A(int)初始化父类部分，类 B 的成员 b 使用 j 初始化。

② 隐式调用父类的构造函数的方法：若子类没有显式使用成员初始化表调用父类的构造函数，则子类的构造函数会调用父类的默认构造函数初始化父类的成员，然后再执行子类的构造函数，若父类没有默认构造函数则出错。由系统自动生成的子类的默认构造函数也是采用隐式方式调用父类的默认构造函数的。比如 class A{public: A(int i){}}; class B:public A{B(){}}; 错误，父类 A 没有默认构造函数。

（4）若父类没有默认构造函数，则子类中所有的构造函数都必须显式使用成员初始化表调用父类中定义的构造函数，否则会因找不到默认构造函数而发生错误，注意：这时子类不能使用系统生成的默认构造函数，因为该构造函数同样会调用父类的默认构造函数。比如 class A{public:A(int i){}}; class B:public A{public: B():A(3){} B(int i){} };，其中 B(int i)是错误的，因为类 A 没有默认构造函数可被调用。

（5）在子类外定义子类的构造函数时，在子类的构造函数中的声明不应包括父类的构造函

数。因为父类的构造函数是在子类的成员初始化表中调用的，而声明构造函数时是不能使用成员初始化表的。比如 class A{};class B:public A{public: B();B(int i):A();};B::B():A(){} B::B(int i):A(){}，其中 B(int i):A();的声明是错误的。

（6）子类只能初始化它的直接父类。比如 class A{}; class B:public A{};class C:public B{public: C():A(){} };，其中 C():A();是错误的，因为类 A 不是类 C 的直接父类。

（7）初始化顺序：在继承时，不管父类的构造函数出现在子类的成员初始化表的什么位置，或者没有出现在子类的成员初始化表中，首先被初始化的都是父类部分，若是多重继承，则按照继承时的顺序进行初始化，其余成员按照成员初始化表的规则进行，即按照声明的顺序初始化。比如 class A{};class B:public A{public: int c;int b; B():b(1),A(),c(2){}}; 或者 class B:public A{ public: int c; int b; B():b(1),c(2){}};，则初始化顺序是首先调用父类 A 的默认构造函数初始化父类部分，然后初始化 c，最后是 b。

（8）当子类中含有类对象成员，特别是含有父类对象成员时，也是首先调用父类的构造函数初始化父类部分。对于父类对象成员，在成员初始化表中也是按照声明的顺序进行初始化的（因为父类对象成员始终是类的成员）。比如 class A{public:A(int i){}}; class B:public A{public: int b; A ma; B():ma(2),b(3),A(4){}};，其初始化顺序是首先调用父类的构造函数初始化父类部分，即 A(4)，然后初始化内置成员 b，最后初始化父类对象成员 ma。

（9）在撤销对象时，总是显式调用父类的析构函数撤销父类部分，使用子类的析构函数撤销子类部分，其析构函数的调用顺序与构造函数相反。

2．继承下的复制构造函数和赋值操作符

（1）基本规则：继承中的父类的复制构造函数复制父类部分，子类的复制构造函数复制子类部分；父类的赋值运算符对父类部分进行赋值，子类的赋值运算符对子类部分进行赋值。

（2）若子类未定义复制构造函数，则子类的默认复制构造函数会调用父类的默认复制构造函数复制父类部分；若父类定义了复制构造函数，则调用该复制构造函数复制父类部分，子类部分由子类的默认复制构造函数进行复制。比如 class A{public: A(){} A(const A& ma){}}; class B:public A{};B mb;，则 B mb(mb1);将使用子类 B 的默认复制构造函数复制子类部分，使用父类 A 中自定义的复制构造函数复制父类部分。

（3）若子类定义了自己的复制构造函数，则该复制构造函数应使用成员初始化表显式调用父类的复制构造函数（正确方式），否则子类的复制构造函数会调用父类的默认构造函数（注意：不是默认复制构造函数，其实这是一种不正确的行为）。记住：复制构造函数就是一个构造函数，只是它的形参为类本身的引用而已。比如 class A{}; class B:public A{public: B(const B& mb){} };，此时子类 B 的复制构造函数将调用父类 A 的默认构造函数初始化该对象的父类部分，因此正确的形式应是 class B:public A{public: B(const B& mb):A(mb){}};。

（4）若子类未定义赋值操作符，则子类的默认赋值操作符会调用父类的默认赋值操作符对父类部分进行赋值；若父类定义了赋值操作符，则调用该赋值操作符对父类部分进行赋值，子类部分由子类的默认赋值操作符进行赋值。比如 class A{public: A& operator =(const A& s){...}}; class B:public A{}; B mb; B mb1;，则 mb=mb1 将使用子类 B 的默认赋值操作符对子类部分进行赋值，使用父类 A 中自定义的赋值操作符对父类部分进行赋值。

（5）若子类定义了自己的赋值操作符，则该赋值操作符应在函数体内显式调用父类的赋值操作符函数对父类部分进行赋值（正确方式），否则子类的赋值操作符不会对父类部分进行赋值（这是不正确的行为）。这一点是与复制构造函数不同的地方。比如 class A{}; class B:public A{public: B& operator =(const B& s){ if(this!=&s){A::operator=(s); ...}}}; B mb; B mb1;，则 mb=mb1 将使用子类自定义的赋值操作符对子类部分进行赋值，调用父类的默认赋值操作符对父类部分进行赋值，若没有 A::operator=(s); 语句，则父类部分不会执行任何操作。

13.1.3　父类与子类间的转换

本节应注意区分类对象的引用（或指针）与类对象的区别。

（1）在常规情形下，C++是不可以把一种类型的指针指向另一种类型的，引用原理类似。但是在继承的情形下，父类的指针或引用可以指向派生类对象，而且不需要显式进行强制类型转换。

（2）子类拥有父类的副本：继承时，C++保证子类拥有父类子对象的完整原样性，即每个子类对象都包含一个父类子对象的完整副本，因此存在从子类类型到父类类型的转换。

（3）自动转换：在公有继承的情况下（此前提见后文），子类对象的引用（指针）可以自动转换为父类对象的引用（指针），这时父类对象的引用（指针）拥有子类对象的父类部分。注意：这种转换是自动执行的，是子类的引用或指针转换为父类的情形，类类型的对象不适合此情形。

（4）子类与父类自动转换详解。

① 若子类对象自动转换为父类的引用，则父类的引用只引用子类的父类子对象。也就是说，被转换后的父类的引用相当于是子类的父类子对象的别名。若是指针，则父类的指针指向的范围只局限于子类的父类子对象部分。假设 B 是 A 的子类，则有 B mb; A &ma=mb;，此时 ma 只是子类对象 mb 的父类子对象的别名，ma 既不是父类的别名，也不是子类 mb 的别名。

② 当子类类型自动转换为父类类型后，父类对象的引用（指针）只能访问从子类继承的父类子对象的成员，子类新增加的成员不能被访问（即使使用作用域解析运算符也不行）。详见示例和图 13.1。

示例如下：

```
class A{public:int a;void f(){}}; class B:public A{public:int a; int b; void f(){} };
B mb; A *pa=&mb; A &pa1=mb; A ma;
```

- pa->b; 错误，pa 只拥有子类 A 的父类部分，而成员 b 是子类 B 新增加的成员，不是 pa 的成员。
- pa->a;或 pa->A::a;正确，只调用父类 A 的数据成员 a。
- pa->f();或 pa->A::f();正确，只调用父类 A 的函数 f。
- pa->B::a;或 pa->B::f();错误，即使使用作用域解析运算符也不能使父类指针调用子类的成员，
- pa1.B::a 或 pa1.B::f();错误，原因同上。
- mb.B::f();或 mb.A::f();正确。
- ma.b; 错误，b 不是 ma 的成员。父类对象不知道子类的内容

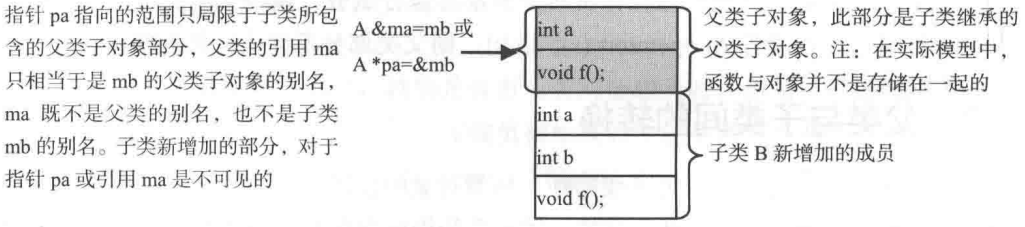


图 13.1 类对象指针或引用之间的自动转换模型

（5）不存在从父类到子类的转换（包括指针、引用与类对象），因为父类不知道子类拥有的成员。比如 class A{};class B:public A{}; A ma;，则 B mb=ma;和 B *pb=&ma;都是错误的。

（6）子类到父类是否能进行自动转换的规则：观察父类的公有成员能不能被访问，若能被访问，则可进行自动转换，否则不可进行自动转换。主要分为以下几种情形。

① 公有继承时（常用情形），可进行自动转换。

② 非公有继承时，在类外是不能进行自动转换的。比如 class A{}; class B:private A{}; B mb;，则 A &ma=mb;错误，因为在类外不能访问父类 A 的公有成员。

③ 不管以何种方式继承，在子类本身的成员内是可以进行自动转换的，因为子类成员总是可以访问父类的公有成员。比如 class A{};class B:private A{public: void f(){B mb; A*pa=&mb; };，其中 A*pa=&mb;的自动转换是正确的。

④ 不管以何种方式继承，在子类的友元函数内是可以进行自动转换的，因为子类的友元函数可以访问父类的公有成员。比如 class A{};class B:private A{friend void f(); };void f(){ B mb; A &ma=mb;}，在友元函数 f 内 A &ma=mb;的自动转换是正确的。

⑤ 若子类以受保护方式继承，则该子类的后续子类不管以何种方式被继承，在后续子类的成员函数内都可以进行自动转换，因为在后续子类的成员函数内是可以访问父类的公有成员的。比如 class A{};class B:protected A{};class C:private B{public: void f(){ B mb; A& ma=mb;}};，其中

A& ma=mb;可进行自动转换，因为在类 C 的成员函数 f 内可以访问类 A 的公有成员。

（7）对象切除（自动转换与初始化/赋值的不同）。

① 自动转换不存在于类对象之间，只存在于类对象的指针（或引用）之间。也就是说，没有从子类对象到父类对象的自动转换（注意：此处指的是类对象，而不是类指针或引用）。这种行为不是错误的，会发生对象切除。

② 对象切除：把子类对象赋值给父类对象时，它们之间是复制或赋值的关系，其原理是以子类对象为实参，调用父类的复制构造函数或赋值操作符对父类对象进行初始化或赋值（注意：这种行为并不是自动转换）。在默认情形下，父类的默认复制构造函数或默认赋值操作符只会复制或赋值子类的父类部分，因此初始化或赋值之后将只保留子类的父类部分，这样子类部分就被“切除”了。比如 class A{}; class B:public A{}; B mb;，则 A ma=mb;是使用 mb 作为实参调用类 A 的默认复制构造函数初始化 ma 的，此处并不是自动转换，而是初始化，类 A 的默认复制构造函数只把子类 mb 的父类部分复制给了父类对象 ma，子类 mb 的子类部分被切除了。对于 ma=mb;原理相同，只不过使用的是默认赋值操作符函数。

③ 自动转换与初始化/赋值区别的具体体现：主要体现在“能否进行自动转换”上，因为能否初始化或赋值的限制与能否进行自动转换的限制是不同的，这又表现在调用复制构造函数或赋值操作符函数进行参数传递时。以复制构造函数为例，父类的默认复制构造函数要求一个父类类型的引用作为实参，而在进行复制初始化时若传递给父类的复制构造函数一个子类对象，那么这就存在子类与父类引用之间的自动转换；若子类是私有继承自父类，则这种初始化就会失败，因为私有继承时不能进行自动转换，但我们可以自定义父类的复制构造函数并把其形参修改为子类的引用，从而避免这种自动转换。

示例如下：

```
class B; class A{public:A(){} A(B& ma){} }; class B:private A{}; B mb;
```

A ma=mb;是正确的，因为 A ma=mb;会以子类对象 mb 为实参调用父类 A 的复制构造函数 A(B&)，但复制构造函数的形参 B&与 mb 的类型是一致的，不存在转换，所以正确。此处可以看到自动转换与复制初始化和赋值的区别。

```
class A{public:A(){}A(A& ma){} };class B:private A{}; B mb;
```

A ma=mb;是错误的，因为 A ma=mb;会以子类对象 mb 为实参调用父类 A 的复制构造函数 A(A&)，这之间存在子类对象 mb 到父类引用 A&之间的自动转换，在私有继承下，这种自动转换是不允许的，所以出错。即使父类 A 使用的是系统合成的默认赋值构造函数，A ma=mb;也是错误的，其原理相同。

13.1.4 继承下的名称解析、名称隐藏及函数重载解析

1. 继承下的名称解析和名称隐藏

（1）名称解析（查找）的常规规则详见 7.5.1 节。类作用域中的名称解析规则详见 8.3.2 节。

（2）父类和子类是两个不同的类作用域：在继承情形下，子类作用域是被嵌套在它的直接父类作用域中的，因此若不能在子类中找到名称，就在外围的父类中查找，这也是子类能够直接访问父类成员的原因。也正因为如此，父类成员的名称会被子类的相同成员名称覆盖或隐藏。

（3）名称隐藏：若子类声明了与父类相同的名称，则子类的名称会隐藏父类中的所有相同名称，此时子类的对象将调用子类中的版本。注意：对于成员函数，子类并不是对父类的函数重载。比如 class A{public: void g(int ,int){} void g(int){}}; class B:public A{public: void g(){}};，则子类 B 中定义的名称 g 隐藏了父类 A 中的所有 g 函数版本，注意此处并不是对父类函数 g 的重载。比如 B mb;，则 mb.g();正确，调用的是子类中无形参的函数 g；而 mb.g(2)或 mb.g(2,3);是错误的，因为父类中的函数 g 对于子类对象 mb 而言不可见（被子类中的同名函数 g 隐藏了）。

（4）函数重载与名称隐藏：函数重载的前提是“重载的函数应在相同的作用域中（带类类型形参的函数除外）”。因为子类和父类是两个不同的作用域，所以子类的函数与父类的函数不会构成函数重载，其名称按照名称解析规则进行解析。比如 class A{public: void f(){}}; class B:public A{public:void f(int i){}};，其中子类中的 f(int)和父类中的 f()两个函数不会形成重载，因为它们不在相同的作用域中。关于带有类类型形参的函数详见 12.3.3 节。

（5）继承下的名称解析规则：与类作用域中的名称解析规则相同，但要注意名称查找总是先查找类作用域。比如 class A{public:int a; };int a; class B:public A{public:void f(){a=3;}};，子类函数 f 中的 a=3;访问的是父类的成员 a，而不是全局变量 a。要访问全局变量应使用::a=3;语法。其名称查找顺序是，首先查找函数 f 中的名称，然后查找子类 B 中声明的名称，接下来查找父类 A 中声明的名称（若此时找到，结束名称查找），最后查找全局名称。

（6）私有成员虽然不会被继承，但在名称解析时，私有成员的名称仍是可见的。比如 class A{private: int a;}; int a; class B:public A{public: void f(){a=3; } };，子类中的 a=3;是错误的，因为此处的 a 指的是父类的私有成员 a，而非全局变量 a，它是不可访问的。虽然父类 a 的私有成员不会被继承，但名称查找时仍是可见的。

（7）访问被隐藏的父类成员及实现函数重载。

① 方法 1：使用作用域解析运算符“::”。

示例如下：

```
class A{public: int a;  void f(){}};
class B:public A{public:int a;  void f(int i){A::a=1;} }; //A::a 指的是父类 A 的成员 a
B mb;  mb.A::f();  mb.A::a=3;  //指的是父类 A 的成员 a 和 f，而非子类的成员 a 和 f
```

② 方法 2：重定义父类中需要使用的重载成员函数，以达到重载的目的，但如果父类有 *N*

个重载的函数，子类就要重定义 *N* 个版本。此方法烦琐。

示例如下：

```
class A{public:void f(int i){}};
class B:public A{public: void f(){}
void f(int j){A::f(j);}};//对父类中函数 f 的重定义，在其内部调用父类的函数 f 以达到重载的目的
```

③ 方法 3：使用 using 声明把父类的名称引入子类中。若 using 声明引入的是父类中函数的名称，则父类的成员函数与子类的成员函数可构成函数重载。比如 class A{public: void f(){}};class B:public A{public: using A::f; void f(int i){} };B mb;，则 mb.f();将调用父类中的成员函数 f，此时父类中的 f()和子类中的 f(int)版本构成函数重载。

④ 方法 4：使用旧式的类名和作用域解析运算符引入父类中的名称。此方法不推荐使用，因为有可能会被 C++淘汰，建议使用 using 声明。比如 class A{public: void f(){}};class B:public A{public: A::f; void f(int i){} };，其中 A::f 就表示引入父类中的成员函数。

（8）使用 using 声明的注意事项：

① 使用 using 声明引入的成员变量在子类中不能被重定义。比如 class A{public: int a;}; class B:public A{public: using A::a; int a;};，其中 int a;会出现重定义错误。

② 函数覆盖：使用 using 声明引入的成员函数是可以在子类中被重定义的，也就是说，可以在子类中覆盖父类中所引入的成员函数的某个版本。例如：

```
class A{public: void f(int i){} void f(){}void f(int i,int j){}};
class B:public A{public: using A::f; void f(int i){}  }; //覆盖使用 using 声明引入的 f(int)
                                                          //版本
```

③ 使用 using 声明引入的成员函数被子类覆盖后，只能使用作用域解析运算符调用父类中被覆盖的版本。例如：

```
class A{public: void f(int i){}  void f(){}  void f(int i,int j){}};
class B:public A{public: using A::f;  void f(int i){A::f(3); } };  //A::f(3);调用父类
                                                                   //A 中的 f(int)
B mb;  mb.A::f(3);    //其中 mb.A::f(3)调用父类 A 中的 f(int)版本
```

2. 继承下的函数重载解析

（1）继承时候选函数的确定。

① 在继承时，若函数是普通调用，且实参是类类型的对象、类类型的引用或类类型的指针，则候选函数在以下范围内查找：调用点上可见的函数；在定义实参所属类或父类的名称空间中的函数，应注意类实际来自于哪个名称空间；在实参所属类或父类中所声明的友元函数。

示例如下：

```
namespace N{class A{friend void f(int i, A& ma){}};void f(char c,A ma){}}
namespace M{class B:public N::A{ friend void f(double d,B mb){}};
void f(float f,B mb){}}
```

说明：对于 M::B mb;，f(3,mb)的调用有 4 个候选函数，分别是实参 mb 所属类 B 所在名称

空间 M 中的函数 f(float,B)、类 B 中的友元函数 f(double,B)、实参 mb 所属类 B 的父类 A 所在名称空间 N 中的函数 f(char,A)和类 A 中的友元函数 f(int,A)。

② 对于使用类对象调用类中的成员函数的情况，主要应注意名称解析的规则。

（2）继承时可行函数的确定：主要应注意单形参构造函数对可行函数的影响，在继承时转换函数可以被继承，而单形参构造函数是不会被继承的，因为它是构造函数。比如 class A{public: A(){}A(int i){} operator int(){return 2;}};class B:public A{};void f(int i){} void g(B mb){} B mb1;，则 f(mb1);的调用是正确的，实参 mb1 会调用父类 A 的转换函数 operator int 把 mb1 转换为 int 类型；而 g(3)的调用是错误的，因为父类 A 的单形参构造函数没有被子类 B 继承，因此无法使用父类的单形参构造函数把整型值 3 转换为类类型 B。

（3）继承时最佳匹配函数的确定：在继承时，存在以下几种类型的转换。

① 从子类对象到父类对象的转换，这是标准转换。

② 从子类指针（引用）到父类指针（引用）的转换，这也是标准转换。

③ 在从子类到父类进行的标准转换中，还可进行等级划分，即父类离子类越近的转换越好。对于多重继承也是如此。注意：这是子类到父类的这种标准转换与其他标准转换不同的地方。

④ 从子类指针到父类指针的转换，比到 void*的转换匹配得好。

⑤ 不存在从父类对象（指针或引用）到子类的转换。

⑥ 标准转换比用户定义的转换（即使用转换函数进行的转换）匹配得好。

示例如下：

```
class A{public:operator int(){return 2;}}; class B:public A{};class C:public B{};
void f(A ma){} void f(B mb){} void f(int i){} void g(B *pb){} void g(void *p){}  C mc;
```

说明：

- 调用 f(mc);，将调用函数 f(B);，其中函数 f 的三个版本都是可行函数。由实参 mc 所属的子类类型 C 转换为父类类型 B，比由子类类型 C 转换为父类类型 A 更近，因此 f(B)比 f(A)匹配得好。对于 f(int)，因为进行的是用户定义的转换，因此 f(A)和 f(B)都比 f(int)匹配得好。
- 调用 g(&mc);，将调用函数 g(B*);，从子类指针到父类指针的转换，比到 void*的转换匹配得好。

示例 13.4：继承时候选函数的确定方法

```
#include<iostream>
using namespace std;
namespace S{class C;}   //前置名称空间 S
namespace N{
   namespace M{class A{};
         void h(S::C &mc){cout<<"HM"<<endl;}
         void h(A &ma){cout<<"HAM"<<endl;}}  //名称空间 M 结束
```

```
    class B{public: friend void h(char c,S::C &mc){cout<<"HNF"<<endl;}};
    void h(int i, S::C &mc)  {cout<<"HN"<<endl;}}  //名称空间 N 结束
namespace S{class C:public N::B,public N::M::A{public: //类 C 继承自类 B 和 A
        friend void h(double d,C mc){cout<<"HSF"<<endl;}};
    void h(float i, C mc){cout<<"HS"<<endl;}    }  //名称空间 S 结束
void main(){S::C mc;
    //本例中实参带类 C 类型的函数 h 的候选函数有以下 6 个
    //定义 mc 所属类 C 的名称空间 S 中的函数 h(float,C)
    //定义类 C 的父类 B 的名称空间 N 中的函数 h(int,S::C&)
    //定义类 C 的父类 A 的名称空间 M 中的函数 h(S::C&)
    //定义类 C 的父类 A 的名称空间 M 中的函数 h(A&)
    //类 C 中声明的友元函数 h(double,C)
    //类 C 的父类 B 中声明的友元函数 h(char,S::C&)
    h(3,mc);          //正确，输出 HN，调用名称空间 N 中的函数 h(int,S::C&)，即第 2 个
    h(mc);            //正确，输出 HM，调用名称空间 M 中的函数 h(S::C&)
    h(3.3, mc);       //正确，输出 HSF，调用在名称空间 S 中定义的类 C 中定义的友元函数 h(double,C)
                      //因为名称空间 S 中的 h(float,C)没有 h(double,C)匹配得好
    h('a',mc); } //正确，输出 HNF，调用在名称空间 N 中定义的类 C 的父类 B 中定义的友元函数 h(char,S::C&)
```

13.1.5　多重继承与虚基类

关于潜在的二义性错误，指的是若进行了不明确的调用才会发生的错误。从某种意义上讲这并不算真正的错误，只要进行明确调用，该错误就不会存在。

1. 多重继承基础

（1）多重继承：指的是一个子类从多个父类继承，各父类之间用逗号隔开，且在每个父类之前还应指定访问级别。同单继承相同，默认的访问级别若是使用 class 声明的，则是私有继承；若是使用 struck 声明的，则是公有继承。比如 class A:public B, public C{};，表示子类 A 以公有方式从父类 B 和 C 继承；class A: public C ,B{};，表示子类 A 以公有方式从父类 C 继承，以私有方式从父类 B 继承。

（2）多级继承：指的是某个被继承的类本身是一个子类。比如 class A{};class B:public A{}; class C:public B{};，其中类 C 的继承就经过了多级，它的父类 B 是类 A 的子类。

（3）在多重继承下，从父类继承的成员的访问级别与单继承相同，根据私有的、公有的、受保护的继承方式而定。比如 class A{public: int a;}; class B{public: int b;}; class C:public B, private A{};，则类 C 继承自类 B 和 A，继承后类 A 的成员 a 在类 C 中为私有的，而类 B 的成员 b 在类 C 中为公有的。

（4）子类与父类间的转换：与单继承相同，多重继承时子类的指针、引用或对象会被自动转换为由公有继承而来的父类的指针、引用或对象。多重继承时编译器还会对 this 指针进行相应的调整，详见第 14 章。

2. 多重继承下的构造函数和复制控制

（1）多重继承下的构造函数：与单继承相同，多重继承时，父类的成员可由子类的构造函数通过成员初始化表的形式调用父类的构造函数进行初始化。比如 class C:public B, public A{public: int c; C(): c(9), B(3),A(){}};，其中 A(): c(9), B(3), A()表示子类 C 的成员 c 由 9 进行初始化；父类 B 的成员使用带一个形参的构造函数进行初始化；父类 A 的成员由默认构造函数进行初始化。

（2）多重继承时初始化顺序：与单继承相同，子类的成员初始化表不能控制初始化顺序，其父类的初始化顺序是按照继承时各父类出现的次序进行的，不管父类的构造函数是否出现在子类的成员初始化表中，父类的构造函数总是最先被调用。

示例如下：

```
class A{}; class B {}; class C:public B{};
class D:public C, public A{public: int d; D():d(8), A(){} };
```

说明：子类 D 初始化的顺序为类 B 的默认构造函数、类 C 的默认构造函数、类 A 的默认构造函数、类 D 的成员 d。原因是，因为在继承时类 C 出现在类 A 之前，因此调用类 C 的默认构造函数初始化类 C 的成员；而类 C 继承自类 B，所以在这之前先调用类 B 的默认构造函数初始化类 B 的成员，然后调用类 A 的默认构造函数初始化类 A 的成员，最后是初始化类 D 的成员 d。可以看到，虽然此处 d 出现在成员初始化表的最前面，但却是最后初始化的。

（3）多重继承下的析构函数按照与调用构造函数相反的顺序进行调用。

（4）多重继承下的复制构造函数和赋值运算符的规则与单继承时相同，只是要注意多重继承下的调用顺序是按照继承时父类出现的次序进行的。

3. 多重继承下的名称冲突及函数重载

（1）多重继承下的名称解析（查找）及二义性错误。

① 多重继承时，所有父类继承子树中的名称查找是并行的（也就是同时进行的），即各父类子树中出现的名称是没有先后顺序之分的。

② 若继承而来的不同父类继承子树中有相同的名称（注意，这并不是错误的行为），并进行了不明确的引用时，将发生二义性错误；若没有这种不明确的引用，则不会发生错误。注意：这里只是一种潜在的二义性错误，从某种意义上讲这并不是真正的错误。

③ 解决名称二义性错误的办法就是使用作用域解析运算符进行明确调用。

示例如下：

```
class A{public: int a;}; class B{public: int a; };class C:public B{};
class D:public C, public A{}; D md;
```

说明：对 md.a;的调用具有二义性，首先在类 D 中查找名称 a，若未找到，然后同时在从子类 D 继承的父类 C 继承子树和父类 A 继承子树中进行查找，同时找到 a，但它们并没有先后顺

序之分，因此调用不明确。注意：父类 C 的继承子树包括类 C 的父类 B。解决方法是使用作用域解析运算符进行明确调用，比如 md.C::a;、md.A::a;和 md.B::a;的调用都是正确的。

（2）多重继承下的函数重载及二义性错误。

① 同时从不同父类继承的成员函数不能在子类中构成函数重载，因为它们在不同的作用域中。

② 若不同的父类中有相同名称的函数（形参是否相同并不重要），则按照多重继承下的名称解析规则进行解析；若进行了不明确调用，则产生潜在的二义性错误。

③ 多重继承时，在子类中实现函数重载的方法与单继承相同，可以使用 using 声明、旧式的类名和作用域解析运算符声明、在子类中重定义等方法。

示例如下：

```
class A{public: void f(){};class B{public: void f(int i){};
class C:public B, public A{}; C mc;
```

说明：对 mc.f()或 mc.f(3);的调用是错误的，因为父类 B 和 A 中的成员函数 f 并不会在子类 C 中构成函数重载，因此此处按照多重继承时的名称解析规则对名称 f 进行解析，分别在父类 A 继承子树和父类 B 继承子树中找到相同的名称 f（它们无先后顺序之分），所以调用出现二义性。而 mc.A::f();和 mc.B::f(9);是正确的，表示分别调用类 A 和类 B 中相应的函数。

（3）子类与父类间转换时的二义性错误：主要存在于调用时对函数形参的转换中，比如 class A{};class B{};class C:public B, public A{};void f(A& a){} void f(B& b){}; C mc;，则 f(mc);的调用存在二义性错误，因为不能确定是把对象 mc 转换为类 A 还是类 B 的引用。

4．共享共同父类副本与虚基类

（1）多重继承下的多个父类副本与二义性错误：若由子类继承而来的多个直接父类又同时继承自某一个共同父类，则这个共同父类会在最终的子类中存在多个相同的副本，这时若子类使用了不明确的调用就会出现潜在的二义性错误。其解决方法有两种，一种是使用作用域解析运算符进行明确调用，一种是使用虚基类。比如 class D{public: void f(){}};class B:public D{};class C:public D{};class A:public B, public C{};A ma;，则 ma.f();的调用存在二义性错误，因为子类 A 同时拥有间接的共同父类 D 的两个相同的副本，一个是由类 B 继承而来的，一个是由类 C 继承而来的，因此解决方法是使用作用域解析运算符进行明确调用，如 ma.B::f();或 ma.C::f();，但 ma.D::f();是错误的，在作用域解析运算符中使用共同父类不能解决二义性错误。

（2）虚基类与虚继承：在继承时访问级别部分使用 virtual 关键字，访问控制符与 virtual 的顺序无关紧要，被继承的父类被称为虚基类，这种继承方式被称为虚继承。注意虚基类不是在声明基类时确定的，而是在继承时才确定的，因此也可以说被某（些）类继承为虚基类，或被继承为某（些）类的虚基类。比如 class B:public virtual D{};就是一种虚继承方式，类 D 被继承为类 B 的虚基类，因此，类 D 是虚基类，虚基类 D 不是在声明类 D 时确定的。

（3）虚基类的作用：虚基类只有一个副本，可避免引起二义性错误。在多重继承时，由子类继承而来的多个直接父类又同时继承自某一个共同父类，若这个共同父类被所有直接父类继承为虚基类，那么该父类就只会有一个副本，这样就避免了在使用最终子类引用该父类的成员时引起二义性错误比如 class D{public: void f(){}};class B:virtual public D{};class C:virtual public D{};class A:public B, public C{};A ma;，则 ma.f();的调用就是正确的，这里共同父类 D 被继承为其所有子类 B 和 C 的虚基类，类 D 相对于类 B 和类 C 而言只有一个副本，调用是明确的。

（4）为保证虚基类在所有子类中只有一个副本，应使该虚基类的所有直接子类都以虚继承方式被继承，否则仍会出现共同父类的多个副本。比如 class A{public:void f(){} }; class B:virtual public A{};class C:virtual public A{};class D:public A{};class E:public B,public C,public D{};E me;，则 me.f();的调用具有二义性。这里类 B 和类 C 都虚继承自类 A，而类 D 非虚继承自类 A，因此对类 E 而言存在两个类 A 的副本（一个是类 D 的副本、一个是类 B 和类 C 的共同虚基类副本），这样在类 E 中引用类 A 的成员时就会出现二义性错误。

5. 虚继承下的构造函数及调用顺序

（1）虚基类对性能影响比较大，因此很少被使用。后文所讲的虚基类都是指其所有直接子类都以虚继承方式被继承的情形。虚基类的直接子类部分以虚继承方式被继承的情形不作讨论。

（2）初始化虚基类的基本原理：在常规情形下，子类只能初始化其直接父类，若虚基类经过它们的各直接子类对其进行初始化，则会产生多个虚基类的副本，这与虚基类只拥有一个副本相矛盾，因此虚基类是允许间接子类直接调用其构造函数对它进行初始化的。比如类 A 被继承为类 B 和类 C 的虚基类，而类 D 又同时继承自类 B 和类 C，若类 D 无法直接初始化虚基类 A，而经过类 B 和类 C 对虚基类 A 进行初始化，则虚基类 A 会拥有类 B 和类 C 产生的两个副本，这与虚基类只拥有一个副本相矛盾。因此 C++允许在类 D 中对虚基类 A 进行直接初始化，这时在类 B 和类 C 中对虚基类 A 的初始化将会被忽略。

（3）最终子类：在虚继承情形下，虚基类的初始化是由最终子类负责的，最终子类指的是创建对象时的类。比如 class A{};class B:public A{};class C:public B{}; A ma; B mb; C mc;，则对于 ma 最终子类是类 A，对于 mb 最终子类是类 B，对于 mc 最终子类是类 C。

（4）初始化虚基类的具体规则：此处的规则适应于虚基类的直接子类都是使用虚继承方式继承的情形，而不适用于 class A{};class B:virtual public A{};class C:public A{};形式，此处类 C 并不是虚继承自虚基类 A 的。

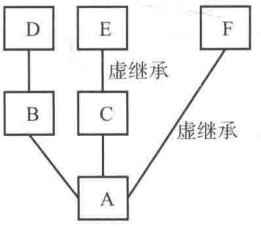
① 虚基类是由最终子类的构造函数使用成员初始化表的形式直接调用其构造函数进行初始化的。

② 若虚基类的构造函数未出现在最终子类的成员初始化表中，则使用虚基类的默认构造函数进行初始化；若虚基类没有默认构造函数，则出错。

③ 若最终子类不是虚基类的直接子类，则最终子类的中间父类对虚基类的初始化被忽略。

（5）虚基类成员的可见性：若某个名称经过多条路径被继承，则子类重定义的名称的优先级高于虚基类中共享的名称。比如 class A{public:int a;}; class B:virtual public A{}; class C:virtual public A{public: int a;}; class D:public B, public C{};D md;，则 md.a 访问的是父类 C 中定义的名称 a，该名称 a 的优先级高于虚基类 A 中共享的名称 a。

（6）有虚基类时的初始化顺序：若继承层次中有虚基类，则虚基类的初始化总是在非虚基类之前进行，详见图 13.2。



假设类 A 的形式为 class A:public C, public B, virtual public F{};，则创建类 A 对象时初始化顺序为：虚基类 E（由类 A 间接初始化）、虚基类 F、类 C、类 D、类 B、类 A

图 13.2　有虚基类时的初始化顺序

（7）有虚基类时析构函数的调用顺序总是与构造函数相反。

（8）初始化顺序可总结为：虚基类、父类、对象成员、数据成员。

示例 13.5：初始化虚基类的规则及虚基类成员的可见性

```
#include<iostream>
using namespace std;
//以下示例是指虚基类的所有直接子类都以虚继承方式被继承的情形
class A {public: A(){cout<<"A"<<endl;} A(int i){cout<<"AI"<<endl;}
    void f(){cout<<"FA"<<endl;}}; //类A结束
class B:virtual public A{public:B(){cout<<"B"<<endl;}
    B(int i):A(i){cout<<"BI"<<endl;}}; //类B结束
ClassC:virtualpublicA{public: C(){cout<<"C"<<endl;} C(int i):A(i){cout<<"CI"<<endl;}
    void f(){cout<<"FB"<<endl;}}; //类C结束
class E:public B,public C{public:    E(){cout<<"E"<<endl;}
   E(int i):B(i),C(i),A(){cout<<"EI"<<endl;} }; //类E结束
class D:public E{public:    D(){cout<<"D"<<endl;}
   D(int i):A(),E(i){cout<<"DI"<<endl;}}; //类D结束
void main(){D md(9);    //调用顺序见下文说明
    md.f();}        //访问父类C中定义的函数f，父类C中函数f的优先级高于虚基类A中共享的函数f
```

创建 md(9)时依次输出 A, BI, CI, EI, DI，构造函数的调用顺序如下。

① 虚基类初始化规则：md 的最终子类是类 D，虚基类由最终子类 D 进行初始化，其余最终子类的中间父类（即类 E、类 C、类 B）对虚基类 A 的初始化都会被忽略。

② 调用虚基类 A 的默认构造函数，由最终子类 D 中带一个形参的构造函数直接调用虚基类 A 的默认构造函数初始化虚基类 A。

③ 调用类 B 中带一个形参的构造函数，由类 E 按照被继承的顺序，使用类 B 中带一个形参的构造函数初始化类 B，此时忽略类 B 的构造函数对虚基类 A 的构造函数的调用（因为相对于最终子类 D 而言，类 B 是虚基类 A 的中间父类）。

④ 调用类 C 中带一个形参的构造函数，其调用原理同类 B。

⑤ 调用类 E 中带一个形参的构造函数，此时忽略类 E 的构造函数对虚基类 A 的构造函数的调用；同理，相对于类 D 而言，类 E 是虚基类 A 的中间父类。

⑥ 调用类 D 中带一个形参的构造函数。

13.2 虚函数与多态性

13.2.1 多态性原理

1. 多态性概念

对多态性最简单的理解就是一种事物有多种形态。在面向对象设计中，多态性指的是向不同的对象发送同一消息时，会产生不同的动作（或行为、功能）。所谓的“向不同的对象发送同一消息”，其实就是指调用不同对象的某个函数；而“产生不同的动作”，指的是该函数会实现不同的功能。没有实现多态性的程序设计语言不能称为真正的面向对象语言。比如语句 pa->f();，若 C++实现了多态性时，则会根据 pa 所指向的类对象的不同而调用相同的函数 f（在语句形式上是相同的），这就是“向不同的对象发送同一消息”，这里的同一消息指的是调用函数 f()，然后程序根据函数 f 所在的类不同会实现不同的功能，这就是“产生不同的动作”。再比如 A、B、C 是同一年级而不同班级的三个学生，当听到上课铃声时，他们会进入不同的教室去上课。A、B、C 三个对象在收到同一消息“上课铃声”时，执行了不同的动作，这就是多态性。至于多态性怎样实现，根据程序设计语言而定。

2. 动态类型与静态类型

（1）静态类型指的是对象（指针、引用）在声明时的类型。这种类型在编译阶段就能确定，仅从表达式的字面形式就能够确定其类型。静态类型在程序运行阶段不会改变。

（2）动态类型指的是当前对象（包括指针和引用）实际指向的类型。这种类型需要在运行阶段才能确定，对象的动态类型可以更改。

（3）编译阶段与运行阶段的区别：在编译阶段，系统只做静态的语法检查，即从表达式的字面形式（或语句形式）就能够确定信息。比如 A* pa，则对于 pa->f();，在编译阶段从表达式的语句形式是无法确定指针 pa 所指向的类型（动态类型）的，因为 pa 可指向不同的子类类型对象，这就需要等到运行阶段才能确定。但 pa 自身的类型（即类 A 类型），在编译时是可以确

定的，就是静态类型。

示例如下：

```
A *pa=&mb;
```

这里 pa 的静态类型是 A*，pa 的动态类型是 mb 的类型。pa 的动态类型可以改变（只需重新指向另一个对象即可），而静态类型无法更改。pa 的动态类型需要在程序运行阶段才能确定，在编译阶段是无法确定的。

```
A* pa=new A();
```

此时 pa 的静态类型和动态类型都是 A*。

3．C++中的动态类型与静态类型

（1）在 C++的继承关系中，指针或引用的动态类型与静态类型可以不同，这是 C++实现多态性的关键。比如 class A{};class B:public A{};B mb;，则对于 A *pa=&mb;，此时 pa 的静态类型为 A*，动态类型为 B*，可见 pa 的静态类型与动态类型并不相同。

（2）在继承关系中，类对象（类指针和类引用除外）的静态类型与动态类型总是相同的。因为在把子类对象赋给父类对象时（注意：父类对象不能赋给子类对象），会把子类的父类部分“切除”，然后把“切除”部分赋给父类对象，这时父类对象的类型相当于是子类被“切除”的父类类型（其实就是父类类型）。比如 class A{};class B:public A{};B mb; A ma=mb;，这时 ma 的动态类型与子类 mb 被“切除”的父类类型 A 相同，即 ma 的动态类型就是类型 A，这与 ma 的静态类型是相同的。

4．动态绑定与静态绑定

（1）绑定（binding）：确定调用的是哪个具体对象的行为。在本文中，绑定一般是指把某个函数名与某一个类对象绑定在一起的行为。

（2）动态绑定：在程序执行时（运行阶段）绑定到对象的动态类型的行为，其特性依赖于该对象的动态类型。动态绑定后的结果就是根据所绑定的动态类型，调用动态类型中的函数。动态绑定又被称为后期绑定。

（3）静态绑定：绑定到对象的静态类型的行为。静态绑定后的结果就是根据所绑定的静态类型，调用静态类型中的函数，该过程在编译阶段进行。静态绑定又被称为早期绑定。

示例如下：

```
class A{public:void f(){}}; class B:public A{public:void f(){}};
B mb; A *pa=&mb; pa->f();
```

① 若函数 f 绑定到指针 pa 的动态类型（即动态绑定），则调用 pa 的动态类型中的函数 f，此时 pa 的动态类型是 B*，因此 pa->f();调用类 B 中的函数 f。可见，动态绑定时调用哪一个函数，是由指针所指向的动态类型决定的。

② 若函数 f 绑定到指针 pa 的静态类型（即静态绑定），则调用 pa 的静态类型中的函数 f，

此时 pa 的静态类型是 A*，因此 pa->f();调用类 A 中的函数 f。

5．动态绑定与多态性、虚函数

（1）C++中的多态表示的是使用一个公有的父类指针（或引用），寻址出一个子类对象。换句话说，就是“一个接口，多个方法”。在非多态情形下，当声明一个父类指针指向子类对象时，这个父类指针只能访问父类中的成员函数，不能访问子类中特有的成员变量或函数（因为父类并不知道子类特有的成员）。C++多态性的实质就是为了实现“使用指向子类对象的父类指针（或引用）访问子类中的成员函数，而不是父类中的成员函数”。比如，若类 B 和类 C 是类 A 的子类，且都有各自的成员函数 f，则对于 B mb; C mc; A* pa;，若 pa=&mb;，那么在多态情形下，pa->f()将调用类 B 中的成员函数 f()；同理若 pa=&mc;，那么 pa->f()将调用类 C 中的成员函数 f；若没有多态，则无论 pa 指向类 A 的哪个子对象，pa->f()都将调用类 A 中的函数 f。这就是 C++中的多态，即根据共有的父类指针 pa，通过 pa 指向不同的子对象，可以调用不同子对象中的成员函数。

（2）C++使用动态绑定实现多态性，静态绑定是不能实现多态性的。比如，若类 B 是类 A 的子类，且都有各自的成员函数 f，则对于 B mb; A* pa;，若 pa=&mb;，要实现多态性就必须使 pa->f()调用子类 B 中的函数 f，而这种调用就是动态绑定，因此只有动态绑定才能实现多态性；若 pa->f()的调用是静态绑定，则无法调用子类 B 中的函数 f。

（3）C++实现动态绑定：在 C++的继承关系中，动态绑定必须使用虚函数和父类类型的指针（或引用）来实现，这两者缺一不可。

① 为什么需要虚函数：在 C++中，只有虚函数才有可能使用动态绑定，虚函数是实现动态绑定的条件之一，非虚函数全部使用静态绑定。

② 为什么需要指针或引用：在 C++中，指针或引用的动态类型与静态类型可以不同，但类对象的静态类型与动态类型总是相同的，只有在静态类型与动态类型不相同时，使用动态绑定才能实现多态性，因此使用类对象无法绑定到动态类型，即无法实现动态绑定，动态绑定必须使用指针或引用才能实现。

③ 为什么必须是父类的指针或引用：在 C++中，子类可以转换为父类，但不存在从父类到子类的转换，因此要实现动态绑定应使用父类类型的指针或引用。不管是通过动态绑定还是静态绑定，把子类对象绑定到子类类型、把父类对象绑定到父类类型都没有实际意义，而把父类对象绑定到子类类型是错误的，因此只有把子类类型动态绑定到父类类型才有意义。

示例如下：

```
class A{public:void f(){}};class B:public A{public:void f(){}};
B mb; A ma; A *pa; B* pb;
```

① 对于 pa=&mb; pa->f();：

- pa 的静态类型为 A*，动态类型为 B*，其静态类型与动态类型是不同的。

- 若函数 f 是虚函数，则通过动态绑定把函数 f 绑定到指针 pa 的动态类型，调用类 B 中的函数 f。只有此情形才可实现多态性。
- 若函数 f 是非虚函数，则通过静态绑定把函数 f 绑定到指针 pa 的静态类型，调用类 A 中的函数 f。因为使用的是静态绑定，所以未实现多态性。

② 对于 pa=&ma; pa->f();：

- pa 的静态类型为 A*，动态类型为 A*，其静态类型与动态类型是相同的。
- 若函数 f 是虚函数，则通过动态绑定把函数 f 绑定到指针 pa 的动态类型，调用类 A 中的函数 f。因为静态类型与动态类型相同，所以未实现多态性。
- 若函数 f 是非虚函数，则通过静态绑定把函数 f 绑定到指针 pa 的静态类型，调用类 A 中的函数 f。因为使用的是静态绑定，所以未实现多态性。

③ pb=&mb; pb->f();，原理同② 。

④ pb=&ma; 或 mb=ma;，错误，不存在从父类到子类的转换。

⑤ ma=mb; ma.f(); mb.f();，此时不管函数 f 是虚函数还是非虚函数，对于类对象使用的都是静态绑定，因此 ma.f()中的函数 f 被绑定到 ma 的静态类型，调用类 A 中的 f 函数。mb.f()中的函数 f 被绑定到 mb 的静态类型，调用类 B 中的函数 f。

⑥ 由以上可见，要实现多态性，必须要满足两个条件，即虚函数和父类类型的指针（或引用）。只有使用虚函数，才可以进行动态绑定；只有使用引用或指针，才能使静态类型与动态类型不同。

（4）**总结**：C++中的多态就是“通过父类的引用（或指针）调用虚函数”，这时发生的是动态绑定，指针或引用的动态类型与静态类型可以不同（因为引用（或指针）既可以指向父类对象，也可以指向子类对象），这是实现动态绑定的关键。用引用（或指针）调用虚函数时，被调用的是引用（或指针）所指对象的实际类型（动态类型）中所定义的虚函数。

13.2.2　虚函数

不管有无虚函数，指针与指针间的赋值使用的都是静态地址(引用原理相同)。比如 class A{}; class B: public A{};A ma; B mb; A* pa=&mb;，则 B* pb=pa; 是错误的，在把 pa 赋给 pb 时，是把 pa 的静态类型的地址（即 A*）赋给指针 pb，因为不存在从父类到子类的转换，所以出错。

1．虚函数的声明

（1）注意：子类必须公有继承自拥有虚函数的父类才能实现多态性，因为只有在公有继承下，类外的子类指针（或引用）才能成功转换为父类指针或引用。比如 class A{}; class B:private A{};B mb;，则 A *pa=&mb;是错误的，因为在私有继承下，子类指针&mb 无法自动转换为父类指针 pa。

（2）包含虚函数的类被称为多态类。

（3）虚函数的声明：需要在父类中声明的函数前面使用 virtual 关键字声明虚函数。比如 class A{public: virtual void f(){};，其中函数 f 就是虚函数。

（4）声明虚函数时的基本规则：

① 只能把类的成员函数声明为虚函数，类外的普通函数不能声明为虚函数。

② 友元函数不能是虚函数，因为它不是成员函数，不过虚函数可以是另一个类的友元。

③ 虚函数不能是静态成员函数。因为 new 和 delete 默认为静态的，因此重载的 new 和 delete 操作符不能是虚拟的。

④ 不能把成员变量声明为虚拟的，即 virtual 关键字不能用在成员变量前面。

⑤ 析构函数可以是虚函数，但构造函数不是。

⑥ 若函数被声明为虚函数，则只有该函数才是虚函数，它的重载版本不会自动成为虚函数。比如 class A{public:virtual void f(){}void f(int i){};，其中只有函数 f()是虚函数，而 f(int)不是虚函数。

⑦ 一旦将函数声明为虚函数，则不管它通过多少层继承，它都是虚函数。比如 D 从 B 继承，而 E 又从 D 继承，那么在 B 中声明的虚函数，在类 E 中仍然是虚函数。

（5）赋值操作符函数不应定义为虚函数，因为子类的赋值操作符函数的形参与父类的赋值操作符函数不同（一般情况下都是对各自类类型的引用）。在形参不同的情况下，若父类的赋值操作符函数为虚拟的，则子类的赋值操作符函数会隐藏父类的赋值操作符函数。比如父类的赋值操作符函数为 virtual A& operator =(A&){return *this;}，则子类的赋值操作符函数应为 B& operator =(B&){return *this;}，子类的 operator =(B&)的形参与父类的 operator =(A&)不同，因此子类的赋值操作符函数会隐藏父类的赋值操作符函数。

2. 虚函数的定义

（1）引入虚函数的类必须定义虚函数，或把虚函数声明为纯虚函数。比如 class A{public: virtual void f();}; 错误，虚函数 f 必须定义或声明为纯虚函数。

（2）可以在类中定义虚函数，也可以在类外定义虚函数。当在类外定义虚函数时，不能再次指定 virtual 关键字。比如 class A{public: virtual void f(); };，则 virtual void A::f(){}是错误的。

（3）在子类中可以对父类中的虚函数进行重定义。若子类未重定义父类中的虚函数，则子类继承父类的虚函数。

（4）若子类只对父类中的虚函数进行重新声明而不定义，则是错误的，因为虚函数必须被定义或把虚函数声明为纯虚函数。

（5）在子类中重定义父类中的虚函数时可以省略 virtual 关键字。

（6）子类重定义虚函数：在子类中重定义虚函数时，该重定义的虚函数必须与父类中的虚

函数的声明完全相同，包括返回类型、存储类说明符、限定修饰符 const 等，否则不是对虚函数的重定义。注意：返回类型有一个特例。

① 若子类中重定义的父类虚函数版本，与父类中的虚函数只有返回类型不同，则是错误的。注意：对于此情形，普通成员函数属于名称隐藏，不会出错。比如 class A{public: virtual void f(){};class B:public A{public:int f(){return 2;};，则类 B 中重定义的虚函数 f 是错误的。

② 隐藏虚函数：如果父类定义了一个虚函数，但子类却定义了这个虚函数的重载版本，那么该函数重载版本并不是虚函数，而且子类的这个函数重载版本会隐藏父类的虚函数，这也是在子类中重定义虚函数时应与父类声明完全相同的原因。注意：子类的函数版本与父类的函数版本并不会形成重载，且对这些同名函数的调用还应遵守有虚函数时的名称查找规则（详见后文）。

③ 若子类中重定义的虚函数被重新定义为 static、const 或只是形参有无 const 的差别，那么这不是对虚函数的重定义，它也不是虚函数，而且它会隐藏父类中的虚函数。注意：若把虚函数重定义为 const，则指的是 const 位于函数名括号后面的情形（即形如 void f()const{}）；若 const 位于函数名之前，则属于重定义后返回类型不同的情形（即形如 int const f(){}），这是一种错误的行为。

示例如下：

```
class A{public:virtual void f(){}; class B:public A{public: static void f(){}};
```

类 B 中的函数 f 不是对类 A 中虚函数的重定义。

```
class A{public:virtual void f(int i){}; class B:public A{public: void f(const int i){}};
```

类 B 中的 f(const int)不是对类 A 中虚函数 f(int)的重定义。注意：普通函数对于内置类型形参有无 const 是指同一函数，但虚函数则不是这样的。

```
class A{public:virtual void f(){};class B:public A{public: void f() const {}};
```

类 B 中的函数 f 不是对类 A 中虚函数的重定义。

```
class A{public: virtual int f(){return 2;}}; class B:public A{public:const int f(){return
2;}};
```

这是错误的，因为返回类型不同。

④ 返回类型的特例（返回类型协变）：若父类虚函数返回的类型是一个类类型的指针或引用，则子类重定义的虚函数的返回类型可以是公有继承自父类虚函数返回类型的子类类型的指针或引用，否则不是对虚函数的重定义。此种特性被称为返回类型协变，因为允许返回类型随类类型的变化而变化。注意：只有返回类型是类类型的指针或引用时才是正确的，若返回的是类类型则不适用此规则。

示例如下：

```
class C{}; class D:public C{};  class D1:private C{};
C mc; D md;
   class E{public:  virtual C* h(){return 0;} virtual C h1(){return mc;}
```

```
    virtual C* h2(){return 0;}   };
class F:public E{public:  D* h(){return 0;}  //正确，是对虚函数的重定义，返回类型协变
    //D h1(){return md;}   //错误，重定义的h1只有返回类型不同。只有返回类型是类型的指
                           //针或引用时才属于返回类型协变，才是正确的
    //D1* h2(){}           //错误，只有在返回类型的类之间是公有继承时，返回类型协变才
                           //是正确的，此处D1私有继承自类C，所以出错
};
```

3. 虚函数的调用

（1）虚函数调用基本原理。

① 当通过父类指针（或引用）调用函数时，若调用的是非虚函数，则无论父类类型的指针指向的实际对象是什么类型，调用的都是父类类型中的函数。若调用的是虚函数，则调用的是父类指针所指向的实际对象所属类型中的函数。

② 若不是通过父类类型的指针（或引用）调用函数，而是通过父类类型的对象调用函数，则无论调用的函数是虚函数还是非虚函数，都将调用父类类型中的函数。

③ 虚函数主要用来支持C++的多态性，在其他情形下，虚函数的表现就像普通成员函数一样。也就是说，在通过父类的指针或引用调用虚函数时，才表现出虚函数的特性，这时被调用的函数被绑定到指针或引用的动态类型（即调用的是父类指针指向的对象所属类中的函数）。这种行为在运行时才能确定。比如 class A{public:virtual void f(){}}; class B:public A{public:void f(){}}; B mb; A ma=mb; A* pa=&mb;，则 pa->f();的调用会表现出虚函数的特性，调用的是子类 B 中的函数 f。而 ma.f();调用的是父类 A 中的函数 f，因为 ma 不是指针或引用，因此并不会表现出虚函数的特性。

（2）虚函数是分层的：若某个子类未重定义父类中的虚函数，则将按继承时的逆序调用最近的一个或最近被重定义的虚函数。

示例如下：

```
class A{public:virtual void f(){}};class B:public A{public:void f(int i){}};
class C:public B{public: };C mc; A *pa=&mc;
```

pa->f();将调用类 A 中的虚函数 f，因为类 B 和类 C 都未重定义父类 A 中的虚函数，只有父类 A 中的虚函数是离调用最近的。

```
class A{public:virtual void f(){};class B:public A{public: static void f(){}};
B mb; A* pa=&mb;
```

pa->f();将调用类 A 中的虚函数 f，因为类 B 中的函数 f 不是对父类 A 中虚函数 f 的重定义。

（3）使用类名对虚函数的静态调用：当使用类名和作用域解析运算符对虚函数进行调用时，会改变虚函数的动态绑定规则，这时虚函数会通过静态绑定规则进行绑定，因此虚函数的多态性被打破。比如 class A{public:virtual void f(){}}; class B:public A{public:void f(){}}; B mb; A *pa=&mb;，则 pa->A::f();将调用类 A 中的成员函数 f，因为函数 f 被静态绑定到对象类 A，此处打破了虚函数的多态性。

（4）需要对虚函数进行静态调用的情形。

① 当子类中的虚函数调用父类中的虚函数时，为了避免编写重复的代码，而直接使用已在父类虚函数中完成的操作。若不使用静态调用而直接调用，将会导致调用函数本身而出现无穷递归的错误。比如 class A{public:virtual void f(){}};class B:public A{public: void f(){A::f();}};，若在类 B 中对函数 f 进行直接调用，将导致出现无穷递归的错误。

② 当子类中的虚函数调用父类中定义的纯虚函数时。

③ 当父类中的非虚函数调用父类中的虚函数时，若不使用静态调用，因为成员函数默认是使用 this 指针调用的，this 指针也能实现多态性，这时调用哪个函数需要根据传递进来的对象类型而定。比如 class A{public:virtual void f(){}void g(){A::f();}}; class B:public A{public:void f(){}}; B mb; A* pa=&mb; pa->g();， 此时函数 g 中的 A::f()将调用类 A 中的函数 f；若在函数 g 内是直接调用函数 f 的，即 void g(){f();}，则 pa->g();将调用子类 B 中的函数 f，而不是父类 A 中的函数 f。因为这时在函数 g 中对函数 f 的调用等价于 this->f();，此时 this 指针是父类类型的指针，函数 f 是虚函数，而 pa->g();中的指针 pa 指向的是类 B 类型的指针，pa 被传递给类 A 的 this，因此 this 也是指向类 B 类型的指针，所以 this->f();将调用类 B 中的函数 f。

（5）在构造函数或析构函数中调用虚函数：在父类构造函数中调用的虚函数总是父类中的虚函数版本，因为此时子类对象只有子类的父类部分被构造（创建）出来，而子类部分没有构造出来，它还不是完整的子类对象。对于在析构函数中调用虚函数也是相同的道理，不过析构函数是因为子类对象已经被释放。

在构造函数中调用虚函数示例如下：

```
class A{public: A();virtual void f(){cout<<"A"<<endl;}};
class B:public A{public: void f(){cout<<"B"<<endl;} };
B mb1;
A::A(){A *pa1=&mb1;     pa1->f();}
void main(){B mb;A *pa=&mb;pa->f();}
```

说明：程序输出 ABB。

① 创建全局子类对象 mb1 时，首先调用子类 B 的父类构造函数创建子类的父类部分，此时子类对象只包含子类的父类部分，因此类 A 构造函数中的 pa1->f()只会调用指针 pa1 所指对象 mb1 的父类部分中的虚函数 f()，因此输出 A。

② 创建局部子类对象 mb 时，首先调用子类的父类构造函数构造（创建）子类的父类部分，这时调用类 A 的构造函数，此时 A *pa1=&mb1;指向的 mb1 已经是完整的子类对象（在语句 B mb1 执行完毕后，mb1 已经是完整的了），因此 pa1->f()将调用 pa1 所指对象 mb1 的子类部分中的虚函数 f()，因此输出 B。

③ main 函数中的 pa->f()会调用类 B 中的虚函数，输出 B。

4. 虚函数与默认实参及有虚函数时的名称解析（查找）

（1）虚函数使用的是动态绑定，但其形参使用的却是静态绑定。也就是说，虚函数的默认实参的值是由调用该函数的静态类型确定的，而不是动态类型。因此在虚函数有默认实参时，子类中的虚函数的形参不论有无默认值，当用父类指针调用子类中的虚函数时都会被父类的默认值覆盖，即子类的默认值不起作用。但子类的对象调用该函数时，就不会出现这种情况。比如 class A{public: virtual void f(int i=1){cout<<i<<endl; }}; class B:public A{public:void f(int i=2){cout<<i<<endl;}}; B mb; A *pa=&mb;，则 pa->f(); 将输出 1，虽然调用的是类 B 中的函数 f，但输出的形参的值却是父类 A 中函数 f 的默认实参的值 1。函数 f 的形参使用的是静态绑定，这时形参 i 被静态绑定到 pa 的静态类型 A*，因此使用类 A 中函数 f 的形参的默认值，父类中的默认实参覆盖了子类中的默认实参。

（2）有虚函数时的名称解析（查找）过程。

① 首先确定调用虚函数的对象(引用或指针)的静态类型。

② 然后在该静态类型的类中查找函数名，若未找到，则在直接父类中继续查找，并重复该过程，直到找到为止，否则出错。

③ 若找到了该函数名，就进行常规的类型检查（即检查实参与形参是否匹配），查看该函数的调用是否合法。此步骤保证了父类的成员函数不是虚函数时的正常调用情形。

④ 若函数调用合法，则查看该函数是否是虚函数且是否是通过引用或指针调用的，若是，则编译器会生成代码以便根据对象的动态类型确定调用哪一个函数版本（此步骤是通过虚函数表进行的，见第 14 章），否则就直接调用该函数。注意：调用虚函数时还应注意其层级关系，即从指针指向的动态类型开始向父类查找，一直找到虚函数的重定义版本或指针所属静态类型的虚函数为止。

示例如下：

```
class A{public: void f(){}};class B:public A{};
class C:public B{public:virtual void f(){}};C mc; B* pb=&mc;
```

pb->f();调用顶级父类 A 中的函数 f。分析如下。

① 首先确定 pb 的静态类型为类 B。

② 然后在类 B 中查找名称 f，若未找到，则在直接父类 A 中查找，找到名称 f，检查 f 是否是虚函数。发现 f 不是虚函数，因此停止名称查找，调用此函数。由此可见，类 C 中的虚函数从未被查找过。

```
class A{public: virtual void f(){}};class B:public A{};
class C:public B{public: void f(){}};C mc;B* pb=&mc;
```

pb->f(); 调用类 C 中的函数 f。分析如下。

① 首先确定 pb 的静态类型为类 B。

② 然后在类 B 中查找名称 f，若未找到，则在直接父类 A 中查找，找到名称 f，检查 f 是

否是虚函数。发现 f 是虚函数，则根据 pb 的动态类型 C，在类 C 中查找虚函数 f 的重定义版本。类 C 中的函数 f 是对虚函数的重定义，调用此函数。

```
class A{public:virtual void f(){}};class B:public A{public:void f(int i){}};
class C:public B{public:void f(){}};C mc; B *pb=&mc;
```

pb->f();调用是错误的。分析如下。

① 首先确定 pb 的静态类型为类 B。

② 然后在类 B 中找到名称 f，检查 f 是否是虚函数。发现 f 不是虚函数，因此停止名称查找，调用此函数。但此函数需要一个 int 类型的形参，因此 pb->f();调用是错误的。注意：顶级父类 A 中的虚函数 f 被类 B 中的函数 f(int)隐藏了。

```
class A{public:virtual void f(){}}; class B:public A{public:static void f(){}};
class C:public B{public:void f(){}}; class D:public C{public: void f(){}};
```

对于 C mc;B *pb=&mc;，pb->f();调用类 B 中的静态成员函数 f。分析如下。

① 首先确定 pb 的静态类型为类 B。

② 然后在类 B 中找到名称 f，检查 f 是否是虚函数。发现 f 不是虚函数，因此停止名称查找，调用此函数。注意：父类 A 中的虚函数 f 被类 B 中的静态成员函数隐藏了。

对于 D md;C *pc=&md;，pc->f();调用类 D 中的虚函数 f。分析如下。

① 首先确定 pc 的静态类型为类 C。

② 然后在类 C 中找到名称 f，检查 f 是否是虚函数。发现 f 是虚函数，则根据 pc 的动态类型 D，在类 D 中查找虚函数 f 的重定义版本。类 D 中的 f 是对虚函数的重定义，调用此函数。注意：一旦将函数声明为虚函数，则不管它通过多少层继承，它都是虚函数。虽然中间父类 B 隐藏了顶级父类 A 中的虚函数，但后继子类仍会继承该虚函数。

对于 B mb;A *pa=&mb;，pa->f();调用类 A 中的虚函数 f。分析如下。

① 首先确定 pa 的静态类型为类 A。

② 然后在类 A 中找到名称 f，检查 f 是否是虚函数。发现 f 是虚函数，则根据 pa 的动态类型 B，在类 B 中查找虚函数 f 的重定义版本。类 B 中的 f 不是对虚函数 f 的重定义，因此继续向父类 A 中查找，类 A 中的 f 是对虚函数的重定义，最后调用此函数。

5. 纯虚函数

（1）纯虚函数的声明形式为：virtual 类型 函数名(参数列表)=0;。

（2）纯虚函数后面的“=0”只是告诉编译器，此函数是纯虚函数。

（3）纯虚函数可以在引入该函数的类中不定义，这是与普通虚函数不同的地方。比如 class A{public: virtual void f()=0;};，其中 f 是纯虚函数，可以不定义。

（4）抽象类：如果类至少有一个纯虚函数，那么这个类就是抽象的。抽象类通常作为基类（父类），因此也被称为抽象基类。

（5）不能用抽象类创建对象。抽象类的作用一般是作为其他类的父类，即抽象类一般作为其他类的公共接口。比如 class A{public:virtual void f()=0;};，则 A ma;错误，不能创建抽象类的对象。

（6）可以用抽象类声明一个指针或引用，这个指针或引用可以指向该类的子类对象，这就使得抽象类能够支持多态性。

（7）子类把纯虚函数重定义为虚函数后，该函数不再是纯虚函数，而只保留了虚函数的性质，其后继子类也不再继承纯虚函数的性质。比如 class A{public: virtual void f()=0;}; class B:public A{public:void f(){}};class C:public B{public:}; B mb; C mc;，由对象 mb 和 mc 的创建，可知类 C 和类 B 不再是抽象类，因此在类 B 中对纯虚函数 f 进行重定义后，f 变为了虚函数，不再是纯虚函数。

（8）也可以在引入纯虚函数的类中对它进行定义，被定义后的纯虚函数就是完整的函数，此时该函数也可被调用（注意：只能被静态调用），但是仍然不能创建抽象类的对象。因此，要在抽象类外部调用被定义的纯虚函数，只能借助抽象类的子类来间接使用类名加作用域解析运算符的方式进行静态调用。比如 class A{public:virtual void f()=0{}}; class B:public A{public:void f(){} };B mb; A *pa=&mb;，则 mb.A::f();和 pa->A::f();都会调用父类 A 中定义的纯虚函数。注意：子类 B 必须对纯虚函数重定义为虚函数，否则 B 是抽象类，无法创建类 B 的对象。

（9）纯虚函数也可以在类外进行定义，但不能再指定 virtual 关键字和小括后面的“=0”。比如 class A{public:virtual void f()=0;};，则 void A::f(){};正确，而 virtual void A::f(){}错误，void A::f()=0(){}错误。

（10）若子类没有对父类的所有纯虚函数进行重定义，那么未被重定义的纯虚函数会被继承，因此该子类也成为抽象类。注意：这并不表示子类必须重定义父类的纯虚函数，只是不重定义纯虚函数，子类会成为抽象类，从而无法创建对象。比如 class A{public:virtual void f()=0;}; class B:public A{};，则 B mb;错误，因为 B 是抽象类。

6．虚析构函数

（1）为什么需要虚析构函数：假设类 A 为类 B 的父类，那么对于 A *pa=new B();，若类 A 的析构函数不是虚拟的，则 delete pa; 将根据 pa 的静态类型（即类型 A）来调用父类 A 的析构函数，这样就只释放了父类部分的内存，而 new B()很明显是使用子类 B 所分配的空间的，因此子类部分的内存并未被释放。若父类的析构函数是虚拟的，则通过指针调用时，将根据指针指向对象的类型来决定（即动态绑定）调用哪个析构函数。因此 delete pa;将调用 pa 所指向的类型（即子类型 B）的析构函数，因为子类的析构函数释放子类部分后，会调用父类的析构函数释放父类部分，这样就能正确地释放由 new B()所分配的内存了。示例如下。

delete 释放非虚析构函数的情形：

```
class A{public: ~A(){cout<<"~A";}}; class B:public A{public: ~B(){cout<<"~B";};
    A* p=new B(); delete p; //只会输出~A，可见子类部分的内存未被释放
```

delete 释放非虚析构函数的情形：

```
class A{public: virtual ~A(){cout<<"~A";}}; class B:public A{public: ~B(){cout<<"~B";}};
    A* p=new B(); delete p;//输出~B~A，可见子类部分的内存被释放了
```

（2）与虚函数相同，虚析构函数也会被继承。若父类中的析构函数是虚函数，则子类中的析构函数自动成为虚函数，而不管子类是否显式定义了析构函数，也不管子类的析构函数名是否与父类的相同。比如 class A{public: virtual ~A(){cout<<"~A";}}; class B:public A{public: ~B(){cout<<"~B";}}; B mb; A *p=&mb;，则 p->~A();将输出~B~A。因为析构函数是虚拟的，所以调用子类 B 的析构函数输出~B；又因为默认析构函数会调用父类 A 的析构函数销毁对象，因此会再次输出~A。

（3）即使父类不需要析构函数，也应显式定义一个空的虚析构函数，以保证在使用 delete 释放对象时能被正确地执行。

（4）虚析构函数也可被声明为纯虚函数，此时该类成为抽象类，不能创建该类的对象；但纯虚析构函数必须定义，只声明而不定义一个纯虚析构函数在链接阶段可能会发生错误（比如使用该类的子类创建对象时），因为当释放子类对象时，必须调用父类的析构函数，但该析构函数又未定义，所以出错。其实对任何函数只声明而不定义都是错误的。比如 class A{public:virtual ~A()=0; };class B:public A{};，则 B mb;是错误的，因为当 mb 的生命期结束时会调用父类 A 的析构函数，但父类中的纯虚析构函数未定义。

第 14 章

对象模型与虚函数表专题

（1）本章主要介绍编译器实现虚函数时隐藏的内部代码及其原理模型。

（2）本章应注意区分类和类对象，如 A ma, ma1，其中 A 是类，ma 和 ma1 是类对象。

（3）在大多数情况下，实体（subobject）与“对象”是相同的，差别只在于实体所指的对象并不是明确创建的，即实体是隐藏创建的对象。

（4）vtbl 指虚函数表，vptr 指虚函数表指针。

（5）在本章中，父类子对象、父类实体、父类部分是指相同的概念，只是为讲解方便用了不同的称呼。

14.1 对象模型与虚函数表基础、内存对齐、函数内部转换

14.1.1 对象模型简介

1. 基础概念

（1）对象模型：研究对象在计算机中怎样被表示出来。对象模型就是指在实现 C++的语法规则时，由编译器所产生出来的一种理论上的模型，编译器都有其内部代码来实现该模型，并且内部代码一般都是使用 C++语言编写的。

（2）对象模型有三种：简单对象模型、表格驱动模型和 C++对象模型。

（3）槽（slot）：其实就是一块内存区域。槽所占内存空间的大小，根据槽所存数据的大小而定。

2. 简单对象模型（见图 14.1）

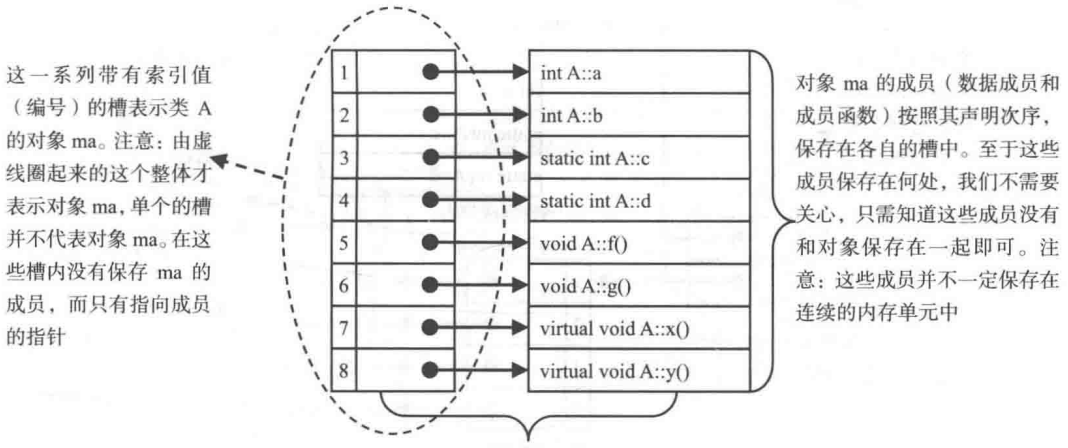


图 14.1　简单对象模型示意图

（1）在简单对象模型中，一个对象被表示为一系列的槽，每一个槽都使用一个“指针”指向对象的一个成员，每一个成员（包括数据成员和函数成员）又按照其声明次序被指定一个槽。

（2）在简单对象模型中，成员并没有被放置在对象中，对象中放置的是指向成员的“指针”，这样做的好处是对象的大小是固定的，不会因为成员的不同类型需要不同大小的存储空间而发生变化。因此此模型对象的大小很容易计算，即“指针所占存储空间的大小乘以类中所声明的成员数”。

（3）在简单对象模型中，对象的成员是以槽的索引值（其实就是一个编号）进行寻址的。此模型并没有被编译器厂家实际使用，但槽和索引值被广泛使用。

3. 表格驱动模型（见图 14.2）

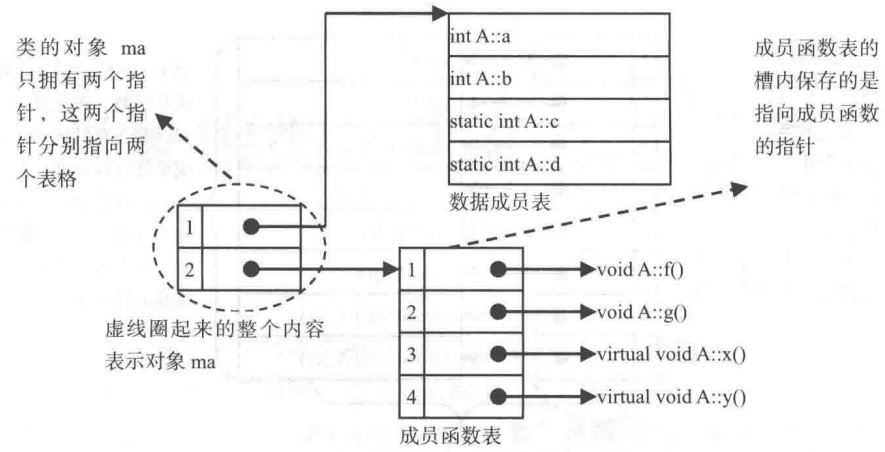


图 14.2　表格驱动模型示意图

槽和索引值的概念在表格驱动模型中依然被使用。

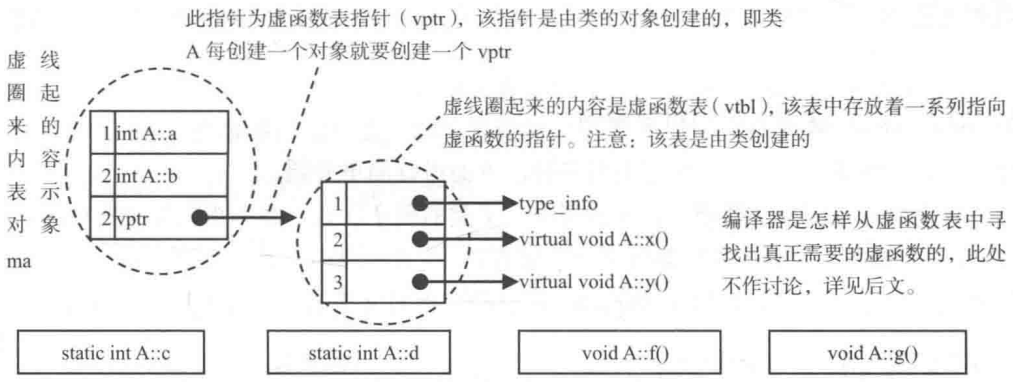
（1）表格驱动模型把数据成员和成员函数的信息分离出来，并分别保存在数据成员表和成员函数表中，在类的对象中则只有指向这两个表的指针。数据成员表同样是一系列的槽，每一个槽保存一个数据成员；成员函数表也是一系列的槽，但每一个槽只保存指向函数的指针。

（2）表格驱动模型可以保证每个类的对象有相同的大小，即“两个指针所占存储空间的大小”。

（3）其实表格驱动模型就是对类中的成员进行了一次更细致的分类。

（4）表格驱动模型也没有被用在 C++编译器上，但成员函数表的概念是支持虚函数的一个有力方案。

4. C++对象模型（虚函数表模型，见图 14.3）



说明：

① 静态数据成员属于整个类，因此其不会被存储在类对象中。

② 因为每个类对象的成员函数的拷贝只有一份，所以成员函数不会被存储在类对象中。

③ 至于静态数据成员和成员函数被存储在何处，我们不需要关心，只需知道它们没有存储在对象中即可。

④ 虚函数是怎样被寻址出来的此处不作讨论。

图 14.3　C++对象模型示意图

槽和索引值的概念在 C++对象模型中依然被使用。

（1）C++对象模型把非静态数据成员存放在每一个类对象中，静态数据成员、静态和非静态成员函数则被存放在所有的类对象外。每个类对象都会拥有自己的数据成员拷贝，但其成员函数的拷贝只有一份，所以成员函数不会被存储在类对象中。C++对象模型示意图大多数编译器厂家都正在使用。

（2）虚函数通过以下方式实现。

① 由类产生一个虚函数表（vtbl），表中存放着一系列指向虚函数的指针。

② 每个类对象都被添加一个指向虚函数表的指针，此指针被称为虚函数表指针（vptr）。虚函数表指针的设置由类的构造函数、析构函数和复制赋值运算符在编译器内部自动完成。

③ 虚函数表是由类产生的，这意味着虚函数表不会因为类的对象而发生变化。虚函数表指针是由类的对象产生的，即由类创建的每一个对象都有一个虚函数表指针。

④ 每个类所关联的类型信息 type_info（主要用于支持 RTTI，即运行期类型识别）也通常被虚函数表中的指针指出，该信息常常被放在虚函数表的第一个槽处。

5. 图解简单对象模型、表格驱动模型和 C++对象模型

假设类 A 有如下声明：

```
class A{public: int a;  int b;          //普通数据成员
static int c; static int d;             //静态数据成员
```

```
void f();void g();                          //普通成员函数
virtual void x();virtual void y();};       //虚拟成员函数
```

由类 A 创建一个对象 A ma，则三种对象模型的内存布局分别如图 14.1、图 14.2 和图 14.3 所示。

6. 加上（虚）继承之后的对象模型

加上（虚）继承之后的对象模型也有多种，下面进行简单介绍。

（1）以简单对象模型为基础，在该模型中，子类的槽中含有一系列指向父类实体的指针。

（2）父类表模型，为所有的父类创建一个父类表，该表的槽中含有一系列指向父类的指针。每个类对象都含有一个父类表指针（bptr），这个指针指向父类表。父类表模型如图 14.4 所示。

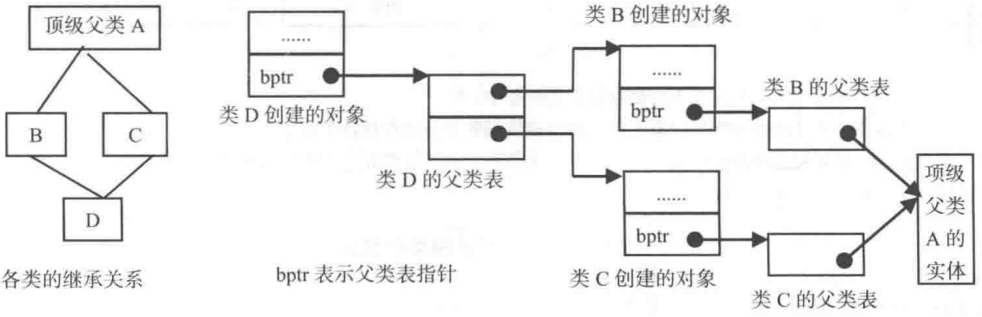


图 14.4 父类表模型示意图

14.1.2 类成员的存储次序与内存对齐

1. 数据成员的存储次序

（1）C++把非静态数据成员放置于类对象中，而成员函数和静态数据成员并未放置于类对象中，因此成员函数和静态数据成员并不会影响类对象的大小。

（2）一个空类占据 1 字节，并不是 0，这个字节是由编译器安插进去的，其作用主要是让类所创建的不同对象可以得到不同的独一无二的内存地址。比如 class A{};，类 A 的大小为 1 字节，若类 A 的大小为 0，则由类 A 所创建的对象如 A ma, ma1;则不能保证 ma 和 ma1 拥有独一无二的不同的内存地址。

（3）C++标准规定，同一个访问区域（即通过 public、private、protected 划分的区域）中，成员的次序需满足较后声明的成员有较高的地址。此规则应注意以下问题：

① 成员的排列不一定非得是连续的。这意味着由编译器产生的内部代码可能会放在类成员之前、之后或之间。典型示例就是由编译器为每一个类对象产生的虚函数表指针（vptr）既可放在类成员的前面、后面，也可放在两个成员之间。传统做法是把 vptr 放在所有成员的最后，

当然也有编译器把 vptr 放所有成员的最前面（比如 VC++编译器）。

② 较后声明的成员有较高的地址，只在同一个访问区域中才需要满足这个要求，在不同的访问区域中就不需要遵守这条规则。比如 class A{public: int a; int b; private: int c;};，其中 b 的地址应高于 a 的地址，但 c 的地址就不必高于 a 或者 b 的地址。也就是说，c 既可存放在 a 之前，也可存放在 b 之后。但各编译器厂家一般都是按照成员的声明次序进行排列的。

2. 内存对齐（边界调整）及类的大小

（1）内存对齐（边界调整）：就是数据存储在内存中的起始地址，必须是某个数 N（如 4、8 等）的倍数，其中 N 被称为“对齐系数”。这种对齐方式也被称为“对齐在 N 上”或“N 字节对齐”。

（2）内存对齐时起始地址必须满足“起始地址%N=0”，其中%是求模运算符，N 表示需要对齐的字节数。为了有更好的性能和可移植，C++需要进行内存对齐。

（3）类内部数据成员的内存对齐：其对齐结果是各数据成员存储的内存起始地址相对于类的起始地址的偏移量 x 必须为对齐系数 N 的整数倍。假设类的起始地址为 101，对齐系数为 4，则类中数据成员存储的起始地址只能是 101、105、109、113 等。

（4）整个类的内存对齐：其对齐结果是类的起始地址是对齐系数 N 的倍数，或类的起始地址%N=0。比如整个类被要求内存对齐，假设对齐系数为 4，则类的起始地址只能是 0、4、8、12、16 等。

（5）注意：类中数据成员的对齐是相对于类的起始地址而言的。

（6）对齐系数有三种：指定对齐系数 n、自身对齐系数 m、有效对齐系数 N。

① 指定对齐系数 n：编译器可通过#pragma pack(n);进行设置，其中 n 就是指定对齐系数，n=1,2,4,8,16……（只能是 2 的幂次方）。n 在各编译器上都有一个默认值，可通过#pragma pack(show)进行查看（以警告信息的形式显示），VC++ 2010 默认值为 8。

② 自身对齐系数 m：就是指数据自身类型大小所占据的长度。比如 int 类型大小为 4 字节，则自身对齐系数就是 4；char 类型大小为 1，则自身对齐系数就是 1。

③ 有效对齐系数 N：指的是在对数据进行对齐时正在使用的对齐系数。有效对齐系数是指自身对齐系数和指定对齐系数中较小的那个值。比如指定对齐系数 n=8，自身对齐系数 m=4，则有效对齐系数 N=min(8,4)=4。

（7）类中数据成员的内存对齐规则。

① 第一个数据成员存放在相对于类的起始地址偏移量为 0 的位置，即它的地址与整个类的地址相同。

② 其后的每个类数据成员分别按照各自的有效对齐系数 N 进行对齐，若有必要则在尾部填充空字符。注意：每个数据成员是独立按照自己的方式对齐的，这意味着对内存填充的空字

节不一定在尾部。对齐后的结果就是各数据成员存放的内存起始地址相对于类的起始地址偏移了有效对齐系数 N 的整数倍。

③ 类的整体对齐和大小：类的整体有效对齐系数为数据成员所用过的所有有效对齐系数中最大的那个值，其对齐后的地址必须为类的整体有效对齐系数的整数倍，类的整体大小也应为类的整体有效对齐系数的整数倍，若有必要则在尾部填充空字符。此规则适用于数组的对齐。

④ 若指定对齐系数 *n* 的值等于或大于所有数据成员长度（即数据成员的自身对齐系数 *m*）时，*n* 的大小不会对对齐产生任何影响。

⑤ 数组成员是按拆分后的成员对齐的。比如 char c[4];的对齐方式与分别声明 4 个 char 是一样的，但数组的各个元素是连续存储的。比如 class A{char c[4];};与 class A{char c1; char c2; char c3; char c4;};是相同的。

（8）内存对齐会影响类的大小，而且数据成员的声明次序不同，也会影响类的大小。

示例 14.1：内存对齐

```
#pragma pack(show)   //以警告消息的形式显示指定对齐系数 n 的默认值，VC++ 2010 默认值为 8
class A{public: char a;double b;short c; int d;};
void main(){A ma;
   cout<<sizeof A<<endl;       //输出 24
   cout<<(int *)&ma.a<<endl;   //输出 0012ff48，第一个数据成员的起始地址或类的起始地址
   cout<<&ma.b<<endl;          //输出 0012ff50，输出的地址值根据机器而定
   cout<<&ma.c<<endl;          //输出 0012ff58
   cout<<&ma.d<<endl;}      //输出 0012ff5c
```

分析：

① 本例的指定对齐系数 *n*=8（默认值）。

② 假设整个类存储的首地址使用内存对齐（根据编译器而定）。为方便讲解，此处假设首地址为 0x0012ff48。

③ ma.a 的地址是类的起始地址，因此 a 相对于类 A 的偏移量是 0。对 ma.a 的对齐要求与对整个类相同，ma.a 的地址为 0x0012ff48。

④ ma.b 是 double 类型，占 8 字节的内存，因此 ma.b 的自身对齐系数是 8；而指定对齐系数也是 8，因此有效对齐系数 *N*=min(8,8)=8。ma.b 的起始地址相对于类的起始地址应该是 8 的整数倍，因此从 ma.a 向后偏移 8 字节，找到地址&ma.b=0x0012ff48+8=0x0012ff50。因为 ma.a 占 1 字节，因此需要在 ma.a 的地址 0x0012ff48 和 ma.b 的起始地址 0x0012ff50 之间填充 7 字节，填充后 ma.a 占据 8 字节。详见图 14.5。

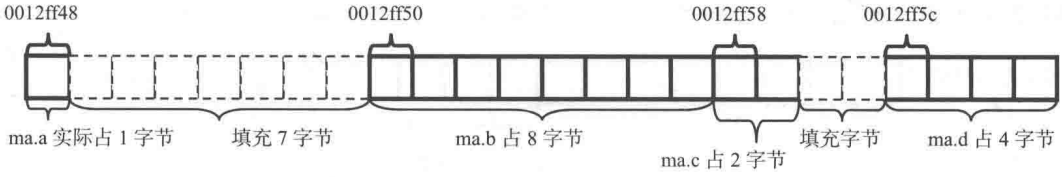


图 14.5　指定对齐系数为 8 时的内存布局（总大小为 24 字节）

⑤ ma.c 的自身对齐系数是 2，因此有效对齐系数 N=min(8,2)=2。ma.c 的起始地址相对于类的起始地址应该是 2 的整数倍，因为 ma.b 已经存储在 0x0012ff50~0x0012ff57 的位置，因此从下一字节 0x0012ff58 开始检测存储 ma.c 的地址。0x0012ff58 相对于类的起始地址 0x0012ff48 的偏移量为 0x10=16 是 2 的整数倍，因此 ma.c 从 0x0012ff58 开始存放。此时 ma.b 占据 8 字节（无填充字符）。

⑥ ma.d 的自身对齐系数为 4，因此 N=min(8,4)=4。ma.d 的起始地址相对于类的起始地址应该是 4 的整数倍，因为 ma.c 已经存储在 0x0012ff58~0x0012ff59 的位置，因此从下一字节 0x0012ff5a 开始检测存储 ma.d 的地址。检测到第一个相对于类的起始地址的偏移量为 4 的整数倍的地址是 0x0012ff5c，其偏移量为 0x0012ff5c–0x12ff48=0x14=20，因此 ma.d 从 0x0012ff5c 开始存储。该地址相对于 ma.c 的首地址偏移了 4 字节，而 ma.c 占据 2 字节，因此在 ma.c 和 ma.d 之间需填充 2 字节，填充后 ma.c 占据 4 字节，此时 ma.d 占据 4 字节。至此，类 A 中的 4 个数据成员 a、b、c、d 共占据 8+8+4+4=24 字节

⑦ 整个类的大小：整个类的有效对齐系数为数据成员所用过的所有有效对齐系数中最大的那个值（注意 ma.a 的有效对齐系数应在内），即 N=max(char,double,short,int)=max(1,8,2,4)=8，因此整个类的大小必须为 8 的倍数。把以上所有成员所占空间相加，ma.a+ma.b+ma.c+ma.d=8+8+4+4=24，正好是 8 的倍数，因此整个类的大小为 24 字节。若要求对整个类进行内存对齐，则整个类的起始地址也应该是 8 的整数倍。本例类的起始地址是 0x0012ff48，是 8 的整数倍，因此整个类是内存对齐的。

⑧ 若使用#pragma pack(4)把指定对齐系数设置为 4，则整个类的大小为 ma.a+填充+ma.b+ma.c+填充+ma.d=1+3+8+2+2+4=20。请按以上规则进行分析，详见图 14.6。

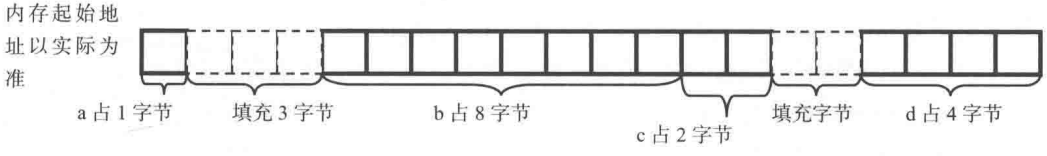


图 14.6　指定对齐系数为 4 时的内存布局（总大小为 20 字节）

⑨ 若将 double b 声明于 char a 之前，则整个类的大小为 ma.b+ma.a+ma.c+ma.d+填充=8+1+2+4+1=16，最后 1 字节为对整个类对齐时所要求的填充字节。详见图 14.7。

图 14.7 把 double 放在 char 之前的情形（总大小为 16 字节）

14.1.3 编译器对函数的内部转换与名称改编

名称改编（name mangling）的本质就是在编译器内部对名称进行“重命名”。名称改编也被称为名称粉碎。

1. 编译器对成员函数的内部转换规则与名称改编

（1）编译器为什么需要名称改编：C++在重载函数时，函数可以有相同的名称，编译器为了区分各个不同的函数，需要有一个内定的算法把重载函数的名称重命名为唯一的名称。这就是 C++的“名称改编”机制，这种机制是由编译器内部实现的，各编译器厂家会有不同的命名规则。

（2）在继承时数据成员也应经过重命名处理。数据成员的重命名比较简单，只需在重命名时加上数据成员来自于哪一个类即可。假设子类同时继承多个父类中相同名称的数据成员，这时在编译器内部就应对被继承的数据成员重新命名。比如 class A{int a;}; class B{int a;}; class C:publicA,public B{}，这时在编译器内部就应对类 C 继承的两个相同数据成员 a 的名称进行重命名以区别它们。

（3）对函数重命名各编译器有不同的命名方法，在 VC++ 2010 中查看被编译器重命名后的函数名称的方法是声明而不定义函数，然后调用该函数，就会在错误提示中出现重命名后的函数名称，类似于"void __cdecl f(void)" (?f@@YAXXZ)。注意：(?f@@YAXXZ)才是微软对函数重命名后的名称。有关微软的重命名机制请参阅相关文章。

（4）编译器对成员函数的内部转换：对于非静态成员函数，C++的设计准则是，非静态成员函数必须至少与一般的非成员函数有相同的效率。因此在编译器内部会将成员函数转换为对等的非成员函数，以实现效率上的一致。其转换步骤大致如下。

① 改写成员函数的形参表，并安插进一些必要的额外形参，其中有一个额外形参的作用是作为“使用对象对该函数进行调用时”的中间桥梁，该形参就是所谓的 this 指针，然后以 this 指针调用类的非静态数据成员。比如 class A{public: void f(){...}};，则函数 f 的形参会被内部转换为 void f(A*const this){...}，其中 this 就是安插进来的额外形参，作为函数调用时的中间桥梁。

其他的额外形参根据成员函数的不同而不同，请见后文。

② 把成员函数重写为外部函数，并对函数名称进行重命名，以使函数名称在编译器内部是独一无二的。

③ 对函数的调用操作进行相应的转换，其方法就是把调用该函数的对象或指针作为实参传递给被内部转换后的函数的形参 this。比如 class A{public: void f(){…}};，则函数 f 的形参会被改写为 void f(A*const this)的形式，对该函数调用时 A ma; A* p=&ma; ma.f();或 p->f();会被转换为 f(&ma)或 f(p)，以与被转换后的函数 f 的 void f(A* const this) 形式相兼容。

④ 对于虚函数的调用操作会被转换为数组的形式。假设 mb 是类 A 的子类对象，f 是一个虚函数，则 A *p=&mb; 调用 p->f(); 会被转换为 (*p->vptr[1])(p);的形式，其中 vptr 表示由编译器产生的位于类对象中的虚函数表指针，这个指针指向虚函数表。当然，在各编译器内部，虚函数表指针的名称并不一定是 vptr。数值“1”是虚函数 f 在虚函数表中所对应的槽的索引值（此处假设 f 位于槽 1）。第二个 p 表示把 p 作为实参传递给被内部转换后的函数 f 的 this 形参。对于使用类对象对虚函数进行调用时，编译器总是把此调用按非静态成员函数那样进行转换，即转换为 f(p)的形式，没必要把这种调用转换为 (*p->vptr[1])(p);的形式，因为使用类对象来调用虚函数并不能实现多态性。

示例 14.2：编译器对成员函数的内部转换

原始代码：

```
class A{public: void f(){x+y; }int x, y;};
```

假设编译器内部对函数 f 的转换代码如下（注意，这是内部代码）：

```
extern void f_8AFA(A* const this, ) //其中 extern 关键字可省略，this 是新增加的额外形参
                                    //用于作为函数调用时的中间桥梁
{ this->x+this->y;}     //被转换后的函数使用 this 指针调用类的非静态数据成员
//以上假设函数名 f 被重命名为 f_8AFA，其中 8 表示函数 f 形参的大小，第一个 A 表示 f 是类 A 的成员函数
//F 表示 f 是一个函数，第二个 A 表示函数返回类型为 A。注意：编译器并不一定是这样命名的
```

2．函数返回类对象时编译器的内部转换规则与 NRV（命名返回值）优化

（1）命名返回值（named return value），即形如 return x;中的 x 就是被命名后的返回值。

（2） 当把一个类对象作为参数传递或作为函数的返回值时，编译器会对其进行一些优化处理。对于形参的优化因各编译器的方法不一样，故本节不作介绍。本节重点讲函数返回命名类对象时的优化处理，即形如 return ma1 的情形。

（3）编译器对函数返回类对象时的内部处理一般分为两个步骤（未做任何优化时）。

① 新增加一个额外的该类类型对象作为形参，该形参用来存放返回值。其实这个形参就是所产生的临时对象。

② 在 return 语句之前调用复制构造函数，把本应传回的返回值复制到新增加的额外形参中，这时该形参中就存储有返回值了。

示例 14.3：编译器对函数返回类对象时的内部转换（未优化）

原始代码：

```
A f(){A aa; return aa;}
```

编译器的内部代码如下：

```
void f(A& _r)      //新增加的额外形参_r，用于放置返回值
{A aa;   //原程序中的代码
          aa.A::A();     //这是编译器的内部代码，用于创建对象 aa
               //...程序中对 aa 的处理，本例中 aa 未做任何动作
_r.A::A(aa);   //编译器内部代码，调用复制构造函数把处理后的 aa 复制到新增加的额外
               //形参_r 中，最后_r 中就存储有返回值了
return; }
```

（4）编译器对返回类对象的函数做了内部转换之后，还应对“对该函数的调用操作”进行内部处理，否则无法调用被处理后的返回类对象的函数。比如 A f();被内部处理为 void f(A& _r);后，若 A ma=f();未对函数 f 的调用做内部处理，则在编译器内部无法调用被处理成 void f(A& _r)的函数（因为形参不匹配）。

（5）编译器调用返回类对象的函数时的内部处理。假如 A f();被内部处理为 void f(A& _r);，则编译器内部处理该函数调用的操作分以下三种情形。注意：函数 f 被转换后的内部形式为 void f(A&)，也就是对函数 f 的调用都应以 void f(A&)的形式为基础。

情形一：将调用函数后的返回值赋给一个已存在的对象。其方法是把被赋值的已存在的对象作为函数的实参进行传递。比如 A ma=f(); 被编译器内部转换为 A ma; f(ma)。

情形二：直接调用该函数或使用该函数的返回值调用类的成员。其方法是创建临时对象，并以临时对象为实参调用被转换后的函数。比如 f().g();被内部转换为 A tmp; (f(tmp), tmp). g();，其中 tmp 是所创建的临时对象；f();会被内部转换为 A tmp; f(tmp);。

情形三：函数指针的形式。其方法是把函数指针转换为与被内部处理后的函数相同的形式。比如 A (*p)(); p=f;被转换为 void (*p)(A&);p=f;，即把函数指针 p 转换为与被内部处理后的函数 f 相同的形式，即 void f(A& _r);。

（6）函数返回类对象时编译器的 NRV 优化方案。

① 该方案是用新增加的形参直接取代函数的返回值，这样就省去了在函数内部使用复制构造函数把返回值复制到额外形参的步骤，最后没有调用复制构造函数。

② 编译器一般把程序中是否有自定义的复制构造函数作为 NRV 优化的开关，若程序自定义了复制构造函数，则编译器开启 NRV 优化（此时返回值不会调用复制构造函数）；若程序未定义复制构造函数，则 NRV 优化关闭。至于是否支持 NRV 优化，或者是否把复制构造函数作为 NRV 的开关，根据编译器而定，C++未作强制规定。

示例 14.4：NRV（命名返回值）优化

原始代码：

```
A f(){A aa; return aa;}
```

NRV 优化后编译器的内部代码如下：

```
void f(A& _r)      //新增加的额外形参_r，用于放置返回值
  {A aa;
  _r.A::A();    //此处对_r直接进行初始化，然后以_r代替返回值aa在程序中的行为
  //...程序中使用_r代替aa，并对_r进行处理，然后把返回结果存于_r中。此过程省略了复制构造函数的调用
  return; }
```

14.1.4　指向虚成员函数的指针

注意：在 VC++中，对成员函数无论怎样取址，其值都始终为 1。

（1）获取一个非虚成员函数的地址，将得到该函数在内存中的实际地址，这与获取数据成员时的地址不同。但要使用该成员函数，还是要必须绑定在某个类对象的地址上才行。比如 void (A::*p)()=&A::f; A ma;，则必须这样调用 f 函数(ma.*p)();，直接使用指针 p 进行调用是错误的，如(*p)()错误。

（2）对一个虚成员函数取其地址，只能得到虚函数位于虚函数表中相对应的索引值。因为虚函数的地址在编译时期是未知的，只知道虚函数所在虚函数表中的索引位置。比如 class A{public:virtual void f(){};virtual void g(){}};，则&A::f;将得到值 0，而&A::g 将得到值 1。

（3）对指向成员函数的指针进行调用，将会因各编译器的实现策略不同而有所不同。

① 指向成员函数的指针 void (A::*pf)();因为指向虚函数和非虚函数成员而有两种不同的值，因此对其调用也与常规函数有所不同。比如 A *pa;，则调用 (pa->*pf)();将因为 pf 指向虚函数还是非虚函数而有不同的值。若指向非虚函数，则 pf 代表一个地址，此调用会被编译器内部转换为 pf(pa)的形式，以把 pa 作为 this 指针的实参进行传递；若指向虚函数，则 pf 代表一个索引值，此调用会被编译器内部转换为(*pa->vptr[(int)pf])(pa);。因此在编译器内部需要对这两种情况加以区分。

② 编译器内部区别指向成员函数的指针是索引值还是内存地址的方法：设计一个中间结构体来处理是索引值还是地址的问题，编译器可根据不同的继承情况而选择不同的成员，从而提供多种形式的指向成员函数的指针。其结构体的形式如下：

```
struct _mptr{int delta;                    //多重继承时this指针的偏移值
   int index;      //当指向虚函数时，设置为虚函数所在表中的索引值，否则设置为-1
   union{ptrtofunc faddr; //当指向非虚函数时设置此值，此值应是指向成员函数的地址
   int v_offset; }; }; //表示虚继承时第二个或以后的虚父类的vptr的位置值
```

因此调用(pa->*pf)()会被转换为(pf.index<0)?(*pf.faddr)(pa):(*pa->vptr[pf.index])(pa);，即如果 index 的值小于 0，则表示该函数是一个非虚函数，使用(*pf.faddr)(pa)的形式调用该函数；否则是一个虚函数，使用(*pa->vptr[pf.index])(pa);的形式进行调用。

14.1.5　对成员函数的各种转换总结

（1）class A{void f();};ma.f(); p->f();被内部转换为 void f(A* const this){}和 f(&ma); f(p);。

（2）class A{A f();};p->f();ma.f();被转换为 void f(A*const this, A& _r);和 A tp; f(&ma, tp); f(p,tp);

（3）class A{virtual void f();}; ma.f(); pa->f();被转换为 f(&ma); (*p->vptr[offset])(p);，其中 offset 为虚函数在虚函数表中的槽的索引值。

14.2 各种 C++对象模型

14.2.1 指针与类型的关系

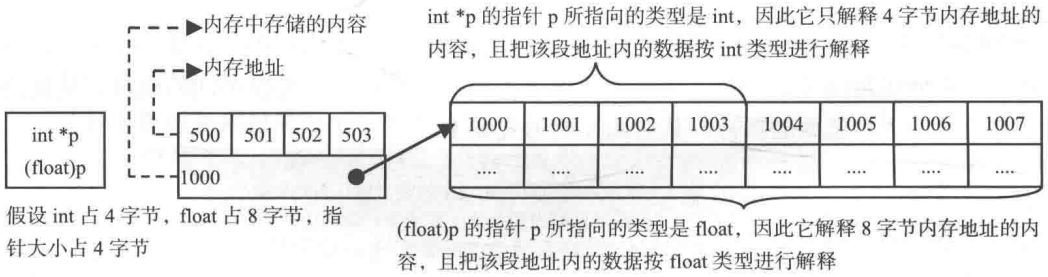
（1）一块连续内存的地址是没有实际意义的，它的意义完全取决于类型，类型决定了怎样解释和处理这块内存。

（2）不管什么类型的指针，它们对内存的需求都是相同的，也就是任何指针占据的内存空间大小是一样的。比如在 32 位机器上一般需要 4 字节或者说一个 word 的内存空间来存放指针，word 的大小根据机器（或正在使用的计算机）而定。

（3）各指针之间的差别在于指针所指向的类型不同，也就是通过指针寻址出来的对象的类型不同。因为类型不同，也就意味着编译器对寻址出来的对象的解释方式不同。

（4）对指针进行强制类型转换并不会改变指针变量的值，也就是不会改变指针所指向的地址。强制类型转换只影响对指针所指向的内存的解释方式，即根据不同的类型确定指针所指向的内存大小和内容。

指针与类型的关系如图 14.8 所示。



说明：① 不管指针是什么类型，其所占据的存储空间都是相同的。

② 不管指针的类型被转换为什么，指针变量的值是不会改变的，也就是不会改变指针所指向的地址。

③ int *p 和强制转换后的(float)p 都被存储在相同的内存地址处，其指针指向的都是以 1000 开始的内存单元。

④ 至于指针所指向对象的内容怎么解释就与指针的类型有关了，详见图中的说明。

图 14.8 指针与类型的关系

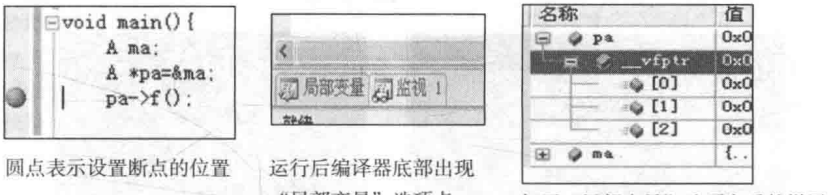
14.2.2　VC++ 2010 访问虚函数表的三种方法

1．使用调试时的局部变量窗口查看虚函数表

（1）此方法不是很可靠，因为局部变量窗口很有可能不会即时刷新新增加或修改后的虚函数，而且无法看到虚函数表的结构。因此不推荐此方法。

（2）在产生虚函数表的地方，设置断点（就是在该行左前面用鼠标点一下，会出现个小圆点），一般在创建一个含有虚函数的类对象处就行。若想查看虚函数表指针（vptr），则应在该指针的类型明确确定处设置断点才能看到，否则指针类型未确定，虚函数表自然就不确定了。比如 A *pa=&mb;，这时 pa 的类型不能确定，而 pa->f();时 pa 的类型就确定了（见图 14.9）。

（3）单击“运行”，窗口会停留，这时回到编译器，底部会多一个“局部变量”选项卡，打开该选项卡即会出现所需的虚函数表，如图 14.9 所示。



圆点表示设置断点的位置　　运行后编译器底部出现“局部变量”选项卡　　打开“局部变量”选项卡后的样子

图 14.9　VC++ 2010 使用局部变量窗口查看虚函数表

（4）若未出现“局部变量”选项卡，则可在暂停状态下选择“调试”→窗口→“局部变量窗口”。

2．使用 cl 命令查看虚函数表

（1）打开 cl 命令的方法。

① cl 命令位于开始菜单中：Microsoft Visual Studio 2010→Visual Studio Tools→Visual Studio 2010 命令提示（建议使用此方法以避免产生不正确的结果）。

② cl 命令可直接在 CMD 窗口中转到 C:\Program Files\Microsoft Visual Studio 10.0\VC 目录下运行。

③ cl 命令可在配置环境变量后，在 CMD 窗口中进入任何目录下运行。

（2）cl 命令用法：cl /d1 reportSingleClassLayoutXXX"文件路径"，其中 XXX 表示需要查看的类。

（3）在 CMD 窗口中进入任何目录下运行 cl 命令的方法（以 VC++ 2010 为例）。

① 设置环境变量如下（若没有，则增加环境变量）：

```
PATH=C:\Program Files\Microsoft Visual Studio 10.0\VC\bin
INCLUDE=C:\Program Files\Microsoft Visual Studio 10.0\VC\include
```

```
LIB=C:\Program Files\Microsoft Visual Studio 10.0\VC\lib
```

② 如果提示找不到 mspdb80.dll 文件，则从 C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE 下拷贝 msobj100.dll、mspdb100.dll、mspdbcore.dll 和 mspdbsrv.exe 四个文件到 C:\Program Files\Microsoft Visual Studio 10.0\VC\bin 下即可。

③ 设置环境变量的位置（以 WindowsXP 为例）：右击“我的电脑”，选择“属性”→“高级”→“环境变量”，然后在 “系统变量”中设置上述值。

（4）注意：若 main 函数内有输出成员地址的语句（如 cout<<&a<<end），那么 cl 命令可能无法显示完整的虚函数表结构。

示例如下：

```
cl /d1 reportSingleClassLayoutB "C:\www\fff.cpp";
```

表示显示 fff.cpp 文件中类 B 的虚函数表结构。

VC++ 2010 使用 cl 命令查看虚函数表结构，如图 14.10 所示（需使用滚动条查看）。

```
class B size(12):
        +---
        | +--- (base class A)
 0      | | (vfptr)
 4      | | a
        | +---
 8      | b
        +---
```

类 B 有一个父类子对象 A 和成员变量 b，子对象 A 有一个虚函数表指针，即 vfptr 和成员变量 a

```
B::$vftable@:
        | &B_meta
        |  0
 0      | &A::f
 1      | &A::g
 2      | &B::h
 3      | &B::x
```

类 B 有 4 个虚函数，分别是由父类继承 A 得到的 f 和 g，以及自己的 h 和 x

图 14.10 虚函数表结构

3. 使用指针直接访问虚函数表

本部分建议学完本章后续内容后再回头阅读。

注意： *在 VC++ 2010 中虚函数表指针位于对象的起始位置处。*

（1）如 A ma; 类对象的地址&ma 就是整个对象的首地址，若类含有虚函数，则此地址指向的就是虚函数表指针（vptr），但&ma 以类类型 A 的形式被解释。虽然&ma 的类型与 vptr 的类型不同，但我们还是知道了虚函数表指针的位置，若虚函数表指针位于类对象末尾，则只需对类对象的地址进行相应偏移即可。当然虚函数表指针的类型我们并不知道，但这无关紧要，我们也不需要知道它的类型。

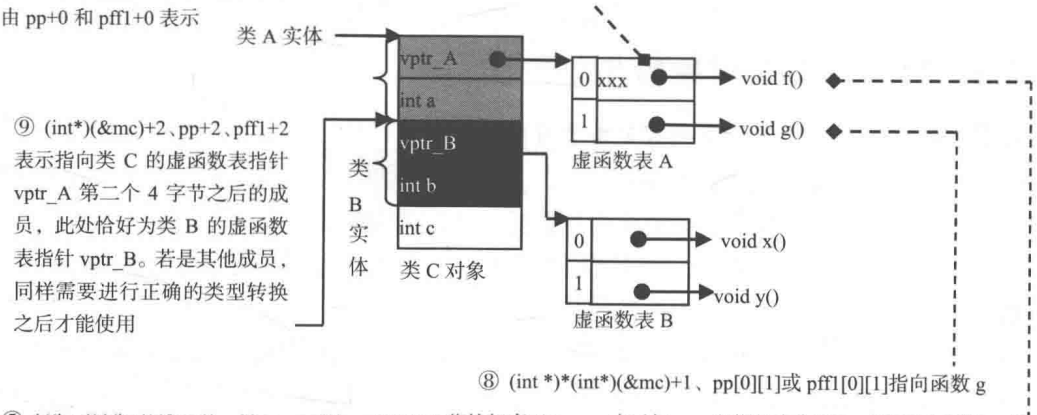
（2）因为指针的类型只影响怎样解释指针指向的内容，因此可以把类对象的首地址做一些强制转换，只要转换之后指针能指向虚函数，并能正确地寻址，就能得到虚函数表中第一个虚函数的地址。使用指针直接访问虚函数表中的虚函数有三种方法，详见以下示例。

① &mc 就是指向的 vptr，只是&mc 的类型是 C 类型的与 vptr 的类型不同而已。

② (int *)(&mc)表示把虚函数表指针（vptr）的类型转换为 int 类型，转换之后就相当于&mc 是 int*类型的指针了。对于二级和三级指针不需要此步强制转换，但此处的位置由 pp+0 和 pff1+0 表示

③ *(int*)(&mc)、*pp、pp[0]、pff1[0]表示虚函数表第一个槽内的内容，即 xxx，它是一个指针，因此*(int*)(&mc)使&mc 成为一个指向指针的指针，即二级指针，但此时的类型还不知道

④ (int *)*(int*)(&mc)表示把虚函数表第一个槽内的指针 xxx 类型转换为 int*，因此(int *)*(int*)(&mc)使&mc 成为一个 int 类型的二级指针，即与 int **pp 相同。对于二级指针 pp 和三级指针 pff1 不需要此步强制转换



⑤ *((int *)*(int*)(&mc))、**pp、pp[0]、pff1[0][0]此处相当于*xxx，表示把 xxx 所指的内容按 int 类型进行解析，但 xxx 所指的内容是一个函数，把函数当作 int 类型解析将得到错误的结果，因此应把*xxx 再转换为函数类型。注意：函数类型其实是一个函数指针

⑥ (pf)*((int *)*(int*)(&mc))、(pf)**pp、(pf)pp[0][0]、(pf)pff1[0][0]表示转换为与虚函数 f 相同的函数类型指针

⑦ ((pf)*((int *)*(int*)(&mc)))()就表示调用虚函数表中的第一个虚函数 f

注意：① 数组下标运算相当于偏移之后再解引用，如 a[3][4]=*(a[3]+4);。具体请参阅第 6 章。

② 多级指针的运算：多级指针加上或减去一个整数，会使指针向前或向后偏移“N*指针指向类型所占字节数”这么多字节。比如对于 double ***p;，因为 p 指向的是一个二级指针，因此 p+2 只会偏移两个指针类型的大小，即 8 字节。关于函数指针请参阅第 7 章。

图 14.11　使用指针直接访问虚函数表中的虚函数时指针的转换步骤

示例 14.5：使用指针直接访问虚函数表（使用 VC++ 2010 编译器）

```
class A{public:int a; virtual void f(){cout<<"AF";}virtual void g(){cout<<"AG";}};
class B{public:int b; virtual void x(){cout<<"BX";}virtual void y(){cout<<"BY";}};
class C:public A,public B{public:int c;};
void main(){C mc;        //创建 C 的类对象，以访问类 C 中的虚函数表
   typedef void(*pf)();  //引入函数指针，此函数指针的类型应与类中虚函数的类型相同
   //方法一：使用一级指针一步一步进行转换，具体见图 14.11
   ((pf)*((int *)*(int*)(&mc)))(); //输出 AF，调用第一个虚函数表 A 中的第一个虚函数 f
   ((pf)*((int *)*((int*)(&mc)+2)+1))(); //输出 BY，调用第二个虚函数表 B 中的第二个虚函数 y
   //方法二：使用二级指针。由图 14.11 也可清楚地看到 vptr 就是一个二级指针，即 vptr 指向的是一个指针
   //（该指针位于虚函数表内，指向真正的虚函数）
   int **pp=(int**)&mc;  //把代表 vptr 地址的&mc 强制转换为二级指针
```

```
((pf)pp[0][0])();    //输出AF，调用第一个虚函数表A中的第一个虚函数f
((pf)pp[2][1])();    //输出BY，调用第二个虚函数表B中的第二个虚函数y
//方法三：使用三级指针。因为函数指针不能进行算术或数组下标运算，因此应使用三级指针，其中前两级
//指针用于寻址出正确的虚函数地址，第三级指针用于指向函数
typedef void (***pff)();
(((pff)&mc)[0][1])();   //输出AG，调用第一个虚函数表A中的第二个虚函数g
(((pff)&mc)[2][1])();  //输出BY，调用第二个虚函数表B中的第二个虚函数y，(pff)&mc)[2]与
                       //*((pff)&mc)+2)相同，在32位机器上将使指针偏移两个4字节，因为
                       //(pff)&mc;指向的类型是一个“指针类型”
pff pff1=(pff)&mc; (pff1[0][1])();(pff1[2][1])();} //创建一个pff类型的中间变量更清晰
```

14.2.3 单继承下的对象模型

注意：继承时C++保证子类拥有父类子对象的完整原样性。

（1）在C++中，虚函数表和虚函数的地址是固定不变的，因此该地址在编译期就可获得，不需要等到运行期。运行期需要做的事情是怎样正确地调用到该虚函数（即激活虚函数）。

（2）虚函数表是由类创建的，一个类只有一个虚函数表，虚函数表中含有虚函数的地址，虚函数表中所指向的虚函数被以索引值的形式访问。

（3）每创建一个类对象时都会创建一个额外的虚函数表指针（vptr）以指向虚函数表。

（4）虚函数表指针的位置：不同编译器厂家对此有不同的策略，vptr 可放置于对象的起始位置，也可放置于对象的末尾。若 vptr 放置于对象的起始位置，则会丧失对 C 语言 struct 的兼容性。微软的 VC++编译器把虚函数表指针放置于对象的起始位置，而 cfront 编译器（由创建C++的组织开发）则放于对象的末尾。

（5）单继承下子类虚函数表包含以下虚函数。

① 从父类继承的虚函数，该虚函数在子类中未被重定义。此时子类继承父类的虚函数，准确地说是父类虚函数的地址会被复制到与子类的虚函数表对应的槽中。

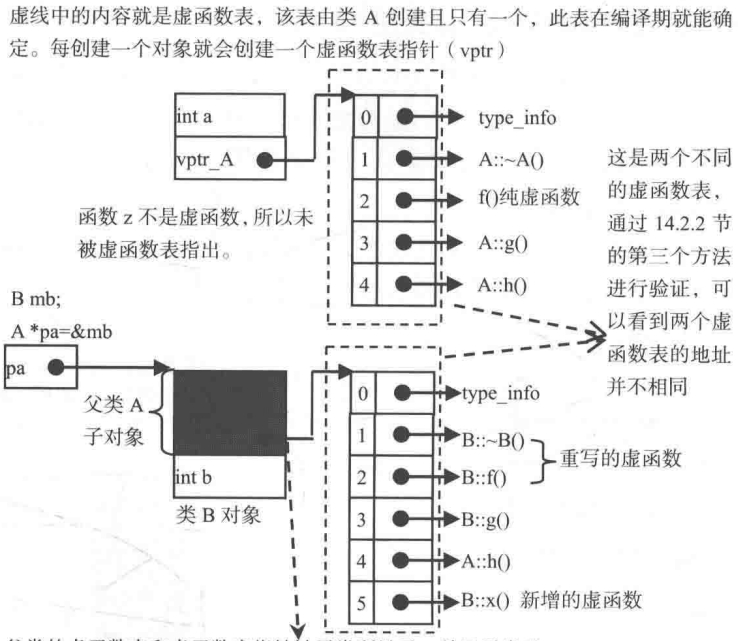
② 该类重定义的从父类继承的已存在的父类虚函数。

③ 纯虚函数。

④ 该类新创建的虚函数，此时需要扩充虚函数表以放置新增加的虚函数。

（6）大多数C++编译器都会继承父类的虚函数表指针（vptr），但 vptr 所指向的虚函数表地址并不相同。同时要注意，子类和父类并不共享同一个虚函数表，它们的虚函数表是相对独立的，虽然子类会继承父类中未被重定义的虚函数，但这种继承只是把虚函数的地址复制到子类的虚函数表相对应的槽中。比如父类 A 有虚函数 f，而子类 B 对虚函数 f 进行了重定义，假设 A 和 B 共享同一个虚函数表，若函数 f 存储在共享的虚函数表的槽 1 位置，那么子类重定义的函数 f 就不可能再存储在该位置，因为它们现在是两个不同的函数，同一个地址不可能存储两个不同的函数。因此父类和子类不可能共享同一个虚函数表。

单继承下的对象模型示意图如图 14.12 所示。



① 注意子类中虚函数的排列次序。

② 此处还应注意，通过 pa->x()直接调用是错误的，因为 x 不是父类 A 的成员。

图 14.12　单继承下的对象模型示意图

示例 14.6：单继承下的虚函数表

```
class A{public:int a;
    virtual ~A(){}
    virtual void f()=0;
    virtual void g(){}
    virtual void h(){}
    void z(){}
};
class B:public A{public:int b;
    ~B(){};
    void f(){};
    void g(){};
    virtual void x(){}
};
```

14.2.4 多重继承下的对象模型与 this 指针调整

1. 多重继承下的理想对象模型（详见图 14.13）

（1）在多重继承下，子类拥有从父类继承的虚函数表，当然子类会对该表的内容进行相应的修改，因此第一个父类之后的其他父类的虚函数表必须向后进行相应的偏移。

（2）子类新增的虚函数会与第一个父类的虚函数表合并（所谓第一个指的是声明时的顺序）。此规则被大多数编译器厂家采用。

（3）若子类重写（或覆盖）了父类的虚函数，不同的厂家会有不同的实现方法，当然理想模型就是直接使用子类重写后的虚函数代替各父类虚函数表中相同的虚函数。

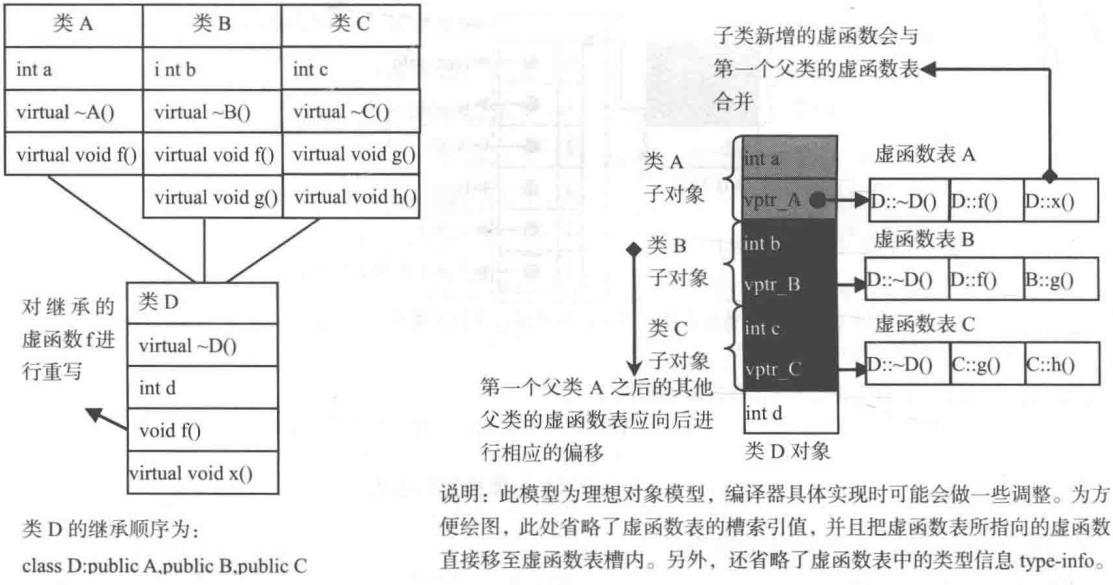


图 14.13 多重继承下有虚函数时的理想对象模型示意图

2. 为什么需要指针调整或地址偏移

（1）在多重继承下，当使用第二个及以后的父类指针指向子类对象时，需要对指针所指向的内存地址做调整。详见图 14.14。

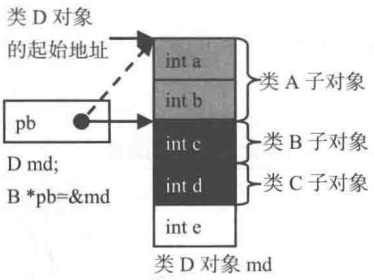


图 14.14　多继承下的指针偏示意图

① 对于 D md; B* pb=&md;，若未对指针进行调整，则此时指针 pb 指向类 D 所创建的对象 md 的起始位置（图中虚线箭头），这是不正确的位置，正确的应是 md 所包含的类 B 子对象的位置（图中实线箭头）。此时编译器就应对指针 pb 指向的地址做相应的调整，以使其指向正确的起始位置，即把 md 的地址向后/前偏移 sizeof(A)这么多字节。因此 pb 在编译器内部可能会被这样处理：B* pb=&md+sizeof(A);。

② 指针调整是在编译器内部自动完成的，我们不需要担心。但要注意 pb 和&md 的地址并不一致，它们相差 sizeof(A)这么多字节。

（2）如果子类含有虚函数而父类没有虚函数，当编译器（比如微软的 VC++编译器）把虚函数表指针安插于类对象最前面的位置时，在继承下也会发生指针调整。详见图 14.15。

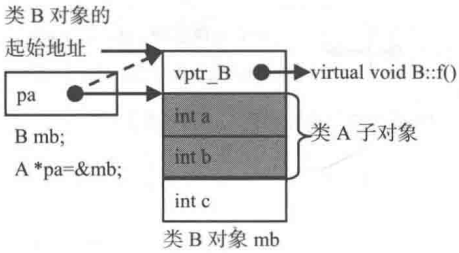


图 14.15 虚函数表指针（vptr）位于类对象最前面时的指针偏移示意图

① 若子类 B 含有虚函数 f 而父类 A 没有虚函数时，则对于 B mb; A* pa=&mb;，若未对指针进行调整，则此时指针 pa 指向类 B 所创建的对象 mb 的起始位置（图中虚线箭头），这是不正确的位置，正确的起始位置应不包含虚函数表指针，即为类 A 子对象的位置（图中实线箭头）。因此应对 pa 所指向的起始地址做相应的调整，以使其指向正确的起始位置，即把 mb 的地址向后/前偏移虚函数表指针所占内存的大小。所以 pb 在编译器内部可能会被这样处理：B* pb=&mb+offset，其中 offset 表示虚函数表指针所占内存的大小，该大小受类中成员的布局和内存对齐的影响。

② 同理，指针调整是在编译器内部自动完成的。但要注意 pa 和&mb 的地址并不一致，当输出它们的地址时 cout<<pa; cout<<&mb;得到的将不是相同的地址，而会相差一些字节。读者可按图中的结构写出程序，自行验证。

3．this 指针调整与 thunk 技术（调整并跳转技术）

（1）在编译器内部，函数都会被重命名，并且类的成员函数都会被转换为等价的外部函数，转换之后就需要增加额外的 this 指针作为形参。在进行调用时，也要对函数进行相应的转换，以给转换后的函数的 this 传递实参。比如 void A::f()函数，被转换为 void f(A*const this)，调用 ma.f()会被转换为 f(&ma)。若 f 是虚函数，则调用 p->f()会被转换为(*p->vptr[1])(p)的形式，其中第二个 p 作为转换后函数的 this 指针的实参。

（2）thunk 技术原理：现在大多数编译器厂家都使用 thunk 技术。使用该技术之后，凡是需要对指针进行调整的虚函数，都让虚函数表指向该虚函数的位置替换为指向一个所谓的“thunk”的地址。thunk 技术其实就是对 this 指针用适当的 offset 值（偏移值）进行调整（其目的是让其指向正确的起始地址），然后就直接跳去执行相关的虚函数。由此可见， thunk 技术就是虚函数表与其所指向的虚函数之间的一个中间转换平台。至于利用 thunk 技术怎样实现地址调整并跳转去执行相应的函数，那是编译器内部完成的，我们不需要关心。详见图 14.16。

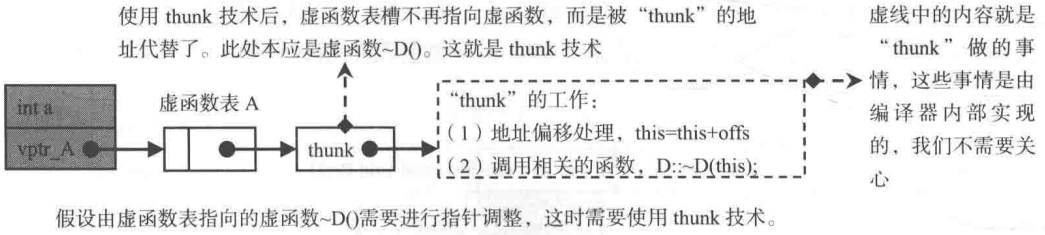


图 14.16 thunk 技术图示

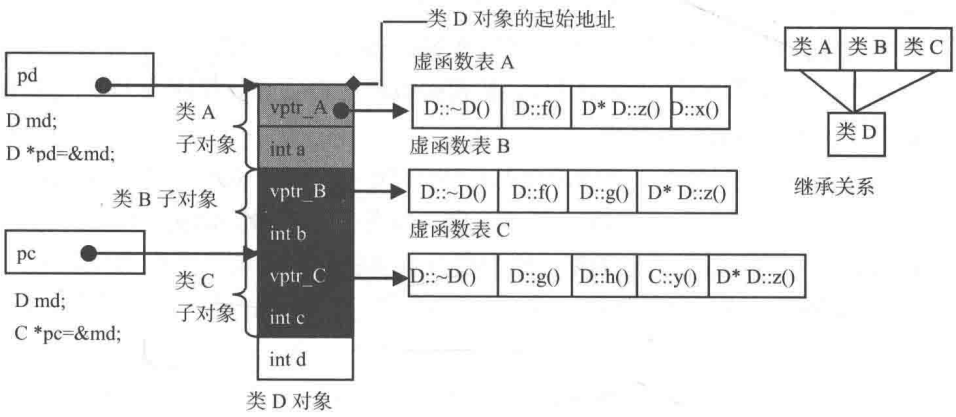
（3）其实利用 thunk 技术就只有两步：指针偏移，比如 this=this+offset；跳转去执行相关的函数，比如 D::~D(this)。

（4）thunk 技术只使用在需要进行指针调整时，若不需要对指针进行调整，则不用使用 thunk 技术，因此使用 thunk 技术不会带来额外的负担。

4．在多重继承下编译器的实际对象模型

这里以 VC++ 2010 为例进行讲解。VC++保证任何有虚函数的类的第一项永远是虚函数表指针。因此在多重继承下，若第一个父类没有虚函数，而第二个父类有虚函数，那么在对象模型中会导致有虚函数的父类放在没有虚函数的父类的前面。

在理想情况下多重继承有虚函数时的对象模型示意图如图 14.17 所示。



说明：

① 此处省略了虚函数表的槽索引值，并且把虚函数表所指向的函数直接移至虚函数表的槽内。

② 此处还省略了虚函数表中的类型信息 type_info。

③ 理想模型没有对虚函数进行指针调整，也未使用 thunk 技术。

④ 此模型把虚函数表指针（vptr）放于类对象的起始地址处（VC++就是如此操作的）。

图 14.17　在理想情况下多重继承有虚函数时的对象模型示意图

示例 14.7：多重继承下的对象模型

```
class A{public:int a; virtual ~A(){} virtual void f(){} virtual A* z(){return 0;}};
class B{public:int b; virtual ~B(){} virtual void f(){} virtual void g(){}
      virtual B* z(){return 0;}};
class C{public:int c; virtual ~C(){} virtual void g(){} virtual void h(){}
      virtual void y(){} virtual C* z(){return 0;}};
class D:public A,public B,public C{public:      //类 D 同时继承自 A、B、C，注意继承的顺序
      int d; virtual ~D(){} void f(){} void g(){}   void h(){}
      virtual void x(){} virtual D* z(){return 0;}};
void main(){}
```

对示例分析如下。

① 对虚析构函数的分析。

- 对于 C *pc=new D;，则 delete pc；会调用虚函数表 C 中被子类 D 重写的析构函数 D::~D()。
- 该析构函数会被编译器内部转换为~D(D*const this)，它需要一个类型为 D*的实参传递给 this 指针，而 delete pc;传入的是类型为 C*的实参，因此必须使 pc 的类型 C*与形参 this 的类型 D*一致。
- 把 pc 的类型 C*转换为 this 的类型 D*是容易的（在 C++中父类对象的地址是不可以赋给子类对象的，但进行强制类型转换后即可进行赋值，因为指针的类型只影响怎样解释指针指向的地址），但问题是实参 pc 进行类型转换并传递给形参 this 之后，this 指向的起始地址是类 D 中类 C 子对象处（见图 14.17），这与 this 所期望的类 D 的起始地址

不一致，因此需要对参数传递之后的 this 指针的起始地址进行调整（注意：使用 thunk 技术并未先对实参 pc 调整之后再进行参数传递，而是在参数传递之后再对形参 this 进行调整，最后再调用相应的函数）。要使参数传递之后 this 指向类 D 的起始地址，本例需要把 pc 向后移 16 字节。

- 若此处使用 thunk 技术，则虚函数表 C 中的 D::~D()应被 thunk 代替，即虚函数表指向“thunk”，而不再指向 D::~D()。同理，虚函数表 B 中的 D::~D()也会做此种调整，但虚函数表 A 不需要进行指针调整，因为类 A 实体的起始地址与类 D 实体相同。

② 对子类重写父类虚函数的分析（请结合图 14.17）。

- 对于 B* pb=new D;，则 pb->f()会调用虚函数表 B 中被子类 D 重写的虚函数 D::f()。
- 该虚函数被编译器内部转换为 f(D* const this)，而调用 pb->f()会被编译器内部转换为(*pb->vptr_B[1])(pb)，其中第二个 pb 是传递给被转换后的函数 f 的实参。由此可见，类 B 中的函数 f 的形参 this 需要一个 D*类型的实参，但实参 pb 的类型是 B*，所以此处必须使 pb 的类型与 this 的类型一致，并对指针所指向的起始位置进行相应的调整。
  - 对于此种情形的指针调整，各编译器厂家有不同的处理方法。下面介绍微软 VC++的处理策略。
  - 对于有虚函数的多重继承，只有当派生类的虚函数覆盖了多个基类的虚函数时，才使用 thunk 技术，否则使用 VC++自己的指针调整策略。
  - VC++自己的指针调整策略：强制把类 B 的虚函数表 vptr_B 中被子类 D 重写的虚函数 f()编译为一个接受类 D 的子对象 B 类型的 this 指针形参，也就是把 D::f()转换为 f(B*const this)，而不再是 f(D*const this)，因此 pb->f()可以直接调用被重写的 D::f()函数，而不需要进行指针调整。
  - 若使用 thunk 技术，本例需要把 pc 向后移 8 字节。
- 由上一点可知，在 VC++中：
  - 类 B 中被重写的函数 f 需要使用 thunk 技术，此时虚函数表中的 D::f()替换为“thunk”。
  - 类 A 中被重写的函数 f 本身不需要进行指针调整，因此不需要使用 thunk 技术。
  - 类 C 中被重写的函数 h 使用 VC++自己的指针调整策略（因为函数 h 未出现在多个父类中），不使用 thunk 技术。
  - 类 B 和类 C 中被类 D 重写的函数 g，只会对类 C 中的函数 g 使用 thunk 技术进行指针调整，调整后的指针指向类 B 子实体的起始处，即虚函数表 C 中的 g 函数。D::g()被调整为 g(B*const this)，而类 B 中的函数 g 不进行指针调整。

③ 对未被子类覆盖的虚函数（函数 y）的分析

使用子类指针调用父类未被覆盖的虚函数时，需要进行指针调整，但这种调整只需简单地加上偏移值即可。这种情况不影响继承下的对象模型。比如 D* pd=new D; pd->y();，在类 C 中

函数 C::y()被内部转换为 y(C*const this)，该函数需要一个 C 类型的形参，而 C 类型子实体与类 D 对象的起始地址不一致，因此需把 pd 的起始地址调整为类 C 实体的起始地址处。这是把子类对象地址赋值给父类对象，因此这种调整是比较简单的。

④ 注意：子类新增的虚函数会与第一个父类的虚函数表合并，因此类 D 中新增的虚函数 x 会与类 A 的虚函数表合并。

⑤ 对返回类型协变的虚函数（函数 z）的分析：

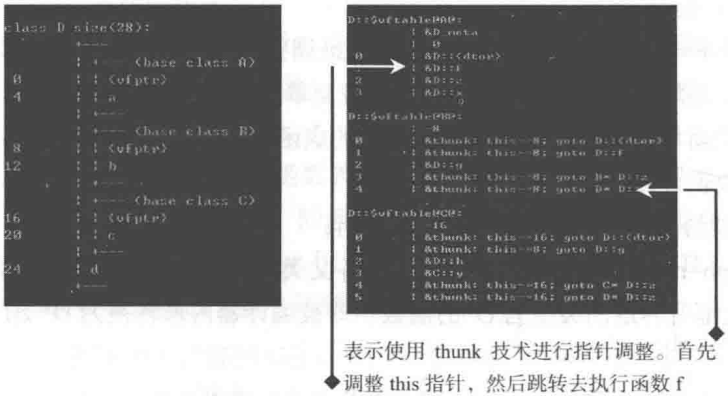
- 对于 B* pb=new D;，本例中虚函数 z 在各父类类型中的返回类型不同，但被子类重写之后，全都变为返回类型为 D*的函数，即被编译器内部转换为 D* z(D* const this)。
- 调用 pb->z()被内部转换为 z(pb)，此时 pb 与 this 的起始地址不同，应把 pb 指针的起始地址调整为指向类 D 的起始处，需要使用 thunk 技术。
- 若把 pb->z()的返回结果再赋给另一个父类对象时，比如 B* pb1=pb->z()，因为 pb->z()返回的类型是 D*，而 pb1 的类型是 B*，两个指针的起始地址又出现不同的情况，因此又需要把 pb->z()返回的指针调整为指针子类实体 B 的起始地址。在此种情形下就有两次指针调整，VC++的实现策略是在两次指针调整时都使用 thunk 技术，这样在对象模型中就保留了返回类型为 B*和 D*两个版本的 z 函数。可通过如下方法进行验证。

把以下程序可写入上面示例程序的 main 函数中，并直接运行，以验证 z 函数的 B*和 D*两个版本。

```
typedef void (*pf)()
D md;
((pf)*((int*)*((int*)(&md)+0)+3))();  //调用添加到虚函数表 A 末尾的子类 D 新增的虚函数 x
((pf)*((int*)*((int*)(&md)+2)+3))();  //调用虚函数表 B 中的第一个函数 z
((pf)*((int*)*((int*)(&md)+2)+4))();  //调用虚函数表 B 中的第二个函数 z，可见返回类型协变的函数
                                      //z 在 VC++中有两个版本
    int **p=(int**)&ma;  ((pf)pp[0][3])();  //二级指针形式
((pf)pp[2][3])(); ((pf)pp[2][4])();
```

⑥ 在 VC++ 2010 中使用 cl 命令查看类 D 的虚函数表的布局，如图 14.18 所示。cl 命令为 cl /d1 reportSingleClassLayoutD "xxxx.cpp"，其中 reportSingleClassLayoutD 表示查看类 D 的内存布局。注意：只有查看类 D 的内存布局才能看到其中所有的虚函数表情况。另外还要注意，需要运行一次程序产生新的 cpp 文件后，再使用 cl 命令，才能看到最新修改的 cpp 文件中的虚函数表结构。

⑦ 在 VC++ 2010 中，关于示例 14.7 在多重继承下有虚函数时的实际对象模型示意图如图 14.19 所示。



表示使用 thunk 技术进行指针调整。首先调整 this 指针，然后跳转去执行函数 f

位于类 D 中的子实体 A 的虚函数表 vftable，可以看到虚函数表 A 中的析构函数、函数 f、z 都被类 D 重写了。而 x 是子类 D 新增的虚函数，位于虚函数表 A 的末尾

图 14.18　使用 cl 命令查看示例 14.7 中类 D 的虚函数表的布局

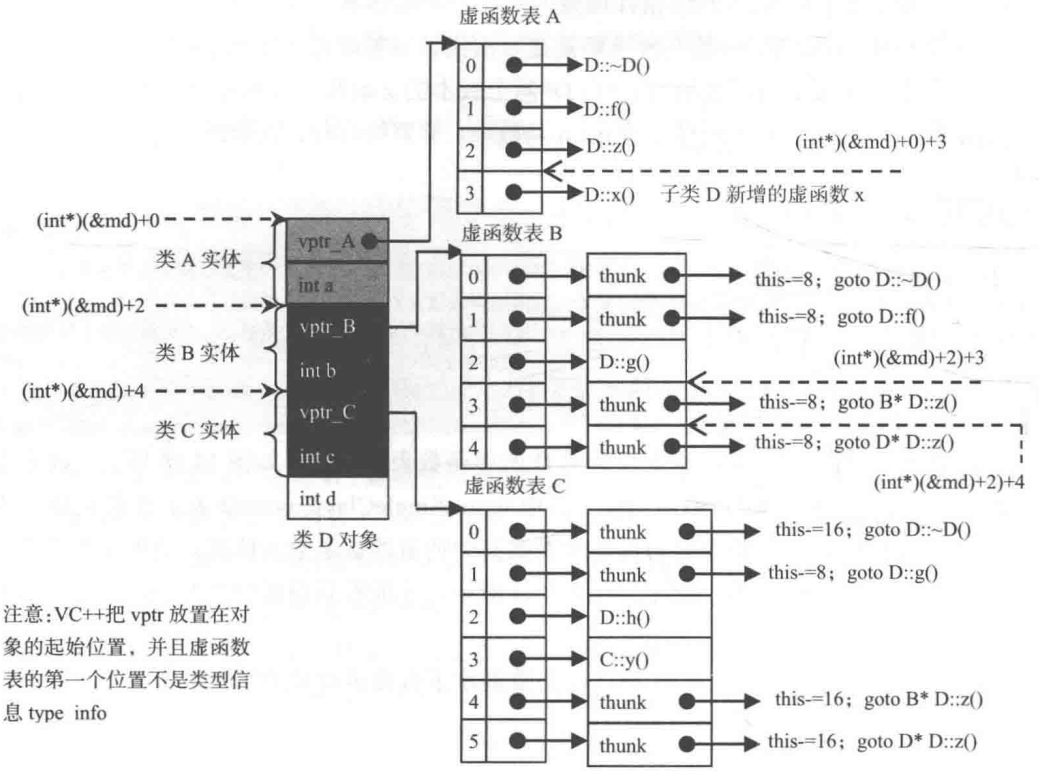


图 14.19　VC++在多重继承下有虚函数时的实际对象模型示意图

14.2.5　虚继承下的对象模型

1. 虚基类对象模型

在非虚继承下，有共同父类的对象模型示意图如图 14.20 所示。

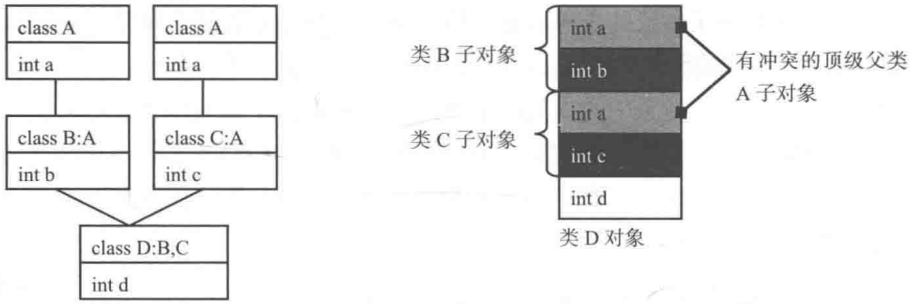


图 14.20　在非虚继承下有共同父类的对象模型示意图

由图 14.20 可见，在非虚继承下因为顶级父类 A 拥有两个副本，因此访问其子对象的成员时会发生冲突。

虚继承下的对象模型示意图如图 14.21 所示。

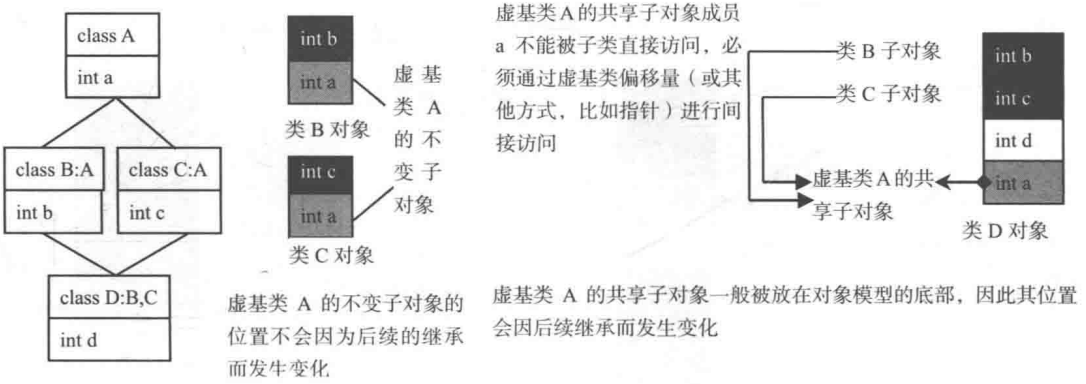


图 14.21　虚继承下的对象模型示意图

（1）解决顶级父类拥有多个副本的方法是在 C++中使用虚继承机制。

（2）在虚继承时，因为只有一个虚基类子对象的副本，因此这个副本一般都是被子类共享的。虚基类共享子对象的位置一般在子类的对象模型的底部，因此子类和虚基类共享子对象地址之间的偏移量是不固定的。若这个子类又被后续继承的话，那么最终子类存放虚基类共享子对象的位置与前一个子类对象不再相同，详见图 14.21 右图。

（3）虚继承下的对象模型的基本原理：编译器在实现虚继承时一般使用的方法是“分割虚基类法”，就是把虚基类分割为两部分，一部分为不变子对象，一部分为共享子对象，详见图14.21。不变子对象的位置不会因为后续继承而发生变化，共享子对象一般存在于后续继承中，其位置会因为后续继承而发生变化，因此对共享子对象的存取操作需要通过另一种策略来完成。

（4）存取虚基类共享子对象的三种策略：指针策略、在虚函数表中存储正负地址策略、虚基类表策略（微软 VC++使用的策略）。

（5）访问虚基类共享子对象中的成员：不管使用哪种策略存取虚基类的共享子对象，其子类都不能直接访问它的成员（包括虚函数），而必须通过虚基类偏移量进行地址计算之后（或其他方式，比如指针），再通过虚基类的共享子对象进行间接访问，访问虚函数还应通过虚基类共享子对象中的虚函数表（见图 14.21）。

2. 存取虚基类共享子对象的策略

（1）指针策略：其方法是编译器在每个子类中都安插一个指针，用于指向虚基类的共享子对象。访问虚基类共享子对象的成员时，只需使用指针间接完成即可。此方法有两个主要缺点，一是因为引入额外的指针而增加了类对象的负担，若虚基类有多个，则需要多个指针；二是若继承层次很深，则需要多次间接存取，这影响了存取的时间。此策略在 cfront 编译器上被采用，详见图 14.22。

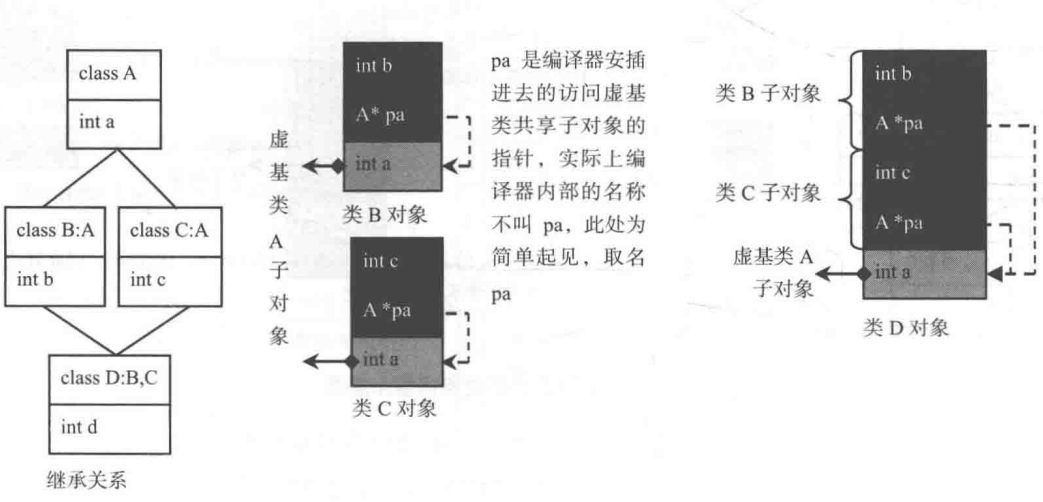


图 14.22　使用指针间接访问虚基类的共享子对象示意图

（2）在虚函数表中存储正负地址策略：其方法是在虚函数表中把虚基类共享子对象的偏移量和虚函数的地址放在一起，并根据正值或负值来进行存取，若为正值则存取虚函数，若为负值则存取虚基类共享子对象的偏移量，然后根据该偏移量找到虚基类的共享子对象。应注意负地址只存储了虚基类共享子对象的偏移量。sun 编译器使用此种策略，详见图 14.23。

（3）虚基类表策略：其方法是向每个继承自虚基类的类对象中增加一个所谓的“虚基类表指针（vbptr）”，该指针指向一个由全类共享的虚基类偏移量表，然后根据表中的偏移量计算出虚基类的位置。微软 VC++使用此策略，详见图 14.24。

（4）在虚继承下微软 VC++的内存布局顺序为：① 虚函数表指针；② 虚基类表指针（vbptr）；③ 数据成员；④整个虚基类的共享子对象。若虚基类的共享子对象中含有虚函数，则虚函数表指针也位于此处。

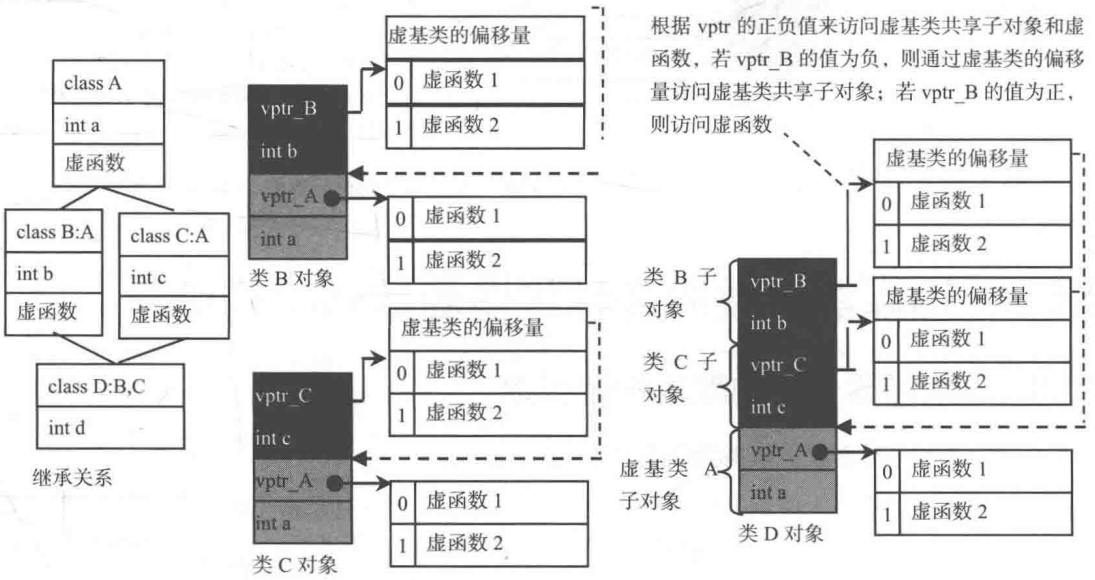


图 14.23　在虚函数表中使用正负地址访问虚基类共享子对象示意图

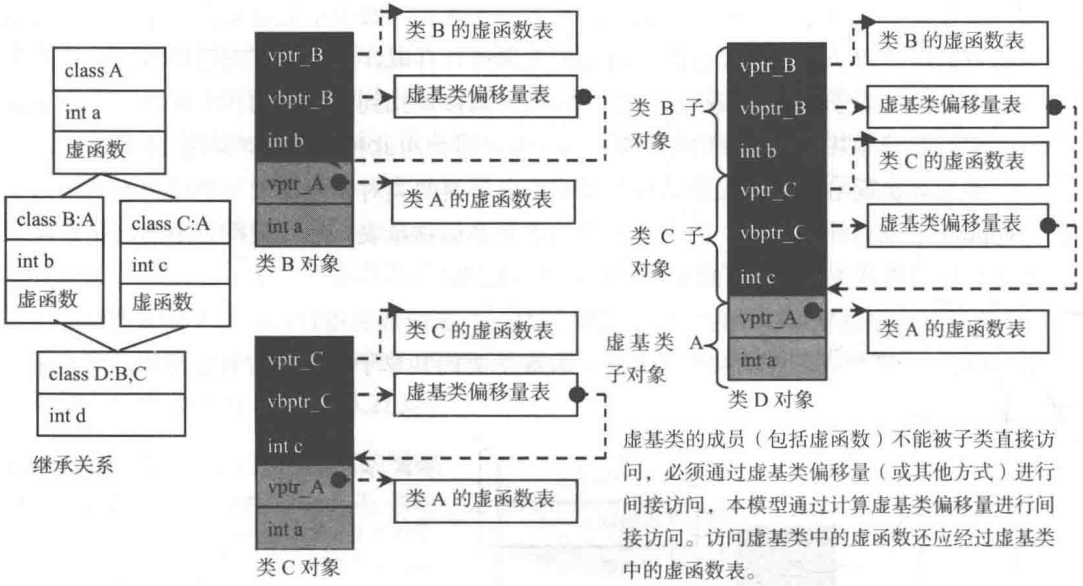


图 14.24　使用虚基类表访问虚基类共享子对象示意图

14.3　编译器合成的各种构造函数和析构函数

14.3.1　编译器合成的默认构造函数

1. 编译器何时合成默认构造函数

（1）如果用户没有为类定义任何构造函数，则编译器会暗中合成（或生成）一个默认构造函数，这个构造函数被称为“合成的默认构造函数”。实际上编译器会不会合成默认构造函数，这要视情况而定。

（2）编译器不会合成无用的默认构造函数，只有编译器认为有用的默认构造函数才会被合成出来。

（3）怎样判断编译器合成的默认构造函数是否有用：若类中成员的初始化是编译器的责任，则这时编译器会合成一个默认构造函数，这个合成的默认构造函数被认为是有用的；若类中成员的初始化是程序员的责任（比如 int 类型数据成员的初始化），而不是编译器的责任，那么合成的默认构造函数是无用的，实际上这种构造函数编译器并不会合成出来。比如 class A{public:int a; int *p;}; A ma;，编译器并不会合成默认构造函数，因为两个数据成员 a 和 p 的初始化并不是编译器的责任，这两个成员也没有被初始化为 0。

（4）在以下 4 种情况下合成的默认构造函数被编译器认为是有用的。

① 类中含有带有默认构造函数的类对象成员时。比如 class A{public:A(){}}; class B{A ma;};，类 B 中会合成一个默认构造函数。

② 父类中含有默认构造函数时。比如 class A{ public:A(){};class B:public A{};，类 B 中会合成一个默认构造函数。

③ 类中含有虚函数或继承含有虚函数的类时。比如 class A{virtual void f(){}};，会合成一个默认构造函数。

④ 类中含有一个虚基类时。

（5）对以上 4 种情况的说明。

① 在以上 4 种情况下，合成的默认构造函数只会初始化应由编译器负责的成员，若类中成员的初始化是程序员的责任（比如 int 类型数据成员的初始化），那么这些成员不会被编译器初始化。比如 class A{public: A(){}}; class B{public: A ma; int a;};，则类 B 中合成的有用的默认构造函数只会初始化类对象成员 ma，而 int 类型数据成员 a 编译器并不会初始化，a 的初始化应由程序员完成。

② 编译器会扩充已有的构造函数：在以上 4 种情况下，若用户已经定义了自己的构造函数，那么因为每一个构造函数都必须由编译器完成一些事情，这时编译器会扩充已存在的构造函数，并在其中安插一些必要的代码，以执行编译器的工作。比如 class A{public: A(){}};class B{public: A ma; B(int i){} B (int i, int j){}};，这时类 B 中定义的成员类对象 ma 并没有在用户定义的构造函数中初始化，但 ma 必须被初始化，这是编译器必须做的，因此编译器会扩充已有的构造函数，以完成对成员类对象的初始化操作。扩充之后形式如 B(int i){ma.A::A();}。

（6）默认构造函数并不是只要类没有定义构造函数就会被编译器合成出来，编译器合成的默认构造函数并不会对类中的每一个数据成员都设定默认值。

2．编译器合成默认构造函数的内部实现原理及相关代码

（1）合成默认构造函数情形一：类中含有带有默认构造函数的成员类对象时。

① 若没有对成员类对象进行显式初始化，则成员类对象的初始化是编译器的责任。因此，在此种情形下合成的默认构造函数是有用的，这个合成的默认构造函数只有在该构造函数真正被调用时才会执行。

② 若用户已经定义了构造函数，则编译器会扩充已存在的构造函数，并在其中安插一些必要的代码以初始化成员类对象，这些代码会在用户的代码之前执行。

③ 若有多个成员类对象，则按成员类对象声明的次序进行初始化。

示例 14.8：含有带有默认构造函数的成员类对象时合成的默认构造函数

```
class A{public: A(){} };                    //类A需含有默认构造函数
```

```
class B{public: int b; A ma; A ma1;   //成员类对象 ma 和 ma1 需含有默认构造函数
   B(int i){b=1;}
   B(int i, int j){}
   //以下为编译器在构造函数 B 内部扩充的代码
   //B(int i)              //为简化程序，未对构造函数的名称、形参、返回值进行内部转换
         //{ma.A::A(); ma1.A::A();   //编译器扩充的代码，调用相应的构造函数，按成员类对象
                                     //声明的次序进行初始化
         //b=1; }
         //B(int i, int j)
         //{ma.A::A(); ma1.A::A();}    //其原理同 B(int i)，但要注意，编译器扩充的代码只初
                                       //始化编译器负责的内容，数据成员 b 的初始化不是编
                                       //译器的责任，因此在 B(int,int)中编译器不会初始化 b
};
void main(){ B mb(9); }//合成的默认构造函数或扩充的代码只有在构造函数真正运行时才会执行
```

（2）合成默认构造函数情形二：父类中含有默认构造函数时，即形如 class B:public A{}，其中父类 A 含有默认构造函数。其原理与带有默认构造函数的成员类对象情形类似，只不过调用的是父类的默认构造函数初始化父类实体。若子类中同时存在成员类对象，则其默认构造函数的调用在所有父类的默认构造函数调用之后。

（3）合成默认构造函数情形三：类中含有虚函数或继承含有虚函数的类时，编译器必须为类对象的虚函数表指针（vptr）设置初始值，以指向合适的虚函数表（vtbl）地址。具体实现时，编译器会合成一个默认的构造函数或扩充已有的构造函数，并在其中安插一些代码来初始化类对象的 vptr，其内部代码形式为 B* B::B(B* this){ this->vptr_B=vtbl_B; }，即把类 B 的虚函数表指针 vptr_B 初始化为指向类 B 虚函数表 vtbl_B 的地址，其中 B* B::B(B* this)是编译器把默认的无参构造函数 B()进行内部转换后的结果。

（4）合成默认构造函数情形四：类中含有一个虚基类时。

① 虚基类在不同编译器中的实现方法差异很大，但有一个共同点——必须使虚基类在每个子类中的位置在执行期就准备好。各编译器厂家实现虚基类的原理不同，因此其初始化方法也不同。

② 一种比较简单的创建虚基类共享子对象的方法是，在子类中使用一个开关变量（或条件测试）来确定该子类是否需要创建虚基类共享子对象（即调用虚基类构造函数）。其原理为“只有当一个类对象被完全定义出来时，才创建虚基类共享子对象，若所创建的类对象只是某个类的子类对象，则不创建虚基类共享子对象”。比如类 D 同时继承自类 B、类 C，而类 B 和类 C 分别继承自类 A，假设 A 是虚基类，则由 D md 创建的对象 md 是完整对象，虚基类子对象由类 D 创建，而不由类 B 和类 C 创建。同理，若 B mb;创建的对象 mb 是完整对象，虚基类由类 B 创建。若类 E 再继承自类 D，则 E me 创建的对象是完整对象，虚基类由类 E 创建，而不由其父类 D、B、C 创建。下面通过示例进行说明。

示例 14.9：使用开关变量（或条件测试）来确定是否创建虚基类共享子对象

```
class A{public: A(){cout<<"A"<<endl;}};
class B:virtual public A{public:
   B(){cout<<"B"<<endl;}
   //以下为编译器在构造函数 B 内部扩充的代码
   //B* B(B* this, bool f)          //f 为确定是否创建虚基类的开关变量，当然在编译器内部并不是
                                    //这个名称。B* B(B* this...)是编译器对构造函数的名称、形
                                    //参、返回值进行内部转换后的结果
       //{ if(f==true) this->A::A(); //若开关变量 f 为真，则创建虚基类共享子对象
       //cout<<"B"<<endl;            //程序员自己写的代码
       //return this; }
       };
class C:virtual public A    {public:
   C(){cout<<"C"<<endl;}
   //以下为编译器在构造函数 C 内部扩充的代码
   //C* C(C* this, bool f)                        //同类 B
       //{ if(f==true) this->A::A();              //同类 B
       //cout<<"C"<<endl;                         //程序员自己写的代码
       //return this; }
       };
class D:public B,public C {public:
   D(){   cout<<"D"<<endl;}
   //以下为编译器在构造函数 D 内部扩充的代码
   //D* D(D* this,bool f)          //同类 B
   //{if(f==true) this->A::A(); //同类 B
       //this->B::B(false);  /*把实参 false 明确地传递给类 B 构造函数的开关变量。这样创建类 D 对象
时，就可以明确地告知类 B 不用再创建虚基类共享子对象了，而由类 D 创建，这就保证了子类 D 所创建的对象只有一个虚基
类共享子对象。因为在创建类 D 对象时，类 D 是完整的对象，虚基类子对象应由类 D 创建。而类 D 中的类 B 只是类 D 的子
类对象，这时类 B 对于类 D 而言不是完整对象，因此虚基类子对象不能由类 B 创建*/
       //this->C::C(false);  //解释同上
       //cout<<"<<D"<<endl;  //程序员自己写的代码
       //return this;}
       };
void main(){D md;}
```

3. 编译器合成的默认构造函数初始化各成员的顺序

（1）调用所有虚基类的构造函数初始化虚基类，顺序为从左至右、从最深到最浅。其初始化规则与普通构造函数相同，即若虚基类的构造函数位于成员初始化表中，则使用该表中明确指定的虚基类的构造函数初始化虚基类；若虚基类有默认构造函数但未出现在成员初始化表中，则调用虚基类的默认构造函数初始化虚基类子对象。注意：若是虚基类共享子对象的初始化，则编译器会使用一些策略来决定怎样初始化。

（2）按声明顺序调用上层父类的构造函数初始化父类部分，其初始化规则与继承时初始化父类的规则相同，此处不再重述。但要注意的是，若是多重继承，则第二个或后继父类的 this

指针的调整也在此时进行。

（3）初始化虚函数表指针（vptr），以使其指向正确的虚函数表。

（4）初始化成员初始化表中的类对象成员（包括有默认构造函数但未出现在成员初始化表中的类对象成员），按照声明类对象成员时的顺序进行。

（5）初始化成员初始化表中的数据成员。

（6）初始化的顺序总结为：虚基类、父类、虚函数表、对象成员、数据成员。

示例 14.10：编译器合成的默认构造函数初始化各成员的顺序

```
class A{public: virtual void f(){}A(){cout<<"A"<<endl;}};
class B{public: B(){cout<<"B"<<endl;}};
class C{public: C(){cout<<"C"<<endl;}};
class D:public A, virtual public B{public: C mc; int d;
   D():mc(),B(),A(),d(9){cout<<"D"<<endl;}  //初始化列表中各成员的顺序并不会影响各成员初始
                                                //化的顺序
    //以下为编译器在构造函数 D 内部扩充的代码
    //D* D(D* this,bool f)   //编译器对构造函数的名称、形参、返回值进行内部转换后的结果
                             //f 为是否创建虚基类子对象的开关变量
      //{{if(f==true) this->B::B();    //1. 按声明次序初始化虚基类子对象。根据开关变量 f
                                       //确定是否创建虚基类子对象
      //this->A::A();                  //2. 按声明次序初始化父类子对象
      //vptr_D=vtbl_D;                 //3. 初始化虚函数表指针 vptr_D，以使该指针指向正确的虚函
                                       //数表地址
      //mc.C::C();                     //4. 按声明次序初始化类中的类对象成员
      //this->d=9;                     //5. 初始化位于成员初始化列表中的数据成员
      //cout<<"D"<<endl;}              //程序员自己写的程序代码
      };
void main(){D md;}
```

14.3.2 编译器合成的复制构造函数与按成员初始化

（1）默认的按成员初始化方法：就是把现有对象的所有非静态成员逐个复制到新创建的对象中，对于对象成员则以递归方式执行按成员初始化方法。

（2）使用一个对象初始化相同类的类对象时，其编译器是按所谓的“默认的按成员初始化”方法来完成的。从概念上讲，“默认的按成员初始化”应该由复制构造函数来实现，但该操作并不要求一定要在复制构造函数内完成（注意理解此点），这意味着若类未定义复制构造函数，则编译器不一定会创建一个合成的复制构造函数。

（3）只有在复制成员是编译器的责任时，编译器才会合成一个复制构造函数，或者说只有有用的复制构造函数，编译器才会合成出来。

（4）编译器何时才会合成默认的复制构造函数（判断合成的复制构造函数是否有用的标准）：类是否表现出按位复制语意。只有当类不表现出按位复制语意时编译器才会合成一个复制

构造函数，或者说这时合成的复制构造函数才是有用的。由此可见，若类不表现出按位复制语意，则使用默认的按成员初始化方法进行初始化。

（5）在以下 4 种情况下不表现出“按位复制语意”（即编译器会合成复制构造函数）。

① 类中有一个成员类对象，而该成员类对象含有复制构造函数时（无论是用户明确声明的还是被编译器合成的）。

② 父类中含有复制构造函数时（无论是用户明确声明的还是被编译器合成的）。

③ 类中含有虚函数时。

④ 类继承自一个继承链，而其父类中有一个或多个是虚基类时。

（6）对以上 4 种情况的说明。

① 若程序员已经定义了复制构造函数，则在以上 4 种情况下，编译器会扩充已有的复制构造函数，以复制应由编译器负责的成员（注意：像内置数据成员等不是由编译器负责的不会被复制）。

② 情况③与④中合成的复制构造函数除了会处理编译器负责的内容外，类中的非静态数据成员会被按默认的按成员初始化方法进行复制，而合成的默认构造函数不会对成员进行处理。这是与合成的默认构造函数第一个不同的地方。

③ 情况①，若自定义了复制构造函数，对于情况②，若子类自定义了复制构造函数，则编译器并不会扩充该复制构造函数以调用成员类对象或父类的复制构造函数复制相应的部分（其实此时调用的是默认构造函数进行初始化），这时对成员类对象或父类部分的复制初始化就变成程序员的责任，而非编译器的责任了。这是与合成的默认构造函数第二个不同的地方。

示例 14.11：类中含有带复制构造函数的对象成员时编译器合成的复制构造函数

```
class A{public:          //对象成员 ma 所属的类 A 需有复制构造函数（或明确声明，或合成出来）
    A(){} A(const A & ma){cout<<"A&"<<endl;}};
class B{public:int b; A ma;};  //若该类合成出复制构造函数，则会对该类中的非静态数据成员进行按
                               //成员初始化处理
class C{public: int c; A ma; C(){}
    C(const C& mc){}};  //此复制构造函数不会被扩充，因为已被明确定义
void main(){ B mb; C mc; mb.b=2; mc.c=3;
    B mb1=mb;//输出 A&，此时类 B 中会合成出复制构造函数，该函数会调用类 A 的复制构造函数
    C mc1=mc; //无输出，显式定义的复制构造函数不会被扩充以调用父类 A 的复制构造函数（其实此时调用的是
              //父类 A 的默认构造函数）
cout<<mb1.b;} //输出 2，可见类 B 合成的复制构造函数对该类中的非静态数据成员进行了按成员初始化
```

编译器对类 B 合成出的复制构造函数如下：

```
//B(const B& mb){ma.A::A(mb.ma); b=mb.b;}//合成的复制构造函数，对类的非静态数据成员进行了处理
```

④ 情况③：类中含有虚函数时，因为每创建一个类对象就会有一个虚函数表指针（vptr）被创建，当使用子类对象对父类对象进行初始化时，若虚函数表指针不能指向正确的虚函数表，将产生可怕的结果，因此此时编译器必须对类对象的虚函数表指针进行正确的初始化。对虚函

数表指针的初始化是编译器的责任，因为程序员无法设置虚函数表指针的值，因此这时该类也就不能再表现出“按位复制语意”了。其处理方式就是使用合成的默认复制构造函数或扩充现有的复制构造函数并安插一些代码，以使父类的 vptr 明确指向父类的 vtbl。另外对于复制初始化的情形“以父类对象初始化另一个父类对象或以子类对象初始化另一个子类对象时”，虚函数表指针可以通过“按位复制语意”进行复制处理。还有一种复制初始化的情形是把子类对象的地址赋给父类指针，这时就实现了多态性。下面通过示例说明编译器必须合成复制构造函数的原理。

示例 14.12：类中含有虚函数时合成的复制构造函数

```
class A{public: A(){} virtual void f(){cout<<"AF"<<endl;};};
class B:public A{public: B(){}
     void f(){cout<<"BF"<<endl;} virtual void G(){cout<<"BG"<<endl;}};
void main(){B mb;
     A ma=mb;  //此时类 A 的复制构造函数会明确设定其对象 ma 中的 vptr 指向由类 A 生成的虚函数表，而不是
               //指向与对象 mb 相同的类 B 生成的虚函数表。详见图 14.25。注意：若 ma 是指针或引用，则此
               //处实现了虚函数的动态绑定
     ma.f();   //输出 AF，可见 ma 的 vptr 指向的是由类 A 生成的虚函数表，否则就会输出 BF
     B mb1=mb;
     mb1.f();} //输出 BF，以父类对象初始化另一个父类对象或以子类对象初始化另一个子类对象时，
               //虚函数表指针可以通过“按位复制语意”进行复制处理。详见图 14.25
```

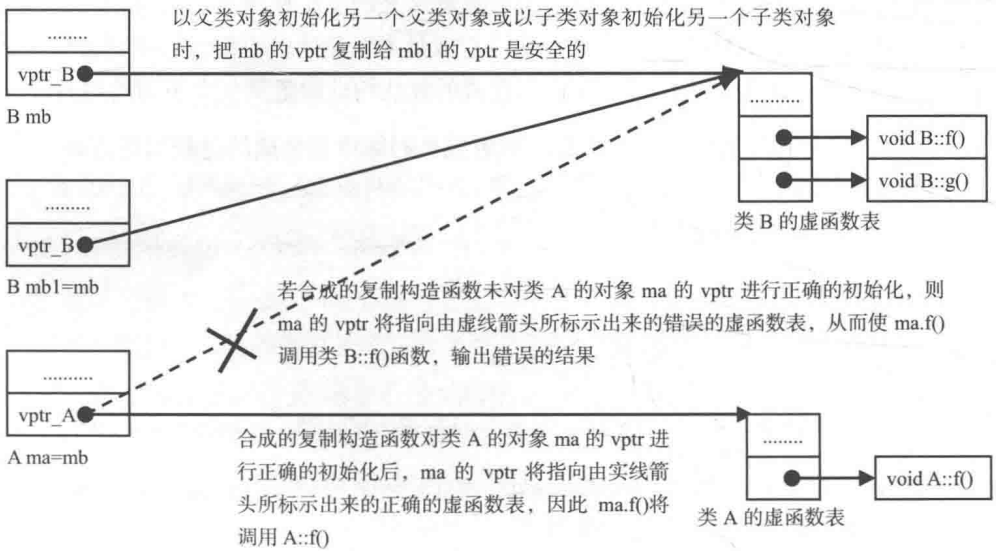


图 14.25 合成的默认复制构造函数对虚函数的处理

⑤ 情况④：类中含有虚基类时，暂不进行介绍。由虚基类的三种对象模型可以看出来，有虚基类时，编译器必须合成一个复制构造函数，以完成使用一个对象初始化另一个对象时访问

共享虚基类的指针偏移问题。

14.3.3　编译器合成的复制赋值操作符函数

（1）复制赋值操作符函数与复制构造函数类似，编译器是否会合成一个复制赋值操作符函数的标准同样是判断“该类是否表现出按位复制语意”，只有当类不表现出“按位复制语意”时编译器才会合成一个复制赋值操作符函数。

（2）在以下 4 种情况下不表现出“按位复制语意”，编译器会合成一个复制赋值操作符函数。

① 类中有一个成员类对象，而该成员类对象含有复制赋值操作符函数时。

② 父类中含有复制赋值操作符函数时。

③ 类中含有虚函数时。

④ 类继承自一个继承链，而其父类中有一个或多个是虚基类时。

（3）合成的复制赋值操作符函数无法解决对虚基类的共享子对象进行多次赋值的问题，C++标准对此未做出强制规定，很多编译器厂家也未解决这个问题，这应该是 C++语言本身的一个弱点，因此在程序中尽量不要对有虚基类的类进行赋值操作。

（4）复制赋值操作符函数的其他规则与复制构造函数类似，请参阅上一节的内容。

（5）编译器合成的复制赋值操作符函数的内部代码如下（假设类 B 继承自类 A）：

```
B& B::operator=(B* const this, const B& mp){
this->A::operator=(mp); }   //编译器内部代码，明确调用父类的复制赋值操作符函数
```

14.3.4　编译器合成的析构函数

（1）编译器合成析构函数的条件：只有在成员类对象或父类拥有析构函数的情况下，编译器才会合成析构函数。注意：虚函数、虚基类不是编译器合成析构函数的必要条件，这是与合成构造函数不同的地方。

（2）与编译器合成的构造函数相同，若程序员已经定义了析构函数，则合成的析构函数会扩充已有的析构函数。

（3）编译器合成的析构函数的调用顺序，与合成的默认构造函数的调用顺序相反。

14.4　类对象创建和销毁时编译器实现原理

1. 全局类对象

（1）全局类对象必须先被创建出来，然后才能在 main 函数中使用，在 main()函数结束之前销毁该全局类对象。

（2）编译器创建全局类对象就是使用静态初始化方法对其进行初始化，与之相对应的是静态内存释放操作。

（3）cfront 编译器使用静态初始化方法的策略是“munch 策略”，其他编译器不一定使用该策略。原理如下：

① 在对象文件中为每一个全局类对象都产生一个用于初始化的以_sti 开头的类似_sti_XXX()的函数，该函数内部有必要的构造函数调用操作。比如假设 ma 是类 A 类型的全局对象，则_sti_XXX 函数为_sti_XXX(){ma.A::A();}。其中 XXX 是编译器经过内部重命名之后的唯一名称，具体命名规则视编译器而定。

② 在对象文件中为每一个全局类对象都产生一个用于销毁对象的以_std 开头类似_std_XXX()的函数，该函数内部有必要的析构函数调用操作。

③ 提供一个_main()函数用于调用对象文件中所有以_sti 开头的_sti_XXX()函数以初始化全局类对象。与之相对应的是，再提供一个 exit()函数调用对象文件中所有以_std 开头的_std_XXX()函数以销毁全局类对象。_main()函数会被编译器安插在 main()函数中的第一个位置，作为 main()函数的第一个指令；同理，exit()函数作为 main 函数的最后一个指令。

④ 用于初始化和销毁对象的以_sti 和_std 开头的一系列函数，可能被存储在一个单独的文件内，具体是什么文件，则依编译器而定。

⑤ 经过以上处理之后，编译器内的 main 函数大致如图 14.26 所示。

（4）对于如何收集所有的以_sti 和_std 开头的函数，一般方法就是创建一个指令或文件专门用于搜索这些函数，然后把结果保存在另一个文件中，再把这些结果文件链接在一起。具体是怎样实施的，各编译器厂家有各自不同的指令或文件，因此不尽相同。

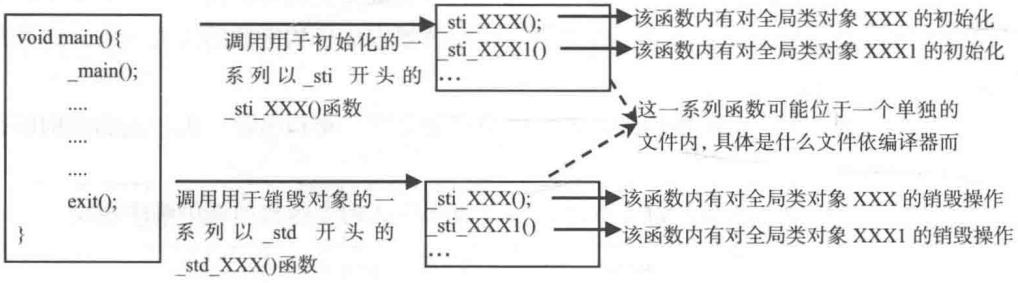


图 14.26 全局类对象静态初始化原理示意图

2. 局部静态类对象

（1）对于局部静态类对象，编译器需要解决的问题是怎样保证构造函数和析构函数只被调用一次。在 cfront 编译器中仍然是使用开关变量，但该开关变量是一个类类型的指针，且它的非零值比较特殊，是类对象的地址，详见以下示例。

示例 14.13：全局类对象和局部静态类对象编译器内部实现原理

（为便于阅读，省略了编译器的重命名和对函数的内部转换。）

```
class A{public: A(){cout<<"A"<<endl;}};
A ma; A ma1; A ma2;          //三个全局类对象
void f(){static A ma3; static A ma4;    //两个局部静态类对象
   //下面是编译器安插进来的处理局部静态类对象的内部代码
   //A *pf=0;//pf 为局部静态类对象的构造函数是否已经被调用过的开关变量，该开关变量是一个类类型的指针
   // if(pf) ;          //如果 pf 为非 0 值，则表示构造函数已经被调用过，程序什么也不做
   //else { ma3.A::A(); ma4.A::A(); pf=&ma3;}   //如果 pf 值为 0，则表示构造函数未被调用过
            //因此调用构造函数创建对象，并把开关变量设置为非 0 值。该非 0 值比较特殊，是类对象的地址
   } //函数 f 结束
   //下面是编译器安插进来的初始化全局类对象的内部代码（销毁全局类对象的内部代码与此类似）
   //1. 用于初始化的以_sti 开头的一系列_sti_XXX()函数，这些函数可能会存储在一个单独的文件内
   //void _sti_Ama(){ma.A::A();}
   //void _sti_Ama1(){ma1.A::A();}
   //void _sti_Ama2(){ma2.A::A();}
   //2. 提供一个_main()函数调用用于初始化的一系列以_sti 开头的_sti_XXX()函数
   //void _main(){_sti_Ama(); _sti_Ama1(); _sti_Ama2();}
void main(){ //_main();      //_main 函数成为 main 函数的第一个指令
   //exit();     // exit 函数成为 main 函数的最后一个指令
}
```

3．类对象数组

（1）如果类未定义任何构造函数，则定义一个类对象数组编译器只需分配足够的内存空间即可。

（2）若类定义了默认构造函数，那么当定义一个类对象数组时，编译器必须保证默认构造函数被应用于每一个数组元素上。编译器实现该功能时使用 vec_new()函数来完成对对象数组中各元素的初始化，使用 vec_delete()函数来完成对对象数组中各元素的销毁。

（3）若类定义了默认构造函数，且数组元素未被显式初始化，则对象数组中各元素的初始化使用 vec_new()函数来完成；否则不需要使用 vec_new()函数来完成初始化。比如假设类 A 定义了相应的构造函数，则对于 A ma[10]={A(), A(2);A(2,3)};，类对象数组 ma 的前 3 个元素被显式初始化了，因此前 3 个元素的初始化不需要使用 vec_new 函数，但未显式初始化的剩余 7 个元素，需要使用 vec_new()函数进行初始化。

（4）vec_new()函数的原型如下（共有 5 个形参）：

```
   void *vec_new(void *ar, size_t s, int c, void (*constructor)(void*), void (*destructor)
(void*,char) )
```

① 形参 ar，表示数组的起始地址。若为 0，则表示使用 new 分配数组。

② 形参 s，表示类对象的大小。size_t 是一个被 typedef 重命名后的 unsigned int 类型。

③ 形参 c，表示数组中元素的数目。

④ 形参 constructor，表示指向类的默认构造函数的指针。注意：默认构造函数被编译器内

部转换后有一个该类类型的 this 指针形参。

⑤ 形参 destructor，表示指向类的析构函数的指针。析构函数被编译器内部转换后有两个形参。

⑥ 理论上讲，是不可以取出构造函数或析构函数的地址的，但对于 vec_new()和 vec_delete()函数是个例外。

⑦ 示例（假设 A 已经定义了默认构造函数）：

- A ma[10]，编译器调用 vec_new()函数的形式为 vec_new(&ma, sizeof(A), 10, &A::A, 0)，其中&A::A 表示取出构造函数的地址。
- A ma[10]={A(), A(2);A(2,3)};，编译器调用 vec_new()函数的形式为 A ma[10]={A(), A(2), A(2,3), vec_new(&ma+3, sizeof(A), 7, &A::A, 0)}，可见已明确初始化的元素，不再需要使用 vec_new()进行初始化了。
- A*pa=new A[10];，编译器调用 vec_new()函数的形式为 vec_new (0,sizeof(A),10, &A::A,&A::~A);，注意第一个形参为 0，最后一个形参为析构函数地址。

（5）vec_delete()函数的原型如下（共 4 个形参，各形参的含义见 vec_new()函数）：

```
void* vec_delete(void *ar, size_t s, int c, void (*destructor)(void*,char));
```

（6）使用 delete[] p 动态释放内存时，编译器如何知道需要释放多少个数组元素的空间，或者说编译器怎样确定数组的实际大小呢？方法之一是使用“cookie 策略”。

（7）cookie 策略的核心思想还是使用开关变量，这种开关变量被称为 cookie，并且其存储和取出操作分别使用两个不同的函数来实现。其具体实现是使用 vec_new()函数初始化数组对象时，就把这个中间变量 cookie 存储起来，其方法是通过调用存储 cookie 的函数来完成。比如有些编译器使用 int _insert_new_array(void* array_key, int elem_count) 函数存储 cookie，其中第一个参数表示数组的地址，第二个参数表示数组的元素数目；使用 int _remove_old_array(void * array_key) 函数取出 cookie。这两个函数的名称比较长，当然编译器内部不一定会使用这两个名称。

示例 14.14：vec_enw()函数内部代码（cfront 编译器）

```
//以下代码未使用编译器的重命名规则，因为重命名后名称太长了，不便阅读，同时省略了异常处理的内容
//以下代码中的_insert_new_array()和constructor()函数是由编译器内部实现的，具体的内部代码从略
void *vec_new(void *par,int c,int s, void (*constructor)(void*) )  //各形参的含义参见正文,
                                                          //此处 c 表示元素数目，s 表示对象的大小
   {int sc=s*c;       //对象数组所占内存空间的总大小
   if(par==0)par=(void*)new char[sc];   //若数组的起始地址为 0，则表示该数组是使用 new 进行动态
                  //分配的，这里使用 new 为对象数组分配足够大小的空间。此处要注意理解指针的类型
   _insert_new_array(par,c); //这就是 cookie 策略。使用_insert_new_array()函数存储数组的大小
   if(constructor){       //若构造函数的地址不为 0，则按以下方式创建对象数组中的每一个元素
      char *pc=(char*)par;   //把数组的地址强制转换为 char 类型，以方便指针位置的偏移计算
      char *pe=pc+sc;        //表示对象数组占据的内存地址最大值，sc 是对象数组所占内存空间的总
```

```
                                //大小，pc 是数组的起始地址
    while(pc<pe){               //使用循环语句，逐个初始化对象数组中的每一个元素
      constructor((void*)pc);  //调用构造函数初始化对象。构造函数被编译器内部转换后，
                               //需要传递一个类对象的地址给构造函数作为形参的值。比如构造函数 A() 会被
                               //编译器内部转换为 A(const A* this)，就需要一个类对象的地址，这里的 pc
                               //被作为 this 指针的值传递进去，然后 pc 向后偏移对象的大小即后面的 pc=pc+s
                               //以初始化对象数组中的下一个元素
      pc=pc+s;}   //while 循环结束
  } //if 语句结束
  return (void*)par; } //返回数组的起始地址
```

示例 14.15：使用 vec_new()函数初始化对象数组

```
class A{public:A(){} A(int i){}};     //类 A 应定义默认构造函数
class B{};                            //类 B 未定义默认构造函数
void main(){A ma1[11]; A ma2[11]={A(),A(3)}; A *pa=new A[11];
    B mb[11];      //因为类 B 未定义默认构造函数，因此 mb 不需要使用 vec_new() 函数进行初始化
//下面为编译器使用 vec_new() 函数初始化类对象数组各元素的扩充代码
//vec_new(&ma1, sizeof(A), 11, &A::A, 0 )   //初始化数组 ma1[11]
//A ma2[11]={A(), A(3), vec_new(&ma2+2, sizeof(A), 9, &A::A, 0 )}  //初始化数组 ma2[11]
                                      //已明确初始化的元素不再需要使用 vec_new() 函数进行初始化
//vec_new(0,sizeof(A),10,&A::A,&A::~A);     //初始化 new 分配的对象数组。注意第一个参数为 0，最后
                                            //一个参数为析构函数的地址
}
```

第 15 章

模板专题

注意：为方便讲解，在定义或声明函数时会省略函数形参名，比如 void f(int i);被简写为 void f(int);，template<class T> void f(T a){};被简写为 template<class T> void f(T){};。

（1）泛型编程：指的是直接应用“使用模板编写好的函数模板或类模板库程序”进行编程。在具体使用时，程序员只需向所使用的库函数或类模板提供类型或值即可。模板是泛型编程的基础。比如使用标准模板库中的容器、迭代器进行编程，就是泛型编程。

（2）易混概念。

① 模板函数与函数模板：一般而言，“模板函数”着重强调这是一个函数，这个函数是使用模板实现的；而“函数模板”着重强调这是一个模板，这个模板实现的是函数的功能。因此函数模板与模板函数并没有本质的区别，都是函数和模板的结合体。

② 类模板与模板类：解释同上。

③ 成员模板函数、函数模板成员、模板函数成员等：这些概念也没什么区别，它们都是模板、函数和成员的结合体，只是强调的方向不同，意思都是一样的。

（3）实例与实体：它们没有本质的区别，有时为了称呼方便会根据情况选择实例或实体。在大多数情况下，本文中实例指的是由模板实例化时编译器所重新编写（或定义）的函数或者类。当然也可以把类的对象称为实例。比如 class A{};，那么对于 A ma;，ma 就是类 A 的一个实例；template<class T>void f(T i){...};，模板函数 f 的 int 类型的实例是 void f(int i){...};，即模板函数 f 的实例就是有一个 int 类型形参的普通函数 f；template<class T> class A{T i;};，类模板 A 的 int 类型的实例是 class A{int i;};。

15.1 模板基础

注意：模板的声明或定义只能在全局范围、命名空间或类内进行，而不能在局部范围、函数内进行声明或定义。比如不能在 main 函数中声明或定义一个模板。

1．为什么要使用模板

（1）使用模板是为了能够让程序员编写出与类型无关的代码。比如有比较两个数大小的函数 int f(int a, int b){return a>b?a:b;}，在不使用模板的情况下，这个函数只有对 int 类型数据进行比较才能得到正确的结果。对于 double、char 等类型就需要另一个用于比较的函数 f 的重载版本，它们之间仅仅是参数的类型不同，而函数体是相同的。使用模板可以让程序的实现与类型无关。比如只需一个用于比较两个数大小的模板函数 f，就可以实现对 int、double、char 等类型数据进行比较，而不需要多个函数 f 的重载版本。因此，如果函数使用模板实现，那么这个模板函数就可以代表一个拥有相同功能但类型不同的函数家族。同理，类也可以使用模板实现，其原理与函数相同。

2．类模板和函数模板的通式

（1）函数模板：template <class 或 typename 模板形参名, ...> 返回类型 函数名(形参列表){...}。

（2）类模板：template<class 或 typename 模板形参名, ...> class 类名{...}。

（3）函数模板示例：template<class T> void f(T a){T i;}，其中 T 是模板形参，i 和 a 是类型为 T 的变量。

调用函数模板：f<int>(4);f(5);，尖括号中的 int 是模板实参，表示把模板形参 T 的值赋为 int。f<int>(4)表示使用显式模板实参调用函数，f(5)表示使用模板实参推演推导出 T=int。

当 T 被确定为 int 之后，函数模板 f 相当于 void f(int a){int i;}，即 T 被 int 所代替。

（4）类模板示例：template<class T1, class T2> class A{};，其中 T1 和 T2 是模板形参。

使用类模板创建对象：A<int,double> ma，类模板必须使用显式模板实参创建对象。

3．理解模板（关键是模板形参、模板实参、实例化三个概念）

（1）模板以关键字 template 开始，并在其后加上尖括号“<>”。

（2）模板形参：在声明或定义模板时尖括号中的就是模板形参，其形式为“class 或 typename 模板形参名（非类型形参除外，详见后文）”。比如 template<class T> void f();，其中 T 就是模板形参，多个模板形参之间使用逗号隔开，模板形参名可以是任意合法的标识符。在这里 typename 和 class 没有区别，只是 typename 是较新的关键字。本文中除非特殊情况，否则使用 class 关键字，以缩短程序的长度。

（3）模板形参表：在 template 关键字之后，由被尖括号括起来的以逗号作为分隔符的所有模板形参组成。比如 template<class T1,class T2> void f();，其中尖括号中的“class T1,class T2”就组成了模板形参表。

（4）模板形参的作用：与函数形参相同，代表的是某个未知的“值”，但这个“值”表示的是某种类型（类型形参）或者某个具体的值（非类型形参）。若模板形参代表的是某种类型，则

可以像使用内置类型那样使用模板形参来指定返回类型、声明变量等；若模板形参代表的是非类型形参，则表示是某个具体的值。当然，有模板形参就有相对应的模板实参，这与函数类似。比如 template<class T> void f(){T i;}，其中模板形参 T 就表示某种未知的类型，在函数 f 中声明了一个名称为“i”的变量，其类型为未知的 T。

（5）模板实参：与模板形参相对应，用来传递给模板形参以确定其“值”，这里的值有可能是某种类型（即类型实参），也有可能是某个具体的值（即非类型实参），而非传统意义上的具体的数值。若模板形参代表的是某种类型，则模板实参就是具体的一种类型，该类型可以是自定义类型（如类类型）、内置类型（如 int、float 等），甚至可以是类模板类型。

（6）实例化（此概念需要先行掌握）：其实就是确定模板形参的“值”的过程，此过程会产生模板的一个具体的实例或实体。模板形参的“值”是通过传递进来的模板实参确定的，确定之后，模板中凡是用到模板形参的地方都会被相应的模板实参指定的“值”所代替。其实在实例化时编译器会使用模板实参代替所有出现的模板形参对模板进行重新编写（或定义），从而创建出该模板的一个实例，即函数或类的一个定义。比如 template<class T> void f(T){T a;}; f<int>(3);，则对 f 实例化后，编译器会对 f 函数模板进行重新编写，函数中的 T 全部被替换为 int，因此创建出的函数 f 的实例形式为 void f(int){int a;};，即 f 函数模板的 int 实例就是对函数 f 的 int 版本的重新编写（或定义）。

4. 模板形参和模板实参的分类及其传递基本规则

（1）模板形参分为：类型（模板）形参、非类型（模板）形参、模板模板形参。

① 类型形参：在尖括号中使用 class 或 typename 关键字所声明的标识符就是“类型形参”，其名字由用户自己确定。类型形参表示的是某一未知的类型，可以像使用内置类型那样使用模板类型形参来指定返回类型、声明变量等。比如 template<class T> void f(){T i;}，其中 T 就是类型形参，i 是类型为 T 的变量，只是 T 的类型暂时未知。

② 非类型形参：与普通的函数形参相同，是使用内置类型声明的模板形参；非类型形参表示的是某一个具体值，它不能表示某一类型。比如 template<class T, int M> class B{};，其中 M 就是非类型形参，表示某一个具体值。

③ 模板模板形参：即模板形参是一个类模板类型的。比如 template<template<class T> class TT> class A{};，其中 TT 就是模板模板形参。

（2）模板实参分为：类型（模板）实参、非类型（模板）实参、模板模板实参。这与模板形参相对应。

（3）模板实参与模板形参之间的传递基本规则：模板实参的个数和顺序应与模板形参相对应，模板实参所使用的“值:（可能为某一类型或一个具体值）应与模板形参相匹配。

5. 模板形参基本要求

（1）模板形参表的内容不能为空（这一点与函数形参不同）。注意：模板全局特化时例外。

（2）多个模板形参之间使用逗号隔开。

（3）模板形参名可以是任意合法的 C++标识符，但习惯上都以大写的 T 或类似名称命名模板形参。

（4）模板形参名可以省略。与函数相似，对模板的不同声明，其模板形参名可以不同，且模板声明与模板定义时的模板形参名也可以不同。比如声明 template<class T> void f();，则定义时可以使用不同的模板形参名，如 template<class T1>void f(){};。

（5）模板形参的名称在同一个模板形参表中只能使用一次，但在函数形参表中可以出现多次。比如 template<class T, class T>void f(){}是错误的，template<class T> void f(T a, T b){}是正确的，在函数形参表中模板形参名 T 可出现多次。

（6）模板形参名可以在不同的模板之间被重复使用。比如 template<class T>void f(){} template<class T>void g(){}，在 g 和 f 中的模板形参名 T 可以是相同的。

（7）若有多个模板类型形参（非类型形参类似），则在每个类型形参之前都应使用关键字 class 或 typename。比如 template<class T, T1>void f(){}错误，因为在 T1 前没有 class 或 typename。

（8）模板形参可以拥有默认值。

（9）后声明的模板形参可以使用先声明的模板形参名。比如 template<class T, T a> void f(){};，这里要注意 T a 应是非类型形参，因此应遵守非类型形参的限制。再比如 template<int a, int b=a>class A{}; A<2> ma; 是正确的，注意函数模板不能使用默认模板实参。而在普通函数中后声明的形参是不允许使用先声明的形参名来作为默认值的，比如 void f(int a, int b=a);是错误的。

（10）在模板内部不能声明与模板形参同名的名称。注意：VC++ 2010 不遵守此规则。比如 template<class T>void f(){typedef int T; T a; } 是错误的，但在 VC++ 2010 中不会出错。

（11）模板形参遵守常规的名称隐藏规则，比如模板形参会隐藏同名的全局名称。

6. 调用模板函数（即实例化函数模板）

调用模板函数的过程就是实例化函数模板的过程，这个过程首先需要指定模板实参或模板实参位于什么位置。调用模板函数有两种方法。

（1）模板实参推演：就是像普通函数一样调用模板函数，该方法根据函数的实参类型推导出函数模板的形参类型。比如 template<class T>void f(T ma){T i};，那么对于 f(4)，因为函数实参 4 的类型为 int，因此由函数头 f(T ma)可推导出模板形参 T 的类型为 int，之后函数 f 中凡是用到模板形参 T 的地方都会被 int 所代替，此时模板函数 f 相当于 void f(int ma){int i}普通函数。

（2）显式模板实参：即明确指定模板实参，该方法是在调用函数（模板）时的尖括号中显式指定模板实参。其形式为：函数名<模板实参,...>(函数实参,...)，尖括号中的内容在显式指定

模板实参时被称为“模板实参表”。比如 f<int>();，其中 int 就是模板实参；再比如 f<int ,double>(4,3);，其中 int 和 double 是模板实参。

7. 创建类模板的对象（或实例化类模板）

（1）创建类模板的对象（实例化类模板）：只能使用显式模板实参创建类模板的对象（不能使用模板实参推演），其形式为：“类名<模板实参, ...>对象名”。对于类模板只能在类名之后的尖括号中明确指定模板实参，模板实参之间使用逗号隔开。比如有类模板 A，则使用类模板创建对象的方法为 A<int>m;，其中 int 就是模板实参，在创建对象 m 之后，类 A 中凡是用到模板形参的地方都会被 int 所代替。再比如 A<int, double> m，模板实参之间用逗号隔开。

（2）在类模板外定义成员函数的方法：比如 template<class T1, class T2> class A{void f();};，则定义的方法为 template<class T1, class T2> void A<T1,T2>::f(){};。

8. 模板的链接性及修饰模板的关键字位置

（1）对模板进行修饰的关键字（如 friend, inline, const, extern, virtual, static 等）应位于 template<模板形参表>之后。比如 template<class T> static void f();，声明 f 为静态函数；再比如 extern template<class T> void f(); 错误，因为 extern 不能在 template 之前。

（2）模板的链接性：模板默认为外部链接。函数模板可以为内部链接，即可以使用 static 修饰。比如 template<class T> static void f(){}，注意 static 关键字应位于 template 之后。

（3）内联函数模板需要使用关键字 inline 明确指定（除非其本身就是内联的，如位于类的内部）。注意：模板的包含模型（详见后文）把函数模板定义于头文件内（这与内联函数相同），从而让人误以为函数模板默认是内联的，但其实不是。

示例 15.1：模板基础示例（快速入门）

```
#include <iostream>
using namespace std;
template<class T>    //声明一个模板函数，其中 T 是类型形参，其名称 T 可以是任意合法的标识符，class
                     //可使用 typename 代替
void f(T){T i;}   //i 是类型为 T 的变量。类型形参 T 表示某一未知类型，可以像内置类型一样使用
//template<> void f1();     //错误，声明或定义模板时其模板形参表不能为空（全局特化时例外）
template<class > void f2();       //正确，模板形参名可省略
template<class T1> void f2();     //正确，对模板的不同声明，其模板形参名可以不同
template<class T2> void f2(){}  //正确，模板声明与模板定义时的模板形参名可以不同
//template<class T, class T> void f3(){}//错误，模板形参名 T 在同一个模板形参表中只能使用一次
template<class T> void f4(T a,T b){}   //正确，模板形参名 T 在函数形参表中可以出现多次
//template<class T, T1>void f5(){}    //错误，在每个类型形参之前都应使用关键字 class 或 typename
template<class T, T a> void f6(){};    //正确，后声明的模板形参可以使用先声明的模板形参名
                                       //此处 T a 应是非类型形参，应遵守非类型形参的限制
template<int a, int b=a>class A{};     //正确，原因同上
//template<int a,int b=a> void f7(){}; //错误，函数模板不能使用默认模板实参
```

```
//void f8(int a, int b=a); //错误，在普通函数中后声明的形参不能使用先声明的形参名作为默认值
//template<class T>void f9(){typedef int T; T a;} //错误，在模板内部不能声明与模板形参同名的
                                                    //名称。但 VC++ 2010 可以
int N=1;
template<class N> void f10(){N i;}    //正确，N 是模板形参而不是全局变量，模板形参遵守常规的名称
                                      //隐藏规则，比如模板形参会隐藏同名的全局名称
//static template<class T> void f11(); //错误，static 应位于 template 的模板形参表之后
template<class T> static void f12(){}  //正确，原因同上
template<class T1,class T2> class B{public: void g(); };
template<class T1,class T2> void B<T1,T2>::g(){} //正确，在类模板外定义成员函数的方法
void main(){
   f(3);     //正确，使用模板实参推演调用函数模板，此处根据 3 的类型 int 推导出 T=int，此时会对 f 进行
             //实例化，实例化之后模板 f 中的所有模板形参 T 都会被模板实参 int 替换
   f<float>(4);        //正确，使用显式模板实参调用函数模板，T=float
   f10<float>();
   //B<3,4.4> mb1;     //错误，不能使用模板实参推演创建类模板的对象
   B<int,float> mb; } //正确，只能使用显式模板实参创建类模板的对象
```

15.2　模板形参与模板实参详解

15.2.1　类型形/实参与非类型形/实参

1. 类型形参与类型实参

（1）类型形参：在尖括号中使用 class 或 typename 关键字所声明的标识符就是“类型形参”，其名字由用户自己确定。类型形参表示的是某一未知的类型。

（2）类型形参的具体类型要根据该模板被调用时传递进来的模板实参而定，这与函数的形参和实参之间的关系是类似的。

（3）类型实参应与类型形参相匹配，因为类型形参表示的是某一类型，因此与其相对应的类型实参可以是任意类型（局部类和局部枚举除外），但不能是一个具体值。比如类型实参可以是指针、内置类型、类类型，甚至可以是一个能匹配的模板形参，还可以是模板类型。比如 template<class T>void f(){}; template<class T1>class A{ void g(){ f<T1>(); }};，则 f<3>();是错误的，因为数值 3 不是类型；再比如 A<int> ma1; f<A>();正确，A<A<int> > ma; 也是正确的，ma 的类型实参为 A<int>类型。同理，在类 A 中对函数 f 的调用 f<T1>()也是正确的，这里传递给函数 f 的模板实参就是类 A 的模板形参 T1。注意：A<A<int> > 后面的两个尖括号“>>”之间应留一个空格，否则会被编译器认为是一个右移运算符。

（4）局部类和局部枚举不能作为类型实参，也就是在函数内部声明的类和枚举不能作为类型实参（VC++ 2010 不遵守此规则）。比如 template<class T> void f(){} void g(){class A{}; f<A>();}，其中 f<A>();是错误的。

（5）在模板中对模板实参所属类型使用的运算符，必须是该类型所支持的运算符。比如template<class T> void f(){T a,b; a+b;} class A{};，则 f<A>();是错误的，错在函数 f 内部的 a+b;语句，因为模板实参所属的类型 A 并不支持“+”运算符操作。

2. 非类型形参与非类型实参

（1）非类型形参：与普通的函数形参相同，是使用内置类型声明的模板形参；非类型形参只表示某一个具体值，它不能表示某一类型的模板形参。比如 template<class T, int M> class B{};，其中 M 就是非类型形参。

（2）非类型形参的限制：只能是整型（包括枚举、char 类型）、指针和引用。比如 template<double M>void f(); 错误，因为非类型形参 M 是 double 类型，而 template<double *P>void f(){} 正确。

（3）非类型形参的基本特点：虽然从语法上看起来是一个变量，但非类型形参总是右值，而且是常量值，因此不能对非类型形参进行取址和赋值。可以把非类型形参作为常量表达式使用，如用于指定数组的长度。比如 template<int a, int b>void f(){ a=3; int c[a];};，其中 a=3;是错误的，因为非类型形参是右值；而 c[a]是正确的。

（4）因为非类型形参是常量值，因此在声明非类型形参时加上 const 是多余的，编译器也会忽略掉 const。比如 template<int a, const int b>void f(){ };，则 const int b;与 int b;是相同的，编译器会忽略掉 const。

（5）有非类型形参的类所创建对象的类型等价原则：只要对实参的求值结果是不同的表达式，那么它们的类型就是不同的。换句话说，只要对实参的求值结果是相同的表达式，那么就认为它们是等同的。此规则只适用于类模板。比如 template<int N> class A{}; A<20> ma;，则 A<10+10> mc=ma;正确，ma 与 mc 是相同的类型；而 A<40> mb=ma;错误，mb 与 ma 是不同的类型。

（6）非类型实参的基本限制（注意：不是形参）。

① 非类型实参的类型必须与相对应的非类型形参的类型相匹配。

② 非类型实参只能是常量值或常量表达式（包括枚举值），即必须能在编译时计算出结果。这些常量值的类型必须与非类型形参的类型相匹配，或者能隐式转换为非类型形参的类型。比如 char 值可隐式转换为 int 值。

③ 具有内部链接性的指针不能作为非类型实参。

④ 具有外部链接性的指针常量可以作为非类型实参。

⑤ 局部作用域中对象的地址不是常量表达式（包括局部指针），不能作为非类型实参。

⑥ 若非类型形参为引用，则具有内部链接性的变量不能作为非类型形参的引用，局部作用域中的对象也不能作为非类型形参的引用。

（7）可以作为非类型实参的表达式。

① 常量表达式、枚举、常量值可作为非类型实参。

② sizeof 表达式的结果是常量表达式，可作为非类型实参。

③ 由常量表达式初始化的 const 变量是常量表达式，可以作为非类型实参。比如 template<int N> void f(){} int b=4; const int a=3; const int c=b;，则 f<a>(); f(3.4)正确，但 f<c>错误，因为 c 不是使用常量表达式初始化的，所以 c 不是常量表达式。

④ 具有外部链接性的名称空间作用域中（包括全局名称空间作用域）任何对象的地址都是常量表达式，可以作为非类型实参。注意：若对象被 const 限制，则对象的地址就变成具有内部链接性了，这时就不能作为非类型实参。此规则同样适用于非类型形参的引用。比如：

```
namespace N{int a=1; const int b=3;} template<int *M>void f(){} int b=2;
void main(){int c=3; f<&N::a>(); f<&N::b>(); f<&b>(); f<&c>();}
```

其中 f<&c>()和 f<&N::b>是错误的，因为 c 是局部变量的地址，b 的地址具有内部链接性；f<&N::a>()和 f<&b>()是正确的。

⑤ 指向外部链接对象的指针常量可以作为非类型实参。在 VC++ 2010 中，声明的指针被认为是变量而不是常量，如 extern const int *const p;，指针 p 仍被 VC++ 2010 认为是变量。比如 int a[4]={0};extern const int *const p=a; template<const int *N> void f(){};，则 f<a>();正确，但 f<p>();错误，因为指针 p 不是常量。

⑥ 指向函数类型的指针常量可以作为非类型实参，这时应注意与非类型形参类型的匹配。比如 void f(){}template<void (*N)()>void g(){};，则 g<f>();是正确的；但对于 void (*p)()=f;，则 g<p>();是错误的，因为 p 是变量而不是常量。

⑦ 静态类成员的地址可以作为非类型实参，这时应注意类型的匹配。比如 class A{public:static int a;}; template<int *N> void f(){};，则 f<&A::a>();是正确的。

⑧ 指向类成员的指针常量可以作为非类型实参，这时应注意与非类型形参类型的匹配。比如 class A{public: int b;}; template<int A::*N> void f(){};，则 f<&A::b>();是正确的；但对于 int A::*p=&A::b;，则 f<p>();是错误的，因为 p 不是常量，p 是指针变量。

（8）不能作为非类型实参的表达式。

① 浮点型值不能作为非类型实参。但 VC++ 2010 是可以的。

② 空指针常量不能作为非类型实参。但 VC++ 2010 是可以的。

③ 非 const 对象的值不是常量表达式，不能作为非类型实参。

④ 字符串常量值不能作为非类型实参。因为在 C++中字符串常量值是一种具有内部链接性的对象，因此不能作为非类型实参。比如 template<char *N>void f(){}，则 f<"AAA">();是错误的。

⑤ 局部作用域中对象的地址不是常量表达式，不能作为非类型实参。

⑥ 具有内部链接性的指针不能作为非类型实参。比如 template<int *p>void f(){} const int p[3]={0};，假设 p 是在全局声明的，则 f<p>();是错误的。这里使用 const 声明的全局变量具有内部链接性。

⑦ 凡是声明的指针（不管是否是 const 指针），在 VC++ 2010 中都被认为是变量，不能作为非类型实参。

⑧ 数组中某个元素的地址不能作为非类型实参。比如 template<int *p>void f(){} int a[11]={0};，则 f<&a[0]>();错误。

⑨ 类中的成员变量不能作为非类型实参。比如 class A{public:int a;}; template<int N> void f(){}; A ma;，则 f<ma.a>();是错误的。

⑩ 存在类型转换时，不会考虑从子类到父类的转换，用户自定义的转换也不会被考虑。比如 class A{}; class B:public A{}; template<A* N> void f(){}; B mb;，则 f<&mb>();是错误的。

示例 15.2：非类型实参的限制

```
#include <iostream>
using namespace std;
template<int N>void f(){}              template<int *P>void f1(){}
template<const int *P>void f2(){}          template<const char *C> void f3(){}
int a=1; const int a1=2;     const int a2=a1; extern const int a3=a; int a4[4]={1,2,3,4};
namespace N{int c=1; const int c1=1; extern const int c2=3;}
class A{public: int d; static int e;}; int A::e=4; int A::*p=&A::d;
template<int A::*N> void f4(){}
extern const int *const p1=&a1;
void main(){int b=3; const int b1=4; const int b2=b; A ma; ma.d=1;
    //f<int>();     //错误，非类型实参应与非类型形参相匹配，非类型实参应是一个具体值
    f<3.3>();       //错误，浮点型值不能作为非类型实参。但 VC++ 2010 是可以的
    //f<a>();       //错误，全局变量 a 不是常量表达式
    f<sizeof(a)>(); //正确，sizeof 的结果是常量表达式
    f<a1>();        //正确，由常量表达式初始化的 const 变量是常量表达式
    f<a2>();        //正确，原因同上
    //f<a3>();      //错误，a3 不是由常量表达式初始化的 const 变量，所以不是常量表达式
    //f<b>();       //错误，局部变量 b 不是常量表达式
    f<b1>();        //正确，由常量表达式初始化的 const 变量是常量表达式
    f<N::c1>();     //正确，同上
    //f<b2>();      //错误，b2 不是由常量表达式初始化的 const 变量，所以不是常量表达式
    //f<ma.d>();    //错误，类中的成员变量不能作为非类型实参
    f1<0>();        //错误，空指针常量不能作为非类型实参。但 VC++ 2010 是可以的
    //f1<&b>();     //错误，局部作用域中对象的地址不是常量表达式，不能作为非类型实参
    //f1<&a4[1]>(); //错误，数组中某个元素的地址不能作为非类型实参
    f1<a4>();       //正确，数组名 a4 是具有外部链接性的指针常量
    f1<&a>();       //正确，全局作用域中任何对象的地址都是常量表达式
    f1<&N::c>();    //正确，名称空间作用域中任何对象的地址都是常量表达式
    //f1<&a1>();    //错误，由 const 限定的全局变量具有内部链接性，其地址不能作为非类型实参
    //f1<&N::c1>(); //错误，原因同上
```

```
    //f1<&a3>();     //错误，不能把 const int*类型的指针赋给 int* const 类型的对象
    //f1<&N::c2>();  //错误，原因同上
    f1<&A::e>();     //正确，静态类成员的地址可以作为非类型实参
    f2<&a3>();      //正确，a3 被使用 extern 声明为具有外部性质
    f2<&N::c2>();   //正确，原因同上
    //f2<p1>();     //错误，在 VC++ 2010 中，凡是声明的指针都被认为是变量而不是常量
    //f3<"AAA">(); //错误，在 C++中字符串常量值是一种具有内部链接性的对象，不能作为非类型实参的地址
    f4<&A::d>();    //正确，指向类成员的指针常量可以作为非类型实参
    //f4<p>();      //错误，指针 p 不是常量，它是指针变量
}
```

（9）非类型形参与非类型实参的类型之间允许存在一些转换，即非类型实参与非类型形参的类型并不要求完全一致，这些转换不会生成新的实例。所允许的转换如下。

① 允许从数组到指针、从函数到指针的转换。比如 template <int *a> class A{}; int b[1];，则 A<b> m;正确。注意 int b[1]应声明为全局变量。

② const 修饰符的转换。比如 template<const int *a> class A{}; int b; A<&b> m;，即从 int *到 const int *的转换。注意 int b 应声明为全局变量。

③ 提升转换。比如 template<int a> class A{}; const short b=2; A<b> m;，即从 short 到 int 的提升转换。

④ 整值转换（即从整值或枚举类型到其他整值类型的转换，不包括提升转换）。比如 template<unsigned int a> class A{}; A<3> m;，即从 int 到 unsigned int 的转换。

⑤ 从 0 到指针的转换是不允许的。但 VC++ 2010 允许。比如 template<int *p> class A{};，则 A<0> ma;在 VC++ 2010 上是正确的。

15.2.2　默认模板实参

（1）默认模板实参：只能为类模板形参提供默认值，而不能为函数模板形参提供默认值，这个默认值就是默认模板实参，其形式与函数形参类似。比如 template<class T1, class T2=int> class A{};，其中 int 就是 T2 的默认模板实参。再比如 template<class T=int>void f(){}错误，不能为函数模板形参设置默认值。

（2）默认模板实参可以是先前声明的模板形参，比如 template<class T1,class T2=T1>class A{};。

（3）类模板类型形参默认值的设定：如果有多个类型形参，在第一个设定了默认值的形参之后的所有模板形参都要设定默认值。比如 template<class T1=int, class T2>class A{};就是错误的，因为 T1 设定了默认值，而 T2 没有设定默认值。

（4）若同一个模板声明语句出现多次，则后续的默认模板实参可以被更早的模板声明指定。比如 template<class T1, class T2, class T3=int> class A;，有相同声明的 template<class T1, class

T2=int, class T3> class A;则是正确的，第二次声明指定了 T2 的默认模板实参，第一次声明指定了比 T2 更早的 T3 的默认模板实参。

（5）当同一模板声明语句被多次声明时，相同的模板形参不能被多次指定默认模板实参。比如 template<class T=int> class A;，则再次声明 template<class T=float>class A;是不允许的

（6）在使用具有默认模板实参的类模板时，可以为默认模板实参重新明确指定实参，指定的模板实参会代替默认模板实参；若不为默认模板实参重新指定实参，则使用默认模板实参。比如 template<class T1, class T2=int> class A{}; A<float> ma; A<float, char> mb;，其中 ma 的 T1 类型为 float，T2 使用默认实参 int；而 mb 的 T1 类型为 float，T2 类型为 char。

（7）在类模板外定义类中的成员时，template 后的形参表应省略默认模板实参，否则是错误的。比如 template<class T1, class T2=int>class A{public: void h();};，在类模板外定义的方法为 template<class T1,class T2> void A<T1,T2>::h(){}。

15.2.3 模板模板形/实参

模板模板形参：指的是模板形参仍然为模板的情形，但必须是类模板，而不能是函数模板，因为模板形参不能是函数。比如 template<template<class T> class TA> void f();，其中的 TA 就是模板模板形参，函数 f 的模板形参 TA 仍是一个模板，T 是模板模板形参 TA 的模板形参；而 template<template<class T> void g()> void f();是错误的。记住：模板模板形参仍然是模板形参，只是这个形参是模板而已。

（1）模板模板形参是一个类模板，这个类模板的声明方法与声明其他类模板相同，但不能是 struct 和 union。因为模板模板形参是一个类模板，因此其实参也必须是一个类模板，其实把模板模板形参叫作“类模板形参”更贴切。比如 template<template<class T> class TA> void f();，其中 TA 就是模板模板形参，它要求一个相对应的类模板作为实参，T 是模板模板形参 TA 的模板形参；再比如 template<class T>class A{};，则正确调用函数 f 的方式为 f<A> ()。注意：f<A<int> >()是错误的，因为 A<int>只能代表类模板 A 的一个实例，而不能代表整个类模板；template<template<class T> struct TA> void f();也是错误的，因为不能是 struct。

（2）模板模板形参的使用：只要在其作用域之中，就可以像使用类模板那样使用模板模板形参。比如 template<template<class T> class TA>void f(){TA<int> mta;};，表示创建一个模板模板形参 TA 的 int 类型实例对象 mta。

（3）模板模板形参的模板形参同样可以设置默认模板实参，比如 template<template<class T=int> class TA> void f();，其中 T=int 中的 int 就是模板模板形参的默认模板实参。

（4）模板模板形参的模板形参的名称，只能在模板模板形参的模板形参表中用来声明其他形参，在其他地方无法使用，因此模板模板形参的模板形参在未被使用时一般都是无名的。比

如 template<template<class T, T i>class TA> void f(){T m;}，其中 T m 是错误的，T i 的声明是正确的，因为模板模板形参 TA 的模板形参 T 只能在模板模板形参 TA 中用来声明其他形参。若模板模板形参的模板形参名未使用，则可省略，比如 template<template<class> class TA> void f(){}，正确。

（5）传递给模板模板形参的模板实参的模板形参拥有默认值时，模板实参的模板形参个数不会减少，由此可能会引发类型不匹配的错误。比如 template<template<class T> class TA> class A{}; template<class T1,class T2=int> class B{};，则 A<B> ma;是错误的，因为类 B 的模板形参 T2 的默认值并没有减少类模板 B 的模板形参个数，B 拥有两个模板形参 T1 和 T2，但类 A 的模板模板形参 TA 只要求拥有一个模板形参的类模板即可，因此出错。

15.3　模板实参推演与显式模板实参

15.3.1　基础

1. 实例化模板的方式（或者说指定模板实参的方式）

（1）实例化其实就是对模板形参的“值”进行确定的过程，因此指定模板实参，就是对模板实例化。

（2）指定模板实参的方法有两种，即模板实参推演和显式模板实参。

① 模板实参推演：指的是从函数实参推演出函数模板实参，从而确定模板类型形参。使用此方法确定模板类型形参时，只需像调用普通函数那样调用模板函数即可。这里要注意两个重点——只能从函数实参进行推演和只能推演出类型形参。比如 template<class T> void f(T a){};，则调用 f(2)就可以根据函数形参和模板形参 T 的关系推演（推导）出模板实参的类型为 int，从而确定函数模板类型形参 T 的类型为 int。

② 显式模板实参：就是明确指定模板实参。

函数模板的形式为：函数名<模板实参列表>(函数实参,...);，比如 f<int,float>(...);。

类模板的形式为：类名<模板实参列表>对象名;，比如 A<int,float> ma;。

可见显式模板实参就是在使用模板时的尖括号中明确指定模板实参，从而确定相应的模板形参类型。这是模板中最基本的指定实参的方法。比如 template<class T1, class T2>void f(T1 a, T2 b){};，则 f<int,float>(2);中的<int,float>就是显式（明确）指定的模板实参，根据模板实参确定模板形参 T1 的类型为 int，T2 的类型为 float。再比如 template<class T1, class T2> class A{};，则显式模板实参的示例为 A<int, float> ma;。

2. 模板实参推演与显式模板实参应用的范围及基本规则

（1）类模板不能使用模板实参推演，因为模板实参是由函数实参推导出来的，而类模板无函数实参（此时只能使用显式模板实参）。比如 template<class T> class A{};，则 A<3>ma 是错误的，此处不能试图通过 3 的类型推导出 T 的类型为 int。

（2）显式模板实参使用在模板实参推演存在二义性或模板实参推演不能进行的情况下，比如类模板就必须使用显式模板实参。

（3）模板实参推演只适用于模板形参与函数形参之间有一定的关联时，否则就无法使用模板实参推演，只能使用显式模板实参。比如 template<class T>void g(int a){};，因为函数 g 的形参 a 与模板形参 T 之间无任何关联，因此在调用函数时无法使用模板实参推演出 T 的类型，即 g(3)错误。要确定 T 的类型只能使用显式模板实参的方式，比如 g<int>(2);。

（4）构造函数不能使用模板实参推演，除了构造函数本身也是模板这种情况。比如 template<class T> class A{public: A(T){} template<class T1> A(int, T1){}};，则 A ma(3);是错误的；但 A<int>ma(4.4);是正确的，此时 T 的类型是由尖括号中的 int 明确指定的，即 T=int;；A<int> ma1(4, 6.5);也是正确的，构造函数 A(int, T1);中的 T1 使用模板实参推演出为 double 类型。

（5）使用模板实参推演时，为同一模板形参推演出的类型应一致，否则会出现二义性错误。比如 template<class T> void f(T x, T y){};，则 f(2, 3.3)是错误的，因为调用 f(2,3.3)推演出的第一个模板实参的类型为 int，第二个为 double，为同一模板形参 T 推演出两个不一致的类型，所以出现二义性错误。

（6）默认函数实参不能用来进行模板实参推演。比如 template<class T> void f(T a=3){};，则 f()是错误的，因为试图使用默认函数实参 3 来推导出 T 的类型。

（7）非类型形参也可以使用模板实参推导出来，但前提是必须在函数形参之中建立起与非类型形参之间的关系。比如 template<int N> void f(int (*p)[N]){} int (*pp)[4]=0;，则 f(pp)正确，推导出 N=4。注意：指针 pp 是作为函数实参而非模板实参传递的，因此 pp 不需要遵守非类型模板实参的限制。另外，数组作为函数形参会被自动转换为指针，因此不能在函数形参中使用一维数组来建立与模板形参的关系。比如 template<int N>void f(int a[N]){}; int c[4];，则 f(c)错误，因为函数形参 int a[N];会被转换为 int *a;，实际上在函数形参之中并未建立起与模板形参 N 的关系。

（8）显式模板实参基本规则：显式指定模板实参时总是从左到右进行模板实参与模板形参的匹配。比如 template<class T1, class T2, class T3> T2 f(T3, T1); f<int, float>(3.3, 4);，则 T1=int; T2=float; T3=double; 显式模板实参顺序为<int, float>，模板形参顺序为<class T1, class T2, class T3>，函数形参顺序为(T3, T1)，因此 T1=int; T2=float;，而 T3 在显式模板实参中并没有被指定，因此 T3 由函数实参 3.3 推导出为 double 类型。

（9）显式模板实参的“值”应与模板形参所要求的“值”相匹配。比如 template<class T> void f(){};，则 f<3>()错误，因为模板形参 T 要求一个类型值，而模板实参传递一个整型值给 T。

（10）被确定的模板实参类型，应保证在模板内部是合法的，否则就会发生错误。比如 template<class T> void f(T x, T y){x+y;}; class A{}; A ma1, ma2;，则调用 f(ma1, ma2)或 f<A>(ma1, ma2)都是错误的，其中 x+y 是错误的，因为类类型 A 未定义相关的“+”操作符。

（11）在使用显式模板实参时，可通过隐式类型转换把函数实参转换为显式指定的模板实参类型；而使用模板实参推演时，则除了所允许的转换之外，是不会进行类型转换的。比如 template<class T> void f(T a, T b){};，则调用 f<int>(2.2, 3)会把函数实参 2.2 转换为显式指定的模板实参类型 int；而调用 f(2.2,3)是错误的，这里通过模板实参推演来确定模板形参 T 的类型，因为函数实参 2.2 和 3 的类型不一致，导致推导出的模板形参 T 的类型不一致。在使用模板实参推演时，不会通过隐式类型转换把整数 3 的 int 类型转换为 2.2 的 double 类型，或进行相反的转换。

（12）在对函数模板使用显式模板实参时，可省略能通过函数实参推演出的尾部的模板实参，此方法不适用于类模板（因为类模板不能进行模板实参推演）。比如 template<class T1, class T2, class T3> T1 h(T2 a, T3 b){};，则 h<int>(3, 3.4)省略了最后两个模板形参 T2 和 T3 对应的模板实参，T2 和 T3 可由调用时的函数实参 3 和 3.4 隐式确定为 int 和 double 类型，T1 被显式确定为 int 类型；而 h<int, , double><2,3.4>错误，只能省略尾部的实参，此处中间的两个逗号试图省略模板形参 T2 对应的模板实参。

（13）零初始化：使用“内置类型 变量名=内置类型()”形式的语法。该方法可以为在模板内部声明的类型形参变量设置默认的初始值，即为 T x 这种形式声明的变量 x 设置初始值。比如 template<class T> void f(){T x=T();}，若 T 为 int 类型，则 x 的默认值为 0；若 T 为类类型，则默认为使用该类的默认构造函数初始化 x。对于在类模板中使用的类型形参变量，还可以在类模板的成员初始化列表中使用类似的语法，如 template<class T>class A{T x; A():x(){}}。

（14）对于函数模板而言不存在 h(int,int)这样的调用，不能在函数调用的参数中指定模板形参的类型，而应通过实参推演或显式模板实参来调用函数模板，即只能进行 h(2,3)或 int a, b; h(a,b)或 h<int,int>(2,3)这样的调用。

（15）在进行模板实参推演的过程中，不会使用单形参构造函数进行隐式转换。比如：

```
template<class T>class A{public:A(){}  A(int){cout<<typeid(T).name()<<endl;} };
template<class T> void f(A<T>){cout<<typeid(T).name()<<endl;}
void main(){  A<int> ma;
   //f(3);          //错误，使用模板实参推演时，不会使用类 A 的单形参构造函数把整型转换为类型 A<int>
                    //从而推导出函数 f 的模板形参 T 的类型为 int
   f<float>(3); }  //正确，这是显式模板实参，首先 f 被实例化为 f<float>，A 也被实例化为 A<float>
                    //此时会调用类 A 的单形参构造函数把整型值 3 转换为 A<float>类型
```

15.3.2 模板实参推演

1. 推演上下文构件简介

（1）复杂类型声明是由基本的构件（比如[], *, (), template, ::等）组成的。

（2）推演上下文（构件）的概念：首先是构成一个复杂声明的构件；其次是对正被推演的模板形参 T 有影响（或相关联）的构件。可见，推演上下文就是一种构件，大多数构件都是推演上下文（构件）。上下文一般指的是上下文相关联，即前后相关联、上下相关联的意思。读者别去纠结推演上下文的字面意思，要注重理解对这个词语的定义。

（3）非推演上下文（构件）：受限名称不是推演上下文构件，比如 A<T>::N，其中的 N 不是推演上下文构件，因为对模板形参 T 的类型推演没有影响（或关联）。受限名称详见下一节。

（4）一般来说，非推演上下文构件不会参与模板实参推演。

2. 模板实参推演原理

（1）注意：模板实参推演只适用于函数模板。

（2）模板实参推演的基本方法：为要推导的模板创建[实参 A, 形参 P]对。

① 实参 A 表示调用函数时实际函数实参的类型。

② 形参 P 表示在声明或定义函数模板时函数形参的类型。

③ 模板实参推演就是通过 A 推导出 P 中的模板形参类型。

④ 比如 template<class T> void f(T* ma){} int a[3]={0}; f(a);，则 A=a=int[3]，而 P=ma=T*，即[A,P]=[int[3], T *]；模板实参推演就是通过 int[3]推导出模板形参 T 的类型。

（3）产生[A,P]对的两种特殊情况：以下两种情况，A 不是产生自调用时的函数实参，或 P 不是产生自声明或定义函数时的函数形参类型。

① 当对函数模板进行取址时，这时 A 产生自被初始化或被赋值的函数指针的类型，而 P 仍然是声明或定义函数模板时的函数形参的类型。比如 template<class T> void f(T, T){}; void (*p)(int, int)=&f;，则 A=(int, int); P=(T,T)。

② 当类中的转换运算符函数是模板时，这时 A 是需要被转换的目标类型，P 是转换运算符函数的返回类型。比如 class A{public: template<class T> operator T*(){int *p=0; return p;}}; void f(int a[3]){}; A ma;，则对于 f(ma); 此时 A 为需要被转换的目标类型 int[3]，有 A=a=int[3]=int *;，P 为转换运算符函数的返回类型，即 P=T*，因此 T 的类型被推导出为 int。

（4）模板实参推演的基本规则。

① 若参数 P 的类型为引用（假设为 T&），则 P 就是被引用的类型（即 T），A 仍然是调用函数时实参的类型，且 A 的外围的 const 和 volatile 限定词不会被忽略。

② 若 P 为非引用类型（假设为 T），则 P 是函数形参的类型；若 A 是数组或函数类型时，

则在推演之前会被自动转换为相应的指针类型，同时会忽略外围的 const 和 volatile 限定词（在此种情形下 VC++ 2010 不会忽略 const 限定词）。发生指针转换的原因是当数组作为函数的形参时，其真实的类型是把数组转换为相应指针类型后的指针类型，详见第 7 章。

- 由以上规则可以推出，若 P 是引用类型（假设为 T&），则当 A 是数组或函数类型时，不会被自动转换为相应的指针类型。
- 转换发生在推演之前。
- 若 P 本身也是数组或函数类型且是非引用的，则在推演之前 P 本身也会进行从数组到指针的转换。
- 二维及多维数组除第一维会被转换为指针之外，第二维及其以后的数组类型不会再次被转换为指针。比如 int a[3][4];转换为指针之后的类型是 int (*)[4];，即“指向有 4 个元素的一维数组的指针”。

示例如下：

```
template<class T> void f(T& p){} int a[3][4]={0};} f(a);
```

说明：推导出 T=int[3][4];，此时 P=T&; A=int[3][4];。因为 P 是引用类型，因此其中的 A 未被转换为相应的指针类型。

```
template<class T> void f(T* p){} int a[3][4]={0}; f(a);
```

说明：推导出 T=int[4]，此时 A=int [3][4];P=T*;。因为 P 为指针，因此在推演之前把 int[3][4]转换为 int (*)[4];，推导出 T=in[4]；若在推演之前不把 A 转换为相应的指针类型 int (*)[4]，则无法根据 int[3][4]推导出 T*中 T 的类型。由此可以看到二维数组 int a[3][4]的第二维不会再次被转换为指针类型，即 int a[3][4]不会被转换为 int **类型。

③ 对于由构件组成的复杂类型，则从最内层的构件开始逐一向外进行模板类型推导。

④ 当把字符串传递给引用时会被转换为数组，而传递给非引用时会被转换为指针，即字符串的类型相当于数组。此规则在应用时应注意指针和数组是两个不同的类型，而且数组的类型还受到数组维数及大小的影响。此规则只适用于函数模板，并且使用模板实参推演的情况，对于类模板和显式模板实参并不适用。比如 template<class T>void f(T a){}; template<class T>void g(T1& a){};，调用 f("AAA")，则 T 的类型为 const char *；调用 g("AAA")，则 T1 的类型为 const char[4]。

⑤ 若 P 通过 A 对模板形参 T 的类型进行成功推导之后，函数形参就相当于使用 typedef 把推导出的 T 类型重命名为 T 后对函数形参的声明，这时对函数形参类型的分析方法就与使用 typedef 重命名的类型创建对象时的类型分析方法相同，在模板使用 T 创建对象时的类型分析方法也是相同的。即：标识符首先与已存在的 const 和声明符组合成一个新的声明符，然后将这个新的声明符与使用 typedef 重命名的新类型结合，即使用 typedef 新命名的类型具有低优先级，详见第 3 章。比如 template<class T> void f(const T* s){};，若 T 被推演为 int 类型，则函数形参

s 与 typedef int* P; const P* x;中 x 的声明一样，s 的类型分析方法也与 x 的类型分析方法相同。

示例如下：

```
template<class T> void f(const T s){} int a=1; int b[3][4]={0}; f(&a); f(b);
```

分析 f(&a)：P=const T s; A=&a=int*;，推导出 T=int *; s=int * const;，很明显可以推导出 T=int *，因此 const T s;中的 s 应首先与 const 结合，组合成 const s，然后再与 T 的类型 int*结合，组合成 int *const s;。

分析 f(b)：P=const T s; A=b=int[3][4]=int(*)[4];，推导出 T=int (*)[4]; s=int (*const)[4];，很明显可以推导出 T=int(*)[4]，因此 const T s 中的 s 应首先与 const 结合，组合成 const s，然后再与 T 的类型结合，组合成 int (*const s)[4];。

```
template<class T> void f(const T* s){}; int a=1; int *p=&a; f(&p);
```

分析：P=const T*; A=int **;，推导出 T=int*;，s=int *const *;，很明显可以推导出 T=int *;，因此 const T* s;中的 s 应首先与 const*结合，组合成 const * s，然后再与 T 的类型 int*结合，组合成 int *const *s;。若是 f(&a)，则 P=cons T*; A=int *;，推导出 T=int，此时 const T* s;中的 s 首先与 const *结合，组合成 const *s, 然后再与 T 的类型 int 结合，组合成 int cons * s;。

```
template<class T> void f(T &s){T *x; }; int a[3]={0}; f(a);
```

分析：T 为引用，推导出 T=int [3];。f 中声明的变量 x 的类型应按 typedef 的方式进行分析，比如 x 首先与*结合成*x，然后再与 T 的类型结合成 int (*x)[3]，因此 x 的类型为 int (*)[3]。

示例 15.3：模板实参推演原理

```
template<class T> void f(T (&a)[3]){}; int *p[3]={0}; f(p);
```

① 分析：P=a=T (&)[3]=T [3]; A=p=int *[3];，推导出 T=int *;。

② 类型转换分析：P=T (&)[3]是引用类型，所以 A 不会发生从数组到指针的转换。

③ 类型推导分析：首先从最内层开始分析，P 的类型为[3]，与 A 最内层的类型[3]相同，因此 T 被推导出为 T=int *。

```
template<class T> void f(T&){}; int *p[3]={0}; f(p);
```

① 分析：P=T&=T; A=p=int *[3];，推导出 T=int *[3];。

② 类型转换分析：P 是引用类型，所以 A 不会发生从数组到指针的转换。

③ 类型推导分析：略（比较明显）。

```
template<class T> void f(T*){} int *p[3]={0}; f(p);
```

① 分析：A=p=int *[3]=int **; P=T*;，推导出 T=int *;。

② 类型转换分析：P=T*是非引用类型，因此会把 A 的数组类型转换为指针类型，即 A=int *[3]会被转换为 int **。注意 int *[3]，首先是与[3]结合的，因此 int*[3]的类型为数组，只是数组存储的内容比较复杂。

③ 类型推导分析：略（比较明显）。

```
template<class T> void f(T*){} int (*p)[3]={0}; f(p);
```

① 分析：P=T*; A=p=int (*)[3];，推导出 T=int [3];。

② 类型转换分析：此处不会发生从数组到指针的转换，因为 A=int (*)[3];的类型为指针并非数组。

③ 类型推导分析：P 最内层的类型为指针*，与 A 最内层的类型相同，因此 T 的类型就应是 A 剩余的类型 int [3]。

```
template<class T> void f(T*){} int a[3][4]={0}; int (*p)[4]=a; f(a)或f(p);
```

① 分析 f(a)：P=T*; A=a=int[3][4]=int (*)[4];，推导出 T=int [4];。

② 类型转换分析：P=T*是非引用类型，因此会把 A 的数组类型转换为相应的指针类型，即将 A=int[3][4]转换为 int (*)[4]。

③ 类型推导分析：P 最内层的类型为指针，与 A 最内层的指针类型匹配，而 A 剩余的类型为 int [4]，因此推导出 T=int [4]，而不是 int *。

④ 分析 f(p)：A=p=int(*)[4]; P=T*;，推导出 T=int[4];。因为 A=int (*)[4];是一个指针而不是数组，因此不会发生从数组到指针的转换。其推导过程与 f(a)相似，最后推导出 T=int[4];。

```
template<class T> void f(T&){}; template<class T> void g(T){}
const int a=2; int b=3; int c[4]={0};
```

① 分析 f(a)：推导出 T=const int;。因为 P 是引用类型，当匹配到引用类型时 const 不会被忽略。

② 分析 f(b)：推导出 T=int;。原因很明显。

③ 分析 f(c)：推导出 T=int [4];。此处 A=int[4]; P=T&=T;，虽然 A 是数组类型，但 P 是引用类型，因此 A 不会发生从数组到指针的转换。

④ 分析 g(a)：推导出 T=int;。当匹配到非引用类型时，const 会被忽略。

⑤ 分析 g(b)：推导出 T=int;。

⑥ 分析 g(c)：推导出 T=int*; A=int [4]=int *; P=T;。这里 P 是非引用类型，因此 A 会从数组转换为指针，即 int [4]会被转换为 int*。

⑦ 分析 f("ddd")：推导出 T=const char[4]; A="ddd"=const char [4];。因为 P=T&=T 是引用类型，A 不会发生从数组到指针的转换。

⑧ 分析 g("ddd")：推导出 T=char *;，此时 A="ddd"=const char [4]= const char *;。因为 P=T 是非引用类型，A 会发生从数组到指针的转换，并且 const 限定词被忽略。但 VC ++2010 不会忽略 const 限定词，从而推导出 T=const char *。

```
template<class T, int N> void f(T s[N]){}; int a[4]={0}; f(a);  //错误,推演失败
```

① 分析：P=s=T [N]=T*; A=a=int [4]=int *;，可推导出 T=int;，但无法推导出 N 的值。

② 类型转换分析。

- 模板形参 A 的类型转换分析——P=T s[N]，其中 P 是非引用类型，因此 A 会发生从数组到指针的转换，这里会把 A=int[4]转换为指针，即 A=int[4]=int *。

- 函数形参 P 的类型转换分析——P=T s[N];，其中 P 是非引用类型，且 P 的类型是一个数组，而且 P 是函数的形参，当数组作为函数的形参时，其真实的类型是把数组转换为相应指针类型后的指针类型。因此，此处的数组 P 会被转换为指针，即 P=T s[N]被转换为 P=T s*=T*。

③ 类型推导分析：A=int*; P=T*;，推导出 T=int，但无法推导出 N 的值。

```
template<class T, int N> void f(T (&s)[N]){}; int a[4]={0}; f(a);
```

① 分析：P=s=T (&)[N]=T [N]; A=a=int [4];，推导出 T=int，N=4。

② 类型转换分析：P=T (&)[N]，可见 P 是引用类型，因此既不会将模板实参 A 的数组转换为指针，也不会将函数形参 P 的数组转换为指针。

③ 类型推导分析：A=int[4]; P=T[N];，T 与 int 相对应，N 与 4 相对应，推导出 T=int; N=4。

```
template<class T> void f(T,T){} int a[3]={0}; int b[4]={0}; f(a,b);
```

分析：推导出 T=int*，因为 P=T 为非引用类型，所以 A=(a,b)中的 a 和 b 的类型都被转换为 int*。

```
template<class T> void f(T&,T&){} int a[3]={0}; int b[4]={0};f(a,b); //错误
```

分析：因为 P=T&为引用类型，A=(a,b)中的 a 和 b 的类型不会被转换为指针，因此为第一个 T&推导出 T=int[3]，为第二个 T&推导出 T=int[4]，但 int[3]和 int[4]是不同的类型，从而产生错误。

3. 模板实参推演所允许的转换

（1）在模板实参推演的过程中，有时函数实参的类型并不会与函数形参的类型完全匹配，这时编译器允许进行从函数实参到函数形参的转换。

（2）以下转换都是指将调用时的函数实参转换为相应的函数形参类型。

① 左值转换（包括从左值到右值、从数组到指针和从函数到指针的转换）。

② 限定修饰符转换：只适用于指针或引用。若函数形参有 const 限定词，函数实参没有 const 限定词，在转换前就把 const 限定词加到函数实参上。比如 template<class T> void h(const T* a){};int b=3; h(&b);，在进行模板实参推演之前，把&b 转换成 const int *，因此类型形参 T 被推导出为 int 类型。

③ 调用时实参到父类的转换（该父类由一个类模板实例化而来）：函数形参可以是调用函数时函数实参所属类型的父类；或者函数实参是指向某类型的指针，则函数形参可以是指向函数实参所指所属类型父类的指针。这里有如下三个限制条件。

- 函数模板的函数形参必须是一个类模板。（条件 1）
- 函数形参应是调用该函数时函数实参的父类。（条件 2）
- 函数形参的实例必须经过函数实参实例化而来。（条件 3）

示例 15.4：调用时实参到父类的转换

```
template<class T1>class C{};
template<class T1> class B:public C<T1>{};
//template<class T1> class B:public C<float>{};
template<class T2> void f(C<T2>& m){}
B<int> n;   f(n);
```

分析 f(n)：

① P=m=C<T2> &=C<T2>; A=n=B<int>=C<int>;，推导出 T2=int。

② 类型转换分析：执行实参到父类的转换，因此在推演之前，把 A=B<int>转换为由 B<int>实例化而来的父类类型实例 C<int>，然后再由函数形参 C<T2>根据实参 C<int>进行推演，推导出 T2=int。

③ 对实参到父类转换的三个条件的分析。

- 函数形参 P=C<T2>是一个类模板。满足条件 1。
- 函数形参 C<T2>是调用该函数时函数实参 B<T1>的父类。满足条件 2。
- 函数形参 C<T2>的实例由函数实参 B<T1>实例化而来。比如创建 B<int>类型的对象时，会根据 B 的继承方式 B:C<T1>创建 B<int>的父类实例 C<int>，父类实例 C<int>是由子类 B 实例化而来的。满足条件 3。
- 遵守实参到父类的转换规则，即把实参 B<int>转换为由 B<int>实例化而来的父类 C<T1>类型的实例 C<int>，即 A=B<int>=C<int>。

④ 注意：转换时的父类类型是子类 B<T1>被实例化时而产生的父类实例；若使用注释掉的继承方式，因为 B 使用 int 实例化时产生的父类类型是 C<float>实例，因此会把 B<int>转换为 C<float>类型，从而推导出 T=float。

4. 含有 const 限定词时模板实参推演总结

（1）对于 P=T，则 A 的 const 会被忽略。若参数 P 为非引用类型，则 A 外围的 const 限定词会被忽略。比如 template<class T> void f(T a){a=1;} const int b=1;，则 f(b)正确，可见 a 的类型未被 const 限定。

（2）对于 P=T*，则 A 的 const 会被忽略。但在 VC++ 2010 中，A 的 const 不会被忽略。比如 template<class T> void f(T *a){*a=1;} const int b=1;，则 f(&b)，对于*a=1 发生错误，因为*a 是常量，可见 a 的类型为 const int *，其 const 未被忽略。

（3）对于 P=T&，则 A 的 const 不会被忽略。若参数 P 为引用类型，则 A 外围的 const 和 volatile 限定词不会被忽略。

（4）对于 P=const T，则 A 的 const 是多余的，会被忽略。注意，对于指针就不是多余的了。

（5）对于 P=const T*;，则 A 无 const 时会被加上 const。

（6）对于 P=const T&;，则 A 无 const 时会被加上 const。

（7）对于 P 含有 const 的情形，最好按照 typedef 的规则进行分析。

（8）注意：VC++使用 typeid(T).name()输出类型时，对于 const int 这样的类型，只会输出 int，而忽略了 const。因此在编写示例时，最好是通过 T a 中 a 的值是否能被修改来验证 const 是否被忽略。

15.3.3 显式模板实参

（1）在不能使用模板实参推演的情况下，就只能明确指定模板实参了。

（2）用法 1：对于类模板只能使用显式模板实参，不能使用模板实参推演。因为类没有函数形参，所以无法通过函数实参推导出模板形参的“值”。比如 template<class M>class A{};，则 A<2> ma;错误，正确形式为 A<int> ma;。

（3）用法 2：若模板形参与函数形参之间没有任何关联时，就无法使用模板实参推演，只能使用显式模板实参。比如 template<class T>void f(int i){};，则 f(3)和 f(2.2)都是错误的，因为无法根据函数实参 3 或 2.2 推演出类型形参 T 的类型，正确形式为 f<int>f(2.2);。

（4）用法 3：使用在同一个模板形参类型不一致的情况下。这时通过模板实参推演不能为同一个模板形参指定两种不同的类型。比如 template<class T> void h(T a, T b){}，则 h(2, 3.2)的调用会出错，因为使用模板实参推演之后的结果是两个实参的类型不一致，第一个为 int 类型，第二个为 double 类型。而调用 h<double>(2,3.2)就是正确的，这里把模板形参显式实例化为 double 类型，这样就允许进行标准的隐式类型转换，即把第一个 int 类型的参数转换为 double 类型的参数。

（5）用法 4：使用在函数模板的返回类型中。若函数的返回类型没有出现在函数形参表中，则无法通过模板实参推演推导出函数的返回类型。比如 template<class T1, class T2, class T3> T1 h(T2 a, T3 b){}，则 int a=h(2,3)和 h(2,4)就会出现模板形参 T1 无法推导的情况。而通过显式模板实参就能解决这个问题。比如 h<int, int, int>(2,3)，即把模板形参 T1 显式指定为 int 类型，T2 和 T3 也指定为 int 类型。

（6）用法 5：使用在模板函数的参数中没有出现模板形参的情况下。若函数没有形参，则是无法通过模板实参推演的。比如 template<class T>void h(){};，则 h()错误，因为无法推导出 T 的类型；而 h<int>()正确，把函数 h 的模板形参 T 指定为 int 类型。

15.4　名称的识别与依赖实参的查询

15.4.1　依赖实参的查询（ADL）

1．名称的分类

（1）受限名称：是指明确使用作用域解析运算符（::）或成员访问运算符（.或->）进行限定的名称。比如 this->a，则 a 是受限名称；若在类内部省略 this 而直接写成员名称 a，则 a 不是受限名称。

（2）依赖型或受控名称：是指以某种方式依赖（或受控于）模板形参的名称。若模板形参的类型已知（假设为 int），则该名称就不是依赖型名称，因为这时名称不依赖模板形参了。具体分类如下。

① 任何含有模板形参的受限或非受限名称都是受控（依赖型）名称。

② 使用成员访问运算符（.或->）限定的受限名称，若运算符左边的表达式类型依赖模板形参，则该名称是依赖型名称。

③ 对于成员名称，比如 this->a 这样的名称，若出现在模板中，则 a 是依赖型名称。

④ 对于调用 f(a,b)，若其中的某个形参的类型依赖模板形参的类型，则名称 f 是依赖型名称。

示例 15.5：受控名称（依赖型名称）

```
template<class T> class A{public:
   typedef int X;
   int i;                    //i 为非受控名称，因为 i 不依赖模板形参 T
   void h(){this->i=3;      //i 为受控名称,因为 this 的类型需要等到模板形参 T 确定之后才能确定,即 this
                            //依赖模板形参 T
   i=4; } };                //i 为非受控名称
template<class T1> void f(){
   T1 b=T1();        //b 为受控名称，因为 b 的类型必须等到 T1 确定之后才能确定，即 b 依赖模板形参 T1
   int c=2;          //c 为非受控名称
   g(b);             //g 为受控名称，因为 g 的实参 b 依赖模板形参 T1
   g(c);             //g 为非受控名称
   T1::X mx;         //X 和 mx 都是受控名称，因为 X 和 mx 都必须等到模板形参 T1 的类型确定之后才能确定
   A<T1> ma;         //ma 是受控名称，同样 ma 也要等到模板形参 T1 确定之后才能确定
   ma.i=4;           //i 是受控名称，因为 ma 是受控名称
   A<int> mb; }      //mb 是非受控名称，因为 mb 不依赖任何模板形参
```

2．名称查询规则

（1）受限名称查询规则：查询受限名称是在限定该名称的构件所决定的作用域内进行的。若该作用域是一个类，则其父类也会被查询，但该类的外围作用域不会被查询。详见 8.3.2 节。

（2）非受限名称查询规则：

① 有两种方法用于查询非受限名称，即常规查询和依赖实参的查询（ADL）。

② 常规查询规则：由内到外在所有的外围作用域中逐层进行查询。前面章节中已讲解过。

③ 依赖实参的查询规则：该规则比较复杂，详见后文。

3. ADL 的启动条件

（1）依赖实参的查询（ADL）：指的是在调用函数时，程序会按照一定规则去查询与实参相关联的类型所在名称空间中的函数名称。比如 namespace N{class A{}; void f(A){}}; N::A ma，则 f(ma)是正确的，此处会去查询与实参 ma 相关联的类型 A 所在名称空间 N 中的函数名称 f。再比如 namespace N{class A{}; void f(A){}}; void f(N::A m){} N::A ma;，则 f(ma)的调用具有二义性。

（2）ADL 一般适用于在进行函数调用操作时查找函数名称，比如调用 f(...)时查找名称 f。

（3）ADL 只适用于查询非受限的非成员函数名称。比如 class A{public: class B{}; void g(B mb){}}; A::B mab;，则 g(mab)调用是错误的，因为 g 是成员函数，不适用于 ADL 机制。再比如 namespace N{class B{}; void g(B mab){}}; A::B mab;，则 g(mab)是正确的。

（4）若在调用函数时，把函数名称用小括号括起来，则不会启动 ADL，即形如 (f)(...) 这样的调用不会启动 ADL。比如 namespace N{class B{}; void g(B m){}}; N::B mnb;，则 g(mnb)是正确的，而(g)(mnb)是错误的。再比如 namespace N{class B{}; void g(B m){}}; void g(N::B m){} N::B mnb;，则(g)(mnb)调用全局作用域中的函数 g，而 g(mnb)的调用则具有二义性。

（5）ADL 与显式模板实参：使用显式模板实参时无法启动 ADL。比如 namespace N{class A{}; template<class T>void f(A*){}} N::A *p=0;，则 f<int>(p)这样的调用是错误的。这里首先要解析的是<int>，即确认<int>是一个模板实参列表，或者首先要确认 f 是一个函数模板才能确定<int>是一个模板实参列表。但是函数 f 定义于名称空间 N 之内，无法被看到，所以程序无法确认<int>是一个模板实参列表，从而把尖括号解释成了大于、小于符号（即编译器并未把名称 f 解释为模板），这样程序根本还没有对函数 f 的实参 p 进行检查时便已出错了，因此 ADL 未启动。但 f(p)调用是正确的，因为这里没有对尖括号的解析，而是直接对实参 p 进行检查。

（6）若通过常规查询可以找到一个成员函数名称或类型名称，则不启动 ADL。

示例 15.6：ADL 的启动条件

```
#include <iostream>
using namespace std;
namespace N{class B{}; void f(B){cout<<"B"<<endl;}
    void h(B){cout<<"C"<<endl;} template<class T>void g(B){}}
class A{public: class B{};  void f(B){cout<<"A"<<endl;}};
void h(N::B){cout<<"D"<<endl;}
void main(){  A::B mab;      N::B mab1;
   //f(mab);        //错误，不启动ADL，因为f是类A的成员函数
```

```
    f(mab1);          //启动 ADL，调用名称空间 N 中的 f(B)，ADL 机制会使程序去查询与实参 mab1 相关联的
                      //类型 B 所在名称空间 N 中的函数名称 f
    //(f)(mab1);      //错误，不启动 ADL，因为函数名称被小括号括起来了
     (h)(mab1);       //不启动 ADL，因此调用全局函数 h(N::B)，原因同上
    //h(mab1);        //错误，启动 ADL，调用出现二义性错误，即全局的 h(N::B)和名称空间 N 中的 h(B)匹
                      //配得一样好
    //g<int>(mab1);//错误，未能启动 ADL。使用显式模板实参时首先应分析尖括号是否代表一个模板。因为
                      //在未看到实参 mab1 之前不会启动 ADL 进行查询，即在分析尖括号之前无法知道函数名
                      //是否代表一个模板，所以尖括号会被解释成大于小于符号，从而程序出错
}
```

4．ADL 规则

（1）ADL 基本规则：利用 ADL 机制会查询调用实参的关联名称空间所组成的集合和关联类所组成的集合中的函数名称。由关联的名称空间组成的集合称为关联名称空间集合，由关联类组成的集合称为关联类集合。比如 namespace N{class A{}; class B{}; void f(A, B){}} N::A ma; N::B mb;，则对于 f(ma,mb)，实参 ma, mb 所关联的类组成的集合是{A,B}，其关联的名称空间组成的集合为{N}。

（2）查询关联类和关联名称空间中的名称规则如下。

① 关联类中的函数若是一个成员函数，则不会启动 ADL，因此该成员函数不会被考虑；若关联类中的函数是一个友元函数，则该友元函数会被考虑。比如 class A{public:void f(A*){} friend void f(A*){}}; A ma;，则 f(&ma)会调用友元函数 f(A*)，而对于成员函数 f(A*)不会启动 ADL 机制，无法找到该名称。

② 查找关联类中的函数名称规则：通过 ADL 机制不但会查询类内部所声明的函数名称，而且还会查询定义该类的外围作用域中的函数名称。注意：成员函数除外，因为成员函数不会启动 ADL 机制。比如 namespace N{class A{public: void f(A*){} friend void f(A*){}}; void f(const A*){}} N::A ma;，则 f(&ma)调用友元函数 f(A*)，此处查询到的函数名称 f 包括类 A 中定义的友元函数 f(A*)和定义类 A 的外围作用域（即名称空间 N）中的函数名称 f(const A*)，但友元函数 f(A*)匹配得更好，因此调用 f(A*)。注意成员函数 f(A*)不包括在此范围内。

③ 查找关联名称空间中的函数名称规则：对于关联名称空间，利用 ADL 机制只会查询该名称空间内部所声明的函数名称，而不会查询该名称空间的外围作用域中定义的函数名称。比如 namespace N{namespace M{ class A{}; }void f(M::A){}} N::M::A ma;，则 f(ma)调用是错误的，对于名称空间 N 中定义的函数 f 是不可见的，因为它位于 ma 所属类 A 所在名称空间 M 的作用域之外。再比如 namespace N{class A{}; namespace M{void f(A){}}} N::A ma;，则 f(ma)错误，找不到函数 f，因为 f 未定义在名称空间 N 之内（注：在外围作用域中是看不到嵌套作用域中的名称的）。

（3）注意：在使用 ADL 时，会忽略在名称空间内部使用的 using 指令。比如 namespace N{ void

f(){}}namespace M{using namespace N; class A{};void f(A){}}; M::A ma;，则对于 f(ma)，因为在 M 中的 using 指令被忽略了，因此 f(ma)只会考虑名称空间 M 中的函数 f(A)，而 N 中的 f()不会被考虑。

5. 创建关联名称空间集合和关联类集合的规则

（1）若实参是内置类型时，则关联名称空间集合和关联类集合是空集合。比如 namespace N{int a=1;void f(int){}};，则 f(N::a)是错误的，这里找不到名称 f，因为 N::a 是内置类型的，其关联的名称空间集合为空。

（2）若实参是指针和数组类型时，则该集合是所引用类型的关联名称空间集合和关联类集合。所引用类型对于指针是指指针所指向的类型，对于数组则是指数组元素的类型。注意：如果所引用类型是内置类型，则该集合为空。

（3）若实参是枚举时，则关联名称空间指的是声明枚举所在的名称空间。比如 namespace N{enum E{a}; void f(E){}}，则 f(N::a)会调用 N 中的函数 f。

（4）若实参是指向类 X 的成员的指针类型时，则该集合包括与 X 及与被指向的成员相关联的类和该关联类所在的名称空间。若这个指针指向的成员是个函数，则还应包括与此函数的形参和返回类型相关联的类和该关联类所在的名称空间。比如 B X::*p，则 f(p)的关联类集合为{X, B}，其中 B 为 p 指向的成员所关联的类，关联名称空间为类 B 和 X 所在的名称空间。

示例 15.7：实参是指向类成员的指针时所关联的类和名称空间

```
#include <iostream>
using namespace std;
class A;    //前置声明
namespace M{class B{}; void f2(B A::*){cout<<"A"<<endl;}}
namespace N{int a=1; void f(int){} enum E{b};  void f1(E){} }
   class A{public:  int a; M::B mb; friend void f2(M::B A::*){cout<<"B"<<endl;}};
void main(){//f(N::a); //错误，找不到名称 f。内置类型的关联名称空间集合和关联类集合是空集合
    f1(N::b);        //正确，实参是枚举型时，关联名称空间指的是声明枚举所在的名称空间
    M::B A::*p=&A::mb;
    //f2(p);      //错误，调用不明确，实参 p 关联的类集合为(A,B)，关联的名称空间集合为 M 和全局作用
                  //域(类 A 所在名称空间)，其中 B 和 M 为指针 p 指向的成员 mb 所关联的类和类
                  //B 所在的名称空间。因此类 A 中的友元函数 f2 和名称空间 M 中的函数 f2 都是候选函数
                  //但这两个函//数匹配得一样好，所以调用出错
}
```

（5）若实参是类类型（含联合 union 类型）时，则关联类集合由该类本身、该类的外围类、该类的直接或间接父类组成，关联名称空间集合由所有与其相关联的类声明时所在的名称空间组成。若实参是一个类模板实例，则关联名称空间集合和关联类集合还应包括模板实参本身的类型、声明模板的模板实参所在的类和名称空间。比如 f(A<B> mab){}，假设 A 和 B 都是类，则实参 mab 的关联类集合有(A,B)及它们的直接或间接父类，关联名称空间集合就是包含类 A

和类 B 及其父类的名称空间。

示例 15.8：实参是类类型时关联类和关联名称空间的创建方法

```
namespace M{class D;} //前置声明。注意：名称空间可以被追加定义
   namespace NN{
      namespace N{
         class A{public:
            class B{friend void f1(M::D *md){cout<<"A1"<<endl;}};
            friend void f2(M::D *md){cout<<"A2"<<endl;}
         }; //类A结束
         void f3(M::D *md){cout<<"A3"<<endl;} } //名称空间N结束
      void f4(M::D *md){cout<<"A4"<<endl;}}    //名称空间NN结束
   namespace M{ class D:public NN::N::A::B{friend void f5(D* md){cout<<"A5"<<endl;} };
   void f6(D* md){cout<<"A6"<<endl;} }
void main(){ M::D md; f1(&md); f2(&md); f3(&md); f5(&md); f6(&md);
   //f4(&md); //错误，找不到名称f4
}
```

分析：

① 对于 M::D md，则调用 fX(&md)，其中 X 表示某个序号（1, 2, 3, 4, 5, 6）中的一个。

② 实参&md 的关联类集合是(D, B, A)，其中 D 是实参&md 所属的类本身，B 是 D 的直接父类，A 是父类 B 所在的外围类。

③ 实参&md 的关联名称空间集合是(M, N)，其中 M 是类 D 所在的名称空间，N 是 D 的父类 B 和父类所在外围类 A 所在的名称空间，它们都是实参&md 的关联名称空间。

④ 名称空间 NN 不属于实参&md 的关联名称空间，因为 NN 不是&md 的关联类所在的名称空间。

⑤ 在关联类集合 A 中找到友元名称 f2，在 B 中找到友元名称 f1，在 D 中找到友元名称 f5，在名称空间 M 中找到 f6，在 N 中找到 f3，因此调用 fX(&md)能找到的名称为有 f1, f2, f3, f5, f6，但 f4 不在查找范围之内，因为 f4 位于名称空间 NN 中，因此对 f4 的调用是错误的。

示例 15.9：实参是模板实例时关联类和关联名称空间的创建方法

```
namespace N{class B;}  //前置声明
namespace M{ template<class T> class A{friend void f1( A<N::B>){cout<<"A"<<endl;}};
   void f2( A<N::B>){cout<<"B"<<endl;}}
namespace N{class B{friend void f3( M::A<B>){cout<<"C"<<endl;}};
   void f4( M::A<B>){cout<<"D"<<endl;}}
void main(){M::A<N::B> mab;f1(mab);f2(mab);f3(mab);f4(mab);}
```

分析：

① 调用 fX(mab)，其中 X 表示某个序号（1, 2, 3, 4）中的一个。

② 实参 mab 的关联类集合是(A,B)，其中 A 是 mab 所属的类模板本身，B 是类模板的模板实参本身的类型。

③ 实参 mab 的关联名称空间集合是(M,N)，其中 M 是类模板 A 本身所属的名称空间，N 是类模板的模板实参 B 所属的名称空间。

④ 在关联类集合 A 中找到友元名称 f1，在 B 中找到友元名称 f3，在关联名称空间集合 M 中找到名称 f2，在 N 中找到名称 f4，因此对 f1, f2, f3, f4 的调用都正确。

（6）若实参是函数类型时，则该集合包括所有与该实参函数的形参类型和返回值类型相关联的类集合和名称空间集合。

示例 15.10：实参是函数类型时关联类和关联名称空间的创建方法

```
namespace M1{class B;}        namespace M{class A;}  //对名称的前置声明
typedef M::A* F(M1::B);       //注意：应返回M::A*指针类型，否则必须先定义类A
namespace M{class A{};        void f1(F){cout<<"A"<<endl;}     }
namespace M1{class B{};       void f2(F){cout<<"B"<<endl; }}
namespace N{class C{};        C f3(F){cout<<"C"<<endl;return C();}}
M::A* g(M1::B){cout<<typeid(g).name()<<endl;return &M::A();}
void main(){  f1(g);   f2(g);   //f3(g); //错误，找不到名称 f }
```

分析：

① 调用 fX(g)，其中 X 表示某个序号（1, 2, 3）中的一个。

② 实参 g 的关联类集合是(A, B)，其中 A 是 g 的返回类型 M::A*所在的类，B 是 g 的形参 M1::B 所在的类。

③ 实参 g 的关联名称空间集合是(M, M1)，其中 M 是 g 的返回类型 M::A*所在的名称空间，M1 是 g 的形参 M1::B 所在的名称空间。

④ 名称空间 N 不属于实参 g 的关联名称空间，因为 g 的形参和返回类型都与 N 无关。

⑤ 在关联类集合 A 和 B 中未定义任何函数，因此未找到任何名称。在关联名称空间 M 中找到名称 f1，在 M1 中找到 f2，因此对 f1 和 f2 的调用是正确的；而对名称空间 N 中 f3 的调用是错误的，因为名称空间 N 不是实参 g 的关联名称空间。

15.4.2 typename 前缀和 template 前缀

1. 使用 typename 前缀指示模板内的名称代表某种类型

（1）typename 前缀用于指示受控受限名称代表一种类型。比如 class A{public:typedef int X;}; template<class T> void f(){T::X *y;}f<A>();，编译器并不能识别 T::X *y 中的 X 代表一种类型还是一个变量，但可以确定 X 是 T 的一个成员。若 X 是类型，则表示声明一个指针变量 y；若 X 不是类型，则表示这是一个乘积运算。再比如 template<class T> class A{public:int X; };template<class T> class B{public: A<T>::X* y;};，对于类 B 中的 A<T>::X* y，编译器也无法识别其中的 X 是否代表一种类型。

（2）C++规定受控受限名称（或依赖型受限名称）默认并不代表一种类型，除非在该名称

前面使用 typename 关键字作为前缀，且此处的关键字 typename 不能使用关键字 class 代替。比如 typename T::X *y;，此处的 X 代表某种类型，因为有 typename 前缀。注意 class T::X* y;是错误的，因为不能使用 class 关键字代替 typename 关键字。

（3）若受控受限名称使用了 typename 前缀，但在模板实参替换之后，该名称实际上并不是类型，则程序会在实例化时产生错误。比如 template<class T> class A{public:int X;}; template<class T> class B{public: typename A<T>::X* y;};，一旦实例化类 B 就会出错，如 B<int> mb;会出错，因为 typename A<T>::X* y;中 A 的模板实参 T 被 int 替换之后，X 并不代表某种类型（实际上是 int X）。

（4）使用 typename 前缀的规则。

① 名称在模板中出现。

② 名称是个受限名称。

③ 名称不用在父类继承列表中，也不用在构造函数的成员初始化列表中。

④ 名称是个受控（依赖型）名称。

⑤ 只有前 3 条都成立，才能使用 typename 前缀；否则，不能使用 typename 前缀。综合起来就是：① typename 只能出现在模板中；② typename 的名称必须是受限名称；③ 在继承列表中不能使用 typename；④ 在构造函数的成员初始化列表中也不能使用 typename。前面 3 个条件用于说明 typenmae 可以在什么地方使用。

⑥ 若前 3 条都成立，第 4 条不成立，则可以使用 typename 前缀，也可以不使用 typename 前缀。比如在模板内部 A<int>::B *p 表示声明一个指针，此时 typename 前缀可有可无，因为第 4 条不满足，此处 A<int>明确指定模板形参为 int，使名称 B 不再是受控的了。

⑦ 若满足第 2 条和第 4 条，且名称表示某种类型，就必须使用 typename 前缀。

示例 15.11：使用 typename 前缀的规则

```
template<typename T>                    //此处的 typename 用于声明模板形参
   class A{public:  class B{}; static int a;};
//template<class T> class C:typename A<int>::B{}; //错误，typename 不能出现在父类列表中
template<class T> class D{public:D(){}
   D(typename A<T>::B mb){}             //正确，typename 是必需的，表示 B 是一个来自模板 A 的类型
   //typename A<T> ma;                  //错误，因为不是受限名称，但在 VC++ 2010 中是正确的
   void f(){typename A<T>::B *p;  //正确，typename 是必需的，表示声明一个指针
      //A<T>::B *p1;                   //错误，应使用 typename 前缀，否则此声明表示一个乘积运算。但此
                                       //语句若出现在函数或成员函数内，VC++ 2010 能识别出是在声明一个
                                       //指针，是正确的
   }
   //A<T>::B *p2;          //错误，必须使用 teypname 前缀，否则此声明表示一个乘积运算
   A<int>::B *p3;          //正确，表示声明一个指针，此处 typename 关键字可有可无，因为 B 不是受控名称
     typename A<int> *p4;  //正确，表示声明一个指针
};   //类 D 结束
```

```
template<class T> class E:public A<T>::B{ public:
   //E(int i):typename A<T>::B(){}  //错误，typename 不能出现在构造函数的成员初始化列表中
   //此错误在实例化阶段才会产生，比如创建类 E 对象 E<int>me(9)时。但在 VC++ 2010 中是正确的
};
//typename A<int>::B mb1;   //错误，typename 前缀不能在模板外使用
A<int>::B mb2;
void main(){D<int> md; A<int>::B mab; D<int> md1(mab); md.f();
    //E<int> me(9); //错误
}
```

2. 使用 template 前缀指示名称代表某个模板

（1）template 前缀用于指示某个名称为模板，即形如 A<T>::B<T> *p;中的名称 B（假设 A 已知是模板），若未使用 template 前缀，则 B 可能就不是模板，该语句就表示(A<T>::B<T)> *p;，此时把 B 之后的尖括号解释成大于、小于符号。要使 B 表示模板则应使用 template 前缀，即 A<T>::template B<T> *p。注意：template 前缀并没有位于名称的最前面，比如 template A<T>::B<T> *p;是错误的，这是显式实例化类模板的语法。

（2）何时需要加上 template 关键字。

① 若限定运算符（::、.、->）前面的名称或表达式受控于（依赖）某个模板形参，且限定运算符后面的名称是 template-id，就必须在限定运算符的后面加上 template 关键字。注意：template 应出现在限定运算符之后。其中 template-id 指的是模板名加上尖括号括起来的模板实参列表，即形如 A<int>、A<T>的形式。

② 上述用法一般出现在模板内部。

③ 比如 typename A<T>::template B<T> *p;，其中的名称 B 是个 template-id，且限定运算符“::”前面的名称 A 受控于模板形参 T，因此应在“::”之后使用 template 关键字。注意：因为 B<T>同时代表某种类型，因此还应在名称 A 的前面加上 typename 前缀。

④ VC++ 2010 并不完全遵守上面的约定。在 VC++ 2010 中不使用 template 关键字不是一种错误，不在模板内部使用如上语法也不是一种错误。

示例 15.12：使用 template 前缀指示名称代表某个模板

```
template<class T> class A{public:
   template<class T1> class B{public:
      template<class T4> static void f(){cout<<"F"<<endl;}  };
   template<class T2> void g(T2){cout<<"G"<<endl;}};
template<class T3> void h(){
   A<float> ma; A<T3> ma1;
   A<T3>::template B<T3> mab;
   //template A<T3>::B<T3> mabb;       //错误，template 前缀应在限定运算符的后面
   A<float>::template B<float> mab1;  //错误，“::”前面的名称不是受控的。但在 VC++ 2010 中是
                                      //正确的
   ma1.template g<int>(4); ma1.g<int>(4);//g 应使用 template。但在 VC++ 2010 中是正确的
```

```
    mab.template f<T3>(); mab.f<T3>();//f 应使用 template。但在 VC++ 2010 中是正确的
    //ma.B<int>::f<int>();  //这是嵌套类模板和外围类模板之间的一种映射关系，在 VC++ 2010 中此种
                            //语法解释是错误的
    //typename mab.template B<T3>::template f<T3>();  //错误。在 VC++ 2010 中，只要不在 B<T3>
                                                      //前面使用 template 就都是正确的，其余情形读者自行验证
    //ma.template g(9);    //注意：使用模板实参推演时，是错误的。因为名称 g 不是 template-id，即
                           //g 之后没有用尖括号括起来的模板实参列表
}
void main(){h<int>(); A<int> ma;
    ma.template g<int>(4);}  //在 VC++ 2010 中，不在模板内部使用 template 语法也是正确的
```

3. 有 using 声明时 typename 和 template 的使用

（1）using 声明可以改变继承时名称的访问级别（即公有的、私有的和受保护的），但使用 using 声明引入的名称是否表示类型也需要使用 typename 关键字加以说明。其形式为 using typename A<T>::Y;，其中 A 是父类，Y 是某种类型。比如 template<class T> class A{public:typedef int Y;}; template<class T> class B:private A<T>{public: using typename A<T>::Y; Y my;};，其中 B 私有继承自类 A，using typename A<T>::Y;就是使用 typename 的示例。

（2）注意：C++不支持在 using 声明中使用 template 来指示某个名称代表模板，即形如 using A<T>::template B;这样的语句是错误的。

15.5　实例化

首先理解声明、定义和实例化的区别。

（1）声明就是让编译器知道有这么一个名称，比如 void h(); class A;就表示声明了一个无参函数 h 和一个类 A。定义就是对函数或者类的实现，比如 void h(){};class A{int a;}; 就表示定义了一个什么也不做的函数 h 和有一个变量 a 的类 A。

（2）声明一个模板的形式为：template<class T> void h(T a); 或 template<class T> class A;，这就表示声明了一个模板函数 h 和一个模板类 A。注意后面有个分号。模板、函数和类的定义，都是以大括号作为开始和结束的。

（3）实例按字面意思理解，就是指实际的例子（或物体）。在这里，实例指的是模板实例化时由编译器所重新编写（或定义）的函数或类。比如 template<class T>void f(T i){};，模板函数 f 的 int 类型实例是 void f(int i){};，即模板函数 f 的实例就是有一个 int 类型形参的普通函数 f 的定义。再比如 template<class T> class A{T i;};，类模板 A 的 int 类型实例是 class A{int i;};。

（4）实例化：以具体类型替换模板形参的过程被称为实例化。实例化时编译器会使用模板实参代替所有出现的模板形参对模板进行重新编写（或定义），从而创建出该模板的一个实例（即函数或类）。比如 template<class T> void f(T){T a;}; f<int>(3);，则对 f 实例化后，编译器会对 f

函数模板进行重新编写，函数中的 T 全部被替换为 int，因此创建出的 f 函数模板的 int 类型实例形式为 void f(int){int a;};，即 f 函数模板的 int 类型实例就是一个函数。

（5）模板定义只是在编译器内部保存模板的形式。模板定义并没有对任何类或函数进行定义，即模板不会创建实例，只有实例化时才会创建实例。比如 template<class T> void h(T); template<class T> void f(T){T a} f(3);，对 h 函数模板的声明和对 f 函数模板的定义都不会创建实例，因此没有对函数 h 或 f 进行定义。只有在调用 f(3)时才会创建出 f 函数模板的一个 int 类型实例，即调用 f(3)之后会生成一个实例，从而对函数 f 进行定义，其形式为 void f(int){int a;}。

（6）特化体：实例化时产生的实例（即函数或类）被称为特化体。可以把特化体理解为实例的别名。

15.5.1 实例化基本规则

（1）实例化重要规则：实例化模板时需要见到模板的完整定义（而不仅仅是声明）。

（2）实例化原理：使用模板可以处理功能相同但类型不同的一族函数或类，但是在具体实现时，编译器并不会把模板编译成处理所有类型的单一实例，而是被编译成多个单个的独立的实例，每一个单个的实例处理一种特定的类型，即对于处理同一种类型的情况模板只生成一个相同的实例，对于处理不同类型的情况模板则会生成不同的实例。比如有函数模板 template<typename T> void f(T a){...};，对于处理 int 类型的函数模板，会被编译器重新编译成一个单独的实例，该实例的形式为 void f(int a){...}；对于处理 float 类型的函数模板，又会被编译成一个单独的实例，其形式为 void f(float a){...}；对于其他类型原理相同。

（3）当模板被实例化时就会创建一个实例，在下次调用到相同的模板实例时就不会生成新的实例了，而会调用以前创建的那个实例。比如有函数模板 template<class T> void h(T a){static int i=1; cout<<i<<endl; i++; }，则调用 h(2)会生成一个 int 类型的函数模板实例，当下一次调用如 h(44)时就会使用以前生成的 int 类型函数模板实例，而不会创建新的实例。可以使用本例中的静态 int 类型变量 i 进行验证。

（4）模板被实例化后就会生成一个新的实例，但这个新生成的实例不存在类型转换（注：有些转换是允许的，如非类型实参所允许的转换、模板实参推演时所允许的转换和显式模板实参所允许的转换）。比如有函数模板 template<class T> void h(T a){}。int a=2; short b=3;，当调用 h(a)时会生成一个 int 类型的实例，但是当下一次调用 h(b)时则不会使用上次生成的 int 类型的实例把 short 转换为 int，而是会生成一个新的 short 类型的实例。

15.5.2 实例化的时机和位置点及两段式名称查询

对模板进行实例化需要解决三个问题：模板以什么方式实例化、模板什么时候实例化（即

实例化的时机）、模板在什么位置实例化（即实例化点）。例如：

```
template<class T> class A{}; A<int> *p; ① A<int > ma; ②
```

（1）模板 A 以什么方式实例化？可以通过模板实参推演和显式模板实参来进行实例化，其中 A<int> *p;和 A<int> ma;就是显式模板实参的方式。当然还可以通过显式实例化、特化的方式来实例化模板。

（2）模板什么时候实例化？A<int>*p;和 A<int>ma;会触发实例化吗？答案是 A<int> *p;不会触发实例化，A<int> ma;会触发实例化。

（3）在 A<int>ma;触发实例化时，是在①处还是②处进行实例化的？这就关系到模板实例化时的具体位置点，即实例化点的问题，本章后面会介绍这个问题。

1．按需实例化和延迟实例化

（1）特化体（重要概念，后文中会反复使用）：其实就是指实例化时生成的一个模板实例，即特化体就是实例的别名。使用模板时并不一定会生成模板特化体，或者说不一定会对模板实例化（比如仅仅声明指针时）。比如 A<int> ma;，此时编译器需要实例化模板 A，这时就会生成一个相应的模板特化体。而 A<int> *p;仅仅声明指针，编译器不需要实例化模板 A，因此也不会生成一个模板特化体。

（2）按需实例化：是指当编译器看到一个模板特化体时就将模板的形参替换为相应的实参，从而创建出模板的一个实例。这个过程是自动隐式完成的，因此按需实例化又被称为隐式实例化或自动实例化。

（3）延迟实例化：只有真正需要被实例化的部分才会进行实例化，这就使得编译器对模板的实例化向后延迟，因此被称为延迟实例化。延迟实例化对类模板中的成员比较明显。

（4）理解按需实例化和延迟实例化。

① 当模板被实例化时，不管是按需实例化还是延迟实例化，其实都是由编译器隐式自动完成的。

② 不管是按需实例化还是延迟实例化，编译器都只会在真正需要时才会实例化模板，只是两种实例化方式的时间点不同。

2．实例化的时机

（1）编译器需要知道某个模板特化体的大小（或完整定义）时，才对模板进行实例化。比如 A<int> *p=new A<int>;，因为此时需要知道模板特化体的大小，所以对模板进行实例化。

（2）仅仅声明类模板类型的指针或引用时，不需要对模板进行实例化。比如 template<class T> class A{};，则 A<int> *p; A<int>* p1=0; void g(A<int> &pa){}都不需要进行实例化。但要注意 A<int> &p;若不是函数的形参，则是错误的，因为必须初始化引用；A<int> &p1=0;也是错误的，因为必须引用一个实例，此处是整数 0，因此编译器首先会对其进行类型匹配，即把整数 0

强制转换为类型 A<int>，若类 A 中无此转换函数，则程序出错。

（3）若程序中使用了模板特化体的某个成员时，需要对模板进行实例化。

（4）创建对象时会使用构造函数，所以构造函数和构造函数调用的成员都会被实例化。比如 template<int N> class B{public: typedef int Y[N];}; template<int N>class A{public:A(){B<N> mb;} A(int){int a[N];} A(int,int){}};，则 A<–1> ma2(2,3);正确，因为 A(int,int)构造函数什么也没做；而 A<–1> ma;和 A<–1> ma1(9);都会发生负下标错误，因为 A()中的 B<N> mb;会被实例化，从而在类 B 中产生负下标，同理 int a[N];也会被例化。

（5）类模板成员实例化的时机：实例化类模板时，并不会实例化类模板中的所有成员。当类模板按需实例化时，类模板中的每个成员声明会同时被实例化，但其对应的定义并不一定会被实例化（此规则有两个例外）。比如 template<int N> class A{void f(int (*p)[N]){} void g(){int a[N];}};，则 A<–1> ma;会发生负下标错误，因为成员函数 f 中的函数形参 int (*p)[N]是一个声明，因此会被实例化，从而产生负下标；而函数 g 不会实例化，因此 int a[N];不会发生负下标错误。下面是此规则的两个例外。

① 类模板中的匿名 union 成员会被实例化。

② 虚成员函数的定义可能会被实例化。大多数编译器会实例化虚成员函数的定义，因为虚函数的内部结构机制要求虚函数应作为一个实体存在于某个结构（如虚函数表）之中。

（6）对于函数的默认实参，只要未被使用，就不会对其进行实例化。也就是说，在调用时指定了默认实参的值，就不会对其进行实例化。

（7）重载函数解析时隐式转换对实例化的影响：比如 template<class T> class A{public:A(int){}}; void f(A<int>){}void f(int){}，则调用 f(33)会调用 f(int)版本，因为 f(int)是完全匹配的。f(33)调用对两个 f 函数来讲都是可行函数，而对于 f(A<int>)调用可能会对模板 A 进行实例化，但这个实例化并不是必需的，因为 f(int)已经完全匹配，这就使得编译器不需要选用 f(A<int>)这个函数，因此对类模板 A 进行实例化就显得没有必要了，但 C++标准并未禁止这种情形。注：f(A<int>)可通过类 A 的单形参构造函数把整型值 33 转换为 A<int>，因此 f(A<int>)对于调用 f(33)来讲也是可行函数。

示例 15.13：类模板成员实例化的时机

```
template<int N> class A{public:typedef int Y[N];};
template<int N> class B{public:
    //int a[N];                    //错误，产生负下标，声明语句会被实例化
    //void g(int (*p)[N]){}        //错误，声明语句会被实例化。即形参 int (*p)[N]中的 N 会被-1 取
                                   //代，从而发生负下标错误
    //void g1(A<N> ma2){}       //错误，形参声明 A<N> ma2 会被实例化。但在 VC++ 2010 中不会实例化
    //void g3(typename A<N>::Y ma3){} //错误，产生负下标
    void g2(){A<N> ma3;}   //正确，成员的定义不会被实例化，因此 g2 未被实例化， A<N>也不会实例化
    //virtual void h(A<N> maa){} //错误，产生负下标。虚函数会被实例化，而且 A<N> maa 还是一个声明
```

```
    //virtual void h(){int a[N]; A<N> ma3; }  //错误。虚函数会被实例化，因此 a[N]和 A<N>都会发
                                                //生负下标错误
    //union{int a[N]; };                      //错误，产生负下标。无名的 union 成员会被实例化
    //static int a[N];                        //错误，产生负下标。声明语句会被实例化
};
void main(){B<-1> mb; }
```

3. 实例化点（POI）

（1）实例化点就是模板实例化时的位置点，在该点编译器会对模板进行实例化，因此需要见到模板的完整定义。

（2）实例化点基本要求：在实例化点对模板进行实例化，实例化之后的代码会在该点插入，这就要求该模板在此实例化点插入的代码必须是合法的，否则就不会在该点进行实例化。比如 template<class T> void f(T){}①void g(){② f<int>(3); ③}④;，本例中对模板 f 的实例化不可能在②和③位置点进行，因为不能在函数内部定义另一个函数。假设在位置点③进行实例化，则实例化之后的代码 void g(){f<int>(); void f(int){}}很明显是错误的，这里的 void f(int){}就是实例化后生成的实例。由此可见，实例化点与使用模板的点并不在同一个位置。

（3）实例化点具体规则。

- 对于非类类型特化体，C++规定其实例化点位于包含此特化体定义或声明的名称空间的紧临后方（注意：编译器的实际实例化点可能位于该编译单元的末尾）。比如 template<class T> void f(){}①void g(){② f<int>();③}④，则实例化点位于④处，因为 f 模板是非类类型特化体，因此其实例化点应在包含特化体 f 的名称空间（即函数 g）的紧临后方，即④处。②和③处不能作为实例化点，因为函数模板在函数内部实例化是非法的（即不能在函数内部定义另一个函数）。
- 对于类类型特化体，其实例化点位于包含此特化体定义或声明的名称空间的紧临前方。比如 template<class T>class A{};①void g(){② A<int> ma;③}④，则实例化点位于①处，因为 A 模板是类类型特化体，因此其实例化点应在包含特化体 A 的名称空间（即函数 g）的紧临前方，即①处。②和③处同样不能作为实例化点，因为在函数内部是不能出现模板的。
- 主要 POI 和次级 POI：此种情况出现在对一个模板进行实例化，在该模板内部又对其他模板进行实例化时。比如：

```
template<class T> class A{public: typedef int X; };①
template<class T> void f(){A<float>::X i=2; typename A<T>::X j=4; };
void main(){f<double>(); }②(2a,2b);
```

在 main 中使用 f<double>()对模板 f 进行实例化时，在 f 函数内部又有两个实例化 A<float>和 A<T>。对于 A<float>，POI 位于①处。而对于 A<T>，因为 A<T>是受控的，若在 1 处实例化，则模板形参 T 仍然受控于模板 f，因此不能在①处实例化；若在②处实例化 A<T>，则实例

化 f<double>时（f<double>也是在②处实例化的），A<double>（即 A<T>）也会同时实例化，这时 A<double>的实例化点就成为次级（或传递）POI。对于非类类型特化体，次级 POI 和主要 POI 的位置相同；但对于类类型特化体，次级 POI 位于主要 POI 之前。在此例中 A<float>的 POI 在①处，而 A<T>的次级 POI 位于 2a 处，f<double>的主要 POI 位于 2b 处。

4．编译器对多个相同实例的解决方案

（1）若使用同一个模板实参对同一个模板实例化多次，就会在程序中出现相同实例的多个实例化点。这时会产生两个问题：编译器可以在任何一个实例化点对模板进行实例化，那么编译器应在哪里进行实例化呢？编译器在不同的实例化点可能产生多个相同的实例，那么编译器怎么处理这些相同的实例呢？比如 template<class T>void f(){}void g(){f<int>();} ①void h(){void f<int>();}；②，编译器是在见到 f 的调用即①和②处就立即对 f 实例化还是等到程序最后同时进行实例化呢？在①和②处的实例化点都会生成一个实例，而且这两个实例是相同的，编译器怎么处理这两个相同的实例呢？

（2）对于多个实例化点，编译器有如下 3 种策略。

- 在每次需要实例化时就立即进行实例化，从而生成特化体。比如 void g(){f<int>();} ① void h(){void f<int>();}；②，编译器会在①和②处立即生成 f 的特化体。
- 在编译单元的最后，生成该编译单元的所有特化体。比如 void g(){f<int>();}① void h(){void f<int>();}；②，假设位于同一个编译单元里，编译器在①处不会马上实例化，等到②处（假设为整个编译单元的最后）编译器会生成所有的特化体。
- 在整个程序的最后，生成该程序的所有特化体。注：一个程序含多个编译单元。

（3）对于实例化时产生的多个相同实例，编译器的处理策略如下。

① 贪婪式实例化。微软 Windows 开发环境采用该策略。其方法是在所有实例化点处都进行实例化，并对所生成的实例进行标记，当链接器使用到这些实例时，对于多个重复的实例只保留其中一个，其余的丢弃。该策略的缺点很明显，就是浪费时间和存储空间。

② 查询式实例化。Sun Microsystems 采用此策略。其方法是使用一个共享的数据库，该数据库中存储了实例化时的所有信息，当编译器遇到需要被实例化的特化体时，就查询该数据库，然后根据数据库中的信息进行实例化。若数据库中已经存在对该特化体实例化的最新信息，则编译器什么也不做，否则就进行实例化，并把信息写入数据库中。此策略原理简单，效率很高，但具体实现时的难点在于需要维护数据库和其他文件之间的相互关系。比如在实例化时，对数据库中的信息进行了更新，那么更新信息应该怎样被反映到其他编译单元中等。

③ 迭代式实例化。cfront 采用此策略。其方法是使用预链接器把所有需要实例化的特化体都进行实例化。因为某个特化体内可能包含有对另一个特化体的实例化，因此此过程会被重复执行。此策略相当费时，比如使用其他策略只需花费 1 小时的时间，而使用此策略可能会花费

几天的时间。

④ 改进的迭代式实例化。EDG 编译器使用此策略。其核心思想是在使用迭代式实例化时，只实例化需要被实例化的特化体，这就大大缩短了时间。其具体步骤是在编译器和链接器之间创建一个文档，该文档记录着需要被实例化的特化体的信息，在预链接器实例化时，查询该文档，并根据该文档的信息对特化体进行实例化，同时把 POI 的位置信息记录在另一个文档中。此策略在 EDG 编译器中表现非常好，并且还据此实现了更高级的 C++模板特性。

（4）大多数编译器对多个相同实例的处理策略：大多数编译器都会把非内联的函数模板的实例化推迟至该编译单元的末尾进行，这样就相当于把该函数模板特化体的 POI 也移到了编译单元的末尾。对于类模板特化体，编译器只保留第一个 POI，后面的 POI 被忽略；对于非类模板特化体，所有 POI 都被保留，但编译器必须保证所有的 POI 处产生的特化体是相等的。注意：inline 指示符对编译器只是一个建议，编译器可以不遵循该建议，因此某些编译器可能并不支持内联函数。

（5）由于各编译器对多个相同实例所采用的处理策略不尽相同，因此应尽量避免使用如下形式的程序：template<class T> void f(){g();}void h(){f<int>();}①void g(){}②，因为编译器可能会在①处对 f 进行实例化，从而找不到名称 g；也可能会在编译单元末尾的②处或更后的位置对 f 进行实例化，这样就能正确地调用函数 g。

5. 两段式名称查询（实例化时的名称查询）规则

（1）注意：编译器并不一定遵守两段式名称查询规则。

（2）C++对模板形参的语法检查：C++总是假设模板形参的使用是最理想的，所以所有与模板形参有关的类型检查都延迟到实例化时再进行。比如 template<int N> class A{public: typedef int Y[N];};，其中 typedef int Y[N];的下标 N 有可能为负值，但编译器总是假设模板形参 N 处于最理想的情况下，即 N 不会为负值，因此该语句不会出错。

（3）受控名称的查询：因为在实例化之前编译器不知道受控名称的类型，因此也就无法对该名称进行解析，所以对于受控（依赖型）名称的解析，编译器需要在实例化点再次进行查询。因此，对受控名称编译器使用的是两段式查询，第一阶段发生在语法解析阶段；第二阶段发生在实例化阶段。比如 template<class T> void f(){T a=T(); g(a);};，对于名称 a，在实例化之前并不知道它的类型，因此无法对 a 进行解析；同时对于名称 g，因为形参 a 的类型在实例化之前不能确定，这也导致了在实例化之前不知道应该调用哪个 g 函数的重载版本。因此对于名称 a 和 g，需要在实例化之后再次进行名称解析。

（4）受控名称的两段式查询规则。

① 非受限受控名称，比如 template<class T> void f(){T i; g(i);};，其中的 g 就是非受限的，但却是受控名称。

② 第一阶段发生在模板定义时。

- 非受控名称在模板定义时使用常规查询规则和 ADL 规则进行解析。
- 非受限受控名称使用常规查询规则进行解析，可是查询的结果并不完整，也不会进行函数重载解析（即确定候选函数集，最后确定最佳匹配的过程），但编译器会记住查询结果。在此阶段可诊断出较多的错误。
- 受限受控名称在此阶段不进行查询。

③ 第二阶段发生在实例化点处。

- 非受控名称在此阶段不再进行查询。即使在此阶段查询到其他名称，也会忽略。
- 对于受限受控名称，编译器在该处使用常规查询规则和 ADL 规则进行查询，其主要目的是把模板形参替换为相应的模板实参。
- 非受限受控名称此时只会使用 ADL 规则对名称进行解析（因为第一阶段已使用常规查询规则进行过查询），然后把此阶段查询到的结果与第一阶段的结果组成一个候选函数集，再运用函数重载解析机制，以便从该集合中选出最佳匹配的函数调用。

④ 总结：根据规则，受限受控名称只在第二阶段（实例化时）进行查询。非受控名称和非受限受控名称最好在使用前进行声明，因为它们都会在第一阶段（定义模板时）使用常规查询规则进行查询。

⑤ VC++ 2010 不遵守以上规则。在 VC++ 2010 中，不管是受控名称还是非受控名称的查询似乎都是等到实例化时才进行的。

（5）在 VC++ 2010 中，成员函数只要未被实例化，就不会对成员函数内部进行任何语法检查，因此此时不管成员函数内部是否有非受控名称都不会进行名称查询。比如 template<class T> class A{public: void f(){djo, kdil kd,cki; "SD", d; ddx } }; A<int> ma;，在 f 函数内部是一些明显错误的代码，但在 VC++ 2010 中是正确的，因为这里并没有对类 A 中的成员函数 f 进行实例化，所以不会对类 A 中的成员函数 f 的内部进行任何语法检查。

示例 15.14：非受限的受控名称的两段式查询

```
template<class T> void f(T i){
   g(2);  //错误。名称 g 是非受控名称，只会在第一阶段（模板定义时）用常规查询和 ADL 规则进行查询，不会
          //在第二阶段（实例化点处）启动名称查询，但在 f 实例化之前是见不到名称 g 的定义的，因此最终
          //程序未找到名称 g，调用出错，但 VC++是正确的
   g(i);} //错误。名称 g 是非受限受控名称，会进行两段式名称查询，在第一阶段（模板定义时）使用常规查询
          //机制未找到名称 g（因为 g 定义于调用点之后）。在第二阶段即实例化点处（本例为 main 函数之后）
          //使用 ADL 机制进行名称查询，但因为 f(4)的调用传递给 i 的类型为 int，对内置类型不会启动 ADL
          //机制，从而导致函数 g(i)的调用产生错误（因为 g 定义于调用点之后，按常规查询是找不到 g 的定
          //义的，必须使用 ADL 机制）。但 VC++是正确的
void g(int){cout<<"G"<<endl;}
void main(){f(4);}
```

示例 15.15：非受限的受控名称的两段式查询与函数重载解析

```
class A{}; class B{};
class X{public:
   operator A(){return A();}  operator B(){return B();}}; //两个转换函数
void g(A){}
template<class T> void f(T i){
   //g(i);   }  //错误。在第一阶段（定义模板时）使用常规查询机制，找到g(A)，在第二阶段即实例化点
              //处（本例为main函数末尾）使用ADL机制查询，找到函数g(B)，从而组合成候选函数集
              //但因为类型X与A和B匹配得一样好，所以调用出错
void g(B){}
void main(){X mx;  f<X>(mx);}  //f函数模板在此处实例化
```

示例 15.16：VC++ 2010 对名称的实际解析规则

```
template<class T> void f(T i){i++;}   //对i的使用是错误的，但在VC++ 2010中正确，对i的名称查
                                      //询只有等到f被实例化后才能进行
int i=0;
template<class T> class A{public:
//以下函数在VC++ 2010中正确。因为在VC++ 2010中成员函数只要未被实例化，就不会对成员函数内部进行
//任何语法检查
void g(){int a[-1]; lkdjfoiuecno j"iek"k};};  //明显错误的非法代码
void main(){f(4); A<int> ma;}
```

6. 把类中的非受控名称修改为受控名称的方法

（1）把非受控名称修改为受限受控名称后，会使名称查询延迟到实例化之后进行，从而可以避免产生一些名称查询时的错误。下面以示例说明。

```
template<class T> class A{public: int g;};
template<class T> class B:public A<T>{public: void f(){ g();} };
template<> class A<int>{public:void g(){}};  //全局特化
void main(){ B<int> mb;  mb.f();  }   //错误（在VC++ 2010中正确）
```

分析：

① 函数 f 中的名称 g 是非受控名称，按照非受控名称的两段式查询规则，名称 g 会被解释成类 A 的 int 类型，而不会被解释成类 A 全局特化后的函数 g，因此出错。

② 若 g 是受限受控名称，则会在实例化之后进行名称查询，这时就能发现名称 g 在类 A 的 int 类型全局特化版本中被重新定义成函数，从而避免产生错误。

（2）通过如下 3 种方法可以把非受控名称修改为受限受控名称。

① 使用 this 指针。

② 使用作用域解析运算符。

③ 使用 using 声明把受控名称带入当前作用域中。

示例 15.17：把非受控名称修改为受限受控名称

```
template<class T> class A{public: int a; int b; int c;};
template<class T>class B:public A<T>{public:
```

```
    using A<T>::c;                    //使用 using 声明把受控名称带入当前作用域中
    void f(){ this->a=1;              //使用 this 指针把非受控名称变为受限受控名称
    A<T>::b=2;                        //使用作用域解析运算符把非受控名称变为受限受控名称
    c=3;} };
void main(){  B<int> mb; mb.f();}
```

7. 包含模型和分离模型

（1）一个程序可以由多个头文件（如.h 文件）和多个源文件（后缀一般为.cpp、.c、.cc 等）组成。在 C++中各个源文件是单独编译的，因此把同一个源文件中的内容（包括使用#include 包含进来的文件）称为编译单元。在源文件中不一定会含有 main 函数。编译器把各个源文件编译成多个分离的.obj 文件后，再由链接器把这些文件链接成一个可执行文件（如.exe 或.dll 文件），这个文件就是整个程序的代码了。

（2）若按传统方式把模板声明写于头文件中，把定义写于源文件中，即头文件+源文件模型，则在编译时不会出错，但在链接时程序会发生错误。因为模板不会被实例化，具体原因如下。

实例化模板时编译器必须知道两个信息，即应该对哪个定义进行实例化，以及使用哪个模板实参对其进行实例化。在传统的头文件+源文件模型中，这两个信息分别位于两个分开编译的文件中。在使用模板的文件中，编译器看不见对模板的定义，因此无法实例化，但编译器可以假设模板定义于其他位置，因而可以等到链接时再实例化；在实例化时，又不知道实例化该模板的哪一个模板实参，因此最终结果是无法对模板进行实例化。比如头文件 a.h 的内容为 template<class T> void f(T);，源文件 a.cpp 的内容为#include "a.h" template<class T> void f(T){}，主文件 b.cpp 的内容为#include "a.h"void main(){f(9); }，则其中的 f(9)调用将出错，函数模板未被实例化，因为程序找不到函数 f 的定义。此处对模板的使用和定义分别位于两个不同的编译文件 b.cpp 和 a.cpp 中。在 b.cpp 中对 f(9)的调用看不到函数模板在 a.cpp 中的定义，因此需要等到链接时再实例化；但在链接阶段对 a.cpp 中的函数模板 f 进行实例化时，又不知道应该使用哪个模板实参对函数模板 f 进行实例化，因为此时的模板实参位于 b.cpp 文件内。因此，最终结果是函数模板 f 未被实例化，从而产生错误。

（3）包含模型和分离模型就是用于解决上述模板实例化问题的两种模型。

（4）包含模型（推荐使用的模型）就是把模板的声明和定义放在同一个文件中（比如都放在头文件中），这样就解决了关于模板不会被实例化的问题。有 3 种方法实现此策略：一是在头文件的最后一行把定义模板的文件使用#include 包含进来；二是在需要使用模板的文件中直接使用#include 把定义模板的文件包含进来；三是直接把模板定义和声明写在同一个文件中。

（5）包含模型的缺点：因为模板的声明和定义位于同一个文件中，这会导致程序包含进来过多的其他文件，增加了编译时间。

（6）分离模型就是在声明模板时使用 export 关键字作为前缀，这样就可以把模板的声明和定义分隔在两个不同的文件中。

（7）分离模型的缺点：首先，大多数编译器都不支持 export 关键字；其次，使用分离模型并不会比包含模型节省多少编译时间。基于以上原因，本书不对 export 作过多介绍。

15.5.3　显式实例化

1. 显式实例化格式

（1）显式实例化可明确指定实例化点。

（2）显式实例化指令的语法格式如下。

显式实例化类模板的格式：template class 类名<模板实参列表>;

显式实例化函数模板的格式：template 返回类型 函数名<模板实参列表>(函数实参列表);

① 显式实例化指令以 template 关键字开头，后接需要实例化的模板特化体声明语句。

② 显式实例化指令是以分号结束的，而不能以大括号“{}”结束。

③ 显式实例化指令与显式特化声明语句很相似，只是显式实例化指令没有 template 后面的尖括号“<>”和定义模板的大括号“{}”。

④ 若函数模板形参可由函数实参推导出来，那么显式实例化时也可省略函数名后的尖括号及其中的内容。

示例 15.18：显式实例化格式

```
template<class T> class A{public: void f(){} template<class X> void g(X){}};
template <class T> void h(T a){}
template void h<int>(int a);          //正确，显式实例化函数模板 h
template void h<>(float);             //正确，h 的模板形参 T 通过模板实参推演确定
template void h(double);              //正确，空的尖括号“<>”可省略
template class A<int>;                //正确，显式实例化类模板 A
template<> class A<char>;             //正确，但这不是显式实例化，而是显式模板特化声明
//template class A<bool>{};           //错误，显式实例化不能以大括号结束
//template void h<char>(char){}       //错误，同上
//template void f<int*>;              //错误，显式实例化函数模板时不能省略函数形参表
//template f<int**>();                //错误，显式实例化函数模板时不能省略返回类型
//显式实例化类模板中成员的方法
//template void A::g<int>(int);       //错误，只能显式实例化类模板的某个具体实例的成员
template void A<int*>::f();           //正确，显式实例化类模板 A 中的成员函数 f
template void A<float*>::g<int>(int); //正确，显式实例化类模板 A 中的成员模板函数 g
template void A<double*>::g(int);     //正确，g 的模板形参通过模板实参推演确定
void main(){}
```

2. 显式实例化的基本规则

（1）不能在局部范围内（包括类作用域内）显式实例化模板，而应在全局范围内，因为模板的声明或定义不能在局部范围或函数内进行。比如 template<class T> void f(T){} void g(){template void f(int);}错误，不能在函数 g 内显式实例化模板 f。

（2）在同一个程序中（注意：这里指的是整个程序，而非单个文件），对同一个实例的显式实例化只能进行一次，否则在链接阶段会发生错误。比如 template<class T> void f(T){} template void f(int);template void f<int>(int);错误，因为对模板 f 的同一个 int 类型实例进行了两次显式实例化。但在 VC++ 2010 中是正确的。

（3）因为对模板实例化时必须见到模板的定义，因此在显式实例化声明所在的编译单元（同一个.cpp 文件中的内容被视为一个编译单元）中，必须给出函数模板的定义，如果定义不可见就会发生错误。

（4）显式实例化类模板，会使其所有成员也被显式实例化。比如 template<int N> class A{public: void f(){ int a[N] };，则 template class A<-1>;错误（a[N]负下标），显式实例化类模板会使成员函数 f 也被实例化；而 A<-1> ma;是正确的，因为实例化类模板时，并不会实例化类模板中成员的定义，因此成员 f 未被实例化。

（5）显式实例化与显式特化的关系：在一个程序中，不能对模板的同一个实例既进行显式实例化又进行显式特化，但对模板的不同实例是可以同时进行显式实例化和显式特化的。比如 template<class T> void f(T){} template void f(int); template<> void f(float); template<> void f(int);，其中 template<> void f(int);是错误的，因为在之前已对 f 的 int 实例进行了显式实例化；而 template<> void f(float); 是正确的，因为对不同实例可以同时进行显式实例化和显式特化。

3．显式实例化的作用

（1）避免在头文件中定义模板。被显式实例化后的函数实例与普通函数类似，无须使用 export 关键字就可以在其他文件中使用，这样显式实例化就可以避免在头文件中定义模板，从而可以使用类似于分离模型的方式编写程序，避免包含重复的头文件。其缺点是每次只能显式实例化一种类型。具体方法是，在头文件中只声明模板，然后在源文件中定义该模板，并对该模板的某个实例进行显式实例化，这样在使用到该模板的这个实例时，只需包含相应的头文件就行，但未被显式实例化的其他实例仍不能使用。比如 aa.h 的内容为 template <class T> void f(T);，aa.cpp 的内容为 template<class T> void f(T){} template void f(int);，主文件 bb.cpp 的内容为 #include "aa.h"void main(){f(3); f(4.4);}，其中 f(3)调用是正确的，因为 f(T)的 int 版本在 aa.cpp 中被显式实例化了；而 f(4.4)调用是错误的，因为未找到 f(double)类型的函数模板定义。

（2）限制产生多个相同的模板实例。在不同的编译单元（文件）中实例化模板时编译器可能会产生多个相同的实例，特别是使用贪婪式实例化的编译器，因此严重浪费了程序的编译时间和存储空间。一种解决方法就是除了需要使用模板的这个实例的文件之外，禁止在其他文件中对模板的这个实例进行实例化。下面介绍 3 种阻止实例化的方案。

方案 1：只允许实例化指定的实例，而阻止其他实例被实例化。在需要实例化的编译单元中定义模板，并对需要在其他编译单元中使用的实例进行显式实例化，在其他编译单元中只声

明而不定义模板，这样就能阻止模板的其他实例在其他编译单元中被实例化。

示例 15.19：仅允许实例化指定的实例，而阻止其他实例被实例化

```
//编译单元（文件）1：假设文件名为 ac.cpp
template<class T> void f(T); //声明而不定义模板，这样模板 f 就不能在此编译单元中被实例化
void g(){f<int>(3);      //被显式实例化后的函数实例与普通函数类似，此处调用的是在文件 ad.cpp 中被显
                         //式实例化的函数模板 f 的 int 版本实例
   //f<float>(9); //错误，函数模板 f 的 float 版本未定义，此处不会调用 ad.cpp 中的 float 版本实例
}
//编译单元（文件）2：假设文件名为 ad.cpp
//源文件不需要被包含进来，在链接阶段，编译器会把两个文件链接在一起
void g();                    //声明函数 g
template<class T> void f(T){ }  //在此文件中定义模板 f
template void f(int);//显式实例化模板 f，此步骤不能省略；否则 ac.cpp 文件中的函数 g 会找不到 f 的定义
void main(){g();}
```

方案 2：阻止某个具体实例被实例化。在各个文件中都可以对模板进行定义（理论上定义应完全相同），但只在需要实例化的编译单元中显式实例化模板，其他编译单元只声明而不定义该模板实例的全局特化版本，这样就能阻止在该编译单元对该特化体类型的实例进行实例化，而其他类型的实例可以正常实例化。

示例 15.20：阻止某个具体实例被实例化

```
//编译单元（文件）1：假设文件名为 ac.cpp
template<class T> void f(T){}    //声明并定义模板，因为是在不同编译单元中定义的模板且未使用 export
                                 //关键字，因此程序不会出现模板重定义的错误
template<> void f<int>(int);   //只声明而不定义模板的 int 版本全局特化体，这样在该文件中就不能实例
                               //化模板函数 f 的 int 版本实例。而其他类型的实例可以正常实例化
void g(){f<int>(3);           //此处调用在文件 ad.cpp 中被显式实例化的 f<int>，而不会调用本文件中
                              //f 的特化体版本 f<int>
   f(3.3);   }              //调用本文件中的模板函数 f 的 float 版本实例
//编译单元（文件）2：假设文件名为 ad.cpp
//源文件不需要被包含进来，在链接阶段，编译器会把两个文件链接在一起
void g();    //声明函数 g
template<class T> void f(T){ }  //在此文件中定义模板 f
template void f(int); //显式实例化模板 f，此步骤不能省略；否则 ac.cpp 文件中的函数 g 会找不到 f 的定义
void main(){g();}
```

方案 3：使用非 C++标准的语法，即在显式实例化指令之前使用 extern 关键字，明确指出不能在该编译单元（文件）内实例化某版本的实例。

示例 15.21：使用 extern 关键字明确指出不能实例化某版本的实例

```
template<class T> void f(T){}
extern template void f(int);    //使用 extern 关键字，明确指出不能在该编译单元（文件）内实例化函
                                //数模板 f 的 int 版本实例
void main(){  //f(9); //错误，程序已禁止实例化 int 类型实例
     f(3.3); }  //正确，程序并未禁止实例化 float 类型实例
```

15.6 类模板中的成员

1. 类模板及其对象的类型

（1）类模板所创建的对象类型：形式为“类名<模板实参,...>”。比如A<int> ma; A<float> mb;，则 ma 的类型为 A<int>，mb 的类型为 A<float>，因此 ma 和 mb 是两个类型不同的对象；再比如 A<int, float> mb;，则 mb 的类型为 A<int, float>。

（2）类模板的类型：形式为“类名<模板形参名,...>”。比如 template <class T1, class T2> class A{};，则类模板 A 的类型为 A<T1,T2>。

（3）类模板类型的作用：一般用于在类模板内部声明自身类型的变量和在类模板外部定义成员函数。比如 template<class T> class A{void f(A<T> const &ma){}};，其中成员函数的形参 A<T>const &ma 就是一个例子。

（4）在类模板内部（包括在类模板外部定义的成员函数内部），类模板名称可以直接当作类模板类型使用，在类模板外部的其他定义中必须使用完整的类模板类型名称。比如 template<class T>class A{A ma; void f(A a){}}; void g(){A<int> ma; A ma1; }，在类模板 A 中的 A ma;和 A a;的类型相当于 A<T>类型。在函数 g 中 A ma1;是错误的语法，因为在类模板外部必须使用完整的类模板类型名称，正确形式为 A<int> ma1;。

（5）内置类模板名称的类型，即在类模板自身的作用域中使用该类模板名称的情形。在此种情形下，单独的类模板名称不能代表模板类型，只能表示类类型。类模板名称后面也可以跟着模板实参，若类模板名称后面没有跟着模板实参，这时该类的模板形参就是该类的模板实参（对于局部特化，则可以使用特化后的实参代表该类的模板实参）。比如 template<class T> class A{C<int> *a; C* a1;};，其中 C<int>后面跟着的<int>是模板实参，但 C<int>不能代表模板类型，它代表的是类模板 C 的一个 int 类型实例，其类型为 C<int>；而 C* a1;中的类名 C 也不能代表模板类型，它表示的是 C<T>，其模板实参是使用该类的模板形参 T 表示的。

（6）使内置类模板名称表示模板类型的方法是，可以使用作用域解析运算符“::”把类模板中的内置类模板名称强制表示为模板类型。比如 template<class T> class B{A< ::B> ma;};，则 A< ::B>中的 B 就表示模板类型。

（7）当类模板名称作为模板形参的实参时，其名称是代表类类型还是模板类型是有明显区别的，详见以下示例。

示例 15.22：内置类模板名称的使用

```
template<template<class T> class X> class A{};  //X是模板模板形参
template<class T> class B{public:
    B *p;              //正确，类模板名称在类模板内部可直接使用，其类型相当于B<T>
    B<T> *p1;          //正确，使用完整的类模板类型的名称声明变量
    void f();
```

```
    //A<B> ma1;          //错误,在类 B 内部的名称 B 只能表示类类型,而不能表示模板类型,B 的类型为 B<T>,
                         //类模板 A 要求一个模板实参
    //A< B<T> > ma3;      //错误,同上
     A< ::B> ma2;        //正确,使用作用域解析运算符强制把名称 B 表示为模板类型。注意:某些编译器需
                         //要在"<"和"::"之间加上空格
    //A<B<int> > ma4;     //错误,B<int>表示的是类模板 B 的 int 类型实例,其类型为 B<int>,不能表示
                         //模板类型
   }; //类模板 B 结束
template<class T> void B<T>::f(){  //类模板类型 B<T>用于定义成员函数
    B *p2; } //在类模板外部定义的成员函数内部,也可直接使用类模板名称,其类型相当于 B<T>
    //B mb;          //错误,在类模板外部不能直接使用类模板名称定义变量
void main(){A<B> ma; B<int> mb; mb.f();
    //B<float> mb2=mb;  //错误,类型不匹配,mb 的类型为 B<int>,而 mb2 的类型为 B<float>,它们是
                        //不同的类型
    B<int> mb1=mb;} //正确,mb1 和 mb 的类型都为 B<int>
```

2. 类模板中的成员函数和嵌套类

(1) 成员函数和嵌套类既可以在类模板内部直接定义，也可以在类模板外部进行定义。在类模板内部直接定义比较简单，本文只介绍在类模板外部进行定义的方法。

(2) 在类模板外部定义成员函数的形式为：

```
template<模板形参列表>返回类型 外围类名<模板形参名列表>::函数名(参数列表){...}
```

示例如下：

```
template<class T1, class T2> void A<T1,T2>::f(...){...}; //假设 f 和 A 都已正确处理
```

(3) 在类模板外部定义嵌套类的形式为：

```
template<模板形参列表> class 外围类名<模板形参名列表>::嵌套类名{类体}
```

示例如下：

```
template<class T1,class T2> class A<T1,T2>::B{};   //假设类 A 已定义,B 已在类 A 中声明
```

说明：

① template 后面的模板形参列表应与要定义的类的模板形参列表一致（包括数量，但模板形参名称除外）。比如 template<class T> class A{void f();};template<class T1> void A<T1>::f(){}; 正确，模板形参名 T1 可以与定义类时的模板形参名 T 不一致；而 template<class T,class T1> void A<T,T1>::f(){};错误，因为 template<class T,class T1>与定义类 A 时的 template<class T>不一致。

②“外围类名<模板形参名列表>”应与“template<模板形参列表>”中的形参名称一致（包括数量、顺序、名称都应一致）。比如 template<class T1, class T2> class A{void f();}; template<class T1, class T2> void A<T1, T3,>:: f(){}错误，因为<T1,T3>与<class T1, class T2>中的 T1 和 T2 名称不一致。

③ “外围类名<模板形参名列表>”表示的是类模板的类型，也就是该成员函数属于哪一个类。在尖括号中只需列出模板形参名称即可。比如 template<class T1, class T2>void A<T1,T2>::f(){};，指出成员函数 f 属于类型为 A<T1,T2>的类模板。

④ “template<模板形参列表>”指出类模板的成员是一个模板（严格来说，非模板成员函数称不上真正的模板）。

（4）类模板中的成员函数和嵌套类本身也是模板，但并不是真正的模板。

（5）调用类模板中的成员函数的方法与调用普通函数相同，直接使用对象名调用即可。比如 A<int> ma; ma.f();。

（6）创建嵌套类对象：只能创建外围类模板的某一个具体实例的嵌套类对象。比如 template<class T> class A{public:class B{}; };，则 A::B mb;错误，A<int>::B mb;正确。

（7）类模板中成员函数的实例化是由调用该函数的对象类型决定的，在调用类模板的成员函数时不会进行模板实参推演，若成员函数的形参是该类模板的形参，则该形参允许进行常规类型转换。比如 template<class T>class A{public: void f(T){}}; A<int> ma;，则 ma.f(3.3);正确，调用时不会执行模板实参推演，即不会把 f(T)中的 T 根据函数实参 3.3 推导出为 double 类型，其类型是由类模板对象 ma 的类型 A<int>决定的，即 T=int，在调用 ma.f(3.3)会执行常规类型转换，把 double 类型的 3.3 转换为 A<int>中的 int 类型。

（8）在类外定义成员的规则同样适用于类模板，即如果出现被定义成员的完全限定名，则直到该成员定义结束，该成员的定义都在类体（类作用域）之中。详见 8.3.1 节

示例 15.23：在类模板外部定义函数和嵌套类

```
template<class T1, class T2> class A{public:void f();void g(); class B;};
                                                          //其中 f、g、B 在类外定义
template<class T1,class T2> void A<T1,T2>::f(){}   //正确形式
template<class T3, class T4> void A<T3,T4>::g(){}; //正确，模板形参名可以省略，即可以不一致
template<class T1,class T2> class A<T1,T2>::B{};   //正确形式
//template<class T1> void A<T1,T2>::f(){};//错误，template 之后的模板形参数量比类 A 的模板形参
                                          //数量少
//template<class T1,class T2> void A<T1>::f(){}     //错误，类名 A 之后的模板形参名数量太少
//template<class T1,class T2> void A<T2, T1>::f(){} //错误，A<T2,T1>的顺序与 template<class
                                                    // T1,class T2>不一致
//template<class T1, class T2> void A<T1,T3>::f(){} //错误，此处 T3 是不能识别的标识符
//template<class T1,class T2> void A<class T1,class T2>::f(){} //错误，类名 A 后面只需列出
                          //模板形参名称即可，即 A<class T1,class T2>应改为 A<T1,T2>的形式
template<class T>class B{public: void f(T){}};
void main(){  A<int,int> ma; ma.f();  //调用成员函数
    //A::B mb;        //错误，只能创建外围类模板的某一个具体实例的嵌套类对象
     A<int,int>::B mab;//正确
     B<int> mb;
     mb.f(3.3); }     //正确，原因见文中叙述
```

3. 类模板中的静态成员

（1）静态成员必须在类模板外部进行定义，该定义不会分配内存空间，只有在实例化静态数据成员时才会分配内存空间。比如 template<class T> class A{static int a;};，定义静态成员的方

法为：template<class T> int A<T>::a=1;。

（2）静态成员只能被类模板的同一个实例共享，不同实例不会共享同一个静态成员。比如 template<class T>class A{public:static int i;}; template<class T> int A<T>::i=1; A<int> ma; A<float> mb;，ma 和 mb 使用的是不同的静态成员 i，对于 ma.i=2;，cout<<mb.i;仍输出 i 的值 1。

（3）类模板的静态成员本身是一个模板，但算不上真正的模板。

4．类模板中的模板成员

（1）类模板中的成员模板函数的声明方式和模板类与普通模板相同，即都是以 template 开始。比如 template<class T> class A{public: template<class T1> void f(){}};，其中 f 就是类模板 A 的成员模板函数。

（2）成员模板函数不能是虚拟的，比如 class A{template<class T> virtual void f();};错误。因为虚函数机制需要一个大小固定的虚函数表，每个虚函数对应该表中的一个条目。然而，若虚函数是模板，则意味着该虚函数可能存在多个实例，只有在编译器编译完整个程序时才能知道有多少个虚函数的实例，因此虚函数表的大小就无法固定。

（3）类模板中的普通成员函数可以是虚拟的，因为当类模板被实例化时，虚函数的数量就确定了。比如 template<class T> class A{public: virtual void f(); };，其中的虚函数 f 是正确的。

（4）成员函数模板和嵌套类模板，同样既可在类模板内部直接定义，也可在类模板外部进行定义。

（5）在类模板外部定义模板成员时，需要使用多重“template<模板形参列表>”子句，以及多重表示所定义的模板成员所属类模板类型的“类名<模板形参名列表>”子句。下面通过示例进行说明。

示例 15.24：在类模板外部定义模板成员

```
template<class T1> class A{public:
   template<class T2> class B; template<class T3> void g(T3 a);};
template<class T1> template<class T2> class A<T1>::B{};     //定义嵌套类模板 B 时有多重
                        //template<...>子句，第一个属于外围类 A 的，第二个属于嵌套类模板 B 的
                        //其中的 A<T1>表示嵌套的类模板 B 属于类型为 A<T1>的模板
template<class T1> template<class T3> void A<T1>::g(T3 a){}   //意义同上
```

（6）在类模板外部定义多层嵌套类模板时，其关键点在于“template<...>”子句应与该模板成员所属类（模板）类型的“类名<...>”子句一致。在定义时每个 template<...>子句需要从最外层的类模板开始写起，然后再把相应的表示该类模板类型的子句“类名<...>”写上，每个 template<...>子句中的模板形参名与表示类模板类型的类名后的尖括号中的模板形参名是一一对应的。若嵌套的类不是模板，则省略 template<...>子句，同时省略相对应的类名后的尖括号。

（7）对已经定义好的 template<...>子句的分析：对于在类外定义的含有嵌套类模板的成员，应从定义语句的含有作用域解析运算符的地方从左到右对类名进行分析，若类名后有尖括号，

则在左边的多重 template<...>子句中从左到右与类名后的尖括号信息依次进行匹配；若类名后没有尖括号，则不用寻找与之对应的 template<...>子句。一直重复下去，直到分析完毕。若分析之后还有一个最内层的 template<....>子句，则表示定义的成员是模板。

（8）创建多层嵌套类模板对象的方法：同嵌套类一样，只能创建某个具体外围类实例的嵌套类模板对象，其方法是使用作用域解析运算符，若嵌套的类是模板，则类名后需要有一个带尖括号的模板实参列表；若嵌套的类不是模板，则不需要尖括号。比如 A<int>::B<float>::C::D<int> md;，假设此语句正确，则说明类 A、B、D 都是模板，而类 C 不是模板，因为名称 C 后面没有尖括号。

示例 15.25：多层嵌套类模板

```
template<class T1> class A{public:
      template<class T2> class B{ public:
           class C{ public:
                template<class T3> class D{public:
                     template<class T4> void f();  void g();};};};};
//在类模板外部定义模板成员函数 f，使用多重 template<...>子句
template<class T1> template<class T2> template<class T3> template<class T4>
              void A<T1>::B<T2>::C::D<T3>:: //成员所属类模板的类型子句“类名<...>”
              f(){}
//在类模板外部定义成员函数 g
template<class T1> template<class T2> template<class T3> //多重 template<...>子句
              void A<T1>::B<T2>::C::D<T3>:://成员所属类模板的类型子句“类名<...>”
              g(){}
A<int>::B<int>::C::D<int> mabcd;  //创建的多层嵌套类模板对象
```

① 对函数 f 的分析：

- 首先从 A<T1>::B<T2>::C:D<T3>::f();处开始分析。
- 作用域解析运算符最左边的类名 A 后面有尖括号，则最左边的 template<class T1>是与之匹配的 template<...>子句。
- 第 2 个类名 B 后面有尖括号，则从左往右数第 2 个 template<class T2>是与之匹配的 template<...>子句。
- 第 3 个类名 C 后面没有尖括号，因此不需要从左边的多重 template<...>子句中寻找匹配的子句。
- 第 4 个类名 D 后面有尖括号，则从左往右数第 3 个 template<class T3>是与之匹配的 template<...>子句。
- 第 5 个是正在被定义的函数名，但左边的多重 template<...>子句中还有一个 template<class T4>没有与之对应的类名，则说明 template<class T4>与正在被定义的函数匹配，说明正在被定义的函数是一个模板。

② 对函数 g 的分析：同函数 f。

15.7　模板特化

（1）泛化模板：即非特化模板，或称为通用模板、常规模板、普通模板。

（2）主模板：不含模板实参列表的模板。也就是在特化的情形下，相对于被特化模板的通用模板。比如 template<class T1,class T2>class A{}; template<class T1> class A<T1,int>{};，其主模板是 template<class T1, class T2> class A{}。

15.7.1　全局特化与局部特化

1. 模板特化基本概念

（1）特化模板的原因：模板适用于所有类型，但是有些特殊类型不需要与模板有相同的操作或者定义，因此就需要为这些特殊的类型进行特别的定义，这些特别的定义就是对模板的特化。比如 template<class T>void h(T a){cout<<"A";}，对所有类型都输出字符 A，若想要对 int 类型输出字符 B，则要定义该模板的特化版本。

（2）特化分为全局特化和偏/局部特化。因为全局特化和局部特化都是显式进行的，因此都可以被称为显式特化。注：很多书籍把全局特化也称为显示特化。

（3）全局特化：把所有的模板形参都替换为需要进行特殊化处理的模板实参。比如 template<class T1, class T2> void f(T1, T2);，进行全局特化后，T1 和 T2 都会被特殊化处理。

（4）局部特化：只把部分模板形参替换为需要进行特殊化处理的模板实参。比如 template<class T1, class T2> void f(T1, T2);，局部特化可能只会特殊化处理其中的 T1 或者 T2，不会把 T1 和 T2 都进行特殊化处理。

2. 全局特化格式及基本语法要求

（1）全局特化函数模板格式：template<> 返回类型 函数名<要特化的模板实参列表>(参数列表) {}。

示例如下：

```
template<class T1, class T2> void f(T1, T2){};
template<> void f<int, float>(int, float){};//形参 T1 被特化为 int 类型，T2 被特化为 float 类型
```

（2）全局特化类模板格式：template<> class 类名<要特化的模板实参列表>{};。

示例如下：

```
template<class T1, class T2> class A{};
template<> class A<int, float> class A{}; //形参 T1 被特化为 int 类型，T2 被特化为 float 类型
```

（3）全局特化基本语法要求如下。

① 以 template<>前缀开头（这是全局特化的标志），表明要全局特化一个模板，然后在函数名或类名后用尖括号括住需要特殊化处理的模板实参。比如 template <class T> void h(T a){},

其 int 类型的全局特化版本为 template<> void h<int>(int a){}。

② 全局特化时指定的需要特化的模板实参列表必须与模板形参列表一一对应，如不能使用一个非类型值实参来替换一个模板类型形参。比如 template<class T1, class T2>class A{};，则 template<> class A<int, 3>{};是错误的，因为 3 是非类型值。

③ 使用全局特化版本的模板时，只需像通用模板一样使用即可，但其模板实参应与全局特化的模板实参相对应，否则就是使用主模板。比如 template <class T> void h(T a){} template<> void h<int>(int a){}，则 h(3)调用全局特化函数 h(int)，而 h(3.3)调用主模板函数 h(T)。

④ 全局特化函数模板时，特化的模板实参类型应与函数的形参类型一致。比如 template <class T> void h(T a){};，则 template<> void h<int>(float a){};错误，因为模板实参<int>和函数形参（float）不一致。

⑤ 在全局特化版本中模板形参是不可见的。比如 template<> void h<int,int>(int a,int b){T1 a;}错误，类型形参 T1 不可见，所以在这里 T1 是未知的标识符。

⑥ 如果可以从函数实参中推导出模板形参，则在全局特化时可以省略能推导出的模板实参部分；若所有实参都能被推导出来，则可省略函数名后的尖括号及其内容。比如 template<class T>void f(T){}，则 template<> void f(int a){}表示把模板形参 T 特化为 int 类型，由模板实参推导出模板形参 T 的类型为 int，此处省略了<int>。当然空的尖括号也可不省略，比如 template<> void f<>(int a){}。

⑦ 在全局特化函数模板时，若使用模板实参推演，则函数形参必须是明确指定的类型，而不能是某个数值。比如 template<class T> void g(T){} template<> void g(3){}错误，很明显不能有类似于 void g(3){}这样的函数定义，函数定义要求函数形参为类型而不是数值。

⑧ 非类型模板形参同样可以被全局特化。比如 template<int N> void f(int){}，其特化形式为 template<> void f<3>(int){}。

⑨ 非类型模板形参在全局特化时也可以使用模板实参推演确定，前提是必须在函数形参中建立起与非类型形参之间的关联。比如 template<int N> void f(int (*)[N]){};，则 template<> void f(int (*)[4]){};正确，推导出 N=4。注意：函数形参中的数组会被转换为指针的形式，比如 template<int N> void f(int a[N]){};，则 int a[N]会被转换为 int *a;，因此函数形参并未与 N 建立起任何关联，从而也就无法使用模板实参推演。

⑩ 对于有默认值的模板形参，在进行全局特化时，模板实参是可选的。比如 template<class T1, class T2=int> class A{};，其全局特化形式有两种，即 template<> class A<int>{}; 和 template<> class A<int,float>{};，其中第一种形式的模板形参 T2 使用默认值。注意：虽然第一种形式只特化了第一个模板形参 T1，但其仍是全局特化，而非局部特化。

⑪ 在函数模板的全局特化版本中函数形参不能拥有默认值，比如 template<class T> void f(T x){};，则 template<> void f<int>(int x=3){}错误。

⑫ 注意：模板全局特化的 template<>前缀不能出现在模板形参列表之后，比如 template<class T> template<>...是错误的语法，这意味着不能只全局特化成员而不全局特化外围类。比如 template<class X> class A{template<class Y> void f(){}};，则 template<X> template<> void A<X>::f<int>(){};错误，此处试图只想特化成员函数模板 f 而不特化外围类模板 A。

3. 局部特化格式及基本语法要求

（1）重要规则：只有类模板才支持局部特化，函数模板是不能进行局部特化的。

（2）局部特化格式：template<不需要要被特化的模板形参列表> class 类名<不需要被特化的模板形参名列表，需要被特化的模板实参列表>{...};

示例如下：

```
template<class X, class Y> class A{};
template<class X> class A<X, int>{};  //模板形参 Y 局部特化为 int 类型，而 X 不被特化
```

（3）局部特化基本语法要求如下。

① 局部特化就是针对模板的部分模板形参进行特化。

② 局部特化时，把不需要被特化的模板形参放在 template 后面的尖括号中，在类名后的尖括号中的模板实参列表列出需要被特化的模板形参的实际类型，不需要被特化的形参位于类名后的尖括号中用作占位符。比如局部特化 template<class X> class A<X,int>{};，其中 X 是不需要被特化的模板形参，位于 template 后的尖括号中，在类名 A 后的 X 被用作占位符，int 表示把需要被特化的模板形参特化为 int 类型。

③ 局部特化的模板实参应与主模板的模板形参一一匹配。比如 template<class X, class Y> class A{};，则 template<class X> class A<X,2>{};错误，因为 2 与主模板中的 class Y 不匹配。

④ 模板形参名与局部特化：因为模板形参名可以省略或不同，因此在局部特化时，是由类名之后的模板实参和主模板形参的顺序相对应来决定特化的是哪个模板形参，而不是由局部特化时 template 后的模板形参名或顺序决定的。比如 template<class X,class Y,class Z> class A{};，则 template<class Y>class A<int,int,Y>{};特化之后主模板的模板形参 X=int, Y=int, Z 未被特化。此处并不是表示对主模板中的模板形参 Y 不特化，而应按 A<int,int,Y>中的顺序与主模板<class X, class Y, class Z>的顺序相对应来判断特化或未特化的模板形参。

⑤ 创建局部特化模板的对象的方法与创建主模板或通用模板的对象相同，但创建对象时的模板实参应与被局部特化的模板实参相对应，否则是使用主模板创建的对象。比如 template<class X, class Y> class A{};template<class X> class A<X,int>{};，则 A<int,int> ma;或 A<float,int> ma1;是使用局部特化版本 A<X,int>创建的对象；而 A<int,float> ma2;或 A<int, double> ma3;是使用主模板版本 A<X,Y>创建的对象。

⑥ 局部特化时，在类名后总是跟着模板实参表。

⑦ 局部特化时，模板形参不能有默认值，它将使用主模板的默认值。注意：此时类名后的

模板实参列表可能会与 template 之后的“不需要被特化的模板形参列表”在数量上相等。比如 template<class T,class X, class Y=int> class A{};，则 template<class X,class Y=int>class A<int,X>{};错误。再比如 template<class X, class Y=int> class A{};，则 template<class X> class A<X>{};正确，模板形参 Y 使用默认值 int 进行局部特化。

⑧ 局部特化时的模板形参表可以与主模板的模板形参表相同。比如 template<class T> class A{};，则 template<class T> class A<T*>{};正确，表示对任何的指针类型进行特化处理。再比如 template<class T> class A<T&>{};template<class T> class A<const T>{};都正确。

⑨ 局部特化时的模板形参的个数可以比主模板的模板形参的个数多或者少。比如 template<class X> class A{};，则 template<class X, int N> class A<X [N]> {};正确。再比如 template<class X, class Y> class A<Y X::*>{};正确，其中 Y X::*表示这个类型是指针，这个指针指向类型为 X 中的一个成员，该成员的类型为 Y，即 Y X::*表示的是指向类中成员的指针。创建以上局部特化对象的方式是使用 typedef 对类型进行重命名，然后使用该重命名的类型名作为模板实参去创建对象。比如 typedef int C[4]; A<C> ma;正确，A<int,4> ma;是错误的。

4．模板特化基本规则

以下规则适用于全局特化与局部特化。

（1）重要规则 1：当程序中有对特化版本的使用需求时，程序会优先使用模板的特化版本，而不会使用通用版本。比如 template<class T>void f(T){}; template<> void f<int>(int){};，则调用 f(3)或 f<int>(3)都会调用函数 f 的全局特化版本。

（2）在使用特化版本时，在实参和形参之间是允许进行一些转换的，这些转换与模板实参推演时所允许的转换相同，即允许左值转换、const 限定转换和到一个基类的转换。比如 template<class T> void f(T){};template<> void f<int*>(int *){};int a[3]={0};，则 f(a)会调用 f 的 int* 特化版本，这里从数组到指针的转换是允许的。

（3）特化的函数版本可以和重载的相同普通函数版本同时存在，因为全局特化和局部特化并不会创建一个新的模板或者模板的实例，它们只是为通用模板（即非特化模板）中已经隐式声明的实例提供另一种定义。这一点是与重载（函数）模板的主要区别。比如 template<>void f<int>f(int){};void f(int){};，两个 f 函数可同时存在。

（4）重要规则 2：在同一个程序中不能同时存在模板的特化体和由通用模板生成的相同实例。比如 template<class T> class A{};A<int> ma; template<> class A<int>{}; 错误，因为使用 A<int> ma;创建的实例与类 A 的 int 类型特化体位于同一个程序中。此处应注意 A<int>ma;的位置，若位于特化体之后将使用特化体创建 ma，而不会使用通用模板创建实例。

（5）重要规则 3：全局特化版本和局部特化版本的实现与通用（非特化）版本的实现之间没有任何关联。比如 template<class T>class A{int f;};，其特化版本可以是 template<>class

A<int>{void f(){}};，在通用模板中把名称 f 定义为 int 类型变量，但在特化版本中把名称 f 定义为一个函数，可以看到，特化版本的内容与通用版本的内容之间是没有任何关联的。

（6）虽然特化版本和通用版本之间没有任何关联，但还是建议类的特化版本应与类模板的通用版本有相同的成员定义，如果不相同，那么当类的特化对象访问到类模板的通用版本的成员时就会出错。比如 template<class T> class A{public:void f(){}void g(){}}; template<>class A<int>{public:void f(){}};A<int> ma;，则 ma.g();错误，因为在类 A 的 int 类型特化版本中没有成员函数 g。

5. 特化与声明、定义

（1）与通用模板相同，特化模板的声明或定义不能在局部范围或函数内进行。

（2）可以对模板的全局/局部特化版本只进行声明，而不定义。在进行声明时只是省略了大括号及其中的内容，其他形式与定义特化版本相同。比如 template<> void h<int>(int a);就是对模板 h 全局特化版本的声明。注意：声明时后面有个分号。

（3）在定义（或声明）特化模板时，必须要见到一个通用模板的声明（可以没有定义）。若这时有针对特化模板的使用，则必须要有特化模板的定义；通用模板可以没有定义，若有定义，则该定义并不会对特化模板产生影响。比如 template<class T> class A; template<> class A<int>{};A<int> ma;正确，此处只有对类 A 的通用模板的声明而无定义，但仍可对类 A 的 int 类型特化模板进行定义；对该特化模板的使用（即 A<int> ma）;也是正确的，因为此时只需要该特化模板的定义即可，不会使用类 A 的通用模板定义。再比如 template<class T>class A{};template<> class A<int>;A<int> ma;错误，此处使用类 A 的 int 类型特化模板，但该特化模板未定义，虽然通用模板已经定义了，但创建 ma 时不会使用通用模板的定义。

（4）当全局特化类模板的成员时，可以在类外对其只进行声明而不定义（局部特化不适用此规则）。注意：对常规类的成员，在类外只进行声明而不定义是错误的。比如 class A{void f();}; void A::f(); 错误，而 template<class T1,class T2> class A{void f();};template<> void A<int,int>::f(); 是正确的，template<class T1> void A<T1,int>::f();是错误的。

（5）在全局特化类模板时，在类外对静态成员变量进行声明的语法，与使用默认构造函数声明的静态数据成员在类外进行定义的语法是相同的，在 C++中对此语法只认为是声明而不是定义（局部特化不适用此规则）。比如 class A{};template<class T> class B{static T s;};　则 template<> A B<A>::s;是对静态成员 s 的全局特化声明语句，而非使用默认构造函数对 s 进行的定义，因此 s 仍未被定义，定义的形式为 template<> A B<A>::s=A();。

（6）不允许在一个文件中使用通用模板实例化某个实例，而在另一个文件中却使用这个实例的特化版本。详见下面示例。

```
//ab.h 头文件中的内容
template<class T> void f(T){};
```

```
//ac.cpp 文件中的内容
#include "ab.h"
template<> void f<int>(int){} void g(){ f(9);};
//主文件 ad.cpp 中的内容
#include "ab.h"
void main(){f(9);}//错误，对 f 的 int 类型版本的实例出现了重定义错误，调用 f(9)使用 ab.h
                                    //中的通用版本创建一个 int 类型实例，该实例与 ac.cpp 中的特化版本相同
```

（7）若一个程序位于不同的编译单元（即不同的.cpp 文件）中，则可以在头文件中使模板特化版本的声明位于通用模板的声明之后，以使模板的全局特化声明在使用该特化版本的每个编译单元中都可见。详见下面示例。

```
//ab.h 头文件中的内容
template<class T> void f(T){};
template<> void f<int>(int);  //增加对模板特化版本的声明
//ac.cpp 文件中的内容
#include "ab.h"
template<> void f<int>(int){}    void g(){ f(9);};
//主文件 ad.cpp 中的内容
#include "ab.h"
void main(){f(9);}   //正确，调用 f 的 int 类型特化版本
```

15.7.2 类模板成员的特化及定义

1. 不含嵌套类模板或成员函数模板的情形

（1）情形 1：在类外部定义全局特化版本中的成员函数时，应省略 template<>前缀，因为类模板已经被全局特化过，因此不能再使用 template<>前缀。但局部特化时不能省略 template<...>前缀。

示例 15.26：在类外定义类模板全局特化后的成员函数的方法

```
template<class T> class A{};
template<> class A<int>{public: void g(); void f();}; //全局特化类 A 时增加成员函数 f、g
void A<int>::f(){} //在类外定义类 A 的全局特化版本中的成员函数 f 的方法，省略了 template<>前缀
//template<> void A<int>::g(){}  //错误，应省略 template<>前缀
```

示例 15.27：在类外定义类模板局部特化版本中的成员函数的方法

```
template<class T1,class T2> class B{};
template<class T1> class B<T1,int>{public:void f();void g();};
template<class T1> void B<T1,int>::f(){} //正确，不能省略 template<...>前缀
//void B<T1,int>::g(){}  //错误，不能省略 template<class T1>前缀。T1 是不能识别的标识符
```

（2）情形 2：类模板不全局特化，直接全局特化类模板中的成员。在这种情形下不能再次对该类模板进行该类型的全局特化。注意：不能直接局部特化类模板中的成员。

示例 15.28：类模板不全局特化，直接全局特化其成员

```
template<class T1,class T2> class A{public:void f(){} };
```

```
template<> void A<int,int>::f(){} //类模板不全局特化，直接对类模板 A 的 int 类型特化版本中的成员 f
                                  //进行重定义的语法
//template<> class A<int,int>{}; //错误，不能再对类模板 A 的 int 类型版本进行全局特化
//template<class T1> void A<T1,int>::f(){}; //错误，局部特化不支持这种行为
```

（3）直接全局特化类模板中的某一个成员，其实是全局特化整个类模板的一种特殊情形。若类模板中有很多成员函数，而只有一、两个成员函数需要进行特殊化定义，其他成员函数定义保持不变，则使用此方法可减少特化整个类模板时对类模板中其他成员函数的重复定义。比如 template<class T> class A{void f(){cout<<"A";} void f1(){} void f2(){}};，假设程序要让函数 f 输出字符 B，而其他函数保持不变，则可以这样特化：template<> void A<int>::f(){cout<<"B";};。若特化整个类模板，然后在类模板中对函数 f 进行特殊化定义。则其他成员函数 f1 和 f2 都需要被重复定义。比如 template<> class A<int>{void f(){cout<<"B";}void f(){} void f2(){}};，可见，程序产生了冗余代码。

2. 含有嵌套类模板或成员函数模板的情形

（1）情形 1：在外围类外定义全局（或局部）特化版本中的成员函数模板、嵌套类模板、嵌套类模板中的成员函数。

示例 15.29：在外围类外定义全局（或局部）特化版本中的模板成员

```
template<class T1,class T2> class A{};
//外围类全局特化的情形
template<> class A<int,int>{template <class Y> class B;  template<class Z> void f();};
template<class Y> class A<int,int>::B{void g(); };   //在类外定义类 A 全局特化版本中的嵌套类 B，
                  //定义时应省略 template<>前缀（因为外围类 A 已被全局特化过）。A<int,int>应与全局
                  //特化类 A 时的 A<int,int>一致。此处增加了一个嵌套类成员 g
template<class Z> void A<int,int>::f(){}          //定义成员函数模板 f 的情形
template<class Y> void A<int,int>::B<Y>::g(){}  //在类外定义嵌套类 B 中的成员函数 g 的语法，应
                //省略 template<>前缀（因为外围类 A 已被全局特化过）。A<int,int>应与全局特化类 A 时的
                //A<int,int>一致，B<Y>应与 template<classY>中的 Y 一致，因为嵌套类 B 未被特化，因
                //此 template<class Y>不能省略
//外围类局部特化的情形
template<class T1> class A<T1,int>{template <class Y> class B;  template<class Z> void
f();};
template<class T1> template<class Y>
   class A<T1,int>::B{void g(); };    //在类外定义类 A 局部特化版本中的嵌套类 B
                                      //template<class T1>前缀不能省略
template<class T1> template<class Z> void A<T1,int>::f(){}//template<class T1>不能省略
template<class T1> template<class Y> void A<T1,int>::B<Y>::g(){}//在类外定义嵌套类 B 中的
                                      //成员函数 g 的语法，template<class T1>前缀不能省略
```

（2）情形 2：外围类模板不进行全局特化，而直接定义外围类全局特化版本中的模板成员，此时不能再次对外围类模板进行该类型的全局特化。在 VC++ 2010 中，不能直接定义外围类局部特化版本中的成员（包括嵌套类、成员函数模板）。

示例 15.30：直接定义外围类特化版本中的模板成员

```
template<class X> class A{public: template<class Y> class B{void h();};
                  template<class Z> void f(){}};
template<> template<class Y> class A<int>::B{void g();};//全局特化外围类A中的嵌套类模板B
                                                        //的语法
template<> template<class Z> void A<int>::f(){} //直接全局特化外围类A中的成员函数模板f的语法
//template<> class A<int>{};              //错误，不能再次对外围类模板进行int类型的全局特化
template<> template<class Y> void A<int>::B<Y>::g(){} //在类外定义被全局特化后的嵌套类B中
            //的成员函数g的语法，此处未省略template<>前缀，因为外围类A没有被全局特化。此处
            //A<int>应与全局特化类A时的A<int>一致，B<Y>应与template<class Y>中的Y一致
template<> template<> void A<float>::B<int>::h(){} //直接全局特化嵌套类模板B中的成员函数h
                          //的语法，这里有两个template<>前缀都没省略，一个表示外围类A未被
                          //全局特化；另一个表示嵌套类B未被全局特化
template<class T1,class T2> class B{public:template<class Y> class C{};};
//template<class T1> template<class Y> class B<T1,int>::C{};  //错误，在VC++ 2010中，不
                                    //能直接局部特化外围类中的成员（包括嵌套类、成员函数模板）
                                    //调用嵌套类模板成员函数的方法
void main(){
          //要访问嵌套类B中的函数g，在创建嵌套类B的对象时应查看定义函数g时的
          //模板实参头A<int>::B<Y>::g()，因此应按如下方法创建类B的对象
   A<int>::B<float> mab;          //mab.g();  //错误，g是私有的，但调用方法是正确的
   A<float>::B<int> mab1;//原理与调用g时相同
   //mab1.h(); //错误，h是私有的，但调用方法是正确的
}
```

说明：

由以上示例可见，对于嵌套类模板或成员函数模板本身并未被真正地特化（只是对它们进行了重定义），因为它们仍然适用于所有的模板实参类型，而不仅仅是某一个模板实参类型。比如 A<int> ma;，则 ma.f<int>();ma.f<float>();都是调用 void A<int>::f()这个版本，函数 f 本身适用于所有类型，但对外围类 A 则只适用于 int 类型。

（3）情形 3：直接对成员函数模板或嵌套类模板本身进行全局特化。

示例 15.31：对成员函数模板或嵌套类模板本身进行全局特化

```
template<class X> class A{public: template<class Y> void f(){}  template<class Z> class
B{}; };
template<> template<> void A<int>::f<float>(){}   //全局特化成员函数模板f本身的语法
template<> template<> class A<int>::B<float>{void g();};   //全局特化嵌套类模板B本身的语法
void A<int>::B<float>::g(){}
                          //在类外定义嵌套类B中的成员函数g的语法，此时应省略两个
                          //template<>前缀(因为外围类和嵌套类都已被全局特化。此处的
                          //int 和 float 应与特化时的 int 和 float 相对应
```

（4）使用类模板中模板成员的方法：首先查看该成员定义时的模板实参头，然后根据该实参头创建相应类型的对象，再使用该对象调用该成员，调用时再次查看该成员的定义，若在左边的 template 前缀中有一个 template 前缀与该成员对应，则说明该成员是模板。

示例如下：

```
template<> template<class Y> void A<int>::B<Y>::g(){}
```

应这样创建对象：A<int>::B<float>ma1;。调用函数 g 时查看定义时的 template 前缀，因为没有与 g 相对应的 template 前缀，因此 g 不是模板，直接调用即可，如 ma1.g()。

```
template<> template<class Z> void A<int>::f(){}
```

应这样创建对象：A<int> ma;。查看 template 前缀，发现 template<class Z>与 f 对应，说明 f 是模板，因此应按模板的方式调用，如 ma.f<int>();。

```
  template<> template<class Z> void A<int>::B::f(){}
```

应这样创建对象：A<int>::B ma。查看 template 前缀，因为 B 后面没有模板实参，因此 B 不与任何 template 前缀对应，那么多余的 template<class Z>应与 f 对应，说明 f 是模板，应像 ma.f<float>();这样调用。

含有嵌套类模板或成员函数模板时各种特化和定义语法总结，如图 15.1 所示。

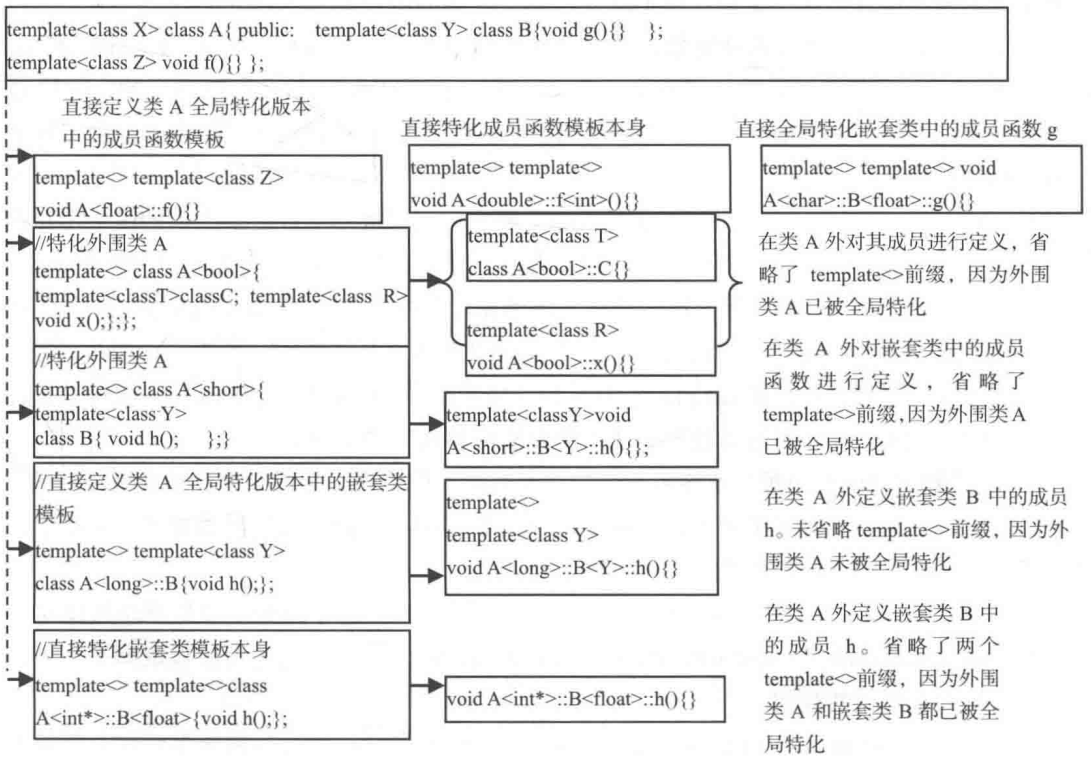


图 15.1　含有嵌套类模板或成员函数模板时各种特化和定义语法总结

15.8 有模板时的函数重载解析

1. 模板签名与重载

（1）重要规则：具有相同名称的两个函数模板，只要它们的签名不相同，即使它们实例化后拥有完全相同的函数形参类型和返回类型，也是可以共同存在的称作函数重载。比如template<class T> void f(T){}template<class T> void f(T*){}int *p=0;，则调用 f(p)后，f(T)被实例化为 f<int*>(int*)，f(T*)被实例化为 f<int>(int*)，虽然它们实例化后的函数形参和返回类型是相同的，但它们可以共同存在。

（2）函数模板签名包括（只要签名不同，这些模板就可以共存）：

① 非受限函数的名称。

② 函数名称所在的类作用域或名称空间作用域，若函数名称是内部链接的，则签名还应包括其所在的编译单元。也就是说，在不同作用域中声明的函数是可以共存的。

③ 若函数是某个类的成员函数，则签名还应包括该函数所带的 const、volatile 或 const volatile 限定词。也就是说，类中带有 const 与不带 const 的成员函数是可以共存的。比如 class A{public: void f(){}void f()const{}};，类中的两个 f 函数是可以共存的，注意 const 应位于函数名后的小括号后。

④ 函数形参的类型。若函数来自函数模板，则指的是模板形参被替换之前的函数形参的类型。也就是说，形参类型不同的函数是可以共存的。注意：仅仅是函数形参使用的模板形参名不同，并不能认为是不同的签名（因为模板形参名是可以省略的）。比如 template<class T1,class T2> void f(T1, T2){}; template<class T3, class T4> void f(T3, T4){};错误，因为两个模板函数 f 的形参具有相同的签名。再比如 template<class T1, class T2> void f(T1,T2); template<class T1, class T2> void f(T2, T1);正确，因为函数形参使用的模板形参的顺序不同。

⑤ 若函数来自函数模板，则签名还应包括函数的返回类型。此规则只对函数模板有效，对普通函数无效，这说明返回类型不同的函数是可以共存的。注意：若返回类型是模板形参，则仅仅是模板形参名不同的函数是不能共存的。比如：

- void f(); int f();错误，因为 f 是普通函数，因为返回类型不同的函数是不能共存的。
- template<class T> void f();template<class T> int f();正确，因为函数模板返回类型不同的函数是可以共存的。
- template<class T1> T1 f();template<class T2> T2 f();错误，因为返回类型仅仅是模板形参名不同的函数是不能共存的。

⑥ 若函数来自函数模板，则签名还应包括模板形参和模板实参。此规则只对函数模板有效，因为常规函数没有模板形参与模板实参。这说明模板形参或者模板实参不同的函数是可以共存的。注意：仅仅是模板形参名或顺序不同并不能认为是不同的签名，还需要模板形参的数量不

同。模板形参和模板实参位于尖括号中，而函数形参和函数实参位于小圆括号中。

示例如下：

```
template<class T> void f(){} template<class T1,class T2> void f(){} //正确
```

说明：两个函数模板形参的数量不同，可以共存。

```
template<class T>void f(){}template<class T1>void f(){}  //错误
```

说明：两个函数仅仅是模板形参名不同，不可以共存。

```
template<class T>void f(T){}template<class T> void f(T*){}  //正确
```

说明：两个函数的实参不同，可以共存。比如 int a=1;，则调用 f(&a)后，f(T)被实例化为 f<int*>(int*)， f(T*)被实例化为 f<int>(int*)，可见，实例化后的两个函数的模板实参列表不同，因此两个函数可以共存。

```
template<int N>void f(){}template<int *N> void f(){}  //正确
```

说明：若调用 f<3>()，则在 VC++ 2010 中会产生二义性错误，因为总是把 f<9>()和<int* N>相匹配。

⑦ 上面讲解的函数模板可以共存，但并不代表在实例化时不会产生二义性错误。比如 template<class T> void f(T){} template<class T> int f(T){return 1;}，则调用 f(9)会产生二义性错误，导致 void f(T)和 int f(T)无法在程序中使用。

（3）重要规则：由模板签名规则可推导出，函数模板产生的实例一定不会与常规函数是等同的，即使它们的类型和函数名称完全相同，它们也可以共存于同一个程序中。比如 void f(int){} template<class T> void f(T){} template<class T> void f(int){}，则 f(3)所产生的两个 int 类型实例与普通函数 f(int)是相同的，但这里的 3 个 f 函数是可以同时存在的。

（4）由上一条规则可得出如下重要结论。

① 由赋值运算符函数模板产生的赋值运算符函数实例不会取代默认的或自定义的常规赋值运算符。

② 由复制构造函数模板产生的复制构造函数实例不会取代默认的或自定义的常规复制构造函数。

③只要创建了自定义的构造函数，就不会再产生默认构造函数，因此若类中把构造函数定义为模板，则该构造函数模板会取代默认构造函数。

（5）当把类的赋值运算符重载为模板时，可使两种不同类型的该类对象进行相互间赋值；但这种方法并不会取代默认的赋值运算符（见上一条规则），相同类型的该类对象之间的赋值，使用的仍然是默认赋值运算符。详见下面示例。

示例 15.32：自定义的赋值运算符模板不会取代默认的赋值运算符

```
template<class T> class A{public:
    template<class T1> A<T1> operator =(T1 ma){A<T1> ma1; cout<<"A"; return ma1;}};
class B{};
void main(){
```

```
A<int> ma,ma2;A<float> ma1; B mb;
ma=ma1;    //正确，输出A，使用重载的operator =(T1)进行赋值。注意：A<int>与A<float>不是
           //相同的类型
ma=ma2;//正确，无输出，调用默认的赋值运算符，可见自定义的赋值运算符模板并没有取代默认的赋值运算符
ma=mb;}    //正确，输出A，调用重载的模板赋值运算符operator=(T1)
```

2. 有函数模板时的重载解析规则

（1）有函数模板时的重载解析也分为 3 个步骤：① 确定候选函数。② 确定可行函数。③ 对可行函数的匹配程度进行等级划分。

（2）有函数模板时候选函数的确定方法。

① 函数名称的识别（即可见性问题），请参阅 15.4 节和 15.5 节。

② 只有模板实参推演成功的函数模板实例才能加入到候选函数集中。比如 template<class T> void f(T,T){}; void f(int, float){}，则 f(3, 4.4)将调用普通函数，因为函数模板 f(T,T)的模板实参推演失败，它并未加入到候选函数集中。

③ 若模板实参推演成功后的实参被显式特化了，则将显式特化的函数加入到候选函数集中，通用模板的该版本实例不会加入到候选函数集中。比如 template<class T> void f(T, T){}; template<> void f<int,int>(int,int){};void f(float,float){};，则 f(3,3)将调用 f(T,T)的特化版本 f<int, int>，此时的候选函数集中只有 f<int, int>和普通函数 f(float, float)，通用函数模板的 int 类型实例不会加入到候选函数集中。

④ 只有模板实参推演成功，才会把模板的显式特化版本函数加入到候选函数集中。比如 template<classT>void f(T,T){};template<>void f<int,int>(int,int){}; void f(float,float){};，则 f(3, 4.4)将调用普通函数 f(float, float)。因为模板实参推演失败，因此通用模板函数和模板函数的 int 类型特化版本都不会加入到候选函数集中，在候选函数集中只有唯一的一个函数，即普通函数 f(float, float)。

（3）有函数模板时，对可行函数匹配程度的等级划分原则如下。

① 在进行类型匹配时，若其他各方面都是相同的，则普通函数优先于模板函数及特化的模板函数。

② 当完全匹配的函数都是模板函数时，更特化（或更具体）的模板函数优先。更特化的判断依据是“部分排序规则（Partial Ordering Rules）”。

③ 可以使用空的模板实参列表（或显式的模板实参列表）让编译器强制调用模板函数，而不调用匹配得一样好的非模板函数。比如 void f(int){} template<class T> void f(T){};，则 f<>(9)或 f<float>(9)都会调用模板函数 f(T)，而不会调用非模板函数 f(int)。

④ 部分排序规则：在完全匹配的情况下，此规则用于判断哪个函数模板更特化，或者说用于选择哪个函数模板匹配得更好。注意：使用此规则的前提条件是，无法区分谁是最佳匹配。有两种方法可判断谁更特化，即交叉互换法和子集法。

交叉互换法：

- 把同名的函数模板的名称重命名为 f1 和 f2。重命名主要是为了便于分析。
- 合成两组虚构的模板实参类型，其方法是使用其他名称去替换每个函数形参中的模板类型形参，替换之后的名称就是虚构的模板实参。说简单点，就是对函数形参中的模板类型形参进行重命名。比如 template<class T> void f(T, T*)(){}，假设以 A 作为虚构的模板实参名称去替换函数形参中的模板类型形参(T,T*)，替换之后为(A, A*)，那么(A, A*)就是替换之后虚构的模板实参，这里的 A 和 T 的性质相同，都是代表任意类型，其实这里只是对 T 的重命名。

采用交叉互换法判断谁更特化的方法如下。

- 在两个函数模板（假设为 f1、f2）与虚构的实参（假设为 A1、A2）之间进行相互交叉的模板实参推演。
- 使用虚构的第一组模板实参 A1 调用第二个函数模板 f2 并进行模板实参推演，即使用 f2(A1)对 f2 中的 T 进行推演。推演时应把 A1 代表的所有实参类型都考虑在内。
- 使用虚构的第二组模板实参 A2 调用第一个函数模板 f1 并进行模板实参推演，即使用 f1(A2)对 f1 中的 T 进行推演。推演时应把 A2 代表的所有实参类型都考虑在内。
- 若第二个函数模板 f2 推演成功，而第一个函数模板 f1 推演失败，则第一个函数模板比第二个函数模板更特化或更具体（即匹配得更好）。
- 反之，若第一个函数模板 f1 推演成功，而第二个函数模板 f2 推演失败，则第二个函数模板比第一个函数模板匹配得更好。
- 若第一个函数模板和第二个函数模板都能推演成功或推演失败，则两个函数模板之间没有谁优谁劣之分（即不存在排序关系），因此调用出现二义性错误。
- 注意：推演时 A1、A2 代表的所有实参类型都必须推演成功才能算成功。

子集法：

此方法相对要简单一些，即若第一个函数模板所能接受的所有类型是第二个函数模板所能接受的类型的子集，但反过来不对，则第一个函数模板更特化，或者说接受类型更有限的函数模板更特化。注意，根据互为子集或相互不为子集都不能判断谁更特化。

示例 15.33：部分排序规则的使用（交叉互换法和子集法）

```
template<class T> void f(T){} template<class T> void f(T*){}
```

谁更特化分析如下。

① 把 f(T)的函数名称重命名为 f1(T)，把 f(T*)的函数名称重命名为 f2(T*)。

② 虚构两组模板实参，由 f1(T)虚构出第一组模板实参为(A1)，由 f2(T*)虚构出第二组模板实参为(A2*)。

③ 交叉使用虚构的模板实参调用函数模板并进行模板实参推演，即使用 f2(A1)对 f2(T*)进行模板实参推演，很明显无法推导出 T 的类型。而使用 f1(A2*)对 f1(T)进行模板实参推演成功（推导出 T=A2*），因此函数模板 f2(T*)匹配得更好，即 f(T*)是更特化版本。

④ 子集法：显然 f(T*)所能接受的类型是 f(T)所能接受的类型的子集，因此 f(T*)更特化。

具体调用示例如下：

① f(0)，调用 f(T)，对于 f(T*)不能由 f(0)推导出正确的类型，因此只能调用 f(T)。

② f((int*)0)，调用 f(T*)，对于 f(T)和 f(T*)都能生成完全匹配的实例 f<int*>(int*)和 f<int>(int*)，但由上面知 f(T*)更特化，因此调用 f(T*)。此时使用部分排序规则。

③ f<int*>((int*)0)，调用 f(T)，对于 f(T)生成的实例为 f<int*>(int*)，对于 f(T*)生成的实例为 f<int*>f(int**)，很明显 f<int*>((int*)0)的调用与 f(T*)的类型不匹配。此时不会使用部分排序规则。

④ f<int>((int*)0)，调用 f(T*)，对于 f(T)生成的实例为 f<int>(int)，很明显与 f<int>((int*)0)的类型不匹配；对于 f(T*)生成的实例为 f<int>(int*)，属于完全匹配，因此调用 f(T*)。此时不会使用部分排序规则。

示例 15.34：部分排序规则的使用（交叉互换法和子集法）

```
template<class T> void f(T*){}template<class T>void f(const T*){}int *p;const int *p1;
```

谁更特化分析如下。

① 把 f(T*)的函数名称重命名为 f1(T*)，把 f(const T*)的函数名称重命名为 f2(const T*)。

② 由 f1(T*)虚构出模板实参(A1*)，由 f2(const T*)虚构出模板实参(const A2*)。

③ 交叉调用函数模板并进行模板实参推演，即使用 f2(A1*)对 f2(const T*)进行模板实参推演，此处推演失败，因为对 A1 所代表的所有类型的实参推演失败。比如，若 A1 是一个指向函数类型的指针，则由 A1*到 const T*的推演就是失败的。使用 f1(const A2*)对 f1(T*)进行模板实参推演成功，推演时会忽略 const，因此 T=A2，f(const T*)更特化。

④ 子集法：显然 f(const T*)是 f(T*)的子集，因此 f(const T*)更特化。

具体调用示例如下。

① f(p)，调用 f(T*)，对于 f(T*)和 f(const T*)都可以生成完全匹配的实例 f<int>(int *)和 f<int>(const int*)，但根据有指针时的完全匹配规则（见第 7 章），没有 const 比有 const 匹配得更好，因此 f(p)调用 f(T*)。此时不会使用部分排序规则。

② f(p1)，调用 f(const T*)，对于 f(T*)和 f(const T*)都可以生成完全匹配的实例 f<const int>(const int*)和 f<int>(const int *)，它们匹配得一样好，但由上面的分析可知 f(const T*)更特化，因此调用 f(const T*)。此时使用部分排序规则。

示例 15.35：确定候选函数集

```
template<class T> void f(T){};template<class T> void f(int){}
```

① f(3)，调用 f(T)，因为 f(int)无法由 f(3)的调用推演出模板形参 T 的类型，因此模板实参推演失败，f(int)不会加入候选函数集中。

② f<int>(4)，调用 f(int)，两个函数匹配得一样好，很明显 f(int)是 f(T)的子集，因此调用 f(int)。

示例 15.36：子集法的直观判断可能会产生错误

```
template<class T> void f(T,T){};  template<class T> void f(T,int){}
```

谁更特化分析如下。

① 把 f(T, T)的函数名称重命名为 f1(T, T)，把 f(T, int)的函数名称重命名为 f2(T, int)。

② 由 f1(T, T)虚构出模板实参(A1, A1)，由 f2(T, int)虚构出模板实参(A2, int)。

③ 由交叉互换法可知，f1(A2,int)无法为 f1(T,T)推导出模板形参 T（因为若 A2 为非 int 类型，则 T 的类型就不一致了），而 f2(A1,A1)也无法为 f2(T, int)推导出模板形参 T 的类型（因为若 A1 为指针等类型，则无法与 int 类型进行匹配，这时模板实参推演会失败），因此两个函数匹配得一样好。

④ 子集法：直观上判断 f(T,int)是 f(T,T)的子集，但实际上并非如此，因为 f(T,int)能接受两个类型不相同的实参，比如 f(double,int)，很明显 f(T,T)不行；而 f(T,T)也不是 f(T,int)的子集，比如对于 f(T,T)中的 f(int*, int*)，很明显就不是 f(T,int)的子集。因此 f(T,T)与 f(T,int)互不为子集，它们没有谁比谁更特化。

具体调用示例如下。

① f(3, 4)，调用不明确，因为由 f(3,4)产生的两个函数实例匹配得一样好，由以上交叉互换法的分析可知，这两个函数没有谁优谁劣之分，所以出错。

② f<int>(3,4)，调用不明确，原因同上。

③ f<float>(1.2, 4)，调用 f(T,int)。由 f(T,T)产生的实例为 f<float>(float, float)，可见两个形参都要进行标准转换；由 f(T,int)产生的实例为 f<float>(float,int)，只有第一个形参需要进行从 double 到 float 的转换，第二个形参是精确匹配，因此 f(T,int)匹配得更好，结果调用 f(T,int)。因为两个函数的实例在匹配等级上并不相同，因此不会使用部分排序规则。

④ f<double>(1.2, 4.2)，调用 f(T,T)，分析方法同③。

⑤ f<float>(1.2, 3.2)，调用不明确，分析方法同③。

⑥ 注意：有多个形参时的匹配规则是，函数的所有形参匹配等级都不低于另一个函数的所有形参，则这个函数的匹配等级更好。

3. 有函数模板时的二义性错误及一些调用规则

（1）当对函数模板进行取址时也可能产生二义性错误。比如 template<class T> void f(T){} template<class T> void f(T*){}template<class T> void g(T){}，则 g(&f<int>)将产生二义性错误，因为对于 f<int>来讲，f(T)和 f(T*)都是完全匹配的。再比如 template<class T> T f(T){}; void g(int (*p)(int)){}void g(float (*p)(float)){}，则 g(&f)错误，因为对于&f 而言，两个 g 函数版本都是正确的，所以存在调用二义性错误。可使用显式模板实参的方式来解决这种二义性错误，比如 g(&f<int>)或 g(&f<float>)都是正确的。

（2）在拥有重载的函数模板时，虽然模板形参替换失败了，但程序并不是错误的。其实此规则在确定有函数模板时的候选函数时就已经讲过，即若模板实参推演失败（即形参替换失败）了，只是该模板函数不能成为候选函数，但程序是没有错误的。

15.9 模板与友元

15.9.1 基础

关于非模板类的友元和友元声明时其名称的可见性问题请参阅第 11 章。

1. 基本规则

（1）友元函数的实际声明位置及可见性：友元函数虽然是在类中声明的，但它的实际声明位置是在包含该友元函数的类所在的最近的名称空间（有可能是全局作用域）中。若友元函数所属的类属于 ADL 的关联类集合，则该友元函数的名称在该名称空间中是可见的，否则不可见。由此可知，友元声明的位置不代表可见性。比如 class A{friend void f(){}friend void f(A*){} };A ma; f(); f(&ma)，表面上看，友元函数 f()和 f(A*)声明于类 A 之中，但实际的声明位置位于包含该友元函数的类（即类 A）所在的最近的名称空间（全局作用域中）。其中，调用 f(&ma)是正确的，因为友元函数 f(A*)所属的类 A 属于 ADL 的关联类集合，该名称在全局可见；而 f()是错误的，因为 f()所在的类 A 不属于 ADL 的关联类集合，其实 f()无实参，根本就未启动 ADL 机制。但对 f 进行声明之后再调用则是正确的，比如 void f()，则 f()正确。

（2）注意：当把类声明为友元时不能对其进行定义。比如 class A{friend class B{};};，其中对友元类 B 的定义是错误的。

（3）把友元函数定义于类模板之中，有可能会在创建多个类模板的实例时引发重定义错误。比如 template<class T> class A{friend void f(){}};，则 A<int> ma; A<float> ma1;会出现错误，因为在类模板 A 的 int 和 float 实例中对同一个函数 f 进行了两次定义。

2. 约束型友元模板和非约束型友元模板

（1）可以建立两种类模板的友元模板，即约束型友元模板和非约束型友元模板。

（2）非约束型友元模板：类模板的友元模板的任一实例都是外围类的任一实例的友元，即外围类和友元模板之间是多对多的关系。比如 template<class T>class A{template<class T1> friend void g(T1 a); template<class T2> friend class B;};，非约束型友元模板和外围类具有不同的模板形参。注意友元模板类的声明，在类名 B 的后面没有尖括号。

（3）约束型友元模板：类模板的友元模板的一个特定实例只是外围类的相关一个实例的友元，即外围类和友元模板之间是一对一的关系。比如：

```
template<class T1> void g(T1 a); template<class T2> void g1(); template<class T3>class B;
template<class T>class A{friend void g<>(T a); friend void g1<T>(); friend class B<T>;};
```

约束型友元模板和外围类具有相同的模板形参。此处建立了三个约束型友元模板，其中 g 和 g1 是函数，而 B 是类。这里 g<int>和类 A<int>相对应，g<double>和 A<double>相对应，它们是一对一的关系。

15.9.2　把模板或其实例声明为友元

（1）把类模板或函数模板的某个实例声明为类或类模板的友元时，在声明之前该模板必须可见（即已被声明或定义），这是与常规函数和类不同的地方。比如 template<class T> class D; class A{friend class B; friend class C<int>;friend class D<int>;};，其中类模板 C 的友元声明是错误的，因为在声明之前类 C 不可见；常规类 B 和类模板 D 的声明是正确的。

（2）有两种方法可以把模板的某个实例声明为友元，即使用尖括号和作用域解析运算符。

1. 使用尖括号把函数模板的实例声明为友元

（1）使用尖括号把函数模板的实例声明为友元时，其函数名称后面必须要有以尖括号括起来的模板实参列表，即使是空的，也不能省略。

（2）若模板实参能由编译器推导出来，则尖括号中的模板实参列表可以为空，但尖括号必须保留。比如 template<class T> void f(T); class A{friend void f<>(int); friend void f(double);};，其中 f 模板的 int 类型版本 f<int>(int)被声明为友元，f(double)只是把普通函数声明为友元，并未把 f 模板的 double 类型版本声明为友元。

（3）把函数模板的实例声明为友元时模板实参推演使用的实参应是具体的类型，而不是某个数值。比如 template<class T> void f(T); class A{friend void f<>(3); friend void f<>(double);};，其中 f<>(3)的声明是错误的，f<>(double)的声明是正确的。

（4）不能对声明为友元的函数模板的实例进行定义。注意：此处并没有说不能对模板的显式特化体和把整个函数模板都声明为友元的函数模板进行定义。比如 template<class T> void f(T);

class A{ template<> friend void f(char){} friend void f<int>(int){}};，其中 f<int>(int)的定义是错误的，f 的特化版 f(char)的定义是正确的。

（5）类模板必须使用尖括号把某个实例声明为友元。

2．使用作用域解析运算符把函数模板的实例声明为友元

（1）若声明的友元函数名称后面没有尖括号，也没有被作用域解析运算符（::）修饰，则表示把一个常规函数声明为友元，而不会把相同名称的函数模板声明为友元。若该常规函数没有被定义，那么在该友元声明处可以被定义。比如 template<class T> class A{ friend void f(int){}};，表示把常规函数 f 声明为友元，并可对其进行定义。

（2）若声明的友元函数名称后面没有尖括号，但函数名称被作用域解析运算符（::）修饰（指的是形如::f 这样的用法），则这个名称就一定会代表（引用）一个函数或者一个函数模板，这时常规函数会优先于函数模板被匹配。也就是说，此方法可以把函数模板的某个实例声明为友元，只是它的优先级没有普通函数高。同时，这样声明的友元不能被定义，模板形参通过模板实参推演来确定。

示例如下：

```
template<class T> void f(T); class A{friend void ::f(int);};
```

表示把模板函数 f 的 int 类型版本声明为友元，f 的模板形参 T 通过模板实参推演来确定。

```
template<class T> void f(T); void f(int); class A{friend void ::f(int);};
```

表示把常规函数 f(int)声明为友元，因为使用作用域解析运算符声明友元时，常规函数的优先级高于函数模板实例。

```
template<class T> void f(T); class A{friend void ::f(int){}};
```

错误，把函数模板实例声明为友元时不能对其进行定义。

（3）因为类模板无法进行模板实参推演，因此它的实例不能仅仅使用作用域解析运算符声明为友元，必须使用尖括号把某个实例声明为友元，因为不使用尖括号无法为类模板指定模板实参。

3．把整个模板声明为友元

（1）把整个类模板声明为友元的形式为：template<模板形参列表> friend class 类名;，比如 class A{template<class T> friend class B;};。

（2）把整个函数模板声明为友元的形式为：template<模板形参列表> friend 返回类型 函数名(函数形参列表);，比如 class A{template<class T> friend void f(T);};。

（3）注意：在声明为友元时 friend 关键字应位于 template 之后。

（4）把整个模板声明为友元时，不需要见到其相应的前向声明或定义。这是与把模板的某个实例声明为友元不同的地方。

（5）把整个模板声明为友元时，在类名或函数名后不能再有用尖括号括起来的模板实参列

表。比如 template<class T> friend class B<int>;，不管该语句是在类外还是类内都是错误的。

（6）与把模板实例声明为友元一样，只有当友元模板是一个非受限的函数名称，且名称后面没有尖括号时，才可以对其进行定义。注意：对于声明的友元类，不管该友元类是模板还是非模板都不能在声明友元时对其进行定义。比如 class A{template<class T> friend void f(T){}};，其中对 f 的定义是正确的。再比如 class A{template<class T> friend class B{};};错误，不能在声明友元类时对其进行定义。

（7）若友元模板声明的是主模板，则任何与该主模板有关的局部特化体和全局特化体都会自动成为友元。比如：

```
class A{int a;
template<class T> friend void f(T);}; //函数模板 f 的所有版本自动成为类 A 的友元
template<class T> void f(T){};
template<> void f<int>(int){A ma; ma.a=3;} //其中对类 A 的私有成员 a 的访问正确
```

15.10　模板与继承

（1）从模板类派生出子类时，子类并不会从通用的模板父类继承而来，只会从父类的某一实例继承而来。因此，有以下几种继承方式。

① 继承时父类是非受控（依赖）的，即子类继承自父类的特定实例。比如 template<class T1>class B:public A<int>{};。

② 继承时父类是受控（依赖）的，即父类是一个和子类相关的实例。比如 template<class T1>class B:public A<T1>{};，当实例化子类 B 时，父类就相应地被实例化为一个和子类相同的实例版本，如 B<int>m，模板类 B 被实例化为 int 版本，这时基类 A 也相应地被实例化为 int 版本。

③ 如果父类是一个特定实例版本，这时子类可以不是模板，比如 class B:public A<int>{}。

（2）非受控父类的名称查找规则：若继承时的父类是非受控的，则对于一个非受限名称，编译器首先查找的是非受控（依赖）型父类中的相同名称，然后才会查找子类模板形参中的名称。

示例 15.37：非受控父类的名称查找规则

```
class A{public: typedef int* T;};
template<class T> class B:public A{public:
    T i;};    //此处的 T 并不是类 B 的模板形参 T，而是类 A 中的 typedef int *T;，查找名称 T 时首先
              //在父类 A 中进行查找，然后才会查找子类 B 的模板形参中的名称
void main(){B<float> mb;
//mb.i=4;       //错误，i 的类型为 int *，而非 float 类型
}
```

（3）受控父类的名称查找规则：只要编译器见到非受控名称，就立即进行名称查找，但不能在受控的父类中查找，对于受控名称的查找一般发生在该名称被实例化之后。（注意：VC++ 2010 不支持这条规则。）

示例 15.38：受控父类的名称查询规则

```
template<class T> class A{public: int a;};
template<class T>class B:public A<T>{public:void f(){ a=1;} };
B<int> mb;  //若编译器支持C++标准，则在a=1语句处会报出错误。因为按照标准，对于非受控名称a
            //会立即进行查找，而且不能在父类中查找，因此出错
```

第 16 章

I/O 专题

C++标准中的 I/O 类都是用模板编写的，因为模板名称太长，再加上模板本身的复杂性，使用模板的形式进行讲述很不方便，因此本文使用的都是模板简化后的版本。

16.1 I/O 流模型及 I/O 类组织结构

16.1.1 I/O 流模型

（1）流：C++把输入输出的过程比喻为流水（数据）从一条管道的一端流向另一端，而 C++中的输入输出就是为适应各种流水（数据）提供的管道，这条管道在 C++中被称为流。

（2）使用流进行输入输出：在输入时，数据从输入设备（比如键盘等）经过 C++提供的输入管道（输入流）流向程序；在输出时，数据从程序经由 C++提供的输出管道（输出流）流向输出设备（比如显示屏等）。C++的 I/O 流在输入输出两端建立起一条通道。

（3）建立 I/O 流的原理：要使用 C++的 I/O 流，就需要在流与输入输出两端建立起联系。

（4）I/O 流的种类：C++根据不同的数据来源，创建了各种类型的流——若数据来源于字符串，则创建专门的字符串输入输出流（简称“字符串流”）；若数据来源于文件，则创建专门的文件输入输出流（简称“文件流”）等。

（5）可见，流就是一个管道，铺好了这条管道也就实现了输入输出，因此实现了流就是实现了输入输出。

（6）流与缓冲区。数据直接从一端流向另一端有时是比较低效的。比如磁盘设备一般都是以 512 个字节为单位进行读写和传输的，而程序一般从磁盘一次读取一个字节，处理后再读取下一个字节，512 个字节就需要读取 512 次，从而产生大量的机械活动（比如移动磁盘头、寻找磁道等），导致速度很慢，使用缓冲区可以解决这个问题。缓冲区一般是内存、寄存器等比硬盘速度快 N 倍的存储器，每次从磁盘读取大量的数据存入缓冲区，然后程序每次从缓冲区中读

取一个字节，这样就解决了读取磁盘低效的问题。把数据写入磁盘的原理与此类似，先把数据写入缓冲区，然后再一次性把数据写入磁盘。

（7）C++使用类来实现流。也就是说，C++使用类来实现输入输出，因此 C++把与输入输出有关的类称为流，每一个类对应处理一种数据来源。比如对于标准输入流，C++使用 istream 类来实现；对于标准输出流，使用 ostream 类来实现；处理文件输入的流使用 ifstream 类实现；处理字符串输入的流使用 istrstream 类来实现等。

（8）对于缓冲区的管理，C++也是使用类来实现的，这个类是 streambuf。

（9）因为 C++是使用类来实现输入输出的，所以 C++标准库中的 I/O 类拥有普通类的性质，要使用这些 I/O 类就必须创建该类类型的对象，然后通过对象调用类中的成员函数或成员变量进行 I/O 操作。比如经常使用的 cout 就是由 ostream_withassign 类创建的对象，cin 是由 istream_withassign 类创建的对象，它们经常调用的是被重载的“<<”和“>>”操作符函数，即重载的“>>”和“<<”操作符函数中包含了怎样对数据进行输入输出的代码。当然，也可以调用 I/O 类中的其他函数进行输出输入，比如 get、put 等。

C++ I/O 流模型原理图如图 16.1 所示。

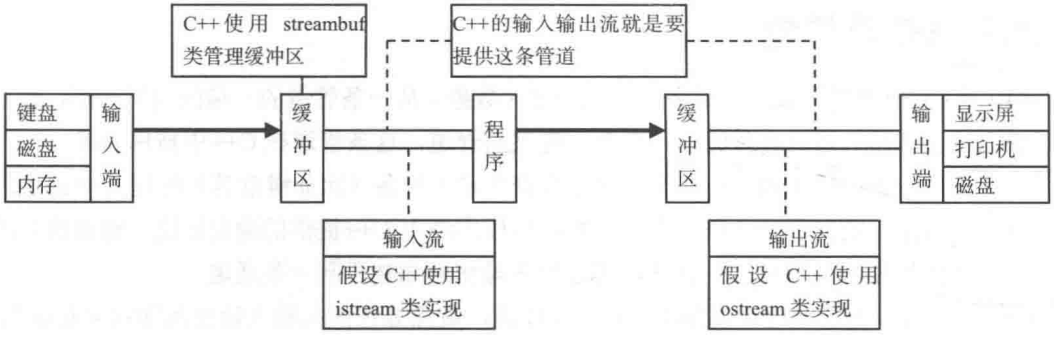


图 16.1 C++ I/O 流模型原理图

16.1.2 I/O 类组织结构

1. I/O 类的继承结构

C++标准把用于输入输出各种数据的类编写在一个 I/O 库中，并在各种头文件中对这些类进行了声明。这些类是通过各种继承关系来实现输入输出的，其继承关系见图 16.2。

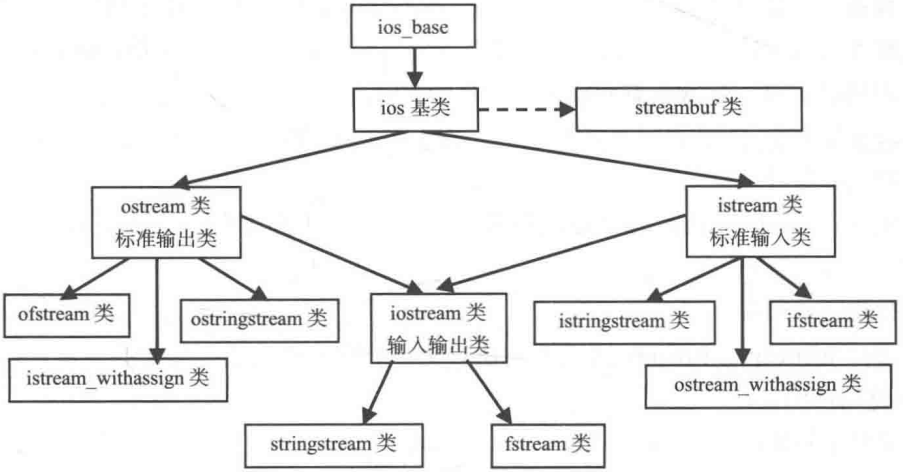


图 16.2　C++中的 I/O 类

2. C++的各种输入输出流（类）简介

（1）独立于输入输出的类。

① streambuf 类——用于对缓冲区进行管理。

② ios_base 类——这是一个比较新的类，从 ios 中提取出独立于 I/O 类型的信息。

③ ios 类——是最基本的其他 I/O 类的父类，该类拥有一个指向 streambuf 类类型对象的指针成员。

（2）标准输入输出类。

① ostream、istream、iostream 这三个类是用于处理常规数据的输入输出流，其中 ostream 类的主要工作是将需要输出的不同类型的数据以字符形式展现在输出设备上，比如浮点数 3.15 中的小数点必须以字符形式输出；istream 类则执行相反的工作。

② 要使用这三个类，需要包含头文件 iostream。注意，这里的 iostream 是头文件，而不是 iostream 类。

③ 在头文件 iostream 中，不仅声明了 I/O 类，还定义了常用的 I/O 流对象。

- cout 流对象——代表标准输出的流对象，因此把 cout 流对象称为标准输出流或简称为 cout 流。cout 流对象被关联到标准输出设备（一般为显示屏）上，由 ostream_withassign 类创建，该类是 ostream 的子类。
- cin 流对象——代表标准输入的流对象，因此把 cin 流对象称为标准输入流或简称为 cin 流。cin 流对象被关联到标准输入设备（一般为键盘）上，由 istream_withassign 类创建，该类是 istream 的子类。
- cerr 流对象——对应于标准错误流，用于显示错误信息。cerr 流对象被关联到标准输出

设备（一般为显示屏）上，由 ostream_withassign 类创建。cerr 流对象在输出错误信息时不会被缓冲，而是直接输出到屏幕。cerr 与 cout 的唯一区别是，cerr 只能把信息输出到显示器，而 cout 则可以把信息重定向输出到其他地方（比如磁盘）。

- clog 流对象——对应于标准错误流。clog 与 cerr 的唯一区别是，clog 在输出错误信息时会被缓冲。
- wcout、wcin、wcerr、wclog 流对象——其功能与上面 4 个流对象相似，只不过它们用于处理宽字符。

（3）处理文件的类。

ifstream、ofstream、fstream 这三个类是对文件的输入输出进行处理的流，它们在头文件 fstream 中进行声明。

（4）处理字符串的类。

① istringstream、ostringstream、stringstream 这三个类是用于处理 string 类型数据的输入输出流，它们在头文件 sstream 中进行声明。

② istrstream、ostrstream、strstream 这三个类是用于处理 C 风格字符串数据的输入输出流，它们在头文件 strstream 中进行声明。

（5）C++的 I/O 流对象不能复制或赋值，这意味着不能把 I/O 流对象作为函数形参或函数的返回类型，若要传递或返回 I/O 流对象，必须使用该对象的指针或引用。

3. 对“>>”（提取）和“<<”（插入）操作符的理解

（1）“>>”和“<<”就是普通的右移和左移操作符，只是它们被 I/O 类进行了重载用于输入输出各种数据。

（2）在 C++中可以通过 I/O 类的对象调用重载的“>>”和“<<”操作符函数进行输入输出，即几乎所有的输入输出类都重载（或继承）了这两个操作符。

（3）对“<<”和“>>”操作符的理解：这两个符号的尖角对着的方向表示内容要到达的地方。比如 cin>>a 表示把左边的内容输入到右边，cout<<a 表示把右边的内容输入到左边。

（4）对“提取”和“插入”名称的理解：提取就是对数据进行输入；插入就是对数据进行输出。比如提取表示从设备（或输入流）提取数据存储至程序中，即提取就是对数据进行输入。

（5）设备（或流对象）只能位于左边（因为重载的操作符只能通过对象调用），所以输入（提取）时只能使用“>>”操作符，输出（插入）时只能使用“<<”操作符。通常把“>>”称为输入操作符，把<<称为输出操作符。

（6）“>>”操作符在 istream 类中进行重载，“<<”操作符在 ostream 类中进行重载。因为 cin 是 istream 类的子类对象，cout 是 ostream 类的子类对象，因此不能把“>>”作用于 cout 对象、“<<”作用于 cin 对象。比如 cout>>，则会出现在 cout 对象所属的类中找不到重载的“>>”

操作符的错误。

（7）因为“<<”操作符是被 ostream 类重载的，因此要使用这个操作符，必须使用 ostream 类的对象进行调用，最常见的是使用 cout 流对象进行调用，比如 cout<<。

（8）重载的“<<”操作符的原型类似于：ostream& operator<<(数据类型)，其中数据类型为 int、float、double 等。由此可见，使用“<<”操作符可以输出基本类型的数据（自定义的类对象除外）。

（9）“<<”操作符可以拼接使用。重载的“<<”操符符返回类型为 ostream 类类型的引用，因此可以拼接（连续）使用，比如 cout<<"A"<<"B"<<"C";。

（10）“<<”操作符的优先级：比如 cout<<a?b::c<<endl;，将被解释为(cout<<a)?b:c;，因为“?:”运算符的优先级低于“<<”，正确的形式为 cout<<(a?b:c)<<endl;。

（11）关于怎样重载“<<”和“>>”操作符对自定义的类进行输入输出的内容，详见第 12 章。

16.2　标准输出流（ostream 类）

16.2.1　使用 ostream 类的成员函数进行输出

（1）要使用 ostream 类的成员函数，必须包含头文件 iostream。

（2）对于 ostream 类的成员函数，需要一个 ostream 类型的对象对其进行调用。本文都以 cout 流对象调用这些成员函数，当然也可以使用 ostream 类创建的其他对象进行调用。

1．使用重载的“<<”操作符成员函数进行输出

（1）注意：平时使用的 cout 只是 ostream 类创建的流对象，因此要对数据进行输出，必须使用对象调用 ostream 类的成员函数，而“<<”就是被 ostream 类重载的成员函数。

（2）对于基本类型的数据，重载的“<<”操作符函数能自动识别，用户自定义的类类型不能使用 ostream 类重载的“<<”操作符函数输出。比如 cout<<"A";，输出字符 A；cout<<3;，输出整数值 3。

（3）使用“<<”操作符输出指针时，会输出指针指向的地址（char*除外）。比如 int *p=&a;，则 cout<<p;将输出 a 的地址。

（4）C++使用指向字符串首地址的指针来表示字符串，因此在使用“<<”操作符对 char*或 char[]类型的数据进行输出时，会以字符串形式输出，直至遇到字符串的终止空字符为止。比如 char *p="AAA";，则 cout<<p;会输出 AAA，而不是 p 的地址。

（5）注意：ostream 类的对象（比如 cout）只能与重载的“<<”操作符联合使用，不能与“>>”

操作符联合使用。比如 cout>>"A";错误，因为在 ostream 类中没有对“>>”操作符进行重载。

2. 使用 ostream 类的其他成员函数进行输出

（1）put 成员函数原型为：ostream &put(char);。

① put 一次输出一个字符（不忽略空白符），对于能转换为 char 类型的值，都会转换为 char 类型后以字符形式输出。比如 cout.put(49);将输出 1，因为 49 在 ASCII 码中表示的字符为 1。

② put 函数可以与“<<”拼接使用。比如 cout.put('a')<<"A"<<endl;输出 aA。

③ put 接受的类型为 char，因此传递给与 char 类型不兼容的实参是错误的。比如 cout.put("A");错误，不能把字符串类型转换为 char 类型。

④ 注意：有些编译器实现了 put 的三个原型，即 put(char)、put(signed char)和 put(unsigned char)，在这种情况下把一个 int 类型变量传递给 put 将出现二义性错误。

（2） write 成员函数原型为：ostream& write(const char *s, streamsize n);。

① 参数 s 是要输出的字符串，n 表示要输出多少个字符。

② write 函数输出指定个数的字符串，在输出时遇到空白字符也不会自动停止，直至输出完成。比如 cout.write("a bcd",3)输出 a b；cout.write("b", 5);则输出 ab***，后面有可能会输出 3 个乱码（依机器而定）。

16.2.2 控制输出时的格式

1. 基本规则

（1）输出时的格式：指的是输出时应以怎样的格式进行显示。比如 cout<<3;应以十进制还是十六进制形式显示。

（2）每一个 I/O 类对象都维持一个控制 I/O 格式的默认格式状态。比如整型值默认以十进制形式输出等。

（3）有关浮点数的说明。

① 浮点数默认输出 6 位数（显示的总位数），末尾的 0 不被显示，当指数大于 6 或小于−5 时使用科学计数法表示。比如 cout<<1.2342234;输出 1.23422，cout<<1.2000;输出 1.2。

② 对于浮点数精度，在默认情况下指的是总位数，在定点（小数点）和科学模式下指的是小数点后的位数。

③ 在定点（小数点）和科学模式下显示的浮点数会输出末尾的 0。

④ 浮点数的默认显示模式没有专门的设置方式，只能使用 setf 函数的标记组合进行间接设置。

（4）C++容纳字段的方式是给字段分配刚好合适的宽度，所以 C++中默认的字段宽度为 0，以适合所有的字段。若设置的字段宽度小于输出的数据，则显示所有的数据而不会进行截断处

理。

（5）格式的持续性：有些格式设置后对以后的输出会一直有效，而有些格式设置后仅仅对下一次输出有效，然后就恢复默认值。比如 cout<<setw(12)<<22<<33;，则只对输出的数值 22 设置 12 个字符的宽度（不足则左侧默认填充空格），对 33 就无效了。再比如 cout<<hex<<13<<14;，则输出 de，该格式状态对以后的输出一直有效，其中 hex 表示以十六进制形式显示数值。

2. 使用标准控制符进行格式输出

下面以 hex 控制符为例进行讲解，hex 表示以十六进制形式显示数值。

（1）hex 控制符的原型为：ostream& hex(ostream &){...};。其他控制符类似。

① 控制符不是 I/O 类的成员函数，但控制符是函数，因此可以像使用普通函数那样使用控制符。

② 控制符要求使用一个 ostream 类类型的对象作为参数，比如 hex(cout);表示以十六进制形式显示。

（2）ostream 类中重载的“<<”操作符函数与控制符有关的原型为：ostream & operator<<(ostream& (*f)(ostream &)){return f(*this); }。重载的“<<”操作符函数要求使用一个类型形式为“ostream& hex(ostream&)” 的函数作为实参。

（3）使用控制符的方式。

① cout<<hex<<13;，输出 d，此方式等同于 cout.operator<<(hex);。

② hex(cout); cout<<13;，输出 d，直接调用控制符函数。

③ cout.operator<<(hex);，输出 d，直接调用重载的“<<”操作符函数。

④ cout<<hex<<hex;，控制符可拼接使用。

推荐使用第一种和第四种方式，第二种方式不能拼接输出，第三种方式不够直观。

（4）不推荐使用点运算符调用控制符函数，因为使用点运算符会发生以下错误。

① cout.hex();是错误的，因为 hex 函数不是 ostream 类及其父类的成员函数。

② cout.endl();和 cout.endl;都是错误的，因为 endl 既不是 ostream 类及其父类的成员函数，也不是其中定义的数据成员。

（5）应用控制符之后的内容将一直按照该格式输出，除非重新对格式进行设置。比如 cout<<hex<<13<<14<<15;将依次输出 def。

标准控制符及其含义如表 16.1 所示。

表 16.1 标准控制符及其含义

控制符	含 义	控制符	含 义
※dec	以十进制形式显示	uppercase	对十六进制形式的字符和基数前缀显示大写
oct	以八进制形式显示	※nouppercase	对十六进制形式的字符和基数前缀不显示大写
hex	以十六进制形式显示	showpoint	显示小数点
boolalpha	将真或假显示为 true 或 false	※noshowpoint	有小数部分时才显示小数点
※noboolalpha	将真或假显示为 1 或 0	fixed	以（小数点）定点形式表示浮点数
showpos	显示正数前的“+”（对八进制、十六进制数无效，因为它们被 C++视为无符号数）	scientific	以科学计数法表示浮点数
※noshowpos	不显示正数前的“+”号	left	左对齐或将填充字符加到数值的右边
showbase	显示进制的前缀基数，八进制以 0 开始，十六进制以 0x 开始	※right	右对齐或将填充字符加到数值的左边
※noshowbase	不显示进制的前缀基数	internal	把填充字符加到符号和值之间
endl	刷新输出缓冲区，插入一个换行符	※skipws	跳过输入中的空白符
flush	刷新输出缓冲区，不插入任何字符	noskipws	不跳过输入中的空白符
ends	刷新输出缓冲区，并插入空字符'\0'	ws	吃掉空白符
unitbuf	在每个输出运算符之后刷新缓冲区，比如 cout<<unitbuf<<3<<4;等于 cout<<3<<flush<<4<<flush;		
※nounitbuf	恢复常规缓冲区刷新模式（主要用于取消 unitbuf 设置的刷新模式）		

说明：※表示是默认值。

3. 使用 iomanip 头文件中的控制符进行格式输出

（1）需要包含头文件 iomanip。

（2）iomanip 头文件中的控制符需要参数。比如 cout<<setw(12);，表示设置显示的字段宽度为 12。

（3）因为 iomanip 中的函数仍是控制符，因此其和“cout<<”联合使用才能有用。比如 setw(12); cout<<12<<endl;，则 setw(12)不起作用，正确形式为 cout<<setw(12);。注意，setw(cout)是错误

的，因为 setw 的形参不是 ostream 类型的。

iomanip 头文件中的控制符及其含义如表 16.2 所示。

表 16.2　iomanip 头文件中的控制符及其含义

控制符	含　义	是否持续有效
setfill(c)	设置填充字符，c 只能是字符型数据，其他类型数据是错误的	是
setprecision(n)	设置浮点数的精度为 n。默认为 6	是
setw(n)	设置显示的字符宽度为 n。默认为 0，以容纳所有字符	否
setbase(n)	设置显示时的进制数，n 只能是八进制、十进制、十六进制数三者之一	是
setiosflags(is)	使用 ios_base 父类中定义的格式常量设置格式状态	是
resetiosflags(is)	清除使用 ios_base 中定义的格式常量设置的格式状态	是

说明：最后两个控制符不常用，它们与 setf 函数类似。

4. 使用 ostream 类的成员函数进行格式输出

注意：ostream 类的成员函数只能使用其对象和点运算符进行明确调用，不能在“<<”操作符后进行输出调用，因为重载的“<<”操作符函数并不接受指向这些成员函数指针的形参。比如 cout<<width(3);错误，正确形式为 cout.width(12);。

ostream 类中有关格式输出的成员函数及其含义如表 16.3 所示。

表 16.3　ostream 类中有关格式输出的成员函数及其含义

控制符	含　义	是否持续有效
int width()	显示当前的字符宽度。默认为 0	否
int width(n)	设置显示的字符宽度为 n，并返回以前的字符宽度	否
fill(c)	将字符 c 设为填充字符。c 可以是可转换为 char 类型的类型	是
precision(n)	设置浮点数的精度为 n。默认为 6	是
setf()和 unsetf()	使用 ios_base 类中的格式常量设置格式状态	是

说明：上面函数应使用点运算符即 cout.width(9);的形式进行调用，因为它们是成员函数。

5. 使用 setf 成员函数进行格式输出

（1）setf 函数通过对“格式标记位”进行设置，从而达到进行格式控制的目的。假设格式标记共有 8 位，比如为 0000 0000，第 1 位代表十六进制输出，第 2 位代表十进制输出，第 3 位代表八进制输出，若要使用八进制输出，则只需使用 setf 函数把第 3 位的 0 设置为 1 即可，因此使用一个与 0010 0000 对应的十进制数值就能设置该位了。实际上使用时不需要专门计算这些十进制数值是多少，因为这些数值在 ios_base 类中已定义了相关的格式常量（见表 16.4）。比如要设置八进制输出，只需使用 ios_base::oct 设置格式标记就行了。

表 16.4 ios_base 类中定义的格式常量

格式常量	含　义	格式常量	含　义
boolalpha	设置 bool 变量显示为 true 和 flase	left	左对齐
showbase	显示基数前缀	right	右对齐
showpoint	显示末尾没有数值时的小数点和尾数 0	internal	在符号和数值之间填充字符
uppercase	对十六进制形式的字母使用大写	dec	以十进制形式显示数值
showpos	显示正数前的“+”号	oct	以八进制形式显示数值
scientific	以科学计数法显示浮点数	hex	以十六进制形式显示数值
fixed	以定点格式（就是小数形式）显示浮点数		

说明：使用这些常量应加限定词，比如 ios_base::showpos，否则可能访问的是相同名称的控制符。

（2）setf 函数有两个版本：一个是有单一形参的；一个是有两个形参的。

（3）单形参 setf 函数的原型为：fmtflags setf(fmtflags);。

① fmtflags 就是格式标记的类型，定义于 ios_base 类中。我们不需要关心此类型是怎么定义的，只管使用就行。

② setf 的返回值为设置新标记以前的标记信息。

③ 单形参 setf 函数用来设置单个位的格式标记，使用比较简单，要设置什么格式状态就向 setf 传递什么状态的 iso_base 格式常量就行了。比如 cout.setf(ios_base::showpos);　cout<<12<<13;，输出+12+13。

（4）双形参 setf 函数的原型为：fmtflags setf(fmtflags, fmtflags);。

① fmtflags 的含义同单形参函数。

② 首先使用第 2 个形参清除第 1 个形参设置的所有标记位，然后使用第 1 个形参设置标记位。

③ 双形参 setf 函数用于处理格式状态相互排斥的情形。比如格式标记为 0000 0000，其中第 1 位表示八进制输出，第 2 位表示十进制输出，第 3 位表示十六进制输出，不能对格式标记中的 2 位或 3 位都进行设置。若之前对其进行设置过，比如以十进制输出，则标记状态变为 0100 0000，这时想要改变为以十六进制输出，单形参 setf 函数就没有办法了，需要使用双形参 setf 函数，首先使用第 2 个形参把使用第 1 个形参设置的前 3 位格式标记清除，即还原为 0000 0000，然后再使用第 1 个形参进行设置，即设置为 0010 0000。注意：第 2 个形参只能清除指定范围内的标记状态，并不能把所有标记状态都清除。比如本例就只能清除前 3 位的标记状态，其余位不会受到影响。

④ 双形参 setf 函数的第 2 个形参见表 16.5，它们也是在 ios_base 中定义的格式常量。

表 16.5　ios_base 类中的格式常量分组

第 1 个参数	第 2 个参数	说　明
idec	basefield	比如：cout.setf(ios_base::hex, ios_base::basefield);，首先使用 basefield 清除指定范围内的标记状态，然后使用 hex 设置为以十六进制输出
ioct		
ihex		
fixed	floatfield	比如：cout.setf(ios_base::fixed,ios_base::floatfield);，首先使用 floatfield 清除指定范围内的标记状态，然后使用 fixed 设置为以定点格式显示浮点数
scientific		
left	adjustfield	比如：cout.setf(ios_base::left,ios_base::adjustfield);，首先使用 adjustfield 清除指定范围内的标记状态，然后使用 left 设置为左对齐进行输出
right		
internal		

说明：该分组主要用在双形参 setf 成员函数中。

（5）使用 unsetf(fmtflags)函数可以清除由 fmtflags 参数指定那一位的标记状态。比如 unsetf(ios_base::fixed);，表示清除浮点数的定点格式标记状态。

（6）单形参 setf、双形参 setf 和 unsetf 函数联合使用：setf 和 unsetf 函数都只能对二进制位的其中一位进行设置和清除值，只要联合使用 unsetf 和单形参 setf 函数，就可以不使用双形参 setf 函数。比如以前是以十进制输出的，现在要改为以十六进制输出，则可以这样做：cout.unsetf(ios_base::dec); cout.setf(ios_base::hex);。若不清除以前的标记状态，使用 setf 函数标记设置十六进制输出是不行的（因为不允许以十进制输出的同时又以十六进制输出）。若不知道以前是以多少进制输出的，则可以使用 unsetf 函数进行两次清除来达到目的，即 cout.unsetf(ios_base::dec); cout.unsetf(ios_base::oct); cout.setf(ios_base::hex);。当然，在这种情况下使用双形参 setf 函数更快捷，即 cout.setf(ios_base::hex, ios_base::basefield);。

（7）因为 setf 和 unsetf 函数是对位进行操作的，因此可以使用位操作符（“|”（或）操作符）对 setf 函数的实参进行计算。比如 cout.setf(ios_base::showpos|ios_base::left);，表示同时设置左对齐和显示正数前的“+”号。

（8）不存在单独设置浮点数的默认显示模式，浮点数的默认显示模式只能使用 setf 函数进行间接设置，只要不是定点或科学计数模式的其他组合就都是默认显示模式，因此可以使用 cout.unsetf(ios_base::floatfield) 直接把两种显示模式都清除来达到目的，也可以使用 cout.setf(0,ios_base::floatfield) 来达到目的，或者已知为科学计数显示模式，使用 cout.setf(ios_base::fixed)来达到目的。

6．刷新输出缓冲区

（1）每个 I/O 类对象都管理一个自己的缓冲区，当程序输入输出时首先把数据输入缓冲区，直至缓冲区被填满或被刷新，然后再把数据从缓冲区发送出去，并清空缓冲区。

（2）输出缓冲区的刷新：向屏幕输出内容时，程序不会等到缓冲区填满才刷新缓冲区。

（3）在输入之前会刷新缓冲区。比如 cout<<"hy"; int a; cin>>a;，在 cin 之前会刷新缓冲区。若不刷新缓冲区，将看不到字符 hy 的输出。

（4）程序正常结束时会清空所有的输出缓冲区。

（5）还可以使用控制符刷新缓冲区（见表 16.1）

（6）系统崩溃时不会自动刷新缓冲区，这有可能导致程序已经执行了输出操作，但内容还未输出到设备上，仍在缓冲区中。因此在对程序进行错误检查时，不能根据输出判断出出错的位置。若想要根据输出判断出出错的位置，就必须确保缓冲区被刷新了，也就是在输出时调用了 flush 或 endl 等控制符。

16.3 标准输入流（istream 类）

16.3.1 流状态

（1）流状态的作用：每一个输入输出流都保留有流的当前状态，以反映当前的流是处于正常状态还是破坏状态。若流处于破坏状态，则将关闭以后的输入输出，因此无法再使用该流进行输入输出。注意：cin 流对象被破坏只会影响到无法使用 cin 流对象进行输入，不会影响到其他 istream 流对象的输入，因为 cin 流对象和其他流对象是相互独立不相关的。

（2）C++流状态实现原理：C++中的流状态是使用 ios_base 类中定义的 iostate 类型进行描述的，该类型使用标记位的形式对流状态进行管理。假设 iostate 类型具有 3 个标记位，比如为 000，第 1 位代表到达文件尾，第 2 位代表流状态破坏，第 3 位代表 I/O 失败等，则可以这样描述流状态：第 2 位为 1 则表示流被破坏，为 0 则表示未被破坏；若 3 个位都为 0，则表示流状态完好等。C++为每个标记位都定义了一个对应的名字，这样方便使用标志位中的每一位，比如为第 2 位命名为 failbit。

（3）流状态位：C++使用 3 个流状态成员来表示流状态，即 eofbit、badbit、failbit，它们被称为“标记位”或“状态位”。其中每一个成员表示一种流的状态，对应一个标记位。它们的类型是 iostate，它们被定义在 ios_base 类中，因此要使用这些状态位，则应使用 ios_base::进行限定修饰。

- eofbit：若到达文件尾，则设置为 1。
- badbit：若流被破坏，则设置为 1。一般不可恢复的错误，设置此位为 1。
- failbit：若读入或输出无效的字符，则设置为 1。一般可恢复的错误，设置此位为 1。

（4）使用 I/O 类成员函数查看流状态位：调用以下成员函数必须使用 I/O 流对象和点运算

符，比如使用 cin 对象。以下成员函数定义于 ios_base 类中。

- bad()：若 badbit 被设置，则返回 true。
- eof()：若 eofbit 被设置，则返回 true。
- fail()：若 badbit 或 failbit 被设置，则返回 true。
- good()：如果没有设置任何状态位，则返回 true，此时表示流状态完好。
- rdstate()：返回流当前的状态，返回类型为 ios_base::iostate。

（5）使用 I/O 类成员函数设置流状态位：调用以下成员函数必须使用 I/O 流对象和点运算符，比如使用 cin 对象。以下成员函数定义于 ios_base 类中。

- setstate(iostate)：设置某个流状态位为 1，其他状态位不变。比如 cin.setstate(ios_base::eofbit);表示把 eofbit 状态位设置为 1，其他状态位不变。
- clear(iostate s=0)：设置某个流状态位，其他状态位设置为 0。若使用默认值 0，则表示所有状态位都设置为 0，此时流状态完好，相当于把流恢复（重置）为正常状态。比如 cin.clear(ios_base::eofbit);表示把 eofbit 状态位设置为 1，其他位设置为 0；cin.clear();表示恢复（重置）流为完好状态。

（6）最简单的测试流状态的方法就是对流对象的返回值直接进行测试。比如 if(cin)或 if(cout)等，若 cin 流或 cout 流处于正常状态，则它们的返回值是非零值；若处于非正常状态，则 cin 流或 cout 流会返回值 0。

（7）因为流状态是对位进行的操作，因此可以使用位操作符（“|”）（或）操作符对流状态的多个位同时进行设置，比如 cin.setstate(ios_base::eofbit | ios_base::badbit);。

（8）可以使用 rdstate 成员函数来保存和恢复以前的流状态。比如 ios_base::iostate m=cin.rdstate(); cin.clear(); cin.clear(m);，第一句表示保存当前的流状态，cin.clear(m);表示恢复以前的流状态。

（9）建议在输入前使用 cin.clear();使流处于正常状态，然后再进行输入，比如 int a;cin.clear();cin>>a;。

（10）流被重置为正常状态，并不会把留在输入流中的字符也清除掉。比如 int a; char c; cin>>a; cin.clear();cin>>c;，输入 f;，则 a 无值，c='f'。分析：f 对于 int 类型的 a 是无效字符，因此设置流状态位 failbit 并关闭输入流 cin，字符 f 留在输入流中，然后使用 cin.clear()恢复输入流为正常状态，再次启用输入流，此时 cin>>c;把留在输入流中的字符 f 读入给变量 c（此时不会提示输入）。若没有 cin.clear()语句，则变量 c 也不会有任何值，因为输入流 cin 被关闭，不会再为变量 c 输入值了。

16.3.2 使用 istream 类的成员函数进行输入

（1）istream 类的成员函数，需要一个 istream 类型的对象对其进行调用。本文都使用 cin 流对象调用这些成员函数，当然也可以使用 istream 类创建的其他对象进行调用。

（2）要使用 istream 类的成员函数，必须包含头文件 iostream。

（3）一旦数据进入输入流，只要未被程序读入，该数据就会一直在输入流中。比如 int a; char c; cin>>a>>c; ，输入 3s，然后回车，则 a=3; c=s;，数字 3 首先被读入 a，字符 s 被留在输入流中，下次读入时直接从输入流中读入字符 s 并赋值给 c，这时不会提示对 c 的输入。但回车换行符仍留在输入流中，下次输入时将从输入流中直接读取回车换行符。

（4）重要术语。

① 空白符，包括空格、制表符、换行符、回车符等。

② 到达文件尾可以通过使用键盘模拟，对于 DOS 一般是使用“Ctrl+Z”，对于 UNIX 是使用“Ctrl+D”，当然也有可能操作系统未实现文件尾的模拟。

③ EOF——到达文件尾时使用常量 EOF 表示，EOF 是 iostream 头文件提供的一个符号常量，通常值为–1，也可能为其他值，依编译器而定。设置为负值，是因为 ASCII 码（或其他编码）中没有使用负数表示的字符。注意 EOF 不等于空白符。

（5）输入流的状态位一旦被设置（即流被破坏），就会关闭输入流，直至流被重置为完好状态。因此在输入前最好使用 clear()成员函数恢复输入流状态，比如 cin.clear();。

1．使用重载的“>>”操作符成员函数进行输入

（1）注意：平时使用的 cin 只是 istream 类创建的流对象，因此要对数据进行输入，必须使用对象调用 istream 类的成员函数，而“>>”就是被 istream 类重载的成员函数。

（2）重载的“>>”操作符的原型为：istream& operator>>(数据类型);，其中数据类型为 int、float、double 等。

（3）对于基本类型的数据，重载的“>>”操作符函数能自动识别，用户自定义的类类型不能使用 istream 类重载的“>>”操作符进行输入，这时需要自定义重载的“>>”操作符（详见 12 章）。

（4）重载的“>>”操作符函数可拼接输入，即形如 cin>>a>>b;的形式。

（5）注意：istream 类的对象（比如 cin）只能与重载的“>>”操作符联合使用，不能与“<<”操作符联合使用。比如 cin<<a ;错误，因为在 istream 类中没有对“<<”操作符进行重载。

（6）使用“>>”操作符读取数据的规则。

① 从输入流读取字符，直到文件结束或遇到一个无效值。在读取数据时，会丢掉所有的空白符。

② 若读取到一个无效值，则 istream 对象会设置 failbit 状态位，且关闭输入流，直至被重

置为完好状态。若读取到一个有效值，则不会设置状态位。

③ 注意理解：读取到无效值和读取到有效值后遇到无效值结束读取的区别。

示例如下：

```
int a; char c; cin>>a; cin>>c;
```

① 输入 F，因为 F 对于 a 是无效字符，从而导致 a 未读取到一个有效值，设置流状态位，关闭输入流，因此 cin>>c 不会被执行，我们也接收不到对 c 的输入提示，c 也不会有任何值。可以使用前文介绍的 cin.clear();来恢复流状态，并打开输入流。

② 输入 3F，则 a=3; c=F;，3 对于 a 是有效值，因此读入到 a 中；字符 F 对于 a 是无效值，结束读取，字符 F 留在输入流中，但 a 已经读取到有效值了，因此不会设置流状态位，输入流是正常的。然后 c 直接从输入流中读取字符 F，但不会对 c 提示输入。

③ 输入 3 z，结果 a=3; c=z;，首先在读取时丢掉所有的空白符，找到字符 3，在 3 之后的空白符对于整型来讲是无效字符，因此读取结束，最后 3 被读入到 a 中，剩余的空白符和 z 留在输入流中。然后 cin>>c 继续从输入流中读取字符（这里同样不会对 c 的输入进行提示），同样忽略空白符，直接读取到字符 z，因此最后 c=z;。

（7）使用“>>”输入字符串，因为键盘不能输入空字符'\0'，因此“>>”使用空白符来确定字符串的结尾位置。使用“>>”读取字符串，在遇到空白符时，就结束读取，并将所输入的字符串放到相应的对象中，然后自动在字符串的末尾添加空白符'\0'，剩余的未读取的字符保留在输入流中，等待下一次读取。比如 char c[7]; cin>>c;，当在键盘上输入“abc def”后，所输入的字符串“abc”将被读取到字符数组 c 中，并在字符串“abc”的末尾自动添加空白符，这时字符数组 c 中就相当于存储了字符串“abc”，剩余的空白符和字符串“def”被留在输入流中，等待下一次读取。

（8）若输入错误时，对 istream 对象进行测试，则结果为 false。比如 cout<<cin;，若 cin 有错误，则输出 0000 0000。一般用在条件测试语句中，比如 if(cin);或 if(!(cin>>a));，注意“>>”的优先级低于“!”，所以小括号不能省略。

（9）可以使用前面介绍的控制符来控制输入时的格式状态。比如：

① 使用 noskipws 控制符读取空白符。如 char c; cin>>noskipws; cin>>c;，输入“ 3”，结果 c 的值为空格符，其 ASCII 值为 32，即 cout<<(int)c;将输出 32。若没有 cin>>noskipws，则会把字符 3 读入到 c 中（空白符会被忽略），其 ASCII 值为 51。

② 输入后的回车换行符会被保留在输入流中，因此使用 noskipws 读取空白符时，会导致回车换行符也会被读取。如 int a,b; cin>>noskipws; cin>>a>>b;，输入 3 后回车，将导致变量 b 没有值被读入，也不会出现提示输入的信息。可以使用 cin.ignore();或 cin.get();读取并丢弃一个字符来解决此问题。如 cin>>a; cin.ignore(); cin>>b;。

③ 使用 skipws 控制符将使空白符被忽略。如 int a; cin>>skipws; cin>>a;，输入“ 3”，结

果 a=3。

④ 使用 hex、oct、dec 等控制符，可使程序以十六进制、八进制、十进制形式读取数据。如 int a; cin>>hex; cin>>a;，输入 13，结果 a=19; ，因为 13 以十六进制形式被赋值给变量 a，十六进制数 13 是十进制数 19。同理，若输入 d，将使 a=13;，这里的 d 在十六进制模式下并不是无效字符。

2. 使用 get 和 getline 成员函数进行输入

（1）istream& get(char& c);：读取单个字符（包括空白符），然后赋值给形参 c。

① 该函数可拼接调用，因为返回类型为 istream&。比如 cin.get(c).get(d)>>e;。

② 到达文件尾时，则停止输入，并设置 failbit 和 eofbit 状态位，然后关闭输入流。

③ 到达文件尾时，get(char&)不会把表示文件尾的符号常量 EOF 读入给形参 c。

④ 注意：get(char&)不能接受 int 类型或其他类型的形参（因为形参是引用）。比如 int a; cin.get(a);是错误的。

（2）int get(void);：读取单个字符（包括空白符），然后转换为 int 类型，并将其值返回。

① 该函数不可拼接调用，因为返回类型为 int。比如 cin.get().get();是错误的。

② 到达文件尾时，则返回值 EOF（通常为–1）。能返回 EOF 值是使用 get(void)函数的主要原因。

③ 若要判断 get()是否到达文件尾，应把 EOF 值与 int 类型值进行比较，因为有些编译器可能无法使用 char 类型来表达 EOF 值。比如有些编译器对于 char c=EOF;的使用有可能是错误的。

④ 使用 get()函数的原因：因为有些编译器会把 char 实现为 unsigned char，这时若把设为负值的 EOF 赋给 char 类型将进行一种内部转换，转换之后的值将不再与 EOF 值相等。若编译器把 char 实现为 singed char，则能保证正确地把 EOF 值赋给 char 类型。无形参的 get()函数就是针对这种编译器对 char 的不同实现而设计的。

⑤ 可以使用 get()函数读取并丢弃一个字符。比如 cin.get();，此处 get()的返回值未赋给任何变量。

（3）使用 get 和 getline 函数输入字符串：

```
istream& get(char*s,int n,char c='\n');   //c 称为终止符或分隔符
istream & getline(char *s, int n, char c='\n');
```

① 以上函数可以接收两个或三个实参，接收两个实参的使用默认实参'\n'。

② 含义：读取 $n-1$ 个字符到 s 所代表的数组中，直至遇到字符 c 指定的终止符，并在结尾加入空字符'\0'。加入空字符是为了把所输入的字符组合成字符串，因此读取的字符数比实际指定的要少一个。默认使用终止符为'\n'（回车换行符）。

③ 遇到这三种情况结束读取：读入了指定数量的字符；遇到终止符 c；到达文件尾。

④ 注意：第一个形参 s 指定的类型是 char 指针，因此不能指定 int 等其他类型的指针实参，

否则是错误的。比如 int s[55];，则 cin.get(s,4);错误。

⑤ 以上函数可拼接调用，比如 cin.get(s,n).getline(s,n);。

⑥ get 和 getline 的区别：get 把终止符留在输入流中，而 getline 会读取并丢弃流中的终止符，但不会把终止符读入到形参 s 中。

⑦ 若使用了非默认的'\n'（回车换行符）终止符，则 get 和 getline 函数会读入'\n'。比如 char s[44]; cin.get(s, 4, 'a');，输入 s↙fa↙ea，则最后 s="s↙f"，其中↙表示回车换行符。

⑧ 流状态位的设置。

- 到达文件尾：get 和 getline 都会设置 eofbit 状态位。
- 未读入任何字符：这时 get 会设置 failbit 状态位，而 getline 不会设置。未读入任何字符可以通过直接输入终止符来验证。比如 char s[4]; get(s, 5,'a'); getline(s, 5, 'a');，输入 a;，则 get 会设置 failbit 状态位，而 getline 不会设置 failbit 状态位。
- 读入的字符数超过了指定的最大字符数，这时 getline 会设置 failbit 状态位，而 get 不会设置。比如 char s[34]; cin.get(s,2); cin.getline(s,2);，假设输入 adssdsd，则 getline 会设置 failbit 状态位，而 get 不会设置 failbit 状态位。

3. 清理输入流的成员函数

（1）ignore 成员函数：istream& ignore(int n=1; int c=EOF);。

① ignore 函数的主要作用是把留在输入流中的字符读入并丢弃（清除）。使用 cin.get()也可以读入并丢弃一个字符。

② 含义：读取并丢弃 *n* 个字符，直至遇到由 c 指定的终止符。丢弃的字符包括终止符在内。默认情况就是读取并丢弃一个字符，或遇到 EOF（文件尾）。此函数可以拼接使用。比如 cin.ignore(4,'a');，输入 bcadef;，则丢弃字符 bca，剩余的字符 def 留在输入流中。

（2）peek()函数：返回输入流中的字符，但不读取输入流中的字符，也就是可以使用 peek 函数查看输入流中的下一个字符。比如 char c=cin.peek();，把查看到的输入流中的字符赋给变量 c。

（3）gcount()函数：返回最后一个以非格式化输入方式读取的字符数，对于 getline 函数，最后的终止符也会被计算在内。非格式化输入方式即 get 和 getline 这样的成员函数，而“>>”操作符属于格式化输入方式，因为“>>”会把所输入的数据转换为匹配的类型。注意：只能返回最后一个非格式化输入中的字符数，也就是在使用 gcount 之前使用了 peek、putback 等函数，则 gcount 将输出 0。比如 char s[44]; cin.get(s,4); cin.peek();，则 cout<<cin.gcount();将输出 0，因为 peek 未读入字符。

（4）read 成员函数：iostream& read(const char* addr,streamsize n);。将 *n* 个将字符读入到 addr 中，此函数不会在读入的字符后面加上空字符'\0'，因此不会把所输入的字符转换为字符串。此

函数可以拼接使用。

（5）putback(c)函数：将字符 c 插入到输入流中最前面的位置上，下一条输入语句将会读取该字符。假设输入流中的内容为"abc"，调用 cin.putback('z');，则输入流中的内容变为"zabc"。

（6）清空输入流中多余字符的方法：一般的做法是使用不带参数的 get 函数重复读取输入流中的字符，直到遇到回车换行符'\n'为止。比如 while(cin.get()!='\n') continue;，若输入流中没有任何内容，此语句却无法判断，若遇到这种情况，会提示输入一次值。

16.4 文件流

一些大型程序往往需要把程序的某些数据保存（输出）到文件（比如数据库）中，其他程序要使用这些数据时，直接从该文件中读取即可。这是使用文件的主要目的，由此可见，文件流是相当重要的。

1. 使用流读写文件

（1）要对文件进行操作，必须包含头文件 fstream。

（2）对文件进行操作需要使用 I/O 类中的 ifstream、ofstream 和 fstream，这三个文件 I/O 类一般统称为文件流或文件 I/O 流。由文件流创建的对象，被称为文件流对象，有时简称“流对象”。

（3）要使用流对文件进行读写应按以下步骤进行。

① 使用文件流创建流对象，即建立管道。创建流对象比较简单，直接使用文件 I/O 类创建即可，比如 ofstream os;。

② 把所创建的流对象与特定的文件相关联，其实际实现时相当于打开文件。比如 os.open("aa");，表示打开文件 aa。

③ 使用文件流及其父类中的成员函数对文件进行读取操作。因为文件流继承自标准输入输出流 istream 和 ostream，因此文件流对象可使用前面介绍的 istream 和 ostream 成员函数。比如 os<<"AAA";和 os.write("AAA",3);都表示向与 os 连接的文件写入字符 AAA。

（4）注意：向文件写入使用的是文件输出流 ostream，从文件读取使用的是文件输入流 istream，因为 C++标准已经把流的另一端与程序相关联了，因此文件的输入输出流就是相对程序而言的，向文件写入数据就是输出，从文件读取数据就是输入。

2. 打开文件

（1）对于文件流有两种方式打开文件：一种是使用构造函数；一种是使用成员函数 open。

（2）使用构造函数打开文件。

ofstream 类的原型（简化形式）为：ofstream(char *f,openmode m=ios_base::out);。

ifstream 类的原型为：ifstream(char *f, openmode m=iosbase::in);。

① 含义：以模式 m 打开由 f 指定的文件。

② 向文件写入内容应使用 ofstream 输出流，从文件读取数据应使用 ifstream 输入流。

③ 文件模式指的是对文件进行写入还是读取，抑或是以二进制方式打开文件。

④ openmode 是由 ios_base 类定义的类型，用于指定文件以什么模式打开。至于 openmode 到底是什么类型，无须关心。

⑤ ofstream 的默认模式 out 表示以写入方式打开文件，并清除文件中原有的内容。若文件不存在，则创建新文件。

⑥ ifstream 的默认模式 in 表示以读取方式打开文件，in 模式不会清空文件，也不会创建新文件。

⑦ 目录的指定方式：在 VC++ 2010 中可以使用“/”和“\\”指定路径，不能使用“\”指定路径，因为“\”表示转义字符。比如 C:/xx/yy/zzz.jj 或 C:\\xx\\yy\\zzz.jj。若不指定路径，则表示文件位于当前目录下。要注意程序有可能会创建文件，但一定不会创建文件夹。也就是说，在把数据写入到文件中时，其文件夹必须已经存在。还应注意中文名可能无法被识别，导致文件不能成功打开。

⑧ 由于历史原因，C++使用 C 风格字符串来指定文件路径，因此不能指定 string 对象作为文件路径，但可以使用 string 的成员函数 c_str 把 string 对象转换为 C 风格字符串。比如 string s="aa"; s.cstr();。

（3）open 成员函数的原型（简化形式）为：open(char *f, openmode m=默认值);。

其形参与 ofstream 和 ifstream 构造函数类似，只是需要使用 ofstream 或 ifstream 类的对象来调用 open。open 函数 m 的默认值，若是使用 ofstream 流对象调用的，则默认模式与 ofstream 相同；若是使用 ifstream 流对象调用的，则默认模式与 ifstream 相同。比如 ofstream os; os.open("a.txt");表示以默认文件模式打开文件 a.txt。

（4）使用 fstream 打开文件。

① 使用 fstream 流创建的对象可以同时对文件进行读取和写入。当需要对一个文件进行反复读取和写入操作时，使用该流对象是很好的选择。

② fstream 类构造函数的原型（简化形式）为：fstream(char *f，openmode m=ios_base::in | ios_base::out);。

③ 默认 fstream 流对象以读取和写入的方式打开文件，此时文件不会被清空，若文件不存在也不会创建新文件。注意：有些编译器的文件模式 m 可能没有设置默认值。

3．使用文件流进行输入输出

（1）当成功打开文件后，ofstream 流对象便可以使用 ofstream 类以及从 ostream 类继承的成员函数对文件进行写入。比如 ofstream os("a.txt"); ，则 os<<"AAA";表示向文件 a.txt 写入 3 个字符 AAA。还可以使用 ostream 类中的其他成员函数，比如 os.put('a'); os.write("AAA");等。

（2）当成功打开文件后，ifstream 流对象便可以使用 ifstream 类以及从 istream 类继承的成员函数从文件进行读取。比如 ifstream is("a.txt"); char c[33];，则 is>>c;表示从文件 a.txt 中读出数据并赋给数组 c。还可以使用 istream 类中的其他成员函数，比如 is.get(); is.getline(c,44);等。

（3）fstream 流对象可以使用 fstream 类以及从 istream 类和 ostream 类继承的成员函数对文件进行写入和读取。比如 fstream fs("a.txt"); char c[33];，则 fs<<"AA"; 和 fs>>c;都是正确的。当然还可以使用 istream 和 ostream 类中的其他成员函数，比如 get、put 等。

4．文件模式

（1）文件模式指的是文件被使用的方式，比如读取、写入、追加等（见表 16.6）。在使用文件流打开文件时必须指定文件模式（若未指定，则使用的是系统设置的默认值）。

表 16.6　文件模式

文件模式	说　明
ios_base::in	打开文件以便读取。若读取到一个不存在的文件，会设置 failbit 状态位
ios_base::out	打开文件以便写入，并清空文件内容。若文件不存在，则创建一个新文件。out 默认与 out | trunc 组合等价。若 out 与 app、in 组合，不会清空文件；与 ate 组合，会清空文件；与 app、ate、trunc 组合，若文件不存在，则会创建新文件
ios_base::ate	打开一个已存在的文件，并把指针移至文件尾。该模式可以修改文件原有的内容
ios_base::app	以写入方式打开文件，但只允许把内容添加到文件尾，不允许修改原有的内容
ios_base::trunc	如果文件存在，则清空文件；若不存在，则创建新文件。若指定的模式为 out，则默认为此模式
ios_base::binary	以二进制方式打开文件，若不指定该模式，则默认以 ASCII（字符）形式打开文件

（2）文件模式的类型为 ios_base 类中定义的 openmode，此类型是使用标记位进行控制的，因此可以使用逻辑操作符（一般为“|”（或）操作符)进行多位值设置。

（3）文件模式可进行组合（详见表 16.7），但并不是所有模式的组合都是有效的。比如 in | app，对于读取文件来说，打开文件且在末尾读取文件的内容，但因为文件末尾没有内容可读，所以该模式组合没有意义。

表 16.7　文件模式组合

模　式	说　明
in | out	打开文件以便同时进行读写，并定位在文件开始处，打开时不清空文件内容，若文件不存在也不创建新文件
out | trunc	打开文件以便写入，如果文件已存在，则清空文件内容。与单独使用 out 模式的功能一样
out | app	以追加的方式打开文件以便写入。注意：数据只能写入到文件的尾部，不允许修改原有的内容。此模式不会清空文件内容，但若文件不存在会创建新文件
out | ate	打开文件，移至文件末尾。此模式会清空文件，若文件不存在会创建新文件
out | binary	以二进制方式打开文件以便写入
in | trunc	这是错误的组合，会设置 failbit 状态位
in | ate	不会清空文件，若文件不存在，则会设置 failbit 状态位
in | app	这样的组合是没有意义的，因为打开文件在末尾无内容可读。此模式不会清空文件，但会创建新文件
in | binary	以二进制方式打开文件以便读取
in | out | binary	以二进制方式打开文件以便能同时读写
in | out | trunc	打开文件以便读写，如果文件已存在，则清除文件内容

说明：out、trunc、app 只能用于指定 ofstream 和 fstream 类对象打开文件，in 只能用于指定 ifstream 和 fstream 类对象打开文件，ate 和 binary 可以用于所有的文件流对象打开文件。

5. 流、文件、文件内指针的关系

（1）文件流是使用缓冲区进行输入输出的，因此创建文件流对象后，会为流对象分配缓冲区。比如 ofstream os; ifstream is;，则会为 os 和 is 分配缓冲区。

（2）每个文件流对象都维护各自的流和缓冲区，它们各不相关。比如 ostream os, os1;，则 os 和 os1 各维护一个文件输出流和缓冲区，若 os 的输出流被破坏，并不会影响到 os1 的输出流。

（3）流与文件是相互独立的，模式是文件的属性，而流状态是流的属性，因此每次打开一个文件时都必须指定文件模式，关闭文件时也不会影响到流状态。

（4）每个文件都维护一个指针，每读取一个字节，指针就向后移一个字节，当指针移到文件尾 EOF（一般值为–1）时，表示文件结束，同时设置 eofbit 状态位为 1。指针的初始位置是由打开文件时的模式指定的。

（5）fstream 流对象虽然既能读取又能写入，但它只维护一个流、一个缓冲区和一个管理文件内容的指针。比如 fstream fs("a.txt"); char c[44]; fs>>c;，假设读到了文件尾，则设置 eofbit 状态位（流被破坏），这时 fs<<"AAA";将不会被写入文件，因为 fs 只维护一个流，且该流被破坏

无法使用。

6. 关闭和打开文件

（1）当文件流对象过期（如程序终止）时，文件会自动关闭。

（2）可以使用 close()成员函数关闭文件。比如 ofstream os("a.txt");，则 os.close();就把流对象 os 打开的文件 a.txt 关闭了。

（3）使用 colse()关闭文件后，并没有把与之关联的流也删除，也不会影响流的状态，只是断开了流与被关闭文件之间的连接。若流处于破坏状态，则关闭文件后，流仍处于破坏状态；若流处于完好状态，则仍可使用该流来重新打开该文件或其他文件。比如 ofstream os("a.txt"); os.close();，则 os 管理的输出流仍然存在，因此可再次使用 os 打开其他文件，如 os.open("b.txt");。

（4）注意：有些编译器会要求在第二次打开同一个文件进行读取之前调用 clear()函数，以恢复流状态。比如 ofstreamos("a.txt"); os.clear(ios_base::failbit); os.close(); os.open("b.txt");，对于 os<<"AA";，若在 VC++ 2010 中，则会把字符"AA"写入到文件 b 中，因为第二次使用 open 时调用了 clear()函数恢复流状态位；但在 VC++ 2005 中，则不会把字符 AA 写入到文件 b 中（流处于破坏状态），但文件 b 仍能被打开，若无此文件则会创建该文件。

（5）关闭流时将使缓冲区被刷新。

7. 打开多个文件

（1）若要打开多个文件，则需要创建多个不同的流分别打开，最多可以打开的文件数依系统而定，通常为 20 个。比如 ofstream os, os1; os.open("a.txt"); os1.open("b.txt");，使用两个流打开两个文件。

（2）使用流对象一旦打开文件，就会保持与该文件的连接，因此不能使用同一个流对象同时打开两个不同的文件，应该在打开另一个文件之前，关闭当前打开的文件。若发生此类错误，则会导致设置流 failbit 状态位，流对象返回值 0，然后就无法再使用该流对象了。比如 ostream os("a.txt");，则 os.open("b.txt");是错误的，正确形式为 os.close(); os.open("b.txt");。

8. 判断文件是否打开成功

（1）不能使用写入文件的流对象对文件进行读取，也不能使用读取文件的流对象对文件进行写入。比如 ofstream os("a.txt"); char c[33];，则 os>>c;是错误的，因为“>>”不是 os 所属类的成员函数。

（2）文件流同样使用流状态位进行控制，即 failbit、eofbit、badbit 状态位被设置，则表示文件流已破坏，不能再使用。

（3）若文件打开失败，则流对象返回值 0；若打开成功，则返回非 0 值，因此可通过直接对文件流对象进行测试，以此判断文件是否打开成功。比如 ofstream os("a.txt");，则可这样测试。

if(os)。

（4）若文件打开失败，则 is_open()函数返回 false；若打开成功，则返回 true。使用 is_open()函数不但可以检测到常规错误，还可以检测到文件以不合适模式打开时的错误。比如 ofstream("a.txt"); if(os.is_open());，若打开成功则返回 1，失败返回 0。

（5）若试图打开一个不存在的文件进行读取时，将设置 failbit 状态位，指的是使用 in 模式打开一个不存在的文件的情形。

（6）若文件打开失败，则会设置 failbit 状态位。若读到文件尾，则会设置 eofbit 状态位。

9．管理文件中的指针位置

（1）C++使用 seekg、seekp、tellg、tellp 成员函数来管理指针的位置，后缀 g 表示 get 对应输入，p 表示 put 对应输出。

（2）seekg 和 tellg 是 istream 类的成员函数，因此凡是继承自 istream 类的子类都可使用这两个成员函数，比如 ifstream 类和 fstream 类。seekp 和 tellp 是 ostream 类的成员函数，同理，ostream 类的子类都可使用这两个成员函数，比如 ofstream 类和 fstream 类。

（3）因为 seekg 和 tellg 是继承自 istream 类的成员函数，因此不能使用 istream 类或不是从 istream 类继承的其他类对象进行调用。seekp 和 tellp 同理。比如 ofstream os;，则 os.tellg();错误，因为 tellg 不是 ofstream 类的成员函数。

（4）seek 和 tell 函数在大多数编译器上都不支持使用 cin、cout、cerr、clog 进行调用，毕竟在使用 cout 输出时定位指针位置没什么用。若使用这些流对象调用 seek 或 tell 函数有可能会破坏流对象，从而使流被关闭。

（5）因为 fstream 流对象只维护一个流和一个指针，因此使用 tellg 和 tellp 函数返回的指针位置的值是相同的。seekg 和 seekp 函数也是对同一个指针的位置进行设置。

（6）seekg 和 tellg 函数原型（简化形式）如下，seekp 和 tellp 原型与之类似。

① istream& seekg(streampos ps);

表示把指针定位于距离文件开始处第 ps 个字节的位置（第 1 个字节编号为 0）。

② istream& seekg(streamoff s, ios_base::seekdir dir);

表示把指针定位于与 dir 距离 s 个字节的位置。dir 可以取 3 个值，其中 ios_base::beg 表示文件起始处；ios_base::cur 表示指针当前位置；ios_base::end 表示文件尾。

③ streampos tellg();

表示返回指针的当前位置（以字节为单位）。

④ 对类型 streampos 的解释。

- streampos 类型接收一个整型值，它是在 I/O 类中使用 typedef 定义的类型。现在该类型是 pos_type 类型的 char 版本的特化体，也就是说，还可以传递 pos_type 类型给 seekg。

- streamoff 与 streampos 类似，streamoff 是 off_type 类型的 char 特化版本。
- pos_type 和 off_type 类型定义于 ios_base 类中，因为 ios_base 类是较新的类，编译器不一定完整实现了它，因此建议使用流来创建该类型的对象。比如正在使用 ofstream 流，则可以这样创建：ofstream::pos_typeps;，而不要这样创建：ios_base::pos_type ps;，因为这样有可能无法识别 pos_type 名称。

（7）示例如下：

```
ifstream is("a.txt");
```

① is.seekg(11);，把指针定位于第 11 个字节处（从文件开始处算起）。

② is.seekg(0);，把指针定位于文件开头。

③ is.seekg(3, ios_base::cur);，从当前位置向后移 3 个字节。

④ is.seekg(−33, ios_base::end);，把指针定位于从文件尾向前的第 33 个字节处。

⑤ ifstream::pos_type ps;或 ifstream::streampos ps1;，则 ps=is.tellg();或 ps1=is.tellg();，表示返回指针的当前位置，并存在 ps 或 ps1 中。

10．二进制文件模式

（1）所有的数据在计算机中都是以二进制形式存储的。以文本形式保存数据，只是为了方便我们阅读，数据在文本和二进制形式之间进行相互转换时，可能会存在精度损失。二进制文件的特点是存取速度快，占用空间小，精度不会损失。但二进制文件在不同的计算机中对同一数据的表示格式可能会不一样。

（2）为什么要使用二进制模式？这里通过示例进行说明，比如 class A{public: int a; char c[44];} A ma; ma.a=3; ma.c="AAABBB";，假设要把整个对象 ma 保存在计算机中，以供其他程序使用，应怎么保存呢？一种方法是使用文本形式分别保存 ma 中各个成员的值；一种方法是使用二进制形式把整个对象 ma 直接保存起来。很明显，二进制形式更具有优点。

（3）使用二进制模式的方法。

① 创建文件流对象时以二进制模式打开文件，比如 ostream os("a.txt", ios_base::out | iso_base::binary);。

② 然后使用成员函数 write 和 read 对数据进行写入和读取。在使用这两个成员函数时，应把要传输的数据强制转换为 char*的格式。假设 ma 是类 A 的对象，则 os.write((char*)&ma, sizeof ma); 表示把整个 ma 对象写入到由 os 打开的文件中。

（4）下面通过示例来说明二进制文件的使用。

示例 16.1：把整个对象 ma 保存到文件 a.txt 中（假设已包含了正确的头文件）

```
class A{public:string c; int a;};   //使用 string 对象保存字符串。注意：类中不能使用指针，比如
                                     //char*c，因为在把数据保存在文件中时，只会保存指针指向的地址，
                                     //当使用另一个文件打开并读取其内容时，只能读取到指针指向的内
```

```
                                        //存地址，而此时不能明确该内存地址在该文件中的具体内容
void main(){A ma;  ma.a=3; ma.c="AAABBB";
     ofstream os("a.txt", ios_base::out | ios_base::binary);  //以二进制形式打开文件，并写入
    os.write((char*)&ma, sizeof ma);   }          //把整个对象 ma 保存到文件 a.txt 中
```

示例 16.2：直接使用文件 a.txt 中保存的 ma 的数据

```
class B{public:string c; int a;};     //类 B 的结构应与示例 16.1 的相同
void main(){  B mb;                    //mb 用于接收文件 a.txt 中保存的数据
   ifstream is("a.txt", ios_base::in | ios_base::binary);  //以二进制形式打开文件，并读取
   is.read((char*)&mb, sizeof mb);       //把文件 a.txt 中的数据读入 mb 中
   cout<<mb.a<<","<<mb.c<<endl; } //输出 3,AAABBB，可见整个对象 ma 的数据被完整地保存并被读取
```

11. 常见的易错情形

示例 16.3：是否读取到了文件尾

```
#include <iostream>
#include <fstream>
using namespace std;
void main(){
     ofstream os("D:\\aa\\a.txt");
     os<<"AA"<<endl;   //文件尾含有一个回车换行符
     os.close();
     os.open("D:\\aa\\b.txt");
     os<<"BB";      //文件尾不含回车换行符
     os.close();
     ifstream is("D:\\aa\\a.txt");
     char c[33];
     is>>c;   //不会设置 eofbit 状态位，因为“>>”会读取到回车换行符并结束读取，此时还未到达文件尾
     is.close();
     is.open("D:\\aa\\b.txt");
     is>>c; }   //设置 eofbit 状态位，因为文件 b.txt 中只有字符 BB 不含末尾的回车换行符，因此“>>”
                //会读到文件尾才结束
```

16.5　字符串流

（1）要使用字符串流，需要包含头文件 sstream。

（2）注意：C++通过 string 类来使用字符串，对于 C 风格字符串，可使用 C 风格字符串流。

（3）使用字符串流的目的，一是让我们能像使用 cout、cin 一样来操控 string 对象；二是让我们可以把数据保存在当前文件的 string 对象中，而不必保存在文件中。

（4）字符串流使用 ostringstream、istringstream 和 stringstream 三个类来实现。使用字符串流与使用文件流原理相似。

（5）ostringstream 构造函数原型（istringstream 和 stringstream 与之类似）如下：

```
ostringstream(); //创建一个空的 ostringstream 流对象，该流对象直接与缓冲区连接
ostringstream(string s); //创建一个与 s 相连接的 ostringstream 流对象，此时缓冲区被 s 初始化
```

（6）字符串流中重要的成员函数 str 用于对流的缓冲区进行操作，其原型如下：

```
os.str(); //把存储在 os 流的缓冲区中的数据转换为 string 对象，并返回
os.str(s); //把 string 类型对象 s 的值复制到流对象 os 的缓冲区，并返回 void
```

（7）ostringstream 流对象的缓冲区与所连接的 string 对象是两个独立的部分，互不影响。对缓冲区的修改并不会影响到与之相连接的流对象，反过来也是一样的。比如 string s="AAA"; ostringstream os(s);，此时缓冲区被 s 初始化，os<<"B";表示把字符 B 写入缓冲区，此时缓冲区中的内容为 BAA，而 s="AAA";其值未变；同理，若 s="BBB";，则缓冲区中的内容仍为 BAA，它们互不影响。

（8）基于上述内容，ostringstream 流对象应与 str 成员函数联合使用。

（9）使用 istringstream 进行读取操作，可以直接把缓冲区中的内容读入到 string 对象中。比如 string s; istringstream is(s); is.str("AAA"); is>>s;，则 s=AAA。

（10）注意：使用 str(s)写入内容时，会清空缓冲区后再重新写入内容，而使用“<<”、put 函数等向缓冲区中写入内容时，不会清空原有内容，而是从头写入数据。比如 ostringstream os; os.str("AAAA"); os<<"BB"; ，此时缓冲区中的内容为 BBAA，若再有语句 os.str("CC");，则缓冲区中的内容为 CC，可通过 cout<<os.str();进行验证。

（11）字符串流的重要应用是格式转换。详见示例。

示例 16.4：把整型等数据类型转换为 string 类型

```
string s; ostringstream os;os<<123; s=os.str();
cout<<s<<endl;  //输出 123，把整型值 123 转换为了 string 类型
```

说明：

① 不能使用 os.str(123);，因为 str 函数要求的形参类型是 string，而 123 是 int 类型。

② string s=123;是错误的，因为 123 是整型值，而 s 是 string 类型，要想把整型转换为 string 类型，应使用 string 流对象。

示例 16.5：把 string 类型转换为整型等数据类型

```
string s="123"; istringstream is(s);  int a;  is>>a; //注意：a=s;是错误的
cout<<a<<endl; //输出 123，把 string 对象转换为了 int 类型
```

示例 16.6：联合使用 istringstream 和 ostringstream 对数据进行格式化转换（也可单独使用 stringstream 流）

```
ostringstream os;
os<<333<<" "<<"end"<<" "<<444<<" "<<"begin"; //以一定格式将数据存储到 os 流中，此时 os 缓冲区
                //中的内容为 333 end 444 begin，空字符是必需的，空白符是向 string 对象输入时的终止符
int a,b; string s1,s2;
istringstream isr(os.str());      //把 os 缓冲区中的内容与 is 流对象连接起来
isr>>a>>s1>>b>>s2;    //此时 a=333，s1=end，b=444，s2=begin，因为“>>”读取数据时遇
```

```
                                //到无效字符就终止读取（若在遇到无效字符前已经读到了有效字符，则不会设置流
                                //状态位），因此 a=333。若无空格作为无效的终止符，则 s1=end444begin，所以
                                //空白符是必需的。若有空白符，则 s1 读完 end 后遇到空白符，结束读取，因此
                                //s1=end。再后面的 b=444，s2=begin
cout<<a<<s1<<b<<s2<<endl;   //输出 333end444begin，格式转换成功
```

16.6　C 风格字符串流

（1）要使用 C 风格字符串流，需要包含头文件 strstream。

（2）C 风格字符串是使用 char 数组或 char 指针进行存储的字符串。

（3）C 风格字符串流（下面简称“字符串流”）使用三个类来实现：istrstream、ostrstream 和 strstream。

（4）使用字符串流，可以像使用 cout、cin 一样使数据在字符数组之间输入输出，同时可以使用字符数组作为数据存储的空间。

（5）字符串流不会在字符串的末尾加入空字符'\0'，该字符需要自己进行设定。

（6）ostrstream 构造函数原型：ostrstream(char* c, int n, int m=ios_base::out);。表示建立一个与 c 相连接的字符串流，以便向 c 中写入字符，最多只能向 c 中写入 *n* 个字符。该流默认使用 out 文件模式（即写入操作，并清空字符数组）。

示例：char c[33]; ostrstream cos(c, 10); cos<<"aaa"<<ends;，表示把字符 aaa 存入字符数组 c 中。因为缓冲区的大小为 10，因此最多可以使用 cos 流对象向 c 中存入 10 个字符的字符串。最后的 ends 表示刷新缓冲区，并在末尾添加一个空字符'\0'。

（7）istrstream 构造函数原型： istrstream(char *c); 和 istrstream(char *c, int n);。表示建立与 c 相关联的字符串流，以读取 c 中的字符。其中第 2 个构造函数表示只能读取 c 中的最多 *n* 个字符。比如 char *c="dddd"; char *s; istrstream cis(c,3); is>>s;，表示从 c 中读取 3 个字符到 s 中。

（8）strstream 构造函数原型：strstream(char *c, int n, int m);。其中 m 是文件模式，这里使用 ios_base::out|ios_base::in 就可以了。比如 strstream scs(c, 4, ios_base::out|ios_base::in)，表示建立与 c 相关联的字符串流，以便进行读取和写入，但一次只能读取或写入 4 个字符。

（9）字符串流可以把 int 类型等数据类型转换为 C 风格字符串类型，或进行相反的转换。

示例 16.7：把整型转换为 C 风格字符串类型

```
char c[33]; ostrstream cos(c,22); cos<<123<<ends; cout<<c<<endl;
```

输出 123，可见此处把整型的 123 转换为 C 风格字符串类型。

示例 16.8：把 char 类型转换为整型

```
char c[33]="123"; int a; istrstream ios(c,22); ios>>a; cout<<a<<endl;
```

输出 123，可见此处把字符串类型的 123 转换为整型。

第 17 章

异常专题

（1）对象：指的是某种类型所占据的一片连续的内存单元。因此对象指的是变量、临时对象和类的对象等，在本文上下文应能区分开来。

（2）本章所有示例都假设包含了正确的头文件和名称空间。

1．使用异常的目的

（1）异常：指的是程序中的差错。

（2）为什么要使用异常。其实处理异常的方法可以使用简单的 if 语句来完成，比如 if(!a=0) b/a;，就可以解决被 0 除的错误，但这种语句必须在每个有可能出现这种错误的函数中进行设置，若一个庞大的程序，函数众多，而且处理异常的情形也众多，则这些函数会包含大量的处理异常的代码，这样程序就变得很复杂、很庞大。C++中的异常就是为解决这种情况而产生的，在使用异常时，在函数内部不需要包含有处理异常的程序，凡是有异常需要处理的情形，就传递一个消息给一个专门处理异常的函数来完成，这样主函数就可以专注于实际的任务，而不再承担处理异常的任务，从而使主函数更简洁、高效。

（3）异常，说简单点就是 if 语句的另一个版本，只是异常的条件判断、跳转方式不同。

2．异常的语法结构

```
try{含有“throw 表达式;”的语句;} catch(异常声明){...} catch(异常声明){...}...
```

示例如下：

```
void f(){ int a; try{ throw a;}catch(int){语句1;} 语句2; }
```

（1）异常执行流程简介：由 try 块内的“throw a;”语句抛出一个异常对象（简称为异常），然后把抛出的异常对象的类型（本例为 a 的类型）与 try 块后的 catch 子句中的异常声明的类型 int 进行匹配，若匹配，则执行该 catch 子句中的语句 1（处理异常），然后继续执行 catch 子句后之后的语句 2。若 throw a 抛出的异常与 catch 子句不匹配，则发生错误。

（2）异常由三部分组成：try 块、throw 语句和 catch 子句。各结构基本语法要求如下。

try 块基本语法要求：

① try 块中可以是任何合法的 C++语句，try 块声明了一个局部作用域，因此在 try 块中声明的名称，在try块外不可引用，包括catch子句都不能引用。比如void f() try{int a; }catch(int){a=2;}错误，名称 a 在 catch 子句中不可见。

② try 块可以使用的地方与无名的大括号{}相似，因此 try 块语句引入的作用域一般都在函数内部，比如 class A{ try{int a;}catch(...){} };错误，不能在类内部引入局部作用域。比如，void f(){ int a; try{int b;}catch(...){}}正确，try{void f(){}}catch(...){}错误。

③ 也可以把整个函数块引入为 try 块，比如 void f() try{int a;} catch(...){}。注意：try 位于小括号后。

④ 异常必须经由 try 块抛出（直接或间接），否则异常将不能被处理，但 try 块也可以不抛出异常。比如 void f(){try{cout<<"A"<<endl;}catch(int){}} 正确，try 块可以不抛出异常。但 void f(){int a=1; throw a;}错误，即抛出的异常未被处理。

catch 子句基本语法要求：

① try-catch 结构是一个语句的整体，因此 try 块和 catch 子句必须一起出现。也就是说，try 块和 catch 子句均不能单独使用。比如，void f() try{};是错误的，void f(){}catch(...){}也是错误的。

② catch 子句列表：try 块之后可以有多个 catch 子句，但一个 catch 子句不能有多个 try 块。多个 catch 子句可组成 catch 子句列表。比如，void f()try{}catch(int){} catch(float){}是正确的，但 void f() try{} try{} catch(...){}是错误的。

③ catch 子句必须紧跟着 try 块之后，中间不能插入任何语句（哪怕仅仅一个分号也不行），比如，void f()try{}cout<<"A"; catch(...){}是错误的。

④ try 块和 catch 子句后的花括号不能省略。

⑤ catch 后的异常声明可以只有类型名而无变量名，catch 根据异常声明的类型是否与 throw 抛出的异常对象的类型相匹配作为判断是否执行 catch 子句中语句的依据。比如，void f()try{throw 3;}catch(int){}是正确的，应执行 catch 块语句。再如，void f() try{throw 3.3;}catch(int){};，因为 3.3 的类型与 int 不匹配，因此不执行 catch 子句。即此语句抛出的异常未被 catch 处理，因而程序会出错。

⑥ catch 后的异常声明也可以有变量名（但类型不能省略），这时变量的值就是 throw 抛出的异常对象的副本。也就是说，catch 后声明的变量与 throw 抛出的异常对象是按值的方式进行传递的。比如 void f()try{int a=4; throw a;} catch(int m){}，则 m 将拥有值 4。

⑦ catch 后的异常声明的类型必须是完全的类型，因此若是类则必须定义，仅有类的声明是不行的，比如 class A;，则 void f() try{}catch(A){}错误。

⑧ catch 一次只能捕获一种类型的异常（省略号除外），比如 void f()try{}catch(int,float){}

错误。

⑨ catch 后的类名可以使用省略号“...”，表示 catch 能捕获所有类型的异常，比如 void f()try{throw 3;}catch(...){}正确，catch 语句会被执行。

throw 语句语法基本要求：

① throw 语句由“throw 表达式;”组成，其中，表达式可以是任何合法的表达式。

② throw 语句可以不直接出现在 try 内，它可以间接地出现在 try 块内部调用的函数中。比如 void g(){throw 3;} void f()try{ g();} catch(int){}正确，throw 出现在 try 块内调用的 g 函数中。这是不在函数内部进行错误处理，而是把错误处理交由专门的一个函数来处理的关键，此处异常的处理由函数 f 进行，函数 g 在遇到异常处时只管抛出异常即可。

③ throw 语句可以抛出任何类型的表达式，包括自定义的类型，但不能只是类型。比如 class A{}; throw A;错误，但 throw A();正确。

3. 基本概念

（1）异常对象、异常信息、异常：异常对象是由 throw 创建，并使用 throw 中的表达式进行初始化的一个特殊对象，简称为异常。异常对象相当于是使用 throw 抛出的表达式的副本，可见异常对象与 throw 表达式是两个不同的主体。异常对象常用以判断应该调用哪一个 catch 子句，因此也被称为异常信息。使用 throw 抛出异常对象的过程称为抛出异常。比如 throw 3;，则创建一个 int 类型的异常对象，并使用整数 3 进行初始化。throw 表达式产生三个相同的名称：异常对象、异常、异常信息。

（2）异常声明、catch 形参、catch 对象：异常声明位于 catch 后的小括号内，其声明形式与函数形参的声明形式相同，因此异常声明也称为 catch 形参，而把异常声明的对象称为 catch 对象。比如 catch(int a){}，其中的 int 就是异常声明，也称为 catch 形参，变量 a 是异常声明的对象，称为 catch 对象。

（3）由以上两点可知，异常对象、异常表达式、catch 对象是三个各不相关且相互独立的三个对象。区分这一点相当重要。

（4）异常处理程序（块、语句）：因为 catch 子句中的语句是用于处理发生异常时的情形的，因此把 catch 子句常称为异常处理程序、异常处理块或异常处理语句。

（5）捕获异常：若 catch 子句中异常声明的类型与 throw 抛出的异常对象的类型相匹配，则称为 catch 捕获异常，此时 catch 子句会执行；若 catch 子句未被执行则表示该异常未被捕获，因此异常也未被处理。比如 void f()try{throw 3;}catch(int){};，此处 int 与 3 匹配，会执行 catch 子句处理异常。再比如 void f()try{throw 3.3;}catch(int){};，此处 int 与 3.3 不匹配，因此 catch 子句不会被执行。

（6）捕获某异常：catch 中异常声明的类型，代表了 catch 能捕获什么类型的异常（对象），

或者说 catch 能处理什么类型的异常。比如 catch(int)，表示 catch 能捕获 int 类型的异常。

（7）异常处理方式：可以以任何形式对异常进行处理，哪怕 catch 后的复合语句是空的，那也是一种处理异常的方式。

（8）异常必须被处理，因为存在异常就表示程序是在非正常的执行，因此若未找到处理异常的 catch 子句，程序就会调用 C++标准库中的 terminate()函数来非正常终止程序。比如 void f(){throw 3;}，则将调用 terminate 函数终止程序。

4．异常原理

（1）try 块的功能：有可能会出现异常的语句应包含在 try 块中（即 try 后面的花括号中）。

（2）throw 语句的功能，见下面示例。

```
void g(){throw 3;} void f()try { g(); }catch(int){}
```

① 其中 throw 语句位于 try 块内调用的函数 g 中，而函数 g 可以位于任何函数 f 可调用的地方。

② throw 语句的功能：throw 语句用于抛出异常对象，异常对象相当于是 if 中的条件表达式，这个对象将用于判断应该调用哪一个 catch 子句，与条件表达式不同的是，这个对象被 throw 抛出，而且 throw 可以位于其他地方。

③ throw 语句的位置：可以在其他任何合法的地方使用 throw 语句抛出异常，不一定要在 try 块内部，try 块内只需包含 throw 语句即可。

④ 函数与代码分开：由示例可见，异常把处理错误的代码与函数实现的代码分离开了，比如函数实现的代码位于函数 g 中，在函数 g 中凡是出现错误（异常）的情形，都可以使用 throw 抛出来，而函数 g 本身可以不处理任何错误。错误的处理由包含 try-catch 结构的函数 f 中的 catch 子句进行处理。也就是说，由调用函数 g 的函数 f 专门负责处理由函数 g 所引发的错误（异常）。

⑤ 调用函数可能引发异常：由示例可见，在调用函数时，若该函数使用 throw 抛出异常，则必须在调用函数时使用 try-catch 结构对该函数抛出的异常进行处理，比如 void g(){f();}，若函数 f 抛出 int 类型异常，则调用时必须进行处理，即 void g(){ try{f();}catch(int){...}}。

⑥ throw 的跳转功能：由于 throw 语句可以从 try 块之外的函数直接跳至与 try 块配对的 catch 子句中，因此 throw 语句具有跳转的功能。

（3）catch 子句的功能：catch 子句与 try 块配对，主要作用是捕获异常，并对异常进行处理。若 catch 语句后的异常声明的类型与 throw 抛出的异常对象的类型匹配，则调用花括号内的语句对异常进行处理。若不匹配，则继续寻找下一个 catch 子句，直到找到匹配的为止。若最后都未找到匹配的 catch 子句，则引发错误。注意，catch 匹配的是 throw 抛出的异常对象的类型，而不是值，只要类型匹配，就是合法的匹配。比如 void f() try{throw 3;}catch(int){cout<< "A"<<endl;}，因为 throw 后的 3 为 int 类型，与 catch(int)匹配，所以执行 catch 中的输出语句，输出字符 A。

此处 throw 3;与 throw 4;、throw5;等都是抛出的同一类型的异常，因此都与 catch(int)相匹配。

（4）根据以上各结构的功能可以看到，异常信息由 try 块内直接或间接的通过 throw 抛出，然后在 catch 后的大括号中进行处理。也就是说，异常发生在 try 块内，而异常的处理发生在 catch 内，它们之间沟通的信息是由 throw 抛出的。而抛出这个信息的位置并不确定。

5. 异常的执行流程

基本流程：

嵌套函数调用的出栈过程见第 9 点（对象的创建与销毁）。

（1）若 try 块未抛出异常，则执行 try 块中的所有代码，与 try 块相关联的所有 catch 子句被忽略，直接执行 catch 子句后面的语句。

（2）若语句通过 throw 抛出异常，则跟在 throw 语句后的所有语句都被跳过，转去执行与之匹配的 catch 子句处理异常，执行完 catch 子句后，程序继续执行在该 catch 子句列表的最后一个 catch 子句之后的语句，注意：程序不会返回到使用 throw 抛出异常的地方继续执行。

示例如下：

```
void f() {int a; try{throw 3;  cout<<"A";}
   catch(int){cout<<"B";}catch(float){cout<<"C";} catch(double){cout<<"D";}
      cout<<"E"; }
void main(){  f();}
```

说明：

① 当 throw 抛出异常后，程序直接跳过 throw 3;后的 cout<<"A"；转到与 throw 3;匹配的 catch(int)子句，并执行该语句中的内容，输出 B。

② 然后程序执行与 try 块相关联的 catch 子句列表的最后一个 catch(double)子句之后的语句，输出 E。

③ 可见程序执行完处理异常的 catch(int)子句后，并没有回到抛出异常的地方 throw 3;，执行其后的 cout<<"A";语句，因此 cout<<"A";一直未被执行。

（3）若最后没有找到与之匹配的 catch 子句处理该异常，则程序调用 terminate()函数，该函数调用 abort()函数，终止程序。注意：VC++ 2010 不调用 terminate 函数。

匹配 catch 子句的过程：

（1）若抛出异常的 throw 语句位于 try 块中，则查找与该 try 块相连的 catch 子句列表，若找到匹配的 catch 子句，则执行该子句处理异常。

（2）若第一步未找到与之匹配的 catch 子句，则在调用该函数的地方继续查找 catch 子句；若找到匹配的 catch 子句，则处理异常；若仍未找到，则继续向上在调用此函数的地方查找 catch 子句，直到找到与之匹配的 catch 子句。若最后没有找到匹配的 catch 子句，则调用 C++标准库中的 terminate 函数终止程序。

示例 17.1：catch 子句匹配过程

```
void f() try{throw 3;  cout<<"A";}catch(char){cout<<"B";}
void f1()try{f();}catch(float){cout<<"C";}
void f2(){f1();}
void f3()try{f2();}catch(int){cout<<"D";}
void main(){f3();}
```

分析：

① f 函数抛出异常对象 throw 3。

② 首先在 f 函数处查找与 try 块关联的 catch(char)子句，因为 throw 3 与 char 不匹配，因此跳至调用 f 函数的函数 f1 处。

③ 在函数 f1 处与 try 相关联的 catch(float)子句也与 throw3 不匹配。再跳至调用函数 f1 的函数 f2 处。

④ 因为函数 f2 没有 try 块，所以直接跳到调用函数 f2 的函数 f3 处。

⑤ 在函数 f3 处与 try 块关联的 catch(int)子句与 throw 3 匹配，执行该 catch 子句，输出 D。

（3）栈展开：第 2 点讲解的寻找 catch 子句的过程也被称为栈展开，即函数名称和形参被压入栈，在离开函数时再弹出栈。

（4）try 块后若有多个 catch 子句列表，则 catch 子句将按出现的次序进行匹配，并且选中第一个找到的相匹配的 catch 子句，之后的 catch 子句即使是相匹配的也不会再被执行。因此对于处理所有类型的 catch(...)子句都应放在 catch 子句列表的末尾，否则在 catch(...)之后的所有 catch 子句将不起作用，而且会产生一个编译错误。比如 void f()try{throw 3;}catch(int){}catch(...){}，则使用的是第一个找到的相匹配的 catch(int)子句，其后的 catch(...)子句即使是匹配的也不会被执行。再如 void f()try{throw 3}catch(...){}catch(int){}会发生编译错误。

（5）继承时，也应把父类的 catch 子句放在子类的 catch 子句之后，详见第 7 点。

6. 异常对象与 throw 表达式

（1）异常对象（简称异常）：是由 throw 创建，并使用 throw 中的表达式进行初始化的一个特殊对象，该对象由编译器管理，并且保证存储在任何可访问的 catch 子句都能访问的地方。比如 throw 3;，则编译器会使用初始值 3 来创建一个 int 型的异常对象，该对象与数值 3 是两个各自独立的主体。

（2）异常对象是使用 throw 表达式进行初始化的独立主体，因此异常对象相当于是 throw 表达式的副本，但 throw 表达式与异常对象是两个互不相关的独立主体，对异常对象的修改不会影响 throw 表达式，比如 class A{}; void f(){A ma; throw ma;}，则 throw ma;之后使用 ma 作为初始值创建的一个异常对象。假设名为 mas，则异常对象使用复制构造函数创建，即 A mas(ma);。

（3）为什么要创建异常对象：由 throw 表达式抛出的对象若是局部对象，则只在该作用域中有效，一旦离开该作用域，那么这个对象就不存在了，这样就不能使用该局部对象去查找与

之匹配的 catch 语句，而创建的异常对象，则可以在找到与之匹配的 catch 之前一直存在，异常对象会一直保持到所有 catch 语句处理完毕才会销毁。比如 void g(){int a=3; throw a;} void f()try{g();}catch(int){};，在 g 函数中抛出的对象 a，是个局部对象，在该对象被抛出后 a 就被销毁了，因此无法使用局部对象 a 查找到位于 f 函数后的与之匹配的 catch(int)语句。因此在 throw a;之后会创建一个异常对象，然后使用异常对象查到与之匹配的在 f 函数之后的 catch(int)语句，执行完 catch 语句之后，异常对象被销毁，在此过程中，异常对象会一直存在。

（4）从异常对象的创建过程可知：在 throw 后的表达式，必须是可复制的类型，且析构函数要能访问。也就是说，复制构造函数和析构函数不能是私有的。比如 class A{A(A&){} public:A(){}}; void f()try{ throw A();}catch(A){}是错误的，因为复制构造函数不可访问。注意：VC++ 2010 不一定完全遵守此规则，见下面示例：

```
class A{ A(A&){cout<<"A&"<<endl;} public:A(){}}; //复制构造函数为私有，不可访问
void f()try{ A ma; throw ma;}//输出 A&，可见在 VC++ 2010 中仍能调用复制构造函数创建异常对象
catch(A&){} //catch 为引用，因此不需要使用复制构造函数，否则会出错
```

（5）不存在数组或函数类型的异常对象：与函数参数传递过程类似，被抛出的数组或函数类型的异常对象，会被自动转换为相应类型的指针，因此不存在数组或函数类型的异常。比如 void f(){int a[3]={0}; try{throw a;}catch(int *){} };，此处 throw a;抛出的数组 a，会被自动转换为 int*指针。

（6）继承时异常对象的类型。

① 因为异常对象是经过复制构造函数而创建的，因此异常对象的类型是指针的静态类型，而不是指针指向的动态类型（实际类型）。也就是说，指针的动态类型不会用来创建异常对象。比如 class A{}; class B:public A{}; void f()try{B mb; A *p=&mb; throw *p; }catch(B){}是错误的，异常未被捕获，因为由*p 创建的异常对象的类型为*p 的静态类型 A，而不是*p 指向的动态类型 B。

② 若 throw 表达式的指针是一个指向子类对象的父类类型指针，则子类对象将被分割为只剩父类部分（其他有关继承的规则都与继承时相同，详见第 13 章）。比如 class A{}; class B:public A{}; void f()try{B mb; A* p=&mb; throw *p;}catch(A){};，则使用 throw* p;创建的异常对象将被分割为只剩父类部分。

（7）抛出局部对象的指针，必须确保该指针在 catch 子句处仍然存在，否则出错，原因与返回指向局部对象的指针相同。

7．异常对象与 catch 对象

（1）异常对象与 catch 对象之间的参数传递规则：异常对象以按值的方式传递给 catch 对象（这与函数形参的参数传递规则相同，详见第 7 章）。

① 若 catch 形参不是引用，则把异常对象的副本复制给异常声明对象，catch 子句将对该副

本进行操作，不会影响到异常对象本身。

② 若 catch 形参是引用，则 catch 对象将是异常对象的别名，在 catch 中对 catch 对象的改变，会影响到异常对象，但 catch 对象的修改并不会影响到 throw 表达式指定的变量，因为 throw 表达式与异常对象是两个独立的主体。此规则对于异常再次抛出时有一定影响。比如 int a=1; void f()try{throw a;}catch(int& m){m=2; cout<<a<<endl;}，则 f();输出 1，可见 catch 中 m=2;对 m 所引用的异常对象的修改，并没有使 throw 表达式指定的变量的值发生改变。

（2）继承关系时的异常对象与 catch 对象之间的参数传递。

① 若 catch 形参是父类类型，则可以捕获到所有子类类型的异常对象。若 catch 形参是子类类型，则不能捕获到父类类型的异常对象。与继承时的规则相同，catch 子句不能使用子类中特有的成员，但会把子类分割为父类子对象。比如 class A{}; class B:public A{}; void f()try{throw B();}catch(A){};正确，父类 catch 形参 A 可以捕获子类异常对象 B();。

② 若 catch 形参是父类类型的指针或引用，则同样可以捕获到所有子类类型的异常对象，而且可以使用虚函数的特性（详见虚函数章节）。

③ 注意：以上规则必须以公有制方式继承才是正确的。比如 class A{}; class B:A{}; void f()try{throw B();}catch(A){};错误，因为类 B 以私有继承方式从类 A 继承而来，错误发生在 catch(A)处，若改为 catch(B)则正确。

（3）异常对象与 catch 对象之间所允许的类型转换。

① 允许从非 const 到 const 的转换，即 throw 抛出非 const 对象，catch 可以是该类型的 const 引用（其他类型类似）。比如 void f()try{int a=2; throw a;}catch(const int &){};正确。

② 允许从子类到父类的转换，即 throw 抛出子类，catch 可以是该类的父类（前文已经讲过）。

③ 数组和函数类型转换为相应类型的指针，其实异常对象根本不存在数组和函数类型，比如 void f()try{int a[3]={0}; throw a;} catch(int *){}正确。

④ 除上述转换之外，其他转换是不允许的。比如 void f()try{short a=3; }catch(int){}错误，异常未被处理，这里 short 不会被提升转换为 int 以与 catch(int)进行匹配。

（4）捕获成员初始化列表中抛出的异常：其方法是把关键字 try 放在成员初始化表之前。比如 class A{public: A(int *p)try:p1(p), a(3){}catch(...){}};。注意，关键字 try 位于“:”之前。这样，在成员函数初始化列表中的成员抛出异常时（比如 p1 抛出异常），构造函数就能捕获到。

8. 再次抛出

（1）可以把抛出的异常再次进行抛出，其语法就是使用空表达式的 throw 语句，即 throw;。

（2）空 throw 语句只能出现在 catch 或在 catch 调用的函数中。

示例如下：

```
void g(){ try{throw;} catch(int){} } void f()try{throw 3;}catch(int){g();} f();
```

g()中的空 throw 语句是正确的，因为函数 g 位于 catch 语句中。但直接调用函数 g();则是错误的，因为 throw;不能在函数中出现。

（3）再次抛出时的异常执行流程：再次抛出异常时，会直接跳至调用抛出异常的函数中查找匹配的 catch 子句。

示例如下：

```
void g()try{int a=1; throw a;}
   catch(int m){ m=2; throw;} //再次抛出由 throw a 抛出的异常，此时直接跳到调用函数 g 所在的 f 函
                              //数中，对再次抛出的异常查找匹配的 catch 子句
void f()try{g();}catch(int n){} //找到与再次抛出异常相匹配的 catch 子句后，在此子句中处理异常
f();   //此处必须调用 f 函数，若调用 g 函数则是错误的
```

（4）被再次抛出的异常是以前的异常对象，而不是 catch 对象。

示例如下：

```
void g()try{int a=1; throw a;}
   catch(int m){ m=2; throw;} //再次抛出的是由 a 初始化的异常对象，其值为 1
void f()try{g();}
   catch(int n){cout<<n;} //输出 1，可见 g 后的 catch 再次抛出的是异常对象，而非 catch 对象 m
f();          //此处必须调用 f 函数，若调用 g 函数则是错误的
```

（5）异常对象能否被修改取决于传递给 catch 对象的方式。再次注意，throw 表达式所表示的变量与异常对象，是两个独立的主体。

① 若 catch 声明是非引用，则异常对象按值传递给 catch 对象，此时 catch 对象是异常对象的副本，对 catch 对象的修改不会对异常对象产生影响。

② 若 catch 声明是一个引用，则 catch 子象的修改会影响到异常对象，这时再次抛出的异常对象将是被修改后的异常对象。但不会影响 throw 表达式。

示例如下：

```
void g()try{int a=1; throw a;}
   catch(int &m){ m=2; throw;} //因为 m 是引用，因此 m=2 修改了异常对象的值
void f()try{g();}
   catch(int n){cout<<n<<endl;} //输出 2，可见再次抛出时，抛出的是修改后的异常对象
f(); //此处必须调用 f 函数，若调用 g 函数则是错误的
```

9. 对象的创建与销毁

（1）异常对象的创建与销毁：异常对象总是在抛出点处被创建，直到异常对象的所有 catch 子句执行完之后才会被销毁。要注意销毁的时机。

（2）局部对象的创建与销毁：在离开抛出异常所在的作用域，以及离开调用该函数的作用域时，被销毁。

（3）C++异常机制保证从 throw 抛出异常一直到被 catch 捕获期间，对已创建的局部对象按

逆序进行销毁。

示例 17.2：异常对象的创建与销毁

```
class A{public:A(A&){cout<<"A&"<<endl;} A(){cout<<"A"<<endl;} ~A(){cout<<"~A" <<endl;} };
void f()try{A ma; throw ma;}catch(A x){cout<<"F"<<endl;}
void main(){f();}
```

分析：

① try 块中共创建了两个对象，局部对象 ma 和使用复制构造函数创建的异常对象，因此首先输出 A,A&。

② 在 f 函数后找到与异常对象匹配的 catch 子句。但此时并未离开 f 函数，无输出

③ 通过异常对象使用复制构造函数创建 catch 对象 x，然后离开 f 函数，跳转到 catch 子句。此时，在 f 中的局部对象 ma 被销毁，此处应注意离开 f 函数的时机，先创建 catch 对象再离开 f 函数。因此输出 A&, ~A。

④ 转入 catch 子句，输出 F。

⑤ catch 子句结束，销毁异常对象和 catch 对象。注意：异常对象到此处才被销毁。输出~A, ~A。

⑥ 程序最后输出 A, A&, A&, ~A, F, ~A, ~A。

⑦ 注意，若 throw A();抛出临时对象，则可能会有不同的结果。

示例 17.3：验证异常对象是在所有的 catch 语句执行完之后才被销毁的

```
class A{public:A(){cout<<"A"<<endl;} ~A(){cout<<"~A"<<endl;} };
void g()try{A ma; throw ma;}catch(A){ throw;}  //创建 ma，异常对象使用复制构造函数创建，结
                                               //束 g 后销毁 ma，输出 A,~A
void f1()try{g();}catch(int){}  //无输出
void f()try{f1();}catch(A){} //输出~A，catch 结束后销毁的是异常对象，因此多输出了一个~A
void main(){  f();}
```

说明：在 g 函数处使用复制构造函数创建异常对象，该对象被再次抛出，异常对象在离开此处的 catch 后不会被销毁，直到 f 函数后的 catch 子句执行完后才被销毁。

示例 17.4：局部对象、异常对象的创建与销毁

```
class A{public:A(){cout<<"A"<<endl;} ~A(){cout<<"~A"<<endl; }};
class B{public:B(){cout<<"B"<<endl;}};
void g(){A ma; throw ma; B mb;}
void g1(){A ma1; g(); B mb;}
void f()try{A ma2; g1();}  catch(A){cout<<"F"<<endl;}
void main(){f();}
```

基本过程：

① 首先找到 f 函数，创建对象 A ma2;，然后调用 g1，转到 g1 函数，输出 A。

② 在 g1 函数内创建对象 A ma1;，然后调用 g，转换 g 函数，后面的 B mb;未被执行，输

出 A。

③ 在 g 函数内创建对象 A ma;，然后抛出异常，B mb;未被执行，此时输出 A。

匹配异常对象：

① 创建异常对象 throw ma;，通过类 A 的复制构造函数，使用 ma 创建异常对象，其形式大约为 A mas(ma);，假设 mas 为异常对象的名称。此时无输出。

② 使用异常对象查找与之匹配的 catch 语句，因为在函数 g 中没有找到与之匹配的 catch 语句，因此跳至调用函数 g 的函数 g1，此时会跳过其后的所有语句，因此 g 函数中的 B mb;未被执行。此时离开函数 g，会销毁在 g 中创建的局部对象 ma，输出~A;。

③ 在 g1 中也未找到与异常对象匹配的 catch 语句，离开函数 g1，跳至调用函数 g1 的函数 f。离开 g1 函数之前，销毁在 g1 中创建的对象 ma1，输出~A;。

④ 在 f 中找到与异常对象匹配的 catch(A)子句，在执行 catch 子句之前(即离开函数 f 时)，销毁 f 中创建的对象 ma2，输出~A;，然后执行 catch 子句，输出 F。

⑤ catch 子句执行完毕，销毁异常对象，输出~A;。

⑥ 整个程序输出 A, A, A, ~A, ~A, ~A, F, ~A，因为异常对象是使用复制构造函数创建的，所以只输出了 3 个 A。

⑦ 若 catch 中声明了对象，即 catch(A)修改为 catch(A x);，则在异常对象与 catch 对象 x 之间还会进行按值传递，从而调用复制构造函数创建 x，在 catch 执行完之后，会销毁 catch 对象 x。

示例 17.5：函数嵌套调用出栈过程与对象的销毁

```
class A{public:A(){cout<<"A"<<endl;} ~A(){cout<<"~A"<<endl; } A(A&){cout<<"A&"<<endl;}};
class B{public:B(){cout<<"B"<<endl;} ~B(){cout<<"~B"<<endl;}};
class C{public:C(){cout<<"C"<<endl;} ~C(){cout<<"~C"<<endl;}};
class D{public:D(){cout<<"D"<<endl;} ~D(){cout<<"~D"<<endl;}};
void g()try{A ma; throw ma; D md1;}  catch(C x){cout<<"gc"<<endl;}  //A, A&,
void g1()try{B mb; g();  D md2;}catch(A  y){cout<<"G1C"<<endl;}      //B, A&,~A,F
void f()try{C mc; g1();D md3; cout<<"FFF"<<endl;} catch(int){cout<<"fc"<<endl;} //C
void main(){cout<<"G"<<endl; f(); cout<<"END"<<endl;}
```

分析：

① 创建类 C、B、A 的对象过程略，按调用顺序创建，依次输出 C, B, A。

② 在函数 g 中遇到 throw ma;，使用复制构造函数创建异常对象，所以输出 A&。

③ 使用异常对象查找匹配的 catch，在函数 g1 中找到 catch(A y);，使用复制构造函数创建 catch 对象 y，因此输出 A&。注意，此时并未离开函数 g，因为程序没有进入其他作用域，所以仍在函数 g 的作用域。最后输出 A&。

④ 因为 catch(A y)子句在函数 g1 作用域之外，因此在进入 catch(A y)子句之前，会离开函数 g 和 g1 的作用域。此时首先销毁 g 中的局部对象 ma，输出~A；然后销毁 g1 中的局部对象

mb，输出~B。此处可见 g1 中的 D md2;被跳过，未被执行。最后输出~A, ~B。

⑤ 执行 catch(A y)中的语句，输出 GIC，然后 catch 子句执行完毕，离开 catch 子句，准备执行 catch 后的语句，在离开 catch 之前销毁 catch 对象 y 和异常对象，因此又输出~A, ~A。最后输出 GIC, ~A, ~A。

⑥ 离开 catch(A y)子句，执行下一条语句。因为 catch 位于函数 g1 之后，表示函数 g1 调用结束，因此返回函数 f 中的调用语句 g1();后，执行其后语句，创建对象 D md3;，输出 D，然后输出 FFF，最后输出 D, FFF。

⑦ 离开 f 函数，销毁 f 中的局部对象，按创建先后顺序，首先销毁对象 md3，输出~D，再销毁对象 mc，输出~C，最后输出~D, ~C.

⑧ 离开 f 函数，返回调用 f 函数的 main 函数，执行其后的语句，最后输出 END。

⑨ 整个程序依次输出：G ,C, B, A, A&, A&, ~A, ~B, G1C, ~A, ~A, D, FFF, ~D, ~C, END。

（4）注意：某些编译器可以不使用临时对象而直接创建异常对象。比如，class A{public:A(A&){cout<<"A";} A(){}}; void f()try{throw A();}catch(A){};，因为 A();创建的是一个临时对象，这时有些编译器可以不使用这个临时对象而直接创建异常对象。若使用了临时对象创建异常对象，则程序会输出 A&，若没有通过临时对象创建异常对象，则无输出。在 VC++ 2010 中，此种情况可直接创建异常对象。

（5）在构造函数内部抛出异常时，则对象有可能只构造了一部分，也就是有些成员被初始化了，有些还没有被初始化，这时即使对象只构造了一部分，C++也必须要适当地销毁已构造的成员。

（6）不建议使用析构函数抛出异常。若在处理某个异常时，在销毁某个类对象的过程中，这个类对象的析构函数又抛出另一个未经处理的异常，则会导致调用 terminate 函数终止程序，比较容易出错，因此不建议使用析构函数抛出异常。

（7）若抛出异常发生在动态分配内存的资源释放之前，则该资源不会被释放。比如 void f()try{int *p=new int; throw 3; delete p;}catch(...){};，则指针 p 的资源未被释放。可以使用类来解决动态内存的释放问题，即在需要使用动态内存的地方，创建该类的对象即可。

10．异常规范（VC++ 2010 不支持异常规范）

（1）异常规范：由函数形参表之后的 throw 关键字与异常类型列表组成。比如 void f()throw(int,float);就是异常规范，表示该函数会抛出 int 和 float 类型的异常。throw 后的小括号中应是类型名，多个类型名应使用逗号分隔。

（2）异常规范的主要作用是告诉编译器，此函数只能抛出由异常规范指定的那些类型的异常，若抛出其他类型的异常则调用 unexpected 函数，该函数最终调用 abort 函数结束程序。

（3）同一函数的重复声明（或定义）必须指定相同类型的异常规范。也就是说，同一函数

的不同异常规范指定的类型是不能叠加的。比如 void f()throw(int);，则 void f()throw(float);错误，因为两个声明的异常规范的类型不相同。

（4）只要在异常离开函数前被 catch 捕获到了（或处理掉了），则函数内部就可以抛出与异常规范类型不相同的异常。比如 void f()throw(int){ int a; try{throw 3.3}catch(double){}}正确，f 内部抛出的 double 型的异常，在离开函数 f 之前被函数内部的 catch 子句处理掉了。

（5）违反函数异常规范的错误在编译时不会产生错误，只有运行时刻才能检测出来。

（6）若异常规范为空，则保证该函数不会抛出任何异常。比如 void f()throw(){throw 3;}错误，函数 f 不抛出任何异常。

（7）若函数未指定异常规范（即常规函数），则该函数可抛出任何类型的异常。

（8）抛出的异常类型与异常规范中指定的类型之间不存在类型转换（有继承关系时除外）。比如 void f()throw(int){short a=3; throw a;}错误，不允许类型转换。

（9）若异常规范指定的是父类类型的对象或指针，则抛出的异常类型可以是子类类型的对象，但该子类必须是以公有继承的方式从异常规范指定的父类继承而来的。比如 class A{}; class B:public A{}; void f()throw(A){throw B();};正确，若类 B 是私有继承自类 A，则错误。

（10）指向函数的指针也可以指定异常规范，其规则与给常规函数指定异常规范相同，比如 void (*pf)()throw(int);。

（11）给具有异常规范的指向函数的指针赋值时，被赋值的函数的异常规范必须比指向函数指针的异常规范更严（或相同）。比如 void (*pf)()throw(int, double); void f()throw(int){} void f1()throw(float){}，则 pf=f 正确，f 的异常规范比 pf 更严，但 p=f1 错误。

（12）子类虚函数的异常规范必须比父类的异常规范更严（或相同），这与函数指针的异常规范相同。

（13）给 const 的成员函数指定异常规范时，异常规范语句应在 const 限定词之后。比如 class A{ void f() const throw(int){}};。

（14）异常规范不适用于模板，因为模板函数引发的异常可能随模板形参的具体化而有所不同。

（15）在类外定义带有异常规范的成员函数时，定义时指定的异常规范，必须与在类中对该成员函数声明时指定的异常规范相同。

11．修改默认的异常处理方式

（1）在默认情况下，若异常未被处理（捕获），则调用 terminate()函数，terminate()默认调用 abort()函数来终止程序，我们可以修改该函数，即调用其他函数来终止程序，其方法就是使用头文件 exception 中的 set_terminate 函数。

（2）未捕获异常，指的是在函数没有异常规范时抛出的异常，若该异常未被捕获（比如没

有匹配的 catch 子句或没有 try 块时抛出的异常），则这种异常被称为未捕获异常。这时程序首先调用 terminate()函数终止程序。

（3）意外异常，指的是抛出的异常与在函数规范中指定的异常不匹配。这时程序调用 unexpected 函数来终止程序，unexpected 函数会调用 terminate()函数。

（4）要想使用以下函数，应包含头文件 exception。

（5）set_terminate 函数原型：typedef void (*pf)(); pf set_terminate(pf s)throw();，表示使 terminate()函数使用 pf 指向的函数 s 来终止程序。

示例如下：

```
void f(){cout<<"A";} void main(){ set_terminate(f); try{throw 3.3;}catch(int){}}
```

说明：该程序抛出的异常为 double 类型，未被捕获，属于未捕获异常，程序使用 terminate()函数调用 f 函数来终止程序。

（6）若使用 set_terminate 函数设置了多次函数，则使用最后一次设置的函数。

（7）设置 unexpected 函数（违反异常规范时调用的函数）。

① 其方法是使用 exception 头文件中的 set_unexpected 函数。

② set_unexpected 函数原型：typedef void (*pf)(); pf set_unexpected(pf s)throw();。

③ 若 pf 函数抛出异常时，则按以下规则与异常规范进行匹配。

- 若 pf 抛出的异常与异常规范的异常匹配，则程序正常处理。
- 若 pf 抛出的异常与异常规范不匹配，且异常规范中没有包括 bad_excpteion 类型的异常，则程序会调用 terminate()终止程序。bad_exception 类型是从 exception 类派生而来的类型。包含在头文件 exception 中。
- 若 pf 抛出的异常与异常规范不匹配，但异常规范包括了 bad_exception 类型的异常，则不匹配的异常将被 bad_exception 异常取代。

示例如下：

```
void f(){cout<<"A"<<endl;}  void g()throw(int){throw 3.3;};
void main(){set_unexpected(f); try{g();}catch(int){}}
```

说明：因为函数 f 未抛出异常，而 try{g();}又引发意外异常，因此使用 unexpected()函数调用 f 函数终止程序。

```
void f(){throw 3.3; }  void g()throw(int){throw 3.3;};
void main(){set_unexpected(f); try{g();}catch(int){}}
```

说明：函数 f 抛出的异常与函数 g 的异常规范不匹配，而函数 g 又未包含 bad_exception 类型的异常，因此虽然使用 f 函数来终止程序，但最后还是会调用 terminate()函数来终止程序。

```
void f(){throw 3.3;}  void g()throw(int,bad_exception){ throw 3.3;}
void main(){set_unexpected(f); try{g();}catch(int){}catch(bad_exception){} }
```

说明：函数 f 抛出的异常与函数 g 的异常规范不匹配，但函数 g 包含有 bad_exception 异常，因此函数 f 抛出的异常与 bad_exception 匹配，程序会执行 catch(bad_exception)子句。

12. 异常标准库

（1）为什么需要了解标准库中的异常类？因为在调用函数时，若调用的函数抛出异常，则必须把该函数包含在 try 块中，让 try 块后的 catch 子句对异常进行处理。若调用的是标准库函数，而该函数在执行期间可能会遇到程序不正常的情况，从而有可能会抛出异常，因此就需要对标准库函数抛出的异常类进行处理。标准库函数抛出的异常是由 C++标准库提供的，若不明白标准库中异常的类型，就无法处理由标准库函数抛出的异常。因此为了弄明白标准库函数抛出的是什么错误的异常，我们有必要了解标准库中定义的异常类。

（2）异常类：此处指的是标准库抛出的异常的类型，这些类型是标准库定义的类类型（说简单点，就是普通的类）。

（3）exception 标准异常类简介。

① C++标准库中的异常的根类是 exception，标准库中的所有异常类都是从该类继承而来的，该类定义于头文件<exception>中。

② exception 类的构造函数、复制构造函数、复制赋值操作符及析构函数全都是公有的，因此任何程序都可以从该类继承，以编写自己的异常类。

③ exception 类拥有一个唯一的名称为 what 的虚拟成员函数，该函数返回 C 风格的字符串，从而为所抛出的异常提供详细的文字描述。调用该函数不需要任何的实参，因为 what 是虚函数，如果 catch 形参是 exception 类型的引用或指针，则调用该函数时，将执行对象实际指向的动态类型的版本。

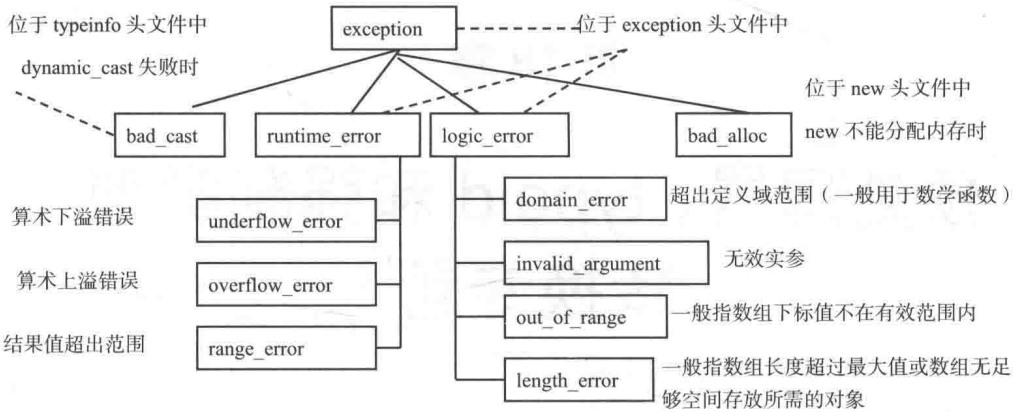
（4）所有的标准库异常类被分别定义在四个头文件中，这四个头文件分别是：exception、stdexcept、new、typeinfo。

（5）逻辑错误：指的是可以避免的错误。通常可通过编程来修复这类问题。

（6）运行时刻错误：只有在运行时才能检测到，一般表示无法避免的错误。

（7）exception、bad_alloc、bad_cast 只定义了无参数的默认构造函数，创建这些类型的对象时，无法为它们提供初值。其他类型定义了一个带 string 类型形参的构造函数，使用该形参主要是为了能提供更多的错误信息。

（8）详细的标准库异常类的继承结构及作用如图 17.1 所示。



说明：实线表示继承关系，虚线表示该类声明于哪个头文件中。

图 17.1　标准库异常类的继承结构及作用

第 18 章

预处理器、typeid 和强制类型转换专题

（1）预处理器：预处理器是编译器把 C++代码编译为机器指令之前执行的一个过程，所有的预处理器都以#开头，以便与 C++语句区分开来，比如#include 预处理器。

（2）注意：预处理器指令必须作为第一个非空白字符，前面不能有其他语句，而且预处理器之后不能有其他与预处理器不相关的语句，也就是非预处理器语句必须另起一行。为方便讲解，本文把预处理器指令的语法会写在一行上，假设读者已做了正确的格式处理。

1．#define 指令

（1）#define 指令格式：#define 标识符 字符序列。比如#define N 3。

① 表示以后凡是使用到“标识符”的地方都被后面的“字符序列”替换。注意，该语句不以分号结束。

② 编译器不会对“字符序列”进行类型检查。

③ 语句中的字符序列可以是任意的字符序列，而不仅仅是数字。比如#define PI HYONG，在遇到 PI 时就会用 HYONG 替换掉 PI。替换之后，HYONG 可能是一个未定义的标识符。

④ 示例：#define N 3。

- 表示以后遇到符号 PI 时都会被置换为 3，比如 int a[N];，定义大小为 3 的数组。
- 虽然 N 看起来和变量一样，但 N 和变量没有任何关系，N 只是一个符号或标志，在程序代码编译前，该符号会用一组指定的字符来替换。
- 可以看到，N 置换成 3 之后，相当于拥有了常量的性质，可以当作常量来使用。但在 C++中最好是用 const 来声名常量，比如 const long double PI=3.1416;。

（2）怎样把预处理指令放在多行上：其方法为使用续行符号“\”，该符号应是上一行的最后一个字符。在字符\后面不能有任何其他字符（包括注释），在下一行的开头也不能有任何其他字符（包括空白符）。比如：

```
#define m \
nop    //等于标识符 mnop，此处假设 nop 前无任何字符
```

（3）删除#define 定义的标志的语法：#define 标识符。

在#define 语句中，不为标识符指定置换字符串时就表示该标识符被删除了。比如#define PI，表示在程序中，删除该语句后面的所有 PI 标识符。比如 int a=PI;，则 PI 不会替换为任何东西，语句 int a=PI，相当于 int a=;，所以程序出错。

（4）使用#undef 取消#define 的定义，其语法为：#undef 标识符。

表示取消在#undef 后面定义的“标识符”。标识符被取消后，就不能再使用该标识符了。

（5）带参数的#define 的格式：#define 标识符（参数列表）置换字符串。

比如#define f(v) cout<<(v)<<endl;，表示将 f(v)用后面的字符串替换。其中，也可以对参数 v 进行替换。比如在程序中调用 f(3);，就会把程序转换为 cout<<(3)<<endl;，程序输出 3。注意括号不会被输出。参数列表也可有多个参数，参数间用逗号隔开，比如#define f(m,n) cout<<(m)<<(n)<<endl;。如果调用 f(3,4);，则程序转换为 cout<<(3)<<(4)<<endl;，输出 34。

（6）使用#define 只能对标识符进行简单的置换，也就是说，会将标识符直接替换为后面表示的字符序列，而不会进行算术优先级或类型检查。比如#define f(m,n) m*n，如果调用 f(4+2, 3);，则该语句会被简单地替换为 4+2*3，这与所希望的(4+2)*3 不一致。要想解决此问题，就需要给参数加上括号，比如#define f(m,n) (m)*(n)，则 f(4+2,3)被替换为(4+2)*(3)。

（7）使用字符串作为#define 参数：比如#define m "kdi"，如果有语句 cout<<m;，则会输出字符串 dki。注意 cout<<"m";，不能在字符前加上双引号以试图输出字符串 kdi，这样只会输出字符串 m，因为程序会把"m"解释为一个字符串，而不会把它解释为 cout<<"kdi"。

（8）把#define 参数指定为字符串：其方法是在参数前加上符号"#"。比如#define f(m) cout<<#m，在调用 f(dikl);时会被替换为 cout<<"dikl";，输出字符串 dikl；再比如#define f(m) #m，则 cout<<f(dikl);会被替换为 cout<<"dikl";，同样会输出字符串 dikl。这里需要注意的是，该方法只能用于参数，而不能用于其他地方，比如#define m #kidkl 就是错误的，这里试图用 m 来代替字符串"kidkl"，这是不成功的，正确方法为#define m "kidkl"或#define f(m) #m。

2. 逻辑预处理器指令

（1）逻辑#if 指令：该指令原理与条件语句 if 相同，如果测试为真就执行后面的语句，如果为假则跳过后面的语句。该指令有两种用法，一是可以用#if 指令测试某个符号以前是否用#define 指令定义过，这是最常用的用法；二是可以用来测试某个条件表达式是否为真。

（2）#if 指令用法一：测试某个符号是否以前用#define 定义过，共有以下几种情形。

① #if 指令语法形式一：

```
语#if defined 标识符 … #endif  //注意关键字 defined 比 define 多一个字母 d
```

缩写形式为：#ifdef 标识符…#endif。注意，应按预处理器格式写在多行上。

意义：表示如果指定的“标识符”已被#define 定义，则中间的语句就包含在源文件中；如果该标识符还未被#define 定义，则跳过#if 和#endif 之间的语句，该语句以#endif 结束。

② #if 指令语法形式二：

```
#if !defined 标识符…#endif   //注意关键字 defined 比 define 多一个字母 d
```

缩写形式为：#ifndef 标识符…#endif。注意，应按预处理器格式写在多行上。

意义：表示如果指定的标识符没有定义，则把#if 和#endif 之间的语句包含在源文件中。如果标识符已定义，则跳过#if 和#endif 之间的代码。系统常使用该语句防止头文件被多次包含。

③ 示例：

```
#define N    //定义标识符 N，此例不需要指定值
void main(){//预处理器指令必须作为第一个非空白字符，因此不能在“{”后使用预处理器指令
    #if defined N    //若标识符 N 被定义，则执行以下语句，注意预处理器指令结束后必须另起一行
    cout<<"A"<<endl; //输出 A
    #endif          //注意，预处理器指令必须另起一行进行输入
    #ifndef N        //若标识符 N 未被#define 定义，则执行以下语句
        cout<<"B"<<endl; //无输出
    #endif
}    //注意：大括号必须另起一行，不能在#endif 预处理器指令的后面
```

（3）防止头文件被包含多次的语法。

```
#ifndef HY #define HY 语句 #endif      //注意，应按预处理器格式写在多行上
```

说明：程序在开始遇到标识符 HY 时没有被定义，执行后面的语句，并使用#define 定义标识符 HY，在第二次被使用时则标识符 HY 已经被定义，这时就不再执行后面的语句，从而避免了同一头文件被包含多次的情况。

（4）#if 指令用法二：测试某个表达式的值是否为真，其语法格式为：#if 常量表达式…#endif。注意，常量表达式的求值结果应是整数常量表达式，比如#if 3+2 … #endif，即测试 3+2 的结果是否为真，若为真则执行#if 与#endif 之间的语句。

（5）多个#if 选择块：和常规的 if 语句一样，#if 也有对应的#else 和#elif 语句，比如#if a!=3 …. #else … #endif，表示如果 a!=3，则执行#if 后面、#else 前面的语句；如果为假，则执行#else 与#endif 间的语句。#elif 用来实现多个选择，该语句和常规语句的 else if 相似，比如#if a!=1 … #elif a!=2 … #elif a!=3… #else … #endif，表示如果 a!=1，则执行#if 后的语句；如果 a!=2，则执行该条件后的语句。

3. RTTI 运行时类型识别（需包含头文件<typeinfo>）

注意：有些编译器需要把 RTTI 的特性打开才能使用，比如 VC++ 2010 就有这项设置，位于项目→属性→展开 C/C++选项卡→选择“语言”，在右侧找到“运行时类型识别”，选择“是”。

（1）静态类型与动态类型：静态类型指的是对象（包括指针和引用）在声明时的类型，动态类型指的是当前对象（包括指针和引用）实际指向的类型。比如 B mb; A* p=&mb;，则 p 的静态类型是 A*，而动态类型是 B*，详见本书第 13 章。

（2）RTTI 运行时类型识别：使用运行时类型识别的主要作用是获得指向父类的指针实际所指向的子类的类型。C++使用两个操作符来实现 RTTI，即 dynamic_cast 和 typeid。

（3）typeid 操作符形式：type_info& typeid(object)，其中 object 既可以是任何类型的对象，也可以是一个类型。typeid 返回对 type_info 类类型的引用。

（4）通过引用 typeid 返回 type_info 的成员函数。可以使用 typeid 获得对象类型的名称，并判断两个类型是否相等。

（5）type_info 类。

① type_info 类的具体实现是依编译器而定的，不同的编译器拥有不同的内容。

② type_info 类的默认构造函数、复制构造函数和赋值操作符都是 private（私有的），因此不能创建、复制 type_info 类型的对象。若要向 type_info 类中增加成员函数，则应该从 type_info 继承。

③ 程序中引用 type_info 类中成员函数的唯一方法是使用 typeid 返回的 type_info 类型的引用。

④ type_info 类的成员函数：C++标准规定，所有编译器必须至少要实现以下成员函数：重载==操作符、重载的!=操作符、成员函数 name()和成员函数 before(const type_info&)。下面分别介绍这 4 个成员函数。

- 重载的“==”和“!=”操作符分别用于判断两个类型是否相等。其中“==”两边相等时则返回 true，“!=”两边不等时返回 true，比如 if(typid(3)!=typid(4.4))。
- name();成员函数：表示返回 C 风格字符串形式的类型名称，返回的名称依编译器而定。也就是说，不同编译器返回的名称可能不同。比如 typeid(3).name();返回字符串“int”。
- before 成员函数主要是为了使 type_info 类的信息能够排序，意义不大。

（6）使用 typeid 操作符示例。

```
cout<<typeid(3).name();  //输出 int，表示返回整型值的类型
class A{};  cout<<typeid(A).name();  //输出 class A，返回类型 A 的类型
if(typeid(3)==typeid(4))cout<<"A";  //输出 A，typeid 比较 3 和 4 的类型，它们都是 int 类型
```

（7）只有当 typeid 的操作数是指向带有虚函数的类型对象的指针时，且对该指针进行解引用后，才会返回指针的动态类型，其余情况都是返回指针的静态类型。

示例 1：

```
class A{public:virtual void f(){}};  class B:public A{};  B mb;  A*pa=&mb;
cout<<typeid(pa).name();  //输出 class A*,因为未对指针解引用，因而返回指针的静态类型
cout<<typeid(*pa).name();   //输出 class B，返回指针实际指向的动态类型
```

示例 2：

```
class A{};  class B:public A{};  B mb;  A* pa=&mb;
cout<<typeid(pa).name();  //输出 class A*,因为未对指针解引用，并且 A 没有虚函数，因而返回指针的静
                          //态类型
cout<<typeid(*pa).name(); //输出 class A;，因为 A 不含虚函数，所以返回指针的静态类型
```

（8）若指针 p 是指向带有虚函数类型的指针，且 p=0;，则 typeid(*p)会抛出一个 bad_typeid 异常，但 typeid(p).name()仍然会得到指针的静态类型。比如 class A{public:virtual void f(){}}; class B{}; B mb; A*pa=0;，则 typeid(*pa).name();抛出 bad_typeid 异常，必须把该语句放入 try 块对异常做处理，否则是错误的。

（9）若指针 p 指向的类型不带虚函数，且 p=0;，则 typeid(*p).name();和 typeid(p).name()都会返回指针的静态类型。

4. 强制类型转换

（1）强制类型转换运算符：C++有四种强制类型转换符，分别是 dynamic_cast、const_cast、static_cast 和 reinterpret_cast。其中，dynamic_cast 与运行时类型转换密切相关。使用这四种强制类型转换，主要是为了提高转换时的安全性，减少传统 C 语言强制转换时的随意性。

（2）dynamic_cast 强制转换运算符格式如下：

```
dynamic_cast<目标类型>(表达式)，//表示把表达式转换为尖括号中的目标类型
```

（3）dynamic_cast 转换符的作用：主要用于把父类类型的指针或引用转换为子类类型的指针或引用，以便能访问子类中除虚函数外的其他特有成员。

示例如下：

```
class A{public:virtual void f(){}}; class B:public A{public: void g(){}}; A *pa=new B();
pa->g();                                          //错误，因为 g 不是虚函数，也不是类 A 的成员函数
dynamic_cast<B*>(pa)->g();  //正确，表示把 pa 强制转换为子类 B 类型的指针，由此便可以访问子类 B 中特
                            //有的成员
```

（4）使用 dynamic_cast 转换成功的条件：若被转换的指针或引用未指向（或引用）目标类型或目标类型的子类型，则转换失败。若表达式所指向的对象不止一个，则表示目标类型的父类转换也是失败的（这种情况出现在多重继承时，详见下面的示例 7）。

① 若使用 dynamic_cast 转换到指针类型时失败，则返回 0。

② 若使用 dynamic_cast 转换到引用类型时失败，则抛出 bad_cast 类型的异常。

③ 简单理解 dynamic_cast 转换成功的条件，比如，dynamic_cast<B&>(ma);，则要求被转换的对象 ma 引用目标类型 B 的对象。假设为 B mb;，则要求有形如 A& ma=mb;这样的语句，才能转换成功。当然，若 ma 的类型就是 B，则也能转换成功。注意，在 A sa; A& ma=sa; B mb; ma=mb;中，并不是使 ma 重新引用 mb，而是使用 mb 对 ma 所引用的对象 sa 赋值。

（5）dynamic_cast 强制转换符的限制。

① dynamic_cast 被转换的对象必须是含有虚函数的类，因此不允许内置类型间的转换。

示例 1：

```
dynamic_cast<int>(3.3);//错误，不能用于内置类型间的转换
```

示例 2：

```
class A{}; class B:public A{};  A* pa=new B();
B *pb=dynamic_cast<B*>(pa);//错误，因为类 A 没有虚函数
```

② dynamic_cast 的目标类型必须是指针或引用。

示例 3：

```
class A{public:virtual void f(){}}; class B:public A{public: void g(){}};  A ma;
dynamic_cast<B>(ma);   //错误。目标类型 B 不是指针或引用
```

③ 除了满足前两个条件外，被转换的指针必须是已经初始化的，否则出错。

示例 4：

```
class A{public:virtual void f(){}}; class B:public A{public: void g(){}};
A *pa=0;  A* pa1;   dynamic_cast<B*>(pa1); //错误。pa1 未被初始化
B *pb=dynamic_cast<B*>(pa); //正确，转换失败，结果为 0，其中 pa 被初始化为 0
```

④ dynamic_cast 的目标类型可以不是多态的，在多重继承下比较明显。

示例 5：

```
class A{public:virtual void f(){}};  class B{public:void g(){cout<<"GB";}};
class C:public A,public B{}; //类 C 同时继承自类 B 和类 A，其中类 B 是非多态的
C mc;     A* pa=&mc;
dynamic_cast<B*>(pa);  //正确，目标类型 B 可以不是多态的，但 pa 必须是多态的
```

⑤ 若被转换的指针（或引用）未指向（或引用）目标类型或目标类型的子类型，则转换失败；若被转换的对象就是目标类型的对象，则转换成功。

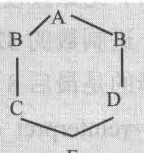
示例6：

```
class A{public:virtual void f(){}}; class B:public A{public: void g(){cout<<"A";}};
        A ma;          B mb;     A ma1=mb;      A &sa=mb;      A &sa1=ma;
        A* pa=&ma;     A* pa1=&mb;          B* pb=&mb;
        //dynamic_cast<B&>(ma);    //错误，ma 未引用目标类型 B 的对象，抛出 bad_cast 异常
        dynamic_cast<B&>(mb);      //正确，mb 的类型就是目标类型 B
        //dynamic_cast<B&>(ma1); //错误，ma1(不是引用)未引用目标类型 B 的对象，抛出异常
        dynamic_cast<B&>(sa);      //正确，sa 引用了目标类型 B 的对象 mb
        //dynamic_cast<B&>(sa1);   //错误，sa1 引用的是类 A 的对象 ma，ma 不是目标类型 B 的对象
                                   //抛出 bad_cast 异常
        cout<<dynamic_cast<B*>(pa)<<endl;  //输出 0，转换失败，pa 未指向目标类型 B
        cout<<dynamic_cast<B*>(pa1)<<endl;  //输出地址，转换成功，pa1 指向目标类型 B
        cout<<dynamic_cast<B*>(pb)<<endl;  //输出地址，转换成功，pb 就是目标类型 B*
```

⑥ 目标类型有多个时会转换失败，此时指针返回 0，在多重继承时能表现出这种错误。

示例 7：

```
class A{public:virtual void f(){}};
class B:public virtual A{public:};  //B 虚拟继承自 A，因此 A 只有一个副本
class C:public B{};  //非虚拟继承，本例会存在类 B 的两个副本
class D:public B{}; //非虚拟继承，本例会存在类 B 的两个副本
class E:public D,public C{};
E me;     A* pa=&me;
cout<<dynamic_cast<B*>(pa);  //输出 0，转换失败，因为存在两个 B 的副本
```



示例 7 继承关系图

（6）从以上各点可以看出，dynamic_cast 操作符在转换时执行两次检测：第一次检测转换是否有效，比如转换的目标类型是否是指针等，只有转换有效才进行转换；第二次检测发生在

运行时刻，比如，对指针转换失败则返回 0，对引用转换失败则抛出 bad_cast 异常。

（7）const_cast 操作符格式如下：

```
const_cast<目标类型>（表达式）；  //表示把表达式转换为尖括号中的目标类型
```

作用：该操作符用于改变 const 和 volatile 限定词（增加或删除），const_cast 最常用的用途就是删除 const 属性，如果某个变量在大多数时候是常量，而在某个时候又是需要修改的，这时就可以使用 const_cast 操作符了。比如 int a=3; const int *p=&a;，则*p=4;错误，但*const_cast<int*>(p)=4;正确，cout<<*p; 输出 4。

（8）const_cast 操作符的限制。

① const_cast 操作符只能改变 const 或 volatile 限定词（增加或删除），不能对其他类型进行转换，即 const_cast 不能进行从 int 到 double 的转换等。比如 int a=4; int *p=&a;，则 const_cast<float*>(p); 错误。

② const_cast 只能用于指针或引用。比如 const int a=3;，则 const_cast<int>(a);错误。

（9）static_cast 操作符：static_cast<目标类型>(表达式)。static_cast 本质上是传统 C 语言强制转换的替代品。该操作符用于非多态类型的转换，任何标准转换都可以使用它，即 static_cast 可以把 int 转换为 double，但不能对两个不相关的类对象进行转换，比如类 A 不能转换为一个不相关的类 B 类型。

（10）reinterpret_cast 操作符格式：reinterpret_cast<目标类型>(表达式)。

① 要想弄明白此操作符的含义，就需要明白内存与类型的关系，详见第 6 章。

② 目标类型必须是一个指针、引用、整型（包括所有 int 类型、char 类型、bool 类型、枚举型）。

③ reinterpret_cast 可以实现从整型到指针、从指针到整型或从一个指针到其他任意指针间的相互转换。

④ 深入理解 reinterpret_cast 操作符：reinterpret_cast 操作符可以用来处理与类型无关的转换，转换后的值与原始表达式所表示的值有完全相同的比特位。简单来说，就是对内存中的比特位重新赋予了一种类型，或者说对比特位重新进行解释。因此这种转换是很危险的，需要小心使用。比如假设在内存中的比特位为 0111 1111 0100 0110，若这串比特位按 int 类型解释，那就是十进制数的 32582；若按 char 类型解释，由于 char 只占 1 个字节，因此只能解释 8 位，假设解释的是最后 8 位，则 0100 0110=70，相当于 char 类型的字符 F，那么假设有 int a=32582;，char c=reinterpret_cast<char&>(a);，则 cout<<c;将输出 F，reinterpret_cast<char&>(a)表示把存储在内存中 a 的值的比特位按 char 类型进行解释，结果就是字符 F。

⑤ 为什么 reinterpret_cast 可以在任意指针之间相互转换：因为指针类型占据的内存宽度相同（一般为 4 个字节），因此指针之间是可以相互转换的，指针指向的类型只影响指针解释的内存的宽度，但对指针本身没有影响。

⑥ 由于内存的地址是用一个数字表示的，因此 reinterpret_cast 操作符允许整型到指针间的相互转换。

⑦ 由于是对比特位重新进行解释，而没有进行相互的值转换，因此不能在不是比特级别的类型之间进行相互转换。比如 reinterpret_cast<int>(3.3);错误。按照常规转换之后的值可能为 3，这会导致对比特位进行重新排列。

⑧ 从上面的原理可以看出，使用 reinterpret_cast 在指针间进行相互转换时，最好是在按比特位进行解释的相同宽度的范围内进行转换。比如 int a; reinterpret_cast<float *>(&a);，则转换之后按 float 类型解释的比特位超过了按 int 类型进行解释的范围。示例如下：

```
int a=3;  float *p=reinterpret_cast<float*>(a);
```

分析：转换后 p 指向内存地址为 0000 00003（十六进制）处的位置，即把 a 的值在内存中的表示形式转换为内存地址。这种转换会发生严重错误，有两个原因，一是地址 0000 00003 的内存地址不知道存储的是什么；二是在解引用时，假设 int 只占 4 个字节，而 float 占 8 个字节，这里会多解释 4 个字节的未知内存的内容。

（11）从 C 语言继承来的传统的(T)e 强制类型转换方法，可以实现 static_cast、reinterpret_cast 和 const_cast 三种组合的任意转换。

第 19 章

string 类专题

（1）string 类用于处理字符串，用于代替使用不方便的 C 风格字符串，使用 string 类表示的字符串可以像处理普通变量那样处理字符串，因此可以对 string 类表示的字符串进行直接的相加、比较、赋值等操作。比如 string s1="abc", s2="def";，则 s1=s1+s2;，结果 s1="abcdef"。对于 s1=s2;，则结果 s1="def"等。C 风格字符串只能使用内置的库函数进行这些操作，用起来很不方便。比如 char c1[]="abc"; char c2[]="def";，则 c1=c2;错误，因为不能改变数组的地址。c1>c2 比较的是两个指针的地址，而不是字符的大小。c1+c2;错误，这是把两个指针的地址相加，而不是把两个字符数组相加。

（2）string 对象创建的字符串的最大特点是：可以自动调整对象大小，以适应所需的字符串。string 对象能存储的最大字符数由 string 类的静态常量 string::npos 设定，通常是最大的 unsigned int 值。

（3）string 的内存分配策略。

C++规定了使用 string 对象能够存储的最大字符数为 string::npos;，这意味着使用的 string 对象最多只能存储这么多个字符，但不代表编译器一开始就会为 string 对象分配这么大的内存。若每创建一个 string 对象就为其分配最大容量的内存，那么会严重浪费存储空间。所以在每创建一个 string 对象时，编译器一般按照 16 的倍数为 string 对象分配内存（容量），若 string 对象存储的字符数超过了当前分配的容量，则按 16 的倍数进行扩展。比如 string s;，一开始编译器可能会为其分配 16 个字节（VC++ 2010 为 15）的容量。若 s 存储的字符数超过了 16 个字符，则自动把容量扩大到 32 个字符（VC++ 2010 为 31），依此类推，但 s 的内存最多只能扩大到 string::npos 指定的数量。string::npos 一般为最大的 unsigned int 值。string 类使用 capacity()成员函数返回为当前 string 对象分配的容量。

1．string 类的原型（需包含 string 头文件）

（1）string 是一个模板类，因此它具有类和模板的特性，也就是说，string 类有构造函数、重载的操作符、成员函数等。因为 string 是模板类，所以应创建一个模板类具体实例化版本的

对象才能使用，然后通过对象调用成员函数使用类。

（2）记住，string s;创建的 s 是一个类的对象，而不是字符串字面值，它们是两种不同的类型。

（3）string 是模板类 basic_string 的 char 特化体版本使用 typedef 命名后的别名，wstring 是模板类 basic_string 的 wchar 特化体版本使用 typedef 命名后的别名。

（4）basic_string 类的原型为：

```
template<class charT, class traits=char_traits<charT>, class Allocator=allocator <charT> >
class basic_string;
```

① charT 是一个类型模板形参，若实例化时传递 char 类型，则 charT=char；若传递 wchar，则 charT=wchar。

② traits 是类型模板形参，描述了字符串的特征，比如字符串是否以'\0'作为结尾等。traits 要求传递一个 char_traits<T>的模板类型作为实参。

③ Allocator 也是一个类模板形参，它的主要作用是用于处理字符串的内存分配问题，默认使用 new 和 delete 分配内存。Allocator 要求传递一个 allocator<T>类型的模板类型作为实参。

④ basic_string 有两个特化体版本，如下所示，当然我们也可以实例化其他类型版本的 base_string 类模板。

```
typedef base_string<char> string;       //即 string 类是使用 typedef 重命名后的
                                        // basic_string 类模板的 char 特化体版本
typedef base_string<wchar_t> wstring;   //主要用于处理宽字符串
```

（5）size_type 类型：size_type 是 basic_string 类中定义的类型，一般被定义为 unsigned 类型。需要使用限定名的方法来使用 size_type 类型，比如 string::size_type a;或 basic_string<char>::size_type a;6。

（6）npos 静态常量：表示 string 对象能够存储的最大字符数，其形式为：static const size_type npos=-1;，可见 npos 是 basic_string 类中定义的静态常量，其类型为 size_type，其值为-1。对于无符号的 size_type 变量，赋值为-1，相当于把最大的无符号数赋值给了它。

2. string 类的构造函数

（1）basic_string 的构造函数与 char 特化版本的 string 构造函数的差别只在于，basic_string 构造函数多了一个用于分配内存的默认形参 const Allocator& a=Allocator(); 。比如 string 类的构造函数 string(const char *s);，其对应的 basic_string 构造函数形式为 basic_string(const charT* s, const Allocator& a=Allocator());。

（2）string 类有 6 种形式的构造函数，本文使用 basic_string 类模板简化后的 string 构造函数版本。注意：字符串从 0 开始计数。

① 原型 1：string();

意义：创建一个默认的 string 对象，长度为 0。

示例：string s;，表示创建一个长度为 0 的字符串对象 s。

② 原型 2：string(const char *s);

意义：将 string 对象初始化为 s 指向的传统 C 风格字符串（即以空字符结束的字符串）。

示例：string s("hyong");，则结果为 s="hyong"。

③ 原型 3：string(size_type n,char c);

意义：创建一个包含 *n* 个元素的 string 对象，其中每个元素都被初始化为字符 c。

示例：string s(4, 'c');，则结果为 s="cccc"。

④ 原型 4：string(const string &str, size_type pos=0, size_type n=npos); //即复制构造函数

意义 1：将 string 对象初始化为 str 中从 pos 开始的 *n* 个字符，或从 pos 开始到结尾的字符，其中 npos 表示 string 对象的最大长度。

示例：string s="ABCDEFG"; string s1(s,2,4);，则结果 s1="CDEF"；再比如 string s2(s,2);，结果为 s2="CDEFG"。

意义 2：该构造函数的默认值形式为复制构造函数，经常被用于创建一个已存在的 string 对象的副本。

示例：string s="CDEF"; string s1(s);，则 s1=s="CDEF"。

意义 3：若指定的字符个数大于原字符串的长度，则使用原字符串到末尾的字符初始化目标 string 对象。

示例：string s("ABDE"); string s1(s,0,22);，则 s1="ABDE"。

意义 4：pos 的值不能大于原字符串的长度，否则会抛出 out_of_range 异常。

示例：string s("ABCD"); string s1(s,22);错误。

⑤ 原型 5：string(const char *s, size_type n);

意义：将 string 对象初始化为 s 指向的传统 C 字符串中的前 *n* 个字符，即使超出了字符串的范围，操作仍会进行。

示例：char c[]="hyong"; string s(c, 5);，结果为 s="hyong"。注意，即使复制的长度超出了数组的长度，操作仍将进行。也就是说，如果把 5 改为 10，将导致 5 个无用的字符被复制到 s 中。

⑥ 原型 6：tempalte<class Iter>string(Iter begin, Iter end);

意义：将 string 对象初始化为[begin, end)间的字符，其中 begin 和 end 类似于指针，用于指定位置，范围包括 begin 在内，但不包括 end。注意，begin 和 end 被看作指针，该方法可用于数组、字符串和 STL 容器。

示例：char c[ ]= "hyongilfmm"; string s(c+2, c+4);，结果 s="on"表示使用从指针 c 指向的字符串"hyongilfmmm"，位于区间[2,4)的字符"on"初始化 s。注意，第 4 个字符不包括在内。字符串从 0 开始计数。

3. string 类对象的特点

（1）不能用单个字符来初始化 string 对象，比如 string a='d';是错误的，但是 string a(1,'d')是正确的，这里将把'd'初始化为"d"。

（2）string 对象只能接收以空字符结尾的字符串，比如语句 char b[ ]={'a', 'b', 'c'}; string a=b;是错误的。

（3）不能将 string 对象的字符串赋值给 char 类型的数组或指针，因为它们是两个不同的类型。

4. string 对象的输入输出

（1）string 对象的输入：可以使用 cin 和 getline 函数对 string 对象进行输入。注意，此处的 geline 函数不是 istream 流的成员函数。

（2）使用 cin 输入 string 对象：有关 cin 的更多特性请参阅第 16 章。

① cin 会忽略开始的所有空白符（包括空格、回车换行符、制表符等）。

② 读取到空白字符时结束对 string 对象的读取。比如 string s,s1; cin>>s>>s1;，若输入 sss　ddd，则 s="sss"; s1="ddd"。

（3）string 类的 getline()函数。

getline 函数原型：

```
template<class charT, class traits, class Allocator> basic_istream<charT, traits>&
getline(basic_istream<charT, traits>& is, basic_string<charT, traits, Allocator>& str,
charT delim='\n');
```

简化的 getline 函数原型：

```
istream<char,traits>& getline(istream& is, string& str, char delim='\n');
```

① 表示把输入流 is 中的字符读到字符串 str 中，直到遇到分界符 delim 或到达文件尾，或到达字符串的最大长度。getline 不会忽略空白符，比如 string s; getline(cin,s,'z');，假设输入 ddd sssz，则 s="ddd　sss"。

② getline 会自动调用 string 对象的大小，以使其刚好能容纳输入的字符，因此在 getline 函数中，不需要指定读到多少个字符的参数。

③ getline 是一个独立的函数，只需直接调用即可，它不是 istream 或 string 的成员函数。比如 string s; getline(cin,s);，s.getline(cin,s);错误，因为 getline 不是类 string 的成员函数。

④ 若第一个字符就是分界符，则 string 对象被设为空 string。

⑤ 分界符 delim 将被读取并丢弃，但不会被赋值给 string 对象。若不指定分界符，则默认为回车换行符'\n'。若指定了分界符，则回车换行符会被当成字符读入 string 对象。比如 string s,s1; getline(cin,s,'z'); cin>>s1;，假设输入 aa↙bbzcc;，则 s="aa↙bb"; s1="cc";，其中↙表示回车换行符。

⑥ getline 返回的是 istream 引用。因此，若到达文件尾，会设置 eofbit 状态。若读取的字符数超过 string 允许的最大数（string::npos 的值），则设置 failbit 位。

⑦ 第一个参数必须是一个输入流的对象，比如标准输入流 cin。

⑧ 这个版本的 getline 函数只能适用于 string 对象，应注意与接受 char*字符的 getline 函数的区别。

提示：下面函数原型使用的都是简化后的版本，即使用 basic_string 的 char 特化版本，并使用 typedef 重命名后的 string 类型。因此只需把下面函数原型中的 string 替换为 basic_string，把 char 替换为 charT，即为 basic_string 的完整版本原型。

5. 与 string 类有关的操作符函数

（1）C++为 string 类实现了赋值、关系、下标、加运算符，这些运算符有些是 string 类的成员，有些不是 string 类的成员。

（2）是否是成员的差别在于左侧是否必须是 string 类的对象，比如 char c[33]; string s="aaa";，则 c=s;错误，因为=操作符是 string 类的成员函数，左侧必须是 string 类对象才是对重载的“=”操作符的正确调用。

（3）赋值操作符“=”。

① 它是 string 类的成员函数。

② 可以使用 string 对象、字符串数组、单个字符对 string 类对象进行赋值。这意味着 C++对这三种类型重载了三个版本的赋值操作符，即形如 operator =(string&)、operator=(char *)、operator=(char)的三个重载版本。

③ 赋值操作符会把以前的数据清空再重新写入，而不是对原有数据进行修改。比如 string s="abcde"; s="AB";，则结果 s="AB"，而不是 s="ABcde"。

④ 示例：string s="aa";，则 s="bb"; s='a';都是正确的。但 string s='a';错误，因为这是初始化，而不是赋值，没有接受 char 类型的 string 构造函数，所以出错。

（4）关系操作符。

① 除“!”关系操作符外，C++为所有的 string 类都实现了其余的关系操作符。

② 关系操作符不是 string 类的成员函数。

③ 可以使用 string 对象、字符串数组的各种组合作为操作数，这意味着所有关系运算符都对这两种类型的各种组合进行了重载。

④ 注意：单个字符不能作为关系运算符的操作数。

⑤ 示例：若 string s="aaa";，则 s>"a"; "A"<s; "AA">"BB"都正确，但 s>'a'; "AA">'A';错误。

⑥ 关系操作符的比较规则。

- 按顺序比较第一个不相同的字符作为结果。

- 排在前面的字符小于排在后面的字符，比如 a 小于 b。
- 若两个 string 对象长度不同，且长的 string 对象的前面部分与短的 string 对象相同，则短的 string 对象更小。
- 大写字母小于任意的小写字母。
- 示例：若 string s="Zabc";，则 s>"afbc";，结果为 0；若 s>"Zab";，则结果为 1；若 s>"Ydfab";，则结果为 1；若 s>"a";，则结果为 0。

（5）下标操作符。

① 它必须是成员函数。

② 返回的是 char 类型的引用，因此下标操作符可以作为左值使用。这意味着可以使用下标操作符对字符串中的某个字符进行修改。

③ 下标操作符不会进行边界检查，若超出数组的范围会引发严重错误。

④ 示例：string s="abcd";，s[1]='c';，则结果 s="accd";。

（6）+（加）操作符。

① 可用于把第二个字符串追加到第一个字符串的后面。

② 加操作符不是 string 类的成员函数。

③ 加操作符不会修改被加的字符串，而是创建一个新字符串。比如 string s="abc"; string s1=s+"de";，则结果为 s="abc"; s1="abcde";，其中 s 的值未被改变。

④ 注意：加操作符的操作数不能是单个的字符。比如 string s="abc";，则 s+'d';错误。

⑤ 加操作符的两个操作数必须至少有一个是 string 类型的。比如"abc"+"de";错误，string s="abc";，则 string s1="d"+s;正确。但 string s1="ab"+"cd"+s;错误，因为子表达式"ab"+"cd"没有一个是 string 类型的。

（7）+=操作符。

① 用于把第二个字符串追加到第一个字符串的后面，比如 string s="abc";，则 s+="de";，结果 s="abcde";。

② +=操作符是成员函数，这意味着+=的左操作数必须是 string 类型的对象。比如 string s="abc";，则"de"+=s;错误。

③ +=操作符的右操作数可以是 string 对象、字符串数组或单个字符。比如 string s="abc"; s+='D';，则 s="abcD"。

6. 与操作符类似的 string 成员函数

（1）at(n)成员函数：该函数与下标操作符相同，唯一的区别是 at 会进行边界检查，若超出数组范围，会抛出 out_of_range 异常，比如 string s="abcde"; s.at(2)='e';，则结果 s="abede";。但 s.at(22);错误，因而抛出 out_of_range 异常。

（2）compare 成员函数：比较。

通用规则：

① 用于比较两个字符串。

② s1.compare(s2)返回值的规则如下。

- s1>s2;返回正数。
- s1<s2;返回负数。
- s1=s2 返回 0。

③ 原型 1：int compare(const string &s) const;

意义：使用 s 与原 string 对象进行比较。

示例：string s1="abc"; string s2="abe"; int n=s1.compare(s2);，结果 s1<s2; n=-1;。

④ 原型 2：int compare(size_type pos, size_type n, const string &s) const;

意义：使用原 string 对象中从位置 pos 开始的 *n* 个字符与 s 进行比较。

若起始位置 pos 超过原 string 对象的长度，则抛出 out_of_range 异常。

示例：string s1="abcdef"; string s2="abcc"; int n=s1.compare(2,3,s2);，此时 s1="cde";，结果 s1>s2; n=1;。

⑤ 原型 3：int compare(size_type pos1,size_type n1, const string &s, size_type pos2, size_type n2) const;

意义：使用原 string 对象中从 pos1 开始的 n1 个字符与 s 中从 pos2 开始的 *n*2 个字符进行比较。

示例：string s1="abcdefgh"; string s2="abcc"; int n=s1.compare(2,3,s2,1,2);，此时 s1="cde"，s2="bc"，结果 s1>s2; n=1。

⑥ 原型 4：int compare(const char *c) const;，原型 1 的 char 数组版本

调用方法：s.compare(c);，表示把 s 与 c 进行比较。

示例：string s1="abc"; char c[]="cde"; int n=s.compare(c);，结果 s<c; n=-1;。

⑦ 原型 5：int compare(size_type pos1, size_type n1, const char* c, size_type n2=npos) const;，原型 3 的 char 版本

调用方法：s.compare(2,3,c,2);，表示使用 s 中从位置 2 开始的 3 个字符与 c 中的前 2 个字符进行比较。

示例：string s1="abcdef"; char c[]="cdef"; int n=s.compare(2,3,c,2);，此时 s="cde"; c="cd";，结果 n=1;。

（3）append()成员函数：追加。

① 该函数用于将一个字符串追加到另一个字符串的后面，其效果与使用+=相似。

② 原型 1：string &append(const string& s);

意义：把字符串 s 的内容追加到原 string 对象的末尾。

示例：string s1="ab"; string s2="cd"; s1.append(s2);，结果 s1="abcd";。

③ 原型 2：string& append(const string& s, size_type pos, size_type n);

意义：把字符串 s 中从位置 pos 开始的 *n* 个字符追加到原 string 对象的末尾。

示例：string s1="ab"; string s2="defg"; s1.append(s2,1,2);，结果 s1="abef";。

④ 原型 3：string& append(const char* c);，原型 1 的 char 版本

意义：把字符数组 c 中的内容追加到原 string 对象的末尾。

示例：string s="ab"; char c[]="cd"; s.append(c);，结果 s="abcd";。

⑤ 原型 4：string& append(const char* c, size_type n);，原型 2 的 char 版本

意义：把字符数组 c 中的前 *n* 个字符追加到原 string 对象的末尾。

示例：string s="ab"; char c[]="abcdef"; s.append(c,2);，结果 s="abab";。

⑥ 原型 5：basic_string& tappend(size_type n, char c);

意义：追加 *n* 个字符 c 到原 string 对象的末尾。

示例：string s="ab"; s.append(3,'c');，结果 s="abccc";。

⑦ 原型 6：template<class InputIterator> string& append(InputIterator first, IputIterator last);

调用方法：s.app(c+2,c+5);表示把指针 c 所指字符串中位于区间[2,5)的字符追加到 s，注意 c 必须是指针。

示例：string s="ab"; char c[]="abcdefgh"; s.append(c+2,c+5);，指针 c 指出的字符串“abcdefgh”，位于区间[2,5)的字符为 cde，注意不包括末尾的区间 5。结果 s="abcde"。

（4）assign()成员函数：赋值。

① 该函数用于将一个字符串赋给另一个字符串，功能与使用赋值运算符（=）相似。

② 原型 1：string& assign(const string& s);

意义：使用字符串 s 中的内容替换原 string 对象中的内容。

示例：string s1="abc"; string s2="def"; s1.assign(s2);，结果 s1="def"。

③ 原型 2：string& assign(const string& s, size_type pos, size_type n);

意义：使用字符串 s 中从位置 pos 开始的 *n* 个字符替换原 string 对象中的内容。

示例：string s1="ab; string s2="abcdef"; s1.assign(s2,2,3);，结果 s1="cde"。

④ 原型 3：string& assign(const char* c, size_type n);

意义：使用字符数组 c 中的前 *n* 个字符替换原 string 对象中的内容。

示例：string s="ab"; char c[]="cdefg"; s.assign(c,3);，结果 s=cde;。

⑤ 原型 4：string& assign(const char* c);

意义：使用字符数组 c 中的内容替换原 string 对象中的内容。

示例：string s="ab"; char c[]="cde"; s.assign(c);，结果 s="cde";。

⑥ 原型 5：string& assign(size_type n, char c);

意义：使用 *n* 个字符 c 替换原 string 对象中的内容。

示例：string s="ab"; s.assign(3,'c');，结果 s="ccc";。

⑦ 原型 6：template<class InputIterator> string& assign(InputIterator first, InputIterator last);

调用方法：s.assign(c+2,c+5);，表示使用指针 c 把所指区间[2,5)中的字符替换 s 中的内容。

示例：string s="ab"; char c[]="abcdefgh"; s.assign(c+2,c+5);，指针 c 所指字符串为"abcdefgh";，在区间[2,5)中的字符为 cde;，注意不包括末尾的区间 5，结果 s="cde"。

（5）以上函数的越界处理规则及通用规则。

① 以上函数若指定的起始位置 pos 超过了该字符串的长度，则会抛出 out_of_range 异常。

② 以上函数若指定的字符个数 *n* 超过字符串长度，虽不会出错，但对于字符数组会进行越界比较，对于 string 对象不会进行越界比较。因为对于下标运算符[]，C++不会进行越界检查，所以字符数组会进行越界比较。

③ 以上函数都是使用子串进行比较。若是对 string 对象进行比较，则要求指定起始位置 pos。对于字符数组都不要求指定起始位置 pos，这意味着若是子串来自字符数组，则要少指定一个参数。

7. 常用的 string 成员函数

（1）substr 成员函数：返回子串。

原型：string substr(size_type pos=0, size_type n=npos)

意义：表示返回由 pos 开始的 *n* 个字符，如果指定的字符数超过了 string 对象的结尾，则 substr()函数只返回从指定位置到 string 对象末尾的字符。如果指定的起始位置超出了 string 对象的结尾，则抛出一个异常。

示例：string s1="abcdefgh"; string s2=s1.substr(2,3); string s3=s1.substr(2,44);，结果 s2="cde"; s3="cdefgh"。

（2）copy()成员函数：复制字符串。

原型：size_type copy(char* c, size_type n, size_type pos=0) const;

意义：将原 string 对象中从位置 pos 开始的地方复制 *n* 个字符到目标数组 c 中。函数返回复制的字符数，该函数不追加空值字符，且不对目标数组 c 进行长度检查。

注意：

- 形参 c 是目标数组而不是原字符串，原字符串来自调用该函数的 string 对象。
- 起始位置和字符数与其他函数的位置相反。
- 复制之后是改写目标数组的内容而不是替换。

- 此函数只能把于 string 对象复制到字符数组中，而不能复制到 string 对象中。
- 若指定的字符数 *n* 超过了字符长度，则复制从指定位置到 string 对象尾的字符。若起始位置 pos 超过了 string 对象长度，则抛出 out_of_range 异常。

示例：string s="ABCDEFG"; char c[]="abcdefg"; s.copy(c,3,2);，结果 c=CDEdefg;，这些情形都会出错。string s1="abc";，则 s.copy(s,1,1);错误，因为不能复制到 string 对象中。s.copy(c,2,33);错误，因为起始位置超过了 string 对象长度。

（3）swap()成员函数原型：void swap(string& s);，表示交换两个 string 对象的内容。

示例：string s1="abc"; string s2="def"; s1.swap(s2);，结果 s1="def"; s2="abc";。

（4）erase()成员函数：从字符串中删除字符。

① 原型 1：string& erase(size_type pos=0, size_type n=npos);

意义：从位置 pos 开始删除 *n* 个字符或删除到字符串尾。

示例：string s="abcdefgh"; s.earse(2,3);，结果 s="abfgh";。

② 原型 2：iterator erase(iterator position);

意义：删除迭代器位置引用的字符，并返回指向下一个元素的迭代器。如果后面没有其他元素，则返回 end();。

③ 原型 3：iterator earse(iterator first, iterator last);

意义：删除区间[first, last)中的字符，即删除从包括 first 开始到不包括 last 之间的字符。它返回一个迭代器，该迭代器指向最后一个被删除元素的后面的一个元素。

（5）其他常用的 string 成员函数如表 19.1 所示。

表 19.1　常用的 string 成员函数

函 数 名	说　明
boole empty() const	若字符串为空则返回 true
string &erase(pos=0, n=npos)	删除从位置 pos 开始的 *n* 个字符
void clear()	删除字符串中的所有字符
size()	返回 string 对象的长度（不含终止字符'\0'），即返回 string 对象中的实际字符个数
length()	与 size()相同，这是较早版本提供的函数
capacity()	返回为 string 对象分配的容量，这个数字可能大于 size()的数目。若字符串的长度大于当前分配的容量，则自动为 string 对象扩展容量

续表

函数名	说　明
max_size()	当前 string 对象能够存储的最大字符长度，该长度比 string::npos 小 1，因为末尾需要存储一个空字符
void reserve(size_type n);	把 capacity()（即容量）设置为大于或等于 *n* 的大小。分配容量时是按 16 的倍数进行的，若分配的容量小于当前的容量则忽略

c_str()	返回指向 C 风格字符串第一个字符的指针。此函数可以把 string 对象转换为 C 风格的字符串，转换之后的指针类型为 const 型
	示例：string s="abcde";
	const char *c=s.c_str();//正确
	//char *c3=s;　//错误，不能把 string 类型转换为 char*类型
	//char *c1=s.c_str();//错误，指针 c1 应是 const 型
	//const char c2[]=s.c_str();//错误，c_str 返回的是指针

data()	与 c_str()相似
void resize(size_type n) void resize(size_type n, char c)	将字符串的长度修改为 *n*，若 *n* 小于字符串的长度，则截短字符串；若 *n* 大于字符串的长度，则第一个 resize 函数使用空字符填充，第二个 resize 函数使用字符 c 填充 示例：string s="abcdef"; s.resize(3); s.resize(6); s.resize(8,'d'); 结果为 s="abc　dd";

示例：长度、容量、最大长度

下面的值会因编译器和操作系统而有所不同。

```
string s="abcdefg";
typedef string::size_type U; //把 size_type 的类型重命名为 U，以缩短程序长度
U n1=s.size();　　//n1=7，当前 s 存储的字符个数
U n2=s.capacity(); //n2=15，为当前 string 对象 s 分配的容量
U n3=s.max_size();　//n3=4294967294，在本机 string 对象能存储的最大字符数（不含空字符）
U n4=string::npos;　//n4=4294967295，在本机 string 对象能存储的最大字符数（含空字符）
s="12345678901234567"; //自动扩容，已超过当前为 s 分配的容量 16（VC++ 2010 为 15）
U n5=s.capacity();　//n5=31，可见自动把容量扩大为了 31 个字节
s.reserve(34); //容量会按 16 的倍数分配，不会分配为 34 个字节
U n6=s.capacity();　//n6=47
s.reserve(15);　//分配的容量小于当前容量，被忽略
U n7=s.capacity();　//n7=47;
```

8．find、insert、replace 成员函数

为使程序简洁，以下程序都使用 int 型保存结果，但有可能会出现数据溢出的情况。

（1）find 成员函数：查找。

① 原型 1：size_type find(const string &str, size_type pos=0) const

意义：从原 string 对象的位置 pos 处开始查找字符串 str，如果找到，则返回该字符串首次出现时其首字符的索引，否则返回 string::npos。

示例：string s1="abcdefghijcdekl"; string s2="cde"; int c1=s1.find(s2,1); int c2=s1.find(s2,3); int c3=s1.find(s2,11);，结果 c1=2; c2=10; c3=−1;。

② 原型 2：size_type find(const char *s, size_type pos=0) const，原型 1 的 char 版本

意义：从原 string 对象的位置 pos 处开始查找 char 字符数组 s。如果找到，则返回该字符串首次出现时其首字符的索引，否则返回 string::npos。

示例：string s1="abcdeghcdek"; int c[]="cde"; int n1=s1.find(c,1); int n2=s1.find("cde",3);，结果 n1=2; n2=7;。

③ 原型 3：size_type find(const char *s, size_type pos, size_type n)

意义：从原 string 对象的 pos 位置处开始，查找 char 字符数组 s 的前 *n* 个字符组成的子字符串。如果找到，则返回该子字符串首次出现时其首字符的索引，否则返回 string::npos。注意，这个版本的 find 函数只适用于 char 类型的数组，而不适用于 string 对象。

示例：

string s1="abcdefghcdk"; string s2="cde"; char c[]="cde";

int n1=s1.find(c,3,2);，由 s1 构成的子串为"defghcdk"，由 c 构成的子串为"cd"，所以结果 n1=8。

int n2=s1.find(c,3,3);，由 s1 构成的子串为"defghcdk"，由 c 构成的子串为"cde"，所以结果 n2=-1;。

int n3=s1.find(s2, 3, 2);，错误，s2 不是 char*类型。

④ 原型 4：size_type find(char c, size_type pos=0) const

意义：从原 string 对象的位置 pos 处开始查找字符 c，若找到，则返回字符首次出现的位置，否则返回 string::npos。

示例：string s1="abcdefghcdk"; int n=s1.find('c',4);，结果 n=8;。

（2）insert 成员函数：插入。

① 原型 1：string& insert(size_type pos1, const string& str);

意义：将字符串 str 插入到原 string 对象 pos1 位置的前面。

示例：string s1="abcdef"; string s2="gh"; s1.insert(2,s2);，结果 s1="abghcdef";。

② 原型 2：string& insert(size_type pos1,const string& str, size_type pos2, size_type n);

意义：将字符串 str 从位置 pos2 开始的 *n* 个字符插入到原 string 对象 pos1 位置的前面。

示例：string s1="abcdef"; string s2="ghiklmn"; s1.insert(2,s2,2,3);，结果 s1="abiklcdef";。

③ 原型 3：string& insert(size_type pos, const char* s, size_type n);

意义：表示把 char 字符数组 s 的前 *n* 个字符插入到原 string 对象 pos 位置的前面。

示例：string s1="abcde"; char c[]="ghikl"; s1.insert(2,c,3);，结果 s1="abghicde";。

④ 原型 4：string& insert(size_type pos, const char* s);，原型 1 的 char 版本

意义：表示将整个 char 字符数组 s 插入到原 string 对象 pos 位置的前面。

示例：string s1="abcde"; char c[]="ghi"; s1.insert(2,c);，结果 s1="abghicde";。

⑤ 原型 5：basic_string& insert(size_type pos, size_type n, char c);

意义：将 *n* 个字符 c 插入到原 string 对象 pos 位置的前面。

示例：string s1="abcde"; s1.insert(2,4,'x');，结果 s1="abxxxxcde";。

⑥ 原型 6：iterator insert(iterator p, charT c=charT());

⑦ 原型 7：void insert(iterator p, size_type n, charT c);

⑧ 原型 8：template<class InputIterator>void insert(iterator p, InputIterator first, InputIterator last);

（3）replace()成员函数：替换字符串。

① 原型 1：string& replace(size_type pos1, size_type n1, const string& str);

意义：把原 string 对象从位置 pos1 开始的 n1 个字符替换为 string 对象 str，不管 str 的长度是否等于要被替换的字符数 n，原 string 对象的 n 个字符都会被替换为 str。

示例：string s1="abcdefg"; string s2="hi"; s1.replace(2,4,s2);，结果 s1="abhig";。

string s1="abcdefg"; string s2="hijklmn"; s1.replace(2,4,s2);，结果 s1="abhijklmng";。

② 原型 2：string& replace(size_type pos1, size_type n1,

const string& str, size_type pos2, size_type n2);

意义：把原 string 对象从 pos1 开始的 n1 个字符替换为字符串 str 从 pos2 开始的 n2 个字符。其他规则见原型 1。

示例：string s1="0123456"; string s2="abcdef"; s1.replace(2,4,s2,1,2);，结果 s1="01bc6";。

string s1="01234"; string s2="abcdef"; s1.replace(2,2,s2,1,3);，结果 s1="01bcd4";。

③ 原型 3：string& replace(size_type pos,size_type n1,const char* c, size_type n2);

意义：把原 string 对象从位置 pos 开始的 n1 个字符替换为 char 字符数组 c 的前 n2 个字符。其他规则见原型 1。

示例：string s1="0123456"; char c[]="abcd"; s1.replace(2,4,c,2);，结果 s1="01ab6"。

string s1="01234"; char c[]="abcdefg"; s1.replace(2,2,c,3);，结果 s1="01abc4"。

④ 原型 4：string& replace(size_type pos, size_type n1, const char* c);，原型 1 的 char 版本

意义：把原 string 对象从位置 pos 开始的 n1 个字符替换为 char 字符数组 c。其他规则见原型 1。

示例：string s1="0123456"; char c[]="abcd"; s1.replace(2,4,c);，结果 s1="01abcd6"。

string s1="0123456"; char c[]="ab"; s1.replace(2,4,c);，结果 s1="01ab6"。

⑤ 原型 5：string& replace(size_type pos, size_type n1, size_type n2, char c);

意义：把原 string 对象从位置 pos 开始的 *n1* 个字符替换为 *n2* 个 char 字符 c。其他规则见原型 1。

示例：string s1="0123456"; s1.replace(2,4,2,'x');，结果 s1="01xx6"。

string s1="0123456"; s1.replace(2,4,6,'x');，结果 s1="01xxxxxx6"。

9. 其他查找成员函数

（1）rfind 成员函数：用于查找字符串最后一次出现的位置。

① 原型 1：size_type rfind(const string& str, size_type pos=npos)const;

意义：在原 string 对象的位置 pos 之前（包括 pos）查找 str 最后一次出现的位置，若未找到则返回 string::npos。

示例：string s1="01ab23456ab789"; string s2="ab";

int n1=s1.rfind(s2);，结果 n1=9;。

int n2=s1.rfind(s2,5);，结果 n2=2;。s1 中位置 5 之前查找 s2 的字符串“ab”最后出现的位置。

int n3=s1.rfind(s2,9);，结果 n3=9;。位置恰好位于 s1 的字符 a 第二次出现的地方，查找时包括位置 pos，因此字符 ab 最后一次出现于此处。

② 原型 2：size_type rfind(const char* c, size_type pos=npos)const;，原型 1 的 char 版本

③ 原型 3：size_type rfind(const char* c,size_type pos, size_type n)const;

意义：在原 string 对象的位置 pos 之前（包括 pos）查找字符数组 c 的前 n 个字符最后一次出现的位置，若未找到则返回 string::npos。

示例：string s1="01ab23456ab789"; char c[]="ab";

int n1=s1.rfind(c,14,2);，结果 n1=9。

int n2=s1.rfind(c,5,2);，结果 n2=2。

int n3=s1.rfind(c,9,2);，结果 n3=9。

④ 原型 4：size_type rfind(char c, size_type pos=npos)const;，原型 1 的 char 单字符版本

（2）find_first_of 成员函数：查找字符串中任意字符首次出现的位置。

① 原型 1：size_type find_first_of(const string& str, size_type pos=0)const;

意义：从原 string 对象的位置 pos 处开始查找 str 字符串中任意字符首次出现的位置。

示例：string s1="01cba23456abc789"; string s2="thabkdc";

int n1=s1.find_first_of(s2);，结果 n1=2;。s1 中的字符 c 是 s2 中的字符，因此最早出现在位置 2。

② 原型 2：size_type find_first_of(const char* c, size_type pos, size_type n)const;

意义：由字符数组 c 中的前 n 个字符组成子串，然后从原 string 对象中的位置 pos 处开始查找该子串中任意字符首次出现的位置。

示例：string s1="01cba23456abc789"; char c[]="thabkdc";

int n1=s1.find_first_of(c,0,4);，结果 n1=3;。由 c 的前 4 个字符构成的子串是“thab”，在 s1 中字符 b 是该子串的字符，而且是最先出现的，因此返回 3。

③ 原型 3：size_type find_first_of(const char*c, size_type pos=0)const;，原型 1 的 char 版本

④ 原型 4：size_type find_first_of(char c, size_type pos=0)const;，原型 1 的 char 单字符版本

（3）find_last_of 成员函数：查找字符串中任意字符最后出现的位置。

① 原型 1：size_type find_last_of(const string& str, size_type pos=npos)const;

意义：在原 string 对象的位置 pos 之前（包括 pos）查找 str 字符串中任意字符最后一次出现的位置。

示例：string s1="01cba23456abc789"; string s2="thabkdc";

int n1=s1.find_last_of(s2); n1=12;，在 s1 中最后一次出现 s2 中的字符位于 7 之前的 c。

int n1=s1.find_last_of(s2,9);，结果 n1=4;。在 s1 的位置 9 处（即数值 5 之前）最后一次出现 s2 中的字符位于 2 之前的 a。

int n1=s1.find_last_of(s2,10);，结果 n1=10;。在 s1 的位置 10 处（即 a）之前最后一次出现 s2 中的字符位于位置 10（因为包括位置 10）处的 a。

② 原型 2：size_type find_last_of(const char* c, size_type pos, size_type n)const;

意义：由字符数组 c 中的前 n 个字符构成子串，然后在原 string 对象的位置 pos 之前（包括 pos）查找该子串中任意字符最后一次出现的位置。

示例：string s1="01cba23456abc789"; char c[]="thabkdc";

int n1=s1.find_last_of(c,14,3);，结果 n1=10;。由字符数组 c 的前 3 个字符组成的子串为"tha";在 s1 中最后含有"tha"中任意字符的位置位于 6 之后的 a。

③ 原型 3：size_type find_last_of(const char* c, size_type pos=npos)const;，原型 1 的 char 版本

④ 原型 4：size_type find_last_of(char c, size_type pos=npos)const;，原型 1 的 char 单字符版本

（4）find_first_not_of 成员函数：在原字符串中查找第一个不是目标字符串中任意字符的位置。

① 原型 1：size_type find_first_not_of(const string& str, size_type pos=0)const;

意义：在原 string 对象的位置 pos 处开始查找第一个不是 str 字符串中任意字符的位置。

示例：string s1="ababc123"; string s2="thabkdc";

int n1=s1.find_first_not_of(s2); n1=5;，数字 1 是最早出现的非字符串 s2 中的字符。

② 原型 2：size_type find_first_not_of(const char* c, size_type pos, size_type n)const;

意义：由字符数组 c 的前 n 个字符构成子串，然后在原 string 对象中的位置 pos 处开始查找第一个不是该子串中任意字符的位置。

示例：string s1="ababc123"; char c[]="thabkdc";

int n1=s1.find_first_not_of(c,0,4);，结果 n1=4;。由字符数组 c 的前 4 个字符构成的子串是“thab”，在 s1 中的字符 c 是第一个非子串中的字符。

③ 原型 3：size_type find_first_not_of(const char* c, size_type pos=0)const;，原型 1 的 char 版本

④ 原型 4：size_type find_first_not_of(char c, size_type pos=0)const;，原型 1 的 char 单字符版本

（5）find_last_not_of 成员函数：在原字符串中查找最后一个不是目标字符串中任意字符的位置。

① 原型 1：size_type find_last_not_of(const string& str, size_type pos=npos)const;

意义：在原 string 对象的位置 pos 之前查找最后一个不是 str 字符串中任意字符的位置。

示例：string s1="ababc123ab45"; string s2="thabkdc";

int n1=s1.find_last_not_of(s2,9);，结果 n1=7;。s1 的位置 9 是字符 b，在这之前最后一个不是 s2 中的字符是数字 3。

② 原型 2：size_type find_last_not_of(const char* c, size_type pos, size_type n)const;

意义：由字符数组 c 的前 n 个字符构成子串，然后在原 string 对象的位置 pos 之前查找最后一个不是该子串中任意字符的位置。

示例：string s1="ababc123ab45"; char c[]="thabkdc";

int n1=s1.find_last_not_of(c,9,4);，结果 n1=7;。由字符数组 c 的前 4 个字符构成的子串是"thab"; 在 s1 中的位置 9（字符 b）之前，最后一个不是"thab"中的字符是 3。

③ 原型 3：size_type find_last_not_of(const char* c, size_type pos=npos)const;，原型 1 的 char 版本

④ 原型 4：size_type find_last_not_of(char c, size_type pos=npos)const;，原型 1 的 char 单字符版本