21 世纪高等学校计算机类课程创新系列教材・微课版

C/C++程序设计导论
——从计算到编程
（微课视频版）

张力生　张化川　何　睿　赵春泽　编著

清华大学出版社
北 京

内 容 简 介

本书以基础理论和编程实践相结合的方式，图文并茂地介绍设计程序所需的数学知识、计算机基础知识和计算机语言基本知识，从计算和数据角度系统地介绍了设计程序的基本原理、基本方法和典型编程模式，并使用 C/C++语言实现设计的程序。全书分为两部分，共 10 章。第一部分为基础篇，包括概述、表达式和数据类型、构造分支、构造循环、函数等知识；第二部分为应用篇，包括程序组织、数组、指针和引用、结构、底层编程等知识。书中的每个知识点都从数学引入，有相应的数学推导、编程步骤、实现代码和编程要点。

本书适合作为全国高等院校计算机及相关专业的程序设计课程的教材，也可供从事软件开发的专业人员自学使用。

图书在版编目（CIP）数据

C/C++程序设计导论：从计算到编程：微课视频版/张力生等编著. —北京：清华大学出版社，2022.3

21 世纪高等学校计算机类课程创新系列教材：微课版

ISBN 978-7-302-59202-0

Ⅰ. ①C… Ⅱ. ①张… Ⅲ. ①C 语言–程序设计–高等学校–教材 ②C++语言–程序设计–高等学校–教材 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2021)第 187906 号

责任编辑：陈景辉 张爱华

封面设计：刘 键

责任校对：李建庄

责任印制：刘海龙

出版发行：清华大学出版社

网 址：http://www.tup.com.cn，http://www.wqbook.com

地 址：北京清华大学学研大厦 A 座 **邮 编：**100084

社 总 机：010-83470000 **邮 购：**010-83470235

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

课 件 下 载：http://www.tup.com.cn，010-83470236

印 装 者：三河市天利华印刷装订有限公司

经 销：全国新华书店

开 本：185mm×260mm **印 张：**22 **字 数：**537 千字

版 次：2022 年 3 月第 1 版 **印 次：**2022 年 3 月第 1 次印刷

印 数：1~1500

定 价：69.90 元

产品编号：091768-01

前　言

在软件无处不在的时代，程序设计课程的重要性毋庸置疑。本书先通过数学带领读者学习计算和描述计算的表达方式，然后讲解如何使用计算机语言编程。

编程具有较强的科学性和系统性，本书强调使用数学模型解决问题，针对我国学生数学基础好但计算思维薄弱的特点，从四则运算、函数和数学归纳法等初等数学内容入手学习计算方法和表达方式，并融入计算理论、程序理论和计算机系统等基本原理，从计算和数据两条主线，由浅入深地讨论编程的基本知识、基本原理和基本方法，旨在培养能够运用数学知识编程的优秀人才。本书的主要范围如图 0.1 所示。

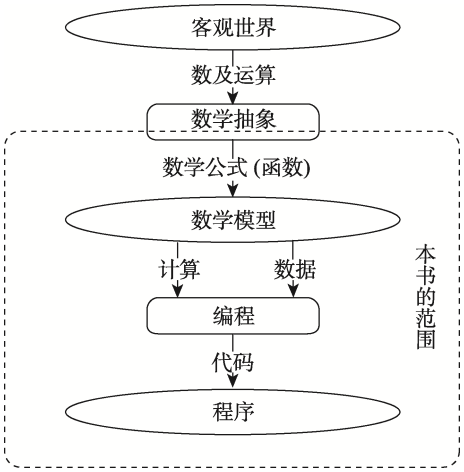


图 0.1　本书的主要范围

编程也是工程性工作，主要涉及详细设计、编码实现两个阶段的工作。详细设计阶段的主要工作是抽象数据和设计计算流程。本书使用流程图、计算顺序图、运算序列图以及内存图等图形语言作为描述数据和计算流程的工具，介绍数学模型中数的表示方法和计算流程，在冯・诺依曼机上详细讨论数据存储方式和运算的实现原理，着重介绍运用数及运算、数学公式（函数）等数学知识设计程序流程、定义数据的方法。

编码实现阶段的主要工作是使用计算机语言描述设计的数据和计算流程，并调试通过。计算机语言的表达方法主要来源于数学和自然语言，本书选用 C/C++语言作为编程语言工具，介绍计算机语言的知识，分层次详细介绍编写代码的步骤和方法，着重介绍怎样运用数学和自然语言的知识描述程序的数据和计算流程。

调试程序是编码实现阶段的重要工作,其工作量在编程中的占比非常高。本书选用Visual Studio 作为集成开发平台，针对表达式、分支、循环、函数和程序模块详细介绍调试程序的步骤和方法。

本书主要内容

全书分为基础篇和应用篇两部分，共 10 章。

第一部分为基础篇，包括第 1~5 章，按照计算机语言的层次关系采用自底向上的方式组织内容，主要从计算角度介绍编程的基本知识和基本方法。

第 1 章为概述。介绍了计算机、计算机语言和程序等基本概念，举例说明了编译和连接的功能和作用，以及调试程序的一般步骤，让读者对编程有一个基本的了解。

第 2 章为表达式和数据类型。从数和四则运算引入表达式和数据类型的概念，主要讨论了基本运算在冯·诺依曼机上的语义和表达式的运算序列，举例说明了计算表达式的步骤和方法，讨论了整型、字符型和实数型等基本数据类型的表示原理和存储格式，介绍了字符流和控制输出格式的方法，并介绍了编写、调试和维护表达式的步骤和方法。

第 3 章为构造分支。从分段函数引入分支的概念，主要介绍使用流程图描述程序流程的方法，深入讨论了通过分段函数构造分支的方法，讨论了关系表达式、逻辑表达式和条件表达式，举例说明了编写分支程序的 4 种典型模式，深入讨论了 I/O 流及其运算，最后介绍了编写、调试和维护分支程序的步骤和方法。

第 4 章为构造循环。从数学归纳法引入了循环的概念，主要介绍了构造循环的基本知识和基本原理，讨论了在数列上采用递推方式构造循环的一般步骤和基本方法，举例说明了递推公式的推导思路，介绍了 while 语句、for 语句和 do…while 语句等循环语句及循环的通用流程框架，讨论了使用循环变量模式和嵌套模式编写程序的方法，最后介绍了调试和维护循环程序的步骤和方法。

第 5 章为函数。从数学函数引入计算机函数的概念,主要介绍了变量和函数的基本知识,介绍了使用“栈”机制管理变量和实现函数调用的原理，讨论了使用“栈”机制分析函数内部执行过程的方法，介绍了定义和调用函数的方法，讨论了描述函数调用关系的方法，举例说明了使用数学递归函数编写计算机递归函数和构造循环的方法，介绍了函数模板以及重载函数技术，最后介绍了调试和维护函数的步骤和基本方法。

第二部分为应用篇，包括第 6~10 章，按照数据类型或应用场景组织内容，主要从数据角度介绍编程的基本知识和基本方法。

第 6 章为程序组织。在总结前面内容的基础上，主要介绍了模块化程序设计思想以及相关的基本知识，着重介绍了多文件结构以及使用方法，介绍了预编译技术以及支持多文件结构程序的基本原理和方法，并介绍了在模块层次上调试程序的基本知识。

第 7 章为数组。介绍了“程序=数据结构+算法”的观点，开始从数据角度介绍编程技术。从数列和矩阵引入数组的概念，介绍了一维数组和二维数组，讨论了其存储结构，举例说明了使用数组管理数据和描述问题的步骤和基本方法。

第 8 章为指针和引用。从内存的逻辑地址引入指针的概念，主要介绍了指针及其运算，

讨论了使用指针访问变量或数组元素的方法，介绍了动态管理堆内存技术，讨论了参数传递的原理，举例说明了各类参数传递的应用场景和编程方法，介绍了字符数组，举例说明了使用字符数组处理字符串的基本方法。介绍了引用及其使用方法，最后讨论了程序安全性。

第 9 章为结构。从向量引入结构的概念，主要介绍了声明结构、定义结构变量以及相关运算等知识，讨论了结构的存储结构和访问结构成员的方法，介绍了使用结构描述二维表并管理其数据的方法，以及使用结构实现链表的基本方法。

第 10 章为底层编程。从整数的记数法引入“按位”计算，介绍十进制、二进制。举例说明了自然数和整数的数值计算方法，讨论了使用补码表示整数的原理，介绍了位运算、联合等底层编程知识和技术。举例说明了基本运算的内部实现原理和方法，详细讲解了在实际应用中抽象、定义底层数据及运算的基本原理和编程方法，介绍了一个最简单的 *R* 进制计算机，讨论了对其扩展的路径和方法，可作为进一步学习、训练的示例。

最后，提供了 ASCII 表和运算表两个附录，以方便读者查阅。其中，运算表详细描述了常用运算的语法和语义，以及优先级和结合性，是编程的基础资料。

本书特色

（1）强化数学基础，夯实计算思维，重点讲解计算方法等内容。本书涉及的数学知识主要来源于初等数学内容，并针对我国中学生计算思维薄弱的特点讲解了计算方法的内容，充分利用我国学生数学基础好的优势，降低学习的门槛。

（2）结构化程序设计思想和面向对象程序设计思想有机融合，旨在节约学生的学习时间。按照先有数再有运算的数学思维，以计算和数据并重的观点组织内容，将结构化程序设计思想和面向对象程序设计思想融合，在程序设计方法上形成一个整体。

（3）图文并茂，通俗易懂。通过使用大量的流程图、计算顺序图、运算序列图和内存图等图形语言，全面讲解程序的计算和数据等内容，以便加强对读者软件设计能力的培养。

（4）紧跟国际工程教育思路，注重实践。将计算机语言、计算机系统、开发环境等回归到工具，强调运用这些工具解决实际问题。

配套资源

为便于教与学，本书配有 530 分钟微课视频、源代码、教学课件、教学大纲、授课计划、习题答案、考试试卷及答案。

（1）获取微课视频方式：读者可以先扫描本书封底的文泉云盘防盗码，再扫描书中相应的视频二维码，观看教学视频。

（2）获取源代码方式：先扫描本书封底的文泉云盘防盗码，再扫描下方二维码，即可获取。

（3）其他配套资源可以扫描本书封底的课件二维码下载。

源代码

进一步学习的内容

当读者完成本书的学习时，是否能成为一名编程的专家呢？答案当然是否定的。但在程序设计领域已经有了一个良好的开始，比较好地掌握了编程所需的基本知识和基本原理，并得到了较好的编程训练。能够编写比较简单的程序，能够从计算角度理解复杂的程序，为进一步的学习打下了良好的理论和实践基础。

当读者完成本书的学习后，进一步学习的最好方法是大量阅读程序并开发一个真正能被别人使用的程序。编程开源区、开发环境中都有很多经典的代码，可供读者选择，但在阅读程序时，需要从数据和计算流程两个方面理解。

还可以进一步从更高层次学习抽象数据和计算的方法，学习面向对象的编程知识和技术。面向对象编程的教材很多，最好选择整合了面向对象设计和编程实现内容的教材，面向对象设计的成果一般会使用 UML 描述，在选择教材时，可通过是否使用了 UML 简单判断是否包含了面向对象设计内容。

读者对象

本书适合作为全国高等院校计算机及相关专业的程序设计课程的教材，也可供从事软件开发的专业人员自学使用。

本书的编写参考了诸多相关资料，在此表示衷心的感谢。限于个人水平和时间仓促，书中难免存在疏漏之处，欢迎读者批评指正。

作　者

2022 年 1 月

目 录

第一部分 基 础 篇

第二部分 应 用 篇

第一部分　基　础　篇

第 1 章

概　　述

编程（programming）可简单理解为程序员与计算机进行交流的过程。程序员使用计算机语言告诉计算机做什么、怎么做，计算机也将做事的过程和结果反馈给程序员，因此，先从程序员与计算机的交流场景开始学习编程。

1.1　计算机

顾名思义，计算机就是计算的机器，它通过执行一系列指令来完成特定的计算功能，被执行的一系列指令统称为程序（program）。一台计算机一般会包含众多的物理设备和各种程序，程序在这些物理设备上运行，实现所需要的功能，这些物理设备统称为硬件（hardware），这些程序统称为软件（software）。

冯·诺依曼机是根据图灵机这个计算模型推导出的计算机逻辑结构，不仅指导人们设计制造计算硬件系统，也是程序员编写程序的基础。

冯·诺依曼机由存储器、运算器、输入设备、输出设备和控制器 5 部分组成。冯·诺依曼机模型如图 1.1 所示。

冯·诺依曼机从功能上描述了计算机的逻辑结构，表明一台计算机应具有 5 大功能部件。存储器也称为内存，其功能为存储程序和数据。运算器也称为（中央）处理器（CPU），其功能为执行程序中的指令，实现规定的运算。输入设备和输出设备完成输入输出功能，控制器用于协调各个功能部件之间协同工作。

经历几十年的发展，“计算机”已发生了根本性的改变，并以各种各样的形态深入到人们的生活和工作，但仍未脱离冯·诺依曼机这个体系结构，也未打破冯·诺依曼机的制约。

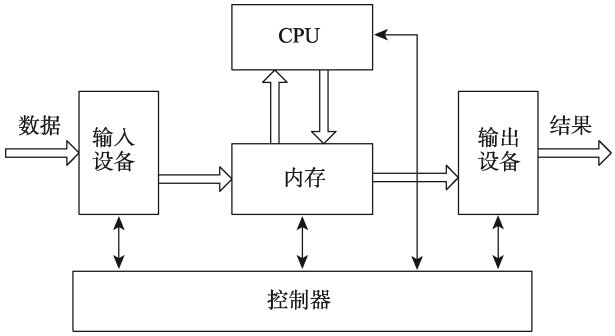


图 1.1　冯・诺依曼机模型

1.2　计算机语言

语言是人类创造的工具，它用来表达意思，交流思想，并借助语言保存和传递人类文明的成果。语言是民族的重要特征之一。

计算机语言也称为程序设计（编程）语言（programming language），是程序员与计算机交流沟通的工具，也是程序员之间进行交流沟通的主要工具。

计算机刚发明时只能理解用 0 和 1 序列表示的机器指令，只有计算机专家能直接用 0 和 1 的序列作为机器指令来编程，编写的程序不仅难以理解，而且效率极低。为了方便专家记忆和编程，用一些符号（助记符）来代替 0 和 1 序列表示的机器指令，出现了汇编语言。汇编语言增强了程序的可读性，也提高了编程的效率，但仍然主要用于专家与计算机交流沟通，不能满足广泛使用的发展需要。

使用机器指令或汇编语言编写程序，主要是用于程序员与计算机之间的交流，编写出的程序与计算机硬件强关联，只能在特定的计算机上运行。机器指令集及相应的汇编语言称为低级语言。

随着计算机的发展，将数学语言和自然语言（英语）中的核心表达特性引入计算机语言，相继诞生了各种各样的高级语言。

如将数学中的代数式和函数等核心表达元素引入计算机语言，程序中出现了“a+b”“y=f(x)”等类似数学的计算机表达式，将英语中的 if 句型引入计算机语言，出现了“if (a>b) {b=a;}”语句。

将数学语言和自然语言（英语）中的核心表达特性引入计算机语言，不仅减少了编写程序的工作量，提高了编程效率，更重要的是，可用数学和自然语言中的方法解决实际问题，可用数学语言和自然语言中的表达方式编写程序，这更符合人们的思维方式，更便于程序员之间进行交流沟通。

程序语言越低级，对过程描写得越具体，指令也就越接近机器的硬件逻辑。相反，程序语言越高级，就越接近对问题的描述与表达，因而更直观、更容易被人们所理解。

高级语言的广泛应用，促使编程人数迅速增加，促进了软件产业突飞猛进的发展。

1.3　为什么选择 C/C++语言

C/C++语言的发展历程和主要特性决定了 C/C++**语言是学习编程的首选语言工具**。

1970 年，AT&T 公司 Bell 实验室的 D. Ritchie 和 K. Thompson 共同发明了 C 语言，研制 C 语言的初衷是用它编写 UNIX 系统程序，C 语言实际上是 UNIX 的“副产品”。C 语言充分结合了汇编语言和高级语言的优点，高效而灵活，又容易移植，所以很受程序设计人员的青睐，成为计算机产业界的宠儿。为此，他们两位获得了 1983 年度的“图灵奖”。

1971 年，瑞士联邦技术学院 N. Wirth 教授发明了 Pascal 语言。Pascal 语言语法严谨，层次分明，程序易写，具有很强的可读性，是第一个结构化的编程语言。它一问世就受到广泛欢迎，为此，N. Wirth 教授获得 1984 年度的“图灵奖”。

20 世纪 70 年代中期，Bjarne Stroustrup 在剑桥大学计算机中心工作。他使用过 Simula 和 ALGOL 语言，实现过低级语言 BCPL，接触过 C 语言，积累了丰富的程序语言经验。1979 年，Bjarne Stroustrup 到了 Bell 实验室，在 C 语言中引 ALGOL 的结构和 Simula 的类，形成了带类的 C（C with classes）。与 C 语言相比，带类的 C 编程更加简单和可靠，运行高效，可移植性更好。1983 年，带类的 C 语言被正式命名为 C++。

20 世纪 90 年代，C 语言慢慢淡出，C++语言逐渐应用。1998 年，ISO/ANSI C++标准正式制定，促进了 C++语言的稳步发展，使其成为一个标志性的计算机语言。

鉴于 C++语言对现代计算机产业的贡献，1995 年 *BYTE* 杂志将 Bjarne Stroustrup 列入“计算机工业 20 个最具影响力的人”。

C++语言对 C 语言的继承是青出于蓝而胜于蓝，它既可以进行 C 语言鼎盛时期所流行的面向过程的**结构化程序设计**，又可以进行以抽象数据类型为特点的**基于对象的程序设计**，还可以进行以继承和多态为特点的**面向对象程序设计**和以模板为特点的泛型程序设计。

C++语言是一种混合型程序设计语言，“混合”体现在可以采用不同的程序设计方法，进行各种目的的编程。“混合”是因为沿革了 C 语言。从本质上说，当今的世界，既有许多规模不大，要求能经济地运行的编程任务，如嵌入式编程，也有越来越多的大规模编程任务，如基于大数据的智能分析系统，因而要求编程语言通用、面广、多样和灵活。“混合”意味着绝不放弃计算机高效运行的实用性特征，而又致力于提高大规模程序的编程质量，提高程序设计语言的问题描述能力。

本书选择 C/C++语言作为学习编程的语言工具，主要是因为它反映了计算机语言的发展历史，它的主要特性和表达能力能够支撑我们学习基本的编程原理和方法，也有助于读者自学其他计算机语言。

需要特别强调的是，本书的主要目的不是学习 C/C++语言，而是学习编程的基本知识、技术和方法，希望能够跳出语言细节而聚焦于程序设计的基本原理和方法，真正为读者深入学习编程打下基础，因此，C/C++语言仅仅作为学习编程的语言工具，主要使用到 C 语言的部分。

1.4 简单程序

下面是一个用 C/C++语言编写的简单程序，它的功能非常简单，就是将输入的两个数相加，并输出结果。两个数相加示例代码如例 1.1 所示。

【例 1.1】 两个数相加。

```
#include<iostream>
using namespace std;
void main(){
    int a, b, sum;
    cout << "请输入两个数:\n";
    cin >> a >> b;
    sum = a + b;                    //两个数相加
    cout << "两数之和: " << sum << endl;
}
```

每个程序都需要一个唯一的入口，在 C/C++语言中，规定了这个入口为 main ( )函数，程序从 main ( )函数的第一行语句开始执行。

cout 表示标准输出设备（output），通常为显示器，<<表示输出，“请输入两个数：\n”是在标准输出设备上输出的内容。

cin 表示输入设备（input），通常为键盘，>>表示输入，“cin >>a >>b”的含义为从输入设备输入两个数到变量 a 和 b。

“sum = a+b”的含义为，将 a 和 b 中的两个数相加，结果放到 sum 中。

后面的语句就是输出 sum 中存放的结果。

只要具有基本的自然语言和数学基础，很容易理解上面的简单程序，但要计算机理解这个程序，还需要给它配备一个“翻译”，即要将高级语言编写的程序“翻译”成计算机能理解的机器语言。

1.5 编译和连接

使用高级语言编程，能够很好地满足程序员之间进行交流沟通的要求，但也必须让计算机硬件能够“理解”程序表达的含义，即程序能够在计算机硬件上运行。为了解决这个问题，专门设计了一组程序，先将高级语言编写的程序“翻译”成一系列机器指令，然后在计算机硬件上逐条运行这些机器指令。

“翻译”例 1.1 的过程如图 1.2 所示，共分为编译和连接（link）两个步骤。先将源程序文件 add.cpp 中的程序“编译”为二进制表示的机器指令序列，并存储到一个目标代码文件 add.obj，然后，将 add.obj 中的代码与系统提供的 cout 和 cin 代码“连接”在一起，生成一个可执行程序文件 add.exe，add.exe 就可以在计算机上运行。

源程序文件（source file）存储程序员编写的程序，以文本形式呈现，常常也将源程序文件中存储的内容称为源代码（source code）。目标代码文件（object file）存储“翻译”后

的机器指令，常常是一个源程序文件对应一个目标代码文件，也将目标代码文件中存储的内容称为目标代码（object code）。可执行程序文件（executable program file）存储程序的完整目标代码，包括各个目标代码文件中的代码，以及系统提供的目标代码。编译、连接程序的过程如图 1.2 所示。

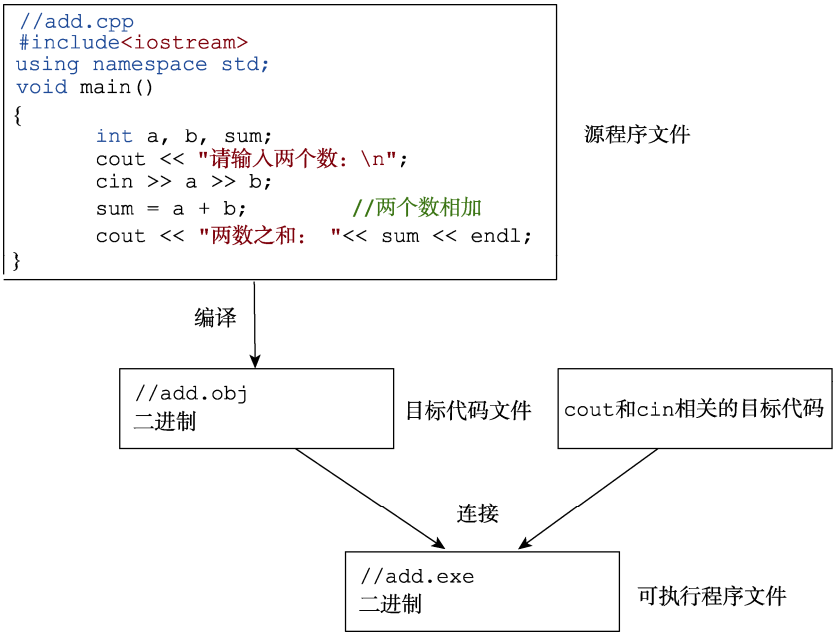


图 1.2　编译、连接程序的过程

C/C++语言提供了许多标准化库，这些库中包含了程序员可以使用的各种程序代码。例如，输入和输出库中提供了一些程序代码，利用这些代码，可以向屏幕输出信息，也可以从键盘获取数据。

通过编译和连接两个步骤将源程序翻译成可执行的程序，是翻译程序的一种方式，这种方式称为编译型。除此之外，还有另外一种称为“解释型”的方式。解释型采用“边翻译边执行”的方式，即翻译一条语句就执行这条语句对应的机器代码，一般不会生成一个完整的可执行文件。解释型有很多应用场景，如前端程序、脚本语言。

程序设计语言发展到现在，无论编译型还是解释型，一般都会提供一个集成开发环境（integrated development environment，IDE）。程序员可以在该环境中完成编辑（edit）、编译（compile）、连接（link、make 或 build）、调试（debug）程序等软件开发工作。

1.6　调试程序

编程是为了解决实际问题，是一件复杂的工作，往往涉及很多方面的事情。首先，在编程前要考虑选择一个程序设计语言作为编程的语言工具，并选择一个相应的集成开发环境。

使用 C/C++语言编程的集成开发环境很多，可适合不同的应用场景。本书选择 Microsoft

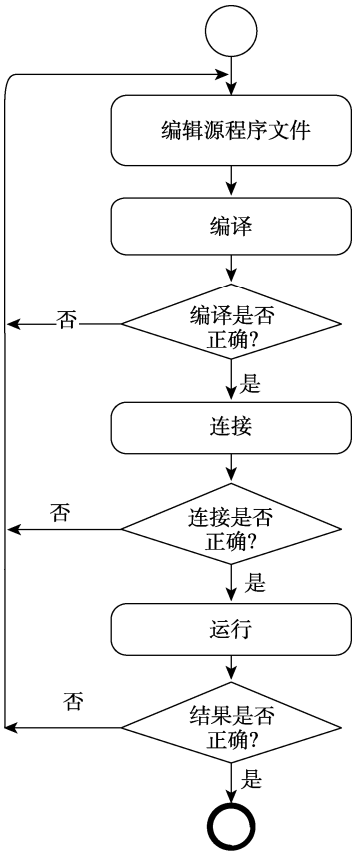


图 1.3　调试程序的一般流程

公司的 Visual Studio 2013（简称 VS2013）集成开发环境，仅仅是因为必须选择一个 IDE 而刚好以前使用过它，并不是必需的。建议读者根据自己的兴趣和具体情况，选择一个适合自己的 IDE 编写、调试程序。

选择好程序设计语言和集成开发环境后，就可以开始编写、调试程序了。调试程序是一个不断迭代的过程，包括编辑、编译、连接、调试 4 个步骤，整个流程循环往复，直至编写出所需要的程序。调试程序的一般流程如图 1.3 所示。

编辑是指程序员在集成开发环境中编写已设计出的程序，主要目的是将程序的源代码输入到计算机，程序以文本形式呈现，这种程序被称为源程序。源程序一般存放在一个或多个文件中，存放 C++源程序的文件常常以.cpp（在 Windows 环境中）作为文件扩展名。

编译是指将程序员编写的源程序翻译成机器指令，生成目标代码，存放在目标代码文件中，后面简称为目标文件。

Windows 环境中的 C++编译器通常以 .obj 作为目标文件的扩展名。目标代码也称为机器代码，是计算机能够识别的指令集合。源程序被编译后生成的目标代码只是一个个独立的程序段，还不能在计算机上运行。

连接是指将分散在各个目标文件和标准库中的相关代码整合成一个完整的程序，生成一个可执行文件，可执行文件通常以 .exe 作为文件扩展名。C/C++程序在编译后，通过连接若干目标文件与若干库文件而创建可执行程序。库文件是系统提供的程序链接资源，属于集成开发环境的一部分。目标文件与库文件连接的结果是生成计算机可执行的程序。

在编译时，需要对源程序进行语法和语义分析，如果没有发现错误，就将源程序翻译为目标代码，并生成目标文件，如果发现错误，就终止编译。在连接时，首先检查所需的目标代码是否完整，如果所需要的目标代码都有了，就将这些目标代码整合在一起，生成相应的可执行文件，否则，终止连接，不生成可执行文件。

通过编译和连接后生成的可执行文件可以执行，但并不代表一定能得到预期的结果。确保程序运行能得到预期的结果这个工作还需要程序员来人工完成，这就是调试程序。

调试程序是根据程序的运行结果发现并改正程序中的错误（bug），这是一个不断迭代的过程。在编程过程中，调试程序的工作量很大。程序员中流行一种说法——编程的主要工作就是不厌其烦地“抓虫”，即找 bug，调试程序。

集成开发环境功能齐全，调试功能很强，程序编好后，可以立刻在 IDE 中调试以获得初步测试结果，然后，可以方便地做成 Beta 版形式，拿到实际环境中进一步测试，最后做成软件发行版。

编程是一种能力，具体体现在 3 个层次：第一，能够运用数学、自然科学和计算科学与技术中的基本知识和基本原理设计程序；第二，能够编写并调试程序；第三，编写的程序能够解决实际问题，投入实际应用。

编程能力不能通过传授方式获得，只能在不断的编程训练中逐步培养。初学者，不仅要学习基本的知识和原理，也需要将学习重点聚焦到编程方法上，并不断上机调试。

1.7　本章小结

本章主要介绍了计算机、计算机语言和程序等基本概念，通过一个简单程序介绍了编译和连接的功能和作用，最后介绍了调试程序的一般步骤，让读者对编程有一个基本的了解，为后面学习编程打下基础。

1.8　习题

1. 计算机的 5 个主要部件是什么？
2. 机器语言程序和高级语言程序的区别是什么？
3. 编译器有什么作用？
4. 什么是源程序？什么是目标程序？
5. 你使用的计算机运行的是什么操作系统？
6. 什么是连接？
7. 在自己的计算机上安装一个适合的 IDE，并调试通过例 1.1 中的程序。

第 2 章

表达式和数据类型

图灵奖得主高德纳（Donald E. Knuth）提到，很多人认为算术运算只是小孩学的简单玩意儿，用计算器就能做，但事实上，它是一个非常有趣和迷人的研究课题，算术运算构成了计算机各种应用的重要基础。

因此，从算术运算开始学习编程，先学习使用表达式描述计算机中的计算，再学习存储和管理数据的基本知识，最后学习编写、调试表达式的基本方法。


视频讲解

2.1 表达式

下面从四则运算开始学习计算机语言中的表达式，学习如何使用一个表达式向计算机描述一个简单的计算。

2.1.1 四则运算中的计算

四则运算是算术运算中最基础也是最重要的内容。四则运算的计算规则是“先乘除后加减”和“从左到右依次计算”。

例如，计算$1\times2+4\div2+3\times5$。

$1\times2+4\div2+3\times5$	计算1×2
$=2+4\div2+3\times5$	计算$4\div2$
$=2+2+3\times5$	计算3×5
$=2+2+15$	计算$2+2$
$=4+15$	计算$4+15$
$=19$	

计算$1\times2+4\div2+3\times5$步骤是，计算1×2得到2，计算$4\div2$得到2，计算3×5得到15，

计算 2+2 得到 4，计算 4+15 得到 19，最后得到算式的结果 19。算式 1×2+4÷2+3×5 的计算步骤如图 2.1 所示。

“先乘除后加减”的原则，将加减乘除划分为两个级别，乘除运算为一个级别，加减运算为一个级别，乘除运算的级别高于加减的级别，并规定先计算级别高的运算，再计算级别低的运算。按照这种方法划分出的级别简称为运算的**优先级**。

当运算的优先级相同时，应按照“从左到右依次计算”的原则确定运算的计算顺序，先计算算式中左边的运算，再计算右边的运算。这条原则称为运算的**结合性**。一个运算的结合性一般是“从左到右”，但也有“从右到左”的情况。

从计算角度讲，使用运算的优先级和结合性能唯一确定一个算式的计算顺序。

加法和乘法还具有交换律，使用交换律对算式进行等式变换，可推导出多个相等的算式。如使用加法的交换律对 1×2+4÷2+3×5 进行变换，可推导出算式 4÷2+1×2+3×5，其计算步骤如图 2.2 所示。

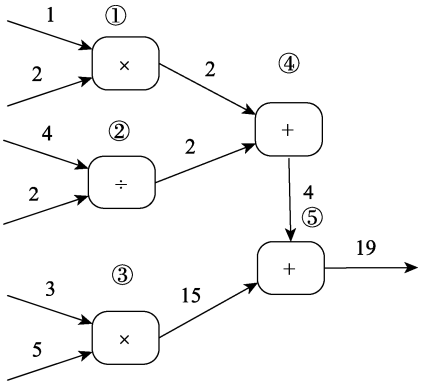


图 2.1　算式 1×2+4÷2+3×5 的计算步骤

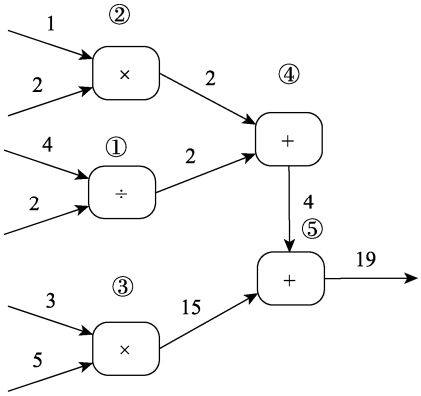


图 2.2　推导出算式 4÷2+1×2+3×5 的计算步骤

使用四则运算的结合律和交换律，可推导出与 1×2+4÷2+3×5 相等的多个算式，按照这些算式的计算顺序都能计算出正确的结果。建议读者试一下，能推导出多少这样的算式。

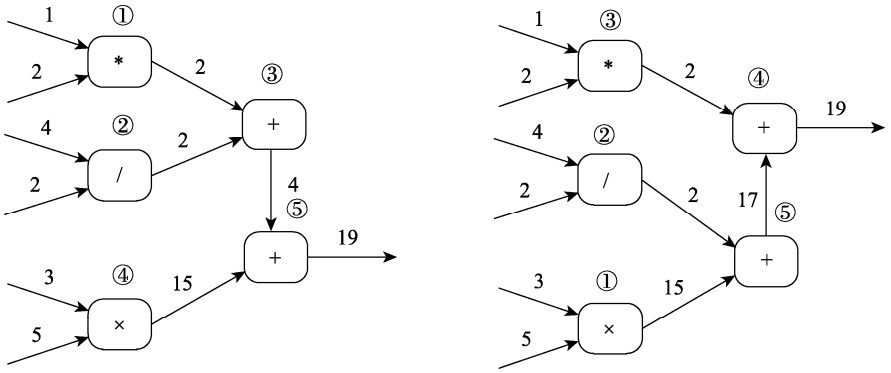
使用数学方法，一个算式可推导出多个相等的算式，按照每个算式规定的计算顺序都能计算出相同的结果。

2.1.2　在计算机中的计算顺序

使用计算机语言描述四则运算算式非常简单。因键盘中没有×和÷两个符号，需用*和/分别替换。如算式1×2+4÷2+3×5，可以改写为

1*2+4/2+3*5

改写后的算式可在计算机中运行，称为计算机表达式，简称表达式。表达式在计算机中有两种可能的计算顺序，即有两种可能的计算步骤。如，表达式 1*2+4/2+3*5 有两种可能的计算步骤，如图 2.3 所示。



(a) 左边优先　　(b) 右边优先

图 2.3　表达式 1*2+4/2+3*5 的两种计算步骤

图 2.3 所示的两种可能的计算步骤，其计算顺序与数学上的计算顺序不同，但每种计算步骤一定能对应与原式相等的一个算式，从而保证了计算结果的正确性。

建议读者分别推导出前面两种计算顺序对应的算式。

一个数学算式存在多个正确的计算顺序，计算机只从中选择一种计算顺序进行计算，所选择的计算顺序往往与数学中的计算顺序有所不同。

在计算机编程中，不能不考虑计算顺序对计算结果的影响。如计算 10÷3×6，其结果应该为 20，但在计算 10÷3 时，由于除不尽会产生计算误差。如果精度保留到两位小数（计算机中必须指定精度），计算结果为 19.98，而不是 20。

总而言之，计算机在计算算式时，只选择数学算式的一种计算顺序进行计算，而计算结果因精度等原因，可能得不到预期的结果，这就要求程序员在编写表达式时，必须清楚表达式的计算顺序，并保证这个计算顺序能得到预期的结果。

2.1.3　表达式的运算序列

中学数学中还学习了代数。在代数中使用字母“代”替算式中的“数”，构成了带有字母的算式，这种带有字母的算式被称为代数式。

代数的常见应用就是数学公式，数学公式中的字母称为变量，代入的数被称为变量的值。数学公式中除了代数式外，一般还包含等号。

如，计算最简单的数学公式 $c = a + b$，可编写如下代码：

```
unsigned int a = 5, b = 4, c;
c = a + b;
```

在计算表达式 c=a+b 之前，需要先将变量 a 和 b 的值存储到内存，然后才能进行计算。表达式 c=a+b 在冯・诺依曼机上的计算过程如图 2.4 所示。

表达式 c=a+b 在冯・诺依曼机上的计算过程分为 4 个步骤。

第 1 步，从变量 a 的内存单元中取出值 5 到 CPU；

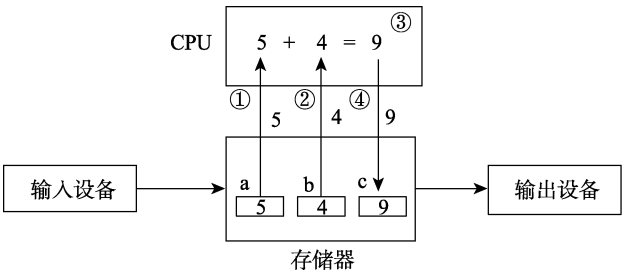


图 2.4　表达式 c=a+b 在冯・诺依曼机上的计算过程

第 2 步，从变量 b 的内存单元中取出值 4 到 CPU；

第 3 步，CPU 计算 5 加 4 得到值 9；

第 4 步，将值 9 存储到变量 c 的内存单元中。

在冯・诺依曼机上计算表达式 c=a+b 时，核心是其中的 4 个计算步骤，因此，可去除冯・诺依曼机等背景信息，仅仅描述这 4 个计算步骤。表达式 c=a+b 的计算步骤如图 2.5 所示。

图 2.5 更加简洁，也能突出重点，但仍然要按照图 2.4 来理解。

一个表达式描述了在冯・诺依曼机上的一组有顺序的计算步骤，一般将这组计算步骤称为一个**运算序列**。

程序员用计算机语言描述数学公式时，往往不知道数学公式中变量的值，在画图时可用一个符号来表示程序运行时变量的值。表达式 c=a+b 的运算序列如图 2.6 所示。

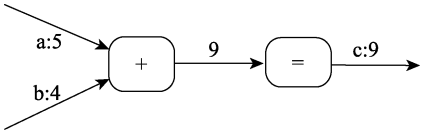


图 2.5　表达式 c=a+b 的计算步骤

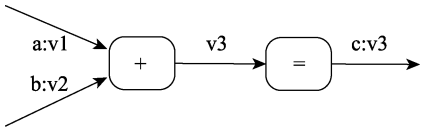


图 2.6　表达式 c=a+b 的运算序列

图 2.6 描述了表达式 c=a+b 定义的运算序列，其中用符号表示各变量的值，这种图称为**运算序列图**。本书约定，在表示运算序列时都以“v+数字”的形式标注其中的变量值。在进行具体计算时再将这些符号替换为变量的具体值，如图 2.5 所示。

计算机表达式描述的是冯・诺依曼机上的一个运算序列。

从本质上讲，每个表达式都定义了一个运算序列，也将定义的运算序列称为表达式的语义。

除了使用运算序列图来表示一个表达式的语义外，还可用文字描述，如，c=a+b 的语义（运算序列）为：从变量 a 取出值 v1，从变量 b 取出值 v2，v1 加 v2 得到值 v3，将值 v3 存储到变量 c 的内存，得到变量 c。

用运算序列图描述表达式的语义，直观清楚，便于理解，但应用场景有限。另外，用文字描述表达式的语义，应用范围更广，但比较抽象，因此，建议结合使用。

2.1.4 计算表达式的基本方法

在实际编程中，程序员的最主要工作之一是编写表达式。编写表达式的基础是将一个数学公式改写为表达式，并保证按照表达式的计算顺序能够得到预期的计算结果。

例如：求一个正方形和一个矩形周长之和。

根据求正方形和矩形的周长公式，很容易写出如下数学公式：

$$c=4l+2(a+b)$$

并改写为如下表达式：

c=4*l+2*(a+b)

按照数学中的方法，先计算优先级高的运算，再计算优先级低的运算，优先级相同时按照结合性依次计算，其核心思想是在算式中寻找最先计算的运算，但计算机采用的是另一种方式，其核心思想是在表达式中寻找最后计算的运算。

在表达式中寻找最后计算的运算时，仍然使用优先级和结合性，但在使用优先级时不是按照从高到低，而是按照从低到高寻找，这与数学计算时的顺序刚好相反。同样，使用结合性时，也是按照结合性相反的顺序寻找。

根据编译原理并结合编程场景专门设计了一种计算表达式的图形方法，称为**计算顺序图**。使用计算顺序图计算表达式，主要包含两个步骤：第 1 步，确定表达式的运算顺序；第 2 步，计算表达式的值。

1. 确定表达式的运算顺序

计算顺序图中包含画水平线和竖线两个操作。水平线的含义是“计算这个表达式”，竖线的含义是“计算这个运算”，水平线到竖线的含义是“要计算这个表达式，先计算这个运算”，竖线到水平线的含义是“要计算这个运算，先计算这个表达式”。

这种句式体现了递归思想，因此，强烈推荐一边画图一边念这两句话，以逐步培养自己的递归思维。

第 1 步：要计算 c=4*l+2*(a+b)，先计算=。

先在整个表达式的正下方画一条水平线，表示“**要计算 c=4*l+2*(a+b)**”，然后在表达式 c=4*l+2*(a+b)中寻找到最后计算的运算=，并在=的正下方画一条与水平线正交的竖线，表示“**先计算运算=**”。整个表达式 c=4*l+2*(a+b)中最后计算的是运算=，如图 2.7 所示。

=是计算机语言中的一个运算，称为赋值运算，其他几个运算的优先级都比它高，因此，它是最后计算的运算。

运算=将表达式分成左右两个表达式，左表达式中只有一个变量 *c*，没有包含运算，不存在寻找运算的问题，右表达式 4*l+2*(a+b)包含了多个运算，需要继续寻找其中最后计算的运算。

第 2 步：要计算=，要先计算 4*l+2*(a+b)。

因=下面的竖线已经画了，只需在 4*l+2*(a+b)的正下方画一条水平线，表示“要先计算 4*l+2*(a+b)”。

第 3 步：要计算 4*l+2*(a+b)，先计算+。

“反向”使用“先乘除后加减”和“从左到右依次计算”两条运算规则，在 4*l+2*(a+b)

中寻找到最后计算的运算+（从左边数第 1 个），然后在它的在正下方画一条竖线。表达式 4*l+2*(a+b)中最后计算的是运算+，如图 2.8 所示。

运算+将表达式分成左右两个表达式，按照左边优先计算的假设，先在左表达式 4*l 中继续寻找最后计算的运算，然后在右表达式 2*(a+b)中寻找。

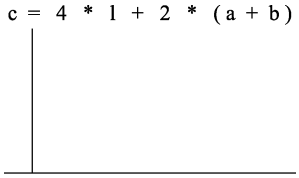


图 2.7　c=4*l+2*(a+b)中最后计算的运算=

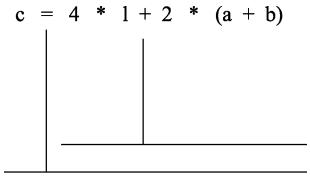


图 2.8　4*l+2*(a+b)中最后计算的运算+

第 4、5 步：要计算+，先计算 4*l；要计算 4*l，先计算*。

按照第 1~3 步中的方法在表达式 4*l 中寻找最后计算的运算*。

第 6 步：为“能够”计算的运算*标注计算顺序号①。

按照“要计算这个运算，先计算这个表达式”的逻辑，计算表达式 4*l 中只有运算*，“能够”计算其中的运算*，因此，在运算*的正上方标注其计算顺序号①，如图 2.9 所示。

在可计算的运算*上方标注计算顺序号后，沿着寻找路径退回到运算+，“**要计算**运算+”，还要“**先计算**表达式 2*(a+b)”。

第 7~10 步：在 2*(a+b)中寻找最后计算的运算并标注计算顺序。

按照寻找最后计算运算的方法，先在 2*(a+b)中寻找到最后计算的运算*，然后在(a+b)中寻找到最后计算的运算+，并标注运算+的计算顺序号②，如图 2.10 所示。

其中，括号不是运算，而是构成表达式的符号，其作用是提高运算的优先级，以保证先计算括号内的运算再计算括号外的运算。

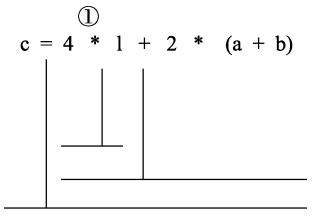


图 2.9　标注运算*的计算顺序①

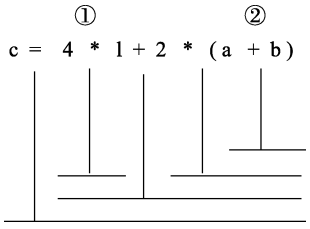


图 2.10　标注运算+的计算顺序②

第 11 步：沿着寻找路径退回到运算*，并标注其计算顺序号③。

2*(a+b)中，因其中的运算+已标注计算顺序号，表示已在前面计算了(a+b)，因此，“能够”计算其中的运算*，在其上方标注计算顺序号③，如图 2.11 所示。

第 12、13 步：按照第 11 步的方法，先退回到运算+，并标注其计算顺序号④，然后退回到运算=，并标注其计算顺序号⑤，最终确定了表达式 c=4*l+2*(a+b)中所有运算的计算顺序，如图 2.12 所示。

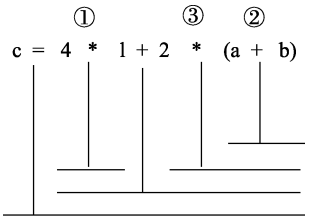


图 2.11　标注运算*的计算顺序③

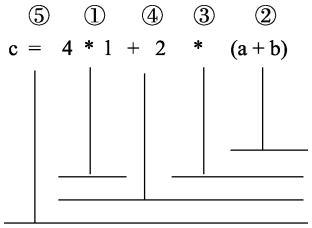


图 2.12　表达式 c=4*l+2*(a+b)中所有运算的计算顺序

经过 13 个步骤，在图 2.12 中标注了表达式 c=4*l+2*(a+b)中所有运算的计算顺序号，确定了所有运算的计算顺序。确定表达式运算顺序的步骤虽然很多，但有大量重复性工作，因此可将其归纳总结为 3 个主要步骤。

第 1 步，寻找“能够”计算的运算。“反向”使用运算的优先级和结合性，在表达式中寻找最后计算的运算，直到寻找到“能够”计算的运算。

第 2 步，沿着寻找路径逐个“退回”并标注计算顺序号。当寻找到“能够”计算的运算时，标注该运算的计算顺序号，并沿着寻找运算的路径逐个“退回”，继续给“能够”计算的运算标注计算顺序号，直到“退回”到不能计算的运算。

第 3 步，跳到第 1 步继续寻找。当“退回”到不能计算的运算时，跳到第 1 步，从这个运算开始继续寻找能够计算的运算并标注其计算顺序号，直到为表达式中的所有运算都标注了计算顺序号。

在画计算顺序图时，需要注意 3 点。

第 1 点，水平线一定要与表达式左右对齐，一样长；

第 2 点，水平线与表达式之间要预留足够空白；

第 3 点，竖线一定要在运算的正下方。

计算顺序图不仅描述了一个表达式中运算的计算顺序，也能表示出表达式的运算序列。根据图 2.12 所示的计算顺序，可画出表达式 c=4*l+2*(a+b)的运算序列图，如图 2.13 所示。

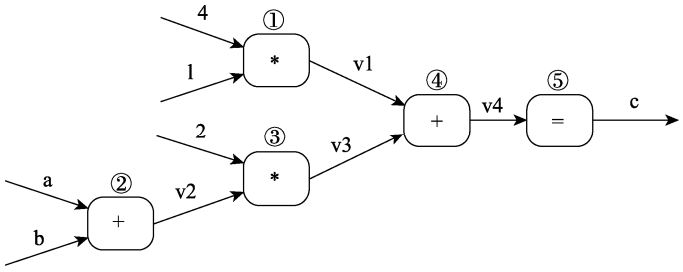


图 2.13　表达式 c=4*l+2*(a+b)的运算序列图

当一个运算将一个表达式分成左右两个表达式时，前面假设了先计算左边的表达式，即采用左边优先的原则。如果采用右边优先的原则，先计算右边的表达式，表达式 c=4*l+2*(a+b)的计算顺序如图 2.14 所示。

在计算机中，表达式有两种不同的计算顺序，但一个编译器只能选择其中一种，不可能同时选用两种。

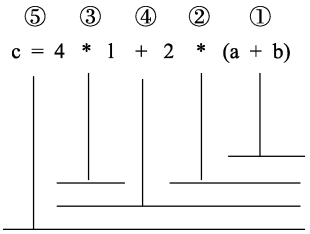


图 2.14　右边优先的计算顺序

2. 计算表达式的值

在计算表达式前，需要先给表达式中的变量指定值。如，给c=4*l+2*(a+b)的所有变量指定值。

```
int l = 2, a = 3, b = 4;
```

在计算顺序图上计算表达式的值，比较简单。具体方法是，按照计算顺序图中标注的计算顺序，根据运算的语义依次计算表达式中的运算，最终计算出表达式的结果。

在计算顺序图中，找到计算顺序号为①的运算*，计算 4*l 得到计算结果 8，并标注在计算顺序图中。具体步骤为，从变量 l 内存中取出整数 2，标注在变量 l 的下方，计算 4*2 得到 8，将结果 8 标注在图中该运算下方的右边直角处。运算*（标号为 1）的计算结果如图 2.15 所示。

按照图 2.15 中标注的计算顺序，依次执行其他运算，并将计算结果标注在计算顺序图中，最终计算出整个表达式的结果。表达式 c=4*l+2*(a+b)的求值过程如图 2.16 所示。

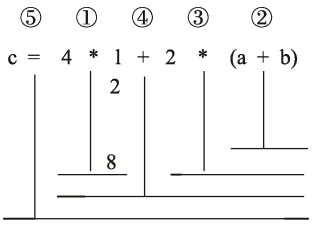


图 2.15　运算*（标号为 1）的计算结果

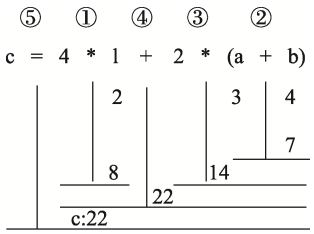


图 2.16　表达式 c=4*l+2*(a+b)的求值过程

从本质上讲，计算表达式就是在冯・诺依曼机上执行表达式的运算序列，因此，也可在表达式的运算序列图上按照标注的计算顺序计算其中的运算，并将每个运算的计算结果标注在运算序列图中，最终计算出表达式的值。如，表达式 c=4*l+2*(a+b)在运算序列图中的计算过程如图 2.17 所示。

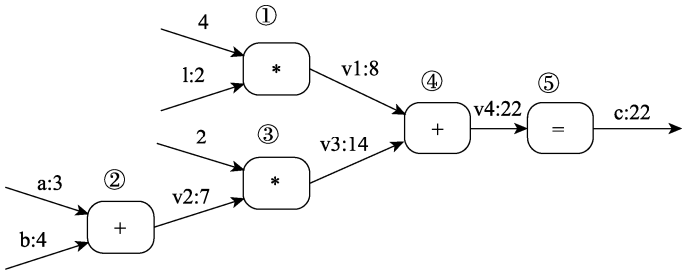


图 2.17　表达式 c=4*l+2*(a+b)在运算序列图中的求值过程

使用计算顺序图和运算序列图都能计算表达式的值。两种方法相比，使用运算序列图进

行人工计算，更能体现表达式在冯·诺依曼机上的计算过程，而使用计算顺序图人工计算，更加简洁，效率更高。


视频讲解

2.2 算术运算

在数学中，将世间万物都抽象为数和运算，再讨论其性质，发现其规律，定义推理规则，经过不断发展，最终形成了庞大的数学理论体系。计算机按照数学这个思想，以数和运算为基础，构造了信息世界的高楼大厦。

计算机语言不仅提供了**算术运算**、**逻辑运算**、**关系运算**等数学中的常见运算，还提供了很多计算机专用运算，如位运算、地址运算等。

在附录 B 中专门提供了 C/C++的运算表，其中以运算序列的方式描述了每个运算的语义，供读者查阅。也建议初学者经常查阅运算表，理解运算的语义，并熟练掌握常用运算的使用。

计算机语言一般提供了加（+）、减（−）、乘（*）、除（/）四则运算，除此之外，还提供负号、模（%）等其他基本的算术运算，但一般不直接提供乘方、开方等比较复杂的运算。

C/C++语言中，常用的算术运算有 5 个，每个算术运算符都有特定的语法语义。算术运算的语法和语义如表 2.1 所示。

表 2.1 算术运算的语法和语义

运算符	名称	结合性	语法	语义或运算序列
*	乘 (multiplication)	从左到右	exp1*exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，v1 乘以 v2 得到值 v3
/	除 (division)	从左到右	exp1/exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，v1 除以 v2 得到值 v3
%	模 (modulus)	从左到右	exp1%exp2	计算 exp1 得到整数类型的值 v1，计算 exp2 得到整数类型的值 v2，v1 除以 v2 取余数得到整数类型的值 v3
+	加 (addition)	从左到右	exp1+exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，v1 加 v2 得到值 v3
−	减 (subtraction)	从左到右	exp1−exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，v1 减 v2 得到值 v3

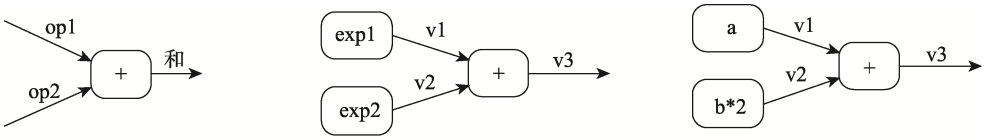
表 2.1 中的算术运算都有两个操作数，称为双目运算。每个运算用一个符号来标识，标识运算的符号被称为运算符。要注意的是，加减运算的运算符与数学中的符号完全相同，但乘除运算的运算符不一样。

算术运算也有优先级和结合性，与数学中的优先级和结合性完全相同。在表 2.1 中，按照从高到低的顺序排列，用加粗的表格线以区分不同优先级的运算，相邻两条加粗表格线之间的运算有相同的优先级。

2.2.1 算术运算的语法和语义

算术运算有语法和语义，算术运算的语法规定了使用该运算构成表达式的法则，算术运算的语义规定了数集上的映射规则，即数据的转换规则。

在数学中，使用操作数 op 描述算术运算的语法和语义。如加法运算的语法为“op1 + op2”，其中，op1 和 op2 为操作数；加法运算的语义为，op1 加 op2 得到两数之和，将 op1 和 op2 两个数映射到它们的和。数学中加法运算的语义如图 2.18(a)所示。



(a) 数学中加法运算的语义　(b) 计算机中加法运算的语义　(c) 表达式a+b*2的语义

图 2.18　加法的语义及应用举例

计算机语言中的算术运算是使用数学的算术运算定义的，并针对计算机的特点进行了扩展，以便于构造和计算表达式。

在语法方面，将数学中的操作数 op 替换为表达式 exp，融入了使用算术运算构造表达式的规则。如加法运算，将其语法“op1+op2”修改为“exp1+exp2”，要求 exp1 的计算结果作为加法的操作数 op1，exp2 的计算结果作为加法的操作数 op2。在构造表达式时使用一个表达式替换其中的操作数，如，可用表达式 a 和 b*2 分别替换 exp1+exp2 中的 exp1 和 exp2，构成表达式 a+b*2。

在语义方面，增加了计算操作数的步骤，规定先计算得到所需的操作数，然后再按照数学中规定的映射规则进行计算，最后得到算术运算的计算结果。如加法运算，在计算加法之前，增加了计算 exp1 和 exp2 的步骤，其计算序列为，计算 exp1 得到值 v1，计算 exp2 得到值 v2，v1 加 v2 得到值 v3。计算机中加法运算的语义如图 2.18(b)所示。

根据算术运算的语义，可推导出表达式的运算序列。如，表达式 a+b*2 由表达式 a 和 b*2 相加而成，使用 a 和 b*2 分别替换加法语义中的 exp1 和 exp2，得到表达式 a+b*2 的运算序列：计算 a 得到值 v1，计算 b*2 得到值 v2，v1 加 v2 得到值 v3。表达式 a+b*2 的语义如图 2.18(c)所示。

一个表达式往往包含多个算术运算，会涉及多个运算的语法和语义。如，表达式 a+b*2 包含加法和乘法两个运算，在计算加法运算前，还需要计算 b*2，需要先执行其运算序列。

除了一个是乘法、一个是加法外，乘法的语法和语义与加法完全相同，按照乘法的语法很容易分析出表达式 b*2 的构成，并按照其语义推导出其运算序列：计算 b 得到值 v21，v21 乘 2 得到值 v2。将表达式 b*2 的运算序列插入如图 2.18(c)所示的运算序列，最终得到表达式 a+b*2 的完整运算序列：计算 a 得到值 v1，计算 b 得到值 v21，v21 乘 2 得到值 v2，v1 加 v2 得到值 v3，如图 2.19 所示。

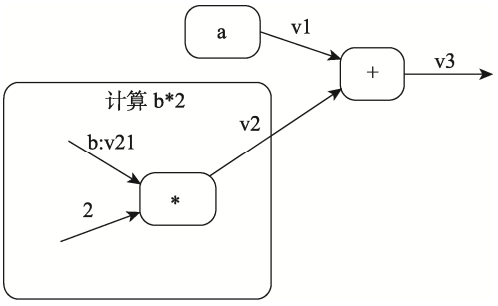


图 2.19　表达式 a+b*2 的完整运算序列

2.2.2 编写表达式

解决实际问题时，一般先从客观世界中抽象出解决问题的数学模型，然后根据数学模型编写程序。在编程时，首先需要将数学模型中的数学公式改写为计算机表达式。

编写表达式的基本思路如图 2.20 所示。

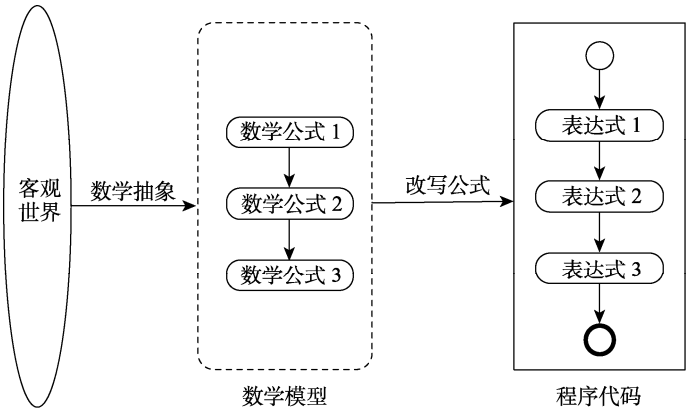


图 2.20 编写表达式的基本思路

在建立数学模型时，程序员一般以数学公式为粒度思考问题，而计算机只能在运算这个粒度上理解表达式。下面仍然以 c=4*l+2*(a+b)为例，在运算粒度上讨论构造表达式的过程。表达式 c=4*l+2*(a+b)的构成如图 2.21 所示。

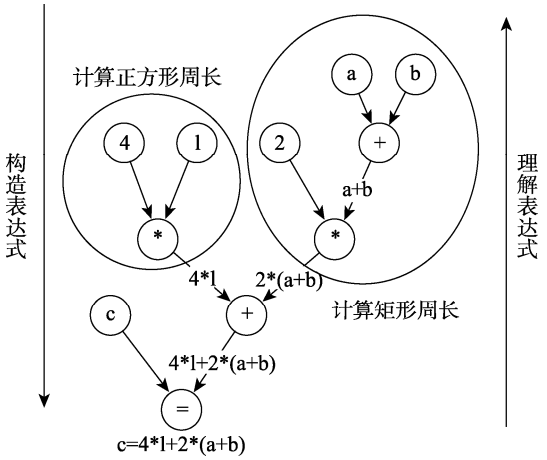


图 2.21 表达式 c=4*l+2*(a+b)的构成

可按照“从右到左，从上到下”的顺序，理解其构造过程：变量 a 和 b 相加构成 a+b，2 和 a+b 相乘构成 2*(a+b)；4 和 l 相乘构成 4*l；4*l 和 2*(a+b)相加构成 4*l+2*(a+b)，最后将 4*l+2*(a+b)的计算结果存到变量 c，构成 c=4*l+2*(a+b)。

也可按照“从左到右，从上到下”的顺序理解表达式 c=4*l+2*(a+b)的构造过程：4 和 l

相乘构成 4*l；变量 a 和 b 相加构成 a+b，2 和 a+b 相乘构成 2*(a+b)；4*l 和 2*(a+b)相加构成 4*l+2*(a+b)，最后将 4*l+2*(a+b)的计算结果存到变量 c，构成 c=4*l+2*(a+b)。

与使用计算顺序图计算表达式对照，读者可以发现，上述两种构造过程刚好与“右边优先”和“左边优先”相对应，可以通过表达式的计算顺序反向推出构造表达式的过程，理解编写表达式的真实意图。

表达式的构成方法与数学中公式的构成方法完全相同，也表示同一个运算序列，因此，编写计算表达式的方法很简单，只需先写出数学公式，然后将数学公式改写为表达式。当然，这个方法针对的是包含四则运算的比较简单的数学公式。

2.2.3　表达式语句

表达式语句是 C/C++中最常用的语句，语法非常简单，只需要在一个表达式后面加个分号（;），就构成一个表达式语句，其语法如下：

```
exp;
```

其中，exp 是一个表达式，分号(;)是一条语句结束标志，C/C++中的每条语句都必须以分号（;）结束，这与我们平时习惯不同。在平时习惯中，往往按 Enter 键（即回车符）换行来表示一句话或一段话结束，需要特别注意。

例如：

```
int l = 2;
int a = 3;
int b = 4;
int c;
c = 4 * l + 2 * (a + b);
```

上面 5 条语句都是表达式语句，其中，c = 4 * l + 2 * (a + b)是一个表达式，这很好理解，在后面加上分号，构成一个表达式语句。

在 C/C++中，定义了丰富的运算，这些运算可以构成实现各种功能的表达式，具有强大的表现能力。但这也导致了一个问题，有很多表达式在形式上与数学算式完全不同，初学者会很不习惯，不过时间长了，自然就习惯了。

很多计算机语言将定义变量作为一条语句，但在 C/C++中，定义变量也是运算，因此 int l = 2、int a = 3、int b = 4 和 int c 都是一个表达式，在后面加上一个分号，就构成了一条表达式语句。

int a = 3 的语义为，在内存中分配 int 规定大小的内存，设置变量名为 a，并用 3 初始化变量 a。int c 的语义为，在内存中分配 int 规定大小的内存，设置变量名为 c，不做初始化。这两个表达式中都只有一个运算，int a = 3 中“=”表示初始化，是定义变量中的操作步骤。

2.3　变量及其运算

数学，顾名思义，就是研究“数”的学科；代数，就是用字母“代”替“数”参与计算。

下面从代数中的字母开始学习编程中的变量，学习计算机存储和管理数据的基本知识和方法。

2.3.1 计算机中的变量

变量和数据类型既是计算机中的核心概念，也是计算机语言的核心内容，同时也是编程的基础。

1. 变量的概念

计算机中的变量来源于数学中的变量，如求一个圆的周长公式

$$l = 2\pi r$$

其中，圆的半径 r 决定了周长的大小。给定一个 r 的数值，就会得到圆的周长 l，r 和 l 是可变的，称为变量。

又如函数的一般形式

$$z = f(x, y)$$

其中，x，y 是自变量，z 为因变量，都是变量。每个变量都有一个取值范围，x，y 的取值范围称为定义域，z 的取值范围称为值域。

从数学中引入了变量的概念，并用计算机中的内存来存储变量的值，形成了计算机变量的概念。计算机变量这个概念不仅包括变量名和变量的值两个数学概念，还包括内存和数据类型两个计算机的专业概念。计算机变量的概念如图 2.22 所示。

计算机内存有众多内存单元（一般以字节为单位），为了区分这些内存单元，为每一个内存单元指定一个编号，这些编号称为内存的地址。编号的编码方法有很多，读者可将内存地址简单理解为一个自然数。

每个计算机变量对应内存中地址连续的一组存储单元，构成一块内存区域，这块内存区域称为**变量的内存**。称内存区域在内存中的开始地址为**首地址**，称内存区域包含的内存单元个数为**偏移**。通常用首地址和偏移表示一块内存区域在内存中的位置和大小。

为了便于计算和存储，必须为每个计算机变量指定一个数据类型。数据类型的作用有两个：第一，指定了计算机变量对应内存区域的大小，即从变量的首地址开始多少字节；第二，指明这个变量是用于存储自然数、整数还是实数。

变量的内存图形象地描述了计算机变量及数据在内存中的状态。为了便于初学者理解，用“长度”替换了“偏移”这个计算机术语。

在定义计算机变量时，必须同时指定一个变量的变量名和数据类型，也可以指定变量的值。如定义一个存储自然数的变量：

```
unsigned short a = 3;
```

上述语句定义了一个变量 a，数据类型为无符号整数(unsigned short)，内存大小为 16 个二进制位，用于存储 0~2^{16}–1 的一个自然数。可用图形表示定义的变量 a，这种图称为变量的内存图。变量 a 的内存图如图 2.23 所示。

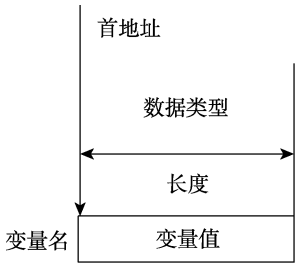


图 2.22　计算机变量的概念

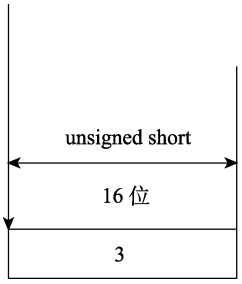


图 2.23　变量 a 的内存图

语句“unsigned short a = 3”中，“=”表示初始化，即给定义的变量指定一个初始值。这条语句的语义为：在内存中为一个变量分配两个字节（16 位）的存储空间，变量取名为 a，并初始化为 3。简单说，就是定义一个 unsigned short 类型的变量 a，并初始化为 3。但需要注意的是，变量 a 的值“3”是自然数，没有正号（+）。

下面定义一个存储整数的变量：

```
short int b = 3;
short int c = -5;
```

上述语句定义了 short int 类型的变量 a 和 b，内存大小为 16 个二进制位，可存储 -2^{15}~$2^{15}-1$ 的一个整数。存储整数的变量内存图如图 2.24 所示。

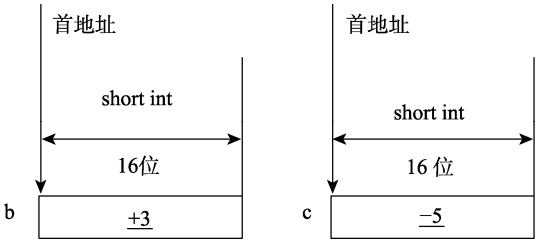


图 2.24　存储整数的变量内存图

2. 标识符和关键字

程序员需要给使用到的每个变量命名，变量名为通用的说法，专业的说法为“标识符”。

标识符由字母、数字、下画线（_）组成，并且第一个字符必须是字母或下画线，不能为数字。大多数计算机语言会区分大写和小写字母，即将大写和小写视为不同的字母，如 C/C++等，但也有些计算机语言不区分大写和小写字母，将大写和小写视为相同的字母。

标识符一般分为关键字、预定义标识符和用户标识符。**关键字**也称系统保留字，是一类特殊的标识符，在计算机语言中有专门的含义，是一个计算机语言的基本符号，决不允许另作他用。预定义标识符是为了程序员使用方便而预先定义的标识符，也指定了特定的含义，最好不要重新定义或另作他用。用户标识符是用户根据自己需要而定义的标识符，可用来命名变量、常量、函数等。

程序中使用标识符命名变量、常量、函数，命名的规范性非常重要，会严重影响程序的可读性和可靠性，命名的规范性也能反映出一个程序员的编程素质。

在 C++的标准文本 ISO/IEC 14882—1998(E)中，共保留了 63 个关键字，如下所示。这些关键字都是英语中的常见单词，很好地体现了其在计算机语言中的含义，提高了程序的可读性。

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

3. 定义变量

变量涉及的运算主要有 3 个，包括定义变量、从变量取值以及将数据存储到变量。

定义变量的基本语法格式为：

```
数据类型  变量名列表;
```

其中，分号(;)表示一条语句结束。

给变量命名应采取“**见名知义，常用从简**”的基本原则。除此之外，还有很多种流行的命名规范可供参考。

例如，在银行程序中，需要为余额、存款、取款交易次数和支票号码定义变量。

```
//银行程序变量定义
float balance;       //分配 float 类型规定的 4 字节（32 位）内存，命名为 balance
float deposit;       //分配 float 类型规定的 4 字节（32 位）内存，命名为 deposit
float withdraw;      //分配 float 类型规定的 4 字节（32 位）内存，命名为 withdraw
//分配 int 类型规定的 4 字节（32 位）内存，命名为 transaction_count
int transaction_count;
int check_number;    //分配 int 类型规定的 4 字节(32 位)内存,命名为 check_number
```

其中，余额、存款和取款都是币值，常用实数表示，依次定义了 balance、deposit 和 withdraw 3 个 float 类型的实型变量，交易次数和支票号码都用整数表示，依次定义了 transaction_count 和 check_number 两个 int 类型的整型变量。定义的银行程序变量如图 2.25 所示。

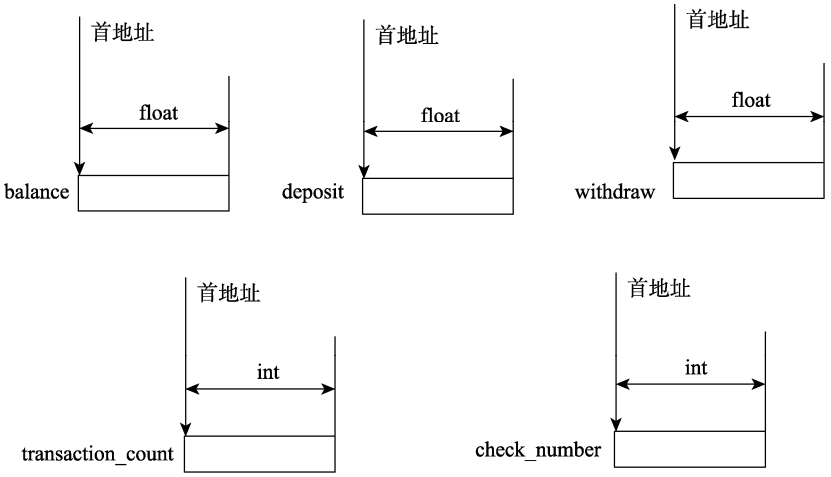


图 2.25　定义的银行程序变量

“//”表示注释，语义为：从“//”开始到本行结束之间的所有字符为注释，其作用是增加程序的可读性。注释是给给程序员阅读的，在编译时会忽略这些信息，因此，如果从源程序中删除注释，不会对程序功能产生任何影响。

程序的可读性，是判断一个程序质量高低最重要的指标之一，在程序中给语句写注释，是程序员的重要工作之一。

可将上面 5 条语句改写为如下两条语句，其语义完全相同。

```
//银行程序变量定义
float balance, deposit, withdraw;
int transaction_count, check_number;
```

在语句 float balance, deposit, withdraw 中，用逗号“,”分隔变量名，语义为依次定义 balance、deposit 和 withdraw 3 个 float 类型的变量。同样，定义了 transaction_count 和 check_number 两个 int 类型的变量。

在定义的同时，可以给变量指定一个初始值，称为变量的初始化。例如：

```
unsigned short width = 5;
```

其中，unsigned short 是 unsigned short int 的缩写，表示数据类型。在变量名后用“=”为所定义变量指定初始值 5。完整的语义为：分配 2 字节的内存，命名为 width，按照 16 位无符号数格式存取数据，并初始化为十进制无符号数 5。变量 width 的内存图如图 2.26 所示。

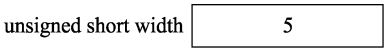


图 2.26　变量 width 的内存图

在定义时也可以初始化多个变量。例如：

```
long width = 7, length = 7;
double area, radius = 23;
```

第 1 条语句 long width=7, length=7 的语义为：分配 4 字节的内存，按照 32 位补码格式存取数据，命名为 width，并初始化为十进制整数 7；分配 4 字节的内存，按照 32 位补码格式存取数据，命名为 length，并初始化为十进制整数 7。

第 2 条语句 double area, radius=23 的语义为：分配 double 类型规定长度的内存，按照 double 类型规定的格式存取数据，命名为 area；分配 double 类型规定长度的内存，按照 double 类型规定的格式存取数据，命名为 radius，并初始化为十进制实数 23。

2.3.2 赋值运算

数学中的代数式包含大量字母，一个代数式规定了一个计算步骤。为了简化计算，一般先采用运算规则化简代数式，然后将每个字母符号对应的数值“代入”字母符号所在的位置，最后计算出代数式的值。

如，简化一个代数式：

$$2a+2b=2(a+b)$$

简化了 $2a+2b$ 代数式，减少了一次乘法运算。如果 a=1、b=2，可将 a、b 的值“代入” $2(a+b)$，计算 2(1+2)。

又如一个函数：

$$f(x,y)=2x+3y$$

计算 f(2,3)的值时，将 2 和 3 分别“代入” $2x+3y$ 中的 x 和 y，计算 2×2+3×3 得到 f(2,3)的值 13。

需要注意的是，前面两个式子都用了等号“=”，但在 $2a+2b=2(a+b)$中，“=”表示演算过程中的两个算式相等，而在 $f(x,y)=2x+3y$ 中，“=”表示函数中的映射，二者的含义不同。

如，用一个字母 z 表示函数值，可将前面的函数改写为

$$z=2x+3y$$

“=”的含义为将数对(x,y)的一个值映射到 z 的一个值。在计算时，将 x、y 的值分别代入 $2x+3y$，然后计算出一个值，这个值就是函数 f 映射到的值。

在计算机中，将 $z=2x+3y$ 中的字母 x、y、z 都视为变量，分配相应的内存，并假设自变量 x、y 的值已存储到变量的内存。计算机在计算函数值时，先从变量 x、y 中分别取出其值，并按照表达式规定的计算序列计算表达式，然后再将计算结果存储到变量 z 的内存，从而实现了函数 f 的映射。

可将上面计算函数值的过程分为两步：第 1 步，计算表达式（前面已学习）；第 2 步，将表达式的值映射到函数的因变量。按照这个思路，计算机语言专门设计了一个运算，将一个表达式的值映射到函数的因变量，这个运算就是赋值运算，其运算符号为“=”。

数学公式中的等号“=”是函数映射，赋值运算实现了数学中函数映射。

使用“=”表示赋值运算，是因为赋值运算的使用频率非常高，为了减少程序员的打字

工作量，就规定用字母“=”代表赋值运算，表示数学中的函数映射，而用“==”代表数学中的相等，表示两个数的比较。这种书写方式已经打破了中学中对算术运算的认知，需要特别注意，决不能将赋值运算符“=”读成“相等”或“等于”。

> 赋值运算符“=”应读作“赋值”，其含义为映射或存入。

在内存中，存数据和取数据是与变量紧密相关的，是变量隐含的两项基本操作，赋值运算实现了对变量的存取操作。

例如：

```
unsigned short  y,x;
x = 5;                          //将自然数 5 赋值给变量 x
y = x;                          //取出变量 x 中的值,然后再赋值给变量 y
```

第 1 条语句定义了两个变量 x 和 y,第 2 条语句 x = 5 的功能为将自然数 5 赋值给变量 x，第 3 条语句 y = x 的功能为取出变量 x 中的值，然后再赋值给变量 y。后面两条语句实现了数学函数 $x = 5$ 和 $y = x$ 定义的映射。

值得注意的是，变量初始化和赋值运算都使用了符号“=”，在大多数情况下，变量初始化与赋值运算在功能上相同，比如上面的第 2 条语句。但在个别情况下，其表示的含义完全不同，比如初始化静态变量，因此需要区分初始化与赋值运算。

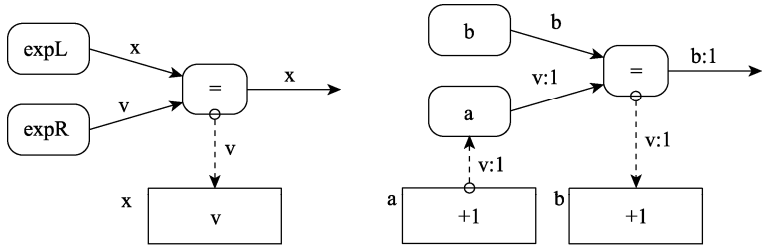
1. 赋值运算的语义

赋值运算语法语义如表 2.2 所示。

表 2.2　赋值运算语法语义

运算符	名称	结合律	语法	语义或运算序列
=	赋值运算	从右到左	expL=expR	计算 expL 得到**变量 x**，计算 expR 得到**值 v**，将值 v 转换为变量 x 的类型规定的存储格式，并存到变量 x 的内存，得到变量 x

赋值运算的语法为 expL=expR，其作用是将表达式 expR 的值存储到 expL 的内存中。赋值运算的语义（或运算序列）为“计算 expL 得到变量 x，计算 expR 得到值 v，将值 v 转换为变量 x 的数据类型规定的存储格式，并存到变量 x 的内存，得到变量 x”。赋值运算=的语义如图 2.27(a)所示。



(a) 赋值运算=的语义　　(b) 表达式b=a的语义

图 2.27　赋值运算的语义

赋值运算规定，左边表达式 expL 的计算结果必须是一个变量 x，赋值运算的运算结果是变量 x，而不是变量 x 中的值 v。这点需要特别注意。

表 2.2 中的语义采用半形式化的方法描述，以方便借用数学中的“代入”对语义进行变换和简化。

例如：

```
short int a = 1, b, c;
b=a;         //从变量 a 的内存中取出值 1,再存放到变量 b 的内存中
c=a + b;     //从变量 a 取出值 1,从变量 b 取值 1,1 加 1 得 2,将 2 存入变量 c 的内存
```

第 2 条语句 b = a 是一条表达式语句。其语义为从变量 a 中取出值 v，将值 v 存储到变量 b 的内存，得到变量 b。程序运行时，将其中的 v 用实际的变量值+1 替换，其实际运行过程为：从变量 a 中读取出值+1，将值+1 存储到变量 b 的内存，得到变量 b。表达式 b=a 的语义如图 2.27(b)所示。

第 3 条语句 c=a+b 也是表达式语句，其语义是从变量 a 中取出值 v1，从变量 b 中取出值 v2, v1 加 v2 得到值 v，直接将值 v 存储到变量 c 的内存，得到变量 c。表达式 c= a+b 的语义如图 2.28 所示。

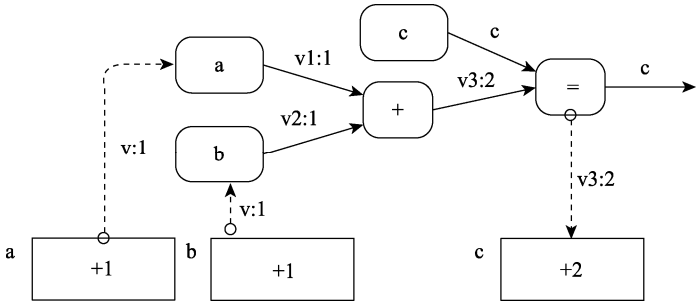


图 2.28　表达式 c= a+b 的语义

当一个变量在赋值运算“=”左边时，其代表变量的内存，用于存储数据，对应的变量操作为存操作，称为变量的“左值”；当一个变量在赋值运算“=”右边时，其代表变量的值，对应的变量操作为从变量的内存中取值，称为变量的“右值”。

2. 复合赋值运算

算术运算都有一个计算结果，而这个计算结果常常需要存储到一个变量中。

例如 a=a*2，其语义为从变量 a 中取出值 v1，v1*2 得到值 v2，将 v2 存储到变量 a，这个语义共包括了 3 个步骤。

程序员在繁重的编程过程中总结出了另一种更加简单的思考方式，即将 2 乘到变量 a，将原来的 3 步变成了一步，这种思考方式最终被广泛接受并写进了 C/C++语言的标准，增加了复合赋值运算。复合赋值减少了运算序列，如图 2.29 所示。

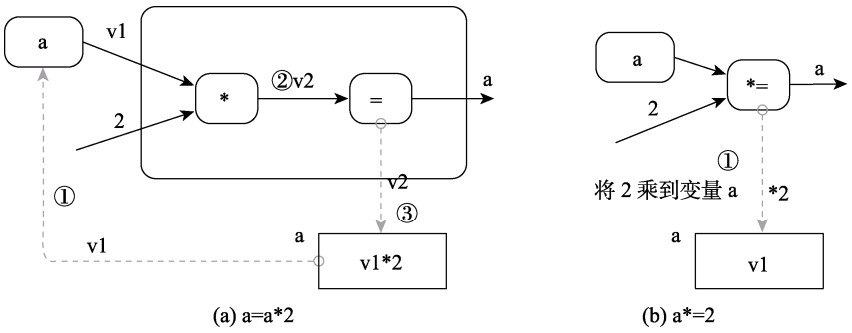


图 2.29　复合赋值减少了运算序列

在赋值运算符“=”之前加上一个运算符，构成一个复合的运算符，以标识一个复合赋值运算。复合赋值运算语法语义如表 2.3 所示。

表 2.3　复合赋值运算语法语义

运算符	名称	结合律	语法	语义或运算序列
*=	multiplication assignment	从右到左	expL*=expR	计算 expL 得到数值类型的变量 x，计算 expR 得到数值 v，将数值 v 乘到变量 x，得到变量 x
/=	division assignment	从右到左	expL/=expR	计算 expL 得到数值类型的变量 x，计算 expR 得到数值 v，变量 x 除以数值 v，得到变量 x
%=	modulus assignment	从右到左	expL%=expR	计算 expL 得到整型变量 x，计算 expR 得到整型值 v，对变量 x 按值 v 取模，得到变量 x
+=	addition assignment	从右到左	expL+=expR	计算 expL 得到数值类型的变量 x，计算 expR 得到数值 v1，将数值 v1 加到变量 x，得到变量 x
–=	subtraction assignment	从右到左	expL–=expR	计算 expL 得到数值类型的变量 x，计算 expR 得到数值 v，从变量 x 减去数值 v，得到变量 x

复合赋值运算的优先级和结合性与赋值运算相同，运算的结果是左边的变量。需要注意的是，复合赋值运算是一个整体，在语义上是一步操作而不能分成多步，目的是减轻程序员的负担。

复合赋值运算+=的语义如图 2.30 所示，计算 expL 得到数值类型的变量 x，计算 expR 得到数值 v1，将数值 v1 加到变量 x，得到变量 x。

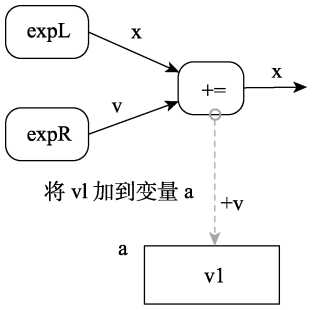


图 2.30　复合赋值运算+=的语义

2.4 整型


视频讲解

整型是计算机使用最多的数据类型，没有之一。程序员在编程时需要理解计算机内部存储整数的方法，并保证计算的正确性。

计算机中都使用二进制来存储自然数和整数，并使用不同的内部存储形式。自然数使用数学上的自然数记数法，没有符号，全部都是数字，而整数一般采用补码，最高位表示整数的符号，其余位数为数字。

2.4.1 理解整数与进制

进制是一种记数的方法，日常生活中主要用到十进制，计算机中使用二进制。程序员不仅需要熟悉十进制、二进制，还需要熟悉八进制和十六进制。

在 x 数轴上表示皮亚诺公理定义的整数，x 数轴上的每个点唯一对应一个整数，可用不同的进制表示这些整数。整数及其表示方法如图 2.31 所示。

0 是 x 数轴上的一个点，将 x 数轴分为左右两边，0 的后继 0′是 x 数轴上右边的第一个点，0 的后继的后继 0″是右边的第二个点，以此类推，用右边的点依次表示所有正整数，同样，0 的前继′0 是左边的第一个点，0 的前继的前继″0 是左边的第二个点，以此类推，用左边的点依次表示所有负整数。

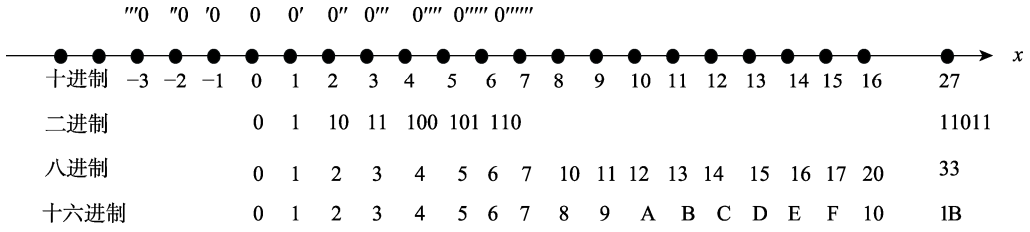


图 2.31 整数及其表示方法

在 x 轴的下面，先用十进制表示整数，然后分别用二进制、八进制和十六进制表示自然数。自然数包括正整数和 0，整数还包括负整数，它们是两种不同的数集。自然数在记数法上只有数字部分，而整数不仅有数字部分，还在数字部分的前面增加了一个符号（正号常省略），它们是两种不同的记数法。

整数的十进制记数法的一般形式为

$$\pm d_n d_{n-1} \cdots d_1 d_0$$

其中，d_i 为 0，1，2，3，4，5，6，7，8，9 中的一个数字。

无论采用十进制还是二进制、八进制或十六进制来表示一个数，数字可能不同，但在坐标上对应的点肯定是相同的，是同一个点。

1. 整数字面值

编程时，除了使用变量来管理数据外，常常需要直接使用整数。在 C/C++中，程序可以

录入用十进制、八进制和十六进制数表示的整数，规定用非 0 数字开头的数字序列表示十进制数，0 开头的数字序列表示八进制数，0X 或 0x 开头的数字和 A，B，C，D，E，F，a，b，c，d，e，f 序列表示十六进制数。例如：

```
int a = 23;                    //十进制数
long int b = 02345;            //八进制数
unsigned int c = 0x79fa;       //十六进制数
```

像上面这样的整数字面值将默认为 int 型整数，即 signed int 型。如果要表示 unsigned int 或者 long int 则可以在整数字面值后面加 U 或 L，大小写都可以，例如：

```
long b = 02345L;               //long int
long c = 235u + 123u;          //unsigned int
```

为了便于读者理解，主要用十进制数来讨论数在计算机中的存储方法。

2. 自然数的固定位数表示法

为了方便内存管理，节约内存资源，计算机中使用固定位数的二进制数来存放和处理数据。为了满足不同场景的需要，常用二进制位数有 8 位、16 位、32 位、64 位。

在固定位数中，无论高位的数值是否为 0，都要写出来，这与人们的常识有很大区别。人们在使用十进制数时，如最高位为 0，往往不会写在纸上，例如，往往只将 010 写成 10。只有在很少的情况下，如写支票时，才被要求在高位写上 0，或做一个标记。

固定位数是通过模运算实现的，下面先介绍模（mod）运算。

模运算定义在自然数集上，两个整数之商的余数就是运算的结果，记为

$$a \bmod b$$

其中，a，b 为自然数，mod 为运算。运算的结果为除法的余数，也形象地称为求余或者取模。

模运算涉及的除法是自然数集上的除法，不是实数集上的除法，16 mod 3 的运算结果是 16 除 3 的余数 1。取余运算和除法运算的计算方法如图 2.32 所示。

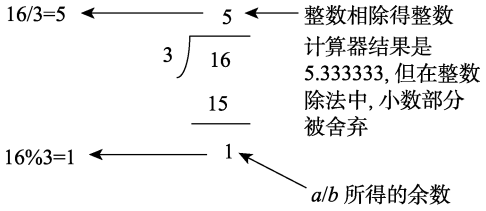


图 2.32　取余运算和除法运算的计算方法

模运算能够将任意一个自然数映射到一个固定大小的区间，如，x mod 8 将 x 的值映射到[0，7]，如图 2.33 所示。

这个特点非常重要，它不仅在计算的内部使用，也被广泛应用到实际编程中。

模运算不仅在数学中具有重要地位，而且是计算机的理论基础。

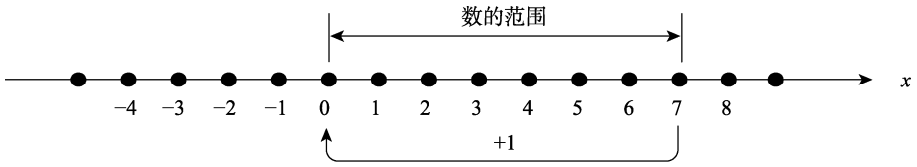


图 2.33　x mod 8 将 x 的值映射到[0,7]

使用固定位数时，为了保证加法、减法和乘法的完备性，都对其计算结果取模，计算公式如下：

$$(x+y) \bmod 2^n$$
$$(x-y) \bmod 2^n$$
$$(x\times y) \bmod 2^n$$

其中，n 表示的是二进制的位数。

例如：

$$(255+1) \bmod 2^8=(255+1) \bmod 256=0$$
$$(255+2) \bmod 2^8=1$$

3. 整数的表示方法

整数表示的核心问题是要解决负数的表示。利用图 2.34 所示的特性，容易找到满足 $(x+y) \bmod 2^n=0$ 的两个自然数。

如果 x 和 y 都是自然数，且满足 $0 \leqslant x<2^n$，$0 \leqslant y<2^n$，则 $(x+y) \bmod 2^n=0$ 成立只有两种情况：一种为 $(x+y)=0$；另一种为 $(x+y)=2^n$。第一种情况，x 和 y 必为 0，不讨论。第二种情况，假设 $x<y$，变形为 $y=2^n-x$，因此，可用 x 表示正整数，用 2^n-x 表示负 x。

当用 8（2^3）取模时，总共有 8 个自然数，用自然数 0、1、2、3 分别表示正整数 0、1、2、3，用自然数 4、5、6、7 分别表示负整数–4、–3、–2，–1。x mod 2^3 时用自然数表示整数的方法如图 2.34 所示。

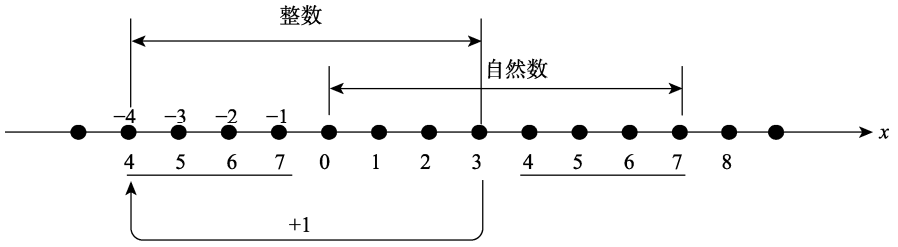


图 2.34　用自然数表示整数的方法

如图 2.34 所示，用$(0,2^2-1)$中的自然数表示正整数，用$[2^2,2^3-1]$中的数表示负整数。

推广到对 2^n 取模，用$(0,2^{n-1}-1)$中的自然数表示正整数，用$[2^{n-1},2^n-1]$中的自然数表示负整数，对任意固定位数的正整数 x，显然有

$$x+(2^n-x)=2^n$$
$$(x+(2^n-x)) \bmod 2^n=0$$

则对 2^n 取模时求 $-x$ 的公式为

$$-x=2^n-x$$

按照这种方法，程序员可以使用上面的公式，采用十进制求出任意正整数的负数。

例如，对 2^8 取模时用(0,127]中的自然数表示正整数，用[128,255]中的数表示负整数。

$$-1=(2^8-1)_{自然数}=(255)_{自然数}$$

$$-6=(2^8-6)_{自然数}=(251)_{自然数}$$

$$-128=(2^8-127)_{自然数}=(128)_{自然数}$$

对 2^8 取模时，用自然数 255 表示最大的负整数–1，自然数 251 表示整数–6，自然数 128 表示最小的整数–128。

例如，对 2^{16} 取模时用(0,32 767]中的自然数表示正整数，用[32 768, 65 535]中的数表示负整数。

$$-1=(2^{16}-1)_{自然数}=(65\ 535)_{自然数}$$

$$-6=(2^{16}-6)_{自然数}=(65\ 530)_{自然数}$$

$$-32\ 768=(2^{16}-32\ 768)_{自然数}=(32\ 768)_{自然数}$$

对 2^{16} 取模时，用自然数 65 535 表示最大的负整数–1，自然数 65 530 表示整数–6，自然数 32 768 表示最小的整数–32 768。

上面的示例使用了十进制表示自然数和整数，实际上，求出任意正整数负数的公式是与进制无关的，可将其推广到 R 进制，对 R^n 取模，求 $-x$ 为

$$-x=R^n-x$$

计算出的 $-x$ 是一个自然数，因此，在对 R^n 取模的情况下，可以使用加法来进行减法运算，将整数的加法和减法统一为自然数的加法运算，其计算公式为

$$(x-y) \bmod R^n = (x+(-y)) \bmod R^n$$

计算机按照上述的原理针对二进制设计了补码，并用自然数的加法运算实现了整数的加减法运算。在 10.1.3 节中将专门讨论补码。

2.4.2　整数的数据类型

客观世界中存在着各种各样的数据，计算机对不同类型的数据采用不同的二进制表示方法。为了区分不同的表示方法，引入了数据类型的概念，用以表示客观世界中的数据与二进制之间相互转换的具体方法以及所需的内存大小。

在 C/C++中，signed 和 unsigned 分别区分的是整数和自然数，signed 称为有符号整数，unsigned 称为无符号整数。如果没有指定是有符号整数还是无符号整数，则编译器自动默认为有符号整数。整数的常见数据类型如表 2.4 所示。

按所需内存大小进行分类，整型常用的长度有 8 位、16 位、32 位，在 C/C++中，分别用 char、short int、long int 来表示。随着计算机的发展，整型的位长也在增加，如 64 位、128 位等。查阅语言或编译器的手册，可了解具体规定。

表 2.4　整数的常见数据类型

二进制位数	有符号整数（整数）	无符号整数（自然数）
8 位	signed char	unsigned char
16 位	signed short int	unsigned short int
32 位	signed long int	unsigned long int
字长	signed int	unsigned int

例如：

```
unsigned short a = 3;
```

定义了一个变量 a，数据类型为无符号整数（unsigned short int，其中 int 可省略），用来存储自然数，内存大小为 16 个二进制位，只能存储 0~2^{16}–1 的数。

这条语句的功能是定义一个 unsigned short 类型的变量 a，并初始化为 3，其运算顺序为：在内存为一个变量分配两个字节（16 位）的存储空间，变量取名为 a，并初始化为 3。

计算机在运行程序时，才为变量 a 分配 16 个二进制位长度的内存空间，并将自然数 3 转换为二进制串 00000000 00000011，然后存储到分配的内存区域。无符号整型变量 a 中存储的二进制串如图 2.35 所示。

为了方便，在变量的内存图中，可用十进制表示变量的值，但需要注意的是，其中的值“3”是自然数，不能有正号“+”。无符号整型变量 a 中存储的自然数如图 2.36 所示。

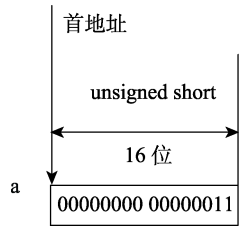


图 2.35　无符号整型变量 a 中存储的二进制串

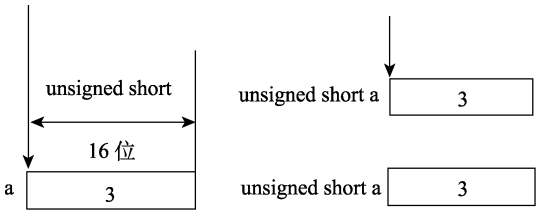


图 2.36　无符号整型变量 a 中存储的自然数

图 2.36 中，还给出了两种简化的变量内存图，它们表达的意思相同，可根据具体情况选用其中一种。

例如：

```
short int a = 3;
```

其中，short int 为 signed short int 的简写，signed 单词的意思为带符号。数据类型 signed short int 规定使用 16 个二进制位存储整数的补码。

存储格式为 $sb_{14}\cdots b_1b_0$，其中，最高位 s 为符号位，0 表示正数，1 表示负数，0~14 位 $b_{14}\cdots b_1b_0$ 表示数字，总共 15 位，存储的整数范围为 $-2^{15}\sim 2^{15}-1$。

语句 short int a = 3 定义了一个有符号整型变量 a，并初始化为+3。变量 a 的内存中实际存储的是其补码 00000000 00000011。特别注意，补码的最高位为符号位，0 表示正整数，这与 unsigned short 类型不同。有符号整型变量 a 中存储的补码和整数如图 2.37 所示。

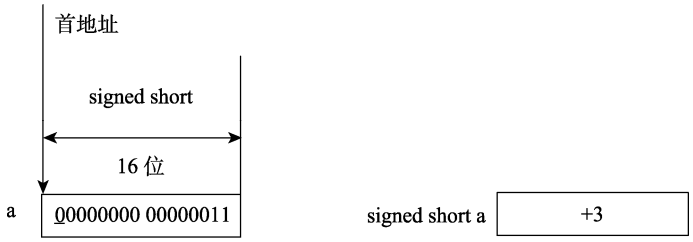


图 2.37　有符号整型变量 a 中存储的补码和整数

C/C++为整数提供了比较丰富数据类型，以适合不同的场景，但每种数据类型有自己的内部存储格式，可编程观察整数内部存储格式，代码如例 2.1 所示。

【例 2.1】 整数内部存储格式。

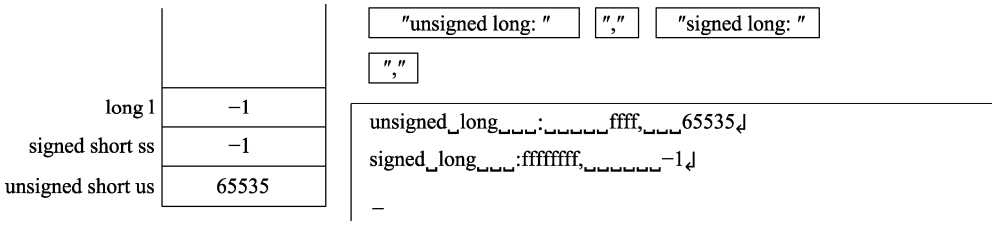
```
#include<iostream>
#include<iomanip>
using namespace std;
void main(){
    unsigned short us = 65535;
    signed short ss = -1;

    long l;
    l = us;//无符号位,高位补 0
    cout << "unsigned long   :";
    cout << setw(8) << hex << l << "," << setw(8) << dec << l << endl;

    l = ss;//符号扩展，高位补 1
    cout << "signed long   :";
    cout << setw(8) << hex << l << "," << setw(8) << dec << l << endl;

}
```

运行中的内存和输出结果：



运行中的内存和输出结果包括三部分，第一部分是右上角的全局数据区，用于存放字符串常量，操作系统在程序运行前会将程序中的"unsigned long:"、"," 、"signed long:"、","等字符串常量存入分配的内存空间中，需要注意的是，这些内存单元不一定是连续的，因此画出的几个内存单元并没有紧挨在一起；第二部分是左边的内存区域，用于存放变量；第三部分是右下角的屏幕区域，用于显示程序的输出结果。

程序中定义了变量 us 和 ss，变量 ss 的数据类型为有符号短整型（signed short int），用自然数 2^{16}–1 表示整数–1，2^{16}–1 刚好就是 65 535，因此两个变量在内存中存储的都是 65 535，

对应的二进制就是全 1。

例 2.1 程序中的如下代码分别按照十六进制和十进制输出变量的值。

```
cout << "signed long   :";
cout << setw(8) << hex << l << "," << setw(8) << dec << l << endl;
```

其中，setw(8)中的 setw 为 setwidth 的缩写，语义为设置显示宽度，setw(8)的语义为设置显示宽度为 8 个字符。hex 的语义为设置为十六进制，后面以十六进制显示变量 l 的内存中存储的二进制串。dec 的语义为设置为十进制。endl 为回车符换行。其中输出结果为：

```
unsigned long : ␣␣␣␣ffff, 65535;
```

变量 l 中存储的仍然是十进制数 65 535，但内部存储的是 0000ffff。从这个显示结果中，充分体会了十六进制的优越性。

例 2.1 程序中的如下代码，先将变量 ss 的值赋值给变量 l，然后分别以十六进制和十进制输出变量的值。

```
l = ss;   //符号扩展,高位补 1
cout << "signed long   :";
cout << setw(8) << hex << l << "," << setw(8) << dec << l << endl;
```

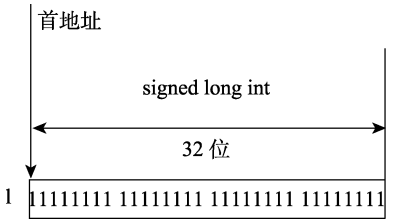


图 2.38　使用符号扩展有符号数的位数

变量 ss 和 l 都是有符号整型，但长度不同，需要从 16 位扩展到 32 位，将–1 的内部表示 2^{16}–1 转换为 2^{32}–1，将 2^{32}–1 用二进制表示出来就是 32 个 1。使用符号扩展有符号数的位数，如图 2.38 所示。

屏幕上的输出结果为：

```
signed␣long␣␣␣:ffffffff,␣␣␣␣␣␣–1↵
–
```

变量 l 中存储的仍然是十进制数–1，但内部存储的是 ffffffff。

前面用字母 l 来命名变量，容易与数字 1 混淆，因此，坚决不要使用字母 l 单独命名变量。

字长是描述计算机处理能力的最主要指标，它表示计算机处理器（CPU）一次处理数据的二进制位数，如在 32 位机上进行一次 32 位的加法运算，其花费的时间不比进行一次 16 位的加法运算长，也不比进行一次 8 位的加法运算长，因此，为了充分发挥计算机的处理能力，并且保证程序的兼容性，大多数计算机语言提供了通过计算机字长来指定整数位数的功能。

在 C/C++的标准文本中，没有规定 int 类型的位数，让编译器按照计算机字长来指定 int 类型的位数，以提高代码的运行效率，因此，在对整数位数没有具体要求的场景下，强烈推荐使用 int 类型。

2.4.3　自增和自减运算

自增和自减是程序员非常喜欢的运算，自增和自减运算语法的语义如表 2.5 所示。

表 2.5　自增和自减运算语法的语义

运算符	名称	结合律	语法	语义或运算序列
++	后自增（后++）	从左到右	exp++	计算 exp 得到变量 x，将 1 加到变量 x，返回原来的值 v
––	后自减（后––）	从左到右	exp––	计算 exp 得到变量 x，将变量 x 减 1，返回原来的值 v
++	前自增（前++）	从右到左	++exp	计算 exp 得到变量 x，将 1 加到变量 x，得到变量 x
––	前自减（后––）	从右到左	––exp	计算 exp 得到变量 x，变量 x 减 1，得到变量 x

自增和自减运算区别于其他非赋值运算的地方在于，它的操作对象必须是变量，自增运算后，变量的值增加 1，自减运算后，变量的值减少 1。

自增和自减运算有前增量和后增量之分：一种是操作数在后操作符在前，俗称前++或前––；另一种是操作数在前操作符在后，俗称后++或后––。前++或前––运算后，得到的是变量，而后++或后––运算后，得到的是变量中原来的值。这两种形式的区别初学者往往会混淆，需要深入掌握。

1. 前++和前––

在实际编程中，常常要将 1 加（减）到一个变量，如 a+=1。为了在编程时输入更少的字符，计算机语言专门定义了两个运算，分别表示为“+=1”和“–=1”，称为前自增和前自减。因在语法上运算符在操作数的前面，因此，俗称前++和前––运算。

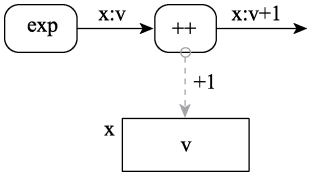


图 2.39　前++的语义

前++的语法为++exp，其语义完全等价于“exp+=1”，其优先级非常高，这样不需使用括号来提高其优先级，写出的表达式会更加简洁。前++的语义如图 2.39 所示。

前++与前––的语法、语义都非常类似，只是一个是减，一个是加。

下面以前++为例，介绍前++与前––的使用方法。示例代码如例 2.2 所示。

【例 2.2】 前++示例。

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

void main(){
    int a = 0, b;

    b = ++a;    //将 1 加到变量 a,得到变量 a;从变量 a 中,取出值 1,赋值给变量 b
    cout << a << " , " << b << endl;

    ++a = 0;    //(++a)的结果为变量 a
    cout << a << endl;
```

```
    b = (a = a + 1); //从变量 a 中取出 0,0+1 得 1,将 1 存到变量 a,得到变量 a
    cout << a << " , " << b << endl;
}
```

程序的难点在于 b = ++a、++a = 0 和 b = (a = a + 1) 3 个表达式语句，其中，表达式 b = ++a 的语义如图 2.40 所示。

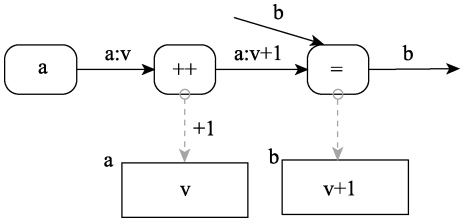


图 2.40　表达式 b = ++a 的语义

3 个表达式的计算顺序如图 2.41 所示。

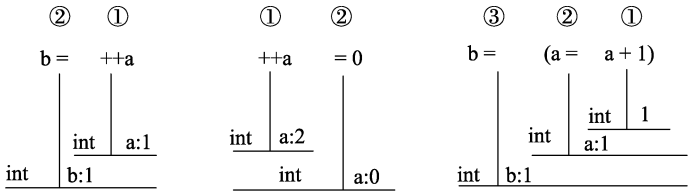
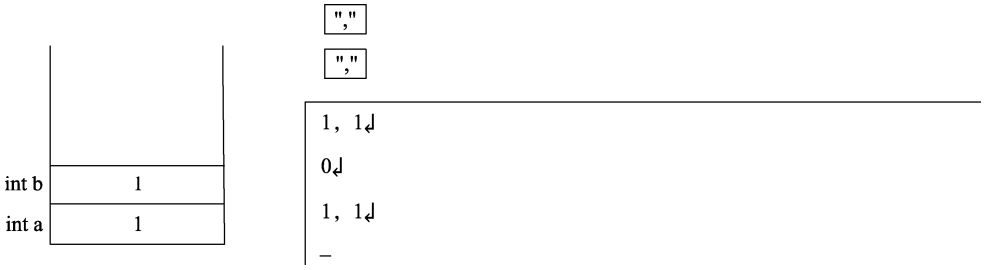


图 2.41　3 个表达式的计算顺序

例 2.2 程序的内存状态和输出结果：



“++a”和“a = a + 1”的结果完全一样，但“++a”只需要程序员思考一步，而“a = a + 1”需要思考 3 步，这就是程序员喜爱自增自减的原因。

2. 后++和后––

前++和前––的运算结果都是变量，变量中存储的是计算后的值，但有时需要使用计算前的值进行后面的运算。为了满足这个编程需求，又定义了后++和后––两个运算。

在语法上，为了与前++区别，后++规定将运算符放在操作数的后面，即 exp++。在语义上，前面两步“计算 exp 得到变量 x，将 1 加到变量 x”与前++完全相同，但后++“返回原来的值 v”，而不是变量 x。后++的语义如图 2.44 所示。

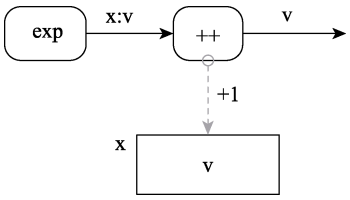


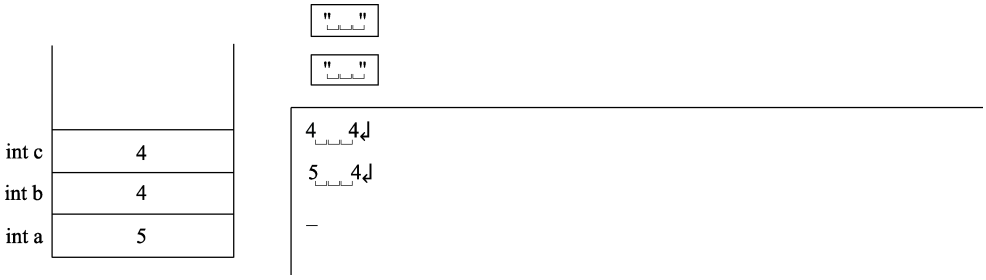
图 2.42　后++的语义

下面通过示例来演示前++和后++的使用方法，代码如例 2.3 所示。

【例 2.3】 前++和后++示例。

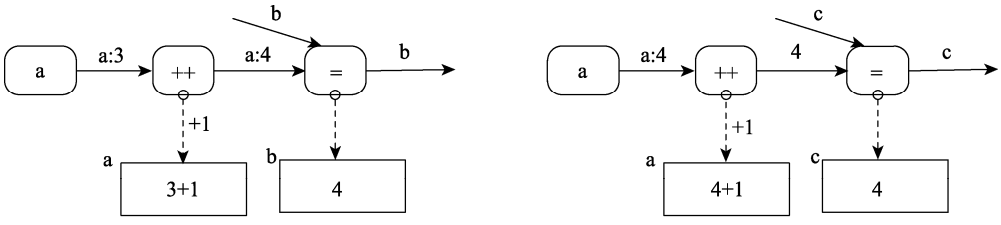
```
#include<iostream>
using namespace std;
void main(){
    int a = 3, b, c;
    b = ++a;
    cout << a << "   " << b << endl;
    c = a++;
    cout << a << "   " << c << endl;
}
```

例 2.3 程序的内存状态和输出结果为：



表达式 b=++a 的语义和计算过程如图 2.43(a)所示，运行 b=++a 后，变量 a 的值为 4，变量 b 的值为 4。

表达式 c=a++的语义和计算过程如图 2.43(b)所示，运行表达式 c = a++后，变量 b 的值为 4，变量 c 的值为 4，变量 a 的值为 5。



(a) 表达式b=++a的语义和计算过程　　(b) 表达式c=a++的语义和计算过程

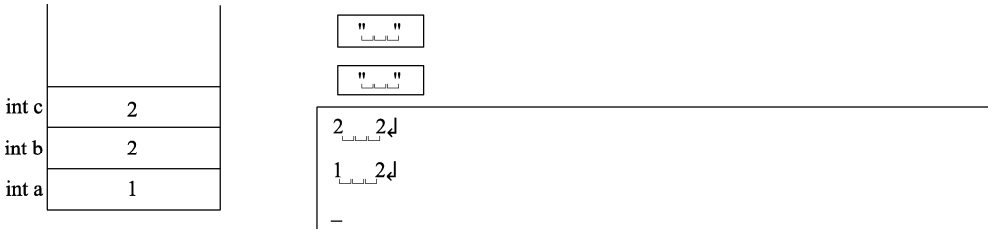
图 2.43　表达式 b=++a 和 c=a++的语义和计算过程

下面通过示例来演示前--和后--的使用方法，代码如例 2.4 所示。

【例 2.4】 前--和后--示例。

```
#include<iostream>
using namespace std;
void main(){
    int a = 3;
    int b = --a;
    cout << a << "   " << b << endl;
    int c = a--;
    cout << a << "   " << c << endl;
}
```

例 2.4 程序的内存状态和输出结果为：



赋值和自增自减运算都会改变变量的值，在一个表达式中可以出现多次，可以在一个表达式中多次修改多个变量的值，但这会导致程序的可读性变差，甚至出现语义混乱，因此，建议在一个表达式中最多对一个变量修改一次值，绝不要在一个表达式中多次修改一个变量的值。


视频讲解

2.5 字符型

在计算机中，不仅要进行数的计算，还需要处理大量文字信息。在英文中，构成文字的基本要素为字母；在中文中，构成文字的基本要素为字。构成文字的字母或字，称为字符。

计算机要处理文字信息，首先要解决的问题是在计算机中怎样表示和存储构成文字的字符。

从本质上讲，计算机中只能存储和处理二进制串，只能存储和处理整数，因此，不能直接存储和处理文字信息中的字符。解决这个问题的方法为，在字符的集合和自然数之间建立一个映射，用一个自然数唯一表示一个字符，计算机存储和处理这些自然数。

一般用表格形式表示字符和自然数之间的映射，就形成一个表示字符与自然数对应关系的表格，称这种表为字符表，也称为字符集。

2.5.1 字符集

目前的文字有很多种，构成各种文字的字母或字也不一样，这就需要针对不同的文字建立不同的字符集，另外，针对同一种文字，也可能有不同的编码方法，这样就产生了很多字符集。

下面只介绍 ASCII 码和国标码两种常用的字符集，读者可自行查阅其他字符集。

1. ASCII 码

ASCII （American Standard Code for Information Interchange，美国标准信息交换代码）是由美国国家标准学会（American National Standard Institute，ANSI）制定的，是一种标准的单字节字符编码方案，后来它被国际标准化组织（International Organization for Standardization，ISO）定为国际标准，称为 ISO 646 标准。

ASCII 码使用 1 字节 8 比特位对字符进行编码，分为标准 ASCII 码和扩展 ASCII 码，标准 ASCII 码的最高位为 0，扩展 ASCII 码的最高位为 1。实际上，这种编码方案主要规定了标准 ASCII 码中的 128 个字符的编号，包括所有的大写和小写字母、数字 0~9、标点符号，以及常用的特殊控制字符。扩展 ASCII 码用于扩展，用来表示附加的 128 个特殊符号字符、外来语字母和图形符号等。

目前主要使用标准 ASCII 码，一般不使用扩展 ASCII 码。附录 A 给出了标准 ASCII 码表，总共有 128 个字符。

2. 国标码

我国国家标准局于 1981 年 5 月颁布了《信息交换用汉字编码字符集——基本集》，代号为 GB 2312—1980，共对 6763 个汉字和 682 个图形字符进行了编码。

编码原则为一个汉字用 2 字节表示，前字节的编码称为区码，后字节的编码称为位码，也称为区位码，每个字节只用 7 位码，为了确保与标准 ASCII 码不冲突，将每个字节的最高位设置为 1。国标码结构如图 2.44 所示。

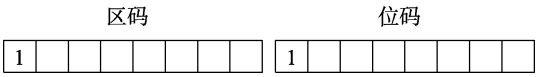


图 2.44　国标码结构

前面介绍了 ASCII 码和汉字国标码两种字符集，一个是单字节码，一个是双字节码，是目前常用字符编码的典型代表，汉字国标码使用的是 ASCII 码的扩展编码空间，一个汉字占用 2 字节，与标准 ASCII 码完全兼容，因此，本书中讨论字符型时，都采用 ASCII 码，但程序中也可使用汉字，汉字在内存中占用连续的 2 字节，每个字符相当于一个字符型变量，换句话说，一个汉字使用两个字符数据表示，占用 2 字节。

2.5.2　使用字符型

字符型专门为存储和处理字符而设计，占用一字节，实际上存储的是字符的编号，是一个 8 位无符号整数，可以使用整型中学习的方法使用字符型，并对字符型变量中存储的字符进行处理。

1. 字符字面量

在计算机语言中，字符的表示方法很简单，但为了与数字、运算符和变量名等相区别，

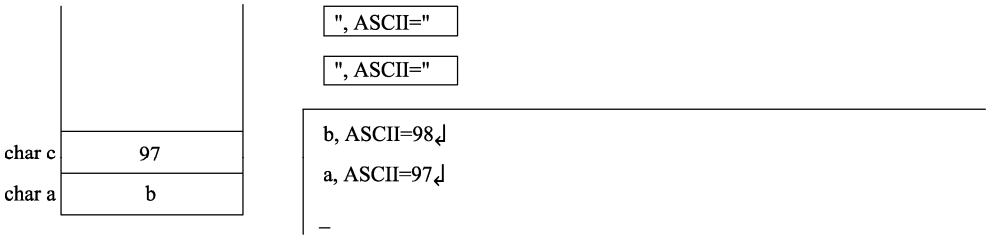
一般规定必须用单引号（''）将一个字符引起来，如， 'a'表示小写字母 a；'B'表示大写字母 B；'9'表示字符 9，而不是数字 9；'+'表示字符+，而不是运算符+。

下面通过示例来演示字符型的使用方法，代码如例 2.5 所示。

【例 2.5】 字符型示例。

```
#include<iostream>
using namespace std;
void main(){
    char a = 'b';   //定义一个字符变量 a,并用字母 b 的 ASCII 码初始化
    cout << a << ", ASCII=" << (int)a << endl;
    char c = 97;    //定义一个字符变量 b,并用整数 97 初始化
    cout << c << ", ASCII=" << (int)c << endl;
}
```

例 2.5 程序的内存状态和输出结果为：



char a = 'b'的语义为，定义一个 char 类型的变量 a，也就是一个 8 位整型变量，然后初始为字符'b'，即将小写字母 b 的 ASCII 码值 98 存储到变量 a 的内存。char c = 97 中，用整数 97 初始化变量 c，整数 97 刚好是字母 a 的 ASCII 码。

cout << a <<", ASCII=" <<(int)a<<endl 中，执行 cout << a 时，因变量的 a 的类型为字符型，cout 从变量 a 中取出整数 98，然后输出 98 在 ASCII 码表中对应的字母 b。执行到 cout << (int)a 时，因增加了一个类型转换，cout 将按照 int 类型来输出，输出变量 a 中存储的 ASCII 码值 98。

在编写程序时，不仅可以用单引号“''”来表示字符，也可以用其 ASCII 码值来表示一个字符。从理论上讲，使用 ASCII 码值可以表示字符集中任意一个字符，但程序员在编写程序时，需要查阅或记住字符的 ASCII 码值，因此，程序员还是喜欢使用单引号表示方式。

2. 转义字符

控制字符或通信专用字符，也称为不可见字符，其 ASCII 码值为 0~31 及 127，共有 33 个。这些字符不能像数字、字母等字符一样在源程序中呈现出来，而是有特定的用途，换句话说，这些字符不能从键盘输入到源程序中单引号的中间。为了解决这个问题，C/C++提供了专门表示这类字符的方法，俗称转义字符（escape character）。用一个特定的字符'\'来改变可见字符的含义，变成不可见字符。

例如，换行符用'\n'表示键盘中的回车符（Enter 键），其中，n 为 newline 的第一个字母，程序员很好记，这就是程序员偏爱它的原因。常用的 C/C++转义字符如表 2.6 所示。

表 2.6　常用的 C/C++转义字符

字符形式	整数值	代表符号	字符形式	整数值	代表符号
\a	0x07	响铃(bell)	\"	0x22	双引号
\b	0x08	退格(backspace)	\'	0x27	单引号
\t	0x09	水平制表符(HT)	\?	0x3F	问号
\n	0x0A	换行(return)	\\	0x5C	反斜杠字符\
\v	0x0B	垂直制表符(VT)	\ddd	0ddd	1~3 位八进制数
\r	0x0D	回车	\xhh	0xhh	1~2 位十六进制数

可使用整数运算处理字符类型的数据。如下例程将大写字母 A 转换为小写字母 a，代码如例 2.6 所示。

【例 2.6】 大小写转换。

```
#include<iostream>
using namespace std;
void main(){
     char ch = 'A';
     ch += 32;
     cout << '\n' << ch;
}
```

想在输出字母前输出一个空行，但因回车符不能录入到代码，就用了转义字符'\n'。

例 2.6 程序的内存状态和输出结果为：

2.6　实数型

前面从自然数、整数出发，重点学习了整型和字符型两种数据类型，下面也从实数开始，学习实数型。与整型相比，实数型是比较复杂的数据类型，但不需要程序员直接管理其中的数据，而由计算机中专门的硬件来处理，因此，使用相对更加简单。

2.6.1　浮点数记数法

计算机中采用浮点数记数法表示和存储实数。

1. 十进制浮点数记数法

十进制浮点数记数法来源于实数的科学记数法，其一般形式为

$$\pm 0.d_1 d_2 \cdots d_m \times 10^{\pm e_n \cdots e_1 e_0}$$

其中，最高位前面为符号，$d_1d_2\cdots d_m$ 为小数部分，小数点后第 1 位 d_1 不能为 0，$\pm e_n\cdots e_1e_0$ 为指数部分，d 和 e 为数字 0、1、2、3、4、5、6、7、8 或 9。与实数的科学记数法相比，十进制浮点数记数法要求整数部分全为 0，小数点后第 1 位非 0。

根据十进制浮点数记数法，当一个十进制浮点数的符号、小数和指数部分确定后，这个数就确定了，因此，只需存储浮点数的符号、小数和指数。浮点数的存储格式如图 2.45 所示。

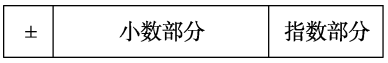


图 2.45　浮点数的存储格式

例如，19 971 400 000 000=+1.997 14×10^{13}=+0.199 714×10^{14}，该正浮点数的存储格式如图 2.46 所示。

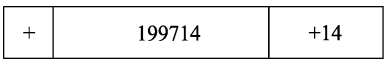


图 2.46　正浮点数的存储格式

在计算机中，为了方便，一般用 E 或 e 表示 10 的幂，如，+0.199 714×10^{14} 表示为 0.199 714E14。

例如，–306.5=–0.3065*10^3，该负浮点数的存储格式如图 2.47 所示。

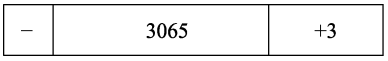


图 2.47　负浮点数的存储格式

其中，–是符号，指数 3 称为阶或阶码，3065 是小数部分，其左右端非 0 数字包起来的最长的数字序列称为有效值（significance），这里的有效值是 3065。小数部分也称为尾数，如 3065 是尾数。

十进制浮点记数法中，一个实数越大，指数就越大，指数的位数也就越多，同样，实数的精度越高，小数的位数就越多；反过来讲，指数的位数越多，能表示的实数就越大，小数的位数越多，能表示实数的精度就越高。

2. 二进制浮点数及存储格式

计算机内部使用二进制浮点数存储实数，二进制浮点数记数法与十进制浮点数类似，包括符号、小数和指数 3 部分，一般形式为

$$\pm 0.b_1b_2\cdots b_m\times 2^{\pm a_n\cdots a_1a_0}$$

其中，最高位前面为符号，$b_1b_2\cdots b_m$ 为小数部分，小数点后第 1 位 b_1 不能为 0，$\pm a_n\cdots a_1a_0$ 为指数部分。a 和 b 为数字 0 或 1。二进制浮点数的存储格式如图 2.48 所示。

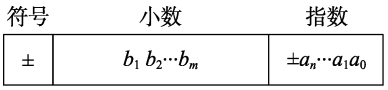


图 2.48　二进制浮点数的存储格式

国际标准 IEEE 754 规定了具体二进制浮点数的存储形式，依次为符号、指数和小数 3 部分，并分别规定了 32 位浮点数和 64 位浮点数中指数和小数的二进制位数。32 位浮点数，符号占 1 位，小数部分占 23 位，指数部分占 8 位，用 32 位二进制位表示一个实数，所表示的实数精度较低，常常称为单精度浮点数，对应 C/C++的 float 类型。64 位浮点数，符号占 1 位，小数部分占 52 位，指数部分占 11 位，用总共 64 位表示一个实数，因所表示的实数精度较高，常常称为双精度浮点数，对应 C/C++中的 double 类型。

2.6.2　实数型分类

在编程实践中，实数主要用于科学计算，有关浮点数的运算算法比较复杂，时间复杂度也很高，计算机中通常有专门的硬件来处理浮点运算，一般不需要程序员深入理解浮点数在计算机内部存储和运算的机制，只需要重点关注两点：第一点，浮点数的大小范围；第二点，浮点数的精度。如果一个实数超出了浮点数的大小范围，就会发生溢出，导致不正确的结果；如果精度太小，计算结果的误差会很大，达不到实际要求。C/C++中的实数型如表 2.7 所示。

表 2.7　C/C++中的实数型

类　型	名　称	位　数	字节数	范　围	有效位数
float	单精度浮点数	32	4	$\pm 3.4\times10^{38}$	7 位
double	双精度浮点数	64	8	$\pm 1.8\times10^{308}$	15 位
long double	长双精度浮点数	80	10	$\pm 1.2\times10^{4932}$	19 位

2.6.3　实数的字面表示

浮点数既可以表示为定点方式（非指数方式），例如 35.623，也可以表示成科学记数法（指数方式），例如 0.35623e+02，即 0.35623 乘以 10 的 2 次方。直接写出的浮点数字面值默认为 double 型，如果要表示成 float 型，则要在浮点数后面加上字母 F 或 f。例如：

```
float f1 = 19.2f;
float f2 = 0.192e+02;          //将 double 数转换为 float
double d1 = 19.2;
double d2 = 0.192e+02f;        //将 float 数转换为 double
long double ld1 = 19.2L;
long double ld2 = 0.192e+02;  //将 double 数转换为 long double
```

2.6.4　实数型的精度和范围

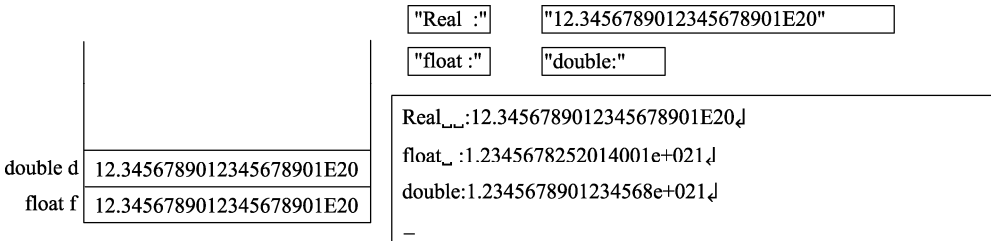
计算机中只能存储实数的近似值，因此，应重点关注实数的精度和大小范围。

下面通过示例来演示实数型的精度及大小范围，代码如例 2.7 所示。

【例 2.7】 实数型的精度及大小范围。

```
#include<iostream>
#include<iomanip>
using namespace std;
void main(){
    cout << setprecision(18) << "Real  :" << "12.3456789012345678901E20"
<< endl;
    float f = 12.3456789012345678901E20;
    cout << setprecision(18) << "float :" << f << endl;
    double d = 12.3456789012345678901E20;
    cout << setprecision(18) << "double:" << d << endl;
}
```

运行中的内存和输出结果：



从输出结果可以看出，float 和 double 输出的都是近似值，float 类型的正确位数为 8，说明 float 类型的有效位数不超过 8 位，double 类型的正确位数为 18，说明 double 类型的有效位数不超过 18 位。

读者可修改 12.345 678 901 234 567 89 01E20 中的指数，观察各类数型的表示范围。


视频讲解

2.7 算术类型转换

在数学中，运算是定义在一个特定数集上的，与数集及相应的记数法紧密相关，而计算机中是用数据类型区分不同的数据集合，不论变量还是值都有一个特定的数据类型。

自然数、整数和实数上都有加减乘除运算，但它们在不同数集上有不同的具体计算方法，在本质上是不同的运算。人具有很高的智力水平，往往会忽略它们之间的差别，但计算机没有这样高的智力水平，只能对相同数据类型的数据进行加减乘除运算。

为了对不同数据类型的数据进行加减乘除运算，计算机语言专门设计了一个运算来解决数据类型的相互转换问题，这个运算称为数据类型转换。

2.7.1 整数的数据类型转换

数据类型规定了数据在计算机中的内部格式和所需的二进制位数。进行四则运算时，当两个操作数的内部格式和二进制位数相同时，才能直接运算，否则需要先进行数据类型转换。

下面以 8 位有符号整数在 16 位机上进行相加为例，讨论数据类型转换。

例如：

```
signed char a = 5, b = -4, c;
c = a + b;
```

第 1 条语句，在内存中定义了 a、b、c 3 个有符号整型变量，每个变量存储一个字符，其中，变量 b 初始化为–4，变量 b 的内存中存储的是–4 的补码 1111 1100。

在 16 位机上执行第 2 条语句 c=a+b，包含 4 个步骤。第 1 步，从变量 a 的内存中取出值 00000101 到 CPU，并按照符号扩展方式扩展为 0000000000000101；第 2 步，从变量 b 的内存中取出值 11111100 到 CPU，并扩展为 1111111111111100；第 3 步，两个二进制数在 CPU 中相加得到 0000000000000001；第 4 步，将计算结果的高 8 位 00000000 扔掉（取模运算）而将低 8 位 00000001 存储到变量 c 的内存。在 16 位机上执行 c=a+b 的过程如图 2.49 所示。

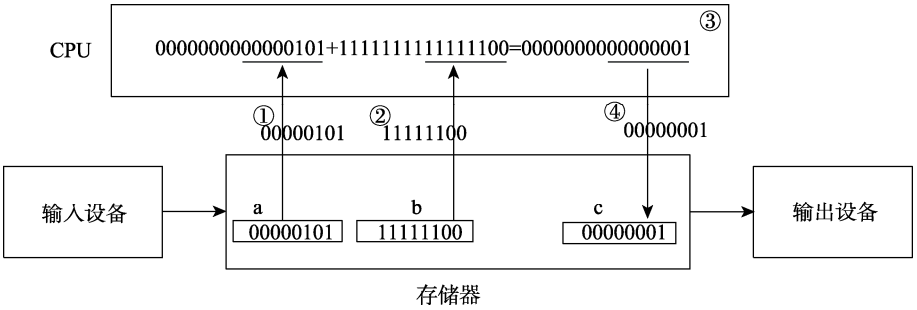


图 2.49　在 16 位机上执行 c=a+b 的过程

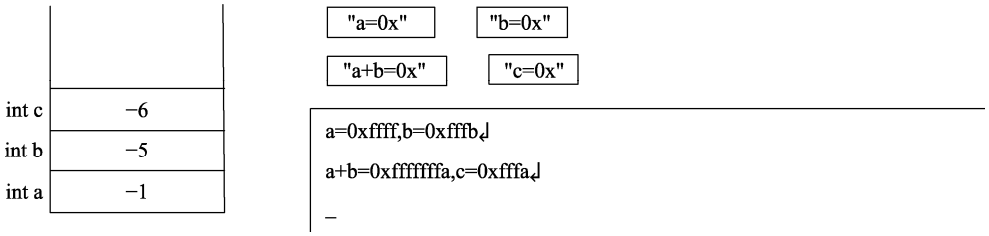
在 16 位机上执行 c=a+b 时，先将 8 位整数转换为 16 位整数，然后再按照 16 位整数进行加法运算，最后将计算结果中的低 8 位存储到变量 c。在计算过程中，自动进行了 3 次数据类型转换。

可编写程序，观察计算表达式 c=a+b 过程中各变量的内存存储数据，以及数据类型转换。整数的内部存储数据及数据类型转换代码如例 2.8 所示。

【例 2.8】 整数的内部存储数据及数据类型转换。

```
#include<iostream>
#include<iomanip>
using namespace std;
//测试整数运算中数据类型的转换
void main(){
    signed short a = -1, b = -5, c;
    cout << "a=0x" << hex << a << "," << "b=0x" << hex << b << endl;
    c = a + b;
    cout << "a+b=0x" << hex << a + b << ","
         << "c=0x" << hex << c << endl;
}
```

例 2.8 程序在 VS2013 的 32 位编译器下的内存状态和输出结果为：



例 2.8 程序的输出结果中，a=0xffff,b=0xfffb 表明变量 a 和 b 中存储的是 16 位二进制数，a+b=0xfffffffa 表明按照 32 位二进制进行加法运算，但 c=0xfffa 表明将 a+b 的计算结果 0xfffffffa 中的低 16 位 0xfffa 存储到变量 c 的内存。

每个计算机的字长是衡量计算机处理能力的重要指标，目前主流个人计算机的字长为 32 位或 64 位，有些单片机的字长为 8 位或 16 位。不论计算机的字长是多少，每个计算机的字长一定是固定的，即 CPU 每次运算的二进制位都是固定的，不能多，也不能少。字长为 16 位的计算机，每次加法运算都是 16 个二进制位相加；字长为 32 位的计算机，每次加法运算都是 32 个二进制位相加。如果在 16 位机上进行 8 位二进制数的加法，需要先将操作数扩展到 16 位，然后再相加；如果进行 32 位加法，则需要将 32 位拆分为两个 16 位来运算。

例 2.8 所示的程序，在不同版本的编译器上编译，会有不同的输出结果，读者可以多试几种编译器，观察其输出结果，分析计算的过程，加深对计算的理解。

2.7.2 算术运算的自动类型规则

为了方便程序员编程，C/C++等计算机语言都提供自动类型转换功能，自动转换操作数的数据类型以满足运算对数据类型的要求。

自动类型转换的一般原则为，位数少的向位数多的转换，有符号向无符号转换，整型向实数转换，以尽量保证信息不丢失，同时兼顾运算的效率。

int 是 C/C++中比较特殊的整型，标准中没有具体规定其二进制位数，其二进制位数往往由编译器根据计算机的字长决定。随着计算机制造技术的发展，常用的个人计算机字长都比较长，至少是 32 位，目前常见的编译器也很多是 32 位的，也有些是 64 位的。因此，int 的长度一般都会不少于 32 位，大多数情况下都会将整型数据转换为 int 类型，这样能充分发挥计算机硬件的处理能力。

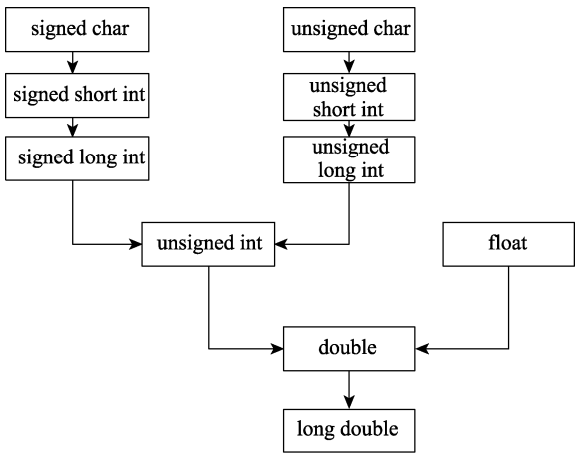


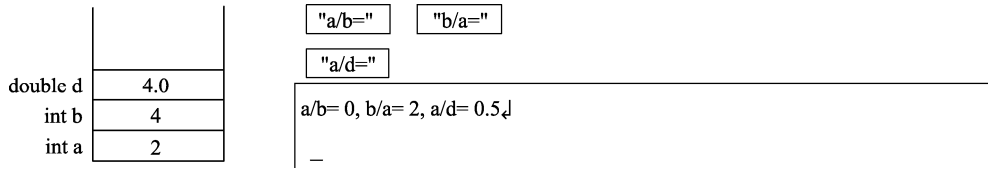
图 2.50 算术运算中数据类型的自动转换规则

VS2013 中的 C++编译器，算术运算中数据类型的转换规则如图 2.50 所示，自动对算术运算中操作数进行数据类型转换。这种数据类型转换是隐性的，初学者往往都感觉不到它的存在，编写表达式时容易出现错误，不能计算出预期结果。

例如：

```
int a = 2, b = 4;
double d = 4.0;
cout << "a/b= "<< a / b << ",b/a= "<< b / a <<",a/d= " << a / d << endl;
```

内存状态和输出结果：



上述代码中，a/b 和 b/a 中的除法运算是整数集上的除法，因除法在整数集上是不完备的，a/b 的结果超出了整数范围，计算机输出不超过 a/b 的最大整数 0。a/d 中，因操作数 d 为 double 类型，计算机将另一外操作数 a 转换为 double 类型，按照实数上的除法进行计算，输出 0.5。

网络中传输的数据流，如图像、音频等数据，规定每个二进制位的含义，一般都用整型表示，对这类数据，应该使用固定长度的整数，甚至使用无符号的，如 char、short、long，而不应该使用 int 类型。只有进行计数或算术计算时才使用 int 类型，以充分发挥硬件的计算能力，同时提高程序的兼容性。

与整型相比，实型是一个更为复杂的数据类型，其中的运算非常复杂，时间复杂度很高，因此，在编程实践中，建议能用整型就用整型，不能用整型才用实型，即优先选用整型。

2.7.3　强制数据类型转换

C/C++计算机语言提供了数据类型转换运算，为了与类型自动转换区分，常常称为强制数据类型转换。类型转换运算( )的语法语义如表 2.8 所示。

表 2.8　类型转换运算( )的语法语义

运算符	名称	结合律	语法	语义或运算序列
( )	类型转换	从右到左	(type) exp	计算 exp 得到值 v1，将值 v1 的类型显式转换成 type 类型，得到 type 类型的值 v2

语法比较简单，在一个表达式的前面加上数据类型并用括号“( )”括起来，其语义为将表达式的值转换为指定的数据类型。类型转换的语义如图 2.51 所示。

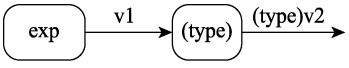


图 2.51　类型转换的语义

如，(double)(2+3)，先计算(2+3)得到整型 5，然后再转换为 double 类型的 5.0。值得注意的是，整型 5 和 double 类型的 5.0 虽然在数学上相等，但计算机内部是用不同的二进制串表示的，是不同的两个“值”。

数据类型转换只能转换“值”，不能转换“变量”的数据类型，如前面的示例：

```
signed char a = 5,b = -4, c;
c = a + b;
```

可以在 c=a+b 前增加类型转换，改写为：

```
c = (signed char)((int)a + (int)b);
```

其语义为，从变量 a 中取出 signed char 类型的值 v1，转换为 int 类型的值 v2；从变量 b 中取出 signed char 类型的值 v3，转换为 int 类型的值 v4；int 类型的值 v2 加 int 类型的值 v4 相加得到 int 类型的值 v5；将 int 类型的值 v5 转换为 signed char 类型的值 v6，存储到 signed char 类型的变量 c。数据类型转换的过程如图 2.52 所示。

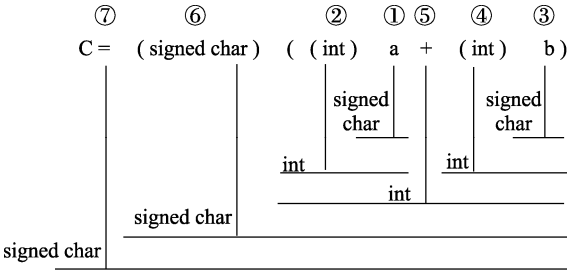


图 2.52　数据类型转换的过程

将 a 和 b 的值代入其语义，运算过程为，从变量 a 中取出 signed char 类型的值 5，转换为 int 类型的值 5；从变量 b 中取出 signed char 类型的值–4，转换为 int 类型的值–4；int 类型的值 5 加 int 类型的值–4 相加得到 int 类型的值 1；将 int 类型的值 1 转换为 signed char 类型的值 1，存储到 signed char 类型的变量 c。带类型转换的运算序列如图 2.53 所示。

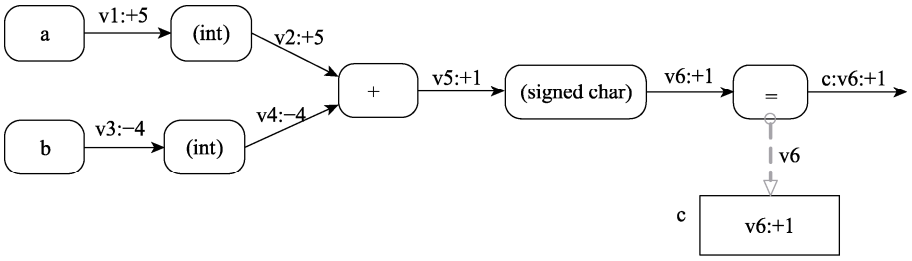


图 2.53　带类型转换的运算序列

数据类型及其转换是程序员不能回避的问题，在编写和理解表达式时要需要特别关注。

2.8　计算表达式的方法

在计算表达式过程中必须考虑数据类型。在 2.1.4 节中计算表达式的基本方法中需要增加数据类型转换，其计算步骤增加到 3 步：第 1 步，确定表达式的运算顺序；第 2 步，标注数据类型；第 3 步，计算表达式的值。

例如，带数据类型转换的表达式。

```
char c;
int i;
unsigned int u;
```

```
float f;
double d,y;
y=c * i + f * d + f / u - (i % 2 - d);
```

上面的表达式中的变量涉及几种不同的数据类型，在计算过程中需要在多种数据类型之间进行类型转换。

2.8.1　确定表达式的运算顺序

依照 2.1.4 节中学习的方法，确定表达式 y=c*i+f*d+f/u–(i%2–d)的计算顺序，如图 2.54 所示。

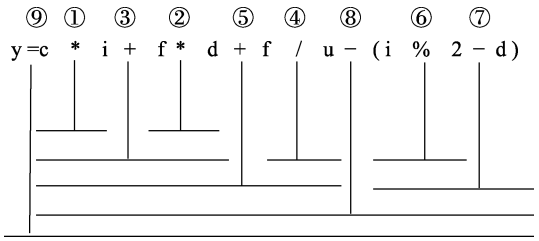


图 2.54　表达式 y=c*i+f*d+f/u–(i%2–d)的计算顺序

根据表达式 y=c*i+f*d+f/u–(i%2–d)的计算顺序，可推导出其运算序列，如图 2.55 所示。

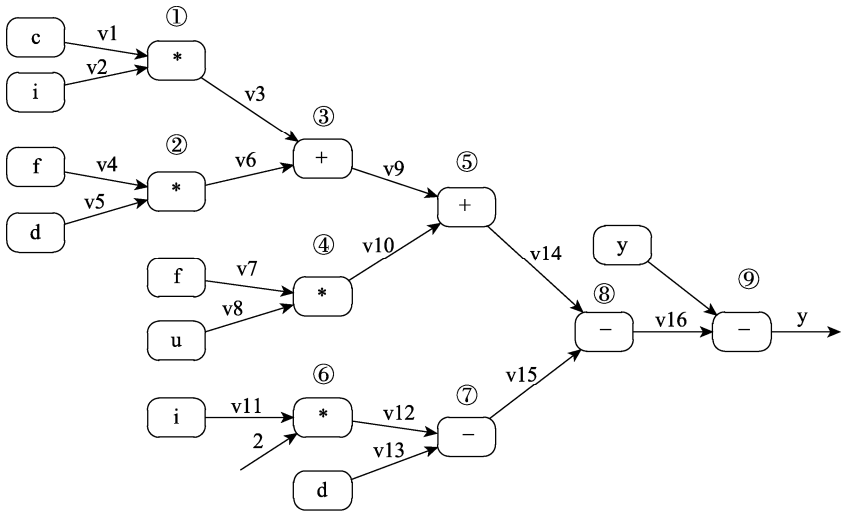


图 2.55　表达式 y=c*i+f*d+f/u–(i%2–d)的运算序列

2.8.2　标注数据类型

根据**运算的语义**和算术运算的**类型转换规则**，按照如图 2.54 所示的计算顺序，依次确定每个运算结果的数据类型，并标注在计算顺序图中，具体标注位置为该运算的左下角。标注数据类型后的计算顺序图如图 2.56 所示。

如图 2.56 所示，标注了各个运算结果的数据类型，根据标注的数据类型，在如图 2.55 所示的运算序列中插入需要增加的类型转换运算，如图 2.57 所示。

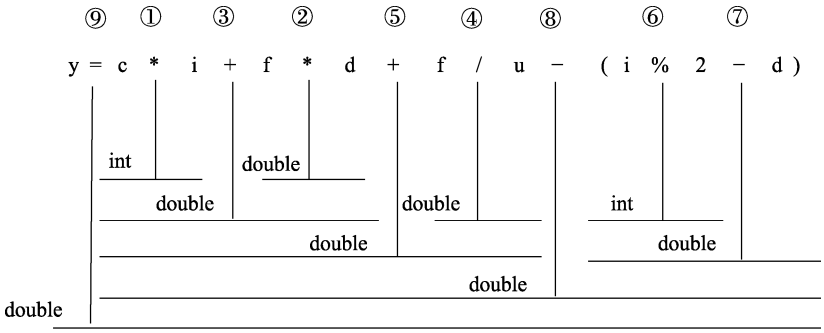


图 2.56　标注数据类型的计算顺序图

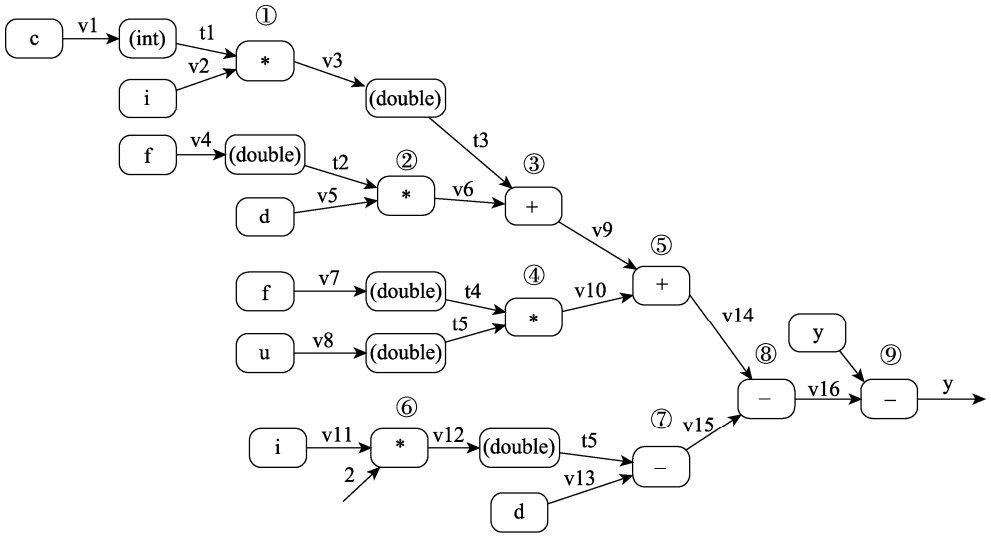


图 2.57　增加类型转换后的运算序列

2.8.3　计算表达式的值

在计算表达式 y=c*i + f*d + f / u –(i % 2 –d)前，先使用如下代码给变量赋值。

```
char c = 'A';
int i = 1;
unsigned int u = 3;
float f = 4.5;
double d = 0.5;
```

执行这些代码后，变量及其内存状态如图 2.58 所示。在这些数据上计算表达式的过程，如图 2.59 所示。最终计算的结果为 68.25，并存储在变量 y 中。

读者也可在如下数据集上计算表达式 y=c*i + f*d + f / u–(i% 2–d)，人工执行其运算序列。

```
char c = 'b';
int i = 2;
unsigned int u = 3;
float f = 3.5;
double d = 2.0;
```

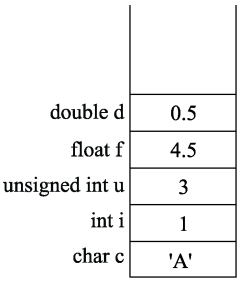


图 2.58　变量及其内存状态

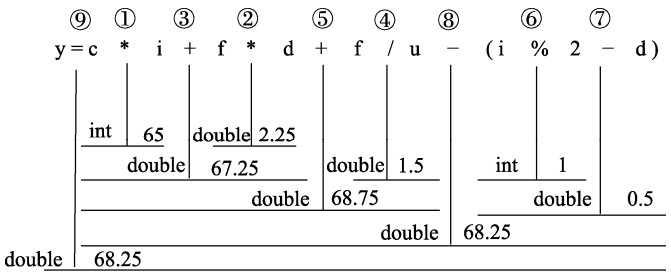


图 2.59　计算表达式的过程

实际上，很多编译器都存在没有严格实现计算机语言标准文本中规定的语义的情况，也存在适配硬件的问题，如，尽量将数据的位数调整到计算机的字长，以充分发挥计算机的性能，因此，编写表达式后必须进行调试，只有经过充分调试的程序才能实际使用。

在调试表达式时，调试器一般以表达式为粒度执行程序，不能观察到计算表达式的步骤，但可以观察到各个变量的值。可将计算顺序图中的变量值与计算机的实际结果比较，判断出表达式是否编写正确，并推导出错的计算步骤，再改正表达式。

在实际编程中，应编写简洁的表达式，而不能编写非常复杂的表达式，特别不能涉及大量的数据类型转换。

2.9　字符流和输出格式

输入输出设备是冯·诺依曼机中的逻辑部件，分别承担输入输出的任务。计算机常用的传统输出设备有显示器和打印机，输入设备有键盘。这 3 个设备成为一个计算机的标配，常常将它们称为标准输入输出设备。

显示器、打印机和键盘都属于字符设备。计算机中，使用“字符流”的方式在标准输入输出设备上输入和输出数据。

2.9.1　字符流的工作原理

字符流，顾名思义，从当前位置流过的字符序列。字符流是一种抽象概念，它将数据视为一个字符序列，这个字符序列从一个固定的位置流过，一般称这个固定位置为当前位置，因此，将从当前位置流过的字符序列称为字符流。字符流的概念如图 2.60 所示。

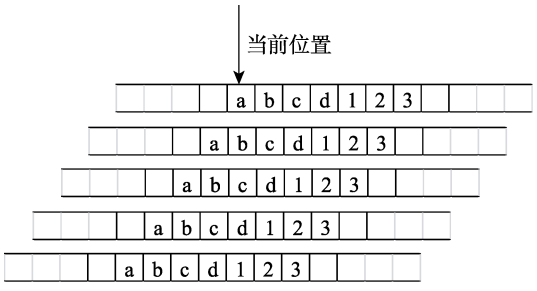


图 2.60　字符流的概念

如，在显示器上输出数据时，先将需要输出的数据转换为一个字符序列，然后再按照字符流的方式输出这个字符序列。显示器上的光标就是当前位置，每次都在光标处输出一个字

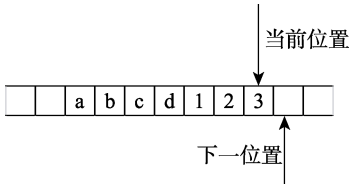


图 2.61　自动向后移动光标位置

符，相当于字符序列从光标位置流过。

为了便于人们阅读，显示器上的字符位置是固定不变的，而通过移动光标实现字符的流动，每显示一个字符就自动将光标向后移动一个位置。自动向后移动光标位置如图 2.61 所示。

与字符流相关的运算有两个：第 1 个运算从当前位置“提取”字符；第 2 个运算从当前位置“插入”字符。可通过这两个运算实现输入和输出。

前面学习的 cout 和 cin 都属于字符流。cout 用于输出，也称为输出字符流，简称输出流，主要运算是将需要输出的字符“插入”当前位置。cin 用于输入，也称为输入字符流，简称输入流，主要运算是从字符流的当前位置“提取”字符。

使用字符流输出一个数据时，一般不会一个字符一个字符地“插入”到字符流，这样太麻烦，而是每次插入一个数据的所有字符，如，一次插入表示一个整数的所有数字、一次插入表示一个实数的数字和小数点。

例如：使用字符流输出一个数据。

```
int a = 123;
cout << a;
```

cout << a 中，cout 是一个输出流。执行 cout << a 时，先将变量 a 的值 123 从内存中取出来，转换为字符串 "123"，然后将字符串 "123" 逐个插入到输出流的当前位置，同时系统将输出流中的字符显示到显示器。使用字符流输出数据的过程如图 2.62 所示。

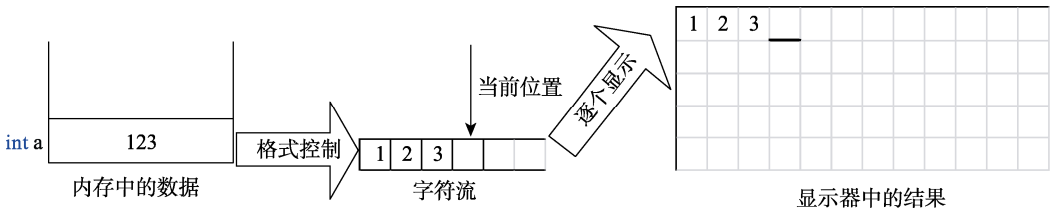


图 2.62　使用字符流输出数据

使用字符流输出一个数据，分为两步：第 1 步，从内存中取出数据并按照指定的显示格式转换为字符串；第 2 步，将字符串插入到输出流的当前位置，其中指定显示格式是编程输出数据的主要工作。

2.9.2　控制输出单元的格式

使用字符流输出数据时，每次输出一个数据，可将一个数据占用的显示区块视为一个单元，并控制这个单元的显示格式。单元的显示格式涉及显示的宽度（字符个数）、对齐方式和填充的字符。单元的显示格式控制方法如图 2.63 所示。

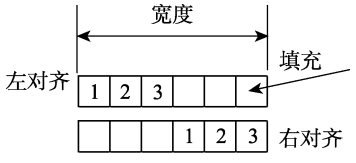


图 2.63　单元的显示格式控制方法

单元的显示格式控制方法与 Excel 表中单元格的控制方法类似，可以参考 Excel 中的控制方法。

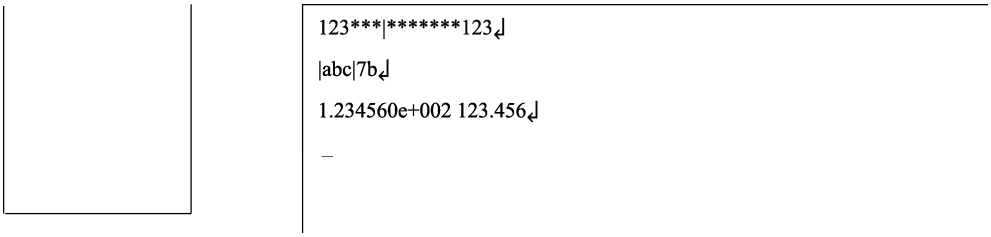
在头文件 iomanip 中提供了控制输出格式的多种手段，下面举一个例子介绍控制输出格式的方法，代码如例 2.9 所示。

【例 2.9】 控制输出格式。

```
#include<iostream>
#include<iomanip>
using namespace std;
void main(){
    cout << cout.fill('*') << setiosflags(ios::left)
         << setw(6) << 123 << '|'
         << resetiosflags(ios::left)
         << setw(10) << 123<<endl
         << "|abc|"
         << hex<<123<<endl;
    cout << setiosflags(ios::scientific)
         << 123.456 << ' '
         << resetiosflags(ios::scientific)
         << 123.456
         << endl;
}
```

代码中，cout.fill('*')设置填充字符，setiosflags(ios::left)设置左对齐，resetiosflags(ios::left)清除左对齐，setw(6)设置宽度，hex 设置十六进制，setiosflags(ios::scientific)设置科学记数法。

例 2.9 在 VS2013 上的输出结果如下：



注意，输出结果中左边的部分表示“栈”，由于没有变量，栈空间中没有数据，所以左边区域是空白的。

控制输出格式涉及大量的计算机语言细节，也与编程环境联系紧密，学习控制输出格式最好的方法是查阅随机文档或查阅标准文本，并针对不同情况在计算机上调试。

2.10　表达式的调试与维护

在程序运行前，编译器首先对表达式进行语法检测，判断是否符合规定的语法，然后将表达式“翻译”成相应的运算序列，最后按照这个运算序列生成表达式的目标代码，并成为可执行程序的一部分。

调试表达式的目标有两个：第一个，确保表达式符合语法，即语法正确；第二个，确保

表达式表达的运算序列符合预期，即语义正确或逻辑正确。

调试表达式中主要做两件事：第 1 件，发现并改正表达式的语法错误；第 2 件，发现并改正表达式的运算序列错误，即逻辑错误。

编译器能很好地发现表达式中的语法错误，但逻辑错误需要程序员来完成，并会消耗大量精力。

调试表达式逻辑错误的关键是按照 2.8 节中的方法人工计算表达式，并在调试器中对比变量的实际值，以推断表达式的正确性，纠正其中的错误。

借助编译器发现代码中的错误，能提高编程的效率，与编译器进行有效交互是调试程序的基础。

2.10.1 调试编译错误

调试表达式时的常见编译错误有：没有提前定义变量；在表达式后没有加分号“;”；运算符使用错误等。除此之外，还很容易将半角字符输出成全角字符。

例如：

```
void main(){
     int s, a, b;
     s = 4 * l + 2 ×(a + b)
}
```

编译时会报如下错误：

		描述	文件	行	列	项目
Error	1	error C2065: 'l' : undeclared identifier	kkk.cpp	9	1	kkk
Error	2	error C2146: syntax error : missing ';' before identifier '×(a'	kkk.cpp	9	1	kkk
Error	3	error C2065: '×(a' : undeclared identifier	kkk.cpp	9	1	kkk
Error	4	error C2065: 'b)' : undeclared identifier	kkk.cpp	9	1	kkk
	5	IntelliSense: identifier "l" is undefined	kkk.cpp	9	10	kkk
	6	IntelliSense: expected a ';'	kkk.cpp	9	16	kkk

这些错误都是因为表达式不正确而引起的，表达式应该是 s=4*l+2*(a+b)，后面还要加分号才能构成表达式语句。另一个原因是变量 l 没有定义，需要先将其定义。

将表达式修改如下：

```
void main(){
     int s, a, b,l;
     s = 4 * l + 2 * (a + b);
}
```

再编译会报如下错误：

		描述	文件	行	列	项目
Error	1	error C4700: uninitialized local variable 'l'	kkk.cpp	9	1	kkk

```
          used
Error  2  error C4700: uninitialized local variable 'a' kkk.cpp  9   1   kkk
          used
Error  3  error C4700: uninitialized local variable 'b' kkk.cpp  9   1   kkk
          used
```

按照表达式的计算顺序，在计算时要使用到 3 个变量 a、b 和 l 的值，必须在计算前给它们赋值。在程序中，可使用初始化、赋值运算或输入语句给这 3 个变量赋值。下面的程序就可通过编译。

```
void main(){
     int s, a=1, b=3,l=3;
     s = 4 * l + 2 * (a + b);
}
```

编译时不仅可以检测到语法错误，还可以检测到简单的逻辑错误。

从程序中还能发现，字母 l 与数学 1 很难区别，不要单独使用字母 l 作为变量名。

2.10.2　整型的溢出

数据类型对表达式的计算结果有很重要的影响，在编写和调试表达式时要高度重视，下面以整数来讨论这个问题。

计算机中整型最终都是使用固定位数二进制数，有一定的表示范围。常见整型的表示范围如表 2.9 所示。

表 2.9　常见整型的表示范围

二进制位数	有符号整数 signed	无符号整数 unsigned
1 字节 8 位 char	$-2^7 \sim 2^7-1$ (−128, 127)	$0 \sim 2^8-1$ (0, 255)
2 字节 16 位 short int	$-2^{15} \sim 2^{15}-1$ (–32 768, 32 767)	$0 \sim 2^{16}-1$ (0, 65 535)
4 字节 32 位 long int	$-2^{31} \sim 2^{31}-1$ (–2 147 483 648, 2 147 483 647)	$0 \sim 2^8-1$ (0, 4 294 967 295)

当数据超过了整型的表示范围时，会导致数据的溢出。下面通过示例来演示整型数据的溢出，代码如例 2.10 所示。

【例 2.10】 整型数据的溢出。

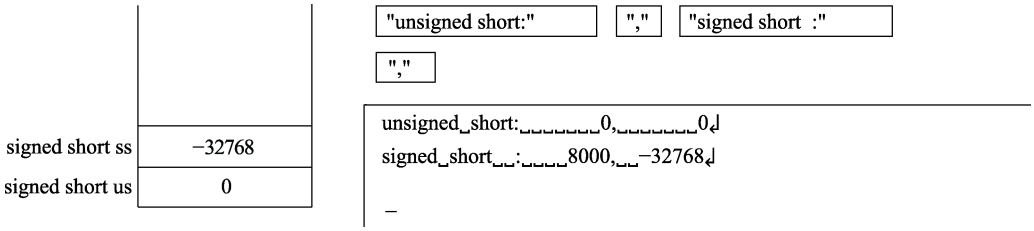
```
#include<iostream>
#include<iomanip>
using namespace std;
void main(){
    unsigned short us = 65535;
    us = us + 1;                  //溢出
    cout << " unsigned short:";
    cout << setw(8) << hex << us << "," << setw(8) << dec << us << endl;

    signed short ss = 32767;
```

```
    ss = ss + 1;          //溢出
    cout << " signed short:";
    cout << setw(8) << hex << ss << "," << setw(8) << dec << ss << endl;

}
```

运行中的内存状态和输出结果：



这与我们的期望一致吗？为什么会得到这样的结果？要回答这些问题，需要清楚，整数是怎样存储在变量中的，变量中存储的是什么。

运用 2.4.1 节理解整数与进制中的基本知识能解决这些问题，并建议读者自己去推导演算。

2.10.3 整数的重要性

整型是计算机语言中最重要的数据类型，是理解其他数据类型的基础。从本质上讲，整型中存储的数据就是固定位数的二进制数，即一串 0、1 数字，整型的运算是固定位数二进制数的计算，计算机在进行整型运算时，一般都假设程序员在编写程序时已保证其正确性，编译时不再增加检测逻辑的代码，以满足应用的多样性和提高程序运行的效率。

换句话讲，程序员必须理解整型的表示原理及运算的逻辑，在编写程序时运用其表示原理及运算的逻辑解决客观世界中的实际问题，并保证其正确性。

直接管理整型数据是一个合格程序员不可推卸的责任。

前面专门讨论自然数和整数的记数法及其运算过程，就是为了程序员理解整型的表示原理及运算的逻辑，能够承担这个责任而准备的。

在实际编程中，常常用整型表示文字、图像、语音等实际应用中的数据，表示通信、网络、物联网中传输的数据，涉及有符号整数、无符号整数，涉及 8 位、16 位、32 位整数，使得整型的应用情况非常复杂，往往需要程序员花费大量的时间和精力，反复做类似的程序调试工作，才能判断是否满足功能要求，满足预期。

2.11 本章小结

本章主要从计算角度学习了表达式和数据类型相关的知识，以及相应的编程方法。

学习了表达式及基本运算，重点学习了算术运算、赋值运算、复合赋值运算和自增自减

运算在冯·诺依曼机上的语义，深入学习了表达式的计算顺序和运算序列以及编写表达式的方法，需要掌握将数学公式改写为计算机表达式的方法，掌握使用计算顺序图人工计算表达式的方法。

学习了变量的概念，重点学习了定义变量在冯·诺依曼机上的语义以及自然数、整数及其记数方法，深入学习了固定位数和负数的方法。学习了整数、字符和实数等基本数据类型，重点学习了其表示原理和存储格式以及数据类型转换及其语义，需要掌握使用变量和操纵数据的方法，能够选择适当的数据类型，并掌握使用内存图描述数据的方法。

学习了字符流，重点学习了控制输出格式的原理，需要掌握控制输出格式的方法。

学习了调试和维护表达式的基本知识，深入学习了发现逻辑错误的方法，需要掌握使用运算序列图、内存图和计算顺序图调试和维护表达式的方法。

2.12　习题

1. 下面给出了一些数学或物理公式，请根据本章所学内容完成各小问。

等差数列的通项公式：$a_n = a_1 + (n-1)d$

余弦定理：$a^2 = b^2 + c^2 - 2bc\cos A$

自由落体距离公式：$h = gt^2 / 2$

并联电阻公式：$R = R_1R_2 / (R_1 + R_2)$

（1）只使用四则运算，能否将所有公式改写为表达式?

（2）如果能，写出其表达式，画出上面表达式的计算顺序图，给变量指定一个值并计算表达式。

（3）根据上面计算顺序图，画出表达式的运算序列图，并以文字描述其运算序列。

2. 按照 2.1.3 节和 2.8 节中的方法,使用图和文字描述下列代码在 32 位机上的计算过程、计算步骤和表达式 c=b–a 的计算序列。

```
int  a=10;
char b=107,c;
c= b-a;
```

要求在计算步骤和计算序列中增加类型转换运算。

3. 画出下列实数存储在内存中的逻辑结构。

（1）3.14165926

（2）–0.333333

（3）17000.3358

（4）–10.94623

4. 上机编程，定义一个无符号整型变量 x，并初始化为–10，分别以十六进制和十进制输出变量 x。再定义一个有符号整型变量 y，将变量 x 的值强制转换为有符号整型，再将转换结果赋值给变量 y，分别以十六进制和十进制输出变量 y。观察输出结果，并用 2.4.1 节中

有关整数表示的知识解释输出结果。

5. 读下面的程序，画出表达式的计算顺序图和内存图，并分析运行结果产生的原因。

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
    unsigned short us = 65535;
    us = us + 1;         //溢出
    cout << " unsigned short:";
    cout << setw(8) << hex << us << "," << setw(8) << dec << us << endl;

    signed short ss = 32767;
    ss = ss + 1;         //溢出
    cout << " signed short:";
    cout << setw(8) << hex << ss << "," << setw(8) << dec << ss << endl;

    return 0;
}
```

6. 以下代码中包含了两个表达式，请按要求回答下列问题。

```
unsigned int k;
char x ='A';
double  a = 9.0;
int y = 2,z = 1;
k=x+a%3*(float)(x+y)%2/z

double u;
int i = 2;
float a = 4;
u=i--,a*=++i
```

（1）分别画出两个表达式的计算顺序图，并计算表达式的值。

（2）根据计算顺序图，分别画出表达式的运算序列图。

（3）上机调试这两个表达式，并改写代码，以观察计算 u=i--后变量 i、u 中的值，计算 a*=++i 后变量 i、a 中的值，并与人工计算结果对比。

第 3 章

构 造 分 支

大千世界，多姿多彩，实际应用中的问题往往非常复杂，一般需要将一个复杂问题分解为多个相对简单的问题，然后再逐个解决这些相对简单的问题。

按照数学的观点，先对需要解决的问题进行层层分解，最终将问题分解为一些能够使用数学公式解决的基本问题，然后再将解决基本问题的数学公式组合起来，构成一个数学模型，最后用这个数学模型去解决实际问题。

计算机，顾名思义，就是“计算”的机器。前面学习了使用表达式和数据类型描述数学公式中的计算，后面开始学习将表达式组合起来展现计“算”艺“术”。

3.1 结构化程序设计


视频讲解

结构化程序设计思想来源于小孩玩的积木游戏。在积木游戏中，购买大量小块的积木，小孩将多个小块的积木组成较大的积木块，再将较大的积木块组成更大的积木块，最后搭建出期望的大积木。

3.1.1 3 种基本结构

1966 年，Borhra 和 Jacopini 提出了表示算法的 3 种基本结构：顺序结构、选择结构和循环结构。3 种基本结构流程如图 3.1 所示。

（1）**顺序结构**。顺序结构表示程序中的各操作是按照它们在程序中出现的先后顺序执行的。前面学习的运算和表达式，基本上都符合“第 1 步做……，第 2 步做……，第 3 步做……”这种模式，都属于顺序结构。顺序结构符合平时的习惯，且便于理解，因此是编程中使用最多的结构，也是编程者最基本的思维模式。

（2）**选择结构**。选择结构表示程序的处理步骤出现了分支，它需要根据某一特定的条

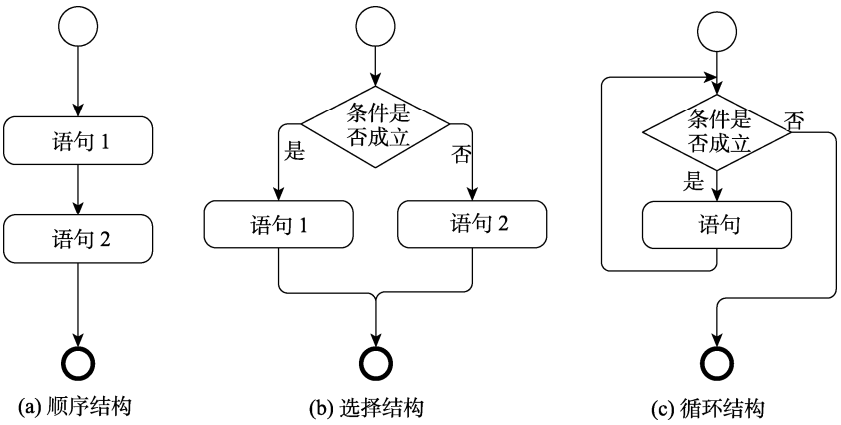


图 3.1　3 种基本结构流程

件选择其中的一个分支执行，适用于“如果……，则……，否则……”这样的思维模式。

（3）**循环结构**。循环结构表示程序反复执行某个或某些操作，直到某条件为假（或为真）时才可终止循环。

从程序流程的层面讲，顺序结构是基础，是程序功能的具体载体；选择结构是方向，决定了程序提供哪些功能；循环结构是能力，在本质上决定了程序计算能力的大小。

在编程中，符合这 3 种结构之一的程序块相当于一个积木，程序员将小的程序块组成较大的程序块，再将较大的程序块组成更大的程序块，最后编写出期望的程序。

结构化程序设计思想尽管简单，但功能强大。理论证明，目前计算机能解决的任何问题，都能用这 3 种基本结构组成的算法来解决。换句话说，用这 3 种基本结构可以描述任何一个程序的流程。

3.1.2　流程图

流程图是描述程序流程的主要工具之一，能够帮助初学者很好地理解程序，理解程序的执行过程，有利于培养编程的思维方式。

业务流程建模与标注（Business Process Model and Notation，BPMN）是 OMG（Object Management Group）组织提出的一个标准，主要目标是提供一组容易理解的符号，这组符号覆盖了业务分析、系统设计到程序的实现整个软件开发过程。BPMN 是一个图形语言，不仅可用于算法描述、程序设计，而且直观，易于理解。

BPMN 中定义的图标很多，有兴趣的读者可以查阅相关标准文档，也可在需求分析时学习。流程图的基本图标及语义如表 3.1 所示。

流程图的图标非常简单，便于理解，画出的流程图直观形象，并能体现程序的主要特征，是学习程序和编写程序的有力工具。本书使用上述图标绘制流程图，在各个层次上描述程序。

3.2　分支结构及条件


视频讲解

分情况解决问题是一种解决问题的基本方法，其核心思想为，先将需要解决的问题分成

表 3.1　流程图的基本图标及语义

图 标 名	图　标	语　义
开始		用于描述程序或语句块的开始，只能有一个
结束		用于描述程序或语句的结束，只能有一个
处理		用于描述程序中的语句或功能，可多个
判断		用于描述分支，需要标注条件
操作流程		连接其他图标，线上可标注数据，用于描述程序的流向及其数据

不同情况，并针对每种情况提出一个解决方案，在解决实际问题时，满足哪种情况就使用哪个解决方案。

数学上的分段函数是描述分情况解决问题的数学工具，分段函数中的条件用于描述问题的不同情况，分段函数中的公式用于描述解决问题的方案。如下面的分段函数：

$$y=\begin{cases}0, & x\leqslant 0\\ 3x, & 0<x\leqslant 40\\ 120, & x>40\end{cases}$$

将一个问题分解成 $x\leqslant 0$ 、$0<x\leqslant 40$ 和 $x>40$ 共 3 种情况，并针对这 3 种情况分别对应 y=0、$y=3x$ 和 y=120 这 3 种解决方案。分段函数的计算流程如图 3.2 所示。

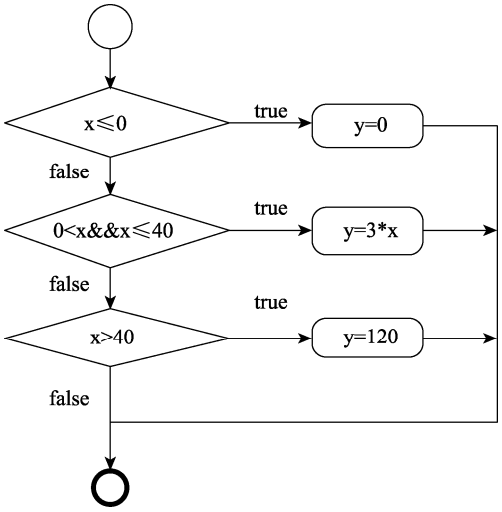


图 3.2　分段函数的计算流程

使用分段函数描述各种情况，先根据不同情况进行复杂问题分解，然后针对每种情况抽

象出一个相对简单的数学公式，最后再将这些数学公式组合在一起，构成解决复杂问题的完整数学模型。

计算机语言中的逻辑运算和条件运算用于描述问题的不同情况，分支语句用于描述不同情况下问题的解决方案。下面先学习分支语句以及逻辑运算和条件运算，再学习构造分支的方法。

3.2.1 if 语句

在自然语言中，有两种常见句型可用于分情况讨论：第 1 种，如果……，则……；第 2 种，如果……，则……，否则……。

在计算机语言中，针对分段函数这种典型场景，借用自然语言中的上述两种句型，设计了 if 语句，用于描述分支结构。

if 语句的语法为：

```
if(exp)
      statStatement;
```

或

```
if(exp)
      statStatement1;
else
      statStatement2;
```

其中，exp 为表达式，exp 的结果为 true（真）或 false（假），作为执行 statStatement 的条件，一般为关系表达式或逻辑表达式。

与分段函数进行对照，exp 对应分段函数中的条件，statStatement 对应分段函数中的数学公式。

在画流程图时，为了与代码中的顺序保持一致，判断的正下方都是 false 分支，需要特别注意。if 语句语义流程如图 3.3 所示。

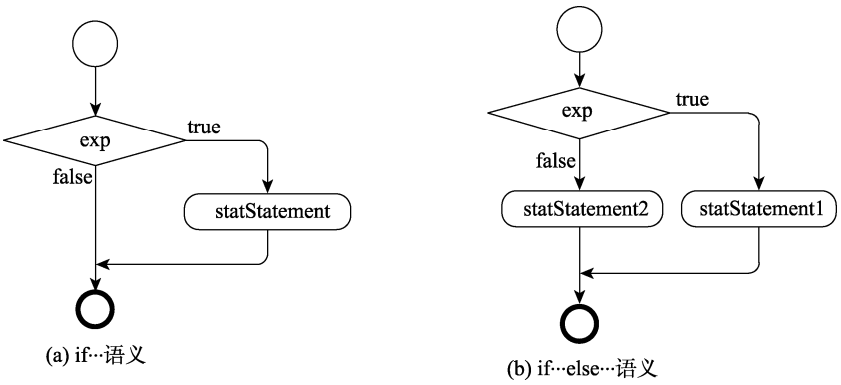


图 3.3　if 语句语义流程

前面的分段函数是 3 个分支，可通过两条 if 语句来描述。

```
if (x <= 0)              //exp
      y = 0;             //statStatement
else if (0 < x&&x <= 40)//exp
      y = 3 * x;         //statStatement
else if (x > 40)         //exp
      y = 120;           //statStatement
```

上面代码使用了关系运算和逻辑运算，下面学习这两种运算。

3.2.2　关系运算

关系运算有等于、不等于、大于、小于、大于或等于和小于或等于，用于比较数据的大小。这些运算都是数学中定义的，计算机语言只是直接使用了这些运算。

关系运算的语法和语义如表 3.2 所示。

表 3.2　关系运算的语法和语义

运算符	名称	结合律	语法	语义或运算序列
<=	小于或等于	从左到右	exp1<=exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，计算 v1<=v2 得到 bool 类型的值
>=	大于或等于	从左到右	exp1>=exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，计算 v1>=v2 得到 bool 类型的值
>	大于	从左到右	exp1>exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，计算 v1>v2 得到 bool 类型的值
<	小于	从左到右	exp1<exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，计算 v1<v2 得到 bool 类型的值
==	等于	从左到右	exp1==exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，计算 v1==v2 得到 bool 类型的值
!=	不等于	从左到右	exp1!=exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，计算 v1!=v2 得到 bool 类型的值

关系运算被分成两个优先级，大于、小于、大于或等于和小于或等于的优先级高于等于和不等于的优先级，关系运算的结果是 bool 类型，只有 true 和 false。

1. 是否相等运算

判断两个数据是否相等的运算有等于（==）和不等于（!=）两种运算，其运算结果都是 bool 类型的值。可编程观察这两个运算得到的结果和数据类型，代码如例 3.1 所示。

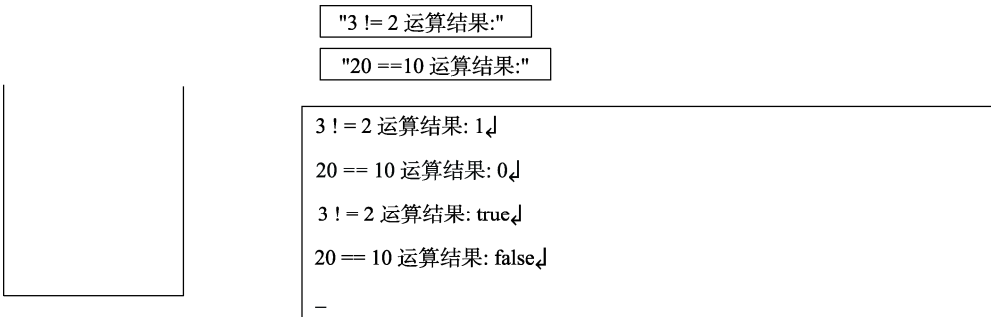
【例 3.1】 比较相等。

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

int main(){
    cout << "3 != 2 运算结果: "
         << (3 != 2) << endl
         << "20 == 10 运算结果: "
```

```
            << (20 == 10) << endl;
     cout << boolalpha           //显示 ture 或 false
            << "3 != 2 运算结果: "
            << (3 != 2) << endl
            << "20 == 10 运算结果: "
            << (20 == 10) << endl;
}
```

例 3.1 程序的内存状态和输出结果：



2. bool 类型

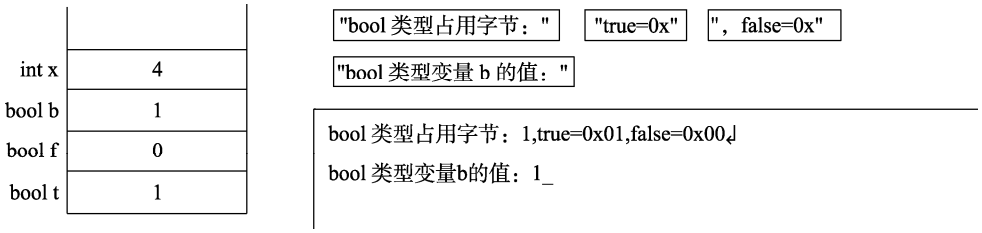
bool 类型只有真和假两个值，在 C/C++中分别用 true（真）和 false（假）表示，在计算机内部分别存储为 1 和 0，可以理解为整型派生出的一个数据类型。可编程观察 bool 类型变量的内存大小和值，代码如例 3.2 所示。

【例 3.2】 bool 类型的内存大小和值。

```
#include<iostream>
#include<iomanip>
using namespace std;
void main(){
    bool t = true, f = false, b;
    int x;
    cout << "bool 类型占用字节："
         << sizeof t             //sizeof 运算的功能取变量的大小
         << ",true=0x"
         << setw((sizeof t) * 2) << setfill('0') << hex << t
         << ",false=0x"
         << setw((sizeof f) * 2) << setfill('0') << hex << f;
    b = (x = 4);                 //将整型转换为 bool 类型
    cout << endl << "bool 类型变量 b 的值：" << b;

}
```

例 3.2 程序运行的内存状态和输出结果：



第一行输出结果，说明 bool 类型占用 1 字节，true 用 8 位整数的 1 表示，false 用 8 位整数的 0 表示。

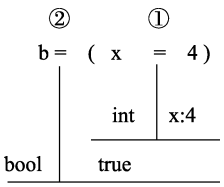


图 3.4　表达式 b = (x = 4)的语义

表达式 b = (x = 4)的语义如图 3.4 所示，将整数 4 赋值给变量 x，得到变量 x，然后从变量 x 取出 int 类型的值 4，将它转换为 bool 类型的值 true，并存储到 bool 类型的变量 b 中。

bool 类型与整型之间转换规则为，0 转换为 false，非 0 转换为 true。初学者在编写程序时容易将等于（==）与赋值（=）运算符混淆，在条件判断中将==误写为赋值（=），从而产生错误。

例如：

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

表达式 x = 0 中，“=”是赋值运算，计算结果为变量 x，变量 x 的值为整数 0，将整数 0 转换为 bool 类型，结果为假（false）。

上述代码中，无论 somevalue 的值是什么，cout 语句都不会被执行。

3. 比较大小运算

比较数据的大小有大于（>）、小于（<）、大于或等于（>=）和小于或等于（<=）4 种运算，运算结果都是 bool 类型的值。可编程观察比较大小运算得到的结果和数据类型，代码如例 3.3 所示。

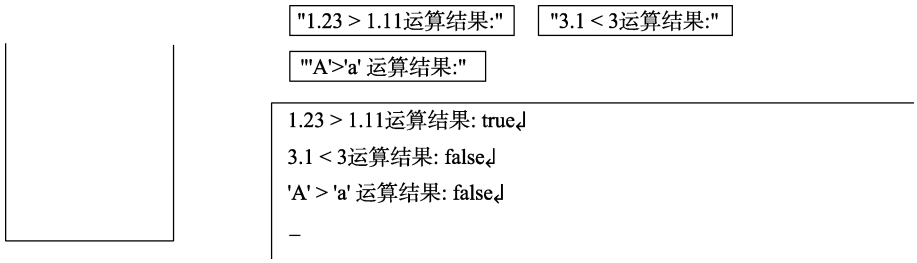
【例 3.3】 比较大小运算。

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

void main()
{
     cout << boolalpha       //显示 ture 或 false
          << "1.23 > 1.11 运算结果: "
          << (1.23 > 1.11) << endl
          << "3.1 < 3 运算结果: "
          << (3.1 < 3) << endl
```

```
            << "'A'>'a' 运算结果: "
            << ('A'>'a') << endl;
}
```

例 3.3 程序的内存状态和输出结果：



前面两行输出结果是实数和整数之间比较大小，相对比较安全，因实数存在精度问题，应避免实数参与相等运算，一般应用于比较运算。最后一行输出结果是比较两字符大小的结果，在计算机中，比较字符大小实际上是比较字符 ASCII 码的大小。

3.2.3 逻辑运算

逻辑运算有非、与和或运算，为了与位运算相区别，分别称它们为逻辑非（logical negation）、逻辑与（logical AND）和逻辑或（logical OR）运算。逻辑运算的语法和语义如表 3.3 所示。

表 3.3 逻辑运算的语法和语义

运算符	名 称	结合律	语法	语义或运算序列
!	逻辑非 (logical negation)	从右到左	!exp	计算 exp 得到 bool 类型的值 b，如果 b 等于 true，则返回 false，否则返回 true
&&	逻辑与 (logical AND)	从左到右	exp1&&exp2	计算 exp1 得到 bool 类型的值 b1，若 b1 值为 false，exp1&&exp2 的结果为 false，否则，计算 exp2 得到 bool 类型的值 b2，计算 b1&&b2 得到 bool 值 b3，exp1&&exp2 的结果为 bool 值 b3
||	逻辑或 (logical OR)	从左到右	exp1||exp2	计算 exp1 得到 bool 类型的值 b1，若 b1 值为 true，则 exp1||exp2 的结果为 true，否则，计算 exp2 得到 bool 类型的值 b2，计算 b1||b2 得到 bool 值 b3，exp1||exp2 的结果为 bool 值 b3

逻辑非、逻辑与和逻辑或运算有不同的优先级，从高到低依次为逻辑非、逻辑与和逻辑或，其中，所有操作数均为 bool 类型，运算结果也是 bool 类型。逻辑与和逻辑或规定了操作数的执行顺序，必须从左到右依次计算。

1. 逻辑运算的运算规则

逻辑非、与、或等逻辑运算的运算规则与数学中的规则相同，计算机中逻辑运算的运算规则如表 3.4 所示。

表 3.4　逻辑运算的运算规则

op1/ exp1	op2/exp2	!op1	op1&&op2	op1||op2
true	true	false	**true**	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	**false**

其中，op1 和 op2 为运算的操作数，是语法中表达式的结果，如 exp1&&exp2 中，exp1 的计算结果记为 op1，exp2 的计算结果记为 op2。

当逻辑与的两个操作数都为真时，结果才为真，否则为假；与之相反，当逻辑或的两个操作数都为假时，结果才为假，否则为真。

2. 逻辑运算的语义

逻辑非运算的语义相对简单，有!true==false 和!false==true 两个等式。逻辑与和逻辑或相对复杂一些，下面重点学习这两种运算。

逻辑与的语法为 exp1&&exp2，语义是计算 exp1 得到 bool 类型的值 b1，若 b1 值为 true，计算 exp2 得到 bool 类型的值 b2，计算 b1&&b2 得到 bool 值 b3；否则 exp1&&exp2 的结果为 false。其中规定了 exp1 和 exp2 的计算顺序，必须从左到右依次计算。

逻辑与和逻辑或类似，但运算规则不同。逻辑与(&&)与逻辑或(||)的语义如图 3.5 所示。

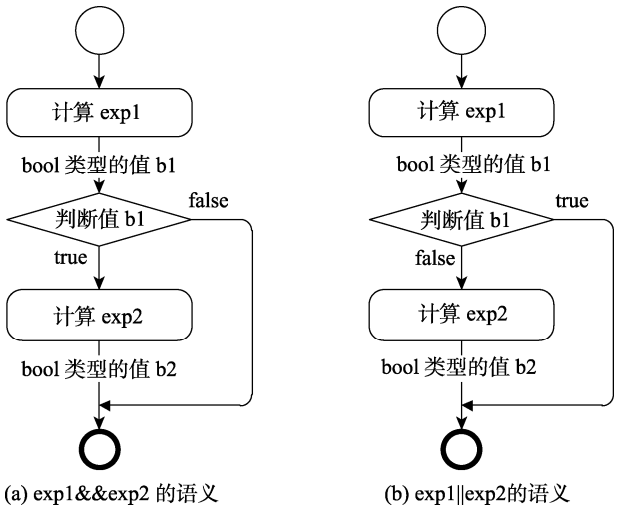


图 3.5　逻辑与与逻辑或的语义

数学中经常表示一个数的取值区间，如 x 的取值区间为(2,3]，用不等式表示为 $2 < x \leqslant 3$，其中包括了 $2 < x$ 和 $x \leqslant 3$ 两个不等式，应改写为逻辑表达式 2<x&&x<=3，而不能写成表达式 2<x<=3。两个表达式的语义差异如图 3.6 所示。

逻辑与和逻辑或的优先级低于关系运算，在同时使用关系运算时一般不用括号“()”，这样写出的表达式更加简明，符合平时的习惯，但也容易出现错误。

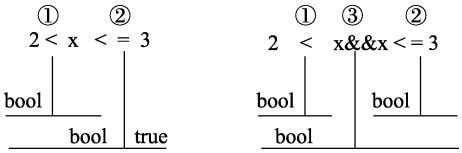


图 3.6　两个表达式的语义差异

表达式 2<x&&x<=3 的语义为，计算 2<x 得到 bool 类型的值 b1，若 b1 值为 false，则 exp1&&exp2 的结果为 false，否则，计算 x<=3 得到 bool 类型的值 b2，计算 b1&&b2 得到 bool 值 b3，exp1&&exp2 的结果为 bool 值 b3。表达式 2<x<=3 的语义为，计算 2<x 得到一个 bool 类型的值 b1；将 bool 类型的值 b1 转换为整型值 t（只可能为 0 或 1），t<3 得到肯定的 bool 类型的值 true。这个语义不符合数学算式 2<x≤3 的意思。

用流程图表示上述两个表达式的语义，更加直观，更容易理解。两个表达式的流程图如图 3.7 所示。

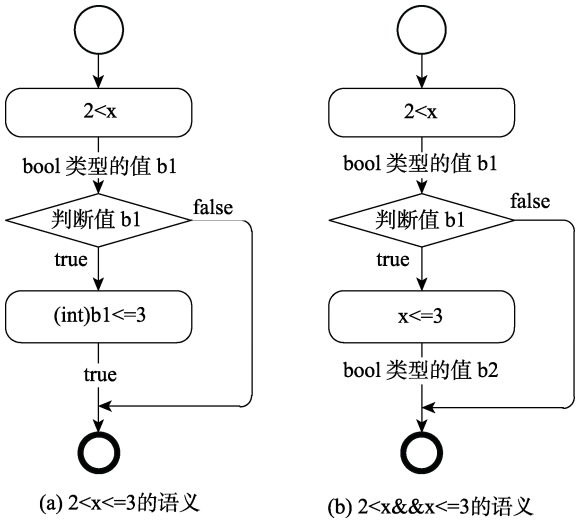


图 3.7　两个表达式的流程图

关系运算和逻辑运算往往结合使用，并且常用在分支和循环语句中，是实现分支和循环流程的基础。

例如，温度和湿度提示的代码如下：

```
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";
```

通过判断房间的温度和湿度，提示舒适程度。其中，使用了两个条件表达式判断舒适程度，并给出相应提示。判断舒适程度的条件表达式的计算顺序如图 3.8 所示。

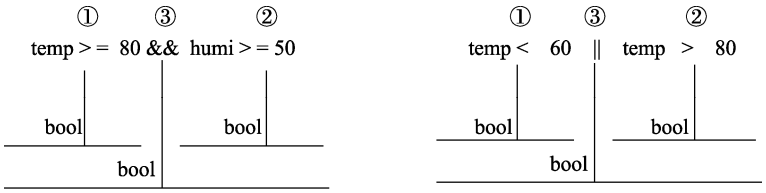
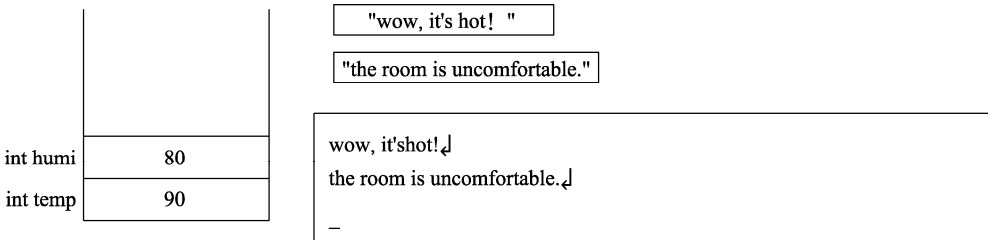


图 3.8　判断舒适程度的条件表达式的计算顺序

运行上述程序的内存状态和输出结果为：

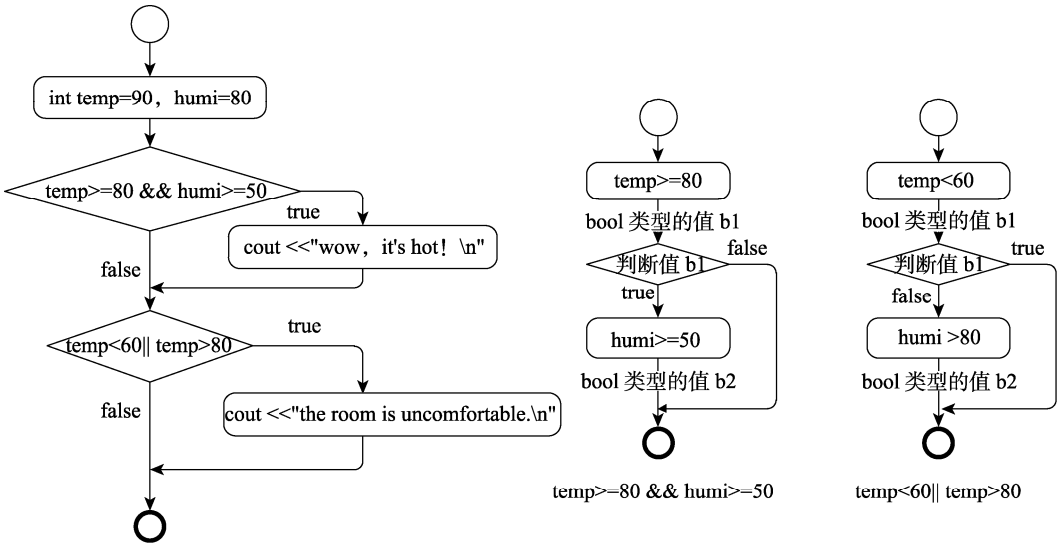


其中：

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

的语义为，如果 temp>=80 && humi>=50 为真，则执行语句 cout <<"wow，it's hot！\n"，输出“wow，it's hot!”，否则，就输出结果。温度和湿度提示的流程如图 3.9(a)所示。

程序的输出结果由 temp>=80 && humi>=50 和 temp<60|| temp>80 控制，两个条件表达式的语义如图 3.9(b)所示。



(a) 温度和湿度提示的流程　　(b) 两个条件表达式的语义

图 3.9　温度和湿度提示的流程及其条件表达式的语义

编写代码是一项细致的工作，不能将!=写为=!，其语义完全不同，如：

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

x!=9 中!=是不等于运算，x=!9 中=!是赋值运算和逻辑非运算，其语义为，将整型值 9 转换为 bool 类型的 true，进行逻辑非运算，得到 false；将 false 转换为整型值 0，存储到变量 x 中。if 语句要求表达式 x=!9 的值必须为 bool 类型，还要将变量 x 的值 0 转换为 bool 类型的值 false，因此，不可能执行语句 cout <<"x isn't 9\n"。

3. 表达式短路

逻辑与和逻辑或的流程图清楚表明，第 1 个操作数的表达式肯定会被执行，但第 2 个操作数的表达式会根据前一个表达式的结果决定是否执行。换句话说，语法中的第 2 个表达式可能会被执行，也可能不会被执行，这个特点是其他大多数运算不具备的，需要特别注意。例如：

```
int n = 3, m = 6;
if (n > 4 && m++ < 10)
       cout << "m should not changed.\n";
cout << "m=" << m << endl;
```

代码中使用了一个条件表达式，由于 n > 4 的值为 false，n>4 && m++ <10 中的 m++ <10 不会执行，m 的值只会是 6。程序员要避免使用这类表达式。表达式短路的计算顺序如图 3.10 所示。

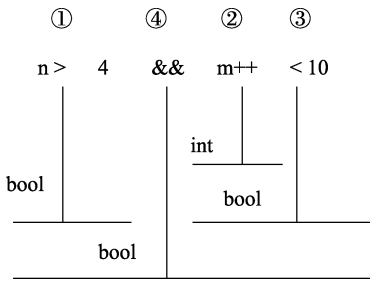
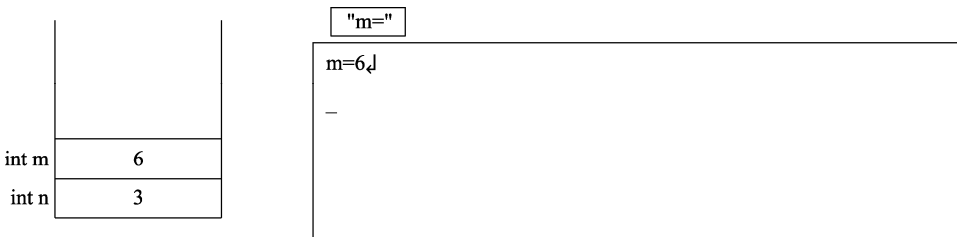


图 3.10　表达式短路的计算顺序

运行上述程序的内存状态和输出结果为：



3.3　构造分支的典型模式

if 语句使用非常灵活，可以构造出很多种分支结构。下面学习单分支、双分支、多分支和嵌套 4 种典型模式。

3.3.1　单分支(if…)

单分支，顾名思义只有一个分支，相当于中文的句型“如果……，就……”。

例如，比较变量 a 和 b 的大小，并根据比较结果输出"hello world"，代码为：

```
int a = 7, b = 4;
if（a > b）
    cout << "\n hello world";
```

按照 if 语句的语义，先计算表达式 a > b，从变量 a 中取出值 7，从变量 b 中取出值 4，判断条件 7>4 为真，则执行 cout << "\n hello world"，在屏幕上输出 hello world。如果定义变量 a 和 b 时，将它们的初值交换，判断条件为假，则不会执行 cout << "\n hello world"，不会输出 hello world。输出 hello world 的流程如图 3.11 所示。

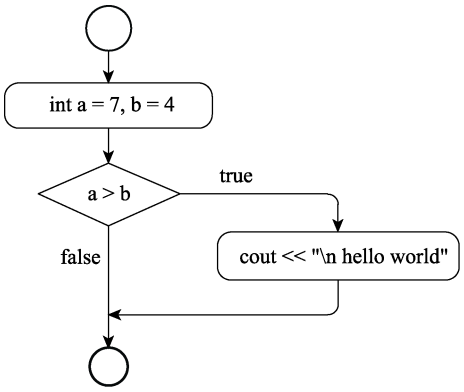


图 3.11　输出 hello world 的流程

例如，输出一个字符，如果是'b'，则响铃，代码如例 3.4 所示。

【例 3.4】 响铃程序。

```
#include<iostream>
#include<conio.h>
using namespace std;
void main()
{
    cout << "please input the b key to hear a bell.\n";
    char ch = getchar();    //从标准输入设备中读取一个字符
    if (ch == 'b')          //如果是'b'，则响铃
       cout << '\a';
}
```

上述程序的 main()函数中包含两条表达式语句和一条 if 语句，计算机按照它们在程序中的顺序逐条执行这 3 条语句，构成一个顺序结构。响铃程序的流程如图 3.12 所示，流程图中非常清楚地表示出了 3 条语句的执行顺序。

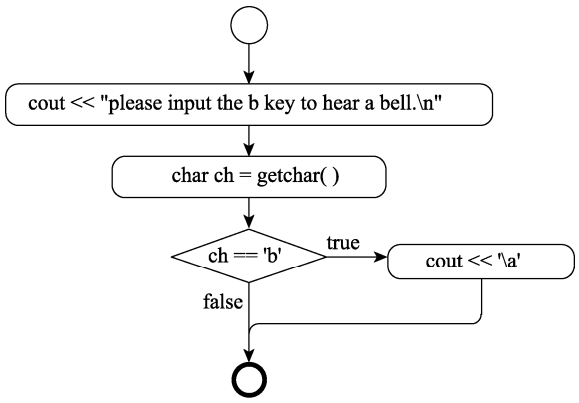


图 3.12　响铃程序的流程

先执行第 1 行语句 cout << "please input the b key to hear a bell.\n"，然后执行第 2 行语句 char ch = getchar()，最后执行 if 语句。if 语句占用两行，语句 cout << '\a'是 if 语句的一部分，当输入到变量 ch 中的字符为'b'时执行。

如果将 if 语句写成一行：

```
if ((ch = getchar())== 'b')  cout << '\a';
```

其功能不变，但不能增加注释信息“如果是 'b'，则响铃”，更不能体现分支结构，因此建议不要写成一行。

当在键盘上输入字符 'b' 和回车（'↵'）后，会发出声音，屏幕上的结果为：

```
please input the b key to hear a bell. ↵
b↵
_
```

再次运行程序，当在键盘上输入字符'1'（或非字符'b'）和回车键'↵'后，如果发出声音，则说明程序编写错了。

3.3.2　复合语句和空语句

if 语句的语法 if (exp) statStatement 中规定，statStatement 只能是一条语句，不能有多条语句，但在实际编程中，常常需要有多条语句，为了解决这个问题，计算机语言提供了复合语句，语法非常简单，用大括号“{}”将多条语句括起来，将大括号中多条语句在语法上“复合”为一条语句，其语法如下：

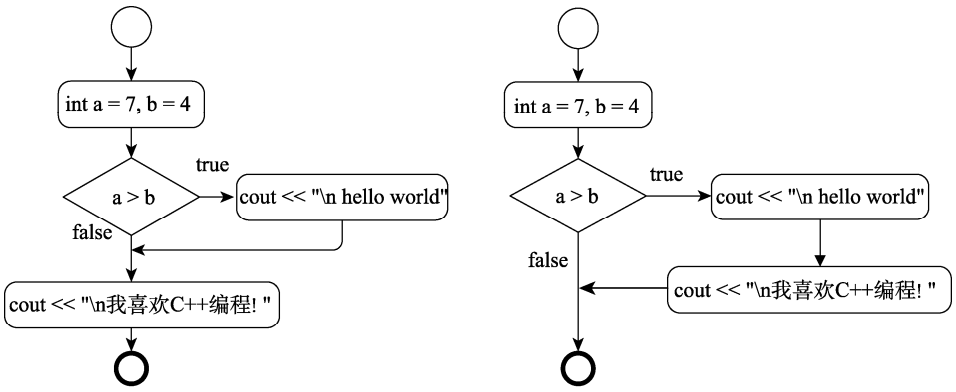
```
{
   statStatement1;
   statStatement2;
```

```
    ...
}
```

复合语句也称为块语句，并将复合语句中的语句称为语句块，意思为将一段程序在语法视为一条语句。如：

```
int a = 7, b = 4;
if (a > b){
       cout << "\n hello world";
       cout << "\n 我喜欢 C++编程! ";
}
```

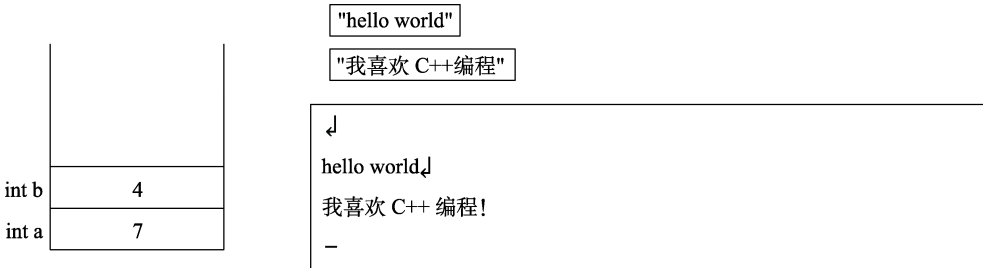
if 语句中使用复合语句，将两个输出语句作为语句块增加到 if 语句，也将语句块的流程嵌入到 if 语句的流程中，其流程如图 3.13(b)所示。



(a) 不加大括号　　(b) 加大括号

图 3.13　使用复合语句后 if 语句的流程

运行上面的程序，内存状态和输出结果为：



程序中条件 a > b 为真，顺序执行 if 语句中的两条 cout 语句，输出上面的结果。

如果上面的例子中删除大括号，则程序的流程完全不一样，变量 a 和 b 的初值无论是什么，一定会输出“我喜欢 C++编程!”，其流程如图 3.13（a）所示。

C/C++中，使用分号“;”表示一条语句结束，分号是语句的结束标志。一般情况下，一条语句的最后都有这个结束标志，但也允许前面没有语句而只有单独的一个分号。针对这种情况下，将单独的一个分号也视为一条语句，称为**空语句**。

空语句的语法为一个分号，并且前面不能是一个完整的语句。空语句只有语法上的作用，

没有任何实际操作，使用非常简单，但容易导致其他语句出现错误，如：

```
int a = 7, b = 4;
if (a > b);
      cout << "\n 我喜欢 C++编程！";
```

在 if 语句的判断条件后加上分号，编译器会将“if (a > b);”视为一条语句，从而导致语句的流程错误。空语句导致的错误流程如图 3.14（b）所示。

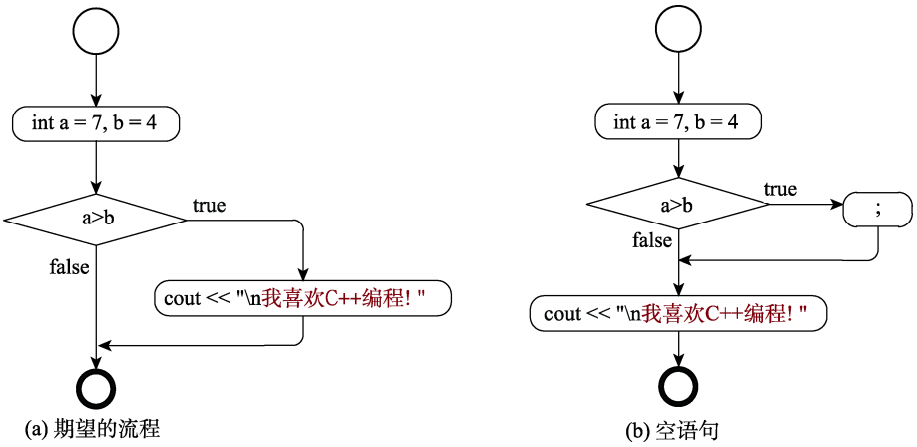


图 3.14　空语句导致的错误流程

按照 if 语句的语法“if (exp) statStatement;”，if 语句在语句结束前应该有一条语句，但“if (a > b);”没有语句就结束了。在这种情况下，编译器将分号视为一条语句，cout << "\n 我喜欢 C++编程！" 被解释为另一条语句，导致了与期望流程不同的流程。

一条 if 语句只包含了一条语句时，也可以将这条语句视为一条复合语句，加上大括号，养成习惯后，可避免很多流程错误，会减少很多 bug（缺陷）。如：

```
int a = 7, b = 4;
if (a > b){
       cout << "\n 我喜欢 C++编程！";
}
```

因此，使用单分支编程模式时，建议按照下面的格式写代码：

```
if (判断条件) {
      语句;
      ...
}
```

或

```
if (判断条件)
{
      语句;
      ...
}
```

按照上述格式写出的代码，与图 3.13 进行比较，它们在结构上非常相似，这有助于理解代码的流程。

3.3.3　双分支（if…else…）

前面学习了单分支结构，下面学习双分支结构，实现双分支的 if 语句语法如下：

```
if (exp)
      statStatement1;
else
      statStatement2;
```

其语义为，如果 exp 的值为真，则执行 statStatement1，否则执行 statStatement2。详细执行过程为，计算 exp1，得到 bool 类型的值 v1，如果 v1 为 true，则执行 statStatement1，否则执行 statStatement2。

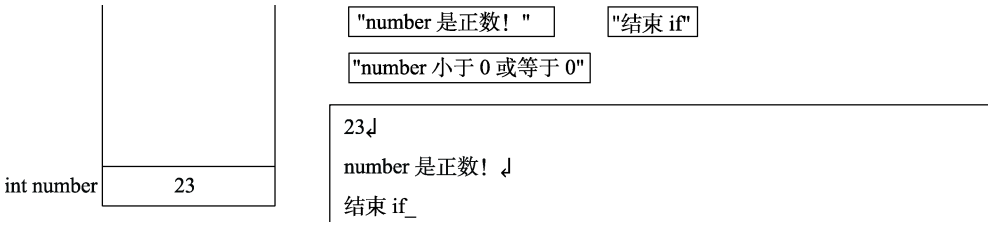
在下面的例子中，判断一个数 number 是正数或非正数（等于 0 或小于 0）。只需要检查 number 是否大于 0，因为任何小于或等于 0 的情况都在 else 语句中处理。

例如：

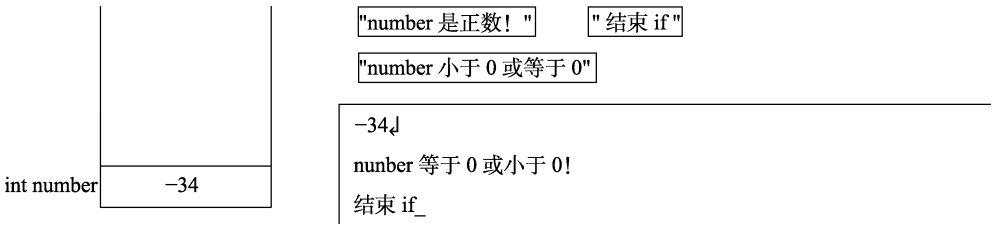
```
int number;
cin >> number;
if (number>0)
      cout << "\n number 是正数! ";
else
      cout << "\n nunber 等于 0 或小于 0! ";
cout << "\n 结束 if";
```

其中，if 语句的流程是，如果 number>0，则执行 cout << "\n number 是正数！"，否则执行 cout << "\n nunber 等于 0 或小于 0！"。两条输出语句执行且只执行一条。判断是否为正数的程序流程如图 3.15 所示。

运行程序，输入 23，内存状态和输出结果为：



再次运行程序，输入一个负数，如–34，内存状态和输出结果：



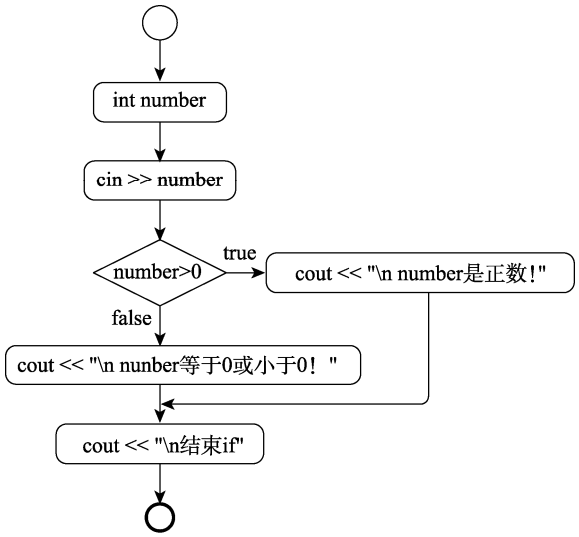


图 3.15　判断是否为正数的程序流程

读者可再次运行程序，输入 0 和↵，观察输出结果，如果和预期不符，则程序一定存在错误。也可以用大括号将 if 语句中的两条语句括起来，这样写出的程序健壮性更好。

```
int number;
cin >> number;
if(number>0) {
      cout << "\n number 是正数! ";
}
else{
      cout << "\n nunber 等于 0 或小于 0! ";
}
cout << "\n 结束 if";
```

3.3.4　if 语句嵌套

if 语句可以包含语句，当然也可以包含 if 语句，if 语句包含 if 语句称为 if 语句嵌套。代码框架如下：

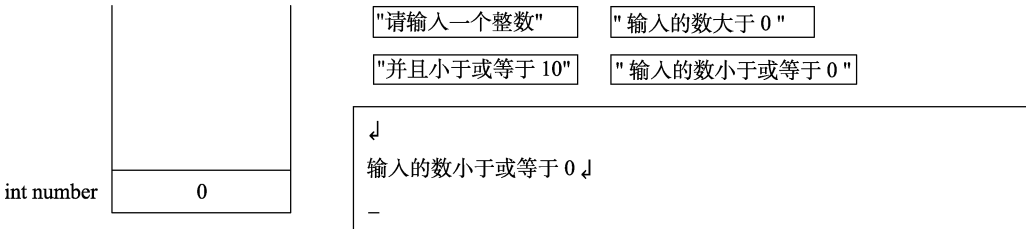
```
if（条件）{                  //第 1 个条件
     if（另一条件）{         //这是嵌套的 if 语句
          //语句
     }
     //其他语句
}
```

例如，编写程序判断一个数的正负。先提示用户输入一个整数，如果这个数小于或等于 0，则输出这个数；如果大于 0，则当这个数在 0~10 时才输出“并且小于或等于 10”。判断一个数的正负，代码如例 3.5 所示。

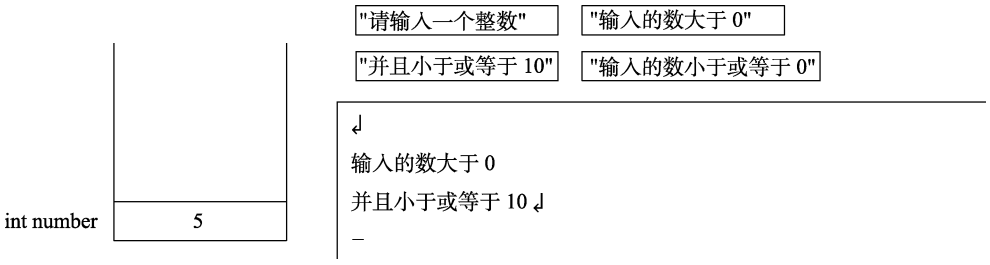
【例 3.5】 判断一个数的正负。

```
#include<iostream>
using namespace std;
void main(){
     int number;
     cout << "\n 请输入一个整数：";
     cin >> number;
     if (number>0)//正数
     {
        cout << "\n 输入的数大于 0" << endl;
        if (number >= 1 && number <= 10){
              cout << "并且小于或等于 10" << endl;
        }
     }
     else{
        cout << "\n 输入的数小于或等于 0" << endl;
     }
}
```

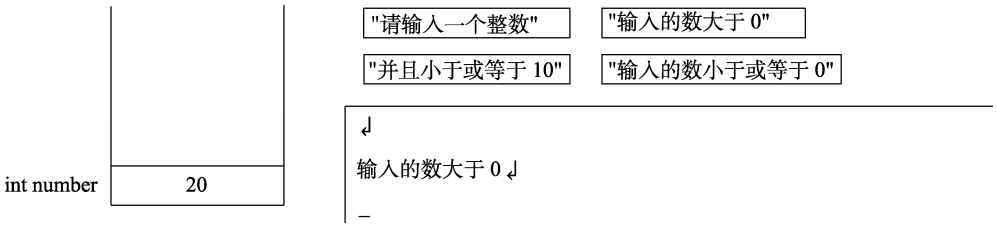
运行例 3.5 程序，输入 0 和↵，内存状态和输出结果为：



输入 5 和↵，内存状态和输出结果为：



输入 20 和↵，内存状态和输出结果为：



例 3.5 程序的流程如图 3.16 所示。

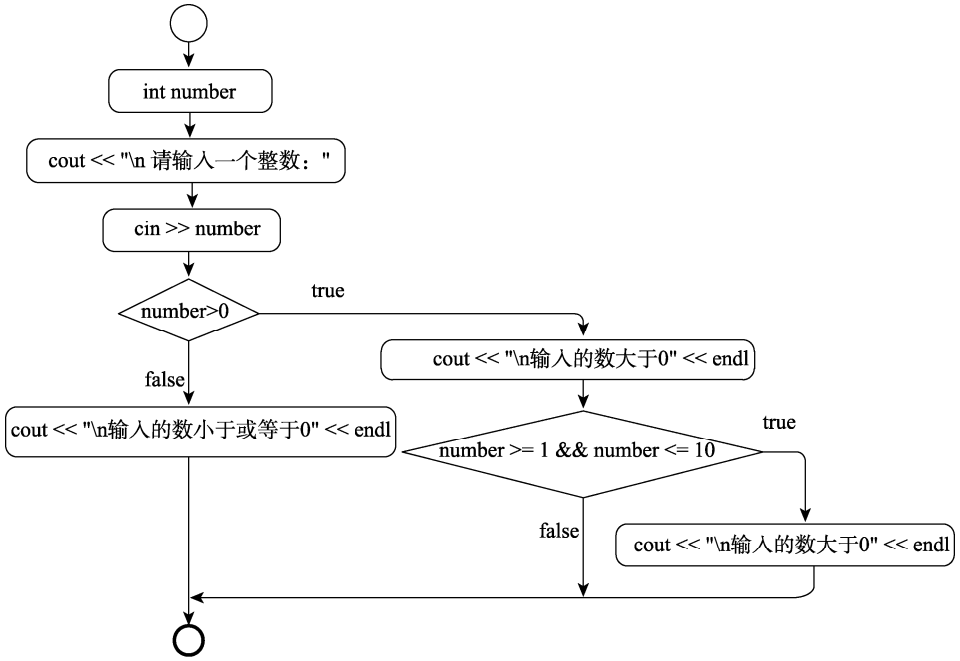


图 3.16　例 3.5 程序的流程

如图 3.16 所示，if 语句的第 1 个分支中包含了一条 if 语句，只有一个分支。if 语句的第 2 个分支中也可包含一条 if 语句。如，根据输入的字符，输出不同的内容，代码如例 3.6 所示。

【例 3.6】 根据输入的字符分情况输出。

```
#include<iostream>
#include<conio.h>
using namespace std;
void main(){
    cout << "please input the b key to hear a bell, \n";
    char ch = getchar();
    if (ch == 'b'){
        cout << '\a';
    }
    else{
        if (ch == '\n')
            cout << "what a boring select on…\n";
        else
            cout << "bye!\n";
    }
}
```

例 3.6 所示的程序，等待输入一个字符，如果是'b'，则响铃，否则，如果是'↵'，则输出"what a boring select on…"，否则，输出"bye!"。

上面这句话逻辑不清，也不符合平时的表达习惯，将它改写成如下代码：

```
cout << "please input the b key to hear a bell,\n";
char ch = getchar();
if (ch == 'b')
     cout << '\a';
else if (ch == '\n')
     cout << "what a boring select on\n";
else
     cout << "bye!\n";
```

将 else if 翻译成“如果”，其语义的逻辑会更清晰，读起来就会更加通顺。如果是'b'，则响铃，如果是'↵'，则输出"what a boring select on…"，否则，输出"bye!"。修改后的流程如图 3.17 所示。

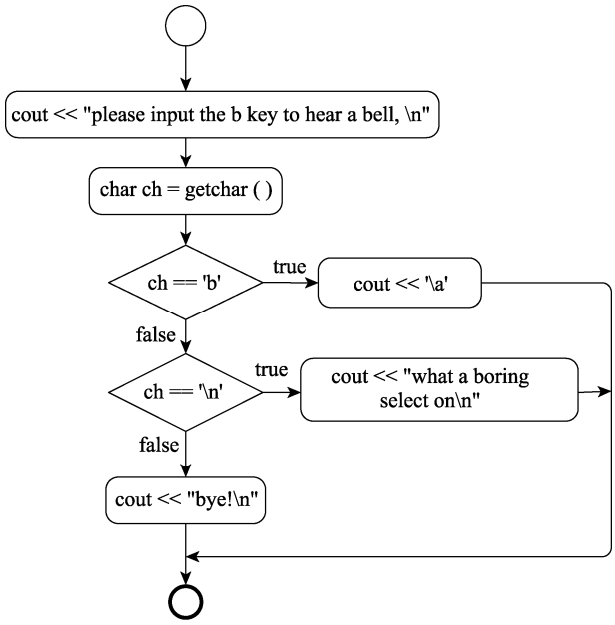


图 3.17　修改后的流程

图 3.17 所示的流程，条件为真时流向右边，为假时往下流动。

如果说表达式来自数学中的公式，那么计算机语句就来自于语文中的语句，因此，学好了语文就能编写出简洁、通顺的代码。

3.3.5　多分支（if…else if…else）

可以通过 if 语句嵌套实现多分支结构，其代码框架如下：

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

```
else if(条件 n)
      语句 n;
else
      语句;
```

其流程简单清晰，依次判断条件是否为真，如果为真，就执行后面的语句并结束，如果所有条件都不满足，则执行最后一条语句。

例如，采用多分支模式实现，提示用户输入一个整数，如果这个整数在 1 和 4 之间，则输出合适的 knick-knack 信息；如果用户输入的数不在这个范围内，则输出一个错误信息。knick-knack 程序代码如例 3.7 所示。

【例 3.7】 knick-knack 程序。

```
#include<iostream>
using namespace std;
void main() {
     int number;
     cout << "\n 请输入一个整数";
     cin >> number;
     cout << "\n He played knick-knack";
     if (number == 1){            //输出 knick-knack 信息
         cout << "with his thumb. \n";
     }
     else if (number == 2){
         cout << "with my shoe.   \n";
     }
     else if (number == 3){
         cout << "on his knee.    \n";
     }
     else if (number == 4){
         cout << "at the door.    \n";
     }
     else{                        //错误检查,其他数字都不合格
         cout << "\n Whoa! He doesn't play knick-knack there!\n\n";
     }
}
```

使用 if…else if…else 多分支模式，逻辑简单清晰，很容易发现程序的错误。如果使用一系列独立的 if 语句，就需要分别检查所有小于 1 和大于 4 的值，编写程序容易出现错误，可读性也非常差。knick-knack 中的多分支流程如图 3.18 所示。

图 3.18 所示的流程中，向下的分支是假，向右的分支是真。在编写程序时，如果不能达到这个要求，可使用逻辑运算！调整其分支方向。


视频讲解

3.4 使用 switch 语句

在数学中有一类特殊的函数，将一个范围内的数映射为一个整数，这类函数常常用于定

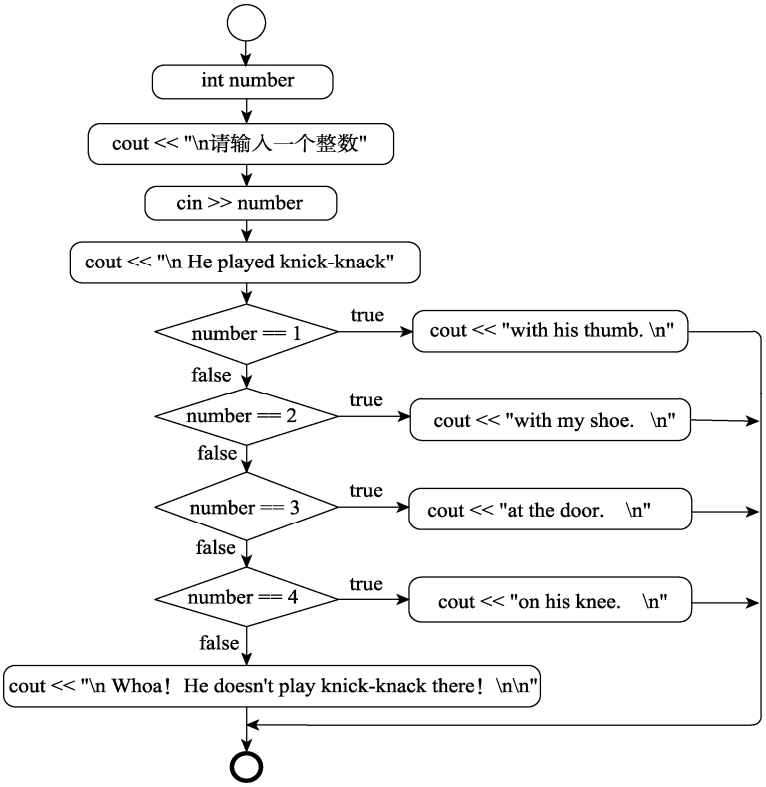


图 3.18　knick-knack 中的多分支流程

性分析，应用非常广泛。如：

$$y=\left\lceil \frac{x}{20} \right\rceil, y=\begin{cases} \left\lceil \frac{x}{10} \right\rceil & x \geqslant 60 \\ 5 & x<60 \end{cases}$$

其中，前一个公式是将百分制的课程成绩转换为 5 分制的成绩，后一个公式可将百分制的课程成绩转换为等级。常常使用类似上面的公式将课程成绩分成几个等级，然后按照等级评价学习效果或制订学习计划。

计算机语言中，除了前面学习的 if 语句外，还提供了 switch 语句用于上述特殊的分支结构。switch 的语法为：

```
switch（exp）
{
      case cexp1:语句块 1;
      case cexp2:语句块 2;
      …
      case cexpn:语句块 n;
      default:语句块 n+1;
}
```

其中，case 后面的表达式都是整型常量表达式，如一个整数、一个字符或一个枚举型的值。switch 的语义为：计算 exp，得到整型、字符型或枚举型的值 v，然后在 case 的常量表达式中依次查找值 v，如果找到，则跳转到该 case 后面的语句，如果没有找到，则跳转到 default 后面的语句。switch 语句的流程如图 3.19 所示。

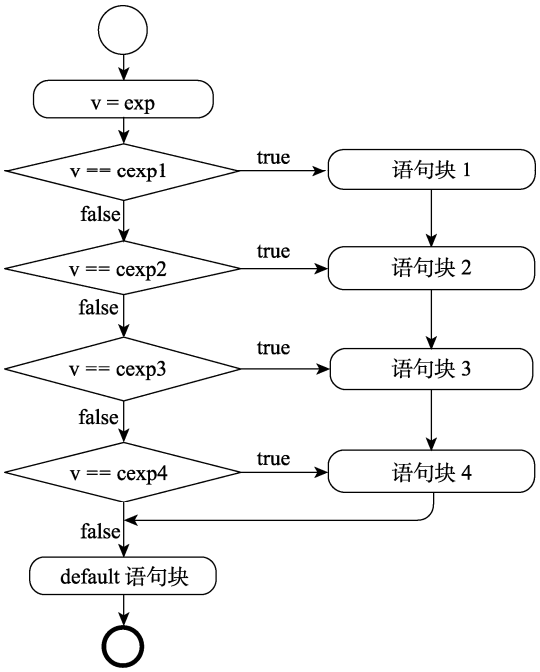


图 3.19　switch 语句的流程

在这个流程图中，首先计算 exp 得到一个值 v，然后再依次与 case 中的常量比较（常量不用计算），效率比每次计算后再比较高，这也是 switch 语句的优势。

细心的读者可以发现，在大部分情况下，我们只想执行一种情况下的语句块，而 switch 的语义显然不满足这个要求，于是，switch 常与转向语句 break 合用，在 switch 的语句块中增加 break。

例如，输入一个年份，查看该年份所处的世纪有什么著名事件。

需要将年份转换为世纪，其数学公式如下。

$$y=\left\lceil \frac{x}{100}+1 \right\rceil$$

按照这个公式，可使用 switch 语句编写程序，代码如例 3.8 所示。

【例 3.8】 按照世纪输出著名事件。

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

```
    int year;
    cout << "请输入一个年份:";
    cin >> year;
    //查看该年份所处的世纪有什么著名事件
    switch (year / 100 + 1)  //exp 是一个表达式
    {
    case 15://常量
        cout << "15 世纪：哥伦布环游世界！\n";
        break;
    case 18:
        cout << "18 世纪：费城公约！\n";
        break;
    case 20:
        cout << "20 世纪：人类登月！\n";
        break;
    default:
        cout << "\n 我不知道这个世纪有什么著名事件。\n";
    }
}
```

编程的关键是用表达式 year/100+1 表示年份 year 所处的世纪，另外，每个 case 语句块的最后一条语句都是 break，用以跳过后面的 case 语句块。按照世纪输出著名事件的流程如图 3.20 所示。

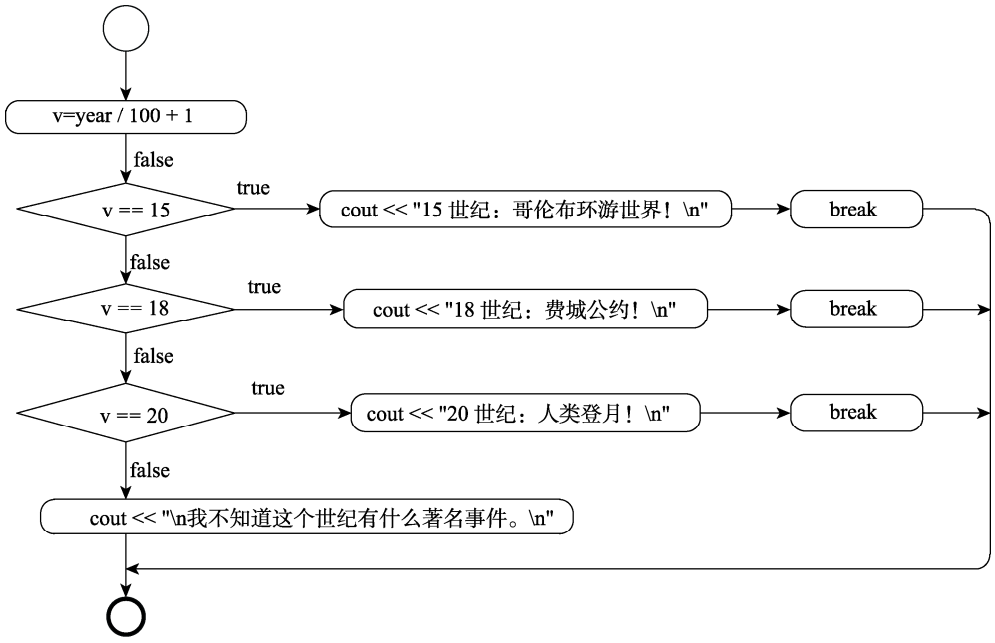


图 3.20　按照世纪输出著名事件的流程

在 switch 语句中，先计算表达式 year / 100 + 1 得到一个值 v，case 后面的都是常量，然后将得到的值 v 依次与这些常量比较，不用再计算表达式 year / 100 + 1，而 if 语句需要多次计算这个表达式，因此，switch 语句的效率比 if 语句高。

3.5 条件运算


视频讲解

一条 if 语句可以包含多个表达式，但表达式不能包含语句，这是计算机语言的规定。表达式定义的运算序列一般都是顺序执行的，不支持分支流程。为了满足这个应用需要，C/C++提供了一个运算，它可以根据一个条件来判断下一步需要进行哪个运算，这个运算就是条件运算。

3.5.1 条件运算的语法语义

条件运算是 C/C++中唯一的三目运算，运算符为“？”和“:”两个单独的字符，其语法为 exp1 ? exp2 : exp3，用“？”和“:”将 3 个操作数隔开，exp1 为条件，根据 exp1 的计算结果，决定计算 exp2 还是 exp3。条件运算的语义和语法如表 3.5 所示。

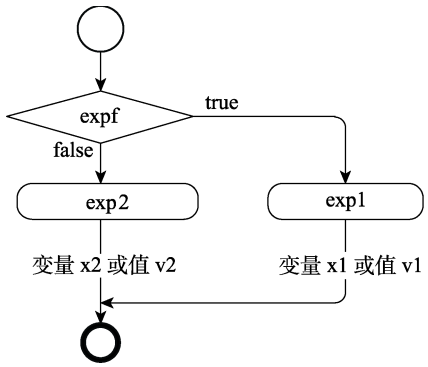


图 3.21　条件运算的流程

与其他运算相比，条件运算比较特殊，有两个“唯一”：第一个唯一是唯一一个三目运算；第二个唯一是唯一支持分支流程的运算。条件运算的流程如图 3.21 所示。

如，使用条件运算求两个数的最大值。

表 3.5　条件运算的语义和语法

运算符	名称	结合律	语法	语义或运算序列
? :	条件运算 (conditional)	从右到左	expf ? exp1 : exp2	计算 expf 得到 bool 值 b，若 b 为 true，则计算 exp1 得到变量 x1 或值 v1；若 b 为 false，则计算 exp2 得到变量 x2 或值 v2

```
int a=1, b=2, c;
c = (a > b) ? a : b;
```

其中，c = (a > b) ? a : b 是一个表达式，与使用 if 语句相比，代码更加简洁。

3.5.2 条件运算表达式举例

字母的大小写转换是常用的功能，用条件运算可以很方便地实现这个功能。下面就以大写字母转换为小写字母为例，学习条件运算。

例如：

```
char c = 'A';
```

```
c = c>='A' && c<='Z' ? c+32 : c;
```

上述表达式的功能是将变量 c 中的大写字母转换为小写字母。

表达式包含了赋值运算、条件运算、算术运算和逻辑运算，比较复杂。下面以这个表达式为例，介绍读表达式的完整步骤和方法：第 1 步，查运算表；第 2 步，标注运算的计算顺序；第 3 步，标注数据类型；第 4 步，写出运算序列；第 5 步，求表达式的值。

1. 查运算表

表达式 c = c>='A' && c<='Z' ? c+32 : c 中包含多个运算，可将这些运算从中找出来，列成一个表，以便后面使用。表达式 c = c>='A' && c<='Z' ? c+32 : c 中的运算如表 3.6 所示。

表 3.6　表达式 c = c>='A' && c<='Z' ? c+32 : c 中的运算

运算符	名称或运算	结合性	语法	语义或运算序列
+	加	从左到右	exp1+exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，v1 加 v2 得到值 v3
<=	小于或等于	从左到右	exp1<=exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，如果 v1<=v2 成立，得到值为 true，否则得到值为 false
>=	大于或等于	从左到右	exp1>=exp2	计算 exp1 得到值 v1，计算 exp2 得到值 v2，如果 v1>=v2 成立，得到值为 true，否则得到值为 false
&&	逻辑与	从左到右	exp1&&exp2	计算 exp1 得到 bool 类型的值 b1，若 b1 值为 true，计算 exp2 得到 bool 类型的值 b2，计算 b1&&b2 得到 bool 类型的值 b3；否则 exp1&&exp2 的结果为 false
? :	条件运算	从右到左	exp1 ? exp2 : exp3	计算 exp1 得到 bool 类型的值 b，若 b 为 true，计算 exp2 得到变量 x1 或值 v1；若 b 为 false，计算 exp3 得到变量 x2 或值 v2
=	赋值	从右到左	expL=expR	计算 expL 得到变量 x，计算 expR 得到值 v，将值 v 转换为变量 x 的类型规定的存储格式，并存到变量 x 的内存中，得到变量 x

2. 标注运算的计算顺序

按照运算的优先级，表达式 c = c>='A' && c<='Z' ? c+32 : c 中，最后计算的运算是=，倒数第 2 个运算就是条件运算。条件运算的语义明确规定了应先计算其中的条件 c>='A' && c<='Z'，按照这个规定，在条件 c>='A' && c<='Z'中寻找最先计算的运算，找到的运算为>。寻找最先计算的运算的过程如图 3.22 所示。

如图 3.22 所示，在确定计算顺序时，用到了条件运算的语义，要特别注意。

确定了条件 c>='A' && c<='Z'中所有运算的计算顺序后，可依次确定表达式 c+32 和 c 中运算的计算顺序。整个条件表达式的计算顺序如图 3.23 所示。

3. 标注数据类型

按照表达式 c = c>='A' && c<='Z' ? c+32 : c 的计算顺序，根据附录 B 中运算的语义，依次确定每个运算结果的数据类型，并在计算顺序图上标注。标注数据类型后的计算顺序图如图 3.24 所示。

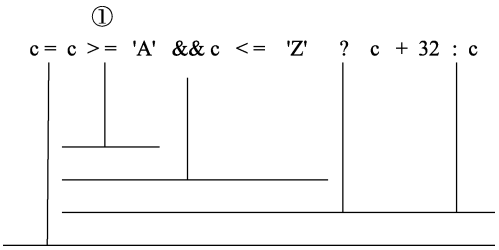


图 3.22　寻找最先计算的运算的过程

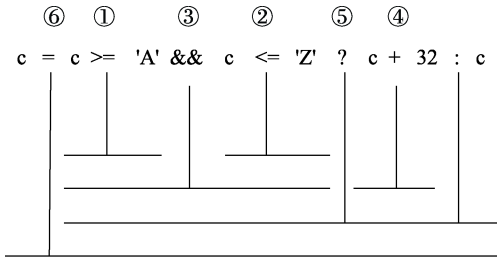


图 3.23　整个条件表达式的计算顺序

如图 3.24 所示，表达式在计算过程中，不需要增加自动类型转换，非常安全。

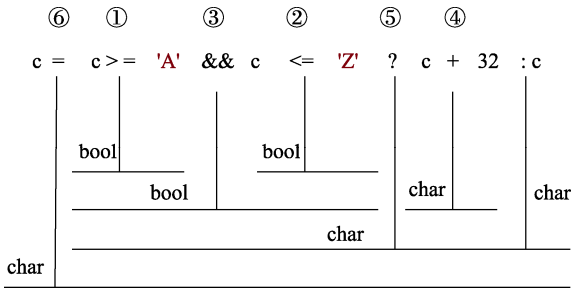


图 3.24　标注数据类型后的计算顺序

条件运算和逻辑运算的语义比较复杂，在标注数据类型过程中，可画出表达式的运算序列图，以方便计算表达式的值。条件表达式的运算序列如图 3.25 所示。

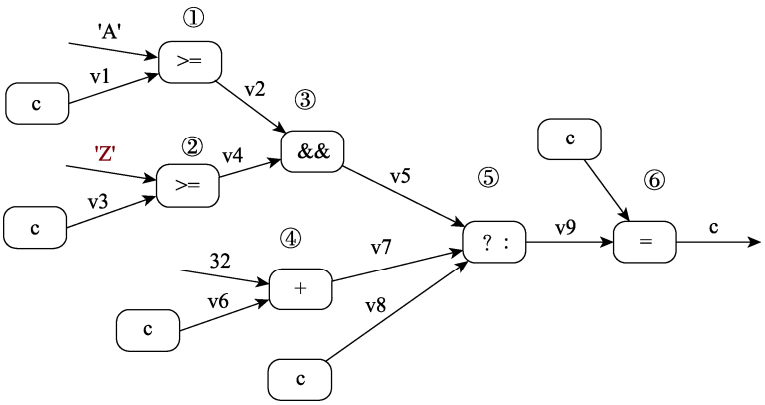


图 3.25　条件表达式的运算序列

在图 3.25 所示的运算序列中，标出了所有的运算步骤，但没有表示出&&和条件运算的语义。在计算表达式时，还需要根据具体的计算结果并结合&&和条件运算的语义，判断第②步和第④步是否执行。

4. 求表达式的值

表达式 c = c>='A' && c<='Z' ? c+32 : c 中包含了条件运算，可分 3 种不同情况选择变量 c 的值，并计算表达式，以验证其正确性。

下面讨论按照&&和条件运算的语义计算条件表达式的过程。

第 1 种情况：当 c 为'A'时的计算过程如图 3.26 所示。

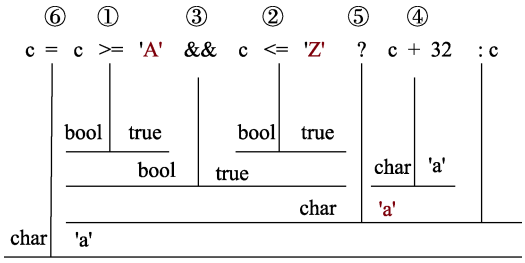


图 3.26　当 c 为'A'时的计算过程

在计算第③步&&时，因 c>='A' 结果为 true，按照&& 的语义，计算第②步中的 c<='Z'。在计算第⑤步的条件运算时，因第③步的结果为 true，按照条件运算的语义，计算第④步 c+32 而没有从变量 c 中取值。

第 2 种情况：当 c 为'c'时的计算过程如图 3.27 所示。

在计算第⑤步的条件运算时，因第③步的结果为 false，按照条件运算的语义，没有计算第④步 c+32 而从变量 c 中取值。

第 3 种情况：当 c 为'6'时的计算过程如图 3.28 所示。

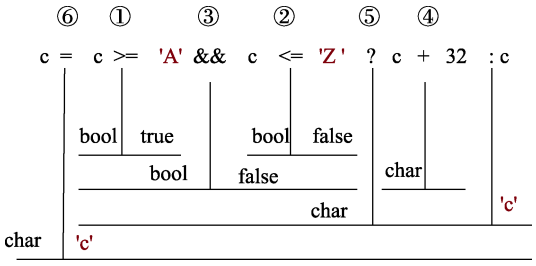


图 3.27　当 c 为'c'时的计算过程

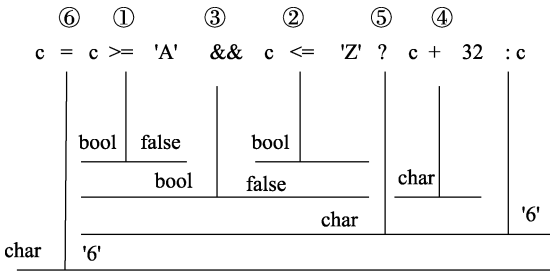


图 3.28　当 c 为'6'时的计算过程

按照&&和条件运算的语义，计算时跳过了第②步、第④步。

3.6　I/O 流及其运算

“流”是一个抽象概念，在计算机中的应用非常广泛。用流描述网络中的数据，称为数据流；描述视频、音频数据，称为视频流、音频流；描述文本数据，称为字符流；描述二进

制数据，称为二进制数据流。

计算机的操作系统预先定义了输入设备、输出设备、错误输出设备和日志输出设备 4 个标准输入输出设备供程序中使用。默认情况下，标准输入设备映射到键盘，标准输出设备和标准错误输出设备映射到显示器，标准日志输出设备映射到打印机。

4 个标准设备都属于字符设备，计算机中使用“字符流”的方式输入输出数据，详见 2.9 节内容。

C++在 iostream 头文件中预定义了 cout、cin、cerr 和 clog 4 个“流”对象，分别对应计算机的输出设备、输入设备、错误输出设备和日志输出设备 4 个标准输入输出设备。

为 cout、cerr 和 clog 定义了插入运算“<<”，实现输出功能，为 cin 定义了插入运算“>>”，实现输入功能。插入运算的语法和语义如表 3.7 所示。

表 3.7　插入运算的语法和语义

运算符	名称或运算	结合性	语法	语义或运算序列
<<	插入运算	从左到右	cout<<exp	计算 exp，得到 T 类型的值 v，将 v1 按照 T 类型和显示格式转换为字符串 s，在标准输出设备的当前光标处依次显示 s 中的字符，光标自动移到下一个位置，返回 cout 对象
>>	插入运算	从左到右	cin>>exp	计算表达式 exp 得到变量 x，按照变量 x 的数据类型从标准输入设备上读入字符串并转换为相应数据类型的值 v1，保存到变量 x 中，得到 cin

学习 cout 和 cin 不仅是为了输出输入数据，而且是为了使用流来进行数据交换，如文件流、字符流、二进制流等。

3.6.1　输出数据

输出插入的运算符为“<<”，语法为 cout<<exp，语义为，计算 exp，得到 T 类型的值 v，将 v 按照 T 类型和显示格式转换为字符串 s，在标准输出设备的当前光标处依次显示 s 中的字符，光标自动移到下一个位置，返回 cout 对象。插入运算<<的语义如图 3.29 所示。

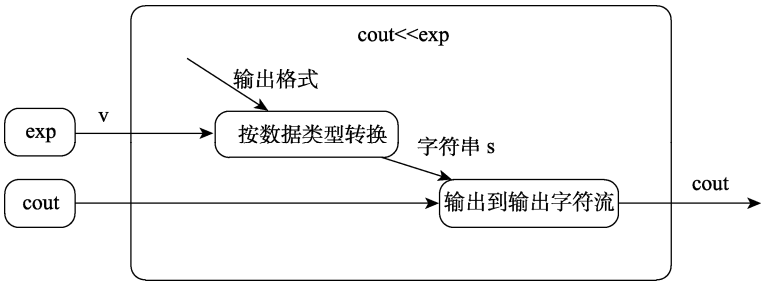


图 3.29　插入运算<<的语义

输出插入运算是根据字符流工作原理定义的，详见 2.9 节内容。

与其他运算一样，输出插入运算也有运算结果，只是运算结果为 cout 对象，而不是显

示的内容。算术运算的结果是所需的功能，运算结果与功能统一，但输出插入运算的结果与功能分离，运算结果是 cout 对象，功能才是输出一个值的内容。

例如：

```
int x = 3;
float y = -7.5;
cout <<"In add(), received "<< x <<" and " << y << endl;
```

其中，使用插入运算实现了输出功能。输出表达式的计算顺序如图 3.30 所示。

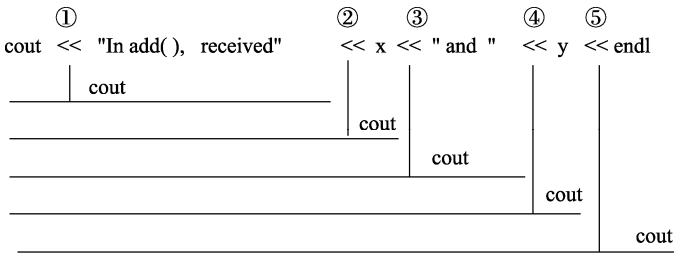
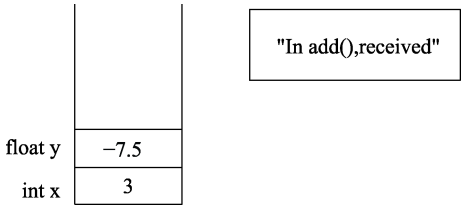


图 3.30　输出表达式的计算顺序

执行到该表达式时的内存状态如下所示。



计算机按照如图 3.30 所示的计算顺序，根据插入运算<<的语义逐个执行输出表达式中的 5 个运算。执行输出表达式前，屏幕上的输出结果：

```
_
```

计算机屏幕上的光标在左上方，空白处的横线代表光标。

① 运行 cout << "In add(), received "。

输出的是字符串"In add(), received "，跳过前面的步骤，直接输出字符串。从当前光标处开始，一个一个地输出" In add(), received "中的字符，每显示一个字符光标自动移到下一位置。屏幕上的输出结果：

```
In␣add(),␣received␣␣
```

② 运行 cout<<x。

输出的是变量 x，从取变量 x 的值开始执行插入运算<<的语义。首先，从 x 指定的内存单元中取出 int 类型的值 3，并按照 int 类型的默认显示格式将整数 3 转换为字符串"3"，然后从光标处开始，在屏幕上显示字符串"3"中的字符，最后返回 cout 对象。屏幕上的输出结果：

```
In␣add(),␣received␣3_
```

③ 运行 cout << " and "。

逐个显示字符串"␣and "中的字符，具体步骤同①。屏幕上的输出结果：

```
In␣add(),␣received␣3␣and␣_
```

④ 运行 cout << y。

输出 int 类型变量 y 的值，具体步骤同②。屏幕上的输出结果：

```
In␣add(),␣received␣3␣and␣-7.5_
```

⑤ 运行 cout << endl。

endl 是 endline 的缩写，对应 ASCII 码中的“回车符”“换行”两个不可见字符，含义为将光标移到下一行的第 1 个位置。屏幕上的输出结果：

```
In␣add(),␣received␣3␣and␣-7.5↵
_
```

至此，整个表达式执行完毕，屏幕上也输出了 In add(), received 3 and -7.5，光标在下一行的第 1 个位置。

3.6.2 输入数据

输入插入的运算符为“>>”，语法为 cin>>exp，其语义为，计算表达式 exp 得到变量 x，按照变量 x 的数据类型从标准输入设备上读入字符串并转换为相应数据类型的值 v1，保存到变量 x 中，得到 cin。

输入插入运算>>是输入，与输出插入运算<<的功能相反，两者的语义也相反，但它们的运算结果都是一个流。输入插入运算>>的语义如图 3.31 所示。

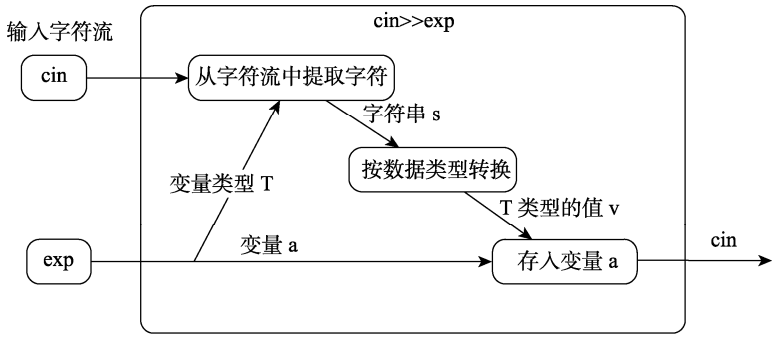


图 3.31 输入插入运算>>的语义

如图 3.31 所示，与输出插入运算相比，计算表达式 exp 得到变量 x，而不是一个值；其核心内容是“按照变量 x 的数据类型从标准输入设备上读入字符串并转换为相应数据类型的

值 v1，保存到变量 x 中”，从键盘读入用户输入的字符串，并按照“字符流”的方式转换为相应数据类型的值 v1，然后保存到变量 x 中，最后的结果为输入流的对象 cin。

例如，输出一个整数到一个变量 a，然后输出变量 a 的值，代码如例 3.9 所示。

【例 3.9】 输入一个整数并输出。

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

void main(){
    //输入整数
    int a;
    cin >> a;
    cout << a << ",0x" << hex << a << endl;
}
```

程序运行到 cin >> a 时，等待用户输入数据。用户在键盘输入"123"，此时屏幕上结果为：

```
123_
```

此时，计算机还没有将用户输入的字符串"123"送给程序，字符串还在计算机的缓冲区。

当用户再输入一个"↵"，计算机才将输入的字符串以字符流的形式送给程序。输入一个整数的字符流如图 3.32 所示，这时，字符流中的当前位置为第 1 个数字"1"。

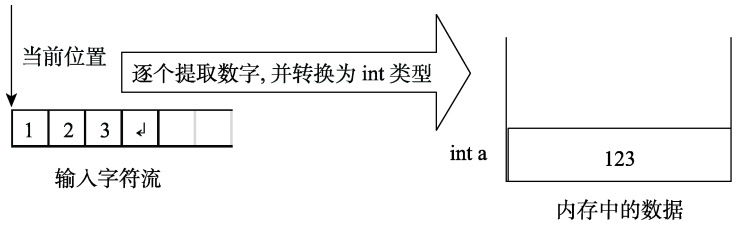


图 3.32　输入一个整数的字符流

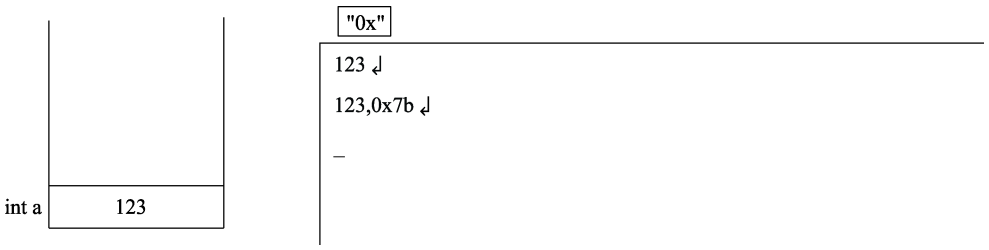
变量 a 是 int 类型，按照整数的记数法

$$\pm d_n d_{n-1} \cdots d_1 d_0$$

只能包含符号和数字，因此，cin 从字符流中提取第 1 个字符'1'，是数字，并将当前位置移到第 2 个字符'2'；继续提取第 2 个字符'2'，是数字，并将当前位置移到第 3 个字符'3'；继续提取第 3 个字符'3'，是数字，并将当前位置移到第 4 个字符'↵'；继续提取第 4 个字符'↵'，不是数字或符号，结束从字符流中提取字符。

cin 从字符流中提取到字符串"123"，然后将字符串"123"转换为整数 123，存储到变量 a 的内存。

后面的输出语句将变量 a 中的值取出来，显示在屏幕上。内存状态和输出结果为：

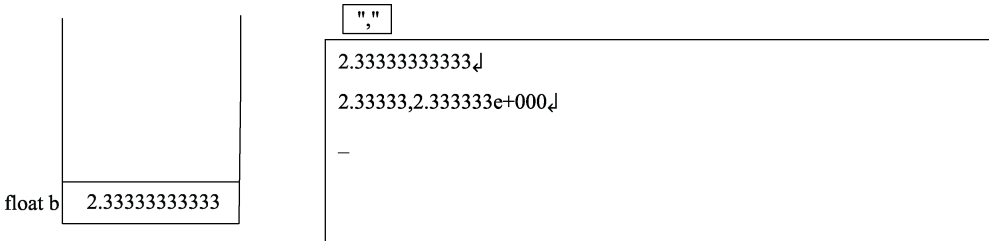


读者可分别输入"12␣3↵"（其中 ␣ 表示空格）、"12.3↵"和"12b3↵"，观察程序的输入输出情况以理解 cin 的工作原理。

例如：输入实数。

```
float b;
cin >> b;
cout << dec << b << "," << scientific << b << endl;
```

运行时输入"2.33333333333↵"，内存状态和输出结果为：



常用的实数记数法为：

$$\pm d_n d_{n-1} \cdots d_1 d_0 . d^1 d^2 \cdots d^m$$

其中，d_i、d^i 为 0~9 中的一个数字。按照这种实数的记数法，一个实数可包含符号、数字和小数点。

cin 从输入字符流中提取字符时，遇到符号、数字和小数点以外的字符，就结束提取。其过程为，cin 从"2.33333333333↵"中读取一个字符'2'，是数字，读取一个字符'.'，是实数中的小数点，读取一个字符'3'，是数字，直到读取一个字符'↵'，不是实数中的符号，结束读取字符，得到一个字符串"2.33333333333"，然后将它转换为 float 数 2.333333，存储到变量 b 中。

将"2.33333333333"转换为 float 数 2.333333 时，因实数存在精度问题，在转换时将超出 float 有效位数的部分扔掉了，只保留了 7 位有效数字（也可能是 6 位有效数字）。

读者可输入+23.45、–12.34 等带符号的实数，也可将变量 b 的数据类型改为 double，再运行程序，观察输入输出情况，理解输入流的工作原理。

实数的科学记数法，其格式如下：

$$\pm 0.d_1 d_2 \cdots d_m \times 10^{\pm e_n \cdots e_1 e_0}$$

可输入–34.56e+4 等科学记数法的实数，观察输入输出情况，理解输入流的工作原理。

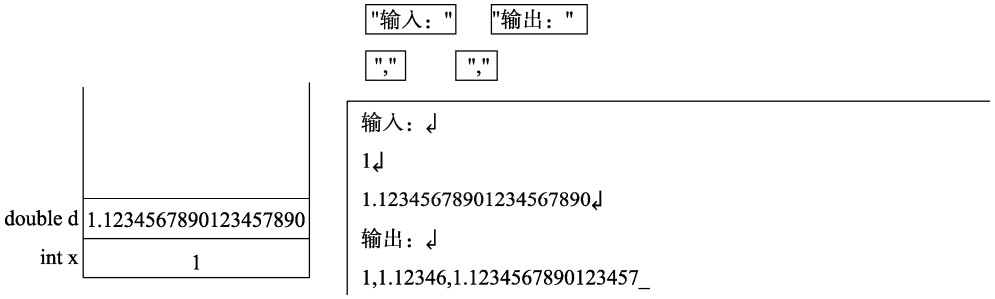
例如：输入多个数。

```
int x;
```

```
    double d;
    cout << "输入："<<endl;
    cin >> x >> d;
    cout << "输出：" << endl;
    cout << x << "," << d << "," << setprecision(18) << d;
```

其中，表达式 cin >> x >> d 的计算顺序如图 3.33 所示。读者可自己写出其语义并画出程序的内存图。

第 1 次运行时，内存状态和输入输出为：



执行 cin >> x 时，用户输入第一个'↲'后，计算机将'↲'送给 cin，cin 判断接收到的是'↲'，扔掉，继续等待；用户输入"1↲"后，计算机将"1↲"送给 cin，cin 从"1↲"读入数字 1，再读入'↲'，判断出不是数字或符号，就将读到的字符'1'，转换为整数 1，并转换为二进制存储到整型变量 x 的内存。最后，返回 cin，跳转去执行 cin >> d。

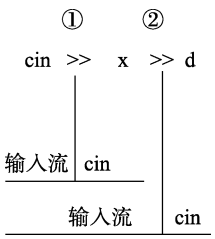


图 3.33　表达式 cin >> x >> d 的计算顺序

执行 cin >> d 时，cin 接收到前面未处理的'↲'，判断后扔掉，继续等待；用户输入"1.123456789012345678901↲"后，计算机将其送给 cin，最终，cin 将其转换实数存储到变量 d。输入过程中字符流的 3 个状态如图 3.34 所示。

第 2 次运行时，内存状态和输入输出为：

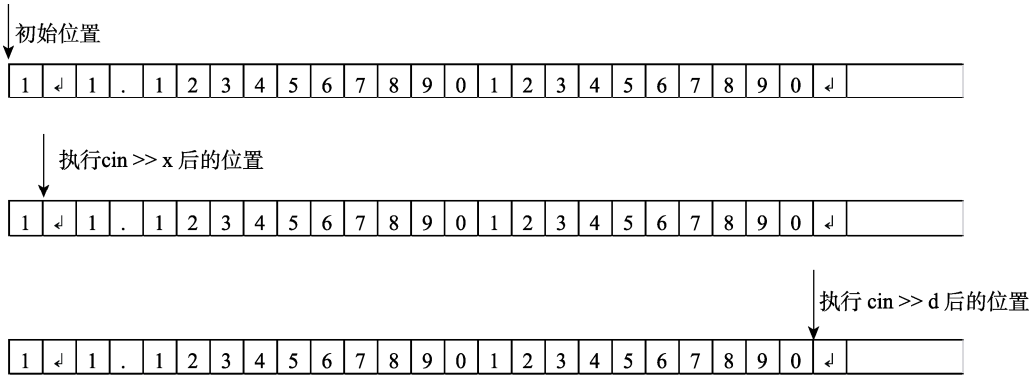
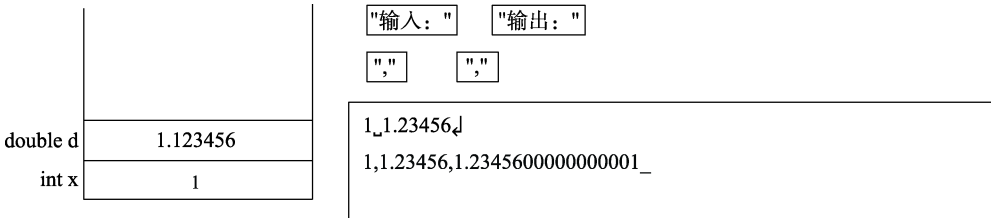


图 3.34　输入过程中字符流的 3 个状态



上面两次运行程序，一次用 '↲' 分隔两个数，一次用空格 '␣' 分隔两个数，都能得到预期结果。

读者可以输入“12,23.8”“23a456”等使用其他字符分隔数字，观察运行时变量的值，理解输入流的工作原理。

输出流的基本特征是，每次输入一个字符，字符在光标处显示，然后光标就自动移到下一个位置；输入流的基本特征是，每次从当前位置读取一个字符，并将当前位置自动后移一个位置。不管是输出流还是输入流，都有个“当前位置”的概念，每输出或读取一个字符，“当前位置”都会自动移到下一个位置，这种处理方式中的字符如水一样不断从人的面前（当前位置）流过，所以形象地称为“流”。

“流”是编程中非常重要的模式，应用广泛，读者可通过学习输入流，培养“流”这种思维模式。

3.7 分支的调试与维护


视频讲解

分支是由分支语句实现的，应从两个层次进行调试：第一，调试分支语句的流程，判断流程是否正确；第二，判断语句中包含的表达式是否正确。前面学习了调试表达式的方法，下面重点介绍调试程序流程的方法。

3.7.1 代码格式的重要性

代码格式的书写规范性是对编写程序的基本要求，也是调试程序的基础，初学者一般意识不到它的重要意义。

目前的编程环境一般都提供了自动规范代码格式的功能，如在 VS2013 编程环境中编辑代码时，会自动调整代码的格式。如果出现语法错误，会给出错误提示，代码格式也会乱。

如例 3.7 knick-knack 程序中的代码格式，借助了 VS2013 编程环境，能反映如图 3.18 所示的多分支结构。

借助编程环境纠正代码中的语言细节错误，并规范代码的结构，是一个好的方法。

按照规范书写程序，在输入代码时就能发现很多语法错误，能缩短调试的时间，提高调试的效率。

代码书写规范是程序员的基本功，也能体现程序员的基本素质。

3.7.2　调试分支的逻辑错误

调试分支的主要目标是确保每种情况都是正确的，基本方法是每种情况选择一组数据，分别在这些数据组上执行程序，通过调试器观察程序执行流程及变量的值，对比人工执行流程和表达式的计算结果，判断程序是否正确。

如例 3.7 中的程序，总共分了 5 种情况，需要为 number 分别选择 1、2、3、4 共 4 个值，加上一个其他的值，至少执行程序 5 次。每次执行程序时，判断每次输入的数是否按照预期的流程执行，是否得到期望的输出结果。

如果执行流程不符合预期，一般都会人工计算条件表达式，并比较计算机中变量的值，确定缺陷的位置，再修改程序。

在跟踪调试分支代码时，总希望能从代码的结构中直接读出代码的流程，因此，要求在设计分支流程时，提前考虑，按照流程写出的代码结构能否体现的分支流程。

例如，从键盘输入一年份，判断该年份是否为闰年，并将结果打印在屏幕上。根据以往的知识知道，如果某年能被 4 整除并且不能被 100 整除，或者能被 400 整除，则该年为闰年。

很可能设计出两个非常类似的流程，其功能相同，流程结构也相同，但它们的判断条件互为“否定”，分支上真假值刚好相反。两个流程如图 3.35 和图 3.36 所示。

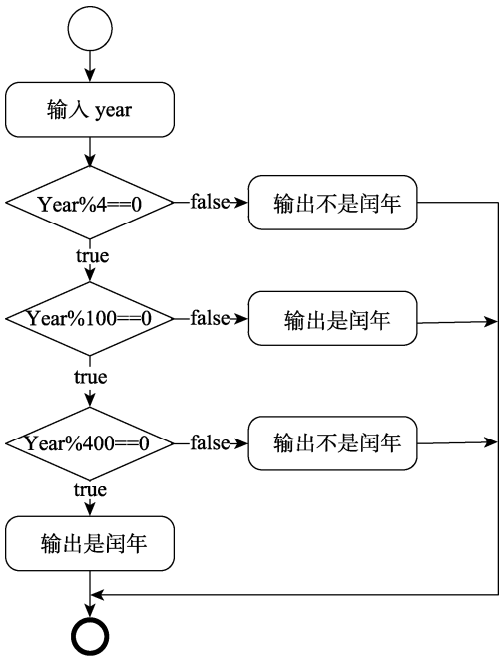


图 3.35　求闰年的流程图 1

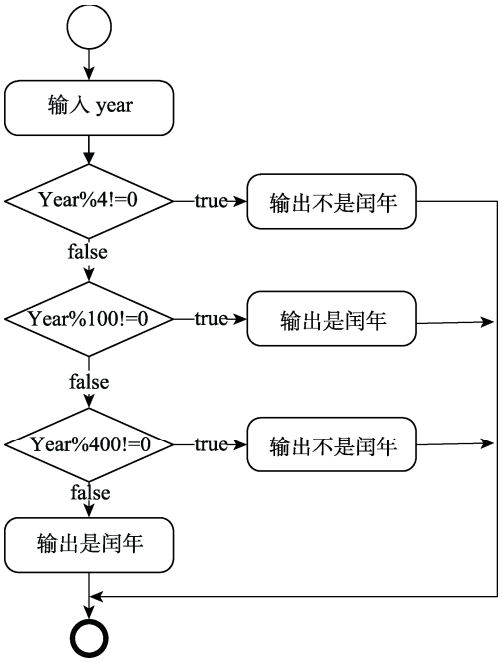


图 3.36　求闰年的流程图 2

编写代码时会发现，按照图 3.35 所示的流程，适合使用 if 嵌套模式描述，写出的代码结构与流程的结构相差太大，从代码中很难发现有 4 个分支，导致设计测试用例困难，增加了调试代码的难度和工作量。

而图 3.36 所示的流程，每个向下的分支都是 false，适合使用多分支模式。使用多分支

模式判断是否为闰年，如例 3.10 所示。

【例 3.10】 使用多分支模式判断是否为闰年。

```
#include<iostream>
using namespace std;
//该程序从键盘输入一年份,判断该年份是否为
//闰年,并将结果打印在屏幕上
//程序编号
void main()
{
    int year;             //定义整型变量 year
    cout << "请输入年份:";
    cin >> year;          //从键盘输入年份
    if (year % 4 != 0)
        cout << year << " is not leap." << endl;
    else if (year % 100 != 0)
        cout << year << " is leap." << endl;
    else if (year % 400 != 0)
        cout << year << " is not leap." << endl;
    else
        cout << year << " is leap." << endl;
}
```

在调试程序时，直接从代码中就可很容易判断有 4 种情况，在调试程序时，为每种情况选择一个具体的年份，覆盖每条分支，如为 year 分别选择 2007、2000、1900 和 2020 共 4 个值，然后分别执行程序 4 次，并判断每次输入的数是否按照预期的流程执行，是否得到期望的输出结果。如果不符合预期，说明程序存在缺陷，需要重新修改程序，再运行调试程序，直到得到执行预期的流程并得到结果。

调试程序时，输入的数据集称为测试用例，设计出的测试用例对调试的工作量和调试质量有很大影响，因此，调试程序的关键是设计测试用例，调试分支程序的基本方法称为“分支覆盖”，即设计出的测试用例刚好覆盖每条分支。

在调试程序时，可采用单步跟踪方式观察程序的执行流程，并观察变量中的值，以判断执行流程是否符合预期、中间结果是否正确，最后通过观察程序的输出结果判断程序是否正确。

还可使用下面的条件表达式判断是否为闰年。

```
(year % 4 == 0 && year % 100 != 0) || year % 400 == 0
```

使用条件表达式编写的代码比较短，但需要对条件表达式进行深入分析才能设计出测试用例，增加了调试的难度。

3.8 本章小结

本章主要学习了构造分支结构的方法和分支程序的 4 种典型编程模式、使用 I/O 流实现

输入输出的方法以及使用流程图描述程序流程的方法。

学习了构造分支结构的基本知识和原理，重点学习了单分支、双分支、嵌套和多分支 4 种典型编程模式以及 if 语句和 switch 语句，需要掌握使用分段函数构造分支结构的方法，能够从分支程序的 4 种典型编程模式中选择适当的编程模式，掌握使用分支语句描述分析结构的方法。

学习了关系运算、逻辑运算和条件运算，深入学习了使用关系运算和逻辑运算描述分支条件的方法以及使用条件运算描述分支结构的方法，需要掌握使用计算顺序图计算关系表达式和逻辑运算表达式的方法。

学习了输入流和输出流的工作方式、I/O 流及两个插入运算、使用 I/O 流实现输入输出的方法，为今后学习文件流、字符串打下基础。

学习使用流程图描述程序流程的方法，需要掌握使用流程图描述顺序结构和分支结构的方法。

学习了调试和维护分支程序的基本知识和基本方法，深入学习了通过分支流程发现逻辑错误的方法，需要学会通过流程图调试程序。

3.9　习题

1. 读程序并完成各问题。

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
    cout << "please input the b key to hear a bell,\n";
    char ch = getchar();
    if (ch == 'b'){
        cout << '\a';
    }
    else {
        if (ch == '\n')
            cout << "what a boring select on…\n";
        else
            cout << "bye!\n";
    }
}
```

（1）程序运行时，分别从键盘输入 a、b、↵，写出运行结果。

（2）将程序中的 ch == 'b'替换为 ch = 'b'，程序运行时，分别从键盘输入 a、b、↵，写出运行结果。

（3）画出程序的流程图。

2. 结合 ASCII 码表编写程序，实现输入一个字符并判断输入的字符是否是英文字母（包

含大小写）。

3. 编程实现输入 3 个整数，输出其中的最大值。先用 if 语句，再用条件运算编程实现。

4. 编程实现输入一个整数，判断其是否能被 2、5 整除，并输出以下信息。

（1）能被 2 整除。

（2）能被 5 整除。

（3）能被 2 和 5 同时整除。

5. 编程实现输入一个整数，判断其是否能被 4、25 整除。

6. 编程实现输入一个百分制的整数成绩，输出相应的 5 分制成绩，90 分以上为 A，80~89 分为 B，70~79 分为 C，60~69 分为 D，60 分以下为 E。

7. 编写程序求一元二次方程 $ax^2+bx+c=0$ 的根，并调试通过。