



清华大学出版社  
第五事业部

官方微信:清华书友 @qinghuashuyou  
扫一扫 初次关注有礼!



Beginning C++

# C++入门经典 (第4版)

[美] Ivor Horton 著  
石磊 译

Apress®

清华大学出版社



清华大学出版社  
第五事业部

官方微信:清华书友 @qinghuashuyou  
扫一扫 初次关注有礼!



# C++入门经典

## (第4版)

[美] Ivor Horton 著  
石磊 译

清华大学出版社

北京



清华大学出版社  
第五事业部

官方微信:清华书友 @qinghuashuyou  
扫一扫 初次关注有礼!



Ivor Horton

Beginning C++

EISBN: 978-1-4842-0008-7

Original English language edition published by Apress Media. Copyright © 2015 by Apress Media. Simplified Chinese-language edition copyright © 2015 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Apress 出版公司授权清华大学出版社出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2015-3298

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

C++入门经典(第4版)/(美)霍尔顿(Horton,I.)著;石磊译. —北京:清华大学出版社,2015

书名原文: Beginning C++

ISBN 978-7-302-40628-0

I. C… II. ①霍…②石… III. ①C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2015)第 150327 号

责任编辑:王军于平

装帧设计:孔祥峰

责任校对:成凤进

责任印制:

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:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, [c-service@tup.tsinghua.edu.cn](mailto:c-service@tup.tsinghua.edu.cn)

质量反馈:010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 刷 者:

装 订 者:

经 销:全国新华书店

开 本:185mm×260mm 印 张:38 字 数:973千字

版 次:2015年8月第1版 印 次:2015年8月第1次印刷

印 数:1~5800

定 价:69.80元

产品编号:



# 译者序

科学计算、分布式应用、嵌入式行业、智能控制、算法研究，乃至学术讨论和上机考试都会有一种语言的身影，那就是大名鼎鼎的 C++ 语言。原汁原味的 C++ 目前已经执行到 C++14 标准。听到这门编程语言，多数人伴随而来的是晦涩、复杂、强大等关键词！这样的感觉实际上是真实的。

对于这门复杂的编程语言，本书首先从一个最简短的 C++ 程序讲起，通过对这个完整的程序的实际编写引申出一些相关的知识，然后在后面的教程中对该程序不断地扩大和完善，这样，读者不至于一上来就被 C++ 吓坏，同时也能深刻地理解 C++ 的各个特性的设置目的。

有抱负的程序员必将面对三重障碍：

首先，必须掌握遍布程序设计语言中的各类术语。术语是专业人士及优秀业余爱好者之间的交流必不可少的，本书不仅详细讲解了这些术语，还强调如何自如地在各种环境下使用它们。

其次，必须理解如何使用语言元素，而不仅仅只是知道它们的概念。本书采用了代码片段来帮助理解语言元素的语法和作用，还用一些实际应用示例展示语言特性如何应用于特定的问题。

最后，必须领会如何在实际场景中应用该语言。本书针对特定的问题应用所学的知识，给出合理的解决方案，不仅帮助读者获得开发应用程序的能力与信心，了解如何联合以及更大范围地应用语言元素；而且能让读者了解设计实际应用程序与管理实际代码会碰到的问题。

读者使用本书时，记住只有通过动手实践才能学会编程，在学习的过程中，肯定会时不时犯许多错误而感到沮丧。当觉得自己完全停滞时，你要做的就是坚持。最终你一定会体验到成功的喜悦，回头想想，你会觉得它并没有你想象中的那么难。

本版与上一版的出版时间相距 10 年，期间 C++ 已经有了很大的变化，所以本版对前一版进行了彻底的修订，不仅内容经过了重新组织，添加了不少新内容，对每个示例都做了相应的修改，使它们能在新的 C++ 14 标准下编译运行。而且显著改善了可读性，充分体现了 C++ 语言的最新进展和当前的业界最佳实践。

本书是 C++ 初学者的权威指南。无论你是从事软件开发还是其他领域的工作，本书将为你打开程序开发之门。本书还是中高级程序员的必备参考。通过观察程序设计大师如何处理编程中的各种问题，使你获得新的领悟和指引。



清华大学出版社

第五事业部

II C++入门经典(第4版)

官方微信:清华书友 @qinghuashuyou  
扫一扫 初次关注有礼!



在这里要感谢清华大学出版社的编辑们,他们为本书的翻译投入了巨大的热情并付出了很多心血。没有你们的帮助和鼓励,本书不可能顺利付梓。

本书由石磊翻译,参加本次翻译活动的还有孔祥亮、陈跃华、杜思明、熊晓磊、曹汉鸣、陶晓云、王通、方峻、李小凤、曹晓松、蒋晓冬、邱培强、洪妍、李亮辉、高娟妮、曹小震、陈笑。对于这本经典之作,译者本着“诚惶诚恐”的态度,在翻译过程中力求“信、达、雅”,但是鉴于译者水平有限,错误和失误在所难免,如有任何意见和建议,请不吝指正。

最后,希望读者通过阅读本书能早日步入C++语言编程的殿堂,领略C++语言之美!

译者



## 作者简介



Ivor Horton 毕业于数学系，被人以获得巨大荣誉且代价极低而诱入信息技术行业。尽管他做了大量的工作，获得的荣誉并不多，但他继续在计算机领域工作。他主要关注编程、系统设计、咨询、非常复杂的项目的管理和实现。

Ivor 在工程设计和生产控制系统的设计、实现方面有多年的经验。他偶尔使用各种编程语言开发有用的应用程序，主要教科学家和工程师开发这些程序。他目前出版的著作包括 C、C++ 和 Java 方面的教材。目前，他不编写编程图书或给他人提供建议时，会钓鱼、旅游、享受生活。



## 技术编辑简介



**Michael Thomas** 作为独立撰稿人、团队领袖、项目经理和工程负责人，在软件开发方面工作了 20 余年。**Michael** 在移动设备方面有 10 余年的经验。他当前的重点是医疗领域，使用移动设备加速患者和医护人员之间的信息传递速度。



# 前 言

欢迎使用《C++入门经典(第4版)》。本书修订并更新了上一版(*Beginning ANSI C++*)。自上一版出版以来, C++语言有了很大的扩展和改进, 但不太可能把 C++的所有内容压缩到一本书中。本书提供的 C++语言基础知识和标准库功能, 足以让读者编写自己的 C++应用程序。掌握了本书介绍的知识, 读者应能毫无困难地扩展 C++专业知识的深度和广度。C++要比许多人想象的更容易理解。本书不需要读者具备任何编程知识。如果你非常渴望学习, 并具备逻辑思考的能力, 掌握 C++就会比想象的更容易。开发 C++技巧, 学习数百万人已在使用的语言, 掌握 C++技能, 它提供了在几乎任何环境下开发应用程序的能力。

本书的 C++语言对应最新的 ISO 标准, 一般称为 C++ 14。C++ 14 对以前的标准 C++ 11 进行了较小的扩展, 所以本书的内容大都不专用于 C++ 14。本书的所有示例都可以使用目前遵循 C++ 11 的编译器来编译和执行。

## 使用本书

要通过本书学习 C++, 需要一个遵循 C++ 11 标准的编译器和一个适合编写程序代码的文本编辑器。目前, 有几个编译器兼容 C++ 11, 其中一些是免费的。

GNU Project 生产的 GCC 编译器全面支持 C++ 11, 是一个开源产品, 可免费下载。安装 GCC 并将它与合适的编辑器一起使用, 对新手而言略有难度。安装 GCC 和合适编译器的一种简单方法是, 从 <http://www.codeblocks.org> 上下载 Code::Blocks。Code::Blocks 是 Linux、Apple Mac OS X 和 Microsoft Windows 的一个免费 IDE, 它允许使用几个编译器(包括用于 GCC、Clang 和 open Watcom 的编译器)开发程序。这表示, 安装 Code::Blocks 会获得 C、C++和 Fortran 的支持。

另一种方法是使用在 Microsoft Windows 下运行的 Microsoft Visual C++, 它不仅完全兼容 C++ 11, 而且已经安装好了。其免费版本是 Microsoft Visual Studio 2013 Express。编写本书时, 它可以编译本书的大多数示例, 最终应能编译所有示例。Microsoft Visual C++可以从 <http://www.microsoft.com/en-us/download/details.aspx?id=43733> 上下载。与 GCC 相比, Microsoft Visual C++编译器的限制多一些, 但根据它对 C++ 11 的支持力度, 这是一个专业的编译器, 还支持其他语言, 例如 C#和 Basic。当然, 也可以安装这两个编辑器。还有其他支持 C++ 11 的编辑器, 在网上搜索会很快找到它们。



本书的内容循序渐进,所以读者应从头开始一直阅读到最后。但是,没有人能仅从一本书中获得所有的编程技巧。本书仅介绍如何使用 C++编程,读者应自己输入所有的例子,而不是从下载文件中复制它们,再编译和执行输入的代码,这似乎很麻烦,但输入 C++语句可以帮助理解 C++,特别是觉得某些地方很难掌握时,自己输入代码就显得非常有帮助。如果例子不工作,不要直接从书中查找原因,而应在自己输入的例子代码中找原因,这是编写 C++代码时必须做的一个工作。

犯错误也是学习过程中不可避免的,练习应提供大量犯错误的机会,最好自己编几个练习题。如果不确定如何编写代码,应翻看前面的内容。犯的错误越多,对 C++的功能和错误的原因认识得就越深刻。读者应完成所有的练习,记住不要看答案,直到肯定不能自己解决问题为止。许多练习都涉及某章内容的一个直接应用,换言之,它们仅是一种实践,但也有一些练习需要多动脑子,甚至需要一点灵感。

希望每个人都能成功驾驭 C++。

—Ivor Horton



# 目 录

第 1 章 基本概念	1	1.11.1 三字符序列	18
1.1 现代 C++	1	1.11.2 转义序列	18
1.2 C++ 程序概念	2	1.12 过程化编程方法和面向 对象编程方法	20
1.2.1 注释和空白	2	1.13 本章小结	21
1.2.2 预处理指令和头文件	3	1.14 练习	22
1.2.3 函数	3	第 2 章 基本数据类型	23
1.2.4 语句	4	2.1 变量、数据和数据类型	23
1.2.5 数据输入输出	4	2.1.1 定义整型变量	24
1.2.6 return 语句	5	2.1.2 定义有固定值的变量	26
1.2.7 名称空间	5	2.2 整型字面量	26
1.2.8 名称和关键字	6	2.2.1 十进制整型字面量	27
1.3 类和对象	6	2.2.2 十六进制的整型字面量	27
1.4 模板	7	2.2.3 八进制的整型字面量	27
1.5 程序文件	7	2.2.4 二进制的整型字面量	28
1.6 标准库	7	2.3 整数的计算	28
1.7 代码的表示样式	7	2.4 op=赋值运算符	33
1.8 创建可执行文件	8	2.5 using 声明和指令	34
1.9 表示数字	9	2.6 sizeof 运算符	34
1.9.1 二进制数	9	2.7 整数的递增和递减	35
1.9.2 十六进制数	11	2.8 定义浮点变量	37
1.9.3 负的二进制数	12	2.8.1 浮点字面量	38
1.9.4 八进制数	14	2.8.2 浮点数的计算	38
1.9.5 Big-Endian 和 Little-Endian 系统	14	2.8.3 缺点	38
1.9.6 浮点数	15	2.8.4 无效的浮点结果	39
1.10 表示字符	16	2.9 数值函数	40
1.10.1 ASCII 码	16	2.10 流输出的格式化	43
1.10.2 UCS 和 Unicode	17	2.11 混合的表达式和类型转换	45
1.11 C++ 源字符	17	2.11.1 显式类型转换	46



- 2.11.2 老式的强制转换 ..... 48
- 2.12 确定数值的上下限 ..... 49
- 2.13 使用字符变量 ..... 50
  - 2.13.1 使用 Unicode 字符 ..... 51
  - 2.13.2 auto 关键字 ..... 52
  - 2.13.3 lvalue 和 rvalue ..... 52
- 2.14 本章小结 ..... 53
- 2.15 练习 ..... 54
- 第 3 章 处理基本数据类型 ..... 55**
  - 3.1 运算符的优先级和相关性 ..... 55
  - 3.2 按位运算符 ..... 57
    - 3.2.1 移位运算符 ..... 58
    - 3.2.2 使用按位与运算符 ..... 60
    - 3.2.3 使用按位或运算符 ..... 61
    - 3.2.4 使用按位异或运算符 ..... 63
  - 3.3 枚举数据类型 ..... 67
  - 3.4 数据类型的同义词 ..... 70
  - 3.5 变量的生存期 ..... 70
    - 3.5.1 定位变量的定义 ..... 71
    - 3.5.2 全局变量 ..... 71
    - 3.5.3 静态变量 ..... 74
    - 3.5.4 外部变量 ..... 75
  - 3.6 本章小结 ..... 75
  - 3.7 练习 ..... 76
- 第 4 章 决策 ..... 77**
  - 4.1 比较数据值 ..... 77
    - 4.1.1 应用比较运算符 ..... 78
    - 4.1.2 比较浮点数值 ..... 79
  - 4.2 if 语句 ..... 80
    - 4.2.1 嵌套的 if 语句 ..... 82
    - 4.2.2 不依赖编码的字符处理 ..... 84
  - 4.3 if-else 语句 ..... 85
    - 4.3.1 嵌套的 if-else 语句 ..... 87
    - 4.3.2 理解嵌套的 if 语句 ..... 88
  - 4.4 逻辑运算符 ..... 89
    - 4.4.1 逻辑与运算符 ..... 90
    - 4.4.2 逻辑或运算符 ..... 90

- 4.4.3 逻辑非运算符 ..... 91
- 4.5 条件运算符 ..... 92
- 4.6 switch 语句 ..... 94
- 4.7 无条件分支 ..... 98
- 4.8 语句块和变量作用域 ..... 99
- 4.9 本章小结 ..... 100
- 4.10 练习 ..... 100
- 第 5 章 数组和循环 ..... 103**
  - 5.1 数据数组 ..... 103
  - 5.2 理解循环 ..... 105
  - 5.3 for 循环 ..... 106
    - 5.3.1 避免幻数 ..... 107
    - 5.3.2 用初始化列表定义数组的大小 ..... 109
    - 5.3.3 确定数组的大小 ..... 109
    - 5.3.4 用浮点数值控制 for 循环 ..... 110
    - 5.3.5 使用更复杂的循环控制表达式 ..... 112
    - 5.3.6 逗号运算符 ..... 113
    - 5.3.7 基于区域的 for 循环 ..... 114
  - 5.4 while 循环 ..... 115
  - 5.5 do-while 循环 ..... 119
  - 5.6 嵌套的循环 ..... 120
  - 5.7 跳过循环迭代 ..... 123
  - 5.8 循环的中断 ..... 125
  - 5.9 字符数组 ..... 128
  - 5.10 多维数组 ..... 131
    - 5.10.1 初始化多维数组 ..... 134
    - 5.10.2 在默认情况下设置维数 ..... 135
    - 5.10.3 多维字符数组 ..... 136
  - 5.11 数组的替代品 ..... 137
    - 5.11.1 使用 array<T,N>容器 ..... 138
    - 5.11.2 使用 std::vector<T> 容器 ..... 142
    - 5.11.3 矢量的容量和大小 ..... 143



5.11.4 删除矢量容器中的 元素.....	145
5.12 本章小结.....	145
5.13 练习.....	146
<b>第 6 章 指针和引用.....</b>	<b>149</b>
6.1 什么是指针.....	149
6.1.1 地址运算符.....	151
6.1.2 间接运算符.....	152
6.1.3 为什么使用指针.....	153
6.2 char 类型的指针.....	154
6.3 常量指针和指向常量的 指针.....	158
6.4 指针和数组.....	159
6.4.1 指针的算术运算.....	160
6.4.2 计算两个指针之间的差.....	162
6.4.3 使用数组名的指针 表示法.....	162
6.5 动态内存分配.....	165
6.5.1 栈和堆.....	165
6.5.2 运算符 new 和 delete.....	166
6.5.3 数组的动态内存分配.....	167
6.5.4 通过指针选择成员.....	169
6.6 动态内存分配的危险.....	169
6.6.1 内存泄漏.....	169
6.6.2 自由存储区的碎片.....	170
6.7 原指针和智能指针.....	170
6.7.1 使用 unique_ptr<T> 指针.....	172
6.7.2 使用 shared_ptr<T> 指针.....	173
6.7.3 比较 shared_ptr<T> 对象.....	177
6.7.4 weak_ptr<T> 指针.....	177
6.8 理解引用.....	178
6.8.1 定义左值引用.....	179
6.8.2 在基于区域的 for 循环中 使用引用变量.....	180
6.8.3 定义右值引用.....	180
6.9 本章小结.....	181
6.10 练习.....	181
<b>第 7 章 操作字符串.....</b>	<b>183</b>
7.1 更好的 string 类型.....	183
7.1.1 定义 string 对象.....	184
7.1.2 string 对象的操作.....	186
7.1.3 访问字符串中的字符.....	188
7.1.4 访问子字符串.....	190
7.1.5 比较字符串.....	191
7.1.6 搜索字符串.....	196
7.1.7 修改字符串.....	203
7.2 国际字符串.....	207
7.3 包含 Unicode 字符串的 对象.....	208
7.4 原字符串字面量.....	208
7.5 本章小结.....	209
7.6 练习.....	210
<b>第 8 章 定义函数.....</b>	<b>211</b>
8.1 程序的分解.....	211
8.1.1 类中的函数.....	212
8.1.2 函数的特征.....	212
8.2 定义函数.....	212
8.2.1 函数体.....	213
8.2.2 函数声明.....	215
8.3 给函数传送参数.....	217
8.3.1 按值传送机制.....	217
8.3.2 按引用传送.....	223
8.3.3 main() 的参数.....	227
8.4 默认的参数值.....	228
8.5 从函数中返回值.....	231
8.5.1 返回指针.....	231
8.5.2 返回引用.....	235
8.6 内联函数.....	236
8.7 静态变量.....	237
8.8 函数的重载.....	239
8.8.1 重载和指针参数.....	241
8.8.2 重载和引用参数.....	241
8.8.3 重载和 const 参数.....	243
8.8.4 重载和默认参数值.....	244
8.9 函数模板.....	245



8.9.1 创建函数模板的实例····· 246

8.9.2 显式指定模板参数····· 247

8.9.3 函数模板的特例····· 248

8.9.4 函数模板和重载····· 249

8.9.5 带有多个参数的  
函数模板····· 250

8.9.6 非类型的模板参数····· 251

8.10 拖尾返回类型····· 252

8.11 函数指针····· 253

8.12 递归····· 256

8.12.1 应用递归····· 259

8.12.2 Quicksort 算法····· 259

8.12.3 main()函数····· 260

8.12.4 extract\_words()函数····· 261

8.12.5 swap()函数····· 262

8.12.6 sort()函数····· 262

8.12.7 max\_word\_length()  
函数····· 263

8.12.8 show\_words()函数····· 264

8.13 本章小结····· 265

8.14 练习····· 266

**第9章 lambda 表达式····· 269**

9.1 lambda 表达式简介····· 269

9.2 定义 lambda 表达式····· 269

9.3 lambda 表达式的命名····· 270

9.4 把 lambda 表达式传递给  
函数····· 272

9.4.1 接受 lambda 表达式  
变元的函数模板····· 272

9.4.2 lambda 变元的函数  
参数类型····· 273

9.4.3 使用 std::function  
模板类型····· 274

9.5 捕获子句····· 277

9.6 在模板中使用 lambda  
表达式····· 279

9.7 lambda 表达式中的递归····· 281

9.8 本章小结····· 283

9.9 练习····· 283

**第10章 程序文件和预处理指令····· 285**

10.1 理解转换单元····· 285

10.1.1 “一个定义”规则····· 286

10.1.2 程序文件和链接····· 286

10.1.3 确定名称的链接属性····· 286

10.1.4 外部名称····· 287

10.1.5 具有外部链接属性的  
const 变量····· 287

10.2 预处理源代码····· 288

10.3 定义预处理标识符····· 289

10.4 包含头文件····· 290

10.5 名称空间····· 292

10.5.1 全局名称空间····· 293

10.5.2 定义名称空间····· 293

10.5.3 应用 using 声明····· 296

10.5.4 函数和名称空间····· 296

10.5.5 未命名的名称空间····· 299

10.5.6 名称空间的别名····· 299

10.5.7 嵌套的名称空间····· 300

10.6 逻辑预处理指令····· 301

10.6.1 逻辑#if 指令····· 301

10.6.2 测试指定标识符的值····· 302

10.6.3 多个代码选择····· 302

10.6.4 标准的预处理宏····· 303

10.7 调试方法····· 304

10.7.1 集成调试器····· 304

10.7.2 调试中的预处理指令····· 305

10.7.3 使用 assert 宏····· 309

10.7.4 关闭断言机制····· 310

10.8 静态断言····· 310

10.9 本章小结····· 312

10.10 练习····· 313

**第11章 定义自己的数据类型····· 315**

11.1 类和面向对象编程····· 315

11.1.1 封装····· 316

11.1.2 继承····· 318



11.1.3	多态性	318	12.1.3	实现重载运算符	366
11.1.4	术语	319	12.1.4	全局运算符函数	369
11.2	定义类	320	12.1.5	提供对运算符的 全部支持	369
11.3	构造函数	322	12.1.6	在类中实现所有的 比较运算符	371
11.3.1	在类的外部定义 构造函数	324	12.2	运算符函数术语	373
11.3.2	默认构造函数的 参数值	326	12.3	默认类成员	374
11.3.3	在构造函数中使用 初始化列表	326	12.3.1	定义析构函数	375
11.3.4	使用 explicit 关键字	327	12.3.2	何时定义副本 构造函数	377
11.3.5	委托构造函数	329	12.3.3	实现赋值运算符	377
11.3.6	默认的副本构造函数	331	12.3.4	实现移动操作	379
11.4	访问私有类成员	332	12.4	重载算术运算符	380
11.5	友元	333	12.4.1	改进输出操作	384
11.5.1	类的友元函数	334	12.4.2	根据一个运算符实现 另一个运算符	386
11.5.2	友元类	336	12.5	重载下标运算符	387
11.6	this 指针	337	12.6	重载类型转换	394
11.7	const 对象和 const 函数成员	338	12.7	重载递增和递减运算符	395
11.8	类的对象数组	340	12.8	函数对象	396
11.9	类对象的大小	342	12.9	本章小结	397
11.10	类的静态成员	342	12.10	练习	398
11.10.1	静态数据成员	342	<b>第 13 章 继承</b>		<b>399</b>
11.10.2	类的静态函数成员	347	13.1	类和面向对象编程	399
11.11	析构函数	347	13.2	类的继承	401
11.12	类对象的指针和引用	350	13.2.1	继承和聚合	401
11.13	将指针作为类的成员	351	13.2.2	派生类	402
11.13.1	定义 Package 类	353	13.3	把类的成员声明为 protected	405
11.13.2	定义 TruckLoad 类	354	13.4	派生类成员的访问级别	405
11.13.3	实现 TruckLoad 类	355	13.4.1	在类层次结构中 使用访问指定符	406
11.14	嵌套类	360	13.4.2	改变继承成员的 访问指定符	408
11.15	本章小结	363	13.5	派生类中的构造 函数操作	408
11.16	练习	363			
<b>第 12 章 运算符重载</b>		<b>365</b>			
12.1	为类实现运算符	365			
12.1.1	运算符重载	366			
12.1.2	可以重载的运算符	366			



13.5.1 派生类中的副本 构造函数..... 412	14.4 通过指针释放对象.....457
13.5.2 派生类中的默认 构造函数..... 414	14.5 本章小结 .....458
13.5.3 继承构造函数..... 414	14.6 练习.....459
13.6 继承中的析构函数..... 415	<b>第 15 章 运行时错误和异常 ..... 461</b>
13.7 重复的成员名..... 417	15.1 处理错误 .....461
13.8 重复的函数成员名..... 418	15.2 理解异常 .....462
13.9 多重继承..... 419	15.2.1 抛出异常 .....463
13.9.1 多个基类..... 419	15.2.2 异常处理过程 .....465
13.9.2 继承成员的模糊性 ..... 420	15.2.3 未处理的异常 .....466
13.9.3 重复的继承 ..... 424	15.2.4 导致抛出异常的代码 .....467
13.9.4 虚基类 ..... 425	15.2.5 嵌套的 try 块.....468
13.10 在相关的类类型之间 转换..... 425	15.3 用类对象作为异常.....472
13.11 本章小结..... 426	15.3.1 匹配 Catch 处理程序和 异常.....473
13.12 练习..... 426	15.3.2 用基类处理程序捕获 派生类异常.....476
<b>第 14 章 多态性.....429</b>	15.3.3 重新抛出异常 .....478
14.1 理解多态性..... 429	15.3.4 捕获所有的异常.....481
14.1.1 使用基类指针..... 429	15.4 抛出异常的函数.....482
14.1.2 调用继承的函数..... 431	15.4.1 函数 try 块.....483
14.1.3 虚函数..... 434	15.4.2 不抛出异常的函数.....483
14.1.4 虚函数中的默认参 数值..... 442	15.4.3 构造函数 try 块.....484
14.1.5 通过智能指针调用 虚函数..... 443	15.4.4 异常和析构函数.....484
14.1.6 通过引用调用虚函数..... 444	15.5 标准库异常.....485
14.1.7 调用虚函数的基类 版本..... 445	15.5.1 异常类的定义 .....486
14.1.8 在指针和类对象之间 转换..... 446	15.5.2 使用标准异常 .....487
14.1.9 动态强制转换..... 447	15.6 本章小结 .....490
14.1.10 转换引用..... 449	15.7 练习.....491
14.1.11 确定多态类型..... 449	<b>第 16 章 类模板 ..... 493</b>
14.2 多态性的成本..... 450	16.1 理解类模板 .....493
14.3 纯虚函数..... 451	16.2 定义类模板 .....494
14.3.1 抽象类..... 452	16.2.1 模板参数 .....495
14.3.2 间接的抽象基类..... 454	16.2.2 简单的类模板 .....495
	16.2.3 定义类模板的函数 成员 .....497
	16.3 创建类模板的实例..... 501
	16.4 类模板的静态成员.....506



16.5	非类型的类模板参数	507	17.3	文件流	540
16.5.1	带有非类型参数的 函数成员的模板	510	17.3.1	在文本模式下写入 文件	540
16.5.2	非类型参数的变元	514	17.3.2	在文本模式下读取 文件	543
16.5.3	把指针和数组用作 非类型参数	514	17.4	设置流打开模式	546
16.6	模板参数的默认值	515	17.5	未格式化的流操作	554
16.7	模板的显式实例化	516	17.5.1	未格式化的流输入 函数	554
16.8	特殊情形	516	17.5.2	未格式化的流输出 函数	557
16.8.1	在类模板中使用 static_assert()	517	17.6	流输入输出中的错误	557
16.8.2	定义类模板特化	518	17.7	二进制模式中的流操作	559
16.8.3	部分模板特化	519	17.8	文件的读写操作	570
16.8.4	从多个部分特化中 选择	519	17.9	字符串流	577
16.9	类模板的友元	520	17.10	对象和流	578
16.10	带有嵌套类的类模板	521	17.10.1	给对象使用插入 运算符	579
16.11	本章小结	528	17.10.2	给对象使用提取 运算符	579
16.12	练习	528	17.10.3	二进制模式中的 对象 I/O	582
<b>第 17 章</b>	<b>文件输入与输出</b>	<b>531</b>	17.10.4	流中更复杂的对象	585
17.1	C++中的输入输出	531	17.11	本章小结	590
17.1.1	理解流	531	17.12	练习	590
17.1.2	使用流的优点	533			
17.2	流类	534			
17.2.1	标准流对象	535			
17.2.2	流的插入和提取操作	535			
17.2.3	流操纵程序	537			



## 第 1 章

## 基本 概念

有时必须在详细解释示例之前使用示例中的元素。本章将概述 C++ 的主要元素及其组合方式,以帮助理解这些元素,还要探讨与计算机中表示数字和字符相关的几个概念。

## 本章主要内容

- 现代 C++ 的含义
- C++ 程序的元素
- 如何注释程序代码
- C++ 代码如何变成可执行程序
- 面向对象的编程方式与过程编程方式的区别
- 二进制、十六进制和八进制数字系统
- Unicode

## 1.1 现代 C++

现代 C++ 使用最新、最好的 C++ 元素编程。这是 C++ 11 标准定义的 C++ 语言,最新标准 C++ 14 对它进行了谨慎的扩展和改进。本书介绍 C++ 14 定义的 C++。

毫无疑问,C++ 是目前世界上使用最广泛、最强大的编程语言。如果打算学习一门编程语言,C++ 就是一个理想的选择。它能在极大范围内的计算设备和环境中高效地开发应用程序:个人电脑、工作站、大型计算机、平板电脑和移动电话。几乎任何程序都可以用 C++ 编写:设备驱动程序、操作系统、薪水管理程序、游戏等。C++ 编译器也是唾手可得的。最新的编译器运行在 PC、工作站和大型机上,常常具备跨编译功能,即可以在一个环境下开发代码,在另一个环境下编译并执行。

C++ 带有一个非常大的标准库,其中包含大量例程和定义,提供了许多程序需要的功能。例如,数值计算、字符串处理、排序和搜索、数据的组织和管理、输入输出等。标准库非常大,本书仅涉及其皮毛。详细描述标准库提供的所有功能,需要好几本书的篇幅。Beginning STL 是使用标准模板库的一个指南,而标准模板库是 C++ 标准库中以各种方式管理和处理数据的一个子集。

就 C++ 语言的范围和库的广度而言,初学者常常觉得 C++ 令人生畏。将 C++ 的全部内容放在一本书里是不可能的,但其实不需要学会 C++ 的所有内容,就可以编写实用的



程序。该语言可以循序渐进地学习,这并不是很难。例如学习开车,没有相关的专业知识和经验,也可以成为非常有竞争力的司机,在印第安纳波利斯安全地开车。通过本书,可以学到使用 C++ 高效编程所需的所有知识。读完本书后,你肯定可以编写自己的应用程序,还可以开始研究 C++ 及其标准库的所有内容。

## 1.2 C++程序概念

本书后面将详细论述本节介绍的所有内容。图 1-1 是一个完全可以工作的完整 C++ 程序,后面将解释该程序的各个部分。这个示例将用作讨论 C++ 一般内容的基础。

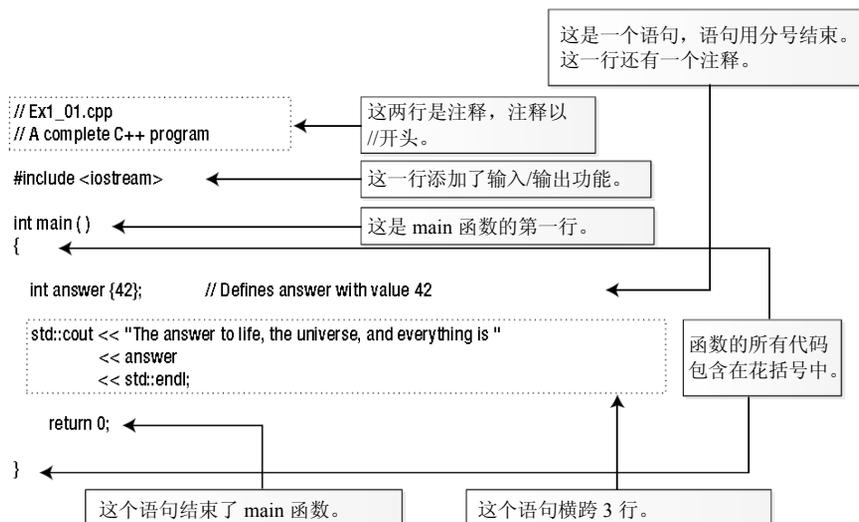


图 1-1 一个完整的 C++ 程序

### 1.2.1 注释和空白

图 1-1 中的前两行是注释。添加注释来解释程序代码,可以使他人更容易理解程序的工作方式。编译器会忽略一行代码中双斜杠后面的所有内容,所以这种注释可以跟在一行代码的后面。第一行注释指出包含代码的文件名。这个文件在本书的下载代码中。本书每个可工作示例的文件都以这种方式给出。文件扩展名 .cpp 表示,这是一个 C++ 源文件。其他扩展名,例如, .cc 也用于表示 C++ 源文件。程序的所有可执行代码放在一个或多个源文件中。

需要把注释放在多行上时,可以使用另一种注释形式。例如:

```
/* This comment is
over two lines. */
```

编译器会忽略 /\* 和 \*/ 之间的所有内容。可以修饰这种注释,使其更加突出。例如:



```
/*  
 * This comment is *  
 * over two lines. *  
 */
```

空白是空格、制表符、换行符、换页符和注释的任意序列。编译器一般会忽略空白，除非由于语法原因需要使用空白把元素区分开来。

## 1.2.2 预处理指令和头文件

图 1-1 的第三行是一个预处理指令。预处理指令会以某种方式修改源代码，之后把它们编译为可执行的形式。这个预处理指令把标准库头文件 `iostream` 的内容添加到这个源文件 `Ex1_01.cpp` 中。头文件的内容插入到 `#include` 指令的位置。

头文件包含源文件中使用的定义。`iostream` 包含使用标准库例程从键盘输入和将文本输出到屏幕上所需的定义。具体而言，它定义了 `std::cout` 和 `std::endl`。每个程序都可以包含一个或多个标准库头文件的内容，也可以创建和使用自己的头文件，以包含本书后面构建的定义。如果在 `Ex1_01.cpp` 中省略了包含 `iostream` 头文件的预处理指令，源文件就不会编译，因为编译器不知道 `std::cout` 和 `std::endl` 是什么。在编译之前，把头文件的内容包含到源文件中。

### 提示：

尖括号和标准头文件名之间没有空格。对于一些编译器而言，尖括号 `<` 和 `>` 之间的空格很重要；如果在这里插入了空格，程序就不会编译。

## 1.2.3 函数

每个 C++ 程序都至少包含一个函数，通常包含许多函数。函数是一个命名的代码块，这些是定义好的操作，例如读取输入的数据，计算平均值，或者输出结果。在程序中使用函数的名称执行或调用函数。程序中的所有可执行代码都放在函数中。程序中必须有一个名为 `main` 的函数，执行总是自动从这个函数开始。`main()` 函数总是调用其他函数，这些函数又可以调用其他函数，以此类推。函数提供了几个重要的优点：

- 程序分解为离散的函数，更容易开发和测试。
- 函数可以在程序的几个不同的地方重用，与在每个需要的地方编写操作代码相比，这会使程序更小。
- 函数常常可以在许多不同的程序中重用，节省了时间和精力。
- 大程序常常由一组程序员共同开发。每个组员负责编写一系列函数，这些函数是整个程序中已定义好的一个子集。没有函数结构，这就是不可能的。

图 1-1 中的程序只包含 `main()` 函数。该函数的第一行是：

```
int main()
```

这称为函数头，标识了函数。其中 `int` 是一个类型名称，它定义了 `main()` 函数执行完



毕时返回的值的类型整数。一般情况下,函数定义中名称后面的圆括号,包含了调用函数时要传递给函数的信息的说明。本实例中的圆括号是空的,但其中可以有内容。第5章将学习如何指定执行函数时传递给函数的信息的类型。文本中总是在函数名的后面加上圆括号,以区分函数与其他代码。函数的可执行代码总是放在花括号中,左花括号跟在函数头的后面。

### 1.2.4 语句

语句是 C++ 程序的基本单元。语句总是用分号结束。分号表示语句的结束,而不是代码行的结束。语句可以定义某个元素,例如计算或者要执行的操作。程序执行的所有操作都是语句指定的。语句按顺序执行,除非有某个语句改变了这个顺序。第4章将学习可以改变执行顺序的语句。图 1-1 的 main() 中有 3 个语句。第一个语句定义了一个变量,变量是一个命名的内存块,用于存储某种数据。在本例中变量的名称是 answer,可以存储整数值:

```
int answer {42}; // Defines answer with the value 42
```

类型 int 放在名称的前面,这指定了可以存储的数据类型——整数。注意 int 和 answer 之间的空格。这里的一个或多个空白字符是必需的,用于分隔类型名称和变量名称。如果没有空格,编译器会把名称看作 intanswer,这是编译器无法理解的。answer 的初始值放在变量名后面的花括号中,所以它最初存储了 42。answer 和 {42} 之间也有一个空格,但这个空格不是必需的。花括号不是名称的一部分,所以无论如何,编译器都可以区分名称和初始值的指定。但是,应以统一的风格使用空白,提高代码的可读性。在第一个语句的末尾有一个多余的注释,解释了上述内容,这还说明,可以在语句中添加注释。// 前面的空白也不是强制的,但最好保留这个空白。

可以把几个语句放在一对花括号 {} 中,此时这些语句就称为语句块。函数体就是一个语句块,如图 1-1 所示,main() 函数体中的语句就放在花括号中。语句块也称为复合语句,因为在许多情况下,语句块可以看作是一个语句,详见第 4 章中的决策功能。在可以放置一个语句的任何地方,都可以放置一个包含在花括号对中的语句块。因此,语句块可以放在其他语句块内部,这个概念称为嵌套。事实上,语句块可以嵌套任意级。

### 1.2.5 数据输入输出

在 C++ 中,输入和输出是使用流来执行的。如果要输出消息,可以把该消息放在输出流中,如果要输入数据,则把它放在输入流中。因此,流是数据源或数据池的一种抽象表示。在程序执行时,每个流都关联着某个设备,关联着数据源的流就是输入流,关联着数据目的地的流就是输出流。对数据源或数据池使用抽象表示的优点是,无论流代表什么设备,编程都是相同的。例如,从磁盘文件中读取数据的方式与从键盘上读取完全相同。在 C++ 中,标准的输出流和输入流称为 cout 和 cin,在默认情况下,它们分别对应计算机的屏幕和键盘。第 2 章将从 cin 中读取输入。



在图 1-1 中, `main()`的下一个语句把文本输出到屏幕:

```
std::cout<< "The answer to life, the universe, and everything is "  
<< answer  
<<std::endl;
```

该语句放在 3 行上, 只是为了说明这么做是可行的。名称 `cout` 和 `endl` 在 `iostream` 头文件中定义。本章后面将解释 `std::`前缀。`<<`是插入操作符, 用于把数据传递到流中。第 2 章会遇到提取操作符`>>`, 它用于从流中读取数据。每个`<<`右边的所有内容都会传递到 `cout` 中。把 `endl` 写入 `std::cout`, 会在流中写入一个换行符, 并刷新输出缓存。刷新输出缓存可确保输出立即显示出来。该语句的结果如下:

```
The answer to life, the universe, and everything is 42
```

可以给每行语句添加注释。例如:

```
std::cout << "The answer to life, the universe, and everything is "  
                                                    // This statement  
<< answer                                     // occupies  
<< std::endl;                                 // three lines
```

双斜杠不必对齐, 但我们常常对齐双斜杠, 使之看起来更整齐, 代码更容易阅读。

## 1.2.6 return 语句

`main()`中的最后一个语句是 `return`。`return` 语句会结束函数, 把控制权返回给调用函数的地方。在本例中它会结束函数, 把控制权返回给操作系统。`return` 语句可能返回一个值或没有返回值。本例的 `return` 语句给操作系统返回 0, 表示程序正常结束。程序可以返回非 0 值, 例如 1、2 等, 表示不同的异常结束条件。Ex1\_01.cpp 中的 `return` 语句是可选的, 可以忽略它。这是因为如果程序执行超过了 `main()`的最后一个语句, 就等价于执行 `return 0`。

## 1.2.7 名称空间

大项目会同时涉及几个程序员。这可能会带来名称问题。不同的程序员可能给不同的元素使用相同的名称, 这可能会带来一些混乱, 使程序出错。标准库定义了许多名称, 很难全部记住。不小心使用了标准库名称也会出问题。名称空间就是用于解决这个问题的。

名称空间类似于姓氏, 它置于该名称空间中声明的所有名称前面。标准库中的名称都在 `std` 名称空间中定义, `cout` 和 `endl` 是标准库中的名称, 所以其全名是 `std::cout` 和 `std::endl`。其中的两个冒号有一个非常奇特的名称: 作用域解析操作符, 详见后面的说明。这里它用于分隔名称空间的名称 `std` 和标准库中的名称, 例如 `cout` 和 `endl`。标准库中的几乎所有名称都有前缀 `std`。

名称空间的代码如下所示:



```
namespace ih_space {  
  
    // All names declared in here need to be prefixed  
    // with ih_space when they are reference from outside.  
    // For example, a min() function defined in here  
    // would be referred to outside this namespace as ih_space::min()  
}
```

花括号对中的所有内容都位于 `ih_space` 名称空间中。

#### 警告:

`main()`函数不能定义在名称空间中,未在名称空间中定义的内容都存在于全局名称空间中,全局名称空间没有名称。

### 1.2.8 名称和关键字

`Ex1_01.cpp` 包含变量 `answer` 的定义,并使用在 `iostream` 标准库头文件中定义的名称 `cout` 和 `endl`。程序中的许多元素都需要名称,定义名称的规则如下:

- 名称可以是包含大小写拉丁字母 `A~Z` 和 `a~z`、数字 `0~9` 和下划线 `_` 的任意序列。
- 名称必须以字母或下划线开头。
- 名称是区分大小写的。

名称可以用下划线开头,但最好不要使用这样的名称,它们可能与 C++ 标准库中的名称冲突,因为 C++ 标准库以这种方式定义了大量的名称。C++ 标准还允许名称有任意长度,但有的编译器对此有某种长度限制,这个限制常常比较宽,不是一个严格的限制。大多数情况下,不需要使用长度超过 12~15 个字符的名称。

下面是一些有效的 C++ 名称:

```
toe_count shoeSize Box democrat Democrat number1 x2 y2 pValue out_of_range
```

大小写字母是有区别的,所以 `democrat` 和 `Democrat` 是不同的名称。编写由两个或多个单词组成的名称时,有几个约定;可以把第二个及以后各个单词的首字母大写,或者用下划线分隔它们。

关键字是 C++ 中有特殊含义的保留字,不能把它们用于其他目的。例如, `class`、`double`、`throw` 和 `catch` 都是保留字。

## 1.3 类和对象

类是定义数据类型的代码块。类的名称就是类型的名称。类类型的数据项称为对象。创建变量,以存储自定义数据类型的对象时,就要使用类类型名称。定义自己的数据类型,就可以根据具体的问题提出解决方案。例如,如果编写一个处理学生信息的程序,就可以定义 `Student` 类型。`Student` 类型可以包含学生的所有特征,例如年龄、性别或学



校记录——这些都是程序需要的。

## 1.4 模板

有时程序需要几个类似的类或函数，其代码中只有所处理的数据类型有区别。编译器可以使用模板给特定的自定义类型自动生成类或函数的代码。编译器使用类模板会生成一个或多个类系列，使用函数模板会生成函数。每个模板都有名称，希望编译器创建模板的实例时，就会使用该名称。标准库大量使用了模板。

## 1.5 程序文件

C++代码存储在两种文件中。源文件包含函数，因此包含程序中的所有可执行代码。源文件的名称通常使用扩展名.cpp，但也使用其他扩展名，例如.cc。头文件包含元素的定义，例如.cpp文件中的可执行代码使用的类和模板。头文件的名称通常使用.h扩展名，但也使用其他扩展名，例如.hpp。当然，实际的程序一般包含其他类型的文件，这些文件包含的内容与C++无关，例如定义图形用户界面(GUI)的资源。

## 1.6 标准库

如果每次编写程序时，都从头开始创建所有内容，就是一件非常枯燥的工作。许多程序都需要相同的功能，例如从键盘上读取数据，计算平方根，将数据记录按特定的顺序排列。C++附带大量预先编写好的代码，提供了所有这些功能，因此不需要自己编写它们。这些标准代码都在标准库中定义。标准库的一个子集是标准模板库(STL)，它包含大量的类模板，用于创建类型，以组织和管理数据。标准库还包含许多函数模板，用于执行各种操作，例如排序和搜索数据集合，进行数值处理等。本书将学习 STL 的几个功能，但全面讨论 STL 需要一整本书的篇幅。Beginning STL 是本书的后续产品，专门介绍 STL。

## 1.7 代码的表示样式

代码排列的方式对代码的可读性有非常重要的影响。这两种基本的方式。首先，可以使用制表符和/或空格缩进程序语句，显示出这些语句的逻辑；再以一致的方式使用定义程序块的匹配花括号，使块之间的关系更清晰。其次，可以把一个语句放在两行或多行上，提高程序的可读性。安排匹配花括号和缩进语句的约定称为表示样式。

代码有许多不同的表示样式。表 1-1 显示了三种常用的代码表示样式。



表 1-1 三种常用的代码表示样式

样式 1	样式 2	样式 3
<pre>namespace mine {     bool has_factor(int x,                     int y)     {         int f{ hcf(x, y) };         if (f &gt; 1)         {             return true;         }         else         {             return false;         }     } }</pre>	<pre>namespace mine{     bool has_factor(int x,                     int y)     {         int f{ hcf(x, y) };         if (f &gt; 1) {             return true;         }         else {             return false;         }     } }</pre>	<pre>namespace mine{     bool has_factor(int x,                     int y) {         int f{ hcf(x, y) };         if (f &gt; 1){             return true;         }         else{             return false;         }     } }</pre>

本书的示例使用样式 1。

## 1.8 创建可执行文件

从 C++源代码中创建可执行的模块需要两步骤。第一步是编译器把每个.cpp 文件转换为对象文件，其中包含了与源文件内容对应的机器码。第二步是链接程序把程序的对象文件合并到包含完整可执行程序的文件中。在这个过程中，链接程序会集成程序使用的标准库函数。

图 1-2 表明，3 个源文件经过编译后，生成 3 个对应的对象文件。用于标识对象文件的文件扩展名在不同的机器环境上是不同的，这里没有显示。组成程序的源文件可以在不同的编译器运行期间单独编译，但大多数编译器都允许在一次运行期间编译它们。无论采用哪种方式，编译器都把每个源文件看作一个独立的实体，为每个.cpp 文件生成一个对象文件。然后在链接步骤中，把程序的对象文件和必要的库函数组合到一个可执行文件中。

实际上，编译是一个迭代的过程，因为在源代码中总是会有输入错误或其他错误。更正了每个源文件中的这些错误后，就可以进入链接步骤，但在这一步可能会发现有更多的错误！即使链接步骤生成了可执行模块，程序仍有可能包含逻辑错误，即程序没有生成希望的结果。为了更正这些错误，必须回过头来修改源代码，再编译。这个过程会继续下去，直到程序按照希望的那样执行为止。如果程序的执行结果不像我们宣称的那样，其他人就有可能找到我们本应发现的许多错误，这是毋庸置疑的。一般说来，如果程序非常大，就总是包含错误。

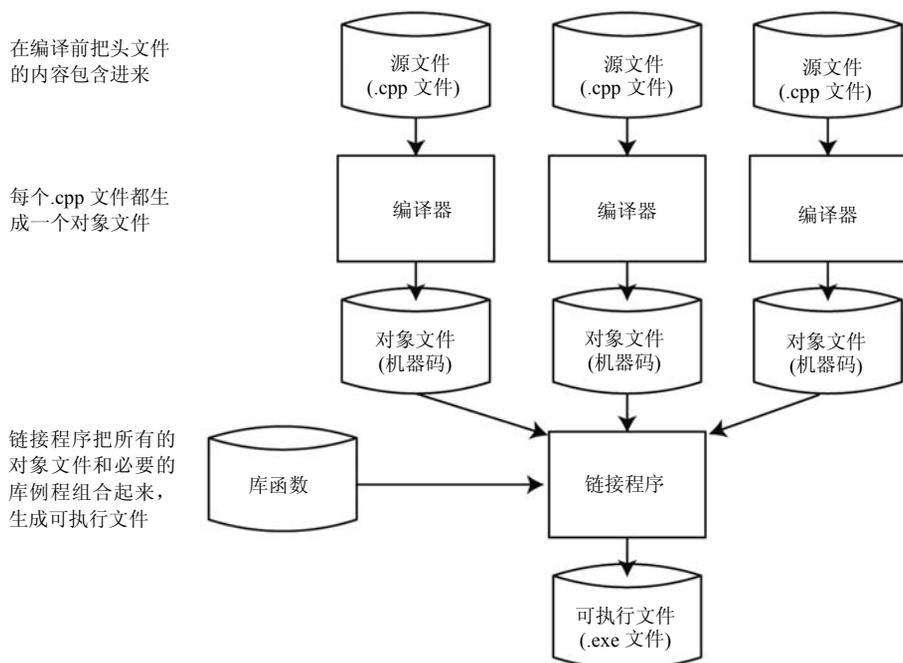


图 1-2 编译和链接过程

## 1.9 表示数字

在 C++ 程序中,数字的表示有许多方式,所以必须理解表示数字的各种可能性。如果很熟悉二进制数、十六进制数和浮点数的表示,就可以跳过本节。

### 1.9.1 二进制数

首先考虑一下常见的十进制数(如 324 或 911)表示什么。显然,324 是表示三百二十四,911 表示九百一十一。这是“三百”加上“二十”再加上“四”的简写形式,或者是“九百”加上“一十”再加上“一”的简写形式。更明确地说,这两个数表示:

$$324 \text{ 是: } 3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0, \text{ 也就是 } 3 \times 10 \times 10 + 2 \times 10 + 4$$
$$911 \text{ 是: } 9 \times 10^2 + 1 \times 10^1 + 1 \times 10^0, \text{ 也就是 } 9 \times 10 \times 10 + 1 \times 10 + 1$$

这称为十进制表示法,因为这是建立在 10 的幂的基础之上。也可以说,这里的数字以 10 为基底来表示,因为每个数位都是 10 的幂。以这种方式表示数值非常方便,因为人有 10 根手指或 10 根脚趾或者 10 个任何类型的附属物。但是,这对 PC 就不太方便了,因为 PC 主要以开关为基础,即开和关,加起来只有 2,而不是 10。这就是计算机用基数 2 而不是用基数 10 来表示数值的主要原因。用基数 2 来表示数值称为二进制计数系统。用基数 10 表示数字,数字可以是 0~9。一般情况下,以任意 n 为基底来表示的数,每



个数位的数字是从 0 到 n-1。

二进制数字只能是 0 或 1，二进制数 1101 就可以分解为：

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0, \text{ 也就是 } 1 \times 2 \times 2 \times 2 + 1 \times 2 \times 2 + 0 \times 2 + 1$$

计算得 13(十进制系统)。在表 1-2 中，列出了用 8 个二进制数字表示的对应的十进制值(二进制数字常常称为位)。

表 1-2 与 8 位二进制值对应的十进制值

二 进 制	十 进 制	二 进 制	十 进 制
0000 0000	0	1000 0000	128
0000 0001	1	1000 0001	129
0000 0010	2	1000 0010	130
...	...	...	...
0001 0000	16	1001 0000	144
0001 0001	17	1001 0001	145
...	...	...	...
0111 1100	124	1111 1100	252
0111 1101	125	1111 1101	253
0111 1110	126	1111 1110	254
0111 1111	127	1111 1111	255

使用前 7 位可以表示从 0~127 的数，一共 128 个不同的数，使用全部 8 位可以表示 256(即 2<sup>8</sup>)个数。一般情况下，如果有 n 位，就可以表示 2<sup>n</sup> 个整数，其正值从 0 到 2<sup>n</sup>-1。

在计算机中，二进制数相加是非常容易的，因为对应数字加起来的进位只能是 0 或 1，所以处理过程会非常简单。图 1-3 中的例子演示了两个 8 位二进制数相加的过程。

二进制	十进制
0001 1101	29
+ 0010 1011	+ 43
-----	-----
0100 1000	72
-----	-----
↑↑↑↑	
进位	

图 1-3 二进制数的相加

相加操作从最右边开始，将操作数中对应的位相加。图 1-3 指出，前 6 位都向左边的下一位进 1，这是因为每个数字只能是 0 或 1。计算 1+1 时，结果不能存储在当前的位中，而需要在左边的下一位中加 1。



## 1.9.2 十六进制数

在处理很大的二进制数时，就会有一个小问题。如：

1111 0101 1011 1001 1110 0001

在实际应用中，二进制表示法显得比较烦琐，如果把这个二进制数表示为十进制数，结果为 16,103,905，这个 8 位的十进制数没有什么价值。还可以用更长的二进制位数表示更大的十进制数。显然，我们需要一种更高效的方式来表示这个数，但十进制并不总是合适的。有时需要能够指定从右开始算起的第 10 位和第 24 位的数字为 1。用十进制整数来完成这个任务是非常麻烦的，而且很容易出现计算错误。比较简单的解决方案是使用十六进制表示法，即数字以 16 为基数表示。

基数为 16 的算术就方便得多，它与二进制也相得益彰。每个十六进制的数字可以是 0~15 的值(从 10 到 15 的数字用 A 到 F 或 a 到 f 表示，如表 1-3 所示)，从 0~15 的数值就分别对应于用 4 个二进制数字表示的值。

表 1-3 十六进制数表示为十进制数和二进制数

十六进制	十进制	二进制
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A 或 a	10	1010
B 或 b	11	1011
C 或 c	12	1100
D 或 d	13	1101
E 或 e	14	1110
F 或 f	15	1111

因为一个十六进制数对应于 4 个二进制数，所以可以把较大的二进制数表示为一个十六进制数，方法是从右开始，把每 4 个二进制数字组成一组，再用对应的十六进制数表示每个组。例如，二进制数：



1111 0101 1011 1001 1110 0001

如果依次提取每 4 个二进制数字,用对应的十六进制数表示每个组,这个数字用十六进制表示,就得到:

F 5 B 9 E 1

所得的 6 个十六进制数字分别对应于 6 组 4 个二进制数字。为了证明这适用于所有的情况,下面用十进制表示法把这个数值直接从十六进制转换为十进制。

这个十六进制数的计算如下。F5B9E1 转换为十进制值:

$$15 \times 16^5 + 5 \times 16^4 + 11 \times 16^3 + 9 \times 16^2 + 14 \times 16^1 + 1 \times 16^0$$

计算得:

$$15,728,640 + 327,680 + 45,056 + 2,304 + 224 + 1$$

最后相加的结果与把二进制数转换为十进制数的结果相同:16,103,905。在 C++ 中,十六进制数要加上前缀 0x 或 0X,所以在代码中,这个值写作 0xF5B9E1。显然,这表示,99 与 0x99 不相等。

十六进制数的另一个非常方便的特点是,现代计算机把整数存储在偶数字节的字中,一般是 2、4、8 或 16 字节,一个字节是 8 位,正好是两个十六进制数字,所以内存中的任意二进制整数总是精确对应于若干个十六进制数字。

### 1.9.3 负的二进制数

二进制算术需要理解的另一个方面是负数。前面一直假定所有的数字都是正的。从乐观的角度来看是这样,所以我们目前已对二进制数有了一半的认识。但在实际中还会遇到负数,从悲观的角度来看,我们对二进制数的认识仅仅是一半。在计算机中,是如何表示负数的?我们只能按照自己的意愿来处理二进制数字,所以解决方案必须是使用其中的一个二进制数字。

对于允许是负数的数值(称为带符号的数值),必须先确定一个固定的长度(换言之,就是二进制数字的位数),再把最左边的二进制数字设置为符号位。必须固定位数,这样才能避免符号位与其他位的混淆。

因为计算机的内存由 8 位字节组成,所以二进制数字要存储在多个 8 位中(通常是 2 的幂),即有些数字是 8 位,有些数字是 16 位,有些数字是 32 位等。只要知道每个数值的位数,就可以找到符号位,它应是最左边的那一位。如果符号位是 0,该数值就是正的,如果它是 1,该数值就是负的。

似乎这就解决了问题,但实际上并非如此。每个数字都是由一个符号位和给定的位数组成,其中符号位为 0 表示正数,符号位为 1 表示负数,给定的位数指定数字的绝对值,换言之,就是无符号的数字。把 +6 改为 -6 只需要把符号位从 0 改为 1。但是,这个表示方法需要大量的系统开销,而且系统开销的大小取决于对这个数字表示方法执行的算术运算的复杂程度。因此,大多数计算机都采用了另一种方法。



理想情况下,当两个整数相加时,计算机不应不检查两个数字是否为负。我们希望使用常规的“加”来生成相应的结果,而不考虑操作数的符号。相加操作把相应的二进制数字合并在一起,得到相应的位,在必要时把1进到下一位上。如果把二进制的-8加到+12上,答案就应是+4。如果把+3加到+8上,也采用相同的方式操作。

如果用简化的解决方案来执行这一操作,也就是把正数的符号位设置为1,使它变成负数,再执行算术运算,并进行常规的进位,答案就是错误的:

12 转换为二进制:                    0000 1100

-8 转换为二进制:                    1000 1000

如果把它们加起来,结果是:    1001 0100。

答案是-20,这可不是我们希望的结果+4,它的二进制应是0000 0100。此时读者会认为,“没有把符号作为另一个位”。但这里就是这么做的。

下面看看计算机如何表示-8,即从+4中减去+12,得到正确的结果:

+4 转换为二进制:                    0000 0100

+12 转换为二进制:                   0000 1100

从+4中减去+12,结果是:           1111 1000。

对于右边的4位数字,必须借1,才能进行减法,这正是我们在执行十进制算术时所进行的操作。结果就是-8,即使它看起来不是-8,但其值的确是-8。再用二进制把该值与+12或+15相加,就会得到正确的结果。当然,如果想得到-8,总是可以从0中减去+8。

在从4中减去12或从0中减去+8时,究竟进行了什么操作?实际上是对负二进制数值采用了2的补码形式。通过一个可以在头脑中进行的简单过程就可以从任何正的二进制数中获得这种形式。这里需要做一个约定,以避免解释它为什么有效。下面看看如何从正数中构建负数的2的补码形式,读者也可以自己证明这是有效的。现在回到前面的例子,给-8构建2的补码形式。

首先把+8转换为二进制:

0000 1000

现在反转每个二进制数字,即把0变成1,把1变成0:

1111 0111

这称为1的补码形式。如果给这个数加上1,就得到了2的补码形式:

1111 1000

这就是从+4中减去+12,得到的-8的二进制表示。为了确保正确,下面对-8和+12进行正常的相加操作:

+12 转换为二进制                    0000 1100

-8 转换为二进制                    1111 1000

把这两个数加在一起,得到:    0000 0100

答案就是4。这是正确的。左边所有的1都向前进位,这样该位的数字就是0。最左



边的数字应进位到第 9 位, 即第 9 位应是 1, 但这里不必担心这个第 9 位, 因为在前面计算-8 时从前面借了一位, 在此正好抵消。实际上, 这里做了一个假定, 符号位 1 或 0 永远放在最左边。读者可以自己试验几个例子, 就会发现这种方法总是有效的。最妙的是, 使用负数的 2 的补码形式使计算机上的算术计算非常简单快速。

### 1.9.4 八进制数

八进制整数是用以 8 为基底来表示的数。八进制的数字是 0 到 7。目前八进制数很少使用。计算机内存用 36 位字来衡量时, 八进制数很有用, 因为可以把 36 位的二进制值指定为 12 个八进制数字。这是很久以前的情形了, 为什么还要介绍八进制? 因为它可能会引起潜在的冲突。在 C++中仍可以编写八进制的常量。八进制值有一个前导 0, 所以 76 是十进制值, 076 是八进制值, 它对应十进制的 64。下面是一条黄金规则:

**注意:**

不要在十进制整数的前面加上前导 0, 否则会得到另一个值, 或者得到编译器的一个错误消息。

### 1.9.5 Big-Endian 和 Little-Endian 系统

整数在内存的一系列连续字节中存储为二进制值, 通常存储为 2、4、8 或 16 个字节。字节采用什么顺序是非常重要的。

把十进制数 262657 存储为 4 字节二进制值。选择这个值是因为它的二进制值是

```
0000 0000 0000 0100 0000 0010 0000 0001
```

每个字节的位模式都很容易与其他字节区分开来。如果使用 Inter 处理器的 PC, 该数字就存储为:

字节地址:	00	01	02	03
数据位:	0000 0001	0000 0010	0000 0100	0000 0000

可以看出, 值中最重要的 8 位是都为 0 的那些位, 它们都存储在地址最高的字节中, 换言之, 就是最右边的字节。最不重要的 8 位存储在地址最低的字节中, 即最左边的字节中。这种安排形式称为 Little-Endian。

如果使用基于 Motorola 处理器的机器, 该数值在内存中存储为:

字节地址:	00	01	02	03
数据位:	0000 0000	0000 0100	0000 0010	0000 0001

字节现在的顺序是反的, 最重要的 8 位存储在最左边的字节中, 即地址最低的字节中。这种安排形式称为 Big-Endian。最新的一些处理器, 例如 SPARC 和 Power-PC 处理器是双向优先的, 这表示数据的字节顺序可以在 Big-Endian 和 Little-Endian 之间切换。

**注意:**

无论字节顺序是 Big-Endian 还是 Little-Endian, 在每个字节中, 最重要的位都放在



左边, 最不重要的位都放在右边。

这非常有趣, 但为什么这很重要? 在大多数情况下, 这并不重要。即使不知道执行代码的计算机是采用 Big-Endian 还是 Little-Endian, 都可以编写出有效的 C++ 程序。但是, 在处理来自另一台机器的二进制数据值, 这就很重要了。二进制数据会写入文件或通过网络传送为一系列字节, 此时必须解释它们。如果数据源所在的机器使用的 Endian 形式与运行代码的机器不同, 就必须反转每个二进制值的字节顺序, 否则就会出错。

对于那些了解一些有趣背景信息的读者来说, 大概知道术语 Big-Endian 和 Little-Endian 取自 Jonathan Swift 撰著的《格利佛游记》。Lilliput 的国王命令所有的国民必须在鸡蛋的小端磕破鸡蛋。这是因为国王的儿子按照在鸡蛋的大端磕破鸡蛋的传统方式玩耍时, 划破了自己的手指。守法的普通 Lilliput 国民称为 Little-Endian, Lilliputian 王国中一些坚持在鸡蛋的大端磕破鸡蛋的反传统主义者就是 Big-Endian。许多人因此被判死刑。

### 1.9.6 浮点数

我们常常要处理非常大的数——例如, 宇宙中的质子数, 它需要 79 位十进制数字。显然在许多情况下, 要处理的数都不仅仅是 4 字节二进制数能表示的 10 位十进制数字。同样, 还有许多非常小的数, 例如, 汽车销售人员接受你对 2001 本田汽车(它仅行驶了 480 000 英里)的报价所需的分钟数。处理这两种数值的机制就称为浮点数。

在十进制记数法中, 数值的浮点表示是带有两个部分的小数值。其中一部分称为尾数, 它大于等于 0.9, 小于 1.0, 且有固定的位数。另一部分称为指数, 浮点数的值是尾数乘以 10 的指数幂。演示这种表示法要比描述它更容易, 所以下面看一些例子。在普通的十进制计数法中, 365 写为浮点数的形式为:

0.3650000E03

E 表示“指数”, 尾数部分 0.3650000 乘以 10 的幂, 就得到需要的值。即:

$0.3650000 \times 10 \times 10 \times 10$

这显然是 365。

这里的尾数有 7 位小数。浮点数中的位数精度取决于给它分配的内存。4 字节的单精度浮点数一般能提供约 7 位小数的精度。这里说“大约”, 是因为在计算机中, 这些数使用二进制的浮点数形式, 23 位的二进制小数不会精确地对应于有 7 位小数位数的十进制小数。双精度浮点数一般对应于约 15 位小数的精度。

下面看一个小数值:

0.3650000E-04

它计算为  $.365 \times 10^{-4}$ , 即 0.000365。

假定有一个数 2 134 311 179, 表示为单精度浮点数:



```
0.2134311E10
```

它们不完全相同,因为丢弃了3个低位数字,初始值就表示为2 134 311 000。这是处理如此大范围的数所付出的代价,这个范围是 $10^{-38}$ 到 $10^{+38}$ (正数和负数)。它们称为浮点数,其原因非常明显:小数点是浮动的,其位置取决于指数值。

除了由于保证精度而带来的固定精度限制之外,还有一个方面要注意。对不同量级的数执行相加或相减操作时要特别小心。这个问题可以用一个简单的例子来说明。把.365E-3和.365E+7加起来,写成十进制的形式:

```
0.000365 + 3,650,000.0
```

结果如下:

```
3,650,000.000365
```

转换为带7位小数的浮点数时,得到:

```
0.3650000E+7
```

把.365E-3和.365E+7加起来,似乎没有什么效果,所以不必担心它。但问题在于结果只有6位或7位小数精度。大数的所有位数都不会受到小数的影响,因为大数的所有有效位数都离小数的有效位数太远了。可笑的是,当两个数几乎相等时,也必须特别小心。如果计算这两个数之差,结果可能只有一两位小数精度。在这种情况下,很容易计算两个完全是垃圾的值。浮点数可以执行没有它们就不可能执行的计算,但要确保结果有效,就必须记住它们的限制。这意味着要考虑要处理的值域及其相对值。

## 1.10 表示字符

计算机中的数据没有内在的含义。机器码指令只是数字,当然数值就是数值,字符也是数值。每个字符都获得了一个独特的整数值,称为代码或代码点。值42可以是6周的天数、生活、宇宙和其他事务的答案,也可以是星号字符。这取决于如何解释它。在C++中,单个字符可以放在单引号中,例如'a'、'?'、'\*',编译器会给它们生成代码值。

### 1.10.1 ASCII 码

20世纪60年代,人们定义了美国信息交换标准码(ASCII),来表示字符。这是一个7位代码,所以共有128个不同的代码值。ASCII值0到31表示各种非打印控制符,例如回车符(代码15)和换页符(代码12)。代码值65到90是大写字母A到Z,代码值141到172对应小写字母a到z。如果查看字母的代码值的对应二进制值,就会发现大小写字母的代码仅在第6位上有区别:小写字母的第6位是0,大写字母的第6位是1。其他代码表示数字0到9、标点符号和其他字符。这适合于美国人或英国人,但法国人或德国人在文本中需要重音和元音变音,但它们没有包含在7位ASCII中。



为了克服 7 位代码的局限,人们定义了 ASCII 的扩展版本,它有 8 位代码。代码值 0 到 127 与 7 位 ASCII 版本表示相同的字符,代码值 128 到 255 是可变的。8 位 ASCII 的一个变体称为 Latin-1,它提供了大多数欧洲语言的字符,以及俄语等语言中的字符。当然,对于韩国人、日本人或阿拉伯人而言,8 位 ASCII 肯定是不够的。为了克服扩展 ASCII 的局限,20 世纪 90 年代出现了通用字符集(Universal Character Set, UCS),UCS 由标准 ISO 10646 定义,其代码有 32 位,提供了超过 20 亿个不同的代码值。

### 1.10.2 UCS 和 Unicode

UCS 定义了字符和整数代码值(称为代码点)之间的映射。代码点与编码是不同的,认识到这一点很重要:代码点是一个整数,编码指定了把给定的代码点表示为一系列字节或字的方式。小于 256 的代码值非常常见,可以用 1 个字节表示。使用 4 字节存储只需要 1 字节的代码值,仅是因为有其他代码需要多个字节,是非常低效的。编码是表示代码点、允许更高效地存储它们的方式。

Unicode 是一个标准,定义了一组字符及其代码点(与 UCS 相同),代码点值从 0 到 0x10ffff。Unicode 还为这些代码点定义了几个不同的编码,包括其他机制,例如处理从右向左阅读的语言(如阿拉伯语),代码点的范围足以包含世界上所有语言的字符集,还包含许多其他的图形化字符,例如数学符号。代码分为 17 个代码段,每段包含 65 536 个代码值。0 段包含 0 到 0xffff 的十六进制代码值,1 段包含 0x10000 到 0x1ffff 的代码,2 段包含 0x20000 到 0x2ffff 的代码,以此类推,17 段包含 0x100000 到 0x10ffff 的代码。大多数国家语言的字符代码包含在 0 段中,其代码值为 0 到 0xffff。因此,大多数语言的字符串可以表示为 16 位代码的一个序列。

Unicode 中一个可能混淆的方面是:它提供了多个字符编码方法。最常用的编码是 UTF-8 和 UTF-16,它们不能表示 Unicode 集合中的所有字符。UTF-8 和 UTF-16 之间的区别是如何表示给定字符的代码点,给定字符的代码数值在这两种表示法中是相同的。下面是这些编码表示字符的方式:

- UTF-8 把字符表示为长度在 1 字节和 4 字节之间变化的序列。ASCII 字符集在 UTF-8 中表示为单字节代码,其代码值与 ASCII 相同。大多数网页都使用 UTF-8 编码文本,0 代码段包含了 UTF-8 中的单字节和双字节代码。
- UTF-16 把字符表示为一个或两个 16 位值。UTF-16 包括了 UTF-8。因为一个 16 位值包含 0 代码段中的所有代码,所以 UTF-16 覆盖了多语言编程环境中的大多数情形。

存储 Unicode 字符有 3 个整数类型: `wchar_t`、`char16_t` 和 `char32_t`。详见第 2 章。

## 1.11 C++ 源字符

编写 C++ 语句要使用基本源字符集,这些是在 C++ 源文件中可以显式使用的字符集。用于定义名称的字符集是上述字符集的一个子集。当然,基本源字符集并没有限制代



码中使用的字符数据。程序可以用各种方式创建没有包含在该字符集中的字符串，基本源字符集包括下述字符：

- 大小写字母 A~Z 和 a~z
- 数字 0~9
- 控制字符，如换行符、水平和垂直制表符、换页符
- 字符 `_{}[]#()<>%:;.*+~/^&|~!=",`

这很简单、直观。一共可以使用 96 个字符，这些字符可以满足大多数要求。大多数情况下，基本源字符集足够用了，但偶尔需要使用不包含在基本源集合中的字符。可以在名称中包含 Unicode 字符。Unicode 字符指定为码点的十六进制表示 `\udddd` 或 `\Uddddddd`(其中 d 是一个十六进制数)。注意第一种形式使用小写 u，第二种形式使用大写 U，两者都是可接受的。但是，不能以这种方式指定基本源字符集中的字符。另外，名称中的字符不能是控制字符。字符和字符串数据都可以包含 Unicode 字符。

### 1.11.1 三字符序列

三字符序列不太常见，但 C++ 标准允许把某些字符指定为三字符序列。三字符序列就是用于标识另一个字符的三个字符序列。以前为了表示 C 语言需要的、7 位 ASCII 没有的字符，这是必不可少的一种方法。表 1-4 列出了以这种方式在 C++ 中指定的字符。

表 1-4 三字符序列的字符

字 符	三字符序列
#	??=
[	??(
]	??)
\	??/
{	??<
}	??>
^	??'
	??!
~	??-

为了兼容，C++ 仍支持这些字符。编译器会用对应的字符替代所有三字符序列，再对源代码进行其他处理。有几个实例需要注意，尤其是使用正则表达式功能(本书未介绍)时。这是因为出现在正则表达式中的序列可能对应于三字符序列，但这不是我们希望的。在大多数情况下，可以忽略三字符序列，尽管有可能不小心指定它们。

### 1.11.2 转义序列

在程序中使用字符常量时，例如单个字符或字符串，某些字符是会出问题的。显然，



不能直接把 `newline` 或 `tab` 这样的字符输入为字符常量,因为它们只完成自己该做的工作:转到新的一行或移动到源代码文件的下一个制表符位置。通过转义序列可以把控制字符输入为字符常量。转义序列是指定字符的一种间接方式,通常以一个反斜杠`\`开头。表示控制字符的转义序列如表 1-5 所示。

表 1-5 表示控制字符的转义序列

转义序列	控制字符
<code>\n</code>	换行符
<code>\t</code>	水平制表符
<code>\v</code>	垂直制表符
<code>\b</code>	退格字符
<code>\r</code>	回车字符
<code>\f</code>	换页字符
<code>\a</code>	警告字符

还有其他一些字符在直接表示时会出问题。显然,表示反斜杠字符本身是很困难的,因为它表示转义序列的开头。用作界定符的单引号和双引号,例如常量`'A'`或字符串`"text"`也有问题。表 1-6 列出了这些转义序列。

表 1-6 用转义序列指定的“问题”字符

转义序列	字符
<code>\\</code>	反斜杠
<code>\'</code>	单引号
<code>\"</code>	双引号
<code>\?</code>	问号

由于反斜杠表示转义序列的开始,因此把反斜杠字符输入为字符常量的唯一方式是使用两个连续的反斜杠`\\`。

下面的程序示例使用转义序列输出要显示在屏幕上的消息。要查看该程序的结果,需要输入、编译、链接和执行下面的程序。

```
// Ex1_02.cpp
// Using escape sequences
#include <iostream>

int main()
{
    std::cout << "\"Least \'said\' \\n\t\tsoonest \'mended\'.\\" <<
std::endl;
}
```



在编译、链接和运行这个程序时,会显示如下结果:

```
"Least 'said' \
                soonest 'mended'."
```

所得的输出由语句中双引号之间的内容确定:

```
std::cout << "\"Least \'said\' \\n\\t\\tsoonest \'mended\'\\.\"" <<
std::endl;
```

原则上,上述语句中外部双引号之间的所有内容都会发送给 `cout`。双引号之间的字符串称为字符串字面量。双引号字符表示该字符串字面量的开始和结束;它们不是字符串的一部分。字符串字面量中的转义序列会被编译器转换为它表示的字符,所以该字符会发送给 `cout`,而不是发送转义序列。字符串字面量中的反斜杠总是表示转义序列的开始,所以发送给 `cout` 的第一个字符是双引号。

接着输出的是 `Least` 后跟一个空格,接着是一个单引号、`said` 和另一个单引号。然后是一个空格和由 `\\` 指定的反斜杠。接着把对应于 `\n` 的换行符写入流,使光标移动到下一行的开头。然后用 `\\t` 给 `cout` 发送两个制表符,让光标向右移动两个制表位置。之后显示字符串 `soonest`、一个空格和单引号中的 `mended`,最后是句号和一个双引号。

## 1.12 过程化编程方法和面向对象编程方法

在历史上,过程化编程方法曾是编写几乎所有程序的方式。要创建问题的过程化编程解决方案,必须考虑程序实现的过程,才能解决问题。一旦需求明确地确定下来,就可以写出完成任务的大致提纲,如下所示:

- 为程序要实现的整个过程创建一个清晰的高级定义。
- 将整个过程分为可工作的计算单元,这些计算单元应尽可能是自包含的。它们常常对应于函数。
- 把每个计算单元的逻辑和工作分解为更详细的动作序列。这就下降到对应于编程语言语句的一级。
- 根据正处理的数据的基本类型:数值数据、单个字符和字符串,来编写函数。

在解决相同问题时,除了开始时对问题进行清晰的说明这一点相同之外,面向对象的编程方法在其他地方都完全不同:

- 根据问题的详细说明确定该问题所涉及的对象类型。例如,如果程序处理的是篮球运动员,就应把 `BaseballPlayer` 标识为程序要处理的一个数据类型。如果程序是一个账户打包程序,就应定义 `Account` 类型和 `Transaction` 类型的对象。还要确定程序需要对每种对象执行的操作集。这将产生一组与应用程序相关的数据类型,用于编写程序。
- 为问题需要的每种新数据类型生成一个详细的设计方案,包括可以对每种对象类型执行的操作。



- 根据已定义的新数据类型及其允许的操作，编写程序的逻辑。

面向对象解决方案的程序代码完全不同于过程化解决方案，理解起来也比较容易，维护也方便得多。面向对象解决方案所需的设计时间要比过程化解决方案长一些。但是，面向对象程序的编码和测试阶段比较短，问题也比较少，所以这两种方式的整个开发时间大致相同。

下面简要论述面向对象方式。假定要实现一个处理各种盒子的程序。这个程序的一个合理要求是把几个小盒子装到另一个大一些盒子中。在过程化程序中，需要在一组变量中存储每个盒子的长度、宽度和高度。包含几个盒子的新盒子尺寸必须根据每个被包含盒子的尺寸，按照为打包一组盒子而定义的规则进行计算。

面向对象的解决方案首先需要定义一个 `Box` 数据类型，这样就可以创建变量，引用 `Box` 类型的对象，并创建 `Box` 对象。然后定义一个操作，把两个 `Box` 对象加在一起，生成包含前两个 `Box` 对象的第三个 `Box` 对象。使用这个操作，就可以编写如下语句：

```
bigBox = box1 + box2 + box3;
```

在这个语句中，`+`操作的含义远远超出了简单的相加。`+`运算符应用于数值，会像以前那样工作，但应用于 `Box` 对象时，就有一个特殊的含义。这个语句中每个变量的类型都是 `Box`，上述代码会创建一个 `Box` 对象，其尺寸足够包含 `box1`、`box2` 和 `box3`。

编写像这样的语句要比分别处理所有的尺寸容易得多，计算过程越复杂，面向对象编程方式的优点就越明显。但这只是一个很粗略的说明，对象的功能要比这里所描述的强大得多。介绍这个例子的目的是让读者对使用面向对象方法来解决一个问题有一个大致的了解。面向对象的编程方法基本上是根据问题涉及的实体来解决问题，而不是根据计算机喜欢使用的实体，即数字和字符，来解决问题。

## 1.13 本章小结

本章简要介绍了 C++ 的一些基本概念。后面将详细讨论本章提及的内容。本章介绍的基本概念如下：

- C++ 程序包含一个或多个函数，其中一个是 `main()` 函数。执行总是从 `main()` 函数开始。
- 函数的可执行部分由包含在一对花括号中的语句组成。
- 一对花括号定义了一个语句块。
- 语句用分号结束。
- 关键字是 C++ 中有特殊含义的一组保留字。程序中的实体不能与 C++ 语言中的任何关键字同名。
- C++ 程序包含在一个或多个文件中。源文件包含可执行代码，头文件包含可执行代码使用的定义。
- 定义函数的代码通常存储在扩展名为 `.cpp` 的源文件中。
- 源文件使用的定义通常存储在扩展名为 `.h` 的头文件中。



- 预处理器指令指定了要对文件中的代码执行的操作。所有的预处理器指令都在编译文件中的代码之前执行。
- 头文件中的代码通过#include 预处理器指令添加到源文件中。
- 标准库提供了支持和扩展 C++语言的大量功能。
- 要访问标准库函数和定义,可以把标准库头文件包含到源文件中。
- 输入和输出是利用流来执行的,需要使用插入和提取运算符,即<<和>>。std::cin 是对应于键盘的标准输入流, std::cout 是把文本写入屏幕的标准输出流。它们都在标准库头文件 iostream 中定义。
- 面向对象的编程方式需要定义专用于某问题的新数据类型。一旦定义好需要的数据类型,就可以根据这些新数据类型来编写程序。
- Unicode 定义的独特整数代码值表示世界上几乎所有语言的字符,以及许多专用的字符集。代码值也称为代码点。Unicode 还定义了代码点如何编码为字节序列。

## 1.14 练习

1. 创建、编译、链接、执行一个程序,在屏幕上输出文本"Hello World"。
2. 创建并执行一个程序,在一行上输出自己的姓名,在下一行输出年龄。
3. 下面的程序有几处编译错误。请指出这些错误并更正,使程序能正确编译并运行。

```
include <iostream>

Int main()
{
    std::cout << "Hello World" << std::endl
}
```

注释:

本书所有练习的答案都可在 Apress 网站 [www.apress.com/source-code/](http://www.apress.com/source-code/)上找到。



## 第 2 章

## 基本数据类型

本章将介绍每个程序都需要的、C++内置的基本数据类型。C++的面向对象功能全部建立在这些基本数据类型的基础之上，因为用户创建的所有数据类型最终都是根据计算机操作的基本数值数据定义的。学习完本章后，读者将能编写传统格式(输入-处理-输出)的简单 C++程序。

## 本章主要内容

- C++中的数据类型
- 变量的声明和初始化
- 字面量的含义和定义方式
- 整数的二进制和十六进制表示
- 计算的过程
- 如何固定变量的值
- 如何创建存储字符的变量
- auto 关键字的作用
- lvalue 和 rvalue

## 2.1 变量、数据和数据类型

变量是用户定义的一个命名的内存段。每个变量都只存储特定类型的数据。每个变量都定义了可以存储的数据类型。每个基本类型都用唯一的类型名称(即关键字)来标识。关键字是 C++中的保留字，不能用于其他目的。

编译器会进行大量的检查，确保在给定的情况下使用正确的数据类型，它还确保在操作中合并不同的类型(例如将两个值相加)时，它们要么有相同的类型，要么可以把一个值转换为另一个值的类型，使它们相互兼容。编译器检测并报告尝试把不兼容的数据类型组合在一起而产生的错误。

数值分为两大类：整型数和浮点数(可以是分数)。在每个大类中，都有几种基本的 C++类型，每种类型都可以存储特定的数值范围。首先介绍整数类型。



## 2.1.1 定义整型变量

下面的语句定义了一个整型变量:

```
int apple_count;
```

这个语句定义了一个 `int` 类型的变量 `apple_count`, 该变量包含某个随机的垃圾值。在定义变量时, 可以而且应该指定初始值, 如下所示:

```
int apple_count {15}; // Number of apples
```

`apple_count` 的初始值放在变量名后面的花括号中, 所以其值为 15。包含初始值的花括号称为初始化列表。在本书后面, 初始化列表常常包含几个值。定义变量时不必初始化它们, 但最好进行初始化。确保变量一开始就有已知的值, 在代码不像预期的那样工作时, 将便于确定出错的位置。

类型 `int` 一般有 4 个字节, 可以存储从 -2,147,483,648 到 +2,147,483,647 的整数。这覆盖了大多数情形, 所以 `int` 是最常用的整数类型。

下面是 3 个 `int` 类型的变量定义:

```
int apple_count {15}; // Number of apples
int orange_count {5}; // Number of oranges
int total_fruit {apple_count + orange_count}; // Total number of fruit
```

`total_fruit` 的初始值是前面定义的两个变量值之和。这说明, 变量的初始值可以是表达式。在源文件中, 定义前述两个变量的语句必须放在定义 `total_fruit` 初始值的表达式前面, 否则 `total_fruit` 的定义不会编译。

花括号中的初始值必须与所定义的变量有相同的类型, 否则, 编译器就必须把它转换为需要的类型。如果转换为值域更小的类型, 就可能丢失信息, 所以编译器不会执行转换, 而是把它标记为一个错误。一个例子是把整型变量的初始值指定为非整数, 如 1.5, 可能原本要给 `apple_count` 输入 15, 却不小心输入了 1.5。转换为值域更小的类型称为缩窄转换。

初始化变量还有另外两种方式。函数表示法如下:

```
int orange_count(5);
int total_fruit(apple_count + orange_count);
```

还可以写为:

```
int orange_count = 5;
int total_fruit = apple_count + orange_count;
```

这两种方式是有效的, 但建议采用初始化列表形式。这是 C++ 11 引入的最新语法, 对初始化进行了标准化。这是首选方法, 因为它允许以相同的方式初始化所有变量。本章后面会解释它的一个例外情形。本书的所有例子都使用初始化列表, 除非不适合使用它。



可以在一个语句中定义和初始化给定类型的多个变量，例如：

```
int foot_count {2}, toe_count {10}, head_count {1};
```

这是合法的，在大多数情况下，最好在单个语句中定义每个变量。这会提高代码的可读性，还可以在注释中解释每个变量的作用。

可以把基本类型的任何变量值写入标准输出流。下面的程序演示了这一点：

```
// Ex2_01.cpp
// Writing values of variables to cout
#include <iostream>

int main()
{
    int apple_count {15}; // Number of apples
    int orange_count {5}; // Number of oranges
    int total_fruit {apple_count + orange_count}; // Total number of fruit

    std::cout << "The value of apple_count is " << apple_count << std::endl;
    std::cout << "The value of orange_count is " << orange_count << std::endl;
    std::cout << "The value of total_fruit is " << total_fruit << std::endl;
}
```

如果编译并执行这个程序，它会输出 3 个变量的值，再输出解释其含义的文本。二进制值会自动转换为字符表示，由插入运算符<<输出。这适用于任意基本类型的值。

### 1. 带符号的整数类型

表 2-1 列出了存储带符号整数(正数和负数)的所有基本类型。每种类型占用的内存和值域随编译器的不同而不同。

表 2-1 带符号的整数类型

类 型 名	类型的大小(字节)	值 域
signed char	1	-128~127
short	2	-256~255
short int		
int	4	-2 147 483 648~+2 147 483 647
long	4	-2 147 483 648~+2 147 483 647
long int		
long long	8	-9 223 372 036 854 775 808~9 223 372 036 854 775 807
long long int		

类型 signed char 一般占 1 字节，其他类型占用的字节数取决于编译器。左列出现两个类型名称时，第一个缩写名称比较常用，所以应总是使用 long，而不是 long int。在列表中，每个类型占用的内存都至少与前一个类型一样多。



## 2. 无符号的整数类型

当然,有时不需要存储负数。班级的学生数或部件中的零件数总是正整数。在带符号的整数类型前面加上 `unsigned` 关键字,例如 `unsigned char`、`unsigned short` 或 `unsigned long`,就可以指定只存储非负值的整数类型。每个不带符号的类型都不同于带符号的类型,但占用相同的内存空间。

类型 `char` 是不同于 `signed char` 和 `unsigned char` 的整数类型。类型 `char` 存储一个字符码,可以是带符号的类型,也可以是不带符号的类型,这取决于编译器。如果 `climits` 头文件中的 `CHAR_MIN` 常量是 0,编译器就把 `char` 类型解释为不带符号的类型,本章后面将详细讨论存储字符的类型。

下面是这些类型的变量示例:

```
signed char ch {20};
long temperature {-50L};
long width {500L};
long long height {250LL};
unsigned int toe_count {10U};
unsigned long angel_count {1000000UL};
```

注意如何编写 `long` 类型和 `long long` 类型的常量。必须给 `long` 类型添加 `L` 后缀,给 `long long` 类型添加 `LL` 后缀。如果没有这些后缀,整型常量的类型就是 `int`。可以使用 `L` 和 `LL` 的小写形式,但建议不要这么做,因为小写的 `l` 很容易与数字 1 混淆。不带符号的整型常量使用 `u` 或 `U` 后缀。

### 2.1.2 定义有固定值的变量

有时希望定义有固定值或值不会改变的变量。在变量的定义中使用 `const` 关键字,就可以定义不能修改的变量。例如:

```
const unsigned int toe_count {2U};
```

`const` 关键字告诉编译器, `toe_count` 的值不能修改。尝试修改 `toe_count` 值的语句会在编译期间标记为错误。使用 `const` 关键字可以固定任何类型的变量值。

## 2.2 整型字面量

任何类型的常量,例如 42、2.71828、'Z'或"Mark Twain",都称为字面量。这些例子分别是整型字面量、浮点字面量、字符字面量和字符串字面量。每个字面量都有特定的类型。下面先介绍整型字面量,再介绍其他类型的字面量。



### 2.2.1 十进制整型字面量

可以用非常直接的方式编写整型字面量。下面是十进制整数的一些例子:

```
-123L +123 123 22333 98U -1234LL 12345ULL
```

如前所述,不带符号的整型字面量有 `u` 或 `U` 后缀。`long` 类型和 `long long` 类型的字面量分别有 `L` 和 `LL` 后缀。如果它们不带符号,就也使用 `u` 或 `U` 后缀。`U`、`L` 和 `LL` 的顺序任意。在第二个例子中,可以省略“+”,因为这是默认的,但为了使该数值的含义更清晰,加上“+”也不会出问题。字面量 `+123` 与 `123` 是相同的,其类型都是 `int`,因为它们没有后缀。第 4 个例子在一般情况下写为 `22,333`,但在整型字面量中不能使用逗号。下面是使用这些字面量的一些语句:

```
unsigned long age {99UL}; // 99uL would be OK too
unsigned short {10u}; // There is no specific literal type for short
long long distance {1234567LL};
```

不能把老式的整数值用作变量的初始值。初始值必须位于变量类型的允许范围内,且类型必须匹配。表达式中的字面量必须在某种类型的范围中。

### 2.2.2 十六进制的整型字面量

可以把整数字面量表示为十六进制数。十六进制字面量要加上 `0x` 或 `0X` 前缀。所以 `0x999` 是一个 `int` 类型的十六进制数,它有 3 个十六进制数字,而 `999` 就是一个 `int` 类型的十进制数,它有 3 个十进制数字,它们是完全不同的。下面是十六进制字面量的一些例子:

```
十六进制值: 0x1AF 0x123U 0xAL 0xcad 0xFF
十进制值:    431 291U 10L 3245 255
```

十六进制的整型字面量主要用于定义位的特定模式。因为每个十六进制位都对应二进制值的 4 位,所以很容易把位的特定模式表达为十六进制的字面量。像素颜色的红、蓝、绿成分(RGB 值)常常表示为 32 位字的 3 个字节。白色可以指定为 `0xFFFFFFFF`,因为在白色中,上述三种颜色成分的强度都是最大值 255,即 `0xFF`。红色则是 `0xff0000`。下面是一些例子:

```
unsigned int color {0x0f0d0eU}; // Unsigned int hexadecimal constant -
decimal 986,382
int mask {0xFF00FF00}; // Four bytes specified as FF, 00, FF, 00
unsigned long value {0xdeadLU}; // Unsigned long hexadecimal literal -
decimal 57,005
```

### 2.2.3 八进制的整型字面量

还可以把整数字面量表示为八进制值,即以 8 为基数。把数值表示为八进制时,要



给它加上一个前导 0。下面列出了八进制值的一些例子。

八进制值:	0657	0443U	012L	06255	0377
对应的十进制整数:	431	291U	10L	3245	255

**注意:**

不要给十进制的整数值加上前导 0。编译器会把这种数值解释为八进制(基数为 8),因此表示为 065 的值就等价于十进制表示法中的 53。

## 2.2.4 二进制的整型字面量

二进制的整型字面量是写为带有前缀 0b 或 0B 的一系列二进制数字(0 或 1)。二进制的整型字面量可以把 L 或 LL 作为后缀,表示其类型是 long 或 long long,把 u 或 U 作为后缀,表示它是不带符号的字面量。例如:

二进制值:	0B110101111	0b100100011U	0b1010L	0B11001101	0b11111111
对应的十进制值:	431	291U	10L	3245	255

二进制的整型字面量在 C++ 14 标准中引入,所以编写本书时,支持它们的编译器不多。下面是其用法示例:

```
int color {0b000011110000110100001110};
int mask {0B11111111000000001111111100000000}; // 4 bytes
unsigned long value {0B1101111010101101UL};
```

前面的代码段已演示了如何编写前缀和后缀的各种组合,例如 0x、0X、UL、LU 或 Lu,但最好坚持使用整数字面量的统一写法。

从编译器的角度来看,它并不在意用户表示整数值时使用什么进制,该数值最终都会在计算机中存储为一个二进制数。在表示整数时使用什么方式只取决于是否方便。应选择适合于当前环境的某种进制。

**注意:**

在整数字面量中可以使用单引号作为分隔符,使字面量更容易阅读,例如 0B1111'1010 或 23'568'987UL。但在编写本书时,支持它的编译器很少,所以这可能适合你。

## 2.3 整数的计算

首先,介绍一些术语。运算(例如计算相加或相乘)是由运算符定义的,例如,+用于相加,\*用于相乘。运算符操作的数值称为操作数,在表达式 2\*3 中,操作数是 2 和 3。乘法运算符需要两个操作数,所以称为二元运算符。只需要一个操作数的运算符称为一元运算符。一元运算符的一个例子是表达式 -width 中的减号。减号对 width 的值取反,



所以该表达式的结果是对其操作数的符号取反。表达式 `width*height` 中的二元乘法运算符与此相反，它作用于两个操作数 `width` 和 `height`。

对整数可以进行的基本算术运算如表 2-2 所示。

表 2-2 基本的算术运算

运 算 符	运 算
+	加
-	减
*	乘
/	除
%	取模(除法运算后的余数)

表 2-2 中的运算符都是二元运算符，其工作方式与我们期望的大致相同。表达式中的乘法、除法和取余操作在加减操作之前执行。下面是其用法示例：

```
long width {4L};
long length {5L};
long area {0L};
long perimeter {0L};
area = width*length;           // Result is 20
perimeter = 2L*width + 2L*length; // Result is 28
```

最后两行是赋值语句，`=`是赋值运算符。计算赋值运算符右边的算术表达式，结果存储在赋值运算符左边的变量中。有两个变量初始化为 `0L`。这里可以忽略初始化列表中的 `0L`，结果是相同的，因为空初始化列表就假定为包含 `0`。第二和第三个语句分别定义了 `area` 和 `perimeter`，它们也可以写作：

```
long area {};
long perimeter {};
```

赋值运算符与代数方程中的`=`大不相同，这很重要。后者表示相等，前者指定了一个操作。考虑下面的赋值语句：

```
int y {5};
y = y + 1;
```

变量 `y` 初始化为 `5`，所以表达式 `y+1` 等于 `6`。这个结果存储回 `y`，所以其效果是 `y` 递增 `1`。

使用圆括号可以控制较复杂表达式的执行顺序。计算 `perimeter` 值的语句可以写作：

```
perimeter = 2L*(width + length);
```

圆括号中的子表达式先计算，其结果乘以 `2`，最终得到与前述语句相同的值。其效率比最初的语句高，因为它需要 `3` 个算术操作，而不是 `4` 个。

圆括号可以嵌套，即圆括号中的子表达式按照从最内层圆括号到最外层圆括号的顺



序计算。嵌套圆括号的表达式示例说明了其工作方式:

```
2*(a + 3*(b + 4*(c + 5*d)))
```

表达式  $5*d$  先计算,再给其结果加上  $c$ ,结果乘以 4,接着加上  $b$ ,然后其结果乘以 3,再加上  $a$ 。最后把结果乘以 2,得到整个表达式的结果。

除法运算略微特殊一点。整数运算总是得到整数结果,例如,表达式  $11/4$  的结果不是 2.75,而是 2。“整数除法”返回被除数和除数相除后的整数部分,舍弃了余数部分。在 C++ 标准看来,除以 0 的结果是不确定的,但特定的实现方式通常定义了其结果,所以读者应查阅产品的文档说明。

图 2-1 说明了除法和取模的不同结果。

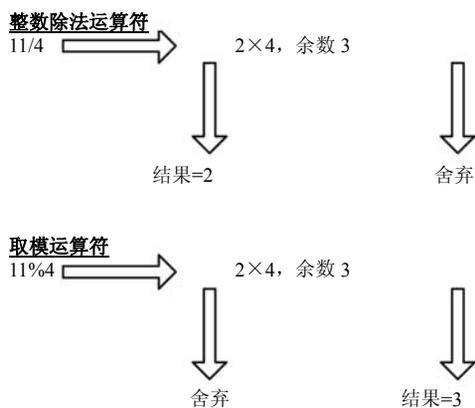


图 2-1 比较除法运算符和取模运算符

取模运算符  $\%$  是对除法运算符的补充,它提供了一种在整数除法运算后获取其余数的方式。取模运算符的一个或两个操作数是负数时,余数的符号由所使用的编译器来确定,因此不同的系统会有不同的结果。使用取模运算符肯定要涉及相除,所以在右操作数为 0 时,其结果是不确定的。

## 赋值操作的更多内容

可以在一个语句中对多个变量赋值。例如:

```
int a {}, b {}, c {5}, d{4};  
a = b = c*c - d*d;
```

第二个语句计算表达式  $c*c - d*d$  的值,并把结果存储在  $b$  中,所以  $b$  设置为 9。接着把  $b$  的值存储在  $a$  中,所以  $a$  也设置为 9,这种重复赋值可以使用任意多次。

等号左边的操作数可以是变量或表达式,但如果它是一个表达式,结果就必须是一个 lvalue。lvalue 表示一个永久的内存位置,所以变量是 lvalue。C++ 中的每个表达式都会得到 lvalue 或 rvalue。rvalue 是一个结果,而不是 lvalue,所以它是临时的。上述语句中表达式  $c*c - d*d$  的结果就是 rvalue。编译器会分配一个临时的内存位置来存储该表达



式的结果, 但一旦该语句执行完毕, 结果及其占用的内存就会舍弃。lvalue 和 rvalue 之间的区别目前并不重要, 但涉及函数和类时, 这个区别就非常重要了。

下面看看示例中的算术运算符。这个程序转换从键盘上输入的距离, 在这个过程中演示了如何使用算术运算符:

```
// Ex2_02.cpp
// Converting distances
#include <iostream>           // For output to the screen

int main()
{
    unsigned int yards {}, feet {}, inches {};

    // Convert a distance in yards, feet, and inches to inches
    std::cout << "Enter a distance as yards, feet, and inches "
               << "with the three values separated by spaces:"
               << std::endl;
    std::cin >> yards >> feet >> inches;

    const unsigned int feet_per_yard {3U};
    const unsigned int inches_per_foot {12U};
    unsigned int total_inches {};
    total_inches = inches + inches_per_foot*(yards*feet_per_yard + feet);
    std::cout << "The distances corresponds to " << total_inches << "
inches.\n";

    // Convert a distances in inches to yards feet and inches
    std::cout << "Enter a distance in inches: ";
    std::cin >> total_inches;
    feet = total_inches/inches_per_foot;
    inches = total_inches%inches_per_foot;
    yards = feet/feet_per_yard;
    feet = feet%feet_per_yard;
    std::cout << "The distances corresponds to "
               << yards << " yards "
               << feet << " feet "
               << inches << " inches." << std::endl;
}
```

这个示例的典型输出如下:

```
Enter a distance as yards, feet, and inches with the three values separated
by spaces:
9 2 11
The distances corresponds to 359 inches.
Enter a distance in inches: 359
The distances corresponds to 9 yards 2 feet 11 inches.
```

main()中的第一个语句定义了 3 个整型变量, 并把它们初始化为 0。它们的类型都是



unsigned int, 因为在这个示例中, 距离值都不会是负的。这个示例在一个语句中定义了 3 个变量, 这是可行的, 因为它们是紧密相关的。

下一条语句把一个输入提示信息输出到 `std::cout`。该语句横跨 3 行, 也可以写成 3 条不同的语句:

```
std::cout << "Enter a distance as yards, feet, and inches ";
std::cout << "with the three values separated by spaces:";
std::cout << std::endl;
```

像最初的语句那样有一系列<<运算符时, 它们按从左到右的顺序执行, 所以上述 3 个语句的输出与最初的语句相同。

下一条语句从 `cin` 中读取值, 并存储到变量 `yards`、`feet` 和 `inches` 中。>>运算符期望读取的值的类型由要存储该值的变量类型确定, 所以需要输入不带符号的整数。<<运算符会忽略空格, 值后跟的第一个空格会终止该操作。这说明, 不能使用<<运算符从流中读取和存储空格, 即使把空格存储在保存字符的变量中, 也是如此。示例中的输入语句还可以写成 3 条独立的语句:

```
std::cin >> yards;
std::cin >> feet;
std::cin >> inches;
```

这些语句的结果与最初的语句相同。

然后定义两个变量 `inches_per_foot` 和 `feet_per_yard`, 用于把码、英尺和英寸转换为英寸, 把英寸转换为码、英尺和英寸。这些变量的值是固定的, 所以把它们指定为 `const`。在代码中可以使用显式值作为转换因子, 但使用 `const` 变量更好, 因为这清楚地说明了代码要做什么。`const` 变量也是正值, 所以把它们定义为 `unsigned int` 类型。转换为英寸用一条语句完成:

```
total_inches = inches + inches_per_foot*(yards*feet_per_yard + feet);
```

圆括号中的表达式先计算, 它把 `yards` 值转换为英尺, 再加上 `feet` 值, 得到总的英尺值。这个结果乘以 `inches_per_foot`, 会得到 `yards` 和 `feet` 的总英寸值。给它加上 `inches` 值, 得到最终的总英寸值, 再用下面的语句输出:

```
std::cout << "The distances corresponds to " << total_inches << "
inches.\n";
```

第一个字符串传递到标准输出流 `cout` 中, 之后输出 `total_inches` 的值。下一个传递到 `cout` 的字符串是 `\n`, 这是最后一个字符, 表示下一个输出从下一行开始。

把一个英寸值转换为码、英尺和英寸需要 4 条语句:

```
feet = total_inches/inches_per_foot;
inches = total_inches%inches_per_foot;
yards = feet/feet_per_yard;
feet = feet%feet_per_yard;
```



再次使用存储前述转换的输入的变量，来存储这个转换的结果。`total_inches` 的值除以 `inches_per_foot`，得到英尺值，存储在 `feet` 中。`%`运算符生成了除法运算的余数，所以下一条语句计算所余的英寸数，并存储在 `inches` 中。再使用这个过程计算码数和最终的英尺数。

最后的输出语句后面没有 `return` 语句，因为不需要它。执行序列超过 `main()`的结尾时，就等价于执行 `return 0`。

## 2.4 op=赋值运算符

在 `Ex2_01.cpp` 中，下面的语句有一个更简略的编写形式：

```
feet = feet%feet_per_yard;
```

这条语句可以用 `op=` 赋值运算符来编写。之所以称为 `op=` 赋值运算符，是因为它们由一个运算符和一个等于号“=”组成。上面的语句可以写成：

```
feet %= feet_per_yard;
```

这条语句执行的操作与上一条语句完全相同。

`op=` 赋值语句的一般形式如下：

```
lhs op= rhs;
```

其中 `lhs` 是某个变量，用于存储该运算符的执行结果。`rhs` 是一个表达式。这等价于语句：

```
lhs = lhs op (rhs);
```

圆括号很重要，因为可以编写这样的语句：

```
x *= y + 1;
```

这等价于：

```
x = x*(y + 1);
```

没有隐含的圆括号，存储在 `x` 中的值就是 `x * y + 1` 的运算结果，这完全不同。

可以对许多运算符使用 `op=`形式的赋值。表 2-3 是一个完整的列表，包括第 3 章将介绍的一些运算符。

表 2-3 op=赋值运算符

操 作	运 算 符	操 作	运 算 符
加	<code>+=</code>	按位与	<code>&amp;=</code>
减	<code>-=</code>	按位或	<code> =</code>
乘	<code>*=</code>	按位异或	<code>^=</code>
除	<code>/=</code>	向左移位	<code>&lt;&lt;=</code>
取模	<code>%=</code>	向右移位	<code>&gt;&gt;=</code>



注意在 `op` 和 “=” 之间没有空格。如果包含空格,就会出现错误。希望给变量递增某个数时,就可以使用 `+=`。例如,下面两条语句的作用相同:

```
y = y + 1;  
y += 1;
```

表 2-3 中的移位运算符 `<<`和`>>`,看起来与用于流的插入和提取运算符相同。编译器可以根据上下文判断出 `<<`和`>>`在语句中的含义。本书后面将解释相同的运算符如何在不同的情形下表示不同的含义。

## 2.5 using 声明和指令

`Ex2_01.cpp` 中有许多个 `std::cin` 和 `std::cout`。在源文件中,使用 `using` 声明就不需要用名称空间名限定名称了。下面是一个示例:

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

这告诉编译器,在编写 `cout` 时,它应解释为 `std::cout`。这个声明位于 `main()`函数的定义之前,所以可以用 `cout` 代替 `std::cout`,减少输入量,使代码不那么烦琐。

可以在 `Ex2_01.cpp` 的开头包含两个 `using` 声明,这样就不必限定 `cin` 和 `cout` 了:

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

当然,仍需要用 `std` 限定 `endl`,但也可以给它添加 `using` 声明。任何名称空间中的名称都可以应用 `using` 声明,而不仅仅是 `std`。

`using` 指令会导入名称空间中的所有名称。下面可以使用 `std` 名称空间中的所有名称,无须限定它们:

```
using namespace std; // Make all the names in std available without  
qualification
```

源文件的开头包含这条语句后,就不需要限定 `std` 名称空间中定义的所有名称了。初看起来这是一个很有吸引力的方法,但问题是它与引入名称空间的主要原因冲突。我们不可能知道 `std` 定义的所有名称,有了这个 `using` 指令,就会增加自己的名称与 `std` 中的名称冲突的可能性。

本书的示例偶尔会给 `std` 名称空间使用 `using` 指令,否则 `using` 声明的数量会非常多。建议仅在在有充足的理由时使用 `using` 指令。

## 2.6 sizeof 运算符

使用 `sizeof` 运算符可以得到某类型、变量或表达式结果占用的字节数。下面是其用



法示例:

```
int height {74};
std::cout << "The height variable occupies " << sizeof height << " bytes."
<< std::endl;
std::cout << "Type \"long long\" occupies " << sizeof (long long) << "
bytes." << std::endl;
std::cout << "The expression height*height/2 occupies "
<< sizeof (height*height/2) << " bytes." << std::endl;
```

这些语句说明了如何输出变量、类型或表达式结果占用的字节数。要使用 `sizeof` 运算符获得类型占用的内存, 必须把类型名放在圆括号中。还需要给 `sizeof` 表达式加上圆括号。不需要给变量名加上圆括号, 但加上也没有害处。因此, 如果总是给 `sizeof` 表达式加上圆括号, 就不会出错。

可以给任何基本类型、类类型或指针类型(指针参见第 5 章)使用 `sizeof`。其结果的类型是 `size_t`, 这是在标准库头文件 `cstdint` 中定义的一个不带符号的整数。类型 `size_t` 的实现是已经定义好的, 但应看看编译器实现它的方式。如果使用了 `size_t`, 代码就可以用于任何编译器。

现在读者应能创建自己的程序, 让编译器列出基本整数类型的大小。

## 2.7 整数的递增和递减

前面介绍了如何使用 `+=` 运算符递增变量的值。显然, 还可以用 `-=` 运算符递减变量的值。另外两个运算符也可以执行递增和递减任务, 它们分别称为递增和递减运算符, 即 `++` 和 `--`。

这两个运算符并不只是递增和递减的另一个选项, 在进一步应用 C++ 的过程中, 可以看出它们的价值。递增和递减运算符是一元运算符, 可以应用于整型变量。下面修改 `count` 的语句有相同的作用:

```
int count {5};
count = count + 1;
count += 1;
++count;
```

上面的每条语句都给变量 `count` 递增 1。显然, 使用递增运算符是最简洁的形式。这个运算符的操作不同于前面介绍的其他运算符, 因为它直接修改其操作数的值。表达式的结果是递增变量的值, 再在表达式中使用已递增的值。例如, 如果 `count` 的值是 5, 则执行下面的语句:

```
total = ++count + 6;
```

递增和递减运算符的优先级高于表达式中的其他二元算术运算符, 因此, `count` 的值先递增为 6, 再在等号右边的表达式中使用这个值 6, 所以变量 `total` 的值就是 12。



可以用相同的方式使用递减运算符:

```
total = --count + 6;
```

在执行这条语句之前,假定 `count` 的值为 6, `--`运算符把 `count` 的值减为 5, 这个值再用来计算存储在 `total` 中的值, 结果是 11。

前面都是把 `++`和`--`运算符放在变量的前面, 这称为前缀形式。`++`和`--`运算符也可以放在变量的后面, 这称为后缀形式, 其结果与前缀形式略有不同。

## 递增和递减运算符的后缀形式

在使用 `++`的后缀形式时, 先在表达式中使用变量的值进行计算, 再递增该变量的值。例如, 把前面的例子改写为:

```
total = count++ + 6;
```

`count` 的初始值还是 5, 但 `total` 的值应是 11, `count` 再递增为 6。上面的语句等价于:

```
total = count + 6;  
++count;
```

在像 `a++ + b`, 甚至 `a+++b` 这样的表达式中, 其含义并不是很明显, 或者不清楚编译器会执行什么操作。这两个表达式的含义是相同的, 但第二个表达式也可能意味着 `a++ + b`, 它的含义就不同了, 等价于另外两个表达式, 如下表达更清晰:

```
total = 6 + count++;
```

另外, 还可以使用括号:

```
total = (count++) + 6;
```

前面应用于递增运算符的规则也适用于递减运算符。例如, 如果 `count` 的初始值是 5, 则语句:

```
total = --count + 6;
```

`total` 的值是 10。如果将语句改写为:

```
total = 6 + count-- ;
```

`total` 的值就是 11。

必须避免在一个表达式中对给定变量多次使用这些运算符的前缀形式。假定变量 `count` 的值是 5, 则语句:

```
total = ++count * 3 + ++count * 5;
```

由于该语句多次修改了变量 `count` 的值, 结果就是不确定的, 编译器会给这个语句生成一个错误消息。



还要注意，下面语句的结果也是不确定的：

```
k = ++k + 1;
```

这条语句递增赋值运算符左边的变量的值，因此在一个表达式中对变量  $k$  的值修改了两次。计算一个表达式的结果只能对每个变量修改一次，变量以前的值只能用于确定要存储的值。根据 C++ 标准，这种表达式是不确定的，但这并不表示编译器不会编译它。这只代表不能保证结果的一致性。

递增和递减运算符通常应用于整数，尤其常用于循环，详见第 5 章。本章后面还把它们应用于浮点数。后面的章节将探讨它们如何应用于某些其他数据类型，并能够得到特别且非常有用的结果。

## 2.8 定义浮点变量

希望使用非整数值时，可以使用浮点变量。浮点数的数据类型有 3 种，如表 2-4 所示。

表 2-4 浮点数的数据类型

数据类型	说明
float	单精度浮点数
double	双精度浮点数
long double	扩展的双精度浮点数

这里的术语“精度”是指尾数中的位数。上述数据类型的精度按从上到下的顺序逐步增加，float 在尾数中的位数最少，long double 的位数最多。注意精度只确定尾数中的位数。某一类型表示的值域主要由指数的可能范围确定。

C++ 标准并没有描述精度和数值范围，所以这些类型的精度和数值范围就由编译器决定，也取决于计算机使用的处理器类型和它使用的浮点数表示方法。long double 类型提供的精度不小于 double 类型提供的精度，double 类型提供的精度不小于 float 类型提供的精度。

通常，float 类型提供 7 位精度，double 类型提供 15 位精度，long double 类型提供 19 位精度，但 double 类型和 long double 类型在一些编译器上的精度是相同的。在 Intel 处理器上，浮点数类型表示的取值范围如表 2-5 所示。

表 2-5 浮点数据类型的取值范围

类型	精度(位数)	取值范围(+或-)
float	7	$1.2 \times 10^{-38} \sim 3.4 \times 10^{38}$
double	15	$2.2 \times 10^{-308} \sim 1.8 \times 10^{308}$
long double	19	$3.3 \times 10^{-4932} \sim 1.2 \times 10^{4932}$



表 2-5 中数字的精度都是大约数。显然, 这些类型都可以精确地表示 0, 但不能表示 0 和正负范围中下限之间的值, 所以这些下限是非 0 值中最小的值。

下面是定义浮点变量的语句:

```
float inches_to_mm {25.4f};
double pi {3.1415926}; // Ratio of circle circumference to diameter
long double root2 {1.4142135623730950488L}; // Square root of 2
```

可以看出, 浮点变量的定义与整数变量相同。在大多数情况下, 使用类型 `double` 就足够了。

## 2.8.1 浮点字面量

从上一节的代码段可以看出, `float` 字面量添加了后缀 `f`(或 `F`), `long double` 字面量添加了后缀 `L`(或 `l`)。没有后缀的浮点字面量是 `double` 类型。浮点字面量包含小数点或指数, 或者两者都包含; 两者都没有的数是整数。

在浮点字面量中, 指数是可选的, 表示 10 的幂乘以该值。指数必须带有前缀 `e` 或 `E`, 其后是指数值。下面是包含指数的一些浮点字面量:

```
5E3 (5000.0) 100.5E2 (10050.0) 2.5e-3 (0.0025) -0.1E-3L (-0.0001L) .345e1F
(3.45F)
```

带指数的每个字面量后面, 圆括号中的值相当于没有指数的字面量。需要表示非常大或非常小的值时, 指数特别有用。

## 2.8.2 浮点数的计算

浮点数的计算与整数计算相同。例如:

```
const double pi {3.1414926}; // Circumference of a circle divided by its
diameter
double a {0.75}; // Thickness of a pizza
double z {5.5}; // Radius of a pizza
double volume {}; // Volume of pizza - to be calculated
volume = pi*z*z*a;
```

取模运算符 `%` 不能用于浮点操作数, 但前面介绍的其他二元算术运算符, 如 `+`、`-`、`*` 和 `/`, 都可以用于浮点操作数。还可以对浮点数变量应用前缀和后缀形式的递增及递减运算符, 其作用与处理整数相同, 变量会递增或递减 1.0。

## 2.8.3 缺点

应了解使用浮点数变量的局限。如果不小心, 结果可能不准确, 甚至不正确。下面是使用浮点数值时常见的错误原因:



- 一些小数值没有准确转换为二进制浮点数值。在计算过程中，很容易把一些小错误放大为大错误。
- 计算两个非常接近的数值之差会丧失精度。如果考虑两个 float 数值之差，而这两个数值仅在第 6 位的数字有区别，那么其结果是只有一或两位是精确的，其他位则可能出错。
- 处理范围相差几个数量级的数值会导致错误。一个简单的例子是：把两个值存储为精度为 7 位的 float 类型的浮点数，可是，其中一个值比另一个值大  $10^8$  倍，对它们执行相加操作。把较小的值加到较大值上任意多次，较大的值是不会有明显变化的。
- <cmath>标准库头文件包含用于编译器的浮点操作信息。其中，它定义了下述值，其前缀 FLT\_、DBL\_ 和 LDBL\_ 表示分别与 float、double 和 long double 类型相关。
- FLT\_MANT\_DIG、DBL\_MANT\_DIG 和 LDBL\_MANT\_DIG 是尾数的位数。
- FLT\_EPSILON、DBL\_EPSILON 和 LDBL\_EPSILON 是可以加到 1.0 上的最小值，并且得到不同的结果。
- FLT\_MAX、DBL\_MAX 和 LDBL\_MAX 是可以表示的最大浮点数。
- FLT\_MIN、DBL\_MIN 和 LDBL\_MIN 是可以表示的最小非 0 浮点数。

这些常量很容易输出。下面是演示它们的完整程序：

```
// Ex2_03.cpp
// Writing floating-point properties to cout
#include <iostream> // For output to the screen
#include <cmath>

int main()
{
    std::cout << "The mantissa for type float has " << FLT_MANT_DIG << " bits."
              << std::endl;
    std::cout << "The maximum value of type float is " << FLT_MAX << std::endl;
    std::cout << "The minimum non-zero value of type float is " << FLT_MIN
              << std::endl;
}
```

下载代码中的这个示例输出的常量比这里多。

## 2.8.4 无效的浮点结果

根据 C++ 标准，除 0 的结果也是不确定的。但一些 C++ 实现方式有自己的处理方式，所以应参阅产品的文档说明。在大多数计算机上，硬件的浮点操作都是根据 IEEE 754 标准(也称为 IEC 559)实现的。浮点标准定义了几个特殊的值，它们的二进制尾数都是 0，指数都是 1，根据其符号表示 +infinity 和 -infinity。一个非 0 的正数除以 0 时，结果就是 +infinity，一个非 0 的负数除以 0 时，结果就是 -infinity。另一个特殊的浮点值称为 Not a Number，通常缩写为 NaN，用于表示数学上没有定义的结果，例如 0 除 0 或无穷大除以无穷大。



一个或两个操作数是 NaN 时,所有后续的操作结果都是 NaN。程序中的一个操作得到值  $\pm\infty$ , 就会影响该值参与的所有后续操作。表 2-6 总结了这些可能性。

表 2-6 NaN 和  $\pm\infty$  操作数的浮点操作

操 作	结 果	操 作	结 果
$\pm\text{value}/0$	$\pm\infty$	$0/0$	NaN
$\pm\infty\pm\text{value}$	$\pm\infty$	$\pm\infty/\pm\infty$	NaN
$\pm\infty*\text{value}$	$\pm\infty$	$\infty-\infty$	NaN
$\pm\infty/\text{value}$	$\pm\infty$	$\infty*0$	NaN

表中的值都是任意非 0 值。把下面的代码放在 main() 中, 就可以看出编译器如何表示这些值。

```
double a{ 1.5 }, b{}, c{}, result{};
result = a / b;
std::cout << a << "/" << b << " = " << result << std::endl;
std::cout << result << " + " << a << " = " << result + a << std::endl;
result = b / c;
std::cout << b << "/" << c << " = " << result << std::endl;
```

运行这个程序, 就可以从输出中看到  $\pm\infty$  和 NaN。

## 2.9 数值函数

<cmath> 标准库头文件定义了许多可以在程序中使用的三角函数和数值函数, 所有函数名都在 std 名称空间中。表 2-7 列出了这个头文件中最有用的函数。

表 2-7 &lt;cmath&gt; 头文件的数值函数

函 数	说 明
abs(arg)	返回 arg 的绝对值, 其类型与 arg 相同, arg 可以是任意浮点类型。在 <cstdlib> 头文件中还声明了变元是任意整数类型的 abs() 函数版本, 其结果也是整数类型
fabs(arg)	返回 arg 的绝对值, 其类型与 arg 相同。变元可以是 int、long、float、double 或 long double
ceil(arg)	返回与 arg 类型相同的一个浮点值, 该值是大于或等于 arg 的最小整数, 所以 std::ceil(2.5) 返回值 3.0, std::ceil(-2.5) 返回值 -2.0。arg 可以是任意浮点类型
floor(arg)	返回与 arg 类型相同的一个浮点值, 该值是小于或等于 arg 的最大整数, 所以 std::floor(2.5) 返回值 2.0, std::floor(-2.5) 返回值 -3.0。arg 可以是任意浮点类型
exp(arg)	返回 $e^{\text{arg}}$ 的值, 其类型与 arg 相同。arg 可以是任意浮点类型
log(arg)	log 函数返回 arg 的自然对数, 其类型与 arg 相同。arg 可以是任意浮点类型
log10(arg)	log10 函数返回 arg 以 10 为底的对数, 其类型与 arg 相同。arg 可以是任意浮点类型
pow(arg1,arg2)	pow 函数返回 arg1 的 arg2 次方, 即 $\text{arg1}^{\text{arg2}}$ 。两个变元的类型都是整数或浮点类型。因此, std::pow(2,3) 的结果是 8, std::pow(1.5,3) 的结果是 3.375



表 2-8 列出了 `<cmath>` 头文件中的三角函数。

表 2-8 `<cmath>` 头文件中的三角函数

函 数	说 明
<code>cos(angle)</code>	返回 <code>angle</code> 的余弦, <code>angle</code> 变元以弧度为单位
<code>sin(angle)</code>	返回 <code>angle</code> 的正弦, <code>angle</code> 变元以弧度为单位
<code>tan(angle)</code>	返回 <code>angle</code> 的正切, <code>angle</code> 变元以弧度为单位
<code>cosh(angle)</code>	返回 <code>angle</code> 的双曲线余弦, <code>angle</code> 变元以弧度为单位。变量 <code>x</code> 的双曲线余弦由公式 $(e^x - e^{-x})/2$ 确定
<code>sinh(angle)</code>	返回 <code>angle</code> 的双曲线正弦, <code>angle</code> 变元以弧度为单位。变量 <code>x</code> 的双曲线正弦由公式 $(e^x + e^{-x})/2$ 确定
<code>tanh(angle)</code>	返回 <code>angle</code> 的双曲线正切, <code>angle</code> 变元以弧度为单位。变量 <code>x</code> 的双曲线余弦是 <code>x</code> 的双曲线正弦除以 <code>x</code> 的双曲线余弦
<code>acos(arg)</code>	返回 <code>arg</code> 的反余弦。变元必须在 $-1$ 和 $+1$ 之间。结果以弧度为单位, 其范围是 $0$ 到 $\pi$
<code>asin(arg)</code>	返回 <code>arg</code> 的正弦。变元必须在 $-1$ 和 $+1$ 之间。结果以弧度为单位, 其范围是 $-\pi/2$ 到 $+\pi/2$
<code>atan(arg)</code>	返回 <code>arg</code> 的正切。结果以弧度为单位, 其范围是 $-\pi/2$ 到 $+\pi/2$
<code>atan2(arg1,arg2)</code>	这个函数需要两个浮点类型的变元, 返回 <code>arg1/arg2</code> 的正切。结果以弧度为单位, 其范围是 $-\pi$ 到 $+\pi$ , 其类型与变元相同

这些函数的变元可以是任意浮点类型, 返回的结果与变元类型相同。

下面是使用这些函数的例子。下面的语句可以计算出一个角度的正弦:

```
double angle {1.5}; // In radians
double sine_value {std::sin(angle)};
```

如果角度以度为单位, 就可以使用  $\pi$  的值把它转换为弧度, 再计算正切:

```
float angle_deg {60.0f}; // Angle in degrees
const float pi {3.14159f};
const float pi_degrees {180.0f};
float tangent {std::tan(pi*angle_deg/pi_degrees)};
```

如果知道教堂尖塔的高度是 100 英尺, 并可以站在距离尖塔底部 50 英尺的地方, 就可以计算出尖塔的顶角, 单位是弧度, 如下所示:

```
double height {100.0} // Steeple height- feet
double distance {50.0} // Distance from base
angle = std::atan2(height, distance); // Result in radians
```

可以使用 `angle` 中的值和 `distance` 中的值计算当前位置到尖塔顶部的距离:

```
double toe_to_tip {distance*std::cos(angle)};
```



当然, Pythagoras of Samos 的拥护者还可以用更简单的方法获得这个结果, 如下所示:

```
double toe_to_tip {std::sqrt(std::pow(distance,2) + std::pow(height, 2)};
```

下面是一个浮点示例。假定要构建一个圆形的池塘来养鱼。通过研究发现, 必须保证该池塘的表面积为 2 平方英尺, 才能确保每条鱼有 6 英寸长。本例需要确定池塘的直径, 以确保鱼有足够的空间。下面就是实现过程:

```
// Ex2_04.cpp
// Sizing a pond for happy fish
#include <iostream>
#include <cmath> // For square root function
using std::cout;
using std::cin;
using std::sqrt;

int main()
{
    // 2 square feet pond surface for every 6 inches of fish
    const double fish_factor {2.0/0.5}; // Area per unit length of fish
    const double inches_per_foot {12.0};
    const double pi {3.14159265};

    double fish_count {}; // Number of fish
    double fish_length {}; // Average length of fish

    cout << "Enter the number of fish you want to keep: ";
    cin >> fish_count;
    cout << "Enter the average fish length in inches: ";
    cin >> fish_length;
    fish_length /=inches_per_foot; // Convert to feet

    // Calculate the required surface area
    double pond_area {fish_count * fish_length * fish_factor};

    // Calculate the pond diameter from the area
    double pond_diameter {2.0 * sqrt(pond_area/pi)};

    cout << "\nPond diameter required for " << fish_count << " fish is "
         << pond_diameter << " feet.\n";
}
```

输入 20 条鱼, 每条鱼的平均长度为 9 英寸, 这个例子的输出如下所示:

```
Enter the number of fish you want to keep: 20
Enter the average fish length in inches: 9
Pond diameter required for 20 fish is 8.74039 feet.
```

3 个 using 声明允许使用没有名称空间名限定的流名和 sqrt() 函数名。首先在 main()



中定义要在计算中使用的 3 个 `const` 变量。注意使用一个常量表达式来指定 `fish_factor` 的初始值。可以使用任意表达式来生成合适类型的结果,以定义变量的初始值。这里把 `fish_factor`、`inches_per_foot` 和 `pi` 声明为 `const`, 因为它们的值是固定的, 不应修改。

接着定义存储用户输入的变量 `fish_count` 和 `fish_length`, 它们的初始值都是 0。

输入的鱼长度使用英寸作为单位, 所以需要把它转换为英尺, 再在鱼池的计算中使用它。使用 `/=` 运算符把初始值转换为英尺。

给池塘的面积定义一个变量, 初始化为一个表达式, 来计算需要的值:

```
double pond_area {fish_count * fish_length * fish_factor};
```

`fish_count` 和 `fish_length` 的乘积给出了所有鱼的总长, 把这个值与 `fish_factor` 相乘, 就得到了需要的面积(单位为平方英尺)。

圆的面积可以由公式  $\pi r^2$  得到, 其中  $r$  是半径。所以, 可以计算出池塘的半径, 即面积除以  $\pi$ , 再开方。直径是半径的两倍, 整个计算过程由下面的语句完成:

```
double pond_diameter {2.0 * sqrt(pond_area / pi)};
```

使用标准头文件 `<cmath>` 中声明的 `sqrt()` 函数可以获得平方根。

当然, 使用一条语句就可以计算出池塘的直径:

```
double pond_diameter {2.0 * sqrt(fish_count * fish_length * fish_factor / pi)};
```

这样就不需要 `pond_area` 变量了, 程序会更小、更短。这是否比最初的版本更好是有争议的, 因为其工作过程不是很明显。

`main()` 中的最后一条语句输出了结果。池塘直径的小数位比需要的多。下面看看如何修改它。

## 2.10 流输出的格式化

数据写入输出流时, 可以使用流操作程序改变数据的格式化方式。流操作程序是在标准库头文件 `iomanip` 中声明的函数, 可以应用于带有插入运算符 `<<` 的输出流。流操作程序要求提供一个参数值。在头文件 `iostream` 中, 把某些预定义的常量插入输出流, 可以影响数据的表示方式。这里仅介绍最有用的操作程序和流常量。

头文件 `iomanip` 提供了如表 2-9 所示的有用的参数化操作程序:

表 2-9 参数化操作程序

参数化操作程序	说 明
<code>std::setprecision(n)</code>	把浮点数的精度或小数位数设置为 <code>n</code> 。如果使用默认的浮点输出形式, <code>n</code> 就指定了输出值中的位数。如果设置了 <code>fixed</code> 或 <code>scientific</code> 格式, <code>n</code> 就是小数点后的位数。 <code>setprecision()</code> 设置的值对后续输出一直有效, 除非修改了它



(续表)

参数化操作程序	说 明
std::setw(n)	把输出的字段宽度设置为 n 个字符, 但仅适用于下一个输出的数据项。后续的输出会恢复默认形式, 其字段宽度设置为容纳数据所需的输出字符数
std::setfill(ch)	字段宽度比输出值的字符数大时, 字段中多余的字符是默认的填充字符, 即空格。该函数把所有后续输出的填充字符设置为 ch

iostream 头文件定义了如表 2-10 的流常量:

表 2-10 流常量

流 常 量	说 明
std::fixed	用小数点固定的格式输出浮点数据
std::scientific	以科学计数法输出所有后续的浮点数据, 它总是包含一个指数和小数点前面的一个数字
std::defaultfloat	恢复为默认的浮点数据表示方法
std::dec	所有后续的整数输出都是十进制
std::hex	所有后续的整数输出都是十六进制
std::oct	所有后续的整数输出都是八进制
std::showbase	给十六和八进制整数值输出基数前缀, 在流中插入 std::noshowbase 会关闭这个选项
std::left	输出在字段中左对齐
std::right	输出在字段中右对齐, 这是默认值
std::internal	填充字符在整数或浮点输出值的内部

在输出流中插入任意流常量, 它们就会一直起作用, 除非修改了它们。下面看看其中一些流常量的作用。考虑这个输出语句:

```
cout << "\nPond diameter required for " << fish_count << " fish is "
    << std::setprecision(2) // Output value is 8.7
    << pond_diameter << " feet.\n";
```

如果用这条语句替换 Ex2\_04.cpp 末尾的输出语句, 浮点数值就会有 2 位精度, 对应于这里的 1 位小数。在浮点输出的默认处理方式起作用之前, setprecision() 的圆括号中的整数指定了浮点数值输出精度, 即小数点前后的总位数。可以使用该参数, 把模式设置为 fixed, 指定小数点后的位数, 换言之, 就是小数位数。例如, 在 Ex2\_04.cpp 中使用如下语句:

```
cout << "\nPond diameter required for " << fish_count << " fish is "
    << std::fixed << std::setprecision(2)
    << pond_diameter << " feet.\n"; // Output value is 8.74
```

把模式设置为 fixed 或 scientific, 就把 setprecision() 的参数解释为输出值中的小数位



数。设置 scientific 模式，会使浮点数输出为科学计数法，它带有指数：

```
cout << "\nPond diameter required for " << fish_count << " fish is "
<< std::scientific << std::setprecision(2)
<< pond_diameter << " feet.\n";           // Output value is 8.74e+000
```

在科学计数法中，小数点前总是有一位。setprecision() 设置的值仍是小数点后的位数。即使指数是 0，指数值也总是有 3 位数字。

下面的语句演示了整数值的一些格式：

```
int a{16}, b{66};
cout << std::setw(5) << a << std::setw(5) << b << std::endl;
cout << std::left << std::setw(5) << a << std::setw(5) << b << std::endl;
cout << " a = " << std::setbase(16) << std::setw(6) << std::showbase << a
<< " b = " << std::setw(6) << b << std::endl;
cout << std::setw(10) << a << std::setw(10) << b << std::endl;
```

这些语句的输出如下：

```
   16   66
  6   66
a = 0x10  b = 0x42
x10   0x42
```

把整数输出为十六或八进制形式时，最好把 showbase 插入流中，这样输出就不会误解为十进制值。建议尝试这些操作程序和流常量的各种组合，了解它们的工作方式。

## 2.11 混合的表达式和类型转换

表达式可以包含不同类型的操作数。例如，可以定义变量来存储鱼的数量，如下：

```
unsigned int fish_count {};           // Number of fish
```

鱼数肯定是整数，所以这是有效的。一英尺的英寸数也是整数，所以可以定义如下变量：

```
const unsigned int inches_per_foot {12};
```

即使变量有不同的类型，计算也是可以完成的，例如：

```
fish_length /=inches_per_foot;       // Convert to feet
double pond_area {fish_count * fish_length * fish_factor};
```

二元算术运算要求两个操作数的类型相同。如果它们的类型不同，编译器就必须把其中一个操作数转换为另一个操作数的类型。这称为隐式转换。其工作方式是把值域较受限的变量类型转换为另一个变量的类型。第一条语句中的 fish\_length 变量是 double 类型。double 类型的值域比 unsigned int 类型大，所以编译器插入了一个转换，把 inches\_per\_foot 的值转换为 double 类型，才能执行除法操作。在第二个语句中，fish\_count



的值转换为 `double` 类型, 使它与 `fish_length` 有相同的类型, 之后执行乘法操作。

在操作数类型不同的操作中, 编译器选择把值域较受限的操作数类型转换为另一个操作数的类型。实际上, 它按如下顺序从高到低给类型排序:

1. long double	2. double	3. float
4. unsigned long long	5. long long	
6. unsigned long	7. long	
8. unsigned int	9. int	

要转换的操作数是位置较低的类型。因此在处理 `long long` 类型和 `unsigned int` 类型的操作中, 要把后者转换为 `long long`。`char`、`signed char`、`unsigned char`、`short` 或 `unsigned short` 类型的操作数总是至少转换为 `int` 类型。

隐式转换可能会得到预料不到的结果。考虑下面的语句:

```
unsigned int x {20u};
int y {30};
std::cout << x - y << std::endl;
```

本来希望输出是 -10, 但其实不是。输出是 4294967286。这是因为 `y` 的值转换为 `unsigned int`, 以匹配 `x` 的类型, 所以减法的结果是一个不带符号的整数值。

等号右边的表达式生成的值的类型不同于其左边的变量类型时, 编译器也会插入一个隐式转换, 例如:

```
int y {};
double z {5.0};
y = z; // Requires implicit conversion
```

最后一条语句需要转换等号右边的表达式值, 以便把它存储为 `int` 类型。编译器会插入一个转换操作, 但在大多数情况下, 还会发出一个可能丢失数据的警告消息。

编写操作数类型不同的整数操作时要小心。不要依靠隐式类型转换来生成希望的结果, 除非肯定可以得到该结果。如果不能肯定, 就需要进行显式类型转换也称为显式强制转换。

### 2.11.1 显式类型转换

要把表达式的值转换为给定的类型, 应编写如下格式的转换语句:

```
static_cast<type_to_convert_to>(表达式)
```

关键字 `static_cast` 表示这个强制转换要进行静态检查, 也就是说, 在程序编译时进行检查。后面在介绍类的处理时, 会遇到动态的强制转换, 这种转换要进行动态检查, 即在程序执行时进行检查。强制转换的结果是把从表达式中计算的数值转换为尖括号中指定的类型。表达式可以是任何内容, 包括从单个变量到包含许多嵌套括号的复杂表达式等所有内容。编写如下语句, 可以避免编译上一节的赋值语句时出现的警告:



```
y = static_cast<int>(z); // No compiler warning this time...
```

下面是使用 `static_cast<>()` 的另一个例子:

```
double value1 {10.5};  
double value2 {15.5};  
int whole_number {static_cast<int>(value1) + static_cast<int>(value2)};
```

因为变量 `whole_number` 的初始值是 `value1` 和 `value2` 的整数部分之和, 所以它们必须分别显式强制转换为 `int` 类型。变量 `whole_number` 的初始值应为 25。强制转换不会影响存储在 `value1` 和 `value2` 中的值, 它们仍然是 10.5 和 15.5。由强制转换得到的值 10 和 15 只是临时存储, 在计算中使用后就删除。这两个强制转换会在计算过程中丢失信息, 但编译器总是假定在显式指定强制转换时, 用户知道会发生什么。

当然, 如果编写如下语句, `whole_number` 的值就会不同:

```
int whole_number {static_cast<int>(value1 + value2)};
```

`value1` 和 `value2` 相加的结果是 26.0, 在转换为 `int` 类型时会得到 26。编译器不会给初始化列表中的值插入隐式的缩窄转换, 所以没有显式的类型转换, 该语句就不会编译。

一般情况下, 很少需要显式强制转换, 特别是在数据为基本类型时。如果必须在代码中包含大量的显式强制转换, 则通常表明应为变量选择更合适的类型。但仍有一些情况需要进行强制转换。下面就介绍一个简单的例子。这个例子把单位为码的长度(小数值)转换为码、英尺和英寸(整数值)。

```
// Ex2_05.cpp  
// Using Explicit Type  
#include <iostream>  
  
int main()  
{  
    const unsigned int feet_per_yard {3};  
    const unsigned int inches_per_foot {12};  
  
    double length {}; // Length as decimal yards  
    unsigned int yards{}; // Whole yards  
    unsigned int feet {}; // Whole feet  
    unsigned int inches {}; // Whole inches  
  
    std::cout << "Enter a length in yards as a decimal: ";  
    std::cin >> length;  
  
    // Get the length as yards, feet, and inches  
    yards = static_cast<unsigned int>(length);  
    feet = static_cast<unsigned int>((length - yards)*feet_per_yard);  
    inches = static_cast<unsigned int>  
    (length*feet_per_yard *inches_per_foot) % inches_per_foot;  
  
    std::cout << length << " yards converts to "
```



```
        << yards << " yards "
        << feet << " feet "
        << inches << " inches." << std::endl;
    }
```

这个程序的典型输出如下:

```
Enter a length in yards as a decimal: 2.75
2.75 yards converts to 2 yards 2 feet 3 inches.
```

`main()`中的前两条语句把转换常量 `feet_per_yard` 和 `inches_per_foot` 声明为整数。把这些变量声明为 `const`, 以防止程序不经意地修改它们。把输入转换为码、英尺和英寸, 其结果存储在 `unsigned int` 类型的变量中, 该变量初始化为 0。

下面的语句从输入值中计算整数码值:

```
yards = static_cast<unsigned int>(length);
```

强制转换会舍弃 `length` 值的小数部分, 把整数部分存储在 `yards` 中。如果这里省略了显式强制转换过程, 编译器就会插入需要的强制转换。但是, 在这种情况下, 应总是编写一个显式强制转换语句。如果忽略了这一步, 就不清楚这个转换的必要性如何, 也没有注意到潜在的数据丢失。

用下面的语句获得长度的英尺值:

```
feet = static_cast<unsigned int>((length - yards)*feet_per_yard);
```

从 `length` 中减去 `yards` 的值, 会把 `length` 中的小数码值自动转换为 `double` 类型。编译器会将 `yards` 的值转换为 `double` 类型, 以进行减法运算。再把 `feet_per_yard` 的值自动转换为 `double` 类型, 进行乘法运算, 最后对乘积进行显式的强制转换, 把它从 `double` 类型转换为 `unsigned int` 类型。

最后一部分计算是获得剩余长度的英寸值:

```
inches = static_cast<unsigned int>
(length*feet_per_yard *inches_per_foot) % inches_per_foot;
```

显式的强制转换应用于 `length` 中的总英寸值, 总英寸值是 `length`、`feet_per_yard` 和 `inches_per_foot` 的乘积。因为 `length` 是 `double` 类型, 两个常量值隐式转换为 `double` 类型, 以计算乘积。`length` 中的整数英寸值除以每英尺的英寸数, 得到了剩余的英寸值。

## 2.11.2 老式的强制转换

前面介绍了 C++ 中的 `static_cast<>()`, 把表达式的结果显式强制转换为另一种类型的过程可以表示为:

```
(type_to_convert_to)表达式
```

表达式的结果强制转换为括号中的类型。例如, 前面例子中计算 `inches` 的语句可以



改写为:

```
inches = (unsigned int)(length*feet_per_yard *inches_per_foot) %
inches_per_foot;
```

目前, C++中有几种不同类型的强制转换, 老式的强制转换语法包含了这几种转换。所以, 使用老式强制转换的代码更容易出错——它并不是很清晰, 可能得不到希望的结果。尽管老式强制转换语法目前仍使用得很广泛(它仍是语言的一部分), 但最好在代码中使用新型的强制转换语法。

## 2.12 确定数值的上下限

前面的示例列出了各种类型的上下限。标准头文件<limits>包含所有基本数据类型的上下限信息, 所以可以访问编译器的这些信息。下面举一个例子。要显示可以存储在double类型的变量中的最大值, 可以使用下面的语句:

```
std::cout << "Maximum value of type double is " <<
std::numeric_limits<double>::max();
```

表达式 `std::numeric_limits<double>::max()` 输出了我们希望的数值。把不同的类型名称放在尖括号中, 就可以得到其他数据类型的最大值。还可以用 `min()` 代替 `max()` 来获得最小值, 但整数和浮点数类型的最小值的含义是不同的。对于整数类型, `min()` 会得到真正最小值, 即带符号的整数类型的负数。对于浮点数类型, `min()` 会返回可以存储的最小正数。

可以检索出各种类型的许多其他信息。例如, 下面的表达式就返回二进制数字的位数:

```
std::numeric_limits<type_name>::digits
```

只需要把自己感兴趣的类型名插入到尖括号中即可。对于浮点数类型, 就会获得尾数中二进制数字的位数。对于带符号的整数类型, 就可以获得除符号位之外的二进制数字的位数。下面的程序显示了数值数据类型的最大值和最小值。

```
// Ex2_06.cpp
// Finding maximum and minimum values for data types
#include <limits>
#include <iostream>

int main()
{
    std::cout << "The range for type short is from "
              << std::numeric_limits<short>::min() << " to "
              << std::numeric_limits<short>::max() << std::endl;
    std::cout << "The range for type int is from "
              << std::numeric_limits<int>::min() << " to "
```



```

        << std::numeric_limits<int>::max() << std::endl;
std::cout << "The range for type long is from "
        << std::numeric_limits<long>::min() << " to "
        << std::numeric_limits<long>::max() << std::endl;
std::cout << "The range for type float is from "
        << std::numeric_limits<float>::min() << " to "
        << std::numeric_limits<float>::max() << std::endl;
std::cout << "The range for type double is from "
        << std::numeric_limits<double>::min() << " to "
        << std::numeric_limits<double>::max() << std::endl;
std::cout << "The range for type long double is from "
        << std::numeric_limits<long double>::min() << " to "
        << std::numeric_limits<long double>::max() << std::endl;
}

```

很容易扩展该程序，包含不带符号的整数类型和存储字符的类型。

## 2.13 使用字符变量

`char` 类型的变量主要用于存储单个字符的编码，占用 1 个字节的内存。C++ 标准没有指定用于表示基本字符集的字符编码，所以这由编译器指定。但一般使用 ASCII 编码。

`char` 类型的变量声明与其他类型的变量声明相同，如下所示：

```

char letter; // Uninitialized - so junk value
char yes {'Y'}, no {'N'}; // Initialized with character literals
char ch {33}; // Integer initializer equivalent to '!'

```

`char` 类型的变量可以用放在单引号中的字符字面量或整数初始化。整数初始化器必须位于 `char` 类型的值域内——这取决于在编译器上 `char` 是带符号还是不带符号的类型。当然，可以把字符指定为第 1 章介绍的某个转义序列。还有一些转义序列可以用于指定用八进制或十六进制值表示的字符编码。八进制字符编码的转义序列是一个反斜杠后跟 3 个八进制数。十六进制字符编码的转义序列是 `\x` 后跟一个或多个十六进制数。在定义字符字面量时，这两种形式都放在单引号中。例如，在 ASCII 编码中，字母 'A' 可以写为十六进制的 `\x41`，或八进制的 `\81`。显然，可以编写不能放在一个字节中的编码，此时结果是实现方式已定义好的内容。

`char` 类型的变量是数值。毕竟，它们存储了表示字符的整数代码，因此它们可以参与算术表达式，就像 `int` 或 `long` 类型的变量一样。例如：

```

char ch {'A'};
char letter {ch + 5}; // letter is 'F'
++ch; // ch is now 'B'
ch += 3; // ch is now 'E'

```

把 `char` 变量写入 `cout` 时，会输出一个字符，而不是整数。如果希望把它输出为一个数值，可以把它强制转换为另一个整数类型。例如：



```
std::cout << "ch is '" << ch
          << "' which is code " << std::hex << std::showbase
          << static_cast<int>(ch) << std::endl;
```

输出如下:

```
ch is 'E' which is code 0x45
```

从流中把数据读入 `char` 类型的变量时, 会存储第一个非空白字符。这意味着不能用这种方式读取空白字符, 而是简单地忽略空白。而且, 不能把数值读入 `char` 类型的变量, 否则, 只会存储第一个数字的字符代码。

### 2.13.1 使用 Unicode 字符

通常, ASCII 对使用拉丁字符的国家语言字符集来说足够了。但是, 如果要同时使用这些语言和拉丁字符, 或者要处理亚洲语言的字符集, 256 个字符编码就远远不够了, 需要使用 Unicode。

类型 `wchar_t` 是一种基本类型, 它可以存储实现方式支持的最大扩展字符集中的所有成员。这个类型名来自于宽字符(wide characters), 因为字符的范围比通常的单字节字符宽。`char` 类型则“比较窄”, 因为可用的字符编码比较有限。

定义宽字符字面量的方式与 `char` 类型的字面量相同, 但要在字面量的前面加上字母 `L`, 例如:

```
wchar_t wch {L'Z'};
```

此语句把变量 `wch` 定义为 `wchar_t` 类型, 并将其初始化为 `Z` 的宽字符表示。

键盘上可能没有表示其他国家语言字符的键, 但仍可以使用十六进制表示法来创建它们。例如:

```
wchar_t wch {L'\x0438'}; //Cyrillic N
```

单引号中的值是一个转义序列, 它指定了字符代码的十六进制表示。反斜杠表示转义序列的开始, 反斜杠之后的 `x` 或 `X` 表示该代码是十六进制的。

`wchar_t` 类型不能很好地处理国际字符, 使用 `char16_t` 类型会更好, 该类型把编码的字符存储为 UTF-16, 也可以使用 `char32_t` 类型, 它存储 UTF-32 编码字符。下面的示例定义了 `char16_t` 类型的变量:

```
char16_t letter {u'B'}; // Initialized with UTF-16 code for B
char16_t cyr {U'\x0438'}; // Initialized with UTF-16 code for cyrillic N
```

字面量的小写字母前缀 `u` 表示 UTF-16, UTF-32 用大写字母 `U` 作为前缀, 例如:

```
char32_t letter {U'B'}; // Initialized with UTF-32 code for B
char32_t cyr {U'\x044f'}; // Initialized with UTF-32 code for cyrillic я
```

当然, 如果编辑器可以接受并显示字符, 就可以把 `cyr` 定义为:



```
char32_t cyr {U'я'};
```

标准库提供了标准输入和输出流 `wcin` 和 `wcout`, 来读写用于 `wchar_t` 类型的字符, 但没有提供处理 `char16_t` 和 `char32_t` 字符数据的库。编译器可能有读写这些类型的工具。

#### 警告:

不要混合 `wcout` 和 `cout` 上的输出操作。两个流上的第一个输出操作都把标准输出流的方向设置为 `narrow` 或 `wide`, 这取决于操作是 `wcout` 还是 `cout`。对于 `wcout` 或 `cout`, 该方向都会应用于后续的输出操作。

### 2.13.2 auto 关键字

使用 `auto` 关键字可以告诉编译器应推断类型。下面是一些示例:

```
auto m = 10;           // m is type int
auto n = 200UL;       // n is type unsigned long
auto pi = 3.14159;    // pi is type double
```

注意, 这里使用 `=` 进行初始化。编译器会根据所提供的初始值推断 `m`、`n` 和 `pi` 的类型。如前所述, 这不是为 `auto` 关键字设计的用法。定义基本类型的变量时, 应显式指定类型, 所以肯定知道变量的类型是什么。本书后面会指出 `auto` 关键字更适合、更有用的场合。

#### 警告:

不应给 `auto` 关键字使用初始化列表, 因为类型是错误的。这是因为初始化列表本身有一个类型。例如, 假定编写如下代码:

```
auto m {10};
```

赋予 `m` 的类型不是 `int`, 而是 `std::initializer_list<int>`, 这是该初始化列表的类型。使用函数表示法和 `auto` 关键字可以指定初始值:

```
auto pi(3.14159); // pi is type double
```

这仍不是使用 `auto` 的方式, 只是把类型指定为 `double`, 并使用了初始化列表。

### 2.13.3 lvalue 和 rvalue

每个表达式都会得到 `lvalue` 或 `rvalue`。`lvalue` 引用了内存中的一个地址, 有时它可以在过程中存储。`rvalue` 是临时存储的结果。之所以称为 `lvalue`, 是因为得到 `lvalue` 的表达式出现在赋值运算符的左边。如果表达式的结果不是 `lvalue`, 它就是 `rvalue`。包含一个命名变量的表达式总是 `lvalue`。

考虑下面的语句:

```
int a {}, b {1}, c {2};
```



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

第一条语句把 `a`、`b` 和 `c` 定义为 `int`，分别初始化为 0、1 和 2。在第二条语句中，`b+c` 的计算结果会临时存储，其值会复制到 `a` 中。执行完该语句后，就舍弃保存了 `b+c` 结果的内存。因此 `b+c` 的计算结果是 `rvalue`。

在第三条语句中，`++a` 表达式是一个 `lvalue`，因为它的结果是 `a` 递增后的值。在第四条语句中，`a++` 表达式是一个 `rvalue`，因为它把 `a` 的值临时存储为表达式的结果，再递增 `a`。

#### 注意：

不可能总是知道表达式是 `lvalue` 还是 `rvalue`。在大多数情况下，不需要考虑表达式是 `lvalue` 还是 `rvalue`，但有时需要考虑。在本书后面学完了类后，就会明白它们的区别何时很重要了。

## 2.14 本章小结

本章介绍了 C++ 中计算的基础知识，学习了该语言提供的大多数基本数据类型，本章的主要内容如下：

- 任何类型的常量都称为字面量，字面量有自己的类型。
- 可以把整数字面量定义为十进制、十六进制、二进制或八进制。
- 浮点字面量必须包含小数点或指数，或两者都包含。如果两者都不包含，它就是一个整数。
- 可以存储整数的基本类型有 `short`、`int`、`long` 和 `long long`。它们存储带符号的整数，也可以在这些类型名称的前面使用类型修饰符 `unsigned`，使该类型占用相同的字节数，但只存储不带符号的整数。
- 浮点数的数据类型有 `float`、`double` 和 `long double`。
- 变量在声明时可以指定初始值，这是一种很好的编程习惯。初始化列表是指定初始值的首选方法。
- `char` 类型的变量可以存储单个字符，占用 1 个字节。`char` 类型可以是带符号的，也可以是不带符号的，这取决于编译器。也可以使用 `signed char` 和 `unsigned char` 类型的变量存储整数。`char`、`signed char` 和 `unsigned char` 是不同的类型。
- 类型 `wchar_t` 可以存储宽字符，占用 2 或 4 个字节，这取决于编译器。类型 `char16_t` 和 `char32_t` 更适合于处理 Unicode 字符。
- 可以用修饰符 `const` 固定变量的值。编译器会在程序源代码文件中检查是否试图修改声明为 `const` 的变量。
- 可以在一个表达式中混合不同类型的变量和常量。操作数的类型不同时，编译器会将二元操作中的一个操作数自动转换为另一个操作数的类型。



- 当等号右边的类型与等号左边的类型不同时,编译器会将表达式结果的类型自动转换为等号左边的类型。当左边的类型不能完全包含与右边类型的信息相同的信息时,就可能丢失信息。例如把 `double` 转换为 `int` 或把 `long` 转换为 `short`。
- 使用 `static_cast<>()` 操作符,可以把一种类型的值显式转换为另一种类型。
- `lvalue` 是出现在等号左边的一个对象或表达式,非 `const` 的变量就是 `lvalue`。`rvalue` 是表达式的临时结果。

## 2.15 练习

1. 编写一个程序,计算圆的面积。该程序应提示从键盘上输入圆的半径,使用公式  $\text{area}=\pi*\text{radius}*\text{radius}$  计算面积,再显示结果。

2. 使用第1题的解决方案,改进代码,使用户可以输入所需的位数,控制输出结果的精度(提示:使用 `setprecision()` 操作程序)。

3. 创建一个程序,把英寸转换为英尺和英寸。例如,输入 77 英寸,程序就应生成 6 英尺 5 英寸的结果。提示用户输入一个单位是英寸的整数值,再进行转换,输出结果(提示:使用 `const` 存储 `inches-to-feet` 转换率,并使用取模运算符)。

4. 在生日那天,你得到了一个卷尺和一个可以确定角度的仪器,例如测量水平线和树高之间的夹角。如果知道自己与树之间的距离 `d` 和眼睛平视量角器的高度 `h`,就可以用公式  $h+d*\tan(\text{angle})$  计算出树的高度。创建一个程序,从键盘上输入 `h`(单位是英寸)、`d`(单位是英尺和英寸)和 `angle`(单位是度),输出树的高度(单位是英尺)。

不必砍树就可以验证程序的准确性。在 `Apress` 网站上检查结果即可。

5. 这个题较难。编写一个程序,提示用户输入两个不同的正整数,在输出中指出哪个较大,哪个较小(这可以用本章学习的内容来完成)。

6. 身体质量指数(Body Mass Index, BMI)是体重`w`(千克)除以身高`h`(米)的平方( $w/(h*h)$ )。编写一个程序,输入体重(磅)和身高(英尺和英寸),来计算BMI。1千克=2.2磅,1英尺=0.3048米。