

SAMS

畅销全球的  
经典C++教程

Sams Teach Yourself C++  
in One Hour a Day  
Seventh Edition



全面涵盖C++11新标准  
帮助读者编写高效的C++应用程序

# 21天学通 C++ (第7版)

[美] Siddhartha Rao 著  
袁国忠 译

 人民邮电出版社  
POSTS & TELECOM PRESS

---

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ1779903665.

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我QQ1779903665。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

**声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。**

对没有任何编程经验的新人和有其他语言编程经验的人来说，这是一本卓越的C++入门图书。

—— 独立评论人

## 21天学通C++ (第7版)

### 倾听来自读者的真实评论，选择最合适的C++教程

本书非常适合初学者，内容很详实，对一些比较抽象的概念会做一些比喻，书中对一些容易弄错或者不容易注意到的问题都特别提出来。书中更为重要的思想是，它不断提醒哪些习惯是优秀的程序员应该有的，哪些习惯容易出问题。总之，这是非常经典的一本书。

—— 当当读者“行动如虫”

C++的经典图书很多，但是这一本书非常有特色……以我的亲身经历，对那些没有任何编程经验的人来说，这本书真的不错。

—— 中国亚马逊读者“后皇嘉树”

这本书完全面向零基础的C++初学者，示例代码很多，讲的也很详细，内容也很全面。

—— 当当读者“风般飘过”

内容浅显易懂，例子简单明了……实在不失为一本好书。

—— 当当读者“小白大帝”

我见过的最好的C++教材，内容有趣，通俗易懂。

—— 中国亚马逊读者“星涛晦瞑”



读者可通过 [www.ptpress.com.cn](http://www.ptpress.com.cn) 或 <http://vdisk.weibo.com/s/gvYGx> 下载本书的所有源代码。



人民邮电出版社-信息技术分社  
<http://weibo.com/ptpitbooks>

美术编辑：王建国

分类建议：计算机 / 程序设计 / C++

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)



ISBN 978-7-115-29624-5



9 787115 296245 >

ISBN 978-7-115-29624-5

定价：59.00 元

# 21天学通 C++ (第7版)

[美] Siddhartha Rao 著  
袁国忠 译



## 图书在版编目 (C I P) 数据

21天学通C++ : 第7版 / (美) 罗奥 (Rao, S.) 著 ;  
袁国忠译. — 北京 : 人民邮电出版社, 2012. 12  
ISBN 978-7-115-29624-5

I. ①2… II. ①罗… ②袁… III. ①C语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字 (2012) 第235664号

## 版权声明

Siddhartha Rao: Sams Teach Yourself C++ in One Hour a Day (7th Edition)

ISBN: 0672335670

Copyright © 2012 by Pearson Education, Inc.

Authorized translation from the English language edition published by Sams.

All rights reserved.

本书中文简体字版由美国 Pearson 公司授权人民邮电出版社出版。未经出版者书面许可, 对本书任何部分不得以任何方式复制或抄袭。

版权所有, 侵权必究。

## 21 天学通 C++ (第 7 版)

- 
- ◆ 著 [美] Siddhartha Rao
  - 译 袁国忠
  - 责任编辑 傅道坤
  - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号  
邮编 100061 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
北京昌平百善印刷厂印刷
  - ◆ 开本: 787×1092 1/16  
印张: 29.5  
字数: 870 千字 2012 年 12 月第 1 版  
印数: 1-3 500 册 2012 年 12 月北京第 1 次印刷

著作权合同登记号 图字: 01-2012-5025 号

ISBN 978-7-115-29624-5

定价: 59.00 元

读者服务热线: (010)67132692 印装质量热线: (010)67129223

反盗版热线: (010)67171154

广告经营许可证: 京崇工商广字第 0021 号

# 内容提要

本书通过大量短小精悍的程序，详细而全面地阐述了 C++ 基本概念和技术以及 C++11 新增的功能，包括管理输入/输出、循环和数组、面向对象编程、模板、使用标准模板库以及 `lambda` 表达式等。这些内容被组织成结构合理、联系紧密的章节，每章都可在 1 小时内阅读完毕；每章都提供了示例程序清单，并辅以示例输出和代码分析，以阐述该章介绍的主题。为加深读者对所学内容的理解，每章末尾都提供了常见问题及其答案以及练习和测验。读者可对照附录 D 提供的测验和练习答案，了解自己对所学内容的掌握程度。

本书是针对 C++ 初学者编写的，不要求读者有 C 语言方面的背景知识，可作为高等院校教授 C++ 课程的教材，也可供初学者自学 C++ 时使用。

# 作者简介

Siddhartha Rao 是全球领先的企业软件提供商 SAP AG 的技术专家。作为 SAP Product Security India 的负责人，其主要职责包括招募产品安全领域的专家以及制定软件开发最佳实践，以保持 SAP 软件的全球竞争力。作为一位 Microsoft Visual C++ MVP，他深信 C++11 有助于编写速度更快、更简洁、更高效的 C++ 应用程序。

Siddhartha 酷爱旅游，不放过任何一次探索新文化的机会。例如，本书就是在 4 个不同的国度创作而成的，其中包括法国布列塔尼一个面朝大西洋的奇特村庄。他期待着您这部全球之作提出宝贵的建议。

# 献词

谨将本书献给我的父母和我的妹妹，他们是我坚强的后盾。

# 致谢

当我为了编写本书而通宵达旦地忙碌时，是我的朋友承担起了我的后勤工作，对此深表谢意。谢谢本书的所有编辑人员，正是他们的勤劳付出，才让本书出现在各位读者的书架上。

# 前 言

对 C++ 来说，2011 是个很特别的年份。在这一年，C++11 终于获批成为新标准，它新增了一些可提高编程效率的关键字和结构，让您能够编写更优质的代码。本书旨在帮助您循序渐进地学习 C++11，其中的章节经过仔细编排，从实用的角度介绍这种面向对象的编程语言的基本知识。读者只需每天花 1 小时，在学完本书后，就能掌握 C++11。

学习 C++ 的最佳方式是动手实践。本书包含丰富的代码示例，有助于读者提高编程技能，请务必亲自动手尝试这些代码。这些代码片段都使用了（在本书编写时）最新版本的编译器进行了测试，具体地说是 Microsoft Visual C++ 2010 和 GNU C++ 编译器 4.6 版，它们都支持大量的 C++11 功能。

## 针对的读者

本书从最基本的 C++ 知识开始介绍，读者只需具备学习 C++ 的愿望及了解工作原理的好奇心即可；虽然具备一些 C++ 知识会有所帮助，但这并非必需的。本书也可供熟悉 C++ 但想了解 C++11 新增功能的读者参考；如果你是专业程序员，第 3 部分“学习标准模板库”可以帮助你创建更优质、更实用的 C++ 应用程序。

## 本书内容

读者可根据自己对 C++ 的熟练程度，阅读感兴趣的部分。本书包含 5 部分。

- 第 1 部分“基本知识”，引导读者编写一些简单的 C++ 应用程序，并介绍一些在 C++ 的未妥协类型安全变量的代码中最常见的关键字。
- 第 2 部分“C++ 面向对象编程基础”，介绍类的概念，您将学习 C++ 如何支持封装、抽象、继承和多态等重要的面向对象编程原则。第 9 章将介绍 C++11 新增的移动构造函数，而第 12 章将介绍移动赋值运算符。这些功能有助于避免不必要的复制步骤，从而提升应用程序的性能。第 14 章是一个跳板，助您编写功能强大的 C++ 通用代码。
- 第 3 部分“学习标准模板库”，将帮助您使用 STL `string` 类和容器编写高效而实用的 C++ 代码。您将了解到，使用 `std::string` 可安全而轻松地拼接字符串，您不再需要使用 C 风格字符串（`char*`）。您可使用 STL 动态数组和链表，而无需自己编写这样的类。
- 第 4 部分“再谈 STL”，专注于算法，您将学习如何通过迭代器对 `vector` 等容器进行排序。在这部分，您将发现，通过使用 C++11 新增的关键字 `auto`，可极大地简化冗长的迭代器声明。第 22 章将介绍 C++11 新增的 `lambda` 表达式，这可极大地简化使用 STL 算法的代码。
- 第 5 部分“高级 C++ 概念”，阐述智能指针和异常处理等 C++ 功能。对 C++ 应用程序来说，这些功能并非必需的，但可极大地提高应用程序的稳定性和品质。在这部分的最后，简要地介绍了有助于编写杰出 C++ 应用程序的最佳实践。



# 本书体例

本书使用了下述提供更多信息的元素：

注意

提供与读者阅读的内容相关的信息。

## C++11

突出 C++11 新增的功能。要使用这些功能，可能需要使用较新的编译器版本。

警告

提醒读者注意在特定情况下可能出现的问题或副作用。

提示

提供 C++编程最佳实践。

应该	不应该
提供当前章介绍的基本原理的摘要。	提供一些有用的信息。

本书使用不同的字体来区分代码和正文，全书都用特殊字体呈现代码、命令以及与编程相关的术语。

# 目 录

第1章 绪论	1	3.1.2 声明变量以访问和使用内存	15
1.1 C++简史	1	3.1.3 声明并初始化多个类型相同的变量	17
1.1.1 与C语言的关系	1	3.1.4 理解变量的作用域	17
1.1.2 C++的优点	1	3.1.5 全局变量	18
1.1.3 C++标准的发展历程	1	3.2 编译器支持的常见C++变量类型	19
1.1.4 哪些人使用C++程序	2	3.2.1 使用bool变量存储布尔值	20
1.2 编写C++应用程序	2	3.2.2 使用char变量存储字符	20
1.2.1 生成可执行文件的步骤	2	3.2.3 有符号整数和无符号整数的概念	20
1.2.2 分析并修复错误	2	3.2.4 有符号整型short、int、long和long long	21
1.2.3 集成开发环境	2	3.2.5 无符号整型unsigned short、unsigned int、unsigned long和unsigned long long	21
1.2.4 编写第一个C++应用程序	3	3.2.6 浮点类型float和double	21
1.2.5 生成并执行第一个C++应用程序	4	3.3 使用sizeof确定变量的长度	22
1.2.6 理解编译错误	4	3.4 使用typedef替换变量类型	24
1.3 C++11新增的功能	5	3.5 什么是常量	24
1.4 总结	5	3.5.1 字面常量	25
1.5 问与答	5	3.5.2 使用const将变量声明为常量	25
1.6 作业	6	3.5.3 使用constexpr声明常量	26
1.6.1 测验	6	3.5.4 枚举常量	26
1.6.2 练习	6	3.5.5 使用#define定义常量	27
第2章 C++程序的组成部分	7	3.6 给变量和常量命名	28
2.1 Hello World程序的组成部分	7	3.7 不能用作常量或变量名的关键字	28
2.1.1 预处理器编译指令#include	7	3.8 总结	29
2.1.2 程序的主体——main()	8	3.9 问与答	29
2.1.3 返回值	8	3.10 作业	30
2.2 名称空间的概念	9	3.10.1 测验	30
2.3 C++代码中的注释	10	3.10.2 练习	30
2.4 C++函数	10	第4章 管理数组和字符串	31
2.5 使用std::cin和std::cout执行基本输入输出操作	12	4.1 什么是数组	31
2.6 总结	13	4.1.1 为何需要数组	31
2.7 问与答	13	4.1.2 声明和初始化静态数组	32
2.8 作业	13	4.1.3 数组中的数据是如何存储的	32
2.8.1 测验	14	4.1.4 访问存储在数组中的数据	33
2.8.2 练习	14	4.1.5 修改存储在数组中的数据	34
第3章 使用变量和常量	15	4.2 多维数组	35
3.1 什么是变量	15	4.2.1 声明和初始化多维数组	36
3.1.1 内存和寻址概述	15		

4.2.2 访问多维数组中的元素 .....	36	6.1.2 有条件地执行多条语句 .....	62
4.3 动态数组 .....	37	6.1.3 嵌套 if 语句 .....	63
4.4 C 风格字符串 .....	38	6.1.4 使用 switch-case 进行条件处理 .....	66
4.5 C++ 字符串: 使用 std::string .....	40	6.1.5 使用运算符?: 进行条件处理 .....	68
4.6 总结 .....	41	6.2 在循环中执行代码 .....	69
4.7 问与答 .....	41	6.2.1 不成熟的 goto 循环 .....	69
4.8 作业 .....	42	6.2.2 while 循环 .....	70
4.8.1 测验 .....	42	6.2.3 do...while 循环 .....	72
4.8.2 练习 .....	42	6.2.4 for 循环 .....	73
第 5 章 使用表达式、语句和运算符 .....	43	6.3 使用 continue 和 break 修改循环的行为 .....	75
5.1 语句 .....	43	6.3.1 不结束的循环——无限循环 .....	75
5.2 复合语句 (语句块) .....	44	6.3.2 控制无限循环 .....	76
5.3 使用运算符 .....	44	6.4 编写嵌套循环 .....	78
5.3.1 赋值运算符 (=) .....	44	6.4.1 使用嵌套循环遍历多维数组 .....	79
5.3.2 理解左值和右值 .....	44	6.4.2 使用嵌套循环计算斐波纳契数列 .....	80
5.3.3 加法运算符 (+)、减法运算符 (-)、 乘法运算符 (*)、除法运算符 (/) 和 求模运算符 (%) .....	44	6.5 总结 .....	81
5.3.4 递增运算符 (++) 和递减 运算符 (--) .....	45	6.6 问与答 .....	81
5.3.5 前缀还是后缀 .....	45	6.7 作业 .....	82
5.3.6 相等运算符 (==) 和不等 运算符 (!=) .....	47	6.7.1 测验 .....	82
5.3.7 关系运算符 .....	48	6.7.2 练习 .....	82
5.3.8 逻辑运算 NOT、AND、OR 和 XOR .....	49	第 7 章 使用函数组织代码 .....	83
5.3.9 使用 C++ 逻辑运算 NOT (!)、 AND (&&) 和 OR (  ) .....	50	7.1 为何需要函数 .....	83
5.3.10 按位运算符 NOT (~)、 AND (&)、OR ( ) 和 XOR (^) .....	53	7.1.1 函数原型是什么 .....	84
5.3.11 按位右移运算符 (>>) 和 左移运算符 (<<) .....	54	7.1.2 函数定义是什么 .....	85
5.3.12 复合赋值运算符 .....	55	7.1.3 函数调用和实参是什么 .....	85
5.3.13 使用运算符 sizeof 确定变量占用的 内存量 .....	56	7.1.4 编写接受多个参数的函数 .....	85
5.3.14 运算符优先级 .....	57	7.1.5 编写没有参数和返回值的函数 .....	86
5.4 总结 .....	58	7.1.6 带默认值的函数参数 .....	87
5.5 问与答 .....	59	7.1.7 递归函数——调用自己的函数 .....	88
5.6 作业 .....	59	7.1.8 包含多条 return 语句的函数 .....	89
5.6.1 测验 .....	59	7.2 使用函数处理不同类型的数据 .....	90
5.6.2 练习 .....	59	7.2.1 函数重载 .....	90
第 6 章 控制程序流程 .....	60	7.2.2 将数组传递给函数 .....	92
6.1 使用 if...else 有条件地执行 .....	60	7.2.3 按引用传递参数 .....	93
6.1.1 使用 if...else 进行条件编程 .....	61	7.3 微处理器如何处理函数调用 .....	94
		7.3.1 内联函数 .....	94
		7.3.2 lambda 函数 .....	96
		7.4 总结 .....	97
		7.5 问与答 .....	97
		7.6 作业 .....	97
		7.6.1 测验 .....	97
		7.6.2 练习 .....	98
		第 8 章 阐述指针和引用 .....	99
		8.1 什么是指针 .....	99

8.1.1 声明指针 .....	99	9.3.6 包含初始化列表的构造函数 .....	133
8.1.2 使用引用运算符 (&) 获取变量的 地址 .....	100	9.4 析构函数 .....	135
8.1.3 使用指针存储地址 .....	101	9.4.1 声明和实现析构函数 .....	135
8.1.4 使用解除引用运算符 (*) 访问指向的数据 .....	102	9.4.2 何时及如何使用析构函数 .....	135
8.1.5 将 sizeof() 用于指针的结果 .....	104	9.5 复制构造函数 .....	137
8.2 动态内存分配 .....	105	9.5.1 浅复制及其存在的问题 .....	137
8.2.1 使用 new 和 delete 动态地分配和 释放内存 .....	105	9.5.2 使用复制构造函数确保深复制 .....	139
8.2.2 将递增和递减运算符 (++和--) 用于指针的结果 .....	107	9.5.3 有助于改善性能的移动构造函数 .....	142
8.2.3 将关键字 const 用于指针 .....	109	9.6 构造函数和析构函数的其他用途 .....	143
8.2.4 将指针传递给函数 .....	109	9.6.1 不允许复制的类 .....	143
8.2.5 数组和指针的类似之处 .....	110	9.6.2 只能有一个实例的单例类 .....	144
8.3 使用指针时常犯的编程错误 .....	112	9.6.3 禁止在栈中实例化的类 .....	146
8.3.1 内存泄露 .....	112	9.7 this 指针 .....	147
8.3.2 指针指向无效的内存单元 .....	112	9.8 将 sizeof() 用于类 .....	148
8.3.3 悬浮指针 (也叫迷途或失控指针) .....	113	9.9 结构不同于类的地方 .....	150
8.4 指针编程最佳实践 .....	114	9.10 声明友元 .....	150
8.4.1 检查使用 new 发出的分配请求 是否得到满足 .....	115	9.11 总结 .....	152
8.5 引用是什么 .....	117	9.12 问与答 .....	152
8.5.1 是什么让引用很有用 .....	117	9.13 作业 .....	153
8.5.2 将关键字 const 用于引用 .....	118	9.13.1 测验 .....	153
8.5.3 按引用向函数传递参数 .....	119	9.13.2 练习 .....	153
8.6 总结 .....	119	第 10 章 实现继承 .....	154
8.7 问与答 .....	120	10.1 继承基础 .....	154
8.8 作业 .....	121	10.1.1 继承和派生 .....	154
8.8.1 测验 .....	121	10.1.2 C++ 派生语法 .....	155
8.8.2 练习 .....	121	10.1.3 访问限定符 protected .....	157
第 9 章 类和对象 .....	122	10.1.4 基类初始化——向基类传递参数 .....	159
9.1 类和对象 .....	122	10.1.5 在派生类中覆盖基类的方法 .....	160
9.1.1 声明类 .....	123	10.1.6 调用基类中被覆盖的方法 .....	162
9.1.2 实例化对象 .....	123	10.1.7 在派生类中调用基类的方法 .....	162
9.1.3 使用句点运算符访问成员 .....	123	10.1.8 在派生类中隐藏基类的方法 .....	164
9.1.4 使用指针运算符 (->) 访问成员 .....	124	10.1.9 构造顺序 .....	165
9.2 关键字 public 和 private .....	125	10.1.10 析构顺序 .....	166
9.2.1 使用关键字 private 实现数据抽象 .....	126	10.2 私有继承 .....	167
9.3 构造函数 .....	127	10.3 保护继承 .....	169
9.3.1 声明和实现构造函数 .....	128	10.4 切除问题 .....	171
9.3.2 何时及如何使用构造函数 .....	128	10.5 多继承 .....	171
9.3.3 重载构造函数 .....	130	10.6 总结 .....	174
9.3.4 没有默认构造函数的类 .....	131	10.7 问与答 .....	174
9.3.5 带默认值的构造函数参数 .....	133	10.8 作业 .....	174
		10.8.1 测验 .....	174
		10.8.2 练习 .....	174
		第 11 章 多态 .....	176
		11.1 多态基础 .....	176

11.1.1 为何需要多态行为	176	类型识别	223
11.1.2 使用虚函数实现多态行为	177	13.3.3 使用 <code>reinterpret_cast</code>	225
11.1.3 为何需要虚构造函数	179	13.3.4 使用 <code>const_cast</code>	226
11.1.4 虚函数的工作原理——理解 虚函数表	182	13.4 C++类型转换运算符存在的问题	226
11.1.5 抽象基类和纯虚函数	184	13.5 总结	227
11.2 使用虚继承解决菱形问题	186	13.6 问与答	227
11.3 可将复制构造函数声明为虚函数吗	189	13.7 作业	228
11.4 总结	191	13.7.1 测验	228
11.5 问与答	192	13.7.2 练习	228
11.6 作业	192	第 14 章 宏和模板简介	229
11.6.1 测验	192	14.1 预处理器与编译器	229
11.6.2 练习	193	14.2 使用 <code>#define</code> 定义常量	229
第 12 章 运算符类型与运算符重载	194	14.3 使用 <code>#define</code> 编写宏函数	232
12.1 C++运算符	194	14.3.1 为什么要使用括号	233
12.2 单目运算符	195	14.3.2 使用 <code>assert</code> 宏验证表达式	234
12.2.1 单目运算符的类型	195	14.3.3 使用宏函数的优点和缺点	235
12.2.2 单目递增与单目递减运算符	195	14.4 模板简介	235
12.2.3 转换运算符	198	14.4.1 模板声明语法	235
12.2.4 解除引用运算符 ( <code>*</code> ) 和成员 选择运算符 ( <code>-&gt;</code> )	199	14.4.2 各种类型的模板声明	236
12.3 双目运算符	202	14.4.3 模板函数	236
12.3.1 双目运算符的类型	202	14.4.4 模板与类型安全	238
12.3.2 双目加法与双目减法运算符	202	14.4.5 模板类	238
12.3.3 实现运算符 <code>+=</code> 与 <code>-=</code>	204	14.4.6 模板的实例化和具体化	239
12.3.4 重载等于运算符 ( <code>==</code> ) 和不等 运算符 ( <code>!=</code> )	206	14.4.7 声明包含多个参数的模板	239
12.3.5 重载运算符 <code>&lt;</code> 、 <code>&gt;</code> 、 <code>&lt;=</code> 和 <code>&gt;=</code>	207	14.4.8 声明包含默认参数的模板	240
12.3.6 重载复制赋值运算符 ( <code>=</code> )	209	14.4.9 一个模板示例	240
12.3.7 下标运算符	211	14.4.10 模板类和静态成员	241
12.4 函数运算符 <code>operator()</code>	214	14.4.11 在实际 C++ 编程中使用模板	243
12.5 不能重载的运算符	219	14.5 总结	243
12.6 总结	219	14.6 问与答	244
12.7 问与答	220	14.7 作业	244
12.8 作业	220	14.7.1 测验	244
12.8.1 测验	220	14.7.2 练习	244
12.8.2 练习	220	第 15 章 标准模板库简介	245
第 13 章 类型转换运算符	221	15.1 STL 容器	245
13.1 为何需要类型转换	221	15.1.1 顺序容器	245
13.2 为何有些 C++ 程序员不喜欢 C 风格类型转换	222	15.1.2 关联容器	246
13.3 C++ 类型转换运算符	222	15.1.3 选择正确的容器	246
13.3.1 使用 <code>static_cast</code>	222	15.1.4 容器适配器	247
13.3.2 使用 <code>dynamic_cast</code> 和运行阶段		15.2 STL 迭代器	247
		15.3 STL 算法	248
		15.4 使用迭代器在容器和算法之间交互	248
		15.5 STL 字符串类	250
		15.6 总结	250

15.7 问与答 .....	250	18.3 对 list 中的元素进行反转和排序 .....	283
15.8 作业 .....	251	18.3.1 使用 list::reverse() 反转元素的 排列顺序 .....	283
第 16 章 STL string 类 .....	252	18.3.2 对元素进行排序 .....	284
16.1 为何需要字符串操作类 .....	252	18.3.3 对包含对象的 list 进行排序 以及删除其中的元素 .....	286
16.2 使用 STL string 类 .....	253	18.4 总结 .....	290
16.2.1 实例化和复制 STL string .....	253	18.5 问与答 .....	290
16.2.2 访问 std::string 的字符内容 .....	254	18.6 作业 .....	291
16.2.3 拼接字符串 .....	256	18.6.1 测验 .....	291
16.2.4 在 string 中查找字符或子字符串 .....	257	18.6.2 练习 .....	291
16.2.5 截短 STL string .....	259	第 19 章 STL 集合类 .....	292
16.2.6 字符串反转 .....	260	19.1 简介 .....	292
16.2.7 字符串的大小写转换 .....	261	19.2 STL set 和 multiset 的基本操作 .....	293
16.3 基于模板的 STL string 实现 .....	262	19.2.1 实例化 std::set 对象 .....	293
16.4 总结 .....	262	19.2.2 在 set 或 multiset 中插入元素 .....	294
16.5 问与答 .....	262	19.2.3 在 STL set 或 multiset 中 查找元素 .....	296
16.6 作业 .....	263	19.2.4 删除 STL set 或 multiset 中的 元素 .....	297
16.6.1 测验 .....	263	19.3 使用 STL set 和 multiset 的优缺点 .....	300
16.6.2 练习 .....	263	19.4 总结 .....	303
第 17 章 STL 动态数组类 .....	264	19.5 问与答 .....	303
17.1 std::vector 的特点 .....	264	19.6 作业 .....	304
17.2 典型的 vector 操作 .....	264	19.6.1 测验 .....	304
17.2.1 实例化 vector .....	264	19.6.2 练习 .....	304
17.2.2 使用 push_back() 在末尾插入元素 .....	266	第 20 章 STL 映射类 .....	305
17.2.3 使用 insert() 在指定位置插入元素 .....	267	20.1 STL 映射类简介 .....	305
17.2.4 使用数组语法访问 vector 中的 元素 .....	269	20.2 std::map 和 std::multimap 的基本操作 .....	306
17.2.5 使用指针语法访问 vector 中的 元素 .....	270	20.2.1 实例化 std::map 和 std::multimap .....	306
17.2.6 删除 vector 中的元素 .....	271	20.2.2 在 STL map 或 multimap 中 插入元素 .....	307
17.3 理解大小和容量 .....	272	20.2.3 在 STL map 或 multimap 中 查找元素 .....	309
17.4 STL deque 类 .....	273	20.2.4 在 STL multimap 中查找元素 .....	311
17.5 总结 .....	275	20.2.5 删除 STL map 或 multimap 中的 元素 .....	312
17.6 问与答 .....	275	20.3 提供自定义的排序谓词 .....	313
17.7 作业 .....	276	20.3.1 散列表的工作原理 .....	316
17.7.1 测验 .....	276	20.3.2 使用 C++11 散列表 unordered_map 和 unordered_multimap .....	316
17.7.2 练习 .....	276	20.4 总结 .....	319
第 18 章 STL list 和 forward_list .....	277	20.5 问与答 .....	319
18.1 std::list 的特点 .....	277	20.6 作业 .....	320
18.2 基本的 list 操作 .....	277		
18.2.1 实例化 std::list 对象 .....	277		
18.2.2 在 list 开头或末尾插入元素 .....	279		
18.2.3 在 list 中间插入元素 .....	280		
18.2.4 删除 list 中的元素 .....	282		

20.6.1 测验 .....	320	23.3.6 使用 <code>for_each()</code> 处理指定范围内的元素 .....	350
20.6.2 练习 .....	320	23.3.7 使用 <code>std::transform()</code> 对范围进行变换 .....	352
第 21 章 理解函数对象 .....	321	23.3.8 复制和删除操作 .....	354
21.1 函数对象与谓词的概念 .....	321	23.3.9 替换值以及替换满足给定条件的元素 .....	356
21.2 函数对象的典型用途 .....	321	23.3.10 排序、在有序集合中搜索以及删除重复元素 .....	357
21.2.1 一元函数 .....	321	23.3.11 将范围分区 .....	359
21.2.2 一元谓词 .....	325	23.3.12 在有序集合中插入元素 .....	360
21.2.3 二元函数 .....	326	23.4 总结 .....	362
21.2.4 二元谓词 .....	328	23.5 问与答 .....	362
21.3 总结 .....	330	23.6 作业 .....	363
21.4 问与答 .....	330	23.6.1 测验 .....	363
21.5 作业 .....	330	23.6.2 练习 .....	363
21.5.1 测验 .....	330	第 24 章 自适应容器：栈和队列 .....	364
21.5.2 练习 .....	330	24.1 栈和队列的行为特征 .....	364
第 22 章 C++ lambda 表达式 .....	331	24.1.1 栈 .....	364
22.1 lambda 表达式是什么 .....	331	24.1.2 队列 .....	365
22.2 如何定义 lambda 表达式 .....	332	24.2 使用 STL stack 类 .....	365
22.3 一元函数对应的 lambda 表达式 .....	332	24.2.1 实例化 stack .....	365
22.4 一元谓词对应的 lambda 表达式 .....	333	24.2.2 stack 的成员函数 .....	366
22.5 通过捕获列表接受状态变量的 lambda 表达式 .....	334	24.2.3 使用 <code>push()</code> 和 <code>pop()</code> 在栈顶插入和删除元素 .....	366
22.6 lambda 表达式的通用语法 .....	335	24.3 使用 STL queue 类 .....	367
22.7 二元函数对应的 lambda 表达式 .....	336	24.3.1 实例化 queue .....	368
22.8 二元谓词对应的 lambda 表达式 .....	337	24.3.2 queue 的成员函数 .....	368
22.9 总结 .....	339	24.3.3 使用 <code>push()</code> 在队尾插入以及使用 <code>pop()</code> 从队首删除 .....	369
22.10 问与答 .....	340	24.4 使用 STL 优先级队列 .....	370
22.11 作业 .....	340	24.4.1 实例化 <code>priority_queue</code> 类 .....	370
22.11.1 测验 .....	340	24.4.2 <code>priority_queue</code> 的成员函数 .....	371
22.11.2 练习 .....	340	24.4.3 使用 <code>push()</code> 在 <code>priority_queue</code> 末尾插入以及使用 <code>pop()</code> 在 <code>priority_queue</code> 开头删除 .....	372
第 23 章 STL 算法 .....	341	24.5 总结 .....	373
23.1 什么是 STL 算法 .....	341	24.6 问与答 .....	373
23.2 STL 算法的分类 .....	341	24.7 作业 .....	374
23.2.1 非变序算法 .....	341	24.7.1 测验 .....	374
23.2.2 变序算法 .....	342	24.7.2 练习 .....	374
23.3 使用 STL 算法 .....	343	第 25 章 使用 STL 位标志 .....	375
23.3.1 根据值或条件查找元素 .....	343	25.1 <code>bitset</code> 类 .....	375
23.3.2 计算包含给定值或满足给定条件的元素数 .....	345	25.2 使用 <code>std::bitset</code> 及其成员 .....	376
23.3.3 在集合中搜索元素或序列 .....	346		
23.3.4 将容器中的元素初始化为指定值 .....	348		
23.3.5 使用 <code>std::generate()</code> 将元素设置为运行阶段生成的值 .....	349		

25.2.1	std::bitset 的运算符	376	27.5	使用 std::fstream 处理文件	398
25.2.2	std::bitset 的成员方法	377	27.5.1	使用 open() 和 close() 打开和关闭文件	398
25.3	vector<bool>	378	27.5.2	使用 open() 创建文本文件并使用运算符<<写入文本	399
25.3.1	实例化 vector<bool>	378	27.5.3	使用 open() 和运算符>>读取文本文件	399
25.3.2	vector<bool> 的成员函数和运算符	379	27.5.4	读写二进制文件	400
25.4	总结	380	27.6	使用 std::stringstream 对字符串进行转换	402
25.5	问与答	380	27.7	总结	403
25.6	作业	381	27.8	问与答	403
25.6.1	测验	381	27.9	作业	403
25.6.2	练习	381	27.9.1	测验	403
第 26 章	理解智能指针	382	27.9.2	练习	404
26.1	什么是智能指针	382	第 28 章	异常处理	405
26.1.1	常规(原始)指针存在的问题	382	28.1	什么是异常	405
26.1.2	智能指针有何帮助	383	28.2	导致异常的原因	405
26.2	智能指针是如何实现的	383	28.3	使用 try 和 catch 捕获异常	406
26.3	智能指针类型	384	28.3.1	使用 catch(...) 处理所有异常	406
26.3.1	深复制	384	28.3.2	捕获特定类型的异常	407
26.3.2	写时复制机制	385	28.3.3	使用 throw 引发特定类型的异常	408
26.3.3	引用计数智能指针	386	28.4	异常处理的工作原理	409
26.3.4	引用链接智能指针	386	28.4.1	std::exception 类	411
26.3.5	破坏性复制	386	28.4.2	从 std::exception 派生出自定义异常类	411
26.4	深受欢迎的智能指针库	389	28.5	总结	413
26.5	总结	389	28.6	问与答	413
26.6	问与答	389	28.7	作业	413
26.7	作业	390	28.7.1	测验	414
26.7.1	测试	390	28.7.2	练习	414
26.7.2	练习	390	第 29 章	继续前行	415
第 27 章	使用流进行输入和输出	391	29.1	当今的处理器有何不同	415
27.1	流的概述	391	29.2	如何更好地利用多个内核	416
27.2	重要的 C++ 流类和流对象	391	29.2.1	线程是什么	416
27.3	使用 std::cout 将指定格式的数据写入控制台	392	29.2.2	为何要编写多线程应用程序	417
27.3.1	使用 std::cout 修改数字的显示格式	393	29.2.3	线程如何交换数据	417
27.3.2	使用 std::cout 对齐文本和设置字段宽度	394	29.2.4	使用互斥量和信号量同步线程	418
27.4	使用 std::cin 进行输入	395	29.2.5	多线程技术带来的问题	418
27.4.1	使用 std::cin 将输入读取到基本类型变量中	395	29.3	编写杰出的 C++ 代码	418
27.4.2	使用 std::cin.get 将输入读取到 char 数组中	396	29.4	更深入地学习 C++	419
27.4.3	使用 std::cin 将输入读取到 std::string 中	397	29.4.1	在线文档	419
			29.4.2	提供指南和帮助的社区	420
			29.5	总结	420



29.6 问与答 .....	420	A.4 不同进制之间的转换 .....	423
29.7 作业 .....	420	A.4.1 通用转换步骤 .....	423
附录 A 二进制和十六进制 .....	421	A.4.2 从十进制转换为二进制 .....	423
A.1 十进制 .....	421	A.4.3 从十进制转换为十六进制 .....	424
A.2 二进制 .....	421	附录 B C++关键字 .....	425
A.2.1 计算机为何使用二进制 .....	422	附录 C 运算符优先级 .....	426
A.2.2 位和字节 .....	422	附录 D 答案 .....	427
A.2.3 1KB 相当于多少字节 .....	422	附录 E ASCII 码 .....	456
A.3 十六进制 .....	422		

# 第1章

## 绪论

欢迎使用本书！通过阅读本章，您将迈出成为高级 C++ 程序员的第一步。

在本章中，您将学习：

- 为何 C++ 是软件开发的标准；
- 输入、编译和链接第一个 C++ 程序；
- C++11 新增的功能。

### 1.1 C++ 简史

编程语言旨在让人更容易使用计算资源。C++ 并非一种新语言，但仍被广泛采用，并在不断改进。2011 年，最新的 C++ 标准获得了 ISO 标准委员会的批准，名为 C++11。

#### 1.1.1 与 C 语言的关系

C++ 最初由 Bjarne Stroustrup 于 1979 年在贝尔实验室开发，被设计为 C 语言的继任者。C 语言是一种过程型语言，程序员使用它定义执行特定操作的函数，而 C++ 是一种面向对象的语言，实现了继承、抽象、多态和封装等概念。C++ 支持类，而类包含成员数据以及操作数据的成员方法（方法类似于 C 语言中的函数）。其结果是，程序员需要考虑数据以及要用它们来做什么。一直以来，C++ 编译器都支持 C 语言，这具有向后与既有代码兼容的优势，但也存在缺点，那就是编译器非常复杂，因为随着 C++ 的发展，编译器既要实现所有的新功能，又要向程序员提供这种向后兼容的功能。

#### 1.1.2 C++ 的优点

C++ 是一种中级编程语言，这意味着使用它既可以高级编程方式编写应用程序，又可以低级编程方式编写与硬件紧密协作的库。在很多程序员看来，C++ 既是一种高级语言，让他们能够开发复杂的应用程序，又提供了极大的灵活性，让开发人员能够控制资源的使用和可用性，从而最大限度地提高性能。

虽然有更新的编程语言面世，如 Java 以及其他基于 .NET 的语言，但 C++ 始终深受欢迎并在不断发展。较新的语言因提供了某些功能（如通过垃圾收集管理内存）让一些程序员钟爱有加，但在需要精确控制应用程序的性能时，他们还是会选择 C++。当前，常常使用 C++ 编写 Web 服务器，并使用 HTML、Java 或 .NET 编写前端应用程序。

#### 1.1.3 C++ 标准的发展历程

经过多年的发展，C++ 得到了广泛接受和采纳，但存在多种不同版本，因为有很多不同的编译

器，它们风格各异。鉴于 C++ 广受欢迎，且不同的版本之间存在差异，这导致了众多互操作性和移植方面的问题，需要对其进行标准化。

1998 年，第一个 C++ 标准获得了 ISO 标准委员会的批准，这就是 ISO/IEC 14882:1998。2003 年进行了修订，即为 ISO/IEC 14882:2003。最新的 C++ 标准于 2011 年 8 月获批，其官方名称为 C++11 (ISO/IEC 14882:2011)，它包含一些雄心勃勃的改进。

#### 注意

网上的很多文档仍称这个 C++ 标准为 C++0x。人们最初预期新标准将于 2008 或 2009 年获批，因此用 x 表示获批的年份。但提议的新标准到 2011 年才获批，因此将其称为 C++11。换句话说，C++11 是新的 C++0x。

### 1.1.4 哪些人使用 C++ 程序

无论你是谁，做什么工作（无论是经验丰富的程序员，还是将计算机用于特定目的的人），都可能经常会用到 C++ 应用程序和库。C++ 常用于开发操作系统、设备驱动程序、办公软件、Web 服务器、基于云的应用程序和搜索引擎，甚至用于编写新编程语言编译器。

## 1.2 编写 C++ 应用程序

当您在计算机上启动 Notepad 或 VI 时，实际上是命令处理器运行该程序的可执行文件。可执行文件是可运行的成品，应按程序员期望的那样做。

### 1.2.1 生成可执行文件的步骤

要创建可在操作系统中运行的可执行文件，第一步是编写一个 C++ 程序。创建 C++ 应用程序的基本步骤如下。

1. 使用文本编辑器编写 C++ 代码。
2. 使用 C++ 编译器对代码进行编译，将代码转换为包含在目标文件中的机器语言版本。
3. 使用链接程序链接编译器的输出，生成一个可执行文件（如 Windows 中的 .exe 文件）。

您在编程时创建的是文本文件，但微处理器无法处理这样的文件。在编译过程中，C++ 代码（通常包含在 .CPP 文本文件中）被转换为处理器能够处理的字节码。编译器每次转换一个代码文件，生成一个扩展名为 .o 或 .obj 的目标文件，并忽略这个 CPP 文件可能对其他文件中代码的依赖。解析这些依存关系的工作由链接程序负责。除将各种目标文件组合起来外，链接程序还建立依存关系，如果链接成功，则创建一个可执行文件，供程序员执行和分发。

### 1.2.2 分析并修复错误

大多数复杂应用程序很少能够一次通过编译并完美地运行，由众多程序员合作开发的应用程序尤其如此。无论使用什么语言（包括 C++）编写，庞大或复杂的应用程序都需要运行很多次，以分析问题和发现 Bug。修复 Bug 后，重新生成程序，再重复上述过程。因此，除编写、编译和链接等三个步骤外，开发过程通常还包括调试步骤。在这个步骤中，程序员使用工具（如监视点）和调试功能（如逐行执行应用程序）对应用程序中的异常和错误进行分析。

### 1.2.3 集成开发环境

很多程序员都喜欢使用集成开发环境（Integrated Development Environment, IDE）。集成开

发环境让您能够在一个统一的用户界面中完成输入、编译和链接步骤，它还提供了调试功能，让您能够更轻松地发现错误并解决问题。

提示

有很多免费的 C++ IDE 和编译器。在 Windows 和 Linux 系统中，最流行的分别是 Microsoft Visual C++ 学习版和 GNU C++ 编译器 g++。如果您使用的是 Linux 系统，要使用 g++ 编译器来开发 C++ 应用程序，可安装免费的 Eclipse IDE。

警告

在编写本书时，还没有全面支持 C++11 标准的编译器，但前述编译器支持 C++11 的很多主要功能。

应该	不应该
可使用简单的文本编辑器(如 Notepad 或 gedit)来创建源代码，也可使用 IDE。 保存文件时，务必使用扩展名.cpp。	不要使用富文本编辑器，因为它们经常会给您编写的代码添加标记。 不要使用扩展名.c，因为很多编译器都将这种文件视为 C 语言代码，而不是 C++ 代码。

### 1.2.4 编写第一个 C++ 应用程序

了解工具和步骤后，该编写第一个 C++ 应用程序了，它在屏幕上打印 Hello World!。如果您使用的是 Windows 操作系统和 Microsoft Visual C++ 学习版，可采取如下步骤。

- 1. 选择菜单“文件”>“新建”>“项目”，以新建一个项目。
- 2. 选择类型“Win32 控制台应用程序”，并取消选择复选框“预编译头”。
- 3. 将项目命名为 Hello，并用程序清单 1.1 所示的代码替换 Hello.cpp 中自动生成的内容。

如果您使用的是 Linux 操作系统，使用简单的文本编辑器（我使用的是 Ubuntu 上的 gedit）创建一个包含程序清单 1.1 所示内容的 CPP 文件。

程序清单 1.1 Hello Wrold 程序（Hello.cpp）

```
1: #include <iostream>
2:
3: int main()
4: {
5:     std::cout << "Hello World!" << std::endl;
6:     return 0;
7: }
```

这个应用程序很简单，只是使用 std::cout 在屏幕上打印一行消息。std::endl 命令 cout 换行。该应用程序退出时向操作系统返回零。

注意

默读程序时，知道特殊字符和关键字的发音可能会有所帮助。  
例如，对于#include，可读作 hash-include、sharp-include 或 pound-include，这取决于您以前的背景。  
同样，对于 std::cout，可读作 standard-c-out。

警告

别忘了，魔鬼隐藏在细节中，这意味着您必须准确地输入程序清单中的代码。编译器对代码的要求非常严格；语句必须以；结尾，如果您错误地输入了:，就会陷入一片混乱。

### 1.2.5 生成并执行第一个 C++ 应用程序

如果您使用的是 Microsoft Visual C++ 学习版, 可在该 IDE 中按 **Ctrl + F5** 直接运行程序。这将编译、链接并执行应用程序。也可依次执行如下步骤。

1. 右击项目并选择“生成”, 准备生成可执行文件。
2. 在命令提示符中, 切换到可执行文件所属的文件夹 (通常是项目文件夹中的 **Debug** 文件夹)。
3. 输入可执行文件的名称以运行它。

在 Microsoft Visual C++ 中编写的程序与图 1.1 极其相似。

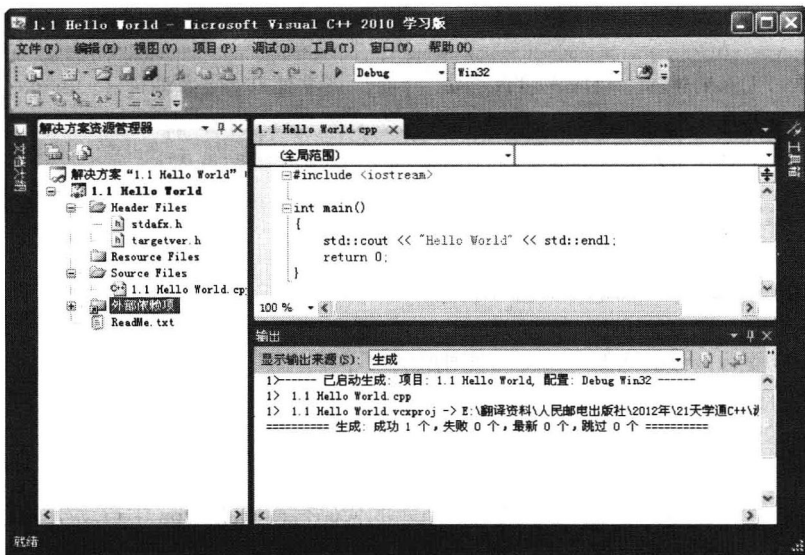


图 1.1 在 Visual C++ 2010 学习版中创建的简单 C++ 程序“Hello World”

如果您使用的是 Linux 系统, 可使用如下命令行调用 **g++** 编译器和链接程序。

```
g++ -o hello Hello.cpp
```

该命令行让 **g++** 编译 C++ 文件 **Hello.cpp**, 并创建一个名为 **hello** 的可执行文件。在 Linux 和 Windows 系统中, 分别执行命令 **./hello** 或 **Hello.exe**, 这将返回如下输出。

```
Hello World!
```

祝贺您踏上了学习有史以来最流行、最强大的编程语言之一的道路!

#### C++ ISO 标准的意义

正如您看到的, 通过遵守标准, 可在多种平台 (操作系统) 中编译和执行程序清单 1.1 所示的代码——前提条件是有遵守标准的 C++ 编译器。因此, 如果要编写让 Windows 用户和 Linux 用户都能运行的软件, 通过在编程中遵守标准 (不使用编译器或平台特有的语义) 是一种获取更多用户的简单方式, 这让您无需针对要支持的每种环境进行编程。当然, 编写不需要在操作系统级进行大量交互的应用程序时, 这种方法的效果最佳。

### 1.2.6 理解编译错误

编译器的要求非常苛刻, 但优秀的编译器会相当明确地指出错误在什么地方。如果您在编译程序清单 1.1 所示的应用程序时遇到问题, 错误消息将与下面的内容极其相似 (这是故意省略第

5 行末尾的分号导致的错误):

```
hello.cpp(6): error C2143: syntax error : missing ';' before 'return'
```

这条错误消息来自 Visual C++ 编译器,对错误进行了详细描述。它指出了包含错误的文件的名称,在哪一行遗漏了分号(这里是第 6 行),还使用错误编号(这里是 C2143)对错误本身进行了描述。在这个例子中,虽然遗漏分号的是第 5 行,但错误消息却指出错误发生在第 6 行,这是因为对编译器来说,只有等它返回语句后,才能确定在返回前,前一条语句必须结束。如果您可在第 6 行开头添加分号,程序将通过编译!

#### 注意

不同于 VBScript 等语言,在 C++ 中语句分行并不能自动结束语句。  
在 C++ 中,一条语句可跨越多行。

## 1.3 C++11 新增的功能

如果您是经验丰富的 C++ 程序员,可能发现程序清单 1.1 所示的基本程序没有任何变化。虽然 C++11 可以与以前的 C++ 版本兼容,但仍然做了大量工作让这种语言使用起来更容易。

**auto** 让您能够定义这样的变量,即编译器将自动推断其类型,这简化了变量声明,同时又不影响类型安全。**Lambda** 函数是没有名称的函数,让您能够编写紧凑的函数对象,而无需提供冗长的类定义,从而极大地减少了代码。C++11 让程序员能够编写可移植的多线程 C++ 应用程序,同时确保它们遵守标准。这些应用程序支持并行执行范式,在用户升级到多核 CPU 以改善硬件配置时,其性能将相应地提升。

本书将讨论 C++11 所做的众多改进,这里列举的只是其中几项。

## 1.4 总结

在本章中,您学习了如何编写、编译、链接和执行第一个 C++ 程序。本章还简要地介绍了 C++ 的发展历程,并演示了如何在不同的操作系统中使用不同的编译器对同一个程序进行编译,从而证明了标准的重要意义。

## 1.5 问与答

问:可以忽略编译器发出的警告消息吗?

答:在有些情况下,编译器会发出警告消息。警告与错误的不同之处在于,相关代码行的语法是正确的,能够通过编译,但可能有更佳的编写方式。优秀编译器在发出警告的同时提供修复建议。

修复建议可能是一种更安全的编程方式,也可能让应用程序能够处理非拉丁语字符和符号。您应该留意警告,并相应地改进应用程序。除非您确定警告是误报,否则不要对其视而不见。

问:解释型语言与编译型语言有何不同?

答:诸如 Windows Script 等语言是解释型的,不需要编译。解释型语言使用解释器,解释器直接读取脚本文件(代码)并执行指定的操作。因此,要在计算机上执行脚本,必须安装解释器。在运行阶段,解释器在微处理器和代码之间充当翻译,因此性能通常会受到影响。

问:什么是运行错误?它与编译错误有何不同?

答:执行应用程序时发生的错误称为运行错误。在较旧的 Windows 版本中,您可能遇到过臭名昭

著的“非法访问 (Access Violation)”错误，它就是运行错误。最终用户不会遇到编译错误，这种错误表明程序存在语法问题，禁止程序员生成可执行文件。

## 1.6 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

### 1.6.1 测验

1. 解释器和编译器有何不同？
2. 链接器的作用是什么？
3. 正常的开发周期包括哪些步骤？
4. C++11 标准如何更好地支持多核 CPU？

### 1.6.2 练习

1. 阅读下面的程序，在不运行它的情况下猜测其功能。

```
1: #include <iostream>
2: int main()
3: {
4:     int x = 8;
5:     int y = 6;
6:     std::cout << std::endl;
7:     std::cout << x - y << " " << x * y << x + y;
8:     std::cout << std::endl;
9:     return 0;
10: }
```

2. 输入练习 1 中的程序，然后编译并链接它。它的功能是什么？与您的猜测相符吗？
3. 下面的程序存在什么样的错误？

```
1: include <iostream>
2: int main()
3: {
4:     std::cout << "Hello Buggy World \n";
5:     return 0;
6: }
```

4. 修复练习 3 中程序的错误，重新编译、链接并运行它。它的功能是什么？

## 第 2 章

# C++程序的组成部分

C++程序由类、函数、变量及其他元素组成。本书的大部分内容将对这些组成部分进行深入解释，但了解程序是如何组合在一起的，必须分析一个完整的工作程序。

在本章中，您将学习：

- C++程序的组成部分；
- 各部分如何协同工作；
- 函数及其用途；
- 基本输入输出操作。

### 2.1 Hello World 程序的组成部分

在第 1 章中，您编写的第一个 C++程序只是将句子 **Hello World** 打印到屏幕上，虽然如此，却包含了 C++程序最重要的基本构件。本节将以程序清单 2.1 为例，对所有程序都包含的重要部分进行分析。

程序清单 2.1 HelloWorldAnalysis.cpp：分析 C++程序

```
1: // Preprocessor directive that includes header iostream
2: #include <iostream>
3:
4: // Start of your program: function block main()
5: int main()
6: {
7:     /* Write to the screen */
8:     std::cout << "Hello World" << std::endl;
9:
10:    // Return a value to the OS
11:    return 0;
12: }
```

可将这个 C++程序划分为两个部分：以#打头的预处理器编译指令以及以 `int main()`打头的程序主体。

**注意**

第 1、4、7 和 10 行以//或/\*打头，它们都是注释，将被编译器忽略。这些注释是供人类阅读的。注释将在下一节更详细地讨论。

#### 2.1.1 预处理器编译指令#include

顾名思义，预处理器是一个在编译前运行的工具。预处理器编译指令是向预处理器发出的命令，总是以符号#打头。在程序清单 2.1 中，第 2 行的`#include<filename>`让预处理器获取指定文件（这里是 `iostream`）的内容，并将它们放在编译指令所处的位置。`iostream` 是一个标准头文件，这里包含它是因为第



8行使用了 `std::cout` 将 `Hello World` 打印到屏幕上，而该头文件包含 `std::cout` 的定义。换句话说，编译器之所以能够编译包含 `std::cout` 的第8行，是因为我们在第2行指示预处理器包含了 `std::cout` 的定义。

**注意**

在专业人员编写的 C++ 应用程序中，包含的文件并非都是标准头文件。编写复杂的应用程序时，通常将其代码放在多个文件中，这使得需要在某些文件中包含其他文件。因此，如果需要在 `FileA` 中使用 `FileB` 中定义的元素，就需要在前者中包含后者。为此，通常在 `FileA` 中使用如下 `include` 语句：

```
#include "...relative path to FileB\FileB"
```

这里使用引号而不是尖括号来包含自己创建的头文件。尖括号 (`<>`) 通常用于包含标准头文件。

## 2.1.2 程序的主体——`main()`

预处理器编译指令的后面是程序的主体——`main()` 函数，执行 C++ 程序时总是从这里开始。声明 `main()` 时，总是在它前面加上 `int`，这是一种标准化约定，表示 `main()` 函数的返回类型为 `int`。

**注意**

在很多 C++ 应用程序中，都使用了类似于下面的 `main()` 函数变种：

```
int main (int argc, char* argv[])
```

这符合标准且可以接受，因为其返回类型为 `int`。括号内的内容是提供给程序的参数。该程序可能允许用户执行时提供命令行参数，如：

```
program.exe /DoSomethingSpecific
```

其中 `/DoSomethingSpecific` 是操作系统传递给程序的参数，以便在 `main` 中进行处理。

下面来看第8行，它实际执行该程序的功能：

```
std::cout << "Hello World" << std::endl;
```

`cout`（控制台输出，读作 *see-out*）是将 `Hello World` 打印到屏幕上的语句。`cout` 是在标准名称空间中定义的一个流（因此这里使用了 `std::cout`），这里使用流插入运算符 `<<` 将文本 `Hello World` 放到这个流中。`std::endl` 用于换行，将其插入流中相当于插入回车。每次需要将新实体插入流中时，都使用了流插入运算符。

C++ 流的优点是，将类似的语义用于另一种类型的流时，将执行不同的操作，如插入到文件而不是控制台。因此，流的用法非常直观，使用过一种流（如将文本写入控制台的 `cout`）后，其他流（如帮助将文本文件写入磁盘的 `fstream`）使用起来就非常容易。

流将在第27章更详细地讨论。

**注意**

实际文本（包括引号）“`Hello World`”被称为字符串字面量。

## 2.1.3 返回值

在 C++ 中，除非明确声明了不返回值，否则函数必须返回一个值。`main()` 也是函数，且总是返回一个整数。这个值返回给操作系统，根据应用程序的性质，这可能很有用，因为大多数操作系统都提供了查询功能，让您能够获悉正常终止的应用程序的返回值。在很多情况下，一个应用程序被另一个应用程序启动，而父应用程序（启动者）想知道子应用程序（被启动者）是否成功地完成了其任务。程序员可使用 `main()` 的返回值向父应用程序传递成功或错误状态。

**注意**

根据约定，程序员在程序运行成功时返回 0，并在出现错误时返回 -1。然而，返回值为整数，程序员可利用整个整数范围，指出众多不同的成功或失败状态。

**警告**

C++ 区分大小写，如果将 `int` 写成了 `Int`、将 `void` 写成了 `Void` 或将 `std::cout` 写成了 `Std::Cout`，程序将不能通过编译。

## 2.2 名称空间的概念

在这个程序中，使用的是 `std::cout` 而不是 `cout`，原因在于 `cout` 位于标准（`std`）名称空间中。

那么什么是名称空间呢？

假设调用 `cout` 时没有使用名称空间限定符，且编译器知道 `cout` 存在于两个地方，编译器应调用哪个呢？当然，这会导致冲突，进而无法通过编译。这就是名称空间的用武之地。名称空间是给代码指定的名称，有助于降低命名冲突的风险。通过使用 `std::cout`，可命令编译器调用名称空间 `std` 中独一无二的 `cout`。

### 注意

您使用 `std`（读作 `standard`）名称空间来调用获得 ISO 标准委员会批准，并在该名称空间中声明的函数、流和工具。

很多程序员发现，使用 `cout` 和 `std` 名称空间中的其他功能时，在代码中添加 `std` 限定符很繁琐。为避免添加该限定符，可使用声明 `using namespace`，如程序清单 2.2 所示。

程序清单 2.2 `using namespace` 声明

```
1: // Pre-processor directive
2: #include <iostream>
3:
4: // Start of your program
5: int main()
6: {
7:     // Tell the compiler what namespace to search in
8:     using namespace std;
9:
10:    /* Write to the screen using std::cout */
11:    cout << "Hello World" << endl;
12:
13:    // Return a value to the OS
14:    return 0;
15: }
```

### ▼ 分析：

请注意第 8 行。通过告诉编译器您要使用名称空间 `std`，在第 11 行使用 `cout` 和 `endl` 时，就无需显式地指定名称空间了。

程序清单 2.3 是程序清单 2.2 的更严谨版本，它没有包含整个名称空间，而只包含要使用的元素。

程序清单 2.3 关键字 `using` 的另一种用法

```
1: // Pre-processor directive
2: #include <iostream>
3:
4: // Start of your program
5: int main()
6: {
7:     using std::cout;
8:     using std::endl;
9:
10:    /* Write to the screen using cout */
11:    cout << "Hello World" << endl;
12:
13:    // Return a value to the OS
14:    return 0;
15: }
```

▼ 分析:

在程序清单 2.3 中，使用第 7~8 行替换了程序清单 2.2 的第 8 行。两者的差别在于，前者让您能够在不显式指定名称空间限定符 `std::` 的情况下使用名称空间 `std` 中的所有元素，而后者让您能够在不显式指定名称空间限定符 `std::` 的情况下使用 `std::cout` 和 `std::endl`。

2.3 C++代码中的注释

在程序清单 2.3 中，第 1、4、10 和 13 行包含口语（这里为英语）文本，但不会影响程序编译，也不会影响程序的输出。这些行被称为注释。注释会被编译器忽略，程序员使用它们来对代码进行解释，因此使用人类能够明白的语言编写。

C++支持下面两种风格的注释。

- `//`指出当前行为注释，例如：

```
// This is a comment
```

- `/*`和`*/`表示它们之间的文本为注释，即便这些文本跨越多行：

```
/* This is a comment  
and it spans two lines */
```

注意

程序员为何要对自己编写的代码进行解释呢？这看起来好像有点奇怪，但程序越大，合作开发同一个模块的程序员越多，编写易于理解的代码就越重要。必须使用清晰的注释对代码的功能以及为何要这样做进行解释，这很重要。

应该	不应该
务必添加注释，对程序中复杂算法和复杂部分的工作原理进行解释。 务必以其他程序员能够理解的方式编写注释。	不要使用注释来解释显而易见的代码。 别忘了，不要因为可以添加注释，就编写晦涩难懂的代码。 别忘了，修改代码时，可能需要相应地更新注释。

2.4 C++函数

C++函数与 C 语言函数相同。函数让您能够将应用程序划分成多个功能单元，并按您选择的顺序调用。函数被调用时，通常将一个值返回给调用它的函数。最著名的函数无疑是 `main()`，它被编译器视为 C++应用程序的起点，必须返回一个整数。

程序员通常需要编写自己的函数。程序清单 2.4 是一个简单的应用程序，它使用一个函数在屏幕上显示内容，而该函数使用各种参数调用了 `std::cout`。

程序清单 2.4 声明、定义和调用函数，该函数演示了 `std::cout` 的一些功能

```
1: #include <iostream>
2: using namespace std;
3:
4: // Function declaration
5: int DemoConsoleOutput();
6:
7: int main()
8: {
9:     // Call i.e. invoke the function
10:    DemoConsoleOutput();
```

```
11:
12:     return 0;
13: }
14:
15: // Function definition
16: int DemoConsoleOutput()
17: {
18:     cout << "This is a simple string literal" << endl;
19:     cout << "Writing number five: " << 5 << endl;
20:     cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
21:     cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
22:     cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
23:
24:     return 0;
25: }
```

---

#### ▼ 输出:

```
This is a simple string literal
Writing number five: 5
Performing division 10 / 5 = 2
Pi when approximated is 22 / 7 = 3
Pi more accurately is 22 / 7 = 3.14286
```

---

#### ▼ 分析:

这里需要注意的是第 5、10 和 15~25 行。第 5 行为函数声明，它告诉编译器您要创建一个函数，该函数名为 `DemoConsoleOutput()`，返回类型为 `int`。正是因为该声明，编译器才会编译第 10 行，并假定后面会有函数定义（即函数的实现），这里是第 15~25 行。

这个函数演示了 `cout` 的各种功能。它不仅能够像前面显示 `Hello World` 那样显示文本，还能显示简单算术运算的结果。第 21 和 22 行都试图显示 `pi` 的值（`22/7`），但后者更精确，因为 `22.0/7` 让编译器将结果视为实数（在 C++ 中为 `float`），而不是整数。

注意到该函数被声明为返回一个整数（这里返回的是 0）。由于它不做任何决策，因此没必要返回其他值。同样，`main()` 也返回 0。鉴于 `main()` 将其所有任务都交给了函数 `DemoConsoleOutput()` 去完成，更明智的做法是，在 `main()` 中返回该函数的返回值，如程序清单 2.5 所示。

---

#### 程序清单 2.5 使用函数的返回值

```
1: #include <iostream>
2: using namespace std;
3:
4: // Function declaration and definition
5: int DemoConsoleOutput()
6: {
7:     cout << "This is a simple string literal" << endl;
8:     cout << "Writing number five: " << 5 << endl;
9:     cout << "Performing division 10 / 5 = " << 10 / 5 << endl;
10:    cout << "Pi when approximated is 22 / 7 = " << 22 / 7 << endl;
11:    cout << "Pi more accurately is 22 / 7 = " << 22.0 / 7 << endl;
12:
13:    return 0;
14: }
15:
16: int main()
17: {
18:     // Function call with return used to exit
19:     return DemoConsoleOutput();
20: }
```

---

### ▼ 分析:

该应用程序的输出与前一个程序清单相同,但编写方式存在细微的差别。首先,由于在 `main()` 前面(第5行)定义了该函数,因此无需声明该函数。较新的 C++ 编译器将其视为函数声明和定义。另外, `main()` 也更简短。第19行调用函数 `DemoConsoleOutput()`,并将该函数的返回值作为应用程序的返回值。

### 注意

在函数无需做任何决策,也无需返回成功/失败状态时,可将其返回类型声明为 `void`:

```
void DemoConsoleOutput()
```

这个函数不能返回值。对于返回类型为 `void` 的函数,不能通过执行它来做决策。

函数可以接受参数,可以递归,可以包含多条返回语句,可以重载,还可声明为内联的,在这种情况下,编译器将展开函数调用。这些概念都将在第7章更详细地介绍。

## 2.5 使用 `std::cin` 和 `std::cout` 执行基本输入输出操作

计算机让用户能够以各种方式与其运行的应用程序交互,还让应用程序能够以众多方式与用户交互。用户通过键盘和鼠标与应用程序交互。您可以在屏幕上以文本和复杂图形的方式显示信息,使用打印机将信息打印到纸上,还可将信息存储到文件系统中,供以后使用。本节讨论最简单的 C++ 输入输出方式:使用控制台读写信息。

要将简单的文本数据写入到控制台,可使用 `std::cout` (读作 **standard see-out**);要从控制台读取文本和数字,可使用 `std::cin` (读作 **standard see-in**)。事实上,在程序清单 2.1 中,在屏幕上显示 **Hello World** 时,您就使用过 `cout`:

```
8:    std::cout << "Hello World" << std::endl;
```

在这条语句中, `cout` 的后面依次为插入运算符 `<<` (帮助将数据插入输出流)、要插入的字符串字面量 **Hello World**、使用 `std::endl` (读作 **standard end-line**) 表示的换行符。

`cin` 的用法也很简单,但它用于输入,因此需要指定要将输入数据存储到其中的变量:

```
std::cin >> Variable;
```

因此, `cin` 后面依次为提取运算符 `>>` (从输入流中提取数据)以及要将数据存储到其中的变量。如果需要将用户输入存储到两个变量中——每个变量包含用空格分隔的数据,可使用一条语句完成这项任务:

```
std::cin >> Variable1 >> Variable2;
```

请注意, `cin` 可用于从用户那里获取文本输入和数字输入,如程序清单 2.6 所示。

### 程序清单 2.6 使用 `cin` 和 `cout` 显示用户的数字输入和文本输入

```
1: #include <iostream>
2: #include <string>
3: using namespace std;
4:
5: int main()
6: {
7:     // Declare a variable to store an integer
8:     int InputNumber;
9:
10:    cout << "Enter an integer: ";
11:
12:    // store integer given user input
13:    cin >> InputNumber;
14:
```

```
15:    // The same with text i.e. string data
16:    cout << "Enter your name: ";
17:    string InputName;
18:    cin >> InputName;
19:
20:    cout << InputName << " entered " << InputNumber << endl;
21:
22:    return 0;
23: }
```

#### ▼ 输出:

```
Enter an integer: 2011
Enter your name: Siddhartha
Siddhartha entered 2011
```

#### ▼ 分析:

第 8 行声明了一个名为 `InputNumber` 的变量，用于存储类型为 `int` 的数据。第 10 行使用 `cout` 让用户输入一个数字，而第 13 行使用 `cin` 将输入的数字存储在该 `int` 变量中。接下来重复相同的操作，存储用户的名字，当然这不能存储在 `int` 变量中，而是存储在 `string` 变量中，如第 17~18 行所示。第 2 行包含了 `<string>`，原因是后面在 `main()` 中使用了类型 `string`。最后，第 20 行使用一条 `cout` 语句显示输入的名字和数字以及连接文本，输出为 `Siddhartha entered 2011`。

这是一个非常简单的示例，演示了 C++ 中输入输出的基本原理。如果您不清楚变量的概念，不用担心，第 3 章将详细阐述。

## 2.6 总结

本章介绍了简单 C++ 程序的基本组成部分。您明白了 `main()` 是什么，了解了名称空间，学习了控制台输入输出的基本知识。现在，您能够在自己编写的任何程序中使用这些知识了。

## 2.7 问与答

问: `#include` 的作用是什么?

答: 这是一个预处理器编译指令。预处理器在您调用编译器时运行。该指令使得预处理器将 `#include` 后面的 `<>` 中的文件读入程序，其效果如同将这个文件输入到源代码中的这个位置。

问: `//` 注释和 `/*` 注释之间有何不同?

答: `//` 注释到行尾结束; `/*` 注释到 `*/` 结束。`//` 注释也被称为单行注释，`/*` 注释通常被称为多行注释。请记住，即使是函数的结尾也不能作为 `/*` 注释的结尾，必须加上注释结尾标记 `*/`，否则将出现编译错误。

问: 什么情况下需要命令行参数?

答: 需要让用户修改程序的行为时。例如，Linux 命令 `ls` 和 Windows 命令 `dir` 都显示当前目录（文件夹）的内容，要查看另一个目录中的文件，需要使用命令行参数指定相应的路径，如 `ls/` 或 `dir\`。

## 2.8 作业

作业包括测验和练习，前者帮助读者加深对所知识的理解，后者提供了使用新学的知识的机会。

请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

### 2.8.1 测验

1. 声明 `Int main()` 有何问题？
2. 注释可以超过一行吗？

### 2.8.2 练习

1. 查错：输入下面的程序并编译它。它为什么不能通过编译？如何修复？

```
1: #include <iostream>
2: void main()
3: {
4:     std::Cout << "Is there a bug here?";
5: }
```

2. 修复练习 1 中的错误，然后重新编译、链接并运行它。
3. 修改程序清单 2.4，以演示减法（使用 `-`）和乘法（使用 `*`）。

## 第 3 章

# 使用变量和常量

变量让程序员能够将数据临时存储一段时间，而常量让程序员能够定义不允许修改的东西。在本章中，您将学习：

- 如何使用 C++11 关键字 `auto` 和 `constexpr`；
- 如何声明和定义变量与常量；
- 如何给变量赋值以及操纵这些值；
- 如何将变量的值显示到屏幕上。

### 3.1 什么是变量

在探索编程语言为何需要使用变量前，先来看看计算机的组成及其工作原理。

#### 3.1.1 内存和寻址概述

所有计算机、智能手机及其他可编程设备都包含微处理器和一定数量的临时存储空间，这种临时存储器被称为随机存取存储器（RAM）。另外，很多设备还让您能够将数据永久性地存储到硬盘等存储设备中。微处理器负责执行应用程序，在此过程中，它从 RAM 中获取要执行的应用程序以及相关的数据，这包括显示到屏幕上的数据以及用户输入的数据。

RAM 类似于宿舍里成排存物柜的存储区域，每个存物柜都有编号，即地址。要访问特定的内存单元，如内存单元 578，需要使用指令要求处理器从这里获取值或将值写入到这里。

#### 3.1.2 声明变量以访问和使用内存

下面的示例将帮助您明白变量是什么。假设您要编写一个程序，它将用户提供的两个数字相乘。用户被要求依次提供被乘数和乘数，而您需要存储它们，以便以后将它们相乘。您可能还需要存储乘法运算的结果，供以后使用，这取决于您要使用这个结果做什么。如果显式地指定用于存储这些数字的内存单元的地址（如 578），既慢又容易出错，因为这需要避免不小心覆盖原有的数据，以后还需避免覆盖您存储的数据。

使用 C++ 等语言编程时，您只需定义用于存储这些值的变量。定义变量非常简单，其语法如下：

```
variable_type variable_name;  
或  
variable_type variable_name = initial_value;
```



变量类型向编译器指出了变量可存储的数据的性质，编译器将为变量预留必要的空间。变量名由程序员选择，它替代了变量值在内存中的存储地址，但更友好。除非给变量赋初值，否则无法确保相应内存单元的内容是什么，这对程序可能不利。因此，初始化虽然是可选的，但对变量初始化通常是一个不错的编程习惯。程序清单 3.1 将用户提供的两个数字相乘，演示了如何在程序中声明、初始化和使用变量。

程序清单 3.1 使用变量存储数字及其相乘的结果

```
1: #include <iostream>
2: using namespace std;
3:
4: int main ()
5: {
6:     cout << "This program will help you multiply two numbers" << endl;
7:
8:     cout << "Enter the first number: ";
9:     int FirstNumber = 0;
10:    cin >> FirstNumber;
11:
12:    cout << "Enter the second number: ";
13:    int SecondNumber = 0;
14:    cin >> SecondNumber;
15:
16:    // Multiply two numbers, store result in a variable
17:    int MultiplicationResult = FirstNumber * SecondNumber;
18:
19:    // Display result
20:    cout << FirstNumber << " x " << SecondNumber;
21:    cout << " = " << MultiplicationResult << endl;
22:
23:    return 0;
24: }
```

▼ 输出:

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

▼ 分析:

这个应用程序要求用户输入两个数字，将它们相乘并显示结果。应用程序要使用用户输入的数字，必须将其存储到内存中。第 9 和 13 行声明了变量 `FirstNumber` 和 `SecondNumber`，用于临时存储用户输入的整数。第 10 和 14 行使用 `std::cin` 获取用户输入，并将其存储到两个整型变量中。第 21 行的 `cout` 语句用于将结果显示到控制台。

下面进一步分析其中的一个变量声明：

```
9:    int FirstNumber = 0;
```

这行代码声明了一个变量，其类型为 `int`（表示整型），名称为 `FirstNumber`，并将该变量的初始值设置为零。

使用汇编语言编程时，需要显式地要求处理器将被乘数存储到特定的位置，如 578，而 C++ 让您能够使用更友好的概念（如变量 `FirstNumber`）来访问内存单元，以检索和存储数据。将变量 `FirstNumber` 关联到内存单元的工作由编译器负责，它还负责为您完成相关的簿记工作（book Keeping）。

这样，程序员就可使用对人类友好的名称，把将变量关联到地址以及创建 `RAM` 访问指令的工作留给编译器去做。

**警告**

为编写易于理解和维护的代码，给变量指定合适的名称很重要。

变量名可包含数字和字母，但不能以数字打头。变量名不能包含空格和算术运算符（+、- 等）。要拉长变量名，可使用下划线。

另外，变量名不能是保留的关键字，例如，将变量命名为 `return` 将导致程序无法通过编译。

### 3.1.3 声明并初始化多个类型相同的变量

在程序清单 3.1 中，变量 `FirstNumber`、`SecondNumber` 和 `MultiplicationResult` 属于同一种类型——都是整型，但在三行中分别声明。如果您愿意，可简化这三个变量的声明，在一行代码中完成，如下所示：

```
int FirstNumber = 0, SecondNumber = 0, MultiplicationResult = 0;
```

**注意**

正如您看到的，在 C++ 中，可同时声明多个类型相同的变量，还可在函数开头声明变量。然而，需要时再声明变量通常是更好的选择，因为这让代码更容易理解——变量的声明离使用它的地方不远时，别人更容易了解变量的类型。

**警告**

存储在变量中的数据被存储在内存中。计算机关闭或应用程序终止时，这样的数据将丢失，除非程序员显式地将其存储到硬盘等永久性存储介质中。

将数据存储到磁盘文件将在第 27 章讨论。

### 3.1.4 理解变量的作用域

常规变量的作用域很明确，只能在作用域域内使用它们，如果您在作用域外使用它们，编译器将无法识别，导致程序无法通过编译。在作用域外面，变量是未定义的实体，编译器对其一无所知。

为让读者更好地理解变量的作用域，程序清单 3.2 将程序清单 3.1 所示的程序重新组织成了一个函数——`MultiplyNumbers()`，它将两个数字相乘并返回结果。

**程序清单 3.2 使用变量存储数字及其相乘的结果**

```
1: #include <iostream>
2: using namespace std;
3:
4: void MultiplyNumbers ()
5: {
6:     cout << "Enter the first number: ";
7:     int FirstNumber = 0;
8:     cin >> FirstNumber;
9:
10:    cout << "Enter the second number: ";
11:    int SecondNumber = 0;
12:    cin >> SecondNumber;
13:
14:    // Multiply two numbers, store result in a variable
15:    int MultiplicationResult = FirstNumber * SecondNumber;
16:
17:    // Display result
18:    cout << FirstNumber << " x " << SecondNumber;
19:    cout << " = " << MultiplicationResult << endl;
20: }
21: int main ()
22: {
23:     cout << "This program will help you multiply two numbers" << endl;
```

```
24:
25:    // Call the function that does all the work
26:    MultiplyNumbers();
27:
28:    // cout << FirstNumber << " x " << SecondNumber;
29:    // cout << " = " << MultiplicationResult << endl;
30:
31:    return 0;
32: }
```

---

**▼ 输出:**

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 24
51 x 24 = 1224
```

---

**▼ 分析:**

程序清单 3.2 的功能与程序清单 3.1 相同, 输出也相同。唯一的差别在于, 将工作交给了函数 `MultiplyNumbers()` 去完成, 并在 `main()` 中调用它。请注意, 不能在函数 `MultiplyNumbers()` 外面使用变量 `FirstNumber` 和 `SecondNumber`。如果您取消对 `main()` 中第 28 或 29 行的注释, 将出现编译错误, 而错误很可能是标识符未声明 (`undeclared identifier`)。

这是因为变量 `FirstNumber` 和 `SecondNumber` 的作用域为局部, 被限定在声明它的函数内, 这里为 `MultiplyNumbers()`。局部变量只能在这样的范围内使用, 即从声明它的语句开始到当前函数的末尾。标识函数结束的花括号 (`}`) 也限定了函数内部声明的变量的作用域。函数结束后, 将销毁所有局部变量, 并归还它们占用的内存。

编译时, 在 `MultiplyNumbers()` 内部声明的变量在该函数结束时不再存在, 如果在 `main()` 中使用它们, 程序将无法通过编译, 因为在 `main()` 中这些变量未声明。

**警告**

如果您在 `main()` 声明另一组同名变量, 就不能指望它们的值与您在 `MultiplyNumbers()` 中赋给同名变量的值相同。

编译器将 `main()` 中声明的变量视为独立的实体, 即便它们与另一个函数中声明的变量同名, 因为这些变量的作用域不同。

### 3.1.5 全局变量

在程序清单 3.2 中, 如果变量是在函数 `MultiplyNumbers()` 外部而不是内部声明的, 则在函数 `main()` 和 `MultiplyNumbers()` 中都可使用它们。程序清单 3.3 演示了全局变量, 它们是程序中作用域最大的变量。

---

**程序清单 3.3 使用全局变量**

```
1: #include <iostream>
2: using namespace std;
3:
4: // three global integers
5: int FirstNumber = 0;
6: int SecondNumber = 0;
7: int MultiplicationResult = 0;
8:
9: void MultiplyNumbers ()
10: {
11:     cout << "Enter the first number: ";
12:     cin >> FirstNumber;
```

```
13:
14:     cout << "Enter the second number: ";
15:     cin >> SecondNumber;
16:
17:     // Multiply two numbers, store result in a variable
18:     MultiplicationResult = FirstNumber * SecondNumber;
19:
20:     // Display result
21:     cout << "Displaying from MultiplyNumbers(): ";
22:     cout << FirstNumber << " x " << SecondNumber;
23:     cout << " = " << MultiplicationResult << endl;
24: }
25: int main ()
26: {
27:     cout << "This program will help you multiply two numbers" << endl;
28:
29:     // Call the function that does all the work
30:     MultiplyNumbers();
31:
32:     cout << "Displaying from main(): ";
33:
34:     // This line will now compile and work!
35:     cout << FirstNumber << " x " << SecondNumber;
36:     cout << " = " << MultiplicationResult << endl;
37:
38:     return 0;
39: }
```

#### ▼ 输出:

```
This program will help you multiply two numbers
Enter the first number: 51
Enter the second number: 19
Displaying from MultiplyNumbers(): 51 x 19 = 969
Displaying from main(): 51 x 19 = 969
```

#### ▼ 分析:

程序清单 3.3 在两个函数中显示了乘法运算的结果，而变量 `FirstNumber`、`SecondNumber` 和 `MultiplicationResult` 都不是在这两个函数内部声明的。这些变量为全局变量，因为声明它们的第 5~7 行不在任何函数内部。注意到第 23 和 36 行使用了这些变量并显示它们的值。尤其要注意的是，虽然 `MultiplicationResult` 的值是在 `MultiplyNumbers()` 中指定的，但仍可在 `main()` 中使用它。

#### 警告

不分青红皂白地使用全局变量通常是一种糟糕的编程习惯。

全局变量可在任何函数中赋值，因此其值可能出乎意料，在不同函数模块由小组中的不同程序员编写时尤其如此。

要像程序清单 3.3 那样在 `main()` 中获取乘法运算的结果，一种更妥善的方式是，让 `MultiplyNumbers()` 将结果返回给 `main()`。

## 3.2 编译器支持的常见 C++ 变量类型

在本书前面的大多数示例中，定义的变量类型都是 `int`（整型），然而 C++ 编译器支持很多基本变量类型，可供程序员选择。选择正确的变量类型犹如根据要做的工作选择正确的工具一样重要！十字螺钉与普通螺帽不匹，同样，无符号整型变量也不能用于存储负值！表 3.1 列出了各种变量类型及其可存储的数据的特征。要编写高效而可靠的 C++ 程序，这些信息非常重要。

表 3.1

变量类型

类型	值	类型	值
bool	true/false	int (16 位)	-32768~32767
char	256 个字符值	int (32 位)	-2147483648~2147483647
unsigned short int	0~65535	unsigned int (16 位)	0~65535
short int	-32768~32767	unsigned int (32 位)	0~4294967295
unsigned long int	0~4294967295	float	1.2e-38~3.4e38
long int	-2147483648~2147483647	double	2.2e-308~1.8e308

接下来的几小节将更详细地介绍一些重要的类型。

### 3.2.1 使用 bool 变量存储布尔值

C++提供了一种专为存储布尔值 **true** 和 **false** 而创建的类型，其中 **true** 和 **false** 都是保留的 C++关键字。对于取值为 ON 或 OFF、有或没有、可用或不可用等设置和标记，非常适合使用这种类型的变量来存储。

下面是一个声明并初始化布尔变量的例子：

```
bool AlwaysOnTop = false;

下面是一个结果为布尔值的表达式：
bool DeleteFile = (UserSelection == "yes");
// evaluates to true only if UserSelection contains "yes", else to false
```

### 3.2.2 使用 char 变量存储字符

**char** 变量用于存储单个字符，下面是一个声明示例：

```
char UserInput = 'Y'; // initialized char to 'Y'
```

请注意，表示内存空间容量大小的单位是位和字节。位的取值为 0 或 1，而字节可以包含字符的数字表示。因此像前面的示例那样使用字符数据时，编译器将把字符转换为可存储到内存中的数字表示。美国信息交换标准 (ASCII) 对拉丁字符 A~Z、a~z、数字 0~9、一些特殊键击 (如 DEL) 和一些特殊字符 (如空格) 的数字表示进行了标准化。

如果您查看附录 E 的 ASCII 码表，赋给变量 **UserInput** 的字符 **Y** 的 ASCII 码为 89，因此编译器将在分配给 **UserInput** 的内存空间中存储 89。

### 3.2.3 有符号整数和无符号整数的概念

符号表示正或负。您在计算机中使用的所有数字都以位和字节的方式存储在内存中。1 字节的内存单元包含 8 位，每位都要么为 0，要么为 1 (即存储这两个值之一)，因此 1 字节的内存单元可以有  $2^8$  (即 256) 个不同的取值。同样，16 位的内存单元可以有  $2^{16}$  (65536) 个不同的取值。

如果这些取值是无符号的 (即为正数)，则 1 个字节的可能取值为 0~255，而 2 个字节的可能取值为 0~65535。从表 3.1 可知，类型 **unsigned short** 的取值范围为 0~65535，因为它占用 16 位内存。因此，使用位和字节表示正值非常容易，如图 3.1 所示。

在这种空间中如何表示负数呢？一种方式是将 1 位用作符号位，指出其他位包含的值是正还是负，如图 3.2 所示。符号位必须是最高有效位 (**most-significant-bit**, **MSB**)，因为最低有效位 (**least-significant-bit**) 需要用于表示小于 2 的数字。当 **MSB** 包含符号信息时，假定 0 表示正，1 表示负，而其他位包含绝对值。

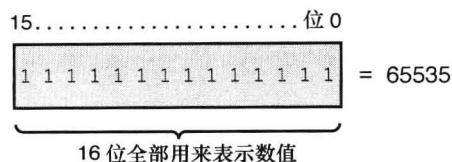


图 3.1 占用 16 位内存的 unsigned short 变量

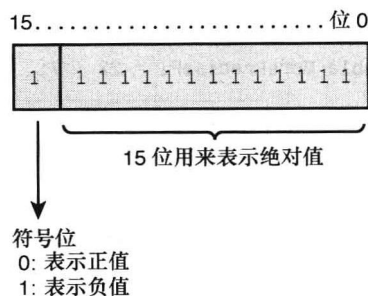


图 3.2 占用 16 位内存的 signed short 变量

因此，占用 8 位的有符号数的取值范围为 $-127 \sim 127$ ，而占用 16 位的有符号数的取值范围为 $-32768 \sim 32768$ 。如果查看表 3.1，将发现类型 `short` 的取值范围为 $-32768 \sim 32768$ 。

### 3.2.4 有符号整型 `short`、`int`、`long` 和 `long long`

这些类型的长度各不相同，因此取值范围也各不相同。`int` 可能是使用得最多的类型，在大多数编译器中，其长度都是 32 位。应根据变量可能存储的最大值给它指定合适的类型。

声明有符号类型的变量非常简单，如下所示：

```
short int SmallNumber = -100;
int LargerNumber = -70000;
long PossiblyLargerThanInt = -70000; // on some platforms, long is an int
long long LargerThanInt = -70000000000;
```

### 3.2.5 无符号整型 `unsigned short`、`unsigned int`、`unsigned long` 和 `unsigned long long`

不同于相应的有符号类型，无符号整型变量不能包含符号信息，因此，它们的最大取值为相应符号类型的两倍。

声明无符号类型变量也很简单，如下所示：

```
unsigned short int SmallNumber = 255;
unsigned int LargerNumber = 70000;
// on some platforms, long is int
unsigned long PossiblyLargerThanInt = 70000;
unsigned long long LargerThanInt = 70000000000;
```

#### 注意

如果预期变量的取值不会为负数，就应将其类型声明为无符号的。因此，如果您要存储苹果的数量，不要使用 `int` 变量，而应使用 `unsigned int` 变量，后者的最大取值为前者的两倍。

#### 警告

对于银行应用程序中用于存储账户余额的变量，将其类型声明为无符号的就可能不合适。

### 3.2.6 浮点类型 `float` 和 `double`

您可能在学校学过，浮点数就是实数，可以是正，也可以是负，还可以包含小数值。因此，如果要使用 C++ 变量存储  $\pi$  ( $22/7$ ) 的值，就应将其声明为浮点类型。

声明浮点类型变量的方式与程序清单 3.1 中声明 `int` 变量的方式相同。要声明一个可存储小数值的 `float` 变量，可像下面这样做：

```
float Pi = 3.14;
```

要声明双精度浮点数 (double) 变量, 可像下面这样做:

```
double MorePrecisePi = 22 / 7;
```

**注意**

表 3.1 列出的数据类型通常被称为 POD (Plain Old Data)。POD 还包含聚合数据类型 (结构、枚举、共用体和类)。

### 3.3 使用 sizeof 确定变量的长度

变量长度指的是: 程序员声明变量时, 编译器将预留多少内存, 用于存储赋给该变量的数据。变量的长度随类型而异, C++ 提供了一个方便的运算符——`sizeof`, 可用于确定变量的长度 (单位为字节) 或类型。

`sizeof` 的用法非常简单。要确定 `int` 变量的长度, 可调用 `sizeof` 并给它传递参数 `int`:

程序清单 3.4 演示了如何获悉各种标准 C++ 变量类型的长度。

```
cout << "Size of an int: " << sizeof (int);
```

**程序清单 3.4 获悉标准 C++ 变量类型的长度**

---

```
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:     cout << "Computing the size of some C++ inbuilt variable types" << endl;
7:
8:     cout << "Size of bool: " << sizeof(bool) << endl;
9:     cout << "Size of char: " << sizeof(char) << endl;
10:    cout << "Size of unsigned short int: " << sizeof(unsigned short) << endl;
11:    cout << "Size of short int: " << sizeof(short) << endl;
12:    cout << "Size of unsigned long int: " << sizeof(unsigned long) << endl;
13:    cout << "Size of long: " << sizeof(long) << endl;
14:    cout << "Size of int: " << sizeof(int) << endl;
15:    cout << "Size of unsigned long long: " << sizeof(unsigned long long) <<
endl;
16:    cout << "Size of long long: " << sizeof(long long) << endl;
17:    cout << "Size of unsigned int: " << sizeof(unsigned int) << endl;
18:    cout << "Size of float: " << sizeof(float) << endl;
19:    cout << "Size of double: " << sizeof(double) << endl;
20:
21:    cout << "The output changes with compiler, hardware and OS" << endl;
22:
23:    return 0;
24: }
```

---

#### ▼ 输出:

---

```
Computing the size of some C++ inbuilt variable types
Size of bool: 1
Size of char: 1
Size of unsigned short int: 2
Size of short int: 2
Size of unsigned long int: 4
Size of long: 4
Size of int: 4
Size of unsigned long long: 8
Size of long long: 8
Size of unsigned int: 4
Size of float: 4
Size of double: 8
The output changes with compiler, hardware and OS
```

**▼ 分析:**

程序清单 3.4 的输出指出了各种类型的长度（单位为字节），这是针对我使用的平台（编译器、操作系统和硬件）而言的。具体地说，这是在 64 位系统中以 32 位模式（使用 32 位编译器进行编译）运行该程序得到的结果。如果使用 64 位编译器进行编译，结果可能不同。我之所以使用 32 位编译器，是因为这样该应用程序在 32 位和 64 位系统上都能运行。输出表明，无符号类型和相应的有符号类型的长度相同，唯一的差别在于，后者的 MSB 包含符号信息。

**注意**

输出中的长度单位为字节。给对象分配内存时，其长度将是一个重要参数，在程序员动态地给对象分配内存时尤其如此。

**C++11****使用 auto——编译器的类型推断功能**

在有些情况下，根据赋给变量的初值，很容易知道其类型。例如，如果将变量的初值设置成了 true，就可推断其类型为 bool。在 C++11 中，可不显式地指定变量的类型，而使用关键字 auto：

```
auto Flag = true;
```

这将指定变量 Flag 的类型的任务留给了编译器。编译器检查赋给变量的初值的性质，再确定将变量声明为什么类型最合适。就这里而言，显然初始值 true 最适合存储到类型为 bool 的变量中。因此，编译器认为变量 Flag 的最佳类型为 bool，并在内部将 Flag 的类型视为 bool，程序清单 3.5 证明了这一点。

**程序清单 3.5 使用关键字 auto 依靠编译器的类型推断功能**

```
1: #include <iostream>
2: using namespace std;
3:
4: int main()
5: {
6:     auto Flag = true;
7:     auto Number = 2500000000000;
8:
9:     cout << "Flag = " << Flag;
10:    cout << " , sizeof(Flag) = " << sizeof(Flag) << endl;
11:    cout << "Number = " << Number;
12:    cout << " , sizeof(Number) = " << sizeof(Number) << endl;
13:
14:    return 0;
15: }
```

**▼ 输出:**

```
Flag = 1 , sizeof(Flag) = 1
Number = 2500000000000 , sizeof(Number) = 8
```

**▼ 分析:**

在第 6 和 7 行声明变量 Flag 和 Number 时，没有将其类型分别指定为 bool 和 long long，而使用了关键字 auto。这让编译器去决定变量的类型，而编译器将根据初始值来确定合适的类型。接下来，使用 sizeof 来检查编译器选择的类型是否符合预期，从程序清单 3.5 的输出可知，确实符合预期。

**注意**

使用 auto 时必须对变量进行初始化，因为编译器需要根据初始值来确定变量的类型。如果将变量的类型声明为 auto，却不对其进行初始化，将出现编译错误。

乍一看，auto 并非什么了不起的功能，但在变量类型非常复杂时，它可让编程容易得多。假设您使用 std::vector 声明了一个名为 MyNumbers 的动态整数数组：



```
std::vector<int> MyNumbers;
```

要访问或遍历该数组中的元素并显示它们，可使用如下代码：

```
for ( vector<int>::const_iterator Iterator = MyNumbers.begin();
      Iterator < MyNumbers.end();
      ++Iterator )
    cout << *Iterator << " ";
```

`std::vector` 和 `for` 循环都还没有介绍，因此即便上述代码看起来像天书，也不用担心。其功能如下：对于矢量中的每个元素——从 `begin()` 开始，到 `end()` 前面的一个元素结束，都使用 `cout` 显示其值。第 1 行代码非常复杂，它声明变量 `Iterator`，并将其初始值设置为 `begin()` 返回的值。这个变量的类型为 `vector<int>::const_iterator`，这对程序员来说，学习和书写起来都非常复杂。程序员无需熟记这一点，而可依赖于 `begin()` 的返回类型，将上述 `for` 循环简化为如下所示：

```
for( auto Iterator = MyNumbers.begin();
      Iterator < MyNumbers.end();
      ++Iterator )
    cout << *Iterator << " ";
```

注意到现在第 1 行有多紧凑。编译器检查 `Iterator` 的初始值——`begin()` 返回的值，并将该变量的类型设置为该返回值的类型。这简化了 C++ 编码工作，在大量使用模板时尤其如此。

## 3.4 使用 typedef 替换变量类型

C++ 允许您将变量类型替换为您认为方便的名称，为此可使用关键字 `typedef`。在下面的示例中，程序员想给 `unsigned int` 指定一个更具描述性的名称——`STRICTLY_POSITIVE_INTEGER`：

```
typedef unsigned int STRICTLY_POSITIVE_INTEGER;
STRICTLY_POSITIVE_INTEGER PosNumber = 4532;
```

编译时，第 1 行告诉编译器，`STRICTLY_POSITIVE_INTEGER` 就是 `unsigned int`。以后编译器再遇到已定义的类型 `STRICTLY_POSITIVE_INTEGER` 时，就会将它替换为 `unsigned int` 并继续编译。

**注意**

涉及语法烦琐的复杂类型，如使用模板的类型时，`typedef`（类型替换）特别方便。

## 3.5 什么是常量

假设您要编写一个程序，计算圆的面积和周长，其公式如下：

```
Area = Pi * Radius * Radius;
```

```
Circumference = 2 * Pi * Radius of circle
```

在这些公式中，`Pi` 为常量，其值为  $22/7$ 。您希望在整个程序中，`Pi` 的值都不变；您也不希望无意间将错误的值赋给 `Pi`，如错误地复制/粘贴或查找/替换。C++ 让您能够将 `Pi` 定义为声明后就不能修改的常量，换句话说，定义常量后，就不能修改它的值。在 C++ 中，给常量赋值会导致编译错误。

因此，在 C++ 中，常量类似于变量，只是不能修改。与变量一样，常量也占用内存空间，并使用名称标识为其预留的空间的地址，但不能覆盖该空间的内容。在 C++ 中，常量可以是：

- 字面常量；
- 使用关键字 `const` 声明的常量；
- 使用关键字 `constexpr` 声明的常量表达式（C++11 新增的）；
- 使用关键字 `enum` 声明的枚举常量；
- 使用 `#define` 定义的常量（已摒弃，不推荐）。

### 3.5.1 字面常量

再来看一下程序清单 3.1——将两个数相乘的简单程序。其中声明了一个名为 `FirstNumber` 的 `int` 变量：

```
9:   int FirstNumber = 0;
```

将这个 `int` 变量的初始值设置成了零。这个零是代码的一部分，将编译到应用程序中，且不可修改，因此称为字面常量。字面常量可以是任何类型：布尔型、整型、字符串等。在您编写的第一个 C++ 程序（程序清单 1.1）中，您使用了如下代码显示 `Hello World`：

```
std::cout << "Hello World" << std::endl;
```

`Hello World` 就是一个字符串字面常量。

### 3.5.2 使用 `const` 将变量声明为常量

从实用和编程的角度看，最重要的 C++ 常量类型是在变量类型前使用关键字 `const` 声明的。通用的声明方式类似于下面这样：

```
const type-name constant-name;
```

来看一个简单的应用程序，它显示常量 `Pi` 的值，如程序清单 3.6 所示。

程序清单 3.6 声明一个名为 `Pi` 的常量

```
1: #include <iostream>
2:
3: int main()
4: {
5:     using namespace std;
6:
7:     const double Pi = 22.0 / 7;
8:     cout << "The value of constant Pi is: " << Pi << endl;
9:
10:    // Uncomment next line to fail compilation
11:    // Pi = 345;
12:
13:    return 0;
14: }
```

#### ▼ 输出：

```
The value of constant Pi is: 3.14286
```

#### ▼ 分析：

请注意常量 `Pi` 的声明（第 7 行）。这里使用了关键字 `const` 来告诉编译器，`Pi` 是一个类型为 `double` 的常量。第 11 行试图给一个常量赋值，如果取消对该行的注释，将出现编译错误，指出不能给常量赋值。因此，常量是一种确保某些数据不能修改的强大方式。

#### 注意

如果变量的值不应改变，就应将其声明为常量，这是一种良好的编程习惯。通过使用关键字 `const`，程序员可确保数据不变，避免应用程序无意间修改该常量。  
在多位程序员合作开发时，这特别有用。

声明在编译期间长度固定的静态数组时，常量很有用。程序清单 4.2 提供了一个示例，演示了如何使用 `in` 常量指定数组长度。

### 3.5.3 使用 constexpr 声明常量

在 C++11 之前, C++ 就支持常量表达式的概念, 只是没有关键字 `constexpr`。在程序清单 3.5 中, 22.0/7 是一个常量表达式, C++11 之前的编译器也支持它。然而, C++11 之前的编译器不允许定义在编译阶段计算的函数。在 C++11 中, 可以编写下面这样的代码:

```
constexpr double GetPi() {return 22.0 / 7;}
```

还可将 `GetPi()` 与另一个常量一起使用, 如下所示:

```
constexpr double TwicePi() {return 2 * GetPi();}
```

乍一看, `const` 和 `constexpr` 之间的差别很小, 但从编译器和应用程序的角度看, 关键字 `constexpr` 提供了优化应用程序的可能性。对于第二条语句, 如果使用 `const`, 将在运行阶段执行计算, 但使用遵守 C++11 的编译器时, 将在编译阶段计算该表达式的值, 这提高了应用程序的运行速度。

#### 注意

编写本书时, Microsoft Visual C++ 学习版还不支持关键字 `constexpr`, 但 GNU 的 g++ 编译器支持。

### 3.5.4 枚举常量

在有些情况下, 变量只能有一组特定的取值。例如, 彩虹不能包含青绿色, 指南针的方位不能为“左”。在这些情况下, 需要定义这样一种变量, 即其可能取值由您指定。为此, 可使用关键字 `enum` 来声明枚举常量。

例如, 下面的枚举常量包含彩虹的颜色:

```
enum RainbowColors
{
    Violet = 0,
    Indigo,
    Blue,
    Green,
    Yellow,
    Orange,
    Red
};
```

下面的枚举常量包含基本方位:

```
enum CardinalDirections
{
    North,
    South,
    East,
    West
};
```

可使用枚举常量来指定变量的类型, 这样声明的变量只能取指定的值。因此, 如果要声明一个变量, 用于存储彩虹的颜色, 可以像下面这样做:

```
RainbowColors MyWorldsColor = Blue; // Initial value
```

上述代码声明了常量 `MyWorldsColor`, 其类型为 `RainbowColors`。这个枚举常量只能取 `RainbowColors` 中指定的值, 而不能取其他值。

#### 注意

声明枚举常量时, 编译器将把枚举值 (Violet 等) 转换为整数, 每个枚举值都比前一个大 1。您可以指定起始值, 如果没有指定, 编译器认为起始值为 0, 因此 North 的值为 0。如果愿意, 还可通过初始化显式地给每个枚举量指定值。

程序清单 3.7 演示了如何使用枚举常量来存储 4 个基本方位，并对第一个方位进行了初始化。

程序清单 3.7 使用枚举值指示基本方位

```
1: #include <iostream>
2: using namespace std;
3:
4: enum CardinalDirections
5: {
6:     North = 25,
7:     South,
8:     East,
9:     West
10: };
11:
12: int main()
13: {
14:     cout << "Displaying directions and their symbolic values" << endl;
15:     cout << "North: " << North << endl;
16:     cout << "South: " << South << endl;
17:     cout << "East: " << East << endl;
18:     cout << "West: " << West << endl;
19:
20:     CardinalDirections WindDirection = South;
21:     cout << "Variable WindDirection = " << WindDirection << endl;
22:
23:     return 0;
24: }
```

#### ▼ 输出:

```
Displaying directions and their symbolic values
North: 25
South: 26
East: 27
West: 28
Variable WindDirection = 26
```

#### ▼ 分析:

这里将 4 个基本方位定义为枚举常量，并将第一个常量（North）的值设置为 25（第 6 行），这自动将随后的常量分别设置为 26、27 和 28，如输出所示。第 20 行创建了一个类型为 `CardinalDirections` 的变量，并将其初始值设置为 `South`。第 21 行显示该变量时，编译器显示的是 `South` 对应的整数值——26。

#### 提示

您可能该看看程序清单 6.4 和程序清单 6.5，它们使用 `enum` 列举一个星期的各天，并使用条件处理指出用户选择的那天是根据哪颗星星命名的。

### 3.5.5 使用 `#define` 定义常量

首先也是最重要的是，编写新程序时，不要使用这种常量。这里介绍使用 `#define` 定义常量，只是为了帮助您理解一些旧程序，它们使用下面的语法定义常量：

```
#define Pi 3.14286
```

这是一个预处理器宏，让预处理器将随后出现的所有 `Pi` 都替换为 3.14286。预处理器将进行文本替换，而不是智能替换。编译器既不知道也不关心常量的类型。

#### 警告

使用 `#define` 定义常量的做法已被摒弃，因此不应采用这种做法。

### 3.6 给变量和常量命名

给变量命名的方式有很多，还有很多不同的约定。有些程序员喜欢在变量名开头用几个字符指出变量的类型，例如：

```
bool bIsLampOn = false;
```

其中 **b** 是程序员添加的前缀，指出变量的类型为 **bool**。这种表示法称为匈牙利表示法，最初由微软发明并倡议。然而，C++是一种强类型安全语言，编译器根据类型定义而不是名称前缀来获悉变量的类型。因此，当前强烈推荐程序员不要采用匈牙利表示法。变量名必须易于理解，哪怕这会导致变量名更长些。在这个示例中，假定该布尔变量表示车前灯开关，下面的变量名将更好些：

```
bool IsHeadLampOn = false;
```

这两个变量名都比下面的变量名更好：

```
bool b = false;
```

应不惜一切代价避免使用非描述性变量名。

应该	不应该
务必给变量指定描述性名称，那怕这会导致变量名很长。 务必确保变量名阐述了变量的用途。 务必站在别人的角度想一想，如果他从未见过您编写的代码，能明白变量名的含义吗？ 务必了解团队是否遵循特定的命名约定，如果是，请遵循这些约定。	不要使用太短或只有一个字符的变量名。 不要在变量名中使用只有您自己明白的怪异缩写。 不要将保留的 C++关键字用作变量名，因为这将导致程序无法通过编译。

### 3.7 不能用作常量或变量名的关键字

有些单词被 C++保留，不能用作变量名。对 C++编译器来说，这些关键字有特殊含义。关键字包括 **if**、**while**、**for**、**main** 等。表 3.2 和附录 B 列出了 C++定义的关键字。您的编译器可能还保留了其他单词，有关完整的关键字列表，请参阅编译器手册。

表 3.2 C++关键字

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void

续表

delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	
另外，下列单词被保留			
And	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

3.8 总结

在本章中，您了解到内存用于临时存储变量和常量的值。您了解到变量的长度取决于其类型，并可使用运算符 `sizeof` 来确定。您学习了各种变量类型，如 `bool`、`int` 等，知道它们用于存储不同类型的数据。选择正确的变量类型至关重要，如果选择的类型太短，可能导致回绕（`wrapping`）错误或溢出。您学习了 C++11 新增的关键字 `auto`，它让编译器根据变量的初始值确定其类型。

您还学习了各种常量，其中最重要的是使用关键字 `const` 和 `enum` 定义的常量。

3.9 问与答

问：既然可以使用常规变量代替常量，为何还要定义常量？

答：通过声明常量（尤其是使用关键字 `const` 时），可告诉编译器，其值是固定的，不允许修改。这样，编译器将确保不给常量赋值，即便另一位程序员接手了您的工作，不小心试图覆盖常量的值。因此，在知道变量的值不应改变时，应将其声明为常量，这是一个不错的编程习惯，可提高应用程序的质量。

问：为何应给变量赋初值？

答：如果不初始化，就无法知道变量包含的初始值。在这种情况下，初始值将是给变量预留的内存单元的内容。下面的语句使得创建变量 `MyFavoriteNumber` 后，就将指定的初始值 `0` 写入到为该变量预留的内存单元：

```
int MyFavoriteNumber = 0;
```

有时候，需要根据变量的值（通常是核实它不为零）做条件处理，如果不对变量进行初始化，这样的逻辑将不可靠，因为未赋值或初始化的变量包含的内容是随机的。

问：C++为何提供变量类型 `short int`、`int` 和 `long int`？为何不始终使用取值范围最大的变量类型呢？

答：C++用于编写各种应用程序，其中很多运行在计算能力和内存资源都很有限的设备上。例如，老式手机的计算能力和内存都有限。在这种情况下，程序员通过选择合适的变量类型，可节省内存并提高速度。如果编写的是常规台式机或高端智能手机程序，选择不同整型带来的性能提升或内存节省将很小，有时甚至可以忽略不计。

问：为何不应频繁地使用全局变量？全局变量在应用程序的任何地方都可用，可避免在函数之间传递值，从而节省一些时间，这种说法对吗？

答：可在应用程序的任何地方读取全局变量的值以及给它赋值，这是个问题，因为在应用程序的

任何地方都可修改它们。假设您与其他几位程序员合作开发一个项目，并将变量声明为全局的。如果队友不小心在其代码中修改这些变量——即使是在另一个.CPP文件中，都将影响代码的可靠性。因此，为确保代码的稳定性，不要为节省几秒钟甚至几分钟而不分青红皂白地使用全局变量。

问：在C++中，可以声明无符号整型变量，其取值只能是零或正整数。如果一个 unsigned int 变量的值为零，将其减 1 的结果如何？

答：这将导致环绕。如果将值为零的 unsigned int 变量减 1，它将环绕到可存储的最大值！从表 3.1 可知，unsigned short 变量的取值范围为 0~65535。为演示这一点，下面的代码声明了一个 unsigned short 变量，将其初始化为 0，并减 1：

```
unsigned short MyShortInt = 0; // Initial Value
MyShortInt = MyShortInt - 1; // Decrement by 1
std::cout << MyShortInt << std::endl; // Output: 65535!
```

这不是 unsigned short 的问题，而是使用它的方式有问题；在变量的取值可能为负时，就不应将其类型指定为 unsigned int、unsigned short 或 unsigned long。如果 MyShortInt 被用于指定动态分配的字节数，允许在它为零时减 1，将导致分配 64KB 的内存！更糟糕的是，如果在访问内存单元是将 MyShortInt 用作索引，很可能访问外部内存单元，进而导致应用程序崩溃！

## 3.10 作业

作业包括测验和练习，前者帮助读者加深对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 的答案。在继续学习下一章前，请务必弄懂这些答案。

### 3.10.1 测验

1. 有符号整型和无符号整型有何不同？
2. 为何不应使用#define 来声明常量？
3. 为何要对变量进行初始化？
4. 给定如下枚举类型，QUEEN 的值是多少？  
enum YOURCARDS {ACE, JACK, QUEEN, KING};
5. 下述变量名有何问题？  
int Integer = 0;

### 3.10.2 练习

1. 修改测验题 4 中的枚举类型 YOURCARDS，让 QUEEN 的值为 45。
2. 编写一个程序，证明 unsigned int 和 int 变量的长度相同，且它们都比 long 变量短。
3. 编写一个程序，让用户输入圆的半径，并计算其面积和周长。
4. 在练习 3 中，如果将面积和圆周存储在 int 变量中，输出将有何不同？
5. 查错：下面的语句有何错误？  
auto Integer;

## 第 4 章

# 管理数组和字符串

在前几章中，您声明存储单个 `int`、`char` 或字符串的变量。然而，您可能想声明一组对象，如 20 个 `int` 变量或一组 `Cat` 对象。

在本章中，您将学习：

- 什么是数组以及如何声明和使用它们；
- 什么是字符串以及如何使用字符数组来表示字符串；
- `std::string` 简介。

### 4.1 什么是数组

`array` 的字典定义与数组的概念很接近。韦氏字典指出，`array` 是一组元素，它们形成一个整体，如一组太阳能电池板。

数组具有如下特点：

- 数组是一系列元素；
- 数组中所有元素的类型都相同；
- 这组元素形成一个完整的集合。

在 C++ 中，数组让您能够按顺序将一系列相同类型的数据存储到内存中。

#### 4.1.1 为何需要数组

假设您要编写一个程序，它让用户输入 5 个整数并显示出来。为此，一种方式是声明 5 个独立的 `int` 变量，并使用它们来存储和显示值。声明类似于下面这样：

```
int FirstNumber = 0;
int SecondNumber = 0;
int ThirdNumber = 0;
int FourthNumber = 0;
int FifthNumber = 0;
```

采用这种方式时，如果用户希望这个程序存储并显示 500 个整数，您将需要声明 500 个 `int` 变量。只要有足够的耐心和时间，这还是可行的。然而，如果用户要求存储并显示 500000 个整数，您该怎么办呢？

您应采取正确而聪明的方式，声明一个包含 5 个 `int` 元素的数组，并将每个元素都初始化为零，如下所示：

```
int MyNumbers [5] = {0};
```

这样，当您被要求支持 500000 个整数时，便可以快速扩大数组，如下所示：



```
int ManyNumbers [500000] = {0};
```

要定义一个包含 5 个字符的数组，可以这样做：

```
char MyCharacters [5];
```

这样的数组被称为静态数组，因为在编译阶段，它们包含的元素数以及占用的内存量都是固定的。

### 4.1.2 声明和初始化静态数组

在前一小节，您声明了一个名为 `MyNumbers` 的数组，它包含 5 个类型为 `int` 的元素（即整数），这些元素都被初始化为 0。在 C++ 中，数组声明遵循如下简单的语法：

```
element-type array-name [number of elements] = {optional initial values}
```

在声明数组时，还可像下面这样分别初始化每个元素，这里将 5 个元素分别初始化为不同的整数：

```
int MyNumbers [5] = {34, 56, -21, 5002, 365};
```

可将数组的所有元素都初始化为相同的值，如下所示：

```
int MyNumbers [5] = {100}; // initialize all integers to 100
```

也可只初始化部分元素，如下所示：

```
int MyNumbers [5] = {34, 56}; // initialize first two elements
```

可将数组长度（即数组包含的元素数）定义为常量，并在数组定义中使用该常量：

```
const int ARRAY_LENGTH = 5;
```

```
int MyNumbers [ARRAY_LENGTH] = {34, 56, -21, 5002, 365};
```

需要在多个地方访问并使用数组的长度（如遍历数组中的元素）时，这很有用。这样就无需在每个地方修改数组的长度，而只需修改 `const int` 声明中的初始值。

#### 注意

如果您只初始化数组的部分元素，有些编译器可能将您忽略的元素初始化为零。

如果知道数组中每个元素的初始值，可不指定数组包含的元素数：

```
int MyNumbers [] = {2011, 2052, -525};
```

上述代码创建一个数组，它包含 3 个 `int` 元素，这些元素的初始值分别为 2011、2052 和 -525。

#### 注意

前面声明的所有数组都是静态数组，因为它们的长度在编译阶段就已确定。这种数组不能存储更多的数据；同时，即便有部分元素未被使用，它们占据的内存也不会减少。

### 4.1.3 数组中的数据是如何存储的

想想书架上放在一起的图书吧，这就是一个一维数组，因为它只沿一个维度延伸，这个维度就是元素数。每本书都是一个数组元素，而书架就像为存储这些图书而预留的内存，如图 4.1 所示。

这里给图书从零开始编号，这并非错误。后面您将看到，在 C++ 中，数组索引从零而不是 1 开始。类似于书架上的 5 本图书，包含 5 个整数的数组 `MyNumbers` 类似于图 4.2。

注意到这个数组占用的内存空间包含 5 块，每块的大小都相同。块大小取决于数组存储的数据类型，这里是 `int`。您可能还记得，第 3 章研究了 `int` 变量的长度，因此编译器为数组 `MyNumbers` 预留的内存量为 `sizeof(int) * 5`。一般而言，编译器为数组预留的内存量为（单位为字节）：

Bytes consumed by an array = sizeof(element-type) \* Number of Elements

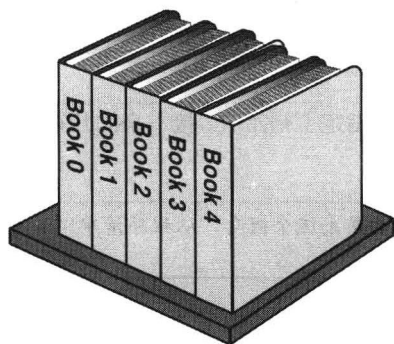


图 4.1 书架上的图书：一维数组

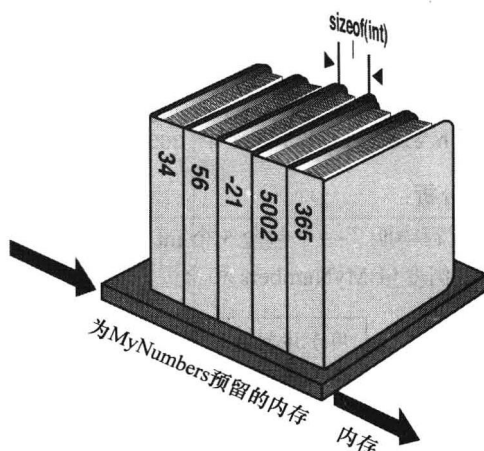


图 4.2 内存中包含 5 个整数的数组 MyNumbers

#### 4.1.4 访问存储在数组中的数据

要访问数组中的元素，可使用从零开始的索引。这些索引之所以被称为从零开始的，是因为数组中第一个元素的索引为零。因此，存储在数组 `MyNumbers` 中的第一个整数值为 `MyNumbers[0]`，第二个为 `MyNumbers[1]`，依此类推。第 5 个元素为 `MyNumbers[4]`，换句话说，数组中最后一个元素的索引总是比数组长度少 1。

被要求访问索引为 `N` 的元素时，编译器以第一个元素（索引为零）的内存地址为起点，加上偏移量 `N*sizeof(element)`，即向前跳 `N` 个元素，到达包含第 `N+1` 个元素的地址。C++ 编译器不会检查索引是否在数组的范围内，您可从只包含 10 个元素的数组中取回索引为 1001 的元素，但这样做将给程序带来安全和稳定性方面的风险。访问数组时，确保不超越其边界是程序员的职责。

#### 警告

访问数组时，如果超越其边界，结果将是无法预料的。在很多情况下，这将导致程序崩溃。应不惜一切代价避免访问数组时超越其边界。

程序清单 4.1 演示了如何声明一个 `int` 数组、初始化其元素并将元素的值显示到屏幕上。

#### 程序清单 4.1 声明一个 `int` 数组并访问其元素

```
0: #include <iostream>
1:
2: using namespace std;
3:
4: int main ()
5: {
6:     int MyNumbers [5] = {34, 56, -21, 5002, 365};
7:
8:     cout << "First element at index 0: " << MyNumbers [0] << endl;
9:     cout << "Second element at index 1: " << MyNumbers [1] << endl;
10:    cout << "Third element at index 2: " << MyNumbers [2] << endl;
11:    cout << "Fourth element at index 3: " << MyNumbers [3] << endl;
12:    cout << "Fifth element at index 4: " << MyNumbers [4] << endl;
13:
14:    return 0;
15: }
```

---

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ1779903665.

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我QQ1779903665。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

**声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，与本人无关，我们仅仅只是帮助你寻找到你要的pdf而已。**