

适合初学者的程序设计教程



易学 C++

EASY C++

上海大学计算机工程与科学学院

潘嘉杰 编著

版权说明

- 1、 本书版权归作者所有，您无权对本书做任何修改，也不能在未经授权的情况下出版本书。
- 2、 作者同意上海大学的学生以学习为目的地传播本书，但禁止将其用于任何商业用途。
- 3、 根据我国《著作权法》，作者完成创作即自动获得著作权。一旦发生版权纠纷，本人保留诉诸法律的权力。
- 4、 禁止任何企业或个人在未得到作者本人授权的情况下转载本书的全部或部分内容。

前 言

高级语言程序设计是各大院校计算机专业的一门专业基础必修课，主要是学习一些程序设计的基础知识和基本思路。学好高级语言程序设计对后继的一些数据结构、编译原理等课程有着重大的影响。如果一名计算机专业的学生将来想在软件行业有所发展，也需要有着扎实的高级语言程序设计基础。因为高级语言是众多计算机语言中使用最多最广泛的。

然而，由于中学阶段的一些教学问题，各中学毕业的学生在计算机水平方面良莠不齐，有些甚至连一点点程序设计的基础也没有。到了大学里，老师们却因为课时紧张，难以把一些很基本的知识很详细地给学生们解释清楚，以至于日积月累之后，学生们还是觉得什么都不懂。于是，如何能够提高学生的专业水平，更快地把大家领进计算机的世界就显得尤为重要。

本书作者自幼接触并自学了 BASIC 语言，从初高中开始自学 PASCAL 语言和 Visual Basic，对如何学习和掌握高级语言有一些自己的想法，在帮助一些没有程序设计基础的同学更快入门有一些自己的方法。于是就把这些想法和方法一一罗列出来，编著成书。

学习程序设计是一个循序渐进的漫长过程，在短短的几个学期内是不可能很好地掌握的。在学习过程中要求精求全实在是很有难度。而且对初学者来说，知道得越多往往就越是迷茫。所以本书将其他教材中所讲述的一些不常用的技术和一些 C 语言的知识删去，添加了一些平时常用的算法介绍和一些可能和后继课程有部分关联的知识，以帮助大家更快地掌握高级语言程序设计的精髓。

本书作者也只是一名普通的大学生，在考虑一些非常专业的问题上可能尚有欠缺。而且由于时间较为仓促，书中难免有一些错误或不合理的地方，希望各位高人能够不吝赐教。如果您对本书有什么建议或者意见，请发送邮件到 tomatostudio@126.com。

作者

2005 年 12 月

使用指南

本书主要是依照计算机本科专业的实际教学需要来编排内容的。虽然内容的条理可能不及某些专业的 C++ 工具书那么清晰，但是这样的次序让初学者比较容易上手。毕竟那些工具书是面向一些已经掌握 C++ 或有较高的高级语言程序设计基础的读者。编写此书的主旨就是不要一下子把什么都说出来，而是一点一点循序渐进地增长读者的能力。这样，读者就不会一下子被那么多难以接受的概念吓住，以至于失去了继续学习的信心。

本书所有程序使用的编译器是 Microsoft Visual C++，并介绍 C++ 标准，对于 Borland C++ 我们不作讨论，以免初学者把各种概念混淆起来，也有效降低了学习的压力。对于一些 C++ 中存在却不常用的内容，本书一般一笔带过或不予提及。因为这些内容在应试方面不作要求，在实际使用上也可以由其他方法代替。但是，如果您是一位初学者，那么就请务必看到本书的每一个角落。您所遗落的一句话就有可能是一个知识的关键点。

本书的内容有四个特点：

1、粗体字：读者必须掌握理解的内容，也是每个知识点的精髓或要点。很多初学者容易犯的错误也在粗体字中予以提醒。

2、试试看：把一些可能与一般情况不符甚至矛盾的情况列举出来，鼓励读者上机试验，以得到深刻的结论。这些结论可能对以后的学习有所帮助。所以建议有条件的读者务必去试试看。对于没有条件上机的读者，则需要牢记书中给出的结论。

3、算法时间：向大家介绍一些程序设计的常用算法。其实大多数情况下一个程序就是把这些算法以不同形式搭建起来。能够掌握这些算法不论是对阅读别人的代码还是对自己设计程序都有着很大的帮助。

4、习题：帮助大家巩固已经学习的知识。有些题型则是符合应试的要求。从难度上来说，都算适中。如果读者已经掌握了章节中的知识，那么做这些习题也不会有什么困难。

本书的定位是 C++ 程序设计的教学辅导书，而不是 C++ 的工具书或语法书。如果您想了解更多深层的内容，请查阅 C++ 的专业工具书。

目 录

前篇 过程化的程序设计	1
第一章 良好的开端	1
1.1 软件与程序.....	1
1.2 程序设计到底要做什么?	1
1.3 选好一种语言.....	2
1.4 Visual C++能够做些什么?	2
1.5 学习程序设计的方法和一些准备.....	3
第二章 HELLO, WORLD!	4
2.1 如何创建一个示例程序.....	4
2.2 创建自己的World.....	6
2.3 输出与输入.....	7
习题.....	9
第三章 会变的箱子	10
3.1 什么是变量?	10
3.2 常用的基本数据类型.....	12
3.3 不会变的箱子.....	12
3.4 算术表达式.....	13
习题.....	14
第四章 要走哪条路?	16
4.1 如果.....	16
4.2 否则.....	20
4.3 爱判断的问号.....	23
4.4 切换的开关.....	23
习题.....	27
第五章 有个圈儿的程序	31
5.1 程序赛车.....	31
5.2 进维修站和退出比赛.....	34
5.3 圈圈里的圈圈.....	36
5.4 当.....	38
习题.....	41
第六章 好用的工具	45
6.1 简单的工具——函数.....	45
6.2 打造自己的工具.....	48
6.3 多功能开瓶器.....	52
6.4 自动的工具.....	55
6.5 给变量和参数起个绰号.....	56
6.6 *函数里的自己	58
习题.....	59
第七章 好大的仓库	63
7.1 方便地让电脑处理更多数据.....	63
7.2 仓库是怎样造成的?	65

7.3 向函数传递数组.....	69
7.4 二维数组.....	71
习题.....	74
第八章 内存里的快捷方式.....	78
8.1 什么是指针.....	78
8.2 指针变量的定义和使用.....	78
8.3 指针的操作.....	80
8.4 指针与保护.....	82
8.5 指针与数组.....	83
8.6 指针与函数.....	84
8.7 更灵活的存储.....	86
习题.....	88
第九章 自己设计的箱子.....	91
9.1 我的类型我做主.....	91
9.2 设计一个收纳箱.....	94
9.3 结构与函数.....	96
9.4 结构数组与结构指针.....	98
9.5 自行车的链条.....	100
9.6 链表的实现.....	101
习题.....	108
中篇 实战程序设计.....	112
第十章 如何阅读程序代码.....	112
10.1 整体把握法.....	112
10.2 经验法.....	114
10.3 模拟法.....	115
习题.....	116
第十一章 如何调试程序代码.....	120
11.1 再谈变量.....	120
11.2 头文件的奥秘.....	126
11.3 更快更好地完成程序调试.....	130
11.4 最麻烦的问题.....	134
11.5 调试工具——Debug.....	138
习题.....	141
第十二章 如何编写程序代码.....	144
12.1 程序设计的基本步骤.....	144
12.2 三类问题.....	144
12.3 函数的递归.....	149
习题.....	154
后篇 面向对象的程序设计.....	156
第十三章 初识对象.....	156
13.1 对象就是物体.....	156
13.2 一个字符串也是对象.....	156
13.3 面向对象特点一：封装性.....	159

13.4 从数组到向量.....	159
习题.....	161
第十四章 再识对象	162
14.1 类是一种数据类型.....	162
14.2 公有和私有.....	163
14.3 成员函数.....	164
14.4 对象、引用和指针.....	167
习题.....	167
第十五章 造物者与毁灭者	171
15.1 麻烦的初始化.....	171
15.2 造物者——构造函数.....	172
15.3 先有鸡还是先有蛋?	175
15.4 克隆技术.....	180
15.5 毁灭者——析构函数.....	187
习题.....	189
第十六章 共有财产·好朋友·操作符	193
16.1 有多少个结点?	193
16.2 类的好朋友.....	197
16.3 多功能的操作符.....	205
习题.....	212
第十七章 父与子	213
17.1 剑士·弓箭手·法师的困惑.....	213
17.2 面向对象特点二：继承性.....	214
17.3 继承的实现.....	214
17.4 子类对象的生灭.....	224
17.5 继承与对象指针.....	228
17.6 面向对象特点三：多态性.....	231
17.7 多态与虚函数.....	231
17.8 虚函数与虚析构函数.....	236
17.9 抽象类与纯虚函数.....	238
17.10 多重继承.....	241
习题.....	242
第十八章 再谈输入与输出	261

各位读者：

感谢大家对我和我的作品的支持！看到众多网友来信与我交流，肯定我的写作风格、向我提出众多宝贵的建议，我感到非常高兴。第 17 章将会是我公布在网上的最后一个章节。之后的 2-3 章或许大家以后可以在书店里看到。近期我会和各大出版社联系，讨论出版事宜，以便更多的读者能够快速踏入 C++ 程序设计的大门。如果您有在出版社工作的朋友，并且看好我的作品，也欢迎您主动来和我联系！我的邮箱地址为 tomatostudio@126.com。

2007 年 7 月 5 日
于上海大学延长校区 计算机学院

前篇 过程化的程序设计

第一章 良好的开端

本章主要讲述一些学习程序设计前需要了解的一些知识和一些学习程序设计的方法。并且对 C++ 语言作了一个简要的介绍。学好这一章，对日后的学习能够起到事半功倍的效果。

1.1 软件与程序

随着电脑的普及和科学技术的发展，无纸化办公、电脑辅助设计（CAD——Computer Aid Design）和电脑辅助制造（CAM——Computer Aid Manufacture）已经渐渐走进我们的日常工作中。有了电脑的帮助，我们的工作效率得到明显的提升。财务人员不必一天到晚扎在账本堆里了；报社编辑点一下鼠标就能够发稿了；设计人员只需要把数据输入电脑，就能显示出一个精确的三维立体模型了。当我们使用电脑的时候，有没有想过人类是如何教电脑学会这些的呢？

其实我们平时对电脑进行的操作是在与电脑软件（Software）打交道。电脑之所以能够帮助人类工作，离不开软件的支持。那么软件到底是什么？其实它是看不见摸不着，但却能够通过电脑为用户所用的一种东西。打一个比方，电脑的各种硬件（Hardware）设备就像是人的肉身，而软件就像是人的灵魂。少了软件这个灵魂，那么电脑只能是一堆废铜烂铁。人们通过编写一款软件，来教会电脑做一些事情。像我们用的 Windows、Word、QQ 甚至游戏都是软件。

那么，软件和我们说的程序（Program）又有着什么样的关系呢？首先，我们要弄清什么是程序。从初学者比较容易理解的角度说，**程序是电脑执行一系列有序的动作的集合**。通过一个程序，可以使电脑完成某一类有着共同特点的工作。如求解一个一元二次方程，或是找出一组数里面最大的一个数。而一款软件，往往是由若干个相关的程序、运行这些程序所需要的数据和一些额外的文档（如软件介绍或帮助文档）等文件组成的。因此，要设计出一款软件，就必须从程序设计开始。

1.2 程序设计到底要做什么？

很多初学者会不解：程序设计到底是要做什么呢？我们该如何教会电脑解决问题呢？

其实，要解决一些看似不同的问题，我们可以将其归结为一种确定的过程和方法。**我们把这种能够解决一类问题的过程和方法称为算法（Algorithm）**。下面，我们以解一元二次方程为例，介绍求解的算法：

- （1）输入二次项系数 a ，一次项系数 b 和常数项 c 。
- （2）计算 $\Delta = b^2 - 4ac$ 。
- （3）判断 Δ 的大小，如果 $\Delta \geq 0$ ，则有实数解，否则就没有实数解。
- （4）如果有实数解，就利用求根公式求出两个解。
- （5）输出方程的两个实数解，或告知无解。

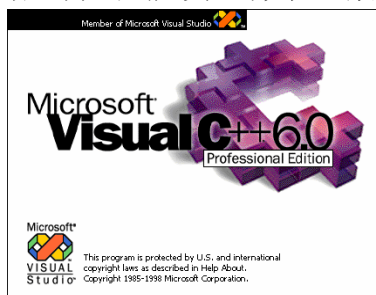
以上便是用自然语言描述的求解一元二次方程的算法。程序设计所要做的便是探求这种能解决一类问题的算法，并且要将这种算法用计算机能够“看懂”的语言表达出来。

1.3 选好一种语言

电脑是无法懂得人类的自然语言的。它有着它自己的语言。电脑中最原始的语言是机器语言，它纯粹由一串数字“0”和“1”组成。这样的语言实在是冗长难记，对一般人来说实在难以入门。接着又发明了汇编语言，机器语言指令被变成人类能够读懂的助记符，如 ADD，MOV。然而，用汇编语言编一个复杂的程序仍然显得有些困难。为了能够让电脑的语言更通俗易懂，更接近人类的自然语言，出现了高级语言。比较著名的高级语言有 Basic、Pascal、C++、Java 等。本书中所说的程序设计是指高级语言的设计。

学习程序设计之前，选好一种语言是十分有必要的。如果你是一名初学者，那么你选的语言可能不需要很强大的功能，但要能很快地让你适应让你入门；如果你将来想从事软件设计工作，那么你务必要选一种比较符合潮流，并且有美好前景的语言。

本书选择微软公司（Microsoft）Visual C++环境下的 C++作为教学语言，一方面是因为它是时下流行的高级语言，与 Java 也有很多共通之处，另一方面是因为它既能够实现以前的结构化程序设计，方便初学者入门，又能够担当现在流行的面向对象的程序设计。



(图 1.2)

1.4 Visual C++能够做些什么？

C++能够制作多种 Windows 下的软件。事实上，Windows 下的应用软件也有很大部分是用 C++开发的。比如设计控制台应用程序可以编出计算量较大的，用于科学计算的小程序；用 MFC 类库可以设计中小型企业的内部管理程序；用一些图形的 API 可以编出大型的 3D 游戏，或者游戏机模拟器；利用 C++能够接触系统底层的特点，可以编出优化软件让内存运作的性能大大提高；利用 C++可以与内存打交道的特点，可以编出游戏修改器；甚至用 C++可以编出手机游戏来。总而言之，C++的功能是非常强大的。

由于 C++是面向对象的高级语言，所以用它来开发软件可以大大减少重复的工作，使得设计程序更为轻松。

为了降低学习的难度，本书主要介绍如何设计控制台应用程序。控制台应用程序是 C++程序设计的基础。它涵盖了 C++程序设计的大部分知识。而更大型的软件设计与之也是触类旁通的。所以，学好如何设计控制台应用程序，便是为将来打下了扎实的基础。

1.5 学习程序设计的方法和一些准备

学习方法：

——四“多”一“有”

- 1、多看：多看别人写的程序，从简单的程序看起，揣摩别人的思想和意图。
- 2、多抄：挑选难度合适的别人编写好的代码，亲自去尝试一下运行的结果。在不断地借鉴别人的代码过程中，你的思维会不断升级。
- 3、多改：正所谓“青出于蓝胜于蓝”，把自己的思想融入别人的思想中，那么你就得到了两种思想。
- 4、多实践：不要用纸和笔来写程序。没有人能保证那样写出来的程序一定能执行。一定要勤上机、勤测试，那样，你的水平才能真正提高。
- 5、有风格：一名优秀的程序员都有着良好的风格习惯。至于这种良好的习惯如何养成，以后会在各章节陆续介绍。

必要准备：


——五“要”

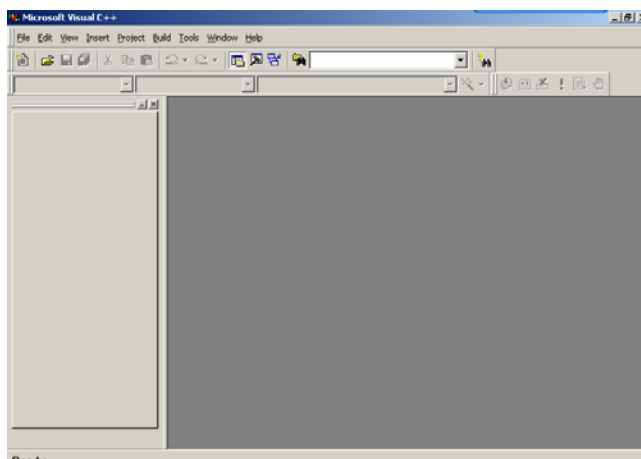
- 1、要有一定能学会的信心和坚持到底的决心。
- 2、要有足够的时间去经常写程序，经常去实践。长时间不写程序，水平就会退步。
- 3、要有良好的身体素质。做程序员很伤身体，废寝忘食更是家常便饭。
- 4、要有一定的电脑常识和实践操作基础。
- 5、要有电脑和相关软件。

第二章 Hello, World!

本章先不介绍很多枯燥的理论知识，而是通过“Hello, World!”这个示例程序教会大家如何自己编一个很简单的程序。在这一章里，我们将介绍输入输出、一个程序的基本结构和字符串等简单知识。你也可以通过本章来了解设计一个程序的基本步骤。

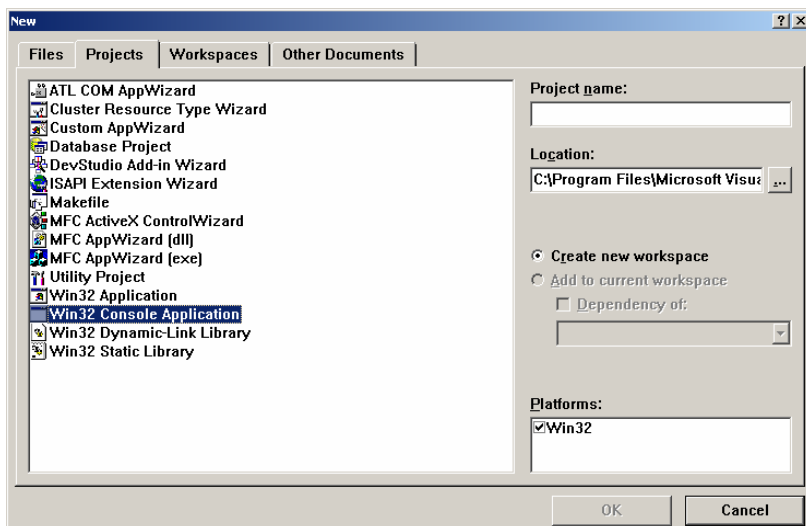
2.1 如何创建一个示例程序

首先，我们要进入 Microsoft Visual C++ 集成开发环境（IDE——Integrated Development Environment），双击图标  即可。进入以后，我们可以看到如下界面。（图 2.1.1）



（图 2.1.1）

单击左上角的 File 菜单（Menu），选择 New，会跳出如下对话框。（图 2.1.2）



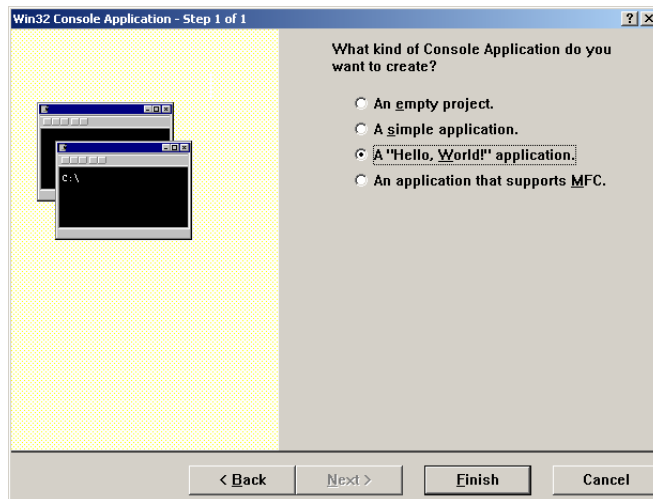
（图 2.1.2）

图 2.1.2 所在的是 Project（工程）选项卡。设计程序就好像造房子需要图纸、建筑材料和建筑工具一样，也需要各种各样的东西，如程序代码、头文件或一些额外的资源。这些东西都是放在一个工程里的。工程能够帮助协调组织好这些文件和资源，使得设计更有序，查

找更方便。注意，每一个工程只能对应一个设计的程序，切莫把多个程序一股脑儿塞在一个工程里！

左面部分是提供工程类型的选择，即我们要设计何种类型的程序。我们要学习的是控制台应用程序，所以选择 Win32 Console Application（如图）。右边的 Project Name 为工程名，应该不难理解。而 Location 则是工程保存的位置，如果你对保存位置这个概念还不清楚，那么请查阅一些初学者的参考书。要说明的是，当在 Location 下新建一个工程后，会在 Location 这个位置下新建一个以工程名命名的文件夹。而通过打开这个文件夹中的“工程名.dsw”文件可以打开该工程。

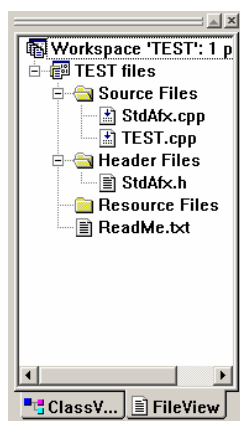
选好类型，填好工程名和保存位置，按“OK”，又出现了如下对话框（图 2.1.3）。



（图 2.1.3）

该对话框问我们要创建哪种控制台应用程序，我们要创建一个“Hello, World!”的示例程序，所以我们选第三项（如图）——A “Hello, World!” application。（思考：如果我们要自己编一个控制台应用程序，我们应该选哪个？）

单击 Finish 后，会弹出一个关于新工程信息的信息框，再次点击 OK 后，示例工程出现。找到 Workspace 框（事实上这个框上没有这个名称，该框在整个集成开发环境左方），单击 File View，将所有树状目录点开，如图 2.1.4。



（图 2.1.4）

我们可以看到三个文件夹结构，分别是 Source Files（源文件）、Header Files（头文件）和 Resource Files（资源文件）。源文件主要是存放设计的主程序代码，头文件主要是存放一些预处理文件（关于什么是预处理文件，后面会另作介绍），资源文件一般是存放一些运行该程序所必需的一些资源，比如图像，文本等类型的文件。不过，这里的文件夹结构并不是磁盘上的文件夹结构，而只是这些文件在该工程中的分类。所以，如果你没有自己创建过这些名字的文件夹，那么在工程中是无法找到这些文件夹的。

双击某个文件，即可查看它的内容。本节只是介绍如何创建一个程序，所以对各文件内的代码不作介绍。

最后要介绍如何让设计好的程序运行起来。先要打开主程序代码文件（本例中为 StdAfx.cpp），然后点 Build 菜单，再点 Compile StdAfx.cpp（编译）。所谓编译，就是使用编译器软件将我们比较容易掌握的高级语言翻译成计算机可以识别的低级语言。如果没有经过编译（或解释），高级语言的程序代码是无法被执行的。

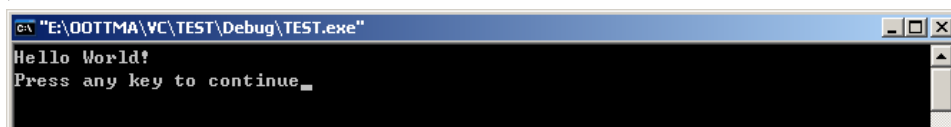
完成编译后，还要再点 Build 菜单，点 Build 工程名.EXE。我们通常把这个过程称为连

接 (Linking)，其作用是将多个源文件的程序模块都整合起来。当一个程序规模比较大的时候，连接也成为保证程序能正常运行的关键步骤之一。

最后再点 Build 菜单里的 Execute 工程名.EXE (执行)，就能运行程序，并查看结果了

(如图 2.1.5)。以后熟练了就可以使用快捷键或快捷按钮。如果代码完全正确

且能正常 Compile，但是 Build 的时候提示有错误，请检查是否上次运行了程序之后没有将它关闭。



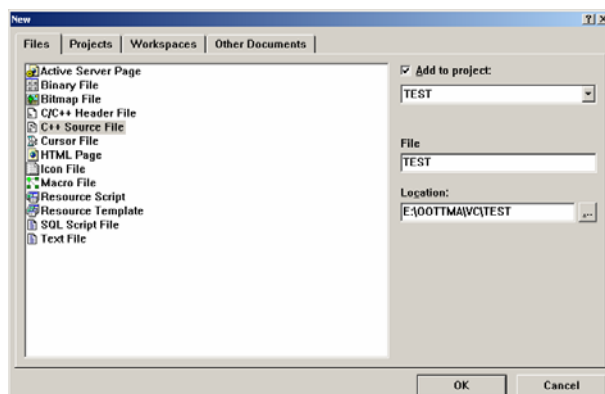
(图 2.1.5)

2.2 创建自己的 World

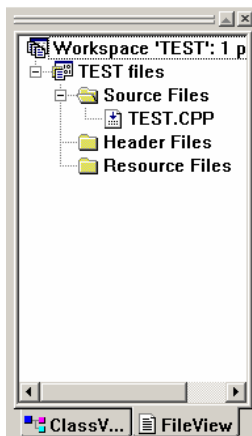
上一节我们介绍了如何创建一个“Hello, World!”示例程序，这一节我们要来自己动手写一个程序。

参照上一节的方法，我们创建一个控制台应用程序的工程，在选择类型的时候选 An Empty Project，即一个空工程。该工程中没有任何文件或代码。

然后，我们要开始准备写代码了。可是，没有任何文件我们写在哪里呢？所以，我们先要新建一个源文件。点击 File 菜单，按 New，这次会出现如下对话框 (图 2.2.1)。



(图 2.2.1)



(图 2.2.2)

当我们打开着一个工程再按 New 的时候，出现的默认选项卡是 Files，即向打开的工程中添加新文件。在左边选择文件类型为 C++ Source File 表示我们要新建一个源文件。在右边的 File 中填入文件名，按 OK 以后，就会在 Workspace 框的 Source Files 里面出现这个新建的文件，文件扩展名为 **cpp**，如左图所示 (图 2.2.2)。

双击该文件，使其处于打开状态，然后在文本框内输入以下代码：

```
#include "iostream.h"
int main()
{
    cout<<"My Own World!";
    return 0;
}
```

我们来试一试运行一下，可以看到屏幕上有这样的结果。

```
My Own World!Press any key to continue
```

下面我们来一句一句地分析一下每个语句的意思。

`#include "iostream.h"`//包含“`iostream.h`”文件，该文件使得设计的程序具有输入输出功能。

`int main()`//主函数，**每个程序只能有并且必须有一个主函数**，在没有系统地介绍函数之前，只能认为这是一种定式。

`cout<<"My Own World!"`;//你可以猜测一下，这个 `cout`（念 C-Out）起到了什么功能。“`<<`”称为插入操作符，在下一节输入输出中将予以介绍。在 `My Own World` 两端加上引号表示它是一个字符串，是一个整体。

`return 0`;//主函数的返回值，暂时也认为是一种定式。

根据程序中的 `cout` 语句，我们可以归纳出 `cout` 的语法格式：

```
cout <<字符串1[<<字符串2 …… <<字符串n];
```

在语法格式中，中括号一般表示根据实际需要，可有可无的内容。**要注意，在实际输入代码的过程中，中括号是不需要输入的。**如果我们要输出多个字符串，则可以通过多加一些插入操作符和字符串来实现。而在语句的最后，我们要加上一个“`;`”。在 C++ 中，分号表示一句语句的结束，但是它的位置是有一定规则的，待介绍了更多语句后，再作归纳。

刚才我们发现在代码中键入“`//`”后的字符串都会变成绿色，而且无论这些字符串是什么，对程序的运行结果都不会产生影响。这叫作注释，是程序员对某句语句的一个解释，方便自己或别人理解这句语句的意思。**写注释是一种好习惯，能够及时把自己当时的思路记录下来，但是这并不是说注释越多越好。在一些很简单的、显而易见的语句后加上注释便是多此一举了。**“`//`”称为单行注释符，当注释要占用多行时，可以用“`/*`”表示注释的开始，用“`*/`”表示注释的结束。尽管如此，注释还是应点到为止，从简为宜。

最后，我们通过这个程序总结出一个简单而完整的 C++ 程序代码结构：

预处理头文件

主函数

```
{
    语句1; //注释
    ……
}
```

试试看：

1、如果去掉第一行的 `#include "iostream.h"`，该程序能否正常运行？

结论：去掉预处理头文件将影响某些功能的实现。

2、如果去掉字符串两端的双引号，是否还能输出这些字符？

结论：输出字符串不能将两端的双引号丢掉。

3、如果去掉 `cout` 语句后的分号，编译时会出现什么错误提示？

4、试试 `cout<<3+4`；会是什么结果。

2.3 输出与输入

我们已经学会如何用 `cout` 语句输出字符串，那么 `cout` 能否输出别的东西呢？如果你已经做了上一节试试看的第四题，那么你会发现，`cout` 还能够输出运算的结果。

你有没有想过，用现在的知识，能否输出一个回车符或者一些键盘上没有的字符呢？事

实上还不能。在此，我们引入一个转义字符的概念，即用多个键盘上有的字符来表示一些键盘上没有或者不方便输出的字符。转义字符仍作字符处理，加在字符串的双引号内。以下给出的是常用转义字符表：

转义字符	功能	转移字符	功能
\a	响铃	\n	换行
\t	制表符(相当于 Tab)	\\	反斜杠
\'	单引号	\"	双引号

在 C++ 中,还有一种更为常用的输出换行的方法,为 `cout <<endl;`,以后再程序中会经常遇到。

试试看：

- 1、试输出 My Own World! 后换行。
- 2、试输出”My Own World! ”后换行。
- 3、试输出 My Own World! 的同时发出响铃。

`out` 是出的意思,那么输入是不是 `in` 呢? 没错,输入就是 `cin` (念 C-In)。它的语法格式和 `cout` 很类似:

```
cin >>变量1 [>>变量2.....>>变量n];
```

我们发现,在 `cin` 语句中,双箭头的方向和插入操作符的方向相反。“>>”叫做抽取操作符。虽然这两个符号的名字都比较难记,但是它们的功能却很好理解。“<<”是箭头从字符串指向外面,好像把东西从字符串里拿出来,所以就是输出功能;而“>>”是箭头指向变量,好像是把东西放进去,所以就是输入功能。

关于变量的具体知识,我们会在下一章作详细的讲解。现在大家只要记住,输入的时候东西一定要放到变量里。下面我们来试一段程序:(程序 2.3)

```
#include "iostream.h"
int main()
{
    char a; //创建一个字符变量 a
    cout <<"请输入字符: "; //输出提示消息
    cin >>a; //把键盘输入的字符放入变量 a
    cout <<"刚才输入的字符是" <<a <<endl; //输出提示消息并把变量 a 里的字符输出
    return 0;
}
```

我们来看看运行结果:

请输入字符: T

刚才输入的字符是 T

Press any key to continue

需要注意的是,如果我们给 `cout <<"刚才输入的字符是" <<a <<endl;`的 `a` 加上双引号,那么无论我们输入什么,输出的始终是一个字符 `a`。所以当我们输出变量中的内容时,千万不能给它加上双引号。

在运行结果中显示的“Press any key to continue”是由系统给出的,表示程序已经运行结束。在书中以后的运行结果中,这句话会被省略。

书中带底纹的字符表示从键盘输入的字符。我们可以看到，我们输入什么，最后的结果就能够输出什么。输入功能达到了我们所需要的效果。而且，通过这个程序，我们还知道了 cout 不仅能够输出字符串和运算结果，还能输出变量。

试试看：

- 1、在上述程序中，如果输入了多个字符，那么最终输出的应该是哪个字符？
- 2、已知对于整数可以通过 `int a,b;` 语句来创建一个名为 a 和 b 的整数变量，试用输入输出语句实现输出任意两个整数的和。
- 3、在执行 `cin` 语句时，输入 `1 + 1` 等表达式电脑是否能够识别？

结论： `cin` 语句中，表达式不能被电脑识别。

习题

- 1、请写出完整的 C++ 代码结构。
- 2、试用一句 `cout` 语句输出以下图形：


```
****
***
**
*
```
- 3、判断下列使用 `cout` 语句输出 Hello,World 的写法是否正确？如果不正确，请将其改正。
 - (1) `cout <<'Hello,World';`
 - (2) `cout >>"Hello,World";`
 - (3) `cout 'Hello,World';`
 - (4) `cout <<"Hello,World"`
- 4、根据运行结果完善代码，并根据已有的代码补全运行结果。

```

_____
int _____
{
    char a;
    cout <<"欢迎光临！" <<endl;
    cout <<endl;

    _____
    cin >>a;

    _____
    return 0;
}

```

运行结果：

请输入您的房间： F

您的房间是 F

第三章 会变的箱子

我们已经学会了如何从键盘获取信息和如何把结果反馈到屏幕。在上一章的最后，我们遇到了变量这个名词。从本章开始，我们要详细介绍什么是变量，还涉及到诸如数据类型、常量、简单表达式等知识。

3.1 什么是变量？

电脑具有存储的功能。我们可以通过 Word 打开一个保存的文章，也可以通过 FPE（整人专家，一款游戏修改软件）来查看或锁定内存中保存的游戏人物的生命值。那么，一个程序是如何把数据存到电脑里，又是如何把电脑里的数据取出来的呢？

在设计程序的时候，我们把要存储的数据放在一个叫变量（Variable）的东西里。它就好像是一个箱子，而数据就是箱子里的物品。当然，在我们放东西和取东西之前必须要创建这么一个箱子，这条创建变量的语句又称为变量的声明（Declaration）。它的语句格式为：

变量数据类型 变量名₁， 变量名₂， ……变量名_n；

数据类型

我们刚才说了，变量就好像是一个箱子。可是不同的东西也要放到对应的箱子里。如果把吃的东西放在文具盒里，把衣服放进饼干盒里，显然是不合适的。变量也是一样的。有些数据是文字（字符或字符串），有些数据是数字（整数或者实数），把它们随便乱放，那么电脑就可能无法理解这些数据的含义了。

我们常用的**基本数据类型**由下表列出：（以 VC 为例）

主类型	分类型	修饰符	占用空间	表示范围
整型	整数型 int	short	2 字节	-32768~32767
		long（默认）	4 字节	$-2^{31} \sim (2^{31}-1)$
		unsigned short	2 字节	0~65535
		unsigned long	4 字节	$0 \sim (2^{32}-1)$
实型	浮点型 float	无	4 字节	-3.4E38~3.4E38
	双精度型 double	long（默认）	8 字节	-1.7E308~1.7E308
字符型	字符型 char	signed（默认）	1 字节	-128~127
		unsigned	1 字节	0~255
逻辑型	布尔型 bool	无	1 字节	0~1

在程序 2.3 中，`char a;`就是声明了一个字符型变量。修饰符是放在分类型之前的，比如要创建一个短整型变量 A，就应该是 `short int A;`了。

要注意，两个数据类型截然不同的变量是不能放在同一个语句中定义的。比如企图通过 `int a, char b;`声明整型变量 a 和字符型变量 b 是不可以的。

我们在选择数据类型时，要尽量选择满足使用要求的类型。比如我们要算一个一元二次方程的解，就应该选择精度较高的浮点型或者双精度型，而不能选一个整数型；**同时，我们也要有“够用就行”的好习惯。**如果我们创建一个双精度型的变量去存储从整数 1 到 100 的和，那就显得大材小用，太浪费了。一个变量所占内存（Memory）的空间是和这个变量

的数据类型有关的。虽然现在电脑的内存已经可以达到 1GB，但是如果在设计大型软件时经常“大材小用”，即使有了更多的内存，也会捉襟见肘的。

变量名

我们创建了一个箱子，用它的时候总应该用一样东西来表示它，那就是变量名。变量名的意义就如同给文件夹起个名字，或者给文件起个名字。不过，其名字也是要讲规则的。具体规则如下：

1、**不能是可能与 C++ 中语句混淆的单词。**（这种单词称为**保留字**，具体哪些是 C++ 的保留字可以查阅书后的附录。凡是在输入代码时，自动变成蓝色的单词，一定是保留字。）如果我们创建一个名为 `int` 的变量，那么这个 `int` 到底是一个变量名还是另一个变量的数据类型呢？电脑糊涂了。

2、**第一个字符必须是字母或者是下划线。**

3、**大小写不同的变量名表示两个不相同的变量。C++ 是大小写敏感的。**所以如果把 C++ 中的语句打成大写字母，就会造成错误。

4、**变量名中不应包括除字母、数字和下划线以外的字符。**因为某些特殊字符在 C++ 中具有分隔功能，电脑无法判断到底这是一个变量还是多个变量。

变量名应该尽量符合变量里面存放东西的特征。这样，自己和别人在阅读代码的时候才能一目了然。我们介绍两种比较常用的变量名标记法：驼峰标记法和匈牙利标记法。驼峰标记法是以小写字母开头，下一个单词用大写字母开头，比如 `numOfStudent`、`typeOfBook` 等等，这些大写字母看起来像驼峰，因此得名。而匈牙利标记法是在变量名首添加一些字符来表示该变量的数据类型，比如 `iNumOfStudent` 是表示学生数的整型变量，`fResult` 是表示结果的浮点型变量等等。不过，如果一个程序实在是非常简单，那么用诸如 `a`，`b`，`c` 作为变量名也未尝不可，只要你能够记住这些变量分别应该存放什么数据就行了。

变量的初始化

前面我们说到，变量是存放在内存里面的。而内存又是有限的，在某些情况下，我们创建一个变量的时候，并不是真的重新造了一个“箱子”，而是把“弃置不用的旧箱子”拿来用。但是那些“旧箱子”里往往是有一些原来的数据，这些数据是不确定的。所以，在我们使用“箱子”之前，需要把原来的旧数据处理掉。这个过程称为**变量的初始化 (Initialization)**。具体格式为：

变量名=初始值；

或者我们可以在声明变量时，在变量名的后面加上“=初始值”即可，如 `int a=5;`。在初始化的时候，要注意设置的初始值要符合变量的数据类型。

试试看：

- 1、试自己编一段程序，分别按整数类和浮点类输出由键盘输入的数值。
- 2、如果在主函数中声明的变量未经初始化，那么里面的数据是什么？

结论：未知的随机数值

3.2 常用的基本数据类型

整型 (Integer)

所谓整型，就是我们平时说的整数。在 VC++ 中，int 默认为 long int，即有符号的长整型数据。一个有符号的长整型变量在内存中占用 4 个字节的空间，它的表示范围是从 -2147483648~2147483647。虽然整型数据无法表示小数，表示的范围也不是非常大，但是在它范围内的运算却是绝对精确的，不会发生四舍五入等情况。

我们常用整型数据来表示人数、天数等可数的事物。对于长度不是很长的编号，如学号、职工号，也可以用整型数据来表示。

实型 (Real)

所谓实型，就是我们平时说的实数。在 C++ 中，实数分为浮点型和双精度型。两者的主要区别是表示范围不同和占用的存储空间不同。我们可以用两种方式来表示实数：

1、小数形式，比如 0.1、0.01、12.34 等等，和我们平时日常生活中的表达并无异样。

2、指数形式，即科学记数法。比如 0.25E5 表示 0.25×10^5 ，E 表示 10 的多少次方，也可以用 e 表示。**注意，在 E 之后的指数必须是整数。**

虽然实数表示范围比整数要大得多，精度也更高，但是用它进行运算却不是绝对精确的。比如同样是 $800000000+1$ ，整数的运算结果是 800000001，而实数的运算结果却是 $8e+008$ 。

我们常用实型数据来进行科学运算，比如计算一个数的平方根，或是纪录地球到月亮的距离。而温度、价格、平均数等可能出现小数的数据，我们也通常也用实型数据来表示。

字符型 (Character)

一个字符型变量可以存放一个半角西文字符或者一个转义字符。同时在字符两端要加上单引号，比如：`char a='a',b='\n'`。

要注意，字符型数据 '1' 和整型数据 1 是不同的。虽然它们输出时的现象是一样的，但是它们存储的内容是不一样的。关于字符型数据的存储，我们将在后续章节中再作介绍。

布尔型 (Boolean)

布尔型数据的取值只能是 0 或 1，也可以分别写作 false 和 true。0 表示假 (false)，1 表示真 (true)。但是，在我们以后学到的内容中，只要数值不等于 0 的都表示为真 (包括负数)。所以，方便地记就称为“有真无假”。

3.3 不会变的箱子

上一节我们介绍了变量，它是一种存储在电脑内存里，在程序中可以改变的数据。然而，有时候我们还会遇到一些数据，它们在程序中不应该被改变。比如圆周率 π 就应该等于 3.1415926……，一年就应该是 12 个月，在程序中不应改变成其它的值。或者说，如果这些值无意之中被改变，会导致整个程序发生错误。这个时候，我们就需要一个不允许改变的“箱子”，我们称它为常量 (Constant)。

常量可以分为两种，一种是文字常量，也叫值常量，比如整数 1，字符'a'就是文字常量；另外一种是需要通过自己定义的常量，它的表达和变量有些类似。

定义一个常量的语法格式为：

const 常量数据类型 常量名=文字常量；

我们可以认为，定义一个常量与定义一个变量的区别是在语句之前加上了 const。但是，定义常量的时候必须对其进行初始化，并且在除定义语句以外的任何地方不允许再对该常量赋值。

特别地，如果一个实型文字常量没有做任何说明，那么默认为双精度型数据。若要表示浮点型数据，要在实数之后加上 F；若要表示长双精度型数据，则要在实数之后加上 L。

在 cout 语句中，我们说它可以输出字符串，这些带着双引号的字符串的全称是字符串常量，它也是一种文字常量。而带着单引号的常量称为字符常量，它与字符串常量是不同的，字符常量只能是一个字符，而字符串常量则可以是一个字符，也可以由若干个字符组成。

事实上，只要在不人为地改变变量值的情况下，常量可以由一个变量来代替。但是从程序的安全和严谨角度考虑，我们不推荐这样做。区别常量和变量的使用是一个优秀程序员需要具有的好习惯。

3.4 算术表达式

我们已经了解了程序设计中，最常用的两种存储方式——常量和变量。本节我们要学习如何在程序中运用常量和变量。我们先来看一段程序：（程序 3.3）

```
#include "iostream.h"
int main()
{
    float r;//创建一个浮点型变量存放半径
    float l;//创建一个浮点型变量存放运算得出的周长结果
    const float pi=3.1415926F;//定义常量 pi 等于 3.1415926，最后的 F 表示这个数是浮点型
    cout <<"请输入半径： ";
    cin >>r;
    l=2*pi*r;//计算周长
    cout <<"这个圆的周长为" <<l <<endl;
    return 0;
}
```

程序的运行结果：

请输入半径： 3

这个圆的周长为 18.8496

我们需要重点研究的是 $l=2*pi*r$ 这句语句。这句语句称为赋值语句，赋值语句的语法格式为：

左值=表达式；

语句中，等号称为赋值操作符。赋值操作符的作用就是把表达式的结果传递给左值。具体的过程是先将右侧的表达式值求出，然后再将它存放到左值中。所以在赋值操作符两边出现相同的变量也是允许的。比如 $a=a+1$ 就是先把原来 a 的值和 1 相加，然后再把结果放回变量 a 中。左值（Left Value，也作 L-Value）的原意是在赋值操作符左边的表达式，它具有存储空间（比如自定义常量或变量），并且要允许存储（自定义常量只能在定义时初始

化)。现在了解的知识中，左值可以理解为变量或定义语句中的自定义常量。

像程序中的 $2*\pi*r$ 我们称为算术表达式。它和平时数学上的表达式没有什么不同。如同四则运算一样，算术表达式中使用的是加减乘除和括号，运算的次序也是遵循“括号最先，先乘除后加减”的原则。**需要注意的是：**

1、表达式中，乘号是不能够省略的，即 $2a$ 、 $4b$ 之类的表达式是无法被识别的。

2、算术表达式中，括号只有小括号 () 一种，并且可以有多种括号。中括号 [] 和大括号 { } 都是不允许使用在算术表达式中的。比如 $((a+b)*4)$ 是正确的写法， $[(a+b)*4]$ 却是错误的写法。

除、整除和取余

在 C++ 中，“/” 有两种含义：当除号两边的数均为整数时为整除，即商的小数部分被截去（不是四舍五入）；除号两边只要有一个是实型数据，那么就做除法，小数部分予以保留，运算结果应当存放在实型变量中。

取余数的操作符为 %，例如 $7\%3$ 的结果是 1。它和乘法类似，在加减法之前执行运算。**注意，在取余数操作符的两边都应该是整数，否则将无法通过编译。**

至此，我们已经学会了输入、输出和简单的运算。运用这些知识，我们已经能够自己设计一些简单的程序，实现一些计算功能。

试试看：

- 1、如果定义一个浮点型的常量时，不在实数之后加上 F，是否能够通过编译？
- 2、假设已定义两个未初始化整型变量 a 和 b，赋值语句 $a=b=1$ 是否是合法的？如果合法，那么 a 和 b 的结果分别是什么？
- 3、7 整除 -2 的结果应该是多少？ $-7\%2$ 的结果应该是多少？请上机验证。

习题

- 1、尝试自己编一段程序，通过键盘输入一个圆的半径 r，经过运算后输出这个圆的面积。
- 2、判断下列数据的数据类型。

3.0 2 '\$' 0 0.0F "ABCDE"

- 3、判断下列变量名是否合法。

3ZH Int _3CQ int 分数 Phy Mark

- 4、指出下列程序的错误之处。

```
#include iostream.h
int main
{
    int b,c=5;
    const int a;
    a=1;
    b=c+1;
    cout >>b >>endl;
    c=c/2.0;
    cout >>a+b+c >>endl;
```

```
    return 0;  
}
```

5、阅读以下代码，写出运行结果。

```
#include "iostream.h"  
int main()  
{  
    int a=5,b=3,c;  
    c=a+b;  
    a=a-1;  
    b=b-1;  
    c=c+1;  
    cout <<a+b+c <<endl;  
    return 0;  
}
```

提示：把变量的初始值和运行每一步语句后的结果写下来，最后就能得到准确无误的答案。

6、猜测以下变量中存放的数据类型和可能的内容。

如：iNumOfStudent——整数型，可能存放学生的人数

iMax

fSum

chTemp

bIsEmpty

lResult

第四章 要走哪条路？

世间万物是变幻莫测的，对于发生的不同状况，可能会有着许多不同的结果。比如考试的成绩高，对应得到的绩点就较高。考试的结果左右着能够得到的绩点。那么，在 C++ 的程序设计过程中，我们要如何来描述这个千变万化，难以预料的世界呢？本章我们将要学习分支结构程序设计，通过分支的方法，使得我们的程序更加完善，能够解决更多的问题。

4.1 如果……

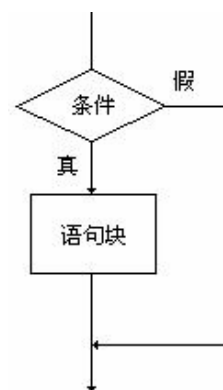
对于可能发生的事情，我们平时经常会说“如果……，那么……”。语文里，我们叫它条件复句。“如果”之后的内容我们称为条件，当条件满足时，就会发生“那么”之后的事件。我们来看这样一句英语：If mark>90, cout <<"GOOD!" <<endl. 把它翻译成中文就是：如果分数大于 90，则输出 GOOD。

其实在程序设计中，也是用“如果”来描述可能发生的情况的。它和刚才的那句英语很相似，具体的语法格式是：

```
if (条件)
{
    语句1;
    语句2;
    .....
}
```

} 语句块

我们把若干句语句放在一个大括号中，称为语句块。运行到该 if 语句，当条件满足时，就会执行语句块内的内容。我们也可以用流程图（图 4.1.1）来表示 if 语句。**请注意，if 语句的结束是没有分号的，分号只是属于语句块中的语句。**



(图 4.1.1)

条件——关系运算

当我们判断一个条件的时候，依赖于这个条件是真是假。说到真和假，我们不难想到布尔型数据（参见 3.1），因为它就是分别用 0 和 1 来表示真和假。显然条件的位置上应该放置一个布尔型的数据。然而，光靠死板的 0 和 1 两个数仍然无法描述可能发生着变化的各种情况。那么我们如何让电脑来根据实际情况做出判断呢？

这里我们要引入关系运算。之前的加减乘除和取余数之类的操作，结果都是整型或实型数据。而关系运算的结果则是布尔型数据，也就是说它们的结果只有两种——真或假。

所谓关系运算，是判断操作符两边数据的关系。这些关系一共有六种，分别是：等于、大于、小于、大于等于、小于等于、不等于。如下表所示：

关系	等于	大于	小于	大于等于	小于等于	不等于
操作符	==	>	<	>=	<=	!=
示例	a==b	a>b	a<b	a>=b	a<=b	a!=b

当操作符两边的数据符合操作符对应的关系时，运算结果为真，否则为假。比如 5>1 的结果是 1（真），'a'=='a' 的结果也是 1（真）；而 3<=2 的结果为 0（假）等等。**特别要注**

意，`==`和`=`是两个不同的操作符，前者是判断操作符两边数据的关系，后者是把右面的表达式的结果赋值给左边。

下面我们来看一段程序：（程序 4.1.1）

```
#include "iostream.h"
int main()
{
    int a,b;
    cout <<"请输入两个数： ";
    cin >>a >>b;
    if(a>b)//如果 a 比 b 大，则将两个数交换
    {
        int temp;//创建一个临时变量
        temp=a;
        a=b;
        b=temp;
    }
    cout <<a <<" " <<b <<endl;//将两个数从小到大输出
    return 0;
}
```

第一次运行结果：

请输入两个数： 1 5

1 5

第二次运行结果：

请输入两个数： 3 2

2 3

算法时间：交换

交换是程序设计中基础最常用的一种操作。它的算法在现实生活中也有着形象的操作。交换两个变量里的数据就好像交换 AB 两个碗中的水。我们必须再拿一个碗来（临时变量），将 A 碗里面的水先倒到这个临时的碗里，再将 B 碗的水倒到空的 A 碗里，最后把临时碗里的水再倒回 B 碗，那么就完成了这个工作。对照着这个过程去阅读代码是不是有些理解了呢？至于这个算法的代码，也是非常好记的。记住把临时变量放在首位，然后把任一变量放在等号的右边，下一句语句开头的必然也是这个变量。简单地记就是首尾相连。（程序 4.1.1 的代码中用相同的颜色表示出“首尾相连”。）

这个程序完成的工作是将两个无序的整数从小到大地输出。即如果第一个数比第二个数大，先交换再输出，否则直接输出。

试试看：

1、如果把程序 4.1 里，if 语句中条件表达式两边的括号去掉，程序是否能够正常运行？

结论：if 之后的条件表达式应该加上括号。

2、在双字符的关系操作符（如`&&`或`||`）中间加入空格，是否能够通过编译？

条件——逻辑运算

学校评三好学生，候选人必须要德智体全面发展才能够评上；学校开运动会，运动员只

要在某一个项目上是全校第一就能够获得冠军。现实生活中，有些条件会很严格，要数项同时满足时才算符合条件；而有些条件又会很松，只要符合其中某一项就算符合条件了。在程序设计中，我们也会遇到这样的问题。

平时，我们往往是用“并且”和“或”两个词来描述这些情况的。而在程序设计中，我们用逻辑运算来描述。我们平时称它们“与”（相当于并且）、“或”、“非”。“逻辑与”的操作符是&&，“逻辑或”的操作符是||，“非”的操作符是!。下面三个真值表说明了各逻辑运算的结果。

逻辑与	真	假	逻辑或	真	假
真	真	假	真	真	真
假	假	假	假	真	假

上面两表的第一行和第一列分别是逻辑操作符两侧的值，右下角带有灰色底纹的四格是经过运算后的结果。

非	真	假
结果	假	真

如果我们用集合 A 和集合 B 分别来描述两个不相同的条件 A 和 B，那么 A&&B 表示要满足集合 A 与集合 B 的交集；A||B 表示要满足集合 A 与集合 B 的并集；!A 表示要满足集合 A 的补集。

在上一章的 3.3 简单表达式中，我们提到了运算的次序。在程序设计中，我们把这种运算的次序称作操作符（Operator）的优先级。那么，关系操作符和逻辑操作符的优先级是怎么样的呢？

和简单表达式一样，括号的优先级仍然是最高的。无论什么情况都应该先从左到右地计算括号内的结果。当算术操作符、关系操作符和逻辑操作符处于同一级的括号中时，则分别从左向右地依次进行逻辑非运算、算术运算（遵循算术运算的优先级）、关系运算、逻辑与运算和逻辑或运算。（记作“不算关羽活”）

试试看：
 试人工计算下列条件表达式的结果，并上机验证结果。
 1 || 0 && 1 && 0
 ! 1+2>1
 0 && !2+5 || 1 && ! (2+!0)

下面我们来实践一下，看一段程序：（程序 4.1.2）

```
#include "iostream.h"
int main()
{
    int mark;
    cout <<"请输入成绩 (0~100): ";
    cin >>mark;
    if (mark>=80 && mark <=100) cout <<"Good!" <<endl;
    if (mark>=60 && mark <80) cout <<"So so" <<endl;
    if (mark>=0 && mark <60) cout <<"Please work harder!" <<endl;
    if (mark<0 || mark >100) cout <<"ERROR!" <<endl;
    return 0;
}
```

第一次运行结果：

请输入成绩 (0~100): 100

Good!

第二次运行结果:

请输入成绩 (0~100): 75

So so

第三次运行结果:

请输入成绩 (0~100): 59

Please work harder!

第四次运行结果:

请输入成绩 (0~100): 105

ERROR!

我们可以看到, 将关系运算和逻辑运算配合使用, 可以将数值有效地分段。以上这段程序的功能就是按照不同段的数值输出不同的结果, 如果输入的数值超出正常的取值范围, 则输出出错信息。

算法时间: 纠错

熟悉电脑软件的同学都知道, 不少软件或程序有时候会有漏洞 (Bug), 使得程序的安全性或稳定性受到影响。而产生这些漏洞的部分原因就是程序员在设计程序时有所疏漏, 忘记了去考虑一些可能引起错误的特殊情况。我们把那些可能引起程序异常的情况称为**临界情况**。比如在 a/b 中, $b=0$ 就是一种临界情况。如果不考虑到这种情况, 则可能导致除数为零而使整个程序崩溃。我们学会了 `if` 语句以后就能够从一定程度上避免一些可以预知的错误, 把那些临界情况引入纠错程序。(比如输出出错信息, 或及时中止程序)

&&和||的妙用

有时候我们做数学题目会遇到这样的问题—— $(1+5*8)/6*0/(5/6+2)$, 当我们发现整个式子是乘式, 并且有一个乘数为 0 的时候, 则会不再做更多的计算, 把结果脱口而出。因为无论后面的乘数是什么, 都无法改变结果了。

根据真值表我们知道, 在逻辑与中, 只要有一个假则整个表达式的结果为假; 在逻辑或中, 只要有一个是真则整个表达式的结果为真。我们发现逻辑与、逻辑或和上面所说的例子有着相似之处, 那么电脑会不会像我们一样, 不再做更多无所谓的计算呢?

答案是肯定的。即在一个或多个连续的逻辑与中, 一旦出现一个假, 则结果为假, 处于该位置以后的条件不再做更多判断; 在一个或多个连续的逻辑或中, 一旦出现一个真, 则结果为真, 处于该位置以后的条件也不再做更多判断。

比如: `if (m!=0 && n/m<1)`

```

{
    cout <<"OK" <<endl;
}

```

当 $m=0$ 时, 电脑不会去尝试用 n/m 了, 而是直接跳过整句语句。这样, 我们就能够避免除数为零的错误了。

试试看:

1、尝试多个复杂的逻辑运算, 摸索能够最准确地表达自己所希望的运算次序的方法。

结论: 使用括号。

2、修改程序 3.3, 使得输入的半径为非正数时输出出错信息。

4.2 否则……

平时我们在说“如果……，那么……”的时候，还经常和“否则……”连用。比如：如果明天天气好，就开运动会，否则就不开。按照我们上一节学的内容，我们只能这样说：如果明天天气好，就开运动会；如果明天天气不好，就不开运动会。虽然这样也能够把意思表达清楚，但是语句显得冗长，要是条件再多一些则更是杂乱。可见，在程序设计中，如果没有“否则……”语句将会多么麻烦。

和平时说话的习惯一样，“否则”应该与“如果”连用，其语法格式为：

```
if (条件)
    语句块 1;
else
    语句块 2;
```

运行到该语句时，当条件满足，则运行语句块 1 中的语句；当条件不满足，则运行语句块 2 中的语句。我们也可以流程图（图 4.2.1）来直观地表示 if……else……语句。和 if 语句一样，else 语句的结尾是没有分号的。

我们来看一段程序：（程序 4.2.1）

```
#include "iostream.h"
int main()
{
    int a,b,max;
    cout <<"请输入两个数: " <<endl;
    cin >>a >>b;
    if (a>=b)//如果 a 大于等于 b, 则把 a 的值放到 max 中
    {
        max=a;
    }
    else//否则把 b 的值放到 max 中
    {
        max=b;
    }
    cout <<"较大的数是" <<max <<endl;
    return 0;
}
```

第一次运行结果：

请输入两个数：

1 5

较大的数是 5

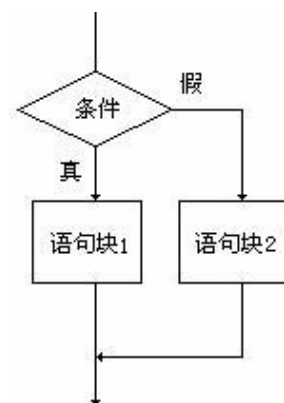
第二次运行结果：

请输入两个数：

5 8

较大的数是 8

通过以上程序，我们基本上可以了解 if……else……的使用了。



（图 4.2.1）

如果里的如果……

我们知道了，if 语句的主要功能是给程序提供一个分支。然而，有时候程序中仅仅多一个分支是远远不够的，甚至有时候程序的分支会很复杂，要在一个分支里面再有一个分支。根据 if 语句的流程图，我们不难想象如果要在分支里再形成分支，就应该在 if 语句中使用 if 语句。这类在一种语句的内部多次使用这种语句的现象叫做嵌套。

我们来看一段程序，熟悉一下 if 的嵌套。（程序 4.2.2）

```
#include "iostream.h"
int main()
{
    float a,b;
    char oper;//创建一个字符型变量用于存放操作符
    cout <<"请输入一个表达式 (eg.1+2): " <<endl;
    cin >>a >>oper >>b;//输入表达式，操作符处于中间
    if (oper=='+')//如果操作符是加号
    {
        cout <<a <<oper <<b <<'=' <<a+b <<endl;//输出两数的和
    }
    else//否则
    {
        if (oper=='-')//如果操作符是减号
        {
            cout <<a <<oper <<b <<'=' <<a-b <<endl;//输出两数的差
        }
        else//否则
        {
            if (oper=='*')//如果操作符是乘号
            {
                cout <<a <<oper <<b <<'=' <<a*b <<endl;//输出两数的积
            }
            else//否则
            {
                if (oper=='/' && b!=0)//如果操作符为除号且除数不为零
                {
                    cout <<a <<oper <<b <<'=' <<a/b <<endl;//输入两数的商
                }
                else//否则
                {
                    cout <<"出错啦！" <<endl;//操作符不正确或除数为零，输出错误信息
                }
            }
        }
    }
    return 0;
}
```

}

第一次运行结果:

请输入一个表达式 (eg.1+2):

1.5+3

1.5+3=4.5

第二次运行结果:

请输入一个表达式 (eg.1+2):

8/0

出错啦!

第三次运行结果:

请输入一个表达式 (eg.1+2):

5p3

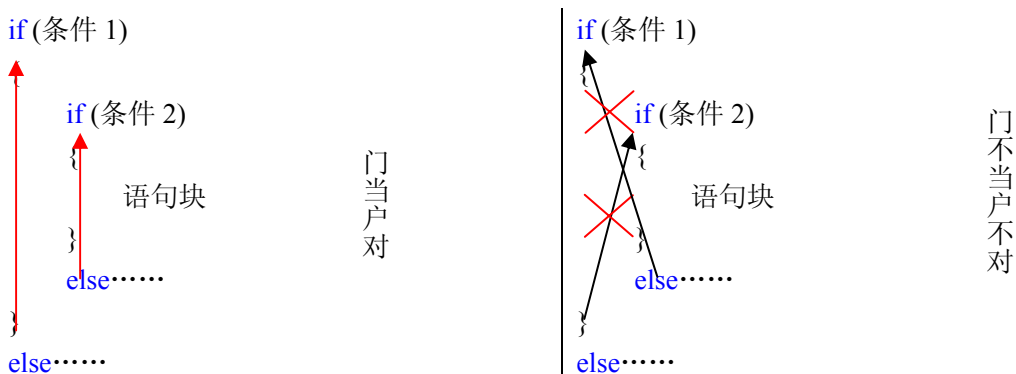
出错啦!

以上这段程序能够基本实现表达式的识别。它所使用的 if 嵌套能够分辨出到底要进行什么运算, 并且把引起错误的操作符或数据分支出来。

找朋友

当一个程序中出现多个 if……else……的时候, 也可能会引来一些麻烦的事情。因为每个 if 都具有和 else 配对的功能。那么我们在阅读一段程序的时候, 怎么才能够知道哪个 if 和哪个 else 是在一起的呢?

如果你尝试过在 VC++ 中输入程序 4.2.2, 那么你会发现, 每输入一次 {}, 括号内部的语句就会自动向右侧缩进一段。而 if……else……正是根据括号和缩进来判断它们是不是匹配的。具体的规则是, else 向上寻找最近的一个和它处于相同缩进位置的 if 配对, 我们把这种规则理解为“门当户对”。很显然, 如果你没有改变过自动产生的缩进位置, else 不会去找一个比它更右边或者更左边的 if 的。



在有些高级语言中, 是没有缩进的。缩进不仅是为了美观, 也是为了让程序的层次更加分明。我们通过缩进就能很容易看出一段代码应该从哪里开始, 运行到哪里结束。如果没有缩进的话, 就要去找保留字, 这给大型程序开发带来了麻烦。所以保持缩进是一种好习惯。

试试看:

当 if……else……语句括号中语句块只有一句语句的时候, 能否将大括号去除? 去除大括号以后是否影响运行结果? 如果去除大括号, 对你阅读程序带来了什么影响?

结论: 当语句块中只有一句语句时, 大括号可以去除。但是在阅读程序时可能会给 if……else……配对带来麻烦。

4.3 爱判断的问号

随着程序越来越复杂，会在代码中出现越来越多的 if 语句。有些时候我们只要电脑做一个简单的判断，就要用占据多行的 if 语句，实在有点吓人，使得程序的可读性受到一定的影响。比如程序 4.2.1 中，使用标准格式写一段将较大数放入 max 中的语句占据了八行。即使是较简便的写法，也至少要占据两行。那么，C++ 是否还提供了更为简便的书写方法呢？

答案是肯定的，我们可以用一个问号来判断一个条件，具体的语法格式为：

(条件表达式) ? (条件为真时的表达式) : (条件为假时的表达式)

“……? ……; ……”称为**条件操作符**，它的运算优先级比逻辑或还低，是目前为止**优先级最低的操作符**。含有条件操作符的表达式称为**条件表达式**。既然是表达式，它就应该有一个计算结果。而这个结果就是已经经过判断而得到的结果。我们可以定义一个变量来存放这个结果，也可以用输出语句把这个结果输出。但是，如果得到结果以后，既没有把它存放起来，也没把它输出来，那么做这个条件运算就失去意义了。

下面我们用条件操作符来改编一下程序 4.2.1，看看条件表达式是如何使用的：

```
#include "iostream.h"
int main()
{
    int a,b,max;
    cout <<"请输入两个数: " <<endl;
    cin >>a >>b;
    max=(a>=b)?a:b;//如果 a 大于等于 b，则把 a 的值放到 max 中，否则把 b 的值放到 max
    中
    cout <<"较大的数是" <<max <<endl;
    return 0;
}
```

运行的结果就如同程序 4.2.1，没有任何区别。而我们也达到了缩短代码的目的，增强了程序的可读性。

试试看：

- 1、仍然使用条件操作符修改程序 4.2.1，要求不能使用除了 a 和 b 以外的任何变量，保持程序的运行结果不变。
- 2、当条件为真假时的表达式都是左值（变量）的时候，能否把条件表达式放在等号的左边而被赋值？

4.4 切换的开关

我们已经了解，if……else……可以用来描述一个“二岔路口”，我们只能选择其中一条路来继续走。然而，有时候我们会遇到一些“多岔路口”的情况，用 if……else……语句来描述这种多岔路口会显得非常麻烦，而且容易把思路搅浑。比如程序 4.2.2 就是一个用 if……else……语句描述的四岔路口（四种操作符），整个程序占据了将近一页。

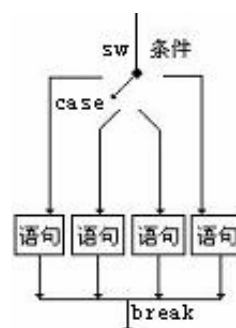
如果我们把这些多岔路看作电路，那么用 if……else……这种“普通双向开关”来选择某一条支路就需要设计一套很复杂的选路器。所以最简便的选路方法当然是做一个像下图那

样的开关。(图 4.4.1)

在 C++中, 也有这样的开关, 那就是 `switch` 语句。它能够很简捷地描述出多岔路口的情况。具体的语法格式为:

switch (表达式)

```
{
  case 常量表达式1:
  {
    语句块1;
    break;
  }
  .....
  case 常量表达式n:
  {
    语句块n;
    break;
  }
  default:
  {
    语句块n+1;
  }
}
```



(图 4.4.1)

在 `switch` 语句中, 我们要记住四个关键词, 分别是 `switch`、`case`、`default` 和 `break`。`switch` 是语句的特征标志 (图中标作 `sw`); `case` 表示当 `switch` 后的表达式满足某个 `case` 后的常量时, 运行该 `case` 以后的语句块。**要注意, 任意两个 `case` 后的常量不能相等, 否则 `switch` 将不知道选择哪条路走。**`default` 表示当表达式没有匹配的 `case` 时, 默认 (`default`) 地运行它之后的语句块 (图 4.4.1 中未标出); `break` 表示分岔路已经到头, 退出 `switch` 语句。

下面, 我们就来用 `switch` 语句来改写程序 4.2.2。箭头表明遇到 `break` 以后的运行情况。

```
#include "iostream.h"
int main()
{
  float a,b;
  char oper;
  cout <<"请输入一个表达式 (eg.1+2): " <<endl;
  cin >>a >>oper >>b;
  switch (oper)
  {
  case '+':
  {
    cout <<a <<oper <<b <<'=' <<a+b <<endl;
    break;
  }
  case '-':
  {
    cout <<a <<oper <<b <<'=' <<a-b <<endl;
```

```

        break;
    }
    case '*':
    {
        cout <<a <<oper <<b <<!=' <<a*b <<endl;
        break;
    }
    case '/':
    {
        if (b!=0) cout <<a <<oper <<b <<!=' <<a/b <<endl;
        else cout <<"出错啦！" <<endl;
        break;
    }
    default:
        cout <<"出错啦！" <<endl;
}
return 0;
}

```

上述程序的运行结果和程序 4.2.2 的运行结果一样。我们发现使用了 switch 语句以后，代码的平均缩进程度有所减少，阅读代码的时候更简洁易懂。所以，使用 switch 语句来描述这种多分支情况是很合适的。

试试看：

1、如果去除了 case 对应的 break，则运行出来会是什么结果？

结论：如果去除了 break，则不会退出 switch 而运行到别的支路里去。

2、如果程序 4.2.2 的 default 没有处在 switch 的结尾部分，那么运行出来会是什么结果？

结论：switch 语句中最后一个分支的 break 可以省略，其它的 break 均不可以。

3、case 后的常量能否是一个浮点型常量或双精度型常量？

巧用 switch

返回去看一下程序 4.1.2，我们不难发现这个程序也是一个多分支结构。可是 switch 语句只能判断表达式是否等于某个值，而不能判断它是否处于某个范围。而要把处于某个范围中的每个值都作为一句 case 以后的常量，显然也太麻烦了。那么我们还能不能使用 switch 语句来描述这种范围型的多分支结构呢？

通过分析，我们发现了主要起区分作用的并不是个位上的数，而是十位上的数。如果我们能把十位上的数取出来，那么最多也就只有十个分支了，不是吗？下面我们就来看一下用 switch 语句改编的程序 4.1.2。

```
#include "iostream.h"
```

```
int main()
```

```
{
```

```
    int mark;
```

```
    cout <<"请输入成绩 (0~100): ";
```

```
    cin >>mark;
```



```

switch(mark/20)
{
case 5:
    {
        if(mark>100)//100 到 119 的情况都是 mark/20==5, 所以要用 if 语句再次过滤
        {
            cout <<"ERROR!" <<endl;
            break;
        }
    }
case 4:
    {
        cout <<"Good!" <<endl;
        break;
    }
case 3:
    {
        cout <<"Soso" <<endl;
        break;
    }
case 2://根据前面试一试的结论, 如果 case 没有对应的 break, 会运行到下一个 case 中
case 1:
case 0:
    {
        if(mark>=0)//同样要用 if 过滤负数
        {
            cout <<"Please work harder!" <<endl;
            break;
        }
    }
default://其它情况都是出错
    cout <<"ERROR!" <<endl;
}
return 0;
}

```

这个程序要比原来的程序 4.1.2 冗长一些。但是这里提到这个程序的目的是要教会大家一种使用 switch 的方法，即“以点盖面”。

算法时间：数据的转换

在程序设计中，我们经常会遇到这样的问题：我们希望处理的数据和电脑能够处理的数据可能有所不符。不符合的情况一般有两种，一种是范围不符合，另一种是类型不符合。对于范围不符合，我们一般考虑的是使用代数式对数据进行处理。比如 C++ 中的随机函数能够产生一个 0~32768 之间的一个整数，如果我们希望得到一个 0~10 之间的随机数，那么就用它对 10 取余数，那么结果一定就在这个范围内。对于类型不符合，我们只好尽量用已有的数据类型来描述这种难以表达的类型。就如同电脑中用 0 和 1 表示真和假一样。

习题

1、把下列自然语言描述的条件转化为逻辑运算和关系运算描述的条件，每条至少写出两种。

①变量 a 不小于变量 b

②变量 a 加上变量 b 之后再乘以变量 c 结果不为零

③当变量 a 等于 1 的时候，变量 b 大于 1；当变量 a 不等于 1 的时候，变量 c 大于 1

④当变量 b 不等于 0 的时候，变量 a 除以变量 b 大于 3

⑤变量 b 大于等于变量 a 并且变量 b 小于变量 a

2、写出下列程序的运算结果，并写出语句执行的先后次序（故意消除缩进）。

①

```
#include "iostream.h"
int main()
{
    int a=8,b=4,c=2,k=4,m=8,n=6;
    cout <<a <<b <<c <<k <<m <<n <<endl;
    if (a!=b || m!=a+b)
    {
        a=2*k!=!m;
        a=a+a;
    }
    if (a+b>=0 && m/3.0>2)
    {
        m=k+3*c;
    }
    else
    {
        k=k*!m!=c;
    }
    cout<<a <<m <<k <<endl;
    return 0;
}
```

②

```
#include "iostream.h"
int main()
{
    int a=0,b=1,c=2;
    switch (a)
    {
        case 0:
            cout <<b+c <<endl;
        case 1:
            {
                a=a+b*c;
                switch (a)
            }
    }
}
```

```
        {
        case 2:
            cout <<b+c <<endl;
        case 5:
            cout <<(a=a+b*c) <<endl;
        default:
            c=c*2;
            break;
        }
    }
    default:
        cout <<a+b+c <<endl;
        break;
    }
    return 0;
}
```

3、指出下列程序的错误之处。

①

```
#include "iostream"
int program()
{
    int a,b,c,temp;
    cout <<'请输入三个数' <<endl;
    cin >>a,b,c;
    if b<c;
    {
        b=temp;
        temp=c;
        c=b;
    }
    if a<c;
    {
        temp=a;
        a=c;
        c=temp;
    }
    if a<b;
    {
        temp=b;
        b=a;
        a=temp;
    }
    cout <<'从大到小排列的顺序为: ' <<a,b,c <<endl;
    return 0;
}
```

}

②

```

#include "iostream.h"
int main()
{
    int a,b,c;
    cin >>a >>b >>c;
    switch a;
    {
        case 0
            cout <<b+c <<endl;
        case b+c
            cout <<a+b+c <<endl;
            break;
        default
            cout <<a+b <<endl;
    }
    return 0;
}

```

4、根据运行结果完善代码。

```

#include "iostream.h"
int main()
{
    int a;
    cout <<"请输入一个数: ";
    cin >>a;
    if (_____) cout <<a <<"是个负数。" <<endl;
    _____ cout <<a <<"是个正数。" <<endl;
    if (_____) cout <<a <<"是个奇数。" <<endl;
    _____ cout <<a <<"是个偶数。" <<endl;
    cout <<a <<"个位上的数是" <<(_____) <<endl;
    return 0;
}

```

第一次运行结果:

请输入一个数: 52

52 是个正数。

52 是个偶数。

52 个位上的数是 2

第二次运行结果:

请输入一个数: 0

0 是个偶数。

0 个位上的数是 0

第三次运行结果:

请输入一个数: -15

-15 是个负数。

-15 是个奇数。

-15 个位上的数是 5

5、根据书中例题改写程序。

①使用 `if……else……` 语句和 `switch` 语句设计一个程序，使其可以识别有两个操作符（加减乘除）的表达式。要注意操作符有优先级。运行时输入输出情况如下：

请输入一个表达式 (eg. `1+2*3`):

`1+2*3`

`1+2*3=7`

提示：在预处理头文件处写上 `#include "stdlib.h"`，可以通过 `exit(1);` 语句直接退出程序。

②现在有一个四位数（0000~9999），请设计一个程序，将其千位、百位、十位、个位的数分别输出。运行时输入输出情况如下：

请输入一个四位数（0000~9999）：`4248`

4248 的千位数是 4，4248 的百位数是 2，4248 的十位数是 4，4248 的个位数是 8。

③使用分支语句设计一个程序，要求输入年月日以后，算出这天是这一年的第几天。运行时输入输出情况如下：

请输入日期 (eg. `2005 2 28`): `2005 2 28`

2005 年 2 月 28 日是 2005 年的第 59 天。

提示：注意闰年，四年一闰、百年不闰、四百年又一闰。使用 `switch` 语句的时候要注意其 `break` 特性，尽量设计出较简洁的代码。

第五章 有个圈儿的程序

电脑之所以能够帮助人类解决各种各样的问题，除了它运算的准确性以外，最重要的就是它非常勤劳，可以反复进行类似或相同的运算而不觉得厌烦。在这一章，我们就主要介绍如何利用程序设计，从复杂却又单调的工作中解脱出来，把那些烦心事都丢给电脑去处理。

5.1 程序赛车

大家看过赛车的话都知道，赛车就是围绕着一个固定的跑道跑一定数量的圈数，如果没有发生意外，那么跑完了指定数量的圈数，比赛就结束了。

我们来设想一下赛车的实际情况，当比赛开始，赛车越出起跑线的时候，车子跑了 0 圈，然后车子开到赛道的某个地方，会看到车迷举着一块标牌。过一会儿，赛车跑完了一圈，这时候已跑圈数还没有达到比赛指定的圈数，所以比赛还要继续，车子还要继续跑……开到刚才那个地方，又看到一次车迷举的标牌……当赛车跑完第 60 圈，也就是最后一圈时，已跑圈数已经等于比赛所要求的圈数，比赛就结束了。

问车手一共看到了几次粉丝举的标牌呢？很显然，答案是 60 次。

如果我们把粉丝的标牌换成了语句 `cout <<"加油!" <<endl;`，那么很显然，屏幕上应该会显示 60 次“加油！”。于是我们有了重复多次输出字符串的基本想法。可是，我们现在还缺少赛车呢，在 C++ 中，是如何造出一辆赛车来的呢？

赛车里最有名的是 Formular 1（一级方程式赛车），于是我们取 Formular 的前三个字母 for 作为造赛车的语句，其具体语法格式为：

for（比赛前的准备；比赛继续的条件；每跑一圈后参数的变化）

语句块；

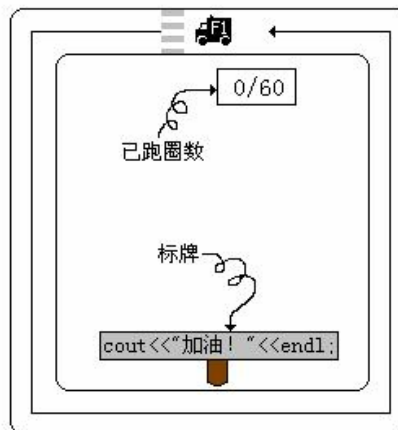
for 语句称为循环语句，大括号内的语句块称为循环体，而这种赛车的结构在 C++ 中称为循环结构。根据上面的语法格式，我们来描述一下前面所说的输出 60 次“加油！”的情况：

```
for (int i=0;i<60;i=i+1)
{
    cout <<"加油!" <<endl;
}
```

我们在比赛开始前，创建一个整型变量 `i` 用于存放赛车已跑的圈数，并且为它赋初值为 0，即比赛开始前已跑了 0 圈。比赛继续的条件是赛车还没跑到 60 圈，即当 `i<=60` 的时候，比赛应该立即中止。（设想如果将此处改成 `i<=60`，赛车实际要跑几圈？）每跑完一圈以后，已跑圈数要增加 1，所以 `i=i+1`。而语句块中的内容相当于在跑道中看到的各种情况……（参见图 5.1.1）

下面我们来看一个完整的 for 语句构成的程序：（程序 5.1.1）

```
#include "iostream.h"
int main()
```



（图 5.1.1）

```

{
    int sum=0;
    for (int i=1;i<=100;i=i+1)
    {
        sum=sum+i;
    }
    cout <<sum <<endl;
    return 0;
}

```

运行结果：

5050

我们在循环之前，创建了两个变量，分别为 sum 和 i。在循环语句中，我们习惯用诸如 i, j, k 之类的字母作为变量名，来控制循环的执行次数。这些变量又称为循环控制变量。而 sum 则表示和的意思，其作用是把一点一点的数值累加起来。我们来通过手工来模拟一下程序 5.1.1 的前三次循环：

创建变量 sum=0→遇到 for 语句，创建变量 i=1→判断 i 是否小于等于 100→满足 (i=1<100)，于是执行循环→sum=sum+i，即 sum=0+i=1→第一次循环完毕，i=i+1，即 i=1+1=2→判断 i 是否小于等于 100→满足 (i=2<100)，于是执行循环→sum=sum+i，即 sum=1+i=3→第二次循环完毕，i=i+1，即 i=2+1=3→判断 i 是否小于等于 100→满足 (i=3<100)，于是执行循环→sum=sum+i，即 sum=3+i=6→第三次循环完毕，i=i+1，即 i=3+1=4……

通过三次循环，我们不难发现 sum 里存放的是 1+2+3……的和。所以，循环 100 次以后输出了结果 5050 也在意料之中了。

算法时间：累加与循环控制变量

在循环结构中，累加是很常用的一种方法。累加分两种：常量累加和变量累加。常量累加就是类似 i=i+1，即在自身的数值上每次递增 1。这种方法一般用来记数，然后利用这个计数器作为条件帮助循环语句或分支语句做一些判断。变量累加一般是用于保存结果的，不管是 1+2+3……+100 还是 1*2+2*3+3*4……+99*100 都需要用到变量累加。变量累加一般和循环控制变量是有关系的，比如程序 5.1.1 中的累加值就是循环控制变量 i，而 1*2+2*3……中的累加值就是 i*(i+1)了。

加加和减减

我们发现，在 for 语句中，会经常用到 i=i+1 之类的语句。于是为了方便表示，C++ 中有了增量表达式和减量表达式。增量的操作符为 ++，减量的操作符为 --。增减量运算的优先级和逻辑非运算处在同一级。所以我们现在可以记作“不曾（增）算关羽活”，但是要注意，逻辑非和增减量操作符应该依次从左向右计算，而没有谁优先的说法（因为它们同级）。

在实际使用中，我们会遇到两种增（减）量操作。一种是 ++i，称为前增量操作，另一种是 i++，称为后增量操作。那么这两种操作有什么不同呢？应该如何记忆呢？

我们刚才说了，增量和减量是表达式，既然是表达式就应该有一个结果。而前增量和后增量的结果是不同的。++i 是先去做 i=i+1，然后再把 i 作为表达式的结果；而 i++ 是先把 i 作为表达式的结果，然后再去做 i=i+1。说到这里，可能有些读者要糊涂了，要是 i=1，执行完了 i++ 和 ++i 的结果不都是 i=2 么？怎么叫结果不一样呢？那么我们来看段程序：（程序 5.1.2）

```
#include "iostream.h"
```

```
int main()
{
    int a,i=1;
    a=i++;
    int b,j=1;
    b=++j;
    cout <<a <<' ' <<b <<' ' <<i <<' ' <<j <<endl;
    return 0;
}
```

运行结果：

1 2 2 2

我们发现，当 i 和 j 同时为 1 分别执行前后自增以后， i 和 j 的值都由 1 变成了 2。但是 a 和 b 的值却是不同的。不同的原因就是在于前面讲的赋值与做加法的顺序不同。 $a=i++$ 是先把没有做过加法的 i 值赋给了 a ，所以 a 的值为 1；而 $b=++i$ 是先做加法，即 $i=2$ 了以后，再把 i 的值赋给 b ，所以此时 b 的值为 2。我们记忆的时候可以按照增量操作符和变量的位置来记，加号在变量前面的称为“先加后赋”，即先做加法在赋值；加号在变量后面的称为“先赋后加”，即先赋值后做加法。由于增减量操作符有赋值操作，所以操作的对象（又称操作数，Operand）必须是左值。比如 $3++$ 就是不允许的。

试试看：

1、修改程序 5.1.1，使之输出以下结果：

① $1+2+3+\dots+50$ ② $1*2*3+\dots*20$ ③ $1/1+1/2+1/3+\dots+1/50$

2、分别使用增量操作和减量操作修改程序 5.1.1，使其运行结果不变。并考察使用前增量和使用后增量是否影响循环程序的运行结果。

for 语句的巧用

我们知道在 for 语句括号内的语句一共有三条，分别是循环前准备、循环继续的条件和每次循环后参数变化。那么这三条内容是不是必需的呢？如果缺少某一句的话，for 语句还能否正常运行呢？

首先要了解，如果省略了某句语句，分号仍然是不能省略的。这里的分号起着分割的作用，如果省略了分号，那么电脑将无法判断到底是省略了哪句语句。

情况一：省略循环前准备

我们以程序 5.1.1 为例，在保证运行结果不变的情况下，可以做这样的修改：

```
#include "iostream.h"
int main()
{
    int sum=0;
    int i=1;//创建循环控制变量，并赋初值为 1
    for (;i<=100;i=i+1)
    {
        sum=sum+i;
    }
    cout <<sum <<endl;
}
```



```

    return 0;
}

```

实际上，我们并不是没有做准备工作，而是早就把准备工作在 for 语句之前就做好了。因此 for 括号内的准备工作就可以省略了。

情况二：省略循环继续的条件

事实上，循环继续的条件也是能够被省略的，但是却不推荐那样做。因为这将使得程序的可读性变差（即不容易让自己或别人看懂），程序的运行变得混乱。**如果循环继续的条件被省略，那么 for 语句就会认为循环始终继续，直到用其他方式将 for 语句的循环打断。**至于如何打断 for 循环我们将在下一节作介绍。

情况三：省略每次循环后的参数变化

我们知道，循环后的参数变化是等到每次循环结束以后才发生的。因此，我们把参数变化放在语句块的最后即可。如下是省略了参数变化的程序 5.1.1:

```

#include "iostream.h"
int main()
{
    int sum=0;
    for (int i=1;i<=100;)//省略参数变化
    {
        sum=sum+i;
        i++;//在语句块最后补上参数的变化
    }
    cout <<sum <<endl;
    return 0;
}

```

虽然省略 for 语句中的成分是允许的，但是在实际使用过程中这种方法却显得比较鸡肋。所以建议不要随意地将 for 语句的成分省略掉，以免给理解程序带来麻烦。

试试看：

1、试输出以下图形：

```

*****
*****
*****

```

2、改写程序 5.1.1，要求只改写 for 语句括号内一处，使其输出 1+3+5……+99 的结果。

5.2 进维修站和退出比赛

不知道大家有没有注意到，在上一节讲述赛车问题的时候有这样一句话：**如果没有发生意外的话**，那么跑完了指定数量的圈数，比赛就结束了。实际上，赛车比赛是会发生各种情况的，比如要进维修站进行维修，或者引擎突然损坏不得不退出比赛。那么 C++的“赛车比赛”会不会进维修站或者退出比赛了呢？

上一节向大家介绍了 for 可以省略循环继续的条件而使其不断循环，但如果我们放任这种无止境的循环，则可能会导致电脑死机。所以我们必须强制停止比赛。这条语句就是 break 语句，其实我们在 4.4 的 switch 语句中已经遇到过了。下面我们还是在程序 5.1.1 的基础上

作修改，看看 `break` 在 `for` 语句中是如何使用的。

```
#include "iostream.h"
int main()
{
    int sum=0;
    for (int i=1;;i++)
    {
        if (i>100) //若 i 大于 100 则退出循环
        {
            break;
        }
        sum=sum+i;
    }
    cout <<sum <<endl;
    return 0;
}
```

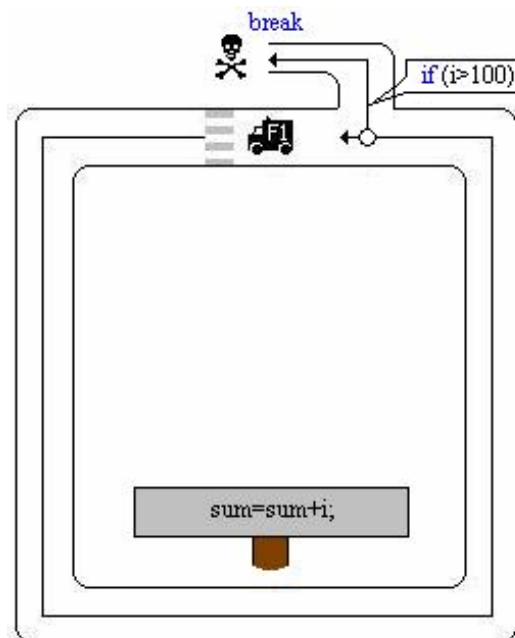
这段代码的意思是，当 $i \leq 100$ 的时候一直执行循环；一旦 $i > 100$ 了，则会运行到 `if` 语句里的 `break` 语句，于是强行中止了循环。以上这段代码可以由图 5.2.1 来表示。我们也不难发现，修改后的程序运行结果应该和程序 5.1.1 的运行结果一样。

那么，进维修站又是怎么回事呢？

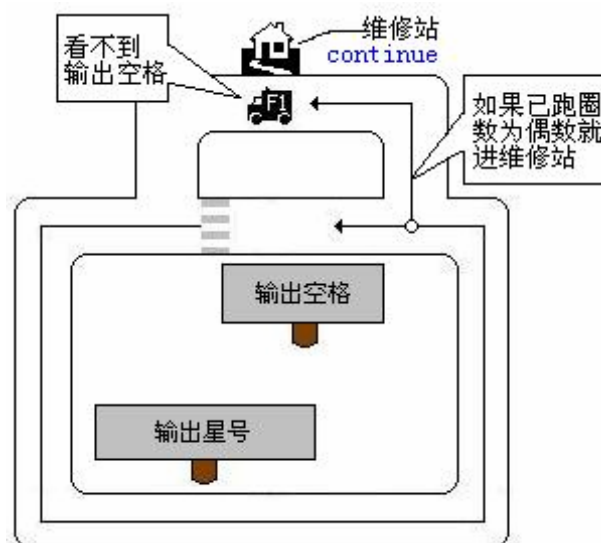
实际上进维修站并不是退出比赛，而是暂时绕开一段，然后重新进入赛道继续下一圈的比赛。那么绕开的赛道上的标牌是无法看到的。在 C++ 的“赛车比赛”中，进维修站是绕开一些语句，重新开始下一次的循环。**进维修站的语句是 `continue`**，下面我们来看一个程序：（程序 5.2.1）

```
#include "iostream.h"
int main()
{
    for (int i=0;i<12;i++)
    {
        cout <<'*';//输出星号
        if (i%2==0)
        {
            continue;
        }
        cout <<' ';//输出空格
    }
    cout <<endl;
    return 0;
}
```

运行结果：



(图 5.2.1)



(图 5.2.2)

*** ** ** ** **

在循环的执行过程中，如果 $i\%2$ 不等于 0，即 i 为奇数的时候，则完成整个循环，输出一个星号和一个空格；如果 i 是个偶数，则跳过输出空格的语句，进行下一次循环。这个程序的运行情况可以由图 5.2.2 来描述。

试试看：

- 1、改写程序 5.1.1，要求使用 `continue` 语句，使其输出 $1+3+5+\dots+99$ 的结果。
- 2、思考 `break` 和 `continue` 语句是否可能会影响循环的次数？为什么？

结论：break 可能影响循环次数，而 continue 不会影响。

5.3 圈圈里的圈圈

在上一章，我们讲到“如果里的如果”，是利用 `if……else……` 语句的嵌套来描述多分支的情况。那么圈圈里的圈圈——`for` 语句的嵌套又是怎么样的一种情况呢？

下面先让我们来看一个程序：（程序 5.3.1）

```
#include "iostream.h"
```

```
int main()
```

```
{
```

```
    int number;
```

```
    for (int i=0;i<=3;i++)
```

```
    {
```

```
        for (int j=0;j<=9;j++)
```

```
        {
```

```
            number=i*10+j;
```

```
            cout <<number <<' ';
```

```
        }
```

```
        cout <<endl;
```

```
    }
```

```
    return 0;
```

```
}
```

运行结果：

```
0 1 2 3 4 5 6 7 8 9
```

```
10 11 12 13 14 15 16 17 18 19
```

```
20 21 22 23 24 25 26 27 28 29
```

```
30 31 32 33 34 35 36 37 38 39
```

我们把最先遇到的循环语句称为外循环，后遇到的循环语句称为内循环。根据运行结果，我们知道这段程序能够输出 0~39 这一些整数。虽然使用一个 `for` 语句也能够做到这个效果，但是它们的原理是不同的。下面我们就来分析一下这两个 `for` 是如何做到输出这些数字的。

创建变量 `number`→遇到第一个 `for` 语句，创建变量 `i=0`，进行循环→遇到第二个 `for` 语句，创建变量 `j=0`，进行循环→`number=0*10+0=0`→输出 0→继续第二个 `for` 语句的循环，`j++`，`number=0*10+1=1`→输出 1→……输出 9→第二个 `for` 语句的循环结束，输出换行，`i++`→`i=1`，小于 3，第一个 `for` 语句的循环继续→再次遇到第二个 `for` 语句，`j=0`→`number=1*10+0=10`→

输出 10→继续第二个 for 语句的循环, j++, number=1*10+1=11→输出 11→……输出 19……

如果你还对 for 语句嵌套的运行方法不能理解, 那么我们可以找出一个生活中的例子。我们都知道, 时钟的运行方式: 分针走完一圈, 时针走一大格, 分针走完 12 圈, 时针才走完一圈。在 for 语句的嵌套中, 内循环就像分针, 而外循环就像是时针, 它走得很慢, 要等到内循环走完一圈它才走一格。

算法时间: 什么时候要用循环的嵌套?

循环的嵌套往往是在由多样东西通过不同搭配而组成一样东西的情况下。比如由一个个位数和一个十位数组成一个两位数就要用到循环的嵌套, 输出处在 x 轴和 y 轴不同位置的点组成的二维图形也要用到循环的嵌套。

试试看:

- 1、改写程序 5.3.1, 使用 for 语句的嵌套输出数字 0~999。
- 2、如果有一个双层的 for 语句嵌套, 完成一次内循环语句块要执行 n 次, 完成外循环语句块需要执行 m 次, 那么完成整个嵌套的 for 语句, 内循环的语句块一共要执行几次?

怎么让输出的东西更好看?

看了程序 5.3.1 的运行结果, 你可能会觉得输出的数字不太整齐。第一行的一位数都挤在了一起, 而第二行开始的两位数都是整整齐齐的。那么, 我们有什么办法让他们排排整齐么? 大家自然就先想到空格了。不过如果为了个这么简单的功能, 还要去编写一段判断一下这个数是几位的, 要加几个空格之类代码就有点麻烦了。其实 C++ 早已经为我们准备好了更方便的方法。这种方法就是设置域宽。

所谓域宽, 就是输出的内容(数值或字符等等)需要占据多少个字符的位置, 如果位置有空余则会自动补足。比如我们要设置域宽为 2, 那么当输出一位数 1 的时候输出的就是“ 1”, 即在 1 前面加了一个空格。空格和数字 1 正好一共占用了两个字符的位置。

那有些时候我们不想在 1 前面补上空格, 而是希望 1 前面补上 0 可不可以? 当然也是可以的。我们可以设置填充字符, 如果我们将 0 设置为填充字符, 那么 1 前面就变成 0 了。

设置域宽的具体语法格式为:

```
cout <<setw(int n) <<被设置的输出内容1 [<<setw(int m) <<被设置的输出内容2 …];
```

设置填充字符的具体语法格式为:

```
cout <<setfill(char n) <<被设置的输出内容1 [<<setfill(char m) <<被设置的输出内容2 …];
```

我们在设置域宽和填充字符的时候要注意几点: ①设置域宽的时候应该填入整数, 设置填充字符的时候应该填入字符。②我们可以对一个要输出的内容同时设置域宽和填充字符, 但是设置好的属性仅对下一个输出的内容有效, 对以后输出要再次设置。即 cout <<setw(2) <<a <<b; 语句中域宽设置仅对 a 有效, 对 b 无效。③setw 和 setfill 被称为输出控制符, 使用时需要在程序开头写上 #include "iomanip.h", 否则无法使用。

下面我们来看一段有关输出图形的循环嵌套程序: (程序 5.3.2)

```
#include "iostream.h"
#include "iomanip.h"
int main()
{
    int a,b;
    cout <<"请输入长方形的长和宽: " <<endl;
```

```

cin >>a >>b;
for (int i=1;i<=b;i++)//控制长方形的宽度
{
    for (int j=1;j<=a;j++)//控制长方形的长度
    {
        cout <<setw(2) <<'*';
    }
    cout <<endl;
}
return 0;
}

```

运行结果:

请输入长方形的长和宽:

5 3

```

* * * * *
* * * * *
* * * * *

```

试试看:

1、请尝试输出一个平行四边形。提示：在每行的开头设置域宽，域宽是表达式。

2、修改程序 5.3.1，使用输出控制符，使得输出格式如下：

```
00 01 02 03 04 05 06 07 08 09
```

```
.....
```

5.4 当……

我们已经学习了 for 语句的循环，并且知道 for 语句习惯上是用在已知循环次数的情况下的。但是，人不具有先知的能力，有些时候我们无法预知一个循环要进行几次，那我们该怎么办呢？

一个循环，最不可缺少的就是开始和终止。如果一个程序的循环只有开始没有终止，那么这个程序是不会有结果的。所以，我们必须知道什么时候让循环终止，即循环继续或循环终止的条件。

于是，一个只包含循环继续条件的循环语句产生了，那就是 while 语句，具体语法格式为：

while (循环继续的条件)

语句块;

while 语句要比 for 语句简练很多，它只负责判断循环是否继续。所以，我们必须人为地在语句块中改变参数，使得循环最终能够被终止。由于 while 循环是在循环语句块之前判断是否继续循环，所以又被称为“当型循环”。

下面让我们来看一段简单的程序：（程序 5.4.1）

```

#include "iostream.h"
#include "iomanip.h"
int main()
{

```

```

int password;
cout <<"请设置一个四位数密码（首位不能是 0）：" <<endl;
cin >>password;
int i=0;
while (i!=password)//如果密码没猜中就继续猜
{
    i++;
}
cout <<"破解成功！密码是" <<i <<endl;
return 0;
}

```

运行结果：

请设置一个四位数密码（首位不能是 0）：

1258

破解成功！密码是 1258

可能有些读者还没看懂，上面这段程序到底是什么意思。其实上面这段程序就是暴力破解密码的基本原理。假设某台电脑内设置了一个四位整数的密码，我们就可以通过循环语句让它不断地去尝试猜测，但是我们无法预知这个密码是多少，也就无法知道循环里的语句块要执行多少次，所以我们应该使用 `while` 循环，而循环继续的条件就是密码没有被猜中。

算法时间：电脑的猜测

很多人认为，电脑没有思维，怎么能猜测呢？其实这样就大错特错了。电脑自己是无法猜测的，但是我们可以使用循环语句教它如何猜测，更确切地说是教它如何找到。**这种使用循环来查找结果的方法我们称为穷举法。**即把所有可能的结果都去试试看，如果哪个能对上号了，就是我们所要的答案。但是在使用它的时候我们要注意严密性，如果自己考虑时漏掉了可能的结果，那么电脑自然不会猜出完美的答案来。穷举法在程序设计中使用时十分广泛，甚至很多人脑难以解决的问题，它都能很快地给出答案。

在实际使用中，我们发现 `while` 语句就像是只有循环条件的 `for` 语句。所以，在某些场合下，`while` 语句和 `for` 语句是可以互相转化的。而 `while` 语句也有着和 `for` 语句类似的嵌套，在这里不作赘述。

导火索——do

在实际生活中会有这样的问题，比如今天是星期一，我们以一周作为一个循环，那么循环结束的条件还是“今天是星期一”。如果我们写 `while (今天!=星期一)`，那么这个循环压根儿就不会运行。因为“今天是星期一”不符合循环继续的条件，已经直接使循环结束了。

其实我们只要让第一次的循环运行起来就是了，然后再写上 `while (今天!=星期一)`，就能达到我们的目的。如果我们把后面可以发生的循环比作能发生连锁反应的炸药，那么我们缺少的只是一根导火索。而在 C++ 中，就有这么一根导火索——`do`。它能够搭配 `while` 语句，使得第一次的循环一定能运行起来。它的语法格式是：

do

语句块；

while (循环继续的条件)；

要注意，这里的 `while` 后面是有一个分号的，如果缺少了这个分号，则会导致错误。下面就让我们来看一个 `do……while` 的程序：（程序 5.4.2）

```

#include "iostream.h"
int main()
{
    char inquiry;
    do
    {
        int n;
        cout <<"你要输出几个星号? " <<endl;
        cin >>n;
        for (int i=0;i<n;i++)//输出 n 个星号
        {
            cout <<'*';
        }
        cout <<endl;
        cout <<"还要再输出一行吗? (n 表示不要) " <<endl;
        cin >>inquiry;
    }while (inquiry!='n' && inquiry!='N');
    return 0;
}

```

运行结果:

你要输出几个星号?

3

还要再输出一行吗? (n 表示不要)

y

你要输出几个星号?

2

**

还要再输出一行吗? (n 表示不要)

y

你要输出几个星号?

1

*

还要再输出一行吗? (n 表示不要)

n

在这段程序中，由 for 语句来控制输出星号的个数。而 do...while 语句则是提供了一个用户交流的方式，一旦用户回答 n，则退出程序。

试试看:

1、根据程序 5.4.2，改写程序 4.2.2（或程序 4.4），要求在完成一个表达式的计算后，询问用户是否要进行下一次运算。

2、思考什么情况下 while 和 do...while 可以互换，什么情况下不可以互换？

结论：要求第一次循环必须运行起来的使用 do...while 语句；循环可能一次都不运行的只能用 while 语句。

算法时间：命令行下的人机交流

我们现在所使用的 Windows 系统称为图形用户界面（GUI——Graphic User Interface），它是一种可以由鼠标控制的直观的操作系统（OS——Operating System）。然而，在图形用户界面的操作系统被开发出来之前，我们只好在 DOS 环境下面对着冷冰冰的电脑，没有好看的图标，也没有方便的鼠标。这种在黑乎乎的屏幕上给电脑下命令的操作模式叫做命令行（Command Line）模式。很显然，这种模式给用户很不友好的感觉。所以，我们在设计一个完美的命令程序时，不仅要求它在功能上质量上的完美，还要求它能够提供更好的人机交流。而程序 5.4.2 中 do……while 语句的用法便是高级语言中简单而常用的提供人机交流的方法。

至此，我们学完了所有常用的的分支语句和循环语句。这些语句称为过程化语句。我们可以发现，除了 do……while 语句以外，所有的过程化语句的末尾是没有分号的，而分号都属于大括号内的语句或者语句块。

过程化语句是一个程序的骨骼。程序的大多数功能都要依赖过程化语句来实现。因此，掌握并且能够灵活运用过程化语句对程序设计来说非常重要。在以后的章节中，我们还会继续学习过程化语句一些更多的使用方法。

习题

1、计算下列表达式的值（表达式运行前 i=2, j=3）。

①(i++)+(++j)

②--j*5+(!j==!i)

③(++i==j)*++j

④!i++*8

2、阅读下列程序，分析程序的运行过程，并写出运行的结果。

①

```
#include "iostream.h"
int main()
{
    int sum=0;
    for (int i=1;i<=5;i++)
    {
        int tmp=1;
        for (int j=1;j<=i;j++)
        {
            tmp=tmp*j;
        }
        sum=sum+tmp;
    }
    cout <<sum <<endl;
    return 0;
}
```

②

```
#include "iostream.h"
```



```

int main()
{
    int sum=0;
    for (int i=0;i<=9;i++)
    {
        for (int j=0;j<=9;j++)
        {
            if (i!=j)
            {
                continue;
            }
            cout <<i <<j <<endl;
            sum++;
        }
    }
    cout <<sum <<endl;
    return 0;
}

```

3、指出下列程序的错误之处。

①要求输出 $(1+3) * (2+4) * (3+5) \dots * (8+10)$

```
#include "iostream.h"
```

```

int mian()
{
    int sum=0;
    for (int i=1;i<8;++i)
    {
        sum=sum*(2i+2);
    }
    cout <<sum <<endl;
    return 0;
}

```

②要求找出“水仙花数”。“水仙花数”是一个各位数字立方和等于该数本身的三位数。例如 $153=1^3+5^3+3^3$ ，所以它是一个水仙花数。

```
#include "iostream.h"
```

```

int main()
{
    for (int i=0;i<=9;i++)
    {
        for (int j=0;j<=9;j++)
        {
            for (int k=0;k<=9;k++)
            {
                if (i*i*i+j*j*j+k*k*k!=i*100+j*10+k)
                {

```

```

        break;
    }
    cout <<k <<j <<i <<endl;
}
}
}
return 0;
}

```

4、根据运行结果完善代码。

```

#include "iostream.h"
#include "stdlib.h"
#include "time.h"
int main()
{
    int a,b,numOfQues=0,numOfRight=0;
    char inquiry;
    srand(time(NULL));//用于产生随机数，不必理会
    cout <<"***欢迎你来做两位数的加减法***" <<endl;
    _____
    {
        int temp=rand()%2;//随机产生 1 或者 0 用于产生随机的加法或者减法
        a=rand()%100;//产生一个 100 以内的随机数
        b=rand()%100;//产生一个 100 以内的随机数
        switch (_____)
        {
            case 0:
                {
                    int ans;//用于存放答案
                    cout <<a <<'+ ' <<b <<!=';
                    cin >>ans;
                    if(_____)
                    {
                        _____++;
                        cout <<"恭喜！答对了！" <<endl;
                    }
                    else cout <<"答错了，再接再厉！" <<endl;
                    _____;
                }
            case 1:
                {
                    int ans; //用于存放答案
                    cout <<a <<'-' <<b <<!=';
                    cin >>ans;
                    if(_____)

```

```

        {
            numOfRight++;
            cout <<"恭喜! 答对了! " <<endl;
        }
        else cout <<"答错了, 再接再厉! " <<endl;
    }
}
_____++;
cout <<"你还要再做一题吗? (N 表示不要) " <<endl;
cin >>inquiry;
}while (_____);//要求大小写的 n 都能退出程序
cout <<"你的答题正确率为" <<_____ <<"%。再见! " <<endl;
return 0;
}

```

运行结果:

欢迎你来做两位数的加减法

81+32=113

恭喜! 答对了!

你还要再做一题吗? (N 表示不要)

y

20-2=18

恭喜! 答对了!

你还要再做一题吗? (N 表示不要)

y

51+34=99

答错了, 再接再厉!

你还要再做一题吗? (N 表示不要)

n

你的答题正确率为 66%。再见!

5、根据书中已有的代码改写程序。

①改写程序 5.3.1, 输出九九乘法口诀表, 显示效果如下:

1*1= 1

1*2= 2 2*2= 4

1*3= 3 2*3= 6 3*3= 9

1*4= 4 2*4= 8 3*4=12 4*4=16

1*5= 5 2*5=10 3*5=15 4*5=20 5*5=25

1*6= 6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36

1*7= 7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49

1*8= 8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64

1*9= 9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81

②根据第四题的改写代码, 要求能随机产生一位数的加减乘除法。要注意除法的除数不能为零, 如果两数除不尽则应该重新选题。在退出时给出评分, 根据不同的评分显示不同的话。比如正确率为 100%时显示“你真棒!”等等。

第六章 好用的工具

在前几章，我们已经学习了过程化语句。使用这些语句已经能够让我们完成一些工作。然而，仅仅靠这些语句是无法解决比较复杂的问题的。这时候，我们就需要有一些“工具”来帮助我们解决一些问题。本章我们将来介绍如何使用工具和如何自己动手造一个合手的工具。

6.1 简单的工具——函数

在日常生活中，我们会经常用到工具。比如我们画一个表格需要用到直尺，它可以告诉我们线段的长度；我们开瓶子的时候需要用开瓶器，它可以把瓶子打开；我们在做计算题的时候需要用到计算器，它能够告诉我们计算结果。

使用工具有些什么好处呢？首先，如果这个工具是现成的，我们就可以不必要自己去做一个这样的工具，直接拿来就能用（比如开瓶器、计算器）。其次，不管是现成的工具，还是自己做的工具（比如自己做的直尺），一定是能够多次反复使用的（比如直尺不是用完一次就不能再用的），而且是能够在各种合适的情况下使用的。（直尺在量程范围内能量这条线段的长度，也能够量那条线段的长度。）

在程序设计中，我们也会有各种各样的“工具”。你告诉比较大小的“工具”两个不相等的数，这个“工具”能够告诉你哪个数大；你告诉求正弦值的“工具”一个弧度，这个工具能够求出这个弧度对应的正弦值等等……**这些工具的名字就是函数 (Function)**。要注意，在程序设计中的函数和数学中的函数有相似的地方，但是它们却完全是两码事，请不要将两者等同起来。

函数和工具的性质是一样的。如果有一个现成求正弦值的函数，我们就不必自己去“造”一个这样的函数。求正弦值的函数是可以多次使用的，并且可以求出任意实数的正弦值（合适的情况下），但是它却求不出一个虚数的正弦值（不合适的情况下）。

工具的说明书

有时候我们会知道一个工具有什么功能，但是却因为对其陌生而不会使用，这时候要使用它可能会发生一些困难。除了自己有空去摸索一下以外，最有效的办法就是去看说明书了。说明书里会告诉你什么东西放在什么位置上，使用了以后会产生什么效果之类的。

同工具一样，**每个函数也有其自己的说明书，告诉用户如何调用（就是使用的意思）这个函数。这份说明书就称为这个函数的原型**。它的格式为：

产生结果类型 函数名(参数₁, 参数₂, ……参数_n);

函数名相当于工具的名字，比如直尺、计算器等等。产生效果类型相当于使用该工具产生的效果，比如直尺能够读出一个长度，计算器能够显示一个结果等等。而参数 (Parameter) 则是表示合适的使用情况，比如直尺应该去量长度而不能去量角度，计算器能计算数值而不能去画图等等。

那么我们如何来阅读函数的“说明书”呢？我们先来看两个例子：

(1) `int max(int a, int b);`

这个函数名称为 `max`，即求出最大的值。运行该函数以后，产生的结果是一个整数。在

数学中，我们会有一元函数比如 $f(x)=2*x+3$ ，也会有多元函数比如 $g(x,y)=x/4+y$ 等等。我们在使用 $f(x)$ 或 $g(x,y)$ 的时候括号内数的位置必须和自变量的字母对应，比如 $g(4,1)=4/4+1=2$ ，此时 $x=4$ 并且 $y=1$ 。我们既不能将其颠倒，也不能写出 $g(4)$ 或者 $g(4,2,1,5)$ 之类的表达式，否则就是错误的。程序设计中参数的作用和自变量 x, y 的作用是类似的。在函数“说明书”中，也交待了哪个位置应该放置什么类型的参数，我们在调用函数的时候要注意参数的类型、顺序、个数都要一一对应。

具体使用请看以下的程序：（程序 6.1.1）

```
#include "iostream.h"
int max(int a,int b);//函数原型，假设函数已经定义
int main()
{
    int r=3,s=5,t;
    t=max(r,s);//使用函数，并记录产生的结果
    cout<<t <<endl;
    return 0;
}
```

运行结果：

5

对于上面这段程序，有两点要说明。首先，调用函数时放入括号内的变量名 r 和 s 与函数原型里 a 和 b 的名字是可以不一样的。就像我可以用量尺各种各样的纸。但是，它们的数据类型必须相同，如果把一个字符型变量放在这个位置上，就如同用尺去量角度一样，无法成功的。其次，调用函数后的结果可以认为是一个表达式的值。我们可以把这个结果赋值给一个变量或者将其输出。当然，我们也可以不保存不输出这个结果，但是那样的话，就像是量了长度却没有把结果记录下来。

(2)`void output(char c);`

这个函数名为 `output`，即输出。`void` 表示空类型，它同整型、实型一样，也是一种数据类型。它表示调用该函数后，不会产生任何定量的结果。这是什么意思呢？我们知道，例如榔头这种工具，它只能产生一些效果，如把钉子砸进木头里，但是它不会给使用者一个定量的结果。不过我们大可不必担心它是否完成了我们要它完成的工作。如果榔头没把钉子砸进木头里，要么是榔头本身质量有问题，要么就是使用者没有按照要求去使用。若这把榔头不是用户自己造的，那么用户没有任何责任。

下面我们就来尝试一下使用这个函数：（程序 6.1.2）

```
#include "iostream.h"
void output(char c);//函数原型，假设函数已经定义
int main()
{
    char temp;
    cin >>temp;
    output(temp);
    return 0;
}
```

运行效果：

T

T

虽然函数没有产生什么定量的结果，但是其在屏幕上输出的功能还是达到了。对于产生 void（空类型）的函数，我们不必去保存结果了。

程序 6.1.1 和 6.1.2 的代码是不完善的，如果仅用这些代码去编译会被告知函数未定义。由于涉及更多的知识，这些代码将在下一节得到完善。

如何使用系统造好的工具

我们经常在程序的一开始就写 `#include` 某个头文件。其实有些头文件中就有不少系统已经造好的函数，它们叫做标准库（Standard Library）函数。我们包含（include）一个头文件，就像是到某个工具库里面去找一个工具一样。所以，要使用系统定义好的一些函数，我们必须知道这些函数在哪个头文件里，就好像使用工具我们必须知道这个工具放在哪个工具库里面。下面是一些函数和相关头文件信息的列举。

头文件	函数原型	备注
stdlib.h	<code>void exit(int a);</code>	退出程序，一般写作 <code>exit(1);</code>
math.h	<code>double sin(double a);</code>	计算正弦值，a 为弧度
	<code>double cos(double a);</code>	计算余弦值，a 为弧度
	<code>double tan(double a);</code>	计算正切值，a 为弧度
	<code>double sqrt(double a);</code>	计算平方根
	<code>double pow(double x,double y);</code>	计算 x 的 y 次方
	<code>int abs(int a);</code>	计算绝对值

其实很多函数系统已经为我们写好，我们只要通过包含头文件就能够使用这些函数。关于更多的函数信息，我们将在附录上作介绍，读者也可以通过网络或者 VC++ 的工具书来查找到这些信息。

下面我们来看一段使用系统造好的函数编写的程序：（程序 6.1.3）

```
#include "iostream.h"
#include "math.h"
#include "stdlib.h"
int main()
{
    const double pi=3.14159265358;
    double a=90;
    cout <<"sin(a)=" <<sin(a/360*2*pi) <<endl;//角度与弧度的转换
    cout <<"cos(a)=" <<cos(a/360*2*pi) <<endl;
    cout <<"sqrt(a)=" <<sqrt(a) <<endl;
    cout <<"pow(a,2)=" <<pow(a,2) <<endl;
    exit(1);
    return 0;
}
```

运行结果：

```
sin(a)=1
cos(a)=4.89659e-012
sqrt(a)=9.48683
pow(a,2)=8100
```

由于电脑的三角函数都是使用弧度作为单位的，所以我们必须用“`a/360*2*pi`”将角度

转化为弧度。至于为什么 $\cos 90^\circ$ 不等于 0，则是因为圆周率 π 无法很精确，所以导致算出来的余弦值是一个接近于 0 的小数，而不是 0。

试试看：

- 1、根据本节函数和头文件的信息表，尝试输出一个数（角度）的正切值、余切值和绝对值。
- 2、用 VC++ 打开 `stdlib.h` 和 `math.h`，看看里面究竟写了些什么。如果不知道文件存放的位置，请使用 Windows 的查找功能。

6.2 打造自己的工具

在上一节，我们已经学会了如何阅读函数原型和如何调用一个函数。然而，仅靠系统给出的标准库函数是不够用的。我们有时候要根据我们的实际要求，写出一个合适自己使用的函数。

那么，我们如何来自己动手编写一个函数呢？

首先，我们要告诉电脑，我们自己编写了一个函数，即这个函数是存在的，这叫作函数的声明 (Declaration)。其次，我们要告诉电脑这个函数是怎么运作的，这叫作函数的定义 (Definition)。显然，函数的声明和函数的定义是两个不同的概念。声明表示该函数存在，而定义则是表示该函数怎么去运行。

我们平时做事都是要有先后顺序的，如果把次序颠倒了可能会惹些麻烦出来。编写函数的时候也一样。我们必须在调用一个函数之前就告诉电脑这个函数已经存在了，否则就成了“马后炮”。所以，我们一般把函数的声明放在主函数前面。

函数的声明

在 C++ 中，函数原型就是函数的声明。所以，函数原型除了向用户说明如何使用一个函数以外，还告诉电脑存在这样一个可以使用的函数。

我们已经介绍了函数原型的结构，只不过“产生结果类型”这个名称是为了方便理解而起的。它应该称为“返回值类型”，用任意一种数据类型来表示，比如 `int` 或者 `char` 等等，当然还包括空类型 `void`。多个参数则构成了“参数表”，表示运行这个函数需要哪些数据。于是，函数原型的结构就是：

返回值类型 函数名(参数表)；

函数声明同变量的声明一样，是一句语句。所以在语句结束要加上分号。函数名、参数名的规则和注意事项同变量名一样，详情请参见第三章。

关于“返回”的概念稍后再作介绍，我们先来说说参数表。我们知道，在声明函数的时候，会写一些参数，而在调用函数的时候需要一一对应地填入这些参数。虽然它们都叫参数，但在不同的情况下，它们的含义是不同的。在声明一个函数的时候，参数是没有实际值的，只是起到一个占位的作用，所以称为形式参数，简称“形参”；在调用一个函数的时候，参数必须有一个确定的值，是真正能够对结果起作用的因素，所以称为实际参数，简称“实参”。我们拿数学中的函数作为例子， $g(x,y)=x/4+y$ 中的 x 和 y 就是形式参数，而 $g(4,1)=4/4+1=2$ 中的 4 和 1 就是实际参数；如果令 $a=4$ 、 $b=1$ ，那么 $g(a,b)$ 中的 a 和 b 也是实际参数。

函数的定义

说完了函数的声明，我们来说函数的定义。其实函数的定义对大家来说是比较熟悉的。因为我们之前所写的程序都是对主函数的定义。函数定义的格式为：

没有分号结尾的函数原型

```
{
    语句块;
}
```

我们把函数定义中没有分号结尾的函数原型称为函数头，把之后的语句块称为函数体。任何一个函数的定义不能出现在另一个函数体内。但函数体内可以调用任何一个函数，包括其本身。

下面我们先来看一个例子，你就会对函数定义有些了解了。（程序 6.2.1）

```
#include "iostream.h"
int max(int a,int b);//函数原型，也是函数声明
① int main()
{
    int r=3,s=5,t;
    t=max(r,s);//调用 max 函数
    cout<<t <<endl;
    return 0;
}
③ int max(int a,int b)//函数头
   {//函数体
     if (a>=b) return a;
     return b;
   }
```

运行结果：

5

程序在运行的时候从 main 函数开始，遇到调用一个用户定义的函数 max，则去查找这个 max 函数的定义，然后运行 max 函数。运行完了以后，回到调用 max 函数的地方，继续后面的语句，直到程序结束。所以整个程序的运行过程如箭头所示。

函数是如何运行的？

我们不难发现，在函数原型的参数表里，就像是多个变量声明的语句。我们可以将其视为创建了若干个变量，然后将实参的值一一赋给这些变量。然后再执行函数体内的语句，进行处理和运算。既然是实参把值赋给了形参，那么在函数体中的数据改变不会影响实参。关于这个问题，我们将在后续章节作详细介绍。

返回语句——return

return 称为返回语句。它的语法格式为：

return 符合返回值类型的表达式；

对于返回，有两层意思。其一是指将表达式的值作为该函数运行的结果反馈给调用函

可以有名称相同的变量或参数。

下面就让我们来看一个实例：（程序 6.2.2）

```
#include "iostream.h"
int max(int a,int b,int c);//求三个整数的最大者
int min(int a,int b,int c);//求三个整数的最小者
void output(int a);//输出功能
int main()
{
    int a=3,b=4,c=2;
    output(max(a,b,c));//把 max 函数的返回值作为 output 函数的实参
    output(min(a,b,c));
    return 0;
}
int max(int a,int b,int c)//不在同一个函数中，参数名重复没关系
{
    if (a>=b && a>=c) return a;
    if (b>=a && b>=c) return b;
    return c;//一旦执行了前面的 return，这句就不会被执行到
}
int min(int a,int b,int c)
{
    if (a<=b && a<=c) return a;
    if (b<=a && b<=c) return b;
    return c;
}
void output(int a)
{
    cout <<a <<endl;
    return;//返回空类型
}
}
```

运行结果：

```
4
2
```

要注意，一旦函数运行结束，那么该函数中声明的参数和变量都将消失。就像下课了，同学们都回家了，老师叫谁都是叫不应的。

函数存在的意义

在第一节，我们已经知道使用工具的好处，即可以重复使用和在各种适用情况下使用。函数和工具一样具有这些好处。但是除此以外，函数的存在还有着其他的意义。

一、现在要设计一个“学生信息处理程序”，需要完成四项工作，分别是记录学生的基本情况、学生成绩统计、优秀学生情况统计和信息输出。如果我们把四项工作全都写的主函数里面，那么我们就很难分清那一段代码在做什么。多层次的缩进和不能重复的变量名给我们阅读程序带来了困难。

如果我们为每一个功能编写一个函数，那么根据函数名每个函数的功能就很清晰了。如

果我们要修改某一个函数的功能，其他的函数也丝毫不会受到影响。所以，**函数的存在增强了程序的可读性。**

二、需要设计一个规模很大的程序，它有几千项功能，把这些功能都编写在一个主函数里就只能由一个人来编写，因为每个人解决问题的思路是不同的，而且在主函数中的变量名是不能重复的，只有编写者自己知道哪些变量名是可以使用的。这样一来，没有一年半载，这个程序是无法完成的。

如果我们把这几千项功能分拆为一些函数，分给几百个人去编写，那么用不了几天时间这些函数就都能够完成了。最后用主函数把这些完成的函数组织一下，一个程序很快就完工了。所以，**函数能够提高团队开发的效率。它就像把各个常用而不相关联的功能做成一块块“积木”。完成了函数的编写，编程就像搭积木一样方便了。**

三、程序会占用一定的内存用来存放数据。如果没有函数，那么在程序的任何一个地方都能够访问或修改这些数据。这种数据的非正常改变对程序的运行是有害的，给调试程序也会带来很多麻烦。

如果我们把若干项功能分拆为函数，则只要把函数原型提供出来就可以了，不需要将数据提供出来。一般情况下，别的函数无法修改本函数内的数据，而函数的实现方法对外也是保密的。**我们把这种特性称为函数的黑盒特性。**

更完整的程序结构

我们认识到一个程序中需要有函数存在，于是一个更完整的程序结构出现了：

```
预处理头文件
各函数声明
主函数
{
  主函数体    //注释
}
各函数定义
```

试试看：

- 1、`void star(int n);`作为函数原型，函数作用是输出 n 个星号，改写程序 5.4.2。
- 2、编写一个函数，使其能够求出 $n!$ ，再求出 $1!+2!+3!+4!+5!$ 。

6.3 多功能开瓶器

我们在开瓶罐的时候，经常会遭遇因各种瓶口规格不同而找不到合适的工具的尴尬。所以有时候就为了开个瓶，家里要备多种规格的开瓶器。同样是开个瓶子嘛，何必这么麻烦？于是有人发明了多功能开瓶器，不管啤酒瓶汽水瓶还是软木塞的红酒瓶都能轻松打开。

然而开瓶器的问题也会发生到程序设计中。比如我们要编写一个函数来求一个数的绝对值，然而整数、浮点型数、双精度型数都有绝对值，但为它们编写的函数返回值类型却是各不相同的。比如：

```
int iabs(int a);
float fabs(float a);
```

```
double dabs(double a);
```

这样是不是有点备了多种开瓶器的感觉？我们能不能在程序设计中也做一个多功能的开瓶器，把所有数据类型的求绝对值都交给 abs 这一个函数呢？

在 C++ 中，我们也能够把具有相同功能的函数整合到一个函数上，而不必去写好多个函数名不同的函数，这叫做函数的重（音 chóng）载（Overload）。重载的本质是多个函数共用同一个函数名。

我们先来看一个函数重载的实例：（程序 6.3）

```
#include "iostream.h"
int abs(int a); //当参数为整型数据时的函数原型
float abs(float a); //当参数为浮点型数据时的函数原型
double abs(double a); //当参数为双精度型数据时的函数原型
int main()
{
    int a=-5,b=3;
    float c=-2.4f,d=8.4f;
    double e=-3e-9,f=3e6;
    cout <<"a=" <<abs(a) <<endl <<"b=" <<abs(b) <<endl; //输出函数返回的结果
    cout <<"c=" <<abs(c) <<endl <<"d=" <<abs(d) <<endl;
    cout <<"e=" <<abs(e) <<endl <<"f=" <<abs(f) <<endl;
    return 0;
}
int abs(int a) //函数定义
{
    cout <<"int abs" <<endl; //显示运行了哪个函数
    return (a>=0?a:-a); //如果 a 大于等于零则返回 a，否则返回-a。
}
float abs(float a)
{
    cout <<"float abs" <<endl;
    return (a>=0?a:-a);
}
double abs(double a)
{
    cout <<"double abs" <<endl;
    return (a>=0?a:-a);
}
```

运行结果：

```
int abs
int abs
a=5
b=3
float abs
float abs
c=2.4
```

```
d=8.4
double abs
double abs
e=3e-009
f=3e+006
```

运行结果表明，abs 函数果然能够处理三种不同数据类型的数据了。那么我们怎样才能自己造一个“多功能工具”呢？

其实要编写一个重载函数并不是很麻烦。首先，我们要告诉电脑，同一个函数名存在了多种定义，所以，我们要给同一个函数名写上多种函数原型（如程序 6.3 的第二到第四行）；其次，我们要对应这些函数原型，分别写上这些函数的定义（如程序 6.3 的主函数体之后，对三个 abs 函数的定义）。

然而电脑又是如何来识别这些使用在不同环境下的“工具”的呢？

在日常生活中使用到多功能工具，如果我们不知道具体应该使用哪个工具，我们会把每个工具放上去试一试，如果只有唯一一个工具适合，那么我们就毫无疑问地能够确定就是使用它了。但是如果出现了两个或者两个以上工具都能适合，我们就分不清到底应该使用哪个是正确的了。

电脑的做法和我们是类似的。电脑是依靠函数声明时参数表中参数个数、各参数的数据类型和顺序来判断到底要运行哪个函数的。因此，当重载函数参数表完全相同的时候，电脑便无法判断应该运行哪个函数，于是程序就出错了。

我们了解了电脑是如何识别重载函数以后，发现要编写一个重载函数还是需要注意一些地方的，那就是：在重载函数中，任意两个函数的参数表中的参数个数、各参数的数据类型和顺序不能完全一样。例如 `int func(int a,char b)` 和 `float func(int c,char d)` 就不能重载，因为它们的参数个数、各参数的类型和顺序完全一样，即使形参名不同、返回值类型不同也是无济于事的。

试试看：

- 1、在程序 6.3 中，若调用重载函数时实参为字符型数据，电脑会运行哪个函数？
- 2、编写一个重载函数，返回类型为 void，函数名为 print，当调用该函数只有一个参数时，以实数形式输出该数（即输出该数本身），当调用时有两个参数，则以复数形式输出该数（如 a+bi）。

在调用一个重载函数时，可能会发生找不到一个完全合适的函数。这时候，就需要进行数据类型的转换。由于这种方法可能导致数据丢失或数据类型不严格符合，且在充分考虑问题后，这种情况是可以尽量避免的，所以这里不再就这个问题展开论述。有兴趣的读者可以查阅其他 C++ 的参考资料。

算法时间：重载函数

从某种意义上说，重载函数是方便了函数的使用者。在前一节我们知道，如果完成了所有函数的编写，那么完成一个程序就像搭积木一样简单了。然而如果功能相似名字却不同的函数太多，那么多“积木”搭起来也未必简单。当函数的编写者充分考虑了不同情况下应该运行稍有不同的函数，函数的使用者就不必为这些小细节而烦恼了。不过重载函数的函数名还是应该符合其功能，如果把功能完全不同的函数重载，那么就大大影响了程序的可读性。

6.4 自动的工具

现在有很多电器都很人性化，比如自动洗衣机，如果想偷个懒你就可以直接把衣服扔进去，使用自动功能，它就能帮你全都搞定；如果哪天要洗个什么大件物品，你也可以人工对其进行设置，同样让你用得得心应手。

我们在调用函数时，可能会要填写很多的参数，那么电脑能否像自动洗衣机一样，让我们偷个懒，帮我们把参数都自动填好呢？

我们知道，所谓自动洗衣功能就是使用其预置好的程序进行洗涤。如果我们也把函数的参数预置好，那么我们同样可以不必填写参数就能让函数运作起来。**这些预置的参数称为默认参数。**

下面我们先来看一个程序，熟悉一下如何来定义默认参数：（程序 6.4）

```
#include "iostream.h"
void create(int n=100);//在函数声明中定义默认参数
int main()
{
    create();//默认实参为 100
    create(5);//人工设置实参
    return 0;
}
void create(int n)//假设该函数的作用是创建空间
{
    cout <<"要创建" <<n <<"个空间" <<endl;
}
```

运行结果：

要创建 100 个空间

要创建 5 个空间

当调用 create 函数，不填写参数时，电脑自动将参数 n 设置为 100 了。而当我们填写参数时，函数也能够按照我们的意愿正常运行。

在定义默认参数时，必须在函数声明中定义。不过，当对多个参数设置默认参数时，会有一些麻烦的情况。

定义默认参数的顺序

当一个函数具有多个参数时，定义默认参数的方向是从右向左的，即以最后一个参数定位的；而匹配参数的方向是从左向右的，即以第一个参数定位的，如下图所示：

如果我们要定义默认参数，那么我们必须从最后一个参数定义起，并且逐渐向前（左）定义，不可以跳过某个参数，直到所有的形参都被定义了默认值。

如果我们调用一个定义了默认参数的函数，那么我们填写的第一个参数一定是和最左边形参匹配，并且逐渐向后（右）匹配，不可以中途省略某一个参数，直到所有未被设置默认值的形参都已经有了参数。

于是，在调用函数时，用户向右自定义的实参至少要和向左来的已定义默认参数的形

```
void func(int a, char b='c', double m=3.0)
    定义默认参数时，方向从右向左

func(3, 'b')=func(3, 'b', 3.0)
    调用函数时，匹配参数方向从左向右
```

参相邻，函数才能够被成功调用。否则这个函数就是缺少参数的。

默认参数和重载函数的混淆

我们在上一节讲了重载函数这个有用的工具，这一节的默认参数也会给我们的程序设计带来方便，然而我们把这两样有用的东西放在一起，却会带来不小的麻烦。我们来看下面这些函数原型：

```
int fn(int a);
int fn(int a,int b=2);
int fn(int a,int b=3,int c=4);
```

这些函数不论是从重载的角度看，还是从默认参数的角度看都是合法的。然而，这样的写法却是不合理的。

当我们调用函数 `fn(1)` 的时候，三个函数都是可以匹配的，因为电脑可以认为我们省略了后面的参数；当我们调用函数 `fn(1,1)` 的时候，后两个函数也都是可以匹配的……由于电脑无法确认我们到底想要调用哪个函数，所以导致了错误的发生。

因此，我们在同时使用重载函数和默认参数的时候，要注意到这一点。

6.5 给变量和参数起个绰号

给别人起绰号是件不好的事情，但是在程序设计中，给变量或参数起个绰号却会给我们带来一些方便。绰号，就是另一种称呼。绰号和本名都是指同一样东西，绰号也是个名称，所以它的命名规则其他的命名规则一样，详情可参见 3.1。另外，“绰号”显然也不能和“本名”相同。

这种给变量起“绰号”的操作称为引用（Reference），“绰号”称为引用名。声明引用的语法格式为：

变量数据类型 &引用名=已声明的变量名；

我们对变量使用了引用以后，对引用的操作就如同对被引用变量的操作。这就好像叫一个人的绰号和叫一个人的本名有着同样的效果。在声明一个引用时，必须告知电脑到底是哪个变量被引用，否则这个“绰号”就显得有些莫名其妙了。

下面我们来看一段简单的程序：（程序 6.5.1）

```
#include "iostream.h"
int main()
{
    int a=2;
    int &b=a;//给变量 a 起了个绰号叫 b
    cout <<"a=" <<a <<endl;
    cout <<"b=" <<b <<endl;
    a++;
    cout <<"a=" <<a <<endl;
    cout <<"b=" <<b <<endl;
    b++;//对 b 的操作也就是对 a 的操作，所以 b++就相当于 a++
    cout <<"a=" <<a <<endl;
    cout <<"b=" <<b <<endl;
    return 0;
}
```

```
}
```

运行结果：

```
a=2
```

```
b=2
```

```
a=3
```

```
b=3
```

```
a=4
```

```
b=4
```

我们在这个程序中，能够验证对引用的操作相当于对被引用变量的操作。或许你还没看出引用到底派了什么大用处，不过马上你就会恍然大悟了。

用引用传递参数

其实引用最有用的地方还是在函数当中，下面我们先来看一个简单的例子：

```
#include "iostream.h"
```

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

```
int main()
```

```
{
```

```
    int a=2,b=3;
```

```
    swap(a,b);
```

```
    cout <<"a=" <<a <<endl;
```

```
    cout <<"b=" <<b <<endl;
```

```
    return 0;
```

```
}
```

```
void swap(int x,int y)
```

```
{
```

```
    int temp;
```

```
    temp=x;
```

```
    x=y;
```

```
    y=temp;
```

```
}
```

运行结果：

```
a=2
```

```
b=3
```

在以上这段程序中，`swap` 函数的语句是我们熟悉的交换语句，可是为什么执行了这个 `swap` 函数以后，`a` 和 `b` 的值并没有交换呢？

在 6.2 中，我们介绍过，函数是将实参的值赋给了形参。这就像本来我们想交换 `a` 碗和 `b` 碗里的水，调用了 `swap` 函数则是拿来了 `x` 碗和 `y` 碗，然后把 `a` 碗里的水分一点到 `x` 碗里，`b` 碗里的水分一点到 `y` 碗里，再把 `x` 碗和 `y` 碗里的水交换。可是，这样的操作有没有将 `a` 碗里的水和 `b` 碗里的水交换呢？没有。而且，我们还知道，一旦函数运行结束，函数中定义的参数和变量就都会消失，所以就连 `x` 碗和 `y` 碗也都没有了。

问题到底在于哪里呢？在于我们传给函数的是“水”，而不是“碗”。如果我们直接把 `a` 碗和 `b` 碗交给函数，那么这个任务就能够完成了。下面我们就来看看如何把“碗”来传递给函数：（程序 6.5.2）

```
#include "iostream.h"
```



```

void swap(int &x,int &y);//用引用传递参数
int main()
{
    int a=2,b=3;
    swap(a,b);
    cout <<"a=" <<a <<endl;
    cout <<"b=" <<b <<endl;
    return 0;
}
void swap(int &x,int &y)//函数定义要和函数原型一致
{
    int temp;
    temp=x;
    x=y;
    y=temp;
}

```

运行结果:

```

a=3
b=2

```

如果我们把没有使用引用的参数认为是 $\text{int } x=a$ 和 $\text{int } y=b$ (把变量 a 和变量 b 的值分别传给参数 x 和参数 y)，那么使用了引用的参数就是 $\text{int } \&x=a$ 和 $\text{int } \&y=b$ 了。也就是说 x 和 y 成为了变量 a 和变量 b 的“绰号”，对参数 x 和 y 的操作就如同对变量 a 和 b 的操作了。

除了在引用变量和通过引用传递参数外，引用还能被函数返回。关于这些内容，有兴趣的读者可以查阅相关书籍资料。

6.6* 函数里的自己

在高中数学中，我们学习过数列。我们知道数列有两种表示方法，一种称为通项公式，即项 a_n 和项数 n 的关系；还有一种称为递推公式，即后一项 a_n 和前一项 $a_{(n-1)}$ 之间的关系。通项公式能够一下子把 a_n 求解出来，而递推公式则要根据首项的值多次推导才能把第 n 项的值慢慢推导出来。如果有一个较为复杂的数列的递推公式，人工将其转化为通项公式或者将其的每一项求出实在非常麻烦，那么我们能不能直接把这个递推公式交给计算机，让它来为我们求解呢？

我们说过，在任何一个函数体内不能出现其它函数的定义。但是，在任一个函数体内可以调用任何函数，包括该函数本身。**直接或者间接地在函数体内调用函数本身的现象称为函数的递归**。正是函数的递归，能够帮我们解决递推公式求解的问题。

现有一个数列，已知 $a_n=2*a_{(n-1)}+3$ ，并且 $a_1=1$ ，求解 a_1 到 a_8 的各项值。我们把数列问题转化为函数问题，认为 $a_n=f(n)$ ， $a_{(n-1)}=f(n-1)\cdots\cdots$ 于是 $f(n)=2*f(n-1)+3$ ， $f(n-1)=2*f(n-1-1)+3\cdots\cdots$ 直到 $f(1)=1$ 。我们根据前面的描述写出以下程序：(程序 6.6)

```

#include "iostream.h"
int f(int n);//看作数列a_n
int main()
{
    for (int i=1;i<=8;i++)

```

```

    {
        cout <<"f(" <<i <<" )=" <<f(i) <<endl;//输出a1到a8的值
    }
    return 0;
}
int f(int n)
{
    if (n==1)
    {
        return 1;//告知a1=1
    }
    else
    {
        return 2*f(n-1)+3;//告诉电脑 f(n)=2*f(n-1)+3
    }
}

```

运行结果:

```

f(1)=1
f(2)=5
f(3)=13
f(4)=29
f(5)=61
f(6)=125
f(7)=253
f(8)=509

```

试试看:

已知数列 $a_n=n*a_{(n-1)}$ ，且知道 $a_1=1$ ，试用递归函数求解 a_1 到 a_5 。

在这里，并不要求大家立刻掌握递归函数。主要是给大家介绍这样一个概念，只要大家能够看懂应用于数列的简单递归函数就可以了。至于如何更广泛地应用递归函数，将在后续章节中再作介绍。

习题

1、根据函数原型，判断下列调用函数是否正确（假设 x 是已定义的整型变量）。

① `int func(char a,int b);`

调用: `x=func('c')(2);`

② `int func(int a,double b,float c,char d);`

调用: `x=func(1,2,3);`

③ `void func(int a,int &b);`

调用: `x=func(x,x);`

④ `int func(int a,int b=100,char c='d');`

调用: `x=func(0,,'s');`

⑤ `int func(int a,int b);`

调用: `x=func(func(3),2);`

2、指出下列函数原型或函数定义的错误。

① `int min(int a,int b,int c);`//找出最小值

```
void min(int a,int b)
{
    if (a>b) return a;
    else return b;
}
```

② `void char(char c);`//输出功能

```
void char(char c);
{
    cout <<c;
    return;
}
```

③ `double add(double result,double a,double b);`//做加法

```
double add(double result,double a,double b)
{
    result=a+b;
}
```

④ `void swap(int a,int b);`//两数交换

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

```
int max2(int a,int b,int c);
```

`int max2(int a,int b,int c);`//求较大的两个数

```
{
    if (a<=b) swap(a,b);
    if (a<=c) swap(a,c);
    if (b<=c) swap(b,c);
    return a,b;
}
```

⑤ `void func(int c);`//求 $a_n=a_{(n-1)}+2$, 且 $a_1=3$

```
void func(int c)
{
    if (c==1) return 3;
    return func(c-1)+2;
}
```

3、判断下列几组重载函数是否能够正常工作。

① `char ch(double a,int b);`

- ```

char ch(double c,int d);
② void f(char c='c',int m=2);
int f();
③ int func(int a,int b,int c);
void func(int a,int b);
float func(float a,int b);
float func(int &a,int b);

```

4、根据要求编写函数，要求写出函数原型和函数定义。

- ①用一个整数表示年份作为参数，判断这年是否是闰年。（提示：布尔型数据）
- ②使用系统给出的正弦函数和平方根函数，自己编写一个余弦函数。（不需要预处理头文件，余弦函数的参数为整型，要求以°为单位）
- ③以一个长整数为参数，求它是否是某个整数的阶乘，如果是则返回这个数，否则返回-1。

5、阅读下列程序，写出运行结果。

```

①
#include "iostream.h"
int func1(int a,int b);
int func1(int a);
float func1(float a,int b=3);
void func2(int &x,int y);
int main()
{
 int a=1,b=2,c=3;
 float d=6.0;
 b=func1(func1(a,b));
 d=func1(d);
 func2(a,c);
 cout <<a <<endl <<b <<endl <<c <<endl <<d <<endl;
 return 0;
}
int func1(int a,int b)
{
 return a==b;
}
int func1(int a)
{
 return a;
}
float func1(float a,int b)
{
 return a/b;
}
void func2(int &x,int y)
{
 x=y;
}

```

```
 return;
}
②
#include "iostream.h"
int func(int a,int b);
int main()
{
 cout <<func(8,3) <<endl;
 return 0;
}
int func(int a,int b)
{
 if (b==0) return 1;
 else return func(a-b,a/b);
}
```

6、根据要求编写程序。

①从键盘输入三个实数a、b、c分别作为一个一元二次方程 $ax^2+bx+c=0$ 的三个系数。使用系统给出的平方根函数，编写一段程序，使之求出这个方程的两个根。其中，求 $\Delta=b^2-4*a*c$ 的功能要以函数形式出现。（提示：求根公式， $\Delta < 0$ 时方程无解）

②已知 $a_1=1$ ， $a_2=1$ ，当n大于等于2时， $a_{(n+1)}=3*a_n-a_{(n-1)}$ ，要求用递归函数输出 $a_1$ 到 $a_8$ 。

## 第七章 好大的仓库

在第三章，我们介绍了像箱子一样的变量。在前几章的学习中，我们也基本掌握了如何使用变量。可是，单个的变量有一个严重的缺陷，就是它能够存储的数据实在是太少了，要么只能存一个数，要么就只能存一个字符。然而，我们有时候要处理很多数据，那这些数据应该怎么放呢？在本章，我们要学习存放数据的大仓库——数组。学会了使用数组，我们就能让电脑处理更多数据了。

### 7.1 方便地让电脑处理更多数据

我们知道，在程序设计中，大多数数据都是存放在变量里的。如果我们要处理较多的数据，增加存放数据的空间最简单的方法就是多开设一些变量。然而，变量多了就难以管理了。这就好像一个班级里的学生名字有长有短，即使没有重复的名字，要在一长串名单里找到一个同学的名字也不是件容易的事情。于是，最方便的方法就是给同学们编上学号了，把名单按学号排列好以后，查找起来只要找学号就可以了。因为数字的排列是从小到大的，是有序的，所以查找起来要比在一堆长短不一的名字中查找要方便多了。

我们受到“学号”的启发，可以给变量也编一个号，把存储着相关内容的变量编在一组内，这就称为数组（Array）。

### 数组的声明

数组的本质也是变量，所以我们在使用数组之前，必须要声明数组。声明一个数组的语法格式为：

**数据类型 数组名[常量表达式];**

和声明变量类似，数据类型仍然是整型、字符型等等，数组的命名规则和变量的命名规则也一样。在这里，我们要说明两个问题：以前我们说过在语法规则中的中括号表示可有可无的东西，然而在数组名后的中括号有着其独特的含义，而不是可有可无的。数组名后的中括号是数组的一个特征，没有这个特征就不是数组了。数组中每个存放数据的变量称为数组元素。中括号内的常量表达式称为数组的大小，即元素的个数。例如 `int a[5]`；这句语句就是声明了一个可以存放五个整型数据的数组，它所能存储的数据相当于五个整型变量。

电脑必须在程序执行之前就已经知道数组的大小，因此中括号内只能是一个常量表达式，而不能含有变量。

通过实验，我们知道我们无法根据程序运行的实际情况来声明一个数组的大小。所以，为了保证程序有足够的存储空间和正常运行，我们尽量要声明一个足够大的数组。要注意，足够大不是无穷大。比如我们要存放一个班级学生的成绩，我们声明一个大小为 70 的数组是足够大，但是声明一个大小为 1000 的数组却是不必要的浪费。

### 数组的操作

我们前面说到，数组就像是给变量编了号。那么我们要访问数组中的某一个元素时自然就要用到这个编号了。给学生编的号称为学号，给数组元素编的号称为下标（Subscript）。我们要表达数组中某一个元素的格式是：数组名[下标]。在 C++ 中，下标是从 0 开始的，所

以一个大小为  $n$  的数组，它的有效下标是  $0 \sim n-1$ 。如果下标不在这个范围内，就会发生错误。和声明数组时不同，操作一个数组时，它的下标既可以是一个常量表达式，也可以是一个变量表达式。

对数组元素的操作就如同对某一相同数据类型的变量的操作。下面我们来看一个简单的例子：（程序 7.1）

```
#include "iostream.h"
int main()
{
 int array[5]; //声明一个可以存放五个整数的数组
 for (int i=0;i<5;i++) //如果写成 i<=5 就要出问题了
 {
 array[i]=i+1; //对各数组元素赋值
 }
 for (int j=0;j<5;j++)
 {
 cout <<array[j] <<" "; //输出各数组元素
 }
 cout <<endl;
 return 0;
}
```

运行结果：

```
1 2 3 4 5
```

阅读了以上程序，我们发现除了要注意下标是否有效，对数组的操作和对变量的操作并无异样。

算法时间：数组的下标和循环控制变量

在一开始，我们就说了给学生编号是为了避免在长短不一的姓名中查找。使用一个数组而不使用多个变量的原因也是类似的。由于循环语句和数组下标的存在，再搭配循环控制变量，就能很方便地对多个数据进行类似的反复操作。（我们一般把循环控制变量作为数组的下标。如程序 7.1.1 中所示。）这种优势是多个变量所没有的。这也是数组存在的重要意义。如果一种高级语言没有数组功能，那么它将很难实现大数据量的复杂程序。

试试看：

输入下列程序，看看在编译的时候会发生什么错误：

```
#include "iostream.h"
int main()
{
 int size;
 int a[size];
 cin >>size;
 return 0;
}
```

## 数组的初始化

我们知道，变量在声明的同时可以进行初始化。同样地，数组在声明的时候进行初始化，声明并初始化数组的语法格式为：

**数据类型 数组名[常量表达式]={初始化值<sub>1</sub>, 初始化值<sub>2</sub>, ……初始化值<sub>n</sub>};**

在初始化数组时，大括号中的值的个数不能大于声明数组的大小，也不能通过添加逗号的方式跳过。但是初始化值的个数可以小于声明数组的大小，此时仅对前面一些有初始化值的元素依次进行初始化。

比如：

```
int array1[3]={0,1,2}; //正确
int array2[3]={0,1,2,3}; //错误，初始化值个数大于数组大小
int array3[3]={0,,2}; //错误，初始化值被跳过
int array4[3]={0,1,}; //错误，初始化值被跳过（即使是最后一个元素，添加逗号也被认为是跳过）
int array5[3]={0,1}; //正确，省略初始化最后一个元素
```

## 省略数组大小

我们已经知道了如何定义和初始化一个数组。然而有时候我们既要赋初值又要数元素个数，有些麻烦。既然我们对各元素赋了初值，电脑能否自己算出有多少个元素呢？

答案是肯定地的。只要我们对各元素赋了初值，电脑会自动计算出声明的数组应该有多大。比如：

```
int array[]={0,3,4,8};
这句话就相当于：
int array[4]={0,3,4,8};
```

这样的写法便于我们对数组元素的插入或修改，我们只需要直接在花括号中对数据进行修改就可以了，而不必去考虑中括号中的数组大小应该怎么变化。但是，这种情况下，我们又怎么才能知道数组的大小呢？我们将在下一节讲述这个问题。

试试看：

1、声明一个大小为 6 的数组，试试看能否输出下标为 6 的元素的数据？能否修改下标为 6 的元素的数据？

**结论：可以输出下标超出有效范围的元素（我们称为越界访问），但是修改下标超出有效范围的元素将可能导致未知的错误。**

2、改写程序 7.1，通过键盘输入给数组的五个元素赋值，再输出这五个元素。

## 7.2 仓库是怎样造成的？

我们以前说变量像箱子，数组像仓库。在这一节，我们要来深入探究一下，这些“箱子”和“仓库”在电脑内部是怎样摆放的。



## 内存和地址

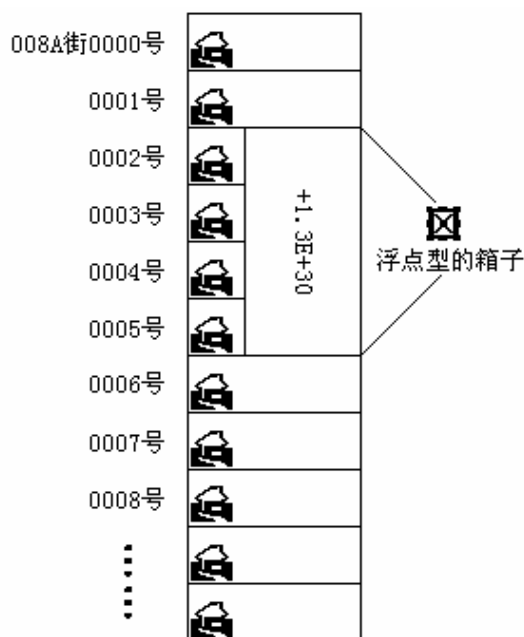
我们知道变量和数组都是放在内存里的，我们有时候还能够听到内存地址（Address）这个词。那么地址究竟是什么意思呢？

其实在内存里，就像是许许多多的街道，每条街道有它的名字，而街道上的每幢房子又按顺序地编了号，于是街道名和房子在街道上的编号就能确定内存中唯一的一幢房子，我们在这里认为所有的数据在内存中都是放在房子里。电脑就是依照这个原理找到所要的访问或修改的数据的。**街道名和房子在街道上的编号就称为这个房子的地址。**我们通常把地址表示为一串十六进制的数。关于十六进制数我们在这里不作展开说明。

那么这些内存中的房子和我们所说的变量和数组是什么关系呢？**在内存里的房子的大小是规定的，每幢房子只能存储一个字节**

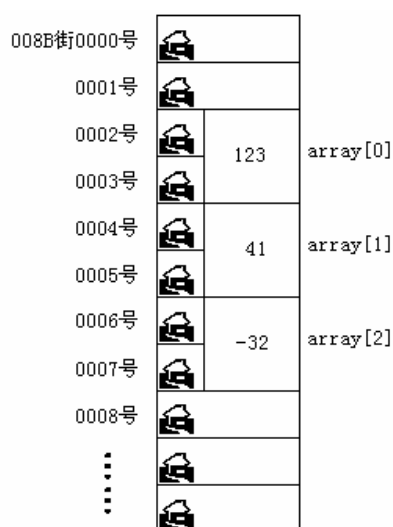
**（Byte）的数据。**（一个字节相当于一个半角的英文字母，一个汉字需要占用两个字节。）有时候，一种类型的变量需要比较大的空间，比如一个浮点型的实数，一幢房子是放不下的，而是需要 4 幢房子的空间才能放得下。于是电脑就把连起来的 4 幢房子拼起来，每幢房子放这个实数的一部分数据。而这连起来的 4 幢房子，构成了一个能够存放浮点型实数的变量。

我们认为，内存中的“房子”是客观存在的，每幢房子的大小一样，没有任何区别；而不同类型的变量“箱子”则是由于若干幢房子拼接而成，箱子在内存中是不存在的，只是我们为了方便理解而臆想出来的。右图就是一个浮点型变量在内存中的情况。（图 7.2.1）



（图 7.2.1）

## 数组在内存中的存储情况



（图 7.2.2）

变量在内存中是由若干个相邻的“房子”拼接而成的，而数组在内存中则是由若干个相邻的数组元素按顺序拼接而成的。每个数组元素又相当于一个变量。左图是一个大小为 3 的短整型（short）数组在内存中的情况。（图 7.2.2）

我们在上一节的最后说到可以省略数组的大小，但是这样一来我们就无法得知数组的大小了，这将可能造成越界访问。当我们了解了数组在内存中的存储情况后，我们就能够知道数组的大小了。在 C++ 中，有一个名为 `sizeof` 的操作符，可以求出一个数组或一种数据类型在内存中占了多少“房子”，它的使用方法是：

**`sizeof(数组名或数据类型);`**

通过左图我们可以理解，要求出数组的大小，

应该是用整个数组占的“房子”数除以每一个数组元素占的“房子”数，即 6 除以 2 等于 3。下面我们来看一个求出数组大小的程序实例：（程序 7.2.1）

```
#include "iostream.h"
int main()
{
 int array[]={3,5,6,8,7,9};
 int size=sizeof(array)/sizeof(int);
 cout <<"size="<<size <<endl;
 for (int i=0;i<size;i++)
 cout <<array[i] <<" ";
 cout <<endl;
 return 0;
}
```

运行结果：

```
size=6
3 5 6 8 7 9
```

通过这个程序，可以成功地知道一个数组的大小，我们也不用为可能发生的越界访问而发愁了。

## 字符的存储情况

电脑是用电来计算和保存信息的。在电脑里，就好像有许许多多的开关，用导通（开）来表示 1，用断开（关）来表示 0。那么这些个“0”和“1”是怎么来表示一些字符的呢？

当只有一个开关的时候，这个开关能表示两种状态，即 0 和 1；当有两个开关的时候，这两个开关可以表示四种状态，即 00、01、10、11……如果你学过排列，就不难理解，当有 8 个开关的时候，可以表示  $2^8=256$  种状态，分别是 0~255。在电脑中，就是用 8 个开关（0 或 1）来表示一个字节的，每一个开关（0 或 1）称为一个“位”（Bit），即 8 位组成一个字节。我们把一个字节所能表示的 256 种状态和 256 个字符按一定的顺序一一对应起来，一个字节就可以表示 256 种不同的字符。这种用 8 位二进制表示一个字符的编码称为 ASCII 码（念 aski），它的全称是美国信息交换标准码（America Standard Code for Information Interchange）。我们需要记住的 ASCII 码有三个，数字 0 的 ASCII 码为十进制的 48，大写字母 A 的 ASCII 码为十进制的 65，小写字母 a 的 ASCII 码为十进制的 97。下面我们就来编写一段程序，输出 ASCII 码表的常用部分：（程序 7.2.2）

```
#include "iostream.h"
#include "iomanip.h"
int main()
{
 char temp;
 for (int i=32;i<=127;i++)
 {
 temp=i;
 cout << setw(2) <<temp;
 if (i%16==15) //从 0~15 正好 16 个，所以余数为 15 的时候换行
 {
 cout <<endl;
 }
 }
}
```



Size of C=6

Hello

Hello 烫藁 ello

Hello

从数组 a、b 和 c 的大小，我们就能看出按字符串和按字符初始化的不同了。你可能还会发现，输出的数组 a 和 c 都是正常的，为什么输出的 b 却夹杂着乱码呢？这是因为 a 和 c 的属性都是字符串的字符数组，而 b 是普通字符数组。b 数组没有结尾符，电脑在输出它的时候就会发生问题了。

数组 a 和 b 在内存中的存储情况如右上图所示（图 7.2.3）。

## 7.3 向函数传递数组

数组的存储空间很大，如果能够把它作为参数传递给函数，那么就能发挥很大的作用了。比如本来一个选出最大数的 max 函数只能在两个数或三个数里面选出一个最大的数字，如果我们把数组传递过去，它就能够在一堆数字中选出最大的数了，这大大提高了程序的效率。当函数中的某个参数是数组时，在参数名后加上一对中括号，比如 int a[]，表示参数 a 是一个数组。下面我们就来看这样一个在一堆正数里面找一个最大数的程序：（程序 7.3.1）

```
#include "iostream.h"
int max(int a[],int size);//size 是数组的大小
int main()
{
 int number[]={2,45,12,6,23,98,13,3};
 cout <<max(number,sizeof(number)/sizeof(int)) <<endl;
 return 0;
}
int max(int a[],int size)
{
 int max=0;
 for (int i=0;i<size;i++)
 {
 if (a[i]>max)
 max=a[i];
 }
 return max;
}
```

运行结果：

98

我们发现，在函数里使用数组也是比较方便的。但大家有没有考虑过一个问题，我们为什么不在函数里面用 sizeof 算出数组的大小，而非要在函数外面算好了，再作为参数传递给函数呢？在这里，我们就有必要讲一下数组作为参数传递给函数的原理了。

我们以前说过，参数的传递是将实参的值赋给形参。然而对于数组来说却是一个例外，因为数组的数据太多了，将其一一赋值既麻烦又浪费空间，所以数组作为参数传递给函数的只是数组首元素的地址，函数在需要用到后面元素时再按照这个地址和数组下标去查找。也就是说后面的元素根本没到函数里来，所以在函数里求不出数组的大小也就不足为奇了。

所以, 当一个函数的参数是一个数组时, 我们必须注意要想办法让这个函数知道数组的大小。

不过, 既然数组传递给函数的是数组首元素在内存中地址, 而数据又都是存在内存里的, 那么在函数中对数组参数的修改会不会影响到实参本身的值呢? 让我们来看一段程序, 验证一下我们的想法: (程序 7.3.2)

```
#include "iostream.h"
#include "iomanip.h"
void sort(int a[],int size);//将数组中的元素从大到小排列
int main()
{
 int num[]={2,3,8,6,4,1,7,9};
 const int size=sizeof(num)/sizeof(int);
 sort(num,size);
 cout<<"排列后的数组元素" <<endl;
 for (int i=0;i<size;i++)//输出排列好以后的数组元素
 {
 cout <<setw(2) <<num[i];
 }
 cout <<endl;
 return 0;
}
void sort(int a[],int size)
{
 cout <<"原来的数组元素" <<endl;
 for (int i=0;i<size;i++)//输出原来的数组元素
 {
 cout <<setw(2) <<a[i];
 }
 cout <<endl;
 for (int j=0;j<size;j++)
 {
 int min=a[j],mink=j;//先假设未排序的首元素是最小的数
 for (int k=j;k<size;k++)//找到尚未排序的元素中最小的数
 {
 if (a[k]<min)
 {
 min=a[k];
 mink=k;
 }
 }
 int temp=a[j];//交换两个元素
 a[j]=a[mink];
 a[mink]=temp;
 }
}
```

|     |   |   |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|---|---|
| 第一次 | 2 | 3 | 8 | 6 | 4 | 1 | 7 | 9 |
| 第二次 | 1 | 3 | 8 | 6 | 4 | 2 | 7 | 9 |
| 第三次 | 1 | 2 | 8 | 6 | 4 | 3 | 7 | 9 |
| 第四次 | 1 | 2 | 3 | 6 | 4 | 8 | 7 | 9 |
| 第五次 | 1 | 2 | 3 | 4 | 6 | 8 | 7 | 9 |
| 第六次 | 1 | 2 | 3 | 4 | 6 | 8 | 7 | 9 |
| 第七次 | 1 | 2 | 3 | 4 | 6 | 7 | 8 | 9 |
| 第八次 | 1 | 2 | 3 | 4 | 6 | 7 | 8 | 9 |

程序 7.3.2 排序过程

(灰: 未排序 白: 已排序 红: 未排序中最小)

运行结果：

原来的数组元素

2 3 8 6 4 1 7 9

排列后的数组元素

1 2 3 4 6 7 8 9

算法时间：排序（Sort）

排序是经常要使用到的一项功能。排序的算法也有多种。程序 7.3.2 所使用的排序方法称为直接选择排序，即在未排序的元素中选择出最小的一个，与未排序的首元素交换，直到所有的元素都已经排序。（如右上表所示）以后大家还会在数据结构课程中学习到一些更高效的排序算法，如快速排序法，插入排序法等等。

我们交换了 sort 函数中参数数组 a 的顺序，却发现回到主函数以后，num 数组的元素次序也发生了变化。正是因为我们在函数中将内存中的数据作了操作，所以影响到了实参。

试试看：

改写程序 7.3.2，给一个字符数组 {'c','a','d','e','b','h'} 按字母表顺序排序，并输出排序前和排序后的结果。

## 7.4 二维数组

我们知道，一维空间是一条线，数学中用一条数轴来表达；二维空间是一个平面，数学中用平面坐标系来表达。那么二维数组又是什么样的呢？

### 线与面

我们用一个下标来描述一维数组中的某个元素，就好像在用数描述一条线上的点。而所有的数据都是存储在一条线上。如果我们采用两个下标，就能形成一个平面，犹如一张表格，有行有列，所有的数据就能够存放到表格里。

|      |         |         |         |         |         |
|------|---------|---------|---------|---------|---------|
| a[0] | a[0][0] | a[0][1] | a[0][2] | a[0][3] | a[0][4] |
| a[1] | a[1][0] | a[1][1] | a[1][2] | a[1][3] | a[1][4] |
| a[2] | a[2][0] | a[2][1] | a[2][2] | a[2][3] | a[2][4] |
| a[3] | a[3][0] | a[3][1] | a[3][2] | a[3][3] | a[3][4] |
| a[4] | a[4][0] | a[4][1] | a[4][2] | a[4][3] | a[4][4] |

一维数组

二维数组

我们把二维数组的两个下标分别称为行下标和列下标，在前面的是行下标，在后面的是列下标。

那么什么时候要用二维数组呢？一般有两种情况，一种是描述一个二维的事物。比如用 1 表示墙，用 0 表示通路，我们可以用二维数组来描述一个迷宫地图；用 1 表示有通路，0 表示没有通路，我们可以用二维数组来描述几个城市之间的交通情况。还有一种是描述多个具有多项属性的事物。比如有多个学生，每个学生有语文、数学和英语三门成绩，我们就可以用二维数组来描述。

对于第二种情况，我们要注意各项属性应该是同一种数据类型，比如三种学科的成绩都是整数。如果出现了姓名（字符串属性），就不能将他们组合到一个二维数组里去。所以不要企图将不同数据类型的属性整合到一个二维数组中去。

## 二维数组的声明和初始化

二维数组的声明和一维数组是类似的，不同之处只是多了一个下标：

**数据类型 数组名[行数][列数]；**

要注意，二维数组的下标也都是从 0 开始的。

二维数组的初始化分为两种，一种是顺序初始化，一种是按行初始化，我们来看一段程序，就能够对它们有所了解：（程序 7.4.1）

```
#include "iostream.h"
#include "iomanip.h"
int main()
{
 int array1[3][2]={4,2,5,6};//顺序初始化
 int array2[3][2]={{4,2},{5},{6}};//按行初始化
 cout <<"array1" <<endl;
 for (int i=0;i<3;i++)//输出数组 array1
 {
 for (int j=0;j<2;j++)
 {
 cout <<setw(2) <<array1[i][j];
 }
 cout <<endl;
 }
 cout <<"array2" <<endl;
 for (int k=0;k<3;k++)//输出数组 array2
 {
 for (int l=0;l<2;l++)
 {
 cout <<setw(2) <<array2[k][l];
 }
 cout <<endl;
 }
 return 0;
}
```

运行结果：

```
array1
 4 2
 5 6
13 4
array2
 4 2
 5 8
 6 8
```

我们可以看出，所谓按顺序初始化就是先从左向右再由上而下地初始化，即第一行所有元素都初始化好以后再对第二行初始化。而按行初始化则是用一对大括号来表示每一行，

跳过前一行没有初始化的元素，在行内从左向右地进行初始化。对于没有初始化的元素，则都是一个不确定的值。

## 省略第一维的大小

我们在第一节学到，一维数组的大小可以省略。可是二维数组的元素个数是行数和列数的乘积，如果我们只告诉电脑元素个数，电脑无法知道究竟这个数组是几行几列。所以，C++规定，在声明和初始化一个二维数组时，只有第一维（行数）可以省略。比如：

```
int array[][3]={1,2,3,4,5,6};
```

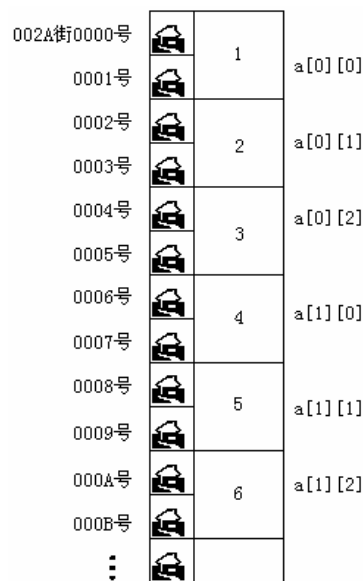
相当于：

```
int array[2][3]={1,2,3,4,5,6};
```

## 二维数组在内存中的存储情况

先前已经说明，内存是依靠地址来确定内存中的唯一一个存储单元的，即只有一个参数。所以在内存中，所有的数据都是像一维数组那样顺序存储的。那么具有两个下标的二维数组是怎样存放到内存中的呢？

在内存中，先将二维数组的第一行按顺序存储，接着就是第二行的数据，然后是第三行的数据……右图（图 7.4）所示的就是一个二维数组在内存中的存储情况。



（图 7.4）

## 向函数传递二维数组

我们知道，数组作为参数传递给函数的是数组首元素的地址。对于二维数组来说亦是如此。不过有两个问题，一个是我们必须让函数知道行数和列数，这就像我们要让函数知道一维数组的大小一样，防止发生越界访问。另一个就是我们必须让电脑知道这个二维数组是怎样的一个表格，即必须告知数组的列数。这和只能省略二维数组的行数道理是一样的。下面我们就来看一个向函数传递二维数组的程序：

```
#include "iostream.h"
#include "iomanip.h"
void disp(int a[][2],int r,int c);//告知数组的列数
int main()
{
 int array[3][2]={4,2,5,6,3,1};
 cout <<"array" <<endl;
 disp(array,3,2);
 return 0;
}
void disp(int a[][2],int r,int c)
{
 for (int i=0;i<r;i++)
 {
```



```

 for (int j=0;j<c;j++)
 {
 cout <<setw(2) <<a[i][j];
 }
 cout <<endl;
}
}

```

运行结果:

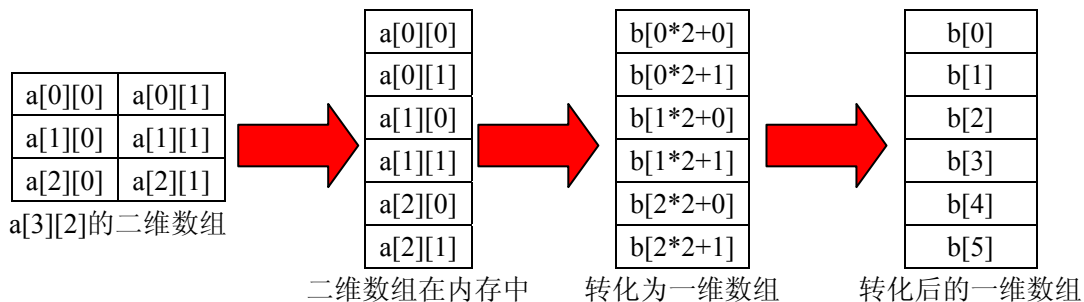
```

array
 4 2
 5 6
 3 1

```

## 二维数组转化成一维数组

有些时候, 我们觉得用二维数组来描述一样事物很方便。比如我们用二维数组来画一个迷宫地图, 行下标和列下标就如同直角坐标系一样。可是在某些情况下, 不能使用二维数组, 或者难以制造一个二维数组。二维数组在内存中的存储情况和一维数组是相同的, 所以我们只好用一个一维数组来代替它了。



于是, 我们不难总结出一个结果, 一个二维数组元素  $a[x][y]$  在一维数组  $b$  中, 是:

$$a[x][y]=b[x*列数+y]$$

## 习题

1、判断下列数组的声明及初始化是否正确, 如有错误, 请指出。

- ① `int array[5]={1,2,,4,5};`
- ② `double array[]={1.0,4,5/3.3,};`
- ③ `char str[]={ 'H', 'e', 'l', 'l', 'o' };`
- ④ `char str1[5]={"Hello"};`
- ⑤ `int a[3][]={1,2,3,4,5};`
- ⑥ `int a[][2]={1,{3,4},5};`

2、根据下列实际情况, 判断最好应该用什么方法存储数据。

- ① 一句英语句子
- ② 一个  $6 \times 6$  的矩阵
- ③ 三种水果每个月的销量的年表
- ④ 乒乓球双循环赛的结果

3、指出下列程序的错误之处。

①//矩阵的加法

```
#include "iostream.h"
void plus(int a[],int b[],int r,int c);
int main()
{
 int matrix1[][3]={3,2,4,-1,5,6,-5,-2,-1};
 int matrix2[][3]={2,1,3,8,-3,1,-1,-2,0};
 plus(matrix1,matrix2,3,3);
 return 0;
}
void plus(int a[],int b[],int r,int c)
{
 int temp[][3];
 for (int i=0;i<r;i++)
 {
 for (int j=0;j<c;j++)
 {
 temp[i][j]=a[i][j]+b[i][j];
 }
 }
 for (int k=0;k<r;k++)//以矩阵形式输出
 {
 for (int l=0;l<c;l++)
 {
 cout <<setw(3) <<temp[k][l];
 }
 }
}
```

②//将数组内的元素倒置

```
#include "iostream.h"
int main()
{
 int array[]={1,2,3,4,5,6,7,8,9};
 for (int i=0;i<sizeof(array)/2;i++)
 {
 temp=array[i];
 array[i]=array[sizeof(array)-i];
 array[sizeof(array) -i]=temp;
 }
 for (int j=0;j<sizeof(array);j++)
 {
 cout <<' ' <<array[j];
 }
}
```

```

 cout <<endl;
 return 0;
}
4、根据运行结果完成代码。
#include "iostream.h"
int squeeze(int a[],int size,int num);
int main()
{
 int temp;
 int number[]={3,16,18,2,4,19,5,15};
 const int size=_____
 for (int i=0;i<3;i++)
 {
 cout <<"请输入一个数" <<endl;
 cin >>temp;
 cout <<"被挤出来的是" <<squeeze(_____) <<endl;
 }
 return 0;
}
int squeeze(int a[],int size,int num)
{
 int min_____,temp,mini;
 cout <<"原来的数组为" <<endl;
 for (int i=0;i<size;i++)
 {
 cout <<' ' <<a[i];
 if(_____)
 {
 min=a[i];

 }
 }
 cout <<endl;
 if(_____)
 {
 temp=a[mini];
 a[mini]=num;
 return temp;
 }
 else
 {

 }
}
}

```

运行结果:

请输入一个数

17

原来的数组为

3 16 18 2 4 19 5 15

被挤出来的是 2

请输入一个数

1

原来的数组为

3 16 18 17 4 19 5 15

被挤出来的是 1

请输入一个数

8

原来的数组为

3 16 18 17 4 19 5 15

被挤出来的是 3

5、根据要求编写程序。

①A 国和 B 国之间爆发了战争，由于需要大量的战况信息，所以需要计算机传送信息。但是 A 国怕某些敏感信息被 B 国盗取，所以想了一个办法给信息加密：把每个字符向后顺移一个，比如 I am Tomato 变成了 J bn Upnbup。请利用这个原理和字符的存储原理，对 I am Tomato 加密，并能够将其还原。

②A、B、C、D 四个学校举行足球赛，比赛采用单循环制，即一共 6 场比赛，比分如下：A 对 B 为 2: 1，A 对 C 为 1: 4，A 对 D 为 2: 2，B 对 C 为 3: 1，B 对 D 为 4: 2，C 对 D 为 1: 1。请使用二维数组，统计出的胜利最多的球队、攻入球数最多的球队和净胜球最多的球队。

## 第八章 内存里的快捷方式

上一章我们学习了数组，了解了地址的概念。本章我们将继续深入学习地址，并引入指针这个概念。C++具有获取地址和操作地址的功能。然而这种功能是强大而又危险的。于是，指针成为了 C++中最难学好的部分。

### 8.1 什么是指针

在我们的桌面上，往往有这样一些图标：在它们的左下角有个小箭头，我们双击它，可以调出本机内的一些程序或文件。然而我们发现这些图标所占的存储空间很小，一般也就几百到几千字节。可是那么小的文件怎么会让上百兆的程序执行起来的呢？

后来，我们知道那些有小箭头的图标文件称为快捷方式。它所存储的内容并不是所要调用的程序本身，而是所要调用的程序在本机磁盘上的位置。（比如 D:\Tencent\QQ\QQ.exe，如图 8.1 所示）使用快捷方式的目的是为了快捷方便，不用去查找程序就能去执行它。不过如果所要调用的程序不存在或位置不正确，那么双击了这个快捷方式就会导致错误发生。



(图 8.1)

在内存中，可能会有一些数据，我们不知道它的变量名，却知道它在内存中的存储位置，即地址。我们能否利用快捷方式的原理对这些数据进行调用和处理呢？

很幸运，在 C++中，也可以给内存中的数据创建“快捷方式”，我们称它为指针(Pointer)。它和整型、字符型、浮点型一样，是一种数据类型。指针中存储的并不是所要调用的数据本身，而是所要调用的数据在内存中的地址。我们可以通过对指针的操作来实现对数据的调用和操作。

### 8.2 指针变量的定义和使用

上一节我们讲述了指针的概念，这节课我们来介绍一下如何在 C++中定义和使用指针变量。

#### 指针的类型

同变量的数据类型类似，指针也有类型。之所以指针会有类型，是为了符合对应的变量或常量数据类型。**要注意，指针本身也是一种数据类型。**

**不同指针类型的本质在于不同的操作。**这点和快捷方式是类似的。比如双击一个可执行文件(.EXE)快捷方式的操作是运行这个可执行文件，而双击一个 Word 文档文件(.DOC)快捷方式的操作是使用 Word 程序打开这个文档。类似地，一个字符型数据在内存中占用一个字节，那么读取数据就应以字符型数据读出一个字节；一个长整型数据在内存中占用四个字节，那么读取数据时就应以长整型数据读出四个字节。如果指针类型与它所指向的数据类

型不匹配，就可能对数据作出错误的操作。

## 指针变量的声明

指针变量也是一种变量。所以在使用之前，必须先声明。声明指针变量的语句格式为：

**指针的类型 \*指针变量名；**

其中，指针类型应该是与指针所指向的数据相符合的数据类型。比如 `int`、`char`、`float` 等等。`*`表示所要定义的是一个指针变量，而不是一个普通变量。指针变量名则应该遵循起变量名的一切规则。

例如：

```
char *cptr;//指向字符型变量的指针变量
```

```
int *iptr;//指向整型变量的指针变量
```

要注意，当我们要声明多个指针变量时，必须在每个指针变量名前加上`*`，例如：

```
int *iptr1,*iptr2,iptr3;//iptr1 和 iptr2 是指向整型变量的指针变量，而 iptr3 是整型变量
```

## 获取地址和指针变量初始化

我们已经声明了指针变量，那么我们如何获得数据在内存中的地址呢？

在 C++中，用`&`操作符可以获取变量或常量在内存中的地址，我们称之为取地址操作符。它的使用格式是：

**&变量名或常量名**

既然能够获取到数据所在的地址，我们就能把这个地址交给指针了。例如：

```
int c=9;
```

```
int *iptr=&c;//声明指向整型变量的指针变量，并作初始化
```

这时，我们称指针 `iptr` 指向了变量 `c`。在第三章我们说过，声明一个未经初始化的变量之后，它的值是无法确定的。所以如果声明了一个指针却不对它作初始化，那么它所指向的内容也是无法确定的，而这种情况是十分危险的。

## 特殊的值——NULL

没有初始化的指针变量是危险的。可是如果在声明变量之后，找不到合适的地址进行初始化，我们该怎么办呢？显然，随便找个地址对指针变量做初始化是不负责任的。

在这里，我们引入一个特殊的地址——`NULL`。它的意思是“空”，即指针没有指向任何东西。比如：

```
int *iptr=NULL;
```

要注意的是，C++是大小写敏感的，`NULL` 与 `null` 是不同的。所以，在使用的时候必须要大写。

## 指针的使用——间接引用

双击一个有效的快捷方式，就能够调用对应的文件，那么我们通过什么方法才能操作指针所指向的变量呢？

在这里，`*`又出现了，它称为间接引用操作符。其作用是获取指针所指向的变量或存储空间。间接引用的指针可以作为左值。（关于左值概念请参见第三章）具体的使用格式为：

**\*指针变量名**

下面，我们来看一段程序，实践一下如何使用指针变量：（程序 8.2）

```
#include "iostream.h"
int main()
{
 int i=3;
 int *iptr=&i;
 int **iptrptr=&iptr;//iptr 也是变量，也能够获取它的地址
 cout <<"Address of Var i=" <<iptr <<endl;//输出 iptr 存储的内容，即 i 在内存中的地址
 cout <<"Data of Var i=" <<*iptr <<endl;//输出 iptr 所指向的变量
 cout <<"Address of Pointer iptr=" <<iptrptr <<endl;//输出 iptr 在内存中的地址
 cout <<"Address of Var i=" <<*iptrptr <<endl;//输出 iptrptr 所指向的变量，即 iptr
 *iptr=2+*iptr;//*iptr 可以作左值
 cout <<"Data of Var i=" <<*iptr <<endl;
 return 0;
}
```

运行结果：

```
Address of Var i=0x0012FF7C
Data of Var i=3
Address of Pointer iptr=0x0012FF78
Address of Var i=0x0012FF7C
Data of Var i=5
```

通过运行结果，我们可以知道变量 i 在内存中的地址是 0012FF7C（前面的 0x 表示这是一个十六进制的数）；指针也是一种变量，在内存中也有地址；间接引用指针以后就和使用指针指向的变量一样方便。

试试看：

- 1、如果将程序 8.2 中的所有整型变量换成字符型变量（把对应的变量数据也换成字符），则执行后会有什么奇怪的现象？请根据第七章的知识，猜想产生这个奇怪现象的原因。
- 2、如果声明一个指针变量后不对其进行初始化，而是将其间接引用，作为左值修改它所指向的内存中的数据，会有什么结果产生？

**结论：在没有保护措施的操作系统中，这样的操作可能会导致系统错误甚至崩溃。**

- 3、能否将一个常量的地址赋值给一个对应类型的指针变量？

**结论：将一个常量的地址赋给指针变量可能导致通过指针更改常量，所以是不合法的。**

## 8.3 指针的操作

既然指针是一种数据类型，那么它也应该有对应的操作或运算，正如整数能做加减乘除一样。但是每一种操作或运算都应该对这种数据类型有意义。比如两个实数可以用关系运算得知哪个大哪个小，而两个虚数却不能使用关系运算，因为比较虚数的大小是没有意义的。

对于指针类型来说，可以使用的运算有：**和整数做加法运算、和整数做减法运算、两指针做关系运算**。很显然，指针类型的乘除法是没有意义的，也是不允许的。

## 指针的加减运算

指针的加减法和数的加减法是不同的。我们认为，指针只能够和整数做加减法运算（包括和整型常量、变量做加减法和自增自减）。其实这也不难理解，内存的存储空间是按“个”计算的，不会出现半个存储空间的情况。那么，指针的加减法是否在地址值上做加减呢？我们先写一段程序来验证一下指针加减法的运算结果：（程序 8.3）

```
#include "iostream.h"
int main()
{
 int a[5]={1,2,3,4,5};
 int *aptr=&a[0];//把数组首元素的地址给指针
 int i=1;
 for (int j=0;j<5;j++)
 {
 cout <<(' <<aptr <<")=" <<*aptr <<endl;//输出指针内存储的地址和该地址的数据
 aptr=aptr+i;//指针和整型变量做加法
 }
 return 0;
}
```

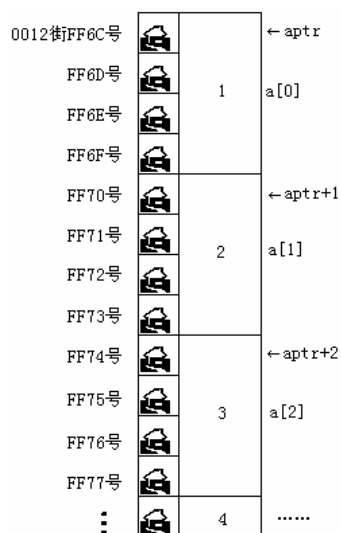
运行结果：

```
(0x0012FF6C)=1
(0x0012FF70)=2
(0x0012FF74)=3
(0x0012FF78)=4
(0x0012FF7C)=5
```

我们发现，每次做了加法以后，地址值并不是相差 1，而是相差了 4。所以指针和整数做加法并不是简单地将地址值和整数相加。我们又发现，每次做了加法以后，能够输出原先指针所指的下一个元素。根据数组在内存中的存储情况我们不难得出这样一个结论：**指针和整数 C 的加减法是指针向前或向后移动 C 个对应类型的存储区域，即可以得到以下公式：**

**新地址=旧地址±C\*每个对应数据类型的变量在内存中所占字节数**

因为每个 int 变量在内存中所占字节数为 4，所以在程序 8.3.1 中每做完一次加法，新地址=旧地址+1\*4=旧地址+4。如右上图 8.3 所示。



（图 8.3）

## 指针的关系运算

在第四章中，我们知道关系运算有等于、大于、小于、大于等于、小于等于和不等六种。对于指针来说，等于和不等就是判断两个指针的值是否相同或不同，即两个指针是否指向了相同或不同的地方。而大于和小于则是判断指针的值哪个大哪个小。值较小的在存储器中的位置比较靠前，值较大的在存储器中的位置比较靠后。

指针的关系运算在数据结构中会经常用到。我们在下一章介绍链表的时候会着重介绍它



的用法。

## 8.4 指针与保护

在第二节，我们说到了，如果没有对指针进行初始化就改变它所指向的内存里的数据是非常危险的。因为不确定的指针有可能指向了一个系统需要的关键数据，数据一旦被更改或破坏系统就会崩溃。在我们使用电脑磁盘的时候，都知道有一种措施叫做“写保护”。（或者称为“只读”，即只可以读，不可以写）那么，我们能否给指针加上写保护呢？

### 对内存只读的指针

为了解决指针可能误修改或破坏内存中的数据，我们可以对内存中的数据加上写保护。即具有这种属性的指针只能够读出内存中的数据，却不能修改内存中的数据。我们把具有这种属性的指针称为指向常量的指针，简称常量指针。

给内存中的数据加写保护的方法是在声明指针时，在指针类型（即各种数据类型）前加上 `const`，表示这些数据是常量，只能读不能写。比如：`const int *iptr`，这时候我们只能够通过指针 `iptr` 读出内存里的数据，但是不能对其写入、修改。

当然，这样的保护措施只是防止通过间接引用 `iptr` 修改内存中的数据，并不保护 `iptr` 本身和 `iptr` 所指向的变量。

试试看：

能否将一个常量的地址赋值给一个对应类型的常量指针？

结论：因为常量指针不能通过指针修改内存里的数据，所以将常量的地址赋值给常量指针是合法的。

### 指针型常量

我们说，指针同整型，字符型一样，是一种数据类型。整型可以有整型常量和整型变量；字符型可以有字符型常量和字符型变量。那么，指针也应该有指针常量。指针常量和常量指针不同，指针常量是指正所指向的位置不能改变，即指针本身是一个常量。但是指针常量可以通过间接引用修改内存中的数据。定义指针常量的语句格式为：

指针类型 \* `const` 指针常量名=&变量名；

下面我们来看一段程序，实践一下指针常量和常量指针：（程序 8.4）

```
#include "iostream.h"
```

```
int main()
```

```
{
```

```
 int a=42;
```

```
 const int b=84;
```

```
 const int *captr=&a;//常量指针
```

```
 int * const acptr=&a;//指针常量
```

```
 int *bptr=&b;//错误，不能把常量的地址给指针变量
```

```
 const int *cbprt=&b;//把常量的地址给常量指针是允许的
```

```
 *captr=68;//错误，间接引用常量指针不可修改内存中的数据
```

```

*acptr=68;//间接引用指针常量可以修改内存中的数据
captr=&b;//常量指针可以指向其他变量
acptr=&b;//错误，指针常量不能指向别的变量
const int * const ccaptr=&a;//常量指针常量，既不能间接引用修改数据，也不能指向别的
变量或常量
*ccaptr=68;//错误，不能间接引用修改数据
ccaptr=&b;//错误，不能指向别的常量或变量
return 0;
}

```

以上程序存在错误，无法通过编译。我们考虑到使用指针的安全性的时候，就能想到要使用以上这些保护措施保护内存中的数据。

## 8.5 指针与数组

我们在上一章说到，在向函数传递数组参数的时候，实质上是传递数组首元素的地址。那么数组和指针有着什么样的关系吗？

### 数组名的实质

数组名并不是一个普通的变量，而是一个指向数组首元素的指针。也就是说，我们可以用数组名来初始化一个对应类型的指针。既然如此，经过初始化的指针能否代替原来的数组名呢？答案是肯定的。

下面我们来看一段程序，了解数组名和指针的用法：（程序 8.5）

```

#include "iostream.h"
int main()
{
 int a[6]={5,3,4,1,2,6};
 int *aptr=a;
 for (int i=0;i<6;i++)
 {
 cout <<a[i] <<aptr[i] <<*(aptr+i) <<*(a+i) <<endl;
 }
 return 0;
}

```

运行结果：

```

5555
3333
4444
1111
2222
6666

```

根据上面的这段程序，可以知道 `a[i]`、`aptr[i]`、`*(aptr+i)`、`*(a+i)`都能够访问到数组的元素。所以，我们说上述四者是等价的。虽然数组名是指针，但它是一个指针常量。也就是说，不带下标的数组名不能作为左值。

## 指针数组

指针是一种数据类型，所以，我们可以用指针类型来创建一个数组。声明一个指针数组的语句格式是：

**指针类型 \* 数组名[常量表达式];**

对指针数组的操作和对指针变量的操作并无不同，在此不作赘述。

试试看：

如果有两个大小相同的数组 a 和 b，我们想把数组 b 的元素都一一复制到数组 a 中，通过一句 a=b;语句是否可以实现？试用本节的知识解释这一现象。

**结论：数组名 a 和 b 都是指向数组首元素的指针，所以 a=b;并不能把数组 b 的数据复制给数组 a。**

## 8.6 指针与函数

指针在函数中的使用也是十分广泛的。某些情况下，将指针作为函数的参数或函数的返回值会给我们带来方便。而某些情况下，我们又不得不将指针作为函数的参数或函数的返回值。

### 指针作为参数

我们在上一章我们已经了解向函数传递数组的实质是向函数传递数组首元素的地址。我们又知道数组名是一个指向数组首元素的指针常量。所以我们认为，**向函数传递数组是将指针作为参数的特殊形式。**

由于指针可以直接操作内存中的数据，所以它可以用来修改实参。这个功能和引用是类似的。

下面我们来看一段程序，了解指针作为参数时的上述两个特点：（程序 8.6.1）

```
#include "iostream.h"
void arrayCopy(int *src,int *dest,int size);//复制数组元素
void display(const int *array,int size);//输出数组元素
int main()
{
 int a[]={3,4,5,6,3,1,6};
 int b[7];
 arrayCopy(a,b,sizeof(a)/sizeof(int));//把数组 a 的元素依次复制到数组 b 中
 cout <<"The data of array a is:";
 display(a,sizeof(a)/sizeof(int));
 cout <<"The data of array b is:";
 display(b,sizeof(b)/sizeof(int));
 return 0;
}
void arrayCopy(int *src,int *dest,int size)
{
```

```

 for (int i=0;i<size;i++)
 {
 dest[i]=src[i];//修改了实参数组元素
 }
 cout <<size <<" data Copied." <<endl;
}
void display(const int *array,int size)//const 用来保护指针指向的数据
{
 for (int i=0;i<size;i++)
 {
 cout <<array[i] <<" ";
 }
 cout <<endl;
}

```

运行结果：

7 data Copied.

The data of array a is:3 4 5 6 3 1 6

The data of array b is:3 4 5 6 3 1 6

根据 arrayCopy 函数，不难看出传递数组和传递指针是完全相同的。而通过指针的间接引用或数组操作，我们可以在函数内实现对实参的修改。这就是 arrayCopy 函数能够实现复制功能的原因。

不过，将指针作为函数参数的副作用仍然不容我们忽视。**指针和引用虽然都能够修改实参，但是指针却更加危险。**因为引用仅限于修改某一个确定的实参，而指针却可以指向内存中的任何一个数据，通过间接引用就能够在函数内修改函数外甚至系统中的数据了。这样一来，函数的黑盒特性就被破坏了，系统也因此变得不再安全。对于程序员来说，将指针作为函数参数可能把函数内的问题引到函数外面去，使得调试程序变得非常困难。所以，我们要认识到使用指针的两面性，谨慎对待指针做函数参数。

为了避免指针作为函数参数导致数据被意外修改，我们可以使用 **const** 来保护指针指向的数据，如程序 8.6.1 中的 **display** 函数。

## 指针作为返回值

和别的数据类型一样，指针也能够作为函数的一种返回值类型。**我们把返回指针的函数称为指针函数。**在某些情况下，函数返回指针可以给我们设计程序带来方便。而且此时通过间接引用，函数的返回值还可以作为左值。

下面我们来看一段程序，了解函数如何返回指针：（程序 8.6.2）

```

#include "iostream.h"
int * max(int *array,int size);//返回值类型是 int *，即整型指针
int main()
{
 int array[]={5,3,6,7,2,1,9,10};
 cout <<"The Max Number is " <<*max(array,sizeof(array)/sizeof(int)) <<endl;//间接引用返回的指针
 return 0;
}

```

```
int * max(int *array,int size)//寻找最大值
{
 int *max=array;
 for (int i=0;i<size;i++)
 {
 if (array[i]>*max)
 max=&array[i];//记录最大值的地址
 }
 return max;
}
```

运行结果:

The Max Number is 10

需要注意的是，返回的指针所指向的数据不能够是函数内声明的变量。道理很简单，我们在第六章已经说明，一个函数一旦运行结束，在函数内声明的变量就会消失。就好像下课同学们都走了，教室里的某一个座位到底有没有坐着谁我们无法确定。所以指针函数必须返回一个函数结束运行后仍然有效的地址值。

## 8.7 更灵活的存储

家里要来客人了，我们要给客人们泡茶。如果规定只能在确定来几位客人之前就把茶泡好，这就会显得很尴尬：茶泡多了会造成浪费，泡少了怕怠慢了客人。所以，最好的方法就是等知道了来几位客人再泡茶，来几位客人就泡几杯茶。

然而，我们在使用数组的时候也会面临这种尴尬：**数组的存储空间必须在程序运行前申请，即数组的大小在编译前必须是已知的常量表达式。**空间申请得太大会造成浪费，空间申请得太小会造成数据溢出而使得程序异常。所以，为了解决这个问题，我们需要能够在程序运行时根据实际情况申请内存空间。

在 C++中，允许我们在程序运行时根据自己的需要申请一定的内存空间，我们把它称为堆内存（Heap）空间。

### 如何获得堆内存空间

我们用操作符 `new` 来申请堆内存空间，其语法格式为：

**new 数据类型[表达式];**

其中，表达式可以是一个整型正常量，也可以是一个有确定值的整型正变量，其作用类似声明数组时的元素个数，所以两旁的中括号不可省略。如果我们只申请一个变量的空间，则该表达式可以被省略，即写作：

**new 数据类型;**

使用 `new` 操作符后，会返回一个对应数据类型的指针，该指针指向了空间的首元素。所以，我们在使用 `new` 操作符之前需要声明一个对应类型的指针，来接受它的返回值。如下面程序段：

```
int *iptr;//声明一个指针
int size;//声明整型变量，用于输入申请空间的大小
cin >>size;//输入一个正整数
iptr=new int[size];//申请堆内存空间，接受 new 的返回值
```

我们又知道，数组名和指向数组首元素的指针是等价的。所以，对于 `iptr` 我们可以认为是一个整型数组。于是，我们实现了在程序运行时，根据实际情况来申请内存空间。

## 有借有还，再借不难

当一个程序运行完毕之后，它所使用的数据就不再需要。由于内存是有限的，所以它原来占据的内存空间也应该释放给别的程序使用。对于普通变量和数组，在程序结束运行以后，系统会自动将它们的空间回收。然而对于我们自己分配的堆内存空间，大多数系统都不会将它们回收。如果我们不人为地对它们进行回收，只“借”不“还”，那么系统资源就会枯竭，电脑的运行速度就会越来越慢，直至整个系统崩溃。我们把这种只申请空间不释放空间的情况称为内存泄露（Memory Leak）。

确认申请的堆内存空间不再使用后，我们用 `delete` 操作符来释放堆内存空间，其语法格式为：

**`delete []` 指向堆内存首元素的指针；**

如果申请的是一个堆内存变量，则 `delete` 后的 `[]` 可以省略；如果申请的是一个堆内存数组，则该 `[]` 不能省略，否则还是会出现内存泄露。另外，我们也不难发现，`delete` 后的指针就是通过 `new` 获得的指针，如果该指针的数据被修改或丢失，也可能造成内存泄露。

下面我们来看一段程序，实践堆内存的申请和回收：（程序 8.7）

```
#include "iostream.h"
int main()
{
 int size;
 float sum=0;
 int *heapArray;
 cout <<"请输入元素个数: ";
 cin >>size;
 heapArray=new int[size];
 cout <<"请输入各元素: " <<endl;
 for (int i=0;i<size;i++)
 {
 cin >>heapArray[i];
 sum=sum+heapArray[i];
 }
 cout <<"这些数的平均值为" <<sum/size <<endl;
 delete [] heapArray;
 return 0;
}
```

运行结果：

请输入元素个数：5

请输入各元素：

1 3 4 6 8

这些数的平均值为 4.4

可见，申请的堆内存数组在使用上和一般的数组并无差异。我们需要记住的是，申请了资源用完了就一定要释放，这是程序员的好习惯，也是一种责任。

那么，我们能不能来申请一个二维的堆内存数组呢？事实上，`new 数据类型[表达式][表`

达式]的写法是不允许的。所以，如果有需要，最简单的方法就是用一个一维数组来代替一个二维数组。这就是上一章最后一小段文字的意义所在。

试试看：

- 1、如果输入的元素个数为 0 会有什么情况？如果输入一个负数呢？
- 2、如果申请了堆内存，却没有释放，对小型程序的运行是否有影响呢？
- 3、如果输入的元素个数是一个很大的数，会发生什么情况？比如 1000000000。

## 习题

1、说出下列语句中包含的指针类型。

- ① `char *cptr;`
- ② `const int *iptr;`
- ③ `float array[4];`
- ④ `int **ipptr;`

2、分析下列函数原型中的参数和返回值情况。

- ① `int max(int *array, int size);`
- ② `char * str(const char *string1, const char *string2);`
- ③ `void main(char * argv[]);`

3、指出下列程序的错误之处。

①

```
#include "iostream.h"
int main()
{
 int *iptr;
 for (int i=0;i<6;i++)
 {
 cin >>iptr[i];
 }
 for (int j=5;j>=0;j--)
 {
 cout <<iptr[j];
 }
 return 0;
}
```

②

```
#include "iostream.h"
void func1(int *array);
void func2(int *array);//功能为输出数组元素
int main()
{
 int array[8];
 func1(array);
}
```

```

 func2(array);
 return 0;
}
void func1(int *array)
{
 int size=sizeof(array)/sizeof(int);
 for (int i=0;i<size;i++)
 {
 array[i]=i;
 }
}
void func2(int *array)
{
 int size=sizeof(array)/sizeof(int);
 for (int i=0;i<size;i++)
 {
 cout <<array++;
 }
}

```

③

```

#include "iostream.h"
int main()
{
 int *iptr=NULL;
 int size;
 cin >>size;//输入的 size 是一个不太大的正整数
 iptr=new int [size];
 for (int i=0;i<size;i++)
 {
 cin >>(*iptr++);
 }
 delete iptr;
 return 0;
}

```

4、根据运行结果完成代码。

```

#include "iostream.h"
void strcpy(char *string1,char *string2);
int main()
{
 char str1[]={ "Tomato Studio"};
 char *str2;
 int size=_____ ;
 str2=_____ ;
 cout <<"STR1 的内容是" <<str1 <<endl;
}

```



```

strcpy(str1,str2);
cout <<"String Copied..." <<endl;
cout <<"STR2 的内容是" <<str2 <<endl;

return 0;
}
void strcpy(char *string1,char *string2)
{
for (char *temp=_____;*temp!='\0';_____)
{

string2++;
}

}

```

运行结果:

STR1 的内容是 Tomato Studio

String Copied...

STR2 的内容是 Tomato Studio

5、根据要求编写程序。

①字符串常量的实质也是指针，比如“Tomato”是一个指向首字母“T”的字符指针，并且字符串常量最后以结尾符'\0'结尾。试编写一段程序，分别比较出两组字符串常量“Tomato”和“Studio”，“SHU”和“Shanghai Univesity”的较长者，并将其输出。如果两个字符串一样长，则都输出。

运行结果示例:

Tomato Studio

Shanghai University

②我们知道无法直接通过 new 来申请一个二维的堆内存数组，于是有人想出了这样一个办法：创建一个一维堆内存指针数组，即每个数组元素是一个指针，然后用 new 给各个指针分配一个一维的堆内存数组，那么最后表示出来就像是一个二维的堆内存数组了。试编写一段程序，依照以上方法实现一个大小为 8×8 的二维堆内存数组，数据类型为整型，并将数组元素依次赋值、输出。

运行结果示例:

```

0 1 2 3 4 5 6 7
8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63

```

## 第九章 自己设计的箱子

在第三章我们已经介绍了一些 C++ 中常用的数据类型。然而，多彩的世界仅靠这些数据来描述显然是不够的。C++ 允许用户自己来设计一些数据类型。在本章，我们要介绍枚举型数据和结构型数据，并介绍一下链表实例，为学习以后的数据结构打好基础。

### 9.1 我的类型我做主

在基本的数据类型中，无外乎就是些数字和字符。但是某些事物是较难用数字和字符来准确地表示的。比如一周有七天，分别是 Sunday、Monday、Tuesday、Wednesday、Thursday、Friday 和 Saturday。如果我们用整数 0、1、2、3、4、5、6 来表示这七天，那么多下来的那些整数该怎么办？而且这样的设置很容易让数据出错，即取值超出范围。我们能否自创一个数据类型，而数据的取值范围就是这七天呢？

C++ 中有一种数据类型称为枚举（Enumeration）类型，它允许用户自己来定义一种数据类型，并且列出该数据类型的取值范围。

我们说变量就好像是一个箱子，而数据类型就好像是箱子的类型，所以我们在创建某个枚举类型的变量的时候，必须先把这个枚举类型设计好，即把箱子的类型设计好。定义枚举类型的语法格式为：

```
enum 类型名{常量1,常量2,……常量n};
```

定义枚举类型的位置应该在程序首次使用该类型名之前，否则程序无法识别该类型。枚举类型中我们列出的常量称为枚举常量。它并不是字符串也不是数值，而只是一些符号。

如果我们要定义一周七天的日期类型，可以这样写：

```
enum day {Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
```

这时候，程序中就有了一种新的数据类型——day，它的取值范围就是 Sunday 到 Saturday 的那七天。我们已经把类型设计好，下面我们就能来创建一个 day 类型的变量了：

```
day today;
```

```
today=Sunday;
```

这样，day 类型的变量 today 的值就是 Tuesday 了。

下面我们来写一段程序来运用一下枚举类型的数据：（程序 9.1）

```
#include "iostream.h"
```

```
enum day {Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday};
```

```
void nextday(day &D);//向后一天是星期几，参数为 day 类型，是程序中首次使用该类型名
```

```
void display(day D);//显示某一天是星期几
```

```
int main()
```

```
{
```

```
 day today=Sunday;
```

```
 for (int i=0;i<7;i++)
```

```
 {
```

```
 cout <<"Data in today=" <<today <<endl;
```

```
 display(today);
```

```
 nextday(today);
```

```
 }
 return 0;
}
void nextday(day &D)
{
 switch(D)
 {
 case Sunday:
 D=Monday;
 break;
 case Monday:
 D=Tuesday;
 break;
 case Tuesday:
 D=Wednesday;
 break;
 case Wednesday:
 D=Thursday;
 break;
 case Thursday:
 D=Friday;
 break;
 case Friday:
 D=Saturday;
 break;
 case Saturday:
 D=Sunday;
 break;
 }
}
void display(day D)
{
 switch(D)
 {
 case Sunday:
 cout <<"Sunday" <<endl;
 break;
 case Monday:
 cout <<"Monday" <<endl;
 break;
 case Tuesday:
 cout <<"Tuesday" <<endl;
 break;
 case Wednesday:
```

```
 cout <<"Wednesday" <<endl;
 break;
 case Thursday:
 cout <<"Thursday" <<endl;
 break;
 case Friday:
 cout <<"Friday" <<endl;
 break;
 case Saturday:
 cout <<"Saturday" <<endl;
 break;
 }
}
```

运行结果:

Data in today=0

Sunday

Data in today=1

Monday

Data in today=2

Tuesday

Data in today=3

Wednesday

Data in today=4

Thursday

Data in today=5

Friday

Data in today=6

Saturday

根据运行结果, 我们发现在 `day` 型变量 `today` 中保存的竟然是整数! 也就是说, 一个整数和一个枚举常量一一对应了起来, 要注意是一一对应, 而不是相等。但是如果我们把整数直接赋值给 `today` 变量, 则会发生错误。虽然枚举类型的实质是整数, 但是电脑还是会仔细检查数据类型, 禁止不同数据类型的数据互相赋值。另外, 在一般情况下, 枚举类型是不能进行算术运算的。

试试看:

1、 在定义一个枚举类型时, 能否有两个相同的枚举常量?

**结论: 不能, 否则会出现重复定义的错误。**

2、 在定义两个不同的枚举类型时, 能否有两个相同的枚举常量?

**结论: 不能, 否则会出现重复定义的错误。**

3、 定义的枚举类型名能否和某一个变量名或者函数名相同?

## 9.2 设计一个收纳箱

学校要统计学生情况，于是 Tomato 同学给出了一张自己的信息表：

|      |                 |
|------|-----------------|
| 学号   | 428004          |
| 姓名   | Tomato          |
| 年龄   | 20              |
| 院系   | ComputerScience |
| 平均成绩 | 84.5            |

从上表来看，我们需要两个字符串分别用来存储姓名，需要两个整型变量分别来存储学号和年龄，还需要一个浮点型变量来存储平均成绩。一个学生已经需要至少 5 个存储空间，更何况一个学校有几千个学生，那将需要几万个存储空间。如果有这么多的变量，显然是很难管理的。

我们把变量比作为箱子。在现实生活中，如果小箱子太多太杂乱了，我们会拿一个大收纳箱来，把小箱子一个个有序地放到收纳箱里面。这样一来，在我们视线里的箱子就变少了，整理起来也会比较方便。那么，我们能否把这么多凌乱的变量整理到一个变量当中呢？

C++中有一种数据类型称为结构(Structure)类型，它允许用户自己定义一种数据类型，并且把描述该类型的各种数据类型一一整合到其中。

|    |           |
|----|-----------|
| 学生 | 学号——整型    |
|    | 姓名——字符串   |
|    | 年龄——整型    |
|    | 院系——字符串   |
|    | 平均成绩——浮点型 |

如上表所示，每个学生的信息成为了一个整体。一个学生拥有学号、姓名、年龄、院系和平均成绩这五项属性，我们把这些属性称为这个结构类型的成员数据(Data Member)。每项属性的数据类型也在旁边做了说明。这样一来，杂乱的数据和每个学生一一对应了起来，方便了我们管理。

定义一种结构类型的语法格式为：

```
struct 结构类型名
{
 数据类型 成员数据1;
 数据类型 成员数据2;

 数据类型 成员数据n;
};
```

和定义枚举类型类似，定义结构类型的位置必须在首次使用该类型名之前，否则程序将无法正确识别该类型。要注意，定义完结构类型后的分号是必不可少的，否则将会引起错误。如果我们要创建前面的学生类型，可以写作：

```
struct student
{
 int idNumber;
 char name[15];
 int age;
 char department[20];
};
```

```
float gpa;
};
```

这时候，就有了一个新的数据类型，称为 student。我们要用这种 student 类型来创建一个变量，并可以依次对它的成员数据进行初始化：

```
student s1={428004, "Tomato", 20, "ComputerScience", 84.5};
```

这样就有了一个 student 类型的变量 s1。s1 有五项属性，它们应该怎么表达呢？如果用自然语言描述，我们会说 s1 的 idNumber、s1 的 name 等等。在 C++ 中，我们用一点“.”来表示“的”，这个“.”称为成员操作符。

下面我们就来看一段程序，了解结构类型的基本使用：（程序 9.2）

```
#include "iostream.h"
struct student
{
 int idNumber;
 char name[15];
 int age;
 char department[20];
 float gpa;
};
int main()
{
 student s1,s2;//首次使用 student 类型名，定义必须在这之前。
 cout <<"输入学号： ";
 cin >>s1.idNumber;//成员数据可以被写入
 cout <<"输入姓名： ";
 cin >>s1.name;
 cout <<"输入年龄： ";
 cin >>s1.age;
 cout <<"输入院系： ";
 cin >>s1.department;
 cout <<"输入成绩： ";
 cin >>s1.gpa;
 cout <<"学生 s1 信息：" <<endl <<"学号：" <<s1.idNumber <<"姓名：" <<s1.name <<"
 年龄：" <<s1.age <<endl <<"院系：" <<s1.department <<"成绩：" <<s1.gpa <<endl;//成员数据
 也能够被读出
 s2=s1;//把 s1 的给各个成员数据值分别复制到 s2 中
 cout <<"学生 s2 信息：" <<endl <<"学号：" <<s2.idNumber <<"姓名：" <<s2.name <<"
 年龄：" <<s2.age <<endl <<"院系：" <<s2.department <<"成绩：" <<s2.gpa <<endl;
 return 0;
}
```

运行结果：

输入学号： 428004

输入姓名： Tomato

输入年龄： 20

输入院系： ComputerScience

输入成绩: 84.5

学生 s1 信息:

学号: 428004 姓名: Tomato 年龄: 20

院系: ComputerScience 成绩: 84.5

学生 s2 信息:

学号: 428004 姓名: Tomato 年龄: 20

院系: ComputerScience 成绩: 84.5

我们看到,结构的成员数据是既可以被读出,也可以被写入的。而且,相同类型的结构变量还能够用一个赋值操作符“=”把一个变量的内容赋值给另一个变量。

试试看:

下列情况中的两个结构变量是否能够通过赋值操作符来进行互相赋值?

- (1) 两者类型名不同,成员数据的个数和名称不同,部分成员数据的数据类型相同。
- (2) 两者类型名不同,成员数据的个数相同,所有成员数据的数据类型相同。
- (3) 两者类型名不同,成员数据的个数和名称都相同,所有成员数据的数据类型相同。

**结论:** 只要两者类型名不同,就不能用赋值操作符互相赋值。

## 9.3 结构与函数

结构也可以用作函数参数或返回值。

### 结构作为参数

我们在前面的一些章节中知道,变量作为函数的参数,了解它是值传递还是地址传递是非常重要的。因为这意味着参数在函数体内的修改是否会影响到该变量本身。

不同于数组,结构是按值传递的。也就是说整个结构的内容都复制给了形参,即使某些成员数据是一个数组。

下面,我们就以一个实例来证明这一点:(程序 9.3.1)

```
#include "iostream.h"
struct student
{
 int idNumber;
 char name[15];
 int age;
 char department[20];
 float gpa;
};
void display(student arg);//结构作为参数
int main()
{
 student s1={428004, "Tomato", 20, "ComputerScience", 84.5};//声明 s1, 并对 s1 初始化
 cout <<"s1.name 的地址" <<&s1.name <<endl;
 display(s1);
}
```

```

 cout <<"形参被修改后……" <<endl;
 display(s1);
 return 0;
}
void display(student arg)
{
 cout <<"学号: " <<arg.idNumber <<"姓名: " <<arg.name <<"年龄: " <<arg.age <<endl <<"
院系: " <<arg.department <<"成绩: " <<arg.gpa <<endl;
 cout <<"arg.name 的地址" <<&arg.name <<endl;
 for (int i=0;i<6;i++)//企图修改参数的成员数据
 {
 arg.name[i]='A';
 }
 arg.age++;
 arg.gpa=99.9f;
}

```

运行结果:

```

s1.name 的地址 0x0012FF54
学号: 428004 姓名: Tomato 年龄: 20
院系: ComputerScience 成绩: 84.5
arg.name 的地址 0x0012FED8
形参被修改后……
学号: 428004 姓名: Tomato 年龄: 20
院系: ComputerScience 成绩: 84.5
arg.name 的地址 0x0012FED8

```

通过上面这个程序，我们发现在函数中修改形参的值对实参是没有影响的。并且通过输出变量 `s1` 和参数 `arg` 的成员数据 `name` 所在地址，我们可以知道两者是不相同的，即整个 `name` 数组也复制给了参数 `arg`。

如果我们希望能在函数修改实参，则可以使用引用的方法。由于结构往往整合了许多的成员数据，它的数据量也绝对不可小觑。使用值传递虽然能够保护实参不被修改，但是却或多或少地影响到程序的运行效率。所以，一般情况下，我们选择引用传递的方法。

关于引用传递，请参见第六章的相关内容，在此不作赘述。

## 结构作为返回值

一般情况下，函数只能返回一个变量。如果要尝试返回多个变量，那么就要通过在参数中使用引用，再把实参作为返回值。然而，这种方法会导致一大堆参数，程序的可读性也较差。

当结构出现以后，我们可以把所有需要返回的变量整合到一个结构中来，问题就解决了。我们通过一段程序来了解如何让函数返回一个结构：（程序 9.3.2）

```

#include "iostream.h"
struct student
{
 int idNumber;
 char name[15];
}

```



```

 int age;
 char department[20];
 float gpa;
};
student initial();//初始化并返回一个结构
void display(student arg);
int main()
{
 display(initial());//输出返回的结构
 return 0;
}
void display(student arg)
{
 cout <<"学号: " <<arg.idNumber <<"姓名: " <<arg.name <<"年龄: " <<arg.age <<endl <<"
院系: " <<arg.department <<"成绩: " <<arg.gpa <<endl;
}
student initial()
{
 student s1={428004, "Tomato",20, "ComputerScience",84.5};//初始化结构变量
 return s1;//返回结构
}

```

运行结果:

学号: 428004 姓名: Tomato 年龄: 20

院系: ComputerScience 成绩: 84.5

## 9.4 结构数组与结构指针

结构是一种数据类型，因此它也有对应的结构数组和指向结构的指针。

### 结构数组

定义结构数组和定义其他类型的数组在语法上并无差别。需要注意的是，在定义结构数组之前，我们必须先定义好这个结构。比如：

```

struct student
{
 int idNumber;
 char name[15];
 int age;
 char department[20];
 float gpa;
};
.....
student S[3]={ {428004, "Tomato",20, "ComputerScience",84.5},
 {428005, "OOTTMA",20, "ComputerScience",85.0},

```

```
{428006, "OTA", 20, "ComputerScience", 89.8}};
```

.....

使用结构数组只要遵循结构和数组使用时的各项规则即可，在此不作赘述。

## 结构指针

在上一章我们了解到指针的一个重要作用就是实现内存的动态分配（堆内存）。待我们学完了这一章，我们会发现结构指针也是一个非常有用的工具。

所谓结构指针就是指向结构的指针。定义好一个结构之后，定义一个结构指针变量的语法格式为：

**结构类型名 \*指针变量名；**

我们知道一般的指针是通过间接引用操作符“\*”来访问它指向的变量。那么我们如何访问结构指针所指向的变量的成员数据呢？这里要介绍箭头操作符“->”，我们用它可以访问到指针指向的变量的成员数据。它的格式为：

**指针变量名->成员数据**

需要注意的是，箭头操作符的左边一定是一个结构指针，而成员操作符的左边一定是一个结构变量，两者不能混淆使用。

下面我们来看一段程序，掌握如何使用结构指针：（程序 9.4）

```
#include "iostream.h"
struct student
{
 int idNumber;
 char name[15];
 int age;
 char department[20];
 float gpa;
};
void display(student *arg); //结构指针作为函数参数
int main()
{
 student s1={428004, "Tomato", 20, "ComputerScience", 84.5}; //初始化结构变量
 student *s1ptr=&s1; //定义结构指针变量，并把 s1 的地址赋值给 s1ptr
 display(s1ptr);
 return 0;
}
void display(student *arg)
{
 cout <<"学号: " <<arg->idNumber <<"姓名: " <<arg->name <<"年龄: " <<arg->age <<endl
 <<"院系: " <<arg->department <<"成绩: " <<arg->gpa <<endl; //用箭头操作符访问成员数据
}
```

运行结果：

学号： 428004 姓名： Tomato 年龄： 20

院系: ComputerScience 成绩: 84.5

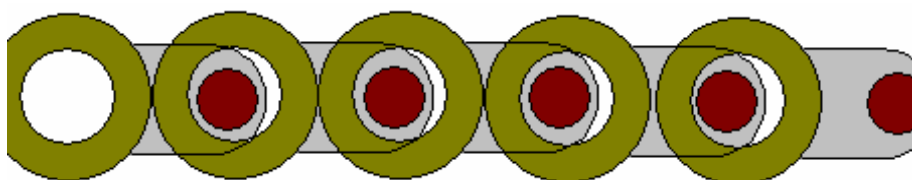
试试看:

1、间接引用指针得到的是一个变量，比如：`int b=&a`，那么 `*b` 就是变量 `a`。根据这个思路，我们能否不使用箭头操作符，而是使用成员操作符来访问一个结构指针所指向的变量的成员数据呢？（提示：在必要的地方加上括号）

2、能否有这样一种类型的结构：结构的成员数据之一是指向这种结构的结构指针，想想看这种结构能够造成一种什么现象。

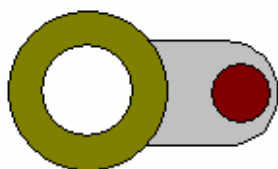
## 9.5 自行车的链条

大家都知道自行车，可是你有没有仔细观察过自行车的链条呢？如下图 9.5.1 就是一段自行车链条的样子。



(图 9.5.1)

我们发现，自行车的链条虽然很长，却是由一个个相同的小环节连接而成的。如左下图 9.5.2 所示。每个环节又可以分成两部分：一部分是一个铁圈，让别的环节能够连接它；另一部分则是一个铁拴，可以去连接别的环节。于是，将这些环节一一连接起来，就形成了长长的链条。



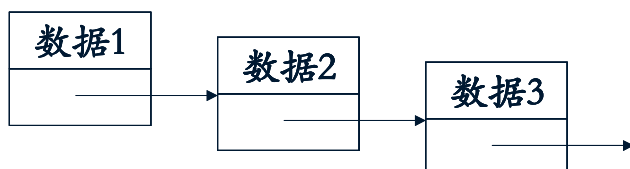
(图 9.5.2)

这时候，我们想到了这样一种结构：

```
struct node
{
 int data;
 node *next;
};
```

这个结构有两个成员数据，一个是整数 `data`，另外一个是指向这种结构的指针 `next`。那么如果有若干个这样的结构

变量，就能像自行车链条一样，把这些变量连接成一条链子。如左下图 9.5.3 所示。



(图 9.5.3)



(图 9.5.4)

我们把这些利用结构指针连接起来的结构变量称为链表 (Link List)，每一个结构变量 (相当于链条中的每个环节) 称为链表的结点 (Node)。如右上图 9.5.4 所示。

和数组一样，链表也可以用来存储一系列的数据，它也是电脑中存储数据的最基本的结构之一。然而，我们已经拥有了数组，也了解了数组的动态分配 (堆内存)，我们为什么还

需要链表呢？

相信很多人都玩过即时战略游戏（RTS），比如时下流行的魔兽争霸、曾红极一时的红色警戒。可是大家有没有考虑过，每个战斗单位都有它们各自的属性，电脑又是如何为我们造出来的部队分配内存的呢？

显然，部队的数量在程序执行之前是未知的。如果用数组来存储这些数据，那么就会造成游戏前期浪费内存（没有那么多的部队），游戏后期存储空间不够（战斗单位数大大增加）的情况。

那么使用数组的动态分配行不行呢？还是不行。因为部队的数量在程序执行的时候仍然是未知的。甚至连玩家自己也不知道要造多少战斗单位，只是根据战斗的实际情况来发展自己的势力。所以，这时候最合理的内存分配方式就是每造一个战斗单位分配一个内存空间。

然而，新问题又出现了：建造各单位的时间一般不可能是完全连续的，根据不同时刻程序运行的实际情况，每个单位分配到的内存空间也不是连续的了。空间不连续就意味着没有了方便的数组下标。我们就很难把这些零零散散的内存空间集中起来管理。

链表的出现改变了这个情况。它可以在程序运行时根据实际需要一个个分配堆内存空间，并且用它的指针可以把一系列的空间串联起来，就像一条链子一样。这样一来，我们就能够利用指针对整个链表进行管理了。

## 9.6 链表的实现

上一节，我们介绍了链表的概念。在这一节，我们将介绍如何用程序来实现一个链表。在具体实现之前，我们先要明确一下我们有哪些任务：

- （1）能够创建一个具有若干个结点的链表。
- （2）能够访问到链表中的每一个结点，即输出每个结点的数据。**这种操作称为遍历。**
- （3）能够根据数据查找到结点所在的位置。
- （4）能够在链表的任意位置插入一个结点。
- （5）能够在链表的任意位置删除一个结点。
- （6）能够在程序结束前清除整个链表，释放内存空间。

### 链表的创建和遍历

我们知道链表也是动态分配的，虽然每次只分配一个结构变量（结点），但却少不了指向这个结构变量的指针。如果任何一个分配给我们的结构变量失去了指向它的指针，那么这个内存空间将无法释放，就造成了内存泄漏。由于指针还维系着各结点之间关系，指针的丢失造成了结点之间断开，整个链表就此被破坏。

所以，我们要保证每个结点都在我们的控制之内，即我们能够通过各种手段，利用指针访问到链表的任一个结点。这也是我们在所有对链表的操作过程中始终要注意的一点。

接下来，我们把链表的创建和遍历分析得更加具体化：

- （1）由于第一个结点也是动态分配的，因此一个链表始终要有一个指针指向它的表头，否则我们将无法找到这个链表。**我们把这个表头指针称为 head。**
- （2）在创建一个多结点的链表时，新的结点总是连接在原链表的尾部的，所以我们必须要有一个指针始终指向链表的尾结点，方便我们操作。**我们把这个表尾指针称为 pEnd。**
- （3）每个结点都是动态分配的，每分配好一个结点会返回一个指针。由于 head 和 pEnd 已经有了各自的岗位，我们还需要一个指针来接受刚分配好的新结点。**我们把这个创建结点的指针称为 pS。**

(4) 在遍历的过程中，需要有一个指针能够灵活动作，指向链表中的任何一个结点，以读取各结点的数据。我们把这个访问指针称为 **pRead**。

(5) 我们把创建链表和遍历各自写为一个函数，方便修改和维护。

做完了这些分析，我们可以开始着手写这个程序了：（程序 9.6.1）

```
#include "iostream.h"
struct node//定义结点结构类型
{
 char data;//用于存放字符数据
 node *next;//用于指向下一个结点（后继结点）
};
node * create();//创建链表的函数，返回表头
void showList(node *head);//遍历链表的函数，参数为表头
int main()
{
 node *head;
 head=create();//以 head 为表头创建一个链表
 showList(head);//遍历以 head 为表头的链表
 return 0;
}
node * create()
{
 node *head=NULL;//表头指针，一开始没有任何结点，所以为 NULL
 node *pEnd=head;//表为指针，一开始没有任何结点，所以指向表头
 node *pS;//创建新结点时使用的指针
 char temp;//用于存放从键盘输入的字符
 cout <<"Please input a string end with '#:'" <<endl;
 do//循环至少运行一次
 {
 cin >>temp;
 if (temp!='#')//如果输入的字符不是结尾符#，则建立新结点
 {
 pS=new node;//创建新结点
 pS->data=temp;//新结点的数据为 temp
 pS->next=NULL;//新结点将成为表尾，所以 next 为 NULL
 if (head==NULL)//如果链表还没有任何结点存在
 {
 head=pS;//则表头指针指向这个新结点
 }
 else//否则
 {
 pEnd->next=pS;//把这个新结点连接在表尾
 }
 pEnd=pS;//这个新结点成为了新的表尾
 }
 }
}
```

```

} while (temp!='#');//一旦输入了结尾符，则跳出循环
return head;//返回表头指针
}
void showList(node *head)
{
 node *pRead=head;//访问指针一开始指向表头
 cout <<"The data of the link list are:" <<endl;
 while (pRead!=NULL)//当访问指针存在时（即没有达到表尾之后）
 {
 cout <<pRead->data;//输出当前访问结点的数据
 pRead=pRead->next;//访问指针向后移动
 }
 cout <<endl;
}

```

运行结果：

Please input a string end with '#':

Tomato#

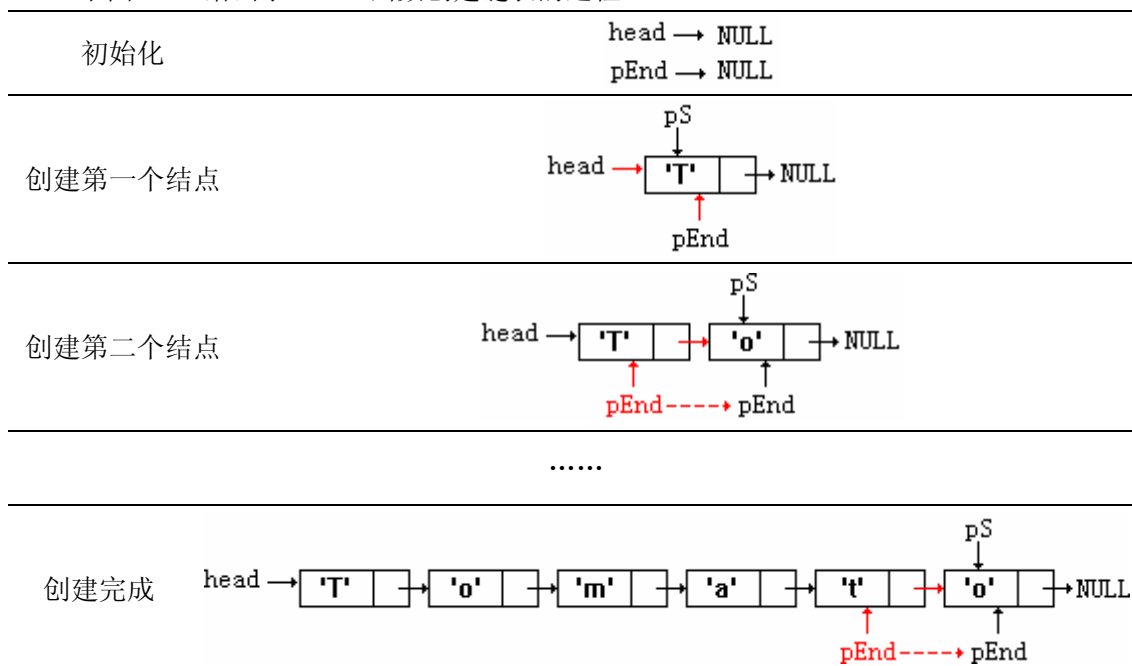
The data of the link list are:

Tomato

这个程序的功能是把输入的字符串保存到链表中，然后把它输出。从程序中我们可以看出，create 函数的主要工作有：

- ①做好表头表尾等指针的初始化。
- ②反复测试输入的数据是否有效，如果有效则新建结点，并做好该结点的赋值工作。将新建结点与原来的链表连接，如果原链表没有结点，则与表头连接。
- ③返回表头指针。

下图 9.6.1 给出了 create 函数创建链表的过程。



(图 9.6.1)

程序中 showList 函数的主要工作有：

- ①初始化访问指针。
- ②如果访问指针不为空，则输出当前结点的数据，否则函数结束。
- ③访问指针向后移动，并重复第二项工作。

注意，虽然上述程序可以运行，但是它没有将内存释放，严格意义上来说，它是一个不完整的程序。

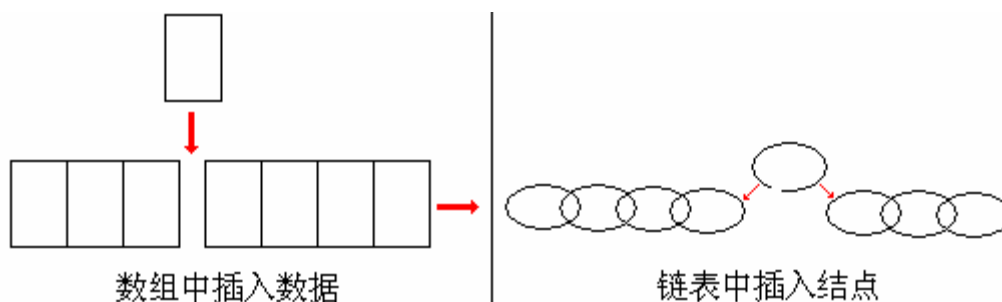
## 链表的查询

在对链表进行各种操作时，需要先对某一个结点进行查询定位。假设链表中没有数据相同的结点，我们可以编写这样一个函数，查找到链表中符合条件的结点：（程序 9.6.2）

```
node * search(node *head,char keyWord)//返回结点的指针
{
 node *pRead=head;
 while (pRead!=NULL)//采用与遍历类似的方法，当访问指针没有到达表尾之后
 {
 if (pRead->data==keyWord)//如果当前结点的数据和查找的数据相符
 {
 return pRead;//则返回当前结点的指针
 }
 pRead=pRead->next;//数据不匹配，pRead 指针向后移动，准备查找下一个结点
 }
 return NULL;//所有的结点都不匹配，返回 NULL
}
```

## 插入结点

数组在内存中是顺序存储的，要在数组中插入一个数据就变得颇为麻烦。这就像是在一排麻将中插入一个牌，必须把后面的牌全部依次顺移。然而，链表中各结点的关系是由指针决定的，所以在链表中插入结点要显得方便一些。这就像是把一条链子先一分为二，然后用一个环节再把它们连接起来。如下图 9.6.2 所示。

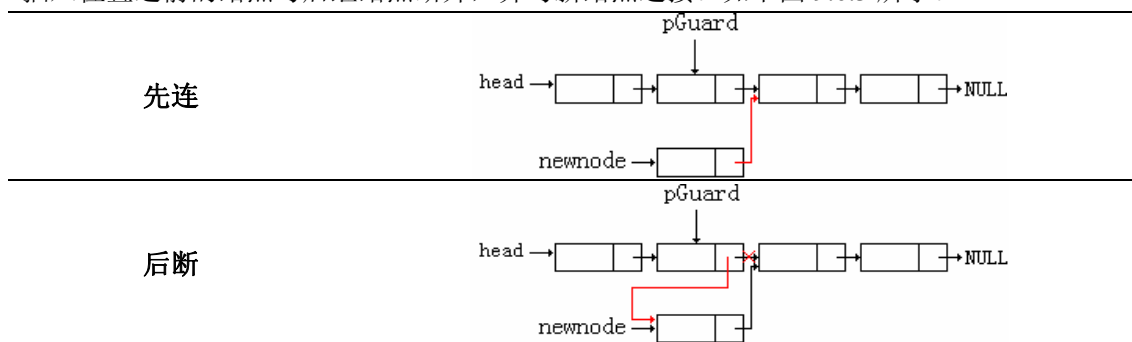


（图 9.6.2）

下面我们先对插入结点这个功能具体分析一下：

- （1）我们必须知道对哪个链表进行操作，所以表头指针 head 是必须知道的。
- （2）为了确定插入位置，插入位置前的结点指针 pGuard 是必须是知道的。
- （3）用一个 newnode 指针来接受新建的结点。
- （4）如果要插入的位置是表头，由于操作的是表头指针而不是一个结点，所以要特殊处理。
- （5）在插入结点的过程中，始终要保持所有的结点都在我们的控制范围内，保证链表的完

整性。为了达到这一点，我们采用先连后断的方式：先把新结点和它的后继结点连接，再把插入位置之前的结点与后继结点断开，并与新结点连接。如下图 9.6.3 所示。



(图 9.6.3)

做完了分析，我们可以开始编写插入函数了。为了简单起见，我们规定新结点插入位置为数据是关键字的结点之后，这样就可以使用刚才编写好的 `search` 函数了。如果该结点不存在，则插入在表头。则插入函数如下：（程序 9.6.3）

```
void insert(node * &head,char keyWord,char newdata)//keyWord 是查找关键字符
{
 node *newnode=new node;//新建结点
 newnode->data=newdata;//newdata 是新结点的数据
 node *pGuard=search(head,keyWord);//pGuard 是插入位置前的结点指针
 if (head==NULL || pGuard==NULL)//如果链表没有结点或找不到关键字结点
 { //则插入表头位置
 newnode->next=head;//先连
 head=newnode;//后断
 }
 else//否则
 { //插入在 pGuard 之后
 newnode->next=pGuard->next;//先连
 pGuard->next=newnode;//后断
 }
}
```

试试看：

如果把 `insert` 函数的第一个参数 `node * &head` 改为 `node *head` 是否可行？为什么？如果改成那样，会发生什么问题？

## 删除结点

与插入数据类似，数组为了保持其顺序存储的特性，在删除某个数据时，其后的数据都要依次前移。而链表中结点的删除仍然只要对结点周围小范围的操作就可以了，不必去修改其他的结点。

仍然我们先要来具体分析删除结点这个功能：

- (1) 我们必须知道对哪个链表进行操作，所以表头指针 `head` 是必须知道的。
- (2) 一般来说，待删除的结点是由结点的数据确定的。然而我们还要操作待删除结点之前的结点（或指针），以连接前后两段链表。之前所写的 `search` 函数只能找到待删除的结点，



却无法找到这个结点的前趋结点。所以，我们只好放弃 search 函数，另起炉灶。

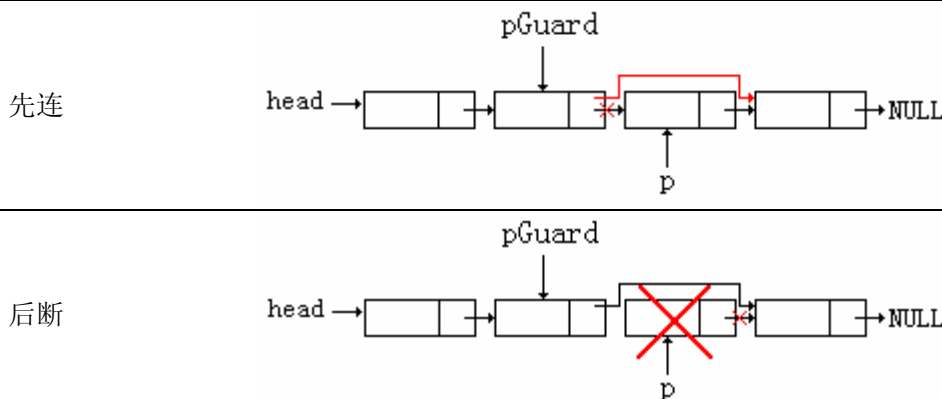
(3) 令 pGuard 指针为待删除结点的前趋结点指针。

(4) 由于要对待删除结点作内存释放，需要有一个指针 p 指向待删除结点。

(5) 如果待删除结点为头结点，则我们要操作表头 head，作为特殊情况处理。

(6) 在删除结点的过程中，仍然要始终保持所有的结点都在我们的控制范围内，保证链表的完整性。为了达到这一点，我们还是采用先连后断的方式：先把待删除结点的前趋结点和它的后继结点连接，再把待删除结点与它的后继结点断开，并释放其空间。如下图 9.6.4 所示。

(7) 如果链表没有结点或找不到待删除结点，则给出提示信息。



(图 9.6.4)

由于 delete 是 C++ 中的保留字，我们无法用它作为函数名，所以只好用 Delete 代替（C++ 是大小写敏感的，Delete 和 delete 是不同的）。都准备好了，我们就可以开始写函数了：（程序 9.6.4）

```
void Delete(node * &head, char keyWord) //可能要操作表头指针，所以 head 是引用
{
 if (head != NULL) //如果链表没有结点，就直接输出提示
 {
 node *p;
 node *pGuard = head; //初始化 pGuard 指针
 if (head->data == keyWord) //如果头结点数据符合关键字
 {
 p = head; //头结点是待删除结点
 head = head->next; //先连
 delete p; //后断
 cout << "The deleted node is " << keyWord << endl;
 return; //结束函数运行
 }
 else //否则
 {
 while (pGuard->next != NULL) //当 pGuard 没有达到表尾
 {
 if (pGuard->next->data == keyWord) //如果 pGuard 后继结点数据符合关键字
 {
 p = pGuard->next; //pGuard 后继结点是待删除结点
```

```

 pGuard->next=p->next;//先连
 delete p;//后断
 cout <<"The deleted node is " <<keyWord <<endl;
 return;//结束函数运行
 }
 pGuard=pGuard->next;//pGuard 指针向后移动
}
}
}
}
cout <<"The keyword node is not found or the link list is empty!" <<endl;//输出提示信息
}

```

## 清除链表

链表的结点也是动态分配的，如果在程序结束之前不释放内存，就会造成内存泄漏。因此，编写一个清除链表的函数就显得相当有必要。我们先来分析一下清除这个功能：

- (1) 我们必须知道对哪个链表进行操作，所以表头指针 head 是必须知道的，并且清除整个链表后要将其改为 NULL。
- (2) 类似于删除结点，我们还需要一个指针 p 来指向待删除结点。
- (3) 类似于删除表头结点的操作，我们仍然要先连后断：先把表头指向头结点的后继，再删除头结点。

下面我们来写一下这个函数：（程序 9.6.5）

```

void destroy(node * &head)
{
 node *p;
 while (head!=NULL)//当还有头结点存在时
 {
 p=head;//头结点是待删除结点
 head=head->next;//先连
 delete p;//后断
 }
 cout <<"The link list has been deleted!" <<endl;
}

```

试试看：

把上面的这些程序段都拼接起来，实践一下链表的各种操作。你也可以按照自己的想法改写这些函数，看看运行出来的结果是否正确。

至此，我们已经学习了链表的所有基本操作。下面来介绍一下数组存储和链表存储各自的优缺点。

| 存储方式 | 优点                                                                                               | 缺点                                                                                           |
|------|--------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| 数组存储 | <ul style="list-style-type: none"> <li>• 不需要指针，存储空间相对较小</li> <li>• 使用数组下标很快能找到第 n 个元素</li> </ul> | <ul style="list-style-type: none"> <li>• 存储空间的灵活性小，会造成浪费</li> <li>• 插入删除元素显得比较麻烦</li> </ul>  |
| 链表存储 | <ul style="list-style-type: none"> <li>• 存储空间可以灵活安排，不会浪费</li> <li>• 插入删除结点不影响前后结点</li> </ul>     | <ul style="list-style-type: none"> <li>• 各结点需要指针来连接前后结点</li> <li>• 无法很快地找到第 n 个结点</li> </ul> |

虽然很多初学者都认为链表非常难以理解,但是只要掌握了插入删除结点时“先连后断”的原则和如何遍历整个链表,所有的问题就迎刃而解了。

## 习题

1、指出下列各枚举类型的变量取值范围。

①enum integer{1,2,3,4,5};

②enum color{red,yellow,blue};

③enum grade{good,pass,failed}

2、指出下列各结构类型的成员数据。

①struct a1

```
{
 int b1;
 char c1;
};
```

②struct a2

```
{
 float b2;
 a2 *c2;
}
```

③struct a3

```
{
 int b3;
 bool c3;
};
```

struct d3

```
{
 a3 e3;
 d3 *f3;
};
```

3、与第二题的各小题对应声明了一些结构变量,请写出这些结构变量各自的成员数据。

①a1 i;

②a2 \*j=new a2;

③d3 k;

4、指出下列程序段的错误之处。

①

```
#include "iostream.h"
```

```
void display(color a);
```

```
int main()
```

```
{
```

```
 enum color{red,yellow,blue};
```

```
 color a=1;
```

```
 display(a);
```

```
 return 0;
```

```
}
```

```
void display(color a)//希望能够输出枚举变量的符号
```

```

{
 cout <<a <<endl;
}
②
#include "iostream.h"
struct employee
{
 int id;
 float salary;
}
void display(employee *p);
int main()
{
 int num;
 employee *emptr;
 cin >>num;
 emptr=new employee[num];
 cout <<"id:";
 cin >>emptr[3]->id;
 cout <<"salary:";
 cin >>emptr[3]->salary;
 display(emptr[3]);
 return 0;
}
void display(employee *p)
{
 cout <<"id:" <<p.id <<endl;
 cout <<"salary:" <<p.salary <<endl;
}

```

5、根据运行结果完成代码。

假设 9.6 节的函数都已经写好，现在要求编写一个函数 reverse，把一个已经存在的链表倒置。

```

#include "iostream.h"
struct node
{
 char data;
 node *next;
};
node * create();//创建链表
void showList(node *head);//遍历链表
node * search(node *head,char keyWord);//查找结点
void insert(node * &head,char keyWord,char newdata);//插入结点
void Delete(node * &head,char keyWord);//删除结点
void destroy(node * &head);//清除链表
void reverse(_____);//实现链表倒置

```

```

int main()
{
 node *head=NULL;
 head=create();
 showList(head);
 cout <<"After reversed..." <<endl;
 reverse(head);
 showList(head);

 return 0;
}
void reverse(_____)
{
 node *newhead_____;
 node *p;
 while (_____)
 {
 p=head;
 head=head->next;

 newhead=p;
 }

}

```

运行结果:

Please input a string end with '#':

Tomato#

The data of the link list are:

Tomato

After reversed...

The data of the link list are:

otamoT

6、根据要求编写程序。

①某公司有一个大型员工数据库，里面存储着很多员工的信息。现要求我们需要编写一个函数，根据员工们的工资高低作降序排列，并且输出这些数据。然而，每个员工的信息实在太多，如果经常改变它们的存储位置代价会很大。我们以结构数组作为这个数据库的模型，要求在不改变各数组元素存储位置的情况下（即不能作移动，交换），对他们的工资作降序排列并输出。数据库中的部分数据如下：

| name  | salary    |
|-------|-----------|
| Jimmy | \$1250.04 |
| Tim   | \$1280.07 |
| Rose  | \$1390.08 |
| Tommy | \$1490.35 |
| Jon   | \$1320.54 |

运行结果示例:

name Tommy salary \$1490.35

name Rose salary \$1390.08

name Jon salary \$1320.54

name Tim salary \$1280.07

name Jimmy salary \$1250.04

②很多软件都具有撤销的功能。实际上,软件是把用户的每一步操作压栈,每撤销一次就退一次栈。那么什么是压栈什么是退栈呢?我们可以认为,压栈就是在链表的表尾插入一个新结点,退栈就是删除一个尾结点。我们用英文字符表示各种操作,用\$表示一次撤销,用#表示操作结束。请根据这串指令,输出用户最终实际做了哪些操作。为简单起见,撤销的次数始终少于有效的操作次数。

运行结果示例:

请输入指令:

ABC\$DEFG\$\$\$HIJ\$KLM\$#

用户的实际操作是:

ABDHIKL

# 中篇 实战程序设计

## 第十章 如何阅读程序代码

阅读代码是程序员必须掌握的技能之一，也是每次考试都会出现的题目。然而，对于没有程序设计基础的读者来说，看一堆代码就犹如在看天书，不知从何处下手。本章主要向初学者介绍一些阅读代码的常用方法，以帮助大家克服对代码的恐惧。

### 10.1 整体把握法

很多初学者问，代码应该怎样读？以怎么样的顺序读？

其实阅读代码和我们读一篇文章是有着相通之处的。我们一篇文章，要看懂它的大意，对每一段快速地扫视。如果出现了难以理解的地方，再根据上下文仔细琢磨它的意思。我们阅读文章，并不是在阅读它的文字，而是在理解它文字中所表达的含义，即语意。

**类似地，我们在读一段代码的时候，要尝试看懂它的大意。**如果出现自己不熟悉的语句，就应该先去查一些相关的工具书，了解语句的意思。这就如同我们读文章遇到了看不懂的字词，需要去借助词典一样。如果出现了难以理解的地方，暂时先放一放（尽管可能看不懂的地方有很多），坚持把整个代码读完，然后再来各个击破。

**要注意，我们在阅读代码的时候也不是在阅读它的语句，而是在理解代码的语意。**就好像把交换操作的三个赋值语句拆开，就没有任何含义了，只有把它们三句看成一个整体，我们才能明白那是交换。

### 阅读代码的顺序

我们现在对一个程序的结构很了解了，即：

```
预处理头文件
各函数声明
主函数
{
 主函数体
}
各函数定义
```

我们阅读代码的时候并不能直接从上到下依次阅读。刚才已经解释了，我们是要理解代码的语意，而不是阅读语句本身。然而从上到下的依次阅读可能会破坏了这种语义，或者让你觉得更加头昏脑胀。**这是因为函数的语意在于调用函数之处的前后，而不是完全在于函数原型或者函数定义中。**特别是一些比较长的代码，即使你先看了函数原型，到了调用的地方，早就忘了这个函数是什么了。

所以，比较正确的读法是从主函数开始，遇到调用函数，则到前面查阅该函数原型，了解返回值类型和参数的含义，如果有必要，再去查看函数定义，了解这个函数是如何运作的。

## 整体把握语意

所谓整体把握，就是不要太在意细节的部分，只要能够做到了解语意就够了。比如一个求正弦值的 `sin` 函数，我们只需要知道它的功能是求正弦值就可以了，对于它是如何求出一个正弦值的却不感兴趣。所以，如果我们已经知道一个函数或语句块是派什么用处的，在不影响语意理解的情况下，就没必要对它的实现过程去深究。

那么，我们是如何去了解一个函数或语句块的作用呢？主要有两个方法。

其一，是猜测。这里的猜测不是漫无目的的乱猜，而是要有根据的。我们知道，一个优秀的程序员在给变量、函数以及参数起名字时，会考虑到它们的实际含义。一般情况下不会出现诸如 `a1`、`a2` 之类不知所云的名称。所以，我们只需要根据函数原型中的函数名以及函数参数名，就能对这个函数的作用略知一二了。如果有必要，则可以到函数的定义中，找到某些具有特征的操作（比如对数组的操作，一般会用 `size` 参数控制循环次数），以证实自己的猜测。

其二，是看注释。一个优秀的程序员会有做注释的好习惯。所以，在一些难以看懂的代码旁边，一般都会有一些注释，以方便阅读者理解。我们在阅读代码的时候，就要充分利用好这些注释，这样对我们理解语意有了方向性的指导。

下面我们用整体把握法来阅读一段代码：（省略函数定义，以下代码无法通过编译）

```
#include "iostream.h"
int size(int array[]);
void insert(int array[],int position,int size,int data);//position 为下标，该函数在 position 位置插入 data，其他数据向后顺移
int del(int array[],int position);//删除 position 位置的数据，其他数据向前顺移，返回被删除的元素
int find(int data);//返回数据为 data 的下标
void display();//显示所有元素
int main()
{
 int array[]={1,3,2,4,5,6,8};
 cout <<"The size of Array is " <<size(array) <<endl;
 insert(array,0,size(array),10);
 insert(array,3,size(array),7);
 cout <<"The number deleted is" <<del(array,find(3)) <<endl;
 cout <<"The size of Array is " <<size(array) <<endl;
 display();
 return 0;
}
```

分析：

首先创建一个数组 `array`，大小为 7，故在屏幕上显示的第一句话应该是“`The size of Array is 7`”。至于这个 `size` 函数是如何求得 `array` 的大小的，我们不必关心。然后调用了 `insert` 函数，我们查阅函数原型得到一系列信息：其中参数 `array` 是操作的数组，`position` 是插入值所在的下标，`size` 是数组大小，`data` 是插入的值。所以，第一次调用 `insert` 函数后，数组内的元素为 10，1，3，2，4，5，6，8，第二次调用 `insert` 函数后，数组内的元素为 10，1，3，7，2，4，5，6，8。接着调用 `del` 函数和 `find` 函数，根据注释我们知道这个调用的意思是删除元素“3”，所以屏幕上显示“`The number deleted is 3`”，这时数组的大小变为 8，所以再输



出 “The size of Array is 8”，最后将这些元素一一输出，即：10, 1, 7, 2, 4, 5, 6, 8。

## 10.2 经验法

在上一节，我们已经学会通过函数名和注释来了解代码的信息。然而，当我们面对一堆语句块的时候，我们应该怎么办？这个时候，就要靠我们的经验了。

**这里的经验是指读者所了解的各种算法和语句的关键点。**对于前者，我们在书的每章节中都有穿插讲述，如果大家对于那些简单的算法都烂熟于胸的话，那么对付那些公式化的语句块（比如交换），就很轻松了。而语句的关键点是本节要着重讲述的。

在代码中，最容易让人思路混乱的，就数分支结构和循环结构了。如果再来一些嵌套，则更是让人觉得没有信心了。所以，对于语句块，我们最先要攻克的，就是分支结构和循环结构了。

在攻克它们之前，我们要做好充分的准备工作，那就是理清语句之间的层次关系，即在第四章中所介绍的“门当户对”。我们根据语句的缩进和大括号来判断语句间的层次，这里不作赘述。接下来，我们就要寻找分支结构和循环结构的关键点了，那就是条件。之所以说它关键是因为**程序到底运行到哪个分支，循环究竟执行几次，终止以后变量的状态，都是由条件来决定的。**

下面我们就来使用经验法来阅读一段代码：

```
#include "iostream.h"
int cmp(char str1[],char str2[],int length);
void result(int num);
int main()
{
 char str1[]{"Hello World!"};
 char str2[]{"Hello World!"};
 char str3[]{"Hello world!"};
 result(cmp(str1,str2,sizeof(str1)));
 result(cmp(str2,str3,sizeof(str2)));
 return 0;
}
int cmp(char str1[],char str2[],int length)
{
 int i;
 for (i=0;i<length && str1[i]==str2[i];i++);
 if (i==length)
 return -1;
 else
 return i+1;
}
void result(int num)
{
 if (num==-1)
 cout <<"They are the same string." <<endl;
 else
```

```

 cout <<"The " <<num <<"th(st/rd) character is different." <<endl;
}

```

分析:

在这段代码中,我们撇开对函数名的猜想。创建了三个字符数组以后,主函数中就只有调用函数了。我们先来看 `cmp` 函数:里面只有一个循环,循环继续的条件是 `i<length` 并且 `str1[i]==str2[i]`,用自然语言描述就是“没有到达字符串末尾并且两个字符串的字符一一相同”。所以,当循环中止的时候,有两种可能:一是到达字符串末尾,二是有两个字符不相同。如果到达了字符串末尾,那么 `i` 的值应该是多少呢?很容易发现, `i` 的值应该是 `length`。

(因为当 `i=length-1` 的时候,还要执行循环,即 `i++`)而如果是两个字符不同时, `i` 的值应该是不同的字符所在的下标。所以,这个函数的功能是,当两个字符串相同时,返回-1,当两个字符串不同时,返回不同的字符所在的位置(位置和下表相差 1,所以要 `i+1`)。接着来看 `result` 函数,它主要就是判断由 `cmp` 函数得出的结果,并在屏幕上显示,即如果 `cmp` 函数返回-1,就要在屏幕上显示两个字符串相同,否则显示第几个字符不相同。

运行结果:

They are the same string.

The 7th(st/rd) character is different.

## 10.3 模拟法

虽然我们说阅读代码应该去理解语意,然而对初学者来说,做到这一点并不容易。可能这个单词不认识,那个算法不熟悉,难道就不能去阅读了么?这个时候,我们就要使用模拟法了。

**所谓模拟法,是指抛开语意的影响,原原本本地按照语句要求模拟电脑的各种操作。**不管是输入、输出还是变量的改变,都要把它想象出来。只要我们对语句的了解是正确的,并且在模拟过程中是仔细的,那么最后模拟出的运行结果应该和电脑上的运行结果是一样的。

不过,一般而言,使用模拟法是需要工具的,就是纸和笔。俗话说“好记性不如烂笔头”,人脑毕竟不是内存,让我们记上十几个变量的变化情况就够呛了,更别说数组了。所以我们要用纸笔将程序每一步的执行情况和变量的变化情况一一记录下来,那么就大大降低了出错的概率了。

下面我们用模拟法来阅读一段代码:

```

#include "iostream.h"
int fun(int a,int b);
int main()
{
 cout <<fun(4,6) <<endl;
 cout <<fun(5,6) <<endl;
 cout <<fun(6,9) <<endl;
 return 0;
}
int fun(int a,int b)
{
 int c;
 while (b!=0)

```

```

 {
 c=a%b;
 a=b;
 b=c;
 }
 return a;
}

```

分析:

主函数里面只有调用三次函数。但是这个函数写得实在不怎么样,不管是函数名还是参数名,都看不出这是在做什么。所以我们只好模拟运行一下。

我们以 fun(4,6)为例稍作讲述。首先调用函数以后, a=4、b=6、c 的值不确定。这时候 b 不等于 0, 执行循环体, c=a%b=4%6=4, a=b=6, b=c=4。这时 b 还不等于 0, 继续执行循环体, c=a%b=2, a=b=4, b=c=2。然后 b 不等于 0, 执行循环体, c=a%b=0, a=b=2, b=c=0, 这时不满足循环条件, 跳出循环, 返回 a=2。下表为三次调用函数过程中, 各参数值的变化情况。

| fun(4,6) |   |   |
|----------|---|---|
| c        | A | b |
| ?        | 4 | 6 |
| 4        | 6 | 4 |
| 2        | 4 | 2 |
| 0        | 2 | 0 |

| fun(5,6) |   |   |
|----------|---|---|
| c        | a | b |
| ?        | 5 | 6 |
| 5        | 6 | 5 |
| 1        | 5 | 1 |
| 0        | 1 | 0 |

| fun(6,9) |   |   |
|----------|---|---|
| c        | a | b |
| ?        | 6 | 9 |
| 6        | 9 | 6 |
| 3        | 6 | 3 |
| 0        | 3 | 0 |

运行结果:

```

2
1
3

```

我们看到, 模拟的结果和程序在电脑上的运行结果一样。多次模拟的结果, 我们也可以揣测出这个函数的作用是求两个整数的最大公约数。

至此, 我们已经学完了阅读代码的三种主要方法。大家只要能够将三者融合运用, 再加上一点征服代码的信心和耐心, 那么即使更长一点的代码, 也并不是可怕的。

## 习题

阅读下列程序, 分析运行过程, 并写出运行结果。

①

```

#include "iostream.h"
int sta(char c);
int main()
{
 int number=0,letter=0,other=0;
 char str[]={ 'a','e','4','%','8','!', 'F','1','!','/' };
 for (int i=0;i<sizeof(str);i++)
 {
 cout <<str[i];
 }
}

```

```

 switch(sta(str[i]))
 {
 case 0:
 number++;
 break;
 case 1:
 letter++;
 break;
 case 2:
 other++;
 break;
 }
 }
 cout <<endl <<number <<endl;
 cout <<letter <<endl;
 cout <<other <<endl;
 return 0;
}
int sta(char c)
{
 if (c>=48 && c<=57) return 0;
 if ((c>=65 && c<=90) || (c>=97 && c<=122)) return 1;
 return 2;
}

```

②

```

#include "iostream.h"
int fun(char str1[],char str2[]);
int main()
{
 char str1[]{"Hello World!"};
 char str2[]{"hello world!"};
 char str3[]{"hello world"};
 char str4[]{"hello"};
 cout <<fun(str1,str2) <<endl;
 cout <<fun(str2,str3) <<endl;
 cout <<fun(str4,str3) <<endl;
}
int fun(char str1[],char str2[])
{
 int i;
 for (i=0;(str1[i]!='\0' && str2[i]!='\0');i++);
 if (str1[i]=='\0')
 {
 if (str2[i]!='\0')

```

```

 return -1;
 else
 return 0;
 }
 return 1;
}

```

③

```

#include "iostream.h"
void calculate(double a1,double b1,double c1,double a2,double b2,double c2);
int main()
{
 calculate(1,2,3,1,1,2);
 calculate(2,4,8,1,3,5);
 calculate(1,1,2,3,3,6);
 return 0;
}
void calculate(double a1,double b1,double c1,double a2,double b2,double c2)
{
 double d=a1*b2-a2*b1;
 double e=a1*c2-a2*c1;
 double f=c1*b2-c2*b1;
 if (d==0)
 {
 cout <<"ERROR!" <<endl;
 return;
 }
 cout <<e/d <<endl;
 cout <<f/d <<endl;
}

```

④

```

#include "iostream.h"
#include "iomanip.h"
int main()
{
 for (int i=0;i<8;i++)
 {
 for (int j=0;j<i+1;j++)
 {
 if (j==0) cout <<setw(8-i) <<i;
 else cout <<setw(2) <<i;
 }
 cout <<endl;
 }
 return 0;
}

```

```
}
⑤
#include "iostream.h"
void sort(int array[],int size);
int main()
{
 int a[]={1,4,5,2,3};
 sort(a,sizeof(a)/sizeof(int));
 for (int i=0;i<sizeof(a)/sizeof(int);i++)
 {
 cout <<' ' <<a[i];
 }
 cout <<endl;
}
void sort(int array[],int size)
{
 int insert,index;
 for (int i=1;i<size;i++)
 {
 insert=array[i];
 index=i-1;
 while (index>=0 && insert<array[index])
 {
 array[index+1]=array[index];
 index--;
 }
 array[index+1]=insert;
 }
}
```

# 第十一章 如何调试程序代码

上一章我们介绍了如何阅读别人的代码。在这一章，我们将深入介绍变量、头文件和一些调试程序的技巧。学好了这一章，我们的实际能力将大大提高。

## 11.1 再谈变量

我们已经在前篇中学习了变量，并且能够熟练地使用它。可是，仅仅靠这些知识，有些问题仍然无法得到解决。

### 标志符

首先要来介绍一下什么是标志符。在程序设计的过程中，经常要给变量、函数甚至是一些数据类型起名字（还包括以后的类名，对象名等）。我们把这些用户根据一些规定，自己定义的各种名字统称为标志符（Identifier）。显然，标志符不允许和任何保留字相同。

### 全局变量和局部变量

在函数这一章节中，我们说过函数体内声明的变量仅在该函数体内有效，别的函数是无法使用的。并且在函数运行结束后，这些变量也将消失了。我们把这些在函数体内声明的变量称为局部变量（Local Variable）。

然而，可能会遇到这样的问题：我们想要创建一个变量作为数据缓冲区（Buffer），分别供数据生成、数据处理和数据输出三个函数使用，三个函数都要能够读取或修改这个变量的值。显然通过传递参数或返回值来解决这个问题是非常麻烦的。

那么，我们能否建立一个变量能够让这三个函数共同使用呢？在 C++ 中，我们可以在函数体外声明一个变量，它称为全局变量（global variable）。所谓全局，是指对于所有函数都能够使用。当然，在该变量声明之前出现的函数是不知道该变量的存在的，于是也就无法使用它了。另外，如果我们声明了一个全局变量之后没有对它进行初始化操作，则编译器会自动将它的值初始化为 0。

下面，我们就用全局变量来实现刚才提出的那个问题：（程序 11.1.1）

```
#include "iostream.h"
#include "stdlib.h"//用于产生随机数，不必理会
#include "time.h"//用于产生随机数，不必理会
#include "iomanip.h"//用于设置域宽
void makenum();
void output();
void cal();
int main()
{
 srand(time(NULL));//用于产生随机数，不必理会
 for (int i=0;i<4;i++)
 {
```

```

 makenum();//产生随机数放入缓冲区
 cal();//对缓冲区的数进行处理
 output();//输出缓冲区的数值
 }
 return 0;
}
int buffer;//定义全局变量，以下函数都能使用它
void makenum()
{
 cout <<"Running make number..." <<endl;
 buffer=rand();//把产生的随机数放入缓冲区
}
void cal()
{
 cout <<"Running calculate..." <<endl;
 buffer=buffer%100;
}
void output()
{
 cout <<"Running output..." <<endl;
 cout <<setw(2) <<buffer <<endl;
}

```

运行结果:

```

Running make number...
Running calculate...
Running output...
48
Running make number...
Running calculate...
Running output...
47
Running make number...
Running calculate...
Running output...
24
Running make number...
Running calculate...
Running output...
90

```

以上为某次运行得到的随机结果。可见，使用全局变量使得多个函数之间可以共享一个数据，同时从理论上实现了函数之间的通讯。



试试看：

- 1、程序 11.1.1 中能否在主函数里使用变量 `buffer` 呢？为什么？
- 2、字符型、浮点型、双精度型和布尔型的全局变量如果没有被我们初始化，它们的值分别都是多少？

## 静态局部变量

全局变量实现了函数之间共享数据，也使得变量不再会因为某个函数的结束而消亡。但是，新问题又出现了：一个密码检测函数根据调用（用户输错密码）的次数来限制他进入系统。如果把调用次数存放在一个局部变量里，显然是不可行的。虽然全局变量可以记录一个函数的运行次数，但是这个变量是被所有函数共享的，每个函数都能修改它，实在很危险。我们现在需要的是一个函数运行结束后不会消失的，并且其他函数无法访问的变量。

C++中，我们可以在函数体内声明一个**静态局部变量 (Static Local Variable)**。它在函数运行结束后不会消失，并且只有声明它的函数中能够使用它。声明一个静态局部变量的方法是在声明局部变量前加上 `static`，例如：

```
static int a;
```

和全局变量类似，如果我们没有对一个静态局部变量做初始化，则编译器会自动将它初始化为 0。

下面，我们就用静态局部变量来模拟一下这个密码检测函数的功能：（程序 11.1.2）

```
#include "iostream.h"
#include "stdlib.h"
bool password();//密码检测函数
int main()
{
 do
 {
 }
 while (password() != true);//反复检测密码直到密码正确
 cout <<"欢迎您进入系统！" <<endl;
 return 0;
}
bool password()
{
 static numOfRun=0;//声明静态局部变量存放函数调用次数
 if (numOfRun<3)
 {
 int psw;
 cout <<"第" <<++numOfRun <<"次输入密码" <<endl;
 cin >>psw;
 if (psw==123456)
 {
 return true;
 }
 }
}
```

```

 else
 {
 cout <<"密码错误！ " <<endl;
 return false;
 }
 }
 else
 {
 cout <<"您已经输错密码三次！ 异常退出！ " <<endl;
 exit(0);//退出程序运行
 }
}

```

第一次运行结果：

第 1 次输入密码

111111

密码错误！

第 2 次输入密码

222222

密码错误！

第 3 次输入密码

0

密码错误！

您已经输错密码三次！ 异常退出！

第二次运行结果：

第 1 次输入密码

000000

密码错误！

第 2 次输入密码

123456

欢迎您进入系统！

使用静态局部变量可以让函数产生的数据更长期更安全地存储。如果一个函数运行和它以前的运行结果有关，那么一般我们就会使用静态局部变量。

## 变量的作用域

在程序的不同位置，可能会声明各种不同类型（这里指静态或非静态）的变量。然而，声明的位置不同、类型不同导致每个变量在程序中可以使用的范围不同。**我们把变量在程序中可以使用的有效范围称为变量的作用域。**

任何变量都必须在声明之后才能被使用，所以**一切变量的作用域都始于变量的声明之处**。那么，它到什么地方终止呢？我们知道 C++ 的程序是一个嵌套的层次结构，即语句块里面还能有语句块。最终语句块由各条语句组成，而语句就是程序中的最内层，是组成程序的一个最小语法单位。**在某一层次声明的变量的作用域就终止于该变量所在层次的末尾。**

举个例子来说明：

```
#include "iostream.h"
```

```

int main()
{
 int a=3,b=4;//变量 a 和 b 的作用域开始
 for (int i=0;i<5;i++)//在 for 语句内声明的变量 i 的作用域开始
 {
 int result=i;//变量 result 的作用域开始
 if (int j=3)//在 if 语句内声明的变量 j 的作用域开始
 {
 int temp=8;//变量 temp 的作用域开始
 result=temp+(a++)-(b--);
 }//变量 temp 的作用域结束
 else
 result=2;//if……else 语句结束，变量 j 的作用域结束
 cout <<result <<endl;
 }//for 语句结束，变量 i 和 result 的作用域结束
 return 0;
} //变量 a 和 b 的作用域结束

```

根据上面这段程序，我们发现每当一个语句块或语句结束，那么在该语句块或语句层次内声明变量的作用域也就结束了。所以，下面的这段程序就存在错误：

```

#include "iostream.h"
int main()
{
 int a=3,b=4;
 for (int i=0;i<5;i++)
 {
 int result=i;
 if (int j=3)
 {
 int temp=8;
 result=temp+(a++)-(b--);
 }
 else
 result=2;
 cout <<j <<result <<endl;//j 的作用域结束，变量未定义
 }
 cout <<result <<endl; //result 的作用域结束，变量未定义
 cout <<i <<endl;//这里居然是正确的，为什么呢？
 return 0;
}

```

变量 j 和 result 无法输出是在意料之中的，但是为什么明明变量 i 的作用域已经结束了，却还是能够正常输出呢？这是微软给我们开的一个玩笑。根据 ANSI C++ 的标准，在 for 语句头中声明的变量的作用域的确应该在 for 语句的末尾结束。然而 VC++ 却没有完全符合这个标准，它认为 for 语句头中声明的变量作用域到包含该 for 语句的最小语句块结束。尽管如此，我们还是应该按照 ANSI C++ 标准来认知变量的作用域。

试试看：

- 1、阅读程序 11.1.1，思考全局变量 `buffer` 的作用域在哪里？
- 2、阅读程序 11.1.2，思考静态局部变量 `numOfRun` 的作用域在哪里？

## 变量的可见性

我们之前介绍过，在某一个函数中，不应该有两个名字相同的变量。可是，我们拿下面这段程序代码（程序 11.1.3）去测试一下，发现居然在同一个函数中可以有二个名字相同的变量。这又是怎么回事呢？编译器又是如何辨别这两个名字相同的变量的呢？

```
#include "iostream.h"
int main()
{
 int a=3,b=4;
 {
 int a=5,b=6;
 {
 char a='a',b='b';
 cout <<a <<b <<endl;
 }
 cout <<a <<b <<endl;
 }
 cout <<a <<b <<endl;
 return 0;
}
```

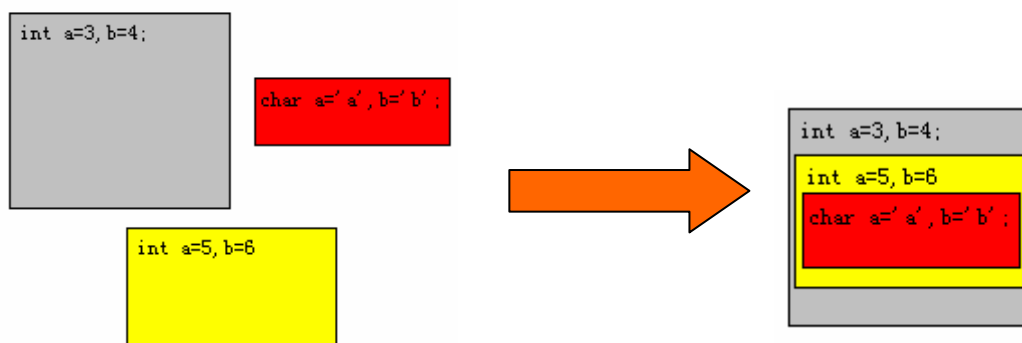
运行结果：

ab

56

34

我们已经说明，变量可以使用的范围是由变量的作用域决定。不同层次的变量的作用域，就好像大小不一的纸片。把它们堆叠起来，就会发生部分纸片被遮盖的情况。我们把这种变量作用域的遮盖情况称为变量的可见性（Visibility）。如下面的图 11.1 所示：



(图 11.1)

编译器正是根据变量的可见性，来判断我们到底引用哪个变量的。具体在程序中就是：

```

#include "iostream.h"
int main()
{
 int a=3,b=4;//整型变量 a=3、b=4 的作用域开始
 {
 int a=5,b=6;//整型变量 a=5、b=6 的作用域开始，整型变量 a=3、b=4 不可见
 {
 char a='a',b='b';//字符型变量 a='a'、b='b'作用域开始，整型变量 a、b 不可见
 cout <<a <<b <<endl;//输出字符型变量，整型变量 a、b 不可见
 }//字符型变量 a='a'、b='b'作用域结束
 cout <<a <<b <<endl;//输出整型变量 a=5、b=6，整型变量 a=3、b=4 不可见
 }//整型变量 a=5、b=6 的作用域结束
 cout <<a <<b <<endl; //输出整型变量 a=3、b=4
 return 0;
} //整型变量 a=3、b=4 的作用域结束

```

然而，当两张纸处于同一个层次，显然两者就不可能发生遮盖了。所以，如果我们在同一个层次中声明两个名字相同的变量，那么他们的作用域就不是遮盖，而是冲突了。

因此，在某个函数的同一语法层次内不能声明多个名字相同的变量。

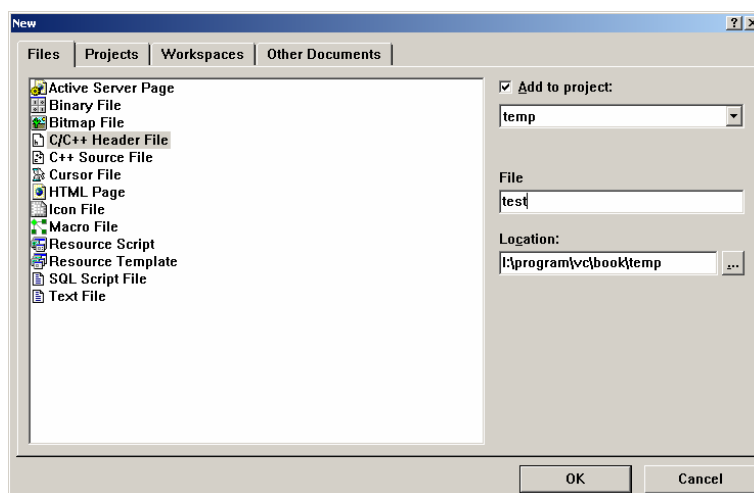
## 11.2 头文件的奥秘

我们在第六章介绍了一下通过使用某些头文件来调用标准库函数。在本节我们将更深入地介绍头文件的概念。

### 如何创建一个头文件

在第二章中，我们看到一个 C++ 的工程里面除了源文件还有头文件。根据以下步骤便能创建一个头文件：

首先要创建或打开一个工程，然后按 File 菜单中的 new。在出现的对话框左边选择 C/C++ Header File，在对话框右边的 File 一栏里填上头文件名，最后点击 OK。如下图 11.2 所示：



(图 11.2)

这时候点击左侧 Workspace 框内树状目录里新建的头文件，就能编辑这个文件了。不过

在我们进行编译的时候，记得要切换回对应的 `cpp` 文件，如果在编辑头文件时编译，就会提示无法编译。

## 头文件里有些什么？

头文件的使用主要体现在两个方面，一个是重（音 **chóng**）用（即多次使用），另一个是共用。

那些提供标准库函数的头文件就是为了重用。很多程序或工程可能会用到这些标准库函数，把它们写在头文件里面，每次使用的时候只需要包含已经完成的头文件就可以了。

头文件的共用主要体现在 C++ 的多文件结构中。由于目前的程序规模较小，尚不需要用到多文件结构，所以在此对头文件的共用不作展开。有兴趣的读者可以查阅相关书籍。

那么，如果我们要自己编写一个可以重用的头文件，里面应该写些什么呢？

类似于标准库函数，我们在头文件里面应该模块化地给出一些函数或功能。另外还应该包括独立实现这些函数或功能的常量、变量和类型的声明。

下面我们就来看一个头文件应用的实例：（程序 11.2）

```
//shape.h
#include "math.h"//在计算三角形面积时要用到正弦函数
const double pi=3.14159265358;//常量定义
struct circle//类型声明
{
 double r;
};
struct square
{
 double a;
};
struct rectangle
{
 double a,b;
};
struct triangle
{
 double a,b,c,alpha,beta,gamma;
};
double perimeter_of_circle(double r)//函数定义
{
 return 2*pi*r;
}
double area_of_circle(double r)
{
 return pi*r*r;
}
double perimeter_of_square(double a)
{
 return 4*a;
}
```

```
}
double area_of_square(double a)
{
 return a*a;
}
double perimeter_of_rectangle(double a,double b)
{
 return 2*(a+b);
}
double area_of_rectangle(double a,double b)
{
 return a*b;
}
double perimeter_of_triangle(double a,double b,double c)
{
 return a+b+c;
}
double area_of_triangle(double a,double b,double gamma)
{
 return sin(gamma/180*pi)*a*b/2;
}
//main.cpp
#include "iostream.h"
#include "shape.h"//包含我们编写好的 shape.h
int main()
{
 circle c={2};
 square s={1};
 rectangle r={2,3};
 triangle t={3,4,5,36.86989,53.13011,90};
 cout <<"Perimeter of circle " <<perimeter_of_circle(c.r) <<endl;
 cout <<"Area of square " <<area_of_square(s.a) <<endl;
 cout <<"Perimeter of rectangle " <<perimeter_of_rectangle(r.a,r.b) <<endl;
 cout <<"Area of triangle " <<area_of_triangle(t.b,t.c,t.alpha) <<endl;
 return 0;
}
```

运行结果：

Perimeter of circle 12.5664

Area of square 1

Perimeter of rectangle 10

Area of triangle 6

我们编写好了 shape.h 头文件，以后用到计算图形周长或面积的时候，就不需要重新编写函数了，只需要包含这个头文件就行了。

## 头文件和源文件

由于头文件是为了重用，所以在一个复杂的程序中，头文件可能会被间接地重复包含。如果头文件里面都是函数声明，那问题还不大。如果头文件里面有函数定义（如程序 11.2），那么就会出现函数被重复定义的错误，程序将无法运行。我们可以采用函数声明和定义分离的方式：把所有的声明都放在 `shape.h` 中，把所有的定义放在 `shape.cpp` 中。注意必须在 `shape.cpp` 中包含 `shape.h`，否则在编译连接时会发生错误。我们在使用时仍然包含 `shape.h`，但由于函数的定义并不在该头文件中，所以就不会被重复定义了。

## 细说#include

我们几乎每次编写程序的时候都要用到 `#include` 命令，那么这条命令到底是什么意思呢？

**#include 是一条编译预处理命令。**什么叫编译预处理命令呢？我们知道，程序中的每一句语句会在运行的时候能得到体现。比如变量或函数的声明会创建一个变量或者函数，输出语句会在屏幕上输出字符。然而编译预处理命令却不会在运行时体现出来，因为它是写给编译器的信息，而不是程序中需要执行的语句。编译预处理命令不仅仅只有 `#include` 一条，在 C++ 中，所有以 `#` 开头的命令都是编译预处理命令，比如 `#if`、`#else`、`#endif`、`#ifdef`、`#ifndef`、`#undef` 和 `#define` 等等。

当编译器遇到了 `#include` 命令后，就把该命令中的文件插入到当前的文件中。不难想象，程序 11.2 的 `main.cpp` 文件实质上包含了 `shape.h` 文件中的所有语句。所以它能够顺利调用 `shape.h` 文件中的各个函数。

试试看：

1、把程序 11.2 的 `main.cpp` 中 `#include "iostream.h"` 移动到 `shape.h` 中，是否会影响程序的运行？为什么？

2、如果有两个头文件 `a.h` 和 `b.h`，在 `a.h` 中有 `#include "b.h"`，在 `b.h` 中有 `#include "a.h"`，那么在编译包含它们的源文件时，会发生什么错误？

**结论：互相包含的两个头文件在编译的时候会导致错误甚至死机。**

## 尖括号和双引号的区别

如果你还看一些别的 C++ 教程，那么你可能很早就发现了，有些书上的 `#include` 命令写作 `#include <文件名>`，但有时候又会出现 `#include "文件名"`。你会很疑惑，到底哪个是对的呢？为什么要有这两种不同的写法呢？

这两种写法都是正确的写法，但是它们却是有区别的。我们知道 C++ 已经有一些编写好的头文件（比如标准函数库等等），它们存放在 VC++ 的 `Include` 文件夹里。当我们使用 `#include <文件名>` 命令时，编译器就到这个文件夹里去找对应的文件。显然，用这种写法去包含一个我们自己编写的头文件（不在那个 `Include` 文件夹里）就会出错了。所以包含 C++ 提供的头文件时，应该使用尖括号。

相反地，`#include "文件名"` 命令则是先在当前文件所在的目录搜索是否有符合的文件，如果没有再到 `Include` 文件夹里去找对应的文件。因此，无论这个文件是 C++ 提供的还是自己编写的，使用 `#include "文件名"` 命令一定是正确的。这也正是书中本节之前的程序一律使用 `#include "文件名"` 命令的原因。



关于标准的尖括号

最新的 C++ 标准中，包含 C++ 提供的头文件并不是写作 `#include <文件名>`，如 `#include <iostream.h>` 的写法是过时的。正确的写法是 `#include <iostream>`，并且要使用 `std` 名字空间。有些程序中会有 `using namespace std;` 就是按照这种标准书写的。名字空间也称为命名空间，主要是用来避免大型程序开发中的标志符冲突。标准还规定了如何在 C++ 中包含 C 的头文件，有兴趣的读者可以到网上查阅这些资料。

尽管以上两种 `#include` 命令都可以正确地被 VC++ 识别了，但是它们却并不符合 C++ 的标准。标准规定，包含 C++ 提供的标准头文件或系统头文件时应使用尖括号，包含自定义头文件时可使用双引号。

鉴于这里已经交代清楚了如何按照标准来包含一个头文件，在之后的章节中，所有程序的 `#include` 命令将按标准来书写。

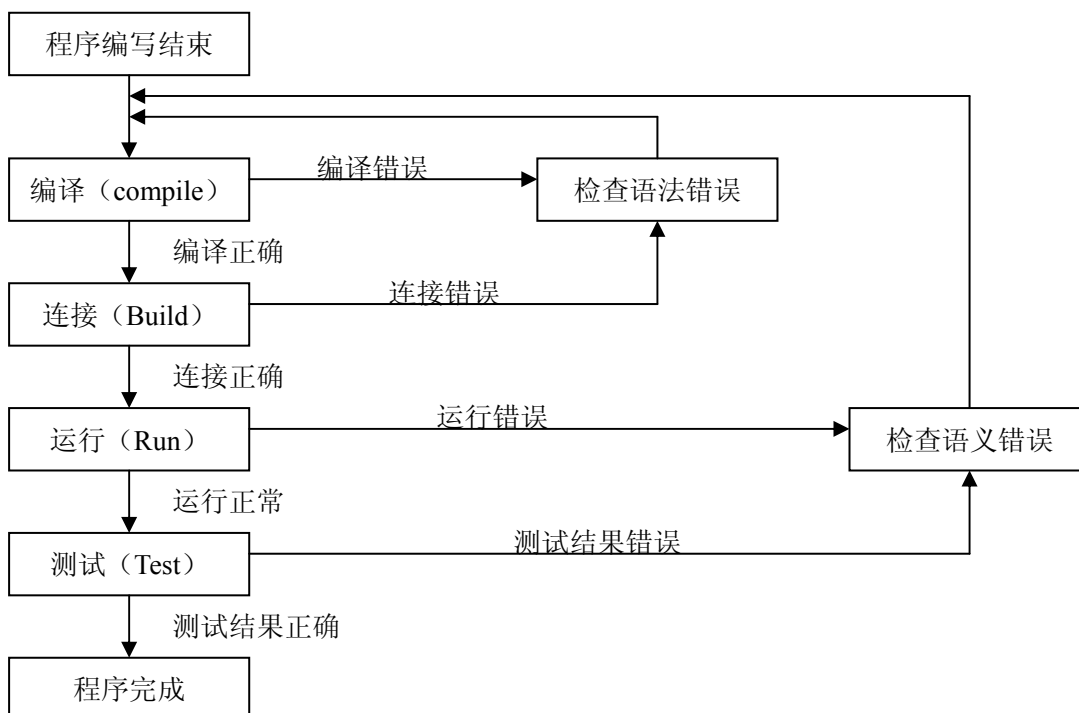
试试看：

如果包含头文件时写作如 `#include <iostream>`，但是没有 `using namespace std;`，即没有使用 `std` 名字空间，能否正常实现输入输出功能？

结论：如果按照这样的写法，必须要使用 `std` 名字空间。

## 11.3 更快更好地完成程序调试

当我们按照自己的思路编写好一个完整的程序，我们就要来对它进行调试 (Debug) 了。调试过程包括我们熟悉的编译和连接，还包括运行和简单的数据测试等等。下面就是调试的一个流程：



(图 11.3)

调试主要分四个步骤和两种处理方式。我们把程序的编译和连接统称为编译阶段，把程

序的运行和测试统称为运行阶段。在编译阶段发生的错误称为编译错误（Compile Error），在运行阶段发生的错误称为运行时错误（Runtime Error）。对于编译错误，我们通过检查并修正语法错误来解决；对于运行时错误，我们通过检查并修正语意（程序设计思想）错误来解决。

## 如何检查语法错误

所谓语法错误是指在书写语句时没有按照相应的语法格式。常见的语法错误有变量未定义、括号不匹配、遗漏了分号等等。大多数的语法错误都是能够被编译器发现的。因此相比于语意错误，语法错误更容易被发现，更容易被解决。

语法检查的工作由编译器完成，很多情况下编译器无法智能地报告出真正的语法错误数和错误位置。比如缺少一个变量的定义，而该变量在程序中被使用了 6 次，则编译器可能会报告 6 个甚至更多的语法错误，而实际上错误只有一个。所以，对编译器来说，任何一个语法错误都可能是“牵一发而动全身”的。

那么在这种可能发生“误报”的情况下，我们如何快速、正确地找到错误的位置呢？

由于编译器是按顺序查找语法错误的，所以它所找到的第一个错误的位置往往是正确的。如果程序规模不大，编译一次的时间不是很长，我们可以每次只修正编译器报告的第一个错误以及由此可以发现的连带错误，直到整个程序没有任何错误为止。

下面我们就用这种方法来检查一个程序的语法错误：（程序 11.3.1）

```
#include <iostream>
main()
{
 int a,b;
 for (i=0,i<3,i++)
 {
 cin >>a >>b;
 c=a+b;
 cout <<c <<endl;
 }
 return 0;
}
```

第一次编译的第一个错误：

```
I:\program\vc\book\11_3_1\main.cpp(5) : error C2065: 'i' : undeclared identifier//未声明的标识符
```

.....

```
main.obj - 7 error(s), 3 warning(s)//一共还有 7 个错误和 3 个警告
```

第一次修改：

```
for (int i=0,i<3,i++)
```

第二次编译的第一个错误：

```
I:\program\vc\book\11_3_1\main.cpp(5) : error C2143: syntax error : missing ',' before '<'//逗号语法错误
```

.....

```
main.obj - 8 error(s), 3 warning(s) //一共还有 8 个错误和 3 个警告
```

第二次修改：

```
for (int i=0;i<3;i++)
```

第三次编译的第一个错误:

```
I:\program\vc\book\11_3_1\main.cpp(7) : error C2065: 'cin' : undeclared identifier//未声明的标识符
```

.....

```
main.obj - 4 error(s), 3 warning(s) //一共还有 4 个错误和 3 个警告
```

第三次修改:

```
添加 using namespace std;
```

第四次编译的第一个错误:

```
I:\program\vc\book\11_3_1\main.cpp(9) : error C2065: 'c' : undeclared identifier//未声明的标识符
```

.....

```
main.obj - 1 error(s), 1 warning(s) //一共还有 1 个错误和 1 个警告
```

第四次修改:

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

第五次编译的第一个错误:

```
I:\program\vc\book\11_3_1\main.cpp(12) : warning C4508: 'mian' : function should return a value; 'void' return type assumed//函数需要返回一个值
```

.....

```
main.obj - 0 error(s), 1 warning(s) //一共还有 1 个警告
```

第五次修改:

```
int mian()
```

第六次编译:

```
main.obj - 0 error(s), 0 warning(s) //编译正确
```

第一次连接的第一个错误:

```
LIBCD.lib(crt0.obj) : error LNK2001: unresolved external symbol _main//没有 main 函数
```

.....

```
11_3_1.exe - 2 error(s), 0 warning(s) //一共还有 2 个错误
```

第六次修改:

```
int main()
```

第七次编译:

```
main.obj - 0 error(s), 0 warning(s) //编译正确
```

第二次连接:

```
11_3_1.exe - 0 error(s), 0 warning(s) //连接正确
```

完整的程序:

```
#include <iostream>
using namespace std;
int main()
{
 int a,b;
 for (int i=0;i<3;i++)
 {
 cin >>a >>b;
 int c=a+b;
 cout <<c <<endl;
 }
}
```

```

}
return 0;
}

```

至此，整个程序的所有语法错误都被检查出来并且被修正。程序编译阶段没有任何错误了。通过对这个程序的语法检查，我们总结出以下几点：

1、**编译器所报告的第一个错误位置往往是有效的，但是报告的错误内容未必正确。**比如第二次编译时报告的错误是“在小于号之前缺少了逗号”，而事实上问题是 for 语句中应该使用分号。所以，报告的错误内容只能参考，却不能完全相信。

2、**编译器报告的错误数目与实际错误数目未必符合。**甚至第一次改正一个错误后，错误数反而增加了。所以，报告的错误数目不能正确描述实际的错误规模。

3、**编译器报告的警告也应当被重视。**有些人认为即使程序存在警告，但是它能正常执行，所以警告可以被忽视。这种想法是错误的。如果一个程序是完美的，为什么编译器还要给出警告呢？警告的存在就说明了这个程序有些地方还不符合正确的语法。

4、I:\program\vc\book\11\_3\_1\main.cpp(5)括号中的 5 表示错误在程序的第 5 行。**我们不需要自己去数行数，只需要双击这个错误就能到达对应的行。**

## 常见语法错误及解决方法

由于大多数用户使用的是 VC++6.0 英文版（市场上的中文版实际上是汉化版），在面对编译器报告的各种错误时，可能会觉得茫然。这也是为什么本书总是在一些专业术语后加上对应的英文名称的原因。读者如果能够掌握这些术语的英文名，看懂错误报告也不会很难。

在下表列出了一些常见的语法错误及解决方法。另外在本书的附录中，有更多的语法错误参考信息。如果读者遇到无法解决的语法错误，可以查阅本书附录或其他语法工具书。

| 错误信息                                                     | 中文翻译                | 可能的解决方法                  |
|----------------------------------------------------------|---------------------|--------------------------|
| Ambiguous call to overloaded function                    | 函数重载发生歧义            | 修改重载函数参数表，避免歧义           |
| Cannot open include file                                 | 无法打开包含文件            | 查找包含文件的正确路径              |
| conversion from 'double' to 'int', possible loss of data | 从双精度变量转换为整型变量可能丢失数据 | 强制类型转换                   |
| function does not take 0 parameters                      | 函数参数表不匹配            | 正确地调用函数                  |
| function should return a value                           | 函数必须返回一个值           | 在函数原型中给出返回值类型            |
| local variable 'a' used without having been initialized  | 局部变量未初始化            | 初始化该变量                   |
| missing ')' before                                       | 缺少括号                | 检查括号是否匹配                 |
| missing ';' before                                       | 缺少分号                | 在语句尾加上分号                 |
| redefinition                                             | 标志符重复定义             | 更改标志符名                   |
| undeclared identifier                                    | 标识符未定义              | ①定义该标志符<br>②检查是否正确包含了头文件 |
| unexpected end of file found                             | 文件异常结束              | 检查语句块括号是否匹配              |
| unreferenced local variable                              | 未引用的局部变量            | ①使用该变量<br>②如果是多余变量，则删去   |
| unresolved external symbol _main                         | 找不到主函数              | 编写一个主函数                  |

## 11.4 最麻烦的问题

在调试过程中，运行时错误是最麻烦的问题。因为编译错误可以由编译器检查出来，而大多数编译器对运行时错误却无能为力。查错和纠错的工作完全由用户自己来完成。

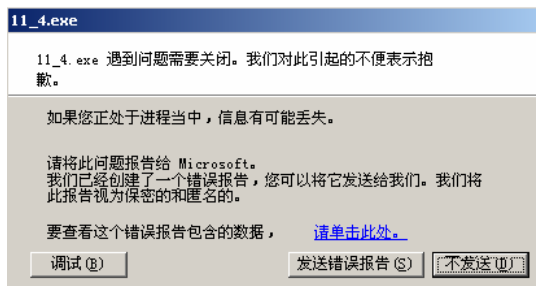
运行时错误还分为两种：一种是由于考虑不周或输入错误导致程序异常（Exception），比如数组越界访问，除数为零，堆栈溢出等等。另一种是由于程序设计思路的错误导致程序异常或难以得到预期的效果。

对于第一类运行时错误，我们不需要重新设计解决问题的思路，认为当前算法是可行的、有效的。我们只需要找出输入的错误或考虑临界情况的处理方法即可。对于第二类运行时错误，不得不遗憾地说，一切都要从头再来。

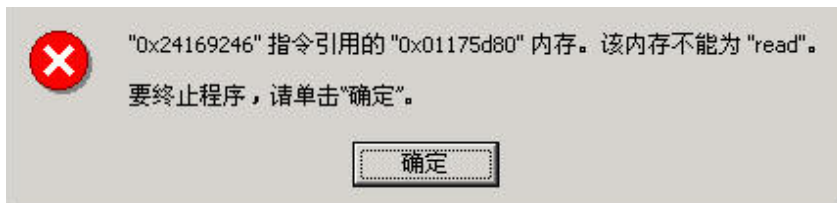
### 见识运行时错误

由于编译器无法发现运行时错误，这些错误往往是在程序运行时以五花八门的形式表现出来。下面就是典型的几种因运行时错误引起的问题：

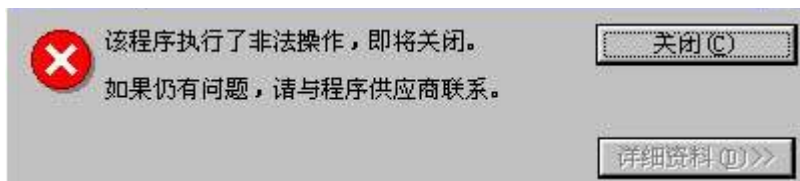
#### (1) WindowsXP 错误报告



#### (2) 内存不能为 Read/Written



#### (3) 非法操作



#### (4) Debug 错误



## 查找错误点

语法错误的位置能很快地被编译器找到，而运行时错误的位置却很难被我们发现。即使我们一条条地检查语句，也未必能检查出什么。所以，在这里要介绍一种查找导致运行时错误的语句的方法。

我们知道，带有运行时错误的程序是可以运行的。当它运行到一个产生错误的语句时，就提示出错了。根据这个特点，我们可以用输出语句来判断程序的运行流程。下面就让我们来看一段有运行时错误的程序：（程序 11.4）

```
#include <iostream>
using namespace std;
int main()
{
 char a[5],b[5];
 int alen=0,blen=0;//记录字符串 a 和 b 的长度
 cin >>a >>b;
 for (int i=0;a[i]!='\0' && b[i]!='\0';i++)//计算字符串的长度
 {
 if (a[i]!='\0')
 alen++;
 if (b[i]!='\0')
 blen++;
 }
 char *c=new char[alen+blen];//申请堆内存，存放连接后的字符串
 for (i=0;i<=alen+blen;i++)//把字符串 a 和 b 连接复制到字符串 c
 {
 if (i<alen)
 c[i]=a[i];
 else
 c[i]=b[i-alen];
 }
 cout <<c <<endl;
 delete [] c;//释放堆内存
 return 0;
}
```

运行结果：

OOTTMA

TomatoStudio

udioTomat 茸茸茸茸痛

在程序运行结束之前，提示 Debug Error，它属于一种运行时错误。而且根据输出的一些内容，发现程序也没有达到连接字符串的目的。所以我们让程序输出更多信息，查找错误原因。首先在计算字符串 a 和 b 的长度后，输出他们的长度，即在第一个 for 语句后添加一句 `cout <<"alen=" <<alen <<"blen=" <<blen <<endl;`。

运行结果：

OOTTMA

```
TomatoStudio
```

```
alen=4blen=4
```

```
udioTomat 葺葺葺葺瘡
```

OOTTMA 字符串长为 6，TomatoStudio 字符串长为 12。根据程序运行结果，我们发现计算出的字符串长度有问题。所以我们必须检查实现该功能的语句。另外，由字符串长度我们可以想到申请空间是否足够的问题。发现数组的空间只能存放 5 个字符，而现在两个字符串都已经超过这个限制。于是把数组空间扩大，该作 `char a[20],b[20];`。

运行结果：

```
OOTTMA
```

```
TomatoStudio
```

```
alen=6blen=6
```

```
OOTTMATomatoS 葺葺瘡
```

发现字符串 a 的长度已经正确，可是字符串 b 的长度为什么不对呢？经过多次尝试，我们发现，正确的字符串长度总是较短的字符串。所以我们想到检查循环继续的条件是否正确，如果过早地终止循环，就会导致这种情况。果然，`a[i]!='\0' && b[i]!='\0'`意味着只要有一个字符串结束，那么长度计算就结束了，故把`&&`改成`||`。

运行结果：

```
OOTTMA
```

```
TomatoStudio
```

```
alen=35blen=41
```

```
OOTTMA
```

这么一改，居然两个长度全都错了。我们不禁要思考为什么会这样了：用一个 for 语句来计算两个字符串的长度，当循环变量越过任一个字符串的结尾符以后又误认为它没有结束，所以输出的长度远远长于字符串的实际长度。我们把计算字符串长度用两个 for 语句来实现。即程序被改写成这样：

```
#include <iostream>
using namespace std;
int main()
{
 char a[20],b[20];
 cin >>a >>b;
 for (int alen=0;a[alen]!='\0';alen++);//计算字符串 a 的长度
 for (int blen=0;b[blen]!='\0';blen++);//计算字符串 b 的长度
 cout <<"alen=" <<alen <<"blen=" <<blen <<endl;
 char *c=new char[alen+blen];
 for (int i=0;i<=alen+blen;i++)
 {
 if (i<alen)
 c[i]=a[i];
 else
 c[i]=b[i-alen];
 }
 cout <<c <<endl;
 delete [] c;
```

```

 return 0;
}

```

运行结果:

```
OOTTMA
```

```
TomatoStudio
```

```
alen=6blen=12
```

```
OOTTMATomatoStudio
```

现在两个字符串的长度都正确了，输出的内容也实现了字符串的连接，但是 Debug Error 仍然存在。继续检查，发现剩下的语句和申请的堆内存空间字符串 c 有关了。于是先检查 c 是否有越界访问。根据 c 申请的空间大小，发现 for 语句中循环继续的条件有错误，导致越界访问，把它改成 `i<alen+blen;`。

运行结果:

```
OOTTMA
```

```
TomatoStudio
```

```
alen=6blen=12
```

```
OOTTMATomatoStudio @
```

Debug Error 已经没有了，看来造成这个错误的原因就是越界了。但是现在输出的字符串后面有乱码，可能是结尾符被忽略了。检查程序，发现 `alen+blen` 是两字符串长度，但是没有考虑结尾符，所以要给字符串 c 增加一个字符的空间。程序改写成如下：

```

#include <iostream>
using namespace std;
int main()
{
 char a[20],b[20];
 cin >>a >>b;
 for (int alen=0;a[alen]!='\0';alen++);
 for (int blen=0;b[blen]!='\0';blen++);
 //cout <<"alen=" <<alen <<"blen=" <<blen <<endl;
 char *c=new char[alen+blen+1];
 for (int i=0;i<alen+blen+1;i++)
 {
 if (i<alen)
 c[i]=a[i];
 else
 c[i]=b[i-alen];
 }
 cout <<c <<endl;
 delete [] c;
 return 0;
}

```

运行结果:

```
OOTTMA
```

```
TomatoStudio
```

```
OOTTMATomatoStudio
```




至此，程序修改完成。在目前的测试数据下，不再出现运行时错误，并且也能实现字符串连接的功能。

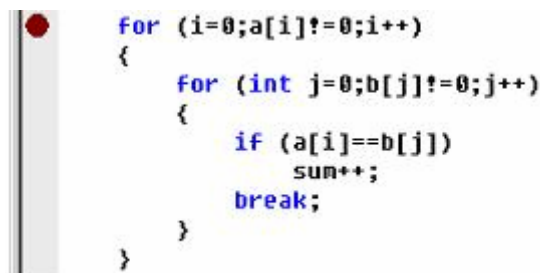
## 11.5 调试工具——Debug

由于引起运行时错误的原因难以被发现，所以我们有时候要利用工具来完成调试工作。Debug 就是 VC++ 提供的一种常用调试工具。它能够让语句一句一句或一段一段执行，并且能够观察程序运行过程中各变量的变化情况。


在介绍如何使用 Debug 工具之前，我们要介绍一下什么是断点（Breakpoint）。当程序运行到断点的时候，它会暂时停止运行后面的语句，供用户观察程序的运行情况，并等待用户发出指令。断点不是语句，而是附在某条语句上的一个标志。

### 如何设置和移除断点


点击需要设置断点的语句，使光标移动到该语句所在的行。按下 F9 键或  按钮就会发现，在该语句之前出现一个红点，这就是断点标志。如下图 11.5.1 所示：



(图 11.5.1)

如果要移除已经设置好的断点，则同样点击断点所在语句，按下 F9 键或  按钮则断点被移除。我们可以给一个程序设置多个断点。

### Go

设置了断点之后，我们就能开始调试程序了。与以前不同，我们不能按执行按钮，而是要按 F5 键或  按钮，或者选择 Build 菜单 Start Debug 中的 Go。一旦按下了 Go，则程序会正常运行直至遇到断点。

我们以下面这个程序（程序 11.5）来演示 Debug 功能的使用。该程序主要目的是统计一个不多于 20 项的正整数数列中，有多少对成双倍关系的项，该数列以 0 结尾。比如数列 1 3 4 2 5 6 0 中，成双倍关系的项有 3 对（1 和 2、2 和 4、3 和 6）。

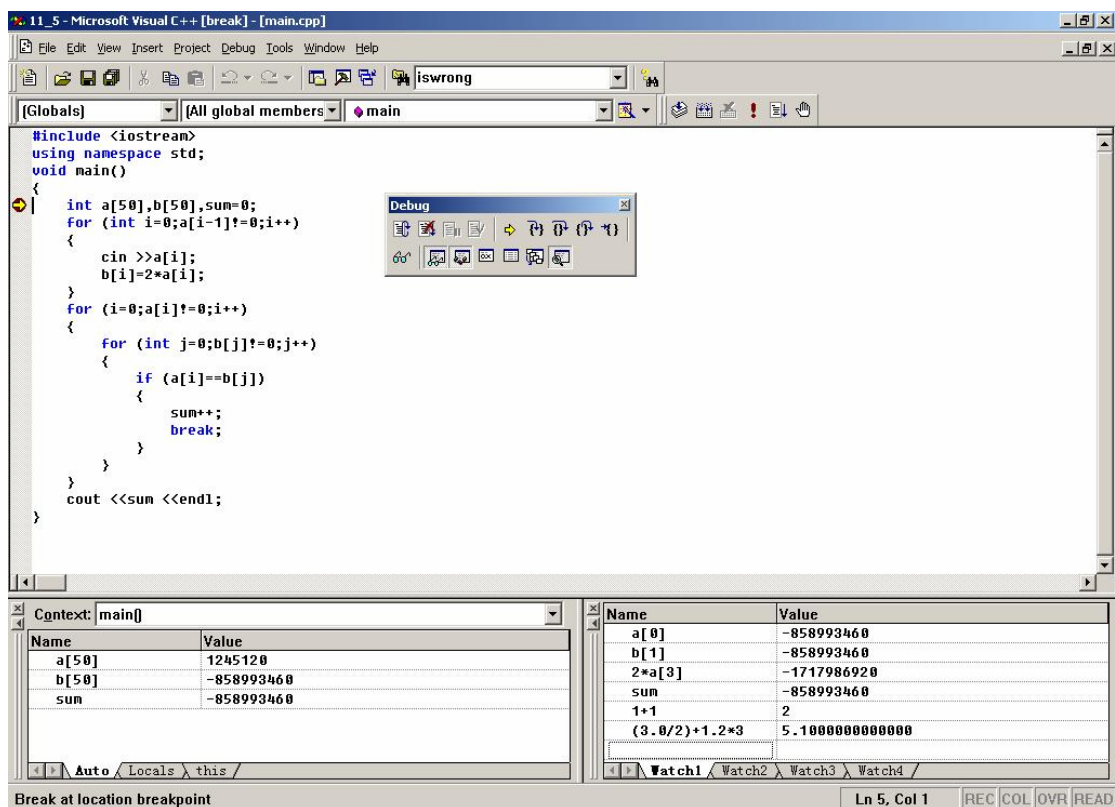
```
#include <iostream>
using namespace std;
int main()
{
 • int a[50],b[50],sum=0;//在此设置断点
 for (int i=0;a[i-1]!=0;i++)
```

```

{
 cin >>a[i];
 b[i]=2*a[i];
}
for (i=0;a[i]!=0;i++)
{
 for (int j=0;b[j]!=0;j++)
 {
 if (a[i]==b[j])
 {
 sum++;
 break;
 }
 }
}
cout <<sum <<endl;
return 0;
}

```

设置好断点，按下 Go 按钮以后，我们可以看到如下的界面：



(图 11.5.2)

在界面中出现了三个我们不熟悉的窗口。在屏幕中间有着很多按钮的小窗口叫 Debug 窗口，里面的按钮可以控制程序继续运行的方式。在屏幕左下方的窗口称为 Variables（变量）窗口，可以观察每句语句执行后变量变化的情况。在屏幕右下方的窗口称为 Watch（监视）窗口，用户可以监视一些变量或简单表达式的变化情况。

## Debug 窗口



Debug 窗口中，第一行按钮是我们常用的。它们依次是：

Restart——重新开始调试。

Stop Debugging——停止当前调试。

Break Execution——停止程序的执行并转回调试状态。

Apply Code Changes——使调试过程中修改的程序代码生效。

Show Next Statement——显示将要执行的下一条语句的位置。在语句之前用黄箭头表示。

Step Into——进入语句调用的函数，并进行调试。

Step Over——不调试语句调用的函数。

Step Out——从当前调试的位置回到调用该函数的位置。

Run to Cursor——正常运行直到光标所在的语句。

我们在调试的时候，不要总是按“Step Into”，因为它对于一些系统提供的函数也是有效的。也就是说我们能够用它详细地看到系统是如何实现一个输出功能的，甚至可以看到这些语句的汇编语言形式。但是，这却并不是我们调试的主要目标。如果不小心进入了系统函数里，我们要及时按“Step Out”以退回到我们所编写的程序中。

在调试过程中，对于大多数语句应该按“Step Over”。如果要调试自己编写的函数，则在调用该函数的语句处按“Step Into”。

## Watch 窗口

在 Watch 窗口中分为两列，一列为 Name，一列为 Value。其中 Name 是可以被编辑的，我们可以在里面输入变量名或简单表达式。如果改变量或表达式是可以被计算的，则会在 Value 中显示它们的值，如下图 11.5.3 所示：

| Name          | Value            |
|---------------|------------------|
| a[0]          | 1                |
| b[1]          | 6                |
| 2*a[3]        | 4                |
| sum           | 3                |
| 1+1           | 2                |
| (3.0/2)+1.2*3 | 5.10000000000000 |

(图 11.5.3)

## 如何用 Debug 找到错误

在 Debug 中，我们可以让语句一句句地执行。如果执行到某一句语句时发生了运行时错误，那么这个错误一般就是由这个语句引起的。

在 Debug 中，我们可以观察每一句语句执行的顺序和执行后变量变化的情况。如果发现程序无法实现既定的功能，我们可以将期望的结果和实际的结果作比对，并分析可能引起这些不同的原因。这样一来，大大加快了我們找到问题和解决问题的速度。

试试看：  
用 Debug 工具调试程序 11.5，观察语句执行的顺序和变量的变化情况。

## 习题

1、写出下列程序段中所有变量的作用域，并描述它们的可见性。

```
int a;
int func()
{
 static int c=0;
 double a=0,b=1;
 return c++;
}
int main()
{
 if (int a=1)
 {
 double a=2,b=5,c;
 c=a+b;
 }
 else
 {
 char a='a',b='b';
 }
 for (int i=0;i<3;i++)
 func();
 return 0;
}
```

2、根据要求调试给出的程序（代码中可能含有编译错误，也可能含有运行时错误）。

①有一个二次函数 $f(x)=ax^2+bx+c$ ，现输入 $f(0)$ ， $f(1)$ 和 $f(2)$ 的值，并输入一个实数 $n$ ，请输出 $f(n)$ 的结果。

运行结果示例：

```
1 2 3 4
```

```
5
```

已经完成的程序：

```
#include "iostream"
double func(int n)
{
 return a*pow(2,n)+b*n+c;
}
void mian()
{
```

```

double a,b,c;
double d,e,f,n;
cin >>d >>e >>f >>n;
a=(f-2*e+d)/2;
b=e-a-d;
c=d;
cout <<func(n) <<endl;
return 0;
}

```

②在计算机网络中的 IP 地址是由 32 位二进制数组成的。为了方便记忆，我们经常将每 8 位二进制转化为一个十进制的数。这个数的范围为 0 到 255。所以，一个 IP 地址就由 4 个这样的十进制数构成。现在输入一个 32 位二进制数形式的 IP 地址，请输出该地址的十进制数形式。

运行结果示例：

```
00101011110010111101110110101011
```

```
43.203.221.171
```

已经完成的程序：

```

#include <iostream>
int main()
{
 char ip[32];
 cin <<ip;
 int sub=0;
 for (int k=0;k<4;k++)
 {
 int address=0,temp=128;
 for (int l=0;l<8;l++)
 {
 address=address+ip[sub]*temp;
 temp=temp/2;
 sub++;
 }
 cout <<address <<'!';
 }
 return 0;
}

```

③任选一个两位数  $x$ ，取它的个位数  $a$  和十位数  $b$ ，将两者相加得到  $c$ ，再用  $x$  减去  $c$ ，得到结果  $d$ 。比如  $x=84$ ，则  $c=8+4=12$ ， $d=84-12=72$ 。请求出当  $x$  的取值范围为 10 到 99 时， $d$  的值域。

运行结果示例：

```

9
18
27
36

```

45

54

63

72

81

已经完成的程序：

```
#include <iostream>
using namespace std;
int main()
{
 int a=0;
 for (int i=10;i<100;i++)
 {
 cout <<i-i/10+i%10 <<endl
 }
 return 0;
}
```

## 第十二章 如何编写程序代码

当我们学会如何阅读代码，如何调试程序后，我们就要更进一步了——学习如何编写程序。如果说之前我们是扶着旁物蹒跚学步，那么现在我们才真正地迈出第一步。本章主要介绍如何把一个实际问题用一个程序来解决，也让大家对程序、软件的开发有一个简单的认识。

### 12.1 程序设计的基本步骤

当我们遇到一个问题，它往往不是一个直接用程序代码描述的问题，比如：统计销售量和利润，寻找出行的公交线路，将中文翻译成英语等等。所以我们先要把实际问题转化成一个人电脑能够解决的问题，而大多数问题一般分为三类：

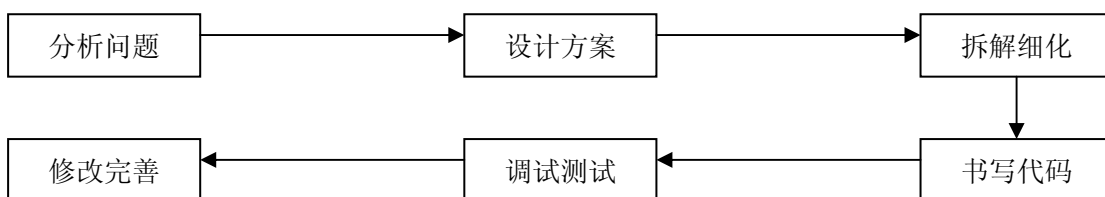
- (1) 算：计算利润，计算一元二次方程的根，计算一个数列的和等等。
- (2) 找：找最大的一个数，找最短的一条路径，找一个字符串的位置等等。
- (3) 实现功能：实现撤销、重做的功能，实现模拟某种操作的功能等等。有时候实现功能问题可以拆解为若干个“算”和“找”的问题。

正确分析了实际问题问题，并将其转化为上述三种电脑能够解决的问题之后，我们便要开始设计解决问题的方案了。**设计解决问题的方案需要考虑算法和数据结构两方面**。在第一章我们已经说明，算法是解决一类问题的过程和方法。而数据结构，目前我们可以简单的理解为数据存储的形式，比如是变量、数组还是链表等等。

设计完方案，我们要将一整套复杂的方案拆解为一个个简单的小模块。拆解的程度根据代码的可读性和可写性来决定。如果某个模块比较复杂，那么会给代码书写和程序调试带来麻烦，也会给日后阅读程序代码带来困难。在实际书写代码的过程中，这些小模块往往以函数的形式出现。

至此，我们已经做完了所有的准备工作，可以上机书写程序代码了。在代码书写过程中可能会遇到一些没有考虑到的小问题，所以可能要适当地对之前设计的方案有所修正。

最后，不断地对程序进行调试和修改，一个程序便能根据要求来解决一些实际问题了。整个程序设计的过程如下图所示：



(图 12.1)

### 12.2 三类问题

我们已经把电脑能解决的问题归结为“算”、“找”和“实现功能”三类。这一节我们通过三个例子，来讲述如何解决这三类问题。

## 算

问题：国际象棋是古印度的一个大臣发明的。国王很喜欢这个国际象棋，于是召见了他，问他要什么。那个大臣说：如果可以的话，陛下就给我一些麦子吧！在棋盘的第一格放一粒麦子，第二格放两粒麦子，第三格放四粒麦子，第四格放八粒麦子，以此类推。于是国王就吩咐属下去拿麦子，可是拿来的一袋麦子很快就放完了。如果一粒麦子的平均重量在 0.02 克，请你帮国王算一下，大臣要的这些麦子一共有多重？

分析：这是一个数列求和的问题，我们只要把 64 格中每格的麦子数加起来，再乘以 0.02，就是所有麦子的重量（克）。由于计算比较简单，就不必将其更加细化了。以下是程序代码：（程序 12.2.1）

```
#include <iostream>
using namespace std;
int main()
{
 double item=1,sum=0;
 for (int i=0;i<64;i++)
 {
 sum=sum+item;
 item=item*2;
 }
 cout <<"这些麦子重" <<sum*0.02/1000000 <<"吨。" <<endl;
 return 0;
}
```

运行结果：

这些麦子重 3.68935e+011 吨。

放满 64 格麦子的总重量居然有上千亿吨！看来国王这下亏大了。不过程序 12.2.1 也很亏，本来可以几句代码解决的问题被搞得那么复杂了。因为等比数列求和在数学上可以用公式解决，那可比用 for 循环一项项计算要快多了！

算法时间：“算”的关键——数学知识

在很多情况下，计算一些数据是可以利用数学方法简化的。比如上述例子中的等比数列求和问题，如果这个数列的项数很多，那么两种方法的性能差距将是非常明显的。所以，要解决好“算”问题，关键是要有良好的数学基础，在分析问题的时候就将其最简化。这样，不论对程序的长度控制还是性能提高都有很大的帮助。

## 找

问题：Peter 以前是一个作家，每天要写很多文章。随着年龄增加，他的记性也不好了。有时候刚写过的东西，他就忘了。于是他请我们帮他编写一个程序，告诉他是否曾经写过某个单词。如果写过，则告知匹配的字符位置；如果没有写过，则告知没有写过。

运行结果示例：

他写过的文字：

Iamawriter,whataboutyou?

查询单词：what

查询结果：12



分析：这是一个字符串匹配的问题。任务的要求是在一个字符串当中查找子串的位置。在第五章介绍过要找到答案，我们常用穷举法。穷举法可以说是“找”问题的基本解决方法。我们把首个字符的位置固定，然后依次向后比对字符串是否符合，如果子串的字符全都匹配成功，说明存在该子串。如下图所示：

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |       |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|-------|
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
| F |   |   |   |   |   |   |   |   |   |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   | F |   |   |   |   |   |   |   |   |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   | F |   |   |   |   |   |   |   |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   | F |   |   |   |   |   |   |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   |   | w | F |   |   |   |   |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   |   |   | F |   |   |   |   |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   |   |   |   | F |   |   |   |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   |   |   |   |   | F |   |   |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   |   |   |   |   |   | F |   |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   |   |   |   |   |   |   | F |   |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   |   |   |   |   |   |   |   | F |   |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   |   |   |   |   |   |   |   |   | F |   |   |   |       |
| I | a | m | a | w | r | i | t | e | r | , | w | h | a | t | ..... |
|   |   |   |   |   |   |   |   |   |   |   |   | w | h | a | t     |

(图 12.2.1，绿色表示匹配成功，红色表示匹配失败)

由于数据是字符串，所以我们考虑采用字符数组来存储这些字符串。以下是程序代码：

(程序 12.2.2)

```
#include <iostream>
using namespace std;
int main()
{
 int i,j;//考虑到变量的作用域问题
 char orgstr[100],substr[100];
 cout <<"他写过的文字: " <<endl;
 cin >>orgstr;
 cout <<"查询单词: ";
 cin >>substr;
 for (i=0;orgstr[i]!='\0';i++)
 {
 for (j=0;substr[j]!='\0';j++)
```

```

 {
 if (orgstr[i+j]!=substr[j])
 {
 break;
 }
 }
 if (substr[j]=='\0')
 break;
}
cout <<"查询结果: ";
if (orgstr[i]=='\0')
{
 cout <<"Peter 没有写过这个单词。" <<endl;
}
else
{
 cout <<i+1 <<endl;//数组下标从 0 开始
}
return 0;
}

```

问题是解决了，但是这样的找法仍然是性能较差的。比如发现 `writer` 的 `r` 和 `what` 的 `h` 不匹配之后，再拿 `what` 的 `w` 和 `writer` 的 `r` 去比较是多余的。因为在上一轮比较中，我们已经知道那是 `r` 而不是 `w` 了。

算法时间：“找”的关键——减少分支，缩小范围

我们在使用穷举法的时候会查找所有的情况，最终找到合理的答案。但是过多无效的查找拖累了程序的运行效率。所以“找”的关键就是减少查找的分支，缩小查找的范围。当然，这里的减少和缩小是在不影响结果的正确性下进行的。我们要缩减的是显而易见的错误答案和可以预知的错误答案。比如上述例子中，如果下一轮比较的起始位置没有确定且当前字符不是子串的首个字符，显然下一轮比较至少要从当前字符的下一个字符开始。这样，就能减少循环执行的次数了。在以后的数据结构课程中，将学习到 KMP 模式匹配算法，那种算法更为高效。

## 实现功能

问题：有  $n$  个孩子围成一个圈，他们按顺时针编号依次为 1 到  $n$ 。有一个整数  $m$ ，现在从第一个孩子开始顺时针数  $m$  个孩子，则那个孩子离开这个圈。从下一个孩子继续数  $m$  个孩子，则那个孩子离开这个圈。如此继续，直到最后剩下的孩子胜出。如果知道孩子的个数  $n$  和整数  $m$ ，请你预测一下编号为多少的孩子会胜出？

运行结果示例：

请输入孩子的个数：8

请输入正整数  $m$ ：2

第 1 个孩子将获得胜利！

分析：这是有名的约瑟夫环问题。程序的要求是模拟一下游戏的过程，并预测游戏结果。而游戏规则已经告知了。这里主要考虑如何直观地表示孩子围成的一个圈。这不难让我们联

想到了链表，如果将它首尾相接，不就是一个圈么？每离开一个孩子就删除一个结点，直到只剩下一个结点。程序可以分为两个模块，一个负责“圈”的初始化，另一个负责模拟游戏过程。以下是程序代码：（程序 12.2.3）

```
#include <iostream>
using namespace std;
struct child
{
 int num;
 child *link;
};
void init(int n);//初始化函数
void gameStart(int n,int m);//模拟游戏函数
child *head;//链表头
child *present;//当前结点
child *end;//链表尾
int main()
{
 int n,m;
 cout <<"请输入孩子的个数: ";
 cin >>n;
 cout <<"请输入正整数 m: ";
 cin >>m;
 init(n);
 gameStart(n,m);
 cout <<"第" <<present->num <<"个孩子将获得胜利! " <<endl;
 delete present;
 return 0;
}
void init(int n)
{
 head=new child;
 head->num=1;
 present=head;
 for (int i=1;i<n;i++)
 {
 present->link=new child;
 present->link->num=i+1;
 present=present->link;
 }
 present->link=head;
 end=present;
 present=head;
}
void gameStart(int n,int m)
```

```

{
 child *pGuard=end;//指向待删除结点的前驱结点，起初应指向表尾
 while (n!=1)
 {
 for (int j=1;j<m;j++)
 {
 pGuard=present;
 present=present->link;
 }
 pGuard->link=present->link;
 delete present;
 present=pGuard->link;
 n--;
 }
}

```

算法时间：“实现功能”的关键——方法正确，简洁高效

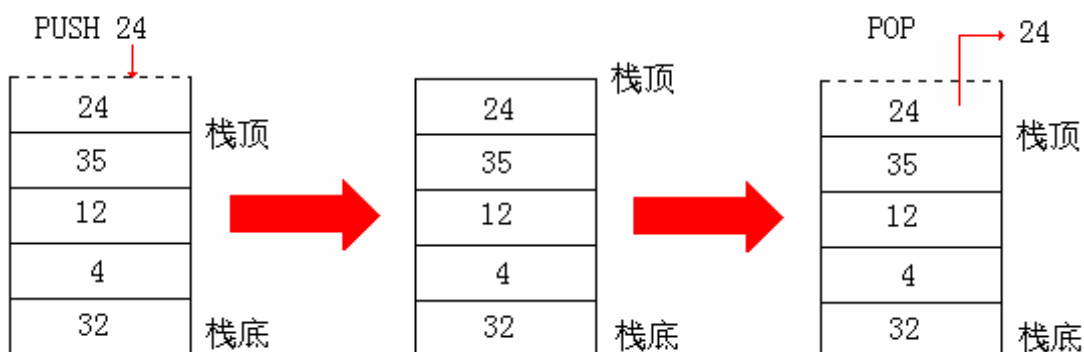
在解决“实现功能”类的问题时，我们要采用合适的方法和数据存储方式，尽量用直观的方式来解决问题。在保证结果的正确性下，我们应尽量选择简洁高效的方法，避免复杂的方法和冗余的操作，以免影响程序的运行效率。

## 12.3 函数的递归

在程序设计中，递归是一种常用的方法。虽然它的运行效率不是很高，但是其代码简洁易懂。在硬件高速发展的今天，递归方法深受程序员的青睐。在详细介绍递归之前，我们先要知道什么是栈，以及函数的调用机制。

### 什么是栈

栈（Stack）是一种存放数据的空间。它的特点是数据先入后出，即最先进入栈的数据需要等到最后才能出栈。就像一把枪的弹匣，先被压入的子弹到最后才被打出来，而最后压入的子弹在一开始就会被打出来。栈有两种操作，分别是压栈（Push）和退栈（Pop），如下图所示：



（图 12.4.1）

如图退栈之后，再退栈的数据依次为：35、12、4、32。最终整个栈为空，里面没有任何数据。

## 函数的调用机制

当调用一个函数时，其实系统内部发生了一系列的动作：

- (1) 开辟该函数的栈空间。
- (2) 将当前运行状态压栈。
- (3) 将返回地址（调用函数的地方）压栈。
- (4) 在栈内分配参数空间，传递参数信息。
- (5) 执行被调用函数，如果有局部变量，则在栈内分配空间。

假设某函数调用了函数 A，函数 A 中又调用了函数 B，那么栈里面的情况是如何的呢？如下图所示：

|          |               |
|----------|---------------|
| B 函数的栈空间 | B 函数中的局部变量    |
|          | 参数            |
|          | 返回地址（A 函数中某处） |
|          | 调用 B 函数时的运行状态 |
| A 函数的栈空间 | A 函数中的局部变量    |
|          | 参数            |
|          | 返回地址          |
|          | 调用 A 函数时的运行状态 |
| 其他函数的栈空间 | .....         |

(图 12.4.2)

当函数运行结束时，系统内部又有一系列的动作，这些恰巧与调用函数时的动作顺序相反：

- (1) 释放栈内局部变量空间。
- (2) 释放栈内参数空间。
- (3) 退栈，得到返回地址，程序跳转回调用函数处等待继续执行。
- (4) 退栈，得到程序运行状态，恢复调用函数前的运行状态。
- (5) 释放该函数的栈空间。

了解函数运行结束后的动作，我们不难理解为何参数和局部变量在函数运行结束后就消失了。因为他们存放在一个栈中，函数运行结束后，这部分空间就会被释放。一旦有新的函数运行，那么栈里面的数据就会发生变化。

以图 12.4.2 为例，如果函数 B 结束运行后，则栈的情况如下图所示：

|          |               |
|----------|---------------|
| A 函数的栈空间 | A 函数中的局部变量    |
|          | 参数            |
|          | 返回地址          |
|          | 调用 A 函数时的运行状态 |
| 其他函数的栈空间 | .....         |

(图 12.4.3)

待 A 函数也运行完之后，那么栈顶的数据就是调用 A 函数的某函数的数据了。如下图所示：

## 栈

|          |       |
|----------|-------|
| 其他函数的栈空间 | ..... |
|----------|-------|

(图 12.4.4)

需要指出的是，栈也是内存中的一部分空间。既然内存空间是有限的资源，那么栈空间也是一种有限的资源。所以如果过于频繁地调用新函数运行而不让已调用的函数结束运行，那么栈资源会渐渐减少，以至于导致栈空间不足，最终因栈溢出（Overflow）而使程序出错。

## 小试牛刀——用递归模拟栈

既然调用函数的实质是栈操作，那么我们可以用递归函数来模拟栈：

我们来编写一个程序，先读取一个字符串，该字符串以“#”结尾。经过程序处理后，让这些字符按输入时的逆序输出，即符合栈“先入后出”的特点。整个程序代码如下：（程序 12.4.1）

```
#include <iostream>
using namespace std;
void stack(char c);
int main()
{
 char d;
 cin >>d;
 stack(d);
 cout <<endl;
 return 0;
}
void stack(char c)
{
 if (c!='#')//判断是否输入结束
 {
 char d;
 cin >>d;//没有结束则继续输入数据
 stack(d);//继续判断输入的数据
 cout <<c;//待输入结束则将字符输出
 return;//返回调用处，去输出上一个字符
 }
 else
 {
 return;
 }
}
```

运行结果：

ABCD#

DCBA

如果仅仅关注参数在栈中的情况，则该程序运行时，栈的情况如下图所示：

|    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|
| 调用 | 调用 | 调用 | 调用 | 调用 | 返回 | 返回 | 返回 | 返回 |
|    |    |    |    | #  |    |    |    |    |
|    |    |    | D  | D  | D  |    |    |    |
|    |    | C  | C  | C  | C  | C  |    |    |
|    | B  | B  | B  | B  | B  | B  | B  |    |
| A  | A  | A  | A  | A  | A  | A  | A  | A  |

(图 12.4.5, 绿色表示输入的字符 d, 红色表示输出的字符 c)

这样的程序是不是比自己编写一个链表栈要简单方便得多了呢?

### \*递归的精髓

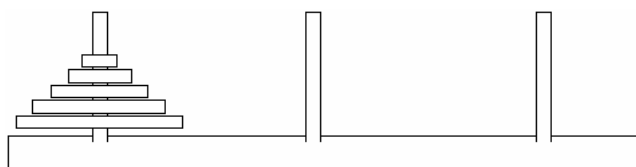
我们必须知道何时可以采用递归。并不是所有问题都能够用递归解决的, 而某些问题却是只能靠递归解决的。如果一个问题可以通过递归来解决, 则我们称这个问题的解法是递归的。那么, 可以通过递归解决的问题有什么特点呢?

可通过递归解决的问题往往能够转化为若干个解决步骤, 并且第 n 步和第 n+1 步有着类似或相同的限制条件。比如一个数列的递推公式:

$a_n=3a_{n-1}+2, a_{n-1}=3a_{n-2}+2, a_{n-2}=3a_{n-3}+2, \dots, a_1=3a_0+2, a_0=1$ , 它的相邻两项有着相同的关系, 即求第 n 项和第 n+1 项有着相同的方法。唯一的不同就是  $a_0=1$ , 我们把这种与众不同的方法称为递归出口, 不断调用函数的递归将在那里终止。用递归来解决问题必须要有递归出口, 否则递归会不断进行, 直到栈溢出。

下面我们以汉诺塔问题为例, 看看如何将问题转化为若干个相似的步骤, 再用递归来解决问题:

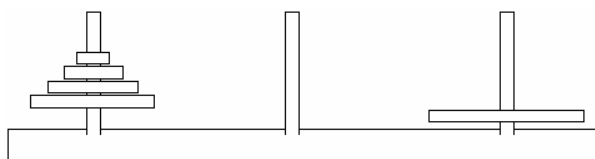
传说印度的一座神庙里有三根石柱, 左侧的石柱从下而上有 64 个逐渐变小的圆盘, 中间的石柱和右侧的石柱则是空着的 (如下图所示)。和尚要把左侧的石柱按一定的规则全都移动到右侧的石柱上去: 每次只能移动一个圆盘, 并且小圆盘必须在大圆盘之上。当这 64 个圆盘全部移动到右侧石柱上去之后, 就是世界末日。



(图 12.4.6)

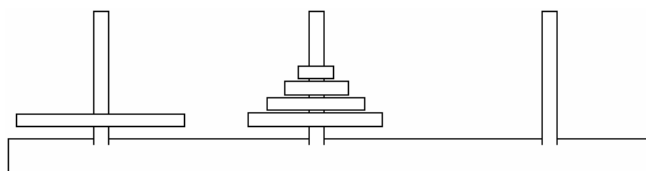
现在我们假设汉诺塔有 n 个盘子, 要用程序来将其移动的方案输出。经过分析, 我们发现:

解决 n 个盘子的汉诺塔问题等价于解决第 n 个盘子在最右侧石柱的 n-1 个盘子的汉诺塔问题, 解决 n-1 个盘子的汉诺塔问题等价于第 n-1 个盘子在最右侧石柱的 n-2 个盘子的汉诺塔问题……如下图所示:

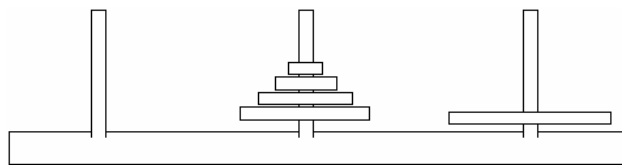


(图 12.4.7)

要由图 12.4.6 的形式转化为图 12.4.7 的形式, 必须经过以下的两个步骤:



(图 12.4.8)



(图 12.4.9)

由于左侧和中间的石柱是等价的（都不是盘子的最终目标），所以我们可以认为，解决上图 12.4.9 的问题等价于解决图 12.4.7 的问题。此时由于在右侧石柱上是最大的盘子，所以可以认为它不存在，其实这已经是一个  $n-1$  个盘子的汉诺塔问题了。接下来应该把  $n-2$  个上面的小盘子借助右侧石柱移动到左侧石柱上，然后把剩下的次大盘移动到右侧石柱，这就便成了一个  $n-3$  个盘子的汉诺塔问题了……如此做下去，直到只剩下一个盘子。

这样，我们可以设计出一个基本的解决汉诺塔问题的递归思路了：

(1) 设左中右石柱名分别为 A、B、C，设每次移动的时候有源石柱、过度石柱和目标石柱分别为 a、b、c，注意 ABC 石柱和 abc 石柱并没有确定的关系。

(2) 当  $n=1$  的时候，直接把 a 石柱上的盘子移动到 c 石柱上。（即递归出口）

(3) 当  $n \neq 1$  的时候，把 a 石柱上的  $n-1$  个盘子通过 c 石柱移动到 b 石柱上（相当于解决  $n-1$  个盘子的汉诺塔问题），把 a 石柱的最后一个盘子移动到 c 石柱上，然后再将 b 石柱上的  $n-1$  个盘子通过 a 石柱移动到 c 石柱上（相当于解决  $n-1$  个盘子的汉诺塔问题）。

于是，整个程序代码如下：（程序 12.4.2）

```
#include <iostream>
using namespace std;
void Hanoi(int n,char a,char b,char c);
void move(char sour,char dest);
int main()
{
 int n;
 cout <<"请输入汉诺塔盘子的个数： ";
 cin >>n;
 Hanoi(n,'A','B','C');
 cout <<"完成！ " <<endl;
 return 0;
}
void Hanoi(int n,char a,char b,char c)
{
 if (n==1)
 {
 move(a,c);
 }
 else
```



```

 {
 Hanoi(n-1,a,c,b);
 move(a,c);
 Hanoi(n-1,b,a,c);
 }
}
void move(char sour,char dest)
{
 cout <<"把" << sour <<"石柱最上面的盘子移动到" << dest <<"石柱上。" << endl;
}

```

运行结果:

请输入汉诺塔盘子的个数: 4

把 A 石柱最上面的盘子移动到 B 石柱上。

把 A 石柱最上面的盘子移动到 C 石柱上。

把 B 石柱最上面的盘子移动到 C 石柱上。

把 A 石柱最上面的盘子移动到 B 石柱上。

把 C 石柱最上面的盘子移动到 A 石柱上。

把 C 石柱最上面的盘子移动到 B 石柱上。

把 A 石柱最上面的盘子移动到 B 石柱上。

把 A 石柱最上面的盘子移动到 C 石柱上。

把 B 石柱最上面的盘子移动到 C 石柱上。

把 B 石柱最上面的盘子移动到 A 石柱上。

把 C 石柱最上面的盘子移动到 A 石柱上。

把 B 石柱最上面的盘子移动到 C 石柱上。

把 A 石柱最上面的盘子移动到 B 石柱上。

把 A 石柱最上面的盘子移动到 C 石柱上。

把 B 石柱最上面的盘子移动到 C 石柱上。

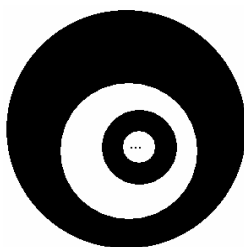
完成!

汉诺塔问题的运算量是随着盘子的增多而夸张地增长。你可以尝试一下输入汉诺塔盘子个数为 16, 却发现一台奔腾 4 级别的电脑在一两分钟内根本无法将其解决。据说如果用家用电脑解决 64 个盘子的汉诺塔问题, 需要花上几百年的时间。如果由人来移动盘子, 则花费的时间比地球的年龄——45 亿年还要多得多!

## 习题

根据要求编写程序。

①面积问题



如上图，有一个半径为  $r$  的黑圆，里面有一个半径为  $r/2$  的白圆，此白圆里又有一个半径为  $r/4$  的黑圆，此黑圆里又有一个半径为  $r/8$  的白圆，如此重复下去，问黑色部分的面积为多少？

运行结果示例：

请输入最外层圆半径  $r$ : 2

黑色部分的面积为  $4\pi$

### ② 阶梯问题

我们用一个正整数列来表示一段地方的高度，当一段地方的高度为一个逐一上升的序列时，我们称它为一个阶梯，例如 4、5、6、7、8 是一个长度为 5 的阶梯。现在给定一个正整数列，请找出它第一个最长的阶梯，并将其输出。如果一个阶梯也没有，输出 “No Ans”。

运行结果示例：

请输入数列的长度: 8

请输入数列: 0 2 3 4 3 4 6 5

结果为 2 3 4

### ③ 食堂问题

食堂里面排队买饭的人很多，某窗口每秒钟都会有一个人来排队买饭，若干秒后有一个人能够买好饭。我们将排队买饭的每个人用不同的字母表示，用 \$ 表示这秒钟有一个人买好了饭，用 # 表示饭卖完了，窗口关闭。假设同一秒钟内不可能有多个人同时买好饭，计时从第一个人到窗口开始。请将买好饭的人按时间序列输出，并告知是从开始计时的第几秒买好的。

运行结果序列：

请输入序列: ABC\$D\$E\$F\$G\$H#

结果：

A 3s

B 4s

C 6s

D 7s

### ④ 配色问题

现在有四种颜色：红  $r$  黄  $y$  绿  $l$  蓝  $b$ ，有一块矩形色板上可以放四种颜色中的任意两种，这两种颜色可以是相同的，比如： $rr$ （红红）、 $ry$ （红黄）。由于矩形色板是对称的，红绿和绿红是相同的配色方案。请用递归的方法将所有的配色方案输出。

运行结果示例：

$rr$   $yr$   $gr$   $br$   $yy$   $gy$   $by$   $gg$   $bg$   $bb$

# 后篇 面向对象的程序设计

## 第十三章 初识对象

C++曾被称为“带‘类’的 C 语言”。虽然这样的称法并不科学，但是不可否认，面向对象的程序设计是 C++的一个重要特征，也是 C++学习过程中的一个难点。本章先不对面向对象的概念作详细的讲述，而是以字符串和向量为例，让读者感性地了解什么是对象，什么是类，并且掌握如何使用类和对象。

### 13.1 对象就是物体

既然称为面向对象（Object Oriented，简称 OO），我们就先要知道什么是对象。其实单词 Object 更直观的翻译应该是物体。世界就是由各种物体组成的，比如某一辆汽车、某一个人、某一个杯子等等，这些都可以看作对象。

任何一个对象往往有一些具体的属性，比如某汽车的品牌、型号、排量，某人的性别、身高、体重，某杯子的口径，材质等等。任何一个对象往往能进行一些操作，比如汽车可以开动、拐弯，人可以走路、吃饭，杯子可以被打破等等。

所以，**对象就是任何我们可以想象出来的具体的物体。**

某些物体具有一些共性，我们可以将他们归类。比如A汽车和B汽车都是汽车，我和你都是人类，大杯子和小杯子都是杯子。**我们把这种能够抽象地描述某一些具有共性的物体的词称为类（Class）。**即汽车是一个类，人类是一个类，杯子也是一个类。

### 13.2 一个字符串也是对象

字符串（String）“abc”是我们可以想象出来的具体物体，它的长度为 3 个字符，我们可以在它的第 2 个字符查找到字母“b”。字符串“abcdefg”是我们可以想象出来的具体物体，它的长度为 7 个字符，我们可以在它的第 3 个字符查找到字母“cd”。由于各个字符串都具有一些属性，都能对其进行一些操作，所以，字符串是一个类。

下面我们先来看一段程序，了解如何使用字符串：（程序 13.2.1）

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
 string a("abc");//创建一个字符串 a，内容为“abc”
 cout <<"Pos 'b'=" <<a.find("b") <<endl;//在字符串 a 中查找子串“b”的位置（从 0 开始）
 cout <<"Length of a=" <<a.length() <<endl;//字符串 a 的长度
 cout <<a <<endl;//输出字符串 a
 string b("abcdefg");//创建一个字符串 b，内容为“abcdefg”
```

```

cout <<"Pos \"cd\"=\"" <<b.find("cd") <<endl; //在字符串 b 中查找子串“cd”的位置
cout <<"Length of b=\"" <<b.length() <<endl; //字符串 b 的长度
cout <<b <<endl; //输出字符串 b
return 0;
}

```

运行结果:

```

Pos 'b'=1
Length of a=3
abc
Pos "cd"=2
Length of b=7
abcdefg

```

程序中 `string` 是类名。`a` 和 `b` 是对象名，它们类似于变量名，也是一种标志符。而 `string a("abc");` 的写法是对象的初始化，在后面的章节我们会作详细介绍。

## 奇妙的点

程序 13.2.1 中出现了诸如 `a.length()` 和 `b.find("cd")` 之类的形式。在第九章，我们说过结构中有这种写法，即用成员操作符“.”来代表“的”，从而能够描述一个结构变量中的成员数据（某种属性）。

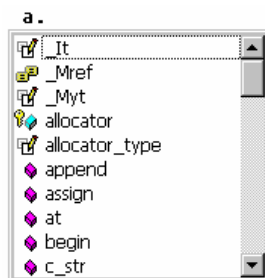
而在程序 13.2.1 中却是在成员操作符后面出现了函数的形式，我们把这种函数称为成员函数（Function Member），有时也称为操作或方法。成员函数就是对某个对象的操作。比如 `a.length()` 就是求字符串 `a` 的长度，`b.find("cd")` 就是在字符串 `b` 中查找子串“cd”的位置。

学过 VB 的朋友看过来

VB 也是面向对象的。它的每个控件都是一个对象。比如一个窗体有 `width`（宽度）属性，有 `height`（高度）属性，这些就是对象的成员数据。一个窗体可以 `hide`（隐藏），可以 `show`（显示），这些就是对象的成员函数。使用控件的经历可以让你更深刻地了解什么是对象。

## 对字符串的操作

对某个对象的操作并不是随心所欲的，而是事先设计好的。就像汽车不能飞，杯子不能吃一样，每个对象所能进行的操作总是和它所属的“类”相关的。那么字符串除了能够求长度和查找子串位置外还能进行一些别的操作么？当我们在输入程序 13.2.1 的时候就已经发现了一个“秘密”，如下图所示。



(图 13.2.1)

我们输入了对象名和成员操作符之后，VC 会给出一个下拉式列表，里面罗列了所有和该对象相关内容。字符串能进行什么操作，就看这个列表啦！下表就是其他一些常用的字符串操作：

| 成员函数名   | 操作内容              | 常用参数格式                    |
|---------|-------------------|---------------------------|
| append  | 在字符串末尾添加内容        | append(string 或字符串常量)     |
| compare | 与另一字符串作比较，相同则返回 0 | compare(string 或字符串常量)    |
| empty   | 判断字符串是否为空，空则返回 1  | empty()                   |
| insert  | 在某个位置插入字符串        | insert(int,string 或字符串常量) |
| swap    | 与另一个字符串内容交换       | swap(string)              |

下面我们通过程序来实践一下这些操作：（程序 13.2.2）

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
 string a("abc");//创建字符串 a
 string b("StringB");
 cout <<"Length of a=" <<a.length() <<endl;//此时 a 的长度为 3
 cout <<a <<endl;//字符串 a 的内容为 "abc"
 a.append("EFG");//在字符串末尾添加 "EFG"
 cout <<"Length of a=" <<a.length() <<endl;//此时字符串长度为 6
 cout <<a <<endl;//字符串 a 的内容为 "abcEFG"
 a.insert(3,b);//在字符串 a 的第三个字符后插入字符串 b
 cout <<a <<endl;//字符串 a 的内容为 "abcStringBEFG"
 cout <<a.compare("ABCDEFG") <<endl;//字符串 a 与 "ABCDEFG" 比较，不同应输出 1
 cout <<a.compare(a) <<endl;//字符串 a 与自己比较，相同应输出 0
 cout <<a.empty() <<endl;//字符串 a 不是空的，应输出 0
 a.swap(b);//字符串 a 和 b 内容交换
 cout <<"String a is " <<a <<endl <<"String b is " <<b <<endl;
 return 0;
}
```

运行结果：

Length of a=3

abc

Length of a=6

abcEFG

abcStringBEFG

1

0

0

String a is StringB

String b is abcStringBEFG

类似于标准库函数，我们不需要记住每种“类”的全部操作，只需要在使用过程中记住一些常用的操作就可以了。如果有需要的话，可以求助于相关书籍或网络。

试试看：

- 1、根据程序 13.2.1 的注释改写程序，要求创建一个字符串 `c`，其内容为“Hello World!”。输出字符串中子串“World”的位置，输出字符串的长度，最后输出字符串 `c`。
- 2、将程序 13.2.2 中的 `string a("abc");`改为 `string a;`，猜测运行结果并上机验证。

## 13.3 面向对象特点一：封装性

在使用字符串类的时候，我们发现它和字符数组一个很明显的不同就是，我们无法对数据进行直接的修改和操作。如果有一个 `char a[]="Hello";`，那么我们可以直接用 `a[0]='h';`来修改存储在内存中的字符，甚至我们可以输出数组的首地址来了解这个数组到底存放在什么位置。而对于一个 `string a("Hello");`，我们却无法直接修改它的数据，因为所有对 `a` 的操作都是由成员函数所定义的。我们只能了解这个字符串的存在，但它具体存储在于内存的什么位置，我们无法通过除了对应操作以外的简单方法得知。（如使用取地址操作符）

由于我们不是字符串类的设计者，当我们对 `string` 进行种种操作时，我们只能了解到它的操作结果，而对它的操作原理和操作实现过程却无法得知。

**我们把类的数据不可知性和操作实现过程不可知性称为类的封装性（Encapsulation）。**

不难理解，作为使用者，我们不需要对数据和操作实现过程感兴趣。就好像买一个手机，我们只关心它是否能够正常通话，正常发短消息，却对它如何接通电话，如何把信号发送出去等等不感兴趣。**类的封装性把类的设计者和类的使用者分隔开，使他们在设计程序时互不干扰，责任明确。**

## 13.4 从数组到向量

向量（Vector）是一个深奥的词。不过这里的向量不是数学里的向量，也不是物理里的向量。在 C++ 中的向量，就是一个存放数据的地方，类似于二维数组和链表。

### 向量的性能

在第九章末尾，我们介绍了数组存储和链表存储的优缺点。数组的缺点是分配空间不灵活；链表的缺点是无法通过下标快速找到结点。**然而这里介绍的向量却吸收了这两种数据结构各自的优点，综合性能较高。**

向量的分配空间是会随着数据的量而变化的，如果空间不够，那么向量的空间会自动增长。类似于数组，我们也可以通过下标来访问向量中的数据元素，增快找到数据的速度。

### 万用的模板

在编写链表程序的时候，我们一定有这样的困惑：链表里面存储的数据类型可能是各种各样的，难道我们要为各种数据类型都写一个链表程序么？我们能不能写一个万用的链表程序呢？

在 PowerPoint 之类的软件中，有一种模板功能。模板提供的文档框架是基本完整的，我们只需要在一些地方填写上自己需要的内容，就是一个完整的文档。在 C++ 中，也有这么一种模板（Template），我们只需要在使用之前填写自己需要的数据类型，就是一个完整的

程序。我们把具有模板功能的类称为模板类，向量就是一个模板类。在这一节，我们只需要了解如何使用向量这个模板类。关于更多模板的知识，将在后面的章节再作介绍。

在上一节中，我们不难看出创建一个对象的方法是：

**类名 对象名 (初始化数据);**

而创建一个模板类对象的方法是：

**类名<数据类型列表> 对象名 (初始化数据);**

即在类名之后填写数据类型，来创建一个符合自己需要的对象。

## 对向量的操作

同字符串一样，向量也有着属于自己的各种操作。下表就是向量常用的一些操作：

| 成员函数名     | 操作内容             | 常用参数格式             |
|-----------|------------------|--------------------|
| back      | 返回最后一个数据元素的值     | back()             |
| clear     | 清空向量中的数据         | clear()            |
| empty     | 判断向量是否为空，空则返回 1  | empty()            |
| front     | 返回第一个数据元素的值      | front()            |
| pop_back  | 删除最后一个数据元素       | pop_back()         |
| push_back | 将参数作为数据元素插入到向量末尾 | push_back(对应类型的数据) |
| size      | 返回向量中数据元素个数      | size()             |
| swap      | 与另一个向量作交换        | swap(vector)       |

由于涉及迭代器 (Iterator) 的知识，我们无法学习向量的插入数据和删除数据操作。有兴趣的读者可以去看一下《C++ Primer》的相关章节。

下面我们用向量来解决习题 9.6.2，模拟一下栈操作：(程序 13.4)

```
#include <vector>
#include <iostream>
using namespace std;
int main()
{
 vector<char> stack(0); //新建一个名为 stack 的存放字符数据的向量，初始元素个数为 0
 char temp;
 cout <<"请输入指令： " <<endl;
 do
 {
 cin >>temp;
 if (temp!='#')
 {
 if (temp!='$')
 {
 stack.push_back(temp); //模拟压栈操作
 }
 else
 {
 stack.pop_back(); //模拟退栈操作
 }
 }
 }
}
```

```

 }
}while (temp!='#');
for (int i=0;i<stack.size();i++)
 cout <<stack[i];//可以用下标访问数据元素
cout <<endl;
return 0;
}

```

运行结果:

请输入指令:

```
ABC$DEFG$$SHIJ$KLMS#
```

```
ABDHIKL
```

不难发现,用现成的向量来实现模拟栈的功能非常方便。我们不需要研究压栈和退栈的详细实现方法,而只需要知道何时操作就行了。

读到这里,可能你还没有完全明白到底什么是类,什么是对象,甚至搞不清创建对象的时候,对象名旁边的括号里面应该填什么。没关系,这些都不是本章所要掌握的内容。你只要会照猫画虎地使用字符串和向量就可以了。

## 习题

1、根据要求写语句。

①创建一个字符串对象,对象名为 str,内容为 Hello,World。

②创建一个存放整型数据的向量对象,对象名为 space,初始元素个数为 10。

2、根据要求编写程序。

①输入字符串 a 和字符串 b,输出它们连接后的字符串,并计算连接后的字符串长度。

运行示例:

请输入字符串 a: Tomato

请输入字符串 b: Studio

连接后的字符串为: TomatoStudio

连接后的字符串长度为: 12

②Peter 在使用他的电脑,有一件事情让他很头痛:他的电脑总是莫名其妙地重新启动,如果没有保存,他打的那些文字就会丢失了。有时候他还会打错字,就要使用撤销功能来删除他之前输入的一个字符。在电脑重新启动后,无法撤销之前保存的内容。我们用英文字母来表示他输入的文字,用\$表示一次撤销,用%表示一次保存,用\*表示一次电脑重新启动,用#表示输入结束并保存。请输出他最终保存下来的所有文字。如果没有任何文字被保存下来,则输出 Nothing。

运行示例:

请输入:

```
Tos$ma%toS*$S$toS%tud*tuD$dio#
```

```
TomatoStudio
```



## 第十四章 再识对象

在上一章我们从使用的角度了解了什么是类和对象，并且学会了如何使用成员函数。本章我们将从设计的角度来研究类和对象，看看类的内部究竟是怎么样的。

### 14.1 类是一种数据类型

我们已经了解了数据类型和变量的关系。数据类型是各个变量的归类，而变量则是某个数据类型的具体表现，我们称变量为数据类型的一个实例（Instance）。各个变量都有他们的属性：内容和占用内存空间等等；各个变量都有他们的操作：算术运算或逻辑运算等等。从这个角度来看，类和对象的关系就如同数据类型和变量的关系。我们不妨将类看作一种自定义的数据类型，那么对象就是一种属于该类型的变量。

### 类与结构

在第九章我们学习了结构类型，知道它是一种由用户自己定义的数据类型。我们已经能够使用结构刻画一些现实生活中的东西，但却无法让它“动起来”。所有对它的操作都要依赖于为它编写的函数。

类与结构是相似的。它也是一种由用户自己定义的数据类型；它也可以通过成员数据来刻画一些现实生活中的东西。不同的是，对它的操作并不是通过普通的函数，而是通过类的成员函数来实现的。

下面我们先来看一下，如何定义一个类和它的成员数据：

```
class 类名
{
 数据类型 成员变量1;
 数据类型 成员变量2;

};
```

看来如果仅仅是定义成员数据，类和结构是非常相似的，唯一的不同就是把保留字 struct 换成了 class。在这里还是要提醒一下，定义完一个类之后务必要在最后加上一个分号。

### 类的声明与定义

如果类的定义和主函数在同一个源文件里，那么就会可能遇到这样的问题：在类定义之前，主函数使用了这个类。这将会导致错误的发生，因为主函数还没有意识到这个类的存在。所以这个时候我们必须在主函数之前声明这个类的存在，其作用类似于函数原型。如：

```
class A;//类的声明
int main();//主函数
{

}
class A//类的定义
```

```

{

}; //千万别忘了这个分号
 我们还可以将一个类定义在一个头文件中，然后在源文件中包含这个头文件。由于包含头文件的动作在主函数运行之前，所以不必在主函数之前声明这个类。比如：
//class.h
class A//类的定义
{

};
//main.cpp
#include "class.h"//要注意这里必须用双引号，而不能使用<>
int main()
{

}

```

## 14.2 公有和私有

在上一章，我们提到了类的封装性。那么我们是如何保证类内部的数据和操作不被外部访问或执行呢？这时候，我们就要说说什么是公有（Public）和私有（Private）了。

**所谓公有，就是外部可以访问的数据或执行的操作。**比如一个人的身高（数据）是可以较直接地获得的，一个人吃东西（操作）是可以受外部控制的。**私有就是外部不能直接访问的数据或执行的操作。**比如一个人的心跳次数（数据）和消化过程（操作），虽然他们都是客观存在，但我们却不能直接地获取心跳数据或控制消化过程。

如果一个类的所有数据和操作都是公有的，那么它将完全暴露在外，同结构一样没有安全性。如果一个类的所有数据和操作都是私有的，那么它将完全与外界隔绝，这样的类也没有存在的意义。

下面我们来看一下如何定义公有和私有的成员数据：

```

class Node//定义一个链表结点类
{
public:
 int idata;//数据能够被外部访问
 char cdata;//数据能够被外部访问
private:
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
};

```

按照上面的写法，如果我们有一个结点对象 lnode，那么 lnode.idata 和 lnode.cdata 都是可以被外界直接访问的，而 lnode.prior 和 lnode.next 则不能被外界直接访问。

需要注意的是，**如果我们在定义或声明时不说明该成员数据（或成员函数）是公有的还是私有的，则默认为私有的。**所以从习惯上我们总是把定义的成员数据和成员函数分为公有和私有两类，先定义公有再定义私有，方便阅读代码时能够区分。虽然在定义时可以有多个 public 或 private 保留字，但是我们不推荐那样的写法。

另外，以后我们还会遇到一个名为 `protected` 的保留，目前它和 `private` 的效果是一样的，即成员数据或成员函数不能被外界直接访问或调用。在以后的章节我们会了解到 `private` 和 `protected` 的区别。

## 14.3 成员函数

我们已经学会如何调用成员函数，那么成员函数又是如何声明和定义的呢？它和普通函数有着什么异同点呢？

普通函数在使用之前必须声明和定义，成员函数也是这样。不过成员函数是属于某一个类的，所以只能在类的内部声明，即在定义类的时候写明成员函数的函数原型，同时要注意此函数是公有的还是私有的。如果一个类的某个成员函数是私有的，那么它只能被这个类的其他成员函数调用。成员函数的函数原型和普通函数的函数原型在写法上是一样的。比如：

```
class Node//定义一个链表结点类
{
public:
 int readi();//通过该函数读取 idata
 char readc();//通过该函数读取 cdata
 bool seti(int i);//通过该函数修改 idata
 bool setc(char c);//通过该函数修改 cdata
 bool setp(Node *p);//通过该函数设置前驱结点
 bool setn(Node *n);//通过该函数设置后继结点
private:
 int idata;//存储数据保密
 char cdata; //存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
};
```

### 常成员函数

由于数据封装在类的内部，在处理一些问题的时候就需要小心翼翼，不能把成员数据破坏了。以前我们介绍使用 `const` 来保护变量（就变成了常量），或保护指针所指向的内容，那么在类中，我们也可以使用 `const` 这个保留字来保护成员数据不被成员函数改变。我们把这种成员函数称为常成员函数。它的写法就是在函数的参数表后面加上一个 `const`，比如：

```
int readi() const;//通过该函数读取 idata，但不能改变任何成员数据
char readc() const;//通过该函数读取 cdata，但不能改变任何成员数据
```

使用常成员函数，就保证了成员数据的安全，在此函数中任何修改成员数据的语句将被编译器拒之门外。

### 成员函数的重载

和普通函数类似，在一个类中也可以有成员函数重载。成员函数的重载在规则上和普通函数也并无差别，这里不再赘述。

最终，我们将链表结点类的定义修改如下：

```

class Node//定义一个链表结点类
{
public:
 int readi() const;//通过该函数读取 idata, 但不能改变任何成员数据
 char readc() const;//通过该函数读取 cdata, 但不能改变任何成员数据
 bool set(int i);//重载, 通过该函数修改 idata
 bool set(char c);//重载, 通过该函数修改 cdata
 bool setp(Node *p);//通过该函数设置前驱结点
 bool setn(Node *n);//通过该函数设置后继结点
private:
 int idata;//存储数据保密
 char cdata; //存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
};

```

## 成员函数的定义

成员函数与普通函数的不同之处, 在于成员函数是属于某一个类的, 而不能被随意地调用。那么, 我们在定义一个成员函数的时候如何来表达它是属于某一个类的呢? **这个时候我们就要用到::操作符, 它表示该函数是属于某一个类的, 称为域解析操作符。**因此在类定义结束后, 定义一个成员函数的格式如下:

```

返回值类型 类名::函数名(参数表)
{
 语句;

}

```

事实上, 成员函数也是可以在类的定义中定义的 (此时不需要域解析操作符), 但是从程序的运行效率、可读性、美观性考虑, 我们建议将成员函数的定义完全放在类定义的外面。于是, 链表结点类和其成员函数的定义如下:

```

//node.h
class Node//定义一个链表结点类
{
public:
 int readi() const;//通过该函数读取 idata, 但不能改变任何成员数据
 char readc() const;//通过该函数读取 cdata, 但不能改变任何成员数据
 bool set(int i);//重载, 通过该函数修改 idata
 bool set(char c);//重载, 通过该函数修改 cdata
 bool setp(Node *p);//通过该函数设置前驱结点
 bool setn(Node *n);//通过该函数设置后继结点
private:
 int idata;//存储数据保密
 char cdata;//存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
}

```

```

}; //类定义结束，分号切勿忘记
int Node::readi() const //成员函数 readi 的定义
{
 return idata;
}
char Node::readc() const
{
 return cdata;
}
bool Node::set(int i) //重载成员函数定义
{
 idata=i;
 return true;
}
bool Node::set(char c)
{
 cdata=c;
 return true;
}
bool Node::setp(Node *p)
{
 prior=p;
 return true;
}
bool Node::setn(Node *n)
{
 next=n;
 return true;
}

```

在上面这些成员函数定义中，我们可以看出成员数据（或成员函数）在成员函数中可以直接使用。平时我们使用一个对象的公有成员数据时，我们要写作“对象名.成员数据”，但是在成员函数中不需要也不能那样写。接下来，我们就能尝试一下使用我们编写的类了：（程序 14.3）

```

//main.cpp
#include <iostream>
#include "node.h" //包含我们编写好的链表结点类头文件，必须用双引号
using namespace std;
int main()
{
 Node a; //创建一个链表结点对象 a
 a.set(1); //设置 idata
 a.set('A'); //设置 cdata
 cout <<a.readi() <<endl;
 cout <<a.readc() <<endl;
}

```

```

 return 0;
}

```

运行结果：

```

1
A

```

注意这个程序有两个文件，一个是头文件 `node.h`，一个是源文件 `main.cpp`。如果你忘记了如何创建一个头文件，那么请看本书的 11.2 节。

## 14.4 对象、引用和指针

我们已经知道，对象就如同一个变量。因此一个对象，也可以有对应的引用和指针。

### 对象的引用

在第六章中，我们说引用就像是给变量起了一个别名，对这个引用的操作就和操作这个变量本身一样。这给我们在设计程序的时候带来了方便。对象也可以有引用，声明一个对象的引用方法是：

```
类名 &对象名 a=对象名 b;
```

此时，对对象 `a` 的访问和操作就如同对对象 `b` 的访问和操作一样，对象 `a` 只是对象 `b` 的一个别名。例如我们已经定义好了一个链表结点类，则有以下程序段：

```

Node b;//声明一个结点对象
Node &a=b;//声明一个引用
a.set(0);//效果与 b.set(0)相同
a.readi();//效果与 b.readi()相同

```

### 对象指针

为了完成一个链表的类定义，我们需要学习一下对象指针的使用方法。所谓对象指针，就是一个指向对象的指针。由于类和结构的相似性，对象指针和结构指针的使用也是相似的。我们也是使用箭头操作符 `->` 来访问该指针所指向的对象的成员数据或成员函数。例如我们已经定义好了一个链表结点类，则有以下程序段：

```

Node b;//声明一个结点对象
Node *a=&b;//声明一个对象指针
a->set(0);//效果与 b.set(0)相同
a->readi();//效果与 b.readi()相同

```

至此，我们已经为编写一个链表类做好了准备。

## 习题

指出下列程序的错误之处。

```

//date.h
class Date
{

```

```
int year;
int month;
int day;
bool IsValid();
bool IsLeapYear() const;
void NextDay() const;
void NextMonth();
void NextYear();
void set(int y,int m,int d);
}
bool IsValid()
{
 switch(month)
 {
 case 1:
 case 3:
 case 5:
 case 7:
 case 8:
 case 10:
 case 12:
 if(day>=1 && day<=31)
 {
 return true;
 }
 else
 {
 return false;
 }
 case 4:
 case 6:
 case 9:
 case 11:
 if(day>=1 && day<=30)
 {
 return true;
 }
 else
 {
 return false;
 }
 case 2:
 if(IsLeapYear())
 {
```

```
 if (day>=1 && day<=29)
 {
 return true;
 }
 else
 {
 return false;
 }
 }
 else
 {
 if (day>=1 && day<=28)
 {
 return true;
 }
 else
 {
 return false;
 }
 }
 default:
 return false;
 }
}
bool IsLeapYear()
{
 return ((year%4==0 && year%100!=0) || year%400==0);
}
void NextDay() const
{
 day++;
}
void NextMonth()
{
 month++;
}
void NextYear()
{
 year++;
}
void set(int y,int m,int d)
{
 year=y;
 month=m;
```



```
 day=d;
}
//main.cpp
#include <iostream>
#include <date.h>
using namespace std;
int main()
{
 date a;//创建一个对象
 date *b=a;//创建一个对象指针
 a.set(2004,11,15);
 cout <<b.IsValid();
 date &c=a;//创建一个引用
 cout <<c.IsLeapYear;
 return 0;
}
```

## 第十五章 造物者与毁灭者

上一章我们介绍了如何编写一个简单的类，了解了公有和私有的概念，掌握了如何编写一个成员函数。但是光光靠这些，还远不能让编写的“类”符合我们的需要。在本章，我们将介绍在创建一个对象和毁灭一个对象的时候，发生的一些事情。

### 15.1 麻烦的初始化

在声明一个局部变量的时候，我们必须对它进行初始化，否则它的数据是一个不确定的值。同样，在声明一个对象的时候，也应该对它进行初始化。不过一个对象可能有许许多多的成员数据，对对象的初始化就意味着对许许多多的成员数据进行初始化。变量的初始化只需要一句赋值语句就能完成，而对象的初始化可能要许许多多的赋值语句才能完成。因此，我们常常把这许许多多的语句写在一个函数中。比如我们为链表结点类编写了一个名为 `init` 的初始化函数：

```
class Node//定义一个链表结点类
{
public:

 void init(int i,char c);

private:
 int idata;//存储数据保密
 char cdata;//存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
};
void Node::init(int i,char c)
{
 idata=i;//初始化 idata
 cdata=c;//初始化 cdata
 prior=NULL;//初始化前驱结点指针
 next=NULL;//初始化后续结点指针
}
```

这下好了，我们创建一个链表结点对象之后只要运行一次 `init` 函数就能将其初始化了：

```
Node a;
a.init(0, '0');
```

既然 `init` 函数担负着初始化对象的重任，那么它就必须和对象的声明“出双入对”了。万一忘记对对象进行初始化，程序就可能会出错。这就像在病毒肆虐的今天，保证电脑安全的病毒防火墙必须在开机之后立刻运行一样。万一哪天开了电脑忘记运行病毒防火墙，那么后果可能很严重。

不过，你使用的病毒防火墙是你每次开机以后自己去点击运行的么？那样岂不是很麻

烦？你是否知道，我们使用的病毒防火墙往往是随系统一起启动的？

这给了我们一些启示：有的程序能够随着系统的启动而自动运行，那么会不会有一种函数，随着对象的创建而自动被调用呢？有！那就是构造函数（Constructor）。

## 15.2 造物者——构造函数

构造函数是一种随着对象创建而自动被调用的函数，它的主要用途是为对象作初始化。那么，构造函数到底是什么样子的呢？

### 构造函数的声明与定义

在 C++ 中，规定与类同名的成员函数就是构造函数。需要注意的是，构造函数应该是一个公有的成员函数，并且构造函数没有返回值类型。以下是我们为链表结点类编写的一个构造函数：（其他成员函数定义见 14.3 节）

```
//node.h
#include <iostream> //如果不包含 iostream 头文件，这个文件里就不能用 cout
using namespace std;
class Node //定义一个链表结点类
{
public:

 Node(); //构造函数的声明，构造函数是公有的成员函数，没有返回值类型

private:
 int idata; //存储数据保密
 char cdata; //存储数据保密
 Node *prior; //前驱结点的存储位置保密
 Node *next; //后继结点的存储位置保密
};
Node::Node() //构造函数的定义
{
 cout << "Node constructor is running..." << endl; //提示构造函数运行
 idata=0; //初始化 idata
 cdata='0'; //初始化 cdata
 prior=NULL; //初始化前驱结点指针
 next=NULL; //初始化后续结点指针
}
```

这时，我们创建一个链表结点对象，构造函数随着对象创建而自动被调用，所以这个对象创建之后 idata 的值为 0，cdata 的值为 '0'，prior 和 next 的值都是 NULL：（程序 15.2.1）

```
//main.cpp
#include <iostream>
#include "node.h"
using namespace std;
int main()
```

```

{
 Node a;//创建一个链表结点对象 a，调用构造函数
 cout <<a.readi() <<endl;
 cout <<a.readc() <<endl;
 return 0;
}

```

运行结果：

```

Node constructor is running...
0
0

```

可是，这样的构造函数还是不太理想。如果每次初始化的值都是固定的，那么有没有构造函数都是一样的。构造函数变成了一种摆设！我们该怎么办？

## 带参数的构造函数

函数的特征之一就是能够在调用时带上参数。既然构造函数也是函数，那么我们就给构造函数带上参数，使用重载或默认参数等方法，从而实现更自由地对对象进行初始化操作。以下便是对链表结点类的进一步修改：（程序 15.2.2）

```

//node.h
#include <iostream>
using namespace std;
class Node//定义一个链表结点类
{
public:
 Node();//构造函数 0
 Node(int i,char c='0');//构造函数重载 1，参数 c 默认为'0'
 Node(int i,char c,Node *p,Node *n);//构造函数重载 2
 int readi() const;//读取 idata
 char readc() const;//读取 cdata
 Node * readp() const;//读取上一个结点的位置
 Node * readn() const;//读取下一个结点的位置
 bool set(int i);//重载，通过该函数修改 idata
 bool set(char c);//重载，通过该函数修改 cdata
 bool setp(Node *p);//通过该函数设置前驱结点
 bool setn(Node *n);//通过该函数设置后继结点
private:
 int idata;//存储数据保密
 char cdata;//存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
};
int Node::readi() const//成员函数 readi 的定义
{
 return idata;
}

```

```
char Node::readc() const
{
 return cdata;
}
Node * Node::readp() const
{
 return prior;
}
Node * Node::readn() const
{
 return next;
}
bool Node::set(int i)//重载成员函数定义
{
 idata=i;
 return true;
}
bool Node::set(char c)
{
 cdata=c;
 return true;
}
bool Node::setp(Node *p)
{
 prior=p;
 return true;
}
bool Node::setn(Node *n)
{
 next=n;
 return true;
}
Node::Node()//构造函数 0 的定义
{
 cout <<"Node constructor is running..." <<endl;//提示构造函数运行
 idata=0;//初始化 idata
 cdata='0'//初始化 cdata
 prior=NULL;//初始化前驱结点指针
 next=NULL;//初始化后续结点指针
}
Node::Node(int i,char c)//构造函数重载 1, 默认参数只需要在函数原型中出现
{
 cout <<"Node constructor is running..." <<endl;
 idata=i;
```

```

 cdata=c;
 prior=NULL;
 next=NULL;
}
Node::Node(int i,char c,Node *p,Node *n)//构造函数重载 2
{
 cout <<"Node constructor is running..." <<endl;
 idata=i;
 cdata=c;
 prior=p;
 next=n;
}
//main.cpp
#include <iostream>
#include "node.h"
using namespace std;
int main()
{
 Node a;//创建一个链表结点对象 a，调用构造函数 0
 Node b(8);//创建一个链表结点对象 b，调用构造函数重载 1，参数 c 默认为'0'
 Node c(8,'F',NULL,NULL);//创建一个链表结点对象 c，调用构造函数重载 2
 cout <<a.readi() <<' ' <<a.readc() <<endl;
 cout <<b.readi() <<' ' <<b.readc() <<endl;
 cout <<c.readi() <<' ' <<c.readc() <<endl;
 return 0;
}

```

运行结果：

```

Node constructor is running...
Node constructor is running...
Node constructor is running...
0 0
8 0
8 F

```

我们看到，在参数和重载的帮助下，我们可以设计出适合各种场合的构造函数。初始化各个对象的成员数据对我们来说已经是小菜一碟了。但是，这时你是否会回想起当初没有编写构造函数时的情形？如果没有编写构造函数，对象的创建是一个怎样的过程呢？

在 C++ 中，每个类都有且必须有构造函数。如果用户没有自行编写构造函数，则 C++ 自动提供一个无参数的构造函数，称为默认构造函数。这个默认构造函数不做任何初始化工作。一旦用户编写了构造函数，则这个无参数的默认构造函数就消失了。如果用户还希望能有一个无参数的构造函数，必须自行编写。

## 15.3 先有鸡还是先有蛋？

链表结点类编写好了，我们可以向链表类进军了。链表是由一个个链表结点组成的，所

以我们会在链表类中使用到链表结点类。链表结点类是一个很简单的类，链表类是一个功能更为强大的类。正是将一个个类不断地组合与扩充，使得面向对象的程序功能越来越强大。

让我们感兴趣的是，假设我们编写的链表需要有一个头结点作为成员数据，那么是先有链表呢，还是先有头结点？我们又该如何在给链表作初始化的同时初始化头结点呢？

当一个对象中包含别的对象时，我们可以在它的构造函数定义中用以下格式调用其成员对象的构造函数：

**类名::构造函数名(参数表):成员对象名<sub>1</sub>(参数表)[,……成员对象名<sub>n</sub>(参数表)]**

前一段和普通的构造函数一样，冒号之后则表示该类中的成员对象怎样调用各自的构造函数。

下面我们来看一个简单的面向对象的链表程序：（程序 15.3）

```
//node.h 同程序 15.2.2
//linklist.h
#include "node.h"//需要使用链表结点类
#include <iostream>
using namespace std;
class Linklist
{
public:
 Linklist(int i,char c);//链表类构造函数
 bool Locate(int i);//根据整数查找结点
 bool Locate(char c);//根据字符查找结点
 bool Insert(int i=0,char c='0');//在当前结点之后插入结点
 bool Delete();//删除当前结点
 void Show();//显示链表所有数据
 void Destroy();//清除整个链表
private:
 Node head;//头结点
 Node * pcurrent;//当前结点指针
};
Linklist::Linklist(int i,char c):head(i,c)//类名::构造函数名(参数表):成员对象名1(参数表), 链表类构造函数, 调用head对象的构造函数重载 1, 详见Node.h文件
{
 cout<<"Linklist constructor is running..."<<endl;
 pcurrent=&head;
}
bool Linklist::Locate(int i)
{
 Node * ptemp=&head;
 while(ptemp!=NULL)
 {
 if(ptemp->readi()==i)
 {
 pcurrent=ptemp;//将当前结点指针指向找到的结点
 return true;
 }
 }
}
```

```
 }
 ptemp=ptemp->readn();//查找下一个结点
}
return false;
}
bool Linklist::Locate(char c)
{
 Node * ptemp=&head;
 while(ptemp!=NULL)
 {
 if(ptemp->readc()==c)
 {
 pcurrent=ptemp;
 return true;
 }
 ptemp=ptemp->readn();
 }
 return false;
}
bool Linklist::Insert(int i,char c)
{
 if(pcurrent!=NULL)
 {
 Node * temp=new Node(i,c,pcurrent,pcurrent->readn());//调用 Node 类构造函数重载 2
 if (pcurrent->readn()!=NULL)
 {
 pcurrent->readn()->setp(temp);
 }
 pcurrent->setn(temp);
 return true;
 }
 else
 {
 return false;
 }
}
bool Linklist::Delete()
{
 if(pcurrent!=NULL && pcurrent!=&head)//head 结点不能删除
 {
 Node * temp=pcurrent;
 if (temp->readn()!=NULL)
 {
 temp->readn()->setp(pcurrent->readp());
 }
 }
}
```



```
 }
 temp->readp()->setn(pcurrent->readn());//先连
 pcurrent=temp->readp();
 delete temp;//后断
 return true;
}
else
{
 return false;
}
}
void Linklist::Show()
{
 Node * ptemp=&head;
 while (ptemp!=NULL)//链表的遍历
 {
 cout <<ptemp->readi() <<'\t' <<ptemp->readc() <<endl;
 ptemp=ptemp->readn();
 }
}
void Linklist::Destroy()
{
 Node * ptemp1=head.readn();
 while (ptemp1!=NULL)
 {
 Node * ptemp2=ptemp1->readn();
 delete ptemp1;
 ptemp1=ptemp2;
 }
 head.setn(NULL);//头结点之后没有其他结点
}
//main.cpp
#include "Linklist.h"
#include <iostream>
using namespace std;
int main()
{
 int tempi;
 char tempc;
 cout <<"请输入一个整数和一个字符: " <<endl;
 cin >>tempi >>tempc;
 Linklist a(tempi,tempc);//创建一个链表, 头结点数据由 tempi 和 tempc 确定
 a.Locate(tempi);
 a.Insert(1,'C');
```

```

a.Insert(2,'B');
a.Insert(3,'F');
cout <<"After Insert" <<endl;
a.Show();
a.Locate('B');
a.Delete();
cout <<"After Delete" <<endl;
a.Show();
a.Destroy();
cout <<"After Destroy" <<endl;
a.Show();
return 0;
}

```

运行结果：

请输入一个整数和一个字符：

4 G

Node constructor is running...

Linklist constructor is running...

Node constructor is running...

Node constructor is running...

Node constructor is running...

After Insert

4 G

3 F

2 B

1 C

After Delete

4 G

3 F

1 C

After Destroy

4 G

根据程序的运行结果，我们发现头结点的构造函数比链表的构造函数优先运行。这也不难理解：构造函数的目的是要初始化成员数据，初始化成员数据的时候这个成员数据是必须存在的。所以当成员数据是一个对象的时候，应当先产生这个成员对象，于是就先调用了成员对象的构造函数。

试试看：

如果有多个成员数据都是对象，那么对象的生成顺序又是怎样的呢？

结论：如果有多个成员数据都是对象，则它们按声明时的顺序依次产生。

## 15.4 克隆技术

我们在程序中常常需要把一些数据复制一份出来备作它用。对于只有基本类型变量的程序来说，这是轻而易举就能做到的——新建一个临时变量，用一句赋值语句就能完成。但如果它是一个有着许许多多成员数据的对象，这就会非常麻烦。最要命的是，那些成员数据还是私有的，根本无法直接访问或修改。那么这时候，我们怎么“克隆”出一个和原来的对象相同的新对象呢？

### 拷贝构造函数

我们知道，构造函数是可以带上参数的，这些参数可以是整型、字符型等等，那么它可不可以是一个对象类型呢？我们把一个原来的对象丢给构造函数，然后让它给我们造一个相同的对象，是否可以呢？下面我们就来试试看：（程序 15.4.1）

```
//node.h
#include <iostream>
using namespace std;
class Node//定义一个链表结点类
{
public:
 Node();//构造函数的声明
 Node(int i,char c='0');//构造函数重载 1
 Node(int i,char c,Node *p,Node *n);//构造函数重载 2
 Node(Node &n);//结点拷贝构造函数，&表示引用
 int readi() const;//读取 idata
 char readc() const;//读取 cdata
 Node * readp() const;//读取上一个结点的位置
 Node * readn() const;//读取下一个结点的位置
 bool set(int i);//重载，通过该函数修改 idata
 bool set(char c);//重载，通过该函数修改 cdata
 bool setp(Node *p);//通过该函数设置前驱结点
 bool setn(Node *n);//通过该函数设置后继结点
private:
 int idata;//存储数据保密
 char cdata;//存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
};
//未定义的函数与程序 15.2.2 相同
Node::Node(Node &n)
{
 idata=n.idata;//可以读出同类对象的私有成员数据
 cdata=n.cdata;
 prior=n.prior;
 next=n.next;
```

```

}
//linklist.h
#include "node.h"//需要使用链表结点类
#include <iostream>
using namespace std;
class Linklist
{
public:
 Linklist(int i,char c);//链表类构造函数
 Linklist(Linklist &l);//链表拷贝构造函数，&表示引用
 bool Locate(int i);//根据整数查找结点
 bool Locate(char c);//根据字符查找结点
 bool Insert(int i=0,char c='0');//在当前结点之后插入结点
 bool Delete();//删除当前结点
 void Show();//显示链表所有数据
 void Destroy();//清除整个链表
private:
 Node head;//头结点
 Node * pcurrent;//当前结点指针
};
//未定义的函数与程序 15.3 相同
Linklist::Linklist(Linklist &l):head(l.head)//调用结点的拷贝构造函数来初始化 head
{
 cout<<"Linklist cloner running..." <<endl;
 pcurrent=l.pcurrent;//指针数据可以直接赋值
}
//main.cpp
#include "Linklist.h"
#include <iostream>
using namespace std;
int main()
{
 int tempi;
 char tempc;
 cout <<"请输入一个整数和一个字符： " <<endl;
 cin >>tempi >>tempc;
 Linklist a(tempi,tempc);
 a.Locate(tempi);
 a.Insert(1,'C');
 a.Insert(2,'B');
 a.Insert(3,'F');
 cout <<"After Insert" <<endl;
 a.Show();
 a.Locate('B');
}

```

```

a.Delete();
cout <<"After Delete" <<endl;
a.Show();
Linklist b(a);//创建一个链表 b，并且将链表 a 复制到链表 b
cout <<"This is Linklist b" <<endl;
b.Show();//显示 b 链表中的内容
a.Destroy();
cout <<"After Destroy" <<endl;
a.Show();
return 0;
}

```

运行结果：

请输入一个整数和一个字符：

4 G

Node constructor is running...

Linklist constructor is running...

Node constructor is running...

Node constructor is running...

Node constructor is running...

After Insert

4 G

3 F

2 B

1 C

After Delete

4 G

3 F

1 C

Linklist cloner running...

This is Linklist b

4 G

3 F

1 C

After Destroy

4 G

根据程序运行的结果，我们发现输出链表 b 的内容的确和链表 a 一样了，并且我们可以得到三个结论：

- (1) 拷贝构造函数可以读出相同类对象的私有成员数据；
- (2) 拷贝构造函数的实质是把参数的成员数据一一复制到新的对象中；
- (3) 拷贝构造函数也是构造函数的一种重载。

## 默认拷贝构造函数

我们已经知道构造函数有默认构造函数，其实拷贝构造函数也有默认的拷贝构造函数。所谓默认拷贝构造函数是指，用户没有自己定义拷贝构造函数时，系统自动给出的一个拷贝

构造函数。默认拷贝构造函数的功能是将对象的成员数据一一赋值给新创建的对象成员数据，如果某些成员数据本身就是对象，则自动调用它们的拷贝构造函数或默认拷贝构造函数。

试试看：

用程序 15.4.1 的 main.cpp 文件中的代码替换程序 15.3 的 main.cpp 中的代码，并察看运行结果。

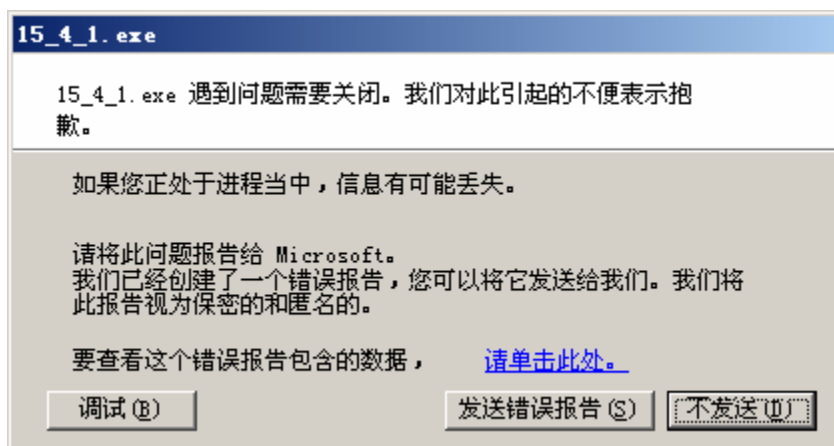
结论：默认拷贝构造函数能够满足我们大多数的需要。

## 拷贝构造函数存在的意义

既然上面说到，默认拷贝构造函数已经能够满足我们大多数的需要，那么自定义的拷贝构造函数是否就可以不用存在了呢？我们修改一下程序 15.4.1 的 main.cpp 文件，看看拷贝构造函数到底有什么意义：

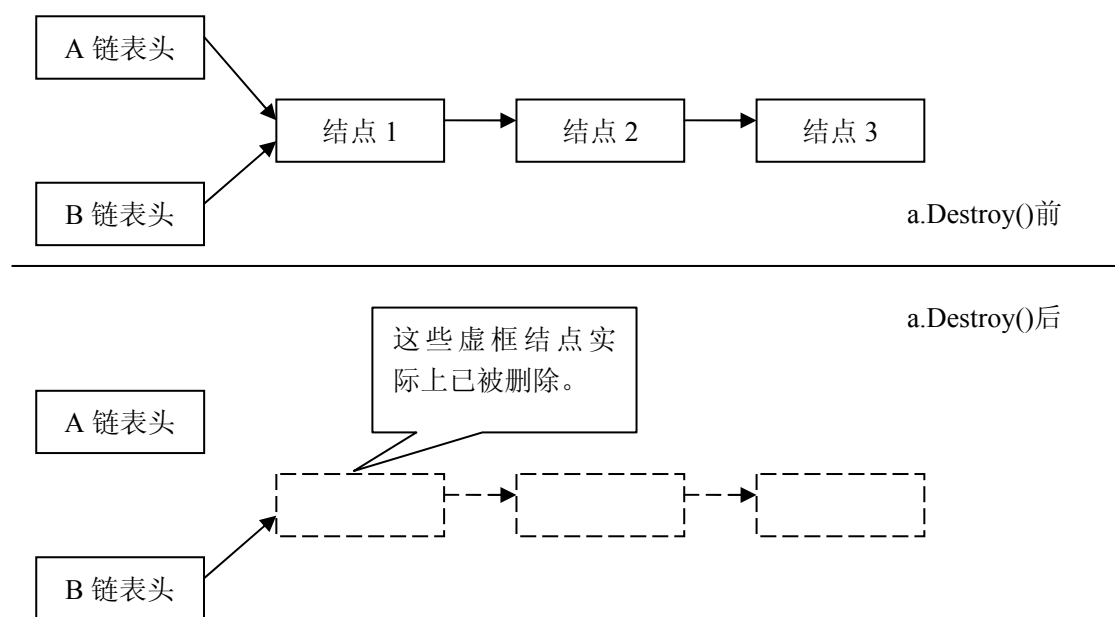
```
#include "Linklist.h"
#include <iostream>
using namespace std;
int main()
{
 int tempi;
 char tempc;
 cout <<"请输入一个整数和一个字符： " <<endl;
 cin >>tempi >>tempc;
 Linklist a(tempi,tempc);
 a.Locate(tempi);
 a.Insert(1,'C');
 a.Insert(2,'B');
 a.Insert(3,'F');
 cout <<"After Insert" <<endl;
 a.Show();
 a.Locate('B');
 a.Delete();
 cout <<"After Delete" <<endl;
 a.Show();
 Linklist b(a);
 cout <<"This is Linklist B" <<endl;
 b.Show();
 a.Destroy();
 cout <<"After Destroy" <<endl;
 a.Show();
 cout <<"This is Linklist b" <<endl;
 b.Show();//关键是在这里加了一条语句，让它显示 b 链表中的内容
 return 0;
}
```

运行结果:



为什么显示链表 b 的内容，却导致了严重的错误呢？

这时我们就要来研究一下这个链表的结构了。在这个链表中，成员数据只有头结点 head 和当前指针 pcurrent，所有的结点都是通过 new 语句动态生成的。而程序 15.4.1 中的拷贝构造函数仅仅是简单地将头结点 head 和当前指针 pcurrent 复制了出来，所以一旦运行 a.Destroy() 之后，链表 a 头结点之后的结点已经全部都删除了，而链表 b 的头结点还傻傻地指向原来 a 链表的结点。如果这时再访问链表 b，肯定就要出问题了。如下所示：



(图 15.4)

程序 15.4.1 中的拷贝构造函数仅仅是把成员数据拷贝了过来，却没有把动态申请的资源拷贝过来，我们把这种拷贝称为浅拷贝。相对地，如果拷贝构造函数不仅把成员数据拷贝过来，连动态申请的资源也拷贝过来，我们则称之为深拷贝。

下面我们来看如何实现深拷贝：(程序 15.4.2)

```
//node.h 同程序 15.4.1
//linklist.h
#include "node.h"//需要使用链表结点类
#include <iostream>
using namespace std;
```

```

class Linklist
{
public:
 Linklist(int i,char c);//链表类构造函数
 Linklist(Linklist &l);//链表深拷贝构造函数
 bool Locate(int i);//根据整数查找结点
 bool Locate(char c);//根据字符查找结点
 bool Insert(int i=0,char c='0');//在当前结点之后插入结点
 bool Delete();//删除当前结点
 void Show();//显示链表所有数据
 void Destroy();//清除整个链表
private:
 Node head;//头结点
 Node * pcurrent;//当前结点指针
};
//未定义的函数与程序 15.3 相同
Linklist::Linklist(Linklist &l):head(l.head)
{
 cout<<"Linklist Deep cloner running..." <<endl;
 pcurrent=&head;
 Node * ptemp1=l.head.readn();//该指针用于指向原链表中被复制的结点
 while(ptemp1!=NULL)
 {
 Node * ptemp2=new Node(ptemp1->readi(),ptemp1->readc(),pcurrent,NULL);//新建结
 点，并复制 idata 和 cdata，思考为何这里不能直接用 Node 的拷贝构造函数？
 pcurrent->setn(ptemp2);
 pcurrent=pcurrent->readn();//指向表尾结点
 ptemp1=ptemp1->readn();//指向下一个被复制结点
 }
}
//main.cpp
#include "Linklist.h"
#include <iostream>
using namespace std;
int main()
{
 int tempi;
 char tempc;
 cout <<"请输入一个整数和一个字符： " <<endl;
 cin >>tempi >>tempc;
 Linklist a(tempi,tempc);
 a.Locate(tempi);
 a.Insert(1,'C');
 a.Insert(2,'B');
}

```



```

a.Insert(3,'F');
cout <<"After Insert" <<endl;
a.Show();
a.Locate('B');
a.Delete();
cout <<"After Delete" <<endl;
a.Show();
Linklist b(a);//创建一个链表 b, 并且将链表 a 复制到链表 b
cout <<"This is Linklist b" <<endl;
b.Show();
a.Destroy();
cout <<"After Destroy" <<endl;
a.Show();
cout <<"This is Linklist b" <<endl;
b.Show();//链表 a 被 Destroy 之后察看链表 b 的内容
return 0;
}

```

运行结果:

请输入一个整数和一个字符:

4 G

Node constructor is running...

Linklist constructor is running...

Node constructor is running...

Node constructor is running...

Node constructor is running...

After Insert

4 G

3 F

2 B

1 C

After Delete

4 G

3 F

1 C

Linklist Deep cloner running...

Node constructor is running...

Node constructor is running...

This is Linklist b

4 G

3 F

1 C

After Destroy

4 G

This is Linklist b

```

4 G
3 F
1 C

```

我们看到，现在即使运行 `a.Destroy()` 之后，链表 `b` 里面的数据仍然能够正常显示。这是因为深拷贝构造函数是真正意义上的复制了链表 `a`，并且使得链表 `a` 和链表 `b` 各自独立，互不干扰。这才是自定义拷贝构造函数存在的重要意义。

## 15.5 毁灭者——析构函数

在学习链表的时候，我们知道结点是动态生成的，如果在程序结束之前不释放内存，就会造成内存泄漏。虽然我们已经编写了成员函数 `Destroy` 来删除所有动态生成的结点，但是如果不知道这个链表对象何时不再使用，那么调用 `Destroy` 的时机对我们来说就是个麻烦了。如果过早地调用，则后面的程序可能会出错。既然有构造函数能随着对象的创建而自动被调用，那么有没有一种函数能随着对象的消亡而自动被调用呢？有！那就是析构函数（`Destructor`）。

析构函数是一种随着对象消亡而自动被调用的函数，它的主要用途是释放动态申请的资源。它没有返回类型，没有参数，也没有重载。析构函数的函数名也是指定的，是在构造函数名之前加一个“~”符号。

下面我们为程序 15.4.2 添上析构函数的功能：（程序 15.5）

```

//node.h
#include <iostream>
using namespace std;
class Node//定义一个链表结点类
{
public:
 Node();//构造函数的声明
 Node(Node &n);//结点拷贝构造函数
 Node(int i,char c='0');//构造函数重载 1
 Node(int i,char c,Node *p,Node *n);//构造函数重载 2
 ~Node();//结点析构函数
 int readi() const;//读取 idata
 char readc() const;//读取 cdata
 Node * readp() const;//读取上一个结点的位置
 Node * readn() const;//读取下一个结点的位置
 bool set(int i);//重载，通过该函数修改 idata
 bool set(char c);//重载，通过该函数修改 cdata
 bool setp(Node *p);//通过该函数设置前驱结点
 bool setn(Node *n);//通过该函数设置后继结点
private:
 int idata;//存储数据保密
 char cdata;//存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
};

```

```

//未定义的函数与程序 15.4.1 相同
Node::~Node()
{
 cout <<"Node destructor is running..." <<endl;
}
//linklist.h
#include "node.h"//需要使用链表结点类
#include <iostream>
using namespace std;
class Linklist
{
public:
 Linklist(int i,char c);//链表类构造函数
 Linklist(Linklist &l);//链表深拷贝构造函数
 ~Linklist();//链表析构函数
 bool Locate(int i);//根据整数查找结点
 bool Locate(char c);//根据字符查找结点
 bool Insert(int i=0,char c='0');//在当前结点之后插入结点
 bool Delete();//删除当前结点
 void Show();//显示链表所有数据
 void Destroy();//清除整个链表
private:
 Node head;//头结点
 Node * pcurrent;//当前结点指针
};
//未定义的函数与程序 15.4.2 相同
Linklist::~Linklist()
{
 cout<<"Linklist destructor is running..."<<endl;
 Destroy();//一个成员函数调用另一个成员函数不需要带上对象名
}

```

//main.cpp 同程序 15.4.2

运行结果:

请输入一个整数和一个字符:

4 G

Node constructor is running...

Linklist constructor is running...

Node constructor is running...

Node constructor is running...

Node constructor is running...

After Insert

4 G

3 F

2 B

```

1 C
Node destructor is running...
After Delete
4 G
3 F
1 C
Linklist Deep cloner running...
Node constructor is running...
Node constructor is running...
This is Linklist b
4 G
3 F
1 C
Node destructor is running...
Node destructor is running...
After Destroy
4 G
This is Linklist b
4 G
3 F
1 C
Linklist destructor is running...
Node destructor is running...
Node destructor is running...
Node destructor is running...
Node destructor is running...
Node destructor is running...
Node destructor is running...

```

在 After Destroy 之前的两条 Node destructor 运行是因为调用了 a.Destroy(), 最后的 6 条 destructor 是因为程序运行结束使得对象自动消亡。可见析构函数是在使用 **delete** 语句删除动态生成的对象或程序结束对象消亡时自动被调用的。

从最后的 2 条 destructor 输出我们发现, 当一个对象的成员数据还是对象时, 析构函数的运行顺序恰好与构造函数的运行顺序相反: 一个大对象先调用析构函数, 瓦解成若干成员数据, 然后各个成员数据再调用各自的析构函数。这体现出构造函数与析构函数的对称性。

## 习题

1、指出下列程序的错误之处。

```

//student.h
#include <iostream>
#include <string>
using namespace std;
class student
{
public:

```

```

 void student(string name,float g=0.0);
 ~student();
 ~student(int i);
private:
 string sname;
 float gpa;
};
void student::student(string name,float g=0.0)
{
 cout <<"student constructor is running..." <<endl;
 sname.append(name);
 gpa=g;
}
student::~~student()
{
 cout <<"student destructor is running..." <<endl;
}
student::~~student(int i)
{
 cout <<"student No. " <<i<<"destructor is running..." <<endl;
}
//main.cpp
#include <iostream>
#include "student.h"
using namespace std;
int main()
{
 student s1("Venus",3.5);
 student s2("Jon");
 student s3(s2);
 cout <<"Student Name:" <<s1.sname <<endl;
 cout <<"Student GPA:" <<s1.gpa <<endl;
 cout <<"Student Name:" <<s2.sname <<endl;
 cout <<"Student GPA:" <<s2.gpa <<endl;
 cout <<"Student Name:" <<s3.sname <<endl;
 cout <<"Student GPA:" <<s3.gpa <<endl;
 return 0;
}

```

2、阅读以下代码，写出运行结果。

```

//class.h
#include <iostream>
using namespace std;
class c1
{

```

```
public:
 c1();
 c1(int i);
 ~c1();
private:
 int d;
};
c1::c1()
{
 cout <<"Original C1 constructor is running..." <<endl;
}
c1::c1(int i)
{
 cout <<"C1 constructor with parameter is running..." <<endl;
 d=i;
}
c1::~~c1()
{
 cout <<"C1 destructor is running..." <<endl;
}
class c2
{
public:
 c2();
 c2(c2 &t);
 ~c2();
private:
 c1 a;
 c1 b;
};
c2::c2():b(3),a()
{
 cout <<"C2 constructor is running..." <<endl;
}
c2::c2(c2 &t):b(t.b),a(t.a)
{
 cout <<"C2 cloner is running..." <<endl;
}
c2::~~c2()
{
 cout <<"C2 destructor is running..." <<endl;
}
//main.cpp
#include <iostream>
```

```
#include "class.h"
using namespace std;
int main()
{
 c2 a;
 c2 b(a);
 return 0;
}
```

3、编写一个字符串类，要求具有以下功能：

- (1) 以字符数组作为成员数据，用于存储字符串，字符串长度不会超过 49。
- (2) 构造函数可以以字符串常量作为参数，对对象进行初始化。
- (3) 构造函数可以以本类对象作为参数，实现对象的拷贝。
- (4) 具有替换字符的功能，如将字符串中的所有字母 r 替换成字母 R。
- (5) 具有求出字符串长度的功能，结尾符不计入长度。
- (6) 具有输出字符串的功能。（用成员函数实现）

## 第十六章 共有财产 · 好朋友 · 操作符

上一章我们学习了构造函数、拷贝构造函数和析构函数，了解了对象的生灭。链表类的各个功能也逐渐丰富起来。不过有些小问题还是不太好解决，本章我们就要来关心一下这些小问题，看看有没有更好的解决方案。

### 16.1 有多少个结点？

由于内存的空间有限，我们常常关心已经使用掉了多少内存空间。如果我们修改上一章的链表程序(程序 15.5)，要能计算出整个程序一共产生了多少链表结点，我们该怎么做呢？

显然，我们需要一个计数器。每产生一个结点，计数器就加一；每消除一个结点，计数器就减一。由于结点的产生和消除只会与链表类或结点类的某些成员函数有关，所以这个计数器只能是一个全局变量了（全局变量的概念见 11.1 节），否则它将无法被各个成员函数访问和修改。

不过使用全局变量会带来严重的安全性问题。产生了多少个链表结点明明是和结点类有关的，却没有被封装在结点类里面。任何函数都能修改这个全局变量，不得不让我们担忧。

封装在类内部的数据是成员数据。想象一下，如果我们给链表结点类添加一个成员数据 `count`，那么链表结点类的定义就是这样：

```
class Node//定义一个链表结点类
{
public:

 //成员函数同程序 15.5
private:
 int idata;//存储数据保密
 char cdata;//存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
 int count;//新来的成员函数，用于记录产生了多少个结点
};
```

现在计数器是一个成员数据，可以被链表结点类的成员函数访问，也保证它不会被随便修改。但如果我们创建了三个结点对象 `a`、`b`、`c` 之后，我们发现 `a.count`、`b.count` 和 `c.count` 是三个互不相关的变量，也就是说它们的值可能是不一致的。更麻烦的是，我们不知道还会产生多少结点对象，如果新增一个结点对象，那么之前的每一个结点对象的 `count` 都要发生变化！

所以，我们需要一种方法，既能把 `count` 封装在类的内部，又能使各个对象的 `count` 相同。

### 静态成员数据

我们将产生的结点个数记为 `count`，它不是某一个结点所具有的属性，而应该是整个链



表结点类所具有的属性，或者说它是各个结点对象的共有属性。

如果我们把 `idata` 和 `cdata` 比作每个结点的私有财产，那么 `count` 就是所有结点的共有财产。`count` 能被任何一个结点使用，但事实上无论有多少个结点，`count` 只有一个。这样就不会发生 `a.count`、`b.count` 和 `c.count` 各不相同的情况了。在 C++ 中，用静态成员数据(Static Data Member)来描述这种共有属性。与一般的成员数据类似，静态成员数据也可以分为公有(Public)的和私有(Private)的。静态成员数据的声明方法为：

**static 数据类型 成员变量名；**

下面我们来看看如何给链表结点类增加一个静态成员数据：

```
class Node//定义一个链表结点类
{
public:

 //成员函数同程序 15.5
private:
 int idata;//存储数据保密
 char cdata;//存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
 static int count;//私有静态成员数据，用于记录产生了多少个结点
};
```

## 静态成员数据的初始化

由于静态成员数据不是仅仅属于某一个具体对象的，所以它不能在构造函数中被初始化。（否则岂不是每创建一个对象，静态成员数据都要被初始化一次？）如果类的头文件会被直接或间接地重复包含，则静态成员数据也会被重复初始化。为了避免这个问题，我们可以将类的声明和定义分离，如果忘记了这个问题可参见 11.2 节。如果类的头文件绝对不会被重复包含，那么把静态成员数据的初始化放在类的头文件中也是可以勉强接受的。

静态成员数据的初始化语句为：

**数据类型 类名::静态成员数据=初始值；**

## 静态成员函数

静态成员数据是某一个类所具有的属性，而不是某一个对象的属性，所以它的存在并不依赖于对象。那么，如果一个类没有任何对象实例时，所有的普通成员函数都无法使用，我们该如何访问私有的静态成员数据呢？

既然成员数据可以属于某一个类而不属于某一个具体的对象，成员函数能否这样呢？答案是肯定的。在 C++ 中，除了有静态成员数据，还有静态成员函数。静态成员函数也是属于某一个类而不属于某一个具体的对象的。静态成员函数的声明方法为：

**static 返回值类型 函数名(参数表)；**

不过，在声明静态成员函数时，却不能出现 **static**。

下面我们来看一下静态成员数据、静态成员函数在程序中如何使用：(程序 16.1)

```
//node.h
class Node//声明一个链表结点类
{
```

```

public:
 Node();//构造函数的声明
 Node(Node &n);
 Node(int i,char c='0');
 Node(int i,char c,Node *p,Node *n);
 ~Node();//析构函数
 int readi() const;
 char readc() const;
 Node * readp() const;
 Node * readn() const;
 bool set(int i);
 bool set(char c);
 bool setp(Node *p);
 bool setn(Node *n);
 static int allocation();//静态成员函数定义，返回已分配结点数
private:
 int idata;//存储数据保密
 char cdata;//存储数据保密
 Node *prior;//前驱结点的存储位置保密
 Node *next;//后继结点的存储位置保密
 static int count;//静态成员数据，存储分配结点数
};
//node.cpp，把类声明和定义拆分开了
#include "node.h"//如果没包含头文件连接时会出现错误
#include <iostream>
using namespace std;
int Node::count=0;//静态成员数据初始化
//未定义的函数与程序 15.5 相同
Node::Node()//构造函数的定义
{
 cout <<"Node constructor is running..." <<endl;
 count++;//分配结点数增加
 idata=0;
 cdata='0';
 prior=NULL;
 next=NULL;
}
Node::Node(int i,char c)//构造函数重载 1
{
 cout <<"Node constructor is running..." <<endl;
 count++;//分配结点数增加
 idata=i;
 cdata=c;
 prior=NULL;
}

```

```

 next=NULL;
}
Node::Node(int i,char c,Node *p,Node *n)//构造函数重载 2
{
 cout <<"Node constructor is running..." <<endl;
 count++;//分配结点数增加
 idata=i;
 cdata=c;
 prior=p;
 next=n;
}
Node::Node(Node &n)
{
 count++;//分配结点数增加
 idata=n.idata;
 cdata=n.cdata;
 prior=n.prior;
 next=n.next;
}
Node::~Node()
{
 count--;//分配结点数减少
 cout <<"Node destructor is running..." <<endl;
}
int Node::allocation()//在定义静态成员函数时不能出现 static
{
 return count;//返回已分配结点数
}
//linklist.h 同程序 15.5
//main.cpp
#include "Linklist.h"
#include <iostream>
using namespace std;
int main()
{
 int tempi;
 char tempc;
 cout <<"请输入一个整数和一个字符: " <<endl;
 cin >>tempi >>tempc;
 Linklist a(tempi,tempc);
 a.Locate(tempi);
 a.Insert(1,'C');
 a.Insert(2,'B');
 cout <<"After Insert" <<endl;
}

```

```

a.Show();
cout <<"Node Allocation:" <<Node::allocation() <<endl;//调用静态成员函数
Node b;
cout <<"An independent node created" <<endl;
cout <<"Node Allocation:" <<b.allocation() <<endl;//调用静态成员函数
return 0;
}

```

运行结果：

请输入一个整数和一个字符：

3 F

Node constructor is running...

Linklist constructor is running...

Node constructor is running...

Node constructor is running...

After Insert

3 F

2 B

1 C

Node Allocation:3

Node constructor is running...

An independent node created

Node Allocation:4

Node destructor is running...

Linklist destructor is running...

Node destructor is running...

Node destructor is running...

Node destructor is running...

可见，记录结点分配情况的功能已经实现。该程序中出现了两种调用静态成员函数的方法，一种是**类名::静态成员函数名（参数表）**，另一种是**对象名.静态成员函数名（参数表）**，这两种调用方法的效果是相同的。由于**静态成员函数是属于类的，不是属于某一个具体对象**，所以它分不清到底是访问哪个对象的非静态成员数据，故而不能访问非静态成员数据。

## 名不符实的 static

在第 11 章中，我们遇到过保留字 `static`，其作用是使局部变量在函数运行结束后继续存在，成为静态变量。（或者说存储空间在编译时静态分配）而本章中 `static` 的含义是“每个类中只含有一个”，与第 11 章中的 `static` 毫不相关。所以说这里的 `static` 是名不符实的。

## 16.2 类的好朋友

在编写链表类的时候我们有着这样的困惑：链表类和链表结点类都是我们编写的，我们能保证链表类对链表结点类的操作都是安全的。但由于类的封装性，我们不得不编写一些成员函数，以便于链表类访问链表结点类的私有成员数据。好在链表结点类的成员数据并不是很多，否则岂不是需要一大堆成员函数来供别的类访问？对于这种情况，我们能否告诉链表

结点类：“链表类是安全的，让它访问你的私有成员吧”？

在 C++ 中，可以用友元来解决这种尴尬的问题。所谓友元，就是作为一个类的“朋友”，可以例外地访问它的私有成员数据或私有成员函数。

## 友元类

类似于链表类和链表结点类的问题，我们可以用友元类来解决。即链表类是链表结点类的“朋友”，可以直接访问链表结点类的私有成员数据或私有成员函数。显然，要做链表结点类的“朋友”，必须要得到链表结点类的认可。所以我们必须在链表结点类的声明中告诉电脑，链表类是它认可的“朋友”，可以访问它的私有成员。声明友元类的语句格式为：

**friend class 类名;**

下面我们来看一下，友元是如何让我们更方便地设计程序的：（程序 16.2.1）

```
//node.h
class Node//声明一个链表结点类
{
 friend class Linklist;//在 Node 类中声明友元类 Linklist
public:
 Node();
 Node(Node &n);
 Node(int i,char c='0');
 Node(int i,char c,Node *p,Node *n);
 ~Node();
 static int allocation();
private:
 int idata;
 char cdata;
 Node *prior;
 Node *next;
 static int count;
};

//node.cpp
#include "node.h"
#include <iostream>
using namespace std;
int Node::count=0;
Node::Node()
{
 cout <<"Node constructor is running..." <<endl;
 count++;
 idata=0;
 cdata='0';
 prior=NULL;
 next=NULL;
}
Node::Node(int i,char c)
```

```
{
 cout <<"Node constructor is running..." <<endl;
 count++;
 idata=i;
 cdata=c;
 prior=NULL;
 next=NULL;
}
Node::Node(int i,char c,Node *p,Node *n)
{
 cout <<"Node constructor is running..." <<endl;
 count++;
 idata=i;
 cdata=c;
 prior=p;
 next=n;
}
Node::Node(Node &n)
{
 count++;
 idata=n.idata;
 cdata=n.cdata;
 prior=n.prior;
 next=n.next;
}
Node::~~Node()
{
 count--;
 cout <<"Node destructor is running..." <<endl;
}
int Node::allocation()
{
 return count;
}
//linklist.h
#include "node.h"
#include <iostream>
using namespace std;
class Linklist//定义一个链表类
{
public:
 Linklist(int i,char c);
 Linklist(Linklist &l);
 ~Linklist();
```

```

 bool Locate(int i);
 bool Locate(char c);
 bool Insert(int i=0,char c='0');
 bool Delete();
 void Show();
 void Destroy();
private:
 Node head;
 Node * pcurrent;
};
Linklist::Linklist(int i,char c):head(i,c)//链表的构造函数
{
 cout<<"Linklist constructor is running..."<<endl;
 pcurrent=&head;
}
Linklist::Linklist(Linklist &l):head(l.head)
{
 cout<<"Linklist Deep cloner running..." <<endl;
 pcurrent=&head;
 Node * ptemp1=l.head.next;//直接访问私有成员数据
 while(ptemp1!=NULL)
 {
 Node * ptemp2=new Node(ptemp1->idata,ptemp1->cdata,pcurrent,NULL);
 pcurrent->next=ptemp2;
 pcurrent=pcurrent->next;
 ptemp1=ptemp1->next;
 }
}
Linklist::~Linklist()
{
 cout<<"Linklist destructor is running..."<<endl;
 Destroy();
}
bool Linklist::Locate(int i)
{
 Node * ptemp=&head;
 while(ptemp!=NULL)
 {
 if(ptemp->idata==i)
 {
 pcurrent=ptemp;
 return true;
 }
 ptemp=ptemp->next;
 }
}

```

```
 }
 return false;
}
bool Linklist::Locate(char c)
{
 Node * ptemp=&head;
 while(ptemp!=NULL)
 {
 if(ptemp->cdata==c)
 {
 pcurrent=ptemp;
 return true;
 }
 ptemp=ptemp->next;
 }
 return false;
}
bool Linklist::Insert(int i,char c)
{
 if(pcurrent!=NULL)
 {
 Node * temp=new Node(i,c,pcurrent,pcurrent->next);
 if (pcurrent->next!=NULL)
 {
 pcurrent->next->prior=temp;
 }
 pcurrent->next=temp;
 return true;
 }
 else
 {
 return false;
 }
}
bool Linklist::Delete()
{
 if(pcurrent!=NULL && pcurrent!=&head)
 {
 Node * temp=pcurrent;
 if (temp->next!=NULL)
 {
 temp->next->prior=pcurrent->prior;
 }
 temp->prior->next=pcurrent->next;
 }
}
```



```

 pcurrent=temp->prior;
 delete temp;
 return true;
 }
 else
 {
 return false;
 }
}
void Linklist::Show()
{
 Node * ptemp=&head;
 while (ptemp!=NULL)
 {
 cout <<ptemp->idata <<"\t" <<ptemp->cdata <<endl;
 ptemp=ptemp->next;
 }
}
void Linklist::Destroy()
{
 Node * ptemp1=head.next;
 while (ptemp1!=NULL)
 {
 Node * ptemp2=ptemp1->next;
 delete ptemp1;
 ptemp1=ptemp2;
 }
 head.next=NULL;
}

```

//main.cpp 同程序 16.1

运行结果:

请输入一个整数和一个字符:

3 F

Node constructor is running...

Linklist constructor is running...

Node constructor is running...

Node constructor is running...

After Insert

3      F

2      B

1      C

Node Allocation:3

Node constructor is running...

An independent node created

Node Allocation:4

Node destructor is running...

Linklist destructor is running...

Node destructor is running...

Node destructor is running...

Node destructor is running...

可以看到，程序的运行结果和程序 16.1 的结果一样，但是链表结点类没有程序 16.1 中那么繁琐。并且在链表类中完全都是直接访问链表结点类的成员数据，大大减少了调用函数产生的开销，这样执行程序的效率也就得以提高了。

## 友元函数

私有成员数据除了可能被别的类访问之外，也可能被别的函数或别的类的部分成员函数访问。为了保证类的封装性，我们可以以函数作为单位，“对外开放”类的私有成员。与声明友元类类似，如果我们想用函数访问链表结点类的私有成员数据，则那些函数必须得到链表结点类的认可。声明友元函数的语句格式为：

**friend 返回值类型 函数名 (参数表);**

如果该函数是某个类的成员函数，则语句格式为：

**friend 返回值类型 类名::函数名 (参数表);**

需要注意的是，在声明友元成员函数时，可能会牵扯出一系列的类的声明顺序问题。当类的结构本身就比较复杂时，友元的使用可能会使得这个问题愈加突出。

下面我们就用友元函数来输出一个结点的信息：（程序 16.2.2）

```
//node.h
class Node
{
 friend class Linklist; //在 Node 类中声明友元类 Linklist
 friend void ShowNode(Node &n); //声明友元函数 ShowNode
public:
 Node();
 Node(Node &n);
 Node(int i,char c='0');
 Node(int i,char c,Node *p,Node *n);
 ~Node();
 static int allocation();
private:
 int idata;
 char cdata;
 Node *prior;
 Node *next;
 static int count;
};
//node.cpp
//其余部分同程序 16.2.1
void ShowNode(Node &n)
{
```

```

 cout <<n.idata <<'\t' <<n.cdata <<endl;//友元函数可以访问私有成员数据
 }
//linklist.h 同程序 16.2.1
//main.cpp
#include <iostream>
#include "Linklist.h"
using namespace std;
int main()
{
 int tempi;
 char tempc;
 cout <<"请输入一个整数和一个字符: " <<endl;
 cin >>tempi >>tempc;
 Linklist a(tempi,tempc);
 a.Locate(tempi);
 a.Insert(1,'C');
 a.Insert(2,'B');
 cout <<"After Insert" <<endl;
 a.Show();
 Node b(4,'F');
 cout <<"An independent node created" <<endl;
 cout <<"Use friend function to show node" <<endl;
 ShowNode(b);//用友元函数输出 b 结点的内容
 return 0;
}

```

运行结果:

请输入一个整数和一个字符:

3 F

Node constructor is running...

Linklist constructor is running...

Node constructor is running...

Node constructor is running...

After Insert

3        F

2        B

1        C

Node constructor is running...

An independent node created

Use friend function to show node

4        G

Node destructor is running...

Linklist destructor is running...

Node destructor is running...

Node destructor is running...

Node destructor is running...

我们看到函数 ShowNode 成功地访问了链表结点 b 的私有成员数据。所以当函数要访问一个或多个对象的私有成员时，我们可以用友元来解决这个问题。

## 友元的利与弊

我们使用了友元之后，发现在设计程序的时候方便了很多。原先的那些私有成员都能轻松地被访问了。于是我们不用去写那些繁琐的成员函数，程序执行的时候也减少了函数的调用次数，提高了运行效率。

一个“好朋友”带来的是效率和方便，而一个“坏朋友”却能带来更多的麻烦。**友元**的存在，破坏了类的封装性。一个类出现问题，就不仅仅是由这个类本身负责了，还可能和它众多的友元有关。这无疑使得检查调试的范围突然扩大了许多，难度也陡然增加。

所以，我们在使用友元的时候，权衡使用友元的利弊，使程序达到最佳的效果。

试试看：

把程序 16.2.1 或 16.2.2 中友元的声明放入 public 或 private 中，看看是否影响程序运行的结果。

结论：友元的声明可以在类声明中的任何位置。

## 16.3 多功能的操作符

在表达式中，我们常会用到各种操作符（运算符），例如  $1+3$  和  $4*2$ 。然而，这些操作符只能用于 C++ 内置的一些基本数据类型。如果我们自己编写一个复数类，它也会有加减法的操作，那么它能否摆脱一串冗长的函数名，而享用加号呢？

在第六章我们学到过，函数是可以重载的，即同名函数针对不同数据类型的参数实现类似的功能。在 C++ 中，操作符也是可以重载的，同一操作符对于不同的自定义数据类型可以进行不同的操作。

### 作为成员函数

在我们学习操作符重载前，我们先看看原先这个复数类是如何定义的：（程序 16.3.1）

```
//complex.h
#include <iostream>
using namespace std;
class Complex//声明一个复数类
{
public:
 Complex(Complex &a);//拷贝构造函数
 Complex(double r=0,double i=0);
 void display();//输出复数的值
 void set(Complex &a);
 Complex plus(Complex a);//复数的加法
 Complex minus(Complex a); //复数的减法
 Complex plus(double r); //复数与实数相加
```

```
Complex minus(double r); //复数与实数相减
private:
 double real;//复数实部
 double img;//复数虚部
};
Complex::Complex(Complex &a)
{
 real=a.real;
 img=a.img;
}
Complex::Complex(double r,double i)
{
 real=r;
 img=i;
}
void Complex::display()
{
 cout <<real <<(img>=0?" "+"") <<img <<"i";//适合显示 1-3i 等虚部为负值的复数
}
void Complex::set(Complex &a)
{
 real=a.real;
 img=a.img;
}
Complex Complex::plus(Complex a)
{
 Complex temp(a.real+real,a.img+img);
 return temp;
}
Complex Complex::minus(Complex a)
{
 Complex temp(real-a.real,img-a.img);
 return temp;
}
Complex Complex::plus(double r)
{
 Complex temp(real+r,img);
 return temp;
}
Complex Complex::minus(double r)
{
 Complex temp(real-r,img);
 return temp;
}
```

```

//main.cpp
#include "complex.h"
#include <iostream>
using namespace std;
int main()
{
 Complex a(3,2),b(5,4),temp;
 temp.set(a.plus(b));//temp=a+b
 temp.display();
 cout <<endl;
 temp.set(a.minus(b));//temp=a-b
 temp.display();
 cout <<endl;
 return 0;
}

```

运行结果:

8+6i

-2-2i

虽然程序 16.3.1 已经实现了复数的加减法，但是其表达形式极为麻烦，如果有复数 a、b、c 和 d，要计算  $a+b-(c+d)$  将会变得非常复杂。如果不是调用函数，而是使用操作符的话，就会直观得多了。

声明一个操作符重载的语句格式为：

**返回值类型 operator 操作符 (参数表);**

事实上，在声明和定义操作符重载的时候，我们可以将其看作函数了，只不过这个函数名是一些操作符。在声明和定义操作符重载时需要注意以下几点：

(1) 操作符只能是 C++ 中存在的一些操作符，自己编造的操作符是不能参与操作符重载的。另外，“::”（域解析操作符）、“.”（成员操作符）、“……? ……: ……”（条件操作符）和 `sizeof` 等操作符不允许重载。

(2) 参数表中罗列的是操作符的各个操作数。重载后操作数的个数应该与原来相同。不过如果操作符作为成员函数，则调用者本身是一个操作数，故而参数表中会减少一个操作数。（请对比程序 16.3.2 与程序 16.3.3）

(3) 各个操作数至少要有一个是自定义类型的数据，如结构或类。

(4) 尽量不要混乱操作符的含义。如果把加号用在减法上，会使程序的可读性大大下降。

下面我们把操作符作为成员函数，来实现复数的加减法：（程序 16.3.2）

```

//complex.h
#include <iostream>
using namespace std;
class Complex//声明一个复数类
{
public:
 Complex(Complex &a);
 Complex(double r=0,double i=0);
 void display();
}

```

```

 void operator =(Complex a);//赋值操作
 Complex operator +(Complex a);//加法操作
 Complex operator -(Complex a);//减法操作
 Complex operator +(double r);//加法操作
 Complex operator -(double r);//减法操作
private:
 double real;
 double img;
};
//未定义的函数与程序 16.3.1 相同
void Complex::operator =(Complex a)
{
 real=a.real;
 img=a.img;
}
Complex Complex::operator +(Complex a)
{
 Complex temp(a.real+real,a.img+img);
 return temp;
}
Complex Complex::operator -(Complex a)
{
 Complex temp(real-a.real,img-a.img);
 return temp;
}
Complex Complex::operator +(double r)
{
 Complex temp(real+r,img);
 return temp;
}
Complex Complex::operator -(double r)
{
 Complex temp(real-r,img);
 return temp;
}
//main.cpp
#include "complex.h"
#include <iostream>
using namespace std;
int main()
{
 Complex a(3,2),b(5,4),c(1,1),d(4,2),temp;
 temp=a+b;//这样的复数加法看上去很直观
 temp.display();
}

```

```

 cout <<endl;
 temp=a-b;
 temp.display();
 cout <<endl;
 temp=a+b-(c+d);//可以和括号一起使用了
 temp.display();
 cout <<endl;
 return 0;
}

```

运行结果：

```

8+6i
-2-2i
3+3i

```

以上程序的 main.cpp 中，复数的加法表达得非常简洁易懂，与程序 16.3.1 相比有了很大的进步。并且，我们发现使用了括号以后，可以更方便地描述各种复杂的运算。**操作符在重载之后，结合性和优先级是不会发生变化的，符合用户本来的使用习惯。**

## 作为友元

前面我们把操作符作为成员函数，实现了复数的加减法。如果我们把操作符作为普通的函数重载，则需要将其声明为友元。这时，参数表中的操作数个数应该与操作符原来要求的操作数个数相同。

下面我们来看一下，用友元和操作符重载来实现复数的加减法：（程序 16.3.3）

```

//complex.h
#include <iostream.h>//由于 VC 编译器存在问题，这里使用标准的写法居然无法通过编译
class Complex
{
public:
 Complex(Complex &a);
 Complex(double r=0,double i=0);
 void display();
 friend Complex operator +(Complex a,Complex b);//作为友元
 friend Complex operator -(Complex a,Complex b);
 friend Complex operator +(Complex a,double r);
 friend Complex operator -(Complex a,double r);
private:
 double real;
 double img;
};
//未定义的函数与程序 16.3.1 相同
Complex operator +(Complex a,Complex b)
{
 Complex temp(a.real+b.real,a.img+b.img);
 return temp;
}

```



```

Complex operator -(Complex a,Complex b)
{
 Complex temp(a.real-b.real,a.img-b.img);
 return temp;
}
Complex operator +(Complex a,double r)
{
 Complex temp(a.real+r,a.img);
 return temp;
}
Complex operator -(Complex a,double r)
{
 Complex temp(a.real-r,a.img);
 return temp;
}
//main.cpp
#include "complex.h"
#include <iostream.h>//由于 VC 编译器存在问题，这里使用标准的写法无法通过编译
int main()
{
 Complex a(3,2),b(5,4),c(1,1),d(4,2),temp;
 temp=a+b;
 temp.display();
 cout <<endl;
 temp=a-b;
 temp.display();
 cout <<endl;
 temp=a+b-(c+d);
 temp.display();
 cout <<endl;
 return 0;
}

```

运行结果：

8+6i

-2-2i

3+3i

在上面这个程序中，加号和减号操作符由成员函数变成了友元函数。细心的读者可能注意到了，那个赋值操作符的定义跑哪儿去了？

事实上，赋值操作符有点类似于默认拷贝构造函数，也具有默认的对象赋值功能。所以即使没有对它进行重载，也能使用它对对象作赋值操作。但是如果要对赋值操作符进行重载，则必须将其作为一个成员函数，否则程序将无法通过编译。

在操作符重载中，友元的优势尽显无遗。特别是当操作数为几个不同类的对象时，友元不失为一种良好的解决办法。

## 又见加加和减减

在第五章我们学习了增减量操作符，并且知道它们有前后之分。那么增减量操作符是如何重载的呢？同样是一个操作数，它又是如何区分前增量和后增量的呢？

前增量操作符是“先增后赋”，在操作符重载中我们理解为先做自增，然后把操作数本身返回。后增量操作符是“先赋后增”，在这里我们理解为先把操作数的值返回，然后操作数自增。所以，前增量操作返回的是操作数本身，而后增量操作返回的只是一个临时的值。

在 C++ 中，为了区分前增量操作符和后增量操作符的重载，规定后增量操作符多一个整型参数。这个参数仅仅是用于区分前增量和后增量操作符，不参与到实际运算中去。

下面我们就来看看，如何重载增量操作符：（程序 16.3.4）

```
//complex.h
#include <iostream.h>//由于 VC 编译器存在问题，这里使用标准的写法无法通过编译
class Complex
{
public:
 Complex(Complex &a);
 Complex(double r=0,double i=0);
 void display();
 friend Complex operator +(Complex a,Complex b);
 friend Complex operator -(Complex a,Complex b);
 friend Complex operator +(Complex a,double r);
 friend Complex operator -(Complex a,double r);
 friend Complex& operator ++(Complex &a);//前增量操作符重载
 friend Complex operator ++(Complex &a,int); //后增量操作符重载
private:
 double real;
 double img;
};
//未定义的函数与程序 16.3.3 相同
Complex& operator ++(Complex &a)
{
 a.img++;
 a.real++;
 return a;//返回类型为 Complex 的引用，即返回操作数 a 本身
}
Complex operator ++(Complex &a,int)//第二个整型参数表示这是后增量操作符
{
 Complex temp(a);
 a.img++;
 a.real++;
 return temp;//返回一个临时的值
}
//main.cpp
#include "complex.h"
```

```
#include <iostream.h> //由于 VC 编译器存在问题，这里使用标准的写法无法通过编译
int main()
{
 Complex a(2,2),b(2,4),temp;
 temp=(a++)+b;
 temp.display();
 cout <<endl;
 temp=b-(++a);
 temp.display();
 cout <<endl;
 a.display();
 cout <<endl;
 return 0;
}
```

运行结果：

4+6i

-2+0i

4+4i

根据运行结果，可以看到 `a++`和`++a` 被区分开来了。而调用后增量操作符的时候，操作数仍然只有一个，与那个用于区分的整型参数无关。

至此，我们已经学完了类的一些基本特性和要素。在接下来的章节中，我们要更深入地发掘面向对象程序设计的优势。

## 习题

- 1、改写程序 16.1，使得链表类也能记录链表对象的个数。
- 2、改写程序 16.2.2，编写一个额外的友元函数（非成员函数），实现链表结点插入到链表中，其函数原型为 `bool Insert(Linklist &list,int i,char c)`；其中参数是 `list` 是链表对象，参数 `i` 对应 `idata`，参数 `c` 对应 `cdata`，返回值表示是否插入成功。
- 3、改写程序 16.3.4，编写实数加虚数、实数减虚数、前减量和后减量的操作符重载。

## 第十七章 父与子

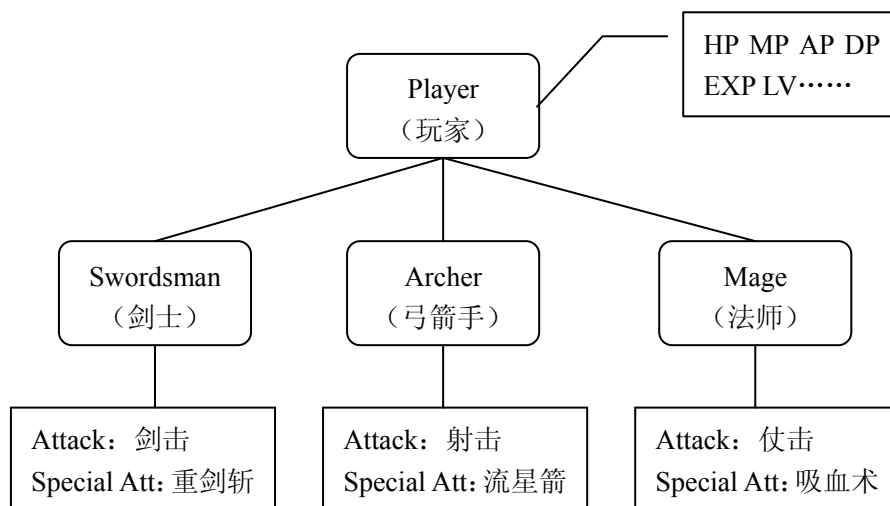
在前面的章节，我们学会了如何编写一个完整的类。然而，面向对象的优点还没有被完全体现出来。特别是在编写一些相似的类时，可能会造成很多的浪费。本章就将以一个文字游戏为例，向大家介绍类的继承问题。

### 17.1 剑士·弓箭手·法师的困惑

在一个角色扮演类游戏（RPG）中，可能有各种不同职业的玩家，比如剑士、弓箭手和法师。虽然他们的职业不同，却有着一些相似之处：他们都具有生命值（Health Point——HP）、魔法值（Magic Point——MP）、攻击力（Attack Point——AP）、防御力（Defense Point——DP）、经验值（Experience——EXP）和等级（Level——LV）。虽然他们有着相似之处，但又不完全相同：剑士和弓箭手都具有普通攻击的技能，只不过剑士用的是剑，而弓箭手用的是弓箭。

这样看来，我们有麻烦了。如果只用一个类来描述三种不同职业的玩家，肯定无法描述清楚。毕竟这三种职业不是完全相同的。如果用三个类来描述这三种职业，那么三者的共同点和内在联系就无法体现出来，并且还造成了相同属性和功能的重复开发。

我们需要有一种好的方法，既能把剑士、弓箭手和法师的特点描述清楚，又能减少重复的开发和冗余的代码。在 C++ 中，有一种称为继承的方法，使我们可以用一种已经编写好的类来扩写成一个新的类。新的类具有原有类的所有属性和操作，也可以在原有类的基础上作一些修改和增补。继承实质上是源于人们对事物的认知过程：从抽象概念到具体事物。下面我们就来看看剑士、弓箭手和法师的逻辑关系：

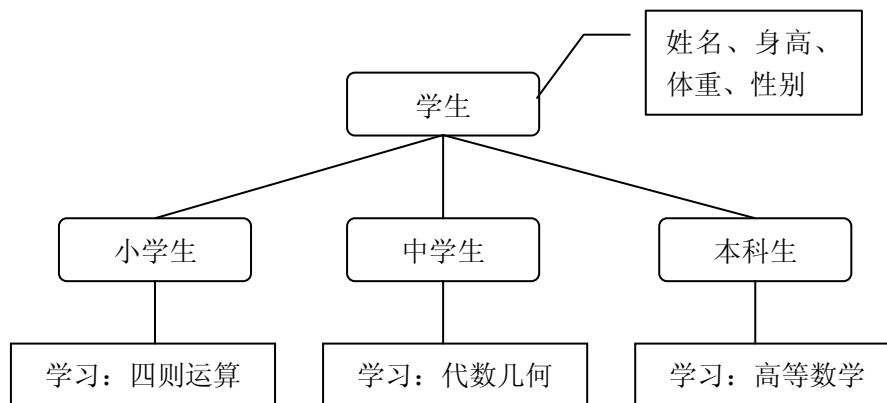


(图 17.1.1)

在上图中玩家是一个抽象的概念，剑士、弓箭手和法师是具体的事物。任何一个玩家都具有生命值、魔法值等属性，具有发动普通攻击和特殊攻击的能力。不同职业的玩家在发动普通攻击和特殊攻击时，有着不同的效果。

如果你不太玩游戏，或者对剑士、弓箭手没有什么概念，那么我们再来看看学生这个例子。学生是一个抽象的概念，具体的有本科生、中学生、小学生等。任何一个学生都具有姓

名、身高、体重、性别等属性，具有学习的能力。不同阶段的学生在学习时，内容会有所不同。小学生学习四则运算，中学生学习代数几何，本科生学习高等数学。如下图：



(图 17.1.2)

为了描写小学生、中学生、本科生，我们可以写三个不同的类，但是会造成部分属性和功能的重复开发。我们也可以先设计一个学生类，描述出各种学生的共同属性或功能，然后再针对不同种类的学生做细节的修改。显然，第二种做法更为省力、合理。

## 17.2 面向对象特点二：继承性

如果有一个类，我们可以将其实例化，成为若干个对象。另外，如果我们希望对这个类加以升级改造，我们可以将这个类继承，形成子类（或者称为派生类），被继承的类则称为父类（或者称为基类）。**实例化和继承是一个类的两种发展方向。继承能够减少我们开发程序的工作量，提高类的重用性。**

如果我们把编写一个类看作是一次生产，那么产品（即编写出来的类）可以有两种用途：一种是将产品直接使用，相当于将类实例化；另一种是将产品用于再生产，相当于将类继承。类在这种不断的“再生产”中变得更为强大、健全。

在第 15 章中，我们曾将链表结点类的实例对象作为链表类的成员数据。这称为对象的组合，它与类的继承也是完全不同的概念。**继承 (Inheritance) 是概念的延续，子类和父类一般都是概念扩展的关系，我们通常把这种关系称为“是”关系。**比如：本科生是学生，自行车是交通工具。而对象的组合是因功能需求产生的从属关系，我们通常把这种关系称为**“有”关系**。比如：链表有一个头结点，电脑有一个中央处理器等等。

关于如何更合理地设置类以及类与类之间关系的问题，会在软件工程这门课中作更详尽的介绍。

## 17.3 继承的实现

了解继承的概念之后，我们就来学习一下如何实现继承。

### 私有和保护

在第 14 章中我们说到，成员函数或成员数据可以是公有或者私有的。如果是公有的，那么它们可以被直接访问；如果是私有的，那么它们无法被直接访问。同时，我们还提到一个 `protected` 保留字，在没有使用继承的时候，它与 `private` 的效果是一样的，即无法被直接

访问。如果使用了继承，我们就能体会到 `protected` 和 `private` 的差别。

**private**（私有）和 **protected**（保护）都能实现类的封装性。**private** 能够对外部和子类保密，即除了成员所在的类本身可以访问之外，别的都不能直接访问。**protected** 能够对外部保密，但允许子类直接访问这些成员。`public`、`private` 和 `protected` 对成员数据或成员函数的保护程度可以用下表来描述：

| 访问方式<br>成员类型     | 类外部  | 子类   | 本类   |
|------------------|------|------|------|
| <b>public</b>    | 允许访问 | 允许访问 | 允许访问 |
| <b>protected</b> | 禁止访问 | 允许访问 | 允许访问 |
| <b>private</b>   | 禁止访问 | 禁止访问 | 允许访问 |

所以，当我们使用到继承的时候，必须考虑清楚：成员数据或成员函数到底应该是私有的还是保护的。

## 一个简单的例子

首先我们以一个学生类为例，介绍继承的写法：（程序 17.3.1）

```
//student.h
#include <iostream>
using namespace std;
class student//学生类作为父类
{
public:
 student(char *n,int a,int h,int w);//带参数的构造函数
 student();//不带参数的构造函数
 void set(char *n,int a,int h,int w);//设置
 char * sname();
 int sage();
 int sheight();
 int sweight();
protected:
 char name[10];//姓名
 int age;//年龄
 int height;//身高
 int weight;//体重
private:
 int test;
};
char * student::sname()
{
 return name;
}
int student::sage()
{
 return age;
}
```

```
}
int student::sheight()
{
 return height;
}
int student::sweight()
{
 return weight;
}
void student::set(char *n,int a,int h,int w)
{
 int i;
 for (i=0;n[i]!='\0';i++)
 {
 name[i]=n[i];
 }
 name[i]='\0';
 age=a;
 height=h;
 weight=w;
 return;
}
student::student(char *n,int a,int h,int w)
{
 cout <<"Constructing a student with parameter..." <<endl;
 set(n,a,h,w);
}
student::student()
{
 cout <<"Constructing a student without parameter..." <<endl;
}
//undergraduate.h
#include "student.h"
class Undergraduate:public student//本科生类作为子类，继承了学生类
{
public:
 double score();
 void setGPA(double g);//设置绩点
 bool isAdult();//判断是否成年
protected:
 double GPA;//本科生绩点
};
double Undergraduate::score()
{
```

```

 return GPA;
}
void Undergraduate::setGPA(double g)
{
 GPA=g;
 return;
}
bool Undergraduate::isAdult()
{
 return age>=18?true:false;//子类访问父类的保护成员数据
}
//main.cpp
#include <iostream>
#include "undergraduate.h"
using namespace std;
int main()
{
 Undergraduate s1;//新建一个本科生对象
 s1.set("Tom",21,178,60);
 s1.setGPA(3.75);
 cout <<s1.sname() <<endl;
 cout <<s1.sage() <<endl;
 cout <<s1.sheight() <<endl;
 cout <<s1.sweight() <<endl;
 cout <<s1.score() <<endl;
 cout <<s1.isAdult() <<endl;
 return 0;
}

```

运行结果:

Constructing a student without parameter...

Tom

21

178

60

3.75

1

试试看:

修改程序 17.3.1, 试试看在 main 函数中, 能否直接访问从学生类继承而来的成员数据 age 和 test。

在使用继承之前, 我们必须保证父类是已经定义好的。如果父类是虚无的、没有被定义的, 那么子类也就没什么好继承的了。定义一个子类的语法格式为:

**class** 子类名: 继承方式 父类名;

根据程序 17.3.1 的运行结果, 我们可以清楚地看到, 学生类里面的公有和保护成员都已



经被继承到本科生类。本科生类可以使用学生类的成员函数，也可以访问学生类的保护成员。而本科生类中定义的成员则是对学生类的补充，并且也能够被使用。

## 继承的方式

在程序 17.3.1 中，我们选择的继承方式是 `public`。和成员的类型一样，除了 `public` 之外，继承方式还有 `protected` 和 `private`。那么，这三种继承方式到底有什么区别呢？

**public 是公有继承，或称为类型继承。**它主要体现的是概念的延伸和扩展，父类所有的公有、保护成员都将按部就班地继承到子类中。父类的公有成员在子类中依然是公有的，父类的保护成员在子类中依然是保护的。比如程序 17.3.1 中的学生类和本科生类就是这样的关系。

**private 是私有继承，或称为私有的实现继承。**它主要体现的是父类成员的重用。父类所有的公有、保护成员继承到子类时，类型会发生改变。父类的公有成员在子类中变成了私有成员，父类的保护成员在子类中也变成了私有成员。这时，我们可以利用从父类继承而来的成员函数来实现子类的成员函数，并且不必担心外部直接访问父类的成员函数，破坏了子类的秩序。比如我们认为栈是一种特殊的链表，它只能从链表尾部添加或删除结点，栈的压栈和退栈功能可以方便地由链表类的成员函数实现。但是，如果外部还能直接访问从链表类继承而来的成员函数，那么就可以在栈的任何位置插入结点，栈就会被破坏。

**protected 是保护继承，或称为保护的实现继承。**与私有继承类似，它也是体现父类成员的重用。只不过父类的公有成员和保护成员在子类中都变成了保护成员。因此，如果有一个孙类继承了子类，那么父类中的成员也将被继承，成为孙类的保护成员。

`public`、`private` 和 `protected` 三种继承方式可以用下表描述。其中右下角的九个单元格表示各种父类成员在对应的继承方式下，成为子类成员后的性质。

| 父类成员<br>继承方式 | public    | protected | private |
|--------------|-----------|-----------|---------|
| public       | public    | protected | 禁止访问    |
| protected    | protected | protected | 禁止访问    |
| private      | private   | private   | 禁止访问    |

在使用继承的时候，我们必须根据实际需要选择合适的继承方式。下面我们以前面提到的栈继承链表为例，理解一下私有继承方式：（程序 17.3.2）

```
//node.h
#include <iostream>
using namespace std;
class Node
{
 friend class Linklist;//链表类作为友元类
 friend class Stack;//栈类作为友元类
public:
 Node();
 Node(Node &n);
 Node(int i,char c='0');
 Node(int i,char c,Node *p,Node *n);
 ~Node();
private:
```

```
 int idata;
 char cdata;
 Node *prior;
 Node *next;
};
Node::Node()
{
 cout <<"Node constructor is running..." <<endl;
 idata=0;
 cdata='0';
 prior=NULL;
 next=NULL;
}
Node::Node(int i,char c)
{
 cout <<"Node constructor is running..." <<endl;
 idata=i;
 cdata=c;
 prior=NULL;
 next=NULL;
}
Node::Node(int i,char c,Node *p,Node *n)
{
 cout <<"Node constructor is running..." <<endl;
 idata=i;
 cdata=c;
 prior=p;
 next=n;
}
Node::Node(Node &n)
{
 idata=n.idata;
 cdata=n.cdata;
 prior=n.prior;
 next=n.next;
}
Node::~~Node()
{
 cout <<"Node destructor is running..." <<endl;
}
//linklist.h
#include "node.h"
#include <iostream>
using namespace std;
```

```

class Linklist
{
public:
 Linklist(int i=0,char c='0');
 Linklist(Linklist &l);
 ~Linklist();
 bool Locate(int i);
 bool Locate(char c);
 bool Insert(int i=0,char c='0');
 bool Delete();
 void Show();
 void Destroy();
protected://原私有成员改为保护成员，以便于 Stack 类继承
 Node head;
 Node * pcurrent;
};
Linklist::Linklist(int i,char c):head(i,c)
{
 cout<<"Linklist constructor is running..."<<endl;
 pcurrent=&head;
}
Linklist::Linklist(Linklist &l):head(l.head)
{
 cout<<"Linklist Deep cloner running..." <<endl;
 pcurrent=&head;
 Node * ptemp1=l.head.next;
 while(ptemp1!=NULL)
 {
 Node * ptemp2=new Node(ptemp1->idata,ptemp1->cdata,pcurrent,NULL);
 pcurrent->next=ptemp2;
 pcurrent=pcurrent->next;
 ptemp1=ptemp1->next;
 }
}
Linklist::~Linklist()
{
 cout<<"Linklist destructor is running..."<<endl;
 Destroy();
}
bool Linklist::Locate(int i)
{
 Node * ptemp=&head;
 while(ptemp!=NULL)
 {

```

```
 if(ptemp->idata==i)
 {
 pcurrent=ptemp;
 return true;
 }
 ptemp=ptemp->next;
 }
 return false;
}
bool Linklist::Locate(char c)
{
 Node * ptemp=&head;
 while(ptemp!=NULL)
 {
 if(ptemp->cdata==c)
 {
 pcurrent=ptemp;
 return true;
 }
 ptemp=ptemp->next;
 }
 return false;
}
bool Linklist::Insert(int i,char c)
{
 if(pcurrent!=NULL)
 {
 Node * temp=new Node(i,c,pcurrent,pcurrent->next);
 if (pcurrent->next!=NULL)
 {
 pcurrent->next->prior=temp;
 }
 pcurrent->next=temp;
 return true;
 }
 else
 {
 return false;
 }
}
bool Linklist::Delete()
{
 if(pcurrent!=NULL && pcurrent!=&head)
 {
```

```
Node * temp=pcurrent;
if (temp->next!=NULL)
{
 temp->next->prior=pcurrent->prior;
}
temp->prior->next=pcurrent->next;
pcurrent=temp->prior;
delete temp;
return true;
}
else
{
 return false;
}
}
void Linklist::Show()
{
 Node * ptemp=&head;
 while (ptemp!=NULL)
 {
 cout <<ptemp->idata <<'\t' <<ptemp->cdata <<endl;
 ptemp=ptemp->next;
 }
}
void Linklist::Destroy()
{
 Node * ptemp1=head.next;
 while (ptemp1!=NULL)
 {
 Node * ptemp2=ptemp1->next;
 delete ptemp1;
 ptemp1=ptemp2;
 }
 head.next=NULL;
}
//stack.h
#include "linklist.h"
class Stack:private Linklist//私有继承链表类
{
public:
 bool push(int i,char c);
 bool pop(int &i,char &c);
 void show();
};
```

```
bool Stack::push(int i,char c)
{
 while (pcurrent->next!=NULL)
 pcurrent=pcurrent->next;
 return Insert(i,c);//用链表类的成员函数实现功能
}
bool Stack::pop(int &i,char &c)
{
 while (pcurrent->next!=NULL)
 pcurrent=pcurrent->next;
 i=pcurrent->idata;
 c=pcurrent->cdata;
 return Delete();//用链表类的成员函数实现功能
}
void Stack::show()
{
 Show();//用链表类的成员函数实现功能
}
//main.cpp
#include <iostream>
#include "stack.h"
int main()
{
 Stack ss;
 int i,j;
 char c;
 for (j=0;j<3;j++)
 {
 cout <<"请输入一个数字和一个字母: " <<endl;
 cin >>i >>c;
 if (ss.push(i,c))
 {
 cout <<"压栈成功! " <<endl;
 }
 }
 ss.show();
 while (ss.pop(i,c))
 {
 cout <<"退栈数据为 i=" <<i <<" c=" <<c <<endl;
 }
 return 0;
}
```

运行结果:

Node constructor is running...

Linklist constructor is running...

请输入一个数字和一个字母:

1 a

Node constructor is running...

压栈成功!

请输入一个数字和一个字母:

2 b

Node constructor is running...

压栈成功!

请输入一个数字和一个字母:

3 c

Node constructor is running...

压栈成功!

0 0

1 a

2 b

3 c

Node destructor is running...

退栈数据为 i=3 c=c

Node destructor is running...

退栈数据为 i=2 c=b

Node destructor is running...

退栈数据为 i=1 c=a

Linklist destructor is running...

Node destructor is running...

我们看到, Stack 类私有继承了 Linklist 类之后, 利用 Linklist 的成员函数, 方便地实现了压栈和退栈功能。

试试看:

1、修改程序 17.3.2, 试试看在 main 函数中, 能否利用 Linklist 类的成员函数向栈 ss 的其他位置添加或删除结点。

2、使用了继承之后, 子类中到底还有没有父类的私有成员呢? 试证明。

## 17.4 子类对象的生灭

对象在使用之前, 始终是要经历“构造”这个过程的。在第 15 章, 我们了解到当一个对象的成员数据是另一个对象的时候, 就先运行成员对象的构造函数, 再运行父对象的构造函数。但是继承的出现, 会引入子类的构造函数。这时候, 这些构造函数的运行顺序又是怎样的呢?

### 子类对象的构造

讨论子类对象的构造, 就是在讨论子类对象的生成方式。它是先生成父类对象的成员,

再对其进行扩展呢，还是先生成子类对象的成员，然后再对其进行补充？我们还是修改一下程序 17.3.2，用事实来解决这个问题：（程序 17.4.1）

//node.h 和 linklist.h 同程序 17.3.2

//stack.h

#include "linklist.h"

class Stack:private Linklist//私有继承链表类

{

public:

bool push(int i,char c);

bool pop(int &i,char &c);

void show();

Stack(int i,char c);

Stack();

};

Stack::Stack(int i,char c):Linklist(i,c)//将子类构造函数的参数传递给父类的构造函数

{

cout <<"Stack constructor with parameter is running..." <<endl;

}

Stack::Stack()//子类构造函数

{

cout <<"Stack constructor is running..." <<endl;

}

bool Stack::push(int i,char c)

{

while (pcurrent->next!=NULL)

pcurrent=pcurrent->next;

return Insert(i,c);

}

bool Stack::pop(int &i,char &c)

{

while (pcurrent->next!=NULL)

pcurrent=pcurrent->next;

i=pcurrent->idata;

c=pcurrent->cdata;

return Delete();

}

void Stack::show()

{

Show();

}

//main.cpp

#include <iostream>

#include "stack.h"

int main()



```

{
 Stack ss(1,'4');//调用带参数的构造函数
 cout <<"Stack ss constructed" <<endl;
 ss.show();
 Stack zz; //调用不带参数的构造函数
 cout <<"Stack zz constructed" <<endl;
 zz.show();
 return 0;
}

```

运行结果：

```

Node constructor is running...
Linklist constructor is running...
Stack constructor with parameter is running...
Stack ss constructed
1 4
Node constructor is running...
Linklist constructor is running...
Stack constructor is running...
Stack zz constructed
0 0
Linklist destructor is running...
Node destructor is running...
Linklist destructor is running...
Node destructor is running...

```

这个程序中有三个类，其中 Stack 类是 Linklist 类的子类，Node 类的对象是 Linklist 类的成员数据。根据程序的运行结果，我们可以确定，父类的成员对象仍然是最先构造的，接着是运行父类的构造函数，最后运行子类的构造函数。也就是说子类对象是在父类对象的基础上扩展而成的。

另外，如果我们希望把子类的构造函数的参数传递给父类的构造函数时，可以在子类的构造函数定义中用以下格式调用父类的构造函数：

**子类名::构造函数名(参数表):父类名(参数表)**

如程序 17.4.1 就是用上述方法实现子类和父类的构造函数参数传递。这样的方法不仅使子类对象的初始化变得简单，并且使子类和父类的构造函数分工明确，易于维护。

试试看：

猜想父类、父类成员对象、子类和子类成员对象的构造函数运行顺序，并设法验证。

**结论：顺序依次为父类成员对象、父类、子类成员对象、子类。各成员对象的构造顺序参照它们在类中声明的顺序。**

## 子类对象的析构

在第 15 章中介绍析构函数的时候，我们就说它的运行顺序往往是和构造函数的运行顺序相反的。那么使用了继承之后，是否依然是这样的规律呢？我们继续修改程序 17.4.1，尝试验证我们的猜想。

```
//node.h 和 linklist.h 同程序 17.3.2
//stack.h
#include "linklist.h"
class Stack:private Linklist
{
public:
 bool push(int i,char c);
 bool pop(int &i,char &c);
 void show();
 Stack(int i,char c);
 Stack();
 ~Stack();//析构函数
};
Stack::Stack(int i,char c):Linklist(i,c)
{
 cout <<"Stack constructor with parameter is running..." <<endl;
}
Stack::Stack()
{
 cout <<"Stack constructor is running..." <<endl;
}
Stack::~Stack()
{
 cout <<"Stack destructor is running..." <<endl;
}
bool Stack::push(int i,char c)
{
 while (pcurrent->next!=NULL)
 pcurrent=pcurrent->next;
 return Insert(i,c);
}
bool Stack::pop(int &i,char &c)
{
 while (pcurrent->next!=NULL)
 pcurrent=pcurrent->next;
 i=pcurrent->idata;
 c=pcurrent->cdata;
 return Delete();
}
void Stack::show()
{
 Show();
}
//main.cpp
```

```

#include <iostream>
#include "stack.h"
int main()
{
 Stack zz;
 cout <<"Stack zz constructed" <<endl;
 zz.show();
 return 0;
}

```

运行结果：

```

Node constructor is running...
Linklist constructor is running...
Stack constructor is running...
Stack zz constructed
0 0
Stack destructor is running...
Linklist destructor is running...
Node destructor is running...

```

根据运行结果，我们可以确认：使用了继承之后，析构函数的运行顺序依然恰好与构造函数的运行顺序相反。

## 17.5 继承与对象指针

我们在第 14 章的最后学习了对象指针，并且在编写链表类的过程中已经能熟练地使用它了。现在有了继承之后，我们的心中有了疑问：父类指针能否指向子类对象？子类指针能否指向父类对象？如果那样使用指针，对象的功能是否会受到限制呢？

### 父类指针与子类对象

我们修改程序 17.3.1，用程序的运行结果来解答我们的疑问：

```

//student.h 和 undergraduate.h 同程序 17.3.1
//main.cpp
#include <iostream>
#include "undergraduate.h"
using namespace std;
int main()
{
 Undergraduate s1;//新建一个本科生对象
 Undergraduate *s1p;//新建一个子类的对象指针
 student s2;
 student *s2p;//新建一个父类的对象指针
 s1p=&s2;//这行程序出错了
 s2p=&s1;
 s1.set("Tom",21,178,60);
}

```

```

cout <<s1.sname <<s1.sage <<endl;
s2p->set("Jon",22,185,68);
cout <<s1.sname <<s1.sage <<endl;
s1p->setGPA(2.5);
s2p->setGPA(3.0); //这行程序出错了
return 0;
}

```

编译结果:

```

main.cpp(10) : error C2440: '=' : cannot convert from 'class student *' to 'class Undergraduate *'
main.cpp(17) : error C2039: 'setGPA' : is not a member of 'student'

```

根据编译结果，我们可以看到，在公有继承情况下父类的对象指针指向子类对象是允许的。如 s2p 学生指针指向本科生 s1，因为本科生也是学生；子类的对象指针指向父类是禁止的。如 s1p 本科生指针不能指向学生 s2，因为学生不一定是本科生。

此外，如果我们用父类的对象指针指向子类对象，那么这个指针无法使用子类中扩展出的成员。如 s2p 指针无法设置本科生的绩点，因为使用了学生指针，本科生就变成了学生的身份，学生身份不再有设置绩点的功能。

我们再次修改程序 17.3.1，使得它能够运行：（程序 17.5）

//student.h 和 undergraduate.h 同程序 17.3.1

//main.cpp

```

#include <iostream>
#include "undergraduate.h"
using namespace std;
int main()
{
 Undergraduate s1;
 student s2;
 student *s2p;
 s2p=&s1;
 s1.set("Tom",21,178,60);
 cout <<s1.sname() <<'\t' <<s1.sage() <<endl;
 s2p->set("Jon",22,185,68);
 cout <<s1.sname() <<'\t' <<s1.sage() <<endl;
 return 0;
}

```

运行结果:

Constructing a student without parameter...

Constructing a student without parameter...

Tom 21

Jon 22

试试看:

- 1、根据程序 17.5 改写程序 17.3.2，尝试在私有继承的情况下，父类的对象指针能否指向子类对象？
- 2、父类对象指针可以指向子类对象，那么父类对象的引用可以作为子类对象的别名么？

现在程序能够正常运行了。可见，用 s1 设置本科生信息和用 s2p 指针设置学生信息都是可行的。

## 猜猜它是谁

假设我们为学生类和本科生类都写了一个名为 study 的成员函数。两者的名称相同，参数表相同，实现却不相同。当子类和父类有着两个名字和参数表完全相同的函数时，我们把这个现象称为覆盖（Overlap）。如下面的代码：

```
//student.h
class student//学生类作为父类
{
public:

 void study();
protected:
 char name[10];
 int age;
 int height;
 int weight;
};
.....
void student::study()
{
 cout <<"随便学些什么。" <<endl;
 return;
}
//undergraduate.h
class Undergraduate:public student
{
public:

 void study();
protected:
 double GPA;//本科生绩点
};
.....
void Undergraduate::study()
{
 cout <<"学习高等数学和大学英语。" <<endl;
 return;
}
```

如果有一个本科生对象 s1 和一个学生对象 s2，那么显然 s1.study()会是学习高等数学和大学英语，s2.study()会是随便学些什么。但是，如果有一个学生类的指针 sp，它也能指向本科生对象，这时调用 sp->study()会是怎么样的呢？我们发现，即使它指向一个本科生对象，它也只能“随便学些什么”。这样的结果在情理之中，却并不是我们期望的。我们希望程序

能够“猜”到 sp 指针指向了哪种对象，并且调用各自的 study 成员函数。这个功能如何才能实现？在之后的几节我们会作讲解。

## 17.6 面向对象特点三：多态性

在本章的开头介绍一个 RPG 游戏的时候，我们就说到不同职业的玩家在发动普通攻击和特殊攻击时，有着不同的效果。在编写程序的时候，我们并不知道用户会选择哪种职业的玩家，那么又该如何保证各种攻击效果和用户选择的玩家是对应的呢？

在使用继承的时候，子类必然是在父类的基础上有所改变。如果两者完全相同，这样的继承就失去了意义。同时，不同子类之间具体实现也是有所区别的，否则就出现了一个多余的类。不同的类的同名成员函数有着不同的表现形式，称为多态性。多态性是符合人的认知规律的，即称呼相同，所指不同。比如，学生类及其子类都有学习这个成员函数，但本科生、中学生、小学生的学习内容并不相同；玩家类的子类都有攻击这项技能，但剑士、弓箭手和魔法师的攻击方法不同。

多态性往往只有在使用对象指针或对象引用时才体现出来。编译器在编译程序的时候完全不知道对象指针可能会指向哪种对象（引用也是类似的情况），只有到程序运行了之后才能明确指针访问的成员函数是属于哪个类的。我们把 C++ 的这种功能称为“滞后联编”。多态性是面向对象的一个标志性特点，没有这个特点，就无法称为面向对象。

## 17.7 多态与虚函数

多态能够方便我们编写程序，可以让不同的类与它独特的成员函数一一对应。即使我们只是简单地“称呼”，程序也会很明白我们的心思。那么，多态应该如何实现呢？

### 多态的实现

在 C++ 中，我们把表现多态的一系列成员函数设置为虚函数。虚函数可能在编译阶段并没有被发现需要调用，但它还是整装待发，随时准备接受指针或引用的“召唤”。设置虚函数的方法为：在成员函数的声明最前面加上保留字 **virtual**。注意，不能把 **virtual** 加到成员函数的定义之前，否则会导致编译错误。

下面我们把各种学生的学习都设置为虚函数，了解如何实现多态：（程序 17.7.1）

```
//student.h
#include <iostream>
using namespace std;
class student
{
public:
 student(char *n,int a,int h,int w);
 student();
 void set(char *n,int a,int h,int w);
 char * sname();
 int sage();
 int sheight();
```

```
int sweight();
virtual void study();//把学习设置为虚函数
protected:
 char name[10];
 int age;
 int height;
 int weight;
};
char * student::sname()
{
 return name;
}
int student::sage()
{
 return age;
}
int student::sheight()
{
 return height;
}
int student::sweight()
{
 return weight;
}
void student::set(char *n,int a,int h,int w)
{
 int i;
 for (i=0;n[i]!='\0';i++)
 {
 name[i]=n[i];
 }
 name[i]='\0';
 age=a;
 height=h;
 weight=w;
 return;
}
student::student(char *n,int a,int h,int w)
{
 cout <<"Constructing a student with parameter..." <<endl;
 set(n,a,h,w);
}
student::student()
{
```

```
 cout <<"Constructing a student without parameter..." <<endl;
 }
 void student::study()//成员函数定义处没有 virtual
 {
 cout <<"随便学些什么。" <<endl;
 return;
 }
//undergraduate.h
#include "student.h"
class Undergraduate:public student
{
public:
 double score();
 void setGPA(double g);
 bool isAdult();
 virtual void study();//把学习设置为虚函数
protected:
 double GPA;
};
double Undergraduate::score()
{
 return GPA;
}
void Undergraduate::setGPA(double g)
{
 GPA=g;
 return;
}
bool Undergraduate::isAdult()
{
 return age>=18?true:false;
}
void Undergraduate::study()//成员函数定义处没有 virtual
{
 cout <<"学习高等数学和大学英语。" <<endl;
 return;
}
//pupil.h
class Pupil:public student
{
public:
 virtual void study();//把学习设置为虚函数
};
void Pupil::study()
```



```

{
 cout <<"学习语数外。" <<endl;
 return;
}
//main.cpp
#include <iostream>
#include "undergraduate.h"
#include "pupil.h"
using namespace std;
int main()
{
 Undergraduate s1;
 student s2;
 Pupil s3;
 student *sp=&s1;//sp 指向本科生对象
 s1.set("Tom",21,178,60);
 sp->study();//体现多态性
 sp=&s2; //sp 指向学生对象
 s2.set("Jon",22,185,68);
 sp->study();//体现多态性
 sp=&s3; //sp 指向小学生对象
 s3.set("Mike",8,148,45);
 sp->study();//体现多态性
 return 0;
}

```

运行结果:

Constructing a student without parameter...

Constructing a student without parameter...

Constructing a student without parameter...

学习高等数学和大学英语。

随便学些什么。

学习语数外。

我们看到，将学习设置为虚函数之后，无论对象指针 `sp` 指向哪种学生对象，`sp->study()` 的执行结果总是与对应的类相符合的。多态就通过虚函数实现了。

我们在编写成员函数的时候，可以把尽可能多的成员函数设置为虚函数。这样做可以充分表现多态性，并且也不会给程序带来不良的副作用。

试试看:

把学生类、小学生类和本科生类的三个 `study` 成员函数中的哪个设置为虚函数对实现多态起了决定性作用?

## 无法实现多态的虚函数

使用虚函数可以实现多态，但是如果在使用虚函数的同时再使用重载，就会可能使虚函

数失效。我们修改程序 17.7.1，看看重载会给虚函数带来些什么麻烦：（程序 17.7.2）

```
//student.h
#include <iostream>
using namespace std;
class student
{
public:
 student(char *n,int a,int h,int w);
 student();
 void set(char *n,int a,int h,int w);
 char * sname();
 int sage();
 int sheight();
 int sweight();
 virtual void study(int c=0);//设置为虚函数，带默认参数
protected:
 char name[10];//姓名
 int age;//年龄
 int height;//身高
 int weight;//体重
};
.....
void student::study(int c)
{
 cout <<"随便学些什么。" <<endl;
 return;
}
//undergraduate.h 和 pupil.h 同程序 17.7.1
//main.cpp
#include <iostream>
#include "undergraduate.h"
#include "pupil.h"
using namespace std;
int main()
{
 Undergraduate s1;
 student s2;
 Pupil s3;
 student *sp=&s1;
 s1.set("Tom",21,178,60);
 sp->study(1);//带参数
 sp=&s2;
 s2.set("Jon",22,185,68);
 sp->study();
}
```

```

sp=&s3;
s3.set("Mike",8,148,45);
sp->study();
return 0;
}

```

运行结果:

Constructing a student without parameter...

Constructing a student without parameter...

Constructing a student without parameter...

随便学些什么。

随便学些什么。

随便学些什么。

当学生类的 study 成员函数和本科生类的 study 成员函数参数格式不同时，即使把学生类中的 study 设置为虚函数，编译器也无法找到本科生类中与之完全相同的 study 函数。多态是在程序员没有指定调用父类还是某个子类的成员函数时，电脑根据程序员的要求，揣测并选择最合适的成员函数去执行。但是当成员函数的参数格式不同时，程序员在调用成员函数的各种参数无疑就是在暗示到底调用哪个成员函数。这时电脑岂敢自作主张揣测人类的心思？因此，要使用虚函数实现多态性，至少要使各个函数的参数格式也完全相同。

试试看:

- 1、如果父类和子类的同名成员函数的参数格式相同，返回值类型不同，并把父类的成员函数设置为虚函数，能否实现多态？
- 2、如果父类和子类的同名成员函数的参数格式相同，父类返回值类型为父类的对象指针（或对象引用），子类返回值类型为子类的对象指针（或对象引用），并把父类的成员函数设置为虚函数，能否实现多态？

## 17.8 虚函数与虚析构函数

在类中，有两个与众不同的成员函数，那就是构造函数和析构函数。当构造函数与析构函数遭遇继承和多态，它们的运行状况又会出现什么变化呢？

多态性是在父类或各子类中执行最合适成员函数。一般来说，只会选择父类或子类中的某一个成员函数来执行。这可给析构函数带来了麻烦！如果有的资源是父类的构造函数申请的，有的资源是子类的构造函数申请的，而虚函数只允许程序执行父类或子类中的某一个析构函数，岂不是注定有一部分资源将无法被释放？为了解决这个问题，虚析构函数变得与众不同。

下面我们就来给析构函数的前面加上保留字 virtual，看看运行的结果会怎么样：（程序 17.8）

```

//animal.h
#include <iostream>
using namespace std;
class Animal
{
public:

```

```

 Animal(int w=0,int a=0);
 virtual ~Animal();//虚析构函数
protected:
 int weight,age;
};
Animal::Animal(int w,int a)
{
 cout <<"Animal consturctor is running..." <<endl;
 weight=w;
 age=a;
}
Animal::~~Animal()
{
 cout <<"Animal destructor is running..." <<endl;
}
//cat.h
#include "animal.h"
class Cat:public Animal
{
public:
 Cat(int w=0,int a=0);
 ~Cat();
};
Cat::Cat(int w,int a):Animal(w,a)
{
 cout <<"Cat constructor is running..." <<endl;
}
Cat::~~Cat()
{
 cout <<"Cat destructor is running..." <<endl;
}
//main.cpp
#include "cat.h"
int main()
{
 Animal *pa=new Cat(2,1);
 Cat *pc=new Cat(2,4);
 cout <<"Delete pa:" <<endl;
 delete pa;
 cout <<"Delete pc:" <<endl;
 delete pc;
 return 0;
}

```

运行结果:

```
Animal constructor is running...
Cat constructor is running...
Animal constructor is running...
Cat constructor is running...
Delete pa:
Cat destructor is running...
Animal destructor is running...
Delete pc:
Cat destructor is running...
Animal destructor is running...
```

我们惊讶地发现，虚析构函数不再是运行父类或子类的某一个析构函数，而是先运行合适的子类析构函数，再运行父类析构函数。即两个类的析构函数都被执行了，如果两块资源分别是由父类构造函数和子类构造函数申请的，那么使用了虚析构函数之后，两块资源都能被及时释放。

我们修改程序 17.8，将 `Animal` 类析构函数前的 `virtual` 去掉，会发现运行结果中删除 `pa` 指向的 `Cat` 对象时，不执行 `Cat` 类的析构函数。如果这时 `Cat` 类的构造函数里申请了内存资源，就会造成内存泄漏了。

所以说，虚函数与虚析构函数的作用是不同的。虚函数是为了实现多态，而虚析构函数是为了同时运行父类和子类的析构函数，使资源得以释放。

试试看：  
构造函数能否设置为虚函数？  
结论：没有虚构造函数。

## 17.9 抽象类与纯虚函数

在本章开头介绍的 RPG 游戏中共有 4 个类。其中玩家类作为父类，剑士类、弓箭手类、魔法师类分别继承玩家类，作为子类。当我们开始游戏时，需要选择创建某一个类的对象，才能进行游戏。然而，我们的选择不应该是 4 个类，而应该只能在剑士类、弓箭手类或魔法师类中做出选择。因为，单纯的玩家类是抽象的、不完整的，任何一个玩家必须有一个确定的职业之后，才能确定他的具体技能。又比如学生类，它也是非常抽象的。让一个小学生、中学生或本科生学习，他们都有各自学习的内容。而一个抽象概念的学生，他却不知道该学些什么。

这时，我们必须要对玩家类或学生类作一些限制了。由于玩家类和学生类直接实例化而创建的对象都是抽象而没有意义的，所以我们希望玩家类和学生类只能用于被继承，而不能用于直接创建对象。在 C++ 中，我们可以把只能用于被继承而不能直接创建对象的类设置为抽象类（Abstract Class）。

之所以要存在抽象类，最主要是因为它具有不确定因素。我们把那些类中的确存在，但是在父类中无法确定具体实现的成员函数称为纯虚函数。纯虚函数是一种特殊的虚函数，它只有声明，没有具体的定义。抽象类中至少存在一个纯虚函数；存在纯虚函数的类一定是抽象类。存在纯虚函数是成为抽象类的充要条件。

那么我们应该如何定义一个纯虚函数呢？纯虚函数的声明有着特殊的语法格式：

**virtual** 返回值类型 成员函数名（参数表）=0;

请注意，纯虚函数应该只有声明，没有具体的定义，即使给出了纯虚函数的定义也会被编译器忽略。下面我们就修改一下程序 17.7.1，将学生类变成一个抽象类：（程序 17.9）

```
//student.h
#include <iostream>
using namespace std;
class student//因为存在纯虚函数 study， student 类自动变成了抽象类
{
public:
 student(char *n,int a,int h,int w);
 student();
 void set(char *n,int a,int h,int w);
 char * sname();
 int sage();
 int sheight();
 int sweight();
 virtual void study()=0;//声明 study 为纯虚函数
protected:
 char name[10];
 int age;
 int height;
 int weight;
};
char * student::sname()
{
 return name;
}
int student::sage()
{
 return age;
}
int student::sheight()
{
 return height;
}
int student::sweight()
{
 return weight;
}
void student::set(char *n,int a,int h,int w)
{
 int i;
 for (i=0;n[i]!='\0';i++)
 {
 name[i]=n[i];
 }
}
```

```

 }
 name[i]='\0';
 age=a;
 height=h;
 weight=w;
 return;
}
student::student(char *n,int a,int h,int w)
{
 cout <<"Constructing a student with parameter..." <<endl;
 set(n,a,h,w);
}
student::student()
{
 cout <<"Constructing a student without parameter..." <<endl;
}

```

//undergraduate.h 和 pupil.h 同程序 17.7.1

//main.cpp

```

#include <iostream>
#include "undergraduate.h"
#include "pupil.h"
using namespace std;
int main()
{
 Undergraduate s1;
 /*student s2; //此时创建学生对象将会出现编译错误*/
 Pupil s3;
 student *sp=&s1;
 s1.set("Tom",21,178,60);
 sp->study();
 sp=&s3;
 s3.set("Mike",8,148,45);
 sp->study();
 return 0;
}

```

运行结果:

Constructing a student without parameter...

Constructing a student without parameter...

学习高等数学和大学英语。

学习语数外。

我们看到，设置了纯虚函数之后并不影响多态的实现，但是却将父类变成了抽象类，限制了父类对象的创建。有了抽象类之后，就不会再出现不确定职业的玩家、不确定身份的学生了。

试试看：

1、如果有一个类继承了学生类，但是却没有定义 `study` 成员函数，那么这个类是不是抽象类？它能否用来创建一个对象？

2、如果去掉学生类中的纯虚函数 `study`，程序 17.9 能否实现多态？

结论：纯虚函数的存在有利于多态的实现。

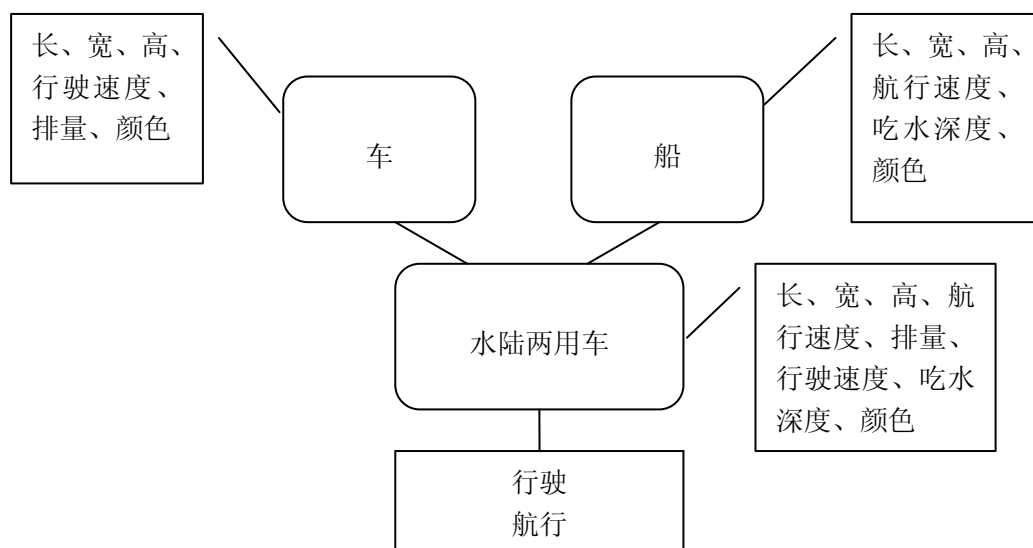
## 17.10 多重继承

2002 年，河北省的一位造车高手奇思妙想设计出了一辆水陆两用型的小跑车。这辆车既可以在公路上奔驰，也可以在水波中荡漾，它同时具有车和船的特性。（如图 17.10.1）



（图 17.10.1 引自新华网）

如果我们用继承的概念来分析水陆两用车，那么它的确是存在继承关系的。与一般的继承不同的是，水陆两用车的父类会有两个：一个是车，一个是船。（如下图 17.10.2）



（图 17.10.2）

C++ 中的确能实现多继承，但是在某些问题上并不是处理得很好。比如车和船同时具有长、宽、高等属性，要搞清水陆两用车的长、宽、高到底是从哪个类继承来的，着实要花费一些功夫。应该说，C++ 中多重继承的思想是优秀的，但是它的实现却是混乱的。有不少人



都认为多重继承是 C++ 的一个败笔，它把原本简单的单一继承复杂化了，使程序员很难再把思路理清。所以，即使是经验丰富的程序员，在大多数情况下也不会去使用多重继承。在此，我们只要能够理解多重继承的概念即可，不必去掌握它的具体实现。有兴趣的读者也可以到网上查找相关的资料。

## 习题

1、阅读一下程序，判断各类的成员数据和成员函数可被访问的情况。

①

```
class a
{
public:
 a();
 ~a();
 int b;
 char c();
protected:
 int d;
 double e;
private:
 char f;
};
class b:public a
{
protected:
 double g;
 int h();
};
```

| 成员名 | 类外部 | 子类 | 本类 |
|-----|-----|----|----|
| b   |     |    |    |
| c() |     |    |    |
| d   |     |    |    |
| e   |     |    |    |
| f   |     |    |    |
| g   |     | -  |    |
| h() |     | -  |    |

②

```
class a
{
public:
 a();
 ~a();
 int b;
```

```

 char c();
protected:
 int d;
 double e;
private:
 char f;
};
class b:private a
{
protected:
 double g;
 int h();
};

```

| 成员名 | 类外部 | 子类 | 本类 |
|-----|-----|----|----|
| b   |     |    |    |
| c() |     |    |    |
| d   |     |    |    |
| e   |     |    |    |
| f   |     |    |    |
| g   |     | -  |    |
| h() |     | -  |    |

2、阅读下列程序，写出运行结果。

```

//animal.h
#include <iostream>
using namespace std;
class Animal
{
public:
 Animal(int w=0,int a=0);
 ~Animal();
 void print();
 virtual void show();
protected:
 int weight,age;
};
Animal::Animal(int w,int a)
{
 cout <<"Animal consturctor is running..." <<endl;
 weight=w;
 age=a;
}
Animal::~~Animal()
{
 cout <<"Animal destructor is running..." <<endl;
}

```

```
}
void Animal::print()
{
 cout <<"Animal Print" <<endl;
}
void Animal::show()
{
 cout <<"Animal Show" <<endl;
}
//cat.h
#include "animal.h"
class Cat:public Animal
{
public:
 void print();
 virtual void show();
 Cat(int w=0,int a=0);
 virtual ~Cat();
};
Cat::Cat(int w,int a):Animal(w,a)
{
 cout <<"Cat constructor is running..." <<endl;
}
Cat::~~Cat()
{
 cout <<"Cat destructor is running..." <<endl;
}
void Cat::print()
{
 cout <<"Cat Print" <<endl;
}
void Cat::show()
{
 cout <<"Cat Show" <<endl;
}
//persiancat.h
#include "cat.h"
class PersianCat:public Cat
{
public:
 void print();
 void show();
 PersianCat(int w=0,int a=0);
 ~PersianCat();
};
```

```

};
PersianCat::PersianCat(int w,int a):Cat(w,a)
{
 cout <<"PersianCat constructor is running..." <<endl;
}
PersianCat::~~PersianCat()
{
 cout <<"PersianCat destructor is running..." <<endl;
}
void PersianCat::print()
{
 cout <<"PersianCat Print" <<endl;
}
void PersianCat::show()
{
 cout <<"PersianCat Show" <<endl;
}
//main.cpp
#include "persiancat.h"
int main()
{
 Animal *pa=new Cat(2,4);
 Cat *pc=new PersianCat(4,6);
 pa->show();
 pa->print();
 pc->show();
 pc->print();
 cout <<"delete pa:" <<endl;
 delete pa;
 cout <<"delete pc:" <<endl;
 delete pc;
 return 0;
}

```

3、根据本章介绍的 RPG 游戏思想，完善以下代码，使之成为一个可以运行的游戏。

```

//container.h
class container//人物的物品栏
{
protected:
 int numOfHeal;//回复剂数
 int numOfMagicWater;//魔法水数
public:
 container();//构造函数
 void set(int i,int j);//重设物品数
 int nOfHeal();//判断是否还有回复剂
}

```

```
int nOfMW();//判断是否还有魔法水
void display();//显示物品情况
bool useHeal();//使用回复剂
bool useMW();//使用魔法水
};
container::container()
{
 set(0,0);
}
void container::set(int i,int j)
{
 numOfHeal=i;
 numOfMagicWater=j;
}
int container::nOfHeal()
{
 return numOfHeal;
}
int container::nOfMW()
{
 return numOfMagicWater;
}
void container::display()
{
 cout <<"回复剂（生命值+100）还有" <<numOfHeal <<"个" <<endl;
 cout <<"魔法水（魔法值+ 80）还有" <<numOfMagicWater <<"个" <<endl;
}
bool container::useHeal()
{
 numOfHeal--;
 return 1;
}
bool container::useMW()
{
 numOfMagicWater--;
 return 1;
}
//player.h
#include <iostream.h>
#include <iomanip.h>//用于设置域宽
#include <stdlib.h>
#include <time.h>//用于产生随机因子
#include "container.h"
enum property {sw,ar,mg};//设置三种职业属性，枚举类型
```

```

class Player//人物类
{
 friend class Swordsman;
 friend class Archer;
 friend class Mage;
 :
 int HP,HPmax,MP,MPmax,AP,DP,speed,EXP,LV;//一般属性
 char name[10];//人物名称
 property role;//人物职业（角色）
 container bag;//物品栏
public:
 //普通攻击
 //特殊攻击
 //升级设定
 void ReFill();//每版结束后人类玩家生命值魔法值恢复
 bool Death();//告知是否人物已经死亡
 void isDead();//判断人物是否死亡
 bool useHeal();//使用回复剂，与职业无关
 bool useMW();//使用魔法水，与职业无关
 void transfer(Player &p);//敌人杀死后，人类玩家获得对方物品
 char showRole();//显示人物的职业
 :
 bool death;//是否死亡
};
void Player::ReFill()
{
 HP=HPmax;
 MP=MPmax;
}
bool Player::Death()
{
 return death;
}
void Player::isDead()
{
 if (HP<=0)
 {
 cout <<name <<"已阵亡。" <<endl;
 system("pause");
 death=1;
 }
}
bool Player::useHeal()

```

```
{
 if (bag.nOfHeal(>0)
 {
 HP=HP+100;
 if (HP>HPmax)
 HP=HPmax;
 cout <<name <<"使用了回复剂， 生命值增加了 100。" <<endl;
 bag.useHeal();
 system("pause");
 return 1;//使用回复剂操作成功
 }
 cout <<"对不起， 您没有回复剂可以使用了。" <<endl;
 system("pause");
 return 0;//使用回复剂操作失败
}
bool Player::useMW()
{
 if (bag.nOfMW(>0)
 {
 MP=MP+80;
 if (MP>MPmax)
 MP=MPmax;
 cout <<name <<"使用了魔法水， 魔法值增加了 80。" <<endl;
 bag.useMW();
 system("pause");
 return 1;
 }
 cout <<"对不起， 您没有魔法水可以使用了。" <<endl;
 system("pause");
 return 0;
}
void Player::transfer(Player &p)
{
 cout <<name <<"获得了" <<p.bag.nOfHeal() <<"个回复剂和" <<p.bag.nOfMW() <<"个魔法水。" <<endl;
 system("pause");
 bag.set(bag.nOfHeal()+p.bag.nOfHeal(),bag.nOfMW()+p.bag.nOfMW());
}
char Player::showRole()
{
 switch(role)
 {
 case sw:
 cout <<setw(6) <<"剑士";
```

```

 break;
 case ar:
 cout <<setw(6) <<"弓箭手";
 break;
 case mg:
 cout <<setw(6) <<"法师";
 break;
 default:
 break;
 }
 return ' ';
}
void showinfo(Player &p1,Player &p2)//纯粹的输出问题
{
 cout <<"#####" <<endl;
 cout <<"# 玩家 " <<setw(11) <<p1.name <<" LV." <<setw(3) <<p1.LV <<"# 敌人 "
<<setw(11) <<p2.name <<" LV." <<setw(3) <<p2.LV <<'#' <<endl;
 cout <<"# HP " <<setw(3) <<(p1.HP<=999?p1.HP:999) <<'/' <<setw(3)
<<(p1.HPmax<=999?p1.HPmax:999) <<" MP " <<setw(3) <<(p1.MP<=999?p1.MP:999) <<'/'
<<setw(3) <<(p1.MPmax<=999?p1.MPmax:999) <<" # HP " <<setw(3)
<<(p2.HP<=999?p2.HP:999) <<'/' <<setw(3) <<(p2.HPmax<=999?p2.HPmax:999) <<" MP "
<<setw(3) <<(p2.MP<=999?p2.MP:999) <<'/' <<setw(3) <<(p2.MPmax<=999?p2.MPmax:999)
<<" #" <<endl;
 cout <<"# AP " <<setw(3) <<(p1.AP<=999?p1.AP:999) <<" DP " <<setw(3)
<<(p1.DP<=999?p1.DP:999) <<" Speed " <<setw(3) <<(p1.speed<=999?p1.speed:999) <<"# AP "
<<setw(3) <<(p2.AP<=999?p2.AP:999) <<" DP " <<setw(3) <<(p2.DP<=999?p2.DP:999) <<"
Speed " <<setw(3) <<(p2.speed<=999?p2.speed:999) <<'#' <<endl;
 cout <<"# EXP" <<setw(7) <<p1.EXP <<" 职业 ";
 cout <<p1.showRole();
 cout <<"# EXP" <<setw(7) <<p2.EXP <<" 职业 ";
 cout <<p2.showRole() <<'#' <<endl;
 cout <<"#####" <<endl;
 p1.bag.display();
}
//swordsman.h
#include "player.h"
class Swordsman:_____//剑士类
{
public:
 Swordsman(int i,char *cptr);//构造函数
 void isLevelUp();
 bool attack(Player &p);
 bool specialatt(Player &p);
 void AI(Player &p);//电脑 AI

```



```

};
Swordsman::Swordsman(int i,char *cptr)
{
 role=sw;
 for (int j=0;j<10;j++)
 name[j]=cptr[j];
 HP=150+8*(i-1);
 HPmax=150+8*(i-1);
 MP=75+2*(i-1);
 MPmax=75+2*(i-1);
 AP=25+4*(i-1);
 DP=25+4*(i-1);
 speed=25+2*(i-1);
 LV=i;
 death=0;
 EXP=LV*LV*75;
 bag.set(i*5,i*5);
}
void Swordsman::isLevelUp()
{
 if (EXP>=LV*LV*75)
 {
 LV++;
 AP=AP+4;
 DP=DP+4;
 HPmax=HPmax+8;
 MPmax=MPmax+2;
 speed=speed+2;
 cout <<name <<"升级了！ " <<endl;
 cout <<"生命值增加了" <<8 <<"点" <<endl;
 cout <<"魔法值增加了" <<2 <<"点" <<endl;
 cout <<"速度增加了" <<2 <<"点" <<endl;
 cout <<"攻击力增加了" <<4 <<"点" <<endl;
 cout <<"防御力增加了" <<4 <<"点" <<endl;
 isLevelUp();//递归调用，如果一次获得经验值太多可能发生升多级的可能
 }
}
bool Swordsman::attack(Player &p)
{
 double HPtemp=0,EXPtemp=0;//敌方伤血和玩家经验值
 double hit=1;//攻击因子，可能产生重击
 srand(time(NULL));//产生随机因子
 if ((speed>p.speed) && (rand()%100<(speed-p.speed)))//两次攻击
 {

```

```

 HPtemp=int((1.0*AP/p.DP)*AP*5/(rand()%4+10));
 cout <<name <<"先发制人，飞剑出鞘，击中" <<p.name <<"的脑门，" <<p.name <<"
生命值减少了" <<HPtemp <<endl;
 p.HP=int(p.HP-HPtemp);
 EXPtemp=int(HPtemp*1.2);
 }
 if((speed<=p.speed) && (rand()%50<1))//敌方躲避
 {
 cout <<name <<"拔出利剑，一阵乱砍，" <<p.name <<"顺势躲开了。" <<endl;
 system("pause");
 return 1;
 }
 if (rand()%100<=10)//10%概率发生重击
 {
 hit=1.5;//攻击因子变为 1.5
 cout <<name <<"力量剧增，发出会心一击。";
 }
 HPtemp=int(hit*(1.0*AP/p.DP)*AP*30/(rand()%8+32));
 cout <<name <<"猛然挥剑，一道银光掠过眼前，" <<p.name <<"被一道剑气穿膛而过，
生命值减少了" <<HPtemp <<endl;
 EXPtemp=int(EXPtemp+HPtemp*1.2);//没有 int 可能会有警告
 p.HP=int(p.HP-HPtemp);
 cout <<name <<"获得了" <<EXPtemp <<"点经验值。" <<endl;
 EXP=int(EXP+EXPtemp);
 system("pause");
 return 1;//攻击操作成功
}
bool Swordsman::specialatt(Player &p)
{
 if (MP<40)
 {
 cout <<"您的魔法值不够！" <<endl;
 system("pause");
 return 0;//魔法不够，特殊攻击操作失败
 }
 else
 {
 MP=MP-40;
 if (rand()%100<=10)//10%敌方躲避
 {
 cout <<name <<"刚刚使出重剑斩，" <<p.name <<"就来了个鹞子翻身，跳开了。
" <<endl;
 system("pause");
 return 1;
 }
 }
}

```

```

 }
 double HPtemp=0,EXPtemp=0;
 double hit=1;
 srand(time(NULL));
 HPtemp=int(AP*1.2+20);
 EXPtemp=int(HPtemp*1.5);
 cout <<name <<"使出浑身解数，大叫一声“重~~~~剑~~~~斩~~~~!”拿起宝剑就向
" <<p.name <<"砸去，顿时鲜血四溅……" <<endl;
 cout <<p.name <<"生命值减少了" <<HPtemp <<"，" <<name <<"获得了"
<<EXPtemp <<"点经验值。" <<endl;
 p.HP=int(p.HP-HPtemp);
 EXP=int(EXP+EXPtemp);
 system("pause");
}
return 1;//特殊攻击操作成功
}
void Swordsman::AI(Player &p)//电脑 AI
{
 if((HP<=int((1.0*p.AP/DP)*p.AP*1.5)) && (HP+100<=1.1*HPmax) && (bag.nOfHeal())>0)
 && (p.HP>int((1.0*p.AP/DP)*p.AP*0.6))//电脑玩家的生命值经不起人类玩家打击且使用回
 复剂不算太浪费且有回复剂可以使用且人类玩家不可能被电脑玩家直接杀死
 {
 useHeal();//则使用回复剂
 }
 else
 {
 if (MP>=40 && HP>0.5*HPmax && rand()%10>7)//魔法值足够且电脑的 HP 足够且
 30%概率
 {
 specialatt(p);//使用特殊攻击
 p.isDead();//判断人类玩家是否死亡了
 }
 else
 {
 if (MP<40 && HP>0.5*HPmax && !bag.nOfMW())//魔法值不够且电脑的 HP 足
 够安全
 {
 useMW();//使用魔法水
 }
 else//其他情况
 {
 attack(p);//使用普通攻击
 p.isDead();//判断人类玩家是否死亡了
 }
 }
 }
}

```

```

 }
}
}
//archer.h
class Archer:_____//弓箭手类和剑士类差不多，就是数值设定不同
{
public:
 Archer(int i,char *cptr);
 void isLevelUp();
 bool attack(Player &p);
 bool specialatt(Player &p);
};
Archer::Archer(int i,char *cptr)
{
 role=ar;
 for (int j=0;j<10;j++)
 name[j]=cptr[j];
 HP=180+6*(i-1);
 HPmax=180+6*(i-1);
 MP=75+8*(i-1);
 MPmax=75+8*(i-1);
 AP=30+3*(i-1);
 DP=24+3*(i-1);
 speed=30+10*(i-1);
 LV=i;
 death=0;
 EXP=(LV-1)*(LV-1)*70;
 bag.set(i*8,i*8);
}
void Archer::isLevelUp()
{
 if (EXP>=LV*LV*70)
 {
 LV++;
 AP=AP+3;
 DP=DP+3;
 HPmax=HPmax+6;
 MPmax=MPmax+8;
 speed=speed+10;
 cout <<name <<"升级了！ " <<endl;
 cout <<"生命值增加了" <<6 <<"点" <<endl;
 cout <<"魔法值增加了" <<8 <<"点" <<endl;
 cout <<"速度增加了" <<10 <<"点" <<endl;
 cout <<"攻击力增加了" <<3 <<"点" <<endl;
 }
}

```

```

 cout <<"防御力增加了" <<3 <<"点" <<endl;
 isLevelUp();
 }
}
bool Archer::attack(Player &p)
{
 double HPtemp=0,EXPtemp=0;
 double hit=1;
 srand(time(NULL));
 if ((speed>p.speed) && (rand()%100<(speed-p.speed))//Double Hit
 {
 HPtemp=int((1.0*AP/p.DP)*AP*2/(rand()%4+10));
 cout <<name <<"先暗放一箭，不偏不倚正好打在" <<p.name <<"的胸口，" <<p.name
<<"生命值减少了" <<HPtemp <<endl;
 p.HP=int(p.HP-HPtemp);
 EXPtemp=int(HPtemp*1.2);
 }
 if (rand()%100<1)
 {
 cout <<name <<"射出一支歪歪扭扭的箭，" <<p.name <<"很轻松地避开了。" <<endl;
 system("pause");
 return 1;
 }
 if (rand()%100<=10)
 {
 hit=1.5;
 cout <<name <<"拉足了弓，发出会心一击。";
 }
 HPtemp=int(hit*(1.0*AP/p.DP)*AP*30/(rand()%8+32));
 cout <<name <<"射出一支长箭，“嗖”地一声，插入了" <<p.name <<"的"
<<(rand()%2==1?"胸膛，":"大腿，") <<p.name <<"生命值减少了" <<HPtemp <<endl;
 EXPtemp=int(EXPtemp+HPtemp*1.2);
 p.HP=int(p.HP-HPtemp);
 cout <<name <<"获得了" <<EXPtemp <<"点经验值。" <<endl;
 EXP=int(EXP+EXPtemp);
 system("pause");
 return 1;
}
bool Archer::specialatt(Player &p)
{
 if (MP<40)
 {
 cout <<"您的魔法值不够！" <<endl;
 system("pause");
 }
}

```

```

 return 0;
 }
 else
 {
 MP=MP-40;
 double HPtemp=0,EXPtemp=0;
 srand(time(NULL));
 HPtemp=int(AP*1.4+18);
 EXPtemp=int(HPtemp*1.5);
 cout <<name <<"拿出三把长箭，大叫一声“流~~~~星~~~~箭~~~~!” 三把长箭径直
向" <<p.name <<"飞去， " <<p.name <<"无处闪躲……" <<endl;
 cout <<p.name <<"生命值减少了" <<HPtemp <<"， " <<name <<"获得了"
<<EXPtemp <<"点经验值。" <<endl;
 p.HP=int(p.HP-HPtemp);
 EXP=int(EXP+EXPtemp);
 system("pause");
 }
 return 1;
}
//mage.h
class Mage:_____//法师类和剑士类差不多，就是数值设定不同
{
public:
 Mage(int i,char *cptr);
 void isLevelUp();
 bool attack(Player &p);
 bool specialatt(Player &p);
};
Mage::Mage(int i,char *cptr)
{
 role=mg;
 for (int j=0;j<10;j++)
 name[j]=cptr[j];
 HP=120+4*(i-1);
 HPmax=120+4*(i-1);
 MP=200+20*(i-1);
 MPmax=200+20*(i-1);
 AP=50+2*(i-1);
 DP=20+2*(i-1);
 speed=25+3*(i-1);
 LV=i;
 death=0;
 EXP=(LV-1)*(LV-1)*65;
 bag.set(i*5,i*30);
}

```

```

}
void Mage::isLevelUp()
{
 if (EXP>=LV*LV*65)
 {
 LV++;
 AP=AP+2;
 DP=DP+2;
 HPmax=HPmax+4;
 MPmax=MPmax+20;
 speed=speed+3;
 cout <<name <<"升级了! " <<endl;
 cout <<"生命值增加了" <<4 <<"点" <<endl;
 cout <<"魔法值增加了" <<20 <<"点" <<endl;
 cout <<"速度增加了" <<3 <<"点" <<endl;
 cout <<"攻击力增加了" <<2 <<"点" <<endl;
 cout <<"防御力增加了" <<2 <<"点" <<endl;
 isLevelUp();
 }
}
bool Mage::attack(Player &p)
{
 double HPtemp=0,EXPtemp=0;
 double hit=1;
 srand(time(NULL));
 if ((speed>p.speed) && (rand()%100<(speed-p.speed))//Double Hit
 {
 HPtemp=int((1.0*AP/p.DP)*AP*3/(rand()%4+10));
 cout <<name <<"先是一记飞腿,踢向" <<p.name <<"的小腹, " <<p.name <<"生命值
减少了" <<HPtemp <<endl;
 p.HP=int(p.HP-HPtemp);
 EXPtemp=int(HPtemp*1.2);
 }
 if (rand()%100<1)
 {
 cout <<name <<"默念咒语, " <<p.name <<"很轻松地避开了。" <<endl;
 system("pause");
 return 1;
 }
 if (rand()%100<=10)
 {
 hit=1.5;
 cout <<name <<"用足力气, 发出会心一击。";
 }
}

```

```

 HPtemp=int(hit*(1.0*AP/p.DP)*AP*25/(rand()%8+35));
 cout <<name <<"抡起法仗,砸向" <<p.name <<"的" <<(rand()%2==1?"脑袋, ":"肩膀, ")
<<p.name <<"生命值减少了" <<HPtemp <<endl;
 EXPtemp=int(EXPtemp+HPtemp*1.2);
 p.HP=int(p.HP-HPtemp);
 cout <<name <<"获得了" <<EXPtemp <<"点经验值。" <<endl;
 EXP=int(EXP+EXPtemp);
 system("pause");
 return 1;
}
bool Mage::specialatt(Player &p)
{
 if (MP<20)
 {
 cout <<"您的魔法值不够! " <<endl;
 system("pause");
 return 0;
 }
 else
 {
 MP=MP-20;
 if (rand()%100<=20)
 {
 cout <<name <<"默念 “&^%@$*……”，射出一道绿光，" <<p.name <<"逃得
远远的。" <<endl;
 system("pause");
 return 1;
 }
 double HPtemp=0,EXPtemp=0;
 srand(time(NULL));
 HPtemp=int(AP*1.1);
 EXPtemp=int(HPtemp*1.3);
 cout <<name <<"默念 “&^%@$*……”，一道绿光射向" <<p.name <<"，" <<name
<<"吸取了" <<p.name <<"的" <<HPtemp <<"点生命值。" <<endl;
 cout <<name <<"获得了" <<EXPtemp <<"点经验值。" <<endl;
 p.HP=int(p.HP-HPtemp);
 EXP=int(EXP+EXPtemp);
 if (HP+HPtemp<=HPmax)
 HP=int(HP+HPtemp);
 else
 HP=HPmax;
 system("pause");
 }
 return 1;
}

```



```
}
//main.cpp
#include <iostream.h>
#include "swordsman.h"
#include "archer.h"
#include "mage.h"
void main()
{
 char temp[10];
 bool success=0;//操作是否成功
 cout <<"请输入玩家名字: ";
 cin >>temp;
 _____//方便实现多态
 int instemp;//存放指令数
 do
 {
 cout <<"请选择职业: 1 剑士 2 弓箭手 3 法师" <<endl;
 cin >>instemp;
 system("cls");
 switch (instemp)//选择职业
 {
 case 1://选择了剑士
 human=new Swordsman(1,temp);
 success=1;//操作成功
 break;
 case 2:
 human=new Archer(1,temp);
 success=1;
 break;
 case 3:
 human=new Mage(1,temp);
 success=1;
 break;
 default:
 break;
 }
 }while (success!=1);//循环选择直到操作成功
 int j=0;//第几版
 for (int i=1;j<5;i=i+2)
 {
 j++;
 system("cls");
 cout <<"STAGE " <<j <<endl;
 cout <<"敌方介绍: 一个" <<i <<"级的剑士。" <<endl;
 }
}
```

```

system("pause");
Swordsman enemy(i,"敌方士兵");//创建一个 i 级的剑士作为敌人
human->ReFill();//人类玩家每过一版生命魔法值恢复
while (!human->Death() && !enemy.Death())//两个人都没死则继续战斗
{
 success=0;
 while (success!=1)//直到操作成功
 {
 system("cls");
 showinfo(*human,enemy);//显示两个玩家信息
 cout <<"请下达指令: " <<endl;
 cout <<"1 攻击 2 特殊攻击 3 使用回复剂 4 使用魔法水 0 退出游戏 "
<<endl;

 cin >>instemp;
 switch (instemp)
 {
 case 0:
 cout <<"是否要退出游戏? Y/N" <<endl;
 char temp;
 cin >>temp;
 if (temp=='Y' || temp=='y')
 {
 exit(0);
 }
 else
 break;
 case 1:
 success=human->attack(enemy);
 human->isLevelUp();
 enemy.isDead();
 break;
 case 2:
 success=human->specialatt(enemy);
 human->isLevelUp();
 enemy.isDead();
 break;
 case 3:
 success=human->useHeal();
 break;
 case 4:
 success=human->useMW();
 break;
 default:
 break;
 }
 }
}

```

```
 }
 }
 if (!enemy.Death())//如果电脑玩家没有死亡
 {
 enemy.AI(*human);//与人类对战
 }
 else
 {
 human->transfer(enemy);//把物品给人类玩家
 }
 if (human->Death())//如果人类玩家死亡
 {
 system("cls");
 cout <<endl <<endl <<endl <<endl <<endl <<setw(50) <<"胜败乃兵家常事，
好男儿请重新再来。" <<endl;

 system("pause");
 exit(0);
 }
}
}
system("cls");
cout <<endl <<endl <<endl <<endl <<endl <<setw(60) <<"所有的敌人都已经被您消灭了！
世界又恢复了往日的和平。" <<endl <<endl <<endl <<setw(35) <<"终" <<endl <<endl <<endl
<<endl <<endl;

system("pause");
}
```

## 第十八章 再谈输入与输出