教育部全国普通高等学校优秀教材奖

普通高等教育“十一五”国家级规划教材

C++程序设计系列教材

C++程序设计教程(修订版)

——设计思想与实现

钱能 著

清华大学出版社

本书特点：

1. 从C++的特色写起，以类型定义、识别、匹配为安全中心，注重编程质量
2. 从自身学习经历写起，将经验与体会作为链接知识与能力的纽带
3. 从初学角度写起，点透要害，巧作取舍
4. 打破文字描述的框框，将概念理解与万事万物联系起来
5. 代码风格独特，实验上手较快，不需要复杂的调试

C++程序设计系列教材

C++程序设计教程(修订版)
——设计思想与实现

C++程序设计习题及解答（已出版）

C++程序设计教程(第二版)（已出版）

C++程序设计教程(第二版)实验指导（已出版）

C++程序设计教程精粹

C++程序设计教程(第二版)习题及解答

C++程序设计教程详解(上)

C++程序设计教程详解(下)

ISBN 978-7-302-20185-4

教育部全国普通高等学校优秀教材奖

普通高等教育“十一五”国家级规划教材

C++程序设计系列教材

C++程序设计教程(修订版)
——设计思想与实现

钱能 著

清华大学出版社

内容简介

C++是一种高效实用的程序设计语言，它既可进行过程化程序设计，也可进行面向对象程序设计，因而成为编程人员最广泛使用的工具。学好C++，很容易触类旁通其他软件，C++架起了通向强大、易用、真正的软件开发应用的桥梁。许多高等院校已经开设了C++程序设计语言课，急需一本实用的教材。本书是作者总结两年教学实践的经验写成的，适合用作大学计算机专业和非计算机专业的程序设计基础课程教材，也可供自学的读者使用。

本书共分两大部分。第一部分，第1章至第10章是基础部分，主要介绍C++程序设计语言、程序结构和过程化基础。第二部分，第11章至第21章，是面向对象程序设计部分，它建立在C++程序设计基础之上，讲述了面向对象程序设计方法。

图书在版编目(CIP)数据

C++程序设计教程(修订版)：设计思想与实现/钱能著. —北京：清华大学出版社，2009.7
(C++程序设计系列教材)
ISBN 978-7-302-20185-4

Ⅰ.C…　Ⅱ.钱…　Ⅲ.C语言－程序设计－高等学校－教材　Ⅳ.TP312

中国版本图书馆CIP数据核字(2009)第069422号

责任编辑：徐培忠　郑寅堃
责任校对：焦丽丽
责任印制：何　芊
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：北京密云胶印厂
装 订 者：三河市李旗庄少明装订厂
经　　销：全国新华书店
开　　本：185×260　**印　张**：30.5　**字　数**：740千字
版　　次：1999年4月第1版　2009年7月第2版
印　　次：2009年7月第1次印刷
印　　数：404001～410000
定　　价：42.00元

修订说明

为了给更多的人创造学习成才的机会，在《C++程序设计教程(第二版)》出版多年之后，再对原《C++程序设计教程》进行重要修订。修订也是为了理清C++学习的前后衔接关系，真正学以为用。

由于教材的撰写是从作者本身的学习经历出发，特别是第二版，更多的是考虑教师和熟练程序员的感受，基于他们的理解能力，而对教材内容进行构思和取舍的，同时在课程教学中以实验内容作为教学主线，更适合直接使用能力衡量型的考核方式。因此，对于非重点大学或非计算机专业的学生有必要采用相对浅近的教材，以相对简明的教学方式来达到C++编程学习和应用的目的。

考虑到原书特色，修订仍然保留了原来的框架和结构体系，并让描述风格保持不变。也就是保留了所有原始的学习经验和灵感，自然地去驾驭当初的学习冲动和写作冲动。

当初采用《C++程序设计教程》教材，切入C++课程教学，是从Pascal与C的技术争议和C与C++的教学争议起步的，以C++标准讨论稿为蓝本，以C为技术支撑，展开以方法为主线的教学模式。随着时间流逝，C的实用性比之Pascal已经无可争议，而C++又继承了C的简捷和高效，弥补了安全问题，支持了类型化编程，在更大规模上编程显得游刃有余。

如今C++各个教学层次的课程教学已经普及，教学经验已很丰富，教学效果已经能够得到保证。C++描述的数据结构课程也已经成熟。而且，选择C或C++作为编程学习起步的争议不再。两者在本质上并无差别：都是从过程化编程学习入手，然后再学习面向对象编程思想。只要明白C++并不等于单纯的面向对象编程语言，而是C的超集即可。

现在采用修订版的《C++程序设计教程》，能够从过程化到对象化编程的学习更连贯、自然。教材中通过展开过程化编程与面向对象编程的两种典型模式比较的讨论来展示C++比之C的技术进步之所在。教材一方面表示，相当一部分应用为小规模编程，其过程化编程方法同样重要；另一方面也表示，在展开规模化编程时，面向对象编程具有诸多的优势。而C++既可以进行过程化编程，又可以进行面向对象编程，两种方法各具特点，但无论采用哪种方法编程，比之C都更加安全、灵活和方便。

由于保留了原书体系和框架，仍以C++标准讨论稿为技术蓝本，所以，代码仍按前C++来描述：编译指令的包含(include)文件操作仍沿用C的习惯；字符串以过渡形式的string类型和字符数组形式的C字符串互相穿插的方式来描述，所以经常会进行C串函数的调用；由于基于当时的16位编译器代码调试，所以int数据类型常以16位长度来举例，相应对指针实体也作2字节长度的假设。这些仅仅是一些刻板的知识约定，可随新的C++标准而调整，对于编程方法来说，不会有任何影响。并且，书中的工程文件虽也保留了旧编译环境中的.prj扩展名，但所表达的程序组织结构仍是清楚的。

在C++标准化的进程中，对C++语言设计所作的更新，也是本版修订的一个原因。全书所有的概念描述都按C++标准予以了调整。其中，新的C++设计中，main函数的返回类型，不再用void，而是统一用int，以保证与操作系统界面沟通的设计一致性，在修订中，全部包

含 main 函数的程序都作了修改；表达式作为可以参加运算的量，在本质上应具有值的属性，使得其与过程(void 函数)调用相区别；常量是具有名字的不变量，而字面量是不具有名字的不变量，决定了后者常作为初始化等场合的一次性使用而与前者相区别，书中关于常量的章节也由此有了变更；原书对数组初始化的描述是沿用 C 规则的，但标准化 C++之后的数组初始化规则最终统一了 C 的相关设计，这也影响到本书修订中对数组初始化原则的调整；在类机制中，静态成员访问约定，在有新创对象引导的情况下，访问顺序不由语言设计确定而由实现(各自的编译器)确定，因而事实上具有不确定性。此外，原书的简易版快速排序算法在本次修订中也进行了更新。

与原书一致的是，所有编号的代码都是可运行代码实例，经作者在 Borland C++Builder 6 中重新测试。虽没有在 VC6 中全部测试，但亦不应有运行问题。在 VC6 中运行，遇有一些可能的运行结果差异，多是因为原 16 位编译器与现 32 位编译器之间存在着差异，没有其他本质上的不同。对于产生的如表达式副作用和编译不确定性所导致的差异，在描述中都有交代。由于多数代码规模不大，涉及知识点集中，所以用 C++工具做实验容易上手，不需要复杂的调试。

学习方式说明

对于初学者来说，C++程序设计编程技术的学习是循序渐进的，有两种成功的学习方式。

一种是直接从相对高强度的编程实践入手，即采用作者《C++程序设计教程(第二版)》作为主教材，《C++程序设计教程(第二版)实验指导》为配套，逐套实验，问题驱动。这种方式的学习相对来说，更需要有一个群体学习的氛围、及时的编程技巧点拨和相对坚韧的心理素质和学习悟性。

另一种是先从 C++的编程基本要素入手，即采用作者修订版的《C++程序设计教程》作为主教材，有选择性地采用《C++程序设计教程(第二版)实验指导》中的实验套题，完成课业之后，再用《C++程序设计教程(第二版)》的教材进行强化(可以自学)。这种方式的学习比较宽松、灵活，对知识基础没有太多要求，然而却一步一个脚印，周期虽拉得长一些，但可以取得与前者同样的效果。

作为指导，可以将本书看作是《C++程序设计教程(第二版)》学习的准备。因为第二版是以 C++标准为蓝本，强调内部特性和抽象编程，鼓励标准模板库的使用，面向对象编程本质揭示得更充分，结合《C++程序设计教程(第二版)实验指导》的使用，其能力导向更明确，操作实践性亦更强。

不管哪种学习方式，都是以 C 的过程化框架搭建为基础的。事实上，C++本质上就是过程化结构的，其学习模式都是从过程化编程学习开始的。只不过，本修订版描述 C++的时候，更多地使用 C 的技术成分和库函数，把 C 看作是其编程的基本和性能优化的基础。当然 C++本身是 C 的超集，对 C 的资源含而用之也是自然的。

本书特点

1. 从 C++的特色写起，以类型定义、识别、匹配为中心，注重编程质量

C++能代替 C，有它内在的原因。编程的复杂性已使得各级软件开发群体的扩张速度不能与硬件发展速度相适应，C 在大规模编程中的安全性问题已经使其应用处处受制。

C++的出现，主要就是解决软件开发的瓶颈问题。C++的特点也是在表现方便和安全编程中体现的：一个是非类特征，包括函数原型(function prototype)、重载函数(overload functions)、内联函数(inline functions)、参数默认(parameter defaults)、内存分配方法、const属性、数据实体的引用(reference)以及实体名作用域；另一个是类特征，它是由类机制来实现编程中的自定义类型能力的，包括数据封装、继承、多态、运算符重载、流以及相关的程序组织方法，除此之外，还有模板机制和异常机制。

C++的这一系列特征都是围绕类型安全或者说类型衔接而展开的。本书从强调函数原型开始，一直高度关注C++编译器的类型检查和匹配。

2. 从自身学习经历写起

本书不是为写语言而写语言，而是在写编程的概念与方法的同时，描述对C++的认识境界。既写与概念相关的作用、重要性、发展趋势和产生原因，也写对概念本质的个人认识。例如，将C++的表达式看作是一种参加运算的值结构量，可以作为任何逻辑判断的结果，也可以作为独立操作成分(语句)；又例如，数据与代码的关系，实体作用域问题，指针的直接与间接访问等问题，如果从程序运行的内存分布全局来看，将会有更为本质的理解。这样，便可从自身学习和把握概念的难易上，自然地理出难点，又从揭示的概念实质上，确立重点。

3. 从初学角度写起，点透要害，巧作取舍

由于本书不是语法参考书，所以不求面面俱到，可以把它看作是一本教会读者如何编程的自学教材。作者致力于强调方法，引导读者针对具体问题采用具体方法，致力于强调透视角度，教会读者如何感悟C++，从何处下手来学C++。

4. 打破文字描述的框框，追求通俗易懂

本书是一本学习编程的教科书，许多概念自然出自理性化的定义，但书中同样也重视用代码来补充说明，增强操作实践的感性认识基础，同时将世界上万事万物都是互相联系的本质运用于知识的理解，不拘概念描述的角度，以通俗易懂贯穿全书。

本书的自成体系和前呼后应，体现在围绕编程方法，从第一部分的过程化语言基础，到第二部分的面向对象程序设计，连贯自然。一个Josephus例子，在许多章节上，都拿出来比较，以表现不同的编程方法，让读者品味到过程化编程和对象化编程的区别和联系，教会读者学习方法，培养读者自学成才。

温馨致谢

在C++教学过程中，许多学生频繁地提出了各种各样的问题，给了作者很多的素材。一些学生阅读了本书初稿，他们从写作风格、可理解性、详略程度等方面都提出了许多宝贵的意见，给了作者很大的启示和帮助。

在C++教学研究中，作者有幸能与一些老教师一起讨论。他们有着几十年的教学经验，浑厚的数学功底和广博的计算机知识，作者在与他们的教学讨论中所获得的提高，也增强了书稿在文字描述上的严谨性。另外，许多同事对撰写此书的关心也给了作者很大的动力。

更为重要的是，本书能够精心构思、顺利完稿，根本上得益于作者恩师的敦促、鼓励和全力帮助。恩师也是作者的人生导师，是他让作者得以保持精力充沛和灵感迭现的状态，并在创作上得到了许多智慧和启迪。

大学 C++程序设计基础教学虽已有多年的积累，但我们深信，更好、更适用的教学方法、手段和形式一定会不断涌现。“抛砖”，是为了“引玉”，我们诚恳期待读者的批评指正和建议，以供再版时参考，使本书日臻完善。

作者的电子邮件地址：qianneng@mail.hz.zj.cn。

钱　能

2009 年春于杭州自在居

前　言

C++是一门高效实用的程序设计语言，它既可进行过程化程序设计，也可进行面向对象程序设计。C++语言强调对高级抽象的支持。C++实现了类的封装、数据隐藏、继承及多态，使得其代码容易维护及高度可重用。随着C++渐渐成为ANSI标准，这种新的面向对象程序设计语言迅速成为程序员最广泛使用的工具。几乎在所有计算机研究和应用领域，都能看到C++的影子。

C++从C进化而来，是C语言的超集。C++在程序结构的本质上与C是一致的，都是用函数驱动机制实现。学过C语言，再来看C++，就会感到C++更简单和容易理解。过程化程序设计与面向对象程序设计之间并无水火不容的矛盾，面向对象程序设计是过程化程序设计的自然升华。

本书对于学过或没有学过C语言的读者都是适用的。如果学过ANSI C，则可以跳过第一部分的程序设计基础，直接阅读第二部分。在学习第二部分时，遇到某些概念不清之处，可以根据章节目录查阅第一部分的有关内容。本书配备有《C++习题解答》，这对自学者尤为方便。

本书适合做大学计算机专业和非计算机专业的程序设计基础课程教材。通过本书可达到以下三个培养目标：

1. 程序设计入门，领略什么是面向对象程序设计；
2. 掌握程序设计方法，领会面向对象程序设计；
3. 把握C++程序设计的灵魂，掌握面向对象程序设计的方法。

纵观当前，C++的发展领导了程序设计语言的潮流，大有取代其他程序语言之趋势。在教学上，它以其面向对象的特征和严密的类型系统而正在悄无声息地取代Pascal；在科学计算功能上，它比Fortran更为可靠和方便；在系统软件的研究开发上，它又是上选的语种；在小规模控制应用上，C++的效率比之C毫不逊色；在大规模应用软件开发上，以Windows环境为代表的C++类库以及组件（组合类库）在迅速发展，它的触角触及各个领域。

C++编译器以Borland C++和Visual C++为典型，其版本在激烈的竞争中快速更新。那些老版本，如Microsoft C和Tubor C已经很少为人所用。即使学习C语言的人们目前也多在C++的编译环境中上机操作与调试。计算机等级考试的程序设计要求也在补充和更新。这就是C++不久即将取代C的无可抗拒的趋势与事实。C++作为程序设计基础教学也将以强有力的态势迅速取代C语言。

C++程序的集成开发器如Borland C++和Visual C++（2.0以下版），作为软件开发工具，已经不再时髦。Borland C++将不再有新版本，Borland公司的C++Builder将取代现在的Borland C++，就像当初Borland C++取代Tubor C一样。软件开发进入了可视化程序开发环境的时代。

一方面是陈旧的软件开发工具在淘汰，另一方面是新的软件开发工具在崛起。作者竭力推崇新的软件开发工具BC++ Builder 4.0和Visual C++ 6.0，因为它们不但是优秀的可

视化开发工具，而且由于它们是基于C++的，可以充分发挥C++语言强大灵活的特点，用面向对象程序设计方法去阅读、理解、思考和分析、设计以及编程。

学好C++，很容易触类旁通其他软件，如基于Java语言的Visual J++和J++ Builder，而且像ActiveX SDK和DirectX SDK等都是C++形式的。C++架起了通向强大、易用、真正的软件开发应用的桥梁。计算机发展到今天，培养一个具有相当的应用软件开发能力的人才，已经不需要很长的周期了。这给每个人都带来了最好的机遇和最大的挑战。我们的战略是，大学一年级学习C++课程，大学二年级，在适当的指导下就可学习如C++ Builder和Visual C++这些快速应用程序开发工具了。这意味着计算机专业的学生，大学二年级就可以冲向计算机应用的前沿，以此与计算机其他课程的学习相辅相成。非计算机专业的学生，同样也可以在应用开发的实践中找到自己的位置。

学程序设计，不仅要在运用程序设计的思想上获得一个大的突破，也应在运用程序设计的开发工具上走向真正的实用。这才是真正的理论联系实际，而且是事半功倍的学习方法。

本书分为两大部分。第一部分，第1章至第10章，是基础部分，主要介绍C++程序设计语言，程序结构和过程化基础。第二部分，第11章至第21章，是面向对象程序设计部分，它建立在C++程序设计基础之上，讲述了面向对象程序设计的方法。

本书内容特点

1. 强化重要概念

函数：函数是理解C/C++语言的重要基础。在书中作了较多的描述。

程序结构：程序结构的理解与实践是从C++语言的学习跨向实际应用的关键。作者在这方面也注入了较多的笔墨。

指针：指针是理解各种C++语言现象的关键，透过指针能够更好地理解语言的表达和程序的工作。这本书对指针重点展开介绍。

链表：链表是数据结构的基础，也是软件设计的重要技术。作者强调链表实现技术，是想从根本上让读者领会程序设计的技术性和艺术性。

引用：引用是C++特有的，C++面向对象的程序能够被人阅读理解，引用起了很大的作用。本书专辟一章讲述。

类：类是面向对象程序设计的首要概念。书中从第11章起，不停地描述类类型机制和面向对象程序设计的各项内容。

继承与虚函数：继承与虚函数是C++实现类的多态性的机制。它因而也是面向对象程序设计的关键之一。理解继承与虚函数能使读者从整体上把握面向对象程序设计的方法，从而走向真正的实用。

2. 舍弃次要内容

联合与位域：本书只介绍结构而不介绍联合和位域，是考虑到这些内容相对整个程序设计的功能来说已经退化。

多级指针：书中只介绍多级指针的概念（多维数组也如此），而不具体展开。

考虑到篇幅不讲成员指针：在《C++习题解答》中，有对成员指针的介绍，作为本书的补充。

3. 不求面面俱到，但求通俗易懂

本书不是语法参考书，是自成体系的C++教科书。它的目的是让读者通过自学或在老师讲授下，能够运用C++语言的基本要素，进行基本的结构化程序设计和面向对象的程序设计。因此全书紧紧围绕着如何进行程序设计而展开。前半部分，讲述C++程序设计基础，在此基础之上，后半部分，介绍面向对象程序设计的具体方法。

4. 以C作为背景

本书讲述的内容以ANSI C++为标准。书中出现的C内容是C++的参照。如关于C库函数printf和scanf，出现在与C++流作比较的场合，库函数malloc和free出现在与C++的new和delete作比较的场合。另外，还有C中的结构与C++中的结构之差异，C中的函数名与C++中的函数名之差异等。

考虑到准备参加C语言等级考试的读者，我们在书中对C的数学和字符串操作库函数作了适当的介绍。

本书编排特点

- 每章首均引出本章将要讲述的内容和学习要求。
- 在每章结束时，都做了小结，给出本章内容的概括性描述，指出本章在全书中的重要程度，并且指导学生进一步学习有关内容。
- 每章末安排的习题都是操作性习题，能通过上机加以验证。
- 书中遇有在理解上特别重要的内容用**黑体标记**。
- 与本书章节内容紧密联系的一些概念和说明用→引导，另起编段。
- 本书的各章与《C++习题解答》中的各章一一对应。教程中没有写进去的内容有些在习题解答中得到了补充。教程中的习题有些在习题解答中有类似的题解。
- 本书强调程序的可读性。使用统一的程序设计风格，并希望读者自始至终地模仿。
- 本书强调程序的可移植性。不以某个C++编译器为标准，而以最新的ANSI C++标准为编书标准。程序中的编译错误，一般用Borland C++(简称BC)中的出错信息给出。有些BC中的警告在Visual C++(简称VC)中是编译错误或反之，则同时也给出VC的出错信息。
- 本书中包含了大量的程序例子，并附有运行结果。凡在程序开头带有程序名编号的，都是完整的程序，它们经过了上机调试，可以直接在计算机上编译运行。运行结果中，如果是手工键入的字符，则用下划线表示。

本书中所有编号的程序，读者可到网站 http://www.tup.com.cn 上下载。

大学C++程序设计基础教学尚在起步，教学方法、手段和形式还在摸索中。本教材难免有不足之处，我们诚恳期待读者的批评指正和建议，以供再版时参考，使本书日臻完善。

作者的电子邮件地址：qianneng@mail.hz.zj.cn
通信地址：杭州市德胜新村12幢1单元101室(邮编310014)

目　录

第一部分　C++过程化语言基础

第二部分　面向对象程序设计

第一部分　C++过程化语言基础

第1章　C++ 入门

C++是一门优秀的程序设计语言。C++比C更容易为人们所学习和掌握，并且以其独特的语言机制在计算机科学领域中得到广泛的应用。学习本章后，要求了解C++语言的概念，了解C与C++之间的关系，了解C++语言对程序设计方法的支持，了解C++程序开发的过程，了解简单的C++程序结构，学会最简单的C++程序开发。

1.1　从C到C++

C语言是贝尔实验室的Dennis Ritchie在B语言的基础上开发出来的，1972年在一台DEC PDP—11计算机上实现了最初的C语言。C是作为UNIX操作系统的开发语言而广为人们所认识的。实际上，当今许多新的重要的操作系统都是用C或C++编写的。在过去20年内，C语言已经能够用在绝大多数计算机上了。C语言是与硬件无关的。由于C语言的严谨设计，使得把用C语言编写的程序移植到大多数计算机上成为可能。到70年代末，C已经演化为现在所说的"传统的C语言"。Kernighan和Ritchie在1978年出版的*The C Programming Language*一书中全面地介绍了传统的C语言，这本书已经成为最成功的计算机学术著作之一。

C语言在各种计算机上的快速推广导致了许多C语言版本。这些版本虽然是类似的，但通常是不兼容的。对希望开发出的代码能够在多种平台上运行的程序开发者来说，这是他们面临的一个严重的问题。显然，人们需要一种标准的C语言版本。为了明确地定义与机器无关的C语言，1989年美国国家标准协会制定了C语言的标准(ANSI C)。Kernighan和Ritchie编著的第二版*The C Programming Language*(1988年版)介绍了ANSI C的全部内容。

至此，C语言以其如下独有的特点风靡了全世界：

(1) 语言简洁、紧凑，使用方便、灵活。C语言只有32个关键字，程序书写形式自由。

(2) 丰富的运算符和数据类型。

(3) C语言可以直接访问内存地址，能进行位操作，使其能够胜任开发操作系统的工作。

(4) 生成的目标代码质量高，程序运行效率高。

(5) 可移植性好。

C 语言盛行的同时，也暴露出它的局限：

(1) C 类型检查机制相对较弱，这使得程序中的一些错误不能在编译时发现。

(2) C 本身几乎没有支持代码重用的语言机制，因此一个程序员精心设计的程序，很难为其他程序所用。

(3) 当程序的规模达到一定的程度时，程序员很难控制程序的复杂性。

为了满足管理程序的复杂性需要，1980 年，贝尔实验室的 Bjarne Stroustrup 开始对 C 进行改进和扩充。最初的成果称为“带类的 C”，1983 年正式取名为 C++，在经历了 3 次 C++修订后，于 1994 年制定了 ANSI C++标准的草案。以后又经过不断完善，成为目前的 C++。C++仍在不断发展中。

C++包含了整个 C，C 是建立 C++的基础。C++包括 C 的全部特征、属性和优点，同时添加了对面向对象编程(OOP)的完全支持。

1.2 程序与语言

1. 程序

程序是以某种语言为工具编制出来的动作序列，它表达了人的思想。计算机程序是用计算机程序设计语言所要求的规范书写出来的一系列动作，它表达了程序员要求计算机执行的操作。

对于计算机来说，一组机器指令就是程序。当我们说机器代码或者机器指令时，都是指的程序，它是按计算机硬件设计规范的要求编制出来的动作序列。

对于使用计算机的人来说，程序员用某高级语言编写的语句序列也是程序。程序通常以文件的形式保存起来，所以，源文件、源程序和源代码都是程序。

程序是任何有目的的、预想好的动作序列。它是一种软件。

计算机要运转起来，需要一整套可运行软件，即计算机程序。

学术界对程序的定义是比较严格的，这里不作详述。

2. 程序语言的发展

最早，程序员使用最原始的计算机指令，即机器语言程序。只有机器语言才能为机器所识别和运行。这些指令由一串二进制的数表示。不久，发明了汇编语言，它可以将机器指令映射为一些能被人读懂的助记符，如 ADD，SUB。程序员运行汇编程序将用助记符写成的源程序转换成机器指令，然后再运行机器指令程序，得到所要的结果。那时，编写程序的都是计算机专业人员，编写程序的语言都是低级的或较低级的。

以后，随着硬件的发展，Fortran，BASIC，Pascal，C 等几十种甚至几百种高级语言应运而生，中间经历了严酷的优胜劣汰过程，最后剩下的是一些比较优秀的高级语言。C++首当其冲。

多年来，计算机程序的主要目标是力求编写出短小的代码以使运行速度更快。因为硬

件成本和上机运行费很高。当计算机变得更小、更廉价、速度更快时，程序员开发程序、维护程序的费用急剧上升，而计算机硬件和运行的成本快速下降，程序设计的目标也就发生了变化。

在程序正确的前提下，可读性，易维护，可移植是程序设计首要的目标。所谓可读，就是使用良好的书写风格和易懂的语句编写程序。所谓易维护，是指当业务需求发生变化时，不需要太多的开销就可以扩展和增强程序的功能。所谓可移植，是指编写的程序在各种计算机和操作系统上都能运行，并且运行结果一样。

3. 高级语言和低级语言

C++语言是高级语言，机器语言是低级语言，汇编语言基本上是低级语言。例如：对于C++语言的语句：

```
a = 3 * a - 2 * b + 1; //3a - 2b + 1 的值赋给 a
```

写成汇编语言和对应的机器语言为：

```
mov eax, DWORD PTR a_$[ebp]          8b 45 fc
lea eax, DWORD PTR [eax + eax * 2]   8d 04 40
mov ecx, DWORD PTR b_$[ebp]          8b 4d f8
add ecx, ecx                         03 c9
sub eax, ecx                         2b c1
inc eax                              40
mov DWORD PTR a_$[ebp], eax          89 45 fc
```

第一条命令是将a放入寄存器eax中(ebp是数据段的指针，a_$是变量a的偏移位置)。

第二条命令是将eax的内容加上2倍的eax内容放到eax中，即eax中值为3 * a。

第三条命令是将b放入寄存器ecx中。

第四条命令是将ecx的内容加上ecx，即ecx中的值为2 * b。

第五条命令是将eax减去ecx的值(3 * a－2 * b)放入eax。

第六条命令是eax的值加1。此时，eax中的值为3 * a－2 * b+1。

最后一条命令是将寄存器eax的值放入a变量中，即实现a=3 * a－2 * b+1。

可以看出，程序语言越低级，描写程序越复杂，指令越难懂。语言越低级，就越靠近机器，越高级，就越靠近人的表达与理解。

程序语言的发展，总是从低级到高级，直到可以用人的自然语言来描述。

程序语言的发展，也是从具体到抽象的发展过程。编制一个表达式，无须将表达式的具体操作过程描述出来，否则人会感到太累，大量的精力会被无谓地浪费，无法进行更大规模的设计与思考。而低级语言则必须详尽地描述任何操作，所以抽象表达能力越强，语言越高级。

4. C与C++

C++语言包括过程性语言部分和类部分。过程性语言部分与C并无本质的差别，无非版本提高了，功能增强了。类部分是C中所没有的，它是面向对象程序设计的主体。要学习面向对象程序设计，首先必须具有过程性语言的基础。所以**学习C++，必先学习其过程性**

语言部分，然后再学类部分。也就是说，先学高版本的 C，再学类。从过程性语言的共有这个意义上说，学习 C++，无须先学 C。

程序设计方法正在从结构化程序设计走向面向对象程序设计。从根本意义上说，C 能够很好地支持结构化程序设计，而 C++能够很好地支持面向对象程序设计。当我们学习了 C 或 C++的过程化语言部分时，面临两种选择：一种是学习结构化程序设计方法，然后逐渐过渡到面向对象程序设计；另一种是直接学习面向对象程序设计。我们的观点是，无须先学结构化程序设计，而是在学了过程性语言之后，直接学习面向对象程序设计。这正是 C++的泰斗 Bjarne Stroustrup 和大多数 C++程序员都一致的观点：先学 C 没有必要。

然而，C 语言程序设计的经验非常有益。因为 C 程序设计开发锻炼了程序员进行抽象程序设计的能力，这正是 C++更为抽象的概念和技术的基础。而且，C++是 C 语言的扩展，它分享了 C 的许多技术风格。C 程序设计特性在 C++中得到频繁使用。一个人使用 C 的经验越丰富，编写 C++程序也就越容易。所以，学过 C 的人，能够促进 C++的学习。

1.3 结构化程序设计

以前，人们把程序看成是处理数据的一系列过程。过程或函数定义为一个接一个顺序执行的一组指令。数据与程序分开存储，编程的主要技巧在于追踪哪些函数调用哪些函数，哪些数据发生了变化。为解决其中可能存在的问题，结构化编程应运而生。

结构化程序设计的主要思想是功能分解并逐步求精。当一些任务十分复杂以致无法描述时，可以将它拆分为一系列较小的功能部件，直到这些自完备的子任务小到易于理解的程度。例如，计算一个公司中每一个职员的平均工资是一项较为复杂的任务。可以将其拆分为以下的子任务：

(1) 找出一个人的收入
(2) 计算总共有多少职员
(3) 计算工资总额
(4) 用职员人数去除工资总额

计算工资总额本身又可分为一系列子任务：

(1) 找出每个职员的档案
(2) 读出工资数额
(3) 把工资加到部分和上
(4) 读出下个职员的档案

类似地，读出每个职员档案中的记录又可以分解为一系列子任务：

(1) 打开职员的档案
(2) 找出正确记录
(3) 从存储设备中读取数据

结构化程序设计成功地为处理复杂问题提供了有力的手段。然而到 80 年代末，它的一些缺点越来越突出。

当数据量增大时，数据与处理这些数据的方法之间的分离使程序变得越来越难以理解。对数据处理能力的需求越强，这种分离所造成的负作用越显著。

采用结构化程序设计方法的程序员发现，每一种相对于老问题的新方法都要带来额外的开销，与可重用性相对，通常称之为重复投入。基于可重用性的思想是指建立一些具有已知特性的部件，在需要时可以插入到程序之中。这是一种模仿硬件组合方式的做法，当工程师需要一个新的晶体管时，他不用自己去发明，只要到仓库去找就行了。对于软件工程师来说，在面向对象程序设计出现之前，虽然市面上有些代码的功能看上去很像是自己需要的，但是修修改改最后还得自己动手重做。

1.4 面向对象程序设计

面向对象程序设计的本质是把数据和处理数据的过程当成一个整体——对象。

C++充分支持面向对象程序设计。面向对象程序设计的实现需要封装和数据隐藏技术，需要继承和多态性技术。

1. 封装和数据隐藏

当一个技术员要安装一台电脑时，他将各个设备组装起来。当他想要一个声卡时，不需要用原始的集成电路芯片和材料去制作一个声卡，而是来到电脑公司，购买一个他所需要的某种功能的声卡。技术员关心的是声卡的功能，并不关心声卡内部的工作原理。声卡是自成一体的。这种自成一体性称为封装性。无须知道封装单元内部是如何工作就能使用的思想称为数据隐藏。

声卡的所有属性都封装在声卡中，不会扩展到声卡之外。因为声卡的数据隐藏在该电路板上。技术员无须知道声卡的工作原理就能有效地使用它。

C++通过建立用户定义类型(类)支持封装性和数据隐藏。完好定义的类一旦建立，就可看成是完全封装的实体，可以作为一个整体单元使用。类的实际内部工作应当隐藏起来，使用完好定义的类之用户不需要知道类是如何工作的，只要知道如何使用它就行。

2. 继承和重用

要制造新的电视机，可以有两种选择：一种是从草图开始，另一种是对现有的型号加以改进。也许现有的型号已经令人满意，但如果再加一个功能，会更加完美。电视机工程师肯定不想从头开始，而是希望制造另一种新型电视机，该机是在原有的型号基础上增加一组电路做成的。新的电视机很快就制造出来了，被赋予一种新的型号，于是新型电视机就诞生了。这是继承和重用的实例。

C++采用继承支持重用的思想，程序可以在扩展现有类型的基础上声明新类型。新子类是从现有类型派生出来的，称为派生类。新型电视机是在原有型号的电视机上增加若干种功能而得到的，所以新型电视机是原有电视机的派生，继承了原有电视机的所有属性，并在此基础上增加了新的功能。

3. 多态性

通过继承的方法构造类，采用多态性为每个类指定表现行为。例如，学生类应该有一个计算成绩的操作。大学生继承了中学生，或者说是中学生的延伸。对于中学生，计算成绩的

操作表示语文、数学、英语等课程的计算，而对于后继的大学生，计算成绩的操作表示高等数学、计算机、普通物理等课程的计算。

继承性和多态性的组合，可以轻易地生成一系列虽类似但独一无二的对象。由于继承性，这些对象共享许多相似的特征。但由于多态性，一个对象可以有独特的表现方式，而对另一个对象有另一种表现方式。

1.5 程序开发过程

大多数现代的编译程序都提供了一个集成开发环境。在这样一个环境中，一般是从菜单中选定 compile 或 make 或 build 命令，来生成可执行的计算机程序。

程序员编制的源程序被编译(compile)后，会生成一个目标文件，这个文件通常以.obj作为文件扩展名。该目标文件为源程序的目标代码，即机器语言指令。但这仍然不是一个可执行的程序，因为目标代码只是一个个的程序块，需要相互衔接成为一个适应一定的操作系统环境的程序整体。为了把它转换为可执行程序，必须进行连接(link)。

C++程序通常是通过同时连接一个或几个目标文件与一个或几个库而创建的。库(.lib)是一组由机器指令构成的程序代码，是可连接文件。库有标准库和用户生成的库。标准库是由 C++提供的，用户生成的库是由软件开发商或程序员提供的。文件与库连接的结果，即生成计算机可执行的程序。

程序员首先在集成开发环境中编辑源程序，或在其他编辑器中输入源程序，然后，在集成环境中启动编译程序将源程序转化成目标文件。编译之后，很有可能产生一些编译错误，于是程序员回到编辑状态重新开始编辑程序和编译的过程。同样在紧接着的连接和运行中也会遇到连接或运行错误，此时，又回到编辑状态修改程序，见图 1-1。

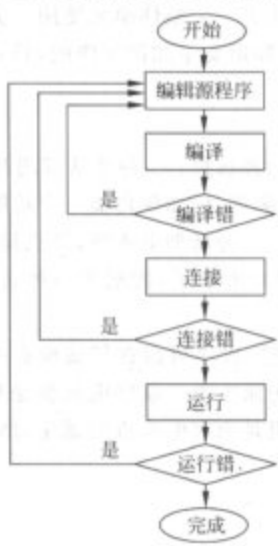

图 1-1 开发 C++程序的步骤

1.6 最简单的程序

我们从最简单的程序例子来分析 C++的程序构成。

例如，下面的代码是一个完整的可以运行的程序，它输出一个句子：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch1_1.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
int main()
{
   cout <<"I am a student.\n";
}
```

运行结果为：

```
I am a student.
```

C++的程序结构由注释、编译预处理和程序主体组成。

注释是程序员为读者作的说明，是提高程序可读性的一种手段。一般可将其分为两种：序言注释和注解性注释。前者用于程序开头，说明程序或文件的名称、用途、编写时间、编写人以及输入输出说明等，后者用于程序中难懂的地方。

C++的注释为"//"之后的内容，直到换行。注释仅供阅读程序使用，是程序的可选部分。在生成可执行程序之前，**C++忽略注释，并把每个注释都视为一个空格。**

另外，C++还兼容了 C 语言的注释，即一对符号"/*"与"*/"之间的内容。它可以占多行，例如：

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   this is the simplest program.
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
```

每个以符号"#"开头的行，称为编译预处理行。如"#include"称为文件包含预处理命令。编译预处理是 C++组织程序的工具，有关内容在 6.8 节中介绍。

"#include <iostream.h>"的作用是在编译之前将文件"iostream.h"的内容增加(包含)到程序 ch1_1.cpp 中，以作为其一部分。iostream.h 是系统定义的一个"头文件"，它设置了 C++的 I/O 相关环境，定义输入输出流对象 cout 与 cin 等。cout 与 cin 的使用方法将在 2.6 节中介绍，其意义将在第 19 章中介绍。

main()表示主函数，每一个 C++程序都必须有一个 main()函数。main()作为从计算机操作系统进入(调用)程序的入口。main 前面的 int 表示函数的返回类型。既然 main()函数被操作系统调用，其最终也将返回到操作系统。main()函数用 int 作为返回类型是 C 和 C++的共同规定。函数体用花括号{}括起来。**描述一个函数所执行算法的过程称为函数定义。**例如，这里的 main()函数头和函数体构成了一个完整的函数定义。

函数名 main 全部都是小写字母构成。C++程序中的名字是大小写"敏感"的，所以在书写标识符的时候要注意其大小写。

在 main()函数体中，cout(全是小写字母)是一个代表标准输出的流设备，它是 C++预

定义的对象(在 iostream.h 中定义),前面包含的头文件就是为了能在这里使用输出设备 cout。当程序要在设备上进行输出时,就需要在程序中指定该对象。输出操作由操作符"<<"来表达,它表示将该操作符右边的数据送到显示设备上。

程序中用双引号括起的数据 I am a student.\n 被称为字符串常量。其中字符"\n"表示一个回车控制符。字符串常量在 2.4 节中介绍。

";"表示一个语句的结束。

例如,下面的程序求一个表达式的值:

```
//* * * * * * * * * * * * * * * * * * * * *
//* *              ch1_2.cpp              * *
//* * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
int main()
{
   int a,b,result;
   cout <<"please input two numbers:\n";
   cin >>a >>b;
   result = 3 * a - 2 * b + 1;
   cout <<"result is " <<result <<endl;
}
```

运行结果为:

```
c>ch1_2 <ENTER>
please input two numbers:
123 45 <ENTER>
result is 280
```

该程序从 main()开始运行。C++中,**一个变量必须在声明之后才能使用**,所以程序首先进行变量定义。"int a,b,result;"表示分别定义 a,b,result 这 3 个 int(整型)变量。C++语言提供的标准数据类型之一是 int。定义变量时,要求在变量之前申明变量的类型。**C++中定义变量,意味着给变量分配内存空间**,用来存放变量值。

随后,在显示"please input two numbers:"之后,执行"cin >>a >>b;",它从标准输入设备(键盘)中输入两个整型数 a 和 b。运行中,屏幕将等待输入,直至输入了两个数 123 和 45。输入时,两个数之间用空格隔开。该两数分别赋给了变量 a 和 b。

"result=3 * a-2 * b+1;"是赋值语句,* 是乘号,将表达式 3 * a-2 * b+1 的值(280)赋给变量 result,使之等于 280。然后,在接下来的语句中将 result 值输出。在 cout 语句中,有 3 个"<<"符号,表示各项内容的连续输出。"<<result"表示输出变量的值,"<<endl"表示输出一个回车符,与"<<'\n'"是等价的。

在输出格式中,<ENTER>表示键入的回车符,在以后的例子中将省略之。

1.7 函数

C++用函数组织程序

虽然 main()也是函数,但它并不是普通的函数。大多数函数是在程序运行时被调用。程

序命令按照它们在源代码中出现的顺序一句一句地顺序执行，直到碰到新的函数调用。然后程序调头去执行函数调用。当函数完成时，程序控制立即返回到调用函数的下一行代码。

这一过程可比喻为查字典。如果你在看书对有一个字不认识，于是，就停止阅读，去查字典。字典查完后，又接着看书。

当程序需要服务时，它可以调用函数实现所需要的服务，然后当函数返回时再从它原来的地方继续执行。

C++程序是函数驱动的

例如，下面的程序实现一个简单的用户函数 max()的调用，来求两个数中的较大值，并调用了标准库函数 sqrt()来求两个数中较大值的平方根：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch1_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <math.h>

double max(double x, double y);

int main()
{
   double a,b,c;
   cout <<"input two numbers:\n";
   cin >>a >>b;

   c = max(a,b);
   cout <<"the squart of maximum = " <<sqrt(c);
}

double max(double x, double y)
{
   if (x>y)
     return x;
   else
     return y;
}
```

运行结果为：

```
c>ch1_3
input two numbers:
123 456
the squart of maximum = 21.3542
```

主函数 main()的开始是 3 个 double 型(双精度类型)变量的定义语句。C++为此给其分配 3 个 double 型变量的内存空间。在输入了两个变量 a,b 的值(运行中输入的 123 赋给 a,456 赋给 b)后，调用了用户自定义的函数 max()。

C++中，**一个函数必须在函数声明后才能使用(被调用)**，所以在主函数 main()的前面，有 max()函数的声明。函数声明告诉编译器，该函数是存在的。后面编译器在看到该函数被调用时就不会觉得大惊小怪了。同时编译器还对函数调用进行正确性检查。**C++函数声明总是由函数原型构成的。**函数原型在 5.2 中介绍。

max()函数调用使程序执行 max()函数中的语句，并将该函数的返回值赋给变量 c。

max()函数是求两个 double 型数中的大者，然后将结果返回给调用它的函数。所以在函数的头上，写有 double 的返回类型。如果一个函数不需要返回值，则可以在头上声明为 void。

函数定义由函数头和函数体构成。函数头又由返回类型、函数名和函数参数构成。上例中的"double max(double x, double y)"就是函数头。函数体是由紧随函数头之后的花括号构成。

函数头中的函数参数允许向函数传递值。max()函数中，x,y 就是函数的参数。**参数声明时，要指出其类型**。函数 max()中的参数声明为"double x,double y"，它指出 x 和 y 的类型都是双精度型。

函数定义中的参数称为形式参数，简称形参。函数 max()中的 x 和 y 就是形参。**调用函数时实际传递的值称为实际参数**，简称实参。主函数 main()在对 max()函数的调用时，用的 a 和 b 就是实参。函数在调用时，将实参值复制给形参，使得形参变量也具有实参的值。实参可以是表达式，它代表赋值的一方。形参只能是变量，因为它要接受赋值。

函数头有返回类型说明时，函数体中要用 return 返回值。同时，return 语句也使函数退出。max()函数中执行"return x"或"return y"即返回一个 double 值到主函数 main()中。

如果函数体中没有 return 语句，函数将在结尾处自动无值返回。如果有返回值，则该返回值应该具有函数头中声明的返回类型。

→main()函数是唯一不是 void 返回类型(int)，而可以忽略 return 语句的函数。因为只有这个函数由操作系统直接控制，其值返回给操作系统而不是其他函数，在技术上可以独立实现返回过程而不影响 C/C++的语言机制。C/C++标准都对 main()函数中不写 return 语句予以了默许。

函数有两种：标准库函数和用户定义函数。上例中的 max()函数是用户定义函数，sqrt()函数是标准库函数。标准库函数简称库函数，它是 C++提供的，可以为任何程序所使用。库函数无须用户声明和定义，但要将含有其函数声明的头文件包含在程序中。sqrt()函数的声明在 math.h 头文件中，所以在上例程序的开头，写有 #include<math.h>。

一个C++程序由一个主函数和若干个函数构成。由主函数调用其他函数，其他函数也可以互相调用。同一个函数可以被一个或多个函数调用任意多次。见图 1-2。

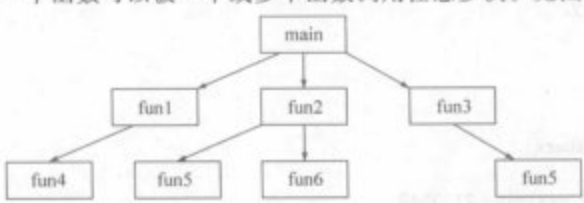

图 1-2　程序中的函数调用

函数定义包含函数声明，所以可以将函数定义放在函数应该声明的位置，而将其他函数的定义放在主函数 main()之前。一个程序中主函数 main()的位置是没有特殊要求的，由此，程序 ch1_3.cpp 可以写成下面的程序：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                ch1_4.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * *
```

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

double max(double x,double y)  //max()函数定义,同时也是函数声明
{
   if (x>y)
     return x;
   else
     return y;
}

int main()
{
   double a,b,c;
   cout <<"input two numbers:\n";
   cin >>a >>b;

   c = max(a,b);

   cout <<"the squart of maximum = " <<sqrt(c);
}
```

该程序的功能和 ch1_3.cpp 是一样的。但通常我们习惯将主函数 main()放在程序的前面,以便阅读程序时很快找到。

→ C++中,每个函数对于程序的其他函数总是可见的。也就是说,任何函数都可以被包括它自己的所有函数所调用。**用户不能定义函数的唯一之处是在另一个函数的定义之中**。函数定义可以以任何顺序出现在程序中。由于 main()启动和终止程序运行,所以 main()函数通常第一个出现在程序中,而其他函数定义紧随其后。

小结

学习 C++,不一定非要学过 C。但学过 C 能促进 C++的学习。

C++程序经过编辑、编译和连接,产生可运行的 exe 文件。

C++程序由函数构成,它总是从主函数 main()开始运行。但并不是说,main()函数非得要写在程序的最前面。

函数有两种:标准库函数和用户定义函数。main()函数是特殊的用户定义函数。每个程序只能有一个 main()函数,并且必须要有一个 main()函数。

函数调用前必须要有函数声明。

函数定义包含函数声明。函数定义由函数头和函数体组成。关于函数,在第 4 章中将详细介绍。

一个语句可以写在多个程序行上,一个程序行可以写多个语句。语句以分号结束。

C++通过标准输入输出流进行输入输出。

程序 ch1_3.cpp 是C++的简单程序结构之样板。认识 C++程序从该程序开始。

程序设计的目标在正确的前提下,其重要性排列次序依次为:可读、可维护、可移植和高效。

第2章　基本数据类型与输入输出

程序中最基本的要素之一是数据类型。确定了数据类型，才能确定变量的空间大小和其上的操作。C++的数据类型检查与控制机制，奠定了C++今天的地位。C++还提供了I/O流机制，完成对输入输出的操作管理。在过程化程序设计中，经常要碰到printf和scanf的输入输出方式，它们是C++对C的兼容。学习本章后，要求搞清数据类型与变量、常量的关系，掌握各种常量的性质和定义，学会I/O流的使用，了解printf和scanf输入输出的作用。

2.1　字符集与保留字

每种语言都使用一组字符来构造有意义的语句。C++程序是用下列字符所组成的字符集写成的：

26个小写字母	abcdefghijklmnopqrstuvwxyz
26个大写字母	ABCDEFGHIJKLMNOPQRSTUVWXYZ
10个数字	0123456789
其他符号	+- * / = , . _ : ; ? \ " ' ~ | ! # % & ( ) [ ] { } ^ <> (空格)

C++中，保留字也称关键字。它是预先定义好的标识符，这些标识符对C++编译程序有着特殊的含义。表2-1列出了C++的保留字。ANSI C规定有32个保留字，表中用黑正体字表示；ANSI C++在此基础上补充了29个保留字，表中用黑斜体字表示。本书不作介绍的表中用白体字表示。为了使语言能更好地适应软件开发环境，BC或VC对保留字进行了扩充，在表中用白斜体字表示。VC与BC对关键字的扩充内容是不同的，这里只是常用的和共同扩充的几个。

表2-1　C++保留字

auto	**break**	**case**	**char**
const	**continue**	**default**	**do**
double	**else**	**enum**	**extern**
float	**for**	**goto**	**if**
int	**long**	register	**return**
short	**signed**	**sizeof**	**static**
struct	**switch**	**typedef**	union
unsigned	**void**	volatile	**while**
bool	***catch***	***class***	***const_cast***
delete	dynamic_cast	explicit	false
friend	***inline***	mutable	namespace
new	***operator***	***private***	***protected***
public	reinterpret_cast		static_cast
template	***this***	***throw***	true

try	typeid	typename	using
virtual	wchar_t		
asm	*cdecl*	*far*	*huge*
interrupt	*near*	*pascal*	*export*
except	*fastcall*	*saveregs*	*stdcall*
seg	*syscall*	*fortran*	*thread*

在程序中用到的其他名字(标识符)不能与 C/C++的关键字有相同的拼法和大小写。关键字也不能重新定义。

2.2 基本数据类型

一个程序要运行,就要先描述其算法。描述一个算法应先说明算法中要用的数据,数据以变量或常量的形式来描述。每个变量或常量都有数据类型。

变量是存储信息的单元,它对应于某个内存空间。用变量名代表其存储空间。程序能在变量中存储值和取出值。

在定义变量时,说明的变量名字和数据类型(如 int)告诉编译器要为变量分配多少内存空间,以及变量中要存储什么类型的值。

内存单元的单位是字节。如果要建立的变量类型的长度是 2 个字节,变量就需要保留 2 个字节的内存。变量的数据类型的作用之一是告诉编译器要为变量分配多少内存。

数据类型简称类型。在不同的计算机上,每个变量类型所占用的内存空间的长度不一定相同。例如,在 16 位计算机中,整型变量占 2 个字节,而在 32 位计算机中,整型变量占 4 个字节。C++的数据类型有基本数据类型和非基本数据类型之分。基本数据类型是 C++内部预先定义的数据类型。非基本数据类型包括指针、数组和结构以及类类型等,非基本数据类型是基本数据类型的合成或用户定义数据类型。

基本数据类型有 char(字符型)、int(整型)、float(浮点型)和 double(双精度型)。在 ANSI C++中,还有 wchar_t(双字节字符型)和 bool(布尔型),见图 2-1。

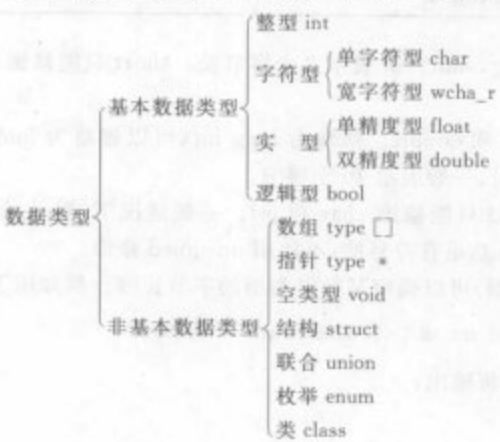

图 2-1 C++的数据类型

图中 type 表示非空数据类型。

除上述一些基本数据类型外，还有一些数据类型修饰符，它用来改变基本类型的意义，以便更准确地适应各种情况的需要。修饰符有 long(长型符)、short(短型符)、signed(有符号)和 unsigned(无符号)。

数据类型的描述确定了其内存所占空间大小，也确定了其表示范围。以在 16 位计算机中表示为例，基本数据类型加上修饰符有表 2-2 的描述。

表 2-2 常用基本数据类型描述

类 型	说 明	长度(字节)	表示范围	备 注
char	字符型	1	−128～127	$-2^7\sim(2^7-1)$
unsigned char	无符号字符型	1	0～255	$0\sim(2^8-1)$
signed char	有符号字符型	1	−128～127	$-2^7\sim(2^7-1)$
int	整型	2	−32768～32767	$-2^{15}\sim(2^{15}-1)$
unsigned int	无符号整型	2	0～65535	$0\sim(2^{16}-1)$
signed int	有符号整型	2	−32768～32767	$-2^{15}\sim(2^{15}-1)$
short int	短整型	2	−32768～32767	$-2^{15}\sim(2^{15}-1)$
unsigned short int	无符号短整型	2	0～65535	$0\sim(2^{16}-1)$
signed short int	有符号短整型	2	−32768～32767	$-2^{15}\sim(2^{15}-1)$
long int	长整型	4	−2147483648～2147483647	$-2^{31}\sim(2^{31}-1)$
signed long int	有符号长整型	4	−2147483648～2147483647	$-2^{31}\sim(2^{31}-1)$
unsigned long int	无符号长型型	4	0～4294967295	$0\sim(2^{32}-1)$
float	浮点型	4	$-3.4\times10^{38}\sim3.4\times10^{38}$	7 位有效位
double	双精度型	8	$-1.7\times10^{308}\sim1.7\times10^{308}$	15 位有效位
long double	长双精度型	10	$-3.4\times10^{4932}\sim1.1\times10^{4932}$	19 位有效位

在大多数计算机上，short int 表示 2 个字节长。short 只能修饰 int，short int 可以省略为 short。

long 只能修饰 int 和 double。修饰为 long int(可以省略为 long)时，一般表示 4 个字节，修饰 long double 时，一般表示 10 个字节。

unsigned 和 signed 只能修饰 char 和 int。一般情况下，默认的 char 和 int 为 signed。实型数 float 和 double 总是有符号的，不能用 unsigned 修饰。

用 sizeof(数据类型)可以确定某数据类型的字节长度。例如用下面的语句：

```
cout <<"size of int is " <<sizeof(int) <<endl;
```

在 16 位计算机上将输出：

```
size of int is 2
```

C++是强类型语言。强类型语言要求程序设计者在使用数据之前对数据的类型明确声明，不能默认。例如，在存储一个整数值之前，首先必须告诉计算机要存储的是一个整数类型。使用强类型语言有很多好处，例如，错把一个整数当成一个职工的编号，编译器就会产生错误信息提示。

强类型语言是通过编译器的功能来体现的。**一个编译器能检查出的错误越多，我们就说该编译器越好。**

在程序设计中，绝大部分的错误是发生在数据类型的误用上，所以现代程序设计语言都要求是强类型语言，因它能够检查出尽可能多的数据类型方面的错误。

例如，如果一个圆半径用浮点数(float)表示，占用4个字节的内存，而一个整数只占用2个字节的内存。如果要清除一个圆半径值，那么就要清除4个字节的内容。如果把一个整数当作一个圆半径来清除，那么除了清除整数的2个字节外，还清除了其他2个字节的内容。也许额外清除的2个字节中包含有重要的信息，这种错误的清除会造成意想不到的后果。

对于一个初学者来说，那种对数据类型要求不严格的语言(如：BASIC)或许更易于操作。但是，它不会像强类型语言那样能够捕获错误，因而调试和运行的程序可能出现更多的错误。

2.3 变量定义

1. 命名变量名

C++程序是大小写敏感的，即大写和小写字母认为是不同的字母。例如变量名something，Something，SOMETHING 和 SomeThing 都是不同的名字。

命名变量名要遵守如下的规则：

(1) 不能是C++关键字。

(2) 第一个字符必须是字母或下划线。

(3) 不要太长，一般不超过31个字符为宜(太长则书写困难，反为不美)。

(4) 中间不能有空格。

(5) 变量名中不能包含“.；，”'+-”之类的特殊符号。实际上，变量名中除了能使用26个英文大小写字母和数字外，只能使用下划线“_”。

(6) 变量名不要与C++中的库函数名、类名和对象名相同。

例如，下面是一些变量名：

```
way_cool, RightOn, Bits32, Case, iPtr, myCar        //合法
case, 52Select, A Lot, -VV                          //非法
sin, cout, string                                   //不合适
```

变量通常具有描述性的名称。例如，看到 numberOfStudents 这个变量名，就立刻可以想到它表示学生人数，即使简写成 numOfStudent 也是一目了然。另一方面，D6Xy 就不是一个好名称，猜不透它代表一个序列号、一个商标，还是一种机器名。命名变量名的方式，决定了程序书写的风格。在整个程序中保持同一风格很重要。

→ 许多程序员喜欢变量名全用小写字母。如果名字需要两个单词(如 my car)，常用的

命名有两种：my_car 和 myCar。后一种形式称为骆驼表示法，因为大写字母看起来像驼峰。

有些人觉得 my_car 较可读，也有人因为它难以输入而避免使用下划线。本书使用骆驼表示法，即以小写字母开头，以后的单词都以大写字母开头，如 myBook，theFox，sizeOfChar。

自定义的类型名（如类和结构类型）则以大写字母开头。一些函数名也以大写字母开头。

还有一种特别流行的方法是匈牙利标记法（Hungarian notation），该法在每个变量名的前面加上若干表示类型的字符，如 iMyCar 表示整型变量，ipMyCar 表示整型指针等。这种方法已经流行于现代软件开发环境中，如 Windows 中的类库和函数库。

2. 变量定义方式

可以在一个语句里建立多个同一类型的变量，方法是在类型后写上多个变量名，中间用逗号隔开。例如：

```
unsigned myAge, myWeight;      //2 个无符号整型变量
long area, width, length;      //3 个长整型变量
```

在同一语句里不能混合定义不同类型的变量。

3. 变量赋值与初始化

用赋值运算符“=”给变量赋值。例如：

```
unsigned short width;
width = 5;          //赋初值
```

也可以在定义时直接给变量赋值。在定义的同时，赋给变量一个初始值，称为变量的初始化。例如：

```
unsigned short width = 5;     //定义并初始化
```

对于整型变量来说，赋初值的形式用 2 条语句完成，初始化的形式只用 1 条语句。它们都是先给变量分配一个整数存放的内存空间，然后将一个整数值赋给该变量。其初始化与赋初值的效果完全一样。然而当定义常量或定义一个对象时，两者的差别则很大。

在定义时也可以初始化多个变量。例如：

```
long width = 7, length = 7;
```

不是所有的变量在定义时都需要初始化。例如：

```
double area, radius = 23;
```

该变量定义并不是将 23 同时赋给这两个变量，而是变量 radius 初始化为 23，area 只是定义，没有被初始化。

4. typedef

用 typedef 可以为一个已有的类型名提供一个同义词。用法是，以 typedef 开始，随后是要表示的类型，最后是新的类型名和分号。例如：

```
typedef double profit;          //定义 double 的同义词
typedef int INT, integer;       //定义两个同义词
INT a;                          //即 int a;
profit d;                       //即 double d;
```

typedef 没有实际地定义一个新的数据类型，在建立一个 typedef 类型时没有分配内存空间，typedef 在程序中起到帮助理解的作用。

2.4 字面量

1. 整型数

整型数即整型字面量。整型数可用 3 种表示方式：

(1) 十进制整数。如 123，−456，0。

(2) 八进制整数。以 0 开头的整数是八进制数。如 0123 表示八进制数$(123)_8$，等于十进制数 83。

(3) 十六进制数。以 0X 或 0x 开头的数是十六进制数。如 0X123 或 0x123 表示十六进制数$(123)_{16}$，等于十进制数 291。

C++中，十进制数有正负之分，但八进制和十六进制数只能表示无符号整数。所以，若写成−011 或−0X345，C++并不能将其理解为负数。

如果在整型数后面加一个字母 L 或 l，则认为是 long int 型整数。例如 123L 是 long int 型整数。

2. 实型数

实型数即实型字面量。实数在 C++中就是浮点数。实数有两种表示：

(1) 小数形式。它由数字和小数点组成(注意必须有小数点)。如 0.123，.234，0.0 等都是十进制数。

(2) 指数形式。如，123e5 或 123E5 都表示 123×10^5。要注意 E 或 e 的前面必须要有数字，且 E 后面的指数必须为整数。如 e3，2.1e3.5，.e3 和 e 等都不是合法的指数形式。

实型数分为单精度(float)、双精度(double)和长双精度(long double)3 类。一般 float 型数据在内存中占 4 个字节，double 型数据占 8 个字节，long double 型数据在内存中占 10 个字节。float 型提供 7 位有效数字，double 型提供 15 位有效数字，long double 提供 19 位有效数字。

在 C++中，一个实型数如果没有任何说明，表示 double 型。要表示 float 型数，则必须在实数后加上 f 或 F。表示 long double 型数，则必须在实数后加 l 或 L。例如：

```
34.5f       //float 型
34.5        //double 型(默认表示)
34.5L       //long double 型
34.5l       //long double 型
34.5e23f    //float 型
34.5e23     //double 型(默认表示)
34.5e23L    //long double 型
34.5e23l    //long double 型
```

```
34.5e400      //long double 型(因为范围超过 double 表示)
```

3. 字符

字符是用单引号括起来的一个字符。如'a','x','?','$'等都是字符。

除了以上形式的字符外,C++中还允许使用一种特殊形式的字符,即以"\"开头的字符序列。如'\n',它代表一个换行符,表 2-3 列出了 C++中常用的以"\"开头的特殊字符。

表 2-3 C++常用特殊字符

字符形式	值	功 能
\a	0x07	响铃
\n	0x0A	换行
\t	0x09	制表符(横向跳格)
\v	0x0B	竖向跳格
\b	0x08	退格
\r	0x0D	回车
\\	0x5C	反斜杠字符"\"
\"	0x22	双引号
\'	0x27	单引号
\ddd		1～3 位八进制数
\xhh		1～2 位十六进制数

表中列出的字符称为转义字符,意思是将反斜杠"\"后面的字符转变成另外的意义。有些是控制字符,如"\n",有些是表示字符的符号,如"'"。

反斜杠可以和八进制或十六进制数值结合起来使用,以表示相应于该数值的 ASCII 码值。例如,'\03'表示 Ctrl-C,'\0A'表示回车。转义字符使用八进制数表示时,最多是 3 位数,不必以 0 开头,如'\03'等价于'\3'。转义字符的八进制数表示的范围是'\000'～'\377',即从 $0\sim255_{10}$。转义字符使用十六进制数表示时,是 2 位数,用 x 或 X 引导,表示的范围是'\x00'～'\xff'。

例如,下面的代码响铃输出一个字符串:

```
cout <<"\x07operating\tsystem\n";
```

其输出内容为在响铃的同时显示:

```
operating       system
```

该结果中两个单词之间的空隙是制表符'\t'产生的作用。

字符变量定义的形式为:

```
char c1,c2;
```

它表示 c1,c2 为字符型变量,各只能放一个字符。初始化字符变量可以用下述形式:

```
char c1 = '\n', c2 = '\x07', c3 = 'B', c4 = '\xff', c5 = 97;
```

将一个字符赋值给字符变量,实际上并不是把该字符本身放到内存单元中,而是将该字符的相应 ASCII 码(整型数)存入。例如,字符'a'的 ASCII 码是 97,上例中"c5=97"即为"c5='a'"。

在内存中,字符数据以 ASCII 码存储,即以整数表示,所以 C++中字符数据和整型数据之间可以相互赋值,只要注意其表示的范围合理。例如:

```
int a = 'b';     //ok:   给一个整型变量赋一个字符值
char c = 97;     //ok:   给一个字符变量赋一个整型值
```

字符数据和整型数据在输出中的表示是不同的。例如,对上面的赋值,输出:

```
cout <<a <<endl;
cout <<c <<endl;
```

其结果为:

```
98
a
```

尽管整型变量 a 赋给了字符值,字符变量 c 赋给了整型值,但它们在内存中都以整数的形式表示。在输出时,a 是整型,所以就按整型数的表示方式输出,c 是字符型,所以就按字符的表示方式输出。

→ '0'与 0 是截然不同的两个数。'0'是数字字符,其 ASCII 码等于值 48 或 0x30。而 0 则是整数值。除此之外,'\0'和 NULL 也表示整数 0。

4. 字符串

字符串是由一对双引号括起来的字符序列。例如:

```
"How do you do?"
"I am a student."
"hello"
```

都是字符串。字符和字符串是不同的。在 C++中,字符串总是以'\0'结束。如果有一个字符串"HELLO",那它在内存中的表示为连续 6 个内存单元,如图 2-2。

H	E	L	L	O	'\0'

图 2-2 字符串的内存表示

不能将字符串赋给字符。例如:

```
char c = "abc";     //error:  不能将字符串转换成字符型
```

一个字符占有一个内存单元,含有一个字符的字符串占有 2 个内存单元,第 2 个内存单元存放 0 结束符。所以,**"0"与'0'是不同的**。在 8.7 节,将进一步介绍字符串。我们将看到,

字符串实际上是字符指针类型。所以，字符与字符串类型不同，占用内存大小不同，处理方式不同，决定了它们的使用方式也不同。

单个字符的字符串与一个字符在输出的表示上没有差别，因为字符串输出时，C++并不把0结束符一起输出。例如：

```
cout <<"a" <<endl;
cout <<'a' <<endl;
```

输出结果为：

```
a
a
```

5. 枚举符

枚举符可以通过建立枚举类型来定义。

定义枚举类型的语法是先写关键字 enum，后跟类型名、花括号，花括号括起来的里面是用逗号隔开的每个枚举符，最后以分号结束定义。例如：

```
enum COLOR{ RED,BLUE,GREEN,WHITE,BLACK };
```

例中，COLOR 是枚举类型名，**它不是变量名，所以不占内存空间**。枚举类型定义规定枚举类型名和枚举的取值范围。

枚举符是一种符号常量。RED，BLUE 等是符号常量，它们表示各个枚举值，在内存中表示以整型数。如果没有专门指定，第1个符号常量的枚举值就是0，其他枚举值依次为1往上加。所以，C++自动给 RED 赋以0，BLUE 赋以1，等等。

可以给枚举符指定枚举值，也可以部分指定枚举值。例如：

```
enum COLOR{ RED = 100, BLUE = 200, GREEN, WHITE = 400};
```

例中 GREEN 没有赋给值，便被自动赋以值201。

定义了枚举类型后，可以定义该枚举类型的变量。变量的取值只能取枚举类型定义时规定的值。例如：

```
COLOR paint = GREEN;
```

该例定义了一个枚举变量 paint，用枚举符 GREEN 所代表的枚举值初始化。

不能用整数值赋给枚举变量。例如：

```
paint = 200;     //error
```

在 VC 中会得到一个“不能将整常数赋给枚举变量”(cannot convert from 'const int' to 'enum paint')的错误。在 BC 中会得到一个“将整数赋给枚举变量 paint”的警告。BC 的警告比 VC 显得缓和一些，但都表示整数赋给枚举型变量是不合适的。**任何警告对于程序设计都应引起与编译错误同等的重视。**

2.5 常量

常量是常数或代表固定不变值(字面值)的名字。

程序中如果想让变量的内容自初始化后一直保持不变，可以定义一个常量。

例如，在圆面积计算中经常要用常数 π，此时，通过命名一个容易理解和记忆的名字来改进程序的可读性，同时在定义中加关键字 const，给它规定为常量性质，以帮助预防程序出错。

如果在整个程序中的许多地方都要用到一个字面值，那么，在这些地方的一个或多个地方写错了这个值就会导致计算错误。如果给字面值取个名字，每处都以该名字代替，那么编译器就能检查名字拼写错误，避免字面值的不一致性。

π 字符不属于 C++语言的字符描述集，不能用来作 C++中的名字。我们用 pi 来表示 π：

```
const float pi = 3.1415926;
```

由于有效位的限制，在下面常量定义中，最后 3 位不起作用：

```
const float pi = 3.141592653;
```

尽管等号后面的字面值是 double 型的，但因为 float 常量只能存储 7 位有效位精度的实数，所以 pi 的实际值为 3.141593（最后 1 位 4 舍 5 入）。如果将常量 pi 的类型改为 double 型，则能全部接受上述 10 位数字。

定义成 const 后的常量，程序中对其只能读不能修改，从而可以防止该值被无意的修改。由于不可修改，不能赋值，所以，常量定义时必须初始化。例如：

```
const float pi;
pi = 3.1415926;   //error
```

常量名不能放在赋值语句的左边。

常量定义中初始化的值可以是一个不依赖于运行的表达式。该表达式在程序运行之前就能计算，所以，编译时就能求值。例如：

```
const int size = 100 * sizeof(int);      //ok
const int number = max(15,23);           //error
```

因为 sizeof 不是函数，而是 C++的基本操作符，该表达式的值在编译之前能确定，所以第一个常量定义语句合法。第二个语句要求函数值，函数一般都要在程序开始运行时才能求值，该表达式不能在编译之前确定其值，所以是错误的。

一般来说，相同类型的变量和常量在内存中占有相同大小的空间。只不过常量不能通过常量名去修改其所处的内存空间，而变量却可以。

→ 关于 #define

在 C 中，另一种定义常量的方法是用编译预定义指令（#define）。例如：

```
#define PI 3.1415926
```

这条语句的格式是 #define 后面跟一个常量名再跟一串字符，中间用空格隔开。由于它不是 C++语句，所以行末不用分号。

当程序被编译时，它要先被编译预处理。当预处理遇到 #define 指令时，就用数值 3.1415926 替换程序中所有的 PI。

尽管它具有常量的所有属性，但是，在编译预处理完成后，PI 不属于 C++程序中的名字了（全部被字串 3.1415926 所替代），所以它不是一个具有一定类型的常量名。随后的编译

无法发现由它引起的数据类型误用的错误。

C++容许#define定义常量是为了兼容C，使C程序能在C++编译器中顺利通过。在C++编程中，常量定义都用const，不用#define。

2.6 I/O流控制

1. I/O的书写格式

I/O流是输入或输出的一系列字节，当程序需要在屏幕上显示输出时，可以使用插入操作符"<<"向cout输出流中插入字符。例如：

```
cout <<"This is a program.\n";
```

当程序需要执行键盘输入时，可以使用抽取操作符">>"从cin输入流中抽取字符。例如：

```
int myAge;
cin >>myAge;
```

不管把什么基本数据类型的名字或值传给流，它都能懂。

例如，下面的函数输出字符串和整数：

```
#include<iostream.h>

int main()
{
  cout <<"My name is Jone\n";
  cout <<"the ID is ";
  cout <<2;
  cout <<endl;
}
```

上面的输出也可以在同一行中串连，下面的输出语句与上例输出同样内容：

```
cout <<"My Name is Jone\n" <<"the ID is " <<2 <<endl;
```

也可以分在几行，提高可读性，下列语句与上例输出同样结果：

```
cout <<"My Name is Jone"        //行末无分号
     <<"the ID is "
     <<2
     <<endl;
```

cin可以和cout一样的方式调整行，它自动识别变量位置和类型。例如：

```
int i; float f; long l;
cin >>i >>f >>l;
```

cin能够知道抽取的变量之类型，它将对i,f,l分别给出一个整型、浮点型和长整型数。

2. 使用控制符

流的默认格式输出有时不能满足特殊要求，如：

```
double average = 9.400067;
cout <<average <<endl;
```

希望显示的是 9.40，即保留两位小数，可是却显示了 9.40007；默认显示 6 位有效位。

用控制符(manipulators)可以对 I/O 流的格式进行控制。控制符是在头文件 iomanip.h 中定义的对象。可以直接将控制符插入流中。常用控制符如表 2-4 所列。

表 2-4 I/O 流的常用控制符

控制符	描 述
dec	置基数为 10
hex	置基数为 16
oct	置基数为 8
setfill(c)	设填充字符为 c
setprecision(n)	设显示小数精度为 n 位
setw(n)	设域宽为 n 个字符
setiosflags(ios::fixed)	固定的浮点显示
setiosflags(ios::scientific)	指数表示
setiosflags(ios::left)	左对齐
setiosflags(ios::right)	右对齐
setiosflags(ios::skipws)	忽略前导空白
setiosflags(ios::uppercase)	十六进制数大写输出
setiosflags(ios::lowercase)	十六进制数小写输出

使用控制符时，要在程序的头上加头文件 iomanip.h。

3. 控制浮点数值显示

使用 setprecision(n)可控制输出流显示浮点数的数字个数。C++默认的流输出数值有效位是 6。

如果 setprecision(n)与 setiosflags(ios::fixed)合用，可以控制小数点右边的数字个数。setiosflags(ios::fixed)是设置确定的小数表示法。

如果与 setiosflags(ios::scientific)合用，可以控制指数表示法的小数位数。setiosflags(ios::scientific)是设置指数方式的小数表示法。

例如，下面的代码分别用浮点、定点和指数方式表示一个实数：

```
//* * * * * * * * * * * * * * * * * * * *
//* *          ch2_1.cpp                * *
//* * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>     //要用到格式控制符

int main()
{
   double amount = 22.0/7;
   cout <<amount <<endl;
   cout <<setprecision(0) <<amount <<endl
```

```
        <<setprecision(1) <<amount <<endl
        <<setprecision(2) <<amount <<endl
        <<setprecision(3) <<amount <<endl
        <<setprecision(4) <<amount <<endl;

    cout <<setiosflags(ios::fixed);
    cout <<setprecision(8) <<amount <<endl;

    cout <<setiosflags(ios::scientific) <<amount <<endl;

    cout <<setprecision(6);     //恢复成原默认设置
}
```

运行结果为：

```
3.14286
3
3
3.1
3.14
3.143
3.14285714
3.14285714e+00
```

该程序在32位机器上运行通过。

在普通表示的输出中，setprecision(n)表示有效位数。

第1行输出数值之前没有设置有效位数，所以用流的有效位数默认设置值6。第2个输出设置了有效位数0，C++最小的有效位数为1，所以作为有效位数设置为1来看待。第3～6行输出按设置的有效位数输出。

在确定表示的输出中，setprecision(n)表示小数位数。

第7行输出是与setiosflags(ios::fixed)合用。所以setprecision(8)设置的是小数点后面的位数，而非全部数字个数。

在指数形式输出时，setprecision(n)表示小数位数。

第8行输出用setiosflags(ios::scientific)来表示指数表示的输出形式。其有效位数沿用上次的设置值8。

小数位数截短显示时，进行4舍5入处理。

4. 设置值的输出宽度

除了使用空格来强行控制输出间隔外，还可以用setw(n)控制符。如果一个值的显示需要比setw(n)确定的字符数更多，则该值将显示所有字符。例如：

```
float amount = 3.14159;
cout <<setw(4) <<amount <<endl;
```

其运行结果为3.14159。它并不按4位宽度，而是按实际宽度输出。

如果一个值的字符数比setw(n)确定的字符个数更少，则在数字字符前显示空白。不同于其他控制符，setw(n)仅仅影响下一个数值输出，换句话说，使用setw设置的间隔方式并不保留其效力。例如：

```
cout <<setw(8)
     <<10
     <<20 <<endl;
```

运行结果为：

```
    10  1020
```

运行结果中的下横线表示空格。整数 20 并没有按宽度 8 输出。setw()的默认值为宽度 0，即 setw(0)，意思是按输出数值的表示宽度输出，所以 20 就紧挨着 10 了。若要每个数值都有宽度 8，则每个值都要设置：

```
cout <<setw(8) <<10
     <<setw(8) <<20 <<endl;
```

5. 输出八进制和十六进制数

3 个常用的控制符是 hex，oct 和 dec，它们分别对应十六进制、八进制和十进制数的显示。这 3 个控制符在 iostream.h 头文件中定义。例如：

```
//* * * * * * * * * * * * * * * * * * * * *
//* *          ch2_2.cpp              * *
//* * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>

int main()
{
   int number = 1001;
   cout <<"Decimal:" <<dec <<number <<endl
        <<"Hexadecimal:" <<hex <<number <<endl
        <<"Octal:" <<oct <<number <<endl;
}
```

运行结果为：

```
Decimal:1001
Hexadecimal:3e9
Octal:1751
```

1001 是一个十进制数，不能把它理解成十六进制或八进制数，因为它不是以 0x 或 0 开头。但在输出时，流根据控制符进行解释操作，使其按一定的进制来显示。

用头文件 iomanip.h 中的 setiosflags(ios::uppercase)可以控制十六进制数大写输出。例如，上例中增加一个头文件，对十六进制数进行大写控制，即：

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

//...

cout <<"Hexadecimal:" <<hex
     <<setiosflags(ios::uppercase)
     <<number <<endl;
```

便能得到十六进制数的大写表示：Hexadecimal:3E9。

6. 设置填充字符

setw 可以用来确定显示的宽度。默认时，流使用空格符来保证字符间的正确间隔。用 setfill 控制符可以确定 setw 所规定的间隔字符。Setfill 在头文件 iomanip. h 中定义。例如：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch2_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>

int main()
{
   cout <<setfill('*')
        <<setw(2) <<21 <<endl
        <<setw(3) <<21 <<endl
        <<setw(4) <<21 <<endl;
   cout <<setfill(' ');     // 恢复默认设置
}
```

运行结果为：

```
21
*21
**21
```

7. 左右对齐输出

默认时，I/O 流左对齐字串，右对齐数值。使用头文件 iomanip. h 中的 setiosflags (ios::left)和(ios::right)标志，可以控制输出对齐。例如：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch2_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>

void main()
{
   cout <<setiosflags(ios::right)
        <<setw(5) <<"1"
        <<setw(5) <<"2"
        <<setw(5) <<"3" <<endl;
   cout <<setiosflags(ios::left)
        <<setw(5) <<1
        <<setw(5) <<2
        <<setw(5) <<3 <<endl;
}
```

运行结果为：

```
    1    2    3
1    2    3
```

8. 强制显示小数点和符号

当程序输出下面的代码时：

```
cout <<10.0/5 <<endl;
```

默认的 I/O 流会简单地显示 2，而非 2.0，因为除法的结果是精确的。当需要显示小数点时，可以用 ios::showpoint 标志。例如：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch2_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>

int main()
{
   cout <<10.0/5 <<endl;

   cout <<setiosflags(ios::showpoint)
        <<10.0/5 <<endl;
}
```

运行结果为：

```
2
2.00000
```

默认时，I/O 流仅在负数之前显示值的符号，根据程序的用途，有时也需要在正数之前加上正号，可以用 ios::showpos 标志。例如：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch2_6.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>

int main()
{
   cout <<10 <<"  " <<-20 <<endl;

   cout <<setiosflags(ios::showpos)
        <<10 <<"  " <<-20 <<endl;
}
```

运行结果为：

```
10  -20
```

```
+10  -20
```

2.7 printf 与 scanf

printf 和 scanf 是标准输入输出函数。它们是 C 程序中所使用的，在头文件 stdio.h 中声明了这两个函数。在 C++面向对象程序设计中，I/O 流代替了它们(见第 19 章)。在过程化程序设计中，printf 和 scanf 在使用习惯上，可作为 C++流的一个补充。

1. printf 函数

(1) printf 函数的一般格式为：

```
printf(格式控制字符串,输出项 1,输出项 2, ...)
```

括号中的格式控制字符串和输出项都是函数参数。printf()函数的功能是将后面的参数按给定的格式输出。

格式控制字符串中有格式说明也有普通字符。格式说明由“%”和格式字符组成，如%d，%f 等。它的作用是将输出的数据转换成指定的格式输出。普通字符就是原样输出的字符。输出项 n 是需要输出的一些数据，可以是表达式。例如：

```
#include <stdio.h>
void f()
{
   int a=10,b=20;
   printf("%d,%d",a,b);
}
```

在上例中，双引号中的字符除了两个“%d”外，还有逗号“,”字符，它是普通字符。a，b 的值分别为 10，20，输出为：

```
10,20
```

(2) %d 格式符

%d 用来输出十进制整数，可以有长度修饰。例如：

```
int a=28,b=38;
long c=289868;
printf("%5d,%5d\n%ld\n",a,b,c);
printf("%3ld\n%7ld\n%d\n",c,c,c);
```

输出结果为：

```
   28,   38
289868
289868
 289868
27724
```

其中%5d 表示输出宽度为 5，%ld 表示输出为长整型，%3ld 表示输出宽度为 3 的长整型数，如果长整型数的位数多于 3，则按长整型的位数输出。如果一个长整型数只按整型数

输出，则会引起整数截断，即输出长整型数低位 2 个字节的值。上例中最后一个数的输出就是截断长整型数 289868 得到的输出。时下的极大多数编译器都是 32 位的，即整型数在内部用 32 位表示，长整型就是整型，%ld 与%d 效果一样，因而最后的输出不会进行截断。

(3) %o 和%x 格式符

%o 和%x 用来以八进制和十六进制数输出。八进制和十六进制数都是无符号整数，输出时不带符号。例如：

```
int a = -3;
printf("%d,%o,%x,%X,%6x\n",a,a,a,a,a);
```

输出结果为：

```
-3,177775,fffd,FFFD, fffd
```

当用%X 时，输出十六进制数时用大写字母，用%x 时，输出用小写字母。

八进制数和十六进制数同样可以用"%lx"输出长整型数，另外也可以指定输出字段的宽度。

(4) %u 格式符

%u 用来以无符号十进制整数方式输出。有符号整数(int 型)可以%u 格式输出，无符号整数(unsigned 型)可以%d 格式输出或以%o，%x 格式输出。另外，还可以指定格式宽度。例如：

```
unsigned int a = 65533;
int b = -2;
printf("a = %d,%o,%x,%u,%7u\n",a,a,a,a,a);
printf("b = %d,%o,%x,%u,%6u\n",b,b,b,b,b);
```

运行结果为：

```
a = -3,177775,fffd,65533, 65533
b = -2,177776,fffe,65534, 65534
```

(5) %c 格式符

%c 用来以字符方式输出。如果一个整数，其值在 0～255 之间，也可以字符方式输出。它们都可以指定格式宽度。例如：

```
char ch = 'a';
int a = 65;
printf("%c,%d,%3c\n",ch,ch,ch);
printf("%c,%d,%3d\n",a,a,a);
```

输出结果为：

```
a,97,  a
A,65,  65
```

(6) %s 格式符

%s 用来以字符串格式输出。可以指定格式宽度。如果字符串长小于指定的宽度时，可以选择左对齐和右对齐。另外还可以选择字符串的前 n 个字符。例如：

```
printf("%s","Hello\n");
```

```
printf("Hello\n");
printf("%3s,%-5.3s,%5.2s\n","Hello","Hello","Hello");
```

输出结果为：

```
Hello
Hello
Hello, Hel  ,   He
```

输出字符串时，要求 printf()的第 2 个参数是字符串，但不一定是字符串常量。我们将在第 8 章中介绍字符串变量的概念。

如果输出的只有一个字符串，可以省略格式参数，如上例中第二条语句，因为格式参数本身可以是原样输出的普通字符串。

“%-5.3s”中的负号表示左对齐，如果没有负号，则默认为右对齐。5 表示格式宽度，3 表示截取字符串中 3 个字符。

(7) %f 格式符

%f 用来以小数方式输出。可以指定格式宽度，也可以指定小数位数，还可以规定左对齐和右对齐。例如：

```
float x=123.456;
double y=321.654321;
long double z=3.141592653;
printf("%f,%-7.2f,%10.4f\n",x,x,x);
printf("%lf,%-7.2lf,%10.4lf\n",y,y,y);
printf("%Lf,%-7.2Lf,%10.4Lf,%14.10Lf\n",z,z,z,z);
```

输出结果为：

```
123.456001,123.46 ,   123.4560
321.654321,321.65 ,   321.6543
3.141593,3.14  ,     3.1416,  3.1415926530
```

以“%f”格式输出时，默认的小数位数为 6，所以输出 123.456001。x 值的有效位是 7 位，超过了 7 位，就不能保证其精度了，所以输出结果并不是想象中的 123.456000。“%-7.2f”表示左对齐，总长度 7 位，小数位数 2 位。

“%lf”是 double 型输出，有效位 15 位。“%Lf”是 long double 型输出。由于默认小数位数是 6，所以看到的输出中，小数位数只有 6 位。

(8) %e 格式符

%e 用来以指数方式输出浮点数。指数的输出格式要求小数点前必须有而且只有一位非 0 数字，默认的小数位数为 6，默认的指数位数为 2(不包括 e+或 e-)。例如：

```
float x=123.456;
double y=321.654321;
long double z=3.141592653e+123;
printf("%e,%-7.2E,%10.4e\n",x,x,x);
printf("%le,%-7.2lE,%10.4le\n",y,y,y);
printf("%Le,%-7.2LE,%10.4Le,%14.10Lf\n",z,z,z,z);
```

输出结果为：

```
1.234560e+02,3.22E+02,1.2346e+01
3.216543e+02,3.22E+02,3.2165e+02
3.141593e+123,3.14E+123,3.1416e+123,3.1415926530e+123
```

其中，e 或 E 控制指数形式的 e 以小写或大写方式输出。

此外，还有%g 格式符，用以输出浮点数，它根据数值的大小，自动选取 f 格式或 e 格式（较短的一种）。

如果要输出%本身，则双写%。例如：

```
printf("%f%%",1.0/3);
```

输出结果为：

```
0.333333%
```

2. scanf 函数

scanf 的一般形式为：

```
scanf(格式控制字符串,地址1,地址2, ...)
```

格式控制字符串的含义同前，地址 n 是变量的地址。

%d 用以输入整数，可以带 l 表示长整数，带 h 表示短整数。

%c 用以输入字符。

%o，%x 用以输入八进制数和十六进制数。%lo 和%lx 分别表示长八进制数和长十六进制数。

%f 用以输入浮点数，%lf 和%Lf 分别表示输入 double 型数和 long double 型数。

%e 与%f 作用相同。

%s 用以输入字符串，以非空字符开始，以空字符或回车结束。

例如：

```
int a,b;
char ch1,ch2;
float f,g;
scanf("%d %d",&a,&b);
scanf("%c%c",&ch1,&ch2);
scanf("%f,%f",&f,&g);
```

输入时：

```
23 456
ab
23.6712,612.97
```

两个输入项之间一般用空格分开。如果规定了分隔字符，如逗号“,”，则必须以逗号分开。如上例中，第 1 个 scanf 语句数值之间必须以空格或回车隔开，第 3 个 scanf 语句必须以逗号隔开。

在用“%c”格式输入时，空格字符和转义字符都作为有效字符输入。如上例中，执行第 2 个 scanf 语句时，若输入“a b”，则'a'赋给 ch1，' '（空格）赋给 ch2，而'b'被忽略。

scanf()中后面的地址参数是重要的，不能是变量名，否则会将输入值存放在变量值作为地址的内存空间中，导致意想不到的运行异常。

小结

变量是程序分配给某个内存位置的名字，它可以存放信息。程序在使用变量前，必须先说明变量名和变量类型。

不同的变量不能同名，变量名应该尽量反映出变量的用途，以增强程序的可读性。

在程序运行中，常量的值不可改变。常量也有各种数据类型，也占有存储空间。各种数据类型的数据表示有一定的范围，越过了该范围，C++就要对该数据进行截取，使得数据不再正确。

利用 cout 可以输出各种数据类型的数据，可以多种方式在屏幕上显示输出信息(包括特殊符号)。

C++兼容 C 的库函数，所以 Printf()和 Scanf()也可照常使用。

练习

2.1 设整数 42486，请定义一个变量，初始化之，并以八进制与十六进制数输出。如果将该整数定义成无符号短整数，当以有符号数输出时，结果是什么，请用补码概念解释。

2.2 取圆周率为 3.1415926，分别输入半径为 40 和 928.335，求圆面积，要求各数据按域宽 10 位输出，先输出圆周率和半径，再输出其面积。

2.3 将常数 e(2.718281828)作为常量定义，然后输出其 10 位有效位数的浮点数、定点方式和 8 位小数位表示的数，以及指数形式和 8 位小数位表示的数。

2.4 设学生常数为 500，编程输出下列结果(引号也要输出)。

```
"How many students here?"
"500"
```

2.5 用 sizeof 操作符，求出表 2-2 中各数据类型的字节长度，并按：

```
size of char      1 byte
size of int       2 byte
...
```

的格式打印输出。

2.6 读下列程序。

(1) 运行时，输入 6,6,8 三个数，写出运行结果；

(2) 解释该程序做了什么，程序的书写为什么要分成几段，各起什么作用？

(3) 用 cout 和 cin 代替 printf()和 scanf()函数，改写程序(注意格式)。

(4) 将求面积部分的程序以调用函数的方式来完成，则函数声明、定义和调用作如何修改？

```
#include <stdio.h>
#include <math.h>
```

```
int main()
{
   float a,b,c,s,area;
   printf("please input 3 sides of one triangle:\n");
   scanf("%f,%f,%f",&a,&b,&c);

   s=(a+b+c)/2;
   area=sqrt(s*(s-a)*(s-b)*(s-c));

   printf("a=%7.2f,b=%7.2f,c=%7.2f\n",a,b,c);
   printf("area of triangle is %10.5f",area);
}
```

2.7 读下列程序,写出运行结果。

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

int add(int x, int y);

int main()
{
   cout <<"In main():\n";
   int a,b,c;
   cout <<"Enter two numbers:\n";
   cin >>a >>b;

   cout <<"\nCalling add():\n";
   c=add(a,b);

   cout <<"\nBack in main():\n";
   cout <<"c was set to " <<c <<endl;

   cout <<"\nExiting...\n";
   getche();
}

int add(int x, int y)
{
   cout <<"In add(),received " <<x <<" and " <<y <<endl;
   cout <<"and return " <<x+y <<endl;
   return x+y;
}
```

2.8 以函数调用的方式,求圆柱体的体积。

主函数中先输入圆柱体的半径和高,然后调用求体积的函数,最后输出结果。

第3章　表达式和语句

程序是一些按次序执行的语句。执行语句是为了要完成某个操作，修改某个数据。程序中大部分的语句是由表达式构成的，因为表达式直截了当地返回值。正因为如此，表达式是C++编译器处理的重要环节。学习本章后，要求理解表达式和语句的概念，掌握表达式中各种运算符的功能与特点，明白产生副作用的原因。更好地理解C++语言的强大与灵活。

3.1　表达式

1. 表达式概述

表达式是操作符、操作数和标点符号组成的序列，其目的是用来说明一个计算过程。

表达式可以嵌套，例如：2+3+(5 * sizeof(int))/345。

表达式根据某些约定、求值次序、结合和优先级规则来进行计算。

所谓约定，即类型转换的约定。例如：

```
float a;
a = 5/2;
```

结果a得到值为2。5/2是整数除法取整，因为5和2都是整数，不会由于a是float型而轻易改变运算的性质。

所谓求值次序，是指表达式中各个操作数的求值次序视编译器不同而不同。见3.9节。

所谓结合性，是指表达式中出现同等优先级的操作符时，该先做哪个操作的规定。例如：

```
d = a + b - c;          //C++规定，加减法先左后右.先做a + b,其结果再减去c
d = a = 3;              //C++规定，等号是先右后左.先做a = 3,其结果再赋给d
```

所谓优先级，是指不同优先级的操作符，总是先做优先级高的操作。例如：

```
d = a + b * c;          //乘法优先级比加法高.先做b * c,其结果再与a相加
```

2. 左值和右值

左值(left value，缩写为lvalue)是能出现在赋值表达式左边的表达式。左值表达式具有存放数据的空间，并且存放是允许的。例如：

```
int a = 3;              //a是变量，所以a是左值
const int b = 4;        //b是常量，所以b不是左值
```

显然常量不是左值，因为C++规定常量的值一旦确定是不能更改的。

右值(right value，缩写为rvalue)只能出现在赋值表达式的右边。左值表达式也可以作为右值表达式。例如：

```
int a,b = 6;
a = b;                    //b 是变量,所以是左值,此处作为右值
a = 8;                    //8 是常量,只能作右值,不能作为左值
```

表达式可产生左值、右值或不产生值。例如：

```
int a;
(a = 4) = 28;             //ok:a = 4 是左值表达式,可以被赋以值 28
void f(){return ;}        //此为函数定义,该函数不返回任何值
```

28 是右值表达式，而 a=4 是左值表达式(C++的语法规定)，所以可以放在赋值语句的左边。该语句表示 a 的值用 28 替代刚刚赋给的值 4。

函数定义本身不是表达式，它说明了一个不返回值的函数。对函数 f()的调用是语句，它实施了一个没有返回值的操作。

3. 优先级和结合性

表 3-1 对操作符的优先级和结合性作了小结。表中包含了 C++所有的操作符，共有 16 级优先级。表中的操作符如重复出现，则第 1 次出现的是单目运算符，第 2 次出现的双目运算符。每一级的结合性，不是从左到右就是从右到左。表达式中，在没有括号的情况下，这些规则决定了表达式运算的次序。

每一级中的操作符是同优先级的。

表 3-1 C++操作符的优先级与结合性

优先级	操 作 符	结合性
1	( ) [ ] -> ::	左→右
2	! ~ + - ++ -- & * (强制转换类型) sizeof new delete	右→左
3	.* ->*	左→右
4	* / %	左→右
5	+ -	左→右
6	<< >>	左→右
7	< <= >= >	左→右
8	== !=	左→右
9	&	左→右
10	^	左→右
11	|	左→右
12	&&	左→右
13	||	左→右
14	?:	右→左
15	= *= /= += -= |= <<= >>=	右→左
16	,	左→右

4. 语句与块

C++中所有的操作运算都通过表达式来实现。由表达式组成的语句称为表达式语句，它由一个表达式后接一个分号“;”组成。

通过计算表达式即执行了表达式语句。大多数表达式语句为赋值语句和函数调用。

语句是用来规定程序执行的控制流。在没有跳转和分支(见第 4 章)的情况下，语句将按照其在源程序中出现的次序顺序执行。

语句可以是空语句。空语句是只有一个分号而没有表达式的语句，其形式为：

```
;
```

它不产生任何操作运算，只作为形式上的语句，被填充在控制结构中。例如：

```
if(x>9)
  ;
else
   cout <<"not large than 9\n";
```

例中判断 x 是否大于 9，如果大于 9，不做任何事，否则输出“not large than 9”和回车。

块(或称复合语句)是指括在一对花括号{}里的语句序列。从语法上来说，块可以被认为是单个语句。例如：

```
if(x>9)
{
   cout <<"The number is perfect.\n";
   cout <<"It is large than 9\n";
}
else
{
   cout <<"not large than 9\n";
}
```

x 若大于 9，则执行两条输出语句，否则，输出“not large than 9”和回车。这两条执行语句必须放在花括号中，因为 if 与 else 之间只能容纳一条语句，或一个语句块。而 else 后面的花括号则可以省略。此外，块还可以嵌套。

3.2 算术运算和赋值

1. 操作符种类

C++提供了算术运算符+，-，*，/，%。

+，-，* 是通常意义的加、减、乘法。

/对于整型数则为除法取整操作。例如，5/2 得到结果 2。

/对于浮点数则为通常意义的除法。例如，5.0/2.0 得到结果为 2.5。

由此可见，/操作符可以对不同的数据类型进行不同的操作。事实上，+，-，*，/，%对不同数据类型的操作都不同。如整数加法是将 2 个整型相加，而浮点数加法是将 2 个浮点数相加，相加的具体操作(在机器指令级上)浮点和整数是不同的。

%只能对整型数进行操作。其操作意义为取余。例如，5%2 得到结果为 1。

%不允许对浮点数操作，如果对浮点数操作，则会引起编译错误。

2. 赋值缩写

算术表达式的赋值表示为：

```
int x,y,z;
x = y * z;
x = y/z;
x = y + z;
x = y - z;
x = y % z;
```

当一变量既出现在表达式的左边又出现在右边时，可以缩写。例如：

```
x = x * y;      缩写为:     x * = y;
x = x + y;      缩写为:     x + = y;
x = x - y;      缩写为:     x - = y;
x = x/y;        缩写为:     x/ = y;
x = x % y;      缩写为:     x % = y;
```

赋值以及缩写都要求左边的表达式为左值，即 x 为左值。

赋值构成一个表达式，因而它具有值。赋值表达式的值为赋值符左边表达式的值。例如：

```
cout <<(x = 5) <<endl;
```

将输出 5。同时 x 被赋予值 5。

赋值表达式为左值。例如：

```
(x = max(5,7)) + = 3;
```

该语句先将 max(5,7)函数调用的值赋给 x，然后在此基础上增值 3。它等价于：

```
x = max(5,7) + 3;
```

缩写格式通常更有效，可读性也不差。缩写与不缩写的差别在于缩写式中左边的变量只出现一次，而不缩写的式中出现 2 次。例如：

```
(x = max(5,7)) = (x = max(5,7)) + 3;
```

上式是合法的表达式，它与前面讨论的表达式不同之处在于 max(5,7)调用了 2 次。如果函数有副作用(见 3.9 节)，则赋值与赋值缩写是不同的。

3. 溢出

进行算术运算时，很可能溢出结果。发生溢出是由于一个变量被赋予一个超出其数据类型表示范围的数值。数值溢出是不会引起编译错误的，只要分母不为 0 也不会引起除 0 运行故障，但会使运行结果发生偏差。

例如，在 16 位机器上进行下面的操作：

```
int weight = 42896;
```

在16位机器中将不能得到值42896，而是－22640。因为有符号整数的表示范围是－32768～32767，所以它只能得到42896的补码－22640(42896－65536)。

一个整数类型的变量，用任何一个超过表示范围的整数初始化，得到的值为用该整数范围作模运算后的值。例如：

```
int weight = 142896;
```

则当weight是2字节整型数时，得到值为11824。因为142896＝2×65536＋11824。而142896－3×65536＝－53712，该数不在有符号整型数表示范围内。

3.3 算术类型转换

C++遇到两种不同数据类型的数值进行运算时，会将两个数作适当的类型转换，然后再进行运算。转换的方向见图3-1。

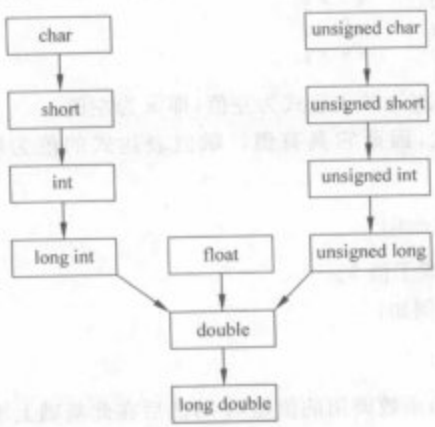

图3-1 类型转换的方向

如果一个char型数和一个int型数相加，则将char型数转换成int型数，然后进行运算。因为在图3-1中，char有向int转换的趋势。如果一个long int型数和一个float型数相加，则先将两个数据类型都转换成double型数，然后进行运算。如果一个int型数和一个unsigned long型数相乘，则先将两数都转换成double型数，然后进行运算。

转换总是朝表达数据能力更强的方向，并且转换总是逐个运算符进行的。例如：

```
float f = 3.5;
int n = 6;
long k = 21;
double ss = f * n + k/2;
```

ss将会得到结果31。计算ss时，首先将f(float型)和n(int型)转换成double型数，算得21，然后计算k/2得整除运算结果10(long int型)，再将long int型的数字10转换成double型数。21和10两个数相加，得到最后结果31。

数据运算过程中自动进行的类型转换称为隐式类型转换。上例的表达式运算过程中进行的数据类型转换就是隐式转换。

有时候，我们会面临下面计算结果不准确的问题：

```
long m = 234 * 456/6;
```

即发现 m 为－4061，而不是 17745。原因是语句先进行 int 型数的乘法运算，结果仍以 int 型数保留起来：234×456＝106704＝2×65536－24368，取模之后得到－24368。该数参加整除运算：－24368/6 得－4061(取整)。由于中间的结果被截断，所以，最后的结果是错的。如果让第 1 次乘法的结果以 long 型数保留下来，就能得到正确的结果。这就要求参加乘法运算的 2 个数至少有一个为 long 型数。

例如，将其中之一标识以 L 或 l(long)，则可保证其正确：

```
cout <<234 * 456L/6 <<endl;
```

输出结果为：

```
17784
```

还可以将整型数强制转换为 long 型：

```
cout <<(long)234 * 456/6 <<endl;
```

该语句使 234 成为 long 型数，与整数 456 相乘，先隐式转换，再相乘，得到一个 long 型数 106704。再与 6 相除取整，从而得到正确结果。

强制转换又称显式转换，其语法是在一个数值或变量前加上带括号的类型名。也可以类型名后跟带括号的数值或表达式。如上面语句也可以写成：

```
cout <<long(234) * 456/6 <<endl;
```

如果类型名是带类型修饰的，则要给类型名加括号。例如：

```
cout <<(unsigned long)234 * 456/6 <<endl;
cout <<unsigned long(234) * 456/6 <<endl; //error
```

请注意下面语句不能产生所期望的效果：

```
cout <<long(234 * 456)/6 <<endl;
```

该语句首先执行括号里的乘法，得到一个 int 型整数(已被取模)－4061，然后强制转换为 long 型数，再参加除 6 取整运算，所以得不到正确结果。

3.4 增量和减量

增量和减量操作符表示为：＋＋和－－。

增量操作表示加 1，减量操作表示减 1。例如：

```
a++;    //等价于 a=a+1;
++a;    //等价于 a=a+1;
a--;    //等价于 a=a-1;
--a;    //等价于 a=a-1;
```

增量操作符有前增量与后增量之分。前增量操作 ++a 的意义为：先修改操作数使之增1，然后将增1过的a值作为表达式的值。而后增量操作 a++ 的意义为：先将变量a的值作为表达式的值确定下来，再将a增1。对于增量和减量操作符，它要求操作数是左值，因为操作数的值要发生变化。例如：

```
int a = 3;
int b = ++a;          //相当于 a = a + 1; b = a;
cout <<a <<"    "  <<b <<endl;
int c = a++;          //相当于 c = a; a = a + 1;
cout <<a <<"  "  <<c <<endl;
```

输出的结果为：

```
4  4
5  4
```

b被赋予了4，因为前增量操作先将a自增为4，然后作为表达式赋值。C被赋予了4，因为后增量操作使表达式的值(a)先赋给c，然后a再自增为5。

由于前增量操作返回的值即修改后的变量值，所以返回的仍是一个左值。例如：

```
int a = 3;
++(++a);              //ok: ++a 是左值，可以接着做 ++ 操作
```

例中得到的a的值为5。

由于后增量操作返回的值是原先a的值，尔后a的值已经发生变化，故返回的不能是当前a，只能是过去的a值，不能是左值。例如：

```
int a = 3;
++(a++);              //error: a++ 不是左值
```

相应的，有前减量 --a 和后减量 a-- 。例如：

```
int a = 3;
int b = --a;          //相当于 a = a - 1; b = a;
cout <<a <<"  " <<b <<endl;
int c = a--;          //相当于 c = a; a = a - 1;
cout <<a <<"  " <<c <<endl;
```

输出的结果为：

```
2  2
1  2
```

由于增量与减量操作包含有赋值操作，所以操作数不能是常量，它必须是一个左值表达式。例如：

```
3++;                  //error
```

增量与减量操作符是两个＋或两个－的一个整体，中间不能有空格。如果有多于两个＋或两个－连写的情况，则编译首先识别前面两个＋或－为增量或减量操作符。

例如，对于“int a＝1，b＝5，c；”的变量定义，下面5个表达式，有些不允许：

```
c = a + b;          //ok:c = 6
c = a ++ b;         //error:编译接收为 a ++ b
c = a +++ b;        //ok:编译接收为 a ++ + b
c = a ++++ b;       //error:编译接收为 a ++ ++ b
c = a +++++ b;      //error:编译接收为 a ++ ++ + b
```

第 2 行中,编译将其理解为 a ++ b。由于 ++ 操作是单目运算符,所以该表达式语法错误。若要合法,应写成 a + + b,表示 a 加上正 b。

第 3 行中,编译将其理解为 a ++ ++ b。同样由于 ++ 是单目操作符,引起编译错误。若要合法,应写成 a +++ + b,表示 a ++ 加上正 b。

第 4 行中,编译将其理解为 a ++ ++ + b。由于 a ++ 是个非左值表达式,所以中间的 ++ 操作符是非法的。若要合法,应写成 a +++ ++ b 或者 a ++ + ++ b,表示 a ++ 加上 ++ b。

3.5 关系与逻辑运算

1. 运算符

关系操作符有:

```
比较( == )
大于(>)
小于(<)
大于等于(>=)
小于等于(<=)
不等于(!=)
```

逻辑运算符有:

```
非(!)
逻辑与(&&)
逻辑或(||)
```

对于>=,<=,!=,==,&&,||都是一个操作符的整体,所以中间不能有空格,而且前 3 个操作符中的字符次序不能颠倒。例如:下面的写法都是不合法的。

```
=<, =>, =!, <  =, >  =, =  =, !  =
```

2. 比较运算符

比较(==)和赋值(=)是两个不同的操作,所以用的操作符也不同。比较用于测试给定的 2 个操作数是否相等。例如:

```
if(x == 999)
  cout <<"x is 999\n";
```

C++中,表达式都产生值,赋值操作符产生的值正是所赋的值。而比较操作符产生的值是比较的结果,可能是 0 或 1,即假或真。

真和假是逻辑值。在C++中，假意味着0，真意味着非0。所以，任意一个非0数都是真，表示为逻辑值就是1。例如：

```
x = somevalue;
if(x = 9)
   cout <<"x is not 0\n";
```

例中，不管x的初值是什么，总是执行cout语句。因为x=9是赋值表达式，其表达式的值是所赋的值9，而9为非0值，所以if语句的条件为真，所以总是执行cout语句。又例如：

```
x = somevalue;
if(x = 0)
   cout <<"x is 0\n";
```

例中，不管x以前是什么值，总是不会执行cout语句。因为x=0是赋值表达式，并且其值为0，为假。

这一般不是编程的本意，但由于**=与==经常不小心搞错**，使得程序不正确地运行。在BC和VC编译器中，在像if语句这样的条件表达式中，遇到=时都会给予警告。这时，就应该有所警觉，以免程序错误地执行。

3. 不等于运算符

当要测试一些东西不是真时，可以使用不等于操作符。例如，如果要在一些东西是真时在屏幕上显示一则消息，则可以用如下语句：

```
if(x!= 9)
   cout <<"x isn't 9\n";
```

要注意的是，如果颠倒!=，则意义完全不同：

```
if(x = !9)
   cout <<"x isn't 9\n";
```

该if条件表达式是一个赋值语句，!9为非真，即0。所以该条件表达式相当于if(x=0)，于是cout语句永远也不会执行。

4. 嵌入赋值

有时候，需要将一个函数值赋给一个变量，然后比较该变量的值与预定值是否相等。例如：

```
x = func();
if(x == somevalue)
   //语句
```

上面的代码与下面的代码等价：

```
if((x = func()) == somevalue)
   //语句
```

因为要给x赋值，然后确定x的值，所以先进行赋值。而赋值表达式的值即x的值可以

作为比较的操作数。这里，由于==操作比=操作优先级高，所以需要额外加一个括号。

这种赋值紧接着比较的表达式称为嵌入赋值，它经常在程序样例中看到。

5. 逻辑非运算符

！改变条件表达式的真假值，即逻辑运算的“非”。原来是0，则！0为1；原来为非0，则！操作使之变为0。例如：

```
if(!(x==9))
   cout <<"x is not 9.\n";
```

因为==比！优先级低，所以额外的括号是需要的。包围“x==9”的括号使比较先进行，然后再做非操作。

6. 逻辑运算

&&和||是两个逻辑运算符，它们的意义为求两个条件表达式的逻辑与和逻辑或。

例如，下面的代码为根据室温打印一则消息：

```
int temp=90,humi=80;

if(temp>=80 && humi>=50)
   cout <<"wow,it's hot!\n";
if(temp<60||temp>80)
   cout <<"the room is uncomfortable.\n";
```

输出结果为：

```
wow,it's hot!
the room is uncomfortable.
```

因为&&比>=优先级低，所以if中的条件表达式为先求temp>=80和humi>=50，然后进行逻辑与&&运算。

7. 短路表达式

如果多个表达式用&&连接，则一个假表达式将使整个连接都为假（此处需要数理逻辑知识）。例如：

```
int n=3,m=6;

if(n>4 && m++<10)
   cout <<"m should not changed.\n";

cout <<"m= " <<m <<endl;
```

输出结果为：

```
m= 6
```

由于n>4的比较值为0，所以整个if条件表达式的值不用看后面就知道为0。C++利用这个特点以产生高效的代码。所以，后面的表达式不被执行。这样，m的值还是6而不是

7。知道了短路表达式在C++中的处理方式，就可以在编写程序时，不但避免不必要的错误，而且还可利用它。例如：

```
if(b!=0 && a/b>2)
  //语句
```

if条件表达式中的b!=0若成立，才会执行后面的关系运算，做分母是b的除法。否则，跳过整个条件语句。

→ 在程序中，如果碰到除0运算，则运行发生异常。如果没有定义异常处理(见第21章)，则整个程序终止运行。

同理，如果多个表达式用||连接，则一个真表达式将使整个连接都为真。例如：

```
if(temp<60||temp>80)
  cout <<"the room is uncomfortable.\n";
```

例中如果temp<60成立，则不会进行temp>80的关系比较，直接执行输出语句。

3.6 if语句

1. if语句

if语句的语法为：

```
if(条件表达式)
  语句;
```

或：

```
if(条件表达式)
{
  语句;
}
```

它的意义为：如果条件表达式进行一次测试，且测试为真，则执行后面的语句。C++中的if语句与其他计算机语言的if语句区别不大。

如果if语句只控制一条语句，则包围该语句的花括号不是必需的。

例如，下面的程序等待键入一字符，如果是'b'，则响铃：

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

void main()
{
  cout <<"please input the b key to hear a bell.\n";
  char ch=getche();
  if(ch=='b')
    cout <<'\a';
}
```

2. 空语句

编译器必须在if条件表达式的后面找到一个作为语句结束符的分号";"，以标志if语

句的结束。这样，如果是下面的代码：

```
if(条件表达式);          //空语句做 if 中的语句

语句;
```

则不管条件表达式为真为假，总是接着执行语句。

3. if...else 语句

语法为：

```
if(条件表达式)
   //语句 1;
else
   //语句 2;
```

见图 3-2 流程图描述。

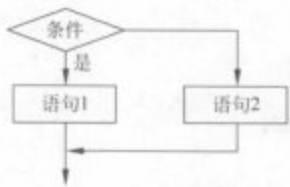

图 3-2　if...else 语句结构

if...else 语句让用户在程序中构造“不是.../就是...”的判断点。例如：

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

int main()
{
   cout <<"please input the b key to hear a bell.\n";
   char ch = getche();

   if(ch == 'b')
      cout <<'\a';
   else
      if(ch == '\n')
         cout <<"what a boring select on...\n";
      else
         cout <<"bye!\n";
}
```

该程序等待输入一个字符，如果是'b'，则响铃，否则，如果是回车，则输出“糟糕……”，不是，则输出“再见”。

else 后面的代码规则和 if 后面的代码规则一样。

else 后跟语句。既然是语句，就可以是 if 语句。上例中跟的就是 if 语句。

4. 解决二义性

如果有下面的代码：

```
int x = 20;
if(x>= 0)
   if (x<50)
      cout <<"x is ok\n";
else
   cout <<"x is not ok\n";
```

编译并不看程序的缩进格式，而只关心语法。该程序能打印什么呢？

该程序有两种解释：

一种是把第 2 个 if 与 else 配对：

```
int x = 20;
if(x>= 0)
{
   if (x<50)
      cout <<"x is ok\n";
   else
      cout <<"x is not ok\n";
}
```

另一种是把第 2 个 if 包在一个程序块中：

```
int x = 20;
if(x>= 0)
{
   if (x<50)
      cout <<"x is ok\n";
}
else
      cout <<"x is not ok\n";
```

C++规定，if...else 语句成对的规则是：else 连接到上面第 1 个没有配对的且为可见的 if 上。所以上例的 else 应属于第二个 if 语句，即第一种解释。

又例如：

```
if(条件)              //第 1 个 if
   if(条件)           //第 2 个 if
   {
      if(条件)        //第 3 个 if
         语句;
   }
   else
     语句;
```

上例的 else 连到第 2 个 if 上，因为第 3 个 if 不可见。第 2 个 if 是 else 最先碰到的没有配对过的 if。

对于程序的可读性要求来说，无论哪一种情况，都不理想，更好的方法是改变程序的实现。例如，将前面的代码合并两个 if 为下面的代码：

```
int x = 20;
if(x>= 0 && x<50)
  cout <<"x is ok\n";
else
  cout <<"x is not ok\n";
```

3.7 条件运算符

条件运算符的语法为：

```
(条件表达式)?(条件为真时的表达式):(条件为假时的表达式)
```

例如：

```
x = a<b? a: b;
```

条件运算符构成一个表达式。它是C++中唯一一个3元运算符，它们之间用“?”和“：”隔开。上例中，把a和b中较小的值赋给x。该例是if...else语句的一个替代：

```
if(a<b)
  x = a;
else
  x = b;
```

条件运算符构成表达式，它是有值的。而if...else语句不能有值，所以if...else语句不能替代条件运算符。例如，下面的代码不能由if...else替代：

```
cout <<(a<b? a:b) <<endl;
```

输出语句要打印一个值，该值是a与b的较小值。由于<<的优先级高于条件运算符，所以输出语句中要将条件运算符构成的表达式用括号括起来。

条件运算符表达式的值与测试值没有直接的关系。例如：

```
cout <<(number == 1?"file":"files") <<endl;
```

该输出语句中，条件运算符表达式的条件若成立，取值为“file”，否则，取值为“files”。其中，条件为两个整型数的比较，而表达式的值为字符串。

条件运算符可以嵌套。例如：

```
x>y?"great than":x == y?"equal to":"less than"
```

它等价于：

```
(x>y)?"great than":((x == y)?"equal to":"less than")
```

当x>y时，值为“great than”，x==y时，值为“equal to”，否则，值为“less than”。条件运算符的嵌套可读性不够好。

在一个条件运算符的表达式中，如果后面两个表达式的值类型相同，均为左值，则该条件运算符表达式的值为左值表达式。例如：

```
int x = 5;
long a,b;
```

```
(x?a:b) = 1;          //ok:因为 a 和 b 都是左值
(x?x:a) = 2;          //error:x 和 a 不同类型.编译器将其解释为(long)x 和 a
(x == 2?1:a) = 3;     //error:1 非左值
```

"(x?a:b)=1"表示当 x 为 0 时,b=1,否则 a=1。这里的括号是必需的,否则将被看作 x?a:(b=1)。"(x?x:a)=2"中,尽管 x 是左值,a 也是左值,但 x 与 a 不同类型,条件运算符要对其进行操作数的隐式转换,使之成为相同的类型。任何被转换的变量都不是左值。

→ 在 C 中,条件运算符是不能作左值的,所以"(x?a:b)=1;"将通不过编译。

3.8 逗号表达式

逗号表达式的语法为:

```
表达式 1, 表达式 2, ...表达式 n
```

C++顺序计算表达式 1,表达式 2,……,表达式 n 的值。例如:

```
int a,b,c;
a=1, b=a+2, c=b+3;
```

由于按顺序求值,所以能够保证 b 一定在 a 赋值之后,c 一定在 b 赋值之后。该逗号表达式可以用下面 3 个有序的赋值语句来表示:

```
a=1;
b=a+2;
c=b+3;
```

逗号表达式是有值的,这一点是语句所不能代替的。逗号表达式的值为第 n 个子表达式的值,即表达式 n 的值。例如:

```
int a,b,c,d;
d=(a=1,b=a+2,c=b+3);
cout <<d <<endl;
```

输出结果为:

```
6
```

上例中输出的结果 d 即为 c 的值。

逗号表达式还可以用于函数调用中的参数。例如:

```
func(n,(j=1,j+4),k);
```

该函数调用有 3 个参数,中间的参数是一个逗号表达式。括号是必须的,否则,该函数有 4 个参数了。逗号表达式作为值的形式,可以用于几乎所有的地方。

C++中,如果逗号表达式的最后一个表达式为左值,则该逗号表达式为左值。例如:

```
(a=1,b,c+1,d) = 5;     //ok:即 d=5
```

→ 在 C 中,逗号表达式是不能作左值的,所以"(a=1,b,c+1,d)=5;"将通不过编译。

3.9 求值次序与副作用

在表达式中，各操作数的求值次序并没有在 ANSI C++标准中规定。于是各个编译器为提高产生目标代码的质量，在不破坏操作符的优先级和结合性的前提下，对操作数(是个表达式)访问进行必要的顺序安排。在顺序安排中，操作数要进行挪动操作，可能会经历求值运算，而求值运算如果修改了另一个表达式中的变量则会产生副作用。

1. 不同的编译器求值顺序不同

例如：

```
int a=3,b=5,c;
c=a*b + ++b;
cout <<c <<endl;
```

c 是 a * b 和++b 的和。在求和之前，先要把这两个操作数安排在加运算的地方。就在安排的时候，可能先安排前一个表达式 a * b，也可能先安排后一个表达式++b。安排时，要对表达式进行计算求值。可是若先安排后一个表达式，求其值之后，变量 b 内存空间中的值却发生变化。在将 a * b 的值放到参加加运算的位置时，面临求 a * b 的问题。到底 b 是取自最初的 b 变量值，还是在前一个操作数放置到加运算的地方之后(b 已经发生变化)，再去取 b 变量的值呢？也就是说，a * b 为 3 * 5 还是 3 * 6 呢？见图 3-3。

编译器可以按 1,2,3 的顺序，也可以按 1',2',3 的顺序来安排求值顺序。

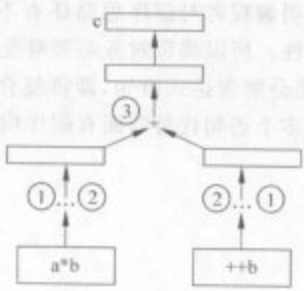

图 3-3 表达式内部的运算次序

该程序段在 BC 中运行得到 24，而在 VC 中运行却得到 21！

2. 求值顺序使交换律失去作用

加法操作我们都知道 a+b=b+a。也就是说，交换律成立。在 C++中，对简单的表达式，交换律是成立的，但对复合表达式，交换律未必成立。例如：

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

与：

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

在 VC 中，运行结果前者为 21，后者为 24。这一现象同样可以用图 3-3 来解释。

3. 求值顺序使括号失去作用

在表达式中，括号的优先级是最高的。例如，a*(b+c)中，先做加法，后做乘法。在 C++中，简单的表达式括号优先可以做到，但复合表达式未必如此。

例如：

```
int a=3,b=5,c;
c= ++b * (a+b);
```

预想的操作应为先做 a+b 得 8，然后乘上 ++b 得 6*8=48，但实际在 VC 中却为 54。这并不是说乘法抢先执行(++b*a，然后再加 b)，而是括号外面的表达式先行求值。

4. 消除副作用

在我们举的三个例子中，原因主要是 ++b 引起的。++b 具有变量 b 的修改(副作用)和它所提供的表达式值两个操作。

同样赋值表达式也会引起副作用。例如：

```
int a,b=20;
a=(b=25)+b;
```

a 是等于 25+20，还是 25+25 呢？分析之后发现，赋值表达式同样有提供表达式值的同时修改变量的行为。

表达式和语句的副作用，说明编程者对程序思路还有不够完善不够周密的地方。它导致可读性下降，也破坏了可移植性。所以编程时务必要避免副作用的产生。

解决表达式副作用的方法是分解表达式语句，即将复合表达式语句写成几个简单的表达式语句。例如，下面的代码用多个语句代替前面有副作用的表达式语句：

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

或者：

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

小结

C++为每个运算符规定了一个优先级和结合性，以控制各运算的顺序，确保表达式计算的一致性。利用括号可以改变表达式的运算顺序。

左值是能出现在赋值表达式左边的表达式。

如果运算结果超过了该数据类型能够表达的范围，则 C++进行截断处理。

参加运算的两个操作数类型不同时，C++将自动作隐式类型转换，但有时候，不得不作强制类型转换。

前增量操作符通知 C++编译器先增加变量的值，然后再使用变量；后增量操作符通知

编译器先使用变量，然后再增加该变量值。

关系运算中，=与==经常要搞错。逻辑运算符 && 和||都是短路运算符。

表达式和语句的一个重要差别是：表达式具有值，而语句是没有值的。

副作用是一个表达式中的嵌套表达式，在提供值的同时，又对某处变量进行修改所引起的。对于副作用，由于其运算结果的不可预料性，所以要尽量避免。

然而，副作用并不是什么都不好，在函数中，正是利用了副作用才使许多代码更精简和可读。事实上函数是产生副作用的温床。指针是最大的"罪魁祸首"。当学习了函数的内部实现机制和指针之后，读者会有所体会。

练习

3.1 写出以下公式的 C++表达式：

(1) $\sqrt{(\sin(x))^{2.5}}$ sqrt(pow(sin(x),2.5))

(2) $\frac{1}{2}\left(ax+\frac{a+x}{4a}\right)$ (a*x+(a+x)/(4*a))/2

(3) $\frac{e^{x^2}}{\sqrt{2\pi}}$ pow(e,x*x)/sqrt(2*M_PI)

在 math.h 头文件中：

正弦函数原型为：double sin(double x)表示 x 弧度的正弦值；

指数函数原型为：double exp(double x)表示 e 的 x 次方；

平方根函数原型为：double sqrt(double x)表示 x 的平方根；

幂指数函数原型为：double pow(double x, double y)表示 x 的 y 次方。

3.2 写出下列表达式的值：

(1)
```
int e = 1, f = 4, g = 2;
float m = 10.5, n = 4.0, k;
k = (e + f)/g + sqrt((double)n) * 1.2/g + m;
```

(2)
```
float x = 2.5, y = 4.7;
int a = 7;
x + a % 3 * (int)(x + y) % 2/4;
```
2.5+7%3*(2/4) = 2.5+0 = 2.5

(3)
```
int a, b;
a = 2, b = 5, a ++ , b ++ , a + b;
```
1.7%2

3.3 写出下列程序的运行结果。

(1)
```
#include <iostream.h>

int main()
{
    int a1, a2;
    int i = 5, j = 7, k = 0;
    a1 = !k;
    a2 = i!= j;
    cout <<"a1 = " <<a1 <<'\t'
         <<"a2 = " <<a2 <<endl;
}
```
a1=1 a2=1

(2) #include <iostream.h>

```
int main()
{
  int x,y,z;
  x=1;
  y=1;
  z=1;
  x=x||y&&z;
  cout <<x <<"," <<(x&&!y||z) <<endl;
}
```

(3) #include <iostream.h>

```
int main()
{
  int a,b,c;
  int s,w,t;
  s=w=t=0;
  a=-1;
  b=3;
  c=3;
  if(c>0)
  s=a+b;
  if(a<=0)
  {
  if(b>0)
  if(c<=0)
    w=a-b;
  }
  else
   if(c>0)
      w=a-b;
   else
      t=c;
   cout <<s <<','
        <<w <<','
        <<t <<endl;
        }
```

(4) #include <iostream.h>

```
int main()
{
  int a,b,c,d,x;
  a=c=0;
  b=1;
  d=20;
  if(a)
  d=d-10;
  else
       if(!b)
          if(!c)
            x=15;
          else
```

```
    x = 25;
  cout <<d <<endl;
}
```

3.4 根据以下函数关系，对输入的每个 x 值，求 y 值。请编制此程序。

```
y = x * (x + 2)      2<x< = 10
y = 2 * x            -1<x< = 2
y = x - 1            x< = -1
```

3.5 编程实现输入一个整数，判断其能否被 3,5,7 整除，并输出以下信息之一：

(1) 能同时被 3,5,7 整除；

(2) 能被其中两数(要指出哪两个)整除；

(3) 能被其中一个数(要指出哪一个)整除；

(4) 不能被 3,5,7 任一个整除。

3.6 编程实现输入一个整数，输出相应的五分制成绩。设 90 分以上为“A”，80 分至 89 分为“B”，70 分至 79 分为“C”，60 分至 69 分为“D”，60 分以下为“E”。

第4章　过程化语句

高级语言源程序的基本组成单位是语句。语句按功能可以分为两类：一类用于描述计算机执行的操作运算(如表达式语句)，即操作运算语句；另一类是控制上述操作运算的执行顺序(如循环控制语句)，即流程控制语句。后一类语句也称为过程化语句。学习本章后，要求掌握C++各种过程化控制语句结构，并理解常用的过程化程序实例，掌握其开发方法。

4.1　while语句

while循环由4个部分组成：循环变量初始化，继续条件，循环体，改变循环变量的值，见图4-1。

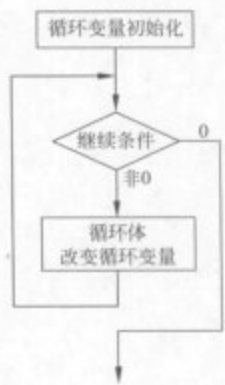

图4-1　while循环结构

while语句的作用是判断一个条件表达式，以便决定是否应当进入和执行循环体，当满足该条件时进入循环，不满足该条件时则不再执行循环。其表现形式为：

```
while(条件表达式)
  循环体
```

语句中的条件表达式就是图中的继续条件。

例如，下面的代码表示了一个while循环：

```
i = 1;                    //循环变量初始化
while(i <= 10)            //继续条件
{                         //循环体
   sum = sum + i;
   i++;                   //改变循环变量的值
}
```

该例是计算sum＝1＋2＋3＋…＋10的代码片段。初始化是对循环变量i而言。在开始循环前给控制变量赋初值(i＝1)是重要的，继续条件(i<＝10)决定循环继续多久。通常在继续条件的表达式中，总是包括循环变量。循环体包括在执行循环时将要做的操作。

图4-1中的继续条件是一个表达式。当表达式为非0时，执行循环体中的语句，否则越过循环。例如，求1＋2＋3＋…＋99＋100的值：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch4_1.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
```

```
{
    int i = 1, sum = 0;        //初始化

    while(i<=100)
    {
      sum = sum + i;
      i = i + 1;
    }

    cout <<"sum = " <<sum <<endl;
}
```

运行结果为：

```
sum = 5050
```

如果循环体包含一个以上的语句，应该用花括号括起来，以块语句形式出现。如果不加花括号，则 while 的范围只到 while 后面第一条语句。例如，上例中的循环体可以写成“sum+=i++;”一条语句，所以，循环体可以省略花括号：

```
while(i<=100)
   sum += i++;

cout <<"sum = " <<sum <<endl;
```

循环体中应该有使循环趋向结束的语句。上例中，i 的初值为 1，循环结束的条件为不满足 i<=100，随着每次循环都改变 i 的值，使得 i 的值越来越大，直到 i>100 为止。如果没有循环体中的“i=i+1;”，则 i 的值始终不改变，循环就永不终止。

→ 不必要的优化

while 循环中的继续条件，是一个表达式，并没有更多的限定，所以，上例可以在继续条件处放上一个逗号表达式以完成同样的功能：

```
i = 1;
while(sum += i++, i<=100);

cout <<"sum = " <<sum <<endl;
```

该代码的 while 语句中的循环体为一个分号，代表空语句。

根据逗号表达式的概念，C++将顺序执行逗号表达式中每个子表达式，并以最后一个子表达式的值作为整个逗号表达式的值。因此，继续条件中的逗号表达式之值是 i<=100，它在前一个子表达式 sum+=i++执行后求取，i 在此处无副作用。

C++有足够的能耐让代码最大限度的优化。该代码也显示了 C++的灵活与技巧，但可读性较差，所以它不是现代程序设计所追求的。我们介绍的用意是让初学者见识这类代码，以达到更好地领会概念的目的。

4.2 do...while 语句

do...while 语句的表现形式为：

```
do
   循环体
while(条件表达式)
```

当流程到达 do 后，立即执行循环体语句，然后再对条件表达式进行测试。若条件表达式的值为真(非 0)，则重复循环，否则退出。

该语句结构使循环至少执行一次。

例如，要从键盘中得到一个范围为 1～10 的数：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch4_2.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   int val;

   do
   {                          //循环体
      cout <<"please enter a number between 1 and 10\n";
      cin >>val;              //修改条件
      if(val<1 || val>10)
         cout <<"the number be not between 1 and 10\n";
   }while( val<1 || val>10 );     //继续条件

   cout <<"you entered a " <<val <<endl;
}
```

运行结果为：

```
please enter a number between 1 and 10
12
the number be not between 1 and 10
please enter a number between 1 and 10
6
you entered a 6
```

该程序读入一个 1～10 之间的数，满足条件后就越过循环，执行显示读入的数值。do-while 循环结构见图 4-2。

图 4-2　do-while 循环结构

do-while 循环至少执行一次，因为直到程序到达循环体的尾部遇到 while 时，才知道继续条件是什么。如果继续条件仍然成立，程序回转到 do-while 循环的顶部，继续循环。

在循环体中，if 语句的条件和 while 的继续条件是同一个，那只是一个巧合，并非必须。

代码中，继续条件的不断变化很重要。如果 val 值恒定不变，则继续条件也永不改变，导致死循环。

do-while 循环同样需要循环变量初始化。do-while 循环在循环体的底部进行继续条件的测试，所以它至少将执行一次循环

体。而 while 语句在循环的顶部测试，有可能永远不执行循环体。

do-while 在许多场合可以做 while 能做的事。例如，上节中求：$\text{sum}=\sum_{n=1}^{100} n$

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch4_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   int i,sum = 0;

   i = 1;
   do
   {
      sum = sum + i;
      i = i + 1;
   }while(i<=100);


   cout <<"sum = " <<sum <<endl;
}
```

运行结果为：

```
sum = 5050
```

→ 书写格式与可读性

do-while 循环中，while(继续条件)后面的分号不要遗忘。不要把 do-while 循环与 while 循环使用空语句作为循环体的形式相混淆。

```
do sum += i++;
 while(i<=100);          //do-while求和代码段

 while(i++ <10000);  //while时间延迟代码段
```

因为它们从局部看都是：

```
while(表达式);
```

为明显区分它们，do-while 循环体即使是一个单语句，习惯上也使用花括号包围起来，并且 while(表达式)直接写在花括号"}"的后面。这样的书写格式可以与 while 循环清楚地区分开来。例如：

```
do
{
  sum += i++;
}while(i<=100);
```

4.3 for 语句

可以用图来描述 for 循环结构，见图 4-3。

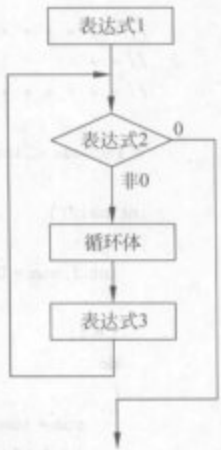

图 4-3 for 循环结构

for 语句的一般表现形式为：

```
for(表达式 1; 表达式 2; 表达式 3)
   循环体
```

它的执行过程如下：

(1) 求解表达式 1；

(2) 求解表达式 2，若为 0，则结束循环，转到(5)；

(3) 若表达式 2 为真，执行循环体，然后求解表达式 3；

(4) 转回(2)；

(5) 执行 for 语句下面的一个语句。

C/C++中的 for 语句相对 while 和 do-while 来说，较为灵活。它不仅可以用于循环次数已经确定的情况，而且可以用于循环次数不确定而只给出循环结束条件的情况。

例如，for 循环对于求和来说，方式更简单、可读：

```
for(i = 1; i <= 100; i ++ )  //初始化，继续条件，步长都在顶部描述
{
   sum += i;                 //循环体相对简洁
}
```

如果将 for 语句的一般形式用 while 来表达，则为如下：

```
表达式 1;
while(表达式 2)
{
   循环体
   表达式 3;
}
```

所以 for 语句是将循环体所用的控制放在循环顶部统一表示，显得更直观。除此之外，for 语句还充分表现了其灵活性：

(1) 表达式 1 可以省略。此时应在 for 语句之前给循环变量赋初值。若省略表达式 1，其后的分号不能省略。

例如，求和运算：

```
i = 1;
for( ; i <= 100; i ++ )      //分号不能省略
   sum += i;
```

执行时，跳过求解表达式 1 这一步，其他不变。由于循环体由一条语句构成，所以花括号可以省略。

(2) 表达式 2 可以省略。即不判断继续条件，循环无终止进行下去。也就是认为表达式 2 始终为真。这时候，需要在循环体中有跳出循环的控制语句。

例如，求和运算：

```
for(i=1; ; i++)          //分号不能省略
{
   sum+=i;
   if(i>100)
      break;
}
```

等价于：

```
for(i=1; 1; i++)         //表达式2为真(1)
{
   sum+=i;
   if(i>100)
      break;
}
```

此处 break 表示退出循环，见后面 4.5 节。

(3) 表达式 3 可以省略。但此时程序员应另外设法让循环变量递进变化，以保证循环能正常结束。

例如，求和运算：

```
for(i=1; i<=100; )       //分号不能省略
   sum+=i++;             //同时改变循环变量
```

在循环体中，必须自己对循环变量进行修改(i++)，其效果与在表达式 3 上设置是一样的。

(4) 表达式 1 和表达式 3 可同时省略。

例如，下面的代码同样能完成求和运算：

```
for( ; i<=100; )
   sum+=i++;
```

(5) 三个表达式都可省略。即不设初值，不判断条件(认为表达式 2 为真)，循环变量不增值，无终止执行循环体。

例如，求和运算可以：

```
for( ; ; )
{
   sum+=i++;
   if(i>100)
     break;
}
```

(6) 表达式 1，表达式 2，表达式 3 都可以为任何表达式。

例如，求和运算中可设置 sum 的初值：

```
for(sum=0; i<=100; i++)
   sum+=i;
```

例如，表达式 1 为逗号表达式：

```
for(sum = 0,i = 1; i<=100; i++)
  sum += i;
```

例如，表达式 1 和表达式 3 都为逗号表达式：

```
for(i = 0,j = 100,k = 0; i<=j; i++,j--)
  k += i * j;
```

例如，表达式 2 和表达式 3 可以为赋值或算术表达式的情况，下面两个语句都可以完成同样的求和运算：

```
for(i = 1; i<=100; sum += i++);      //循环体为空语句

for(i = 1; sum += i++,i<=100;);      //表达式 3 省略，循环体为空语句
```

注意，在了解以上各种编程方法的同时，不要忘了可读性。

(7) 表达式 1 可以是循环变量定义。C++的变量定义可以在任何语句的位置，for 循环中也不例外。例如，下面的代码完成求和运算：

```
for(int i=1; i<=100; i++)
  sum += i;
```

for 循环使得所有的循环细节都可在语句中描述。使得程序又精炼又可读。只要循环变量不在程序的其他地方使用，在 for 头部定义循环变量是最好的，该变量只在循环体中有效，循环退出后自行消失。关于作用域规则见 6.3 节。

4.4 switch 语句

switch 语句是多分支选择语句。if 语句是二分支选择语句，但在实际问题中常常需要用到多分支的选择。例如，学生成绩分类(90 分以上为 A，80 分至 89 分为 B，70 分至 79 分为 C 等)，人口统计分类(按年龄分为老、中、青、少、儿、幼)等。嵌套的 if 语句也可以处理多分支选择，但是，switch 更加直观。

switch 语句的一般形式为：

```
switch(表达式)
{
   case 常量表达式 1: 语句组 1
   case 常量表达式 2: 语句组 2
   ...
   case 常量表达式 n: 语句组 n
   default: 语句组 n + 1
}
```

例如，根据考试成绩的等级输出百分制分数段：

```
char grade;
//...

switch(grade)
{
   case'A': cout <<"85～100\n";
```

```
    case'B': cout <<"70~84\n";
    case'C': cout <<"60~69\n";
    case'D': cout <<"<60\n";
    default: cout <<"error\n";
}
```

(1) switch 后面括号中的表达式只能是整型、字符型或枚举型表达式。case 后面的常量表达式之类型必须与其匹配。例如,下面的代码错误地用浮点类型作 switch 的表达式,它会引起编译错误:

```
float f = 4.0;

switch(f) //error
{
  //...
}
```

(2) 当表达式的值与某一个 case 后面的常量表达式值相等时,就执行此 case 后面的语句,若所有 case 中的常量表达式值都没有与表达式值匹配,就执行 default 后面的语句。

(3) **case 语句起标号的作用**。标号不能重名,所以每一个 case 常量表达式的值必须互不相同,否则就会出现编译错误。例如,下面的代码中 case 出现相同常量值:

```
case 'A': cout <<"this is A\n";
case 65 : cout <<"this is 65\n";     //error:'A'等值于 65
```

(4) 因为 case 语句起语句标号的作用,所以 case 与 default 并不改变控制流程。

例如,在最初的例子中,若 grade 的值等于'A',则将连续输出:

```
85~100
70~84
60~69
<60
error
```

case 通常与 break 语句联用,以保证多路分支的正确实现。例如,改写上例以使输出某个成绩段后终止 switch 语句:

```
char grade = 'B';

switch(grade)
{
  case'A': cout <<"85~100\n"; break;
  case'B': cout <<"70~84\n"; break;
  case'C': cout <<"60~69\n"; break;
  case'D': cout <<"<60\n"; break;
  default: cout <<"error\n";
}
```

输出结果为:

```
70~84
```

最后一个分支可省略 break 语句。

(5) 各个 case(包括 default)的出现次序可以任意。在每个 case 分支都带有 break 的情况下,case 次序不影响执行结果。

例如,上面的代码可以写成下面的代码:

```
char grade;
//...

switch(grade)
{
   case'C': cout <<"60~69\n"; break;
   default: cout <<"error\n"; break;
   case'D': cout <<"<60\n"; break;
   case'A': cout <<"85~100\n"; break;
   case'B': cout <<"70~84\n"; break;
}
```

当 grade 为'B'值时,输出结果也为:

```
70~84
```

(6) 多个 case 可以共用一组执行语句。例如:

```
//...
case'A':
case'B':
case'C': cout <<">60\n";
```

当 grade 的值为'A','B'和'C'时,都输出“>60”。

几个状态都执行同一操作时,不能将值用逗号隔开,图谋在一个 case 中实现。例如:

```
case 1,2,3: cout <<"hello";  //error
```

是错的,应为:

```
case 1:
case 2:
case 3: cout <<"hello";
```

(7) default 语句是可选的。当 default 不出现时,则当表达式的值与所有常量表达式的值都不相等时,越过 switch 语句。

(8) switch 不一定非要包含复合语句块。例如,下面两条语句是等价的:

```
switch(i) case 1: cout <<"ok\n";
if(i == 1) cout <<"ok\n";  //与上一条语句等价
```

(9) switch 语句可以嵌套。case 与 default 标号是与包含它的最小的 switch 相联系的。例如:

```
int i,j;
//...

switch(i)
{
   case 1:  //...
```

```
    case 2:
      switch(j)     //嵌套 switch
      {
        case 1:  //...
        case 2:  //...
        //...
      }
      case 3:  //...
    //...
}
```

switch(j)中的 case 1 标号不会与外面的 switch(i)中的 case 1 标号相混淆。

(10) 用 if 语句与 switch 语句可以互相补充。

例如,根据学生的分数输出其成绩等级:

```
int grade;
//...

if(grade>=85 && grade<=100)
   cout<<"A\n";
else if(grade>=70 && grade<85)
   cout<<"B\n";
else if(grade>=60 && grade<70)
   cout<<"C\n";
else if(grade<60 && grade>=0)
   cout<<'D';
else
  cout<<"error\n";
```

最后的 else 等价于 switch 中的 default。

由于 grade 是表示分数,若用 switch 中的 case 标号来表达,则标号需要很多,程序会很长。因为 switch 语句只能对等式进行测试,如果测试值包含一个较大的范围,就需要关系表达式比较,这时候用 if 语句较好。

if...else 语句的执行体等价于 switch 语句的 case 中含有 break 的语句组。

写成上面这样的格式,可以提高 if 语句的可读性。

另外,switch 语句只能对整型数进行测试,如果对浮点数进行测试,也需要 if 语句。

若测试一个整型变量取几个不同的值,用 switch 语句比较简明。

4.5 转向语句

1. break 语句

break 语句用在 while,do...while,for 和 switch 语句中。

在 switch 语句中,break 用来使流程跳出 switch 语句,继续执行 switch 后的语句。

在循环语句中,break 用来从最近的封闭循环体内跳出。

例如,下面的代码在执行了 break 之后,继续执行"a=1;"处的语句,而不是跳出所有的循环:

```
for( ; ; )
{
   for ( ; ; )
   {
      //...
      if(i == 1)
       break;
      //...
   }
   a = 1;     //break 跳至此处
   //...
}
```

2. continue 语句

continue 语句用在循环语句中，作用为结束本次循环，即跳过循环体中尚未执行的语句，接着进行下一次是否执行循环的判定。

例如，下面的代码把 100～200 之间的不能被 3 整除的数输出：

```
for(int n = 100; n< = 200; n++ )
{
   if(n % 3 == 0)
        continue;

   cout <<n <<endl;
   //...
}
```

当 n 被 3 整除时，执行 continue 语句，结束本次循环，即跳过 cout 语句。只有 n 不能被 3 整除时，才执行 cout 函数。

由于多条语句可以组成块，所以上述代码也可以写成：

```
for(int n = 100; n< = 200; n++ )
{
   if(n % 3!= 0)        //条件相反
   {
        cout <<n <<endl;
        //...
   }
}
```

从而省略了 continue 语句。事实上，由于在 C++中，有块语句的支持，所以，经常使用适当 if 语句，把 continue 后面的语句以块的形式包含在 if 语句之中，可以避免使用 continue 语句。

continue 语句和 break 语句的区别是：continue 语句只结束本次循环，而不是终止整个循环的执行；而 break 语句则是终止整个循环，不再进行条件判断。对于 for 语句，其两者的差别见图 4-4。

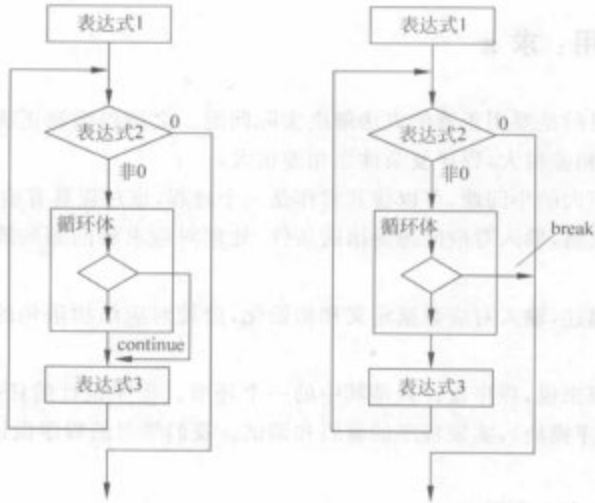

图 4-4　continue 与 break 语句的区别

3. goto 语句

goto 语句将控制从它所在的地方转移到标识符所标识的语句处。例如，求 1 加到 100 的和：

```
i = 1; sum = 0;

Loop:
  sum += i ++ ;
  if(i< = 100)
    goto Loop;
  cout <<"sum is" <<sum <<endl;
```

语句标号用标识符表示，它的命名规则与变量名相同。

用 goto 语句实现的循环完全可以用 while 或 for 循环来表示。现代程序设计方法主张限制用 goto 语句，因为，滥用 goto 语句将使程序流程无规则，可读性差。goto 语句只在一个地方有使用价值：当要从多重循环深处直接跳转到循环之外时，如果用 break 语句，将要用多次，而且可读性并不好，这时 goto 可以发挥作用。

例如，下面的代码找到满足乘积为 27 的两个小于 10 的整数后即输出该数，运行结束：

```
for(int i = 1; i<10; i ++ )
  for(int j = 1; j<10; j ++ )
      if(i * j == 50)
        goto End;

End:          //循环体外
  cout <<i <<" * " <<j <<"50\n";
```

4.6 过程应用：求 π

程序设计的目的是要用正确的方法解决实际问题。之所以强调正确，是因为不同的程序设计方法，难易相差很大，程序复杂性也相差很大。

一个复杂性不大的小问题，可以将其看作是一个过程，该过程具有输入、处理和输出。

对于问题的求解，输入对应问题给出的条件，处理对应求解问题的算法，输出对应问题的解。

对于程序的描述，输入对应数据定义和初始化，处理对应结构语句的一个序列，输出对应打印输出语句。

对于软件工程来说，程序设计只是其中的一个环节。程序设计的任务是根据给定的数据定义和算法(程序模块)，实现程序的编码和调试。我们学习的程序设计，所指的概念包括了程序设计方法。

程序设计方法有 3 个层次：

(1) 简单的问题求解分析方法(过程化方法)。它适用于简单、孤立的问题求解。一般定义 2～3 个函数便可解决。

(2) 结构化程序设计方法。它适用于一个问题大小适中，能够方便地分解成相对独立的几个功能模块，从而用几个程序文件分别描述并调试实现之。

(3) 面向对象程序设计方法。它面向求解一个用常规方法并不能简单理清头绪的问题。它将问题看作为包含有若干个小对象的大对象，层层分解对象，研究里面的数据和行为。当一个问题分解成不同层次的对象结构时，程序设计的描述也同时完成。

这里介绍的是第 1 种方法。

例如，用公式“$\frac{\pi}{4}\approx 1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$”求 π 的近似值，直到最后一项的绝对值小于 10^{-8} 为止。

分析：

(1) π 的表示用 double 型。因为 float 型的有效位数是 7 位，而该问题中的最小项的精度要求达到小数点后 8 位。

(2) 先求 π/4，然后求 π。

(3) 分析数列的通项：数列的第 1 项是 1，第 2 项是 −1/3，…，第 n 项是 $(-1)^{n-1}/(2\times n-1)$。第 n 项与第 n−1 项的关系为符号变一下，分母加 2。

根据前后项的关系，可以设计一个循环，每次循环将原项分母加 2，符号变更一下，求得新项。最初的分母变量(类型为 long)值是 1。最初的符号变量值是 +1。于是可以用图 4-5 所示的框图来表示算法。

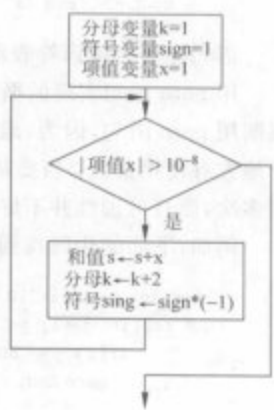

图 4-5 求 π/4 算法一

根据该算法，添上头文件，定义相应的变量，实现 while 循环，最后根据 π/4，输出 π 值。

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch4_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>
#include <math.h>

int main()
{
   double s=0,x=1;                          //初始值
   long k=1;
   int sign=1;

   while(fabs(x)>1e-8)                      //项值在比较前要先求绝对值
   {
       s+=x;
       k+=2;
       sign*=-1;
       x=sign/double(k);                    //强制转换使x得到浮点数值
   }

   s*=4;                                    //π值
   cout<<"the pi is "<<s<<endl;             //输出
       <<setiosflags(ios::fixed)
       <<setpreasion(8)
       <<s<<endl;
}
```

运行结果为：

```
the pi is 3.14159263
```

该程序中的变量名意义在前面的框图中描述。如果只有源程序，那么，变量名的命名要能反映出其意义，以使程序能被人读懂。

该程序在 Pentium133 主频的 PC 机中运行，耗时 40 秒。

求解的算法一般都不是唯一的。如果注意到前后项关系的另一种表达：设第 n−1 项为 x，则下一项为 x*(−1)*(2*n−3)/(2*n−1)，则 for 循环的结构显得更自然一些。见图 4-6 描述。

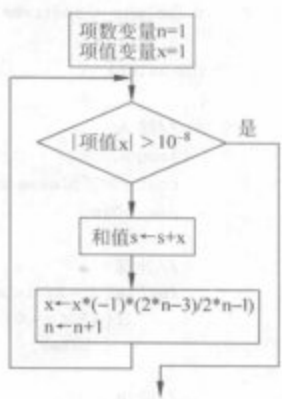

图 4-6　求 π/4 算法二

根据此算法，可以得到下面的程序：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch4_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <math.h>

int main()
{
   double s=0,x=1;                          //初始值
```

```
    for(int n=1; fabs(x)>1e-8; n++,x*=(-1.0)*(2*n-3)/(2*n-1))
        s+=x;

    s*=4;                                    //π值
    cout<<"the pi is "<<s<<endl;             //输出
}
```

运行结果为：

```
the pi is 3.14159
```

如果从效率上去分析，该程序不如前者，因为求项数时，要做4次乘法，一次除法，而ch4_4.cpp中只做一次除法。

在步长表达式中，-1.0表示一个浮点数，以使项值x能取到浮点数值。

for的循环体只有一条语句，所以就省去了花括号。

该程序与上面的程序比较，可读性稍好，但程序运行比前者多用了2秒钟。在算法分析时，多考虑效率，但编程中，我们要更多考虑可读性。从这个意义上说，本程序比上个程序优。然而，本程序还不是最优的，它完全可以做到既不损失效率，也不损害可读性。

如果条件指明到n=1000000项停止，则程序如何修改，哪个程序更方便修改？请读者思考。

4.7 过程应用：判明素数

给定一个整数m，判断其是否为素数。

分析：m是素数的条件是不能被$2,3,\cdots,m-1$整除。

根据这一条件，立即可以写出一个循环判明该数是否为素数：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch4_6.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
    //输入
    long m;
    cout<<"please input a number:\n";
    cin>>m;

    //处理
    for(int i=2; i<m; i++)          //找m的因数
        if(m%i==0)
            break;

    //输出
    if(m==i)                        //判断m是否被小于m的数整除
        cout<<"m is prime.\n";
```

```
    else
      cout <<"m isn't prime.\n";
}
```

在程序的输出部分，判断是否 m==i，for 循环有两种退出的情况：一种是不满足 i<m 的循环条件而正常退出，此时 i 正好等于 m；另一种是发现 m 整除 i 时的退出，此时 i 小于 m。所以，判断 m 与 i 是否相等，就能知道 m 是否为素数。

该程序最直接反映了数学定义。但是，当给定的 m 很大时，运算量也很大，能否改进一下算法，使运算量急剧下降？

假定 m 不是素数，则可表示为：m=i*j。i<=j，$i<=\sqrt{m}$，$j>=\sqrt{m}$。

也就是说，如果 m 不是素数，一定能找到一个整数 i 能整除 m，即 m%i 为 0。

于是，循环可以在 $2\sim\sqrt{m}$ 内进行。

于是改进的算法可以写为：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch4_7.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <math.h>

int main()
{
   //输入
   long m;
   cout <<"please input a number:\n";
   cin >>m;

   //处理
   double sqrtm = sqrt(m);              //用到 math.h
   for(int i = 2; i<= sqrtm; i++)
      if(m % i == 0)
        break;

   //输出
   if(sqrtm<i)                          //注意 == 与 =
      cout <<"m is prime.\n";
   else
      cout <<"m isn't prime.\n";
}
```

程序中，求 m 的平方根特地放在 for 循环外面来做，本可以直接写为：

```
for(int i = 2; i<sqrt(m); i++)
```

但如果这样，则每次比较都要求一次平方根。为了明显提高循环的效率，在可读性不受影响的前提下，可以适当对程序进行优化。

如果要求从 a～b 的数段内所有的素数，则应该对每个 a～b 中的数（循环）都进行上述程序的判断（循环），所以它为二重循环。

事实上，从 a～b 的循环步长可以是 2，因为偶数不是素数。但循环前，先要判明 a 是否偶数。程序如下：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch4_8.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>
#include <math.h>

int main()
{
  //输入
  long a,b,l = 0;
  cout <<"please input two numbers:\n";
  cin >>a >>b;
  cout <<"primes from " <<a <<" to " <<b <<" is:\n";

  //处理
  if(a = = 2)
    cont<<"2";
  if(a % 2 == 0)          //若为偶数,则增 1,确保下面循环变量从奇数开始
    a ++ ;

  for(long m = a; m< = b; m += 2)          //步长为 2
  {
    int sqrtm = sqrt(m);
    int i;
    for(i = 2; i< = sqrtm; i ++ )          //判明素数
      if(m % i == 0)
        break;

    //输出
    if(i>sqrtm)              //素数
    {
      if(l ++ % 10 == 0)
        cout <<endl;
      cout <<setw(5) <<m;
    }
  }
}
```

运行结果为：

```
c:>ch4_8
please input two numbers:
12 78
primes from 12 to 78 is:
   13   17   19   23   29   31   37   41   43   47
   53   59   61   67   71   73
```

该程序中设置了一个打印位置变量，每次打印一个素数，该变量加 1，当该变量满足模 10 为 0 时，打印一个回车。该代码起到控制每行输出数据个数的作用。

程序要运行可靠，输入的校验很重要，如果本程序输入 87 和 12，则该程序会有什么反应，能否改进程序使之对输入进行校验？

4.8 过程应用：求积分

数学上求积分靠公式推导，求一条函数曲线 f(x)在 x 轴上从 a～b 的投影所包围之面积可以看成是一重积分。我们用一种变步长的辛普生递推公式来进行计算机求解积分问题。积分问题为将该积分看作为求如图 4-7 所示的封闭区域面积。

求解积分的步骤如下：

(1) 用梯形公式计算面积近似值

$$I_n = T_n = \frac{h}{2}(f(a) + f(b))$$

其中 $n=1, h=b-a$。如图 4-8 所示。

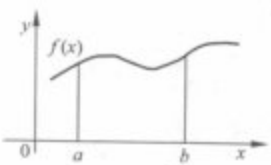

图 4-7 $[a,b]$区域的 $f(x)$函数所包围的面积

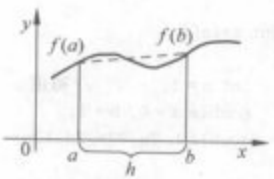

图 4-8 用梯形公式计算面积

(2) 用变步长梯形法计算面积近似值

$$T_{2n} = \frac{T_n}{2} + \frac{h}{2}\sum_{k=0}^{n-1} f\left(x_k + \frac{h}{2}\right)$$

$2n$ 意味着将区间$[a,b]$划分成 $2n$ 等分。显然开始时，$n=1$，即 $2n=2$ 等分。该公式是说，一半的面积由前面的近似公式给出，另一半则由原 n 等分的中值小矩形和的一半给出，如图 4-9。

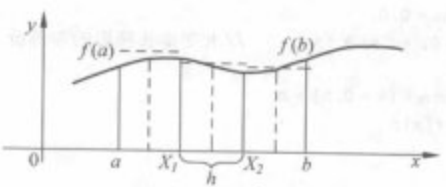

图 4-9 变步长梯形法计算区域划分

图中，$x_0=a, x_n=b, h=(b-a)/n$。中值小矩形即某等分中线的函数值与 h 的乘积。

(3) 用辛普生公式计算积分近似值

$$I_{2n} = \frac{4T_{2n} - T_n}{3}$$

(4) 只要上次求的积分值与本次求的积分值之差在一个非常小的 ε 范围内，则认为所求积分的近似度已经达到要求。即若

$$|I_{2n} - I_n| < \varepsilon$$

则结束，I_{2n}即为所求。否则，$n \leftarrow 2n, h \leftarrow h/2$，重复第(2)步和第(3)步。

下面的程序是求积分

$$I=\int_0^1 \frac{e^x}{1+x^2}dx$$

其中 ε 取 10^{-8}。

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch4_9.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <iomanip.h>
#include <math.h>

double f(double x);

int main()
{
   int n=1;          //初值
   double a=0, b=1;
   double h,Tn,T2n,In,I2n;

   const double eps=1e-8;

   h=b-a;
   T2n=I2n=h*(f(a)+f(b))/2;
   In=0;

   while(fabs(I2n-In)>=eps)          //求积分
   {
       Tn=T2n;
       In=I2n;

       double sigma=0.0;
       for(int k=0; k<n; k++)        //求变步长梯形的和部分
       {
         double x=a+(k+0.5)*h;
         sigma+=f(x);
       }

       T2n=(Tn+h*sigma)/2.0;         //变步长梯形
       I2n=(4*T2n-Tn)/3.0;           //辛普生公式

       n*=2;                         //划分
       h/=2;
   }

   cout<<"the integral of f(x) from "
       <<a<<" to "<<b<<" is \n"
       <<setiosflags(ios::fixed)
       <<setprecision(8)
       <<setw(10)<<I2n<<endl;        //输出结果
}
```

```
double f(double x)
{
   return exp(x)/(1 + x * x);
}
```

运行结果为：

```
the integral of f(x) from 0 to 1 is
1.27072414
```

该程序将所求积分的函数 f(x)从程序的主函数中分离出来，以便当所求积分的函数发生变更时，只要 f(x)函数的定义更改即可。

在求积分的循环中，初始值 T2n 和 I2n 首先被赋给 Tn 和 In，以实现新的一轮逼近积分值的循环。考虑到这一点，所以在 while 循环之前，先将初始的梯形公式值赋给 T2n 和 I2n。

该循环可以用 for 循环来实现：

```
for(T2n = I2n = h * (f(a) + f(b))/2; fabs(I2n - In) >= eps; n * = 2, h/ = 2)
{
  //...
}
```

小结

循环是一组语句，计算机反复执行这组语句直到满足终止条件为止。

可以通过循环变量来控制循环。如果事先不知道循环次数，可以在循环体中通过条件判断中间跳转的方法终止循环。

while，do...while 和 for 语句都是循环语句，它们可以相互转化。

switch 是多分支语句，它是 if 语句的一个补充，但并不是必需的，当用它编制程序会带来可读性良好的效果时，就采用它。

用这些语句编制的程序，其结构性比较好，所以可读性也比较好。现代程序设计反对用 goto 编程，因为它破坏程序过程中的结构，使之不可读，难维护。

求 π 是常规的级数求和问题，任何级数的求和求积，都可进行类似的简单分析和实现。

判明素数以及求一定数域的素数分析，能够帮助读者认识到，算法能够相当程度地影响程序运行的效率。

求积分问题是要说明，解决问题的算法很重要，而这种算法并不是程序设计中学习的内容。

程序设计有三种方法。过程化方法最直接和简明，但它只能解决一些小问题。

对一个具体算法问题，编制程序相对来说是比较难的，因为涉及计算方法问题。利用现有的数学方法和结论，有助于问题的解决。

学习程序设计的主要任务是学习如何组织程序，表达实际问题的已有解决方法，而不是去寻找实际问题的解决方法。

寻找实际问题的解决方法属于系统分析与设计的范畴。本书介绍的只是常规和简单的

问题求解方法和思路。使之通过程序设计的经验积累，升华到对优秀程序设计（面向对象程序设计）方法学习和掌握的向往，进而研究问题的分析与实现。

程序设计更多的是体现其艺术性，可读性是我们追求的重要目标。

练习

4.1 计算级数

$$1+x-\frac{x^2}{2!}+\frac{x^3}{3!}-\cdots+(-1)^{n+1}\frac{x^n}{n!}$$

要求精度为 10^{-8}。并分别用 do...while、while 和 for 语句编写程序。

4.2 编程求 1!+2!+3!+4!+…+12!

4.3 编程求“水仙花数”。所谓“水仙花数”是指一个三位数，其各位数字立方和等于该数本身。例如，153 是水仙花数，因为 $153=1^3+5^3+3^3$

4.4 编程求 1000 之内的所有“完数”。所谓“完数”是指一个数恰好等于它的因子之和。例如，6 是完数，因为 6=1+2+3。

4.5 一球从 100m 高度落下，每次落地后反跳回原高度的一半，再落下。编程求它在第 10 次落地时，共经过多少 m？第 10 次反弹多高？

4.6 猴子吃桃问题。猴子第一天摘下若干桃子，当即吃了一半，还不过瘾，又多吃了一个。第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第 10 天早上想再吃时，见只剩一个桃子了。编程求第一天共摘下多少桃子。

4.7 用迭代法编程求 $x=\sqrt{a}$。

求平方根的迭代公式为：

$$x_{n+1}=\frac{1}{2}\left(x_n+\frac{a}{x_n}\right)$$

要求前后两次求出的 x 的差的绝对值小于 10^{-7}。

4.8 用循环语句编程打印如下图案：

(1)

```
#
# # #
# # # # #
# # # # # # #
# # # # # # # # #
# # # # # # # # # # #
# # # # # # # # # # # # #
# # # # # # # # # # # # # # #
# # # # # # # # # # # # # # # # #
```

(2)

```
# # # # # # # # # # # # # # # # # # #
 # # # # # # # # # # # # # # # # # #
  # # # # # # # # # # # # # # # # #
   # # # # # # # # # # # # # # # #
    # # # # # # # # # # # # # # #
     # # # # # # # # # # # # # #
      # # # # # # # # # # # # #
       # # # # # # # # # # # #
        # # # # # # # # # # #
         # # # # # # # # # #
          # # # # # # # # #
```

4.9 编程打印乘法九九表：

(1)

*	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

(2)

*	1	2	3	4	5	6	7	8	9
1	1								
2	2	4							
3	3	6	9						
4	4	8	12	16					
5	5	10	15	20	25				
6	6	12	18	24	30	36			
7	7	14	21	28	35	42	49		
8	8	16	24	32	40	48	56	64	
9	9	18	27	36	45	54	63	72	81

(3)

*	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2		4	6	8	10	12	14	16	18
3			9	12	15	18	21	24	27
4				16	20	24	28	32	36
5					25	30	35	40	45
6						36	42	48	54
7							49	56	63
8								64	72
9									81

4.10 编程求解问题。若一头小母牛，从出生起第四个年头开始每年生一头母牛，按此规律，第 n 年时有多少头母牛？

第5章 函 数

要编好程序，就要会合理地划分程序中的各个程序块，C++称之为函数。函数有各种表现形态，但都离不开函数调用的实质。所以要用好函数，必须先把握函数调用机制。学习本章后，要求领会函数调用的内部实现机制，区分函数声明与定义，掌握全局变量、静态局部变量和局部变量之间的区别，理解并运用递归、内联、重载和默认参数的函数。

5.1 函数概述

程序通常是非常复杂而冗长的。实际编程中，有些程序需要几万甚至几百万行的代码。在编写一个很长的程序时，可以采用一种好的策略，就是把这个大的程序分割成一些相对独立而且便于管理和阅读的小块程序。这样，无论对程序员还是其他阅读者都很方便。

把相关的语句组织在一起，并给它们注明相应的名称，利用这种方法把程序分块，这种形式的组合就称为函数。函数通常也称为例程或过程。

函数的使用是通过函数调用实现的。函数调用指定了被调用函数的名字和调用函数所需的信息(参数)，这和请一个上门服务的修理工形式类似。主人(相当于调用函数)要求修理工人(相当于被调用函数)按照要求(函数参数)完成某个任务，并在完成这项工作后由主人验收(函数返回)。如果不符合要求，则修理工人就面临拿不到工钱的局面。

程序员编写完成指定任务的函数是用户定义的函数，标准库函数是C++提供的可以在任何程序中使用的公共函数。程序总是从main()函数开始启动。

可以通过结合已有函数的方法建立新的函数。由多个小函数建立大函数，这能使程序易写、易读和易调试。

图5-1反映了main()函数用层次式管理方式与被调用函数的关系。一个函数可以被函数调用也可以调用函数。

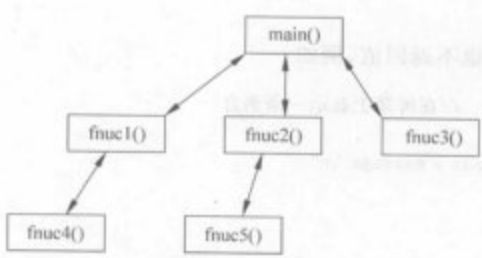

图5-1 调用与被调用函数的层次关系

C++不允许函数定义嵌套，即在函数定义中再定义一个函数是非法的。

例如，下面的代码在主函数中非法嵌套了一个func()函数定义：

```
void main()
{
   void func()
   {
//...
   }
}
```

C++函数是一个独立完成某个功能的语句块，函数与函数之间通过输入参数和返回值(输出)来联系。可以把函数看作是一个“黑盒(black box)”，除了输入输出，其他什么也看不见。

例如我们只需了解怎样连接电源和天线及按钮操作，就能看到电视节目(输入与输出)。而电视机内部的电子元件如何工作无须我们操心，这样的电子元件就称为“黑盒”。

函数的类型：

(1) 获取参数并返回值，例如：

```
int bigger(int a, int b)
{
   return (a>b) ? a : b;
}
```

(2) 获取参数但不返回值，例如：

```
void delay(long a)
{
   for(int i = 1; i<=a; i++); //延迟一个小的时间片
}
```

(3) 没有获取参数但返回值，例如：

```
int geti() //从键盘上获取一个整型数
{
   int x;
   cout <<"please input a integer:\n";
   cin >>x;
   return x;
}
```

(4) 没有获取参数也不返回值，例如：

```
void message()     //在屏幕上显示一条消息
{
   cout <<"This is a message.\n"
}
```

5.2 函数原型

标准库函数的函数原型都在头文件中提供，程序可以用#include指令包含这些原型文件。对于用户自定义函数，程序员必须在源代码中说明函数原型。

函数原型是一条程序语句，即它必须以分号结束。它由函数返回类型、函数名和参数表

构成，形式为：

```
返回类型 function(参数表);
```

参数表包含所有参数的数据类型，参数之间用逗号分开。**在C++中，函数声明就是函数原型。**

函数原型和函数定义在返回类型、函数名和参数表上必须完全一致。如果它们不一致，就会发生编译错误。

函数原型不必包含参数的名字，而只要包含参数的类型。下面的函数原型声明是合法的。

```
int Area(int,int);
```

等价于：

```
int Area(int length,int width);
```

对于标准库函数(简称库函数)来说，编译器从来不把其实际代码看成是程序的组成部分。编译器能够确认是否正确地调用库函数，这是必要的。在头文件中内含的函数声明都是函数原型。

如果函数原型不正确，编译器会及时报告错误。

例如，对于程序 ch1_3.cpp，主函数中的函数调用写成：

```
c = max(a,b,56);     //error:extra parameter in function call
```

则编译器将会报告一个“函数调用中遇到过多的函数参数”之错误。

又例如，下面的代码中，函数声明与函数定义的函数原型不一致：

```
void funcA(int,float);

int main()
{
   int a;
   float b;
   funcA(a,b);
}

void funcA(int,int)
{
   //...
}
```

该代码能够正确通过编译，因为函数声明的原型与函数调用相吻合。但在连接时，发现没有与函数声明相一致的函数定义，结果产生“不能确定的外部函数”的连接错误。

函数返回在声明时约定数据类型。例如：

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

例中，函数返回的变量 a 或 b 是 int 型的。

如果返回的是其他基本数据类型，则在返回时，先作隐含的类型转换，然后再返回。例如，下面的代码中，主函数中的变量 a 被初始化为 3：

```
int f()
{
   return 3.5;
}

int main()
{
   int a = f();
}
```

因为函数 f()定义的返回类型是 int，所以 return 语句的值 3.5 被转换成 int 型数 3 之后，返回给主函数，赋给了变量 a。

如果函数返回的是不相容的数据类型(比如，后面介绍的类对象)，则函数将在编译时给出一个“不能将类转换成 int”的错误。

函数的返回值也称函数值。返回的不是函数本身，而是一个值。

return 语句后面的括号是任选的，例如，“return (3);”等价于“return 3;”。

return 语句可用于改变执行顺序。例如：

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

在这里，return 语句还起到了改变计算顺序的作用。因为 return 是返回语句，它将退出函数体，所以该语句之后的语句不会被执行了。

无返回的函数也可以使用 return，但不能返回值。例如：

```
void message(int a)
{
   if(a>10)
     return;
   //...
}
```

在这里，return 语句起了一个改变语句顺序的作用。

在有返回类型的函数中，return 后面所跟的表达式并不直接替换调用函数，而是先经过求值计算，必要的时候进行类型转换，然后将其存放到内存的某个区域。该区域视编译器的不同而不同，也视返回类型的不同而不同。例如，BC 将一个返回整型的数存放在栈区(见 5.4 节)的位置。在存放到一个专用的区域后，再将其变量的地址传给调用函数，以使调用

函数把它作为函数返回值。

编译器遇到一个函数调用时，需要判断该函数调用是否正确，该机制即函数原型。

5.3 全局变量与局部变量

1. 程序的内存区域

并不是所有的变量时时刻刻都是可知的。一些变量在整个程序中都是可见的，它们称为全局变量。一些变量只能在一个函数中可知，称为局部变量。要了解变量的这些属性，应先弄清程序在内存中的分布区域，见图 5-2。

图 5-2 程序在内存中的区域

一个程序将操作系统分配给其运行的内存块分为 4 个区域：

(1) 代码区，存放程序的代码，即程序中的各个函数代码块。

(2) 全局数据区，存放程序的全局数据和静态数据。

(3) 堆区，存放程序的动态数据。

(4) 栈区，存放程序的局部数据，即各个函数中的数据。

2. 全局变量

在函数外边访问的变量被认为是全局变量，并在程序的每个函数中是可见的。全局变量存放在内存的全局数据区。全局变量由编译器建立，并且初始化为 0，在定义全局变量时，进行专门初始化的除外。

例如，下面的代码定义并使用了全局变量 n：

```
int n = 5;     //全局变量

int main()
{
   int m = n;
   //...
}
void func()
{
   int s;
```

```
    n = s;
    //...
}
```

n 在任何函数的外部定义。n 被初始化为 5，如果 n 不在定义时初始化，则 C++将其初始化为 0。main()函数使用变量 n，函数 func()修改变量 n。两个函数都访问了同一个内存区域。这样定义的全局变量 n 在所有函数中都可见。如果一个函数修改了 n，则所有其他的函数都会看到修改后的变量。

全局变量在主函数 main()运行之前就开始存在了。所以主函数中可以访问 n 变量。全局变量通常在程序顶部定义。全局变量一旦定义后就在程序的任何地方可知。可以在程序中间的任何地方定义全局变量，但要在任何函数之外。**全局变量定义之前的所有函数定义，不会知道该变量。**例如：

```
int main()
{
    int m = n;          //error: n 无定义
    //...
}

int n;                  //全局变量

void func()
{
    int s;
    n = s;
    //...
}
```

该代码中的全局变量 n 不能被主函数 main()访问。编译该代码，将会引起 main()中的 m 初始化语句报告一个“n 无定义”的错误。

3. 局部变量

在函数内部定义的变量仅在该函数内是可见的。另外，局部变量的类型修饰是 auto，表示该变量在栈中分配空间，但习惯上都省略 auto。例如：

```
int main()
{
    int n;              //等价于 auto int n;
    //...
}

void func()
{
    int n;
    //...
}
```

代码中两个函数都包含一个变量定义语句。在函数内定义的变量局部于该函数。main()函数中有一个变量 n，func()函数中也有一个变量 n，但它们是两个不同位置的变量。

一个函数可以为局部变量定义任何名字，而不用担心其他函数使用过同样的名字。这

个特点和局部变量的存在性使 C++适合于由多个程序员共同参与的编程项目。项目管理员为程序员指定编写函数的任务，并为程序提供参数和期望的返回值。然后，程序员着手编写函数，而不用了解程序的其他部分和项目中其他程序员所使用的变量名。

函数中的局部变量存放在栈区。在函数开始运行时，局部变量在栈区被分配空间，函数退出时，局部变量随之消失。

局部变量没有默认初始化。如果局部变量不被显式初始化，那么，其内容是不可预料的。例如：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch5_1.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
int func1();
int func2();

int main()
{
   func1();
   cout <<func2() <<endl;
}

int func1()
{
   int n = 12345;
   return n;
}

int func2()
{
   int m;
   return m;        //warning:possible use of 'm' before definition
}
```

运行结果为：

```
12345
```

主函数 main()先后调用了函数 func1()和 func2()，它们都是无参并返回整数的函数。在 func1()中，定义了局部变量 n，并给其初始化为 12345。在 func2()中，定义了局部变量 m，没有初始化。可是在将该变量值返回后，在主函数中输出该值，却发现为 12345，恰好就是 func1()函数中初始化的值。这说明，func2()中，没有显式初始化的局部变量 m，C++也未给其默认初始化，其值保留为原内存位置的值。那么，原内存位置为什么恰巧是存放值 12345 的位置呢？请见下节“函数调用机制”。

5.4 函数调用机制

栈是一种数据结构，它的工作原理就像在子弹匣中压子弹一样，最先压入的子弹要等到

最后才飞射出去，而最后压入的子弹则首先飞射出去。

C++的函数调用过程，需要调用初始化和善后处理的环节。函数调用的整个过程就是栈空间操作的过程。函数调用时，C++首先：

(1) 建立被调函数的栈空间。

(2) 保护调用函数的运行状态和返回地址。

(3) 传递参数。

(4) 将控制转交被调函数。

例如，下面的代码在主函数中调用一个函数，该函数又调用了另一个函数，它得到图 5-3 中所示的内存布局：

```
void funcA(int,int);
void funcB(int);

int main()
{
   int a = 6,b = 12;
   funcA(a,b);
}

void funcA(int aa,int bb)
{
   int n = 5;
   //...
   funcB(n);
}

void funcB(int s)
{
   int x;
   //...
  }
```

该函数运行的栈区分布图是示意性的。**函数的栈操作原理是一致的，但具体的实现因编译器不同而不同**。介绍函数栈的操作具体过程之目的是要让读者对函数运行的机制有一个感性的认识。

在图 5-3 中，主函数 main()返回的地址应该是在操作系统环境中。其参数是操作系统传递过来的，在 8.9 节中将对其进行介绍。主函数定义了两个局部变量 a 和 b，a 和 b 的次序并不一定如图所示，这些实现的细节都是与 C++标准无关的。我们观察栈内存时，可以忽略返回地址和调用函数运行状态。

当主函数调用 funcA()时，funcA()着手保护调用函数的地址等数据，分配两个形参的空间，将主函数的实参传递过来。所以 funcA()中的形参 aa 和 bb 分别为 6 和 12。

funcA()的栈区和 main()的栈区是互相独立的。在 funcA()中不能访问 main()中的局部变量 a 和 b。通过参数传递，使得 funcA()中的形参赋有 main()函数中的实参值。funcA()可以修改其变量 aa 和 bb，但始终不会影响 main()中的 a 和 b。这便是 C++函数的参数传值特性。

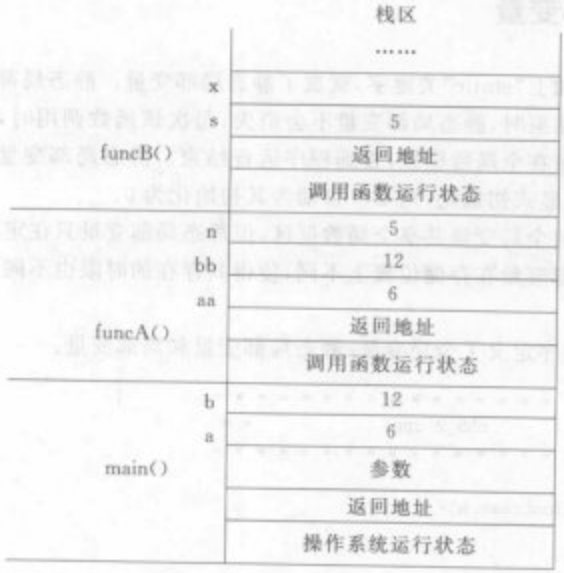

图 5-3　函数调用机制中的栈结构

在函数 funcA()中，定义了一个局部变量 n，并以该变量作实参来调用函数 funcB()。同样，funcB()建立了它自己的栈空间，保护 funcA()的返回地址，获取参数值，随后运行它自己的函数体语句。

栈是有限的资源，每次调用一个函数，剩余的栈空间会逐渐减少。可以看出，如果一层层地调用下去，或者，过多的定义局部变量特别是数组（将在第 7 章介绍），最后可能导致栈空间枯竭而引起程序运行出错。如果程序确实要占用相当大的栈空间，可以在连接前通过设置栈空间大小来改善。

函数在返回时，将把返回值保护在临时变量空间中（如果有返回值的话）。然后恢复调用函数的运行状态，释放栈空间，使其属于调用函数栈空间的一部分，最后根据返回地址，回到调用函数。

图 5-4 是 funcB()返回到 funcA()，funcA()又回到 main()时的栈内容。可以看出，返回到主函数后，funcA()和 funcB()的局部变量和形参都消失，但对应内存中的内容还是存在着，这就是上节中，程序 ch5_1.cpp 得到该运行结果的原因。

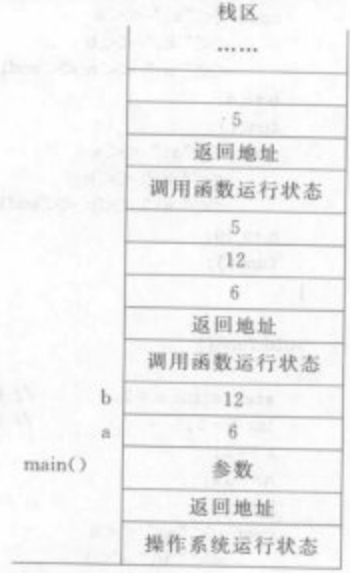

图 5-4　函数返回到 main()时的内存情况

5.5 静态局部变量

在局部变量前加上“static”关键字，就成了静态局部变量。静态局部变量存放在内存的全局数据区。函数结束时，静态局部变量不会消失，每次该函数调用时，也不会为其重新分配空间。它始终驻留在全局数据区，直到程序运行结束。静态局部变量的初始化与全局变量类似，如果不为其显式初始化，则 C++ 自动为其初始化为 0。

静态局部变量与全局变量共享全局数据区，但静态局部变量只在定义它的函数中可见。静态局部变量与局部变量在存储位置上不同，使得其存在的时限也不同，导致对这两者操作的运行结果也不同。

例如，下面的程序定义了全局变量、静态局部变量和局部变量：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch5_2.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void func();
int n = 1;        //全局变量

int main()
{
   static int a;              // 静态局部变量
   int b = -10;               // 局部变量
   cout <<"a:" <<a
        <<" b:" <<b
        <<" n:" <<n <<endl;
   b += 4;
   func();
   cout <<"a:" <<a
        <<" b:" <<b
        <<" n:" <<n <<endl;
   n += 10;
   func();
}

void func()
{
   static int a = 2;          // 静态局部变量
   int b = 5;                 // 局部变量
   a += 2;
   n += 12;
   b += 5;
   cout <<"a:" <<a
        <<" b:" <<b
        <<" n:" <<n <<endl;
}
```

运行结果为：

```
a:0 b:-10 n:1
a:4 b:10 n:13
a:0 b:-6 n:13
a:6 b:10 n:35
```

程序中主函数 main()两次调用了 func()函数，从运行结果可以看出，程序控制每次进入 func()函数时，局部变量 b 都被初始化。而静态局部变量 a 仅在第一次调用时被初始化，第二次进入该函数时，不再进行初始化，这时它的值是第一次调用后的结果值 4。main()函数中的变量 a 和 b 与 func()函数中的变量 a 和 b 空间位置是不一样的，所以相应的值也不一样。关于变量作用域和可见性的进一步讨论见第 6 章。

静态局部变量的用途有许多：可以使用它确定某函数是否被调用过。使用它保留多次调用的值。

5.6 递归函数

1. 什么是递归函数(recursive function)

递归函数即自调用函数，在函数体内部直接或间接地自己调用自己，即函数的嵌套调用是函数本身。

例如，下面的程序为求 n!：

```
long fact(int n)
{
   if(n==1)
    return 1;

   return fact(n-1)*n;      //出现函数自调用
}
```

2. 函数调用机制的说明

任何函数之间不能嵌套定义，调用函数与被调用函数之间相互独立(彼此可以调用)。

发生函数调用时，被调函数中保护了调用函数的运行环境和返回地址，使得调用函数的状态可以在被调函数运行返回后完全恢复，而且该状态与被调函数无关。

被调函数运行的代码虽是同一个函数的代码体，但由于调用点、调用时状态、返回点的不同，可以看作是函数的一个副本，与调用函数的代码无关，所以函数的代码是独立的。

被调函数运行的栈空间独立于调用函数的栈空间，所以与调用函数之间的数据也是无关的。函数之间靠参数传递和返回值来联系，函数看作为黑盒。

这种机制决定了 C/C++允许函数递归调用。

3. 递归调用的形式

递归调用有直接递归调用和间接递归调用两种形式。

直接递归即在函数中出现调用函数本身。

例如，下面的代码求斐波那契数列第 n 项。斐波那契数列的第一和第二项是 1，后面每

一项是前二项之和，即 1,1,2,3,5,8,13,...。代码中采用直接递归调用：

```
long fib(int x)
{
   if(x>2)
      return (fib(x-1)+fib(x-2));  // 直接递归
   else
      return 1;
}
```

间接递归调用是指函数中调用了其他函数，而该其他函数却又调用了本函数。

例如，下面的代码定义两个函数，它们构成了间接递归调用：

```
int fn1(int a)
{
   int b;
   b= fn2(a + 1);     // 间接递归
   //...
}

int fn2(int s)
{
   int c;
   c= fn1(s - 1);     // 间接递归
   //...
}
```

上例中，fn1()函数调用了 fn2()函数，而 fn2()函数又调用了 fn1()函数。

4. 递归的条件

(1) 须有完成函数任务的语句。

例如，下面的代码定义了一个递归函数：

```
#include <iostream.h>

void count(int val)     //递归函数可以没有返回值
{
   if(val>1)
      count(val - 1);

   cout <<"ok:" <<val <<endl;     //此语句完成函数任务
}
```

该函数的任务是在输出设备上显示“ok:整数值”。

(2) 一个确定是否能避免递归调用的测试。

例如，上例的代码中，语句“if(val>1)”便是一个测试，如果不满足条件，就不进行递归调用。

(3) 一个递归调用语句。

该递归调用语句的参数应该逐渐逼近不满足条件，以致最后断绝递归。

例如，上面的代码中，“count(val - 1);”是一个递归调用，参数值在渐渐变小，这种发

展趋势能使测试"if(val>1)"最终不满足。

(4) 先测试,后递归调用。

在递归函数定义中,必须先测试,后递归调用。也就是说,递归调用是有条件的,满足了条件后,才可以递归。

例如,下面的代码无条件调用函数自己,造成无限制递归,终将使栈空间溢出:

```
#include <iostream.h>

void count(int val)
{
   count(val - 1);      //无限制递归
   if(val>1)            //该语句无法到达
       cout <<"ok:" <<val <<endl;
}
```

5. 消去递归

大多数递归函数都能用非递归函数来代替。例如,下面的代码求两个整数 a,b 的最大公约数,用递归和非递归函数分别定义之:

```
long gcd1(int a, int b)      //递归版
{
   if(a % b == 0)
       return b;
   return gcd1(b, a % b);
}

long gcd2(int a, int b)      //非递归版
{
   int temp;
   while(b!= 0)
   {
       temp = a % b;
       a = b;
       b = temp;
   }
   return a;
}
```

思考:将求 n! 的递归函数非递归化。

6. 递归的评价

递归的目的是简化程序设计,使程序易读。

但递归增加了系统开销。时间上,执行调用与返回的额外工作要占用 CPU 时间。空间上,随着每递归一次,栈内存就多占用一截。

相应的非递归函数虽然效率高,但却比较难编程,而且相对来说可读性差。

现代程序设计的目标主要是可读性好。随着计算机硬件性能的不断提高,程序在更多的场合强调可读性,它似乎在鼓励递归设计。但是,递归函数如果很缓慢地逼近到递归结束条件,会使性能大大下降,所以实际上只有很少一部分类似于最大公约数计算的递归设计才被接受。

5.7 内联函数

1. 内联函数的需要性

内联函数也称内嵌函数，它主要是解决程序的运行效率。

函数调用需要建立栈内存环境，进行参数传递，并产生程序执行转移，这些工作都需要一些时间开销。

有些函数使用频率高，但代码却很短。

例如，下面的代码中，频繁地调用一个小函数：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch5_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int isnumber(char);              //函数声明

int main()
{
   char c;
   while((c = cin.get())!= '\n')
   {
     if( isnumber(c) )           //调用一个小函数
       cout <<"you entered a digit\n";
     else
       cout <<"you entered a non-digit\n";
   }
}

int isnumber(char ch)            //函数定义
{
   return (ch>= '0' && ch<= '9')? 1:0;
}
```

程序中不断到设备中读取数据，频繁调用 isnumber()函数。为了提高效率，可将程序改为：

```
#include <iostream.h>
int main()
{
   char c;
   while((c = cin.getc())!= '\n')
   {
     if((ch>= '0' && ch<= '9')? 1:0 )     //修改处：直接计算表达式
       cout <<"you entered a digit\n";
     else
```

```
        cout <<"you entered a non-digit\n";
    }
}
```

该程序在 if 语句中用表达式替换了函数调用。在程序运行上，提高了一些执行效率，因为免去了大量的函数调用开销。

由于 isnumber()比相应的表达式可读，所以若程序中多处出现 isnumber()的替换，就会降低程序的可读性。我们既要用函数调用来体现其结构化和可读性，又要使效率尽可能地高。

2. 解决办法

将 isnumber()函数声明为 inline，即在函数声明和定义中：

```
inline int isnumber(char);

inline int isnumber(char c)
{
    return (ch> = '0' && ch< = '9')?1:0;
}
```

编译器看到 inline 后，为该函数创建一段代码，以便在后面每次碰到该函数的调用都用相应的一段代码来替换。内联函数可以在一开始仅声明一次。例如下面的代码表达了一个内联函数：

```
# include <iostream.h>
inline int isnumber(char);      //inline 函数声明
int main()
{
    char c;
    while((c = cin.getc())!= '\n')
    {
        if( isnumber(c) )      //调用一个小函数
            cout <<"you entered a digit\n";
        else
            cout <<"you entered a non - digit\n";
    }
}

int isnumber(char ch)      //此处无 inline,视为 inline
{
    return (ch> = '0' && ch< = '9')? 1:0;
}
```

3. 先声明后调用

内联函数必须在被调用之前声明或定义。因为内联函数的代码必须在被替换之前已经生成被替换的代码，因此，下面的代码不会像预计的那样被编译：

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

```
}

int isnumber(char);     //此处无 inline

int main()
{
  char c;
  while((c = cin.getc())!= '\n')
  {
    if( isnumber(c) )     //调用一个小函数
      cout <<"you entered a digit\n";
    else
      cout <<"you entered a non - digit\n";
  }
}

inline int isnumber(char ch)     //此处为 inline
{
  return (ch>= '0' && ch<= '9')? 1:0;
}
```

编译程序不认为那是内联函数，对待该函数如普通函数那样，产生该函数的调用代码，并进行连接。

4. 内联函数的函数体限制

内联函数中，不能含有复杂的结构控制语句，如 switch 和 while。如果内联函数有这些语句，则编译将该函数视同普通函数那样产生函数调用代码。

另外，递归函数（自己调用自己的函数）是不能被用来做内联函数的。

内联函数只适合于只有 1～5 行的小函数。对一个含有许多语句的大函数，函数调用和返回的开销相对来说微不足道，所以也没有必要用内联函数实现。

→ 内联函数与宏定义

在 C 中，常用预处理语句 #define 来代替一个函数定义。例如：

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

该语句使得程序中每个出现 MAX(a,b)函数调用的地方都被宏定义中后面的表达式((a)>(b)? (a):(b))所替换。

宏定义语句的书写格式有过分的讲究，MAX 与括号之间不能有空格，所有的参数都要放在括号里。尽管如此，它还是有麻烦：

```
int a = 1,b = 0;
MAX(a ++ ,b);          //a 被增值 2 次
MAX(a ++ ,b + 10);     //a 被增值 1 次
MAX(a,"Hello");        //错误地比较 int 和字符串，没有参数类型检查
```

MAX()函数的求值会由于两个参数值的大小不同而产生不同的副作用。

MAX(a ++ ,b)的值为 2，同时 a 的值为 3；

MAX(a ++ ,b+10)的值为 10，同时 a 的值为 2。

如果是普通函数，则 MAX(a,"Hello")会受到函数调用的检查，但此处不会因为两个参数类型不同而被编译拒之门外。幸运的是，通过一个内联函数可以得到所有宏的替换效能

和所有可预见的状态以及常规函数的类型检查：

```
inline int MAX(int a,int b)
{
  return a>b?a:b;
}
```

5.8 重载函数

1. 重载的需要性

在C中，每个函数必须有其唯一的名字，但有时，这会令人生厌。

例如，求一个数的绝对值，由于命名唯一，所以对于不同的类型需要不同名字的函数：

```
int abs(int);
long labs(long);
double fabs(double);
```

这几个函数所做的事情是一样的，都是求绝对值。因此，使用3个不同的函数名，看上去很笨拙，若给以同样的名字就会方便得多。对于在不同类型上作不同运算而又用同样的名字的情况，则称之为重载。这种技术在C++中早已用于基本数据类型运算，如加法只有一个名字+，但它可以用来加整数值、浮点值和指针值。

例如，上述3个函数的声明可以改为：

```
int abs(int);
long abs(long);
double abs(double);
```

C++用一种函数命名技术可以准确判断出应该使用哪个abs()函数。例如：

```
abs(-10);              //调用 int abs(int);
abs(-1000000);         //调用 long abs(long);
abs(-12.23);           //调用 double abs(double);
```

2. 匹配重载函数的顺序

在调用一个重载函数f()时，编译器必须搞清函数名f究竟是指哪个函数。这是靠将实参类型和所有被调用的f()函数的形参类型一一比较来判定的。按下述3个步骤的先后顺序找到并调用那个函数：

(1) 寻找一个严格的匹配，如果找到了，就用那个函数。

(2) 通过内部转换寻求一个匹配，只要找到了，就用那个函数。

(3) 通过用户定义的转换寻求一个匹配，若能查出有唯一的一组转换，就用那个函数(见18.5节)。

例如，重载函数print()的匹配：

```
void print(double);
void print(int);
```

```
void func()
{
   print(1);            //匹配 void print(int);
   print(1.0);          //匹配 void print(double);
   print('a');          //匹配 void print(int);
   print(3.1415f);      //匹配 void print(double);
}
```

按严格匹配规则，保证把实参 1 作为整数打印，1.0 作为浮点数打印。

对于 int 形参，0，char 和 short int 都是严格匹配。

对于 double 形参，float 也是严格匹配。

C++允许 int 到 long，int 到 double 的转换。当实参是整数，而重载函数一为 long 型参数，一为 double 型参数时，应该给以一个显式转换。

例如，对于重载函数 print()声明，其下面的函数调用将引起错误：

```
void print(long);
void print(double);

void func(int a)
{
   print(a);            //error: 因为有二义性
}
```

该代码在 BC 中会导致二义性编译错误(Ambiguity between "print(long)" and "print(double)")，应该指明是"print(long(a));"还是"print(double(a));"，以分辨这种含混的调用。

3. 使用说明

(1) C++的函数如果在返回类型、参数类型、参数个数、参数顺序上有所不同，则认为是不同的。但重载函数如果仅仅是返回类型不同，则是不够的。

例如，下面的声明是错误的：

```
void func(int);
int func(int);
```

编译器无法区分函数调用"func(3)"是指上述哪一个重载函数。因此**重载函数至少在参数个数、参数类型或参数顺序上有所不同。**

(2) typedef 定义的类型只能使之相同于一个已存在的类型，而不能建立新的类型，所以不能用 typedef 定义的类型名来区分重载函数声明中的参数。

例如，下面的代码实际上是同一个函数：

```
typedef INT int;

void func(int x) { //... }
void func(INT x) { //... }     //error: 函数重复定义
```

编译器不能区分这两个函数的差别，INT 只不过是 int 的另一种称呼而已。

(3) 让重载执行不同的功能，是不好的编程风格。同名函数应该具有相同的功能。如果定义一个 abs()函数而返回的却是一个数的平方根，则该程序的可读性受到破坏。

→ 关于重载函数的内部实现

C++用名字粉碎(name mangling)的方法来改变函数名,以区分参数不同的同名函数。名字粉碎是十分简单的过程,一系列代码被附加到函数名上以指示参数类型以及它们出现的次序。例如,用 v,c,i,f,l,d,r 分别表示 void,char,int,float,long,double,long double,则重载函数:

```
int f(char a);
int f(char a, int b, double c);
```

在内部分别被表示为:f_c 和 f_cid,而函数名为:

```
int f_cid();
```

则在内部表示为:f_cidv。它与上面的重载函数相区别。

名字粉碎是看不到的,它通常放在目标文件中,遇到连接错误时,有时可能会看到粉碎了的名字。

5.9 默认参数的函数

1. 默认参数的目的

C++可以给函数定义默认参数值。通常,调用函数时,要为函数的每个参数给定对应的实参。例如:

```
void delay(int loops);          //函数声明

void delay(int loops)           //函数定义
{
   if(loops == 0)
        return;
   for(int i = 0; i<loops; i++);
}
```

无论何时调用 delay()函数,都必须给 loops 传一个值以确定时间。但有时需要用相同的实参反复调用 delay()函数。C++可以给参数定义默认值。如果将 delay()函数中的 loops 定义成默认值 1000,只需简单地把函数声明改为:

```
void delay(int loops = 1000);
```

这样,无论何时调用 delay()函数,都不用给 loops 赋值,程序会自动将它当作值 1000 进行处理。例如,调用:

```
delay(2500);        //loops 设置为 2500
delay();            //ok: loops 采用默认值 1000
```

调用中,若不给出参数,则按指定的默认值进行工作。

允许函数默认参数值,是为了让编程简单,让编译器做更多的检查错误工作。

2. 默认参数的声明

默认参数在函数声明中提供,**当又有声明又有定义时,定义中不允许默认参数**。如果函

数只有定义，则默认参数才可出现在函数定义中。例如：

```
void point(int = 3, int = 4);      //声明中给出默认值

void point(int x, int y)           //定义中不允许再给出默认值
{
   cout <<x <<endl;
   cout <<y <<endl;
}
```

3. 默认参数的顺序规定

如果一个函数中有多个默认参数，则形参分布中，默认参数应从右至左逐渐定义。当调用函数时，只能向左匹配参数。例如：

```
void func(int a = 1, int b, int c = 3, int d = 4);      //error

void func(int a, int b = 2, int c = 3, int d = 4);      //ok
```

对于第 2 个函数声明，其调用的方法规定为：

```
func(10,15,20,30);          //ok: 调用时给出所有实参
func();                     //error: 参数 a 没有默认值

func(12,12);                //ok: 参数 c 和 d 默认

func(2,15,,20);             //error: 只能从右到左顺序匹配默认
```

4. 默认参数与函数重载

默认参数可将一系列简单的重载函数合成为一个。例如，下面 3 个重载函数：

```
void point(int,int) {  //... }
void point(int a) { return point(a,4); }
void point() { return point(3,4); }
```

可以用下面的默认参数的函数来替代：

```
void point(int = 3, int = 4);
```

当调用“point();”时，即调用“point(3,4);”它是第 3 个声明的重载函数。

当调用“point(6);”时，即调用“point(6,4);”，它是第 2 个声明的重载函数。

当调用“point(7,8);”时，即调用第 1 个声明的重载函数。

如果一组重载函数（可能带有默认参数）都允许相同实参个数的调用，将会引起调用的二义性。例如：

```
void func(int);                 //重载函数之一
void func(int, int = 4);        //重载函数之二：带有默认参数
void func(int = 3, int = 4);    //重载函数之三：带有默认参数

func(7);                        //error: 到底调用 3 个重载函数中的哪个?
func(20,30)                     //error: 到底调用后面 2 个重载函数的哪个?
```

5. 默认值的限定

默认值可以是全局变量、全局常量，甚至是一个函数。例如：

```
int a = 1;
int fun(int);

int g(int x = fun(a));     //ok: 允许默认值为函数
```

默认值不可以是局部变量，因为默认参数的函数调用是在编译时确定的，而局部变量的位置与值在编译时均无法确定。例如：

```
void fun()
{
   int i;
   void g(int x = i);     //error:处理 g()函数声明时，i 不可见
}
```

小结

随着程序量和程序复杂度的不断增加，最好的办法是把程序分成更小，更容易管理的模块，这种模块就是函数。

函数名最好能反映出所要完成的任务。

函数可以把数据返回给调用者，若函数要返回一个值，必须在函数名前规定返回值的类型，若函数没有返回值，则类型为 void。

程序通过参数把信息传递给函数，若函数需要接受参数，就必须给参数指定名称及类型。

C++必须知道函数的返回类型以及接受的参数个数和类型，如果函数的定义出现在函数调用之后，就必须在程序的开始部分用函数原型进行说明。

局部变量是在函数内部定义的，只能被定义该变量的函数访问。全局变量是指其作用域贯穿程序始终的变量。定义全局变量要在程序开始时进行，并且放在所有函数的外面。静态局部变量是在函数内部定义，但生命期却随函数的第一次被调用而产生，随程序的结束而结束，静态局部变量只能在定义该变量的函数中可见。

函数调用机制是由栈操作的过程实现的。函数可以递归调用。函数定义不能放在任何函数定义的里面。

内联函数是为了提高编程效率而实现的，它克服了用 # define 宏定义所带来的弊病。

函数重载允许用同一个函数名定义多个函数。连接程序会根据传递给函数的参数数目、类型和顺序调用相应的函数。函数重载使程序设计简单化，程序员只要记住一个函数名，就可以完成一系列相关的任务。

在函数定义中通过赋值运算，即可指定默认参数值。一旦程序在调用函数时默认了参数值，函数就使用默认参数值。不允许在参数中间使用默认值。指定默认参数值可以使函数的使用更为简单，同时也增强了函数的可重用性。

练习

5.1 将 ch4_8.cpp 的程序改写为主函数调用 isprime()函数的形式，以确定该数是否为素数。

5.2 将 ch4_9.cpp 的程序改写为主函数调用 integral()函数的形式，以求积分值。

5.3 将习题 4.9 打印乘法九九表改用函数调用的形式，适当取函数名，分别调用三种函数以输出不同格式。

5.4 分析下列程序，写出运行结果。

```
#include <iostream.h>

void func();
int n = 1;

int main()
{
   static int x = 5;
   int y;
   y = n;
   cout << "Main -- x = " << x
        << ",y = " << y
        << ",n = " << n << endl;
   func();
   cout << "Main -- x = " << x
        << ",y = " << y
        << ",n = " << n << endl;
   func();
}

void func()
{
   static int x = 4;
   int y = 10;

   x += 2;
   n += 10;
   y += n;
   cout << "Func -- x = " << x
        << ",y = " << y
        << ",n = " << n << endl;
}
```

5 1 1
6 21 11
5 1 11

8 31 21

5.5 用非递归的函数调用形式求斐波那契数列第 n 项。

5.6 以下函数 poly 是用递归方法计算 x 的 n 阶勒让德多项式的值。已有调用语句"p(n,x);"，编写 poly 函数。递归公式如下：

$poly_n(x) = 1$　　当 $n = 0$ 时；

$poly_n(x) = x$　　当 $n = 1$ 时；

$poly_n(x) = ((2n-1) * x * poly_{n-1}(x) - (n-1) * poly_{n-2}(x))/n$ 当 $n > 1$ 时。

5.7 已知 $f(x)=\cos(x)-x$。x 的初始值为 3.14159/4，用牛顿法求解方程 $f(x)=0$ 的近似解，要求精确到 10^{-4}。$f(x)$的牛顿法为：

$$x_{n+1}=x_n-(\cos(x_n)-x_n)/(\sin(x_n)-1)$$

5.8 编写程序，其中包含三个重载的 display()函数。第一个函数输出一个 double 值，前面用字符串“A double:”引导；第二个函数输出一个 int 值，前面用字符串“A int:”引导；第三个函数输出一个 char 字符，前面用字符串“A char:”引导。在主函数中，分别用 double，float，int，char 和 short 型变量去调用 display()函数，并对结果做简要说明。

5.9 将习题 4.10 用递归函数方法求解。

第6章 程序结构

要编好C++程序，就必须要对C++的程序结构有一个全面的了解。所有的C++程序都是由一个或多个函数构成的。一个C++程序可以由一个或多个包含若干函数定义的源文件组成。C++的编译器和连接器把构成一个程序的若干源文件有机地联络在一起，最终产生可执行程序。学习本章后，要求掌握外部存储类型和静态存储类型在多文件程序中的联络作用，理解作用域、可见性与生命期的概念，学会使用头文件，理解多文件结构，理解编译预处理的概念。

6.1 外部存储类型

一个程序在很小的规模下，可以用一个源文件来完整表达。本章之前示例的程序都是由单个文件构成的完整程序。一般具有应用价值的程序由多个源文件组成。根据C++程序的定义，其中只有一个源文件具有主函数main()，而其他的文件不能含有main()，否则程序不知道该从何处开始执行了。

构成一个程序的多个源文件之间，通过声明数据或函数为外部的(extern)来进行沟通。例如，下面两个文件构成了一个程序，该程序由一个工程文件ch6_1.prj定义。工程文件和源文件中的内容分别为：

```
//- - - - - - - - - - - - - - - - -
//- -          ch6_1.prj          - -
//- - - - - - - - - - - - - - - - -

ch6_1.cpp
ch6_1_1.cpp
```

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                ch6_1.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void fn1();
void fn2();
int n;

int main()
{
   n = 3;
   fn1();              //fn1()函数的定义在本文件中

   cout <<n <<endl;
}
```

```
void fn1()
{
   fn2();              //fn2()函数的定义不在本文件中
}


//* * * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch6_1_1.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * *

extern int n;          //n由另一个源文件定义

void fn2()             //fn2()函数用于另一个源文件

{
   n=8;                //使用n
}
```

运行结果为：

```
c:\>ch6_1
8
```

文件 ch6_1.cpp 中含有主函数 main()，main()中调用了函数 fn1()，函数 fn1()调用了函数 fn2()，所以在 main()函数的前面应有函数 fn1()的声明，在函数 fn1()的定义之前应有函数 fn2()的声明。**所有函数声明一般都放在源文件的开始位置。**

文件 ch6_1.cpp 中定义了全局变量 n 以供 main()函数使用。

文件 ch6_1.cpp 中只含有函数 main()和 fn1()的定义，fn2()函数的定义在 ch6_1_1.cpp 中。

文件 ch6_1_1.cpp 中定义了函数 fn2()。函数 fn2()中要使用在 ch6_1.cpp 中定义的全局变量 n，为此在文件开头声明了带 extern 的 int n，它表示该变量 n 不在本文件中分配空间，而在程序的其他文件中分配空间(变量定义)。

默认的函数声明或定义总是 extern 的，所以文件 ch6_1.cpp 中，为了调用 fn1()和 fn2()，在文件一开始声明的函数原型等价于：

```
extern void fn1();
extern void fn2();
```

它们告诉连接程序，在所有组成该程序的文件(这里是 ch6_1.cpp 和 ch6_1_1.cpp)中搜索该函数的定义。其中，fn1()在本文件中定义，fn2()在 ch6_1_1.cpp 中定义。

假如一个程序由十个源文件构造而成，每个源文件都必须访问一个全局变量。在这种情形下，其中的九个文件必须把变量声明为 extern，另外一个则不能。虽然在包含 main()函数的源文件中分配变量是最合理的，但哪个文件真正分配该变量(全局变量定义)是无关紧要的。

带 extern 的变量说明是变量声明，不是变量定义。

如果共同的变量一次都没有定义，或者在各个文件中分别定义，造成定义多次，或者声明的类型不一致，都会造成直接或间接的错误。例如：

```
//file1.cpp
```

```
int a = 5;
int b = 6;
extern int c;

//file2.cpp

int a;                  //error: 多次定义
extern double b;        //error: 类型不一致
extern int c;           //error: 无定义
```

在上面的代码中，两个源文件都以常规方式定义变量 a，没有一个显式声明 extern，这时，其行为将依赖于编译器。VC 会通过编译，但在连接时，给出一个“变量 a 的定义已经在前一个文件中定义过”(int a already defined in file1.obj)的错误。但 BC 则将每个文件的变量定义都看作是全局静态变量(见 6.2 节)。所以，程序将会运行，但两个文件中的变量 a 是互不相干的。

两个文件中 b 的类型不一样，这时，VC 将在连接时报告一个“未定的外部名”(link error:unresolved external)错误，而 BC 却不能发现该错误而使程序错误地运行下去。

两个源文件又都声明变量 c 为 extern，这时，编译也不会发生问题，但连接时却会找不到该变量，产生一个连接错误，因为没有一个文件为该变量分配空间。

不论各编译器和连接器对外部存储类型的反应如何，只要理解了 extern 的作用，就可保证编程的正确性。

6.2 静态存储类型

1. 静态全局变量

在全局变量前加一个 static，使该变量只在这个源文件中可用，称之为全局静态变量。全局静态变量就是静态全局变量。

在前面几章，由于程序都是由一个源文件实现，所以全局变量与全局静态变量是没有区别的。但是在多文件组成的程序里，全局变量与全局静态变量是不同的。全局静态变量使得该变量成为定义该变量的源文件所独享。

例如，两个源文件组成一个程序，其工程文件为 ch6_2.prj。其中一个源文件定义了“int n;”另一个源文件定义了“static int n;”则程序给它们分别分配了空间，两个值互不干扰：

```
//- - - - - - - - - - - - - - - - - -
//- -          ch6_2.prj          - -
//- - - - - - - - - - - - - - - - - -

ch6_2.cpp
ch6_2_1.cpp

//* * * * * * * * * * * * * * * * * * * * * *
//* *                ch6_2.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
```

```
int n;
void fn();
int main()
{
   n = 20;
   cout <<n <<endl;
   fn();
}
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch6_2_1.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *
#include <iostream.h>

static int n;                                    //默认初始化为 0

void fn()
{
   n++;
   cout <<n <<endl;
}
```

运行结果为：

```
c:\>ch6_2
20
1
```

该程序分别编译后，连接为 ch6_2.exe。运行 ch6_2 便得到上述结果。

函数 fn()输出 1 而不是 21，表示两个变量互不干涉。

静态全局变量对组成该程序的其他源文件是无效的。

例如，下面的代码在 file1.cpp 中声明全局变量 n，在 file2.cpp 中定义全局静态变量 n：

```
//file1.cpp

#include <iostream.h>

void fn();
extern int n;                                    //表示 n 仅是一个声明，将由别的文件定义
int main()
{
   n = 20;
   cout <<n <<endl;
   fn();
}

//file2.cpp

#include <iostream.h>

static int n;                                    //默认初始化为 0

void fn()
```

```
{
   n++;
   cout <<n <<endl;
}
```

文件 file1.cpp 和 file2.cpp 分别都能通过编译，但连接时，file1.cpp 中的变量 n 找不到定义，产生连接错误。

如果把文件 file1.cpp 中的“extern int n;”去掉，那么编译会由于没有 n 的定义而出错。这说明，file2.cpp 中的静态变量 n 与 file1.cpp 中的变量 n 毫不相干。

使一个变量只在一个源文件中全局使用有时是必要的。第一，不必担心另外源文件使用它的名字，该名字在源文件中是唯一的。第二，源文件的全局变量不能被其他源文件所用，不能被其他源文件所修改，保证变量的值是可靠的。

2. 静态函数

函数的声明和定义默认情况下在整个程序中是 extern 的。有时候，你可能需要使某个函数只在一个源文件中有效，不能被其他源文件所用，这时在函数前面加上 static。

例如，下面的代码在 file2.cpp 中定义的函数不能被 file1.cpp 调用：

```
//file1.cpp

void fn();
void staticFn();                                    //link error

int main()
{
   fn();
   staticFn();
}

//file2.cpp

# include <iostream.h>

static void staticFn();
void fn();

void fn()
{
   staticFn();
   cout <<"this is fn()\n";
}

void staticFn()
{
   cout <<"this is staticFn()\n";
}
```

在文件 file1.cpp 中，主函数 main()调用了 fn()和 staticFn()，但由于函数 staticFn()没有定义，所以尽管通过了编译，但通不过连接，产生一个找不到函数 staticFn()定义的连接错误。如果把 file1.cpp 中的 staticFn()声明拿掉，则会产生没有 staticFn()函数定义的编

译错误。这说明，文件 file1.cpp 无法共享 file2.cpp 中的静态函数。

在文件 file1.cpp 中，去掉关于函数 staticFn()的声明和调用，便能通过编译和连接了。

例如，下面的程序修改了上面代码的内容，其工程文件为 ch6_3.prj。其工程文件和源文件内容分别为：

```
//- - - - - - - - - - - - - - - - -
//- -        ch6_3.prj          - -
//- - - - - - - - - - - - - - - - -

ch6_3.cpp
ch6_3_1.cpp

//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch6_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

void fn();

int main()
{
  fn();
}

//* * * * * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch6_3_1.cpp                    * *
//* * * * * * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>

static void staticFn();
void fn();

void fn()
{
   staticFn();
   cout <<"this is fn()\n";
}

void staticFn()
{
   cout <<"this is staticFn()\n";
}
```

运行结果为：

```
c:\>ch6_3
this is staticFn()
this is fn()
```

主函数 main()调用了函数 fn()，fn()在文件 file2.cpp 中定义，所以它在 file2.cpp 中有效，于是可以随意调用该文件的另一个函数 staticFn()。

文件 file1.cpp 由于没有使用输出语句"cout << ...",所以不用包含头文件 iostream.h。静态有两个效果：第一，它允许其他源文件建立并使用同名的函数，而不相互冲突，在很大的编程项目中它是一个优势；第二，声明为静态的函数不能被其他源文件所调用，因为它的名字不能得到。假设要编写逻辑上只能由函数 A 调用的函数 B，其他函数对函数 B 的调用是无意义的，但是要求其他源文件能调用函数 A，把函数 B 声明为静态即能实现。

在文件作用域下声明的 inline 函数默认为 static 存储类型。在文件作用域下声明的 const 的常量默认为 static 存储类型。它们如果加上 extern，则为外部存储类型。

6.3 作用域

作用域是标识符在程序中有效的范围，标识符的引入与声明有关，作用域开始于标识符的声明处。C++的作用域范围分为：局部作用域(块作用域)，函数作用域，函数原型作用域，文件作用域和类作用域(见 11.7 节)。

1. 局部作用域

当标识符的声明出现在由一对花括号所括起来的一段程序(块)内时，该标识符的作用域从声明点开始，到块结束处为止。作用域的范围具有局部性。

例如，下面的代码描述了局部作用域：

```
#include <iostream.h>

void fn(int y)                                    //y 的作用域从此开始
{
   int x = 1;                                     //x 的作用域从此开始
   if(x>y)
     cout <<x <<endl;
   else
     cout <<y <<endl;
   //...
}                                                 //x 和 y 的作用域到此结束
```

语句是一个程序单位，如果在 if 语句和 switch 语句中进行条件测试的表达式中声明标识符，则该标识符的作用域在该语句内。

例如，下面的代码在 if 语句中声明了一个局部作用域的标识符：

```
#include <iostream.h>

void fn()
{
   if(int i = 5)          //i 的作用域从此开始
     i = 2 * i;
   else
     i = 100;
   //i 的作用域到此结束
   cout <<i <<endl;     //error i 无定义
}
```

又例如，下面的代码在 switch 语句中声明了一个块作用域的标识符：

```
#include <iostream.h>

void fn()
{
   switch(int i = f())   //i 作用域从此开始
   {
        case 1:
          cout <<i <<endl;
        //...
   }                     //i 作用域到此结束
   cout <<i <<endl;  //error i 无定义
}
```

在 if 条件表达式判断之后的语句和 else 之后的语句为一个语句块，尽管有时仅仅只是一条语句。

例如，下面的代码错用了 if 语句中声明的变量：

```
#include <iostream.h>

void fn(int n)
{
   if(n>5)
       int i = n;        //整型 i 的作用域从此开始，到此结束
   else
       double i = n;     //double i 的作用域从此开始，到此结束

   cout <<i <<endl;  //error i 无定义
}
```

在 for 语句的第一个表达式中声明的标识符，其作用域在该语句内。

```
#include <iostream.h>

void fn()
{
   int number,1;                    //i 的作用域从此开始
   for(i = 0; i<10; i++)
   {
       int halfNumber;
       if(i % 2)
         number + = 1;
   }

   halfNumber = number/2;           //error halfNumber 无定义
   number = i;                      //ok
}
```

→ 在 C++早些时候的版本，例如 TC++ 3.0，允许 for 语句头中声明的变量之作用域延伸至包含 for 的最小块结束。后来的 ANSI C++标准规定，for 语句头中声明的变量之作用域只在该 for 语句中。但有些编译器仍保留其另一种意义的选择。如 BORLAND C++ 5.0 在 Options|Project|C++Options|C++Compatibility 中，可以选择 for 语句头中声明的变量

之作用域范围。

2. 函数作用域

标号是唯一具有函数作用域的标识符。goto 语句使用标号。标号声明使得该标识符在一个函数内的任何位置均可以被使用。

例如，下面的代码声明了两个标号：

```
#include <iostream.h>

void fn()
{
   goto S;
   int b;
   cin >>b;
   if(b>0)
   {
S:
       goto End;
   }
End:
   Cout <<"All right\n";
}
```

goto 或 switch 语句不应使控制从一个声明的作用域之外跳到该声明的作用域内，因为这种跳转越过了变量的声明语句，使变量不能被初始化。

例如，下面的代码在 switch 语句内作了一个永远也到不了的变量说明：

```
switch(s)
{
   int b = 5;                          //warning 无法到达此处
   case 1:
     b++;                              //error: 变量 b 无定义
   //...
}
```

局部变量不具有函数作用域。

例如，下面的代码描述一个局部变量在声明之前进行了使用：

```
void fn()
{
   int n = 1;
   n = k + 1;                          //error: 因为 k 不是函数作用域，只能从定义处起有效
   int k;                              //k 为块作用域
   //...
}
```

3. 函数原型作用域

函数原型声明(不是函数定义)中所作的参数声明在该作用域中。这个作用域开始于函数原型声明的左括号，结束于函数原型声明的右括号。

例如，下面的代码是函数 Area()的原型声明：

```
void Area(double width, double length);
```

由于参数声明：

```
double width, double length
```

只在括号之内有效，在程序的其他地方使用 width 和 length 必须另外有定义，否则会引起无定义的标识符错。

例如，下面的代码引起一个无定义的标识符编译错误：

```
void Area(double width,double length);

length = 50;                          //error length 无定义
```

所以，在这个函数原型声明中的标识符 width 和 length 是可有可无的。即上面的函数原型等价于下面的函数原型声明：

```
void Area(double, double);
```

但是，参数中有了标识符，可以增强可读性。上面参数中带标识符的函数原型声明使人一看就明白一个参数是宽度值，另一个是长度值。所以，习惯上，在函数原型声明中，都为参数指定一个有说明意义的标识符，而且一般总是与该函数定义中参数的标识符一致。即：

```
void fn(int number);                  //函数声明

//...

void fn(int number)                   //函数定义
{
   //...
}
```

4. 文件作用域

文件作用域是在所有函数定义之外说明的，其作用域从说明点开始，一直延伸到源文件结束。

例如，下面的代码声明了三个全局变量，但由于声明点的差异，使得在函数中，出现了编译错误：

```
//abc.cpp

int number;
static int value;

int main()
{
   a = 1;                             //error a 无定义
   number = 0;
   value = number + 1;
}
```

```
int a;                                    //定义全局变量 a
```

value 是静态全局变量，它是文件作用域的。

静态全局变量是文件作用域的，静态函数也是文件作用域的。所以，文件作用域即为静态全局的。

在头文件的文件作用域中所进行的声明，若该头文件被一个源文件嵌入，则声明的作用域也扩展到该源文件中，直到源文件结束。

例如，cout 和 cin 都是在头文件 iostream.h 的文件作用域中声明的标识符，这两个标识符的作用域延伸到嵌入 iostream.h 的源文件中。

6.4 可见性

可见性从另一角度表现标识符的有效性，标识符在某个位置可见，表示该标识符可以被引用。可见性与作用域是一致的。作用域指的是标识符有效的范围，而可见性是分析在某一位置标识符的有效性。

可见性在分析两个同名标识符作用域嵌套的特殊情况时，非常有用。在内层作用域中，外层作用域中声明的同名标识符是不可见的，当在内层作用域中引用这个标识符时，表示的是对内层作用域中声明的标识符的引用。

例如，下面的程序定义 3 个不同作用域的同名变量，在访问它们时，可见性起了作用：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch6_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int id=3;

int main()
{
   id=5;
   {
      int id;
      id=7;
      cout <<"id=" <<id <<endl;     //输出 7
   }
   cout <<"id=" <<id <<endl; //输出 5
}
```

运行结果为：

```
id=7
id=5
```

这里外层的变量 id 有文件作用域，内层的变量 id 有块作用域。内层的变量 id 隐藏了外层的变量 id，因此，在主函数 main()内，定义了变量 id 之后，就不能简单访问文件作用域的变量 id 了。

在最内层的块作用域结束后，次外层的块作用域又变得可见了，所以运行结果的第二行

输出 id 的值为 5。

标识符的可见性范围不超过作用域，作用域则包含可见范围。

例如，下面的代码进一步说明可见性与作用域的关系：

```
{
   int i;
   char ch;
   i = 3;
   {
    double i;
    i = 3.0e3;                    //int i 被隐藏
    ch = 'A';                     //char ch 仍可见
   }                              //double i 的作用域结束
   i + = 1;                       //int i 可见
}                                 //int i,char ch 作用域结束
```

该代码的图示见图 6-1。

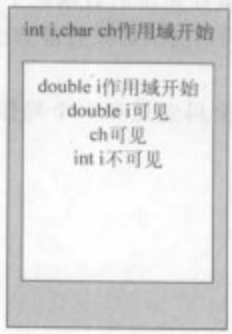

图 6-1　作用域与可见性

图中阴影部分为 int i 和 char ch 的作用域。进入内部块后，double i 遮住了 int i，使得 int i 不可见，但变量 ch 仍可见。所以在内部块中，double i 的作用域和可见性是一致的，int i 作用域存在，但不可见。在外部块中，ch 的作用域与可见性是一致的，因为 ch 的可见性渗透至内部块中。

如果被隐藏的是全局变量，则可用符号::来引用该全局变量。相关的内容可参见 11.3 节。

例如，下面的代码定义了同名的全局变量和局部变量，但仍可在局部作用域中访问全局变量：

```
int s = 0;

void f()
{
   float s = 3.0;
   int a;
   {
    float a = 2.0;
    ::a = 1;                      //error 没有全局变量 a
```

```
        ::s = 1;                    //ok 全局变量 s
        s = 2.0;                    //ok 指 float s
    }
}
```

6.5 生命期

生命期也叫生存期。生命期与存储区域密切相关，存储区域主要有代码区(code area)、数据区(data area)、栈区(stack area)和堆区(heap area)，对应的生命期为静态生命期、局部生命期和动态生命期。

1. 静态生命期

这种生命期与程序的运行期相同，只要程序一开始运行，这种生命期的变量就存在，当程序结束时，其生命期就结束。变量在固定的数据区中分配空间的，具有静态生命期。所以，全局变量、静态全局变量、静态局部变量都具有静态生命期。**具有文件作用域的变量具有静态生命期。**

例如，下面的代码声明了一个全局变量和一个局部变量，由于生命期不同，其命运也不同：

```
#include <iostream.h>

int n;

int main()
{
    double m = 3.8;
    cout <<"n = " <<n <<endl;      //ok
    fn();
}

void fn()
{
    cout <<"m = " <<m <<endl;      //error
    cout <<"n = " <<n <<endl;      //ok
}
```

因为 n 是全局变量，所以任何函数都能够访问它。而 m 是局部变量，尽管它在调用函数 fn()之前有了定义，但 fn()仍无法访问它，原因是 m 不具静态生命期，在 fn()中不存在。**被调用函数不能享有使用调用函数中数据的权利。**

静态生命期的变量，若无显式初始化，则自动初始化为 0。

函数驻在代码区，也具有静态生命期。在函数内部可以声明静态生命期的变量，即静态局部变量。

2. 局部生命期

在函数内部声明的变量或者是块中声明的变量具有局部生命期。这种变量的生命期开

始于程序执行经过其声明点时，而结束于其作用域结束处。所以具有局部生命期的变量也具有局部作用域。但反之不然，具有局部作用域的变量若为局部变量，则具有局部生命期，若为静态局部变量，则具有静态生命期。

具有局部生命期的变量驻在内存的栈区。

具有局部生命期的变量如果未被初始化，则内容不可知。

3. 动态生命期

这种生命期由程序中特定的函数调用(malloc()和 free())或操作符(new 和 delete)来创建和释放。见 8.4 节。

具有这种生命期的变量驻在内存的堆中。当用函数 malloc()或 new 为变量分配空间时，生命期开始，当用 free()或 delete 释放该变量的空间或程序结束时，生命期结束。

6.6 头文件

一个程序经常由多个源文件组成，每个源文件是一个可编译的程序单位。在将程序分解成多个源文件之后，必须计划在每个源文件中哪些信息可以被其他文件可见，哪些不可见。C++提供了开放和隐藏信息的工具，那就是使变量或函数具有外部或静态存储类型或都不具有。

在程序中可能有：

```
全局变量声明，如 extern int n;
全局变量定义，如 int n;
静态全局变量定义，如 static int n;
静态函数声明，如 static void fn();
函数声明，如 void fn();
函数定义，如 void fn(){ //... }
类型声明，如 enum COLOR{ //... };
全局常量声明，如 extern const float pi;
全局常量定义，如 const float pi = 3.14;
内联函数定义，如 inline void fn();
以及非外部或静态存储类型名字的声明及定义.
```

同一名字的声明可以多次，具有外部存储类型的声明可以在多个源文件中引用，因此方便的方法是将它们放在头文件中。头文件起着源文件之间接口的作用。

下述经验规则说明哪些可以，哪些不可以放在头文件中。不是语言要这么做，而是对#include机制使用方法的一个合理建议。

头文件一般可包含：

```
类型声明，如 enum COLOR{ //... }
函数声明，如 extern int fn(char s);
内联函数定义，如 inline char fn(char p){ return *p++; }
常量定义，如 const float pi = 3.14;
数据声明，如 extern int m; extern int a[];
枚举，如 enum BOOLEAN{ false, true};
包含指令(可嵌套)，如 #include <iostream.h>
宏定义，如 #define Case break;case
```

注释，如//check for end of file

但头文件不宜于包含：

一般函数定义，如 char fn(char p){return * p++;}
数据定义，如 int a; int b[5];
常量聚集定义，如 const int c[] = {1,2,3};

例如，下面的程序由一个工程文件 ch6_5.prj 定义。该工程文件由三个源文件组成：

```
//- - - - - - - - - - - - - - - - -
//- -        ch6_5.prj          - -
//- - - - - - - - - - - - - - - - -

ch6_5.cpp
mycircle.cpp
myrect.cpp
```

该三个源文件中，都包含了自己定义的头文件 myarea.h。三个源文件分别为：调用不同功能计算面积，定义计算圆面积和定义计算矩形面积的函数：

```
//* * * * * * * * * * * * * * * * * * * * *
//* *              myarea.h                * *
//* * * * * * * * * * * * * * * * * * * * *

double circle(double radius);
double rect(double width, double length);


//* * * * * * * * * * * * * * * * * * * * * * * * * *
//* *              mycircle.cpp                    * *
//* *               计算圆面积                     * *
//* * * * * * * * * * * * * * * * * * * * * * * * * *

# include "myarea.h"

const float pi = 3.14;

double circle(double radius)
{
   return pi * radius * radius;
}

//* * * * * * * * * * * * * * * * * * * * * * *
//* *              myrect.cpp                 * *
//* *              计算矩形面积               * *
//* * * * * * * * * * * * * * * * * * * * * * *

# include "myarea.h"

double rect(double width,double length)
{
   return width * length;
}
```

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                ch6_5.cpp                * *
//* *                 主函数                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include "myarea.h"

  int main()
{
   double width,length;
   cout <<"please enter two numbers:\n";
   cin >>width >>length;

   cout <<"area of rectangle is " <<rect(width,length) <<endl;
   double radius;
   cout <<"please enter a radius:\n";
   cin >>radius;

   cout <<"area of circle is " <<circle(radius) <<endl;
}
```

运行结果为：

```
c:\>ch6_5
please enter two numbers:
5.5  2.0
area of rectangle is 11
please enter a radius:
2.1
area of circle is 13.8474
```

要开发该程序，首先分别编辑头文件 myarea.h 和其他 C++源文件。

由于 ch6_4.cpp 中用到了流输出，所以必须包含 C++I/O 流的头文件 iostream.h。

对工程文件进行编译和连接，产生一个名为 ch6_5.exe 的执行程序，运行之，就会得到应有的结果。

工程文件中的 3 个文件都含有 myarea.h 的头文件，这样可以使信息共用，保证程序的一致，在大型程序开发中尤为必要。

6.7 多文件结构

程序开发的示意图见图 6-2。

图中，源文件中含有包含头文件的预编译语句，经过预编译后，产生翻译单元，该翻译单元以临时文件的形式存放在计算机中。之后编译，进行语法检查，产生目标文件(.obj)。若干个目标文件经过连接，产生可执行文件(.exe)。连接包括 C++库函数的连接和标准类库的连接。

许多小程序可以由单个源文件建立，它编译成一个目标文件(.obj)，然后输给连接器，

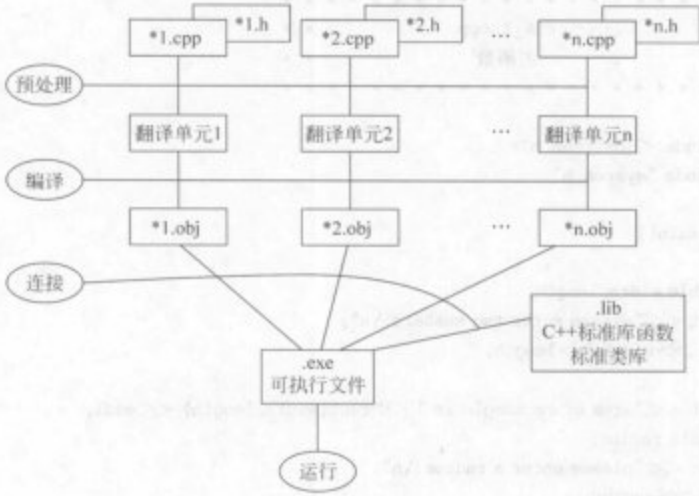

图 6-2　C++程序开发示意图

产生运行程序。这样的程序维护方便。如果修改了源文件中的任何函数，只需再次启动编译器。

大程序倾向于分成多个源文件，其理由为：

(1) 避免一而再、再而三地重复编译函数。因为编译器总是以文件为单位工作的。如果一个文件中包含的函数太多，则由于被修改的函数总是少数的几个，所以大多数正确的函数都得重新编译一次。

(2) 使程序更加容易管理。可以将程序按逻辑功能划分，分解成各个源文件，便于程序员的任务安排，以及程序调试。

(3) 把相关函数放到一特定源文件中。例如，所有输入函数放在一个源文件中。

6.8　编译预处理

预处理程序也称预处理器，它包含在编译器中。预处理程序首先读源文件。预处理的输出是“翻译单元”，它是存放在内存中的临时文件。编译器接受预处理的输出，并把源代码转化成包含机器语言指令的目标文件。

预处理程序对源文件进行第一次处理，它处理的是预处理指令。我们介绍三类预处理指令：#include，#define 和 #if。

1. #include 包含指令

include 命令让预处理器把一个源文件嵌入到当前源文件中该点处。它有两种格式，一种格式是：

```
#include <文件名>
```

这种格式用于嵌入 C++ 提供的头文件。这些头文件一般存于 C++ 系统目录中的 include 子目录下。C++ 预处理器遇到这条指令后，就到 include 子目录下搜索给出的文件，并把它嵌入到当前文件中。这种方式是标准方式。

另一种格式是：

```
#include "文件名"
```

预处理器遇到这种格式的包含指令后，首先在当前文件所在目录中进行搜索，如果找不到，再按标准方式进行搜索。这种方式适合于规定用户自己建立的头文件。

include 文件可以嵌套，即在头文件中，还可以有包含指令。

2. #define 宏定义指令

在 C 中，#define 最常用的方法是建立常量，但已经被 C++ 的 const 定义语句所代替。见 2.5 节。

#define 还可以定义带参数的宏，但也已经被 C++ 的 inline 内嵌函数所代替。见 5.7 节。

#define 的一个有效的使用是在条件编译指令中。

3. 条件编译指令

条件编译的指令有：#if，#else，#elif，#endif，#ifdef，#ifndef 和 #undef。

条件编译的一个有效使用是协调多个头文件。

例如，符号 NULL 在 6 个不同的头文件中都有定义：locate. h，stddef. h，stdio. h，stdlib. h，string. h 和 time. h。一个源文件可能包含其中的几个头文件，这样会使得编译给出“一个符号重复定义多次”的错误。这时，需要在每个头文件中使用条件编译指令：

```
#ifndef NULL
   #define NULL ((void*)0)
#endif
```

上面的代码能够保证符号 NULL 在一个程序中只有一次定义((void*)0)。而当再次遇到头文件时，一切定义的企图都被 #ifndef 给“挡驾”了。

使用 #undef 可以取消符号定义，这样可以根据需要打开和关闭符号。

小结

存储类型决定了名字在内存中的位置，存储类型也规定了在多文件程序中的连接特性。

非静态的全局名字具有外部存储类型的属性，它使得程序中的诸文件之间共享该名字，前提是在程序的各文件中当且仅当只有一个名字定义而其他皆为声明。

静态就是让变量和函数在声明的区域内成为私有。这使得多文件协作编程的数据交错使用状态下可靠性增强。

作用域规则规定了程序中名字的有效范围，它给名字的可见性提供了依据。所有的变量都有作用域和可见性。

为使在不同源文件中保持声明的一致性，采用头文件。

预处理器对源文件进行初次处理，处理时，它忽略注释语句，加入.h 头文件，并按宏定义进行替换。

在面向对象程序设计中，程序结构的意义扩展到了类。类作用域，名空间，类及对象的连接特性，以及程序的合理分解都是新的内容，这些内容将在 11.7 节中继续介绍。

练习

6.1 指出下列程序的错误。

(1)

```
//file1.cpp

int x = 1;
int func()
{
   //...
}

//file2.cpp

extern int x;
int func();
void g()
{
   x = func();
}
//file3.cpp

extern int x = 2;
int g();

void main()
{
   x = g();
   //...
}
```

(2)

```
//file1.cpp

int x = 5;
int y = 8;

extern int z;

//file2.cpp

int x;
```

```
extern double y;
extern int z;
```

6.2 写出下列程序的运行结果。

```
//file1.cpp

static int i = 20;
int x;

void f(int v)
{
   x = g(v);
}

static int g(int p)
{
   return i + p;
}
```

```
//file2.cpp

#include <iostream.h>

extern int x;
void f(int);

int main()
{
   int i = 5;
   f(i);
   cout <<x;
}
```

6.3 将习题 4.9 分解为四个源文件实现。一个文件含有主函数，调用其他三个函数。其他三个文件分别含有一个乘法九九表输出格式的函数定义。要求用一个头文件作为相互联络的接口。

第7章 数 组

在这之前，所介绍的数据都是基本数据类型（整型，字符型，浮点型）。人们经常需要使用大量集中在一起的数据来工作，C++支持数组处理来满足这一需求。数组可以是一维的，也可以是多维的，许多重要的应用都是基于数组的。学习本章后，要求理解数组下标，掌握初始化数组的方法，学会把数组用作函数参数，学会二维数组的使用，并学习数组应用的技术。

7.1 数组定义

数组是一个由若干同类型变量组成的集合。一维数组的说明方法为数据类型加数组名，再加方括号，里面含有元素个数。即：

```
类型说明符 数组名[常量表达式];
```

例如，下面的代码说明一个字符数组：

```
int main()
{
   char buffer[5];
   //...
}
```

主函数中定义了一个字符数组，存储该数组占5个字节。这个字符数组可以是最长为4个字符的单词，因为数组第5个字节用于'\0'字符，用'\0'字符结束的字符数组构成一个字符串。

数组名的命名规则和变量名是一样的。数组名后是用方括号括起来的常量表达式，不能用圆括号。例如，下面的用法不对：

```
int a(5);                    //并非数组，而是初值为5的变量a定义
```

常量表达式表示元素的个数，即数组长度。“char a[5]”表示a数组有5个元素，每个元素的类型是字符型。数组下标从0开始，分别是a[0]，a[1]，a[2]，a[3]，a[4]。注意，a[5]不属于该数组的空间范围。数组的内存排列见图7-1。

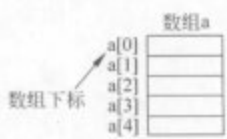

图7-1 数组的内存排列

下标是数组元素到数组开始的偏移量。第1个元素的偏移量是0，第2个元素的偏移量是1，依次类推。由此，数组是一系列大小相同的连续项，每项到公共基点的偏移量是固

定的。

数组定义的方括号中，常量表达式可以包含枚举常量和字符常量，该常量表达式的值是在编译时确定的。一个数组定义是具有确定含义的操作，它分配固定大小的空间。如果方括号的值不能在编译时确定，那就只能在运行时确定，即在函数调用时，即兴分配数组空间。这使得为局部作用域的数组分配数据空间的语句，具有不同的意义，它随每次函数调用的不同而不同，这是不允许的。

C++允许堆内存分配来建立数组，8.4 节介绍了这种方法。

数组的作用域规则和单个变量相同。定义为局部作用域的数组，刚分配完空间时，其内容是不定的。全局作用域数组和静态局部作用域数组初始为全 0。例如，下面的代码用 3 种方法定义数组：

```
int iArray[10];     //全局数组
void funcA();
void funcB();
int main()
{
   funcA();
   funcB();
}

void funcA()
{
   static int iStaticLocal[30];     //局部静态数组
   //...
}

void funcB()
{
   int iLocal[20];     //局部数组
   //...
}
```

数组 iArray 在所有函数外面定义，它是全局的，因此，可以被任何函数访问。这个数组没有初始化，所以数组的每个元素都初始为 0。在 16 位机器上，这个数组中每个元素占 2 个字节，所以共占 20 个字节内存空间。

函数 funcA()定义了一个静态局部数组，它有 30 个整型数，这个数组不在栈中，而是在全局数据区。它没有显式初始化，所以默认初始值为 0，与全局数组相同。但在函数 funcB()中，iStaticLocal 是不可见的。因为是静态的，数组在 funcA()调用之间保持其值。如果在第一次调用 funcA()时改变了 iStaticLocal 的值，则第 2 次调用 funcA()时，funcA()拥有这些改变了的值。详见 5.5 节。

函数 funcB()中定义了在栈中分配的数组，它有 20 个整型数。由于该数组在栈中，它受到了栈空间大小的限制。如果定义数组的元素很多(如 500000)，则有可能使程序运行由于不能满足数组分配而突然终止。**不能满足内存分配要到程序运行时才知道**，因为**编译只管语法检查**，而不管运行环境。程序连接时，才确定各内存空间包括栈空间的大小。确定栈空间大小后，程序运行中遇到大容量数组分配而不能满足之时，才会有所表示。

编程时，如果要定义一个很大的数组，可以通过将其定义为静态或全局来解决，也可以

将其在堆内存中分配(见 8.4 节)。

在编译时,数组定义中的下标必须确定。例如,下面的代码不能通过编译:

```
int main()
{
   int size = 50;
   int array[size];     //error: 不能用变量来描述数组定义中的元素个数
//...
}
```

尽管变量 size 已经赋有值 50,紧接着就是数组的定义,阅读上都能理解,但是 size 是变量这一性质,是编译不能原谅的。对于常量,编译可以用一个值直接代替,但对于变量,编译不能用值来代替,而是编译为取该变量的值。**取值是一个操作,不是值本身**,不能决定数组下标。程序运行中,通常通过常量来决定数组大小。

用全局变量的值来确定数组下标也是不允许的。例如:

```
int size = 50;

int main()
{
   int array[size];     //error
   //...
}
```

任何函数都在程序运行中被调用,函数被调用的先后次序是未知的。在函数被调用时,全局变量的值也是不可预测的,所以,数组的下标无法确定。

例如,下面的代码用常量来规定数组元素个数:

```
const int size = 50;
const int n = size * sizeof(int);

int main()
{
   int array[size];     //ok
   char charray[n];     //ok
   //...
}
```

size 和 n 都是常量,n 的常量定义中,初始化值是个表达式,但在编译时,该表达式的值能被确定下来。因此,编译总是认可数组定义中用常量说明的元素个数。

7.2 访问数组元素

数组中特定的元素通过下标访问。在 C++中,所有数组均由连续的存储单元组成,起始地址对应于数组的第一个元素,下标是距数组开始的偏移量。长度为 n 的数组,其下标范围为 0～(n－1)。

在内存的表示中,地址是从 0 开始的。如果表示数组的下标从 1 开始,则需要进行额外的机器操作,C 或 C++的处理方法使其编译器更简单有效,使代码效率更高。数组直接从 0 下标开始则不必进行这种多余的调整。

在数组定义后，给数组赋初值时，必须一个个元素逐个访问。例如，下面的代码给一个数组赋一组 Fibonacci 数：

```
void main()
{
   int iArray[10];     //数组定义

   iArray[0] = 1;
   iArray[1] = 1;
   iArray[2] = 2;
   iArray[3] = 3;
   //...
   iArray[9] = 55;
   //...
}
```

如果知道元素之间的规律，上面的赋值也可以通过循环来完成：

```
int main()
{
   int iArray[10];
   iArray[0] = 1;
   iArray[1] = 1;
   for(int i = 2; i<10; i ++ )
     iArray[i] = iArray[i - 1] + iArray[i - 2];
   //...
}
```

for 循环的终止条件为 i<10。当 i==10 时，它终止循环。必须保证没有超出数组边界。如果循环条件误写成“i<=10”，那么程序将会执行到包括“iArray[10]=90;”的语句，改变不属于数组空间的内存单元（见图 7-2）。这个代码的失误，不会在程序的编译与连接中反映出来，而是可能一直运行下去，直到出现结果不正确，或严重时导致死机。

图 7-2　数组越界的内存表示

字符数组，实际上是 1 字节的整数数组。处理字符数组的方法与处理其他数组相同。字符数组若用来存储字符串，则要考虑字符串末尾的'\0'结束符。

例如，下面的程序输出一个字符串：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch7_1.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *
```

```
#include <iostream.h>

int main()
{
   char chArray[30];
   cin.get(chArray,30);
   for(int i = 0; chArray[i]!= '\0'; i++)
     cout <<chArray[i];
   cout <<endl;
}
```

例中 get()是输入流的成员函数(我们暂且模仿之,到面向对象程序设计中,再去讨论其实现机制)。它使用时,前面必须加"cin."。它输入一系列字符,直到输入流中出现结束符或所读字符个数已达到要求读的字符个数。它的原型为:

```
get(char * target,int count,char delimeter = '\n');
```

其中 target 为存放一系列字符的空间地址,count 为限制最长的读取字符个数,delimeter 为规定的结束符。遇到此结束符时,尽管还没有到达读取字符最大个数,也还是结束读入过程。

这里 for 循环保持下去的条件为:

```
chArray[i]!= '\0';
```

实际上没有必要让 i 达到 30。用户也许不会键入这么多字符。cin.get()在用户输入的最后字符后面加上'\0'。人们对'\0'之后的无定义字符没有兴趣。

可以用"chArray[i];"来代替前面的终止条件,也即 for 循环可以写成:

```
for(int i<0; chArray[i]; i++)
```

因为字符串中的所有字符,除最后的'\0'外,都是非 0(真值)值,所以,较短的格式是可行的。这两种表示在应用中都很普遍。

字符串的处理在 8.7 节中将进一步展开讨论,读者将会看到字符串的输出无须逐个字符输出,可以一种更简捷的方式进行。

7.3 初始化数组

1. 数组的初始化

数组可以初始化,即在定义时,使它包含程序马上能使用的值。

例如,下面的代码定义了一个全局数组,并用一组 Fibonacci 数初始化:

```
int iArray[10] = {1,1,2,3,5,8,13,21,34,55};    //初始化

int main()
{
  //...
}
```

初始化数组的值的个数不能多于数组元素个数,初始化数组的值也不能通过跳过逗号

的方式来省略，这在C中是允许的，但在C++中不允许。

例如，下面的代码对数组进行初始化是错误的：

```
int array1[5] = {1,2,3,4,5,6};      //error:初始化值个数多于数组元素个数
int array2[5] = {1,,2,3,4};         //error: 初始化值不能省略
int array3[5] = {1,2,3,};           //error: 初始化值不能省略
int array4[5] = {};                 //error: 语法格式错误

void main()
{
  //...
}
```

初始化值的个数可少于数组元素个数。当初始化值的个数少于数组元素个数时，前面的按序初始化相应值，后面的初始化为0。

例如，下面的程序对数组进行初始化：

```
//* * * * * * * * * * * * * * * * * * * * * *
//* *              ch7_2.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int array1[5] = {1,2,3};
static int array2[5] = {1};

int main()
{
  int arr1[5] = {2};
  static int arr2[5] = {1,2};

  int n;
  cout <<"global:\n";
  for(n = 0; n<5; n++)
       cout <<"  " <<array1[n];

  cout <<"\nglobal static:\n";
  for(n = 0; n<5; n++)
       cout <<" " <<array2[n];

  cout <<"\nlocal:\n";
  for(n = 0; n<5; n++)
       cout <<" " <<arr1[n];
  cout <<"\nlocal static:\n";
  for(n = 0; n<5; n++)
       cout <<" " <<arr2[n];
  cout <<endl;
}
```

运行结果为：

```
global:
  1  2  3  0  0
global static:
```

```
  1  0  0  0  0
local:
  2  0  0  0  0
local static:
  1  2  0  0  0
```

例中，全局数组和全局静态数组的初始化是在主函数运行之前完成的，而局部数组和局部静态数组的初始化是在进入主函数后完成的。

全局数组 array1[5]对于初始化表的值按序初始化为 1,2,3，还有两个元素的值则按默认初始化为 0。

全局静态数组 array2[5]与全局数组的初始化情况一样，初始化表值{1}表示第 1 个元素的值，而不是指全部数组元素都为 1。

局部数组 arr1[5]根据初始化表值的内容按序初始化，初始化表值虽只有 1 个，但因启动了初始化，使得其余元素都被默认初始化为 0 了。

局部静态数组 arr2[5]先根据初始化表按序初始化，其余 3 个数组元素的值默认初始化为 0。

2. 初始化字符数组

初始化字符数组有两种方法，一种是：

```
char array[10] = {"hello"};
```

另一种是：

```
char array[10] = {'h','e','l','l','o','\0'};
```

第一种方法用途较广，初始化时，系统自动在数组没有填值的位置用'\0'补上。另外，这种方法中的花括号可以省略，即能表示成：

```
char array[10] = "hello";
```

第二种方法一次一个元素地初始化数组，如同初始化整型数组。这种方法通常用于输入不容易在键盘上生成的那些不可见字符。

例如，下面的代码中初始化值为若干制表符：

```
char chArray[5] = {'\t','\t','\t','\t','\0'};
```

这里不要忘记为最后的'\0'分配空间。如果要初始化一个字符串"hello"，那为它定义的数组至少有 6 个数组元素。

例如，下面的代码给数组初始化，但会引起不可预料的错误：

```
char array[5] = "hello";
```

该代码不会引起编译错误，但由于改写了数组空间以外的内存单元，所以是危险的。

3. 省略数组大小

有初始化的数组定义可以省略方括号中的数组大小。

例如，下面的代码中数组定义为 5 个元素：

```
int a[] = {2,4,6,8,10};
```

编译时必须知道数组的大小。通常，声明数组时方括号内的数字决定了数组的大小。有初始化的数组定义又省略方括号中的数组大小时，编译器统计花括号之间的元素个数，以求出数组的大小。

例如，下面的代码产生相同的数组空间：

```
static int a1[5] = {1,2,3,4,5};
static int a2[] = {1,2,3,4,5};
```

让编译器得出初始化数组的大小有几个好处。它常常用于初始化一个元素个数在初始化中确定的数组，提供程序员修改元素个数的机会。

在没有规定数组大小的情况下，怎么知道数组的大小呢？sizeof 操作解决了该问题。

例如，下面的代码用 sizeof 确定数组的大小：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch7_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   static int a[] = {1,2,4,8,16};
   for(int i = 0; i<(sizeof(a)/sizeof(int)); i++)
   cout <<a[i] <<"  ";
   cout <<endl;
}
```

运行结果为：

```
1  2  4  8  16
```

sizeof 操作使 for 循环自动调整次数。如果要从初始化 a 数组的集合中增删元素，只需重新编译即可，其他内容无须更动。

每个数组所占的存储量都可以用 sizeof 操作来确定。sizeof 返回指定项的字节数。sizeof 常用于数组，使代码可在 16 位机器和 32 位机器之间移植。

对于字符串的初始化，要注意数组实际分配的空间大小是字符串中字符个数加上末尾的'\0'结束符。

例如，下面的代码定义一个字符数组：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch7_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   char ch[] = "how are you";
```

```
    cout <<"size of array: " <<sizeof(ch) <<endl;
    cout <<"size of string: " <<strlen("how are you") <<endl;
}
```

运行结果为：

```
size of array: 12
size of string: 11
```

例中，数组大小为 12，而字符串长度为 11。

省略数组大小只能在有初始化的数组定义中。

例如，下面的代码将产生一个编译错误：

```
int a[]; //error: 没有确定数组大小
```

在定义数组的场合，无论如何，编译器必须知道数组的大小。

7.4 向函数传递数组

无论何时，将数组作为参数传给函数，实际上只是把数组的地址传给函数。

物理上，把整个数组放在栈中是不合理的，因为栈大小是一定且有限的。如果把传送给函数的整个数组都放在栈中（内存的大块复制），则很快会把栈空间用光。

1. 传递给标准库函数

C++中有一个 memset()的函数，它可以一字节一字节地把整个数组设置为一个指定的值。memset()函数在 mem.h 头文件中声明，它把数组的起始地址作为其第一个参数，第二个参数是设置数组每个字节的值，第三个参数是数组的长度（字节数，不是元素个数）。其函数原型为：

```
void* memset(void*, int, unsigned);
```

其中 void* 表示地址，详细介绍见 8.6 节。

例如，下面的代码用数组做参数传递给标准函数 memset()，以让其将数组设置成全 0：

```
#include <mem.h>

int main()
{
   int ia1[50];
   int ia2[500];
   memset(ia1, 0, 50 * sizeof(int));
   memset(ia2, 0, 500 * sizeof(int));
   //...
}
```

memset()的第一个实参是数组名，**数组名作参数即数组作参数，它仅仅只是一个数组的起始地址而已。**

实现第一个 memset()函数调用的内存布局见图 7-3。在函数 memset()栈区，从返回地址往上依次为第 1，2，3 个参数。第 1 个参数中的内容是 main()函数中定义的数组 ia1 的起

始地址。第 2 个参数是给数组设置的值(0),第 3 个参数是数组的长度(50 * 2)。函数返回时,main()函数的数组中内容全置为 0。

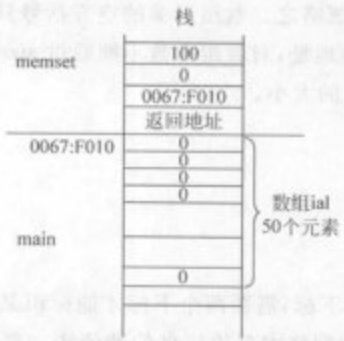

图 7-3 memset 函数调用的内存布局

2. 传递给自定义函数

若要让一个函数求数组元素的和,需传递一个数组参数和数组大小参数。因为从传递的数组参数(地址)中,没有数组大小的信息。

例如,下面的程序调用一个函数求数组元素之和:

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                ch7_5.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int sum(int [], int);

int main()
{
   static int ia[5] = {2,3,6,8,10};
   int sumOfArray;
   sumOfArray = sum(ia, 5);
   cout <<"sum of array: " <<sumOfArray <<endl;
  }

int sum(int array[], int len)
{
   int iSum = 0;
   for(int i = 0; i<len; i++)
     iSum + = array[i];
   return iSum;
}
```

运行结果为:

```
sum of array: 29
```

sum()函数以整数数组作为第一个参数，以整数作为第二个参数。由于传递数组实际上传递的是地址，所以函数原型中，数组参数的书写形式无须在方括号中写明数组大小。如果写明了数组大小，编译器将忽略之。数组形参的空方括号只是告诉函数，该参数是个数组的起始地址。由于数组参数是地址，对数组参数不能通过 sizeof 求得数组大小，所以 sum()函数必须要第二个参数：数组的大小。

7.5 二维数组

1. 二维数组定义

C++中的数组可以有多个下标，需要两个下标才能标识某个元素的数组称为二维数组。二维数组经常用来表示按行和列格式存放信息的数值表。要识别表中某个特定的元素，必须指定两个下标。习惯上，第一个下标表示该元素所在行，第二个下标表示该元素所在列。

图 7-4 中表示了一个名为 a 的 3 行×4 列的整型二维数组。可以看到，第一个下标范围是 0～2，第二个下标范围是 0～3。二维数组是按先行后列的顺序在内存中线性排列的。它的定义如下：

```
int a[3][4];
```

通常把有 m 行和 n 列的数组称为 m×n 数组。

图 7-4 3×4 数组排列的内存表示

数组 a 中的每个元素用元素名 a[i][j]识别，其中，a 是数组名，i 和 j 是唯一标识数组 a 中每个元素的下标。

2. 初始化

和一维数组一样，二维数组也能在定义时被初始化。

例如，下面定义一个 2×3 的整型数组，并初始化：

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

其中的值是按行用花括号组合在一起的。{1,2,3}初始化了 b[0][0]，b[0][1]和 b[0][2]，{4,5,6}初始化了 b[1][0]，b[1][1]和 b[1][2]。如果某行没有足够的初始化值，那么该行中的剩余元素都被初始化为 0。初始化还可以将多个花括号简化为一个花括号。

例如，下面的代码初始化二维数组：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch7_6.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *
#include <iostream.h>

int main()
{
   int array1[2][3] = {1,2,3,4,5};
   int array2[2][3] = {{1,2},{4}};

   int i,j;
   for(i = 0; i<2; i ++ )        //按行列输出数组 array1
   {
      for(j = 0; j<3; j ++ )
         cout <<array1[i][j] <<",";
      cout <<endl;
   }

   cout <<endl;
   for(i = 0; i<2; i ++ )        //按行列输出数组 array2
   {
      for(j = 0; j<3; j ++ )
         cout <<array2[i][j] <<",";
      cout <<endl;
   }
}
```

运行结果为：

```
1,2,3,
4,5,0,

1,2,0,
4,0,0,
```

数组 array1 提供了 5 个初始化值，这些初始化值先赋给第一行元素，然后再赋给第二行元素。还有最后一个元素 array1[1][2]没有被明确初始化，在这里被默认初始化为 0。数组 array2 的定义语句在两个花括号中提供了 3 个初始化值。用于第一行的初始化值表把第一行的头两个元素明确地初始化为 1 和 2，第三个元素为 0。用于第二行的初始化值表把第二行的第一个元素明确地初始化为 4，其余为 0。

3. 省略第一维大小

如果对全部元素都赋初值，则定义数组时对第一维的大小可以忽略，但第二维的大小不能省。例如：

```
int a[][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

与下面的代码是等价的：

```
int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

编译器会根据数据总个数分配空间，每行 4 列，所以确定该数组为 3 行。

在定义时，也可以只对部分元素赋初值而省略第一维的大小，但应分行赋初值。例如：

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

该数组定义表示 3×4 的整型数组，没有明确初始化值的都默认初始化为 0，所以等价于下面的定义：

```
static int a[3][4] = {{1,2,3},{0,0,0},{4,5,0}};
```

访问二维数组中所有的元素通常需要一个二重循环。例如，上例输出数组 array1 和 array2 所有元素的值。

4. 作为参数传递

作为参数传递一个二维数组给函数，其意义也为内存地址，所以原型中，声明整数数组参数的形式只能省略左边的方括号。

例如，下面的程序定义一个 3×4 的数组，表示 3 个学生，每个学生有 4 次测验成绩，求所有学生中的最好成绩。问题化作遍历二维数组找最大值，传递函数参数时，除了传递数组名外，还要传递行数和列数：

```
//* * * * * * * * * * * * * * * * * * * * *
//* *              ch7_7.cpp              * *
//* * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>

int maximum(int[][4],int,int);

int main()
{
   int sg[3][4] = {{68,77,73,86},
                   {87,96,78,89},
                   {90,70,81,86}};

   cout <<"the max grade is " <<maximum(sg,3,4) <<endl;
}
int maximum(int grade[][4],int pupils,int tests)
{
   int max = 0;
   for(int i = 0; i<pupils; i++)
     for(int j = 0; j<tests; j++)
       if(grade[i][j]>max)
         max = grade[i][j];

   return max;
}
```

运行结果为：

```
the max grade is 96
```

5. 降维处理

由于二维数组在内存中是线性排列的，传递一维数组和传递二维数组都是传的地址，所以可以在被调用的函数中用单重循环来遍历二维数组中的所有元素，此时只要传递数组名和元素总个数。要注意被传递的数组地址不要用数组名表示，要用第一个元素的地址表示，因为数组名表示的是二维数组的首地址，尽管地址值相同，但操作不同，在第 8 章中将详细解释地址与指针的差别。

例如，下面的程序是上例程序的改进。主函数将第一个数组元素地址作为一维数组首地址传递给 maximum()函数(实参)，maximum()函数也以一维整型数组的首地址来接受，求取学生成绩的最大值：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch7_8.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int maximum(int[],int);

int main()
{
   int sg[3][4]={{68,77,73,86},
                 {87,96,78,89},
                 {90,70,81,86}};
   cout <<"the max grade is "
        <<maximum(&sg[0][0],3*4)     //传递第一个元素地址和元素个数
        <<endl;
}

int maximum(int grade[],int num)
{
   int max=0;
   for(int i=0; i<num; i++)
     if(grade[i]>max)
       max=grade[i];
   return max;
}
```

运行结果为：

```
the max grade is 96
```

函数调用时，**数组参数的实参为整型变量的地址，函数原型中，数组参数的形参为整型数组的首地址**，两者类型是匹配的。进一步的讨论见 8.3 节和 8.8 节。

7.6 数组应用：排序

数据排序是重要的计算应用之一。排序方法有很多，我们先介绍最简单的排序法：冒

泡排序法(bubble sort),然后再介绍最常用的插入排序法(insert sort),最后介绍最快的排序法:快速排序法(quick sort)。

1. 冒泡排序法(bubble sort)

冒泡排序法可以形象地描述为:使较小的值像空气泡一样逐渐"上浮"到数组的顶部,或者较大的值逐渐"下沉"到数组的底部。这种排序技术要排序好几轮,每一轮都要比较连续的数组元素对。如果某对数值是按升序排列的,那就保持原样。如果按降序排列,就交换它们的值。见图 7-5。

原数据	第一轮	第二轮	第三轮	第四轮
B	B	B	B	A
E	D	C	A	B
D	C	A	C	C
C	A	D	D	D
A	E	E	E	E

图 7-5 冒泡排序法

图 7-5 是将原数据序列 BEDCA 排序。第一轮结束时,E 沉底,第二轮结束时 D 下沉,第三轮结束时 C 下沉,第四轮结束时 B 下沉,从而得到从小到大的顺序排列。

例如,下面的程序把有 10 个元素的数组用冒泡排序法按升序排列:

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch7_9.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void bubble(int[],int);

int main()
{
   int array[] = {55,2,6,4,32,12,9,73,26,37};

   int len = sizeof(array)/sizeof(int);     //元素个数
   for(int i = 0; i<len; i++)      //原始顺序输出
    cout <<array[i] <<",";

   cout <<endl<<endl;

   bubble(array, len);     //调用排序函数

}

void bubble(int a[],int size)     //冒泡排序
{
   int i,temp;

   for(int pass = 1; pass<size; pass++)      //共比较 size-1 轮
```

```
    {
     for(i=0; i<size-pass; i++)      //比较一轮
        if(a[i]>a[i+1])
          {                          //一次交换
           temp=a[i];
           a[i]=a[i+1];
           a[i+1]=temp;
          }

        for(i=0; i<size; i++)        //比较一轮后就输出
            cout <<a[i] <<",";
        cout <<endl;
    }
}
```

运行结果为：

```
55,2,6,4,32,12,9,73,26,37,

2,6,4,32,12,9,55,26,37,73,
2,4,6,12,9,32,26,37,55,73,
2,4,6,9,12,26,32,37,55,73,
2,4,6,9,12,26,32,37,55,73,
2,4,6,9,12,26,32,37,55,73,
2,4,6,9,12,26,32,37,55,73,
2,4,6,9,12,26,32,37,55,73,
2,4,6,9,12,26,32,37,55,73,
```

排序过程是用嵌套的 for 循环完成的。对 10 个元素的数组，一共进行 9 轮比较(pass 从 1～9)。每轮要进行 size－pass 次比较，以决出一个最大值。

程序首先进行第一轮比较(pass＝1)，即先比较 a[0]和 a[1]，再比较 a[1]和 a[2]，然后比较 a[2]和 a[3]，……，直到比较完 a[8]和 a[9]后为止。数组有 10 个元素，但只比较了 9 次(0～8)。因为是朝一个方向二个二个连续比较的，较大的值在比较之后总是放在两者的后面。9 次比较使 10 个元素都参与了比较，并在第一轮比较完后，最大的值“下沉”在数组底部，即 a[9]。

程序然后进行第二轮比较(pass＝2)，因为最后一个元素 a[9]已经确定为最大，无须与前面的元素比较，比较只须在前面 9 个元素中进行，所以该轮只需 8 次比较(0～7)。比较完第二轮后，次大的数组元素“下沉”到 a[8]。

程序再进行第三轮比较(pass＝3)，该轮比较了 7 次，确定了 a[7]。

直到程序进行第九轮比较(pass＝9)，该轮比较只进行 1 次，以确定 a[0]和 a[1]中的大者放在 a[1]中。

从结果中看出，经过 3 轮比较，就将数组从小到大排序完毕。只有在最坏情况，即全部元素从大到小排列时，才需要全部 9 轮的排序。

冒泡排序法比较易于实现，但是不论情况好坏，都要进行所有轮的比较，运行速度较慢。

2. 插入排序法(insert sort)

插入排序法是一个简单，但相对比较高效的排序算法。

插入排序通过把数组中的元素插入到适当的位置来进行排序。步骤为：

(1) 将数组中的头两个元素按排序顺序排列。

(2) 把下一个元素(第 3 个)插入到其对应于已排序元素的排序位置。

(3) 对于数组中的每个元素重复(2)。即把第 4 个元素插入到适当位置，然后是第 5 个元素，等等。

例如，对于 WVLMABCONP 字母序列的一个插入排序见图 7-6。

原数据	第一轮	第二轮	第三轮	第四轮	第五轮	第六轮	第七轮	第八轮	第九轮
W	**V**	**L**	**L**	**A**	**A**	**A**	**A**	**A**	**A**
V	**W**	**V**	**M**	**L**	**B**	**B**	**B**	**B**	**B**
L	L	**W**	**V**	**M**	**L**	**C**	**C**	**C**	**C**
M	M	M	**W**	**V**	**M**	**L**	**L**	**L**	**L**
A	A	A	A	**W**	**V**	**M**	**M**	**M**	**M**
B	B	B	B	B	**W**	**V**	**O**	**N**	**N**
C	C	C	C	C	C	**W**	**V**	**O**	**O**
O	O	O	O	O	O	O	**W**	**V**	**P**
N	N	N	N	N	N	N	N	**W**	**V**
P	P	P	P	P	P	P	P	P	**W**

图 7-6　插入排序法

图中加黑的字体部分是已经排好序的。

第一轮，是 W 和 V 比较，按升序，则交换位置。

第二轮，L 插入到已排序的 VW 中，形成 LVW。L 先跟最大的 W 比较，顺便将 W 移到 L 的位置，再与 V 比较，将 V 移到原 W 的位置，原 V 的位置放 L。

第三轮，M 插入到已排序的 LVW 中，形成 LMVW。一边比较，一边移动位置，发现 M 比 L 大时，立即停止该轮的比较，因为 M 的位置已经找到。

第四轮，类似的过程。直到第九轮全部排序完毕。

例如，下面的程序在主函数中调用一个整数插入排序函数：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch7_10.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void isort(int * a, int size);

int main()
{
   int array[ ] = {55,2,6,4,32,12,9,73,26,37};

   int len = sizeof(array)/sizeof(int);     //元素个数
   for(int i = 0; i<size; i ++ )     //原始顺序输出
     cout <<a[i] <<",";

   cout <<endl<<endl;
```

```
    isort(array, len);     //调用排序函数
}

void isort(int a[],int size)     //插入排序
{
   int inserter,index;
   for(int i=1; i<size; i++)     //共执行 size-1 轮
   {
    inserter=a[i];
    index=i-1;
    while(index>=0 && inserter<a[index])     //寻找插入点
    {
        a[index+1]=a[index];     //后挪一个位置
        index--;
    }
    a[index+1]=inserter;     //插入

    for(int j=0; j<size; j++)     //比较一轮后就输出
    {
       cout<<a[j]<<",";
       if(j==i) //已排序与未排序的分界线
         cout<<'|';
    }
    cout<<endl;

   }
}
```

运行结果为：

```
55,2,6,4,32,12,9,73,26,37,

2,55,|6,4,32,12,9,73,26,37,
2,6,55,|4,32,12,9,73,26,37,
2,4,6,55,|32,12,9,73,26,37,
2,4,6,32,55,|12,9,73,26,37,
2,4,6,12,32,55,|9,73,26,37,
2,4,6,9,12,32,55,|73,26,37,
2,4,6,9,12,32,55,73,|26,37,
2,4,6,9,12,26,32,55,73,|37,
2,4,6,9,12,26,32,37,55,73,|
```

排序函数 isort()中的 inserter 是待插入元素，index 是当前准备与插入元素比较的元素下标。

该插入排序算法好在边比较边挪位，找到插入点的同时，挪位工作也完成。挪位是赋值操作，不是交换操作，所以工作量减轻很多。但另一方面，插入排序的每轮比较都是不可缺少的，无法进一步优化算法。

3. 快速排序法(quick sort)

快速排序法被认为是效率较高的排序算法。

快速排序算法是建立在把数组分为许多部分的思想上。其工作过程是首先选择一个分

界值，把数组分成两部分，大于等于分界值的元素集中到数组的某一部分，小于分界值的元素集中到数组的另一部分。对于分出来的两部分，又分别重复这个过程，直到整个数组被排序完毕。

例如，设给定一个字符数组 fedacb，选定 d 作为分界值，则在第一遍划分后数组将重新安排如下：

```
初 始  f e d a c b
第1遍  b c a d e f
```

在第一遍划分为 bca 和 def 两部分后，又分别对这两部分进行划分。这个过程是递归的。采用递归算法使程序相对可读。

分界值可以用两种不同的方法去确定。一种办法是随机确定；另一种办法是算出被排序部分各元素的中间值。当分界值正好等于被排序部分各元素的中间值时，排序速度最快。然而，找到这个中间值却不是一件容易事。即使在最坏情况下，即分界值完全偏向一方，取了一个极大或极小值，快速排序法的性能仍然是比较好的。

例如，下面的程序给有 10 个元素的数组用快速排序法排序，选择数组中间项的值作为分界值，虽然这样做的效果并不总是最佳，但它是既简单又快捷的办法：

```
//* * * * * * * * * * * * * * * * * * * * * *
//* *              ch7_11.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void qsort(int[],int,int);
void swap(int& a, int& b){ int t=a; a=b; b=t; }

int main()
{
   int a[]={55,2,6,4,11,12,9,73,26,37};
   int len=sizeof(a)/sizeof(int);
   for(int i=0; i<len; i++)                    //原始顺序输出
     cout<<a[i]<<",";
   cout<<"\n\n";

   qsort(a,0,len-1);                           //调用排序函数

   for(int i=0; i<len; i++)                    //排序结果输出
     cout<<a[i]<<",";
   cout<<"\n";
}
void qsort(int a[],int left,int right)         //快速排序法
{
   int pivot=a[right], l=left, r=right;
   while(l<r)
   {
     swap(a[l], a[r]);
     while(l<r && a[r]>pivot) --r;
```

```
        while(l<r && a[l]<=pivot) ++l;
    }
    swap(a[left], a[r]);                    //使得a[r]成为分界元
    if(left<r-1) qsort(a,left,r-1);
    if(r+1<right) qsort(a,r+1,right);
}
```

运行结果为：

```
55,2,6,4,32,12,9,73,26,37
2,4,6,9,12,26,32,37,55,73
```

快速排序法采用3个参数，一个为数组，另两个为数组的上下界。因为递归调用也是这样的形式，所以简化代码的书写。3个参数的形式并不是必需的，因为数组参数传递的是地址，所以对于2个参数的函数原型：

```
qsort(int a[],int len);
```

其中的left为0，right为len－1，递归调用改成如下即可：

```
if(left<r) qsort(&a[left],l-left);
if(r<right) qsort(&a[r+1],right-r);
```

语句中“&”表示取该变量的地址，进一步的介绍见8.1节。

根据快速排序法，我们来分析上面字符数组的第一遍划分结果：

划分之前的准备，left=0，right=5，分界值pivot=a[(left+right)/2]= a[2]='d'，并且l=0，r=5。

每次交换的结果如下：

```
f e d a c b      //划分之前
b e d a c f      //第一次交换结果
b c d a e f      //第二次交换结果
b c a d e f      //第三次交换结果
```

首先，由于a[l]='f'＞ 'd'=pivot，所以“while(a[l]<pivot) ++l;”循环语句结束，l=0；

又由于a[r]='b'＜ 'd'=pivot，所以“while(a[r]>pivot) --r;”循环语句结束，r=5。

此时，a[0]与a[5]即'f'与'b'交换，得到第一次交换结果。

随即，l=1，r=4。

其次，由于a[l]='e'＞ 'd'=pivot，所以“while(a[l]<pivot) ++l;”循环语句结束，l=1；

由于a[r]='c'＜ 'd'=pivot，所以“while(a[r]>pivot) --r;”循环语句结束，r=4。

此时，a[1]与a[4]即'e'与'c'交换，得到第二次交换结果。

随即，l=2，r=3。

再次，由于a[l]='d' = pivot，所以“while(a[l]<pivot) ++l;”循环语句结束，l=2；

由于a[r]='a'＜ 'd'=pivot，所以“while(a[r]>pivot) --r;”循环结束，r=3。此时，a[2]与a[3]即'd'与'a'交换，得到第三次交换结果。

随即，l=3，r=2。这时，l>r，不满足外循环条件，本次划分到此结束。

读者可以分析程序 ch7_10.cpp 中的运行结果是经过几次划分（每次划分即一次递归调用）得到的。

快速排序法在一般情况下较其他排序算法为快，但是程序的理解稍微有点复杂。

7.7 数组应用：Josephus 问题

Josephus 问题是说，一群小孩围成一圈，任意假定一个数 m，从第一个小孩起，顺时针方向数，每数到第 m 个小孩时，该小孩便离开。小孩不断离开，圈子不断缩小。最后，剩下的一个小孩便是胜利者。究竟胜利者是第几个小孩呢？

解答这个问题，首先要定义一个数组，其元素个数就是小孩个数。必须预先设置一个小孩个数常量，以便定义一个数组。

对每个小孩赋以一个序号值作为小孩的标志。由于数组是局部作用域的，一旦分配之后，就删不去，得等到作用域结束才会自动抹去，所以当小孩离开时，只能修改数组元素值来标识小孩的离开。

数组是线性排列的，小孩是围成圈的，用数组表示小孩围成圈，要有一种从数组尾部跳到其头部的技巧，这就是“加 1 求模”。当数到数组尾的时候，下一个数组下标值可以算得为 0，从而回到数组首以继续整个过程。试看解答问题的程序：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *        Josephus 问题解法 1                * *
//* *        josel.cpp                          * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
  //建立小孩数组
  const int num = 10;     //小孩数
  int interval;     //每次数 interval 个小孩，便让该小孩离开
  int a[num];     //小孩数组

  //给小孩编号
  for(int i = 0; i<num; i++)     //小孩的编号只与小孩数有关
    a[i] = i + 1;

  //输入数小孩间隔
  cout <<"please input the interval: ";     //输入一个数小孩个数
  cin >>interval;

  //将全体参加的小孩输出
  for(int i = 0; i<num; i++)     //顺序输出开始时的小孩编号
    cout <<a[i] <<",";
  cout <<endl;
```

```
    int k = 1;          //标识处理第 k 个离开的小孩
    int i = -1;         //数组下标(下一个值 0 就是第一个小孩的下标)

    //处理获胜前的小孩
    while(1)
    {
      //在圈中数 interval 个小孩
      for(int j = 0; j<interval; )
      {
        i = (i + 1) % num;          //对下标加 1 求模
        if(a[i]!= 0)                //如果该元素的小孩在圈中,则承认数数有效
          j++;
      }

      if(k == num) break;           //该小孩是最后一个(胜利者)吗?

      cout <<a[i] <<",";    //输出离开的小孩之编号
      a[i] = 0;             //标识该小孩已离开

      k++;                  //准备处理下一个圈中小孩
    }

    //break 语句跳转到此
    cout <<"\nNo." <<a[i] <<" boy've won.\n";      //输出胜利者
}
```

运行结果为:

```
c:\>josel
please input the interval: 8
1,2,3,4,5,6,7,8,9,10
8,6,5,7,10,3,2,9,4
No.1 boy've won.
c:\>josel
please input the interval: 2
1,2,3,4,5,6,7,8,9,10
2,4,6,8,10,3,7,1,9
No.5 boy've won.
```

程序运行了两遍。第一遍,m 取值为 8,得到胜利者是第 1 个小孩;第二遍,m 取值为 2,得到胜利者是第 5 个小孩。

程序中,小孩数 num 用常量定义,这样数组定义的大小就可用此常量表示。用一个循环给小孩编号,依次为 1,2,3,…,不管小孩有几个,小孩的编号只与小孩个数有关。

随机输入的 m 值应大于 0,一般不能超过小孩数。读者可思考如何进行控制?

在处理离开小孩的循环前,初始化正要处理第 1 个小孩给 k,初始化数组的下标为-1,因为下一个值 0 下标表示数组第一个元素,即起始第一个小孩。

在 while 循环中的 for 循环完成数 m 个小孩的工作。数组中含有离开的和未离开的小孩，标识为 0 的是离开的小孩，否则，数组元素的值是小孩的编号。因此，往前数一下，须确认该小孩含有非 0 值。另外，下标的移动是重要的，i 值加 1 是下一个下标，但该下标有可能越过数组的边界，所以对其进行模运算就能保证下标在数组范围内循环移动。

每次处理小孩离开时，都要遍历整个数组，所以该程序效率是不高的。用数组来表示小孩围成的圈，数据结构的描述是简单的，程序语句的行数也不多，但处理是富于技巧的，理解比较费事。原始的 C 程序设计方法，多与此相像。

7.8 数组应用：矩阵乘法

如果矩阵 A 乘以 B 得到 C，则必须满足如下规则：

(1) 矩阵 A 的列数与矩阵 B 的行数相等；

(2) 矩阵 A 的行数等于矩阵 C 的行数；

(3) 矩阵 B 的列数等于矩阵 C 的列数。

例如，下面的例子说明两个矩阵是如何相乘的：

$$\begin{pmatrix} 5 & 7 \\ 8 & 3 \\ 7 & 4 \end{pmatrix} \times \begin{pmatrix} 12 & 3 & 6 \\ 4 & 2 & 7 \end{pmatrix} = \begin{pmatrix} 88 & 29 & 79 \\ 108 & 30 & 69 \\ 100 & 29 & 70 \end{pmatrix}$$

在结果矩阵中，第 1 行第 1 列的元素是 74，它通过下列计算得来：

$$5\times12+7\times4=88$$

即若矩阵 $A_{mn}\times B_{nl}=C_{ml}$，则：

$$c_{ij}=\sum_{k=1}^{n} a_{ik}\times b_{kj}$$

其中，A_{mn} 表示 $m\times n$ 矩阵，c_{ij} 是矩阵 C 的第 i 行 j 列元素。

下列程序是求 $A_{34}\times B_{45}=C_{35}$ 的矩阵乘法：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                  ch7_12.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>
# include <iomanip.h>

int a[3][4] = { { 5, 7, 8, 2},
                { - 2, 4, 1, 1},
                { 1, 2, 3, 4} };

int b[4][5] = { {4, - 2, 3, 3, 9},
                {4, 3, 8, - 1, 2},
                {2, 3, 5, 2, 7},
                {1, 0, 6, 3, 4} };

int c[3][5];

int MultiMatrix(int a[][4], int arow, int acol,
```

```
                int b[][5], int brow, int bcol,
                int c[][5], int crow, int ccol);     //函数声明

int main()
{
  if(MultiMatrix(a,3,4, b,4,5, c,3,5))
  {
   cout <<"illegal matrix multiplication.\n";
   return;
  }
  for(int i = 0; i<3; i++)     //输出矩阵乘法的结果
  {
   for(int j = 0; j<5; j++)
       cout <<setw(5) <<c[i][j];
     cout <<endl;
  }
}

int MultiMatrix(int a[][4], int arow, int acol,
                int b[][5], int brow, int bcol,
                int c[][5], int crow, int ccol)
{
  if(acol!= brow)        //正确性检查
   return 1;
  if(crow!= arow)
   return 1;
  if(ccol!= bcol)
   return 1;

  for(int i = 0; i<crow; i++)              //行
   for(int j = 0; j<ccol; j++)             //列
     {
        c[i][j] = 0;                       //此句可以省略,因为c是全局数组
        for(int n = 0; n<acol; n++)  //求一个元素
          c[i][j] += a[i][n] * b[n][j];
     }

  return 0;
}
```

运行结果为：

```
66  35  123  30  123
11  19   37  -5    1
22  13   58  19   50
```

该程序先定义两个全局数组，然后调用矩阵乘法函数，对应地，矩阵乘法函数的参数是三个二维数组表示的矩阵，分别有两个行列项。二维数组的大小最好也要一一对应。

矩阵乘法函数中，先对行列值进行校验，如果不符合要求，及时返回一个出错信息。在用二重循环计算矩阵结果时，又用了一个循环计算对应元素积之和。另外，由于数组 c 是全局的，默认值为全 0，所以求结果时，语句“c[i][j]=0;”可以省略。

小结

数组是一个由若干同类型变量组成的集合，数组中特定的元素通过下标来访问。数组由连续存储单元组成，其起始地址对应于数组的第一个元素。数组名本身是地址，使得数组作为函数参数来传递从空间利用上显得合理。由于C字符串的'\0'结束符特性，所以在字符数组中，要多考虑一个字节安排该字符。

数组应用是广泛的，排序是一种典型的应用。此处的Josephus问题解是在数组中采用了一些技巧。矩阵乘法是二维数组的一个应用，内含三重循环的处理。

在C++中，数组和指针是密切相关的，下一章将讨论指针，只有读完这两章，才能透彻理解C++语言的这些构成。

练习

7.1 一个10个整数的数组(34,91,83,56,29,93,56,12,88,72)，找出最小数和其下标，并在主函数中打印最小数和下标。

7.2 n个数，已按从小到大顺序排列。在主函数中输入一个数，调用一个函数，它把输入的数插入到原有数列中，保持大小顺序，输出插入前后的两个数组，并将被挤出的最大数(有可能就是被插入数)返回给主函数输出。

7.3 17个人围成圈，编号为1～17，从第1号开始报数，报到3的倍数的人离开，一直数下去，直到最后只剩下一人。求此人的编号。

7.4 改进ch7_9.cpp中的冒泡排序算法，使之在新一轮比较中，若没有发生元素交换，则认为已排序完毕。

7.5 输入一个$n \times n$的矩阵，求出两条对角线元素值之和

7.6 5个学生，4门课，要求主函数分别调用各函数实现：

(1) 找出成绩最高的学生序号和课程。

(2) 找出不及格课程的学生序号及其各门课的全部成绩。

(3) 求全部学生各门课程的平均分数，并输出。

7.7 编程求矩阵的加法：

$$\begin{bmatrix} 5 & 7 & 8 \\ 2 & -2 & 4 \\ 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 4 & -2 & 3 \\ 3 & 9 & 4 \\ 8 & -1 & 2 \end{bmatrix}$$

第8章 指 针

C++语言拥有在运行时获得变量的地址和操纵地址的能力，在其他任何语言中，理解它们是如何工作的或许都不如在C++中这么必不可少。这种用来操纵地址的特殊类型变量就是指针。指针用于数组，作为函数参数，用于内存访问和堆内存操作。指针对于成功地进行C++语言程序设计是至关重要的。"指针功能最强，但又最危险。学习本章后，要求能够使用指针，能够用指针给函数传递参数，理解指针、数组和字符串之间的紧密联系，能够声明和使用字符串数组，正确理解命令行参数，理解函数指针的用法。

8.1 指针概念

1. 指针类型

我们学过基本数据类型，如int，float，char，double等，其中每一种基本数据类型都有相应的指针类型。如可以建立整型指针以处理整型数，建立字符指针以处理字符等。

2. 定义指针变量

在这之前，"*"是乘法符号，在这里，用作定义指针。

例如，指向整型数的指针是包含该整型数地址的变量或常量：

```
int * ip;
const int * icp;        //因为常量也具有地址，所以有指向常量的指针
```

*表示指针，int表示该指针的类型是整型，ip和icp是指针变量的名字。

关于指向常量的指针见8.5节。

又例如，指向字符的指针是包含字符地址的变量或常量：

```
char * cptr;
const char * ccptr;     //指向字符常量的指针
```

char表示该指针的类型是字符型，cptr和ccptr是指针变量的名字。

指针变量的定义语句，由数据类型后跟星号，再跟随指针变量名组成。

通常，每个指针都有一个类型(void*指针除外，见8.6节)。指针是变量，因此它与其他基本数据类型一样，凡可声明变量的地方，就可声明指针变量，它可以是全局、静态全局、静态局部和局部的。

上面ip和cptr两个指针定义都分配了空间，但是都没有指向任何内容。正如定义整型或浮点变量没有给它赋值一样，指针也没有被赋值。

定义名为iPtr的指针可以写成：

```
int * iPtr;      //*靠左
int *iPtr;       //*靠右
```

```
int * iPtr;      //* 两边都不靠
```

它们表示同一个意思。**在指针定义中，一个 * 只能表示一个指针**。定义语句：

```
int * iPtr1,iPtr2;
```

表示定义一个 iPtr1 指针变量和一个 iPtr2 的整型变量。如果要定义两个指针变量，须：

```
int * iPtr, * iPtr;
```

3. 建立指针

建立指针包括定义指针和给指针赋初值。

变量存在于内存中的某位置(地址)。例如，一旦有了变量，则放置该变量的地方就用内存地址描述。

用 & 操作符可以获取变量的地址，指针变量用于存放地址。

例如，定义整型指针 iPtr，定义整型变量 iCount，把变量 iCount 的地址赋给指针 iPtr：

```
int * iPtr;
int iCount = 18;
iPtr = &iCount;        //将地址赋给存放地址的变量(指针)
```

指针的工作方式见图 8-1。存放变量 iCount 的地址是 0000：F822，用 &iCount 表示，上例将该地址赋给指针变量 iPtr。等号右边是一个地址，其左边是一个地址左值，即 iPtr，其类型也是一个地址(整型数地址)，等号两边匹配。

这时候的指针 iPtr 赋有了值 0000：F822。

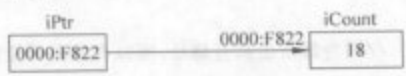

图 8-1　指针的工作方式

4. 间接引用指针

“ * ”是乘法，又可以用于定义指针，在这里可用于指针的间接引用(* 的第三个用途)。

间接引用指针时，可获得由该指针指向的变量内容。

例如，下面的程序间接引用指针 iPtr，输出 iCount 的内容：

```
//* * * * * * * * * * * * * * * * * * * * *
//* *          ch8_1.cpp          * *
//* * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   int * iPtr;
   int iCount = 18;
   iPtr = &iCount;
```

```
    cout << * iPtr <<endl;      //间接引用指针
}
```

运行结果为：

```
18
```

该运行结果就是指针 iPtr 所指向的变量 iCount 中的内容。

*** 放在可执行语句中的指针之前，称为间接引用操作符，* 放在指针定义中时，称指针定义符。**

非指针变量是不能用间接引用操作符的，因为 * 只能作用于地址。例如：

```
cout << * iCount;      //error: 非指针变量不能用间接引用操作符
```

间接引用的指针既可用于右值，也可用于左值。

例如，上例 ch8_1.cpp 中，通过 iPtr 改变变量 iCount 的原始值：

```
 * iPtr = 58;      //指针的间接引用作左值
cout << iCount <<endl;      //将显示 58
```

改变 * iPtr 就是改变 iCount。所以，输出结果为 58。

设想图书馆的卡片，每个卡片都是一个指向书的指针，卡片中有书的位置。看到书的位置，到书库中找到该书，此时就间接地引用了那张卡片。

还书时，把书放回书架，管理员查到书背上的号码，号码与所使用的卡片号码相对应。管理员把书放到书架的固定位置，也间接引用了指针。

5. 指针变量的地址

指针也是变量，**是变量就具有内存地址**。所以指针也有地址。

例如，下面的程序输出 iCount 变量值，以及 iPtr 和 iCount 的地址值：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch8_2.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
  int iCount = 18;
  int * iPtr = &iCount;
  * iPtr = 58;

  cout <<iCount <<endl;
  cout <<iPtr <<endl;
  cout <<&iCount <<endl;          //与 iPtr 值相同
  cout << * iPtr <<endl;          //与 iCount 值相同
  cout <<&iPtr <<endl;            //指针本身的地址
}
```

运行结果为：

```
58
0x0067fe00
0x0067fe00
58
0x0067fdfc
```

0x0067fe00 是指针 iPtr 的值，即变量 iCount 的地址。0x0067fdfc 是指针变量 iPtr 存放的地址。两者是有区别的，见图 8-2。注意地址 0067:FDFC 和 0x0067fdfc 是同一个地址的两种不同表示。

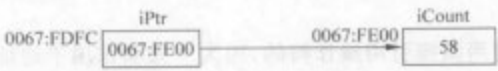

图 8-2　指针值与指针的地址值是不同的

*** iPtr 的类型是整型，指针 iPtr 指向该整数，所以 iPtr 的类型是整型指针，而 iPtr 的地址(即 &iPtr)的类型是整型指针的地址，即指向整型指针的指针。三者都不相同。**在 8.8 节中将会看到，**指针的地址就是二级指针。**

有了取地址操作符“&”和间接引用操作符“*”后，我们有 6 种对变量的操作，并且应该理解这 6 种操作的意义。它们是 iPtr，iCount，* iCount，* iPtr，&iPtr，&iCount。其中，* iCount 是非法的。如果访问 * iCount，BC 将报告“无效的间接引用”(invalid indirection)错误，VC 将报告“非法间接引用”(illegal indirection)错误。

6. 指针与整型数的区别

指针在使用中必须类型匹配。例如：

```
int iCount = 26;
int *  iPtr = &iCount;          //定义语句：* 在此处作定义指针变量用，而非间接引用
* iPtr = &iCount;               //error:不能将整型地址转换成整型数
* iPtr = 50;                    //执行语句：* 在此处作间接引用
```

例中前面两句是定义语句，后面两句是执行语句。* iPtr 的类型是整型，&iCount 的类型是整型指针，**指针值不是整型数**，所以赋值语句“* iPtr＝&iCount;”在 BC 中会引起类型转换的错误(cannot convert int * to int)。

在 32 位机器中，整数和指针都占 4 个字节，内存表示的方式也都是二进制整数，但指针和整数表示的是不同的类型。

强制转换是合法的。例如，允许语句“* iPtr＝(int)&iCount;”，但要注意其赋值的意义。该语句表示将变量 iCount 的地址值作为一个整型数赋给变量 * iPtr，即 iCount 变量。

7. 指针的初始化

普通变量在定义时可以初始化，指针变量也可在定义时初始化。**指针变量初始化的值是该指针类型的地址值。**例如：

```
int iCount = 26;
int *  iPtr = &iCount;      //初始化为整型地址
* iPtr = &iCount;           //error
```

不要将“int * iPtr=&iCount;”与“* iPtr=&iCount;”混淆。前者是定义语句，*是指针定义符，C++为 iPtr 指针分配一个指针空间，并用 iCount 的地址值初始化，后者是赋值语句，左右两边类型不匹配。

* 操作符在指针上的两种用途要区分开：定义或声明时，建立一指针；执行时，间接引用一指针。

指针在使用前，要进行初始化。例如，下面的代码是危险的：

```
int count;
int * iPtr;
* iPtr = 58;     //!
```

指针忘了赋值比整型变量忘了赋值危险得多。

iPtr 当前指向什么地方？该代码能通过编译，但没有赋初值的指针 iPtr 是一个随机地址。“* iPtr=58;”是把 58 赋到内存中的随机位置，因此将改写另一存储位置的数值，甚至修改了栈中的函数返回地址，计算机将死机或进入死循环。

8. 指针类型与实际存储的匹配

指针是有类型的，给指针赋值，不但必须是一个地址，而且应该是一个该指针类型相符的变量或常量的地址。

例如，下面的代码错将浮点类型的变量地址赋给整型指针：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_3.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   float f = 34.5;
   float * fPtr = &f;        //浮点指针
   int * iPtr = &f;          //warning: 将浮点变量的地址赋给整型指针
   iPtr = fPtr;              //warning: 将浮点指针赋给整型指针
   cout <<f <<endl;
   cout <<"iPtr:" << iPtr <<" = >" << * iPtr <<endl;
   cout <<"fPtr:" << fPtr <<" = >" << * fPtr <<endl <<endl;

   * iPtr = * fPtr;          //隐式数据转换

   cout <<f <<endl;
   cout << * iPtr <<endl;
   cout << * fPtr <<endl;
}
```

运行结果为：

```
34.5
iPtr:0x0067:fdfc = >1107951616
fPtr:0x0067:fdfc = >34.5

4.76441e-44
```

```
34
4.76441e-44
```

该程序在BC5(32位机器环境)中运行。程序中,定义整型指针iPtr时,赋给一个指向浮点变量f的地址。在BC中,"int * iPtr=&f;"和"iPtr=fPtr;"都会引起"可疑的指针转换"(Suspiciuos pointer conversion)警告。在VC中该程序通不过编译,将给出"不能将浮点指针转换成整型指针"(cannot convert from 'float *' to 'int *')的错误。

输出f的内容与输出*fPtr的内容是一致的。但是输出*iPtr的内容与*fPtr不一致,尽管其地址是相同的,但所表示的类型不同。**iPtr是整型指针,它总是访问该地址中的整型数,而fPtr是浮点指针,总是访问该地址的浮点数。**

"*iPtr=*fPtr;"是将浮点数赋给整型变量,它是合法的语句,但会引起隐式类型转换(见3.3节),截断小数位数,失去一定的精度。在赋值中,C++自动进行数据类型转换,得到整型数34。由于类型不一致,在VC中将给出"转换浮点到整型"(conversion from 'float' to 'int')的警告。

由于fPtr与iPtr都指向同一地址,所以赋值之后,在该地址上,浮点数被整型数34覆盖了。此时,若要按浮点方式来读取整型数位模式下的34,则会得到一个"意想不到"(若知道浮点数的表示方法,则可推算出该浮点数的值)的浮点数。

程序运行的结果证实,iPtr并不是一个单纯指向f的地址。***iPtr中的内容是起始点为f的地址的一个整型数。**

指针具有一定类型,它是值为地址的变量,该地址是内存中另一个该类型变量的存储位置。或者说指针是具有某个类型的地址。

8.2 指针运算

指针可以进行加减运算。

数组名本身,没有方括号和下标,它实际上是地址,表示数组起始地址。整型数组的数组名本身得到一整数地址,字符数组的数组名得到一字符型地址。

可以把数组起始地址赋给一指针,通过移动指针(加减指针)来对数组元素进行操作。

例如,下面的程序用指针运算来计算数组元素的和:

```
//*********************
//*        ch8_4.cpp    *
//*********************

#include <iostream.h>

int main()
{
   int iArray[10];
   int sum=0;
   int * iPtr=iArray;      //用数组名iArray给指针初始化

   for(int i=0;i<10;i++)     //给数组赋值
     iArray[i]=i*2;
```

```
for(int index = 0; index <10; index ++ )      //计算数组元素之和
{
  sum += * iPtr;
  iPtr ++ ;
}
cout <<"sum is " <<sum <<endl;
}
```

运行结果为：

```
sum is 90
```

指针 iPtr 被初始化为数组起始地址，因此，上例的指针定义语句“int * iPtr=iArray;”中，左右两边的类型是匹配的。它可以写成：

```
int * iPtr;
iPtr = iArray;
```

其中，“iPtr=iArray;”还可以改写成：

```
iPtr = &iArray[0];     //同样表示数组的第一个元素的地址
```

在程序例中，循环重复 10 次，指针遍历数组每个元素。假设在 16 位机器上，数组起始地址是 0x00000100，则指针 iPtr 的值似乎是：

```
0x00000100
0x00000101
0x00000102
0x00000103
0x00000104
...
```

但我们知道 16 位机器中，整数是占 2 个字节的，所以数组元素的地址应该是：

```
0x00000100
0x00000102
0x00000104
0x00000106
0x00000108
...
```

由于指针是具有某个数据类型的地址，所以指针运算都是以数据类型为单位展开的。即，iPtr 是个整型指针，iPtr ++ 使指针指向下一个整型数。因而，iPtr 的地址值其实增加了 2，见图 8-3。

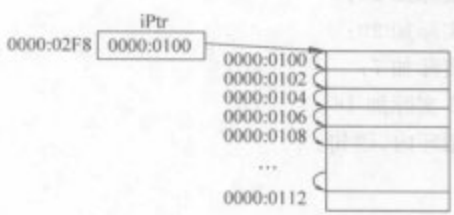

图 8-3　指针 iPtr ++ 的移动

例如，下面的程序显示了指针移动时其地址的变化和指向的内容：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                ch8_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   int iArray[10];
   int* iPtr = iArray;      //用数组名 iArray 给指针初始化

   for(int i = 0;i<10;i++)     //给数组赋值
     iArray[i] = i * 2;

   for(int index = 0; index <10; index++)     //显示每个数组元素
   {
     cout <<"&Array[" <<index <<"]:" <<iPtr
          <<" = >" << * iPtr <<endl;
     iPtr++;
   }

}
```

运行结果为：

```
&iArray[0]:0x0067fdd4 = >0
&iArray[1]:0x0067fdd8 = >2
&iArray[2]:0x0067fddc = >4
&iArray[3]:0x0067fde0 = >6
&iArray[4]:0x0067fde4 = >8
&iArray[5]:0x0067fde8 = >10
&iArray[6]:0x0067fdec = >12
&iArray[7]:0x0067fdf0 = >14
&iArray[8]:0x0067fdf4 = >16
&iArray[9]:0x0067fdf8 = >18
```

只有加法和减法可用于指针运算。

在 16 位机器上，浮点数占 4 字节，长整数占 4 字节，字符占 1 字节，双精度数占 8 字节，所以：

对浮点型指针加 6 实际加 24；

对长整型指针加 5 实际加 20；

对字符型指针加 7 实际加 7；

对双精度型指针加 2 实际加 16。

在 ch8_4.cpp 中的循环内，语句：

```
sum += * iPtr;
iPtr++;
```

可压缩成一条语句：

```
sum += *(iPtr++);
```

以更有效地处理数组。由于++与*的操作符优先级相同，它们是右结合的，所以括号可以省略。即可表示为：

```
sum += *iPtr++;
```

"*iPtr++"这种形式的表达式在C++中很常见。

指针减法的原理与加法相同。只是，不论指针的加法还是减法，其访问操作都必须是有意义的，否则是危险的。例如：

```
int a[5];
int* iPtr=&a[1];
iPtr--;              //指向 &a[0]
*iPtr=3;             //ok
iPtr--;              //指向 &a[-1],dangerous!
*iPtr=6;             //damage!
```

8.3 指针与数组

数组名可以拿来初始化指针，数组名就是数组第一个元素地址。即对于数组a，有：a等于&a[0]。此外，对于：

```
int a[100];
int* iPtr=a;
```

我们有第i个元素：

```
a[i]  等价于  *(a+i)  等价于  iPtr[i]  等价于  *(iPtr+i)
```

a[i]表示数组的第i个元素的值。而a+i表示第i个元素的地址，对其间接访问，即*(a+i)就表示第i个元素的值。另外，**下标操作是针对地址而不仅仅是针对数组名的**，所以iPtr[i]也表示第i个元素的值。上面4个式子等价的事实使得数组与指针相互转换非常灵活。

不但如此，相应的，我们还有第i个元素的地址：

```
&a[i]  等价于  a+i  等价于  iPtr+i  等价于  &iPtr[i]
```

数组名本身是一指针，它的类型是指向数组元素的指针。&a[i]表示数组第i个元素的地址。该4式的等价与前面4式的等价是对应的。

例如，对于前面数组的求和运算，可以有下面5种方法：

```
//*********************
//**          ch8_6.cpp          **
//*********************

#include <iostream.h>

int sum1,sum2,sum3,sum4,sum5;      //存放每种方法的结果
```

```
int iArray[ ] = {1,4,2,7,13,32,21,48,16,30};     //全局数组
int * iPtr;

int main()
{
   int size,n;
   size = sizeof(iArray)/sizeof( * iArray);     //元素个数

   for (n = 0; n<size; n ++ )     //方法 1
   {
    sum1 += iArray[n];
   }

   iPtr = iArray;
   for(n = 0; n<size; n ++ )     //方法 2
   {
    sum2 += * iPtr ++ ;
   }

   iPtr = iArray;        //此句不能省略,因为方法 2 修改了 iPtr
   for(n = 0; n<size; n ++ )     //方法 3
   {
    sum3 += * (iPtr + n);
   }

   iPtr = iArray;        //此句可以省略,因为方法 3 没有修改 iPtr
   for(n = 0; n<size; n ++ )     //方法 4
   {
    sum4 += iPtr[n];
   }

   for(n = 0; n<size; n ++ )     //方法 5
   {
    sum5 += * (iArray + n);
   }

   cout <<sum1 <<endl
        <<sum2 <<endl
        <<sum3 <<endl
        <<sum4 <<endl
        <<sum5 <<endl;
}
```

运行结果为：

```
174
174
174
174
174
```

程序中求元素个数的表达式中，sizeof(* iArray)表示数组元素类型所占的字节数，它可以表示成 sizeof(int)。没有这样做的原因是，该语句可以适应 float 或 double 等类型的数组，任凭全局数据的类型如何变化，主函数中的语句都不用作修改。

一般来说，在机器指令的实现上，指针表示不比下标表示效率更低。所以，*(iPtr+n)(指针表示)或*(iArray+n)的表示总是不差于iPtr[n](下标表示)或iArray[n]的表示，但从可读性来说，后者似乎优于前者。

数组名是指针常量，区别于指针变量，所以，给数组名赋值是错误的。

例如，下面的代码对数组求和，企图以数组名的增量来实现：

```
int iArray[100];
int sum = 0;
//...
for(int i = 0; i<100; i ++ )
{
   sum += iArray;
   iArray ++ ;        //error:数组名不是左值
}
```

由于**指针常量不是左值**，"iArray ++ ;"意味着"iArray=iArray+1;"，即要求iArray是一个左值。所以，上例中，BC编译器将给出"Lvalue required"的错误，VC编译器将给出"left operand must be an lvalue"的错误。

对于编译器来说，**数组名表示内存中分配了数组的固定位置，修改了这个数组名，就会丢失数组空间**，所以数组名所代表的地址不能被修改。

8.4 堆内存分配

1. 堆内存

堆(heap)是内存空间。堆是区别于栈区、全局数据区和代码区的另一个内存区域。**堆允许程序在运行时(而不是在编译时)，申请某个大小的内存空间。**

在通常情况，一旦定义了一个数组，那么不管这个数组是局部的(在栈中分配)还是全局的(在全局数据区分配)，它的大小在程序编译时即是已知的，因为必须用一个常数对数组的大小进行声明：

```
int i = 10;
//...
int a[i];     //error: 定义时不允许数组元素个数为变量
int b[20];    //ok
```

但是，在编写程序时不是总能知道数组应该定义成多大，如果定义数组元素多了就浪费内存了，更何况有时候根本不知道需要使用多少个数组元素。因此，需要在程序运行时从系统中获取内存。

程序在编译和连接时不予确定这种在运行中获取的内存空间，这种内存需求随着程序运行的进展而时大时小，这种运行中申请的内存就是堆内存，所以堆内存是动态的。堆内存也称动态内存。

2. 获得堆内存

函数malloc()是C程序获得堆内存的一种方法，它在alloc.h头文件中声明。malloc()

函数的原型为：

```
void * malloc(size_t size);
```

size_t 即 unsigned long。

该函数从堆内存中“切下”一块 size 大小的内存，将指向该内存的地址返回。该内存中的内容是未知的。

例如，下面的程序从堆中获取一个整数数组，赋值并打印：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                ch8_7.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <alloc.h>

int main()
{
   int arraysize;     //元素个数
   int * array;
   cout <<"please input a number of array elements:\n";
   cin >>arraysize;

   array = (int * )malloc(arraysize * sizeof(int));     //堆内存分配

   for(int count = 0; count<arraysize; count ++ )
    array[count] = count * 2;

   for(int count = 0; count<arraysize; count ++ )
    cout <<array[count] <<"    ";

   cout <<endl;
}
```

运行结果为：

```
C>ch8_7
please input a number of array elements:
10
0  2  4  6  8  10  12  14  16  18
```

程序编译和连接时，在栈中分配了 arraysize 整型变量和 array 整型指针变量空间。程序运行中，调用函数 malloc()并以键盘输入的整数值作为参数。malloc()函数在堆中寻找未被使用的内存，找够所需的字节数后返回该内存的起始地址。因为 malloc()函数并不知道用这些内存干什么，所以它返回一个没有类型的指针(见 8.6 节)。但对整数指针 array 来说，malloc()函数的返回值必须显式转换成整数类型指针才能被接受(ANSI C++标准)。

一个拥有内存的指针完全可以被看作为一个数组，而且位于堆中的数组和位于栈中的数组结构是一样的。表达式“array[count]=count * 2;”正是这样应用的。

上例中并没有保证一定可以从堆中获得所需内存。有时，系统能提供的堆空间不够分配，这时系统会返回一个空指针值 NULL。这时所有对该指针的访问都是破坏性的，因此

调用 malloc()函数更完善的代码应该如下：

```
if(array = (int * )malloc(arraysize * sizeof(int)) == NULL)
{
   cout <<"can't allocate more memory, terminating.\n";
   exit(1);
}
```

3. 释放堆内存

我们把堆看作是可以按要求进行分配的资源或内存池。程序对内存的需求量随时会增大或缩小。程序在运行中可能经常会不再需要由 malloc()函数分配的内存，而且程序还未运行结束，这时就需要把先前所占用的内存释放回堆以供程序的其他部分所用。

函数 free()返还由 malloc()函数分配的堆内存，其函数原型为：

```
void free(void * );
```

free()参数是先前调用 malloc()函数时返回的地址。把其他值传给 free()很可能会造成灾难性的后果。

例如，下面的程序完善程序 ch8_7.cpp：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_8.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>
# include <alloc.h>

int main()
{
   int arraysize;     //元素个数
   int * array;
   cout <<"please input a number of array elements:\n";
   cin >>arraysize;

   if((array = (int * )malloc(arraysize * sizeof(int))) == NULL)
   {
     cout <<"can't allocate more memory, terminating.\n";
     exit(1);
   }

   for(int count = 0; count<arraysize; count ++ )
    array[count] = count * 2;

   for(int count = 0; count<arraysize; count ++ )
    cout <<array[count] <<"  ";

   cout <<endl;
   free(array);          //释放堆内存
}
```

运行结果为：

```
C>ch8_8
please input a number of array elements:
10
0  2  4  6  8  10  12  14  16  18
```

4. new 与 delete

new 和 delete 是 C++专有的操作符，它们不用头文件声明。new 类似于函数 malloc()，分配堆内存，但比 malloc()更简练。new 的操作数为数据类型，它可以带初始化值表或单元个数。new 返回一个具有操作数之数据类型的指针。

返回 delete 类似于函数 free()，释放堆内存。delete 的操作数是 new 返回的指针，当返回的是 new 分配的数组时，应该带[]。

例如，下面的程序是程序 ch8_8.cpp 的新版：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_9.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>
//# include <alloc.h>     //无须头文件

int main()
{
   int arraysize;     //元素个数
   int * array;
   cout <<"please input a number of array elements:\n";
   cin >>arraysize;

     if((array = new int[arraysize]) = = NULL)     //分配堆内存
     {
     cout <<"can't allocate more memory, terminating.\n";
     exit(1);
    }

    for(int count = 0; count<arraysize; count ++ )
     array[count] = count * 2;

    for(int count = 0; count<arraysize; count ++ )
     cout <<array[count] <<"  ";

    cout <<endl;
    delete[ ]array;     //释放堆内存
}
```

运行结果为：

```
C>ch8_9
please input a number of array elements:
10
0  2  4  6  8  10  12  14  16  18
```

上例中可以看到，new 的返回值无须显式转换类型，直接赋给整数指针 array。new 的

操作数是 int[arraysize],它只要指明什么类型和要几个元素就可以了,它比函数 malloc()更简捷。虽然 new 和 delete 在性能上略逊于函数 malloc()和 free(),但却更安全并具有更丰富的功能。在第 14 章将进一步展开 new 和 delete 的讨论。

8.5 const 指针

对于下面涉及指针定义和操作的语句:

```
int a = 1;
int * pi;
pi = &a;
* pi = 58;
```

可以看到,一个指针涉及到两个变量,指针本身 pi 和指向的变量 a。修改这两个变量的对应操作为"pi=&a;"和"*pi=58;"。

1. 指向常量的指针(常量指针)

在指针定义语句的类型前加 const,表示指向的对象是常量。例如:

```
const int a = 78;
const int b = 28;
int c = 18;
const int * pi = &a;      //指针类型前加 const
* pi = 58;                //error: 不能修改指针指向的常量
pi = &b;                  //ok:指针值可以修改
* pi = 68;                //error: 同上
pi = &c;
* pi = 88;                //error: 同上
c = 98;                   //ok
```

a 是常量,将 a 的地址赋给指向常量的指针 pi,使 a 的常量性有了保证。如果企图修改 a,则会引起"不能修改常量对象"(Cannot modify a const object)的编译错误。

可以将另一个常量地址赋给指针"pi=&b;"(指针值可以修改),这时,仍不能进行"*pi=68;"的赋值操作,从而保护了被指向的常量不被修改。见图 8-4,图中阴影部分表示不能被修改。可以将一个变量地址赋给指针"pi=&c;",这时,由于不能进行"*pi=88;"的赋值操作,从而保护了被指向的变量在指针操作中不被修改。**定义指向常量的指针只限制指针的间接访问操作,而不能规定指针指向的值本身的操作规定性。**例如,变量 c 可以修改,这在函数传递中经常被使用。

图 8-4 指向常量的指针

下面的程序将两个一样大小的数组传递给一个函数,让其完成复制字符串的工作,为了防止作为源数据的数组遭到破坏,声明该形参为常量字符串:

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_10.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void mystrcpy(char * dest, const char * source)
{
   while( * dest ++  =  * source ++ );
}

int main()
{
   char a[20] = "How are you!";
   char b[20];
   mystrcpy(b,a);
   cout <<b <<endl;
}
```

运行结果为：

```
How are you!
```

变量字符串 a 传递给函数 mystrcpy()中的 source，使之成为常量，不允许进行任何修改。但在主函数中，a 却是一个普通数组，没有任何约束，可以被修改。

由于数组 a 不能直接赋值给 b(即不允许 b=a)，所以通过一个函数实现将数组 a 的内容复制给数组 b。

函数 mystrcpy()中的语句是一个空循环，条件表达式是一个赋值语句。随着一个赋值动作，便将一个 a 数组中的字符赋给了 b 数组中对应的元素，同时两个数组参数都进行增量操作以指向下一个元素。只要所赋的字符值不是'\0'(条件表达式取假值)，则循环就一直进行下去。

指针常量定义"const int * pi=&a;"告诉编译，* pi 是常量，不能将 * pi 作为左值进行操作。

2. 指针常量

在指针定义语句的指针名前加 const，表示指针本身是常量。例如：

```
char * const pc = "asdf";      //指针名前加 const 定义指针常量
pc = "dfgh";                   //error: 指针常量不能改变其指针值
 * pc = 'b';                   //ok: pc 内容为"bsdf"
 * (pc + 1) = 'c';             //ok: pc 内容为"bcdf"
 * pc ++ = 'y';                //error: 指针常量不能改变其指针值
const int b = 28;
int * const pi = &b;           //error: 不能将 const int * 转换成 int *
```

pc 是指针常量，**在定义指针常量时必须初始化**，就像常量初始化一样。这里初始化的值是字符串常量的地址，见图 8-5。

图 8-5　指向变量的指针常量图

由于 pc 是指针常量，所以不能修改该指针值。“pc="dfgh";”将引起一个“不能修改常量对象”(Cannot modify a const object)的编译错误。

pc 所指向的地址中存放的值并不受指针常量的约束，即 * pc 不是常量，所以“ * pc='b';”和“ * (pc+1)='c';”的赋值操作是允许的。但“ * pc++='y';”是不允许的，因为该语句修改 * pc 的同时也修改了指针值。

由于此处 * pc 是不受约束的，所以，将一个常量的地址赋给该指针“int * const pi=&b;”是非法的，它将导致一个不能将 const int * 转换成 int * 的编译错误，因为那样将使修改常量(如：“ * pc=38;”)合法化。

指针常量定义“int * const pc=&b;”告诉编译，pc 是常量，不能作为左值进行操作，但是允许修改间接访问值，即 * pc 可以修改。

3. 指向常量的指针常量(常量指针常量)

可以定义一个指向常量的指针常量，它必须在定义时进行初始化。例如：

```
const int ci = 7;
int ai;
const int * const cpc = &ci;      //指向常量的指针常量
const int * const cpi = &ai;      //ok
cpi = &ai;                        //error: 指针值不能修改
 * cpi = 39;                      //error: 不能修改所指向的对象
ai = 39;                          //ok
```

cpc 和 cpi 都是指向常量的指针常量，它们既不允许修改指针值，也不允许修改 * cpc 的值，见图 8-6。

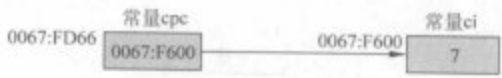

图 8-6 指向常量的指针常量

如果初始化的值是变量地址(如 &ai)，那么不能通过该指针来修改该变量的值。也即“ * cpi=39;”是错误的，将引起“不能修改常量对象”(Cannot modify a const object)的编译错误。但“ai=39;”是合法的。

常量指针常量定义“const int * const cpc=&b;”告诉编译，cpc 和 * cpc 都是常量，它们都不能作为左值进行操作。

8.6 指针与函数

1. 传递数组的指针性质

一旦把数组作为参数传递到函数中，则在栈上定义了指针，可以对该指针进行递增、递减操作。

例如，下面的代码传递一个数组，并对其进行求和运算：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_11.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void Sum(int array[],int n)
{
   int sum = 0;
   for(int i=0;i<n; i++)
   {
    sum += *array;
    array++;      //ok:array是一个指针,可以作为左值
   }
   cout <<sum <<endl;
}

int main()
{
   int a[10]={1,2,3,4,5,6,7,8,9,10};
   Sum(a,10);
}
```

运行结果为：

```
55
```

在主函数中，对 Sum()函数进行了调用。传递的数组参数在 Sum()中，实质上是一个指针，所以**声明 Sum(int *array[],int n)与 Sum(int *array,int n)是等价的**。调用的示意见图 8-7。

图 8-7　数组作为参数的函数调用

由于形参 array 是指针变量而不是数组，所以它所占的空间是指针变量大小而不是数组空间大小。不能用 sizeof(array)/sizeof(*array)来求取数组元素个数，这就是第二参数 n(表示数组元素个数)必须要给的原因。

尽管形参中说明的形式是 array[]数组的形式，但 C++已经明确告诉作为参数传递的数组就是指针变量，所以 array 可以作为左值进行 array++运算。

2. 使用指针修改函数参数

通过对函数的调用，函数返回一值。然而，在很多情形下，希望函数返回不止一个值。例如，下面是想通过 swap()函数调用来交换两个整数值的程序：

```
//* * * * * * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch8_12.cpp                        * *
//* * * * * * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void swap(int, int);

int main()
{
   int a = 3, b = 8;
   cout <<"a = " <<a <<", b = " <<b <<endl;
   swap(a,b);
   cout <<"after swapping...\n";
   cout <<"a = " <<a <<", b = " <<b <<endl;
}

void swap(int x, int y)
{
   int temp = x;          //交换两个形参
   x = y;
   y = temp;
}
```

运行结果为：

```
a = 3, b = 8
after swapping...
a = 3, b = 8
```

该 swap()函数无法返回更多的值，而且也无法改变 a 和 b 的值。由于函数参数值的传递是实参到形参的复制，被调函数内部对形参的修改并不反映到上层函数的实参中，致使 swap()中 x 和 y 作了交换，而 main()中 a 和 b 并没有交换。该函数的内存栈结构见图 8-8。

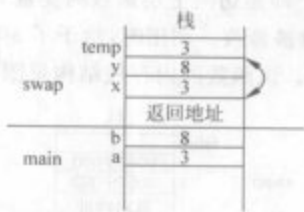

图 8-8　传递整数值的内存布局

传递指针可以使函数“返回”更多的值。这里的“返回”不是函数返回类型描述返回值的返回，而是反映了上层函数中的变量给被调函数修改了。

例如，下面的程序通过传递指针来实现整数值的交换：

```
//* * * * * * * * * * * * * * * * * * * *
//* *          ch8_13.cpp                * *
//* * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void swap(int *, int *);

int main()
{
   int a = 3, b = 8;
   cout <<"a = " <<a <<", b = " <<b <<endl;
   swap(&a,&b);
   cout <<"after swapping...\n";
   cout <<"a = " <<a <<", b = " <<b <<endl;
}

void swap(int * x,int * y)
{
   int temp = *x;
   *x = *y;
   *y = temp;
}
```

运行结果为：

```
a = 3, b = 8
after swapping...
a = 8, b = 3
```

传递指针的函数调用实现过程为：

(1) 函数声明中指明指针参数，即上例的 void swap(int * x,int * y)；

(2) 函数调用中传递以变量的地址，即上例的 swap(&a,&b)；

(3) 函数定义中对形参进行间接访问。

对 * x 和 * y 的操作，实际上即是访问上层函数的变量 a 和 b。通过函数中的局部变量 temp 的中介，使变量 a 和 b 的值被修改。调用时，给予了 a 和 b 的地址作为参数，swap()函数运行后，“返回”了 a 和 b 的值。该函数的内存栈结构见图 8-9。

图 8-9　传递指针的内存布局

由于指针可以间接访问变量，使得函数调用中值的返回变得灵活多样，可以更方便地实现函数之间的数据传递。

但是要看到，指针的灵活是以破坏函数的黑盒特性为代价的。它使函数可以访问本函数的栈空间以外的内存区域（函数的副作用初露端倪），以致引起了：

（1）可读性问题：因为间接访问比直接访问相对难理解，传递地址比传递值的直观性要差，函数声明与定义也相对比较复杂。

（2）重用性问题：函数调用依赖于上层函数或整个外部内存空间的环境，丧失了黑盒的特性，所以无法作为公共的函数模块来使用。

（3）调试复杂性问题：跟踪错误的区域从函数的局部栈空间扩大到整个内存空间。不但要跟踪变量，还要跟踪地址。错误现象从简单的不能得到相应的返回结果，扩展到系统环境遭破坏甚至死机。

3. 指针函数

返回指针的函数称为指针函数。**指针函数不能把在它内部说明的具有局部作用域的数据地址作为返回值。**

例如，下面的程序打印一个用整型指针指向的整数值：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch8_14.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int * getInt(char * str)     //指针函数
{
   int value = 20;
   cout <<str <<endl;
   return &value;     //warning: 将局部变量的地址返回是不妥的
}

void somefn(char * str)
{
   int a = 40;
   cout <<str <<endl;
}

int main()
{
   int * pr = getInt("input a value:");     //赋值取自返回的指针值
   cout << * pr <<endl;                     //第一次输出 * pr
   somefn("It is uncertain.");
   cout << * pr <<endl;                     //第二次输出 * pr
}
```

运行结果为：

```
input a value:
20
It is uncertain.
4435500
```

该程序中的 getInt()函数返回一个局部作用域变量的地址是不妥的。因为 getInt()函数结束时,其栈中的变量 value 随之消失。在 BC 编译器中,将给出"可疑的指针转换"(suspicious pointer conversion)警告;在 VC 编译器中将给出"返回了一个局部变量或临时空间的地址"(returning address of local variable or temporary)警告。

在主函数中,指针 pr 得到一个局部变量的地址,输出该地址中的内容 20。随后,调用了另一个函数 somefn(),该函数将前一个被调函数的栈空间位置作为自己的栈工作空间,所以函数返回后,栈中内容发生了变化。而主函数下一个语句输出的是该变化了的内存空间内容。

可以返回堆地址,可以返回全局或静态变量的地址,但不要返回局部变量的地址。

4. void 指针

void 指针是空类型指针,它不指向任何类型,即 void 指针仅仅只是一个地址。所以**空类型指针不能进行指针运算,也不能进行间接引用**,因为指针运算和间接引用都需要指针的类型信息。例如:

```
void * p;      //仅仅表示 p 存放一个地址
p++            //error: +、- 运算离不开指针类型
* p = 20.5;    //error: 访问 p 指向的变量空间需要变量类型信息
```

由于其他指针都包含有地址信息,所以将其他指针的值赋给空类型指针是合法的,反之,将空类型指针赋给其他指针则不被允许,除非进行显式转换。例如:

```
int a = 20;
int * pr = &a;
void * p = pr;      //ok: 将整型指针值赋给空类型指针
pr = p;             //error: 不能将空类型指针赋给其他指针
pr = (int * )p;     //ok: 显式转换被允许
```

上面的代码中,将空类型指针赋给其他指针时,在 BC 编译器中将给出"可疑的指针转换"(suspicious pointer conversion)警告;在 VC 编译器中将给出"不能将空类型指针转换成整型指针"(cannot convert from 'void * ' to 'int * ')的编译错误。

下面的程序是空类型指针的一个应用。函数 memcpy()在头文件 mem.h 中声明,它的功能为从源 src 中拷贝 n 个字节到目标 dest 中:

```
//* * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_15.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <mem.h>

int main()
{
   char src[10] = "* * * * * * * * *";
   char dest[10];
   char * pc = (char * )memcpy(dest,src,10);
   cout <<pc <<endl;
}
```

运行结果为：

```
* * * * * * * * *
```

函数 memcpy()的原型为：

```
void* memcpy(void* d,const void* s, size_t n);
```

其中 size_t 为 unsigned int。该函数返回空类型指针 dest 的值，d 形参值是主函数中的实参 dest 的数组首地址。

8.7 字符指针

1. 字符数组和字符串常量

在 C++中，作为字面值，有两类字符串，一类用于字符数组初始化，它在完成将内容填写到所创建的字符数组之中后，随即消失；另一类作为表达式，或输出，或赋值，或参加运算，这类字符串在运行中有它自己的存储空间，可以寻址访问。例如：

```
char buffer[] = "hello";     //字符数组
cout <<"good" <<endl;        //字符串常量
```

在例中，"hello"用来给数组初始化，属于前一类。"good"字符串属于后一类。

字符串的类型是指向字符的指针（字符指针 char *），它与字符数组名同属于一种类型。字符串在内存中以'\0'结尾。这种类型的字符串称为 C 字符串，或 ASCIIZ 字符串。

2. 字符串的属性

字符串通常存放在内存 data 区中的 const 区。而字符数组是根据其数据存储的特点存放在相应的位置上。如果字符数组是全局变量，就存放在内存的 data 区中全局或静态区，如果字符数组是局部数据，就存放在内存的栈区等，见图 8-10。

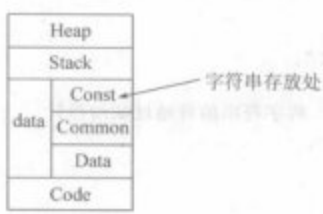

图 8-10 字符串的内存位置

当编译器遇到一字符串时，就把它放到字符串池（data 区的 const 区）中，以'\0'作结束符，记下其起始地址，在所构成的代码中使用该地址。这样，字符串就"变成"了地址。

由于字符串的地址属性，所以**两个同样字符组成的字符串的地址是不相等的。**

例如，下面的程序不会输出"equal"字符串：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_16.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   if("join" == "join")
    cout <<"equal\n";
   else
     cout <<"not equal\n";
}
```

运行结果为：

```
not equal
```

程序中两个字符串的比较实质上是两个地址的比较。在编译时，给了这两个字符串不同的存放地点，所以两个"join"字符串的地址是不同的。要使两个字符串真正从字面上进行比较，可以用库函数 strcmp()，见稍后的描述。

3. 字符指针

字符串，字符数组名，字符指针均属于同一种数据类型。

例如，下面的程序描述了字符指针的操作：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_17.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   char buffer[10] = "ABC ";
   char * pc;
   pc = "hello";     //ok: 将字符串的首地址赋给指针
   cout <<pc <<endl;
   pc ++ ;
   cout <<pc <<endl;
   cout << * pc <<endl;
   pc = buffer;
   cout <<pc;
}
```

运行结果为：

```
hello
ello
e
ABC
```

buffer 初始化为"ABC",buffer[3]='\0',buffer[4]~buffer[9]的内容不重要。

pc 是字符指针,定义时分配该变量空间但没有初始化,之后将"hello"赋给 pc。由于字符串是地址,所以"pc="hello";"语句完全合法。pc 实际上指向"hello"中的'h'字符。当 pc++ 时,pc 就指向这个字符串中的'e'字符。

输出字符指针就是输出字符串。所以输出 pc 时,便从'e'字符的地址开始,直到遇到'\0'结束。

输出字符指针的间接引用,就是输出单个字符。当输出 * pc 时,便是输出 pc 所指向的字符,见图 8-11。

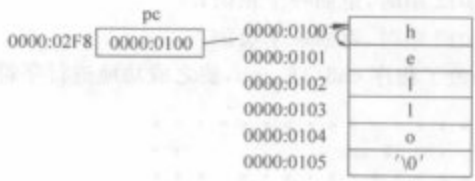

图 8-11 字符指针指向字符串常量

Buffer 是字符数组名,所以"pc=buffer;"也是合法的。之后的输出只输出字符数组中的前 3 个字符,遇到'\0'就结束了。

当 pc 指向 buffer 后,字符串常量"hello"仍逗留在内存的 data 区,但是再也访问不到该字符串(数据丢失)了。所以对于字符串赋给字符指针的情形,指针一般不再重新赋值。

4. 字符串比较

我们已经知道,两个字符串的比较是地址的比较。同理,两个数组名的比较也是地址的比较。

例如,下面的代码不会输出"equal":

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch8_18.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   char buffer1[10] = "hello";
   char buffer2[10] = "hello";

   if(buffer1 = = buffer2)
     cout <<"equal\n";
   else
     cout <<"not equal\n";
}
```

运行结果为:

```
not equal
```

因为 buffer1 和 buffer2 都是地址，buffer1 与 buffer2 是不同的数组，占有不同的内存空间，所以其地址也不同，即 buffer1 与 buffer2 不相等。那么，要进行字符串比较，应该怎么办？

字符串比较应该是逐个字符一一比较，通常使用标准库函数 strcmp()，它在 string.h 头文件中声明，其原型为：

```
int strcmp(const char * str1, const char * str2);
```

其返回值如下：

(1) 当 str1 串等于 str2 串时，返回值 0；

(2) 当 str1 串大于 str2 串时，返回一个正值；

(3) 当 str1 串小于 str2 串时，返回一个负值。

例如，下面的程序修改了程序 ch8_18.cpp，使之成功地进行字符串比较：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_19.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>
#include <string.h>

int main()
{
   char buffer1[10] = "hello";
   char buffer2[10] = "hello";

   if(strcmp(buffer1,buffer2) = = 0)
     cout <<"equal\n";
   else
     cout <<"not equal\n";
}
```

运行结果为：

```
equal
```

5. 字符串赋值

C++中可以用字符串去初始化字符数组，但是不能对字符数组赋予一个字符串，原因是数组名是常量指针，不是左值。

例如，下面的代码不能通过编译：

```
char buffer[10];
buffer = "hello"; //error
```

可以使用标准函数 strcpy()来对字符数组进行赋值。strcpy()的声明在头文件 string.h 中，它的原型为：

```
char * strcpy(char * dest, const char * src);
```

strcpy()函数的返回值为 dest 值，一般都舍弃该值。

例如，下面的代码对字符数组进行赋值：

```
char buffer1[10];
char buffer2[10];
strcpy(buffer1,"hello");
strcpy(buffer2,buffer1);
```

函数 strcpy()仅能对以'\0'作结束符的字符数组进行操作。若要对其他类型的数组赋值可调用函数 memcpy()：

```
int intarray1[5] = {1,3,5,7,9};
int intarray2[5];
memcpy(intarray2,intarray1,5 * sizeof(int));
```

8.8 指针数组

1. 定义指针数组

一个数组中若每个元素都是一个指针，则为指针数组。

例如，下面的代码定义一个字符指针数组，并对其初始化：

```
char * proname[] = { "Fortran",
                     "C ",
                     "C++" };
```

该指针数组具有变量属性，所以可以是全局的、静态的和局部的。proname 数组中每个元素都存放一个字符指针(char *)，初始化中的每个值都是一个字符串字面量，这些字面量存储在内存 data 区的 const 子区中，而不管 proname 是全局还是局部变量。字符串在 const 子区中有可能连续分布，也可能不是，但这无关紧要，重要的是指向这些字符串的指针是连续排列在指针数组中的。在 16 位机器中，proname 数组空间占 6 个字节以存放 3 个字符指针，见图 8-12。

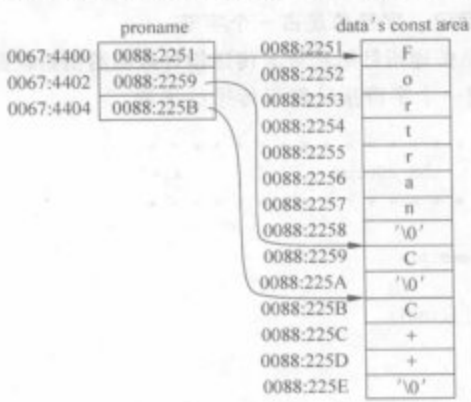

图 8-12 字符数组的内存表示

2. 指针数组与二维数组

指针数组与二维数组是有区别的。前面看到字符指针数组的内存表示，指针所指向的字符串是不规则长度的。

proname 数组本身含有 6 个字节(假设每个指针占 2 个字节);

proname[0]指向的字符串长 8 个字节(包括'\0');

proname[1]指向的字符串长 2 个字节;

proname[2]指向的字符串长 4 个字节;

proname 所占内存总数为 20 个字节。

使用二维数组时的情况:

```
char name[3][8] = { "Fortran",
                    "C ",
                    "C++"};
```

在二维数组情况下,每一列的大小必须是一样的,因此只能将数组中所要存储的字符串中最长列的大小作为数组的列的大小。这样,共需 3 * 8=24 字节。

如果被赋值的各字符串长短相差悬殊,则二维数组空间就浪费多一些。

3. 指向指针的指针

指针数组是数组,是数组则其名字便为指针常量。指针具有类型,那么指针数组名是什么类型呢?

指针数组名是指向指针的指针(即二级指针)。

例如,下面的代码描述了指向字符指针的指针:

```
char * pc[] = {"a","b","c"};
char * * ppc;
ppc = pc;  //ok
```

这里的初始化值"a","b","c"必须为双引号,如果单引号则表示字符而不是字符串,而且字符没有'\0'的结束标记。字符总是占一个字节。

传递数组给函数就是传递指针给函数。**传递指针数组给函数就是传递二级指针给函数。**

例如,下面的程序把一个字符指针数组传递给函数:

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_20.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void Print(char * [],int);

int main()
{
   char * pn[] = {"Fred","Barney","Wilma","Betty"};
   int num = sizeof(pn)/sizeof(char * );
   Print(pn,num);
}

void Print(char * arr[], int len)
{
 for(int i = 0; i<len; i++)     //输出各字符串
```

```
    cout <<(int)arr[i] <<"  "          //输出字符串地址
         <<arr[i] <<endl;              //输出字符串
}
```

运行结果为：

```
4202628 Fred
4202633 Barney
4202640 Wilma
4202646 Betty
```

arr 是指向字符指针的指针，所以 * arr 是字符指针，*(arr+i)是 arr 开始的第 i 个字符指针，*(arr+i)就是 arr[i]。

arr 是指针而不是数组，尽管形参声明为指针数组。这是由传递数组参数即为指针的特性决定的。所以，函数的输出语句可以写为：

```
cout <<(int) * arr;
cout << * arr ++ <<endl;      //arr 值可以改变
```

输出字符指针就是输出字符串。如果要输出字符指针的地址值，应该将字符指针强制转换为整型："cout <<(int)arr[i];"。例中输出的地址是 10 进制表示的整数。若要以十六进制表示输出，则须用流控制的方法实现。

传递字符指针的指针之内存布局见图 8-13。

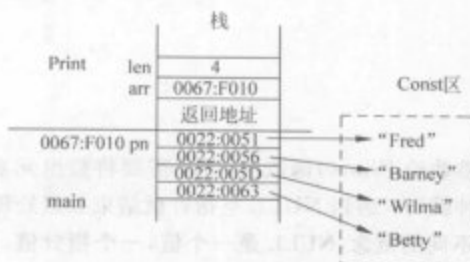

图 8-13　传递字符指针的指针之内存布局

4. NULL 指针值

NULL 是空指针值，它不指向任何地方。不同的操作系统，会使编译取不同的 NULL 值。因此，NULL 是个不确定值。在 PC 中的 BC 和 VC、NULL 都取 0 值。如果严格考虑到移植，那么不应养成 NULL 即为 0 的习惯。

例如，下面的代码可移植性差：

```
char ch = NULL;      //NULL 是个不确定值
char * pc = NULL;    //ok: 将空指针值赋给 pc
```

在程序中，如果一个数组大小不定，则处理指针数组时可以利用在数组末尾设置 NULL 来解决。

例如，下面的程序通过对 ch8_20.cpp 中的指针数组设置 NULL 的方法进行处理：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch8_21.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void Print(char * []);

int main()
{
  char * pn[] = {"Fred","Barney","Wilma","Betty",NULL};
  Print(pn);
}

void Print(char * arr[])
{
  while( * arr!= NULL)     //输出各字符串
  {
    cout <<(int) * arr <<"     "
         << * arr <<endl;
    arr++;
  }
}
```

运行结果为：

```
4202628  Fred
4202633  Barney
4202640  Wilma
4202646  Betty
```

在主函数传递数组参数给 Print()函数时，并不需要将数组元素个数传递过去，因为程序中函数之间达成了一种默契：遇到 NULL 空指针就结束数组处理。

NULL 与 void * 是不同的概念，NULL 是一个值，一个指针值，任何类型的指针都可赋予该值。而 void * 是一种类型，是一种无任何类型的指针。

8.9 命令行参数

1. 命令行参数的概念

C++程序只不过是操作系统调用的函数。

很多程序都需要从命令行输入参数。例如，在 DOS 中，拷贝命令需要两个参数，type 命令需要一个参数：

```
c>copy filea fileb
c>type c:autoexec.bat
```

操作系统将命令行参数以字符串的形式传递给 main()。因此 main()的第一行的形式应为：

```
int main(int argc, char * argv[])
```

```
{
  //...
}
```

这种形式是固定不变的，不能将参数改成其他类型。但由于数组传递的实质上是指针，所以第二个参数“char * argv[]”也可表示成“char * * argv”。

整数 argc 和字符指针数组 argv 一直都在栈中，只是以前的：

```
int main()
{
  //...
}
```

形式中，程序忽略了它们。

参数的取名是任意的，可以把 main()的输入参数改成任何喜欢的称呼：

```
int main(int count,char * str[])
```

但是，全世界都已经习惯 argc、argv 这样的称呼，所以习惯成自然了。

2. 打印命令行参数

argc 表示参数个数，argv 表示参数数组。

例如，下面的程序打印命令行参数，从中可看到命令行参数在 argc 和 argv 中的体现：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_22.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main(int argc,char * argv[])
{
  int iCount = 0;
  while(iCount<argc)
  {
   cout <<"arg " <<iCount <<": " <<argv[iCount] <<endl;
   iCount++;
  }
}
```

运行结果为：

```
C>ch8_22 aBcD eFg hIJkL
arg 0: ch8_22
arg 1: aBcD
arg 2: eFg
arg 3: hIJkL
```

再一次运行结果为：

```
C>ch8_22 c:\file1.img d:\abc\file2.img
arg 0: ch8_22
arg 1: c:|file1.img
```

```
arg 2: d:\abc\file2.img
```

参数的个数与内容决定于在程序运行时命令行的表示。argv 是字符指针数组，所以，argv[iCount]是字符指针，指向字符串。命令行参数的栈表示见图 8-14。

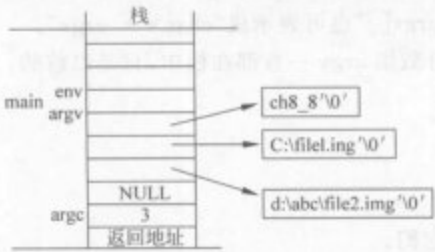

图 8-14　程序的命令行参数栈结构

知道了命令行参数的栈表示，就能更加灵活地运行命令行参数。例如，打印命令行参数的程序，方法不止一种，下面的程序与程序 ch8_21.cpp 的功能一样：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_23.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *
#include <iostream.h>

int main(int argc,char * argv[])
{
   int i = 0;
   while( * argv)
     cout <<"arg " <<i ++ <<": " << * argv ++ <<endl;
}
```

3. 命令行参数使用形式

操作系统启动程序时，总是把 3 个参数(int argc,char * argv[],char * env[])传递给 main()函数，env 用得很少，这里不作介绍。程序可以按下面 4 种方式来声明 main()函数：

```
int main()
int main(int argc)
int main(int argc,char * argv[])
int main(int argc,char * argv[],char * env[])
```

其中第二种情况是合法的，但不常见，因为在程序中很少有只用 argc，而不用 argv[]的情况。

在命令行参数中，有时某个参数含有空格，而操作系统是以空格作为区分下一个参数的标志。解决的方法是将该参数用引号括起来。

Ch8_22.cpp 的程序运行不加引号时，结果为：

```
C>ch8_22 Hello how are you
arg 0: ch8_22
```

```
arg 1: Hello
arg 2: how
arg 3: are
arg 4: you
```

Ch8_22.cpp 的程序运行加引号时，结果为：

```
C>ch8_22 "Hello how are you"
arg 0: ch8_22
arg 1: Hello how are you
```

4. main()函数的返回

一般说来，main()函数返回程序运行的状态码，例如：

```
int main()
{
  //...
  return 1;
}
```

程序将返回 1。当用户程序使用 exit()子程序终止出口时，它返回用户设定的值。例如：

```
exit(1);
```

返回状态为 1。效果与“return 1”等价。使用 exit()时，可以不论 main()的返回类型。例如：

```
void main()
{
  //...
  exit(1);
}
```

尽管 main()函数返回类型为 void，即无返回值，但仍可使用 exit()给操作系统返回一个值。任何其他的函数使用 exit()，即意味着程序终止，返回到操作系统中。

8.10 函数指针

在程序运行中，全局变量存放在 data 区，局部变量存放在栈区，动态变量存放在堆区。函数代码是程序的算法指令部分，它们同样也占有内存空间，存放在代码(code)区。每个函数都有地址。指向函数地址的指针称为函数指针。函数指针指向代码区中的某个函数，通过函数指针可以调用相应的函数。

1. 定义函数指针

函数指针的定义为：

```
int ( * func)(char a, char b);
```

int 为函数的返回类型；被括号括住的 * 和名字表示该名字是一个指针；后面的()表示该指针变量是函数类型或函数指针，这里是函数指针。该函数具有两个字符型参数 a 和 b。

定义函数指针后就产生了指针变量，它有全局、静态和局部之分。它占有内存空间。变量所占的空间大小与其他类型的指针变量一样。

由于()的优先级大于 *，所以下面是函数定义而不是函数指针定义：

```
int *func(char a, char b);
```

定义中，func 先与()结合构成函数的声明，然后再得到其返回类型为整型指针 int *。

2. 函数指针的内在差别

不含下标访问(即方括号[])的数组名是地址，**不作函数调用(即括号())的函数名也是地址**，所以可以将省略了()的函数名作为函数地址赋给函数指针。另一方面，函数是有差别的。不同的参数，不同的返回类型，构成了不同的函数(函数类型是指函数的返回类型，这里函数的差别不但是返回类型的差异，还有参数的差异，所以，函数类型的差异不能完全说明函数的差别)。

例如，下面的代码表示函数和函数指针操作的相互关系：

```
int fn1(char x, char y);          //两个字符参数和返回整型值的函数
int * fn2(char x, char y);        //两个字符参数和返回整型指针的函数
int fn3(int a);                   //一个整型参数和返回整型值的函数

int (*fp1)(char a, char b);       //两个字符参数和返回整型值的函数指针
int (*fp2)(int s);                //一个整型参数和返回整型值的函数指针

fp1 = fn1;        //ok: fn1 函数与指针 fp1 指向的函数一致
fp1 = fn2;        //error: fn2 函数与 fp1 指向的函数不一致
fp2 = fn3;        //ok: 函数参数与返回类型相一致，函数名赋给函数指针
fp2 = fp1;        //error: 两个指针指向的函数不一致
fp2 = fn3(5);     //error: 函数赋给函数指针时，不能加括号
```

由于函数的差异，导致了函数指针的差异。一个函数不能赋给一个不一致的函数指针。语句“fp1=fn2;”在 BC 编译器中会导致“可疑的指针转换”(suspicious pointer conversion)警告，而在 VC 编译器中，会导致“不能将甲函数赋给指向乙函数的指针”的编译错误。两个函数指针的相互赋值同样也要求函数一致。

函数名加上括号就变成了函数调用，所以，语句“fp2=fn3(5);”并不是一个函数地址的赋值，作为函数调用，它返回的是整型数，而 fp2 是一个函数指针，所以会引起数据类型不匹配的编译错误。

函数指针与其他数据类型的指针尽管都是地址，但在类型上有很大的差别。两类指针之间不允许互相赋值，甚至显式转换也不行。因为从意义上来说，函数指针指向程序的 code 区，是程序运行的指令代码，而数据指针指向 data 数据区、stack 栈区和 heap 堆区，是程序赖以运行的各种数据。例如：

```
int * ip;
void (*fp)();     //函数指针
fp = ip;     //error: 不能相互赋值
ip = fp;     //error: 同上
```

3. 通过函数指针来调用函数

首先必须给函数指针赋值：

```
fp2 = fn3;          //意义见上例
```

然后：

```
fp2(5); 或 (*fp2)(5);
```

第一种方式 fp2(5)是 ANSI C++标准，第二种方式(*fp2)(5)是为了兼容 C 的形式，两种方式都表示同一个函数调用的意义。常用的是第一种方式。

4. 用 typedef 来简化指针

有时，经常要定义同一种函数指针，最好将该函数指针类型用别名声明下来。以后凡是要用到该函数指针定义，就会方便许多。例如：

```
typedef int(*FUN)(int a, int b);      //声明 FUN 是一个函数指针类型
FUN funp;        //funp 为一个返回整型和两个整型参数的函数指针
```

FUN 是一个函数指针类型，该指针类型中的指针指向一个函数，它有两个整数参数，返回一个整型数。FUN 不是指针变量，只是一个指针类型名。通过第二个语句的函数指针定义，才确定一个函数指针变量 funp。

5. 函数指针用作函数参数

例如，下面的程序计算以 0.10 为步长，特定范围内的三角函数之和：

```
//*************************
//**          ch8_24.cpp          **
//*************************

#include <iostream.h>
#include <math.h>

double sigma(double(*func)(double), double dl, double du)
{
   double dt = 0.0;
   for(double d = dl; d<du; d += 0.1)
     dt += func(d);        //用函数指针调用函数
   return dt;
}

int main()
{
   double dsum;
   dsum = sigma(sin, 0.1, 1.0);      //sin 函数为实参赋给函数指针 func
   cout <<"the sum of sin from 0.1 to 1.0 is " <<dsum <<endl;
   dsum = sigma(cos, 0.5, 3.0);      //cos 函数赋给函数指针 func
   cout <<"the sum of cos from 0.5 to 3.0 is " <<dsum <<endl;
}
```

运行结果为：

```
the sum of sin from 0.1 to 1.0 is 5.01388
the sum of cos from 0.5 to 3.0 is -2.44645
```

程序中，sigma()函数的第一个参数为函数指针，该指针指向的函数有一个 double 参数并返回 double 类型数。sin 和 cos 就是这样的函数，它们作为实参赋给函数指针 func。其原型在 math.h 的头文件中声明。

又例如，标准库函数 qsort()可对任何类型的数组排序。其头文件在 stdlib.h 中。qsort()的原型为：

```
void qsort(void *, size_t nelem, size_t width,
           int(*fcmp)(const void *, const void *));
```

其中，第一个参数为待排序数组；nelem 是数组元素个数；width 是元素类型的长度；fcmp 是函数指针，比较两个参数，如果相等，则返回 0，如果参数 1 大于参数 2，则返回值为正，否则返回值为负。

现排序一个字符串数组，其比较函数用 compare()：

```
//*********************************
//*          ch8_25.cpp           *
//*********************************

#include <iostream.h>
#include <stdlib.h>
#include <string.h>

int compare(const void* a, const void* b);

char list[5][4] = {"cat","car","cab","cap","can"};

int main()
{
   qsort((void*)list,5,sizeof(list[0]),compare);

   for(int i=0; i<5; i++)
     cout <<list[i] <<endl;
}

int compare(const void* a, const void* b)
{
   return strcmp((char**)a, (char**)b);
}
```

运行结果为：

```
cab
can
cap
car
cat
```

compare()是自定义函数，本可以直接用标准库函数 strcmp()的，为了要与 qsort()函

数中的第四个参数相匹配，以使 qsort()函数调用更可读一些。

6. 函数指针可构成指针数组

例如，下面的程序例示了一个用菜单驱动函数调用的方法，各个函数调用用函数指针数组来实现：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch8_26.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

typedef void (*MenuFun)();              //声明函数指针类型
void f1(){ cout <<"good!\n";}
void f2(){ cout <<"better!\n";}
void f3(){ cout <<"best!\n";}

MenuFun fun[] = {f1,f2,f3};             //全局函数指针数组

int main()
{
   int choice;
   do{
     cout <<"1------- display good\n"
          <<"2------- dispaly better\n"
          <<"3------- dispaly best\n"
          <<"0------- exit\n"
          <<"Enter your choice: ";
     cin >>choice;
     switch(choice)
     {
       case 1: fun[0](); break;
       case 2: fun[1](); break;
       case 3: fun[2](); break;
       case 0: return;
       defaut:cout<<"you entered a wrong key.\n";
     }
   }while(choice);
}
```

7. 函数的返回类型可以是函数指针

例如：

```
typedef int (*SIG)();          //声明返回整型且无参函数的指针类型 SIG
typedef void (* SIGARG)();     //声明无返回且无参函数的指针类型
SIG signal(int, SIGARG);       //声明返回函数指针的函数
```

最后一行声明一个函数，该函数返回一个函数指针，且有一个整型参数和一个函数指针参数。返回的函数指针是指向返回整型且无参数的函数。作为函数指针参数的指针指向无返回且无参数的函数。

小结

指针不仅仅是地址，还有对数据类型的操作性规定。这是理解指针的关键。

当程序把一个数组传递给函数时，C++实际上把数组第一个元素的地址传递给该函数。通过递增指针的值，可以使指针直接指向数组的下一个元素。

对字符串操作的函数一般用指针遍历字符串，直至指针指向 NULL。当指针指向除字符串以外的其他数组时，函数要知道数组中元素的个数，或者数组中的最后一个元素。

堆允许程序在运行时，而不是在编译时，确定所申请的内存之大小。在 C++中，堆分配一般用 new 和 delete 两个操作符，malloc()和 free()函数则相对过时，只在阅读用 C 编制的程序时，才需要这些知识。在第 14 章，将进一步讨论 new 和 delete 的使用。

指针使函数的功能大大增强，使之可以访问局部栈以外的内存空间。但也随之带来了副作用，函数的黑盒性受到威胁。为了限制副作用，可以声明函数形参为指针常量和指向常量的指针。

函数指针指向程序的代码区，通过函数指针可以调用函数。程序设计深入下去的时候，必然要触及函数指针，函数指针使程序更加简洁和强大。

指针数组是 C++程序中最有用的结构之一，指针数组和二维数组是有区别的。

命令行参数是让程序员与操作系统之间发生关系，main()是操作系统调用的一个函数。main()从操作系统获取参数值。命令行参数也是指针数组的一个应用。

练习

8.1 下面的程序中，调用了 findmax()函数，该函数寻找数组中的最大元素，将该元素的下标通过参数返回，并返回其地址值，编程实现 findmax()函数。

```
#include <iostream.h>

int * findmax(int * array, int size, int * index);

int main()
{
   int a[10] = {33,91,54,67,82,37,85,63,19,68};
   int * maxaddr;
   int idx;

   maxaddr = findmax(a, sizeof(a)/sizeof( * a), &idx);

   cout <<"the index of maximum element is " <<idx <<endl
        <<"the address of it is " <<maxaddr <<endl
        <<"the value of it is " <<a[idx] <<endl;
}
```

8.2 编制程序，改进 Josephus 问题的解(josel.cpp)，使之在运行时确定小孩数(用 new 和 delete 操作符)，并进行输入检查：小孩数不能小于 1，数小孩的间隔数不能小于 1，也不能大于小孩数。发现输入错误应让其选择：停止运行，重输，以默认值 10 个小孩和

数小孩间隔 3 让其运行。

8.3 编制程序，使用标准库函数 qsort()，对各类数组进行排序。

(1) 对整数数组进行排序。比较是以一个整数的各位数字之和的大小为依据，从小到大排列。数组中元素值为：

```
12, 32, 42, 51, 8, 16, 51, 21, 19, 9
```

(2) 对浮点数组进行排序。从小到大排列。数组中元素为：

```
32.1, 456.87, 332.67, 442.0, 98.12,
451.79, 340.12, 54.55, 99.87, 72.5
```

(3) 对字符串数组进行排序。比较是以各字符串的长度为依据，如果长度相等，再比较字符串的值，从小到大排列。数组中的元素为：

```
enter, number, size, begin, of, cat, case,
program, certain, a
```

8.4 编制程序，将输入的一行字符加密和解密。加密时，每个字符依次反复加上“4962873”中的数字，如果范围超过 ASCII 码的 032(空格)～122('z')，则进行模运算。解密与加密的顺序相反。编制加密和解密函数，打印各个过程的结果。

例如，加密：the result of 3 and 2 is not 8

```
(t)116 + 4,(h)104 + 9,(e)101 + 6,( )32 + 2,(r)114 + 8,(e)101 + 7,(s)115 + 3,
(u)117 + 4,(l)108 + 9,(t)116 + 6,(o)111 + 2,(f)102 + 8,( )32 + 7,(3)51 + 3,
( )32 + 4,(a)97 + 9,(n)110 + 6,(d)100 + 2,( )32 + 8,(2)50 + 7,( )32 + 3,
(i)105 + 4,(s)115 + 9,( )32 + 6,(n)110 + 2,(o)111 + 8,(t)116 + 7,( )32 + 3,
(8)56 + 4
```

得到密文为：

```
xqk"zlvyuz"wm#7>gpl's$rg" vvw $ A
```

8.5 编写程序，实现两个字符串比较的自定义版：

```
int strcmp(const char * str1,const char * str2);

//当 str1>str2 时，返回正数；
//当 str1 = = str2 时，返回 0；
//当 str1<str2 时，返回负数。
```

8.6 编写程序，实现复制字符串的自定义版：

```
char * strcpy(char * dest, const char * source);

//该函数返回 dest 的值，即字符串首地址
```

8.7 编写程序，输入命令行参数为两个字符串，用练习 8.5 的 strcmp()比较并输出比较结果。

8.8 编写程序。

(1) 初始化一个矩阵 A(5×5)，元素值取自随机函数，并输出；

(2) 将其传递给函数，实现矩阵转置；

(3) 在主函数中输出结果。

随机函数的原型为：

```
int rand();
```

它产生一个 0～65535 中的随机数(16 位机器中)。

8.9 编写程序。测试堆内存的容量：每次申请一个数组，内含 100 个整数，直到分配失败，并打印堆容量报告。

第9章 引　　用

程序设计语言的进化使用户从被迫解决细节问题中解脱出来，转向允许用户花更多时间来考虑“大的蓝图”。根据这种精神，C++中包含了一个称为引用的特性，它允许程序来负责确定把参数传递给函数的方法。学习了本章后，应该掌握引用的语法，用引用传递函数的方法，理解C++在函数原型中声明引用的目的，正确使用引用，避免不恰当的引用返回，明辨引用与指针的区别。

9.1 引用的概念

引用是个别名，当建立引用时，程序用另一个变量或对象(目标)的名字初始化它。从那时起，引用作为目标的别名而使用，对引用的改动实际就是对目标的改动。

为建立引用，先写上目标的类型，后跟引用运算符“&”，然后是引用的名字。引用能使用任何合法变量名。

例如，引用一个整型变量：

```
int someInt;
int& rInt = someInt;
```

声明rInt是对整数的引用，初始化为引用someInt。在这里，要求someInt已经有声明或定义，而引用仅仅是它的别名，不能喧宾夺主。

引用不是值，不占存储空间，声明引用时，目标的存储状态不会改变。所以，既然定义的概念有具体分配空间的含义，那么引用只有声明，没有定义。

例如，下面的程序建立和使用引用：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch9_1.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int main()
{
   int intOne;
   int& rInt = intOne;

   intOne = 5;
   cout <<"intOne:" <<intOne <<endl;
   cout <<"rInt:" <<rInt <<endl;

   rInt = 7;
   cout <<"intOne:" <<intOne <<endl;
   cout <<"rInt:" <<rInt <<endl;
}
```

运行结果为：

```
intOne:5
rInt:5
intOne:7
rInt:7
```

引用 rInt 用 intOne 来初始化。以后，无论改变 intOne 还是 rInt，实际上都是指 intOne，两者的值都一样。

引用在声明时必须被初始化，否则会产生编译错误。

→ 引用运算符与地址操作符使用相同的符号。尽管它们显然是彼此相关的，但它们不一样。

引用运算符只在声明的时候使用，它放在类型名后面，例如：

```
int& rInt = intOne;
```

任何其他“&”的使用都是地址操作符，例如：

```
int * ip = &intOne;
cout <<&ip;
```

→ 与指针类似，下面三种声明引用的方法都是合法的：

```
int& rInt;
int &rInt;
int & rInt;
```

下面的语句包含一个引用的声明和一个变量的定义：

```
int& rInt, sa;        //会误以为声明了两个引用
```

为了提高可读性，不应在同一行上同时声明引用、指针和变量。

9.2 引用的操作

如果程序寻找引用的地址，它只能找到所引用的目标的地址。

例如，下面的程序取引用的地址：

```
//* * * * * * * * * * * * * * * * * * * * * *
//* *              ch9_2.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void main()
{
   int intOne;
   int& rInt = intOne;

   intOne = 5;
   cout <<"intOne:" <<intOne <<endl;
   cout <<"rInt:" <<rInt <<endl;
```

```
    cout <<"&intOne:" <<&intOne <<endl;
    cout <<"&rInt:" <<&rInt <<endl;
}
```

运行结果为：

```
intOne:5
rInt:5
&intOne:00F3:5300
&rInt:00F3:5300
```

C++没有提供访问引用本身地址的方法，因为它与指针或其他变量的地址不同，它没有任何意义。引用在建立时就初始化，而且总是作为目标的别名使用，即使在应用地址操作符时也是如此。对引用的理解可以见图 9-1。

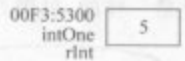

图 9-1　定义 rInt 引用与变量的关系

引用一旦初始化，它就维系在一定的目标上，再也不分开。任何对该引用的赋值，都是对引用所维系的目标赋值，而不是将引用维系到另一个目标上。

例如，下面的程序给引用赋新值：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                ch9_3.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

void main()
{
    int intOne;
    int& rInt = intOne;

    intOne = 5;
    cout <<"intOne:" <<intOne <<endl;
    cout <<"rInt:" <<rInt <<endl;

    cout <<"&intOne:" <<&intOne <<endl;
    cout <<"&rInt:" <<&rInt <<endl;

    int intTwo = 8;
    rInt = intTwo;
    cout <<"intOne:" <<intOne <<endl;
    cout <<"intTwo:" <<intTwo <<endl;
    cout <<"rInt:" <<rInt <<endl;

    cout <<"&intOne:" <<&intOne <<endl;
    cout <<"&intTwo:" <<&intTwo <<endl;
    cout <<"&rInt:" <<&rInt <<endl;
}
```

运行结果为：

```
intOne:5
rInt:5
&intOne:0110:F150
&rInt:0110:F150
intOne:8
intTwo:8
rInt:8
&intOne:0110:F150
&intTwo:0110:F14E
&rInt:0110:F150
```

在程序中，引用 rInt 被重新赋值为变量 intTwo。从运行结果看出，rInt 仍然维系在原 intOne 上，因为 rInt 与 intOne 的地址是一样的，见图 9-2。

图 9-2　引用被赋值的意义

```
rInt = intTwo;      等价于      intOne = intTwo;
```

引用与指针有很大的差别，指针是个变量，可以把它再赋值成指向别处的地址，然而，建立引用时必须进行初始化并且绝不会再指向其他不同的变量。

9.3　什么能被引用

若一个变量声明为 T&，即引用时，它必须用 T 类型的变量或对象，或能够转换成 T 类型的对象进行初始化。

如果引用类型 T 的初始值不是一个左值，那么将建立一个 T 类型的目标并用初始值初始化，那个目标的地址变成引用的值。

例如，下面的代码是合法的：

```
double& rr = 1;
```

在这种情况下：

(1) 首先作必要的类型转换；

(2) 然后将结果置于临时变量；

(3) 最后，把临时变量的地址作为初始化的值。

所以上面的语句解释为：

```
double temp;
temp = double(1);
double& rr = temp;
```

上面的语句中将临时变量显式地表示出来，事实上，临时变量并不在存放局部变量的栈

区。由于指针也是变量，所以可以有指针变量的引用：

```
int * a;
int * &p = a;     //表示 int * 的引用 p 初始化为 a
int b = 8;
p = &b;           //ok! p 是 a 的别名，是一个指针
```

指针变量的引用，见图 9-3。

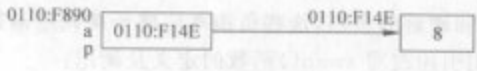

图 9-3　指针变量的引用

对 void 进行引用是不允许的。例如：

```
void& a = 3;     //error
```

void 只是在语法上相当于一个类型，本质上不是类型，没有任何一个变量或对象，其类型为 void。

不能建立引用的数组：

```
int a[10];
int& ra[10] = a;     //error
```

因为数组是某个数据类型元素的集合，数组名表示该元素集合空间的起始地址，它自己不是一个名副其实的数据类型。

引用本身不是一种数据类型，所以没有引用的引用，也没有引用的指针。例如：

```
int a;
int& ra = a;
int& * p = &ra;     //error 企图定义一个引用的指针
```

引用不是类型，定义引用在概念上不产生实体，所以，在引用之上的引用不存在，而且，引用的指针指向谁呢？

引用不能用类型来初始化：

```
int& ra = int;     //error
```

因为引用是变量或对象的引用，而不是类型的引用。

有空指针，无空引用。不应有下面的引用声明，否则会有运行错误：

```
int& ri = NULL;     //毫无意义
```

9.4　用引用传递函数参数

1. 引用传递参数

传递引用给函数与传递指针的效果一样，传递的是原来的变量或对象，而不是在函数作用域内建立变量或对象的副本。

在 8.6 节中，我们看到对 swap(int,int)传值方式函数的调用不影响调用函数中的实参，结果并未达到交换数据的预想目的。

使用指针传递方式的 swap(int *,int *)函数的调用,能够达到预定的目的(见 8.6 节),但是函数的语法相对传值方式来说比较累赘。首先,在 swap()函数内需要重复传递引用(dereference)(*px),这容易产生错误且难以阅读。其次,调用函数需要传递变量地址,使 swap()内部的工作对用户太过显然。而且还有 swap(&x,&y)的形式会造成一种交换两个变量地址的错觉。

C++的目标之一就是让使用函数的用户无须考虑函数是如何工作的。传递指针给使用函数的用户增加了编程和理解的负担,这些负担本应属于被调用函数。

例如,下面的程序用引用改写 swap()函数的定义及调用:

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch9_4.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

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

int main()
{
   int x = 5;
   int y = 6;
   cout <<"before swap, x:" <<x <<" ,y:" <<y <<endl;

   swap(x,y);

   cout <<"after swap, x:" <<x <<" ,y:" <<y <<endl;
}

 void swap(int &  rx,int &  ry)
{
   int temp;
   temp = rx;
   rx = ry;
   ry = temp;
}
```

运行结果为:

```
before swap, x:5 ,y:6
after swap, x:6 ,y:5
```

在主函数中,调用 swap()函数的参数是 x 和 y,简单地传递变量而不是它们的地址。而事实上,传递的是它们的地址。引用传递的内存布局与指针相仿,只是操作完全不同。每当使用引用时,C++就去求该引用所含地址中的变量值,见图 9-4。

引用具有指针的威力,但是调用引用传递的函数时,可读性却比指针传递好。引用具有传值方式函数调用语法的简单性与可读性,但是威力却比传值方式强。

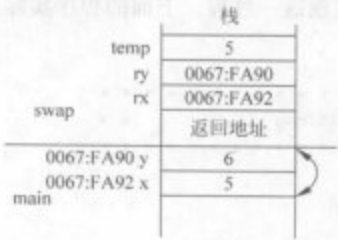

图 9-4　传递引用的内存布局

2. 引用存在的问题

尽管引用可以表达清晰并让程序员负责了解如何传递参数，但是在有些情况下它们能隐藏错误。

例如，下面的代码在没有看到函数原型之前可能会误认为实参 a 和 b 是通过值来传递的，从而不能通过函数调用来修改它，而事实上却能够修改：

```
int a = 10;
int b = 20;
swap(a,b);
```

因为引用隐藏了函数所使用的参数传递的类型，所以无法从所看到的函数调用判断其是值传递还是引用传递。正因为此，下面的代码中两个重载函数将引起编译报错：

```
void fn(int a)
{
   //...
}

void fn(int& t)
{
   //...
}

int main()
{
   int a = 5;
   fn(a);      //error 匹配哪一个函数?
}
```

9.5　返回多个值

函数只能返回一个值。如果程序需要从函数返回两个值怎么办？解决这一问题的办法之一是用引用给函数传递两个参数，然后由函数往目标中填入正确的值。因为用引用传递允许函数改变原来的目标，这一方法实际上让函数返回两个信息。这一策略绕过了函数的返回值，使得可以把返回值保留给函数，作报告运行成败或错误原因用。

引用和指针都可以用来实现这一过程。下面的程序实际上返回了三个值，两个是引用，另一个是函数返回值：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch9_5.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

int Factor(int, int&, int&);

int main()
{
   int number,squared,cubed,error;
   cout <<"Enter a number(0 ~20): ";
   cin >>number;

   error = Factor(number,squared,cubed);

   if(error)
    cout <<"Error encountered!\n";
   else
   {
    cout <<"Number: " <<number <<endl;
    cout <<"Squared: " <<squared <<endl;
    cout <<"Cubed: " <<cubed <<endl;
   }
}

int Factor(int n, int& rSquared, int& rCubed)
{
   if(n>20 && n<0)
    return 1;
   rSquared = n * n;
   rCubed = n * n * n;
   return 0;
}
```

运行结果为：

```
Enter a number(0~20): 3
Number: 3
Squared: 9
Cubed: 27
```

Factor()函数检查用值传递的第一参数。如果不在0～20的范围内，它就简单地返回错误值(假设程序正常返回为0)。程序所真正需要的值squared和cubed是通过改变传递给函数的引用返回的，而没有使用函数返回值。

9.6 用引用返回值

函数返回值时，要生成一个值的副本。而用引用返回值时，不生成值的副本。

例如，下面的程序是有关引用返回的4种形式：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch9_6.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

float temp;

float fn1(float r)
{
   temp = r * r * 3.14;
   return temp;
}

float& fn2(float r)
{
   temp = r * r * 3.14;
   return temp;
}

int main()
{
   float a = fn1(5.0);       //1
   float& b = fn1(5.0);      //2:warning
   float c = fn2(5.0);       //3
   float& d = fn2(5.0);      //4
   cout<<a<<endl;
   cout<<b<<endl;
   cout<<c<<endl;
   cout<<d<<endl;
}
```

运行结果为：

```
78.5
78.5
78.5
78.5
```

对主函数的4种引用返回的形式，程序的运行结果是一样的。但是它们在内存中的活动情况是各不相同的。其中变量temp是全局数据，驻留在全局数据区data。函数main()、函数fn1()或函数fn2()驻留在栈区stack。

第一种情况：见图9-5。

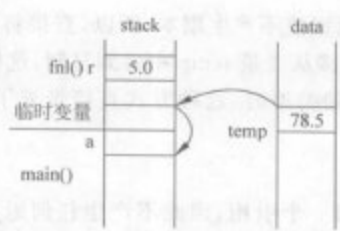

图9-5　返回值方式的内存布局

这种情况是一般的函数返回值方式。返回全局变量 temp 值时，C++创建临时变量并将 temp 的值 78.5 复制给该临时变量。返回到主函数后，赋值语句 a=fn1(5.0)把临时变量的值 78.5 复制给 a。

第二种情况：见图 9-6。

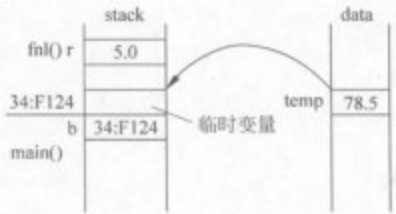

图 9-6　返回值初始引用的情形

这种情况下，函数 fn1()是以值方式返回的，返回时，复制 temp 的值给临时变量。返回到主函数后，引用 b 以该临时变量来初始化，使得 b 成为该临时变量的别名。由于临时变量的作用域短暂，所以 b 面临无效的危险。根据 C++标准，临时变量或对象的生命期在一个完整的语句表达式结束后便宣告结束，也即在“float& b=fn1(5.0);”之后，临时变量不再存在。所以引用 b 以后的值是个无法确定的值。BC 对 C++标准进行了扩展，规定如果临时变量或对象作为引用的初始化时，则其生命期与该引用一致。14.7 节将进一步介绍这一内容。这样的程序，依赖于编译器的具体实现，所以移植性是差的。

若要以返回值初始化一个引用，应该先创建一个变量，将函数返回值赋给这个变量，然后再以该变量来初始化引用，就像下面这样：

```
int x = fn1(5.0);
int& b = x;
```

第三种情况：见图 9-7。

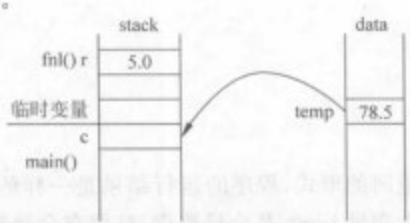

图 9-7　返回引用方式

这种情况，函数 fn2()的返回值不产生副本，所以，直接将变量 temp 返回给主函数。主函数的赋值语句中的左值 c 直接从变量 temp 中得到复制，这样避免了临时变量的产生。当变量 temp 是一个用户自定义的类型时，这种方式直接带来了程序执行效率和空间利用的利益。

第四种情况：见图 9-8。

这种情况，函数 fn2()返回一个引用，因此不产生任何返回值的副本。在主函数中，一个引用声明 d 用该返回值来初始化，使得 d 成为 temp 的别名。由于 temp 是全局变量，所

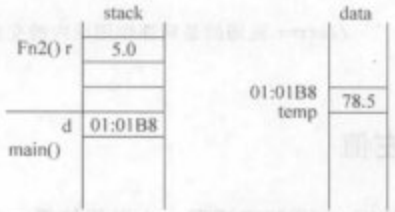

图 9-8　返回引用方式的值作为引用的初始化

以在 d 的有效期内 temp 始终保持有效。这样做法是安全的。

但是，如果返回不在作用域范围内的变量或对象的引用，那就有问题了。这与返回一个局部作用域指针的性质一样严重。BC 作为编译错误，VC 作为警告，来提请编程者注意。

例如，下面的代码返回一个引用，来给主函数的引用声明初始化：

```
float& fn2(float r)
{
   float temp;
   temp = r * r * 3.14;
   return temp;
}

 int main()
{
   float& d = fn2(5.0);     //error 返回的引用是个局部变量
}
```

见图 9-9 说明。

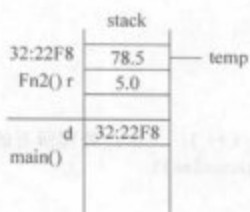

图 9-9　返回的引用是局部变量

如果返回的引用是作为一个左值进行运算，也是程序员最犯忌的。所以，如果程序中有下面的代码，则一定要剔除：

```
float& fn2(float r)
{
   float temp;
   temp = r * r * 3.14;
   return temp;
}

 int main()
```

```
{
   fn2(5.0) = 12.4;        //error 返回的是局部作用域内的变量
}
```

9.7 函数调用作为左值

在上节中，对于第三种情况，也意味着**返回一个引用使得一个函数调用表达式成为左值表达式**。只要避免将局部栈中变量的地址返回，就能使函数调用表达式作为左值来使用运行得很好。

例如，下面的程序是统计学生中 A 类学生与 B 类学生各占多少。A 类学生的标准是平均分在 80 分以上，其余都是 B 类学生，先看不返回引用的情况：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *                ch9_7.cpp                * *
//* * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>

int array[6][4] = {{60,80,90,75},
                   {75,85,65,77},
                   {80,88,90,98},
                   {89,100,78,81},
                   {62,68,69,75},
                   {85,85,77,91} };

int getLevel(int grade[], int size);

int main()
{
     int typeA = 0,typeB = 0;
     int student = 6;
     int gradesize = 4;

   for(int i = 0; i<student; i ++ )        //处理所有的学生
    if(getLevel(array[i], gradesize))
       typeA ++ ;
    else
       typeB ++ ;

   cout <<"number of type A is " <<typeA <<endl;
   cout <<"number of type B is " <<typeB <<endl;
  }

int getLevel(int grade[], int size)
{
   int sum = 0;
   for(int i = 0; i<size; i ++ )     //成绩总分
    sum += grade[i];

   sum/= size;        //平均分
```

```
    if(sum>=80)
      return 1;        //type A student
    else
      return 0;        //type B student
}
```

运行结果为：

```
number of type A is 3
number of type B is 3
```

该程序通过函数调用判明该学生成绩属于A类还是B类，然后给A类学生人数增量或给B类学生人数增量。

该程序也可以通过返回引用来实现。返回的引用作为左值直接增量。例如：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch9_8.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

  int array[6][4]={{60,80,90,75},
                   {75,85,65,77},
                   {80,88,90,98},
                   {89,100,78,81},
                   {62,68,69,75},
                   {85,85,77,91}};

int& level(int grade[], int size, int& tA, int& tB);

int main()
{
   int typeA=0,typeB=0;
   int student=6;
   int gradesize=4;

   for(int i=0; i<student; i++)        //处理所有的学生
    level(array[i], gradesize, typeA, typeB)++;     //函数调用作为左值

   cout<<"number of type A is "<<typeA<<endl;
   cout<<"number of type B is "<<typeB<<endl;
}

int& level(int grade[], int size,int& tA, int& tB)
{
   int sum=0;
   for(int i=0; i<size; i++)     //成绩总分
    sum+=grade[i];

   sum/=size;        //平均分

   if(sum>=80)
    return tA;     //type A student
   else
```

```
    return tB;      //type B student
}
```

该程序中的 level()函数返回一个引用，为了返回一个非局部变量的引用，就要传递两个引用参数 typeA 和 typeB。当该学生属于 A 类时，就返回 typeA 的引用，否则就返回 typeB 的引用。

由于返回的是引用，所以可以作为左值直接进行增量操作。该函数调用代表 typeA 还是 typeB 的左值视具体的学生成绩统计结果而定。

本例说明：返回引用的函数，可以使函数成为左值。在后面章节中，我们将会看到，这一应用是很多的，最典型的是 cout 和 cin 的操作符重载。

→ 上面各个图中，对引用的表示依赖于实现。引用变量概念上是不占空间的，引用变量被理解为粘附在初始化的实体上，它的实现对用户来说不可见。但并不等于具体实现的时候，非得不占任何空间。为了帮助理解引用，让你有个引用实现的感性认识，表示了一种实现的方案，图中引用空间用来存放所代表变量的地址。

9.8 用 const 限定引用

传递指针和引用更大的目的是效率。当一个数据类型很大(后面章节中介绍的自定义类型)时，因为传值要复制副本，所以不可取。

另一方面，传递指针和引用存在传值所没有的危险。程序有时候不允许传递的指针所指向的值被修改或者传递的引用被修改，但传递的地址特征使得所传的参数处于随时被修改的危险之中。

保护实参不被修改的办法是传递 const 指针和引用。

例如，下面的程序传递一个 const double 型的常量指针，返回一个指针：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch9_9.cpp                  * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

double* fn(const double* pd)
{
   static double ad = 32;
   ad += *pd;
   cout <<"fn being called...the value is: " << *pd <<endl;
   return &ad;
}

int main()
{
   double a = 345.6;
   const double* pa = fn(&a);
   cout << *pa <<endl;
   a = 55.5;
   pa = fn(&a);
   cout << *pa <<endl;
}
```

运行结果为：

```
fn being called...the value is: 345.6
377.6
fn being called...the value is: 55.5
433.1
```

程序中 fn()函数声明的参数为 double 型的常量指针，返回 double 型的指针。函数 fn()中，没有生成实参 a 的副本，访问 * pd 就是直接访问 a。尽管 a 是变量，但由于限定了 pd 的性质，所以在 fn()函数的作用域范围内，pd 只能以 * pd 的形式读出 a，而不能修改 a。

函数 fn()中，定义了一个静态局部变量 ad。在返回时，将其地址返回给了主函数中的常量 double 的指针 pa，使之通过 * pa 能读出 ad 的值而不能修改之。但在函数 fn()中，ad 是变量，是可以被修改的。

将上面的程序改成传递引用与返回引用，函数处理起来会更容易，可读性也更好：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *                ch9_10.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

double& fn(const double& pd)
{
   static double ad = 32;
   ad += pd;
   cout <<"fn being called...the value is: " <<pd <<endl;
   return ad;
}

int main()
{
   double a = 345.6;
   double& pa = fn(a);
   cout <<pa <<endl;
   a = 55.5;
   pa = fn(a);
   cout <<pa <<endl;
}
```

运行结果为：

```
fn being called...the value is: 345.6
377.6
fn being called...the value is: 55.5
433.1
```

程序 ch9_10.cpp 与 ch9_9.cpp 的输出是一样的。唯一明显的区别是现在的函数 fn()以 const double 的引用为参数，返回 double 的引用。用引用比用指针更简单些，而且程序达到相同的效率，也具有使用 const 所提供的安全性。

→ C++不区分变量的 const 引用和 const 变量的引用。程序绝不能给引用本身重新赋值，使它指向另一个变量，因此引用总是 const 的。如果对引用应用关键词 const，其作用就

是使目标成为 const 变量。即没有：

```
const double const& a = 1;
```

只有：

```
const double& a = 1;
```

9.9 返回堆中变量的引用

对引用的初始化，可以是变量，可以是常量，也可以是一定类型的堆空间变量。但是，由于引用不是指针，所以，下面的代码直接从堆中获得的变量空间来初始化引用是错的：

```
int& a = new int(2);     //a 不是指针
```

考虑操作符 new。如果 new 不能在堆空间成功地获得内存分配，它返回 NULL。因为引用不能是 NULL，在程序确认它不是 NULL 之前，程序不能用这一内存初始化引用。

例如，下面的代码说明如何处理这一校验：

```
# include <iostream.h>

void fn()
{
   int * pInt = new int;
   if(pInt == NULL)
   {
     cout <<"error memory allocation!";
     return 1;
   }
   int& rInt = * pInt;
   //...
}
```

int 指针 pInt 获得 new 返回的值，程序测试 pInt 中的地址，如果它是 NULL，则报告错误信息并返回。如果它不是 NULL，则将 * pInt 初始化引用 rInt。如此，rInt 成为 new 返回的 int 的别名。

用堆空间来初始化引用，要求该引用在适当时候释放堆空间。

例如，下面的程序在堆中分配空间，求值，然后释放堆空间：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch9_11.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>

int CircleArea()
{
   double * pd = new double;
   if(!pd)
   {
     cout <<"error memory allocation!";
```

```
        return 1;
    }
    double& rd = * pd;
    cout <<"the radius is: ";
    cin >>rd;
    cout <<"the area of circle is " <<rd * rd * 3.14 <<endl;
    delete &rd;
    return 0;
}

int main()
{
    if(CircleArea())
     cout <<"program failed.\n";
    else
     cout <<"program successed.\n";
}
```

运行结果为：

```
the radius is: 12
the area of circle is 452.16
program successed.
```

在计算圆面积的CircleArea()中，double指针接受new返回的堆空间地址，然后进行有效性校验。如果有效，则将 * pd 初始化引用 rd，rd 接受键盘输入，计算，打印输出圆面积，返还堆空间，并正常返回。否则输出错误信息，返回出错标志。

这里，返还堆空间有两种方式，一个是 delete pd，另一个是 delete &rd，因为 &rd 和 pd 都指向同一个堆空间地址。对引用来说，同样存在由一个函数建立的堆内存由另一个函数释放的问题，我们在第 8 章有过类似的说明。

对使用堆的引用，有下面的经验：

- 必要时用值传递参数
- 必要时返回值
- 不要返回有可能退出作用域的引用
- 不要引用空目标

→ 引用和指针使函数的“黑盒”性被打破。函数可以访问不属于自己栈空间的内存。这对把握不住C++的人来说是危险的，而对熟练的程序员来说，正是引用和指针才使函数只能返回单一值的状态被打破，使得函数功能更趋强大。函数的副作用是良性还是恶性，各人自有评说，对于函数潜在的破坏性，是放任自流，还是要适当地抑制，专家们动尽了脑筋，直到C++类机制的实现，才使函数的恶性作用得以控制。

小结

引用是C++独有的特性。指针存在有种种问题，间接引用指针会使代码可读性差，易编程出错。而引用正好扬弃了指针。引用的使用中，单纯取个别名是毫无意义的，引用的目的主要用于在函数参数传递中，解决大对象的传递效率和空间都不如意的问题。本章主要介

绍引用的原理和各种语法现象，未涉及到大对象，它在以后各章陆续介绍。引用能够保证参数传递中不产生副本，从而发挥指针的威力，提高传递的效率，通过 const 的使用，保证了引用传递的安全性。

引用是 C++语言学习中的一个难点。有的人 C 的背景知识不是很强，他们经常一开始不管在什么地方都使用引用，除非有的地方必须使用指针。而另一些学习 C++的 C 程序员则通常避免使用引用，只是把它们作为另一种传递地址的方法来考虑。由于指针也能做这项工作，所以他们只使用指针。这都是片面的。

引用具有表达清晰的优点。引用将对传递的参数的责任附给了编写函数的程序员，而不是使用它们的各个用户。引用是对操作符重载必不可少的补充(见 14.7 节)。引用通过传递地址提高了函数运行的效率。引用传递与值传递在使用方法上比较，唯一的区别在函数的形式参数声明。

不允许声明引用数组，可以用常量来初始化引用声明。返回引用时，要注意局部对象返回的危险。要注意引用隐藏函数所使用的参数传递的类型。

练习

9.1 读下列程序。

(1) 将其改写为传递引用参数；

(2) 说出其功能；

(3) 将 findmax()函数改写成非递归函数(重新考虑参数个数)。

```
#include <iostream.h>
const size = 10;

void findmax(int * a, int n, int i, int * pk);

int main()
{
   int a[size];
   int n = 0;
   cout <<"please input " <<size <<"datas:\n";
   for(int i = 0; i<size; i ++ )
   {
     cin >>a[i];
   }

   findmax(a, size, 0, &n);

   cout <<"the maximum is " <<a[n] <<endl
        <<"It's index is " <<n <<endl;
}

void findmax(int * a,int n,int i,int * pk)
{
   if(i<n)
   {
```

```
        if(a[i]>a[ * pk])
          * pk = i;
        findmax(a,n,i + 1,&( * pk));
      }
  }
```

9.2 读下列程序，该程序生成有十个整数的安全数组。要把值放入数组中，使用 put()函数，然后取出该值，使用 get()函数，put()和 get()中若遇下标越界立即终止程序运行。其运行结果为后面所示，请完成两个未写出的函数定义。

```
# include <iostream.h>

int& put(int n);      //put value into the array
int get(int n);       //obtain a value from the array

int vals[10];
int error = -1;

int main()
{
   put(0) = 10;     //put values into the array
   put(1) = 20;
   put(9) = 30;

   cout <<get(0) <<endl;
   cout <<get(1) <<endl;
   cout <<get(9) <<endl;

   put(12) = 1;       //out of range

}

//...
```

运行结果为：

```
10
20
30
range error in put() value!
```

9.3 编制程序，调用传递引用的参数，实现两个字符串变量的交换。例如开始：

```
char * ap = "hello";
char * bp = "how are you?";
```

交换的结果使得 ap 和 bp 指向的内容分别为：

```
ap:    "how are you?"
bp:    "hello"
```

第10章 结 构

到目前为止，所见到的数据类型都只包含一种类型信息，即使是多个元素的数组。但是，信息的逻辑关系，要求各个数据类型组合在一起考虑会更加方便。结构就能表达这种数据聚集。另外，结构是实现链表结构的理想表现手段。学习本章后，应能掌握结构声明、结构变量定义与访问结构成员的方法，掌握结构作为参数传递与返回结构的函数方法，掌握链表结构的各项基本操作。

10.1 结构概述

1. 为什么要用结构

C++用数组存储许多相同类型和意义的相关信息，但是有些数据信息是由若干不同数据类型和不同意义的数据所组成。例如，一个人事记录包括姓名、职工编号、工资、地址、电话等，这些数据信息的类型是不一样的，不能用数组的形式把它们组织起来。用结构变量就可有组织地把这些不同类型的数据信息存放在一起。否则程序需要若干个不同数据类型的变量分别存储职工的信息，不便于程序管理。

例如，定义了一个职工的若干数据后，在函数参数传递时感到麻烦，在返回一个职工信息时遇到了困难：

```
void fn(char * ,long,float,char * ,char * )
{
   //处理职工数据
   //...
   //不知如何返回这5个数据?
}

int main()
{
   char name[20];     //定义一个职工要下列5个变量定义语句
   long code;
   float salary;
   char address[50];
   char phone[11];
   //...

   fn(name,cade,salary,address,phone);     //调用时要传递5个变量
   //如何得到fn()处理后返回的5个变量呢
}
```

2. 结构的概念

结构是用户自定义的新数据类型，除此之外，它可与 int，float 等基本数据类型同等看

待。声明结构类型时，首先指定关键字 struct 和结构名，然后用一对花括号将若干个结构成员数据类型说明括起来。

通常情况下，结构声明在所有函数之外，位于 main()函数之前。这使新声明的数据类型在程序的任何地方都可以被使用。

例如，声明一个职工 Employee 结构数据类型，它包括姓名、职工编号、工资、地址、电话。用一个结构数据类型的变量可以存放所有这些相关的信息：

```
struct Employee     //名为 Employee 的结构声明
{
   char name[20];
   long code;
   float salary;
   char address[50];
   char phone[11];
};          //分号是必需的

int main()
{
   Employee person;     //定义一个 Employee 结构的变量，分配变量空间
   //使用这个结构变量
}
```

声明一个结构并不分配内存，内存分配发生在定义这个新数据类型的变量中。

结构中包含的数据变量称为该结构的成员。如 code，salary 是结构 Employee 的成员。

3. 访问结构成员

一旦通过定义相应结构变量，分配了空间，就可以使用点操作符“.”(或称结构成员操作符)来访问结构中的成员。左操作数为结构类型变量，右操作数为结构中的成员。点操作符运算的结果可以是左值，也可以是右值(见 3.1 节)。

在数组中，我们称数组分量为元素，在结构中，我们称结构分量为成员。数组的[]运算符与结构的点运算符具有相同的运算优先级，它们是所有运算符中，优先级最高的。

例如，下面的程序说明了访问结构成员的方法：

```
//* * * * * * * * * * * * * * * * * * * * * * * *
//* *              ch10_1.cpp                   * *
//* * * * * * * * * * * * * * * * * * * * * * * *

# include <iostream.h>

struct Weather
{
   double temp;     //温度
   double wind;     //风力
};

int main()
{
   Weather today;
```

```
    today.temp = 30.5;    //作为左值使用
    today.wind = 10.1;

    cout <<"Temp = " <<today.temp <<endl;    //作为右值使用
    cout <<"Wind = " <<today.wind <<endl;
}
```

运行结果为：

```
Temp = 30.5
Wind = 10.1
```

程序中 today.temp 被赋值为 30.5，today.wind 被赋值为 10.1。在点操作符的左右两边都没有空格，这不是必需的，只是一般的程序书写风格。即你也可以写成下面这样：

```
today  .  wind = 10.1;
```

在 Weather 结构中，含有两个 double 类型的成员 temp 与 wind，所有成员的数据类型相同，为什么不用数组来实现呢？即为什么不写成：

```
double Weather[2] = {30.5, 10.1};
```

因为我们用结构来实现的目的，就是要区分一个数据聚集中的每个分量的类型和意义，每个分量数据类型可以相异，更重要的是，为了区分数据内容的意义，结构的成员有自己单独的名字，也就能区分每个成员各自的意义。

4. 给结构赋值

在数组中，数组是不能彼此赋值的。

例如，下面数组的赋值语句会导致一个编译错误：

```
void f()
{
   char a[10],b[10];
   a = b;     //error
   //...
}
```

这主要是因为数组名是一个常量指针，不允许被赋值。**数组是单纯空间意义上同类数据实体的集合**，数组只是空间的代表，可用数组下标[]操作对单个元素进行访问，但不能对数组（空间地址）作批量元素操作，如 a 与 b。

因而无法看作是同类型数据之间的赋值。

结构就不同了，它大小固定，可以被赋值。

例如，下面的程序对结构变量赋值：

```
//* * * * * * * * * * * * * * * * * * * * * * *
//* *              ch10_2.cpp                 * *
//* * * * * * * * * * * * * * * * * * * * * * *

#include <iostream.h>

struct Person
{
   char name[20];
   unsigned long id;
```

```
    float salary;
};

Person pr1 = {"Frank Voltaire",12345678,3.35};

int main()
{
    Person pr2;
    pr2 = pr1;          //结构变量的赋值
    cout <<pr2.name <<"     "
         <<pr2.id <<"     "
         <<pr2.salary <<endl;
}
```

运行结果为：

```
Frank Voltaire     12345678     3.35
```

程序中定义了一个全局 Person 结构变量 pr1，它使用与初始化数组相似的方法进行初始化。在 main()函数中，定义了一个结构变量 pr2，然后使用赋值运算符将 pr1 的内容赋值给 pr2。

在结构 Person 中，成员 name 是一个字符数组，通过结构变量的赋值，该数组作为成员也被赋值了。

两个不同结构名的变量是不允许相互赋值的，即使两者包含有同样的成员。

例如，下面的代码，不能将结构 Employee 的变量赋给结构 Person 的变量：

```
struct Person
{
    char name[20];
    unsigned long id;
    float salary;
};

struct Employee     //Employee 与 Person 具有相同的成员
{
    char name[20];
    unsigned long id;
    float salary;
};

int main()
{
    Person pr1 = {"Frank Voltaire",12345678,3.35};
    Employee er1;
    er1 = pr1;      //error 类型不匹配
    //...
}
```

→ 在 C 中(而不是 C++)，结构变量定义在结构类型名前必须有 struct 关键字。

例如，定义结构变量 pr1：

```
struct Person pr1 = {"Frank Voltaire",12345678,3.35};
```

否则 C 编译器报错。