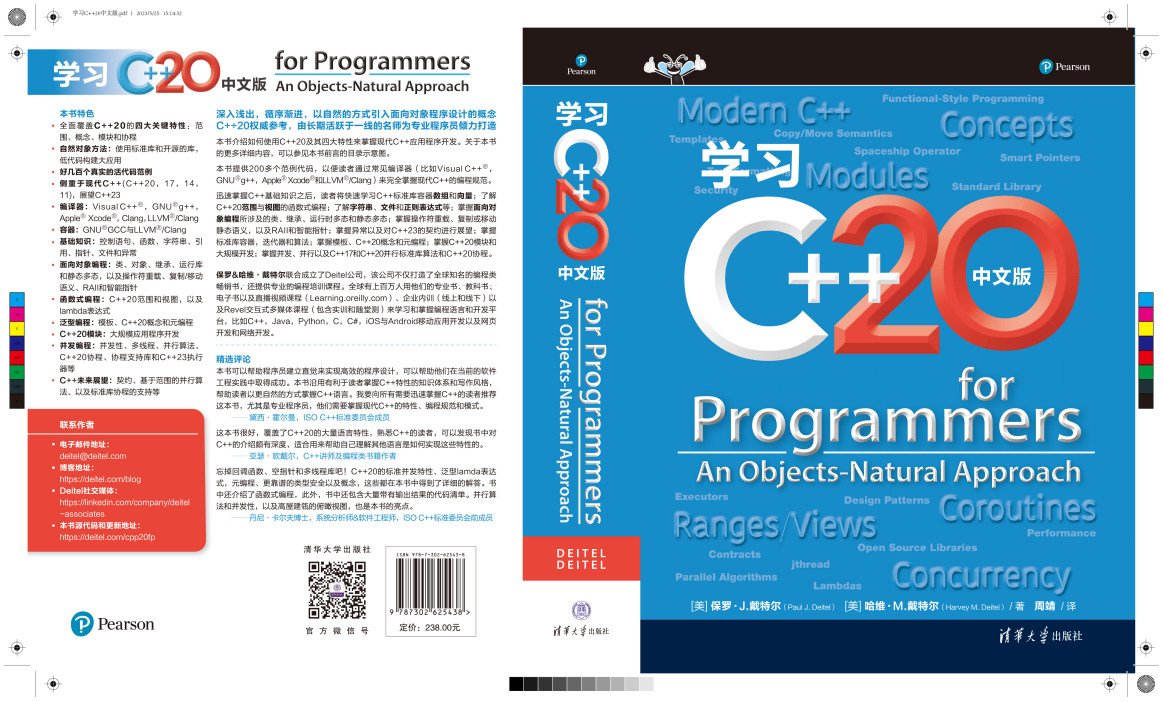
学习 C++20（中文版）

C++20 for Programmers: An Objects-Natural Approach

Paul Deitel / Harvey Deitel 著

周 靖 译



中文试读版 1~5 章和附录。翻译原稿，仅供参考，

更多精彩内容，请购买正版。

购买链接：京东 淘宝

配套资源和试读下载： ys168 网盘>> 百度网盘>>

访问中文版博客，提交评论和勘误

清华大学出版社

学习 C++20
C++20 for Programmers: An Objects-Natural Approach

致 ISO C++标准委员会的成员：

感谢你们为发展世界上最重要的高性能、关键任务和关键业务应用程序编程语言

而做的努力。

Paul Deitel

Harvey Deitel

本书特色：

- 涵盖了 C++20 的四大特性：范围、概念、模块和协程。
- 采用“对象自然”（Objects-Natural）法：使用标准库和开源库，以最少的代码构建功能齐全的应用程序
- 数以百计的真实世界、实时编码示例
- 现代 C++：涵盖 C++20、17、14、11，并展望了 C++23
- 编译器：Visual C++®, GNU® g++, Apple Xcode® Clang, LLVM®/Clang
- Docker：GNU® GCC, LLVM®/Clang
- 基础知识：控制语句、函数、字符串、引用、指针、文件、异常
- 面向对象编程：类、对象、继承、运行时和静态多态性、操作符重载、拷贝/移动语义、RAII、智能指针
- 函数式编程：C++20 的范围和视图，lambda 表达式
- 泛型编程：模板、C++20 概念和元编程
- C++20“模块”：大规模开发
- 并发编程：并发、多线程、并行算法、C++20 协程、协程支持库、C++23 执行器
- 未来：对契约的展望、基于范围的并行算法、标准库的协程支持等

目录

前言 **74**

P.1 现代 C++ *74*

保持话题性 74

C++版本 75

P.2 目标读者 *75*

P.3 “实时编码”方法和代码下载 *75*

P.4 三种工业强度的编译器 *75*

P.5 编程技巧和 C++20 关键特性 *76*

P.6 “对象自然”法 *76*

何谓对象自然？ 76

免费的类 77

Boost 项目 77

对象自然案例研究 77

P.7 本书导读 *78*

第 1 部分：编程基础知识快速入门 79

第 2 部分：数组、指针和字符串 80

第 3 部分：面向对象编程 81

第 4 部分：标准库容器、迭代器和算法 82

第 5 部分：高级主题 83

附录 84

P.8 如何访问在线章节和附录 *84*

在线章节 84

在线附录 85

deitel.com 提供的在线资料 85
P.9 C++核心准则 *85*
准则支持库 86
P.10 工业强度的静态代码分析工具 *86*
P.11 教学方法 *87*
使用不同字体进行强调 87
学习目标和大纲 87
表格和插图 87
编程技巧和关键特性 87
P.12 开发者资源 *88*
StackOverflow 88
GitHub 88
P.13 Docker *89*
P.14 一些关键的 C++文档和资源 *89*
文档 90
博客 90
其他资源 90
P.15 自己寻找答案 *91*
与作者沟通 91
P.16 加入 Deitel & Associates 公司的社交媒体社区 *91*
P.17 O'Reilly Online Learning 上的 Deitel Pearson 产品 *91*
教科书和专业书 92
LiveLessons 视频产品 92
现场 Full-Throttle 培训课程 92

P.18 讲师指导的现场培训 *92*

P.19 本书的大学教科书版 *93*

P.20 致谢 *93*

审稿人 94

Arthur O'Dwyer 94

GitHub 95

Matt Godbolt 和 Compiler Explorer 95

Dietmar Kühl 95

Rainer Grimm 95

Brian Goetz 95

开源贡献者和博客作者 96

谷歌搜索 96

Grammarly 96

作者简介 *97*

Deitel® & Associates 公司简介 *97*

准备工作 **99**

字体和命名约定 *99*

获得代码示例 *99*

本书使用的编译器 *100*

某些例子不能在全部三种编译器上编译和运行 *100*

在 Windows 上安装 Visual Studio 社区版 *100*

在 macOS 上安装 Xcode *101*

安装最新的 GNU C++版本 *101*

用 WSL 运行 Ubuntu Linux 并安装 GNU 编译器集合 *101*

Docker 和 Docker 容器 *102*

安装 Docker 102
下载 GNU Compiler Collection Docker 容器 103
下载 Clang Docker 容器 103
C++答疑解惑 103
在线 C++文档 104
关于{fmt}文本格式化库 104
静态代码分析工具 105
第 1 章 介绍免费、流行的 C++编译器并试运行 107
1.1 导读 107
本书的上层建筑 107
C++历史 108
1.2 试运行一个 C++20 应用程序 109
编译器和 IDE 试运行总结 109
1.2.1 在 Windows 上用 Visual Studio 2022 社区版编译和运行 C++20 应用程序 110
步骤 1：检查安装 110
步骤 2：启动 Visual Studio 110
步骤 3：创建项目 110
步骤 4：将 GuessNumber.cpp 文件添加到项目 111
步骤 5：配置项目以使用 C++20 111
步骤 6：编译和运行项目 111
步骤 7：输入第一次猜测 112
步骤 8：输入另一个猜测 112
步骤 9：输入其他猜测 112
步骤 10：再玩一次游戏或退出应用程序 113

在后续的例子中重用该项目 113
在 Linux 的 Windows 子系统中使用 Ubuntu Linux 113
1.2.2 在 macOS 上用 Xcode 编译和运行 C++20 应用程序 114
步骤 1：检查安装 114
步骤 2：启动 Xcode 114
步骤 3：创建项目 114
步骤 4：配置项目以使用 C++20 115
步骤 5：从项目中删除 main.cpp 文件 115
步骤 6：将 GuessNumber.cpp 文件添加到项目 115
步骤 7：编译和运行项目 116
步骤 8：输入第一次猜测 116
步骤 9：输入另一个猜测 116
步骤 10：输入其他猜测 117
步骤 11：再玩一次游戏或退出应用程序 117
在后续的例子中重用该项目 117
1.2.3 在 Linux 上用 GNU C++运行 C++20 应用程序 118
步骤 1：切换到 ch01 文件夹 118
步骤 2：编译应用程序 118
步骤 3：运行应用程序 118
步骤 4：输入第一次猜测 119
步骤 5：输入另一个猜测 119
步骤 6：输入其他猜测 119
步骤 7：再玩一次游戏或退出应用程序 120
1.2.4 在 GCC Docker 容器中编译和运行 C++20 应用程序 120
执行 GNU 编译器集合（GCC）Docker 容器 120

在 Docker 容器中切换到 ch01 文件夹 121
终止 Docker 容器 121
1.2.5 在 Docker 容器中使用 clang++来编译和运行 C++20 应用程序 122
执行 teeks99/clang-ubuntu Docker 容器 122
在 Docker 容器中切换到 ch01 文件夹 123
编译应用程序 123
运行应用程序 123
终止 Docker 容器 123
1.3 摩尔定律、多核处理器和并发编程 124
摩尔定律 124
多核处理器和性能 124
1.4 面向对象简单回顾 125
汽车作为对象 125
函数、成员函数和类 126
实例化 126
重用 126
消息和成员函数调用 126
属性和数据成员 127
封装 127
继承 127
面向对象分析与设计 127
1.5 小结 128
第 2 章 C++编程入门 129
2.1 导读 129

“对象自然”学习方法 129
编译和运行程序 130
2.2 第一个 C++程序：显示单行文本 *130*
注释 130
#include 预处理指令 131
空行和空白 131
main 函数 131
输出语句 131
缩进 132
std 命令空间 132
流插入操作符和转义序列 132
return 语句 133
2.3 修改第一个 C++程序 *133*
多个语句显示单行文本 133
单个语句显示多行文本 134
2.4 另一个 C++程序：整数相加 *134*
变量声明和大括号初始化 135
一次性声明多个变量 136
基本类型 136
标识符和驼峰式命名法 136
变量声明位置 136
从用户处获取第一个值 137
从用户处获取第二个值 137
计算用户输入值之和 138
显示结果 138

2.5 算术运算 *139*
整数除法 139
求余操作符 139
用圆括号对子表达式进行分组 139
操作符优先级规则 140
操作符分组 140
2.6 决策：相等性和关系操作符 *140*
混淆==和= 141
使用 `if` 语句 141
using 声明 143
using 指令 143
变量声明和从用户处读取输入 143
比较数字 144
大括号和代码块 144
常见逻辑错误：在条件之后添加分号 144
分割冗长的语句 145
操作符优先级和分组 145
2.7 对象自然案例研究：创建和使用标准库类 `string` 的对象 *145*
使用 `string` 类 145
实例化对象 147
`string` 的 `length` 成员函数 147
用相等性操作符比较字符串对象 148
string 的 empty 成员函数 148
字符串连接和赋值 148

C++20 string 的成员函数 starts_with 和 ends_with 149
2.8 小结 *149*
第 3 章 控制语句(第 1 部分) **150**
3.1 导读 *150*
3.2 控制结构 *150*
3.2.1 顺序结构 151
3.2.2 选择语句 152
3.2.3 循环语句 152
关键字 152
3.2.4 控制语句小结 154
3.3 `if` 选择语句 *154*
`bool` 数据类型 155
`if` 语句的 UML 活动图 155
3.4 `if…else` 双选语句 *155*
`if…else` 语句的 UML 活动图 156
3.4.1 嵌套 `if…else` 语句 156
3.4.2 代码块 157
空语句 157
3.4.3 条件操作符(`?:`) 158
3.5 `while` 循环语句 *158*
`while` 语句的 UML 活动图 158
3.6 计数器控制的循环 *159*
3.6.1 实现计数器控制的循环 159
`main` 中的局部变量 160
初始化 `total` 和 `gradeCounter` 变量 161

从用户处读入 10 个成绩 161
计算并显示班级平均成绩 161
3.6.2 整数除法和截断 161
3.7 哨兵值控制的循环 *161*
3.7.1 实现哨兵值控制的循环 162
哨兵值控制的循环与计数器控制的循环的程序逻辑 163
3.7.2 基础类型之间的显式和隐式转换 164
`static_cast` 操作符 164
提升 165
任意类型的强制类型转换操作符 165
3.7.3 格式化浮点数 165
`setprecision` 参数化流操纵元 165
`fixed` 非参数化流操纵元 165
浮点数的四舍五入 166
3.8 嵌套控制语句 *166*
3.8.1 问题陈述 166
3.8.2 实现程序 167
3.8.3 用大括号初始化防止收缩转换 169
回顾图 3.1 170
3.9 复合赋值操作符 *170*
3.10 递增和递减操作符 *171*
对比前缀递增和后缀递增 172
用算术复式赋值、递增和递减操作符简化语句 173
操作符优先级和分组 173

3.11 基本类型不可移植 *174*

3.12 对象自然案例研究：任意大小的整数 *175*

一些应用需要 long long 范围以外的数字 175

使用 BigNumber 类来实现任意精度的整数 175

包含不在 C++标准库中的头文件 177

超出 long long 整数的取值范围会发生什么？ 177

用 BigNumber 对象执行同样的操作 178

用 long long 整数计算 100 000 000 的乘方 178

用 BigNumber 对象计算 100 000 000 的乘方 178

在 Microsoft Visual Studio 中编译和运行示例 179

在 GNU g++中编译和运行示例 179

在 Apple Xcode 中编译和运行示例 179

3.13 C++20：用 format 函数格式化文本 *180*

C++20 字符串格式化尚未全面实现 180

格式字符串占位符 181

点位符从左到右替换 181

在 Microsoft Visual Studio 中编译和运行示例 181

在 GNU g++中编译和运行示例 182

在 Apple Xcode 中编译和运行示例 182

3.14 小结 *182*

第 4 章 控制语句(第 2 部分) **183**

4.1 导读 *183*

4.2 计数器控制的循环的本质 *183*

4.3 for 循环语句 *184*

剖析 for 语句头 185

for 语句的常规格式 186
for 语句的控制变量的作用域 186
for 语句头中的表达式是可选的 186
在语句的主体中使用 for 语句的控制变量 187
for 语句的 UML 活动图 187
4.4 for 循环的例子 187
4.5 应用程序：累加偶数整数 188
4.6 应用程序：复利计算 189
用域宽和对齐来格式化 190
用标准库函数 pow 计算利息 191
浮点数精度和内存需求 191
浮点数是近似值 192
关于显示舍入值的警告 192
即使是普通的美元金额也会出现浮点表示误差 193
4.7 do…while 循环语句 193
do…while 循环语句的 UML 活动图 193
4.8 switch 多选语句 194
用 switch 语句统计 A，B，C，D 和 F 成绩数 194
从用户处取成绩 197
处理成绩 197
不用 break 的 case——C++17 的[[fallthrough]]特性 198
default case 199
报告成绩 199
switch 语句的 UML 活动图 199

关于 case 的注意事项 200
4.9 使用带初始化器的 C++17 选择语句 201
带初始化器的选择语句的语法 201
初始化器所声明的变量的作用域 201
4.10 break 和 continue 语句 202
break 语句 202
continue 语句 203
4.11 逻辑操作符 204
4.11.1 逻辑 AND(&&)操作符 204
4.11.2 逻辑 OR(||)操作符 205
4.11.3 短路求值 205
4.11.4 逻辑非(!)操作符 206
4.11.5 示例：生成逻辑操作符真值表 206
到目前为止的操作符的优先级和分组 207
4.12 混淆相等性(==)和赋值(=)操作符 208
左值和右值 209
该用=的时候用了== 209
启用警告 210
4.13 对象自然案例研究：使用 miniz-cpp 库读写 ZIP 文件 210
ZIP 文件 210
开源 miniz-cpp 库 211
用 getline 从用户处输入一行文本 212
创建样本内容并在 ZIP 文件写入一个单独的文件 212
创建一个 zip_file 对象 213
在 zip_file 对象中创建文件，并将该对象存储到磁盘 213

ZIP 文件似乎包含的是随机符号 213
读取 ZIP 压缩包的内容 213
显示 ZIP 文件的名称和内容 214
获取并显示 ZIP 压缩包中一个特定文件的信息 214
解压 intro.txt 并显示其原始内容 215
4.14 用域宽和精度进行 C++20 文本格式化 215
格式化本金和利率 216
格式化 Year 和 Amount-on-Deposit 列标题 217
在 for 循环中格式化 Year 和 Amount-on-Deposit 列的值 217
4.15 小结 217
第 5 章 函数和函数模板入门 219
5.1 导读 219
5.2 C++程序组件 220
5.3 数学库函数 220
C++11 引入的数学函数 221
C++20 新增的数学常数和<numbers>头文件 222
C++17 数学特殊函数 222
5.4 函数定义和函数原型 223
maximum 函数 225
maximum 的函数原型 225
函数原型中的参数名称 225
编译器用 maximum 的函数原型做什么？ 225
将控制从函数返回给调用者 226
5.5 函数实参的求值顺序 226

5.6 关于函数原型和实参强制类型转换的注意事项 *226*
5.6.1 函数签名和函数原型 227
5.6.2 实参强制类型转换 227
5.6.3 实参提升规则和隐式转换 227
转换可能造成不正确的值 229
配合“准则支持库”的收缩转换 229
5.7 C++标准库头文件 *229*
5.8 案例研究：随机数生成 *232*
<random>头文件 232
5.8.1 掷六面骰子 233
5.8.2 六面骰子掷 6000 万次 234
5.8.3 为给随机数生成器提供种子 236
为 default_random_engine 提供种子 236
5.8.4 用 random_device 为随机数生成器提供种子 237
5.9 案例研究：概率游戏；介绍有作用域的 enum *238*
实现游戏 239
C++11 有作用域的 enum 240
第一次掷骰的输赢 240
继续掷骰 241
显示玩家是赢还是输 242
rollDice 函数 242
有作用域的 enum 的其他注意事项 243
C++11 之前的枚举类型 243
C++11：指定枚举常量的类型 243
C++20：使用 enum 声明 244

5.10 作用域规则 *244*
块作用域 244
全局命名空间作用域 245
演示作用域 245
`main` 函数 246
`useLocal` 函数 247
`useStaticLocal` 函数 248
`useGlobal` 函数 248
5.11 内联函数 *248*
5.12 引用和引用参数 *249*
引用参数 250
通过传值和传引用来传递实参 250
将引用作为函数中的别名 251
`const` 引用 252
返回对局部变量的引用可能会很危险 252
5.13 默认参数 *253*
带默认参数的 `boxVolume` 函数 253
默认参数注意事项 254
5.14 一元作用域解析操作符 *254*
5.15 函数重载 *255*
重载的 `square` 函数 255
编译器如何区分不同的重载版本？ 256
重载解析详情 258
重载的操作符 258

5.16 函数模板 *258*

maximum 函数模板 259

使用 maximum 函数模板 259

针对 int 类型的 maximum 函数模板特化 261

5.17 递归 *261*

递归的概念 261

阶乘 262

以循环方式求阶乘 262

以递归方式求阶乘 262

求值 5! 263

递归调用 factorial 函数来计算阶乘 263

阶乘值快速增长 264

5.18 递归示例：斐波那契数列 *265*

斐波那契的递归定义 265

求值 fibonacci(3) 267

操作数的求值顺序 267

指定了求值顺序的操作符 267

指数级复杂度 268

5.19 对比递归和循环 *268*

阶乘的循环版本 269

递归的缺点 270

递归和循环的选择时机 270

5.20 Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz *270*

密码学 270

凯撒密码 271

维吉尼亚密码 271
使用现成的 Cipher 类 272
Cipher 类的成员函数 273
5.21 小结 *273*
第 6 章 数组、向量、范围和函数式编程 275
6.1 导读 *275*
完全限定的标准库名称 276
6.2 数组 *276*
6.3 声明数组 *276*
6.4 用循环初始化数组元素 *277*
用循环向数组元素赋值 278
size_t 类型 278
显示数组元素 278
避免安全漏洞：数组索引的边界检查 279
6.5 用初始化器列表初始化数组 *279*
初始化值少于数组元素 280
初始化值多于数组元素 281
6.6 C++11 基于范围的 for 和 C++20 带初始化器的基于范围的 for *281*
用基于范围的 for 显示数组内容 282
使用基于范围的 for 修改数组内容 283
C++20 带初始化器的基于范围的 for 283
避免索引 283
外部迭代与内部迭代 283
6.7 计算数组元素值并理解 constexpr *284*

常量 284
constexpr 和编译时 285
6.8 累加数组元素 285
6.9 使用简陋的条形图以图形方式显示数组数据 286
6.10 数组元素作为计数器使用 288
6.11 使用数组来汇总调查结果 289
6.12 数组排序和查找 291
演示 sort 和 binary_search 函数 291
6.13 多维数组 293
声明由数组构成的数组 294
显示由数组构成的数组 295
嵌套的基于范围的 for 语句 295
嵌套的、由计数器控制的 for 语句 295
用完整的大括号初始化列表来初始化由数组构成的数组 296
常见二维数组操作：设置一行的值 296
常见二维数组操作：用嵌套的、由计数器控制的 for 循环求所有元素之和 296
常见二维数组操作：用嵌套的、基于范围的 for 循环求所有元素之和 297
C++23 mdarray 297
6.14 函数式编程入门 297
6.14.1 做什么和怎么做 297
用 accumulate 进行函数式归约 298
6.14.2 函数作为实参传给其他函数：理解 lambda 表达式 298
用一个具名函数调用 accumulate 299
用 lambda 表达式调用 accumulate 300
6.14.3 过滤器、映射和归约：理解 C++20 的“范围”库 300

用于显示程序运行结果的 showValues lambda 301
用 views::iota 生成一个连续的整数范围 301
用 views::filter 过滤数据项 302
用 views::transform 映射数据项 303
将过滤和映射操作合并到一个管道中 303
用 accumulate 归约范围管道 304
过滤和映射现有容器的元素 304
6.15 对象自然案例研究：C++标准库类模板 vector *304*
创建 vector 对象 305
vector 成员函数 size 和 outputVector 函数 306
inputVector 函数 306
判断两个 vector 是否不相等 307
用一个 **vector** 的内容初始化另一个 **vector** 307
vector 赋值和 vector 的相等性比较 308
使用 at 成员函数访问和修改 vector 的元素 308
异常处理：处理超出范围的索引 309
try 语句 310
执行 catch 块 310
异常对象的 what 成员函数 310
更改 vector 的大小 311
outputVector 和 inputVector 函数 311
直线型代码 312
6.16 小结 *312*
第 7 章 现代 C++对指针的淡化 **313**

7.1 导读 *313*
在现代 C++中淡化指针 313
有时仍然需要指针 314
和指针有关的“C++核心准则” 314
C++20 如何帮你避免使用指针 314
本章介绍的其他概念 315
7.2 声明和初始化指针变量 *315*
7.2.1 声明指针 315
7.2.2 初始化指针 316
7.2.3 C++11 之前的空指针 316
7.3 指针操作符 *316*
7.3.1 取址(&)操作符 316
7.3.2 间接寻址(*)操作符 317
未定义的行为 317
7.3.3 使用取址(&)和间接寻址(*)操作符 318
7.4 用指针传引用 *319*
传值的例子 319
用指针传引用的例子 320
事实：“用指针传引用”实际是“以传值方式传指针” 321
图示传引用和传值 321
7.5 内置数组 *323*
7.5.1 声明和访问内置数组 323
7.5.2 初始化内建数组 324
7.5.3 向函数传递内置数组 324
7.5.4 声明内置数组参数 324

7.5.5 C++11 标准库函数 begin 和 end 325
7.5.6 内置数组的限制 325
7.6 使用 C++20 to_array 将内置数组转换成 std::array 326
用 to_array 从内置数组创建 std::array 327
使用 to_array 从初始化列表创建 std::array 327
7.7 为指针和它指向的数据使用 const 327
7.7.1 指向非常量数据的非常量指针 328
7.7.2 指向常量数据的非常量指针 328
7.7.3 指向非常量数据的常量指针 329
7.7.4 指向常量数据的常量指针 330
7.8 sizeof 操作符 331
判断基本类型、内置数组和指针的大小 331
7.9 指针表达式和指针算术 333
7.9.1 在指针上加减整数 334
7.9.2 从指针上减一个指针 335
7.9.3 指针赋值 335
7.9.4 不能解引用 void* 335
7.9.5 指针比较 336
7.10 对象自然案例研究：C++20 span——连续容器元素的视图 336
displayArray 函数 337
displaySpan 函数 337
times2 函数 338
将数组传给函数以显示内容 339
隐式创建 span 并把它们传给函数 339

修改 span 的元素会同时修改原始数据 340

手动创建一个 span 并与之交互 340

为 span 使用标准库的 accumulate 算法 341

创建子视图 341

```
68      // spans can be used to create subviews of a container
69      std::cout << "\n\nFirst three elements of mySpan: ";
70      displaySpan(mySpan.first(3));
71      std::cout << "\nLast three elements of mySpan: ";
72      displaySpan(mySpan.last(3));
73      std::cout << "\nMiddle three elements of mySpan: ";
74      displaySpan(mySpan.subspan(1, 3));
75
```

```
First three elements of mySpan: 2 4 6
Last three elements of mySpan: 6 8 10
Middle three elements of mySpan: 4 6 8
```

修改子视图的元素会同时修改原始数据 342

通过[]操作符访问视图的元素 342

7.11 理解基于指针的字符串 342

字符和字符常量 343

基于指针的字符串 343

字符串字面值作为初始化器 343

C 字符串的问题 344

显示 C 字符串 344

7.11.1 命令行参数 344

7.11.2 再论 C++20 的 to_array 函数 346

从字符串字面值初始化一个 std::array 来创建单元素 array 346

将字符串字面值传给 to_array 来创建由 char 构成的一个 std::array 347

7.12 展望其他指针主题 347

7.13 小结 347

第 8 章 string、string_view、文本文件、CSV 文件和正则表达式 349

8.1 导读 349

std::string 349

C++17 string_view 349

文本文件和字符串流处理 349

对象自然案例研究：CSV 文件和泰坦尼克号灾难数据集 350

对象自然案例研究：使用正则表达式来查找字符串的模式、校验数据和替换子串 350

8.2 字符串赋值和连接 350

字符串赋值 351

按索引访问字符串中的元素 352

字符串连接 352

8.3 字符串比较 352

用关系和相等性操作符来比较字符串 354

用成员函数 compare 比较字符串 354

8.4 子串 354

8.5 交换字符串 355

8.6 收集 string 特征信息 356

C++20 对 string 成员函数 reserve 的更新 358

8.7 在字符串中查找子串和字符 358

成员函数 find 和 rfind 359

成员函数 find_first_of 359

成员函数 find_last_of 360

成员函数 find_first_not_of 360

8.8 替换和删除字符串中的字符 361

8.9 在字符串中插入字符 *363*

8.10 C++11 数值转换 *363*

数值转换为 string 对象 363

string 对象转换为数值 364

将 string 转换为整型的函数 364

将 string 转换为浮点类型的函数 365

8.11 C++17 string_view *365*

创建 string_view 366

string_view 能“看到”对所查看字符的更改 366

string_view 可与 std::string 或其他 string_view 比较 367

string_view 可以移除前缀或后缀 367

string_view 是可迭代的 367

string_view 支持对 C 字符串的多种 string 操作 368

8.12 文件和流 *368*

文件处理流 369

8.13 创建顺序文件 *369*

打开文件 370

通过 open 成员函数打开文件 371

测试文件是否成功打开 371

处理数据 372

关闭文件 372

示例执行 373

8.14 从顺序文件读取数据 *373*

打开文件进行输入 374

从文件中读取 374

文件位置指针 374
8.15 C++14 读取和写入引号文本 *375*
读取引号文本 376
写入带引号的文本 376
8.16 更新顺序文件 *376*
8.17 字符串流处理 377
演示 ostringstream 377
演示 istringstream 378
8.18 原始字符串字面值 *380*
8.19 对象自然案例研究：读取和分析包含泰坦尼克号灾难数据的 CSV 文件 *381*
数据集 381
account.csv 381
8.19.1 使用 rapidcsv 读取 CSV 文件的内容 381
警告：CSV 数据字段中的逗号 383
警告：CSV 文件中缺失和多余的逗号 383
8.19.2 读取和分析泰坦尼克号灾难数据集 383
了解数据 384
缺失的数据 384
加载数据集 384
加载数据以进行分析 385
查看泰坦尼克号数据集中的一些行 386
基本的描述性统计 387
清理后的年龄列的基本描述性统计 387
确定各种舱位的乘客人数 388

存活列的基本描述性统计 389
按性别和舱位统计幸存人数 390
计算幸存者百分比 391
8.20 对象自然案例研究：理解正则表达式 391
数据校验 391
正则表达式的其他用途 392
支持的正则表达式“流派” 392
8.20.1 将完整字符串与模式相匹配 392
匹配字符字面值 393
元字符、字符类和量词 393
匹配数字 394
自定义字符类 394
*和+量词 395
其他量词 396
8.20.2 替换子串 397
8.20.3 查找匹配 398
在字符串的任何地方寻找匹配 398
忽略正则表达式中的大小写并查看匹配文本 399
查找字符串中的所有匹配 400
8.21 小结 401
第 9 章 自定义类 402
9.1 导读 402
9.2 体验 Account 对象 402
实例化对象 403
头文件和源代码文件 404

调用 Account 类的 getName 成员函数 404
调用 Account 类的 setName 成员函数 404
显示用户输入的姓名 404
9.3 具有赋值和取值成员函数的 Account 类 405
9.3.1 类定义 405
关键字 class 和类的主体 405
std::string 类型的数据成员 m_name 406
在头文件中通过 std::使用标准库的组件 406
setName 成员函数 406
getName 成员函数 406
const 成员函数 407
9.3.2 访问说明符 private 和 public 407
类成员的默认访问级别 408
9.4 Account 类：自定义构造函数 408
Account 类的定义 408
Account 类的自定义构造函数 409
explicit 关键字 410
在创建 Account 对象时初始化它们 410
默认构造函数 411
类只要显式定义了构造函数就没有默认构造函数 412
C++的特殊成员函数 412
9.5 赋值和取值成员函数的软件工程优势 412
含有封装数据的 Account 类的概念性视图 413
9.6 含有余额的 Account 类 413

数据成员 balance 414
双参数构造函数 415
deposit 成员函数 415
getBalance 成员函数 415
用余额操纵 Account 对象 415
从用户处读取存入金额并进行存款 416
9.7 Time 类案例研究：分离接口与实现 417
C++20“模块”改变了分离接口与实现的方式 418
9.7.1 类的接口 418
9.7.2 分离接口与实现 418
9.7.3 类定义 419
#pragma once 420
9.7.4 成员函数 420
9.7.5 在源代码文件中包含类的头文件 421
9.7.6 作用域解析操作符(::) 421
9.7.7 成员函数 setTime 和抛出异常 422
9.7.8 成员函数 to24HourString 和 to12HourString 422
9.7.9 隐式内联的成员函数 423
9.7.10 成员函数与全局函数 423
9.7.11 使用 Time 类 423
用无效值调用 setTime 425
9.7.12 对象的大小 425
9.8 编译和链接过程 425
编译包含两个或多个源代码文件的程序 426
9.9 类作用域以及对类成员的访问 426

点(.)和箭头(->)成员选择操作符 427
通过对象、引用和指针访问公共类成员 427
9.10 访问函数和实用函数 427
访问函数 427
实用函数 427
9.11 Time 类案例研究：带有默认参数的构造函数 428
9.11.1 Time 类 428
Time 类的构造函数定义 429
测试更新的 Time 类 431
关于 Time 类的赋值和取值函数以及构造函数的软件工程说明 433
9.11.2 重载构造函数和 C++11 委托构造函数 433
9.12 析构函数 434
9.13 什么时候调用构造函数和析构函数 435
全局作用域中的对象的构造函数和析构函数 435
非静态局部对象的构造函数和析构函数 435
静态局部对象的构造函数和析构函数 435
演示在什么时候调用构造函数和析构函数 435
9.14 Time 类案例研究：返回到 private 数据成员的引用或指针时要小心 438
9.15 默认赋值操作符 441
拷贝构造函数 443
9.16 const 对象和 const 成员函数 443
使用 const 和非 const 成员函数 444
9.17 合成：对象作为类成员 445
Employee 构造函数的成员初始化列表 449

Date 类的默认拷贝构造函数 450
测试 Date 类和 Employee 类 450
不使用成员初始化列表会发生什么？ 451
9.18 友元函数和友元类 451
声明友元 452
友元规则 452
友元不受访问说明符的影响 452
用友元函数修改类的私有数据 452
9.19 this 指针 453
使用 this 指针避免命名冲突 454
this 指针的类型 454
9.19.1 隐式和显式使用 this 指针访问对象的数据成员 454
9.19.2 使用 this 指针来实现级联函数调用 456
9.20 静态类成员：类级数据和成员函数 459
什么时候需要类级数据 459
静态数据成员的作用域和初始化 460
访问静态数据成员 460
演示静态数据成员 460
静态成员函数的注意事项 464
9.21 C++20 中的聚合 464
9.21.1 初始化聚合 465
9.21.2 C++20：指定初始化器 466
指定初始化器的其他好处 466
9.22 对象自然案例研究：用 JSON 序列化 466
JSON 数据格式 466

序列化 467
序列化的安全性 467
cereal 单文件序列化库 468
9.22.1 序列化由包含 public 数据的对象构成的 vector 468
聚合类型 Record 468
Record 对象的 serialize 函数 469
displayRecords 函数 470
创建要序列化的 Record 对象 470
用 cereal::JSONOutputArchive 序列化 Record 对象 470
records.json 的内容 471
用 cereal::JSONInputArchive 反序列化 Record 对象 471
9.22.2 序列化由包含 private 数据的对象构成的 vector 472
9.23 小结 *475*
第 10 章 OOP：继承和运行时多态性 477
has-a 和 is-a 关系 477
运行时多态性 478
实现可扩展性 478
讨论用虚函数实现的运行时多态性的幕后机制 478
非虚接口惯用法 478
接口和依赖注入 479
使用 std::variant 和 std::visit 实现运行时多态性 479
多继承 479
实现多态性的其他方式 479
本章目标 479

10.2 基类和派生类 *480*
10.2.1 CommunityMember 类层次结构 480
10.2.2 Shape 类层次结构和 public 继承 481
10.3 基类和派生类的关系 *482*
10.3.1 创建和使用 SalariedEmployee 类 482
SalariedEmployee 构造函数 484
SalariedEmployee 的成员函数 earnings 和 toString 485
测试 SalariedEmployee 类 485
10.3.2 创建 SalariedEmployee/SalariedCommissionEmployee 继承层次结构 486
继承关系 486
SalariedCommissionEmployee 的成员函数 487
SalariedCommissionEmployee 类的实现 487
SalariedCommissionEmployee 的成员函数 earnings 489
SalariedCommissionEmployee 的成员函数 toString 489
测试 SalariedCommissionEmployee 类 489
派生类构造函数必须调用基类构造函数 490
派生类构造函数的注意事项 491
基类 private 成员在派生类中不能直接访问 491
在派生类头文件中包含基类头文件 491
通过“实现继承”消除重复代码 492
10.4 派生类中的构造函数和析构函数 *492*
构造函数调用顺序 492
析构函数调用顺序 492
合成对象的构造函数和析构函数 493
10.5 运行时多态性入门：多态性电子游戏 *493*

屏幕管理器 493
向系统添加新类 493
10.6 继承层次结构中对象之间的关系 494
10.6.1 从派生类对象调用基类函数 494
创建对象并显示其内容 496
基类指针指向基类对象 496
派生类指针指向派生类对象 497
基类指针指向派生类对象 497
10.6.2 派生类指针指向基类对象 497
10.6.3 通过基类指针调用派生类成员函数 498
10.7 虚函数和虚析构函数 499
10.7.1 为什么虚函数这么有用？ 499
10.7.2 声明虚函数 500
10.7.3 调用虚函数 500
10.7.4 `SalariedEmployee` 层次结构中的虚函数 500
运行时多态性行为 501
不要从构造函数和析构函数中调用虚函数 503
C++11 `override` 关键字 504
10.7.5 虚析构函数 504
10.7.6 final 成员函数和类 504
10.8 抽象类和纯虚函数 506
10.8.1 纯虚函数 506
10.8.2 设备驱动程序：操作系统中的多态性 506
10.9 案例研究：使用运行时多态性的薪资系统 507

10.9.1 创建抽象基类 Employee 508
Employee 类的头文件 509
Employee 类成员函数定义 510
10.9.2 创建派生的具体类 SalariedEmployee 510
SalariedEmployee 类的成员函数定义 511
10.9.3 创建派生的具体类 CommissionEmployee 512
10.9.4 演示运行时多态性处理 515
创建包含 Employee 指针的一个 vector 517
virtualViaPointer 函数 517
virtualViaReference 函数 518
10.10 运行时多态性、虚函数和动态绑定的幕后机制 518
虚函数表 518
Employee 类的 vtable 519
SalariedEmployee 类的 vtable 519
CommissionEmployee 类的 vtable 520
继承具体虚函数 520
用于实现运行时多态性的三级指针 520
10.11 非虚接口(NVI)惯用法 521
重构 Employee 类以应用 NVI 惯用法 522
更新的 SalariedEmployee 类 524
更新的 CommissionEmployee 类 526
使用 NVI 的 Employee 层次结构的运行时多态性 529
10.12 藉由接口来编程，而不要藉由实现 530
重新思考 Employee 层次结构：合成和依赖注入 530
接口继承最灵活 531

10.12.1 重新思考 Employee 层次结构：CompensationModel 接口 531
10.12.2 Employee 类 532
10.12.3 实现 CompensationModel 534
CompensationModel 的 Salaried 派生类 534
CompensationModel 的 Commission 派生类 535
10.12.4 测试新层次结构 537
10.12.5 依赖注入在设计上的优势 538
CompensationModel 发生变化时的灵活性 538
员工升职时的灵活性 538
对 Employee 类进行增强时的灵活性 538
10.13 使用 std::variant 和 std::visit 实现运行时多态性 539
Salaried 收入模型 539
Commission 收入模型 540
Employee 类定义 542
类型安全的 union 543
Employee 构造函数和 setCompensationModel 成员函数 543
Employee 的 earnings 和 toString 成员函数：用 std::visit 调用成员函数 544
测试用 std::variant 和 std::visit 实现的运行时多态性 544
10.14 多继承 546
多继承示例 546
解决派生类从多个基类继承同名成员函数时存在歧义的问题 548
测试多继承层次结构 548
演示多继承中的 is-a 关系 550
10.14.1 菱形继承 550

菱形继承中出现歧义时产生的编译错误 551
10.14.2 用虚基类继承消除重复的子对象 553
具有虚基类的多继承层次结构中的构造函数 555
10.15 深入理解 protected 类成员 555
protected 数据的问题 556
protected 基类成员函数 556
10.16 public、protected 和 private 继承 557
对比类和结构体的默认继承 558
10.17 更多运行时多态性技术和编译时多态性 558
10.17.1 其他运行时多态性技术 558
运行时概念惯用法 559
带有类型擦除的运行时多态性 559
运行时类型信息(RTTI) 560
10.17.2 编译时(静态)多态性技术 560
替换失败并非错误(SFINAE) 561
C++20“概念” 562
Tag Dispatch 562
10.17.3 其他多态性概念 562
10.18 小结 563
第 11 章 操作符重载、拷贝/移动语义和智能指针 565
11.1 导读 565
演示 string 类的重载操作符 566
动态内存管理和智能指针 566
MyArray 案例研究 566
C++20 的三路比较操作符(<=>) 567

转换操作符 567
11.2 使用标准库 string 类的重载操作符 567
创建 string 和 string_view 对象并用 cout 和操作符<<来显示 567
用相等性和关系操作符比较 string 对象 568
string 的 empty 成员函数 569
string 拷贝赋值操作符 569
字符串连接和 C++14 string 对象字面值 569
string 的 substr 成员函数 570
string 拷贝构造函数 571
用 string 拷贝赋值操作符测试自赋值 571
用一个 string_view 初始化 string 571
string 的[]操作符 572
string 的 at 成员函数 572
11.3 操作符重载基础 573
11.3.1 操作符不会自动重载 573
11.3.2 不能重载的操作符 573
11.3.3 不必重载的操作符 574
11.3.4 操作符重载的规则和限制 574
11.4 用 new 和 delete 进行动态内存管理(过时技术) 575
老办法：操作符 new 和 delete 575
用 new 获取动态内存 575
用 delete 释放动态内存 576
初始化动态分配的对象 576
用 new[]动态分配内置数组 576

用 delete[]释放动态分配的内置数组 577
基于范围的 for 对动态分配的内置数组不起作用 577
11.5 现代 C++动态内存管理：RAII 和智能指针 577
11.5.1 智能指针 578
11.5.2 演示 unique_ptr 578
11.5.3 unique_ptr 的所有权 580
11.5.4 指向内置数组的 unique_ptr 580
11.6 MyArray 案例研究：通过操作符重载来打造有价值的类 581
11.6.1 特殊成员函数 582
11.6.2 使用 MyArray 类 582
getArrayByValue 函数 582
创建 MyArray 对象并显示它的大小和内容 583
用圆括号而不是大括号来调用构造函数 584
使用重载流提取操作符来填充 MyArray 对象 584
使用重载的不相等操作符(!=) 584
用现有 MyArray 的拷贝初始化一个新 MyArray 585
使用重载的拷贝赋值操作符(=)。 586
使用重载的相等性操作符(==) 586
使用重载的下标操作符([]) 587
创建 MyArray ints4 并用 getArrayByValue 函数返回的 MyArray 来初始化 588
具名返回值优化(NRVO) 588
创建 MyArray ints5 并用 std::move 函数返回的右值来初始化 589
用移动赋值操作符将 MyArray ints5 赋给 ints4 590
将 MyArray ints5 转换成 bool 值来测试是否为空 591
用重载的++操作符前递增 ints4 的每个元素 591

用重载的++操作符后递增 ints4 的每个元素 592
用重载的+=操作符在 ints4 的每个元素上加一个值 592
销毁剩余的 MyArray 对象 592
11.6.3 MyArray 类定义 593
11.6.4 指定了 MyArray 大小的构造函数 595
11.6.5 C++11 向构造函数传递一个大括号初始化器 595
11.6.6 拷贝构造函数和拷贝赋值操作符 596
五法则 597
浅拷贝 597
深拷贝 598
实现拷贝构造函数 598
拷贝赋值操作符(=) 599
11.6.7 移动构造函数和移动赋值操作符 600
noexcept 说明符 601
MyArray 类的移动构造函数 601
其实什么都没移动 602
MyArray 类的移动赋值操作符(=) 603
移动操作应该 noexcept 603
11.6.8 析构函数 604
11.6.9 toString 和 size 函数 604
11.6.10 重载相等性(==)和不相等(!=)操作符 605
编译器自动生成!=操作符函数 605
将比较操作符定义为非成员函数 606
11.6.11 重载下标([])操作符 607

11.6.12 重载一元 bool 转换操作符 608
11.6.13 重载前递增操作符 608
11.6.14 重载后递增操作符 609
11.6.15 重载加赋值操作符(+=) 610
11.6.16 重载二元流提取(>>)和流插入(<<)操作符 611
实现流提取操作符 611
实现流插入操作符 612
为什么 operator>>和 operator<<必须是非成员函数 612
在成员或非成员函数之间做出选择 613
11.6.17 友元函数 swap 614
11.7 C++20 三路比较操作符(<=>) *614*
<=>使编译器支持所有比较操作符 616
显式使用<=> 617
11.8 类型之间的转换 *618*
转换操作符 618
隐式调用转换操作符和转换构造函数 618
11.9 explicit 构造函数和转换操作符 *619*
意外地将单参数构造函数用作转换构造函数 619
禁止单参数构造函数的隐式转换 620
C++11 explicit 转换操作符 622
11.10 重载函数调用操作符() *622*
11.11 小结 *622*
第 12 章 异常和对契约的展望 624
12.1 导读 *624*
异常和异常处理 625

异常处理、栈展开和重新抛出异常 626
何时使用异常和异常安全保证 626
构造函数和析构函数中的异常 626
处理动态内存分配失败 626
标准库异常层次结构和自定义异常类 627
异常没有广泛使用 627
展望“契约” 627
异常使你能将错误处理与程序逻辑分离 627
通过返回值来指示错误 628
12.2 异常处理控制流 628
12.2.1 定义一个异常类来表示可能发生的问题类型 629
12.2.2 演示异常处理 629
12.2.3 将代码封闭到 `try` 块中 631
12.2.4 为 `DivideByZeroException` 定义 `catch` 处理程序 631
通过引用来捕捉以防止切割 631
`catch` 参数名 632
`catch` 处理程序执行的任务 632
定义 `catch` 处理程序时常犯的错误 632
12.2.5 异常处理的终止模型 632
12.2.6 用户输入非零分母时的控制流 633
12.2.7 用户输入零分母时的控制流 633
12.3 异常安全保证和 `noexcept` 633
12.4 重新抛出异常 634
需要重新抛出异常的用例 635

演示重新抛出异常 635
12.5 栈展开和未捕捉的异常 637
未捕捉的异常 639
12.6 什么时候使用异常处理 639
什么时候不使用异常处理 640
12.6.1 assert 宏 641
12.6.2 快速失败 642
12.7 构造函数、析构函数和异常处理 642
12.7.1 从构造函数抛出异常 642
12.7.2 通过函数 try 块在构造函数中捕获异常 643
12.7.3 异常和析构函数：再论 noexcept(false) 646
12.8 处理 new 的失败 646
在持有指向动态分配内存的一个原始指针时不要抛出异常 647
12.8.1 new 在失败时抛出 bad_alloc 647
12.8.2 new 在失败时返回 nullptr 648
12.8.3 使用 set_new_handler 函数处理 new 的失败 649
12.9 标准库异常层次结构 650
基类 exception 651
exception 的派生类 651
runtime_error 的派生类 652
logic_error 的派生类 652
捕捉因继承而相关的异常类型 652
捕捉所有异常 653
12.10 C++的 finally 块替代方案：资源获取即初始化(RAII) 653
12.11 一些库同时支持异常和错误码 653

12.12 日志记录 *655*

12.13 展望“契约” *655*

违反前置和后置条件 656

不变式 656

契约式设计 656

C++标准库逐渐转向契约 657

契约属性 658

契约等级 658

指定前置条件、后置条件和断言 658

抢先体验 659

示例：除以零 659

契约延续模式 661

示例：查找函数要求一个有序 vector 661

自定义契约违反处理程序 664

12.14 小结 *664*

第 13 章 标准库容器和迭代器 **666**

13.1 导读 *666*

容器、迭代器和算法 666

容器的通用成员函数 666

迭代器 667

算法 667

C++20“范围” 667

自定义模板化数据结构 668

13.2 容器简介 *668*

序列容器 668
关联式容器 669
容器适配器 670
近似容器 670
13.2.1 序列和关联式容器中的通用嵌套类型 670
13.2.2 通用容器成员和非成员函数 671
容器特殊成员函数 672
非成员关系和相等性操作符 672
返回迭代器的成员函数 672
其他成员函数 673
13.2.3 对容器元素的要求 674
13.3 使用迭代器 *675*
13.3.1 使用 istream_iterator 进行输入，使用 ostream_iterator 进行输出 675
istream_iterator 676
ostream_iterator 676
13.3.2 迭代器的类别 676
13.3.3 容器对迭代器的支持 677
13.3.4 预定义迭代器类型名称 678
13.3.5 迭代器操作符 679
13.4 算法简介 *681*
13.5 序列容器 *681*
性能和对合适容器的选择 681
13.6 vector 序列容器 *682*
13.6.1 使用 vector 和迭代器 682
创建 vector 并显示其初始大小和容量 682

push_back 函数 683

修改 vector 后更新大小和容量 684

g++的 vector 扩容方式 684

调整 vector 大小时的其他考虑 684

用迭代器输出 vector 内容 685

用 const_reverse_iterator 反向显示 vector 的内容 685

C++11：shrink_to_fit 686

13.6.2 vector 元素处理函数 686

ostream_iterator 687

copy 算法 687

```
16      std::cout << "integers contains: ";
17      std::copy(integers.cbegin(), integers.cend(), output);
18
```

```
integers contains: 1 2 3 4 5
```

普通范围 687

vector 成员函数 front 和 back 687

访问 vector 的元素 688

vector 成员函数 insert 688

C++20“范围”的 copy 算法 689

vector 成员函数 erase 689

vector 的三参数成员函数 insert（范围插入） 690

vector 成员函数 clear 690

13.7 list 顺序容器 690

forward_list 容器 691

list 的成员函数 691

创建 list 对象 692
list 成员函数 sort 692
list 成员函数 splice 693
list 成员函数 merge 693
list 成员函数 pop_front 和 pop_back 694
list 成员函数 unique 694
list 成员函数 swap 695
list 成员函数 assign 和 remove 695
13.8 deque 序列容器 696
13.9 关联式容器 698
13.9.1 multiset 关联式容器 698
创建 multiset 699
multiset 成员函数 count 699
multiset 成员函数 insert 699
multiset 成员函数 find 700
multiset 成员函数 contains(C++20) 700
将另一个容器的元素插入 multiset 701
multiset 成员函数 lower_bound 和 upper_bound 701
pair 对象和 multiset 成员函数 equal_range 702
C++14：异质查找 702
13.9.2 set 关联式容器 703
创建 set 703
在 set 中插入新值 704
在 set 中插入重复值 704
13.9.3 multimap 关联式容器 704

创建包含键值对的 multimap 705
统计特定键的“键值对”数量 705
插入键值对 706
使用大括号初始化器而不是 make_pair 插入键值对 706
C++11：对键值对容器进行大括号初始化 707
13.9.4 map 关联式容器 707
13.10 容器适配器 *709*
13.10.1 stack 适配器 709
13.10.2 queue 适配器 711
13.10.3 priority_queue 适配器 712
13.11 bitset 近似容器 *713*
13.12 选读：Big O 简介 *715*
O(1)算法 716
O(n)算法 716
O(n2)算法 716
线性查找的效率O(n) 717
二叉查找效率 O(log n) 717
常用的 Big *O* 符号 718
13.13 选读：哈希表简介 *718*
13.14 小结 *720*
第 14 章 标准库算法和 C++20 范围/视图 **722**
14.1 导读 *722*
最低算法要求和 C++20“概念” 723
比较基于 C++20“范围”的算法与早期的“普通范围”算法 723

lambda、函数指针和函数对象 723
C++20“视图”和进行惰性求值的函数式编程 723
并行算法 724
展望 C++23 的“范围” 724
14.2 算法要求：C++20“概念” 724
迭代器要求 724
C++20“概念” 725
C++20 迭代器概念 726
C++20 范围概念 726
14.3 lambda 和算法 726
copy 算法以及“普通范围”和 C++20“范围”对迭代器的要求 727
for_each 算法 728
具有空 introducer 的 lambda 729
具有非空 introducer 的 lambda：捕获局部变量 729
lambda introducers [&]和[=] 730
lambda 返回类型 730
14.4 算法 730
14.4.1 fill、fill_n、generate 和 generate_n 730
fill 算法 731
fill_n 算法 732
generate 算法 732
generate_n 算法 733
通过 lambda 来使用 generate_n 算法 733
14.4.2 equal、 mismatch 和 lexicographical_compare 733
equal 算法 734

为 equal 算法使用二元谓词函数 735
mismatch 算法 735
关于 auto 和算法返回类型的注意事项 736
lexicographical_compare 算法 736
14.4.3 remove、remove_if、remove_copy 和 remove_copy_if 737
remove 算法 737
Erase–Remove 惯用法 738
C++20 std::erase 和 std::erase_if 738
remove_copy 算法 739
迭代器适配器：back_inserter、front_inserter 和 inserter 739
remove_if 算法 740
remove_copy_if 算法 740
14.4.4 replace、replace_if、replace_copy 和 replace_copy_if 741
replace 算法 741
replace_copy 算法 742
replace_if 算法 742
replace_copy_if 算法 743
14.4.5 打散、计数和最小/最大元素算法 743
shuffle 算法 744
count 算法 744
count_if 算法 745
min_element 算法 745
max_element 算法 746
minmax_element 算法 746

C++17 结构化绑定 747
transform 算法 747
14.4.6 查找和排序算法 748
find 算法 748
find_if 算法 749
sort 算法 750
binary_search 算法 750
all_of 算法 751
any_of 算法 751
none_of 算法 751
find_if_not 算法 752
14.4.7 swap、iter_swap 和 swap_ranges 752
swap 算法 753
iter_swap 算法 753
swap_ranges 算法 754
14.4.8 copy_backward、merge、unique、reverse、copy_if 和 copy_n 755
copy_backward 算法 756
move 和 move_backward 算法 756
merge 算法 756
unique 算法 757
reverse 算法 757
copy_if 算法 758
copy_n 算法 758
14.4.9 inplace_merge、unique_copy 和 reverse_copy 759
inplace_merge 算法 759

unique_copy 算法 760
reverse_copy 算法 760
14.4.10 集合操作 761
includes 算法 761
set_difference 算法 762
set_intersection 算法 762
set_symmetric_difference 算法 763
set_union 算法 763
14.4.11 lower_bound、upper_bound 和 equal_range 764
lower_bound 算法 764
upper_bound 算法 765
equal_range 算法 765
在有序序列中定位插入点 766
14.4.12 min、max 和 minmax 766
获取两个实参的 min 和 max 算法 766
获取两个实参的 C++11 minmax 算法 767
用于操作 C++20 范围的 minmax 算法 767
14.4.13 来自头文件<numeric>的 gcd、lcm、iota、reduce 和 partial_sum 算法 768
gcd 算法 768
lcm 算法 769
iota 算法 769
reduce 算法 770
对比 reduce 和 accumulate 770
partial_sum 算法 771

14.4.14 堆排序和优先队列 772
堆数据结构 772
和堆相关的算法 772
初始化并显示 heapArray 773
make_heap 算法 773
sort_heap 算法 774
用 push_heap 和 pop_heap 来维护堆 774
演示堆数据结构 775
14.5 函数对象(仿函数) *777*
函数对象的优势 777
标准库的预定义函数对象 778
使用 accumulate 算法 779
sumSquares 函数 780
SumSquaresClass 类 781
调用 accumulate 算法 781
14.6 投射 *783*
定义并显示一个 array<Employee> 783
使用投射按升序对 array<Employee>排序 784
投射的简化形式 785
投射可以是指向 public 数据成员的指针 785
使用投射按降序对 array<Employee>排序 785
14.7 C++20 视图和函数式编程 *786*
14.7.1 范围适配器 786
14.7.2 使用范围适配器和视图 788
用 std::views::iota 创建无限范围 788

take 范围适配器 789
take_while 范围适配器 790
reverse 范围适配器 790
transform 范围适配器 790
drop 范围适配器 791
drop_while 范围适配器 791
创建并显示罗马数字及其十进制值的 map 792
keys 和 values 范围适配器 792
elements 范围适配器 793
14.8 并行算法简介 *793*
14.9 标准库算法小结 *795*
会修改序列的算法 796
不修改序列的算法 796
排序及相关算法 797
数值算法 797
特殊内存操作 798
14.10 C++23“范围”前瞻 *798*
“A Plan for C++23 Ranges” 799
14.11 小结 *800*
第 15 章 模板、C++20“概念”和元编程 801
15.1 导读 *801*
自定义模板 803
C++20 的模板特性 803
类型 traits 803

构建自定义容器、迭代器和算法 803
可变参数模板和折叠表达式 804
模板元编程 804
15.2 自定义类模板和编译时多态性 804
创建类模板 Stack<T> 805
template 头 806
类模板 Stack<T>的数据表示 807
类模板 Stack<T>的成员函数 807
测试类模板 Stack<T> 808
在类模板外部定义类模板成员函数 810
15.3 C++20 对函数模板的增强 811
15.3.1 C++20 缩写函数模板 811
有时需要传统的函数模板语法 812
为 printContainer 使用不兼容的类型 813
15.3.2 C++20 模板化 lambda 813
15.4 C++20“概念”初探 813
“概念”的动机与目标 814
15.4.1 无约束的函数模板 multiply 815
在 multiply 中使用不兼容的类型 816
15.4.2 带有 C++20“概念”的 requires 子句的有约束的函数模板 818
requires 子句 818
析取和合取 820
用满足约束的实参调用 multiply 820
用不满足约束的实参调用 multiply 820
15.4.3 C++20 预定义概念 821

15.5 类型 traits *823*
在 C++20 之前使用类型 traits 823
C++20 预定义概念经常使用类型 traits 823
演示类型 traits 823
15.6 C++20 概念：深入了解 *828*
15.6.1 创建自定义概念 828
15.6.2 使用概念 829
放在 `template` 头后面的 `requires` 子句 829
放在函数模板签名后面的 `requires` 子句 829
概念作为 `template` 头中的一个类型 829
15.6.3 在缩写函数模板中使用概念 830
15.6.4 基于概念的重载 831
重载函数的匹配过程 832
重载函数模板 832
使用概念来重载函数模板 832
15.6.5 `requires` 表达式 834
简单要求 835
类型要求 836
复合要求 836
嵌套要求 837
`requires requires`——临时约束 837
15.6.6 C++20 仅供参详的概念 838
15.6.7 C++20“概念”之前的技术：SFINAE 和 Tag Dispatch 839
SFINAE——替换失败不是错误 839

Tag Dispatch 840

15.7 用 static_assert 测试 C++20 概念 840

15.8 创建自定义算法 843

自定义 NumericInputRange 概念 845

自定义 average 算法 845

对标准库容器使用 average 算法 845

15.9 创建自定义容器和迭代器 846

单一头文件 846

MyArray 支持双向迭代器 846

为什么要实现双向迭代器而不是随机访问迭代器？ 847

基本迭代器要求 848

15.9.1 类模板 ConstIterator 849

标准迭代器嵌套类型名称 851

构造函数 852

++操作符 852

重载的*和->操作符 852

双向迭代器比较 852

--操作符 852

15.9.2 类模板 Iterator 852

标准迭代器嵌套类型名称 854

构造函数 854

++和--操作符 854

重载的*和->操作符 854

15.9.3 类模板 MyArray 854

MyArray 的 template 头 856

标准容器嵌套类型名称 856
返回迭代器的 MyArray 成员函数 857
MyArray 的重载操作符 857
15.9.4 针对大括号初始化的 MyArray 推导指引 857
ConstIterator 和 Iterator——MyArray 推导指引 858
15.9.5 将 MyArray 及其自定义迭代器用于 std::ranges 算法 859
创建 MyArray 并用 std::ranges::copy 显示 861
用基于范围的 for 语句显示 MyArray 861
用 std::ranges::copy 拷贝 MyArray 861
用 std::ranges::minmax_element 查找 MyArray 中的最小和最大元素 862
用 std::ranges::reverse 来反转 MyArray 862
尝试将 std::ranges::sort 用于 MyArray 862
15.10 模板类型参数的默认实参 863
15.11 变量模板 864
15.12 可变参数模板和折叠表达式 864
15.12.1 tuple 可变参数类模板 865
打包 tuple 867
用 std::make_tuple 创建 tuple 867
通过结构化绑定来解包 tuple 867
使用 get<*index*>按索引获取一个 tuple 成员 867
C++14 使用 get<*type*>按类型获取一个 tuple 成员 868
tuple 的其他特性 868
15.12.2 可变参数函数模板和 C++17 折叠表达式简介 869
编译时递归 871

调用有一个实参的 sum 871
调用有两个实参的 sum 871
调用有三个实参的 sum 872
用三个 string 对象调用 sum 872
C++17 折叠表达式 872
15.12.3 折叠表达式的类型 873
15.12.4 一元折叠表达式如何应用它们的操作符 874
一元左折和右折加法 875
一元左折和右折减法 876
一元折叠表达式中的参数包必须非空 876
15.12.5 二元折叠表达式如何应用它们的操作符 877
15.12.6 使用逗号操作符重复执行一个操作 879
15.12.7 将参数包中的元素约束为同一类型 879
可变参数模板的自定义概念 881
调用 foldingSum 882
15.13 模板元编程 882
15.13.1 C++模板是图灵完备的 883
15.13.2 在编译时计算值 884
元函数 886
Factorial 元函数 887
“调用”元函数 887
Factorial 元函数针对基本情况的特化——Factorial<0> 887
编译器如何为实参 0 求值元函数 Factorial 888
编译器如何为实参 3 求值元函数 Factorial 888
函数式编程 889

使用 constexpr 函数执行编译时计算 889
15.13.3 用模板元编程和 constexpr if 进行条件编译 889
15.13.4 类型元函数 892
为类型添加 const 894
从指针类型移除* 894
为非指针类型实例化 my_remove_ptr 895
为指针类型实例化 my_remove_ptr 895
向类型添加左值和右值引用 896
从引用类型中移除左值和右值引用 896
15.14 小结 *896*
第 16 章 C++20 模块：大规模开发 898
16.1 导读 *898*
模块简史 899
模块的编译器支持 899
用于 g++ 11.2 和 clang++ 13 的 Docker 容器 900
C++20 之前的 C++编译和链接 900
模块的优点和目标 900
模块和模块工具的未来 900
参考资料和模块词汇表 900
16.2 C++20 之前的编译和链接 *901*
当前的头文件/源代码文件模型存在的问题 901
16.3 模块的优点与目标 *902*
模块的缺点 903
16.4 示例：过渡到模块——头单元 *903*

头单元和头文件的区别 904
将所有头文件作为头单元导入(如果可能的话) 905
在 Microsoft Visual Studio 中用头单元进行编译 905
在 g++中用头单元进行编译 906
在 clang++用头单元进行编译 906
16.5 模块可以减少翻译单元的大小和编译时间 907
参考资料 907
编译器分析工具 908
16.6 示例：创建并使用模块 908
16.6.1 模块接口单元的 `module` 声明 909
模块声明和模块命名 910
模块接口文件扩展名 910
g++和 clang++的模块接口文件扩展名 911
其他文件扩展名 911
16.6.2 导出声明 911
在模块中定义模板、`constexpr` 函数和 `inline` 函数 912
16.6.3 导出一组声明 912
16.6.4 导出命名空间 912
定义和导出命名空间 912
访问命名空间的成员 912
16.6.5 导出命名空间的成员 913
16.6.6 导入模块以使用其导出的声明 913
在 Visual C++中编译这个例子 914
在 g++中编译这个例子 914
在 clang++中编译这个例子 914

16.6.7 示例：试图访问未导出的模块内容 915
g++的错误消息 917
clang++的错误消息 917
16.7 全局模块片断 *917*
16.8 将接口与实现分开 *918*
16.8.1 示例：模块实现单元 918
主模块接口单元 918
模块实现单元 919
使用模块 920
在 Visual C++中编译这个例子 920
在 g++中编译这个例子 921
在 clang++中编译这个例子 921
16.8.2 示例：模块化一个类 921
`deitel.time` 主模块接口单元 922
`deitel.time` 模块实现单元 922
在 `deitel.time` 模块中使用 `Time` 类 923
在 Visual C++中编译这个例子 924
在 g++中编译这个例子 924
在 clang++中编译这个例子 925
16.8.3 `:private` 模块片断 925
16.9 分区 *926*
16.9.1 示例：模块接口分区单元 927
`deitel.math:powers` 模块接口分区单元 927
`deitel.math:roots` 模块接口分区单元 927

deitel.math 主模块接口单元 928
使用 deitel.math 模块 929
在 Visual C++中编译这个例子 930
在 g++中编译这个例子 930
16.9.2 模块实现分区单元 930
16.9.3 示例：“子模块”和分区 931
deitel.math.powers 主模块接口单元 931
在 Visual C++中编译这个例子 932
在 g++中编译这个例子 932
在 clang++中编译这个例子 932
deitel.math.roots 主模块接口单元 933
在 Visual C++中编译这个例子 933
在 g++中编译这个例子 934
在 clang++中编译这个例子 934
deitel.math 主模块接口单元 934
在 Visual C++中编译这个例子 935
在 g++中编译这个例子 935
在 clang++中编译这个例子 936
16.10 其他模块示例 *936*
16.10.1 示例：将 C++标准库作为模块导入 936
在 Visual C++中编译程序 937
在 clang++中修改和编译程序 938
16.10.2 示例：不允许循环依赖 938
16.10.3 示例：导入不具传递性 939
16.10.4 示例：可见性和可达性 940

16.11 将代码迁移到模块 *942*

16.12 模块和模块工具的未来 *943*

参考资料 944

16.13 小结 *945*

附录：模块相关视频 946

附录：模块相关文章 947

文档 949

附录：模块词汇表 950

第 17 章 并行算法和并发性：高级观点 952

17.1 导读 *952*

多任务的顺序、并发和并行操作 952

C++并发性 953

并发编程用例：视频流式传输(串流) 953

线程安全 953

并发编程很复杂 954

C++11 和 C++14：提供低级并发特性 954

C++17 和 C++20：提供方便的高级并发特性 955

17.2 标准库并行算法(C++17) *955*

17.2.1 示例：分析顺序排序和并行排序算法 956

设置随机数生成 958

创建数组 958

用 `std::chrono` 计时 958

顺序排序 959

使用执行策略 `std::execution::par` 进行并行排序 959

警告：并行化并非一定更快 959
17.2.2 什么时候使用并行算法 960
17.2.3 执行策略 960
17.2.4 示例：分析并行化和矢量化运算 961
17.2.5 并行算法的其他注意事项 964
并行算法名称 964
限制条件 965
17.3 多线程编程 965
17.3.1 线程状态和线程生命周期 965
新建和就绪状态 966
运行状态 966
等待状态 967
计时等待状态 967
阻塞状态 967
终止状态 968
线程调度 968
17.3.2 死锁和无限期推迟 968
死锁 968
死锁的 4 个必要条件 969
无限期推迟 969
预防死锁 969
等待有危险 970
17.4 用 `std::jthread` 启动线程 970
`jthread` 中的异常 971
17.4.1 定义在线程中执行的任务 971

17.4.2 在一个 jthread 中执行任务 973
等待之前调度的任务终止 975
主线程 975
示例输出 976
17.4.3 jthread 对 thread 的修正 976
17.5 生产者-消费者关系：首次尝试 977
生产者-消费者关系的另一个例子 977
同步 978
状态依赖 978
缺少同步而导致的逻辑错误 978
UnsynchronizedBuffer 979
这个缓冲区不受同步的保护 979
main 应用程序 980
生产者线程 983
消费者线程 983
随机数生成 983
调用 std::this_thread::sleep_for 只是出于演示目的 984
分析输出 984
17.6 生产者-消费者：同步对共享可变数据的访问 985
线程同步、互斥和临界区 985
一组操作当作一个操作来执行 986
不可变的数据不需要同步 986
17.6.1 SynchronizedBuffer 类：互斥体、锁和条件变量 986
SynchronizedBuffer 的 m_buffer 和 m_occupied 数据成员 989

SynchronizedBuffer 的 std::condition_variable 数据成员 989
SynchronizedBuffer 的 std::mutex 数据成员 989
SynchronizedBuffer 的成员函数 990
成员函数 put 和生产者线程 990
获得锁并检查是否轮到生产者 991
在条件变量上等待 991
虚假唤醒 992
更新 SynchronizedBuffer 的状态 992
释放锁 992
通知消费者线程继续 993
成员函数 get 和消费者线程 993
在条件变量上等待 993
更新 SynchronizedBuffer 的状态并通知生产者继续 993
等待和通知必须成对 994
17.6.2 测试 SynchronizedBuffer 994
分析输出 997
关于同步示例中输出语句的说明 998
17.7 生产者-消费者：用循环缓冲区最小化等待时间 998
循环缓冲区 998
实现循环缓冲区 999
数据成员 1003
CircularBuffer 的 put 函数 1003
CircularBuffer 的 get 函数 1003
CircularBuffer 的 displayState 函数 1004
测试 CircularBuffer 类 1004

分析输出 1007
17.8 读者和写者 *1008*
17.9 协作式取消 jthread *1009*
可选的 stop_callback 1012
分析输出 1012
17.10 用 std::async 启动任务 *1012*
安全性、加密和大整数质因数分解 1013
本例概述 1013
类型别名 1014
对整数进行质因数分解的 getFactors 函数 1015
用于确认质因数分解 proveFactors 函数 1017
显示质因数分解的 displayResults 函数 1018
创建并执行任务：函数模板 std::async 1019
std::future、std::promise 和线程间通信 1020
std::packaged_task 和 std::async 1021
17.11 线程安全的一次性初始化 *1021*
static 局部变量 1022
once_flag 和 call_once 1022
17.12 原子类型简介 *1023*
在两个并发线程中递增 int、std::atomic<int>和 std::atomic_ref<int> 1023
定义计数器 1025
递增计数器 1025
分析示例输出 1026
关于原子类型的更多话题 1026

C++20 原子智能指针和原子指针 1027
C++20 std::atomic_ref 类模板 1027
17.13 用 C++20 闭锁和栅栏来协同线程 1027
17.13.1 C++20 std::latch 1028
在门户永久性开放时试图等待 1028
演示闭锁 1028
17.13.2 C++20 std::barrier 1031
17.14 C++20 信号量 1035
使用 C++20 std::binary_semaphore 的生产者-消费者关系 1037
SynchronizedBuffer 的 put 成员函数 1039
SynchronizedBuffer 的 get 成员函数 1040
17.15 C++23：C++并发性未来展望 1040
17.15.1 并行范围算法 1040
17.15.2 并发容器 1040
17.15.3 其他和并发性相关的提案 1041
17.16 小结 1042
第 18 章 C++20 协程 1044
18.1 导读 1044
协程用例 1045
18.2 协程支持库 1046
18.3 安装 concurrencpp 和 generator 库 1047
concurrencpp 库 1047
generator 库 1048
18.4 用 co_yield 和 generator 库创建生成器协程 1048
将线程 ID 转换为 std::string 的 id 函数 1050

fibonacciGenerator 协程 1050
main 函数 1050
生成器协程的控制流示意图 1051
协程是无栈的 1052
18.5 用 concurrencpp 启动任务 1052
id 函数 1055
printTask 函数 1055
main 函数 1055
concurrencpp 的任务和结果 1056
concurrencpp::thread_pool_executor 1056
等待任务完成执行 1056
concurrencpp 的 executors 小结 1057
任务并非一定要用单独的线程运行 1057
18.6 用 co_await 和 co_return 创建协程 1058
本例概述 1058
#include 指令和 id 函数 1058
用两个任务来执行的 sortCoroutine 1059
在 C++20 的 co_await 表达式中使用 when_all 1061
决定是否暂停协程执行 1062
恢复协程执行 1062
concurrencpp 的 when_any 实用函数 1062
main 程序 1062
示例输出 1064
co_await 和 co_return 的协程的控制流 1065

18.7 低级协程概念 1066

协程的限制 1067

promise 对象 1068

协程状态 1068

协程句柄 1069

可等待对象 1069

18.8 C++23 的协程改进计划 1069

18.9 小结 1070

附录 A 操作符优先级和分组 1071

附录 B 字符集 1074

网上章节和附录

第 19 章 流 I/O 和深入理解 C++20 文本格式化

第 20 章 其他主题和对 C++23 的展望

附录 C 数字系统

附录 D 预处理器

附录 E 位操作

前言

欢迎阅读《C++20 程序设计》。本书为软件开发人员介绍前沿的计算技术，符合 ISO C++标准委员会于 2020 年 9 月批准[①]的 C++20 标准（共 1834 页）。[②]

C++编程语言适合构建高性能的关键业务和关键任务计算系统，包括操作系统、实时系统、嵌入式系统、游戏系统、银行系统、空中管制系统、通信系统等。本书是 C++20 的一个从入门到中级的教程，涵盖了 C++语言本身（世界上最流行的编程语言之一[③]）及其相关的标准库。我们对 C++20 进行了友好的、现代的、代码密集的、面向案例的介绍。在这篇前言中，我们探讨了这本书的“灵魂”。

P.1 现代 C++

本书专注于“现代 C++”，其中包括四个最新的 C++标准：C++20、C++17、C++14 和 C++11，并对 C++23 以及更以后的关键特性进行了展望。本书致力于探讨用 C++进行编码的一些新的和改进的方法。我们采用的是最佳实践，强调当今专业软件开发的“现代 C++”惯例，并将重点放在性能、安全性和软件工程问题上面。

保持话题性

“胆敢教授的人，绝不能停止进修。”（约翰·科顿·达纳）[④]

为了把握住“现代 C++”的脉搏，改变开发人员写 C++程序的方式，我们阅读、浏览或观看了大约 6000 篇最新的文章、研究论文、白皮书、文档、博客帖子、论坛帖子和视频。

[①] Herb Sutter, “C++20 Approved, C++23 Meetings and Schedule Update”，2020 年 9 月 6 日。详情请访问 https://herbsutter.com/2020/09/06/c20-approved-c23-meetings-and-schedule-update/。

[②] C++标准的最终草案在此：https://timsong-cpp.github.io/cppwp/n4861/，该版本免费。要购买最终发行版（ISO/IEC 14882:2020），请访问 https://www.iso.org/standard/79358.html。

[③] 出自“TIOBE 编程社区指数”，网址是 https://www.tiobe.com/tiobe-index。

[④] 出自 https://www.bartleby.com/73/1799.html：新泽西州纽瓦克市的图书馆馆长达纳被要求提供适合铭刻在新泽西州尤尼恩市纽瓦克州立学院（现基恩大学）新大楼上的拉丁语引文。由于找不到合适的引文，达纳创作了后来成为校训的这段话。——《纽约时报书评》，1967 年 3 月 5 日，第 55 页。

C++版本

20 作为开发人员，你可能被要求在 C++遗留代码或特定 C++版本的项目上工作。因此，本书使用了有特色的页边图标，例如这里显示的“20”。这样一来，每次提到某个“现代 C++”语言特性时，你就知道它首次出现于哪个 C++版本。这些图标帮助你体会 C++的发展历程：通常都是从低级别的细节，逐渐发展到更容易使用的高级别的表达形式。这个趋势有利于缩短开发时间，提高性能、安全性和系统可维护性。

P.2 目标读者

本书面向的读者包括：

- 希望在完全覆盖、专业风格的教程中学习最新 C++20 特性的 C++软件开发人员。
- 需要做一个 C++项目，并想学习最新语言特性的非 C++软件开发人员。
- 在大学里学过 C++，或者在一段时间内因为职业需要使用过 C++，并希望在 C++20 的背景下更新自己的 C++知识的软件开发人员。
- 想要开发 C++20 课程的职业 C++培训师。

P.3 “实时编码”方法和代码下载

本书的核心是 Deitel 标志性的实时编码（Live-Code）方法。我们不是展示代码片段，而是在数百个完整的、可运行的、具有实时输出的真实世界的 C++程序的背景下使用 C++。

请阅读“前言”之后的“准备工作”部分，了解如何设置你的 Windows、macOS 或 Linux 计算机来运行由大约 15 000 行代码组成的 200 多个代码示例。所有源代码都可以在以下网站免费下载：

- https://github.com/pdeitel/CPlusPlus20ForProgrammers
- https://www.deitel.com/books/c-plus-plus-20-for-programmers
- https://informit.com/title/9780136905691 (参见 P.8 节)

为了提供方便，我们以 C++源代码（.cpp 和.h）文件的形式提供本书的例子，供集成开发环境和命令行编译器使用。参见第 1 章的“试运行”一节（第 1.2 节），了解如何用我们推荐的三种编译器来编译和运行代码示例。在看书的同时执行每个程序，你的学习经历会“活”起来。遇到任何问题，可以发送邮件至 deitel@deitel.com 联系我们。

P.4 三种工业强度的编译器

我们在以下最新版本的平台上测试了所有代码示例：

- Windows®平台：Microsoft® Visual Studio®社区版的 Visual C++®。
- MacOS®平台：Apple® Xcode®以及一个 Docker®容器中的 Clang C++（clang++）。
- Linux®平台：GNU 编译器集合（GCC）Docker®容器中的 GNU® C++（g++）。

撰写本书的时候，大多数 C++20 特性已被所有三个编译器完全实现，有些被三个编译器中的一个子集实现，有些还没有被任何编译器实现。我们会酌情指出这些差异，并在编译器厂商实现剩余的 C++20 特性时更新我们的数字内容。我们还会在本书的 GitHub 仓库中发布代码更新：

https://github.com/pdeitel/CPlusPlus20ForProgrammers

在本书配套网站上，会同时提供对代码和正文的更新：

https://www.deitel.com/books/c-plus-plus-20-for-programmers
https://informit.com/title/9780136905691

本书中文版网站会同步提供上述内容，并提供后期勘误：

https://bookzhou.com

P.5 编程技巧和 C++20 关键特性

本书使用多种图标来提醒你注意**软件开发技巧**以及 C++20 的模块和概念特征：

- **软件工程观察**强调为了正确开发软件而需要注意的架构和设计问题，特别是对于较大型的系统而言。
- **安全性最佳实践**帮助你加强程序以抵御攻击。
- **性能提示**给出了使你的程序运行得更快或尽量减少占用内存的窍门。
- **常见编程错误**有助于减少你犯同样错误的可能性。
- C++**核心准则**建议（P.9 节会具体说明）。
- C++20 新的**模块**特性。
- C++20 新的**概念**特性。

P.6 “对象自然”法

第 9 章会开始介绍如何开发自定义的 C++20 类，并在余下章节继续讨论面向对象编程。

何谓对象自然？

在第 9 章之前的各章，你使用的是一些现有的类，它们能完成很多重要的工作。会快速地

创建这些类的对象，并通过最少数量的简单 C++语句让它们“大显身手”。我们把这称为“对象自然法”（Objects-Natural Approach）。

鉴于 C++社区已经创建了大量免费、开源的类库，在第 9 章学习如何创建自己的 C++类之前，你就已经能够执行强大的任务了。这是使用面向对象语言（特别是 C++这样成熟的面向对象语言）最吸引人的方面之一。

免费的类

我们鼓励使用 C++生态系统现有的大量有价值的免费类，它们通常来自：

- C++标准库。
- 平台特有的库，如 Microsoft Windows、Apple macOS 或者各种 Linux 版本所提供的库。
- 免费的第三方 C++库，通常由开源社区创建。
- 其他开发者，比如你所在组织的开发者。

我们鼓励你查看大量免费的、开源的 C++代码实例（可在 GitHub 等网站上找到）以寻找灵感。

Boost 项目

Boost 提供了 168 个开源的 C++库[①]。它也是最终被纳入 C++标准库的新特性的“温床”。一些已被添加到“现代 C++”的特性包括多线程、随机数生成、智能指针、元组、正则表达式、文件系统和 `string_view`[②]。以下 StackOverflow 答案列出了从 Boost 库演变而来的“现代 C++”库和语言特性[③]：

https://stackoverflow.com/a/8852421

对象自然案例研究

第 1 章主要是解释对象技术的基本概念和术语。然后，到第 9 章之前，本书的重点都是创建并使用现有类的对象。在第 9 章及之后各章，才会开始创建自定义类。我们的对象自然案例研究包括：

---

① “Boost 1.78.0 Library Documentation”，https://www.boost.org/doc/libs/1_78_0/。

② https://en.wikipedia.org/wiki/Boost_(C++_libraries)。

③ “Which Boost Features Overlap with C++11?”，Kennytm 的回答，https://stackoverflow.com/a/8852421。

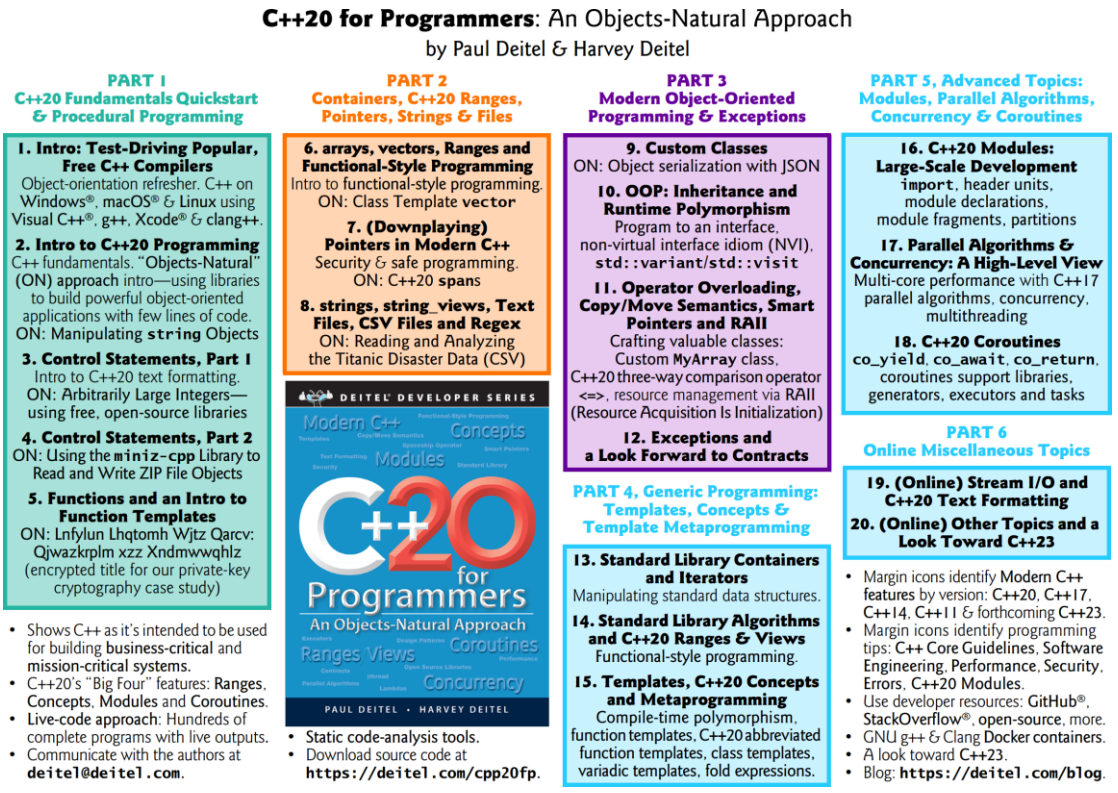
- 2.7 节——创建并使用标准库 `string` 类的对象
- 3.12 节——任意大小的整数
- 4.13 节——使用 `miniz-cpp` 库读写 ZIP 文件
- 5.20 节——Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz（这是我们私钥加密案例研究的加密标题）
- 6.15 节——C++标准库模板类 `vector`
- 7.10 节——C++20 `span`：连续容器元素的视图
- 8.19 节——读取/分析包含泰坦尼克号灾难数据的 CSV 文件
- 8.20 节——正则表达式简介
- 9.22 节——用 JSON 序列化对象

对象自然法的一个完美例子是直接使用现有类的对象，如 `array` 和 `vector`（第 6 章），而不需要知道如何编写自定义类，特别是不需要知道如何编写上述这些类。本书会广泛地使用现有的 C++标准库的功能。

P.7 本书导读

本书英文版封面内页提供了一幅全彩目录图，它揭示了本书的模块化结构。阅读本节时，你也可以参考该图。该图和本节将帮助你快速“浏览”本书的内容。下面展示了这幅英文版的目录图[1]：

[1] 原图下载地址：https://deitel.com/cpp20fpTOCdiagram。



本节描述了本书的许多关键特性。早期的章节为 C++20 的基础知识打下了坚实的基础。中、高端章节和案例研究使你轻松进入基于 C++20 的现代软件开发。贯穿全书，我们将讨论 C++20 的几种编程模型：

- 程序化编程
- 函数式编程
- 面向对象编程。
- 泛型编程
- 模板元编程

第 1 部分：编程基础知识快速入门

第 1 章“介绍免费、流行的 C++编译器并试运行”：本书是为专业软件开发人员编写的，所以第 1 章会这样安排：

- 提供一个简单的介绍。
- 讨论了摩尔定律、多核处理器以及为什么标准化的并发编程在现代 C++中很重要。
- 提供了关于面向对象一个简单回顾，介绍了贯穿全书的术语。

然后，直接进入试运行环节，演示如何用我们首选的三个免费编译器编译和执行 C++代码。

- Windows 平台：作为 Visual Studio 一部分提供的 Microsoft Visual C++
- macOS 平台：Apple Xcode
- Linux 平台：GNU g++

我们使用每一种环境对书中的代码实例进行了测试，指出了少数编译器不支持某一特定语言特性的情况。请选择你自己喜欢的程序开发环境。本书在其他 C++20 编译器中也能很好地工作。

我们还演示了 GNU 编译器集合(GCC) Docker 容器中的 GNU g++以及一个 Docker 容器中的 Clang C++。这使你可以在 Windows、macOS 或 Linux 上运行最新的 GNU g++和 clang++命令行编译器。关于 Docker 这种重要的开发工具的更多信息，请参见本“前言”的 P.13 节。要了解安装说明，请阅读本“前言”之后的“准备工作”部分。

Windows 用户也可以直接安装 Linux，我们会指导你阅读 Microsoft 提供的指示，通过 Windows Subsystem for Linux（WSL）在 Windows 上安装 Linux。这是在 Windows 上使用 g++和 clang++编译器的另一种方式。

第 2 章“C++编程入门”介绍了 C++的基础知识，说明了关键的语言特性，包括输入、输出、基本数据类型、算术操作符及其优先级以及决策。2.7 节的对象自然案例研究演示了如何创建并使用标准库提供的 `string` 类的对象。在这个时候，你不必知道如何开发自定义类，特别是不必知道如何实现如此大型且复杂的类。

第 3 章“控制语句（第 1 部分）”的重点是控制语句。你将使用 `if` 和 `if...else` 选择语句，用计数器控制和用哨兵控制的 `while` 循环语句，以及递增、递减和赋值操作符。3.12 节的对象自然案例研究演示了如何使用第三方库来创建任意大小的整数。

第 4 章“控制语句（第 2 部分）”介绍了 C++的其他控制语句，包括 `for`，`do...while`，`switch`，`break` 和 `continue`，还介绍了逻辑操作符。4.13 节的对象自然案例研究演示了如何使用 `miniz-cpp` 库以编程方式写入和读取 ZIP 文件。

第 5 章“函数和函数模板入门”，介绍了自定义函数。我们用随机数生成来演示模拟技术。C++从 C 语言继承的随机数生成函数 `rand` 不具备良好的统计特征，是可以预测的。我们讲解了 C++11 引入的更安全的随机数库，它能生成非确定性的随机数——一组无法预测的随机数。这种随机数生成器应用于那些不容许预测的模拟和安全场景。我们还讨论了函数之间的信息传递以及递归。5.20 节的对象自然案例研究演示了私钥加密。

第 2 部分：数组、指针和字符串

第 6 章“数组、向量、范围和函数式编程”初步讨论了 C++标准库的容器、迭代器和算法。我们介绍了 C++标准库中用于表示值的列表和表格的 `array` 容器。将定义和初始化数组，并访问其元素。讨论了如何将数组传给函数，对数组进行排序和查找，以及操作多维数组。

我们通过介绍 lambda 表达式（匿名函数）和 C++20 的“范围”——C++20 的“四大”特性之一——来开始接触函数式编程。6.15 节的对象自然案例研究演示了 C++标准库的 vector 模板。本章实质是 array 和 vector 的一个大型的“对象自然”案例研究，其中的代码很好地演示了“现代 C++”编码惯例（coding idioms）。

第 7 章“现代 C++对指针的淡化”全面介绍了指针，并探讨了内置指针、基于指针的数组和基于指针的字符串（也称为 C 字符串）之间的密切关系，所有这些都是 C++从 C 语言继承的。指针很强大，但使用起来很有挑战性。为此，“现代 C++”设计了一些特性来消除对大多数指针的需求，使代码变得更健壮、更安全。这些特性包括 array 和 vector、C++20 span 以及 C++17 string_view。我们还是会讨论内置数组，它们在 C++中依然有用，也是你看懂遗留代码所需要的。不过，如果是开发新项目，那么应该首选“现代 C++”提供的能力。7.10 节的对象自然案例研究展示的就是这样的一种能力：C++20 span。它使你能查看和操作连续容器（例如基于指针的数组以及标准库的 array 和 vector）的元素，同时无需直接使用指针。本章再次强调了“现代 C++”的编码惯例。

第 8 章“string、string_view、文本文件、CSV 文件和正则表达式”介绍了标准库 string 类的许多特性；展示了如何读写纯文本文件和以逗号分隔的 CSV 文件（常用于表示数据集）；并讨论了如何用标准库的正则表达式（regex）功能进行字符串模式匹配。C++支持两种类型的字符串：string 对象和 C 风格的、基于指针的字符串。我们使用 string 类的对象使程序变得更健壮，同时避免 C 字符串的许多安全问题。如果是开发新项目，那么应该首选 string 对象。我们还介绍了 C++17 的 string_view，它是将任何类型的字符串传递给函数的一种轻量级的、灵活的机制。本章有两个对象自然案例研究：

- 8.19 节通过读取并分析一个包含泰坦尼克号灾难数据集的 CSV 文件来介绍数据分析，这是向初学者介绍数据分析的一个流行的数据集。
- 8.20 节介绍了正则表达式模式匹配和文本替换。

第 3 部分：面向对象编程

第 9 章“自定义类”正式讨论面向对象编程，我们会编写一些有价值的自定义类。C++是一种可扩展的语言——你编写的每个类都会成为一个新类型，可基于它创建对象。9.22 节的对象自然案例研究使用第三方库 cereal 将对象转换为 JSON 格式（序列化），并从其 JSON 表示中重新创建这些对象（反序列化）。

第 10 章“OOP：继承和运行时多态性”主要讨论继承层次结构中类之间的关系以及这些关系所带来的强大的运行时多态处理能力。本章的一个重要方面是了解多态性是如何工作的。我们用详细的插图和文字解释了 C++于幕后如何实现多态性、虚函数和动态绑定。你会看到，语言使用了一种优雅的、基于指针的数据结构。我们还介绍了用于实现运行时多态性的其他机制，其中包括非虚接口（NVI）惯用法和 std::variant/std::visit。本章还讨论了一个重要的编程原则：藉由接口来编程，而不要藉由实现（programming to an

interface, not an implementation）。

错误

第 11 章“操作符重载、拷贝/移动语义和智能指针”介绍如何使 C++现有的操作符与自定义类的对象一起工作，并讨论了智能指针和动态内存管理。智能指针提供了内置指针所不具备的额外功能，能帮助你避免动态内存管理错误。`unique_ptr` 会在本章讨论，而 `shared_ptr` 和 `weak_ptr` 会在本书在线内容的第 20 章讨论。本章的一个关键点是教你打造有价值的类。首先展示一个 `string` 类的例子，它优雅地运用了操作符重载技术。然后，教你如何创建自定义类来实现该功能。接着是本书最重要的案例研究之一，你将利用重载操作符和其他功能创建自定义的 `MyArray` 类，从而解决 C++原生的、基于指针的数组所存在的各种问题[①]。我们介绍并实现了 5 种可在所有类中定义的特殊成员函数：拷贝构造函数、拷贝赋值操作符、移动构造函数、移动赋值操作符和析构函数。我们讨论了拷贝语义和移动语义，它们使编译器能够将资源从一个对象移动到另一个对象，以避免昂贵的、不必要的拷贝。我们介绍了 C++20 的三路比较操作符`<=>`（也称为“太空飞船操作符”），并演示了如何实现自定义转换操作符。第 15 章会着手将 `MyArray` 转换为一个可用于存储指定类型的元素的类模板，届时将真正打造出一个有价值的类。

性能

20

错误

第 12 章“异常和对契约的展望”继续讨论第 6 章开始引入的异常处理。我们讨论了何时使用异常、异常安全保证、构造函数/析构函数中的异常、如何处理动态内存分配失败以及为什么有的项目不使用异常处理。本章最后还介绍了契约，这是未来 C++可能会纳入的一个特性。我们通过 godbolt.org 网站上的一个实验性契约实现来进行说明。契约的一个目标是将大多数函数都变成 `noexcept`，即指定函数不抛出异常。这可能允许编译器执行额外的优化，同时消除异常处理的开销和复杂性。

性能

第 4 部分：标准库容器、迭代器和算法

第 13 章“标准库容器和迭代器”开始更广泛和更深入地讨论三个关键的 C++标准库组件：

- 容器（模板化数据结构）。
- 迭代器（用于访问容器元素）
- 算法（用迭代器操作容器）

本章将讨论容器、容器适配器和近似容器。会讲到 C++标准库提供的一些常用数据结构，这样就不必自行创建——我们的绝大多数数据结构需求都可通过重用这些标准库的功能来满足。我们演示了大多数标准库容器的用法，并介绍了如何利用迭代器将算法应用于各种容器类型。本章还会提到，不同容器支持的是不同种类的迭代器。本章会继续演示如何利

20

[①] 工业强度的系统应该使用标准库的类来做这件事，但这个例子便于我们演示许多关键的“现代 C++”概念。

用 C++20“范围”来简化代码。

20

第 14 章“标准库算法和 C++20 范围/视图”从标准库总计 115 种算法中选择了一部分来介绍，重点放在常见的容器操作上，其中包括用值填充容器、生成值、比较元素或整个容器、移除元素、替换元素、数学运算、查找、排序、交换、拷贝、合并、集合操作、确定边界以及计算最小值和最大值等。我们讨论了对迭代器的最低要求，这样就可以确定每种算法能使用哪些容器。本章开始讨论 C++20“四大”特性中的另一个：概念。C++20 `std::ranges` 命名空间中的算法通过 C++20“概念”来指定它们的要求。我们将通过 C++20“范围”和“视图”来拓展对 C++函数式编程的讨论。

第 5 部分：高级主题

第 15 章“模板、C++20 概念和元编程”讨论了如何使用模板进行泛型编程。自 1998 年 C++标准发布以来，C++语言就一直支持“模板”的概念。随着 C++每个新版本的发布，模板的重要性也日益增加。“现代 C++”的一个重要趋势就是在编译时做更多的事情，以获得更好的类型检查和更好的运行时性能。换言之，如果问题能在编译时解决，就能有效地避免运行时开销，使系统更快。如你所见，模板（特别是模板元编程）是实现强大的编译时处理的关键。本章将深入研究模板，展示如何开发自定义类模板，并探索 C++20 新增的“概念”。你将创建自己的“概念”，将第 11 章的 `MyArray` 案例研究转换为具有自己的迭代器的类模板，并与可接收任意数量模板实参的“可变参数模板”（variadic templates）协同工作。本章会讨论如何进行 C++元编程。

性能

第 16 章“C++20 模块：大规模开发”介绍了 C++20 的另一个“四大”特性。模块（modules）是一种新的代码组织方法，能精确控制向客户代码公开哪些声明，同时封装实现细节。模块有助于提高开发人员的效率，特别是在构建、维护和演化大型软件系统的时候。模块使这些系统能更快地构建，并为其赋予更佳的可伸缩性。C++的缔造者 Bjarne Stroustrup 说：“模块为改善 C++的代码整洁性和编译速度提供了一个历史性的机会（将 C++带入 21 世纪）。”[1]你会发现，即使是一些小型系统，模块也能通过消除对 C++预处理器的需求，从而为每个程序带来直接的好处。我们本来想在程序中集成模块，但在写作本书时，市面上的主流编译器仍然缺失了多种模块功能。

模块

性能

安全性

第 17 章“并行算法和并发性：高级观点”是本书中最重要的章节之一，讲解了 C++如何为创建和管理多个任务的应用程序提供支持。这能显著提升程序的性能和可响应性（responsiveness）。我们展示了如何使用 C++17 的预打包并行算法来创建多线程程序。这种程序在当今的多核计算机架构上运行得更快（而且通常快得多）。例如，我们实验了对

17

性能

20

[1] Bjarne Stroustrup，“Modules and Macros”，http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0955r0.pdf。

1 亿个值先进行顺序排序，再进行并行排序。在逐渐增大核心数量的前提下，我们利用C++的<chrono>库来分析在当今流行的多核系统上获得的性能提升。你会看到，在使用了八核英特尔处理器的 64 位 Windows 10 计算机上，并行排序的运行速度达到了顺序排序的 6.76 倍。我们讨论了生产者–消费者关系，并演示了使用低级和高级 C++并发基元（原语）来实现它的多种方法。这些基元包括 C++20 新增的 latch（闭锁）、barrier（栅栏）和 semaphore（信号量）。有必要强调一点，并发编程很难正确实现，所以应该尽可能地使用高级并发特性。像信号量和原子类型（atomic）这样的低级特性可以用来实现像闭锁这样的高级特性。

20 软件工程

第 18 章“C++20 协程”介绍了所谓的“协程”，这是 C++20“四大”特性中的最后一个。协程（coroutine）是一种可以暂停执行并在稍后由程序的另一部分继续执行的函数。为其提供支持的机制完全由编译器自动为你生成的代码来处理。任何函数只要使用了 co_await、co_yield 和 co_return 关键字之一，就可以被认为是一个协程。协程使你能以简单的顺序编码风格进行并发编程。协程需要一个复杂的基础结构，你可以自己写，但那样会很复杂、乏味而且容易出错。相反，大多数专家都认为，我们应尽可能使用高级协程支持库，本节演示的就是这种方法。开源社区已经创建了好几个实验性的库，使我们能快速、方便地开发协程，本书利用了其中两个。预计 C++23 会开始提供对协程的标准库支持。

23

附录

附录 A“操作符优先级和分组”按从高到低的优先级顺序列出了 C++支持的操作符。

附录 B“字符集”列出了字符及其数字编码。

P.8 如何访问在线章节和附录

我们在 informit.com 上提供了几个在线章节和附录。请按以下步骤来注册本书并访问以下在线内容：

1. 访问 https://informit.com/register，用现有帐户登录或者创建新帐户。
2. 在“Register a Product”区域的 ISBN 框中输入 9780136905691，单击“Submit”。
3. 在你的帐户页面的“My Registered Products”区域，单击“C++20 for Programmers: An Objects-Natural Approach”下的“Access Bonus Content”链接。

随后将显示本书的在线内容页面，单击下载链接即可获取全部内容（40 多 MB）。

在线章节

20

第 19 章“流 I/O 和深入理解 C++20 文本格式化”讨论了标准的 C++输入/输出功能和

<iomanip>库遗留的格式化特性。之所以讨论这些格式化特性，是因为程序员可能在遗留C++代码中遇到它们。另外，本章还更深入地讨论了C++20新的文本格式化特性。

23 第20章“其他主题和对C++23的展望”介绍了杂七杂八的C++主题，并展望了C++23以及后续版本预期的新特性。

在线附录

附录C“数字系统”概述了二进制、八进制、十进制和十六进制数字系统。

附录D“预处理器”讨论了C++预处理器的其他特性。我们在讨论模板元编程（第15章）和C++20模块（第16章）时，已经讨论了C++预处理器的许多特性，本附录只是起到查漏补缺的作用。

附录E“位操作”讨论了按位（bitwise）操作符，它们对整数操作数的单独的二进制位进行操作。另外，本附录还讨论了用于紧凑表示整数数据的“位域”（bit fields）

deitel.com提供的在线资料

我们在deitel.com网站为本书设立了专页：

https://deitel.com/c-plus-plus-20-for-programmers

其中包含以下附加资源：

- 到GitHub仓库的链接，提供本书C++源代码的下载
- 博客文章：https://deitel.com/blog
- 本书后续更新

要想更多地了解如何下载源代码以及C++开发环境的设置，请参见稍后的“准备工作”部分。

P.9 C++核心准则

核心准则 C++“核心准则”打印下来约有500页：

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

它们由Bjarne Stroustrup（C++的缔造者）和Herb Sutter（ISO C++标准委员会召集人）编

辑，是旨在“帮助人们有效使用现代 C++”[①]的一些建议。以下内容来自它的“摘要”：

软件工程
错误
性能

“该准则聚焦于相对高层次的问题，例如接口、资源管理、内存管理以及并发等。这些规则影响的是应用程序架构和库的设计。和现今在代码中常见的情况相比，遵循这些规则可以保证代码的静态类型安全，无资源泄漏，并能捕捉更多的编程逻辑错误。除此之外，还能使代码运行得更快，你将更容易地把事情做对。”[②]

核心准则

贯穿全书，我们将尽可能坚持这些准则。许多在“核心准则”中提供的建议我们都会用相应的图标指明。几百条核心准则被划分为多个类别和子类别。虽然信息量听起来似乎有点过度，但利用 P.10 节描述的静态代码分析工具，可以根据这些准则自动化检查你的代码。

准则支持库

“C++核心准则”会经常提到一个“准则支持库”（Guidelines Support Library，GSL），它实现了用于为各种建议提供支持的辅助类和函数[③]。Microsoft 在 GitHub 上提供了一个开源的 GSL 实现：

https://github.com/Microsoft/GSL

本书早期章节的一些例子已经在使用 GSL 特性。其中一些 GSL 特性后来被纳入了 C++标准库。

P.10 工业强度的静态代码分析工具

错误
安全性

利用静态代码分析工具，可以快速检查代码是否存在常见的错误和安全问题，并提供代码改进意见。使用这些工具就像是让世界级的专家帮你检查代码。为了帮助我们遵守“C++核心准则”，并从总体上改进代码，我们使用了以下静态代码分析工具：

- `clang-tidy`——https://clang.llvm.org/extra/clang-tidy/
- `cppcheck`——https://cppcheck.sourceforge.io/
- Microsoft 的“C++核心准则”静态代码分析工具，它内置于 Visual Studio 的静态代码

[①] C++ Core Guidelines，“Abstract”，https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-abstract。

[②] C++ Core Guidelines，“Abstract”。

[③] C++ Core Guidelines，“GSL: Guidelines Support Library”，https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-gsl。

分析器中

使用上述三种工具，我们对本书代码示例进行检查以确认：

- 遵循 C++核心准则
- 遵循编码标准
- 遵循“现代 C++”惯例
- 是否存在可能的安全问题
- 是否存在常见 bug
- 是否存在可能的性能问题
- 代码的可读性
- 以及更多……

错误

我们还在 GNU g++和 Clang C++编译器中使用了编译器标志`-Wall`来启用所有编译器警告。除了一些超出本书范围的警告，我们确保程序在编译时没有警告消息。要更多地了解静态分析工具的配置，请参见稍后的“准备工作”一节。

P.11 教学方法

安全性

本书包含丰富的“实时编码”示例。我们强调程序的清晰性，并专注于构建良好工程的软件。

使用不同字体进行强调

关键术语**加黑**使其更醒目，C++代码使用等宽字体（例如，`x = 5`），并使用引号来表示屏上元素（例如，“文件”菜单）。

学习目标和大纲

每一章都从“学习目标”开始，简单概括你希望在本章达成的目标。

表格和插图

本书使用了丰富的表格和插图。

编程技巧和关键特性

编程技巧和关键特性在页边用相应的图标表示，详情参见本“前言”的 P.5 节。

P.12 开发者资源

StackOverflow

StackOverflow 是最受欢迎的面向开发人员的问答网站之一。程序员遇到的许多问题已经在这里讨论过了，所以是寻找这些问题的解决方案和发布新问题的一个好地方。在本书写作过程中，当我们在谷歌上查询各种问题(往往是复杂的那种)时，都会将 StackOverflow 的答案作为首选。

GitHub

“准备[成为程序员]的最佳途径就是编写程序并研究其他人已经写好的优秀程序。我自己以前就是去翻计算机科学中心的垃圾桶，找出他们的操作系统的程序清单。”[①]

——比尔·盖茨

安全性

GitHub 是寻找免费的开源代码并集成到自己项目中的一个很好的地方。如果你愿意的话，还可以将自己的代码贡献给开源社区。目前有超过 8300 万名开发人员在使用 GitHub[②]。网站拥有超过 2 亿个以多种编程语言编写的代码仓库[③]。仅仅是过去一年，开发者就为 6100 多万个代码仓库做出了贡献[④]。GitHub 为专业软件开发人员提供了版本控制工具，可以帮助开发团队管理公共开源项目和私人项目。

C++开源社区很庞大。在 GitHub 上，有超过 47 000 个 C++代码仓库[⑤]。你可以在 GitHub 上查看其他人的 C++代码。如果喜欢，甚至可以在此基础上构建自己的。这是一种很好的学习方式，是我们的“实时编码”教学方法的自然延伸[⑥]。

2018 年，Microsoft 斥资 75 亿美元收购了 GitHub。作为软件开发人员，你几乎肯定会经常使用 GitHub。根据 Microsoft 首席执行官萨蒂亚·纳德拉（Satya Nadella）的说法，该公司

---

① 引自苏珊·拉默斯所著的《Programmers at Work: Interviews with 19 Programmers Who Shaped the Computer Industry》（编程大师访谈录），Microsoft Press，1986，p. 83。

② 2022 年 7 月 10 日的数据。最新数据请访问 https://github.com/about。

③ 2022 年 7 月 10 日的数据。最新数据请访问 https://github.com/about。

④ 2022 年 7 月 10 日的数据。最新数据请访问 https://octoverse.github.com。

⑤ 2022 年 7 月 10 日的数据。C++最新统计数据请访问 https://github.com/topics/cpp。

⑥ 学生需要熟悉 GitHub 上软件的各种开源许可证。

收购 GitHub 是为了“尽一切努力让每个开发人员构建、创新和解决世界上最紧迫的挑战”[①]。

我们鼓励你研究并执行 GitHub 上大量的开源 C++代码，并贡献你自己的代码。

P.13 Docker

安全性

Docker 是一种将软件打包成容器的工具，它捆绑了跨平台方便地执行该软件所需的一切。某些软件包要求进行复杂的安装和配置。对于许多这样的软件，你都可以下载现成的、免费的 Docker 容器，从而避免复杂的安装问题。然后，可以简单地在台式机或笔记本上本地执行软件，这使 Docker 成为帮助你快速、方便和经济地开始使用新技术的一个好方法。

我们展示了如何安装和执行 Docker 容器，它们预先配置了：

- GNU 编译器集合（GCC），其中包括 g++编译器
- 最新版 Clang 的 clang++编译器

上述每一种都可以在 Windows、macOS 和 Linux 平台上的 Docker 中运行。

Docker 还有利于“可重复性”或“再现性”（reproducibility）。自定义 Docker 容器可以使用你当前选择的软件和库来配置。这使其他人能重建你的环境，并再现你的工作，也能帮助你再现自己的结果。可重复性在科学和医学领域尤其重要——例如，在研究人员想要对已发表文章中的工作进行证明和扩展的时候。

P.14 一些关键的 C++文档和资源

本书引用了超过 900 多个视频、博客、文章和在线文档。在阅读本书的过程中，建议你根据需要访问这些资源中的一些，以研究更多的高级特性和编程惯例。zh.cppreference.com 网站已成为事实上的 C++文档网站。我们时常引用它，方便你获得关于标准 C++类和函数的更多细节。我们还会经常引用 C++20 标准文件的最终草案，它可从 GitHub 上免费获取，网址是：

https://timsong-cpp.github.io/cppwp/n4861/

在阅读本书的过程中，以下 C++资源也会很有帮助：

[①] “Microsoft to Acquire GitHub for $7.5 Billion”，https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/。

文档

- 由 C++标准委员会通过的 C++20 标准文档最终草案：
  https://timsong-cpp.github.io/cppwp/n4861/
- zh.cppreference.com 提供的“C++参考手册”：
  https://zh.cppreference.com/
- Microsoft 的 C++语言文档：
  https://docs.microsoft.com/zh-cn/cpp/cpp
- GNU C++标准库参考手册：
  https://gcc.gnu.org/onlinedocs/libstdc++/manual/index.html

博客

- Sutter’s Mill 博客，这是 Herb Sutter 的软件开发博客：
  https://herbsutter.com/
- Microsoft C++团队的博客：
  https://devblogs.microsoft.com/cppblog
- Marius Bancila 的博客：
  https://mariusbancila.ro/blog/
- Jonathan Boccara 的博客：
  https://www.fluentcpp.com/
- Bartlomiej Filipek 的博客：
  https://www.cppstories.com/
- Rainer Grimm 的博客：
  http://modernescpp.com/
- Arthur O'Dwyer 的博客：
  https://quuxplusone.github.io/blog/

其他资源

- Bjarne Stroustrup 的网站：
  https://stroustrup.com/
- Standard C++ Foundation 网站：
- https://isocpp.org/
- C++标准委员会网站：
  http://www.open-std.org/jtc1/sc22/wg21/

P.15 自己寻找答案

下面是一些关于 C++和常规编程的网上论坛：

- https://stackoverflow.com
- https://www.reddit.com/r/cpp/
- https://groups.google.com/g/comp.lang.c++
- https://www.dreamincode.net/forums/forum/15-c-and-c/

其他有价值的网站的一个清单可参见 https://tinyurl.com/2npmj3ld。

另外，厂商经常会为他们的工具和库提供论坛。许多库是在 github.com 上管理和维护的。某些库的维护者会通过该库的 GitHub 页面上的“议题”（issues）标签提供支持。

与作者沟通

阅读本书的过程中遇到任何问题，可以通过以下方式联系我们：

deitel@deitel.com

我们会及时做出答复。

P.16 加入 Deitel & Associates 公司的社交媒体社区

Deitel 的社交媒体帐号包括：

- LinkedIn®——https://bit.ly/DeitelLinkedIn
- YouTube®——https://youtube.com/DeitelTV
- Twitter®——https://twitter.com/deitel
- Facebook®——https://facebook.com/DeitelFan

P.17 O'Reilly Online Learning 上的 Deitel Pearson 产品

你所在的公司或大学或许已经订阅了 O'Reilly Online Learning，使你能免费访问 Deitel 在 Pearson 出版的所有电子书和 LiveLessons 视频。除此之外，还能免费访问 Paul Deitel 提供的为期一天的 Full Throttle 培训课程。个人可在以下网址注册 10 天免费试用：

https://learning.oreilly.com/register/

要完整了解 O'Reilly Online Learning 提供的最新产品和课程，请访问：

https://deitel.com/LearnWithDeitel

教科书和专业书

O'Reilly Online Learning 上的每一本 Deitel 电子书都是全彩的，提供了全面的索引和文本查找功能。我们每写一本专业书，都会提前发布到 O'Reilly Online Learning 上，供早期的“粗加工”（rough cut）观看。书定版之后，会用最终内容取代这些粗加工内容。本书英文版的最终版电子书可访问以下网址获得：

https://learning.oreilly.com/library/view/c-20-for-programmers/9780136905776

LiveLessons 视频产品

和 Paul Deitel 一起动手学，他在视频中介绍了 C++、Java、Python 以及 Python 数据科学/人工智能中引人注目的前沿计算技术（还有更多即将推出）。O'Reilly 订阅者可通过以下网址访问本书的 LiveLessons 视频：

https://learning.oreilly.com/videos/c-20-fundamentals-parts/9780136875185

这些视频是自己掌握学习进度时的理想选择。本书写作时，我们仍在录制这个产品。其他视频会在 2022 年第 1 和第 2 季度发布。最终的视频产品将包含 50~60 小时的视频，约相当于两个大学学期的课程。

现场 Full-Throttle 培训课程

Paul Deitel 在 O'Reilly Online Learning 网站提供现场 Full-Throttle（快节奏）培训课程：

https://deitel.com/LearnWithDeitel

这是我们推出的为期一天的、仅授课（无学生互动）、快节奏的、代码密集型的现场授课方式，主题涉及 Python、Python 数据科学/AI、Java、C++20 基础和 C++20 标准库。这些课程的目标学员是有经验的开发人员和软件项目经理，他们准备选择一种新的语言来完成项目，所以需要大致了解一下。在完成一节 Full-Throttle 课程后，学员通常可以选择观看相应的 LiveLessons 视频课程，其授课时间更长，并符合课堂学习的节奏。

P.18 讲师指导的现场培训

Paul Deitel 三十年来一直在为开发人员教授编程语言。他开设了各种为期一至五天的 C++、Python 和 Java 企业培训课程，并为加州大学洛杉矶分校安德森管理学院的商业分析科学硕士（MSBA）课程教授 Python 与数据科学入门。他可以在全球范围内现场或虚拟授课。请

联系 deitel@deitel.com，获得为满足贵公司或学术项目的需求而定制的方案。

P.19 本书的大学教科书版

本书的大学教科书版是《C++ How to Program》第 11 版，它将以三种数字格式提供：

- 主流电子书供应商提供的网上电子书。
- 交互式 Pearson eText（见下）。
- 带考评功能的交互式 Pearson Revel（见下）。

所有这些教科书版本都支持标准的“How to Program”特色，例如：

- 专门用一章介绍硬件、软件和互联网的概念。
- 面向新手的编程介绍。
- 节末的编程和非编程检查点自测练习（附答案）。
- 章末练习。

Deitel Pearson eText 和 Revel 版包括：

- Paul Deitel 讲解书中核心章节材料的视频。
- 交互式的编程和非编程检查点自测题（附答案）
- 抽认卡（flashcards）和其他学习工具。

此外，Pearson Revel 还支持交互式编程和非编程的自动考评练习，还提供了教师课程管理工具，如成绩册。

如果符合资格的大学教师用教科书进行教学，还可获得以下补充资源：

- 教参，包括大多数章末练习的答案。
- 课堂作业文件，包括分为基于代码和非代码的多选题及答案。
- 可定制的 PowerPoint 课堂幻灯片。

请发送邮件至 deitel@deitel.com 了解详情。

P.20 致谢

感谢 Barbara Deitel 为本项目付出的长时间的网上研究。很幸运能与 Pearson 的专业出版团队合作。感谢朋友和同事 Pearson IT Professional Group 副总裁 Mark L. Taub 的努力和长达 27 年的指导。Mark 和他的团队出版了我们的专业书籍和 LiveLessons 视频产品，并赞助了我们通过 O'Reilly Online Learning 服务提供的实时在线培训：

https://learning.oreilly.com/

Charvi Arora 招募了本书的审稿人团队并管理审稿过程。Julie Nahil 管理本书的生产。Chuti Prasertsith 设计了封面。

审稿人

在本项目中，我们很幸运地请到了 10 位杰出的专业人士来审校书稿。大多数审稿人要么是 ISO C++标准委员会的成员，要么曾在该委员会任职，要么与委员会有工作关系。许多人都为该语言做出了一些贡献。他们帮助我们制作了一本更好的书，书中如果还有任何错漏，那么肯定是我们自己的问题。

- Andreas Fertig，独立 C++培训师和顾问，cppin-sights.io 的创建者，《Programming with C++20》一书的作者。
- Marc Gregoire，Nikon Metrology 公司的软件架构师，Micrisoft Visual C++ MVP 和《Professional C++》第 5 版的作者（本书针对 C++20）。
- Dr. Daisy Hollman，ISO C++标准委员会成员。
- Danny Kalev，博士和认证系统分析师/软件工程师，ISO C++标准委员会前成员。
- Dietmar Kühl，Bloomberg L.P.高级软件开发师，ISO C++标准委员会成员。
- Inbal Levi，SolarEdge Technologies，ISO C++基金会总监，ISO C++ SG9（“范围”）主席，ISO C++标准委员会成员
- Arthur O'Dwyer，C++培训师，CppCon 的“Back to Basics”分主题主席，几个已被接受的 C++17/20/23 提案以及《Mastering the C++17 STL》一书的作者。
- Saar Raz，Swimm.io 高级软件工程师，Clang 中 C++20“概念”的实现者。
- José Antonio González Seco，安达卢西亚议会议员
- Anthony Williams，英国标准协会 C++标准小组成员，Just Software Solutions Ltd.的总监，《C++ Concurrency in Action》第 2 版的作者（Anthony 是许多 C++标准委员会文件的作者或共同作者，这些文件最终促成了 C++标准化的“并发”特性）。

Arthur O’Dwyer

我们想强调 Arthur O'Dwyer 为审校我们的手稿所付出的非凡努力。通过看他的批注，我们学到了很多关于 C++的微妙之处，特别是一些“现代 C++”编码惯例。除了仔细标记我们发给他的每一章 PDF 文件，Arthur 还单独提供了一份综合性的文档，其中详细解释了他的意见，还经常会重写代码，并提供外部参考资源以提供额外的见解。当我们综合所有审稿人的意见时，我们总是期待着来自 Arthur 的，特别是和更具挑战性的问题有关的那些。他是一个忙碌的专业人士，但他对自己的时间很慷慨，而且总是富有建设性。他坚持要我们“把它做好”，并努力帮助我们做到这一点。Arthur 向专业人士教授 C++。他教会了我们很多关于如何正确使用 C++的知识。

GitHub

感谢 GitHub 使我们能轻松分享代码并保持更新，感谢它提供的工具，使 8300 多万开发者能够为 2 亿多的代码仓库做出贡献[①]。这些工具支持大规模的开源社区，为当今流行的编程语言提供库，使开发者更容易创建强大的应用程序，同时避免“重新发明车轮”。

Matt Godbolt 和 Compiler Explorer

感谢 Compiler Explorer 的创造者 Matt Godbolt（https://godbolt.org）。这个工具使你能编译和运行以多种编程语言写的程序。通过这个网站，你可以这样测试自己的代码：

- 在大多数流行的 C++编译器上测试，其中包括我们首选的三个编译器
- 在许多已发布的、开发中的和实验性的编译器版本上测试

例如，我们使用了一个实验性的 g++编译器版本来演示“契约”（参见第 12 章），该特性有望在未来的 C++语言版本中标准化。我们的几位审稿人使用 godbolt.org 向我们展示了建议的修改，帮助我们改进本书。

Dietmar Kühl

感谢 Bloomberg L.P.高级软件开发师和 ISO C++委员会成员 Dietmar Kühl，他与我们分享了他对继承、静态/动态多态性的看法。他的见解帮助我们完善了第 10 章和第 15 章对这些主题的讲述。

Rainer Grimm

感谢 Rainer Grimm（http://modernescpp.com/），他是“现代 C++”社区最多产的博主之一。随着我们对 C++20 的深入了解，我们的谷歌搜索经常指向他的著作。Rainer Grimm 是一位专业的 C++培训师，提供德语和英语课程。他是几本 C++书籍的作者，包括《C++20: Get the Details ++20：了解细节》、《Concurrency with Modern C++》、《The C++ Standard Library》第 3 版和《C++ Core Guidelines Explained》。他已开始在博客上介绍有望在 C++23 中出现的特性。

Brian Goetz

我们有幸请到了 Oracle 的 Java 语言架构师和《Java Concurrency in Practice》的合著者 Brian Goetz 对我们的另一本书《Java How to Program》第 10 版进行审稿。他为我们提供了许多

[①] 2022 年 7 月 10 日的数据。最新数据请访问 https://github.com/about。

见解和建设性的意见，特别是关于：

- 继承层次的设计，这影响了本书第 10 章的几个例子的设计决策。
- Java 并发性，这影响了本书第 17 章对 C++20 并发性的讲述。

开源贡献者和博客作者

特别感谢全世界为开源运动做出贡献并在网上发表博客的技术人员，并感谢乐意促进开放软件和信息的组织。

谷歌搜索

感谢谷歌，它的搜索引擎回答了我们源源不断的查询，每次都在几分之一秒内完成，无论白天还是晚上，而且还免费。这是我们过去 20 年的研究过程中所享受到的最好的生产力提升工具，没有之一。

Grammarly

我们现在对所有稿件都使用 Grammarly 的付费版本。他们将这个工具描述成“人工智能驱动的写作助手”，能帮助你“写出大胆、清晰、没有错误的文章”[①]。 他们还说：“利用各种创新方法——包括先进的机器学习和深度学习——我们不断在自然语言处理（NLP）研究方面取得新的突破，为用户提供无与伦比的帮助。”[②]Grammarly 提供了免费工具，可以将其整合到流行的 Web 浏览器、Microsoft® Office 365™和 Google Docs™中。他们还提供更强大的高级和商业工具。可以访问以下网址查看他们的免费和付费套餐：

https://www.grammarly.com/plans

在阅读本书并使用代码示例时，我们感谢你的评论、批评、指正和改进建议。包括疑问在内的所有信件请发送到：

deitel@deitel.com

我们会及时做出答复。

欢迎来到令人振奋的 C++20 编程世界。过去 30 年，我们总共写了 11 版的学术和专业 C++书籍。我们希望本书能为您提供一个信息丰富、富有挑战性和娱乐性的学习体验，并帮助

[①] Grammarly 的主页是 https://www.grammarly.com。

[②] “Our Mission”，https://www.grammarly.com/about。

您迅速进入“现代 C++”软件开发的状态。

Paul Deitel

Harvey Deitel

作者简介

Paul J. Deitel 是 Deitel & Associates 公司 CEO 兼 CTO，具有 42 年计算机行业的工作经验，毕业于麻省理工学院。Paul 是世界上最有经验的编程语言培训师之一，自 1992 年以来一直在为软件开发人员教授专业课程。他向 Deitel & Associates 公司的院校、行业、政府和军事客户提供了数以百计的编程课程，这些客户包括：加州大学洛杉矶分校（UCLA）、思科（Cisco）、IBM、西门子（Siemens）、Sun Microsystems（现在的 Oracle）、戴尔（Dell）、Fidelity、肯尼迪航天中心的 NASA、美国国家强风暴实验室（National Severe Storm Laboratory）、白沙导弹发射场（White Sands Missile Range）、Rogue Wave Software、波音（Boeing）、彪马（Puma）、iRobot 等等。他和他的合著者 Harvey M. Deitel 博士是世界上最畅销的编程语言课本、专业书籍、视频和互动多媒体电子学习内容的作者以及虚拟/现场培训的主持人。

Harvey M. Deitel 博士是 Deitel & Associates 公司主席兼首席战略官，具有 61 年以上的计算机行业工作经验。Deitel 博士在麻省理工学院获得电子工程学士和硕士学位，在波士顿大学获得数学博士学位——在将计算机科学专业从这些专业分离出去之前，Deitel 博士已经学习过计算机知识。他具有丰富的大学教学经验，在与儿子 Paul 于 1991 年创立 Deitel & Associates 公司之前，Deitel 博士是波士顿大学计算机科学系主任并获得了终身教职。他们的出版物已经赢得了国际声誉，并被翻译成了日文、德文、俄文、西班牙文、法文、波兰文、意大利文、简体中文、繁体中文、韩文、葡萄牙文、希腊文、乌尔都文和土耳其文。Deitel 博士为许多大公司、学术机构、政府部门和军事机构提供了数百场的专业编程培训。

Deitel® & Associates 公司简介

Deitel & Associates 公司由 Paul Deitel 和 Harvey Deitel 创立，是一家国际知名的创作和企业培训组织，擅长计算机编程语言、对象技术、移动应用开发以及互联网和 Web 软件技术。公司的培训客户包括一些全球最大型的公司、政府机构、军事部门和学术机构。公司提供讲师现场指导的培训课程，为世界各地的客户提供虚拟或现场培训，并在 O'Reilly Online Learning（前称是 Safari Books Online）上为 Pearson Education 提供虚拟培训（https://learning.oreilly.com）。

通过与 Pearson 公司 47 年的出版合作，Deitel & Associates 以印刷和电子书形式发布了大量行业领先的编程专业书籍和大学教科书、LiveLessons 视频课程、O'Reilly Online Learning

实时培训课程和 Revel™交互式多媒体大学课程。

如需联系 Deitel & Associates 公司和作者，或要求提供全球范围内的虚拟或现场由讲师指导的培训方案，请致函：

deitel@deitel.com

要更多地了解 Deitel 的虚拟和现场企业培训，请访问：

https://deitel.com/training

个人如需购买 Deitel 的书籍，请访问：

https://amazon.com

https://www.barnesandnoble.com/

企业、政府、军队和学术机构的大宗订单应直接联系 Pearson 公司。企业和政府销售请致函：

corpsales@pearsoned.com

Deitel 电子书支持各种格式，请访问以下网站购买：

https://www.amazon.com/ https://www.vitalsource.com/

https://www.barnesandnoble.com/ https://www.redshelf.com/

https://www.informit.com/ https://www.chegg.com/

如需注册 10 天免费的 O'Reilly Online Learning，请访问：

https://learning.oreilly.com/register/

准备工作

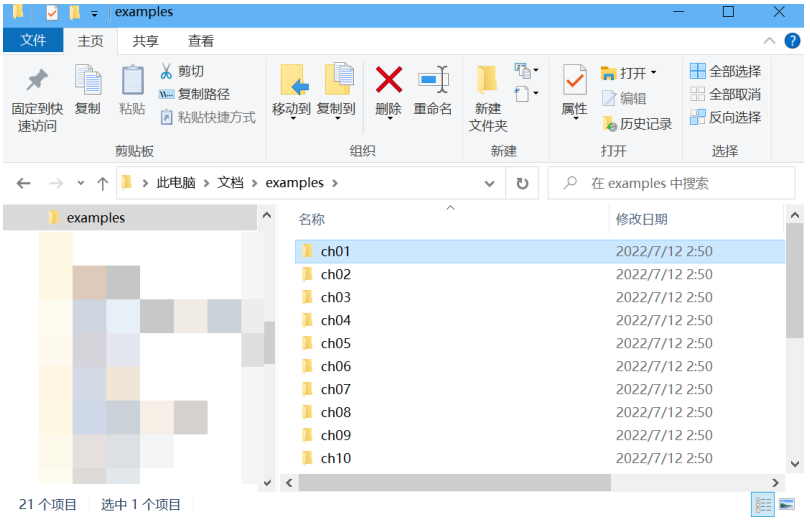
使用本书之前，请先了解本书采用的约定，并设置好你的计算机以编译和运行示例程序。

字体和命名约定

- 重要术语用黑体显示，例如“这种函数称为类的**成员函数**”。
- C++代码元素使用一种等宽代码字体，例如 `sqrt(9)`。
- 提到屏幕元素时会使用引号，例如“文件”菜单。
- 提到菜单操作的时候，会用“|”来表示从一个菜单中选择菜单项。所以，“文件”|“打开”意味着从“文件”菜单中选择“打开”。
- 提到组合键的时候用加号分隔，例如“按 Ctrl+F5”。
- 要求用户输入的内容加粗显示，例如“请输入 **500**”。

获得代码示例

我们在一个 GitHub 仓库中维护本书的代码示例。本书主页（https://deitel.com/cpp20fp）的 Source Code 区域提供了到该 GitHub 仓库的链接，还提供了源代码 ZIP 文件的下载链接。如果你熟悉 Git 和 GitHub，就把仓库克隆到你的系统中。如果下载了 ZIP 文件，一定要解压其中的内容。本书假定这些示例已存储到你的用户帐户下的“文档”文件夹中的一个名为 examples 的子文件夹中。如下图所示。



如果不熟悉 Git 和 GitHub，但有兴趣了解这些基本的开发工具，请查看他们的指南：

https://guides.github.com/activities/hello-world/

本书使用的编译器

在阅读本书之前，请确保你已经安装了一个最新的 C++编译器。我们使用以下免费编译器测试了本书的全部代码示例：

- 在 Microsoft Windows 上使用的是 Microsoft Visual Studio 社区版[①]，其中包括 Visual C++编译器和其他 Microsoft 开发工具。
- 在 macOS 上使用的是 Apple Xcode[②] C++编译器，它使用 Clang C++编译器的一个版本。
- 在 Linux 上使用的是 GNU C++编译器[③]——GNU Compiler Collection（GCC）的一部分。GNU C++在大多数 Linux 系统上已经安装（虽然可能需要将编译器更新到一个较新的版本），并且可以安装到 macOS 和 Windows 系统上。
- 还可以通过 Docker 容器在 Windows、macOS 和 Linux 上方便地运行最新版本的 GNU C++和 Clang C++。请阅读稍后的“Docker 和 Docker 容器”一节。

稍后会讲述编译器和 Docker 的安装。1.2 节的试运行环节演示了如何使用这些编译器编译和运行 C++程序。

某些例子不能在全部三种编译器上编译和运行

本书写作时（2022 年 2 月），编译器厂商还没有完全实现 C++20 的一些新特性。当这些特性可用时，我们会重新测试代码，更新我们的数字产品，并在以下网站发布纸质版的更新信息：

https://deitel.com/cpp20fp

在 Windows 上安装 Visual Studio 社区版

如果你是 Windows 用户，首先确定自己的系统满足 Microsoft Visual Studio 社区版的系统需求：

https://docs.microsoft.com/zh-cn/visualstudio/releases/2022/system-requirements

---

[①] 本书写作时使用的是 Visual Studio 2022 社区版。

[②] 本书写作时使用的是 Xcode 13.2.1。

[③] 本书写作时使用的是 GNU g++ 11.2。

接着访问：

https://visualstudio.microsoft.com/downloads/

再执行以下安装步骤：

1. 单击“社区”区域的“免费下载”。
2. 取决于所用的 Web 浏览器，文件下载回来后可能会在屏幕底部弹出一个提示，单击“运行”即可开始安装。如果是 Chrome 浏览器，单击下载回来的文件名即可开始安装。否则，请自行启动“下载”文件夹中的安装程序。
3. 如果出现“用户帐户控制”对话框，单击“是”允许安装程序修改你的系统。
4. 在“Visual Studio Installer”对话框中，单击“继续”允许安装程序下载配置安装所需的一些组件。
5. 对于本书的示例，请勾选“使用 C++的桌面开发”，其中已包含 Visual C++编译器和 C++标准库。
6. 单击“安装”。取决于你的网速，安装过程可能要花一定的时间。

在 macOS 上安装 Xcode

在 macOS 上执行以下步骤安装 Xcode：

1. 单击苹果图标，选择“App Store...”，或者单击 Mac 屏幕底部 Dock 栏上的 App Store 图标。
2. 在 App Store 的搜索栏中，输入 Xcode。
3. 单击“获取”按钮并安装 Xcode。

安装最新的 GNU C++版本

目前存在许多 Linux 发行版，它们通常采用了不同的软件升级技术。请查阅你所使用的发行版的联机文档，了解将 GNU C++ 升级到最新版本的正确方法。也可以从以下网站下载适用于各种平台的 GNU C++：

https://gcc.gnu.org/install/binaries.html

用 WSL 运行 Ubuntu Linux 并安装 GNU 编译器集合

可以通过“适用于 Linux 的 Windows 子系统”（Windows Subsystem for Linux，WSL）在 Windows 上安装 GNU Compiler Collection。WSL 使你能在 Windows 上运行 Linux。Ubuntu Linux 在 Windows 商店中提供了一个易于使用的安装程序，但你首先必须安装好 WSL：

1. 在任务栏的搜索框中，输入“启用或关闭”，在搜索结果中单击“启用或关闭 Windows 功能”。
2. 在“Windows 功能”对话框中找到“适用于 Linux 的 Windows 子系统”。如果它已被选中，就表明 WSL 已经安装。否则，选中它并单击“确定”。Windows 将安装 WSL 并要求你重启系统。
3. 系统重启后，打开 Microsot Store 应用并搜索 Ubuntu，选择名为 Ubuntu 的应用程序并点击安装。这就安装了最新版本的 Ubuntu Linux。
4. 安装好后单击“打开”按钮来显示 Ubuntu Linux 命令行窗口，这将继续安装过程。会要求你为这个 Ubuntu 安装创建用户名和密码（不需要与你的 Windows 用户名和密码相同）。
5. Ubuntu 安装完成后，执行以下两个命令来安装 GCC 和 GNU 调试器，可能会要求你为第 4 步创建的账户输入密码：

```
sudo apt-get update
sudo apt-get install build-essential gdb
```

6. 执行以下命令来确认 g++已成功安装：

```
g++ --version
```

为了访问我们的代码文件，使用 `cd` 将文件夹更改到：

```
cd /mnt/c/Users/你的用户名/Documents/examples
```

将用户名换成你自己的，将路径更新为你系统中存储了本书示例的位置。

Docker 和 Docker 容器

Docker 是一个将软件打包到容器（也叫映像）中的工具。容器捆绑了跨平台执行该软件所需的一切，这对需要进行复杂安装和配置的软件包特别有用。许多这样的软件包都提供了免费的、现有的 Docker 容器（通常放在 https://hub.docker.com），你可以下载并在自己的系统上本地执行。Docker 是开始使用新技术的一种快速、方便的方法，也特别适合试验新的编译器版本。

安装 Docker

使用 Docker 容器必须先安装好 Docker。Windows 和 macOS 用户请从以下网站下载并运行 Docker Desktop 安装程序：

https://www.docker.com/get-started

然后按照屏幕上的指示操作。另外，可以在这个网页上注册一个 Docker Hub 帐户，以便利

用 https://hub.docker.com 提供的各种容器。Linux 用户应该从以下网站安装 Docker Engine：

https://docs.docker.com/engine/install/

下载 GNU Compiler Collection Docker 容器

GNU 团队在以下网页维护他们的官方 Docker 容器：

https://hub.docker.com/_/gcc

Docker 安装好并开始运行后，请打开一个“命令提示符”[①] (Windows)、Terminal(macOS/Linux)或者 shell(Linux)，然后执行以下命令：

```
docker pull gcc:latest
```

Docker 会下载 GNU 编译器集合（GCC）容器的最新版本（本书写作时是版本 11.2）。在 1.2 节的一次试运行中，我们会演示如何执行容器并用它编译和运行 C++程序。

下载 Clang Docker 容器

目前，Clang 团队没有提供官方 Docker 容器，但 https://hub.docker.com 提供了许多可用的。本书使用了其中一个比较流行的容器：

https://hub.docker.com/r/teeks99/clang-ubuntu

打开一个“命令提示符”[②](Windows)、Terminal(macOS/Linux)或者 shell(Linux)，然后执行以下命令：

```
docker pull teeks99/clang-ubuntu:latest
```

Docker 会下载 Clang 容器的最新版本（本书写作时是版本 13）。在 1.2 节的一次试运行中，我们会演示如何执行容器并用它编译和运行 C++程序。

C++答疑解惑

如果在阅读本书的过程中遇到任何问题，请致函：

[①] Windows 用户应选择以管理员身份打开“命令提示符”窗口。

[②] Windows 用户应选择以管理员身份打开“命令提示符”窗口。

deitel@deitel.com

或者访问：

https://deitel.com/contact-us

我们会及时做出答复。

网上有丰富的编程资料可供参考。其中一个宝贵的资源是 https://stackoverflow.com 网站，它无论对于程序员还是非程序员都非常好用。你可以在这里：

- 查找最常见的编程问题的答案。
- 查找错误消息，了解出错原因。
- 提出编程问题，从世界各地的程序员那里获得答案
- 获得关于编程的宝贵见解

如果想参与实时的 C++讨论，请访问 Slack 的 cpplang 频道：

https://cpplang-inviter.cppalliance.org

以及 Discord 服务器#include<C++>：

https://www.includecpp.org/discord/

在线 C++文档

要查看 C++标准库文档，请访问：

https://zh.cppreference.com

另外，一定要看一下 C++ FAQ：

https://isocpp.org/faq

关于{fmt}文本格式化库

本书许多程序都包含下面这一行代码：

```
#include <fmt/format.h>
```

这使得我们的程序能够使用开源的{fmt}库的文本格式化功能[1]。这些程序包含了对fmt::format 函数的调用。

新的 C++20 文本格式化功能是{fmt}库所提供的特性一个子集。在 C++20 中，上面这行代码应写成：

```
#include <format>
```

还应修改函数调用，变成调用 std::format 函数。

但在本书写作时，只有 Microsoft Visual C++支持新的 C++20 文本格式化功能。所以，本书的例子使用了开源{fmt}库，以确保大部分示例能在我们首选的全部编译器上执行。

静态代码分析工具

我们使用以下静态代码分析工具来检查我们的代码实例是否遵循“C++核心准则”、是否遵循编码标准、是否遵循“现代 C++”惯例、可能的安全问题、常见 bug、可能的性能问题以及代码可读性等等：

- clang-tidy—https://clang.llvm.org/extra/clang-tidy/
- cppcheck—https://cppcheck.sourceforge.io/
- Microsoft 的 C++核心准则静态代码分析工具，它内置于 Visual Studio 的静态代码分析器中

可用以下命令在 Linux 上安装 clang-tidy：

```
sudo apt-get update -y
sudo apt-get install -y clang-tidy
```

可以按照 https://cppcheck.sourceforge.io/上的说明，为多种操作系统平台安装 cppcheck。对于 Visual C++，一旦通过 1.2 节的试运行环节学会了如何创建项目，就可以按以下方式配置 Microsoft 的 C++核心准则静态代码分析工具：

1. 在解决方案资源管理器中右击项目名称，选择“属性”。
2. 在随后出现的对话框中，在左栏中选择“代码分析”|“常规”，然后在右栏中将“生成时启用代码分析”设为“是”。
3. 接着，在左栏中选择“代码分析”|“Microsoft”，然后在右栏中，从下拉列表中选择分析规则的一个特定子集。我们使用选项“<选择多个规则集...>”来选择所有以“C++ Core Check”开头的规则集。单击“另存为...”，为你的自定义规则集起个名字，

[1] 详情请访问 https://github.com/fmtlib/fmt。

点击“保存”，然后单击“应用”。 (注意，这会为本书示例中使用的`{fmt}`文本格式化库产生大量警告。)

第 1 章 介绍免费、流行的 C++编译器并试运行

学习目标：

- 高屋建瓴认识本书架构和 C++20 这种大型、复杂和强大的编程语言。
- 使用我们的三个首选编译器（Windows 上是由 Microsoft Visual Studio 提供的 Visual C++，macOS 上是 Xcode 中的 Clang C++，Linux 上是 GNU g++）尝试编译和运行一个 C++应用程序。
- 了解如何为 g++和 clang++命令行编译器执行 Docker 容器，以便在 Windows、macOS 或 Linux 上使用这些编译器。
- 了解有哪些资源可供了解 C++的历史和 40 多年来的里程碑。
- 了解在“现代 C++”中，为什么并发编程是从当今的多核处理器中挖掘最大性能的关键。
- 概述在本书早期章节的对象自然案例研究中使用的对象技术概念。本书会从第 9 章开始正式讲解面向对象编程技术。

1.1 导读

欢迎来到 C++——世界上最流行的编程语言之一[①]。我们在 C++20 的背景下介绍“现代 C++”。C++20 是由国际标准化组织（ISO）标准化的最新版本。本章将帮助你：

- 快速了解本书的上层建筑。
- 试运行几个最流行的免费 C++编译器。
- 理解摩尔定律和多核处理器，以及为什么并发编程对在“现代 C++”中构建高性能应用程序至关重要。
- 理解面向对象编程的概念和术语。

本书的上层建筑

在开始深入学习本书之前，建议你对本书的上层建筑有一个“高屋建瓴”的认识，以掌控在学习 C++20 这种大型、复杂和强大的编程语言时的方向。为此，我们建议你先查看下面这些内容：

- 本书英文版封面内页的单页全彩目录图，它提供了本书的高层概览。这幅图的 PDF 版

[①] 出自“TIOBE 编程社区指数”，网址是 https://www.tiobe.com/tiobe-index。

本可从以下网址下载：https://deitel.com/cpp20fpTOCdiagram。

- 本书英文版封底包含了对该书的简要介绍、关键特性列表和一些书评。封底内页还包含了更多内容。这些书评来自对印前手稿进行审查的 C++行业专家。阅读它们可以让你很好地了解审稿人认为的本书重要特色。这些书评也被张贴在本书主页上，网址是 https://deitel.com/cpp20fp。
- 本书“前言”展示了本书的大纲，解释了本书如何围绕“现代 C++”编程方法展开。我们介绍了“对象自然方法”，它鼓励使用少量的、简单的 C++语句来指挥强大的类执行重要的任务，同时无需创建自定义类。请务必阅读前言中的“本书导读”一节，它描述了每一章的主要特色。在看这些导读的过程中，可以同时参考前面所说的“目录图”。

C++历史

1979 年，Bjarne Stroustrup 开始创建 C++，他当时称之为“C with Classes”（包含类的 C）[①]。现在，至少有 500 万开发人员（有的估计高达 750 万[②③]）使用 C++构建广泛的关键业务和关键任务系统和应用软件[④⑤]。当今流行的桌面操作系统 Windows[⑥]和 macOS[⑦]也有一部分是用 C++编写的。许多流行的应用程序也部分用 C++编写，包括 Web 浏览器（如 Google

[①] https://en.wikipedia.org/wiki/Bjarne_Stroustrup

[②] “State of the Developer Nation, 21st Edition”，Q3 2021，https://www.slashdata.co/free-resources/state-of-the-developer-nation-21st-edition。

[③] Tim Anderson，“Report: World's Population of Developers Expands, Javascript Reigns, C# Overtakes PHP”，April 26, 2021，https://www.theregister.com/2021/04/26/report_developers_slashdata/。

[④] “Top 10 Reasons to Learn C++”，https://www.geeksforgeeks.org/top-10-reasons-to-learn-c-plus-plus/。

[⑤] “What Is C++ Used For? Top 12 Real-World Applications and Uses of C++”，https://www.softwaretestinghelp.com/cpp-applications/。

[⑥] “What Programming Language Is Windows Written In?”，https://social.microsoft.com/Forums/en-US/65a1fe05-9c1d-48bf-bd40-148e6b3da9f1/what-programming-language-is-windows-written-in。

[⑦] https://zh.wikipedia.org/wiki/MacOS

Chrome[①]和 Mozilla Firefox[②]）、数据库管理系统（如 MySQL[③]和 MongoDB[④]）等等。

C++的历史和重要的里程碑事件在下面这些地方进行了详细的记录：

- 维基百科的 C++页面提供了一个详细的 C++历史，还罗列了许多参考文献：https://zh.wikipedia.org/wiki/C++
- C++的缔造者 Bjarne Stroustrup 提供了语言及其从开始到 C++20 的设计过程的详细历史：https://www.stroustrup.com/C++.html#design
- zh.cppreference.com 提供了一个自 C++诞生以来的里程碑列表，其中包括许多参考文献：https://zh.cppreference.com/w/cpp/language/history

1.2 试运行一个 C++20 应用程序

在本节中，你将编译、运行并和你的第一个 C++应用程序交互[⑤]。这是一个猜数游戏，它从 1 到 1000 中随机挑选一个数字，并提示你去猜它。如果你猜中，游戏结束。如果猜错，应用程序会显示你猜的数是比正确数字高还是低。猜的次数没有限制。

编译器和 IDE 试运行总结

我们将展示如何使用以下编译器来编译和执行 C++代码：

- Windows 平台上的 Microsoft Visual Studio 2022 社区版(1.2.1 节)
- macOS 平台上的 Apple Xcode 中的 Clang(1.2.2 节)
- Linux 平台上用一个外壳程序执行的 GNU g++(1.2.3 节)
- 在 GNU Compiler Collection(GCC) Docker 容器内部，用一个外壳程序执行的 g++(1.2.4 节)
- 在一个 Docker 容器内部，用一个外壳程序执行的 clang++，这是 Clange C++编译器的

---

① https://zh.wikipedia.org/wiki/Google_Chrome

② https://zh.wikipedia.org/wiki/Firefox

③ https://zh.wikipedia.org/wiki/MySQL

④ https://zh.wikipedia.org/wiki/MongoDB

⑤ 这里故意不涉及该 C++程序的代码。这个例子只是为了演示如何使用本节讨论的每种编译器来编译和运行程序。我们将在第 5 章讨论随机数生成。

命令行版本(1.2.5 节)

注意，可以只选读和你的平台对应的小节。要使用 g++和 clang++的 Docker 容器，必须先安装并运行 Docker，详情请参见本章之前描述的“准备工作”。

1.2.1 在 Windows 上用 Visual Studio 2022 社区版编译和运行 C++20 应用程序

本节演示如何使用 Microsoft Visual Studio 2022 社区版在 Windows 上运行一个 C++程序。Visual Studio 有几个版本；在某些版本中，我们介绍的选项、菜单和操作指示可能略有区别。后文将简单地说“Visual Studio”或“IDE”。

步骤 1：检查安装

如果尚未安装，请参考之前的“准备工作”了解如何安装 IDE 并下载本书代码示例。

步骤 2：启动 Visual Studio

启动 Visual Studio。如果是新安装的 Visual Studio，可以需要等一段时间才能就绪。出现启动窗口后，请按 Esc 键将其关闭。注意不要单击右上角的 X，否则会终止 Visual Studio。可以在任何时候选择“文件”|“启动窗口”显示该窗口。注意，本书在涉及菜单操作的时候，会用“|”来表示从一个菜单中选择菜单项。所以，“文件”|“打开”意味着从“文件”菜单中选择“打开”。

步骤 3：创建项目

“项目”（project）是指一组相关的文件，例如构成一个应用程序的 C++源代码文件。Visual Studio 用项目和解决方案来组织应用程序。“解决方案”（solution）包含一个或多个项目。多项目解决方案常用于创建大规模的应用程序。本书每个应用程序都是只包含单个项目的解决方案。对于我们的代码示例，你将从一个空项目开始，并向其中添加文件。请按以下步骤创建一个项目：

1. 选择“文件”|“新建”|“项目”，显示“创建新项目”对话框。
2. 首先选择“C++”、“Windows”和“控制台”这三个标签，再选择“空项目”模板。这个项目模板适用于在“命令提示符”窗口的命令行上执行的程序。取决于你的 Visual Studio 版本及其安装选项，可能还会安装其他许多项目模板。可以使用顶部的“搜索模板”文本框和它下方的下拉列表来过滤选择。单击“下一步”来显示“配置新项目”对话框。
3. 指定项目名称和位置。为项目名称输入 **cpp20_test**，为位置选择本书的 examples 文件

夹。单击创建，随后会在 Visual Studio 中打开这个新项目。

Visual Studio 会创建项目，将该项目文件夹放到以下文件夹中，然后显示主界面：

```
C:\Users\你的用户名\Documents\examples
```

编辑 C++代码时，Visual Studio 将每个文件作为一个单独的标签页显示。停靠于 Visual Studio 左侧或右侧的“解决方案资源管理器”用于查看和管理你的应用程序的文件。在本书的例子中，通常会把每个程序的代码文件放在“源文件”文件夹中。如果解决方案资源管理器没有显示，可以选择“视图”|“解决方案资源管理器”来显示它。

步骤 4：将 GuessNumber.cpp 文件添加到项目

接着，将 GuessNumber.cpp 添加到步骤 3 创建的项目中。在解决方案资源管理器中执行以下操作：

1. 右击“源文件”文件夹，选择“添加”|“现有项”。
2. 在出现的对话框中，切换到本书示例文件夹中的 ch01 子文件夹，选择 GuessNumber.cpp。单击“添加”[①]。

步骤 5：配置项目以使用 C++20

Visual Studio 中的 Visual C++编译器支持多个版本的 C++标准。本书使用 C++20，必须在项目设置中进行配置。

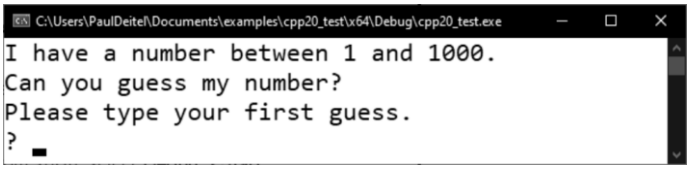
1. 在解决方案资源管理器中右击加粗显示的项目名称，选择“属性”来打开 cpp20_test 项目的属性页。
2. 从“配置”下拉列表中选择“所有配置”，再从“平台”下拉列表中选择“所有平台”。
3. 在左栏中展开“C/C++”节点，选择“语言”。
4. 在右栏中单击“C++语言标准”右侧的字段。单击下箭头，选择“ISO C++20 标准（/std:c++20）”，单击“确定”。

步骤 6：编译和运行项目

为了编译并运行项目，以便试运行该应用程序，请选择“调试”|“开始执行(不调试)”，或者直接按 Ctrl+F5。如果程序正确编译，Visual Studio 会打开一个“命令提示符”窗口并

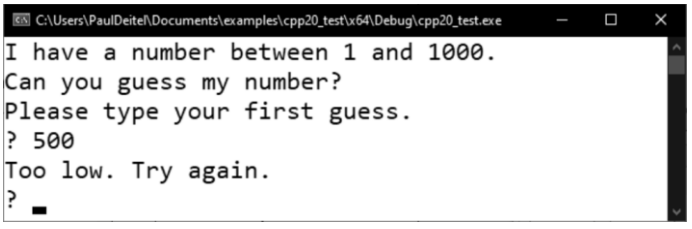
[①] 对于包含多个源代码文件的程序（本书以后会遇到），请选择程序的所有文件。开始自己创建程序时，可以右击“源文件”文件夹，选择“添加”|“新建项”来显示一个“添加新项”对话框。

执行该程序。为方便阅读，我们修改了“命令提示符”窗口的颜色方案和字号。



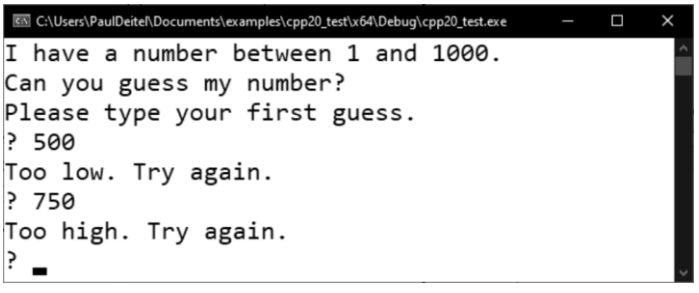
步骤 7：输入第一次猜测

在?提示符下输入 **500** 并按回车键。每次运行程序的输出结果都有所不同。在我们的例子中，应用程序显示“Too low. Try again.”（太低了，再试一次），这表明猜的数字小于应用程序选择的目标数字。



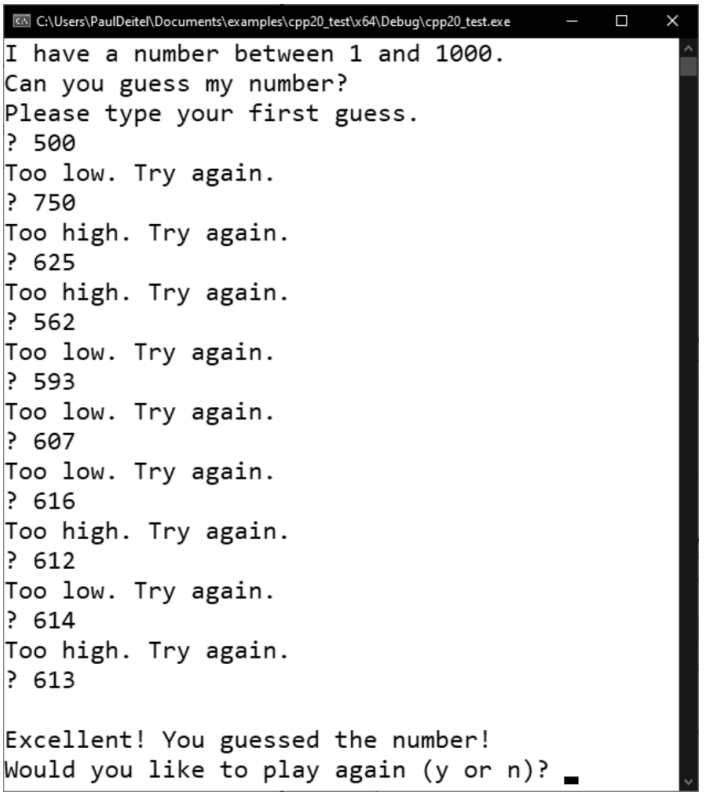
步骤 8：输入另一个猜测

在下一个提示中，如果系统说第一次猜得太低，请输入 **750** 并按回车键；否则，输入 **250** 并按回车键。我们的例子是输入 **750**。然后，应用程序显示“Too high. Try again.”（太高了，再试一次。”这表明猜的数字比目标数字大。



步骤 9：输入其他猜测

继续通过输入数字玩这个游戏，直到猜出正确数字。如果猜对，应用程序会显示“Excellent! You guessed the number!”（太好了! 你猜对了!）。



步骤 10：再玩一次游戏或退出应用程序

猜中正确的数字后，应用程序会问是否想再玩一次。看到“Would you like to play again (y or n)?”（想再玩一次吗(y 或 n)）？”时，输入 **y** 会使应用程序选择一个新的目标数，并开始一次新的游戏。输入 **n** 则终止应用程序。

在后续的例子中重用该项目

可以按照本节的步骤为本书的每个应用程序都创建一个单独的项目。然而，就我们的例子而言，从项目中删除当前程序，再添加一个新程序会更方便。要从项目（但不是你的系统）中删除一个文件，请在解决方案资源管理器中选定它，按 Del（或 Delete）键即可。然后，重复步骤 4 向项目中添加一个不同的程序。

在 Linux 的 Windows 子系统中使用 Ubuntu Linux

一些 Windows 用户可能想在 Windows 上使用 GNU gcc 编译器。可以使用 GNU 编译器集合 Docker 容器（1.2.4 节），也可以通过 WSL 来运行 Ubuntu Linux，并在其中使用 gcc。要安装 WSL，请按以下网址的指示进行操作：

https://docs.microsoft.com/en-us/windows/wsl/install

在 Windows 上安装并启动 Ubuntu 应用后，使用以下命令切换到包含代码示例的文件夹：

```
cd /mnt/c/Users/你的用户名/Documents/examples/ch01
```

然后，从 1.2.3 节的步骤 2 继续。

1.2.2 在 macOS 上用 Xcode 编译和运行 C++20 应用程序

本节在 Apple Xcode IDE 中使用 Apple 版本的 Clang 编译器在 macOS 上运行 C++程序。

步骤 1：检查安装

如果尚未安装，请参考之前的“准备工作”了解如何安装 IDE 并下载本书代码示例。

步骤 2：启动 Xcode

打开“访达”（Finder）窗口，选择“应用程序”并双击 Xcode 图标：

如果是首次运行 Xcode，会出现 Welcome to Xcode 窗口。关闭这个窗口（以后可以选择“Window”|“Welcome to Xcode”重新打开它。注意，本书在涉及菜单操作的时候，会用“|”来表示从一个菜单中选择菜单项。所以，“文件”|“打开”意味着从“文件”菜单中选择“打开”。

步骤 3：创建项目

项目（project）是指一组相关的文件，例如构成一个应用程序的 C++源代码文件。本书创建的 Xcode 项目是“Command Line Tool”项目，将直接在 IDE 中执行。请按以下步骤创建一个项目：

1. 选择“File”|“New”|“Project”。
2. 在“Choose a template for your new project”对话框顶部单击“macOS”。
3. 在“Application”下单击“Command Line Tool”并单击“Next”。
4. 在“Product Name”中输入项目名称，这里输入的是 **cpp20_test**。
5. 从“Language”下拉列表中选择“C++”，单击“Next”。
6. 指定项目的存储位置。这里选择包含本书代码示例的 examples 文件夹。
7. 单击“Create”。

Xcode会创建项目，并显示工作区（Workspace）窗口，其中最开始有三个区域：Navigator区域(左)，Editor区域(中)和Utilities区域(右)。

左侧的Navigator区域顶部显示了一些导航器图标，本书主要使用其中两个：

- Project（）——显示项目中的所有文件和文件夹。
- Issue（）——显示由编译器生成的警告和错误。单击一个导航器按钮会显示相应的导航器面板。

中间的 Editor 区域用于管理项目设置和编辑源代码。这个区域在你的工作区窗口中始终显示。在项目导航器中选择一个文件时，该文件的内容会出现在Editor区域。右侧的Utilities区域通常显示检查器。例如，如果构建的是包含可触摸按钮的iPhone应用，就可以在这个区域配置按钮的属性（标签、大小、位置等）。本书用不到 Utilities 区域。另外还有一个Debug 区域，将在这里与运行中的猜数程序进行交互。它会出现在 Editor 区域下方。利用工作区窗口的工具栏，可以执行程序、显示 Xcode 中执行的任务的进度以及隐藏或显示左侧（Navigator）和右侧（Utilities）的区域。

步骤 4：配置项目以使用 C++20

Xcode中的Apple Clang编译器支持多个版本的C++标准。本书使用C++20，必须在项目设置中进行配置。

1. 在Project导航器中，选择你的项目名称（cpp20_test）。
2. 在Editor区域左侧，在TARGETS下选择你的项目名称。
3. 在Editor区域的顶部，单击“Build Settings”（生成设置），单击它下方的“All”。
4. 滚动到“Apple Clang – Language - C++”区域。
5. 单击“C++ Language Dialect”右边的值，选择“GNU++20 [-std=gnu++20]”。
6. 单击“C++ Standard Library”右边的值，选择“Compiler Default”（编译器默认）。

步骤 5：从项目中删除 main.cpp 文件

Xcode 默认创建了一个 main.cpp 源代码文件，其中包含一个能显示“Hello, World!”的简单程序。这次试运行不会用到 main.cpp，所以应删除该文件。在 Project 导航器中，右击main.cpp文件，选择“Delete”。在随后出现的对话框中，选择“Move to Trash”。除非清空废纸篓，否则该文件不会从你的系统中删除。

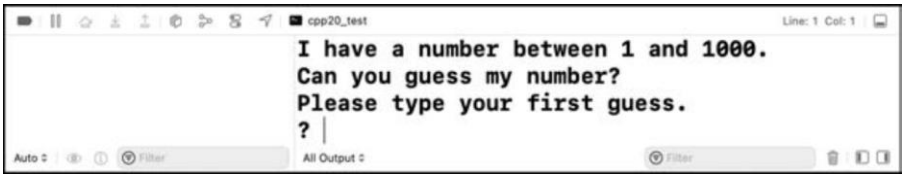
步骤 6：将 GuessNumber.cpp 文件添加到项目

在Finder窗口中打开本书的examples文件夹，将GuessNumber.cpp拖放到Project导航器的cpp20_test 文件夹。在随后出现的对话框中，确定已勾选了“Copy items if needed”，单击

“Finish”[1]。

步骤 7：编译和运行项目

为了编译和运行项目，以便测试该应用程序，请直接单击 Xcode 工具栏上的运行按钮（▶）。如果程序正确编译，Xcode 会打开 Debug 区域，并在该区域的右半边执行程序。程序将显示“Please type your first guess.”，并显示一个问号（?）来提示输入。



步骤 8：输入第一次猜测

单击 Debug 区域，输入 **500** 并按回车键。每次运行程序的输出结果都有所不同。在我们的例子中，应用程序显示“Too high. Try again.”（太高了，再试一次），这表明猜的数字大于应用程序选择的目标数字。

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

步骤 9：输入另一个猜测

在下一个提示中，如果系统说第一次猜得太低，请输入 **750** 并按回车键；否则，输入 **250** 并按回车键。我们的例子是输入 **250**。然后，应用程序显示“Too high. Try again.”（太高了，再试一次。”这表明猜的数字还是比目标数字大。

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
?
```

[1] 对于包含多个源代码文件的程序（本书以后会遇到），请将程序的所有文件拖放到项目的文件夹。开始自己创建程序时，可以右击项目的文件夹，选择“New File…”来显示一个用于添加新文件的对话框。

步骤 10：输入其他猜测

继续通过输入数字玩这个游戏，直到猜出正确数字。如果猜对，应用程序会显示“Excellent! You guessed the number!”（太好了! 你猜对了!）。

步骤 11：再玩一次游戏或退出应用程序

猜中正确的数字后，应用程序会问是否想再玩一次。看到“Would you like to play again (y or n)?”（想再玩一次吗(y 或 n)）？”时，输入 **y** 会使应用程序选择一个新的目标数，并开始一次新的游戏。输入 **n** 则终止应用程序。

在后续的例子中重用该项目

可以按照本节的步骤为本书的每个应用程序都创建一个单独的项目。然而，就我们的例子而言，从项目中删除当前程序，再添加一个新程序会更方便。要从项目（但不是你的系统）中删除一个文件，请在 Project 导航器中右击文件并选择“Delete”。然后，重复步骤 6 向项目中添加一个不同的程序。

1.2.3 在 Linux 上用 GNU C++运行 C++20 应用程序

本节在 Linux 上使用 GNU C++编译器（g++）运行 C++程序[①]。对于这个例子，我们假设你已经阅读了前言之后的“准备工作”部分，并已将本书的示例代码放到了自己用户帐户下的 Documents/examples 文件夹。

步骤 1：切换到 ch01 文件夹

在一个 Linux 外壳中，使用 `cd` 命令切换到本书 examples 文件夹中的 ch01 子文件夹。

```
~$ cd ~/Documents/examples/ch01
~/Documents/examples/ch01$
```

注意，本书是用**加粗**的内容来表示用户输入。另外，在 Ubuntu Linux 外壳中，提示符用波浪号(~)表示 home 目录（即用户根目录）。每个提示符都以美元符号($)结尾。在你的 Linux 系统上，提示符可能有所区别。

步骤 2：编译应用程序

运行应用程序之前，必须先用以下 g++命令编译它[②]：

```
~/Documents/examples/ch01$ g++ -std=c++20 GuessNumber.cpp -o GuessNumber
~/Documents/examples/ch01$
```

注意，`-std=c++20` 选项表明我们要使用 C++20。`-o` 选项指定用于运行程序的可执行文件的名称（`GuessNumber`）。如果不指定该选项，g++默认将可执行文件命名为 `a.out`。

步骤 3：运行应用程序

在命令提示符后输入`./GuessNumber`，按回车键来运行程序。注意，`GuessNumber` 前的`./`告

[①] 本书写作时所用的 g++版本是 11.2。可以使用 `g++ --version` 命令来确定系统上安装的 g++版本。如果使用的是较旧的 g++版本，可在网上搜索为你的 Linux 发行版升级 GCC 的说明，或者考虑使用 1.2.4 节介绍的 GCC Docker 容器。

[②] 如果安装了多个 g++版本，可能需要使用 `g++-##`，其中##是 g++的版本号。例如，在你的计算机上运行最新版本的 g++ 11.x 可能需要使用 `g++-11` 命令。

诉 Linux 从当前目录运行 GuessNumber 程序。

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

步骤 4：输入第一次猜测

应用程序显示“Please type your first guess.”，并显示一个问号（?）来提示输入。我们输入 **500**，但程序的输出在每次运行时都会有所区别。

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

在本例中，应用程序显示“Too high. Try again.”（太高了，再试一次），这表明猜的数字大于应用程序选择的目标数字。

步骤 5：输入另一个猜测

在下一个提示中，如果系统说第一次猜得太低，请输入 **750** 并按回车键；否则，输入 **250** 并按回车键。我们的例子是输入 **250**。然后，应用程序显示“Too high. Try again.”（太高了，再试一次。”这表明猜的数字还是比目标数字大。

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
?
```

步骤 6：输入其他猜测

继续通过输入数字玩这个游戏，直到猜出正确数字。如果猜对，应用程序会显示

“Excellent! You guessed the number!”（太好了！你猜对了！）。

```
? 125
Too high. Try again.
? 62
Too low. Try again.
? 93
Too low. Try again.
? 109
Too high. Try again.
? 101
Too low. Try again.
? 105
Too high. Try again.
? 103
Too high. Try again.
? 102

Excellent! You guessed the number!
Would you like to play again (y or n)?
```

步骤 7：再玩一次游戏或退出应用程序

猜中正确的数字后，应用程序会问是否想再玩一次。看到“Would you like to play again (y or n)?”（想再玩一次吗(y 或 n)）？”时，输入 **y** 会使应用程序选择一个新的目标数，并开始一次新的游戏。输入 **n** 则终止应用程序。

1.2.4 在 GCC Docker 容器中编译和运行 C++20 应用程序

无论什么操作系统都可以使用最新的 GNU C++编译器。最方便的跨平台方法之一是使用 GNU 编译器集合（GCC）的 Docker 容器。本节假设你已经按照“准备工作”的指示安装好了 Docker Desktop（Windows 和 macOS）或者 Docker Engine（Linux）。

执行 GNU 编译器集合（GCC）Docker 容器

打一个“命令提示符”窗口（Windows）、终端（macOS/Linux）或外壳程序（Linux），然后执行以下步骤来启动 GCC Docker 容器。

1. 使用 `cd` 命令切换到包含本书代码示例的 examples 文件夹。
2. Windows 用户执行以下命令来启动 GCC Docker 容器[①]：

[①] 可能出现一个通知，要求你允许 Docker 访问当前文件夹中的文件。必须允许，否则无法在 Docker 中访问本书的源代码文件。

```
docker run --rm -it -v "%CD%":/usr/src gcc:latest
```

3. macOS/Linux 用户执行以下命令来启动 GCC Docker 容器：

```
docker run --rm -it -v "$(pwd)":/usr/src gcc:latest
```

在上述命令中：

- `--rm` 在你最后关闭容器时清理容器的资源。
- `-it` 以交互模式运行容器，这样就可以输入命令来更改文件夹，并使用 GNU C++编译器来编译和运行程序。
- `-v "%CD%":/usr/src`（Windows）或 `-v "$(pwd)":/usr/src`（macOS/Linux）允许 Docker 容器访问你从中执行 `docker run` 命令的那个文件夹中的文件。在 Docker 容器中，你将用 `cd` 命令切换到/usr/src 的子文件夹以编译和运行本书的示例。换言之，你的本地系统文件夹将被映射到 Docker 容器中的/usr/src 文件夹。
- `gcc:latest` 是容器的名称。`:latest` 指出你要使用最新版本的 gcc 容器[①]。

容器运行后，会出现像下面这样的一个提示符：

```
root@67773f59d9ea:/#
```

容器使用一个 Linux 操作系统。在它的提示符中，会在:和#符号之间显示当前文件夹位置。

在 Docker 容器中切换到 ch01 文件夹

之前执行的 `docker run` 命令将你的 examples 文件夹连接到容器的/usr/src 文件夹。在 Docker 容器中，使用 `cd` 命令来切换到/usr/src 下的 ch01 子文件夹。

```
root@01b4d47cadc6:/# cd /usr/src/ch01
root@01b4d47cadc6:/usr/src/ch01#
```

在 Docker 容器中编译、运行 GuessNumber 应用程序并与之交互的过程请遵循 1.2.3 节的 GNU C++的步骤 2~步骤 7。

终止 Docker 容器

可以在容器的提示符下按 Ctrl+d 来终止 Docker 容器。

---

① 如果希望 GCC 容器保持最新的版本，可以在运行容器之前执行命令 `docker pull gcc:update`。

1.2.5 在 Docker 容器中使用 clang++来编译和运行 C++20 应用程序

和 g++一样，无论什么操作系统都可以使用最新的 LLVM/Clang C++（clang++）命令行编译器。目前，LLVM/Clang 团队没有一个官方的 Docker 容器，但在 https://hub.docker.com 上有许多好用的容器可供下载。本节假设你已经按照“准备工作”的指示安装好了 Docker Desktop（Windows 和 macOS）或者 Docker Engine（Linux）。

我们选择的是带有 clang++ 13 的一个最新的、下载量很大的容器，可执行以下命令获取[①]：

```
docker pull teeks99/clang-ubuntu:13
```

执行 teeks99/clang-ubuntu Docker 容器

打一个“命令提示符”窗口（Windows）、终端（macOS/Linux）或外壳程序（Linux），然后执行以下步骤来启动 teeks99/clang-ubuntu Docker 容器。

1. 使用 `cd` 命令切换到包含本书代码示例的 examples 文件夹。
2. Windows 用户执行以下命令来启动 GCC Docker 容器[②]：

```
docker run --rm -it -v "%CD%":/usr/src teeks99/clang-ubuntu:13
```

3. macOS/Linux 用户执行以下命令来启动 GCC Docker 容器：

```
docker run --rm -it -v "$(pwd)":/usr/src teeks99/clang-ubuntu:13
```

在上述命令中：

- `--rm` 在你最后关闭容器时清理容器的资源。
- `-it` 以交互模式运行容器，这样就可以输入命令来更改文件夹，并使用 GNU C++编译器来编译和运行程序。
- `-v "%CD%":/usr/src`（Windows）或 `-v "$(pwd)":/usr/src`（macOS/Linux）允许 Docker 容器访问你从中执行 `docker run` 命令的那个文件夹中的文件。在 Docker 容器中，你将用 `cd` 命令切换到/usr/src 的子文件夹以编译和运行本书的示例。换言之，你

[①] Xcode 中使用的 Clang C++编译器的版本并不是最新版本，所以不像 LLVM/Clang 团队的直供版本那样实现了许多 C++20 特性。另外，本书写作时，在 docker pull 命令中将 13 换成 latest，会得到一个带有 clang++ 12 而不是 13 的 Docker 容器。

[②] 可能出现一个通知，要求你允许 Docker 访问当前文件夹中的文件。必须允许，否则无法在 Docker 中访问本书的源代码文件。

的本地系统文件夹将被映射到 Docker 容器中的/usr/src 文件夹。

- `teeks99/clang-ubuntu:13` 是容器的名称。

容器运行后，会出现下面这样的一个提示符：

```
root@9753bace2e87:/#
```

容器使用一个 Linux 操作系统。在它的提示符中，会在:和#符号之间显示当前文件夹位置。

在 Docker 容器中切换到 ch01 文件夹

之前执行的 `docker run` 命令将你的 examples 文件夹连接到容器的/usr/src 文件夹。在 Docker 容器中，使用 `cd` 命令来切换到/usr/src 下的 ch01 子文件夹。

```
root@9753bace2e87:/# cd /usr/src/ch01
root@9753bace2e87:/usr/src/ch01#
```

编译应用程序

应用程序运行前必须编译。该容器使用 `clang++-13` 命令来开始编译，如下所示：

```
clang++-13 -std=c++20 GuessNumber.cpp -o GuessNumber
```

其中：

- `-std=c++20` 选项表明我们要使用 C++20。
- `-o` 选项指定用于运行程序的可执行文件的名称（`GuessNumber`）。如果不指定该选项，g++默认将可执行文件命名为 `a.out`。

运行应用程序

在 Docker 容器中运行 GuessNumber 应用程序并与之交互的过程请遵循 1.2.3 节的 GNU C++的步骤 3~步骤 7。

终止 Docker 容器

可以在容器的提示符下按 Ctrl+d 来终止 Docker 容器。

1.3 摩尔定律、多核处理器和并发编程

今天的许多个人电脑可以在一秒钟内做数十亿次计算——比人类一辈子所能做的还要多。超级计算机每秒已能执行数千万亿条指令。日本的富岳（Fugaku）超级计算机每秒能进行超过 442 千万亿次计算（442.01 petaflops）[①]。如果这样算，富岳超级计算机在一秒钟内可以为地球上的每个人进行大约 4000 万次计算！与此同时，超级计算的“上限”正在迅速增长。

性能

摩尔定律

由于通胀的存在，我们每年花在大多数产品和服务上的钱恐怕都是增加的。但计算机和通信领域的情况刚好相反，特别是硬件。多年来，硬件成本呈现出不升反降的趋势。

性能

几十年来，计算机处理能力大约每两年就会以低廉的成本翻上一番。这一明显的趋势通常被称为摩尔定律，以英特尔公司的联合创始人高登·摩尔命名，他在 20 世纪 60 年代发现了这一趋势。英特尔是当今计算机和嵌入式系统（智能家用电器、家庭安全系统、机器人、智能交通路口等等）所用的处理器的领先制造商。摩尔定律及其相关的观察尤其适用于：

- 为程序和数据准备的内存容量。
- 用来存储程序和数据的辅助存储（二级存储）。
- 处理器速度，即计算机执行程序以完成其工作的速度。

性能

不过，计算机处理器公司 NVIDIA 和 Arm 的核心管理层表示，摩尔定律或许已经不再适用[②③]。计算机处理能力虽然还是会增加，但现在主要依赖于新的处理器设计，尤其是多核处理器。

多核处理器和性能

今天，大多数计算机使用的都是多核处理器，也就是在一个集成电路芯片上经济地实现多个处理器。例如，双核处理器有两个 CPU，四核处理器有四个，八核处理器则有八个。我们的主要测试电脑使用的就是八核英特尔处理器。苹果最新的 M1 Pro 和 M1 Max 处理器有

[①] https://zh.wikipedia.org/wiki/TOP500#TOP_500

[②] Esther Shein，“Moore’s Law Turns 55: Is It Still Relevant?”，https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant。

[③] Nick Heath，“Moore’s Law Is Dead: Three Predictions About the Computers of Tomorrow”，https://www.techrepublic.com/article/moores-law-is-dead-three-predictions-about-the-computers-of-tomorrow/。

10 个 CPU 核心。此外，顶级的 M1 Pro 有一个 16 核的 GPU，而顶级的 M1 Max 处理器有一个 32 核的 GPU，两者都有一个 16 核的“神经引擎”用于机器学习[①②]。英特尔的一些处理器有多达 72 个核心，并正在开发多达 80 个核心的处理器[③]。AMD 在开发 192 和 256 核的 CPU[④]。核心数量还会继续呈增长态势。

在多核系统中，不同处理器可以处理任务的不同部分，从而使任务能更快地完成。为了充分利用多核架构的优势，你需要编写多线程应用程序。如果一个程序将任务分割成多个独立的线程，那么只要有足够数量的内核，多核系统就能并行运行这些线程。

随着多核系统的普及，人们对多线程的兴趣日益增大。标准 C++多线程是 C++11 引入的最重要的更新之一。 随后的每一个 C++标准都增加了更高级的功能来简化多线程应用程序开发。第 17 章讨论了如何创建和管理多线程 C++应用程序。第 18 章介绍了 C++20“协程”，它能以简单的、类似顺序编码的风格来实现并发编程。

1.4 面向对象简单回顾

虽然对新的和更强大的软件的需求激增，但是快速、正确和经济地构建软件仍然是一个难以达成的目标。**对象**（object），或者更准确地说，对象所来自的**类**（class）本质上是可重用的软件组件（参见第 9 章）。有日期对象、时间对象、音频对象、视频对象、汽车对象、人物对象等等。几乎任何名词都可以通过属性（例如名称、颜色和大小）和行为（例如计算、移动和通信）来合理地表示为一个软件对象。软件开发人员发现，如果使用模块化的、面向对象的设计和实现方法，那么相较于使用早期的技术，软件开发团队的效率会更高。这是由于面向对象的程序通常更容易理解、纠正和修改。

汽车作为对象

让我们从一个简单的类比开始。我们都知道，开车时踩油门踏板可以加速。但是，为了做

[①] Juli Clover，“Apple’s M1 Pro Chip: Everything You Need to Know”，November 3, 2021，https://www.macrumors.com/guide/m1-pro/。

[②] https://zh.wikipedia.org/wiki/Apple_M1

[③] Anton Shilov，“Intel's Sapphire Rapids Could Have 72–80 Cores, According to New Die Shots”，April 30, 2021，https://www.tomshardware.com/news/intel-sapphire-rapids-could-feature-80-cores。

[④] Joel Hruska，“Future 256-Core AMD Epyc CPU Might Sport Remarkably Low 600W TDP”，November 1, 2021，https://www.extremetech.com/computing/328692-future-256-core-amd-epyc-cpu-might-sport-remarkably-low-600w-tdp。

到这一点，之前必须发生什么？好吧，在汽车能开起来之前，必须先有人设计它。这通常是从工程图纸开始的，类似于描述房屋设计的蓝图。图纸中包括一个油门踏板的设计。踏板向司机隐藏了使汽车加速的复杂机制，就像刹车踏板隐藏了使汽车减速的机制，而方向盘隐藏了使汽车转向的机制。这样一来，那些对发动机、制动和转向机构如何工作知之甚少或一无所知的人也能轻松驾驶汽车。

汽车要想开起来，必须先根据它的工程图纸来完成建造。一辆完工的汽车有一个实际的加速踏板，它能使汽车跑得更快，但即使这样还不够：汽车不会自己加速（幸好如此！），所以司机必须自己踩下踏板来加速。

函数、成员函数和类

现在用汽车的例子来介绍一些关键的面向对象编程概念。在程序中执行一项任务需要一个函数。函数中包含执行任务所需的程序语句。它向用户隐藏了这些语句，这类似于汽车的油门踏板向司机隐藏了使汽车加速的机制。在 C++中，我们经常创建一个称为类的程序单元来容纳执行该类的任务所需的一组函数，这种函数称为类的**成员函数**（member functions）。例如，代表银行账户的类可能用一个成员函数向账户存钱，用另一个从账户取钱，用第三个查询账户当前余额。类就像是汽车的工程图纸，它包含了油门踏板、刹车踏板、方向盘等的设计。

实例化

就像汽车在能够开起来之前，必须有人先根据工程图纸把汽车建造出来一样，你必须在程序执行类的成员函数所定义的任务之前，根据类来创建一个对象。这个过程称为**实例化**（instantiation），而对象称为其类的一个**实例**（instance）。

重用

软件工程

就像汽车的工程图纸可以重复使用很多次来建造大量汽车一样，你也可以多次重用一个类来创建许多对象。类的重用可以节省大量时间和精力。重用还有助于构建更可靠、更高效的系统，因为现有的类和组件通常已经过广泛的测试、调试和性能优化。正如可互换零件的概念对于工业革命至关重要一样，可重用类对于由对象技术推动的软件革命也至关重要。

消息和成员函数调用

驾驶汽车时，是踩下油门踏板向汽车发送一条消息，让它执行一项任务，即“开快点”。我们以类似的方式向对象发送消息。每条消息都被实现为一个成员函数调用，指示对象的一个成员函数执行其任务。例如，程序可调用特定银行账户对象的存款（`deposit`）成员函数以增加余额。

属性和数据成员

除了具有完成任务的能力，汽车还具有属性，例如它的颜色、车门数量、油箱油量、当前速度和总行驶里程的记录（即里程表读数）。和它的能力一样，汽车的属性在其工程图纸中被表示为其设计的一部分（例如，要求它配备一个里程表和一个燃油表）。驾驶汽车时，这些属性会跟着汽车走。换言之，每辆车都维护着自己的属性。例如，每辆车都知道自己的油箱里还有多少油，但不知道其他车还有多少油。

同样，对象亦有属性，在程序中使用时，属性会跟着对象一起走。这些属性被设计为类的一部分。例如，银行账户对象可能有一个余额（`balance`）属性。每个银行账户对象知道它所对应的账户的余额，但不知道银行中其他账户的余额。属性是类的一种**数据成员**（data member）。

封装

类将属性和成员函数**封装**（encapsulate）到由这些类创建的对象中。一个对象的属性和成员函数是密切相关的。对象之间可以交流，但它们通常不允许知道其他对象的实现细节。这些细节被隐藏在对象本身中。正如我们将看到的，这种信息隐藏对于良好的软件工程至关重要。

继承

通过**继承**（inheritance），可以快速、方便地创建新类。新类接收了现有类的特征，并可对其进行自定义，从而增加自己的独特特征。在我们的汽车类比中，“敞篷车”显然也是一种“汽车”，但它的车顶可以伸缩。

面向对象分析与设计

你很快就会用 C++写程序。那么，如何为程序创建代码？也许，像许多程序员一样，你会直接启动电脑，然后直接开始敲代码。这种方法可能适合小程序（就像本书早期章节介绍的那些）。但是，如果要求你创建一个软件系统来控制一家大银行的数千台自动取款机呢？或者，如果要求你在一个由数千名软件开发人员组成的团队中工作，建立下一代空中交通管制系统呢？对于如此庞大和复杂的项目，肯定不是直接坐下来开始敲代码那么简单。

为了创建最好的解决方案，应遵循一个详细的分析过程来确定项目的需求（即定义系统应该做什么），并开发一个满足这些需求的设计（即决定系统应该如何做）。理想情况下，在写任何代码之前，都应经历这个过程并仔细审查设计（同时让其他软件专业人士审查这个设计）。如果这个过程涉及到从面向对象的角度分析和设计系统，就称其为**面向对象分析与设计**（Object-Oriented Analysis and Design，OOAD）过程。像 C++这样的语言就是面向

对象的。用这样的语言编程称为**面向对象编程**（Object-Oriented Programming，OOP）。

1.5 小结

本章算是入门性质，我们讨论了如何使用三种首选的编译器（Windows 上的 Visual Studio 2022 社区版中的 Visual C++，macOS 上的 Xcode 中的 Clang 以及 Linux 上的 GNU g++）来编译和运行应用程序。我们列出了一个 Microsoft 的资源，它允许你通过 WSL 安装 Ubuntu Linux，这样就可以在 Windows 上运行 g++。还展示了如何启动跨平台的 Docker 容器，这样就可以在 Windows、macOS 或 Linux 上使用最新的 g++和 clang++版本。我们给你指出了几个学习 C++历史和设计的资源，包括由 C++的创造者 Bjarne Stroustrup 提供的资源。接着，我们讨论了摩尔定律、多核处理器以及为什么现代 C++的并发编程功能对于利用多核处理器的威能至关重要。最后，我们简要回顾了面向对象编程的概念和术语，它们将在本书以后广泛地使用。

下一章将讲解 C++编程的基础知识，包括基本输入和输出语句、基本数据类型、算术运算、决策和我们的第一个对象自然案例研究，它使用了 C++标准库提供的 `string` 类。

第 2 章 C++编程入门

学习目标：

- 写简单的 C++应用程序。
- 使用输入和输出语句。
- 使用基本数据类型。
- 使用算术操作符。
- 理解算术操作符的优先级。
- 编写决策语句。
- 使用关系和相等性操作符。
- 在开始创建自定义类之前，通过创建和使用 C++标准库的 `string` 类的对象，理解“对象自然”学习方法。

2.1 导读

本章展示了几个代码示例，解释程序如何显示消息并从用户那里获取数据以进行处理。前三个例子在屏幕上显示消息。下一个例子要求用户从键盘输入两个数字，求和并显示结果。在此期间，我们讨论了 C++的算术操作符。第 5 个例子演示了决策语句，教你如何比较两个数字，然后根据结果显示信息。

“对象自然”学习方法

有许多现成的类可供利用，它们已进行了精心的开发和测试。我们在开发程序时，可以创建和使用这些类的对象，从而以最少的代码执行重要的任务。这些类通常来自：

- C++标准库。
- 平台特有的库（如 Microsoft 提供的用于创建 Windows 应用程序的库，或者 Apple 提供的用于创建 macOS 应用程序的库）
- 免费的第三方库，通常由围绕所有主流编程语言发展起来的大型开源社区创建。

为了在本书的早期体验这种编程风格，在学会创建自定义类之前，你将先学会创建和使用现有的、来自 C++标准库的类的对象。我们称这种方法为“对象自然”。在本章最后的例子中，你将创建和使用 `string` 类的对象。以后学会创建自定义类后，就知道 C++允许程序员自行修改有价值的类，以便自己使用，或者供其他程序员重用。

编译和运行程序

关于在 Microsoft Visual Studio、Apple Xcode 和 GNU C++中编译和运行程序的说明，请参见第 1 章的说明或者我们在以下网址提供的视频解说：

http://deitel.com/c-plus-plus-20-for-programmers

2.2 第一个 C++程序：显示单行文本

图 2.1 是一个显示单行文本的简单程序。注意，行号不是代码的一部分。

```
1  // fig02_01.cpp
2  // Text-printing program.
3  #include <iostream> // enables program to output data to the screen
4
5  // function main begins program execution
6  int main() {
7     std::cout << "Welcome to C++!\n"; // display message
8
9     return 0; // indicate that program ended successfully
10 } // end function main
```

```
Welcome to C++!
```

图 2.1 文本打印程序

注释

第 1 行和第 2 行：

```
// fig02_01.cpp
// Text-printing program.
```

各自都以//开头，表明本行剩余的所有内容都是**注释**（comment）。本书每个示例程序的第一行注释都包含程序文件名。注释"Text-printing program."描述了该程序的作用。以//开头的注释称为**单行注释**，因其在当前行末就结束了。将文本封闭到一对/*和*/之间，即可创建单行或**多行注释**，例如：

```
/* fig02_01.cpp: Text-printing program. */
```

或者：

```
/* fig02_01.cpp
Text-printing program. */
```

#include 预处理指令

第 3 行：

```
#include <iostream> // enables program to output data to the screen
```

是一条**预处理指令**（preprocessing directive），它是向 C++预处理器发送的一条消息，编译器在对程序进行编译之前会先调用这种指令。上面这一行的作用是通知预处理器在程序中包含输入/输出流的头文件`<iostream>`的内容。任何程序只要使用 C++流输入/输出功能向屏幕输出或者从键盘输入，编译器在编译它时就需要来自这个头文件的信息。图 2.1 的程序将数据输出到屏幕。第 5 章会更详细地讨论头文件，本书放在网上的第 19 章更详细地解释了`<iostream>`的内容。

空行和空白

第 4 行就是一个空行。写代码时，我们使用空行、空格和制表符使程序更容易阅读。所有这些字符一般统称为**空白**（whitespace）——会直接被编译忽略。

main 函数

第 6 行：

```
int main() {
```

是每个 C++程序都要有的一部分。`main` 后的圆括号表明它是一个**函数**（function）。C++程序通常由一个或多个函数和类组成。每个程序必须有一个且只能有一个 `main` 函数，这是 C++程序开始执行代码的地方，称为程序的**入口点**（entry point）。关键字 `int` 表示在 `main` 执行完毕后，它会“返回”一个整数值。**关键字**（keywords）是 C++为特定用途而保留的。我们会在第 3 章展示 C++关键字的完整列表。第 5 章讲述如何创建你自己的函数时，会解释函数“返回一个值”是什么意思。现在，只需在你的每个程序的 `main` 左边加上关键字 `int` 即可。

第 6 行末尾的左大括号{用于开始函数的**主体**（body）。函数主体包含了函数要执行的指令。配对的右大括号}（第 10 行）用于结束函数主体。

输出语句

第 7 行：

```
std::cout << "Welcome to C++!\n"; // display message
```

显示包含在双引号之间的字符。引号及其之间的字符统称为一个**字符串**（string）。平时，

我们也会直接将双引号之间的字符称为字符串或者**字符串字面值**（string literal）。注意，字符串中的空白字符（空格、制表符等）不会被编译器忽略。

整个第 7 行——包括 `std::cout`、`<<`操作符、字符串`"Welcome to C++!/n"`和分号（`;`）——统称为一个**语句**（statement）。大多数 C++语句以分号结尾。如果需要分号而在 C++语句结尾处省略分号，就属于一种语法错误。预处理指令（例如`#include`）不是 C++语句，不以分号结尾。

⊗错误

通常，C++中的输出和输入通过数据**流**（stream）来进行。上述语句执行时，会将字符流 `Welcome to C++!\n` 发送给标准输出流对象 `std::cout`，而它通常“连接”的是屏幕。

缩进

在用于主体定界的大括号内，整个函数主体最好缩进一级。这使程序的功能结构更醒目，使程序更容易阅读。为你喜欢的缩进大小设定惯例，然后统一应用。是可以按 Tab 键来缩进，但制表位的距离在不同系统上可能有所区别。我们喜欢每级缩进三个空格。

`std` 命令空间

如果准备使用从标准库头文件（例如`<iostream>`）带入程序的名称，就需要在 `cout` 前添加 `std::`前缀。`std::cout` 指出我们要使用一个从属于 `std` 命名空间[①]的名称（本例是 `cout`）。我们将在第 17 章介绍命名空间。现在，只需记住在程序中每次引用 `cout`、`cin` 和 `cerr` 时都添加 `std::`前缀。这自然很繁琐，但我们很快就会讲到 `using` 声明和 `using` 指令，它们使你能在每次使用来自 `std` 命名空间的名字时省略 `std::`前缀。

流插入操作符和转义序列

`cout` 语句中的`<<`操作符称为**流插入操作符**（stream insertion operator）。这个程序执行时，操作符右边的值（右操作数）被插入输出流[②]。注意，箭头方向就是数据的去向。一般情况下，显示的内容会和字符串的内容一模一样。但是，在图 2.1 的示例输出中，并没有显示字符`\n`。 反斜杠（`\`）称为**转义字符**（escape character）。它表示要输出一个“特殊”字符。在字符串中遇到反斜杠时，下个字符和这个反斜杠就构成了一个**转义序列**（escape sequence）。转义序列`\n` 代表换行符，它使光标（即当前屏幕位置指示器）移动到屏幕下

[①] “`std::`”应该怎么读？不要读作单独的字母 s，t 和 d，而是直接读作“standard”。

[②] 译注：operator 在本书统一翻译为“操作符”，而不是“运算符”。相应地，operand 翻译为“操作数”，而不是“运算子”。

一行的开头位置。下表总结了一些常见的转义序列。

转义序列	说明
\n	换行符，将屏幕光标定位到下一行开头。
\t	水平制表符，将屏幕光标移到下一个制表位。
\r	回车符。将屏幕光标定位到当前行的开头；不前进到下一行。
\a	警报，发出系统响铃声。
\\	反斜杠，在字符串中包含一个反斜杠字符。
\'	单引号，在字符串中包含一个单引号字符。
\"	双引号，在字符串中包括一个双引号字符。

return 语句

第 9 行：

```
return 0; // indicate that program ended successfully
```

展示了我们用来退出函数的几种方法之一。在 main 结尾的这个 return 语句中，数值 0 标志着程序成功终止。如果程序执行到 main 的结束大括号时没有遇到 return 语句，C++会默认已经遇到 return 0;并假定程序成功终止。因此，在本书后续的程序中，我们会省略 main 最后的 return 语句，默认程序将成功终止。

2.3 修改第一个 C++程序

接着两个例子将修改图 2.1 的程序。第一个用多个语句显示单行文本。第二个用单个语句显示多行文本。

多个语句显示单行文本

图 2.2 的程序在多个语句中执行流插入操作（第 7 行~第 8 行），但产生的输出和图 2.1 一

样。每一次流插入操作都会在上一次停止的地方继续。第 7 行显示了 Welcome 和一个空格，由于这个字符串没有以\n 结尾，所以第 8 行在空格后于同一行开始显示。

```
1  // fig02_02.cpp
2  // Displaying a line of text with multiple statements.
3  #include <iostream> // enables program to output data to the screen
4
5  // function main begins program execution
6  int main() {
7     std::cout << "Welcome ";
8     std::cout << "to C++!\n";
9  } // end function main
```

```
Welcome to C++!
```

图 2.2 多个语句显示单行文本

单个语句显示多行文本

单个语句显示多行文本是通过额外的换行符实现的，如图 2.3 的第 7 行所示。每次在输出流中遇到\n(换行符)转义序列，屏幕光标就会指向下一行开头。要输出空行，像第 7 行那样连续使用两个换行符即可。

```
1  // fig02_03.cpp
2  // Displaying multiple lines of text with a single statement.
3  #include <iostream> // enables program to output data to the screen
4
5  // function main begins program execution
6  int main() {
7     std::cout << "Welcome\nto\n\nC++!\n";
8  } // end function main
```

```
Welcome
to

C++!
```

图 2.3 单个语句显示多行文本

2.4 另一个 C++程序：整数相加

下个程序获取用户从键盘输入的两个整数，求和，并用 std::cout 输出结果。图 2.4 展示了程序的代码以及示例输入和输出。注意，由用户输入的内容加粗显示。

```
1  // fig02_04.cpp
2  // Addition program that displays the sum of two integers.
3  #include <iostream> // enables program to perform input and output
4
5  // function main begins program execution
6  int main() {
7     // declaring and initializing variables
8     int number1{0}; // first integer to add (initialized to 0)
9     int number2{0}; // second integer to add (initialized to 0)
10    int sum{0}; // sum of number1 and number2 (initialized to 0)
11
12    std::cout << "Enter first integer: "; // prompt user for data
13    std::cin >> number1; // read first integer from user into number1
14
15    std::cout << "Enter second integer: "; // prompt user for data
16    std::cin >> number2; // read second integer from user into number2
17
18    sum = number1 + number2; // add the numbers; store result in sum
19
20    std::cout << "Sum is " << sum << "\n"; // display sum
21 } // end function main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

图 2.4 求两个整数之和

变量声明和大括号初始化

第 8 行~第 10 行：

```
int number1{0}; // first integer to add (initialized to 0)
int number2{0}; // second integer to add (initialized to 0)
int sum{0}; // sum of number1 and number2 (initialized to 0)
```

都是**声明**（declarations）。其中，`number1`，`number2` 和 `sum` 都是**变量**（variables）的名称，分别代表两个加数和两者之和。三个变量都是用于容纳整数值（例如 7，–11，0 和 31914）的 `int` 类型。注意，所有变量声明都必须包含一个数据类型和一个名称。

在第 8 行~第 10 行的每个变量声明中，变量名后都在一对大括号（`{}`）中包含了值 `0`。这
11 种语法称为**大括号初始化**（braced initialization），是 C++11 引入的一个特性。虽然并非一定要显式初始化每个变量，但这样做有助于防止多种多样的问题。在 C++11 之前，第 8 行~第 10 行是这样写的：

```
int number1 = 0; // first integer to add (initialized to 0)
int number2 = 0; // second integer to add (initialized to 0)
int sum = 0; // sum of number1 and number2 (initialized to 0)
```

在一些遗留的 C++程序中，你会看到使用这种旧式 C++编码风格的初始化语句。我们以后会讨论大括号初始化的优点。

一次性声明多个变量

同一类型的变量可以一次性声明。例如，可以用逗号分隔的列表来声明并初始化全部三个变量，如下所示：

```
int number1{0}, number2{0}, sum{0};
```

但这会影响程序的可读性，而且不方便用注释来说明每个变量的作用。

基本类型

我们将很快讨论用于指定实数的 double 类型和用于指定字符数据的 char 类型。实数是带小数点的数，例如 3.4、0.0 和−11.19。char 变量只能容纳一个小写字母、大写字母、数字或特殊字符（如$或*）。像 int、double、char 和 long long 这样的类型称为**基本类型**（fundamental types）。基本类型的名称通常由一个或多个关键字组成，而且必须全部小写。要查询 C++基本类型的完整列表及其取值范围，请访问：

https://zh.cppreference.com/w/cpp/language/types

标识符和驼峰式命名法

变量名（如 number1）可以是任何有效的标识符。标识符（identifier）是由字母、数字和下划线（_）组成的一系列字符，不能以数字开头，也不能是语言的关键字。C++区分大小写，同一个字母的大小写形式是两个不同的字符。所以，a1 和 A1 是两个不同的标识符。

C++允许任何长度的标识符。不要以下划线和一个大写字母开头，也不要以两个下划线开头，这种形式的标识符是 C++编译器内部使用的。

根据惯例，变量名标识符以小写字母开头，而且名称中第一个单词之后的每个单词都以大写字母开头。例如，firstNumber 的第二个单词 Number 以大写字母 N 开头。这种命名惯例称为**驼峰式命名法**（camel case），因为大写字母就像骆驼的驼峰一样突出。

变量声明位置

变量声明可以放到程序的几乎任何地方，但必须在变量使用前声明。以第 8 行的声明为例：

```
int number1{0}; // first integer to add (initialized to 0)
```

它可以放到第 13 行之前：

```
std::cin >> number1; // read first integer from user into number1
```

第 9 行的声明：

```
int number2{0}; // second integer to add (initialized to 0)
```

可以放到第 16 行之前：

```
std::cin >> number2; // read second integer from user into number2
```

而第 10 行的声明：

```
int sum{0}; // sum of number1 and number2 (initialized to 0)
```

可以放到第 18 行之前：

```
sum = number1 + number2; // add the numbers; store result in sum
```

事实上，第 10 行和第 18 行可以合并为以下声明，并放到第 20 行之前：

```
int sum{number1 + number2}; // initialize sum with number1 + number2
```

从用户处获取第一个值

第 12 行：

```
std::cout << "Enter first integer: "; // prompt user for data
```

会在屏幕上显示 `Enter first integer:`，并后跟一个空格。这种消息称为**提示**（prompt），因为它提示用户采取一个特定的操作。第 13 行：

```
std::cin >> number1; // read first integer from user into number1
```

使用标准输入流对象 `cin`（来自 `std` 命令空间）和**流提取操作符**（stream extraction operator）`>>`从键盘获取一个值，注意箭头的方向即数据的流向。

上述语句执行时，程序等待你为变量 `number1` 输入一个值。你的回应是输入一个整数（此时是作为字符），然后按回车键将其发送给程序。`cin` 对象将数字的字符形式转换为整数值，并将该值赋给变量 `number1`。按回车键还会使光标移动到屏幕上下一行的开头。

当程序期望用户输入一个整数时，用户实际可能输入字母、特殊符号（如#或@）或者带有小数点的数字（如 73.5）等。在本书这些早期的程序中，我们假设用户输入的是有效数据。以后会讨论用于处理数据输入问题的各种技术。

从用户处获取第二个值

第 15 行：

```
std::cout << "Enter second integer: "; // prompt user for data
```

在屏幕上显示 Enter second integer:和一个空格，提示用户采取行动。第 16 行：

```
std::cin >> number2; // read second integer from user into number2
```

从用户处获取变量 number2 的值。

计算用户输入值之和

第 18 行是一个赋值语句：

```
sum = number1 + number2; // add the numbers; store result in sum
```

它使 number1 和 number2 的值相加，使用赋值操作符=将结果赋给 sum。

主要计算在赋值语句中进行。这里的=操作符和+操作符都是**二元操作符**（binary operators），因为每个操作符都有两个操作数。其中，+操作符的两个操作数是 number1 和 number2，而=操作符的两个操作数是 sum 和表达式 number1 + number2 的求值结果。在二元操作符两侧添加空格可以使操作符更醒目，使程序更易读。

显示结果

第 20 行：

```
std::cout << "Sum is " << sum << "\n"; // display sum
```

显示字符串"Sum is "，后跟变量 sum 的值和一个换行符。

注意，上述语句输出了多个不同类型的值。流插入操作符<<“知道”如何输出每种类型的数据。在单个语句中使用多个<<操作符，这称为对多个流插入操作进行**连接**（concatenating）。

还可以直接在输出语句中执行计算。将第 18 行和第 20 行的语句合并成下面这个语句，即可避免使用变量 sum：

```
std::cout << "Sum is " << number1 + number2 << "\n";
```

C++的标志特征就是可以创建自己的数据类型，称为类（将从第 9 章开始讨论）。然后，你也可以“教”C++如何使用>>和<<操作符来输入和输出这些新类型的值。这就是所谓的**操作符重载**（operator overloading），将于第 11 章讨论。

2.5 算术运算

下表总结了算术操作符。

操作	算术操作符	代数式	C++表达式
加	+	$f + 7$	f + 7
减	-	$p - c$	p - c
乘	*	bm 或 $b \cdot m$	b * m
除	/	x / y，$\frac{x}{y}$或$x \div y$	x / y
求余	%	$r \bmod s$	r % s

注意这里使用了多个在代数中没有的符号。星号（*）表示乘法，而百分号（%）是求余操作符（马上就会讲到）。这些算术操作符都是二元操作符。

整数除法

分子和分母都是整数的整数除法会生成一个整数商。例如，表达式 7/4 的求值结果是 1，表达式 17/5 的求值结果是 3。整数除法结果中的任何小数部分都被截断，不会发生四舍五入。

求余操作符

求余操作符%（也称为模除操作符）生成整数除法后的余数，而且只能应用于整数操作数。x % y 生成 x 除以 y 后的余数。因此，7 % 4 的结果是 3，17 % 5 的结果是 2。

用圆括号对子表达式进行分组

圆括号在 C++表达式中的用法和在代数式中一样。例如，要求 b 和 c 之和，再和 a 相乘，可以写成 a * (b + c)。

操作符优先级规则

C++在算术表达式中应用各个操作符的顺序由以下操作符优先级规则决定，这些规则通常与代数中的规则一致：

1. 圆括号内的表达式优先求值。圆括号具有“最高优先级”。对于嵌套或嵌入圆括号的情况，例如：

   ```
   (a * (b + c))
   ```

   最内层圆括号中的表达式首先求值。

2. 其次求值的是乘法、除法和求余操作。如果一个表达式包含多个这样的操作，它们按从左到右的顺序应用。我们说这三个操作符具有相同的优先级。
3. 最后求值的是加法和减法操作。如果一个表达式包含多个这样的操作，它们按从左到右的顺序应用。加法和减法也具有相同的优先级。

附录 A 提供了完整的操作符优先级表。注意。在诸如`(a + b) * (c - d)`这样的表达式中，两组圆括号没有嵌套，但处于“同一级”上，C++标准没有规定这些圆括号的子表达式的求值顺序。

操作符分组

当我们说 C++从左到右应用某些操作符时，我们说的是操作符的**分组**（grouping），有时也称为操作符的**关联性**（associativity）。例如，在以下表达式中：

```
a + b + c
```

加法操作符（`+`）从左到右分组，等同于`(a+ b)+ c`。大多数相同优先级的 C++操作符都从左到右分组。以后会提到一些从右到左分组的操作符。

2.6 决策：相等性和关系操作符

下面要介绍的是 C++的 `if` 语句，它允许程序根据一个条件的真假来采取不同的行动。`if` 语句中的条件可用下表总结的**关系操作符**（relational operators）和**相等性操作符**（equality operators）来形成。

代数关系或相等性操作符	**C++关系或相等性操作符**	**示例 C++条件**	**C++条件的含义**
关系操作符			

>	>	x > y	x 大于 y
<	<	x < y	x 小于 y
≥	>=	x >= y	x 大于或等于 y
≤	<=	x <= y	x 小于或等于 y
相等性操作符			
=	==	x == y	x 等于 y
≠	!=	x != y	x 不等于 y

所有关系操作符具有相同的优先级，从左到右分组。两个相等性操作符都具有相同的优先级，但低于关系操作符的优先级，而且也是从左到右分组。

错误

颠倒操作符!=、>=和<=中一对符号的顺序（分别写成=!、=>和=<）通常是一个语法错误。某些时候，将!=写成=!并不是语法错误，但几乎可以肯定，这是一个会在执行时产生影响的逻辑错误。4.11 节讨论逻辑操作符时，你会明白其中的原因。

混淆==和=

错误

混淆相等性操作符==与赋值操作符=会造成逻辑错误。混淆这些操作符不一定会造成容易发现的语法错误，但可能会造成微妙的逻辑错误。编译器一般会对此提出警告。

使用 if 语句

图 2.5 的程序使用 6 个 if 语句来比较用户输入的两个整数。如果一个给定的 if 语句的条件为真，其主体中的输出语句就会执行。如果条件为假，其主体中的输出语句就不执行。

```cpp
// fig02_05.cpp
// Comparing integers using if statements, relational operators
// and equality operators.
#include <iostream> // enables program to perform input and output

using std::cout; // program uses cout
using std::cin; // program uses cin

// function main begins program execution
int main() {
   int number1{0}; // first integer to compare (initialized to 0)
   int number2{0}; // second integer to compare (initialized to 0)

   cout << "Enter two integers to compare: "; // prompt user for data
   cin >> number1 >> number2; // read two integers from user

   if (number1 == number2) {
      cout << number1 << " == " << number2 << "\n";
   }

   if (number1 != number2) {
      cout << number1 << " != " << number2 << "\n";
   }

   if (number1 < number2) {
      cout << number1 << " < " << number2 << "\n";
   }

   if (number1 > number2) {
      cout << number1 << " > " << number2 << "\n";
   }

   if (number1 <= number2) {
      cout << number1 << " <= " << number2 << "\n";
   }

   if (number1 >= number2) {
      cout << number1 << " >= " << number2 << "\n";
   }
} // end function main
```

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
```

```
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

图 2.5 使用 `if` 语句、关系操作符和相等性操作符比较两个整数

using 声明

第 6 行~第 7 行：

```
using std::cout; // program uses cout
using std::cin; // program uses cin
```

称为 `using` 声明，作用是避免在之前的程序中重复添加 `std::`前缀的必要。现在可以在程序剩余的部分直接写 `cout` 而不是 `std::cout`，直接写 `cin` 而不是 `std::cin`。

using 指令

但是，许多程序员不会像第 6 行~第 7 行那样写，他们更愿意使用 `using` 指令：

```
using namespace std;
```

这样就可以直接使用来自 `std` 命名空间的所有名称，不需要用 `std::`前缀进行限定。在本书早期的章节中，我们将在程序中使用这一指令来简化代码[1]。

变量声明和从用户处读取输入

第 11 行~第 12 行：

```
int number1{0}; // first integer to compare (initialized to 0)
```

---

[1] 第 19 章会讨论在大型系统中使用 `using` 指令的缺点。

```
int number2{0}; // second integer to compare (initialized to 0)
```

声明要在程序中使用的变量并把它们初始化为 0。

第 15 行：

```
cin >> number1 >> number2; // read two integers from user
```

使用连接的几个流提取操作符>>从标准输入设备 cin（这里是键盘）输入两个整数。由于第 7 行的存在，所以可以直接写 cin 而不是 std::cin。输入的第一个值被读入 number1，第二个值被读入 number2。

比较数字

第 17 行~第 19 行的 if 语句：

```
if (number1 == number2) {
   cout << number1 << " == " << number2 << "\n";
}
```

判断变量 number1 和 number2 的值是否相等。如果是，cout 语句显示一行文本，表明数字相等。对于从第 21 行、第 25 行、第 29 行、第 33 行和第 37 行开始的其余 if 语句，均是一旦条件为真，相应的 cout 语句就显示一行适当的文本。

大括号和代码块

在图 2.5 中，每个 if 语句的主体都只有一个语句。为了提高可读性，该语句被缩进。另外要注意的是，我们将每个主体语句都用一对大括号{}括起来，这形成了一个所谓的**复合语句**，或者称为**代码块**、**块**（block）。

虽然单语句的主体不必用大括号{}括起来，但多语句的主体必须用大括号括起来。如果忘记用大括号包围多语句的主体，那么肯定会导致错误。为了避免这种错误，请养成将所有 if 语句的主体语句都用大括号括起来的习惯。

错误

常见逻辑错误：在条件之后添加分号

错误

在 if 语句的条件的右圆括号后紧跟一个分号往往是逻辑错误（虽然不是语法错误）。分号导致 if 语句的主体为空，因此该 if 语句不会执行任何操作，无论其条件是否为真。更糟的是，if 语句原来的主体语句现在变成了在 if 语句之后总是执行的一个语句（块），这常常导致程序产生错误的结果。大多数编译器会对这种逻辑错误发出警告。

分割冗长的语句

在代码编辑器中，一个冗长的语句可能自动散布于多行。如果出于可读性的考虑，必须对其进行分割，那么请选择有意义的分割点，例如在逗号分隔列表的一个逗号之后，或者在冗长表达式中的一个操作符之后。如果一个语句被分割为两行或多行，缩进所有后续行是一个好的做法。

操作符优先级和分组

除了赋值操作符=，本章介绍的所有操作符都从左到右分组。赋值操作符（=）从右到左分组。因此，表达式 `x = y = 0` 等同于 `x = (y = 0)`，即从右到左，首先将 `0` 赋给 `y`，再将 `y` 的求值结果（即 `0`）赋给 `x`。

在写包含大量操作符的表达式时，请参考附录 A 完整的操作符优先级表。确认表达式中的操作符是按照你期望的顺序执行的。但凡不确定复杂表达式中的求值顺序，就用多个语句对其进行分割，或者使用圆括号来强制求值顺序，就像在代数式中做的那样。

2.7 对象自然案例研究：创建和使用标准库类 string 的对象

本书强调尽可能使用 C++标准库和 C++开源社区的各种开源库所提供的、有价值的类。你将专注于了解存在哪些库，选择你的应用程序适用的，使用库所提供的类来创建对象，并发挥这些对象的作用。“对象自然”的意思是，在你学会创建自定义类之前，就能使用功能强大的对象进行编程。

之前其实已经用过 C++对象，具体就是 `cout` 和 `cin` 对象，它们分别封装了输出和输入机制。在幕后，这些对象是使用头文件`<iostream>`所提供的类来创建的。本节将创建 C++标准库的 `string` 类的对象并与之交互[1]。

使用 string 类

类自己是不能执行的。一个 `Person` 对象可以告诉一个 `Car` 对象需要做什么（开快点、开慢点、左转、右转等等）来“驱动”这个 `Car` 对象，同时不需要知道汽车的内部工作机制。同样地，`main` 函数可以通过调用 `string` 对象的成员函数来“驱动”它，同时不需要知道

[1] 在后续章节，你还会更多地了解 `string` 类的功能。第 8 章详细讨论了 `string` 类，试验了它的更多成员函数。

该类具体是如何实现的。从这个角度说，下面这个程序中的 main 可以称为驱动程序。图 2.6 的 main 函数“驱动”了几个字符串成员函数。

```
// fig02_06.cpp
// Standard library string class test program.
#include <iostream>
#include <string>
using namespace std;

int main() {
   string s1{"happy"};
   string s2{" birthday"};
   string s3; // creates an empty string

   // display the strings and show their lengths
   cout << "s1: \"" << s1 << "\"; length: " << s1.length()
      << "\ns2: \"" << s2 << "\"; length: " << s2.length()
      << "\ns3: \"" << s3 << "\"; length: " << s3.length();

   // compare strings with == and !=
   cout << "\n\nThe results of comparing s2 and s1:" << boolalpha
      << "\ns2 == s1: " << (s2 == s1)
      << "\ns2 != s1: " << (s2 != s1);

   // test string member function empty
   cout << "\n\nTesting s3.empty():\n";

   if (s3.empty()) {
      cout << "s3 is empty; assigning to s3;\n";
      s3 = s1 + s2; // assign s3 the result of concatenating s1 and s2
      cout << "s3: \"" << s3 << "\"";
   }

   // testing new C++20 string member functions
   cout << "\n\ns1 starts with \"ha\": " << s1.starts_with("ha") << "\n";
   cout << "s2 starts with \"ha\": " << s2.starts_with("ha") << "\n";
   cout << "s1 ends with \"ay\": " << s1.ends_with("ay") << "\n";
   cout << "s2 ends with \"ay\": " << s2.ends_with("ay") << "\n";
}
```

```
s1: "happy"; length: 5
s2: " birthday"; length: 9
s3: ""; length: 0

The results of comparing s2 and s1:
s2 == s1: false
s2 != s1: true

Testing s3.empty():
s3 is empty; assigning to s3;
s3: "happy birthday"

s1 starts with "ha": true
s2 starts with "ha": false
s1 ends with "ay": false
s2 ends with "ay": true
```

图 2.6 标准库 string 类的测试程序

实例化对象

一般情况下，除非创建类的对象，否则不能调用该类的成员函数[①]。创建类的对象也称为**实例化**（instantiating）一个对象。第 8 行~第 10 行创建了三个 string 对象。

- s1 初始化为一个字符串字面值拷贝"happy"。
- s2 初始化为一个字符串字面值拷贝" birthday"。
- s3 默认初始化为空字符串，即""。

以前在声明 int 变量时，编译器事先就知道 int 是什么，因为它是 C++内置的一个基本类型。但在本例的第 8 行~第 10 行，如果不事先做一些准备，那么编译器并不知道 string 是什么。事实上，它是 C++标准库中的一个类类型（class type）。

只要进行了恰当的封装，类就可以被其他程序员重用。这是像 C++这样的面向对象编程语言所具备的最显著的优势之一。这些语言提供了多种多样的库，其中预置了功能强大的类。例如，可以在任何程序中重用 C++标准库中的类，只要包含恰当的头文件即可，本例包含的是<string>头文件（第 4 行）。string 这个名称，就像 cout 一样，从属于命名空间 std。

string 的 length 成员函数

第 13 行~第 15 行输出每个字符串及其长度。string 类的 length 成员函数返回一个特定的

[①] 9.20 节会讲到，可直接在类上调用它的静态成员函数，而不必先创建类的对象。

string 对象中的字符数。第 13 行的以下表达式：

```
s1.length()
```

通过调用 s1 对象的 length 成员函数来返回该对象的长度。要为一个特定的对象调用成员函数，需要先指定对象名称（s1），后跟点操作符（.），再后跟成员函数名称（length）和一组圆括号。空的圆括号表示 length 函数不需要任何额外的信息就能执行其任务。你很快就会看到，还有一些成员函数需要称为**实参**（arguments）的额外信息才能执行其任务。

从 main 的角度看，在 length 成员函数被调用时：

1. 程序将执行从调用（main 中的第 13 行）转移到成员函数 length。因为 length 是通过 s1 对象调用的，所以 length“知道”要操作哪个对象的数据。
2. 接着，成员函数 length 执行它的任务——也就是把 s1 的长度返回给调用该函数的第 13 行。main 函数不知道 length 具体如何执行任务，就像驾驶员不知道发动机、变速箱、转向机构和刹车系统是如何实现的。
3. cout 对象显示成员函数 length 返回的字符数，程序继续执行，显示字符串 s2 和 s3 及其长度。

用相等性操作符比较字符串对象

和数字一样，字符串也能相互比较。第 18 行~第 20 行使用相等必操作符比较 s2 和 s1。注意，字符串比较区分大小写[①]。

默认情况下，在输出一个条件的值时，C++显示 0 代表假，1 代表真。来自<iostream>头文件的流操纵元 boolalpha（第 18 行）告诉输出流将条件值显示为 false 或 true。

string 的 empty 成员函数

第 25 行调用 string 的 empty 成员函数。如果字符串为空，它会返回 true。这里的“空”（empty）是指字符串长度为 0。如果字符串不为空，它会返回 false。我们故意让 s3 对象默认初始化为空字符串，所以会执行 if 语句的主体。

字符串连接和赋值

第 27 行使用+操作符使字符串 s1 和 s2 相加，并将结果赋给 s3。两个字符串相加的操作称为**字符串连接**（string concatenation）。在赋值之后，s3 包含 s1 的字符加上 s2 的字符，即

[①] 第 8 章会讲到，是用字符串中代表每个字符的数值进行字典序（lexicographical）比较。

"happy birthday "。第 28 行输出 s3 以证明赋值成功。

C++20 string 的成员函数 starts_with 和 ends_with

20 第 32 行~第 35 行演示了新的 C++20 string 成员函数 start_with 和 ends_with。如果字符串以指定的子串开始或结束，这两个成员函数将分别返回 true；否则，它们返回 false。第 32 行和第 33 行显示 s1 是以"ha"开头，但 s2 不是。第 34 行和第 35 行显示 s1 不是以"ay"结束，但 s2 是。

2.8 小结

本章介绍了 C++许多重要的基本特性，包括在屏幕上显示数据、从键盘输入数据和声明基本类型的变量。具体地说，你学会了使用输出流对象 cout 和输入流对象 cin 来构建简单的交互式程序。我们声明并初始化了变量，并使用算术操作符来进行计算。讨论了 C++应用操作符的顺序（即操作符优先规则），以及操作符的分组（也称为操作符的关联性）。你学习了如何利用 C++的 if 语句让程序做出决策。讨论了相等性和关系操作符，我们用它们来形成 if 语句中的条件。

本章最后演示了学习 C++的“对象自然”方法，即创建 C++标准库类 string 的对象，并使用相等性操作符和各种字符串成员函数与之交互。在后续章节中，还会创建并使用许多现有类的对象，从而用最少的代码完成重要的任务。然后，在第 9 章~第 11 章，你将开始创建自定义类。届时会理解 C++如何使你能够对现有的、有价值的类进行修改，从而打造符合自己需要的类。下一章开始介绍控制语句，它们指定了一个程序的行动顺序。

第 3 章 控制语句(第 1 部分)

学习目标：

- 使用 `if` 和 `if...else` 选择语句在备选行动之间做出选择。
- 使用 `while` 循环语句来重复执行程序中的语句。
- 使用计数器控制的循环和哨兵值控制的循环。
- 使用嵌套控制语句。
- 使用复合赋值操作符和递增/递减操作符。
- 理解为什么基本数据类型不可移植。
- 用一个案例研究继续我们的“对象自然”方法，创建和操作任意大小的整数。
- 使用 C++20 新的文本格式化功能，它比早期 C++版本的更简洁、更强大。

3.1 导读

本章和下一章将介绍结构化编程的理论和原则。这些概念在构建类和操作对象方面至关重要。我们更详细地讨论了 if 语句，并介绍了 if...else 和 while 语句。还介绍了复合赋值操作符以及递增和递减操作符。

我们讨论了为什么基本类型是不可移植的。我们通过一个关于任意大小的整数的案例研究继续我们的“对象自然”方法，这种整数可以表示超出计算机硬件所支持的整数范围的值。

本章将开始介绍 C++20 新的文本格式化功能，它们基于 Python、Microsoft .NET 语言（例

如 C#和 Visual Basic）和 Rust 中的功能[①]。和早期 C++版本提供的功能相比，C++20 提供的这些功能更简洁、更强大。

3.2 控制结构

20 世纪 60 年代，人们清楚地认识到，滥用控制权的转移是软件开发团队遇到的许多问题的根源。人们把责任归咎于 `goto` 语句（当时大多数编程语言都设计了这个语句），它允许将控制权转移到程序中的几乎任何位置。Böhm 和 Jacopini 的研究[②]表明，不使用任何 `goto`

[①] Victor Zverovich，“Text Formatting”，July 16, 2019，https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html。

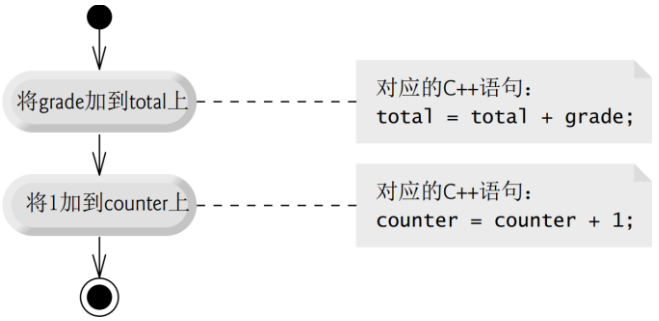
[②] C. Böhm 和 G. Jacopini，“Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules”，Communications of the ACM, Vol. 9, No. 5，May 1966, pp. 336–371。

语句也能编写程序。那个时代的程序员所面临的挑战是如何将他们的风格转变为“无goto编程”。**结构化编程**（structured programming）这个词几乎成了“消除 goto”的同义词。结果令人印象深刻。软件开发团队报告了更短的开发时间，更经常的按时交付，而且更经常地在预算内完成软件项目。这一系列成功的关键在于，结构化的程序更清晰，更容易调试和修改，而且更有可能从一开始就避免犯错。

Böhm 和 Jacopini 的研究表明，所有程序都可以只用三种控制结构来编写：顺序结构、选择结构和循环结构。我们将讨论 C++如何实现其中的每一个。

3.2.1 顺序结构

C++的语句默认就是顺序结构。除非另有指示，否则语句会按它们在程序中出现的顺序依次执行。以下 UML[①]**活动图**（activity diagram）展示了一个典型的顺序结构，其中的两个计算是按顺序进行的。



C++允许在顺序结构中采取任意数量的行动。很快就会看到，在任何可以放置一个行动的地方，都可以按顺序放置多个行动。

活动图建模的是软件系统的一个部分的**工作流**（workflow，也称为**活动**）。这样的工作流可能包括算法的一部分，就像上图的顺序结构。活动图由各种符号组成，例如行动状态符号（圆角矩形）、菱形和小圆圈。这些符号用过渡箭头连接，代表活动的流程，也就是行动的顺序。

上述顺序结构活动图包含两个**行动状态**（action states），每个都包含一个**行动表达式**（action expression），例如“将 grade 加到 total 上”或者“将 1 加到 counter 上”。行动

[①] 本章和第 4 章使用 UML 来展示控制语句中的控制流。第 9 章~第 10 章讨论自定义类的开发时，会再次用到 UML。

表达式指定了要执行的特定行动。活动图中的箭头代表**过渡**[1]（transitions），表示行动状态所代表的行动的发生顺序。

活动图顶部的实心圆代表**初始状态**（initial state）——即程序在执行所建模的行动之前，工作流的开始。图底部内部实心的空心圆代表**最终状态**（final state），即程序执行完行动之后，工作流的结束。

顺序结构活动图还包括右上角折叠的矩形。这些是 UML **注释**（类似于 C++注释），它对图中的符号的作用进行解释。一条虚线将每个注释与它所描述的元素连接起来。上图的 UML 注释显示了该图与每个行动状态的 C++代码的关系。活动图一般不显示 C++代码。

3.2.2 选择语句

C++有三种类型的选择语句。`if` 语句在条件为真时执行（选择）一个（或一组）行动，在条件为假时跳过该行动。`if...else` 语句在条件为真时执行一个（或一组）行动，在条件为假时执行另一个（或一组）行动。`switch` 语句（第 4 章）根据一个表达式的值，从多个不同的行动（或行动组）中选择一个来执行。

`if` 语句被称为**单选语句**，因为它选择或忽略了一个行动（或行动组）。`if...else` 语句被称为**双选语句**，因为它在两个不同的行动（或行动组）之间选择。`switch` 语句则被称为**多选语句**，因为它从许多不同的行动（或行动组）中选择一个。

3.2.3 循环语句

C++提供了四种用于在循环持续条件为真时重复执行语句的循环语句，包括 `while`、`do...while`、`for` 和基于范围的 `for`。`while` 和 `for` 语句执行其行动（或行动组）零次或多次。如果循环继续条件一开始就为假，那么它的行动（或行动组）一次都不会执行。相反，`do...while` 语句执行其行动（或行动组）至少一次。第 4 章介绍了 `do...while` 和 `for` 语句。第 6 章介绍了基于范围的 for 语句。

关键字

`if`、`else`、`switch`、`while`、`do` 和 `for` 都是 C++关键字。关键字不能作为变量名这样的标识符使用，并且只包含小写字母（有时会有下划线）。下面是 C++关键字的完整列表。

C++关键字

[1] 译注：一说“转换”。

alignas	alignof	and	and_eq	asm
auto	bitand	bitor	bool	break
case	catch	char	char16_t	char32_t
class	compl	const	const_cast	constexpr
continue	decltype	default	delete	Do
double	dynamic_cast	else	enum	explicit
export	extern	false	final	float
for	friend	goto	if	import
inline	int	long	module	mutable
namespace	new	noexcept	not	not_eq
nullptr	operator	or	or_eq	override
private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static
static_assert	static_cast	struct	switch	template
this	thread_local	throw	true	try
typedef	typeid	typename	union	unsigned
using	void	volatile	virtual	wchar_t

while	xor	xor_eq		
C++20 新增关键字				
char8_t	concept	consteval	constinit	co_await
co_return	co_yield	requires		

3.2.4 控制语句小结

C++只有三种控制结构，从现在起，我们把它们称为控制语句：

- 顺序
- 选择（if、if…else 和 switch）
- 循环（while、do…while、for 和基于范围的 for）

针对想要实现的算法，我们通过适当地组合这些语句来构建程序。每个控制语句都可以建模为一个活动图。每个图都包含一个初始状态和一个最终状态，分别代表控制语句的入口和出口。利用单入/单出的控制语句，可以很容易构建出可读性好的程序。为此，只需采用**控制语句堆叠**（control-statement stacking）技术，将一个控制语句的出口与下一个控制语句的入口连接。除此之外，只有另一种方式可以连接控制语句，即**控制语句嵌套**（control-statement nesting），也就是将一个控制语句中放到另一个控制语句中。因此，C++程序中的算法仅由三种控制语句构成，仅以两种方式组合。这就是所谓的“大道至简”。

3.3 if 选择语句

2.6 节简单介绍了 if 单选语句。程序使用选择语句在备选的行动路线中进行选择。例如，假设一次考试的及格线是 60 分，以下 C++语句判断条件 studentGrade >= 60 是否为真。

```
if (studentGrade >= 60) {
   cout << "Passed";
}
```

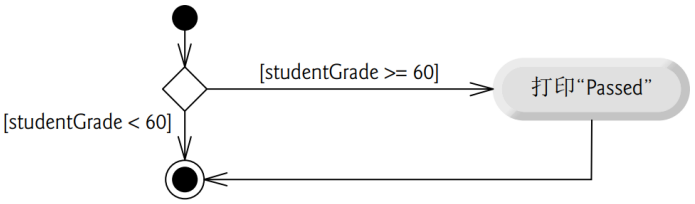
如果条件为真，就打印"Passed"，然后顺序执行下个语句。如果条件为假，就忽略主体中的输出语句，顺序执行下个语句。这个选择语句第二行的缩进虽然可选，但强烈建议加上以提高程序的可读性。

bool 数据类型

第 2 章是用关系或相等性操作符创建条件。实际上，任何求值为零或非零的表达式都可以用作条件。其中，零被视为假，而非零被视为真。C++还为布尔变量专门提供了 bool 数据类型，这种变量只能容纳 true 和 false 两个值之一，两者都是 C++关键字。编译器可以隐式地将 true（真）转换成 1，将 false（假）转换成 0。

if 语句的 UML 活动图

下图展示了上述单选 if 语句的控制流。



这张图包含了活动图中最重要的符号：菱形，或者称为**决策符号**（decision symbol），它表示要做出一个决定。工作流沿着与该符号关联的**保护条件**（guard conditions）所决定的路径继续。保护条件要么为真，要么为假。每个从决策符号引出的过渡箭头都附有一个保护条件（在箭头旁边的方括号中指定）。保护条件为真，工作流就进入过渡箭头所指向的行动状态。上图显示，如果成绩大于或等于 60 分（即条件为真），程序会打印"Passed"，然后过渡到活动的最终状态。如果成绩小于 60 分（即条件为假），程序则立即过渡到最终状态，而不打印消息。if 语句是一种单入/单出的控制语句。

3.4 if…else 双选语句

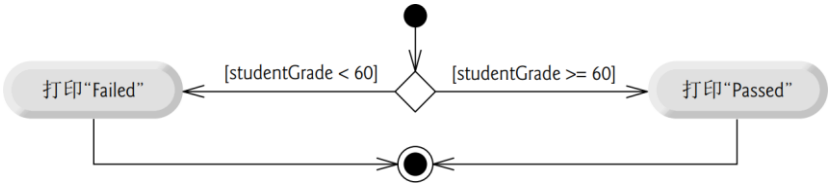
if 单选语句只在条件为真时执行指定的行动。if...else 双选语句则允许指定在条件为真时执行一个行动，在条件为假时执行另一个行动。例如，以下 C++语句在 studentGrade >= 60 时打印"Passed"，在小于 60 时打印"Failed"。

```
if (studentGrade >= 60)
{
    cout << "Passed";
}
else {
    cout << "Failed";
}
```

不管哪种情况，在打印行动完毕后，都会顺序执行下一个语句。注意，else 的主体也进行了缩进。不管采用的是什么缩进惯例，在程序中都要保持一致。

if…else 语句的 UML 活动图

下图展示了上述 if…else 语句的控制流。



3.4.1 嵌套 if…else 语句

程序可以将 if...else 语句放到其他 if...else 语句中来测试多个条件，这样创建的是嵌套 if...else 语句。例如，下面这个嵌套 if...else 语句对考试成绩大于或等于 90 分的学生打印"A"，对 80~89 分的学生打印"B"，对 70~79 分的学生打印"C"，对 60~69 分的学生打印"D"，对其他所有成绩打印"F"。我们用阴影来强调嵌套。

```
if (studentGrade >= 90) {
   cout << "A";
}
else {
   if (studentGrade >= 80) {
      cout << "B";
   }
   else {
      if (studentGrade >= 70) {
         cout << "C";
      }
      else {
         if (studentGrade >= 60) {
            cout << "D";
         }
         else {
            cout << "F";
         }
      }
   }
}
```

如果变量 studentGrade 大于或等于 90，嵌套 if...else 语句中的前四个条件都为真，但只有第一个 if...else 语句的 if 主体语句会执行。在该语句执行后，“最外层”if...else 语句的整个 else 部分会被跳过。上述 if...else 语句也可以写成以下形式，效果一样，但使用的大括号更少，空白和缩进也更少。

```
if (studentGrade >= 90) {
   cout << "A";
}
else if (studentGrade >= 80) {
   cout << "B";
}
```

```
else if (studentGrade >= 70) {
   cout << "C";
}
else if (studentGrade >= 60) {
   cout << "D";
}
else {
   cout << "F";
}
```

这样写能避免代码向右边深度缩进。缩进过深，可能造成一行代码发生自动换行。本书总是使用大括号（{和}）将控制语句主体括起来，这样能避免一种称为“空悬 else”的逻辑错误。

3.4.2 代码块

如果 if 语句的主体要包含多个语句，用大括号把它们括起来既可。无论主体语句是否只有一个，始终使用大括号是一个好的做法。一对大括号中的语句（例如控制语句或函数的主体）构成了一个所谓的**块**或**代码块**（block）。函数中能放入一个语句的地方，都能放入多个语句。

下例在 if...else 语句的 else 子句中包含了一个多语句块：

```
if (studentGrade >= 60) {
   cout << "Passed";
}
else {
   cout << "Failed\n";
   cout << "You must retake this course.";
}
```

如果 studentGrade 小于 60，程序会执行 else 主体中的两个语句，即打印下面两行文本：

```
Failed
You must retake this course.
```

如果不用大括号将 else 子句中的两个语句括起来，以下语句：

```
cout << "You must retake this course.";
```

会跑到 if...else 语句的 else 子句的主体外面，无论 studentGrade 是否小于 60 都会执行，这是一个逻辑错误。

空语句

在任何可以放单个语句的地方，都可以放一个代码块。类似地，在任何通常可以放一个语句的地方，都可以放一个空语句。空语句就是一个分号（;），它没有任何作用。

3.4.3 条件操作符(?:)

C++提供了条件操作符（?:）来代替 if...else 语句。这可以使代码更短、更清晰。条件操作符是 C++唯一的三元操作符（即需要三个操作数的操作符）。操作数和?:符号共同构成一个**条件表达式**（conditional expression）。例如，以下语句打印条件表达式的求值结果：

```
cout << (studentGrade >= 60 ? "Passed" : "Failed");
```

?左侧的操作数是条件，第二个操作数（?和:之间）是条件为真时整个条件表达式的求值结果。:右侧的操作数则是条件为假时整个条件表达式的求值结果。在本例中，如果条件 studentGrade >= 60 为真，条件表达式将求值为字符串"Passed"，否则求值为字符串"Failed"。所以，上述语句用条件操作符来执行和3.4节第一个if…else语句一样的功能。注意，条件操作符的优先级较低，所以一般将整个条件表达式放到圆括号中。

3.5 while 循环语句

循环语句指定程序在某个条件保持为真时重复一个行动。作为C++ while循环语句的一个例子，请考虑如何发现大于100的3的第一个乘幂。以下while语句执行后，变量product将包含结果。

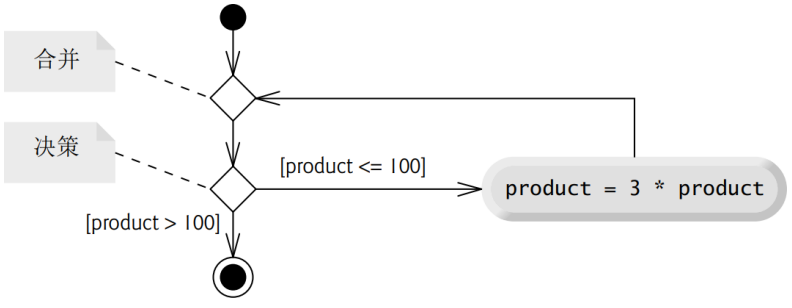
```
int product{3};

while (product <= 100) {
   product = 3 * product;
}
```

while语句的每一次循环都会使product乘以3，所以product的值依次变成9、27、81和243。当 product 变成 243 时，product<=100 条件变成假，这就终止了循环。所以，product的终值是243。随后，程序从while语句之后的语句继续执行。

while 语句的 UML 活动图

以下while语句的UML活动图引入了**合并符号**（merge symbol）。



UML 将合并符号和决策符号都表示为菱形。合并符号将两个活动流连接成一个。在上图中，合并符号连接了来自初始状态和行动状态的过渡，使两者都流向决定循环是否应该开始（或继续）执行的决策。

可以根据进出的过渡箭头的数量来区分决策与合并符号。决策符号有一个箭头指向菱形，有两个或更多箭头离开菱形，代表该决策可能发生的状态过渡。另外，每个离开决策符号的箭头都有一个保护条件。相反，合并符号有两个或更多过渡箭头指向它，但只有一个箭头离开它，表示它合并了多个活动流以继续活动。与合并符号关联的过渡箭头都没有保护条件。

3.6 计数器控制的循环

考虑以下问题陈述：

10 个学生的一个班级进行了一次测验。你可以获得这次测验的成绩（0~100 的整数）。计算此次测验的班级平均成绩。

班级平均成绩等于成绩之和除以学生人数。程序必须输入每个成绩，将所有输入的成绩相加，求平均值并打印结果。

我们使用**计数器控制的循环**（counter-controlled iteration）来一次输入一个成绩。这个技术要求用计数器控制一组语句的执行次数。在本例中，循环在计数器超过 10 后终止。

3.6.1 实现计数器控制的循环

在图 3.1 中，`main` 函数用计数器控制的一个循环来计算班级平均成绩。它允许用户输入 10 个成绩，然后计算并显示平均成绩。

```
1  fig03_01.cpp
2  // Solving the class-average problem using counter-controlled iteration.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // initialization phase
8     int total{0}; // initialize sum of grades entered by the user
9     int gradeCounter{1}; // initialize grade # to be entered next
10
11    // processing phase uses counter-controlled iteration
12    while (gradeCounter <= 10) { // loop 10 times
13       cout << "Enter grade: "; // prompt
14       int grade;
15       cin >> grade; // input next grade
16       total = total + grade; // add grade to total
17       gradeCounter = gradeCounter + 1; // increment counter by 1
18    }
19
20    // termination phase
21    int average{total / 10}; // int division yields int result
22
23    // display total and average of grades
24    cout << "\nTotal of all 10 grades is " << total;
25    cout << "\nClass average is " << average << "\n";
26 }
```

```
Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

图 3.1 使用计数器控制的循环计算班级平均成绩

main 中的局部变量

第 8 行、第 9 行、第 14 行和第 21 行分别声明 int 变量 total、gradeCounter、grade 和 average。变量 grade 用于存储用户的输入。在代码块（如函数的主体）中声明的变量是所谓的**局部变量**（local variable），其**作用域**（scope）从声明位置开始，到代码块的右括号结束。所有局部变量在使用前都必须声明。在 while 循环主体中声明的 grade 变量只能在这个代码块中使用。

初始化 total 和 gradeCounter 变量

第 8 行~第 9 行声明变量 total 和 gradeCounter，并分别初始化为 0 和 1。注意，这些变量必须先声明并初始化，然后才能在计算中使用。

从用户处读入 10 个成绩

只要 gradeCounter 的值小于或等于 10，while 语句（第 12 行~第 18 行）就会继续下一次循环。第 13 行提示"Enter grade: "（输入成绩）。第 15 行将用户通过键盘输入的成绩赋给 grade 变量。然后，第 16 行将用户输入的新成绩加到总分上，并将结果赋给 total，取代它之前的值。第 17 行在计数器变量 gradeCounter 上加 1，表示程序已处理好了一个成绩，准备好由用户输入下一个成绩。gradeCounter 不断递增，最终导致它超过 10，从而终止循环。

计算并显示班级平均成绩

循环终止时，第 21 行在 average 变量的大括号初始化器（braced initializer）中计算平均成绩。第 24 行显示文本 "Total of all 10 grades is "（总成绩），后跟 total 变量的值。然后，第 25 行显示文本"Class average is "（平均成绩），后跟 average 变量的值。执行到第 26 行时，程序终止。

3.6.2 整数除法和截断

这个例子的平均成绩计算产生的是一个 int 结果。从程序的示例执行可以看出，总成绩是 846 分，除以 10 应得 84.6。像 84.6 这样含小数点的数字称为**浮点数**（floating-point numbers）。但是，在求班级平均成绩的示例程序中，由于 total 和 10 都是整数，所以 total / 10 的结果也是整数，即 84。在 C++程序中，两个整数相除称为**整数除法**（integer division），结果中的任何小数部分都会被截断。下一节将介绍如何在计算平均成绩是获得更准确的浮点结果。例如，7 / 4 在常规算术中的结果应该是 1.75，但在整数除法中被截断为 1，而不是四舍五入的 2。

3.7 哨兵值控制的循环

下面对 3.6 节的班级平均成绩问题进行了一般化：

开发一个求班级平均成绩的程序，每次运行都能处理输入的任意数量的成绩。

在上一个班级平均成绩例子中，问题陈述中限死了学生数量，所以有多少个成绩（10 个）是事先知道的。而在一般化后，我们并不知道用户在程序执行过程中会输入多少个成绩。程序必须能处理任意数量的成绩。

解决这个问题的一个方案是使用**哨兵值**（sentinel value）来指示“数据输入结束”。用户输入所有有效的成绩，然后输入一个哨兵值，表示没有更多成绩需要输入了。

哨兵值的选择必须得当，不能和任何要实际计算的输入值混淆。测试成绩肯定全是非负的整数，所以–1 就是一个可接受的哨兵值。例如，用户连续输入 95，96，75，74，89 和–1，程序就可以正常地计算前 5 个输入值的平均值。由于–1 是哨兵值，所以不参与实际的平均值计算。

注意，用户可能在输入任何有效的成绩前就输入–1，这会导致成绩数量为零。所以，必须在计算平均成绩前测试这种情况。根据 C++标准，在浮点算术中除以零的结果“未定义”（undefined）。执行除法（`/`）或求余（`%`）计算时，由于右操作数可能为零，所以应测试这种情况并予以相应的处理（例如，显示一条错误消息），而不是让计算继续。

3.7.1 实现哨兵值控制的循环

图 3.2 实现了哨兵值控制的循环。虽然用户输入的每个成绩都是整数，但平均成绩计算可能会产生一个浮点数，而 `int` 不能表示这样的数字。C++提供了数据类型 `float`、`double` 和 `long double` 来存储内存中的浮点数。这些类型的主要区别在于，`double` 变量能存储比 `float` 变量更大、更精确的值；即小数点右边的数字更多，这称为数字的**精度**（precision）。类似，`long double` 存储的值比 `double` 更大、更精确。第 4 章会更多地讨论各种浮点类型。

```cpp
// fig03_02.cpp
// Solving the class-average problem using sentinel-controlled iteration.
#include <iostream>
#include <iomanip> // parameterized stream manipulators
using namespace std;

int main() {
   // initialization phase
   int total{0}; // initialize sum of grades
   int gradeCounter{0}; // initialize # of grades entered so far

   // processing phase
   // prompt for input and read grade from user
   cout << "Enter grade or -1 to quit: ";
   int grade;
   cin >> grade;

   // loop until sentinel value is read from user
   while (grade != -1)  {
      total = total + grade; // add grade to total
      gradeCounter = gradeCounter + 1; // increment counter

      // prompt for input and read next grade from user
      cout << "Enter grade or -1 to quit: ";
      cin >> grade;
   }

   // termination phase
   // if user entered at least one grade...
   if (gradeCounter != 0) {
      // use number with decimal point to calculate average of grades
      double average{static_cast<double>(total) / gradeCounter};

      // display total and average (with two digits of precision)
      cout << "\nTotal of the " << gradeCounter
         << " grades entered is " << total;
      cout << setprecision(2) << fixed;
      cout << "\nClass average is " << average << "\n";
   }
   else { // no grades were entered, so output appropriate message
      cout << "No grades were entered\n";
   }
}
```

```
Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

图 3.2 使用哨兵值控制的循环计算班级平均成绩

记住，整数除法产生的是整数结果。但是，这个程序引入了一个**强制类型转换操作符**（cast operator），强制平均成绩计算产生一个浮点结果。这个程序还在 `while` 语句后连接了一个 `if...else` 语句。该程序的大部分代码与图 3.1 相同，所以我们重点讨论新概念。

哨兵值控制的循环与计数器控制的循环的程序逻辑

第 10 行将 `gradeCounter` 初始化为 `0`，因为此时尚未输入任何成绩。记住，该程序使用哨

兵值控制的循环来输入成绩。只有在用户输入一个有效的成绩时，程序才会递增计数器 gradeCounter。第 32 行声明 double 变量 average，它将计算出来的班级平均成绩作为浮点数来存储。

比较这个程序由哨兵值控制的循环和图 3.1 由计数器控制的循环的程序逻辑。在计数器控制的循环中，while 语句的每一次循环（图 3.1 的第 12 行~第 18 行）都从用户处读取指定次循环的值。而在哨兵值控制的迭代中，程序在进入 while 之前提示并读取用户输入的第一个值（图 3.2 的第 14 行和第 16 行）。该值决定了控制流是否应该进入 while 的主体。如果条件为假，即用户输入了哨兵值，没有输入任何有效成绩，那么循环主体一次都不会执行。如果条件为真，就开始执行 while 循环的主体，将 grade 值加到 total 上并递增 gradeCounter。然后，循环主体的第 24 行~第 25 行输入用户的下一个值。接着，程序控制在第 26 行到达循环主体的右大括号，所以继续执行 while 条件的测试（第 19 行）。该条件使用用户上次输入的成绩来判断循环主体是否应该再次执行。

下一个成绩总是在测试 while 条件之前由用户输入。这允许程序在处理该值（即把它加到总成绩上）之前判断刚刚输入的值是不是哨兵值。如果输入的是哨兵值，循环就会终止，程序不会把-1 加到 total 上。

循环终止后，第 30 行~第 42 行的 if...else 语句开始执行。第 30 行的条件判断是否输入了任何成绩。如果没有输入任何成绩，就会执行 if...else 语句的 else 部分，并显示消息"No grades were entered"（没有输入成绩）。if...else 语句执行完毕后，整个程序终止。

3.7.2 基础类型之间的显式和隐式转换

如果至少输入一个有效成绩，图 3.2 的第 32 行：

```
double average{static_cast<double>(total) / gradeCounter};
```

就会计算平均成绩。之前在讲图 3.1 的程序时提到过，整数除法肯定产生一个整数结果。即使将 average 声明为 double 类型，再将第 32 行换成下面这样的写法：

```
double average{total / gradeCounter};
```

在除法结果被用于初始化 average 之前，它还是会失去商的小数部分。

static_cast 操作符

在本例中，为了用整数执行浮点计算，首先要用 static_cast 操作符创建临时的浮点值。第 32 行将其圆括号中的操作数（total）的临时拷贝转换为尖括号中指定的类型（double）。此时，存储在 int 变量 total 中的值仍然是一个整数。以这种方式使用强制类型转换操作符称为**显式转换**（explicit conversion）。static_cast 是我们将要讨论的几个

强制类型转换（cast）操作符之一。

提升

进行了强制类型转换之后，现在的计算式是 `total` 的临时 `double` 拷贝除以 `int` 类型的 `gradeCounter`。执行算术运算时，编译器只知道如何求值所有操作数类型都一样的表达式。所以，编译器会对选定的操作数执行一种称为**提升**（promotion）的操作，这也称为**隐式转换**（implicit conversion）。在同时含有 `int` 和 `double` 数据类型的表达式中，C++将 `int` 操作数提升为 `double` 值。所以在第 32 行，C++会将 `gradeCounter` 的值的临时副本提升为 `double` 类型，然后再执行除法运算。最终，`average` 被初始化为浮点结果。5.6 节会讨论哪些基本类型允许提升。

任意类型的强制类型转换操作符

第 9 章一开始就会讲到，强制类型转换操作符可应用于所有基本类型和其他类型，只需在 `static_cast` 关键字后面的尖括号（`<`和`>`）中指定要转换成的类型即可。它是**一元操作符**，只有一个操作数。其他一元操作符还有应用于表达式的一元加（`+`）和减（`-`）操作符，例如`-7` 或`+5`。强制类型转换操作算符的优先级是第二高的。

3.7.3 格式化浮点数

下面简单说明一下图 3.2 采用的格式化特性。更深入的解释请参见本书放在网上的第 19 章。

`setprecision` 参数化流操纵元

第 37 行调用 `setprecision(2)`将浮点数的精度设为两位小数（例如 92.37）。`setprecision` 是一种**参数化流操纵元**（parameterized stream manipulator），因其需要一个实参（本例是 2）来执行其任务。使用参数化流操纵元的程序必须包含头文件`<iomanip>`（第 4 行）。

`fixed` 非参数化流操纵元

非参数化流操纵元 `fixed`（第 37 行）不需要实参，表示浮点值应该以**定点格式**（fixed-point format）输出。这正好与**科学记数法**（scientific notation）[1]相反。科学记数法将数字显示为1.0~10.0乘以10的幂。所以，在科学记数法中，3100.0这个值显示为$3.1e+03$（即3.1×10^3）。这种格式对于显示极大或极小的值很有用。

[1] 网上的第 19 章会进一步解释如何使用科学记数法进行格式化。

定点格式则强制浮点数在显示时不使用科学记数法。即使是整数，定点格式也能强制打印小数点和尾随的零，例如 88.00。如果不指定定点格式，88.00 会打印成 88，没有小数点和尾随的零。

流操纵元 `setprecision` 和 `fixed` 进行的是所谓的**粘性设置**（sticky settings）。一旦指定，程序之后所有浮点值的格式都会应用这些设置，直至你再次改变它们。本书放在网上的第 19 章介绍了如何在应用粘性设置之前捕获当前的流格式化设置，这样就可以在以后恢复原始格式化设置。

浮点数的四舍五入

使用流操纵元 `fixed` 和 `setprecision` 时，打印的值被四舍五入到当前精度所指定的小数位数。内存中的值保持不变。例如，如果将精度设为 2，那么 87.946 和 67.543 会分别四舍五入为 87.95 和 67.54[①]。

图 3.2 的第 37 行和第 38 行共同输出四舍五入后的班级平均成绩，保留两位小数。程序执行时，我们输入的三个成绩总计 257 分，平均成绩应该是 85.666...分，但显示的是四舍五入后的结果 85.67。

3.8 嵌套控制语句

如前所述，控制语句可以按顺序堆叠（连接）在一起。在这个案例研究中，我们将讨论控制语句的另一种结构化连接方式：一个控制语句嵌套在另一个控制语句中。

3.8.1 问题陈述

考虑以下问题陈述：

大学开设了一门课程，为学生参加该州的房地产经纪人执照考试做准备。去年，完成该课程的学生中有 10 人参加了考试。学校想知道学生在考试中的表现如何。你需要写一个程序来汇总结果。已经拿到了这 10 名学生的名单。如果学生通过了考试，名字旁边会写 1；如果没有通过考试，则会写 2。

程序应该对考试的结果进行分析，具体包括：

1. 输入每个考试结果（即 1 或 2）。每次程序要求另一个考试结果时，在屏幕上显示

[①] 在图 3.2 中，如果不指定 `setprecision` 和 `fixed`，C++默认保留 4 位小数。如果只指定 `fixed`，C++保留 6 位小数。

“Enter result”（输入结果）消息。

2. *统计每种考试结果的数量。*
3. *显示测试结果的摘要，说明通过的学生人数和没有通过的人数。*
4. *如果超过 8 名学生通过考试，打印“Bonus to instructor!”（给老师发奖金）。*

3.8.2 实现程序

图 3.3 用计数器控制的循环实现该程序，并显示了两个示例执行的情况。第 8 行~第 10 行和第 16 行声明用于处理考试结果的变量。`while` 语句（第 13 行~第 29 行）循环 10 次。每次循环都输入并处理一个考试结果。注意对每个结果进行处理的 `if...else` 语句（第 20 行~第 25 行）是嵌套在 `while` 语句中的。如果 `result` 为 1，`if...else` 语句会递增 `passes` 变量，否则，它就假设 `result` 为 2[①]，并递增 `failures` 变量。在第 13 行重新测试循环条件之前，第 28 行先完成 `studentCounter` 变量的递增。

[①] 正常编码不要假设 `result` 为 2，因为用户可能输入无效数据。以后会讨论数据校验技术。

```cpp
// fig03_03.cpp
// Analysis of examination results using nested control statements.
#include <iostream>
using namespace std;

int main() {
   // initializing variables in declarations
   int passes{0};
   int failures{0};
   int studentCounter{1};

   // process 10 students using counter-controlled loop
   while (studentCounter <= 10) {
      // prompt user for input and obtain value from user
      cout << "Enter result (1 = pass, 2 = fail): ";
      int result;
      cin >> result;

      // if...else is nested in the while statement
      if (result == 1) {
         passes = passes + 1;
      }
      else {
         failures = failures + 1;
      }

      // increment studentCounter so loop eventually terminates
      studentCounter = studentCounter + 1;
   }

   // termination phase; prepare and display results
   cout << "Passed: " << passes << "\nFailed: " << failures << "\n";

   // determine whether more than 8 students passed
   if (passes > 8) {
      cout << "Bonus to instructor!\n";
   }
}
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

图 3.3 使用嵌套控制语句分析考试结果

输入 10 个值后，循环终止，第 32 行显示通过和没有通过考试的人数（`passes` 和 `failures` 的当前值）。第 35 行~第 37 行的 `if` 语句判断是否有超过 8 名学生通过考试；如果有，就输出消息`"Bonus to instructor!"`。

图 3.3 显示了两次示例执行的输入和输出。第一次，第 35 行的条件为真，即超过 8 名学生通过了考试，所以程序输出给老师发奖金的消息。

3.8.3 用大括号初始化防止收缩转换

11 考虑图 3.3 第 10 行自 C++11 引入的大括号初始化：

```
int studentCounter{1};
```

在 C++11 之前只能这样写：

```
int studentCounter = 1;
```

对于基本类型的变量，大括号初始化防止了可能会造成数据丢失的**收缩转换**[①]（narrowing conversions）。来看看以下声明：

---

① 译注：也称为“窄化转换”。

```
int x = 12.7;
```

它试图将 double 值 12.7 赋给 int 变量 x。如果这样写，C++会将 double 值的浮点部分（.7）截断，将该值转换成一个 int。这就是一种会造成数据丢失的收缩转换。所以，上述语句最终将值 12 赋给 x。对于这种情况，编译器通常会显示一个警告，但仍然允许编译。

但是，如果使用大括号初始化，例如：

```
int x{12.7};
```

就会发生编译错误，帮助你避免一个不容易发现的逻辑错误。即使指定的是整数形式的 double 值，例如 12.0，那么仍然会得到编译错误。初始化器的类型（double）而不是它的值（12.0）决定了是否发生编译错误。C++标准文档没有具体规定错误消息的措辞。对于上述声明，Apple Xcode 编译器报告的错误是：

```
'double'类型不能在初始化列表中收缩为'int'。
```

Visual Studio 报告的错误是：

```
C2397：从“double”转换到“int”需要收缩转换
```

GNU C++报告的错误是：

```
'double'类型不能在初始化列表中收缩为'int' [-Wc++11-narrowing]
```

本书以后会讨论更多的大括号初始化特性。

回顾图 3.1

回头看看图 3.1 的以下语句：

```
int average{total / 10}; // int division yields int result
```

你可能会觉得该语句会发生收缩转换。但是，total 和 10 均为 int，所以整个初始化器的值也是一个 int。相反，如果 total 是 double，或者使用 double 字面值 10.0 作为分母，那么初始化器的值就是 double 类型。在这种情况下，编译器才会因为发生了收缩转换而报错。

3.9 复合赋值操作符

以下语句：

```
c = c + 3;
```

可用**加法复合赋值操作符**（addition compound assignment operator）+=改写成：

```
c += 3;
```

+=操作符将右侧表达式的值与左侧变量的值相加，结果存入左侧的变量中。因此，赋值表达式 c +=3 将 3 加到 c 上，结果存回 c。下表列出了所有算术复合赋值操作符，提供了示例表达式，并解释了操作符的作用。

操作符	示例表达式	解释	赋值结果
假设：int c = 3, d = 5, e = 4, f = 6, g = 12;			
+=	c += 7	c = c + 7	10 赋给 c
-=	d -= 4	d = d - 4	1 赋给 d
*=	e *= 5	e = e * 5	20 赋给 e
/=	f /= 3	f = f / 3	2 赋给 f
%=	g %= 9	g = g % 9	3 赋给 g

3.10 递增和递减操作符

下表总结了 C++的一元递增和递减操作符++和--，它们专门用于数值变量的值加 1 或减 1。

操作符	操作符名称	示例表达式	解释
++	前缀递增	++number	先使 number 递增 1，再在 number 所在的表达式中使用 number 的新值
++	后缀递增	number++	先在 number 所在的表达式中使用 number 的当前值，再使 number 递增 1
++	前缀递减	--number	先使 number 递减 1，再在 number 所在的表达式中使用 number 的新值
++	后缀递减	number--	先在 number 所在的表达式中使用 number 的当前值，再使 number 递减 1

递增或递减操作符的前缀形式分别称为**前缀递增**（prefix increment）或**前缀递减**（prefix decrement）操作符。后缀形式则分别称为**后缀递增**（postfix increment）或**后缀递减**（postfix decrement）操作符。

前缀递增（或递减）操作符会先使一个变量加 1 或减 1，然后在变量所在的表达式中使用

新值。

而后缀递增（或递减）操作符会先在变量所在的表达式中使用变量的当前值，再使变量加 1 或减 1。和二元操作符不同，按照惯例，一元递增和递减操作符应紧靠在操作数旁边，中间不要有空格。

对比前缀递增和后缀递增

图 3.4 演示了++递增操作符的前缀和后缀版本的区别[①]。递减操作符（--）与之类似。

```
1  // fig03_04.cpp
2  // Prefix increment and postfix increment operators.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // demonstrate postfix increment operator
8     int c{5};
9     cout << "c before postincrement: " << c << "\n"; // prints 5
10    cout << "    postincrementing c: " << c++ << "\n"; // prints 5
11    cout << " c after postincrement: " << c << "\n"; // prints 6
12
13    cout << "\n"; // skip a line
14
15    // demonstrate prefix increment operator
16    c = 5;
17    cout << " c before preincrement: " << c << "\n"; // prints 5
18    cout << "     preincrementing c: " << ++c << "\n"; // prints 6
19    cout << "  c after preincrement: " << c << "\n"; // prints 6
20 }
```

```
c before postincrement: 5
    postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
     preincrementing c: 6
  c after preincrement: 6
```

图 3.4 前缀递增和后缀递增操作符

第 8 行将变量 c 初始化为 5，并在第 9 行输出这个初始值。第 10 行输出表达式 c++的值。这个表达式对变量 c 进行的是后递增，所以会先输出 c 的原始值（5），再递增 c（变成 6）。因此，第 10 行输出的是 c 的初始值（5）。第 11 行则输出 c 的新值（6），以证明该变量的值确实在第 10 行递增了。第 16 行将 c 的值重置为 5，并在第 17 行输出该值。第 18 行输

[①] 译注：本书会混用“前缀递增”和“前递增”，以及“后缀递增”和“后递增”。递减版本同样如此。

出表达式++c 的值。这个表达式对变量 c 进行的是前递增，所以会先递增它的值，再输出这个新值（6）。第 19 行再次输出 c 的值，证明在执行了第 18 行后，c 的值仍然是 6。

用算术复式赋值、递增和递减操作符简化语句

可以使用算术复合赋值操作符以及递减和递减操作符来简化程序语句。以图 3.3 的三个赋值语句（第 21 行、第 24 行和第 28 行）为例：

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

它们可用复合赋值操作符简化为：

```
passes += 1;
failures += 1;
studentCounter += 1;
```

甚至可以用前缀递增操作符进一步简化为：

```
++passes;
++failures;
++studentCounter;
```

或者使用后缀版本：

```
passes++;
failures++;
studentCounter++;
```

如果变量递增或递减自成一个语句，那么前缀和后缀版本的效果一样（无论递增还是递减）。只有当变量用在一个更大的表达式中时，前缀和后缀版本才有不同的效果。

错误

试图在可以赋值的表达式以外的其他表达式上使用递增或递减操作符会造成编译错误。例如，++(x + 1)会报告一个语法错误，因为(x + 1)不是可修改的变量。

操作符优先级和分组

下表总结了到目前为止介绍的操作符的优先级和分组。操作符从上到下按优先级递减的顺序排列。第二列表示每一级操作符的分组情况（关联性）。注意，只有条件操作符(?:)、一元前递增(++)、前递减(--)、加(+)、减(-)操作符以及赋值操作符=、+=、 -=、*=、 /=和%=从右到左分组。这个表格中的其他所有操作符均是从左到右分组。

操作符	分组(关联性)

++ -- static_cast<*type*>()	从左到右
++ -- + -	从右到左
* / %	从左到右
+ -	从左到右
<< >>	从左到右
< <= > >=	从左到右
== !=	从左到右
?:	从右到左
= += -= *= /= %=	从右到左

3.11 基本类型不可移植

C++基本类型及其典型取值范围的完整列表可参考以下网页：

https://zh.cppreference.com/w/cpp/language/types

在 C 和 C++中，一个 int 在一台机器上可能使用 16 位（2 字节）来表示，在另一台机器上可能使用 32 位（4 字节），在第三台机器上则可能使用 64 位（8 字节）。由于这个原因，使用整数的代码并非肯定能保证跨平台移植。可以写多个版本的程序，在不同平台上使用不同的整数类型。还可以使用一些技术来实现不同程度的可移植性[①]。下一节将展示一种实现可移植性的方法。

C++的整数类型包括 int、long 和 long long。C++标准要求 int 类型至少为 16 位，long 类型至少为 32 位，long long 类型至少为 64 位。标准还要求 int 要小于或等于 long 的大

[①] 头文件<cstdint>中的整数类型（https://zh.cppreference.com/w/cpp/types/integer）可用来保证整数变量在不同平台上都有正确的大小。

小，long 要小于或等于 long long 的大小。这种“软弱”的要求带来了可移植性的挑战，但允许编译器的实现者通过将基本类型的大小与机器的硬件相匹配来优化性能。

3.12 对象自然案例研究：任意大小的整数

一个整数类型的取值范围取决于在特定计算机上用来表示该类型的字节数。例如，一个 4 字节的 int 可以在–2 147 483 648 到 2 147 483 647 的范围内存储2^{32}个可能的值。在大多数系统中，一个 long long 整数占用 8 个字节，可以在–9 223 372 036 854 775 808 到 9 223 372 036 854 775 807 的范围内存储2^{64}个可能的值。

一些应用需要 long long 范围以外的数字

现在考虑一下阶乘计算。阶乘是指从 1 到一个给定值的整数的乘积。5 的阶乘（写成 5!）是 1*2*3*4*5，结果是 120。我们用 64 位 long long 整数表示的最高阶乘值是 20!，也就是 2 432 902 008 176 640 000。阶乘值迅速增长，很快就会超出 long long 整数所能表示的范围。随着要处理的数据迅速变大，越来越多的现实世界的应用将超过 long long 整数的限制。

安全性

另一个需要巨大整数的应用是密码学，它是确保计算机通过互联网安全传输数据的关键。许多密码学算法都使用 128 位或 256 位整数值进行计算，这远远大于我们用 C++的基本类型所能表示的。

使用 BigNumber 类来实现任意精度的整数

如果需要超出 long long 范围的整数，任何应用都需要进行特殊的处理。不幸的是，C++标准库（还）没有为任意精度的整数提供一个类。所以，本例将深入广阔的开源类库世界，展示众多 C++类中的一个，你可以用它创建和处理任意精度的整数。这里使用的是 BigNumber 类，它来自：

https://github.com/limeoats/BigNumber

为方便起见，我们已将其下载到 examples 文件夹的 library 文件夹中。请务必阅读 LICENSE.md 文件中的开源许可条款。

BigNumber 类可以直接使用，不必了解它具体如何实现[1]。包含它的头文件（bignumber.h），

[1] 对 C++有了深入的理解之后，可考虑看看 BigNumber 的源代码（大约 1000 行），了解它具体是如何实现的。本书后面会讨论面向对象编程，你可以利用其中讲到的各种技术实现你自己的大整数类。

即可轻松创建该类的对象，并在代码中使用这些对象。图 3.5 的程序展示了如何使用 BigNumber 类，并提供了一个示例输出。本例将使用最大的 long long 整数值来证明可以用 BigNumber 创建一个更大的整数。本节最后会演示如何用自己喜欢的编译器来编译和运行程序。

```
// fig03_05.cpp
// Integer ranges and arbitrary-precision integers.
#include <iostream>
#include "bignumber.h"
using namespace std;

int main() {
   // use the maximum long long fundamental type value in calculations
   const long long value1{9'223'372'036'854'775'807LL}; // long long max
   cout << "long long value1: " << value1
      << "\nvalue1 - 1 = " << value1 - 1 // OK
      << "\nvalue1 + 1 = " << value1 + 1; // result is undefined

   // use an arbitrary-precision integer
   const BigNumber value2{value1};
   cout << "\n\nBigNumber value2: " << value2
      << "\nvalue2 - 1 = " << value2 - 1 // OK
      << "\nvalue2 + 1 = " << value2 + 1; // OK

   // powers of 100,000,000 with long long
   long long value3{100'000'000};
   cout << "\n\nvalue3: " << value3;

   int counter{2};

   while (counter <= 5) {
      value3 *= 100'000'000; // quickly exceeds maximum long long value
      cout << "\nvalue3 to the power " << counter << ": " << value3;
      ++counter;
   }

   // powers of 100,000,000 with BigNumber
   BigNumber value4{100'000'000};
   cout << "\n\nvalue4: " << value4 << "\n";

   counter = 2;

   while (counter <= 5) {
      cout << "value4.pow(" << counter << "): "
         << value4.pow(counter) << "\n";
      ++counter;
   }

   cout << "\n";
}
```

```
long long value1: 9223372036854775807
value1 - 1: 9223372036854775806                    OK
value1 + 1: -9223372036854775808                   不正确的结果

BigNumber value2: 9223372036854775807
value2 - 1: 9223372036854775806                    OK
value2 + 1: 9223372036854775808                    OK
```

```
value3: 100000000
value3 to the power 2: 10000000000000000                    OK
value3 to the power 3: 2003764205206896640                  不正确的结果
value3 to the power 4: -8814407033341083648                 不正确的结果
value3 to the power 5: -5047021154770878464                 不正确的结果

value4: 100000000
value4.pow(2): 10000000000000000                            OK
value4.pow(3): 1000000000000000000000000                    OK
value4.pow(4): 100000000000000000000000000000000            OK
value4.pow(5): 10000000000000000000000000000000000000000    OK
```

图 3.5 整数范围和任意精度的整数

包含不在 C++标准库中的头文件

如第 4 行所示，在#include 指令中，和应用程序同一文件夹或某个子文件夹中的头文件通常被放在双引号（""）中，而不放在尖括号（<>）中。双引号告诉编译器，这个头文件在当前应用程序所在的文件夹或者由你指定的另一个文件夹中。

超出 long long 整数的取值范围会发生什么？

第 9 行将 value1 变量初始化为系统支持的最大 long long 值[①]：

```
long long value1{9'223'372'036'854'775'807LL}; // max long long value
```

位数过多的数字字面值在输入时容易出错。为了增强这种字面值的可读性，并减少打字错
14 误，C++14 引入了数字分隔符'（单引号字符）。可以在数值字面值的各个分组之间插入该分隔符，这里是用它分隔三个数字的分组。另外，请注意字面值末尾的 LL，它将该字面值指定为 long long 整数。

第 10 行显示 value1，第 11 行从中减 1，以演示一个有效的计算。接着，第 12 行在 value1 上加 1，变量此时已包含了最大的 long long 值。我们测试的所有编译器显示的结果都是最小的 long long 值。C++标准指出，这种计算是“未定义”的行为。未定义的行为在不同系统之间可能会有所区别，我们测试的系统会显示一个不正确的值，但其他系统可能终止程序并显示一条错误信息。这进一步证明了为什么基本整数类型是不可移植的。

[①] 在我们对本书示例代码进行测试的所有平台上，最大 long long 整数值都是 9 223 372 036 854 775 807。可在自己的程序中使用表达式 std::numeric_limits<long long>::max()来确定该值，它使用了 C++标准库头文件<limits>中的 numeric_limits 类。这个表达式中使用的<>和::符号将在以后的章节中介绍。经过上述实验后，我们在图 3.5 的程序中使用了字面值 9 223 372 036 854 775 807。

用 BigNumber 对象执行同样的操作

第 15 行~第 18 行使用一个 BigNumber 对象来重复第 9 行~第 12 行的操作。我们创建了一个名为 value2 的 BigNumber 对象，并用 value1 对其进行初始化，该对象包含 long long 整数的最大值。

```
BigNumber value2{value1};
```

接着，我们显示 BigNumber 对象，然后从中减 1 并显示结果。第 18 行在 value2 上加 1。此时，value2 包含的是 long long 整数的最大值。BigNumber 能处理任意精度的整数，所以它正确执行了这个计算。其结果是使用 C++基本整数类型在我们的系统上无法处理的一个值。

BigNumber 支持所有典型的算术运算，包括本程序使用的+和-。编译器知道如何为基本数值类型使用算术操作符，但你必须教会它如何将这些操作符应用于类的对象。我们将在第 11 章讨论这个技术，即所谓的操作符重载。

用 long long 整数计算 100 000 000 的乘方

第 21 行~第 30 行使用 long long 整数计算 100 000 000 的乘方。首先创建 value3 变量并显示它的值。第 26~第 30 行循环 4 次。以下计算：

```
value3 *= 100'000'000; // quickly exceeds maximum long long value
```

会将 value3 的当前值乘以 100 000 000，从而获得 value3 的下一个乘方值。如程序的输出所示，只有循环的第一次迭代生成的结果才是正确的。

用 BigNumber 对象计算 100 000 000 的乘方

为了证明 BigNumber 能处理比基本整数类型大得多的值，第 33 行~第 42 行计算了 100 000 000 的乘方。首先创建 BigNumber 对象 value4 并显示其初始值。第 38 行~第 42 行循环 4 次。以下计算：

```
value4.pow(counter)
```

会调用 BigNumber 的成员函数 pow 来计算 value4 的 counter 次方[①]。BigNumber 能正确处理每个计算，生成的整数远超出各种操作系统的基本整数类型的支持范围。

虽然可用 BigNumber 表示任意整数值，但由于它超出了系统硬件的支持范围，所以必然是

[①] 这里选择演示 BigNumber 的 pow 成员函数，但也可以像第 27 行的 long long 变量那样使用*=。

以牺牲性能的方式来换取 `BigNumber` 所提供的灵活性。

在 Microsoft Visual Studio 中编译和运行示例

在 Microsoft Visual Studio 中：

1. 按照 1.2 节的指示创建一个新的空项目。
2. 在解决方案资源管理器中，右击项目的“源文件”文件夹，选择“添加”|“现有项”。
3. 切换到 fig03_05 文件夹，选择 fig03_05.cpp，单击“添加”。
4. 重复步骤 2~步骤 3，添加来自 examples\libraries\BigNumber\src 文件夹的 bignumber.cpp。
5. 在解决方案资源管理器中右击项目名称，选择“属性”。
6. 在“配置属性”下选定“C/C++”。在对话框右侧，为“附加包含目录”添加 BigNumber\src 在你的系统上的完整路径。在我们的系统上，这个路径是：

   ```
   C:\Users\你的用户名\Documents\examples\libraries\BigNumber\src
   ```

7. 单击“确定”。
8. 按 Ctrl+F5 编译并运行程序[1]。

在 GNU g++中编译和运行示例

对于 GNU g++(在 macOS 的一个“终端”窗口运行时也可按这些指示操作)：

1. 在命令行中，切换到本例的 fig03_05 文件夹。
2. 用以下命令编译程序。其中，`-I` 选项指定编译器在查找头文件时的额外文件夹（所有内容在一行输入）：

   ```
   g++ -std=c++2a -I ../../libraries/BigNumber/src fig03_05.cpp
      ../../libraries/BigNumber/src/bignumber.cpp -o fig03_05
   ```

3. 输入以下命令来执行程序

   ```
   ./fig03_05
   ```

在 Apple Xcode 中编译和运行示例

在 Apple Xcode 中：

1. 按照 1.2 节的指示创建一个新项目，删除 main.cpp。

---

[1] Visual C++可能为 `BigNumber` 类显示警告消息。

2. 将 fig03_05.cpp 文件从“访达”（Finder）中的 fig03_05 文件夹拖放到 Xcode 中的项目源代码文件夹。在出现的对话框中单击“完成”。
3. 采用类似的操作，将 bignumber.h 和 bignumber.cpp 从 examples/libraries/ BigNumber/src 文件夹拖放到 Xcode 中的源代码文件夹，同样单击“完成”。
4. 按⌘+R 编译并运行程序。

3.13 C++20：用 format 函数格式化文本

C++20 通过 format 函数引入了强大的、新的字符串格式化功能（需包含<format>头文件），
20 它使用和 Python、Microsoft .NET 语言（例如 C#和 Visual Basic）以及较新的 Rust 语言相似的格式化语法大幅简化了 C++的格式化功能[①]。C++20 的文本格式化功能比早期 C++版本更加简洁和强大。本书以后将主要使用这些新的文本格式化功能。考虑到一些遗留软件仍在使用，所以本书网上的第 19 章介绍了旧的格式化功能的细节。

C++20 字符串格式化尚未全面实现

本书写作时，仅 Visual C++实现了 C++20 新的文本格式化功能。https://github.com/fmtlib/fmt 提供了一个开源的{fmt}库，它完整了实现了新特性。[②③]在 C++20 编译器全面实现文本格式化之前，我们将使用这个库[④]。为方便起见，我们在 examples 文件夹的 library 子文件夹中包含了已经下载好的完整的{fmt}库。详情请参见例子之后的编译步骤。该库的许可条款请参见：

https://github.com/fmtlib/fmt/blob/master/LICENSE.rst

[①] Victor Zverovich，“Text Formatting”，July 16, 2019，http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html。

[②] 根据 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html，它是 C++标准委员会的 C++20 文本格式化提案。

[③] C++20 的文本格式化特性是{fmt}库提供的特性的一个子集。

[④] 我们的一些 C++20 特性模拟（Feature Mock-Up）小节展示了不能编译或运行的代码。一旦编译器实现了这些特性，我们就会重新测试代码，更新我们的数字产品，并在 https://deitel.com/c-plus-plus-20-for-programmers 发布针对纸质书的更新。本例的代码可以运行，但使用了{fmt}开源库来演示 C++20 编译器最终会支持的功能。

.

格式字符串占位符

format 函数的第一个实参是**格式字符串**（format string），包含以大括号（{和}）分隔的占位符。如图 3.6 所示，函数会用其他实参的值来替换占位符。

```
1  // fig03_06.cpp
2  // C++20 string formatting.
3  #include <iostream>
4  #include <fmt/format.h> // C++20: This will be #include <format>
5  using namespace std;
6  using namespace fmt; // not needed in C++20
7
8  int main() {
9     string student{"Paul"};
10    int grade{87};
11
12    cout << format("{}'s grade is {}\n", student, grade);
13 }
```

```
Paul's grade is 87
```

图 3.6 C++20 字符串格式化

点位符从左到右替换

format 函数默认从左到右替换其格式字符串实参中的占位符。所以，对于以下格式字符串：

```
"{}'s grade is {}"
```

第 12 行的 format 调用会在第一个占位符中插入 student 的值（"Paul"），在第二个占位符中插入 grade 的值（87），然后返回以下完整字符串：

```
"Paul's grade is 87"
```

在 Microsoft Visual Studio 中编译和运行示例

步骤和 3.13 节一样，只是进行了以下修改：

- 在步骤 3，切换到 fig03_06 文件夹，将 fig03_06.cpp 添加到项目的“源文件”文件夹。
- 在步骤 4，切换到 examples\libraries\fmt\src 文件夹，将 format.cc 添加到项目的“源文件”文件夹。
- 在步骤 6，为“附加包含目录”添加{fmt}库的 include 文件夹的完整路径。在我们的系统上，这个路径是：

  C:\Users*你的用户名*\Documents\examples\libraries\fmt\include

在 GNU g++中编译和运行示例

对于 GNU g++(在 macOS 的一个“终端”窗口运行时也可按这些指示操作)：

1. 在命令行中，切换到本例的 fig03_06 文件夹。
2. 用以下命令编译程序，注意所有内容在一行输入：

```
g++ -std=c++2a -I ../../libraries/fmt/include fig03_06.cpp
   ../../libraries/fmt/src/format.cc -o fig03_06
```

3. 输入以下命令来执行程序

```
./fig03_06
```

在 Apple Xcode 中编译和运行示例

步骤和 3.12 节一样，只需进行一处修改。在步骤 2 中，改为将文件 fig03_06.cpp、format.cc 和文件夹 fmt 从“访达”（Finder）中的 fig03_06 文件夹拖放到 Xcode 中的项目源代码文件夹。

3.14 小结

开发任何算法都只需要三种类型的控制语句：顺序、选择和循环。本章介绍了 `if` 单选语句、`if...else` 双选语句和 `while` 循环语句。我们使用计数器和哨兵值控制的循环，通过控制语句的堆叠（连接）来汇总和计算一组学生成绩的平均值。然后，我们使用嵌套控制语句来分析一组考试结果并做出决策。我们还介绍了复合赋值操作符以及递增和递减操作符。

我们讨论了为什么 C++的基本类型不能保证在不同平台上具有相同的大小，并使用开源类 `BigNumber` 的对象来执行整数运算，它支持超出系统所支持范围的大数。最后，我们通过开源`{fmt}`库介绍了 C++20 新的文本格式化功能。一旦你喜欢的编译器开始完全支持 C++20 的`<format>`头，就不再需要单独的`{fmt}`库了。第 4 章将继续讨论控制语句，会介绍 `for`、`do...while` 和 `switch` 语句。还会介绍用于创建复合条件的逻辑操作符。

第 4 章 控制语句(第 2 部分)

学习目标：

- 使用 `for` 和 `do...while` 循环语句。
- 使用 `switch` 选择语句进行多选。
- 在 `switch` 语句中使用 C++17 的`[[fallthrough]]`特性。
- 使用带初始化器的 C++17 选择语句。
- 使用 `break` 和 `continue` 语句改变控制流。
- 使用逻辑操作符在控制语句中构建复合条件。
- 理解使用浮点数据保存币值时可能出现的表示误差。
- 使用一个开源的 ZIP 压缩/解压缩库来创建和读取 ZIP 文件，继续我们的“对象自然”案例研究。
- 使用更多的 C++20 文本格式化功能。

4.1 导读

本章介绍了 `for`、`do...while`、`switch`、`break` 和 `continue` 控制语句。我们探讨了由计数器控制的循环的基本原理。我们通过复利计算开始研究货币金额的处理问题，首先探讨的是和浮点类型有关的表示误差。我们使用 `switch` 语句针对一组数值成绩来判断对应的 A、B、C、D 和 F 成绩数量。我们展示了 C++17 的增强功能，它允许在 `if` 和 `switch` 语句的头部初始化一个或多个同类型的变量。我们讨论了逻辑操作符，它们允许合并简单的条件以形成复合条件。我们的“对象自然”案例研究将继续使用现成类的对象，使用 miniz-cpp 开源库来创建和读取 ZIP 压缩文件。最后，我们介绍了更多 C++20 强大而富有表现力的文本格式化功能。

4.2 计数器控制的循环的本质

本节使用第 3 章介绍的 `while` 循环语句来正式说明计数器控制的循环的构成元素：

1. 一个控制变量（或循环计数器）
2. 控制变量的初始值
3. 在循环的每次迭代中应用的控制变量的增量
4. 决定循环是否应该继续的循环继续条件。

图 4.1 用一个循环来显示从 1 到 10 的数字。

```cpp
// fig04_01.cpp
// Counter-controlled iteration with the while iteration statement.
#include <iostream>
using namespace std;

int main() {
   int counter{1}; // declare and initialize control variable

   while (counter <= 10) { // loop-continuation condition
      cout << counter << "  ";
      ++counter; // increment control variable
   }

   cout << "\n";
}
```

```
1  2  3  4  5  6  7  8  9  10
```

图 4.1 计数器控制的 while 循环

在图 4.1 中，第 7 行、第 9 行和第 11 行定义了计数器控制的循环的各个元素。第 7 行声明 int 类型的控制变量 counter，在内存中为其保留空间，并将其初始值设为 1。我们用可执行语句来声明需要初始化的变量，这称为变量的**定义**（definitions）。但是，我们通常使用**声明**（declaration）这个术语，除非它和“定义”有重要的区别。

第 10 行在循环的每次迭代中显示一次 counter 的值。第 11 行在循环的每次迭代中使控制变量递增 1。while 的循环继续条件（第 9 行）测试控制变量的值是否小于或等于 10（10 是条件为真时的最后一个值）。一旦控制变量超过 10，循环终止。

由于浮点变量只能存储近似值，所以用浮点变量控制计数循环可能造成不精确的计数器值和不准确的终止测试，结果是造成循环无法正常终止。有鉴于此，总是用整数变量控制计数循环。

4.3 for 循环语句

for 循环语句用一行代码指定计数器控制的循环的所有细节。图 4.2 用 for 语句重新实现了图 4.1 的程序。

```
1  // fig04_02.cpp
2  // Counter-controlled iteration with the for iteration statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // for statement header includes initialization,
8     // loop-continuation condition and increment
9     for (int counter{1}; counter <= 10; ++counter) {
10       cout << counter << "  ";
11    }
12
13    cout << "\n";
14 }
```

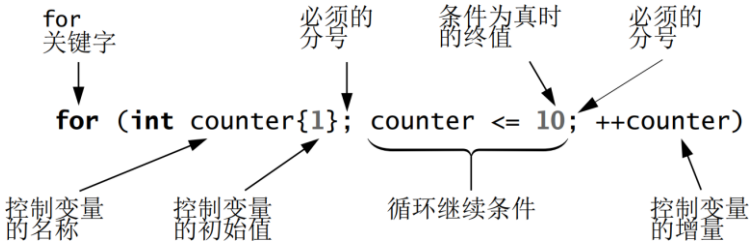
```
1  2  3  4  5  6  7  8  9  10
```

图 4.2 计数器控制的 for 循环

for 语句（第 9 行~第 11 行）开始执行时，先声明控制变量 counter 被把它初始化为 1。接着，程序测试两个分号之间的循环继续条件（counter <= 10）。因为 counter 的初始值是 1，所以该条件为真。所以，第 10 行显示 counter 的值（1）。执行第 10 行后，第二个分号右侧的++counter 使 counter 递增。然后，程序再次测试循环继续条件，判断是否应继续下一次迭代。这时，counter 的值是 2，条件仍然为真，所以程序再次执行第 10 行。这个过程一直持续到循环显示 1~10 的所有数字，此时 counter 的值变成 11，所以对循环继续条件的测试失败，循环终止，程序继续执行 for 循环后的第一个语句（第 13 行）。

剖析 for 语句头

下图详细剖析了图 4.2 的 for 语句头。



第一行——包括关键字 for 和之后圆括号内的一切（图 4.2 的第 9 行）——有时称为 for 语句头（header）。for 语句头承包了这种循环的全部控制工作，它用一个控制变量指定了计数器控制的循环所需的每个元素。

for 语句的常规格式

for 语句的常规格式如下所示：

```
for (初始化; 循环继续条件; 递增) {
    语句
}
```

其中：

- “初始化”命名循环的控制变量并提供其初始值。
- 两个必须的分号之间的“循环继续条件”决定循环是否应该继续。
- “递增”修改控制变量的值，使循环持续条件最终变为假。

如果循环继续条件一开始就为假，程序不会执行 for 语句的主体，而是直接执行 for 语句之后的语句。

for 语句的控制变量的作用域

如果“初始化表达式”声明了控制变量，那么该变量只能在该 for 语句内部使用，不能超出该语句的范围。这种限制性的范围有一个专业称呼，即变量的**作用域**（scope），它定义了变量的寿命和它在程序中可以使用的地方。例如，变量的作用域是从它的声明位置开始，一直到代码块的结束右大括号。第 5 章会讲到，将变量限制在所需的最小作用域是一个好的做法。

for 语句头中的表达式是可选的

for 头中的三个表达式全都是可选的。如果省略“循环继续条件”，那么该条件会一直为真，从而形成一个无限循环。如果程序在循环之前就初始化好了控制变量，那么可以省略“初始化表达式”。如果程序在循环主体中递增控制变量，或者不需要递增，那么可以省略“递增”表达式。

for 的递增表达式相当于放在 for 主体末尾的一个独立的语句。所以，以下递增表达式：

```
counter = counter + 1
counter += 1
++counter
counter++
```

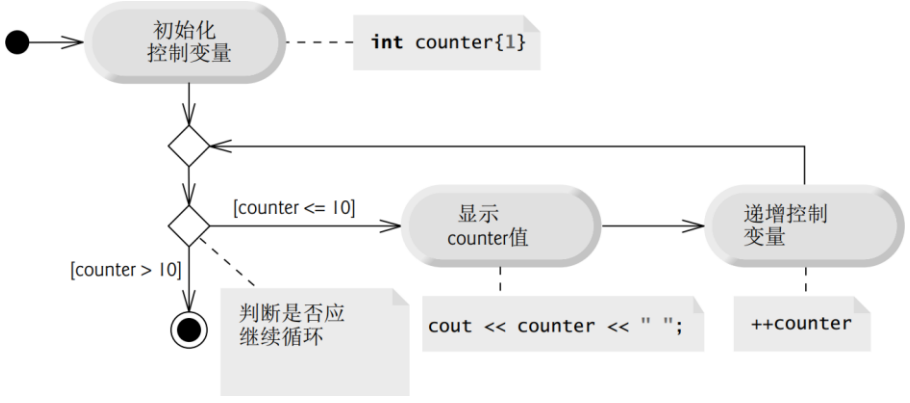
这些 for 语句是完全等价的。在本例中，由于递增表达式不会出现在一个更大的表达式中，所以前递增和后递增的效果一样。我们更喜欢前递增。第 11 章在讨论操作符重载时，会讲到前递增能提供性能上的优势。

在语句的主体中使用 for 语句的控制变量

程序经常显示控制变量的值，或者在循环主体的计算中使用它，但这并不是必要的。虽然能在 for 循环的主体内部更改控制变量的值，但这样做可能造成不容易发现的错误。如果必须在循环主体内部修改控制变量的值，最好使用 while 而不是 for。

for 语句的 UML 活动图

下面是图 4.2 的 for 语句的 UML 活动图，它清楚地表明，初始化仅在首次测试条件前发生一次，而递增在执行了主体语句之后发生。



4.4 for 循环的例子

下面的例子展示了在 for 语句中更改控制变量的技术。每个例子只提供相应的 for 头。在控制变量递减的循环中，要注意关系操作符的变化。

a) 将控制变量从 1 变到 100，每次递增 1。

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

b) 将控制变量从 100 变到 1，每次递减 1。

```
for (int i{100}; i >= 1; --i)
```

c) 将控制变量从 7 变到 77，每次递增 7。

```
for (int i{7}; i <= 77; i += 7)
```

d) 将控制变量从 20 变到 2，每次递减 2。

```
for (int i{20}; i >= 2; i -= 2)
```

e) 使控制变量依次在 2，5，8，11，14，17，20 之间变化。

```
for (int i{2}; i <= 20; i += 3)
```

f) 使控制变量依次在 99，88，77，66，55，44，33，22，11，0 之间变化。

```
for (int i{99}; i >= 0; i -= 11)
```

如果循环控制变量的增量或减量超过 1，就不要在循环继续条件中使用相等性操作符（!=或==）。例如在以下 for 语句头中：

```
for (int counter{1}; counter != 10; counter += 2)
```

counter != 10 永远都不会变成假（从而造成无限循环），因为每次迭代后 counter 会递增 2，所以只会生成奇数值：3，5，7，9，11，…… 。

4.5 应用程序：累加偶数整数

图 4.3 的应用程序使用 for 语句对 2~20 的偶数整数进行求和，并将结果存储到 int 变量 total 中。循环的每一次迭代（第 10 行~第 12 行）都将控制变量 number 的值加到 total 变量上。

```
1  // fig04_03.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int total{0};
8
9     // total even integers from 2 through 20
10    for (int number{2}; number <= 20; number += 2) {
11       total += number;
12    }
13
14    cout << "Sum is " << total << "\n";
15 }
```

```
Sum is 110
```

图 4.3 用 for 语句累加偶数整数

for 语句的初始化和递增表达式可以是以逗号分隔的列表，其中包含多个初始化表达式或多个递增表达式。虽然不鼓励这样做，但确实可以使用逗号操作符将 for 语句的主体（第 11 行）合并到 for 头的“递增”部分。

```
for (int number{2}; number <= 20; total += number, number += 2) { }
```

表达式 total += number 和 number += 2 之间的逗号是**逗号操作符**（comma operator），它

保证表达式列表从左到右求值。在所有 C++操作符中，逗号操作符的优先级最低。对于一个以逗号分隔的表达式列表，其值和类型分别是最右边的表达式的值和类型。需要多个初始化表达式或多个递增表达式的 for 语句经常使用逗号操作符。

4.6 应用程序：复利计算

本节用 for 语句计算复利，以下是问题陈述：

一个人将 1000 美元投资在一个年利率为 5%的储蓄账户上。假设所有利息都留存，计算并打印 10 年内每年年底的账户余额。使用以下公式进行计算：

$$a = p\,(1 + r)^n$$

其中：

- *p* 是初始投资金额（即本金）
- *r* 是年利率（例如，用 0.05 表示 5%）。
- *n* 是年数
- *a* 是第 *n* 年年底的余额

解决方案（图 4.4）使用一个循环计算 10 年内每一年的余额。我们先用 double 值进行货币计算。然后，将讨论使用浮点类型表示货币金额的问题。对于需要精确货币计算和舍入控制的金融应用，可以考虑使用像 Boost.Multiprecision 这样的开源库[①]。

第 12 行~第 13 行将 double 变量 principal（本金）初始化为 1000.00，将 double 变量 rate（利率）初始化为 0.05。C++默认将 1000.00 和 0.05 这样的浮点字面值视为 double 类型。类似地，默认将 7 和-22 这样的整数视为 int 类型[②]。第 15 行~第 16 行显示初始本金和利率。

[①] John Maddock 和 Christopher Kormanyos，“Chapter 1. Boost.Multiprecision”，https://www.boost.org/doc/libs/master/libs/multiprecision/doc/html/index.html。

[②] 3.12 节讲过，C++的整数类型并不能表示所有整数值。要根据希望表示的值的范围选择恰当的类型。另外，松在整数字面值后面附加 L 或 LL 来分别指定 long 类型或 long long 类型。

```
1  // fig04_04.cpp
2  // Compound-interest calculations with for.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath> // for pow function
6  using namespace std;
7
8  int main() {
9     // set floating-point number format
10    cout << fixed << setprecision(2);
11
12    double principal{1000.00}; // initial amount before interest
13    double rate{0.05}; // interest rate
14
15    cout << "Initial principal: " << principal << "\n";
16    cout << "    Interest rate:    " << rate << "\n";
17
18    // display headers
19    cout << "\nYear" << setw(20) << "Amount on deposit" << "\n";
20
21    // calculate amount on deposit for each of ten years
22    for (int year{1}; year <= 10; ++year) {
23       // calculate amount on deposit at the end of the specified year
24       double amount{principal * pow(1.0 + rate, year)} ;
25
26       // display the year and the amount
27       cout << setw(4) << year << setw(20) << amount << "\n";
28    }
29 }
```

```
Initial principal: 1000.00
    Interest rate:    0.05

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

图 4.4 用 for 进行复利计算

用域宽和对齐来格式化

年份和金额的打印是循环之前的第 10 行和循环内部的第 27 行共同控制的。我们用参数化流操纵元 setprecision 和 setw 以及非参数化流操纵元 fixed 来格式化输出。其中，流操纵元 setw(4)指定下个输出值应该在一个 4 字符宽度的**域**（或称**字段**）内打印。换言之，

cout <<打印的值至少要占 4 个字符的宽度。如果输出的值需要的位置少于 4 个字符，该值在域内默认右对齐。如果要输出的值需要占用 4 个以上的字符位置，C++将域宽向右扩展以装下整个值。如果不想默认的右对齐，而是想左对齐，可以输出一个非参数化的流操纵元left（位于头文件<iostream>中）。要恢复右对齐，输出一个非参数化的流操纵元right即可。

输出语句中的其他格式化设置将amount变量显示为带小数点的定点值（第10行的fixed）[1]，在 20 个字符位置的域内右对齐（第 27 行的 setw(20)），两位小数精度（第 10 行的setprecision(2)）。我们在 for 循环之前对输出流 cout 应用了粘性的流操纵元 fixed 和setprecision，这些格式化设置在被更改之前将一直有效（所谓的粘性），而且不需要在每次循环迭代时重复应用。但是，用 setw 指定的域宽只适用于下一个输出值。本书网上的第 19 章会详细讨论 cin 的和 cout 的格式化功能。4.14 节会继续讨论 C++20 新的、强大的文本格式化功能。

用标准库函数 pow 计算利息

for 语句（第 22 行~第 28 行）总共迭代 10 次，int 类型的控制变量 year 从 1 变到 10，每次递增 1，它代表问题陈述中的 n。

C++没有内建幂操作符，我们使用的是来自头文件<cmath>（第 5 行）的标准库函数 pow（第 24 行）。pow(x, y)计算 x 的 y 次方的结果。该函数接收两个 double 实参并返回一个 double 结果。第 24 行实际计算$a = p\,(1 + r)^n$，其中 a 对应 amount，p 对应 principal，r 对应 rate，n 对应 year。

pow 函数的第一个实参是 1.0 + rate。注意，该表达式在循环的每次迭代中都会产生相同的结果，所以每次都重复计算纯属浪费。为了提升程序性能，如今许多优化编译器都会在编译后的代码中把这种计算放到循环之前。

性能

浮点数精度和内存需求

float 值表示的是一个**单精度浮点数**（single-precision floating-point number），大多数系统都用 4 个字节来存储它，其中约有 7 个有效数字（数位）。double 值表示的是一个**双精度浮点数**（double-precision floating-point number），大多数系统都用 8 个字节来存储它，其中约有 15 个有效数字——约为 float 精度的两倍。大多数程序员都会使用 double 类型。C++将程序源代码中的浮点数（例如 3.14159）默认看成是 double 值。源代码中的这种（直接写出来的）值称为**浮点字面值**（floating-point literals）。

---

[1] 3.7.3 节说过，fixed 导致输出符合我们平时习惯的、带小数点的浮点数，而不是使用科学计数法。

C++标准没有规定各种浮点类型需要用多大的内存空间来存储，只规定 double 类型提供至少和 float 一样的精度。另外还有一种 long double 类型，它要求提供至少和 double 一样的精度。要获得 C++基本类型的完整列表及其典型范围，请参考：

https://zh.cppreference.com/w/cpp/language/types

浮点数是近似值

在传统算术中，浮点数往往是由除法产生的。10 除以 3，结果是 3.3333333...，3 的序列会无限重复。但计算机不一样，它是分配固定大小的空间来容纳这种值，所以存储的只能是一个近似值。像 double 这样的浮点类型存在着所谓的**表示误差**（representational error）。如果想当然地以为浮点数表示的是准确的值（例如，在相等性比较中使用），就会导致错误的结果。

浮点数有很多应用，尤其是在测量值方面。例如，平时我们说一个人的体温为摄氏 36.5 度时，并不需要精确到小数点后面很多位。温度计显示 36.5 时，它实际可能是 36.494732103。对于大多数体温计算来说，称这个数字为 36.5 是可以接受的。一般来说，double 比 float 更受欢迎，因为 double 能更精确地表示浮点数[①]。

关于显示舍入值的警告

本例将 amount、principal 和 rate 都声明为 double 类型。遗憾的是，浮点数会为货币金额的小数点部分带来麻烦。下面简单解释了用浮点表示两位小数的货币金额时会出现什么问题。假定机器中存储的两个计算好的金额是 14.234（出于显示的目的而四舍五入为 14.23）和 18.673（出于显示的目的而四舍五入为 18.67）。两个金额相加，它们内部的结果应该是 32.907。但是，出于显示的目的，通常会被四舍五入为 32.91。因此，你的输出是这样显示的：

```
  14.23
+ 18.67
  32.91
```

但是，如果一个人把上面显示的各个数字相加，得到的总和将是 32.90。别说我们没有警告你！

[①] 当前的浮点数算术标准是 IEEE 754（https://zh.wikipedia.org/wiki/IEEE_754）。

即使是普通的美元金额也会出现浮点表示误差

即使是简单的货币金额，把它们作为 double 值存储时，也可能出现表示误差。为了证明这一点，我们创建一个简单的程序并这样定义变量 d：

```
double d{123.02};
```

然后，以 20 位小数的精度来显示 d 的值，结果会将 123.02 显示为 123.0199999...，这是另一个表示误差的例子。虽然一些货币数额能精确地表示为 double 值，但许多都不能。这是许多编程语言常见的问题。本书以后会创建并使用能精确处理货币金额的类。

4.7 do…while 循环语句

while 循环语句在执行主体之前会先测试循环继续条件。如果为假，主体一次都不会执行。相反，do...while 循环语句先执行一次循环主体，然后才测试循环继续条件。所以，主体至少会执行一次。图 4.5 使用 do...while 来输出数字 1~10。第 7 行声明并初始化控制变量 counter。进入 do...while 语句后，第 10 行输出 counter 的值，第 11 行递增 counter。然后，程序在循环的底部（第 12 行）对循环继续条件进行求值。如果结果为真，循环从主体的第一个语句（第 10 行）继续。如果结果为假，则循环终止，程序从循环后的下一个语句继续。

```
1  // fig04_05.cpp
2  // do...while iteration statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int counter{1};
8
9     do {
10       cout << counter << "  ";
11       ++counter;
12    } while (counter <= 10); // end do...while
13
14    cout << "\n";
15 }
```

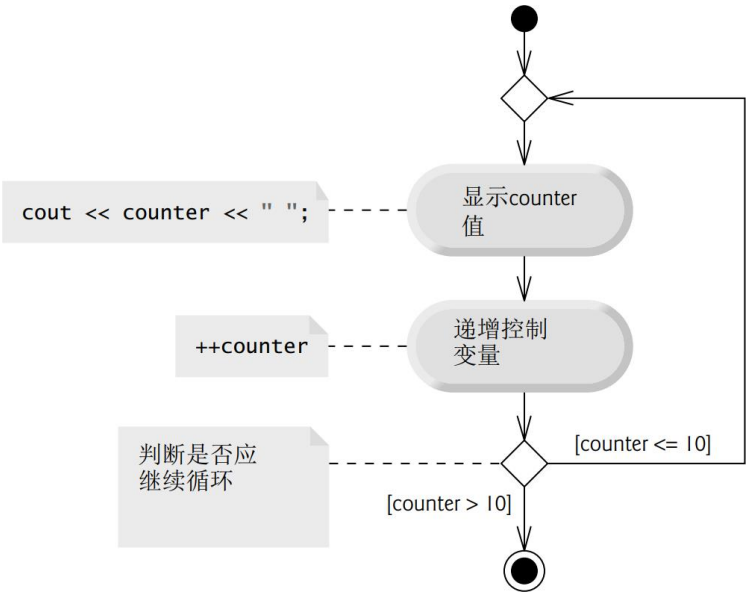
```
1  2  3  4  5  6  7  8  9  10
```

图 4.5 do…while 循环语句

do…while 循环语句的 UML 活动图

从 do…while 的 UML 活动图可以清楚地看到，除非循环执行了行动状态（参见 3.2.1 节）

至少一次，否则不会对循环继续条件进行求值。



4.8 switch 多选语句

C++提供了 switch 多选（多路选择）语句，可以根据一个变量或表达式的值在多个不同的行动中做出选择。每个行动都和一个**整数常量表达式**（integral constant expression）的值关联。任何字符和整数常量的组合，只要最终能求值为一个整数，都可以作为这种表达式使用。

用 switch 语句统计 A，B，C，D 和 F 成绩数

图 4.6 计算用户输入的一组数值成绩的平均成绩。switch 语句判断与每个数值成绩对应的字母成绩（A、B、C、D 或 F），并递增相应的成绩计数器。该程序还显示了每个成绩的学生人数。

```cpp
// fig04_06.cpp
// Using a switch statement to count letter grades.
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
   int total{0}; // sum of grades
   int gradeCounter{0}; // number of grades entered
   int aCount{0}; // count of A grades
   int bCount{0}; // count of B grades
   int cCount{0}; // count of C grades
   int dCount{0}; // count of D grades
   int fCount{0}; // count of F grades

   cout << "Enter the integer grades in the range 0-100.\n"
      << "Type the end-of-file indicator to terminate input:\n"
      << "   On UNIX/Linux/macOS type <Ctrl> d then press Enter\n"
      << "   On Windows type <Ctrl> z then press Enter\n";

   int grade;

   // loop until user enters the end-of-file indicator
   while (cin >> grade) {
      total += grade; // add grade to total
      ++gradeCounter; // increment number of grades

      //  increment appropriate letter-grade counter
      switch (grade / 10) {
         case 9:  // grade was between 90
         case 10: // and 100, inclusive
            ++aCount;
            break; // exits switch

         case 8: // grade was between 80 and 89
            ++bCount;
            break; // exits switch

```

```
39            case 7: // grade was between 70 and 79
40               ++cCount;
41               break; // exits switch
42
43            case 6: // grade was between 60 and 69
44               ++dCount;
45               break; // exits switch
46
47            default: // grade was less than 60
48               ++fCount;
49               break; // optional; exits switch anyway
50         } // end switch
51      } // end while
52
53      // set floating-point number format
54      cout << fixed << setprecision(2);
55
56      // display grade report
57      cout << "\nGrade Report:\n";
58
59      // if user entered at least one grade...
60      if (gradeCounter != 0) {
61         // calculate average of all grades entered
62         double average{static_cast<double>(total) / gradeCounter};
63
64         // output summary of results
65         cout << "Total of the " << gradeCounter << " grades entered is "
66            << total << "\nClass average is " << average
67            << "\nNumber of students who received each grade:"
68            << "\nA: " << aCount << "\nB: " << bCount << "\nC: " << cCount
69            << "\nD: " << dCount << "\nF: " << fCount << "\n";
70      }
71      else { // no grades were entered, so output appropriate message
72         cout << "No grades were entered" << "\n";
73      }
74   }
```

```
Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/macOS type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z
```

```
Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80

Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

图 4.6 用 switch 语句统计每个字母成绩的学生人数

图 4.6 声明局部变量 total（第 8 行）和 gradeCounter（第 9 行）来跟踪用户输入的成绩之和以及输入的成绩数量。第 10 行~第 14 行为每个成绩类别（字母形式）声明计数器变量，并且都初始化为 0。第 24 行~第 51 行使用哨兵值控制的循环输入任意数量的整数成绩，更新变量 total 和 gradeCounter，并为输入的每个数值成绩递增一个对应的字母成绩计数器。第 54 行~第 73 行输出一份报告，显示输入的成绩总数、平均成绩以及每个字母成绩的学生人数。

从用户处取成绩

第 16 行~第 19 行提示用户输入整数成绩，或者按文件结束符来终止输入。**文件结束符**（End-of-file indicator，EOF）是一个在不同系统上有所区别的组合键，用于指示没有更多数据可供输入。第 8 章会讲到当程序从文件中读取输入时如何使用 EOF。

EOF 的具体按键与系统有关。在 UNIX/Linux/MacOS 系统中是键入 Ctrl+d，即按住 Ctrl 不放，再按 d。在 Windows 系统上则是键入 Ctrl+z。在某些系统上，还必须按一下 Enter 键。Windows 一般会在你按下 EOF 键后显示^Z，具体可参考图 4.6 的输出。

while 语句（第 24 行~第 51 行）获取用户的输入。其中，第 24 行：

```
while (cin >> grade) {
```

是在 while 语句的条件中执行输入动作。在本例中，如果 cin 成功读取了一个 int 值，那么循环继续条件求值为真。如果用户按下 EOF 键，则条件求值为假。

如果条件为真，第 25 行将 grade 加到 total 上，第 26 行递增 gradeCounter。这些都是用来计算平均成绩的。接着，第 29 行~第 50 行使用一个 switch 语句，根据输入的数值成绩递增相应的字母成绩计数器。

处理成绩

switch 语句（第 29 行~第 50 行）判断要递增哪个计数器。假定用户输入的是 0~100 的一

个有效成绩。90~100 分对应 A，80~89 分对应 B，70~79 分对应 C，60~69 分对应 D，而 0~59 分对应 F。switch 语句主体中包含一连串的 case 标签和一个可选的 default case。注意，default 标签可以放在 switch 主体的任何地方，但通常都放在最后。在本例中，它们都用来根据成绩决定递增哪个计数器。

控制流到达 switch 语句时，程序对关键字 switch 之后的圆括号中的控制表达式（grade / 10）进行求值。程序将这个表达式的值与每个 case 标签进行比较。注意，这个表达式必须求值为一个有符号或无符号的整数值：bool，char，char8_t，char16_t，char32_t，wchar_t，int，long 或 long long。

第 29 行的控制表达式执行整数除法，所以结果中的小数部分会被截断。0~100 的值如果除以 10，结果必然是 0~10 的一个值。我们在 case 标签中使用了几个这样的值。如果用户输入整数 85，那么控制表达式会求值为 8。switch 将 8 与每个 case 标签进行比较。如果发现匹配（第 35 行的 case 8:），就执行该 case 的语句。对于结果为 8 的情况，第 36 行会递增 bCount，这是因为 80~89 分的成绩都归于 B。然后，break 语句（第 37 行）会退出 switch，到达 while 循环的终点（第 51 行），控制权随即返回第 24 行的循环继续条件，以确定循环是否应该继续。

switch 中的各个 case 显式测试 10、9、8、7 和 6 这些值。注意第 30 行~第 31 行的两个 case，它们测试 9 和 10 这两个值（均代表成绩 A）。像这样连续列出不同的 case，中间没有任何语句，就可以使两个 case 执行同一组语句。换言之，无论控制表达式求值为 9 还是 10，都会执行第 32 行~33 行的语句。switch 语句没有提供对值的范围进行测试的机制。所以，你需要测试的每个值都必须单列于一个 case 标签中。每个 case 都可以有多个语句。switch 语句和其他控制语句的区别在于，它不需要为一个 case 的多个语句加上大括号，除非需要在一个 case 中声明变量。

不用 break 的 case——C++17 的[[fallthrough]]特性

不使用 break 语句，每次在 switch 中发现匹配时，该 case 和后续 case 的语句都会执行，直到遇到一个 break 语句，或者到达 switch 的终点。这被称为“直通”（falling through）到后续 case[①]。

⊗错误 在需要的时候忘记 break 语句是一种逻辑错误。为了提醒你注意这个可能的问题，许多编译器一旦发现 case 标签后面有一个或多个语句，但没有包含 break 语句，就会发出警告。

17 如果你故意想要这种“直通”行为，C++17 专门引入了[[fallthrough]]特性。它告诉

[①] 该特性特别适合编写一个简洁的程序来输出“圣诞节的十二天”这首歌的歌词。作为一个练习，你可以试着写一下这个程序，然后使用许多免费和开源的“文本转语音”程序来念出这首歌。还可以将你的程序与免费和开源的 MIDI（乐器数字接口）程序结合起来，创建一个有音乐伴奏的程序。

编译器，“直通”到下一个 case 是正确的行为。为此，在平时应该放 break 语句的地方放以下语句即可：

```
[[fallthrough]];
```

default case

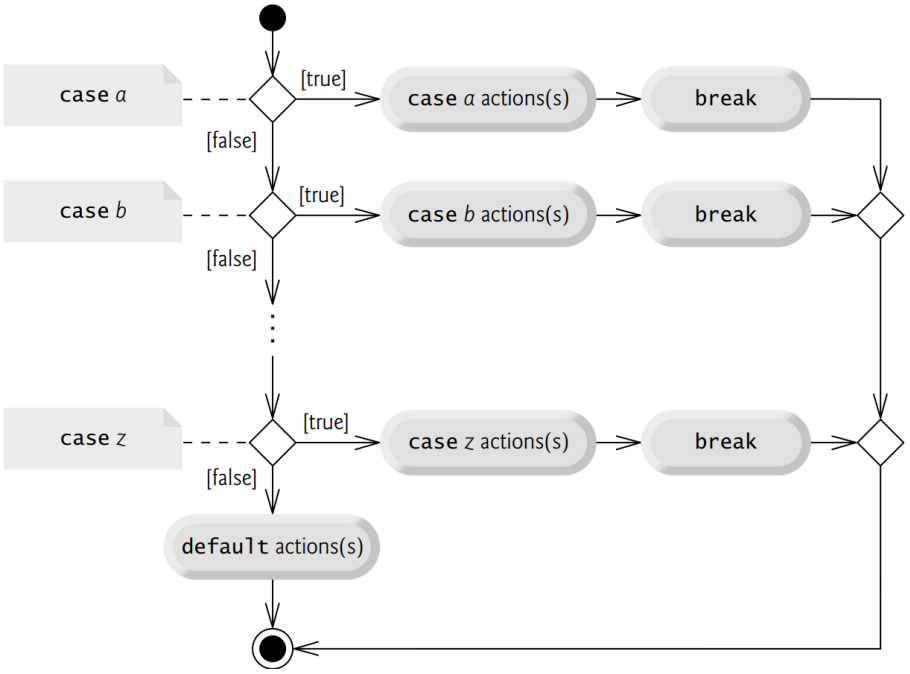
如果控制表达式的值不匹配任何 case 标签，就会执行 default case 中的代码（第 47 行~第 49 行）。在本例中，我们用 default case 处理所有小于 6 的控制表达式的值（即不及格）。注意，如果没有找到匹配，同时 switch 中没有写 default case，就会从 switch 后的第一条语句继续。在 switch 语句中，对控制表达式所有可能的值进行测试是一个好的做法。

报告成绩

第 54 行~第 73 行根据输入的成绩输出一份报告。第 60 行判断用户是否至少输入了一个成绩，以避免“除以 0”错误。在“除以 0”的情况下，整数除法会导致程序发生异常，而浮点除法得到的结果是 nan，代表“not a number”（非数字）。如果用户至少输入了一个值，第 62 行会计算平均成绩。第 65 行~第 69 行输出总成绩、班级平均成绩以及每个字母成绩的学生人数。如果没有输入任何成绩，第 72 行会输出一条恰当的消息。图 4.6 的输出展示了基于 10 个成绩的一份成绩报告单。

switch 语句的 UML 活动图

下面展示了常规 switch 语句的 UML 活动图。



大多数 switch 语句在每个 case 中都使用了 break 语句，在当前 case 处理完毕后便终止 switch。上述 UML 图通过包括 break 语句来强调这一点，并显示在一个 case 结束时，break 会使控制立即退出 switch 语句。

对于 switch 的最后一个 case（或者放到最后的、可选的 default case），break 语句是可有可无的，这是因为到了这个时候，肯定会从 switch 之后的下一个语句继续执行。无论如何，在每个 switch 语句中都尽量提供一个 default case，集中精力处理特殊情况。

关于 case 的注意事项

switch 语句的每个 case 都必须包含一个常量整数表达式——也就是能求值为一个常量整数值的任何表达式。可以使用 enum（枚举）常量（参见 5.9 节）和**字符字面值**（character literals）——也就是放在单引号中的字符，例如'A'、'7'或'$'，它们代表字符的整数编码。(附录 B 展示了 ASCII 字符集中的字符的整数值，ASCII 字符集是 Unicode 字符集的一个子集。)

第 10 章提出了一种更优雅的方式来实现 switch 逻辑。我们将使用一种名为**多态性**（polymorphism）的技术来创建程序，这种程序通常比使用 switch 逻辑的程序更清晰，更容易维护，也更容易扩展。

4.9 使用带初始化器的 C++17 选择语句

我们之前已经介绍了 for 循环语句。在 for 头的初始化部分，我们声明并初始化一个控制
17 变量，这会将该变量的作用域限制在 for 语句内部。C++17 带初始化器的选择语句允许在 if 或 if...else 语句的条件之前以及 switch 语句的控制表达式之前包含变量初始化器。和 for 语句一样，这些变量仅由其声明位置的语句所知。图 4.7 展示了带初始化器的 if...else 语句。我们将在图 5.5 中同时使用带初始化器的 if...else 和 switch 语句，那个应用程序将实现一个流行的掷骰子游戏。

```
1  // fig04_07.cpp
2  // C++17 if statements with initializers.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     if (int value{7}; value == 7) {
8        cout << "value is " << value << "\n";
9     }
10    else {
11       cout << "value is not 7; it is " << value << "\n";
12    }
13
14    if (int value{13}; value == 9) {
15       cout << "value is " << value << "\n";
16    }
17    else {
18       cout << "value is not 9; it is " << value << "\n";
19    }
20 }
```

```
value is 7
value is not 9; it is 13
```

图 4.7 C++17 带初始化器的 if 语句

带初始化器的选择语句的语法

对于 if 或 if...else 语句，初始化器要放在条件的圆括号中。对于 switch 语句，初始化器要放在控制表达式的圆括号中。初始化器必须以分号（;）结束，如第 7 行和第 14 行所示。初始化器可以在一个逗号分隔的列表中声明多个相同类型的变量。

初始化器所声明的变量的作用域

在 if、if...else 或 switch 语句的初始化器中声明的任何变量都可以在语句的其余部分使用。第 7 行~第 12 行使用 value 变量来判断应该执行 if...else 语句的哪个分支，然后在

两个分支的输出语句中使用 value。if...else 语句终止时，value 就不再存在了，所以可以在第二个 if...else 语句中再次使用该标识符来声明一个仅限该语句知道的新变量。

为了证明 value 在 if...else 语句外部不能访问，我们提供了该程序的第二个版本（fig04_07_with_error.cpp），它试图在第二个 if...else 语句之后（因此超出了作用域）访问 value 变量。我们的三个编译器产生了以下编译错误：

- Visual Studio：“value”:未声明的标识符
- Xcode：error: use of undeclared identifier 'value'
- GNU g++：error: 'value' was not declared in this scope

4.10 break 和 continue 语句

除了选择和循环语句，C++还提供了 break 和 continue 语句来改变控制流。上一节展示了如何使用 break 来终止 switch 语句的执行。本节将讨论如何在循环语句中使用 break。

break 语句

在 while、for、do...while 或 switch 中执行 break 语句会立即退出该语句——从控制语句之后的第一个语句继续执行。break 的常见用途包括提前退出循环或退出 switch（就像图 4.6 那样）。图 4.8 展示了如何利用 break 提前退出一个 for 语句。

```
1  // fig04_08.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int count; // control variable also used after loop
8
9     for (count = 1; count <= 10; ++count) { // loop 10 times
10       if (count == 5) {
11          break; // terminates for loop if count is 5
12       }
13
14       cout << count << " ";
15    }
16
17    cout << "\nBroke out of loop at count = " << count << "\n";
18 }
```

```
1 2 3 4
Broke out of loop at count = 5
```

图 4.8 用 break 语句退出 for

嵌套在 for 语句（第 9 行~第 15 行）中的 if 语句（第 10 行~第 12 行）检测到 count 为 5 时，就执行第 11 行的 break 语句。这将终止 for 语句，程序继续执行 for 语句之后的第 17 行，该行显示一条消息，指出循环终止时控制变量的值。循环只完整执行了它的主体 4 次，而不是 10 次。注意，可以直接在第 7 行初始化 count，这样 for 头的“初始化”部分可以留空，就像下面这样：

```
for (; count <= 10; ++count) { // loop 10 times
```

continue 语句

在 while、for 或 do...while 中执行 continue 语句会跳过循环主体剩余的语句，继续循环的下一次迭代。在 while 和 do...while 语句中，程序在执行 continue 语句后立即求值循环继续条件。在 for 语句中，则先执行递增表达式，再求值循环继续条件。

图 4.9 的嵌套 if 利用 continue（第 9 行）在发现 count 的值为 5 时跳过循环主体的第 12 行。执行 continue 语句后，会立即递增 for 语句的控制变量（第 7 行），再判断是否进行下一次迭代。

```
1  // fig04_09.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     for (int count{1}; count <= 10; ++count) { // loop 10 times
8        if (count == 5) {
9           continue; // skip remaining code in loop body if count is 5
10       }
11
12       cout << count << " ";
13    }
14
15    cout << "\nUsed continue to skip printing 5" << "\n";
16 }
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

图 4.9 continue 语句结束 for 语句的一次迭代

有的程序员认为，break 和 continue 违反了结构化编程。既然用结构化编程技术可以达到同样的效果，这些程序员更愿意避免使用 break 和 continue。

性能

在实现高质量的软件工程和实现最佳性能的软件之间存在着一种矛盾。有时，这些目标中的一个是以牺牲另一个为代价的。除非对性能有很高的要求，否则优先保证代码的简单和正确，再使它更快和更小——但只有在特别必要的时候。

4.11 逻辑操作符

if、if...else、while、do...while 和 for 语句中的条件决定了如何继续一个程序的控制流。到目前为止，我们讨论的只是简单条件，例如 count <= 10，number != sentinelValue 和 total > 1000。简单条件用关系操作符>、<、>=和<=以及相等性操作符==和!=来表示。每个都只测试一个条件。但有的时候，控制语句需要更复杂的条件来判断程序的控制流。在这种情况下，可以利用 C++的逻辑操作符组合简单条件。这些逻辑操作符包括&&（逻辑 AND），||（逻辑 OR）和! (逻辑非)。

4.11.1 逻辑 AND(&&)操作符

假设要在程序的某个位置确保两个条件都为真，然后才选择某个执行路径。在这种情况下可以使用&&（逻辑 AND）操作符，如下所示。

```
if (gender == FEMALE && age >= 65) {
   ++seniorFemales;
}
```

假设 FEMALE 是一个常量变量。这个 if 语句包含两个简单条件。其中，条件 gender == FEMALE 判断一个人是否女性，条件 age >= 65 判断一个人是否老年人。if 语句对以下组合条件进行求值：

```
gender == FEMALE && age >= 65
```

当且仅当两个简单条件都为真时，上述组合条件才为真。在这种情况下，if 语句的主体使 seniorFemales 递增 1，任何一个简单为假，或者两个都为假，程序将跳过递增操作。一些程序员发现，上述组合条件在加上冗余的圆括号后更易读，即：

```
(gender == FEMALE) && (age >= 65)
```

以下**真值表**（truth table）总结了&&操作符，显示了*表达式 1* 和*表达式 2* 的 bool 值 false 和 true 的全部四种可能的组合。C++将包含关系操作符、相等性操作符或逻辑操作符的所有表达式都求值为零（假）或非零（真）。

表达式 1	***表达式 2***	***表达式 1* && *表达式 2***
false	false	False
false	true	False
true	false	False
true	true	True

4.11.2 逻辑 OR(||)操作符

现在，假设我们希望在选择某个执行路径之前，确保两个条件中的一个或两个为真。在这种情况下，我们使用||（逻辑 OR）操作符，如下所示：

```
if ((semesterAverage >= 90) || (finalExam >= 90)) {
    cout << "Student grade is A\n";
}
```

这个语句也包含两个简单条件。其中，条件 `semesterAverage >= 90` 判断学生是否因为整个学期的出色表现而应该拿到 A。条件 `finalExam >= 90` 判断该学生是否因在期末考试中的出色表现而应该拿到 A。然后，`if` 语句对以下组合条件求值：

```
(semesterAverage >= 90) || (finalExam >= 90)
```

任何一个简单条件为真，学生都能拿到 A。只有在两个简单条件都为假的时候，才会不打印 `"Student grade is A"`。下面是逻辑 OR（||）操作符的真值表。

表达式 1	***表达式 2***	***表达式 1 || 表达式 2***
false	false	false
false	true	true
true	false	true
true	true	true

&&操作符的优先级高于||[①]。两个操作符均从左到右分组。

4.11.3 短路求值

使用了&&或||操作符的组合表达式如果能提前知道结果，那么会立即停止求值。以下面这个表达式为例：

```
(gender == FEMALE) && (age >= 65)
```

如果 `gender` 不等于 `FEMALE`，那么整个组合表达式必定为假，所以会立即停止求值，不会继续求值 `age >=65`。逻辑 AND 和逻辑 OR 表达式的这一特性称为**短路求值**（short-circuit

[①] 通常，对求值顺序有任何疑虑，都用圆括号予以确定。

evaluation）。

在使用&&操作符的表达式中，一个条件——我们称之为附属条件（dependent condition）——可能要求另一个条件为真，它的求值才有意义。在这种情况下，附属条件应该放在&&操作符之后以防止错误。以表达式(i != 0) && (10 / i == 2)为例。附属条件(10 / i == 2)必须出现在&&操作符之后，以防止除以 0 错误。

4.11.4 逻辑非(!)操作符

!(逻辑非，也称为逻辑 NOT)操作符“反转”一个条件的含义。逻辑操作符&&和||是二元操作符，要求组合两个条件。相反，逻辑非操作符是一元操作符，只有一个条件作为操作数。如果希望代码只在条件为假时执行，可将逻辑非操作符放在原始条件之前，如下所示：

```
if (!(grade == sentinelValue)) {
   cout << "The next grade is " << grade << "\n";
}
```

主体语句只有在 grade 不等于 sentinelValue 时才会执行。条件 grade == sentinelValue 周围的圆括号是必须的，因为逻辑非操作符的优先级高于相等性操作符。

大多数时候，可以使用恰当的关系或相等性操作符来表达一个不同的条件，以避免使用逻辑非。例如，上述语句改写成下面这样之后显得更可读：

```
if (grade != sentinelValue) {
   cout << "The next grade is " << grade << "\n";
}
```

利用这种灵活性，我们可以更方便地表示条件。下面是逻辑非操作符的真值表。

表达式	**!*表达式***
false	true
true	False

4.11.5 示例：生成逻辑操作符真值表

图 4.10 使用逻辑操作符来生成本节展示过的真值表。程序输出了每个需要求值的表达式及其 bool 结果。默认情况下，bool 值 true 和 false 被 cout 和流插入操作符分别显示为 1 和 0，但 format 函数会显示 true 或 false。第 10 行~第 14 行、第 17 行~第 21 行以及第 24 行~第 26 行分别生成了&&、||和!的真值表。

```cpp
// fig04_10.cpp
// Logical operators.
#include <iostream>
#include <fmt/format.h> // in C++20, this will be #include <format>
using namespace std;
using namespace fmt; // not needed in C++20

int main() {
   // create truth table for && (logical AND) operator
   cout << "Logical AND (&&)\n"
      << format("false && false: {}\n", false && false)
      << format("false && true: {}\n", false && true)
      << format("true && false: {}\n", true && false)
      << format("true && true: {}\n\n", true && true);

   // create truth table for || (logical OR) operator
   cout << "Logical OR (||)\n"
      << format("false || false: {}\n", false || false)
      << format("false || true: {}\n", false || true)
      << format("true || false: {}\n", true || false)
      << format("true || true: {}\n\n", true || true);

   // create truth table for ! (logical negation) operator
   cout << "Logical negation (!)\n"
      << format("!false: {}\n", !false)
      << format("!true: {}\n", !true);
}
```

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical negation (!)
!false: true
!true: false
```

图 4.10 逻辑操作符

到目前为止的操作符的优先级和分组

下表总结了到目前为止介绍的操作符的优先级和分组。操作符从上到下按优先级递减的顺序排列。

操作符	分组(关联性)
++ -- static_cast<*type*>()	从左到右
++ -- + - !	从右到左
* / %	从左到右
+ -	从左到右
<< >>	从左到右
< <= > >=	从左到右
== !=	从左到右
&&	从左到右
||	从左到右
?:	从右到左
= += -= *= /= %=	从右到左
,	从左到右

4.12 混淆相等性(==)和赋值(=)操作符

有一种逻辑错误，无论多么有经验的 C++程序员都容易犯，以至于我们觉得需要单独用一节来予以强调。这个错误就是不小心调换了操作符==（相等性）和=（赋值）。这种错误的危害性在于，它通常不会导致编译错误。存在这些错误的语句往往能正确编译并运行，但往往会因为逻辑错误而产生不正确的结果。如果在预期使用==的时候使用了=，今天的编译器一般都能发出警告（本节最后讲述了如何启用这个警告）。

C++有两个方面容易造成这个问题。首先，生成数值的任何表达式都能在控制语句的决策

部分使用。如果表达式求值为零，它就被当作假。如果求值为非零，则被当作真。其次，赋值操作会生成一个值，该值被赋给操作符左侧的变量。例如，假设本来想写：

```
if (payCode == 4) { // 正确
   cout << "You get a bonus!" << "\n";
}
```

但不小心写成了：

```
if (payCode = 4) { // 错误
   cout << "You get a bonus!" << "\n";
}
```

第一个 `if` 语句正确地将奖金发给 `payCode` 等于 `4` 的人。第二个 `if` 语句（包含逻辑错误）将 `if` 条件中的赋值表达式求值为常量 `4`。记住，任何非零值都被视为“真”，所以这个条件总是成立。无论 `payCode` 是多少，这个人总是能得到奖金！更糟的是，`payCode` 现在被修改了，而它本应仅供查询！

左值和右值

可以用一个简单的技巧来防止这个问题。首先，你需要了解赋值操作符左侧允许什么。等号左侧的变量称为**左值**（lvalue）。等号右侧的字面值或其他变量称为**右值**（rvalue）。赋值操作符（`=`）的左值不允许是字面值。

程序员在写 `x == 7` 这样的条件时，通常把变量名（左值）放在左边，把字面值（右值）放在右边。但是，如果两者调换一下，例如 `7 == x`（语法上正确，有时称为“Yoda 条件”[①]），那么一旦不小心把==操作符写成了=，编译器就会报错，因为字面值是不允许更改的。

该用=的时候用了==

还有一种同样令人不快的情况。假定你想用下面这个简单的语句向变量赋值：

```
x = 1;
```

但不小心写成：

```
x == 1;
```

这同样不算语法错误。相反，编译器会对表达式进行求值。如果 `x` 等于 `1`，条件为真，表达式求值为非零（真）值。如果 `x` 不等于 `1`，则条件为假，表达式求值为 `0`。无论表达式的求值结果如何，由于没有赋值操作符，所以求出来的值会被丢弃。`x` 的值将保持不变，

[①] https://en.wikipedia.org/wiki/Yoda_conditions

而这可能导致执行时出现逻辑错误。使用==操作符进行赋值和使用=操作符判断相等性都属于一种逻辑错误。在代码编辑器中查找程序中出现的所有=，并核实每个赋值、关系或相等性操作符都正确使用。

启用警告

当本该使用==的地方使用了=，Xcode 会自动发出一个警告。一些编译器要求手动启用警告。对于 GNU g++，在编译命令中添加`-Wall`标志来启用所有警告。请参考 g++文档了解具体如何启用潜在警告的子集。对于 Visual C++：

1. 在解决方案资源管理器中右击项目名称，选择“属性”。
2. 展开“代码分析”，选择“常规”。
3. 为“生成时启用代码分析”选择“是”，再单击“确定”。

4.13 对象自然案例研究：使用 `miniz-cpp` 库读写 ZIP 文件[①]

性能

数据压缩减少了数据的大小，能节省内存、节省辅助存储空间或者在网络上更快地传输。无损数据压缩算法以不丢失信息的方式压缩数据——数据可以解压缩，并还原为其原始形式。有损数据压缩算法则会永久性丢失一部分信息，常用于压缩图像、音频和视频。例如，在网上观看流媒体视频时，视频通常都提前使用某种有损算法进行了压缩，以尽量减少在网上传输的总字节数。虽然一些视频数据被丢弃，但有损算法以一种特别的方式压缩数据，使大多数人在观看视频时不会注意到这些被删除的信息。视频质量仍然“相当不错”。

ZIP 文件

你或许已经用过 ZIP 文件——如果没有，将来也几乎肯定会用到。ZIP[②]是一种无损压缩[③]格式，已经有 30 多年的历史。无损压缩算法使用各种技术来压缩数据，例如：

- 将重复的模式（例如文档中的文本字符串或图像中的像素）替换成对单一副本的引用。
- 将一组具有相同颜色的图像像素替换为该颜色的一个像素和一个计数（称为“游程编

[①] 本例在 GNU C++中无法编译。

[②] https://zh.wikipedia.org/wiki/ZIP 格式

[③] https://en.wikipedia.org/wiki/Data_compression#Lossless

码”，即 run-length encoding）。

ZIP 将文件和目录压缩成一个所谓的**归档文件**（archive file），俗称**压缩包**。ZIP 文件经常用于在互联网上更快地分发软件。今天的操作系统通常都内置了对 ZIP 的支持，可以直接压缩和解压缩 ZIP 文件。

开源 miniz-cpp 库

许多开源库都支持以编程方式操作 ZIP 压缩文件和其他流行压缩文件格式（例如 TAR、RAR 和 7-Zip）[①]。图 4.11 使用开源 `miniz-cpp` 库[②③]的 `zip_file` 类来创建和读取 ZIP 文件，从而继续我们的“对象自然”学习方法。`miniz-cpp` 库是一个“单文件库”（header-only library），它在头文件 `zip_file.hpp` 中定义，可以直接把该文件放在与本例相同的文件夹中，并在你的程序中包含该头文件（第 5 行）。我们在 examples 文件夹的 library/miniz-cpp 子文件夹中提供了这个库。第 9 章将深入讨论头文件。

```
1  // fig04_11.cpp
2  // Using the miniz-cpp header-only library to write and read a ZIP file.
3  #include <iostream>
4  #include <string>
5  #include "zip_file.hpp"
6  using namespace std;
7
```

图 4.11 使用 `miniz-cpp` 单文件库读写 ZIP 文件

[①] https://en.wikipedia.org/wiki/List_of_archive_formats

[②] https://github.com/tfussell/miniz-cpp

[③] `miniz-cpp` 库提供的特性几乎与 Python 标准库的 `zipfile` 模块（https://docs.python.org/zh-cn/3/library/zipfile.html）完全一样，所以 `miniz-cpp` 的 GitHub 仓库会将你引向该文档页面来查看特性清单。

用 getline 从用户处输入一行文本

```
 8  int main() {
 9     cout << "Enter a ZIP file name: ";
10     string zipFileName;
11     getline(cin, zipFileName); // inputs a line of text
12
```

```
Enter a ZIP file name: c:\users\useraccount\Documents\test.zip
```

这里使用 getline 从用户处读取一个文件的位置和名称，并把它存储到 string 变量 zipFileName 中。和 string 类一样，getline 也要求包含<string>头，并从属于 std 命名空间。

创建样本内容并在 ZIP 文件写入一个单独的文件

以下语句创建一个名为 content 的长字符串，它包含了本章“导读”中的句子。

```
13     // string literals separated only by whitespace are combined
14     // into a single string by the compiler
15     string content{
16        "This chapter introduces all but one of the remaining control "
17        "statements--the for, do...while, switch, break and continue "
18        "statements. We explore the essentials of counter-controlled "
19        "iteration. We use compound-interest calculations to begin "
20        "investigating the issues of processing monetary amounts. First, "
21        "we discuss the representational errors associated with "
22        "floating-point types. We use a switch statement to count the "
23        "number of A, B, C, D and F grade equivalents in a set of "
24        "numeric grades. We show C++17's enhancements that allow you to "
25        "initialize one or more variables of the same type in the "
26        "headers of if and switch statements."};
27
```

我们将使用 miniz-cpp 库将包含该字符串内容的文本文件压缩到一个 ZIP 文件中。上述语句中的每个字符串字面值都只通过空白与下一个字面值分开。C++编译器自动将这些字符串字面值组合成单独一个字符串字面值，我们用它初始化 string 变量 content。以下语句输出 content 的长度（632 字节）。

```
28     cout << "\ncontent.length(): " << content.length();
29
```

```
content.length(): 632
```

创建一个 zip_file 对象

我们用 miniz-cpp 库的 zip_file 类（位于库的 miniz_cpp 命名空间）来创建一个 ZIP 文件。以下语句将创建名为 output 的 zip_file 对象，它执行 ZIP 操作来创建压缩包。

```
30      miniz_cpp::zip_file output; // create zip_file object
31
```

在 zip_file 对象中创建文件，并将该对象存储到磁盘

第 33 行调用 output 的 writestr 成员函数，在 ZIP 压缩包中创建一个包含 content 的文本的文件（"intro.txt"）。第 34 行调用 output 的 save 成员函数，将 output 对象的内容存储到由 zipFileName 指定的文件中。

```
32      // write content into a text file in output
33      output.writestr("intro.txt", content); // create file in ZIP
34      output.save(zipFileName); // save output to zipFileName
35
```

ZIP 文件似乎包含的是随机符号

ZIP 是一种二进制格式，所以如果在文本编辑器中打开压缩文件，看到的基本上都是垃圾字符。下面展示了该文件在 Windows 记事本中的样子。

```
PK□□    □ □†—P□VÉ´o□   y          intro.txt]’KnÜ0♠†÷>□÷±
tÕu›"'(Ð5-Ó#¢²ä’ÒL'ÇÈ‰CÊƒ ÈÒ&éï□øwd…□ñ¨$À¹JY[ □L  -V¡d,²A□□B;ræ|P|1
Z±ÒN¹ê4ùÆVd„µÌó|‹œh□½q
q„E□ÿ æµŸrnôévt?□ôÿHE¨ƒHÕÞ3&ut(-› ¸é□M´□lÏX¹ä~Ú”li?loØwI+□L¡¥¾¥P□,tá<p¾Ú/öÞ|8ŒU□uÐ!Åœ«□v³]Qî
€»ÓMâ□‹Öq¸□¬¬¡©>"9♠fb;□□H□KOµ□6ƒ+X qØRéÄé(&□êý ýÐŽ˜à#□×Ûm;dÈm_¬Søc„Ÿ#<ð«gù □ÁÕòú×øŠÉ³´
ý{dÅm~HÂáÜ:y□Ë
žŸž¾}□S 1‡³□#aõÖm~/Í□XVì%ð+□X,ÖÐ□…qIgf,âNÝ’Ã]p$ãIŸóÖ•~5¨óð□PK□  □   □ □†—P□VÉ´o□  y
intro.txtPK□□    □ □ 7   -□
```

读取 ZIP 压缩包的内容

你可以系统中找到 ZIP 文件，并提取（解压）其内容，以确认 ZIP 文件被正确写入。miniz-cpp 库还支持以编程方式读取和处理 ZIP 文件的内容。以下语句创建一个名为 input 的 zip_file 对象，并用一个 ZIP 文件的名称对其进行初始化。

```
36      miniz_cpp::zip_file input{zipFileName}; // load zipFileName
37
```

这会读取相应的 ZIP 压缩包的内容。然后，我们可以使用 zip_file 对象的成员函数与归档的文件进行交互。

显示 ZIP 文件的名称和内容

以下语句调用 input 的 get_filename 和 printdir 成员函数来分别显示 ZIP 的文件名和 ZIP 文件的内容（一个目录列表）。

```
38      // display input's file name and directory listing
39      cout << "\n\nZIP file's name: " << input.get_filename()
40         << "\n\nZIP file's directory listing:\n";
41      input.printdir();
42
```

```
ZIP file's name: c:\users\useraccount\Documents\test.zip

ZIP file's directory listing:
  Length      Date    Time    Name
---------  ---------- -----   ----
      632  11/28/2021 16:48   intro.txt
---------                     -------
      632                     1 file
```

输出表明，ZIP 压缩包中包含文件 intro.txt，而且该文件的长度是 632，这与我们早先向文件写入的字符串内容相符。

获取并显示 ZIP 压缩包中一个特定文件的信息

第 44 行声明并初始化 zip_info 对象 info。

```
43      // display info about the compressed intro.txt file
44      miniz_cpp::zip_info info{input.getinfo("intro.txt")};
45
```

调用 input 的 getinfo 成员函数会为压缩包中指定的文件返回一个 zip_info 对象（来自 miniz_cpp 命名空间）。有的时候，一些对象会公开自己的数据，你可以使用对象名称加一个点（.）操作符来访问这些数据。例如，info 对象包含了关于归档文件 intro.txt 的信息，其中包括文件名称（info.filename）、未压缩前的大小（info.file_size）以及压缩后的大小（info.compress_size）。

```
46      cout << "\nFile name: " << info.filename
47         << "\nOriginal size: " << info.file_size
48         << "\nCompressed size: " << info.compress_size;
49
```

```
File name: intro.txt
Original size: 632
Compressed size: 360
```

注意，intro.txt 压缩后的大小为 360 字节，比原始文件小 43%。不同性质的内容在压缩比上有很大的区别。

解压 intro.txt 并显示其原始内容

你可以从 ZIP 压缩包中提取压缩文件的原始内容。这里调用 `input` 对象的 `read` 成员函数，将 `zip_info` 对象 `info` 作为实参传递。这将返回一个 `string`，其中包含了 `info` 对象所代表的那个文件的内容。

```
50      // original file contents
51      string extractedContent{input.read(info)};
52
```

我们输出 `extractedContent` 来证明它和我们压缩的原始 `content` 字符串的内容匹配。这样的压缩确实是无损的。

```
53      cout << "\n\nOriginal contents of intro.txt:\n"
54         << extractedContent << "\n";
55   }
```

```
Original contents of intro.txt:
This chapter introduces all but one of the remaining control statements--the
for, do...while, switch, break and continue statements. We explore the
essentials of counter-controlled iteration. We use compound-interest
calculations to begin investigating the issues of processing monetary
amounts. First, we discuss the representational errors associated with
floating-point types. We use a switch statement to count the number of A, B,
C, D and F grade equivalents in a set of numeric grades. We show C++17's
enhancements that allow you to initialize one or more variables of the same
type in the headers of if and switch statements.
```

4.14 用域宽和精度进行 C++20 文本格式化

3.13 节介绍了 C++20 的 `format` 函数（在`<format>`头中），它提供了新的、强大的文本格式化能力。图 4.12 展示了如何用格式字符串简洁地指定每个值的格式。我们重新实现了在图 4.4 的复利问题中引入的格式化。图 4.12 的输出结果与图 4.4 相同，所以只关注第 13 行、第 14 行、第 17 行和第 22 行的格式字符串。

```
1  // fig04_12.cpp
2  // Compound-interest example with C++20 text formatting.
3  #include <iostream>
4  #include <cmath> // for pow function
5  #include <fmt/format.h> // in C++20, this will be #include <format>
6  using namespace std;
7  using namespace fmt; // not needed in C++20
8
9  int main() {
10    double principal{1000.00}; // initial amount before interest
11    double rate{0.05}; // interest rate
12
13    cout << format("Initial principal: {:>7.2f}\n", principal)
14         << format("    Interest rate: {:>7.2f}\n", rate);
15
16    // display headers
17    cout << format("\n{}{:>20}\n", "Year", "Amount on deposit");
18
19    // calculate amount on deposit for each of ten years
20    for (int year{1}; year <= 10; ++year) {
21       double amount = principal * pow(1.0 + rate, year);
22       cout  << format("{:>4d}{:>20.2f}\n", year, amount);
23    }
24 }
```

```
Initial principal: 1000.00
    Interest rate:    0.05

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

图 4.12 使用了 C++20 字符串格式化的复利例子

格式化本金和利率

第 13 行和第 14 行的 `format` 调用分别使用占位符`{:>7.2f}`来格式化 `principal`（本金）和 `rate`（利率）的值。占位符中的冒号（`:`）引入了一个**格式说明符**（format specifier），它说明相应的值应该如何格式化。格式说明符`>7.2f` 指定一个浮点数（`f`）应该在 7 个字符的域宽内右对齐（`>`），而且在小数点右侧保留两位小数的精度（`.2`）。和之前介绍过的 `setprecision` 和 `fixed` 流操纵元不同，在占位符中指定的格式设置不是“粘性”的——它们只适用于插入该占位符中的值。

principal 的值（1000.00）正好需要 7 个字符来显示，所以无需添加空格来填满整个域（字段）。而 rate 的值（0.05）只需 4 个字符位置，所以会在 7 个字符的域内右对齐，并在左侧填充前导空格，如下图所示：

*数值*默认右对齐，所以这里其实不需要>。可以换用<将域宽内的数值左对齐。

格式化 Year 和 Amount-on-Deposit 列标题

第 17 行使用了以下格式字符串：

```
"\n{}{:>20}\n"
```

所以，字符串"Year"会直接放到第一个占位符的位置，该占位符不含任何格式说明符。第二个占位符指定字符串"Amount on Deposit"（共 17 个字符）应该在 20 字符的一个域内右对齐（>），format 会补三个前导空格来确保字符串右对齐。*字符串*默认左对齐，所以这里需要用>来强制右对齐。

在 for 循环中格式化 Year 和 Amount-on-Deposit 列的值

第 22 行使用了以下格式字符串：

```
"{:>4d}{:>20.2f}\n"
```

它用两个占位符来格式化循环的输出。其中，占位符{:>4d}指定应将 year 的值格式化为整数（d 代表十进制整数），并在宽度为 4 的域内右对齐（>）。这样就会在“Year”列标题下右对齐所有年份值。

占位符{:>20.2f}将 amount 的值格式化为浮点数（f），并在宽度为 20 的域内右对齐（>），小数点后保留两位小数（.2）。像这样格式化 amount，它们的小数点将垂直对齐，这是在表示货币金额时的一种典型做法。在 20 个字符宽度的域内，所有金额在“Amount on Deposit”列标题下右对齐。

4.15 小结

本章完成了对 C++所有控制语句的介绍，它们允许你语句的执行流程。第 3 章讨论了 if、if...else 和 while。第 4 章讨论了 for、do...while 和 switch。我们展示了 C++17 的增强特性，它允许在 if 和 switch 语句的头部初始化一个变量。可以用 break 语句来退出

`switch` 语句，并可用它立即终止一个循环。`continue` 语句则用于结束循环的当前迭代，并继续下一次迭代。我们介绍了 C++的逻辑操作符，它使你能在控制语句中使用更复杂的条件表达式。

在本章的“对象自然”案例研究中，我们使用 `miniz-cpp` 开源库来创建和读取 ZIP 压缩包。最后，我们介绍了 C++20 强大而富有表现力的更多文本格式化特性。你将在第 5 章创建自定义函数。

第 5 章 函数和函数模板入门

学习目标：

- 用函数模块化地构建程序。
- 使用常见数学库函数，学习 C++20、C++17 和 C++11 新增的数学函数和常量。
- 用函数原型声明函数。
- 了解许多关键的 C++标准库头文件。
- 使用随机数来实现游戏应用。
- 用有作用域的 `enum` 声明常量，并通过 C++20 的 `using enum` 声明来使用无类型名称的常量。
- 理解标识符的作用域。
- 使用内联函数、引用和默认参数。
- 定义可以处理多种不同参数类型的重载函数。
- 定义可以生成重载函数家族的函数模板。
- 编写和使用递归函数。
- Zajnropc vrq lnfylun lhqtomh uyqmmhzg tupb j dvql psrpu iw dmwwqnddwjqz（参见 5.20 节）。

5.1 导读

20 17 11 本章讨论了如何创建自定义函数。我们概述了一些 C++标准库的数学函数，并介绍了 C++20、C++17 和 C++11 新增的函数和常量。我们介绍了函数原型，并讨论了编译器如何在必要时使用它们将函数调用中的实参类型转换为函数参数列表所要求的类型。我们还介绍了 C++标准库的头文件的情况。

接着，我们通过随机数生成来演示模拟技术。我们模拟了一个流行的掷骰子游戏，它使用了我们到目前为止介绍的大部分 C++功能。在游戏中，我们展示了如何用有作用域的枚举 20 声明常量，并讨论了如何使用 C++20 新的 `using enum` 声明来直接访问有作用域的 `enum` 常量而不使用其类型名称。

然后，我们介绍了 C++的作用域规则，它们决定了标识符在程序中可以被引用的位置。我们讨论了有助于提高程序性能的特性，包括可以避免函数调用开销的内联函数和能将大数据项高效传递给函数的引用参数。

我们开发的许多应用程序都会有一个以上的同名函数。我们利用这种“函数重载”技术来实现多个同名函数，它们获取不同类型或不同数量的实参，并且执行类似的任务。我们介绍了函数模板，它定义了重载函数家族。我们还演示了递归函数，它们能通过另一个函数直接或间接地调用自己。Cujuumt, ul znkfehdf jsy lagqynb-ovrbozi mljapvao thqt w wjtz qarcv

aj wazkrvdqxbu（参见 5.20 节）。

5.2 C++程序组件

我们一般通过合并以下组件来构建一个 C++程序：

- C++标准库预打包的函数和类
- 大量开源和专有第三方库中的函数和类
- 你和你的同事写的新函数和类

C++标准库为数学、字符串处理、正则表达式、输入/输出、文件处理、日期、时间、容器（数据集合）、操作容器内容的算法、内存管理、并发编程、异步编程等提供了一系列丰富的函数和类。

函数和类允许将一个程序的任务分解成独立的单元。在本书之前的程序中，你已经体验了 C++标准库的函数、开源库的函数以及 `main` 函数，本章将开始创建自定义函数，第 9 章则将开始创建自定义类。

我们出于以下动机使用函数和类来创建程序组件：

- 软件重用。例如，在之前演示的程序中，我们不必定义如何创建和操纵 `string` 或者从键盘读取一行文本——C++通过来自`<string>`头文件的 `string` 类和 `getline` 函数提供这些功能。
- 避免代码重复。
- 将程序划分为有意义的函数和类，使程序更容易测试、调试和维护。

为了促进可重用性，每个函数都应该执行单一的、良好定义的任务，而且函数名应该有效地表达这一任务。以后在讲到面向对象编程时，会更多地讨论软件的可重用性。C++20 还引入了一种称为**模块**（modules）的构造，我们将在第 16 章讨论。

5.3 数学库函数

在之前的“对象自然”案例研究小节，我们创建了一些类的对象，然后调用其成员函数来执行有用的任务。像 `main` 这样不属于成员函数的函数称为**全局函数**（global functions）。

`<cmath>`头文件定义了许多全局函数（位于 `std` 命名空间）来执行常见的数学计算。例如：

```
sqrt(900.0)
```

它计算 `900.0` 的平方根并返回结果，即 `30.0`。`sqrt` 函数接受一个 `double` 实参，并返回一个 `double` 结果。调用 `sqrt` 函数前不需要创建任何对象。调用这种函数时，只需指定函数名称，后跟包含实参的圆括号。下表总结了一些常用的数学库函数。`x` 和 `y` 变量是 `double`

类型。

函数	说明	示例
ceil(x)	将 x 取整为不小于 x 的最小整数(向上取整)	ceil(9.2) 结果是 10.0，ceil(-9.8)结果是-9.0
cos(x)	x 的余弦(x 的单位是弧度)	cos(0.0)结果是 1.0
exp(x)	指数函数e^x	exp(1.0)结果是 2.718282，exp(2.0)结果是 7.389056
fabs(x)	x 的绝对值	fabs(5.1) 结果是 5.1，fabs(0.0) 结果是 0.0，fabs(-8.76)结果是 8.76
floor(x)	将 x 取整为不大于 x 的最大整数(向下取整)	floor(9.2) 结果是 9.0，floor(-9.8)结果是-10.0
fmod(x, y)	作为浮点数的 x/y 的余数	fmod(2.6, 1.2)结果是 0.2
log(x)	x 的自然对数(以 e 为底)	log(2.718282)结果是 1.0，log(7.389056)结果是 2.0
log10(x)	x 的对数(以 10 为底)	log10(10.0) 结果是 1.0，log10(100.0)结果是 2.0
pow(x, y)	x 的 y 次方(x^y)	pow(2, 7) 结果是 128，pow(9, .5)结果是 3
sin(x)	x 的正弦(x 的单位是弧度)	sin(0.0)结果是 0
sqrt(x)	x 的平方根(x 非负)	sqrt(9.0)结果是 3.0
tan(x)	x 的正切(x 的单位是弧度)	tan(0.0)结果是 0

C++11 引入的数学函数

C++11 在<cmath>头文件中引入了几十个新的数学函数。有的全新，有的是现有函数的扩充版本，提供对 float 或 long double 实参的支持（之前的版本只支持 double）。例如，双参数的 hypot 函数可以计算直角三角形的斜边。C++17 新增了一个三参数版本的 hypot 来计算三维空间中的斜边。以下网址提供了<cmath>头文件定义的所有函数的一个完整列表：

https://zh.cppreference.com/w/cpp/numeric/math

还可以参考 C++标准中关于浮点类型的数学函数的小节：

https://timsong-cpp.github.io/cppwp/n4861/c.math

C++20 新增的数学常数和<numbers>头文件

20 在 C++20 之前，C++没有内建对常用的数学常数的支持。某些 C++实现通过预处理器宏[①]定义了`M_PI`（代表π）和`M_E`（代表e）以及其他数学常数。预处理器执行时，会用`double`浮点值替换这些宏名称。遗憾的是，并不是所有 C++实现都提供了这些预处理器宏。C++20 新`<numbers>`头文件[②]对许多科学和工程应用程序中常用的数学常数进行了标准化，其中包括：

常数	数学表达式	常数	数学表达式
`numbers::e`	e	`numbers::inv_sqrtpi`	$\frac{1}{\sqrt{\pi}}$
`numbers::log2e`	$\log_2 e$	`numbers::sqrt2`	$\sqrt{2}$
`numbers::log10e`	$\log_{10} e$	`numbers::sqrt3`	$\sqrt{3}$
`numbers::ln2`	$\log_e(2)$	`numbers::inv_sqrt3`	$\frac{1}{\sqrt{3}}$
`numbers::ln10`	$\log_e(10)$	`numbers::egamma`	欧拉-马歇罗尼常数γ
`numbers::pi`	π	`numbers::phi`	$\frac{(1+\sqrt{5})}{2}$
`numbers::inv_pi`	$\frac{1}{\pi}$		

C++17 数学特殊函数

17 C++17 在`<cmath>`头文件中为工程和科学界新增了几十个**数学特殊函数**（mathematical

[①] 本书放在网上的附录 D 讨论了预处理器和宏的问题。

[②] Lev Minkovsky 和 John McFarlane，“Math Constants”， https://wg21.link/p0631r8。

special functions）[①]。可以访问 cppreference.com 查看完整列表和每个函数的简单例子[②]。下表中的每个函数都有接受 `float`、`double` 和 `long double` 参数的版本。

C++17 数学特殊函数	
连带拉盖尔多项式	不规则变形圆柱贝塞尔函数
连带勒让德多项式	圆柱诺依曼函数
beta 函数	指数积分
第一类完全椭圆积分	埃尔米特多项式
第一类不完全椭圆积分	勒让德多项式
第二类完全椭圆积分	拉盖尔多项式
第二类不完全椭圆积分	黎曼 zeta 函数
第三类完全椭圆积分	第一类球面贝塞尔函数
第三类不完全椭圆积分	球面连带勒让德函数
规则变形圆柱贝塞尔函数	球面诺依曼函数
第一类圆柱贝塞尔函数	

5.4 函数定义和函数原型

下面创建一个名为 `maximum` 的用户自定义函数，返回其三个 `int` 实参中最大的一个。图 5.1 的程序执行时，`main` 从用户处读取三个整数。然后，第 16 行调用 `maximum`（在第 20 行~第 34 行定义）。在第 33 行，`maximum` 函数将最大值返回给它的调用者——在本例中，第 16 行直接显示返回的值。

[①] Walter E. Brown，Axel Naumann 和 Edward Smith-Rowland，“Mathematical Special Functions for C++17, v5”，February 29, 2016。https://wg21.link/p0226r1。

[②] “数学特殊函数”，https://zh.cppreference.com/w/cpp/numeric/special_functions。

```cpp
// fig05_01.cpp
// maximum function with a function prototype.
#include <iostream>
#include <iomanip>
using namespace std;

int maximum(int x, int y, int z); // function prototype

int main() {
   cout << "Enter three integer values: ";
   int int1, int2, int3;
   cin >> int1 >> int2 >> int3;

   // invoke maximum
   cout << "The maximum integer value is: "
      << maximum(int1, int2, int3) << '\n';
}

// returns the largest of three integers
int maximum(int x, int y, int z) {
   int maximumValue{x}; // assume x is the largest to start

   // determine whether y is greater than maximumValue
   if (y > maximumValue) {
      maximumValue = y; // make y the new maximumValue
   }

   // determine whether z is greater than maximumValue
   if (z > maximumValue) {
      maximumValue = z; // make z the new maximumValue
   }

   return maximumValue;
}
```

```
Enter three integer grades: 86 67 75
The maximum integer value is: 86
```

```
Enter three integer grades: 67 86 75
The maximum integer value is: 86
```

```
Enter three integer grades: 67 75 86
The maximum integer value is: 86
```

图 5.1 maximum 函数的原型和定义

maximum 函数

通常，函数定义的第一行要指定其返回类型、函数名和包含参数列表的一对圆括号。**参数列表**（parameter list）指定函数执行任务所需的任何额外信息。函数的第一行也称为函数的**头**（header）。参数列表可以包含零个或多个**参数**（parameters，也称为**形参**），每个参数都用一个类型和一个名称声明。两个或多个参数用逗号分隔的列表指定。maximum 函数有三个 int 参数，分别命名为 x、y 和 z。调用函数时，每个参数都从函数调用中接收相应的**实参**（argument）值。

maximum 函数首先假定参数 x 包含最大值，所以第 21 行将 maximumValue 初始化为 x 的值。当然，参数 y 或 z 也可能包含最大值，所以它们分别与 maximumValue 进行比较。第 24 行~第 26 行判断 y 是否大于 maximumValue；如果是，就将 y 赋给 maximumValue。第 29 行~第 31 行判断 z 是否大于 maximumValue，如果是，就将 z 赋给 maximumValue。现在，maximumValue 包含的是真正的最大值，所以第 33 行向调用者返回该值的拷贝。

maximum 的函数原型

函数在使用之前必须定义或声明。第 7 行声明了 maximum 函数：

```
int maximum(int x, int y, int z); // function prototype
```

这个**函数原型**（function prototype）描述了 maximum 的接口，同时没有公开它的内部实现细节。函数原型告诉编译器函数的名称、它的返回类型和参数类型，而且必须以一个分号（;）结束。第 7 行表示 maximum 返回一个 int，并需要三个 int 参数来执行其任务。函数原型中的类型必须与函数定义头中的类型一致（第 20 行）。函数原型的参数名称通常与函数定义中的参数名称一致，但并不强求。

函数原型中的参数名称

函数原型中的参数名称是可选的（编译器会忽略它们），但建议加上这些名称以便编写文档。

编译器用 maximum 的函数原型做什么？

在编译程序时，编译器使用该原型来做下面这些事情：

- 检查 maximum 函数头（第 20 行）是否与其原型（第 7 行）相符。
- 检查对 maximum 的调用（第 16 行）是否包含正确的实参数量和实参类型，而且实参的类型是否有正确的顺序（在本例中，所有实参都是同一类型）。
- 检查函数的返回值是否能在调用函数的表达式中正确使用。例如，如果函数声明了

void 返回类型（因而不会返回值），不能在期望获得一个值的地方调用它，例如在赋值操作符的右侧或者 cout 语句中。

- 检查每个实参是否与对应的形参类型一致。例如，一个 double 类型的参数可以接收 7.35、22 或者-0.03456 这样的值，但不能接收 "hello"这样的字符串。如果传给函数的实参与函数原型中指定的形参类型不一致，编译器会尝试将实参转换为要求的类型。5.6 节讨论了这个转换过程以及如果无法转换会发生什么。

⊗错误 如果函数原型、函数定义头和函数调用在参数数量/类型/顺序以及函数的返回类型上不一致，就会发生编译错误。

将控制从函数返回给调用者

当程序调用一个函数时，该函数会执行其任务，然后将控制（可能还有一个值）返回给函数的调用位置。在不返回值的函数中（即返回类型为 void），当程序遇到函数的结束右大括号时，控制就会返回。除此之外，在这种函数中，还可以在函数主体的任何地方执行 return;来显式地将控制返回给调用者。

5.5 函数实参的求值顺序

图 5.1 第 16 行以 maximum 函数的实参进行分隔的逗号不是我们之前说过的逗号操作符。逗号操作符保证其操作数从左到右计算。但是，C++标准并没有规定函数实参的求值顺序。因此，不同的编译器可以按照不同的顺序对传给函数的实参进行求值。

有的时候，当一个函数的实参是某个表达式时（例如对其他函数的调用），编译器对实参进行求值的顺序就可能影响一个或多个实参的值。如果不同编译器使用了不同的求值顺序，传给函数的实参值就有可能出现区别，从而导致难以发现的逻辑错误。

如果对函数实参的求值顺序以及该顺序是否会影响传给函数的值有疑虑，可以在调用函数前将实参赋给变量，再将这些变量作为实参传给函数。

5.6 关于函数原型和实参强制类型转换的注意事项

⊗错误 除非函数在使用前定义，否则必须列出函数的原型。如果函数在调用前就定义好了，那么它的定义也可以作为函数原型，所以不需要单独列出原型。如果一个函数在调用的时候既没有定义，也没有原型，就会发生编译错误。

使用像 sqrt 这样的标准库函数时，由于不能访问它的定义，所以在调用该函数之前不能在自己的代码中定义它。相反，必须包含该函数原型所在的头文件（本例是<cmath>）。虽然有函数定义就可以省略函数原型（前提是先定义再调用），但提供原型可避免将你的代

码与函数的定义顺序绑定，而函数的定义顺序很容易随着程序的演进而发生变化。

5.6.1 函数签名和函数原型

函数的名称及其参数类型统称为**函数签名**（function signature），或者简单地称为**签名**（signature）。函数的返回类型不是函数签名的一部分。函数的作用域是程序中知道该函数的存在，并可访问该函数的区域。同一作用域内的每个函数都必须有唯一的签名。我们将在第 5.10 节详细讨论作用域。

在图 5.1 中，如果将第 7 行的函数原型写成：

```
void maximum(int x, int y, int z);
```

错误 那么编译器会报错，因为原型的 `void` 返回类型和函数头的 `int` 返回类型不一致。类似地，这样的一个原型会导致以下语句产生编译错误，因为它依赖 `maximum` 返回一个供显示的值：

```
cout << maximum(6, 7, 0);
```

错误 函数原型可以帮助你在编译时发现错误，这总比在运行时发现错误好。

5.6.2 实参强制类型转换

函数原型的一个重要特征是**实参强制类型转换**（argument coercion）——也就是将实参的类型强制转换为参数（形参）声明所指定的恰当类型。例如，虽然函数原型指定的是 double 参数，但程序在调用该函数时可以传递一个整型实参。只要发生的不是收缩转换（参见 3.8.3 节），函数就仍然能正确工作。如果函数调用中的实参不能被隐式转换为函数原型中指定的预期类型，就会发生编译错误。

5.6.3 实参提升规则和隐式转换

有的时候，和函数原型中的参数类型不完全一致的实参值可能在函数被调用之前由编译器转换为适当的类型。这种转换根据的是 C++**提升规则**（promotion rules），它们规定了基本类型之间允许的隐式转换[①]。`double` 也能转换为 `int`，但这种收缩转换会截断 `double` 值的小数部分[②]。记住，`double` 变量可以容纳比 `int` 变量大得多的数字，所以收缩转换时发生数据损失的可能是相当大的。

[①] 除了这里讨论的之外，还有其他一些提升和转换规则。详情参考 C++标准的 7.3 节和 7.4 节。https://timsong-cpp.github.io/cppwp/n4861/conv 和 https://timsong-cpp.github.io/cppwp/n4861/expr.arith.conv。

[②] 3.8.3 节说过，C++11 大括号初始化器不允收缩转换。

将大整型转换为小整型（例如，long 转换成 short），有符号类型转换为无符号类型，或者无符号类型转换为有符号类型时，值也可能被修改。unsigned 整型变量可表示从 0 到大约两倍于对应的有符号整数类型的正值范围的值。unsigned 类型主要用于二进制位操作（网上附录 E）。“C++核心准则”指出，使用 unsigned 类型并不能真正消除负值的可能性[①]。

核心准则

提升规则也适用于**混合类型表达式**（mixed-type expressions），其中的值具有两个或多个数据类型。每个值的类型都被提升到表达式的“最高”类型。提升时使用的是每个值的临时拷贝——原始值保持不变。下表按照从“最高类型”到“最低类型”的顺序列出了数据类型。

数据类型	
long double	
double	
float	
unsigned long long int	(等同于 unsigned long long)
long long int	(等同于 long long)
unsigned long int	(等同于 unsigned long)
long int	(等同于 long)
unsigned int	(等同于 unsigned)
int	
unsigned short int	(等同于 unsigned short)
short int	(等同于 short)
unsigned char	
char and signed char	
Bool	

[①] C++ Core Guidelines，“ES.106: Don't Try to Avoid Negative Values By Using unsigned”，https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-nonnegative。

转换可能造成不正确的值

将数值转换为“较低”的类型可能因为收缩转换而导致错误或警告。如果将 double 实参传给一个指定了 int 参数的 square（平方） 函数，该实参会被转换为 int（一个较低的类型，因此是收缩转换），而 square 可能返回一个不正确的值。例如，square(4.5)将返回 16，而不是 20.25。一些编译器会对此发出警告。例如，Microsoft Visual C++显示的警告是：

```
“参数”：从“double”转换到“int”，可能丢失数据
```

配合“准则支持库”的收缩转换

核心准则

如果必须进行显式的收缩转换，“C++核心准则”推荐使用“准则支持库”（GSL）中的 narrow_cast[①]。这个库有几个实现。其中，Microsoft 的开源版本已在众多平台/编译器组合上进行了测试，其中包括我们首选的三个编译器和平台。可从以下网址下载 GSL：

https://github.com/Microsoft/GSL

为方便起见，我们在本书配套资源的 libraries/GSL 子文件夹中提供了该 GSL。

GSL 是一个单文件（header-only）库，所以在程序中包含头文件<gsl/gsl>即可直接使用。当然，必须先将你的编译器指向 GSL 文件夹的 include 子文件夹，这样编译器才知道在哪里可以找到头文件，具体做法可参考 3.12 节末尾使用 BigNumber 类时的操作。以下语句使用一个 narrow_cast (来自命名空间 gsl) 来将 double 值 7.5 转换为 int 值 7。

```
gsl::narrow_cast<int>(7.5)
```

圆括号中的值被强制转换为尖括号（<>）中指定的类型。

5.7 C++标准库头文件

C++标准库被划分为许多头文件。每个都包含该头文件中相关函数的函数原型。头文件还包含各种类（class）类型和函数的定义，以及这些函数所需的常量。头文件告诉编译器如何与库和用户编写的组件建立接口。

下表列出了一些常用的 C++标准库头文件，其中许多都会在本书后面讨论。注意，这个表格中的“宏”将在网上附录 D“预处理器”中详细讨论。96 个 C++20 标准库头文件的完整列表请参考：

[①] C++ Core Guidelines，http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-narrowing。

https://zh.cppreference.com/w/cpp/header

注意，这个网页列出了30多个额外的、标记为弃用（deprecated）或移除（removed）头文件。废弃的头文件不应再用，而移除的头文件则不再包含在C++标准库中。

标准库头文件	**说明**
<iostream>	C++标准输入和输出函数的函数原型，已在第 2 章介绍，在网上的第19章进行了更详细的说明。
<iomanip>	用于格式化数据流的流操纵元（stream manipulators）的函数原型。这个头文件首次在3.7节使用，并在网上的第19章进行了详细讨论。
<cmath>	数学库函数的函数原型（5.3节）。
<cstdlib>	数字到文本转换、文本到数字转换、内存分配、随机数和其他各种实用函数的函数原型。头文件的部分内容将在 5.8节、第11章和第12章讨论。
<random>	C++11 的随机数生成功能（会在本章讨论，并会在后续章节中使用）。
<ctime>，<chrono>	定义了用于处理时间和日期的函数原型和类型。5.8 节使用了<ctime>， <chrono>自C++11引入，并在C++20中得到了增强。第17章使用了几个<chrono>计时功能。
<array>，<vector>，<list>，<tuple>，<forward_list>，<deque>，<queue>，<stack>，<map>，<unordered_map>，<unordered_set>，<set>，<bitset>	这些头文件包含实现C++标准库容器的类。容器（containers）是常用数据结构的标准实现。<array>和<vector>头文件首次在第6章讨论。第13章将讨论所有这些头文件。<array>、<forward_list>、<tuple>、<unordered_map>和<unordered_set>是自C++11引入的。
<cctype>	用于测试字符的某些属性（例如字符是否为数字或标点符号）的函数原型，还有将小写字母转换成大写字母（或相反）的函数原型。
<cstring>	C字符串处理函数原型。
<typeinfo>	用于运行时类型识别的类（在程序执行时确定数据类型）。

<exception>，<stdexcept>	异常处理类（第 12 章）。
<memory>	用于管理内存分配的类和函数。这个头文件将在第 12 章使用。
<fstream>	磁盘文件输入/输出函数原型（第 8 章）。
<string>	string 类以及 getline 和 to_string 函数的定义（后者在第 8 章讨论）。
<sstream>	从内存中的字符串输入和向内存中的字符串输出的函数原型（第 8 章）。
<functional>	C++标准库算法所用的类和函数。这个头文件将在第 14 章使用。
<iterator>	用于访问 C++标准库容器中的数据的类。这个头文件将在第 13 章使用。
<algorithm>	对 C++标准库容器中的数据进行处理的函数。这个头文件将在第 13 章使用。
<cassert>	用于添加有助于程序调试的诊断机制的宏。这个头文件在网上附录 D 中使用。
<cfloat>	系统的浮点大小限制。
<climits>	系统的整数大小限制。
<cstdio>	C 标准输入/输出库函数原型。
<locale>	流处理常用的类和函数，以不同语言的自然形式处理数据（例如，货币格式、字符串排序、字符呈现）。
<limits>	用于定义每种计算机平台上的数值数据类型限制的类——这是 C++的<climits>和<cfloat>版本。
<utility>	许多 C++标准库头文件都要用到的类和函数。
<thread>，<mutex>，<shared_mutex>，<future>，<condition_variable>	在 C++11 和 C++14 中加入的多线程应用程序开发功能，使你的应用能够利用多核处理器的优势（第 17 章）。

C++17 引入的一些关键头文件	
<any>	用于容纳任何可拷贝类型的单个值的类。
<optional>	用于表示一个对象的模板，该对象可能有、也可能没有值（第 13 章）。
<variant>	用于创建和操作指定类型集合的对象的特性（第 10 章）。
<execution>	和标准模板库的并行算法一起使用的特性（第 17 章）。
<filesystem>	与本地文件系统的文件和文件夹进行交互。
C++20 引入的一些关键的头文件	
<concepts>	限制可与模板一起使用的类型（第 15 章）。
<coroutine>	用协程进行异步编程（第 17 章）
<compare>	用于支持新的三路比较操作符<=>（第 11 章）。
<format>	支持新的、简洁和强大的文本格式化功能（贯穿全书）。
<ranges>	为函数式编程提供支持（第 6 章和第 13 章）。
	将视图创建为连续的对象序列的能力（第 17 章）。
<bit>	支持标准化的位操作。
<stop_token>，<semaphore>，<latch>，<barrier>	提供额外的功能来支持 C++11 和 C++14 引入的多线程应用开发特性（第 17 章）。

5.8 案例研究：随机数生成

现在让我们放松一下，来看看一种流行的编程应用：模拟和玩游戏。本节和下一节将开发一个用到了多个函数的游戏程序。本节的示例程序输出是用 Visual Studio 生成的。在其他编译器和平台上，输出可能会有所不同。

<random>头文件

利用 C++11 <random>头文件提供的功能，可以将概率元素引入你的应用程序中。这些功能取代了被弃用的 rand 函数，后者是 C++从 C 语言标准库继承的。rand 函数不具备“良好

的统计特征”，是可以预测的[1]。所以，凡是使用了 rand 的程序都不太安全。

安全性 C++11 提供了一个更安全的随机数功能库，可以生成非确定性的随机数——一组无法预测的随机数。这种随机数生成器应用于那些不容许预测的模拟和安全场景。

随机数生成是一个复杂的主题，数学家们为此开发了许多具有不同统计特征（statistical properties）的算法。为了在程序中灵活地使用随机数，C++11 提供了许多类来表示各种**随机数生成引擎**（random-number generation engines）和**分布**（distributions）：

- **引擎**实现了生成随机数的一种随机数生成算法。
- **分布**控制引擎所生成的数值的范围、数值的类型（如 int、double 等）以及数值的统计特征。

在本例中，我们使用的是默认的随机数生成引擎 default_random_engine。分布使用的则是 uniform_int_distribution，它将随机整数均匀分布在一个指定的范围内。默认范围是从 0 到你使用的平台所支持的最大 int 值。如果 default_random_engine 和 uniform_int_distribution 真的能随机生成整数，那么每次程序请求随机数时，0 到最大 int 值之间的每个数字都有均等的机会（或概率）被选中。

default_random_engine 直接生成的数值范围往往有别于应用程序的具体要求。例如，模拟抛硬币的程序可能只需要 0 代表“正面”，1 代表“反面”。而模拟掷六面骰子的程序需要 1~6 的随机整数。另外，假定总共有 4 种类型的飞船，程序随机预测游戏中从远处飞过的下一种飞船，那么需要的是 0~3 的随机整数。为了指定取值范围，应用程序需要用目标范围的起始值和终止值来初始化 uniform_int_distribution。

要获得引擎和分布的完整列表，请访问：

https://zh.cppreference.com/w/cpp/header/random

要更多地了解 C++11 随机数生成功能，请参考白皮书“Random Number Generation in C++11”[2]。

5.8.1 掷六面骰子

图 5.2 模拟并显示将一颗六面骰子掷 10 次的结果。第 9 行创建一个名为 engine 的

---

[1] Fred Long，“Do Not Use the rand() Function for Generating Pseudorandom Numbers”，最后由 Jill Britton 修改于 2021 年 11 月 30 日，https://tinyurl.com/ycxrvd6y。

[2] Walter E. Brown，“Random Number Generation in C++11”，March 12, 2013，https://isocpp.org/files/papers/n3551.pdf。

default_random_engine 对象来生成随机数。第 12 行用 {1, 6} 来初始化 uniform_int_distribution 对象 randomDie，指定要生成的是范围在 1 到 6 之间的 int 值。第 16 行的表达式 randomDie(engine)返回 1~6 的一个随机 int。

```
1  // fig05_02.cpp
2  // Producing random integers in the range 1 through 6.
3  #include <iostream>
4  #include <random> // contains C++11 random-number generation features
5  using namespace std;
6
7  int main() {
8     // engine that produces random numbers
9     default_random_engine engine{};
10
11    // distribution that produces the int values 1-6 with equal likelihood
12    uniform_int_distribution randomDie{1, 6};
13
14    // display 10 random die rolls
15    for (int counter{1}; counter <= 10; ++counter) {
16       cout << randomDie(engine) << " ";
17    }
18
19    cout << '\n';
20 }
```

```
3 1 3 6 5 2 6 6 1 2
```

图 5.2 生成 1~6 的随机整数

5.8.2 六面骰子掷 6000 万次

为了证实上一个程序生成的随机数的出现概率大致均等，图 5.3 模拟了 6000 万次掷骰子的过程[①]。范围在 1 到 6 之间的每个 int 都应出现大约 1000 万次（掷六次出一次）。程序的输出证实了这一点。在 switch 的初始化器中（第 23 行），我们在 face 变量的定义前附加了一个 const。对于任何初始化后不应改变的变量来说，这是一个好的做法。用 const 声明后，如果不小心修改了这个变量，编译器就会报错。

[①] 1976 年，当本书作者之一 Harvey Deitel 首次在他的课堂上实现这个例子时，只掷了 600 次骰子——如果掷 6000 次，花费的时间太长了。如今，在我们的系统上，这个程序花了大约 5 秒钟就完成了 6000 万次掷骰子！如果掷 6 亿次，则大概要花一分钟。该程序掷骰子的过程是顺序的。以后讨论到异步编程时，会探讨如何将应用程序并行化，以利用当今多核计算机的优势。

```cpp
// fig05_03.cpp
// Rolling a six-sided die 60,000,000 times.
#include <fmt/format.h>
#include <iostream>
#include <random>
using namespace std;

int main() {
   // set up random-number generation
   default_random_engine engine{};
   uniform_int_distribution randomDie{1, 6};

   int frequency1{0}; // count of 1s rolled
   int frequency2{0}; // count of 2s rolled
   int frequency3{0}; // count of 3s rolled
   int frequency4{0}; // count of 4s rolled
   int frequency5{0}; // count of 5s rolled
   int frequency6{0}; // count of 6s rolled

   // summarize results of 60,000,000 rolls of a die
   for (int roll{1}; roll <= 60'000'000; ++roll) {
      // determine roll value 1-6 and increment appropriate counter
      switch (const int face{randomDie(engine)}) {
         case 1:
            ++frequency1; // increment the 1s counter
            break;
         case 2:
            ++frequency2; // increment the 2s counter
            break;
         case 3:
            ++frequency3; // increment the 3s counter
            break;
         case 4:
            ++frequency4; // increment the 4s counter
            break;
         case 5:
            ++frequency5; // increment the 5s counter
            break;
         case 6:
            ++frequency6; // increment the 6s counter
            break;
         default: // invalid value
            cout << "Program should never get here!";
            break;
      }
   }

   cout << fmt::format("{:>4}{:>13}\n", "Face", "Frequency"); // headers
   cout << fmt::format("{:>4d}{:>13d}\n", 1, frequency1)
      << fmt::format("{:>4d}{:>13d}\n", 2, frequency2)
      << fmt::format("{:>4d}{:>13d}\n", 3, frequency3)
      << fmt::format("{:>4d}{:>13d}\n", 4, frequency4)
      << fmt::format("{:>4d}{:>13d}\n", 5, frequency5)
      << fmt::format("{:>4d}{:>13d}\n", 6, frequency6);
}
```

```
Face    Frequency
   1      9997896
   2     10000608
   3      9996800
   4     10000729
   5     10003444
   6     10000523
```

图 5.3 六面骰子掷 6000 万次

switch 的 default case（第 42 行~第 44 行）应该永远都执行不到，因为控制表达式（face）的值总是在 1 到 6 的范围内。许多程序员在每个 switch 语句中都提供了一个 default case 以捕捉可能发生的错误——即使他们觉得自己的程序没有错误。第 6 章在介绍了数组之后，我们将展示如何用一个单行语句优雅地取代图 5.3 中的整个 switch。

5.8.3 为给随机数生成器提供种子

再次执行图 5.2 的程序，我们得到了以下结果：

```
3 1 3 6 5 2 6 6 1 2
```

还是和图 5.2 一样的输出。这算哪门子的“随机”数？default_random_engine 生成的实际上是**伪随机数**（pseudorandom numbers）。重复执行图 5.2 和 5.3 的程序，生成的是貌似随机的数字序列。每次执行这些程序，这些序列实际上都会重复。对模拟程序进行调试时，一定要检查随机数的序列是不是在重复。

一旦彻底调试了你的模拟程序的输出，就可以为它设定条件，让它每次执行时生成不同的随机数序列，即所谓的**随机化**（randomizing）。为此，要用一个 unsigned int 实参初始化 default_random_engine，该参数为 default_random_engine 提供**种子**（seed），使其在每次执行时产生一个不同的随机数序列。

为 default_random_engine 提供种子

图 5.4 的程序演示了如何用通过键盘输入的一个 unsigned int（第 12 行）为 default_random_engine 提供种子（第 15 行）。每次输入一个不同的种子，程序都会生成一个不同的随机数序列。我们在第一个和第三个示例输出中使用了相同的种子，所以每次都输出同一个 10 数字的序列。为安全性起见，必须确保程序每次执行时都为随机数生成器提供不同的种子（只需提供一次）；否则，攻击者就能预知即将生成的（伪）随机数序列。

安全性

```cpp
// fig05_04.cpp
// Randomizing the die-rolling program.
#include <iostream>
#include <iomanip>
#include <random>
using namespace std;

int main() {
   unsigned int seed{0}; // stores the seed entered by the user

   cout << "Enter seed: ";
   cin >> seed;

   // set up random-number generation
   default_random_engine engine{seed}; // seed the engine
   uniform_int_distribution randomDie{1, 6};

   // display 10 random die rolls
   for (int counter{1}; counter <= 10; ++counter) {
      cout << randomDie(engine) << " ";
   }

   cout << '\n';
}
```

```
Enter seed: 67
6 2 5 6 6 2 2 6 2 1
```

```
Enter seed: 432
3 5 6 1 6 1 4 4 2 2
```

```
Enter seed: 67
6 2 5 6 6 2 2 6 2 1
```

图 5.4 对掷骰子程序进行随机化

5.8.4 用 random_device 为随机数生成器提供种子

为了实现随机化，正确方式是像图 5.5 那样使用 C++11 引入的 random_device 对象（来自头文件<random>）作为随机数生成器引擎的种子。random_device 生成均匀分布的、非确定性的随机整数，这些整数是无法预测的[1]。但是，random_device 的文档指出，出于性能方面的原因，它通常只用于为随机数生成引擎提供种子。另外，random_device 在某些平

安全性
性能
安全性

[1] https://en.wikipedia.org/wiki/Nondeterministic_algorithm

台上可能是确定性的[1]。因此，在安全应用中依赖它之前，一定要检查你的编译器的文档。例如，Visual C++的实现就提供了非确定性的、密码安全（cryptographically secure）的随机数。

5.9 案例研究：概率游戏；介绍有作用域的 enum

最流行的概率游戏（game of chance）之一是“掷骰子”（craps），它在全世界的范围内都比较流行。游戏规则很简单：

玩家掷两颗骰子。每颗骰子六个面，点数分别是 1，2，3，4，5 和 6 点。骰子静止后，计算两个向上的面的点数之和。如果第一次掷出的点数是 7 或 11，玩家胜。如果第一次掷出的点数是 2、3 或 12（称为“craps”），玩家输（庄家胜）。如果第一次掷出的点数是 4、5、6、8、9 或 10，那么这个点数就成为“你的点”（point）。要想获胜，必须继续掷骰子，直到再次掷出“你的点”（make your point）。如果玩家在掷出“你的点”之前掷出 7，玩家输。

在规则中，注意玩家第一次和之后都要掷两颗骰子。我们定义一个 `rollDice` 函数来模拟掷骰子的动作，计算并显示两颗骰子的点数之和。该函数可被多次调用。第一次掷骰子需要调用它。如果玩家在第一次掷骰子后既没赢也没输，则可能被调用更多次。下面是几个示例输出，它们显示的是：

- 第一次掷出 7 点，玩家赢。
- 第一次掷出 6 点，成为“你的点”，然后再多掷几次，在掷出 7 点前再次掷出 6 点，玩家赢。
- 第一次掷出 12 点，玩家输。
- 在再次掷出“你的点”之前掷了一个 7 点，玩家输。

[1] https://zh.cppreference.com/w/cpp/numeric/random/random_device

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

实现游戏

掷骰子游戏（图 5.5）使用两个函数（`main` 和 `rollDice`）以及 `switch`、`while`、`if…else` 和嵌套 `if…else` 语句来模拟游戏。`rollDice` 函数的原型（第 8 行）规定函数不获取任何实参（一对空的圆括号）并返回一个 `int`（两颗骰子点数之和）。

```
1  // fig05_05.cpp
2  // Craps simulation.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <random>
6  using namespace std;
7
8  int rollDice(); // rolls dice, calculates and displays sum
9
```

图 5.5 模拟掷骰子

C++11 有作用域的 enum

11 玩家第一次掷骰子或随后任何一次掷骰子，都可能赢，也可能输。程序通过 gameStatus 变量来跟踪输赢，第 15 行将该变量声明为新的 Status 类型。第 12 行声明由用户定义的一个有作用域的枚举，该枚举类型由关键字 enum class 引入，后跟类型名称（Status）和一组代表整数常量的标识符。

```
10  int main() {
11     // scoped enumeration with constants that represent the game status
12     enum class Status {keepRolling, won, lost};
13
14     int myPoint{0}; // point if no win or loss on first roll
15     Status gameStatus{Status::keepRolling}; // game is not over
16
```

这些**枚举常量**（enumeration constants）的基础值是 int 类型的，从 0 开始，默认情况下以 1 递增。在 Status 枚举中，常量 keepRolling（继续掷骰子）值为 0，win（赢）的值为 1，lost（输）的值为 2。enum class 中的标识符必须是唯一的，但多个标识符可以有相同的（整数）值。只能将枚举中声明的常量赋给用户自定义的 Status 类型的变量。

核心准则

按照惯例，enum class 名称的首字母应该大写。如果名称由多个单词组成，则后续每个单词的首字母也应大写（例如，ProductCode）。“C++核心准则”指出，enum class 中的常量应使用和变量相同的命名规则[①②]。

为了引用一个有作用域的枚举常量，要使用枚举类名（即 Status）和作用域解析操作符（::）来限定要访问的常量，如第 15 行所示，该行将 gameStatus 初始化为 Status::keepRolling。如果胜了，程序将 gameStatus 设为 Status::win。如果输了，程序将 gameStatus 设为 Status::lost。

第一次掷骰的输赢

以下 switch 语句判断玩家在第一次掷骰子时是赢还是输。

① C++ Core Guidelines，https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-caps。

② 在遗留 C++代码中经常会看到全部大写的 enum 常量，这个实践目前已弃用。

```
17      // determine game status and point (if needed) based on first roll
18      switch (const int sumOfDice{rollDice()}) {
19         case 7: // win with 7 on first roll
20         case 11: // win with 11 on first roll
21            gameStatus = Status::won;
22            break;
23         case 2: // lose with 2 on first roll
24         case 3: // lose with 3 on first roll
25         case 12: // lose with 12 on first roll
26            gameStatus = Status::lost;
27            break;
28         default: // did not win or lose, so remember point
29            myPoint = sumOfDice; // remember the point
30            cout << fmt::format("Point is {}\n", myPoint);
31            break; // optional (but recommended) at end of switch
32      }
33
```

switch 的初始化器（第 18 行）创建 sumOfDice 变量，并通过调用 rollDice 对其进行初始化。如果掷出的点数是 7 或 11，第 21 行将 gameStatus 设为 Status::win。如果掷出的点数是 2、3 或 12，第 26 行将 gameStatus 设为 Status::lost。对于其他值，gameStatus 保持不变（Status::keepRolling），第 29 行将 sumOfDice 保存到 myPoint 中，第 30 行显示 myPoint。

继续掷骰

第一次掷完骰子后，如果 gameStatus 是 Status::keepRolling，将通过以下 while 语句继续程序执行。

```
34      // while game is not complete
35      while (Status::keepRolling == gameStatus) { // not won or lost
36         // roll dice again and determine game status
37         if (const int sumOfDice{rollDice()}; sumOfDice == myPoint) {
38            gameStatus = Status::won;
39         }
40         else if (sumOfDice == 7) { // lose by rolling 7 before point
41            gameStatus = Status::lost;
42         }
43      }
44
```

17 在每一次循环迭代中，if 语句的初始化器（第 37 行）都调用 rollDice 来生成一个新的 sumOfDice。如果 sumOfDice 与 myPoint 匹配，第 38 行将 gameStatus 设为 Status::won，然后循环结束。如果 sumOfDice 是 7，程序将 gameStatus 设为 Status::lost（第 41 行），然后循环结束。否则，循环将继续执行。

显示玩家是赢还是输

上述循环终止后，程序进入下面的 if...else 语句，如果 gameStatus 是 Status::win，则打印 "Player wins"（玩家赢了）。如果 gameStatus 是 Status::lost，则打印 "Player loses"（玩家输了）。

```
45      // display won or lost message
46      if (Status::won == gameStatus) {
47         cout << "Player wins\n";
48      }
49      else {
50         cout << "Player loses\n";
51      }
52   }
53
```

rollDice 函数

rollDice 函数模拟掷两个骰子（第 61 行~第 62 行），计算点数之和（第 63 行），打印每个骰子掷了多少点和两者之和（第 66 行），并返回点数之和（第 68 行）。

```
54   // roll dice, calculate sum and display results
55   int rollDice() {
56      // set up random-number generation
57      static random_device rd; // used to seed the default_random_engine
58      static default_random_engine engine{rd()}; // rd() produces a seed
59      static uniform_int_distribution randomDie{1, 6};
60
61      const int die1{randomDie(engine)}; // first die roll
62      const int die2{randomDie(engine)}; // second die roll
63      const int sum{die1 + die2}; // compute sum of die values
64
65      // display results of this roll
66      cout << fmt::format("Player rolled {} + {} = {}\n", die1, die2, sum);
67
68      return sum;
69   }
```

一般来说，每个要用到随机数的程序都会创建一个随机数生成器引擎，为它提供种子，然后在整个程序中使用它。本例只有 rollDice 函数需要访问随机数生成器，所以第 57 行~第 59 行将 random_device、default_random_engine 和 uniform_int_distribution 对象定义为**静态局部变量**（static local variables）。其他局部变量在一次函数调用结束后就会消失。相反，静态局部变量会在不同函数调用之间保留其值。将第 57 行~第 59 行的对象声明为 static 可确保它们只在第一次调用 rollDice 时被创建。然后，它们会在后续所有 rollDice 调用中重用。第 58 行的表达式 rd()从 random_device 对象中获得一个非确定的随机整数，并使用它作为 default_random_engine 的种子，使程序在每次执行时产生不同

的结果。

有作用域的 enum 的其他注意事项

用类型名称和::来限定一个 enum class 类型的常量，会将该常量显式限定在该类型的作用域内。如果另一个 enum class 也包含相同的标识符，由于类型名称和::是必须的，所以具体使用的是哪个常量不会出现歧义。一般来说，使用唯一的枚举常量值有助于防止以后难以发现的逻辑错误。

月份是另一种流行的、有作用域的枚举：

```
enum class Month {jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};
```

该语句创建用户自定义的 enum class 类型 Month，其枚举常量代表一年中的各个月份。第一个值被显式设为 1，所以其余的值从 1 开始递增，形成了 1 到 12 的值。任何常量都可以在 enum class 定义中被显式赋予一个整数值。后续常量值总是比上一个大 1，直到被再次显式设置。

C++11 之前的枚举类型

枚举也可以直接用关键字 enum 来定义，后跟类型名称和一组枚举常量。例如：

```
enum Status {keepRolling, won, lost};
```

这些枚举常量是无作用域的，可直接使用 keepRolling、win 和 lost 等名称来引用它们，不需要在之前限定作用域。但是，如果两个或更多的无作用域枚举含有同名枚举常量，就可能导致命名冲突和编译错误。“C++核心准则”建议始终使用 enum class[①]。

核心准则

C++11：指定枚举常量的类型

枚举常量具有整数值。对于无作用域的 enum，其基础整型取决于它的常量的值，并且保证足够大以存储它们。而对于有作用域的 enum，其基础整型是 int，但可以通过在类型名称后面加上冒号（:）和一个类型名称来指定其他整型。例如，可用以下语句将 enum class Status 中的常量类型指定为 short。

```
enum class Status : short {keepRolling, won, lost};
```


[①] C++ Core Guidelines，“Enum.3: Prefer Class enums over 'Plain' enums”，https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-class。

C++20：使用 enum 声明

如果一个 enum class 的常量类型依据其使用上下文是显而易见的（例如我们的掷骰子例
20 子），那么 C++20 的 using enum 声明允许引用一个 enum class 的常量，而不必附加类型名称和作用域解析操作符（::）前缀[①②]。例如，可在 enum class 声明后添加以下语句：

```
using enum Status;
```

它允许程序的剩余部分直接使用 keepRolling、won 和 lost，而不必分别写成 Status::keepRolling、Status::won 和 Status::lost。另外，使用以下形式的 using 声明，还可以直接使用一个单独的 enum class 常量：

```
using Status::keepRolling;
```

这样就能在不加 Status::限定符的前提下直接使用 keepRolling。通常，应该将这种 using 声明放在使用它们的那个代码块中。

5.10 作用域规则

一个标识符能在程序的什么部分使用，这一部分就是该标识符的**作用域**（scope）。例如，在代码块中声明了一个局部变量后，它的作用域就被限制到：

- 从声明位置到该代码块的末尾；
- 声明位置之后的那个代码块中的嵌套块。

本节讨论了块作用域和全局命名空间作用域。函数原型中的参数名具有**函数参数作用域**（function parameter scope），仅在该原型中可见。以后还会讨论其他作用域，包括第 9 章的**类作用域**（class scope）和网上第 20 章的**函数作用域**（function scope）和**命名空间作用域**（namespace scope）。

块作用域

代码块中声明的标识符具有**块作用域**（block scope），它始于标识符的声明位置，终于代码块的结束右大括号（}）。局部变量具有块作用域，函数的参数（形参）亦是如此。任何块都可以包含变量声明。在嵌套块中，如果外层块中的标识符与内层块中的标识符同名，那么外层块中的标识符将被“隐藏”，直到内层块结束。内层块“看到”的是自己的局部

---

① Gašper Ažman 和 Jonathan Müller，“Using Enum”，July 16, 2019，https://wg21.link/p1099r5。

② 本书写作时，这一特性仅 Microsoft Visual C++编译器支持。

变量的值，而不是包围它的那个块的同名变量的值。如果不小心为内层块中的标识符使用了和外层块的标识符相同的名称，而事实上你想让外层块中的标识符延续到内层块，那么通常都是一个逻辑错误。要注意避免内层作用域中的变量名称“隐藏”外层作用域中的名称。大多数编译器都会警告你这一点。

如图 5.5 所示，局部变量也可以声明为 `static`（静态）。这样的变量也具有块作用域，但和其他局部变量不同的是，静态局部变量在函数返回给调用者后仍然保留其值。下次调用该函数时，静态局部变量包含的是它在该函数上次执行后的值。以下语句声明一个静态局部变量 `count` 并将其初始化为 `1`：

```
static int count{1};
```

数值类型的静态局部变量默认初始化为零——虽然你应该尽量显式初始化。对于非基本类型变量，其默认初始化视类型而定。例如，`string` 变量的默认值是空字符串（`""`）。我们将在以后的章节中更多地讨论默认初始化的问题。

全局命名空间作用域

如果声明的一个标识符不在任何函数或类的内部，该标识符就具有**全局命名空间作用域**（global namespace scope）。这种标识符一经声明，就被之后的所有函数所知。函数定义、放在函数外部的函数原型、类定义和全局变量均具有全局命名空间作用域。**全局变量**（global variables）是在所有类或函数定义的外部声明的。这种变量在程序执行期间会一直保留其值。

但是，如果将一个变量声明为全局而不是局部，那么当一个不需要访问该变量的函数意外或恶意地修改它时，就会产生意想不到的**副作用**（side effects）。除非是真正的全局资源，例如 `cin` 和 `cout`，否则应避免使用全局变量。这是**最小权限原则**（principle of least privilege）的一个例子，该原则对于优秀的软件工程至关重要。它指出，代码应该只被授予完成其目标任务所需的权限和访问，而不是更多。一个例子是局部变量的作用域，不需要它的时候，它就不应该可见。局部变量在函数被调用时创建，在函数执行时使用，然后在函数返回时消失。最小权限原则防止代码意外（或恶意）修改不应被它访问的变量值，从而使程序更健壮。它还使程序更容易阅读和维护。

一般来说，为变量提供所需的最窄作用域即可。只在特定函数中使用的变量应声明为该函数的局部变量，而不应声明为全局变量。

演示作用域

图 `5.6` 演示了全局变量、局部变量和静态局部变量的作用域问题。为方便讨论，我们把这个例子分成几个部分，并附有相应的输出。只有第一部分有图题——本书以后许多例子都会做类似的处理。第 `10` 行声明全局变量 `x` 并初始化为 `1`。在任何声明了同名变量的代码块（或函数）中，全局变量 `x`

都会被“隐藏”。

```
1  // fig05_06.cpp
2  // Scoping example.
3  #include <iostream>
4  using namespace std;
5
6  void useLocal(); // function prototype
7  void useStaticLocal(); // function prototype
8  void useGlobal(); // function prototype
9
10 int x{1}; // global variable
11
```

图 5.6 作用域示例

main 函数

在 main 中，第 13 行显示全局变量 x 的值。第 15 行将局部变量 x 初始化为 5。第 17 行输出该变量，证明全局变量 x 在 main 中被隐藏。接着，第 19 行~第 23 行在 main 中定义了一个新的代码块，其中另一个局部变量 x 被初始化为 7（第 20 行）。第 22 行输出该变量，证明它同时隐藏了 main 的外层块中的 x 以及全局 x。内层块退出时，值为 7 的 x 被自动销毁。接着，第 25 行在 main 的外层块中输出局部变量 x，表明它不再被隐藏。

```
12 int main() {
13    cout << "global x in main is " << x << '\n';
14
15    const int x{5}; // local variable to main
16
17    cout << "local x in main's outer scope is " << x << '\n';
18
19    { // block starts a new scope
20       const int x{7}; // hides both x in outer scope and global x
21
22       cout << "local x in main's inner scope is " << x << '\n';
23    }
24
25    cout << "local x in main's outer scope is " << x << '\n';
26
```

```
global x in main is 1
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5
```

为了演示其他作用域，程序定义了三个函数：useLocal、useStaticLocal 和 useGlobal。所有函数均无参，也不返回任何值。main 的其余部分（见下）在第 27 行~第 32 行调用每个函数两次。在分别执行了函数 useLocal、useStaticLocal 和 useGlobal 两次后，程序再

次打印 main 中的局部变量 x，以证明对这些函数的调用都没有修改 main 中的 x 值，因为这些函数引用的全都是其他作用域中的变量。

```
27      useLocal(); // useLocal has local x
28      useStaticLocal(); // useStaticLocal has static local x
29      useGlobal(); // useGlobal uses global x
30      useLocal(); // useLocal reinitializes its local x
31      useStaticLocal(); // static local x retains its prior value
32      useGlobal(); // global x also retains its prior value
33
34      cout << "\nlocal x in main is " << x << '\n';
35   }
36
```

```
local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal
```

```
global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

useLocal 函数

useLocal 函数将局部变量 x 初始化为 25（第 39 行）。当 main 调用 useLocal 时（第 27 行和第 30 行），该函数打印变量 x，将其递增并再次打印，然后将程序控制返回给它的调用者。程序每次调用该函数时，该函数都会重新创建局部变量 x 并将其重新初始化为 25。

```
37   // useLocal reinitializes local variable x during each call
38   void useLocal() {
39      int x{25}; // initialized each time useLocal is called
40
41      cout << "\nlocal x is " << x << " on entering useLocal\n";
42      ++x;
43      cout << "local x is " << x << " on exiting useLocal\n";
44   }
45
```

useStaticLocal 函数

useStaticLocal 函数声明静态变量 x，将其初始化为 50。局部静态变量即使离开作用域（即它所在的函数不再执行时），也会保留其值。当 main 中的第 28 行调用 useStaticLocal 函数时，函数会打印局部变量 x，在函数将程序控制返回给它的调用者之前，对 x 进行递增并再次打印。在对该函数的下一次调用中（第 31 行），静态局部变量 x 将包含值 51。由于附加了 static 关键字，所以第 50 行的变量初始化仅在首次调用 useStaticLocal 时（第 28 行）发生。

```
46  // useStaticLocal initializes static local variable x only the
47  // first time the function is called; value of x is saved
48  // between calls to this function
49  void useStaticLocal() {
50     static int x{50}; // initialized first time useStaticLocal is called
51
52     cout << "\nlocal static x is " << x
53        << " on entering useStaticLocal\n";
54     ++x;
55     cout << "local static x is " << x
56        << " on exiting useStaticLocal\n";
57  }
58
```

useGlobal 函数

useGlobal 函数没有声明任何变量。因此，当它引用变量 x 时，引用的是全局 x（第 10 行，在 main 前面）。当 main 调用 useGlobal（第 29 行）时，函数打印全局变量 x，将其乘以 10，并在函数将控制返回给调用者之前再次打印。下次 main 调用 useGlobal 时（第 32 行），全局变量容纳的是修改后的值，即 10。

```
59  // useGlobal modifies global variable x during each call
60  void useGlobal() {
61     cout << "\nglobal x is " << x << " on entering useGlobal\n";
62     x *= 10;
63     cout << "global x is " << x << " on exiting useGlobal\n";
64  }
```

5.11 内联函数

从软件工程的角度来看，将程序作为一组函数来实现是不错，但函数调用涉及执行时间和资源的开销。C++提供了**内联函数**（inline functions）来帮助减少函数调用开销。在函数定义的返回类型前附加一个 inline 关键字，即可让编译器在调用该函数的每个地方都生成函数主体代码的一份拷贝（如果可以的话），从而避免函数调用。这通常会使程序变大。

编译器有可能忽略 `inline` 限定符。可重用的内联函数通常放在头文件中，这样它们的定义就可以在使用了它们的每个源文件中内联。

核心准则

更改了内联函数的定义后，必须重新编译调用该函数的所有代码。虽然编译器可能自动内联你没有显式添加 `inline` 的代码，但“C++核心准则”指出，只应对“小的、对执行时间要求高”的函数进行内联[1]。

以下网址提供了关于内联函数微妙之处的一份非常全面的 FAQ：

https://isocpp.org/wiki/faq/inline-functions

图 5.7 使用内联函数 `cube`（第 9 行~第 11 行）来计算立方体的体积。.

```
1  // fig05_07.cpp
2  // inline function that calculates the volume of a cube.
3  #include <iostream>
4  using namespace std;
5
6  // Definition of inline function cube. Definition of function appears
7  // before function is called, so a function prototype is not required.
8  // First line of function definition also acts as the prototype.
9  inline double cube(double side) {
10    return side * side * side; // calculate cube
11 }
12
13 int main() {
14    double sideValue; // stores value entered by user
15    cout << "Enter the side length of your cube: ";
16    cin >> sideValue; // read value from user
17
18    // calculate cube of sideValue and display result
19    cout << "Volume of cube with side "
20       << sideValue << " is " << cube(sideValue) << '\n';
21 }
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

图 5.7 计算立方体体积的内联函数

5.12 引用和引用参数

性能

在许多编程语言中，向函数传递实参都采用两种方式：**传值**（pass-by-value）和**传引用**

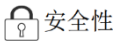
[1] C++ Core Guidelines，https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-inline。

（pass-by-reference）。如果采用传值方式，会创建参数值的一个拷贝，并将该拷贝传给被调用的函数。对该拷贝的更改不会影响调用者的原始变量值。这就防止了意外产生的副作用，这种副作用会极大妨碍我们开发正确和可靠的软件系统。到目前为止，本书每个实参都是传值的。但是，对于大的数据项来说，传值的缺点在于，对数据进行拷贝可能需要相当多的执行时间和内存空间。

引用参数

本节介绍引用参数——传引用的两种机制之一[①②]。如果传递的是对变量的引用，调用者相当于允许被调用的函数直接访问在调用者中的这个变量，而且允许对方修改该变量。

传引用有利于性能，因其可以避免传值时对大量数据进行拷贝的开销。但是，传引用不利于安全性，因为被调用的函数可能破坏调用者的数据。



在本节的例子之后，我们将展示如何在获得传引用的性能优势的同时，使调用者的数据免遭破坏。

引用参数（reference parameter）是调用函数时向其传递的实参的别名。为了指定一个函数参数是传引用的，需在函数原型中的参数类型后附加一个&符号。在函数（定义）头中列出参数类型时，也要使用同样的约定。以下面这个参数声明为例：

```
int& number
```

它从右到左读作“number 是对一个 int 的引用”。同样地，函数原型和函数头必须一致。

调用函数时则不需要添加&符号。直接使用变量名，即可传递对它的引用。在被调用函数的主体中，引用参数（如 number）引用的是调用者的原始变量，可直接由被调用的函数修改。

通过传值和传引用来传递实参

图 5.8 对比了传递实参时的传值和传引用方式。调用 squareByValue 和 squareByReference 这两个函数时，采用的“风格”是一致的，都是在函数调用中直接引用变量名。但是，编译器会检查函数原型和定义，以确定变量是以传值还是传引用的方式传给函数。

---

① 第 7 章会讨论指针，它是“传引用”的另一种机制。

② 第 11 章会讨论“引用”的另一种形式，称为“右值引用”（Rvalue reference）。

```cpp
// fig05_08.cpp
// Passing arguments by value and by reference.
#include <iostream>
using namespace std;

int squareByValue(int number); // prototype (for value pass)
void squareByReference(int& numberRef); // prototype (for reference pass)

int main() {
   int x{2}; // value to square using squareByValue
   int z{4}; // value to square using squareByReference

   // demonstrate squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue(x) << '\n';
   cout << "x = " << x << " after squareByValue\n\n";

   // demonstrate squareByReference
   cout << "z = " << z << " before squareByReference\n";
   squareByReference(z);
   cout << "z = " << z << " after squareByReference\n";
}

// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue(int number) {
   return number *= number; // caller's argument not modified
}

// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in function main
void squareByReference(int& numberRef) {
   numberRef *= numberRef; // caller's argument modified
}
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

图 5.8 传值和传引用

将引用作为函数中的别名

引用也可以作为一个函数中其他变量的别名使用（虽然它们通常像图 5.8 那样和函数一起使用）。例如，以下代码使用 count 的别名 cRef 使其递增 1：

```cpp
int count{1}; // declare integer variable count
int& cRef{count}; // create cRef as an alias for count
```

```
++cRef; // increment count (using its alias cRef)
```

引用变量必须在声明时初始化，而且不能作为别名重新赋给其他变量。从这个角度说，引用是常量。所有对别名（即引用）进行的操作实际都是在原始变量上进行的。别名只是原始变量的另一个名字。除非是对常量的引用（后面讨论），否则作为一个引用，它的初始化器必须是一个**左值**（lvalue）——即任何可以出现在赋值操作符左侧的东西，例如变量名。引用不能用常量或**右值**（rvalue）表达式来初始化。所谓右值表达式，就是只能出现在赋值操作右侧的东西，例如计算结果。

const 引用

为了指定一个引用参数不允许在被调用的函数中修改，可在参数声明中为类型名称附加 const 限定符。以下面这个 displayName 函数为例：

```
void displayName(std::string name) {
   std::cout << name << '\n';
}
```

调用这个函数时，函数会接收其 string 实参的一个拷贝。由于 string 对象可能很大，这种拷贝操作会影响应用程序的性能。因此，string 对象（以及一般的对象）应该以“传引用”的方式传给函数。

另外，displayName 函数不需要修改其实参。所以，遵循“最小权限原则”，我们应将参数声明为：

```
const std::string& name
```

性能

从右到左这样读：name 参数是对一个 string 的引用，该引用是常量。通过传递对 string 的引用，我们获得了性能上的优势。另外，displayName 将实参视为常量，所以 displayName 不能修改调用者中的值。

返回对局部变量的引用可能会很危险

如果返回对局部非 static 变量的引用，该引用会指向（引用）一个在函数返回时被丢弃的变量。试图访问这样的变量会产生未定义的行为，往往会造成程序崩溃或数据破坏[①]。

错误

对未定义变量的引用称为**空悬引用**（dangling references）。这属于一种逻辑错误，编译器通常会对此发出警告。软件工程团队通常有规定，代码在部署之前必须在没有任何警告的情况下进行编译。你可以利用编译器提供的选项，将警告视为错误，从而强制你的团队遵守这个规定。

软件工程

[①] C++ Core Guidelines，https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-dangle。

5.13 默认参数

程序从多个地方调用一个函数，并为一个特定的参数使用相同的参数值，这是很常见的一种情况。在这种情况下，可为该参数指定一个**默认参数**（default argument）——也就是要为该参数传递的一个默认值。如果程序在函数调用中省略了一个带有默认参数的参数，编译器会自动改写函数调用，插入该参数的默认值。

带默认参数的 boxVolume 函数

图 5.9 演示了如何使用默认参数来计算一个箱子的体积（长*宽*高）。`boxVolume` 的函数原型（第 7 行）在每个参数右侧添加`=1`，从而将全部三个参数的默认值设为 `1`。

```
1  // fig05_09.cpp
2  // Using default arguments.
3  #include <iostream>
4  using namespace std;
5
6  // function prototype that specifies default arguments
7  int boxVolume(int length = 1, int width = 1, int height = 1);
8
9  int main() {
10    // no arguments--use default values for all dimensions
11    cout << "The default box volume is: " << boxVolume();
12
13    // specify length; default width and height
14    cout << "\n\nThe volume of a box with length 10,\n"
15       << "width 1 and height 1 is: " << boxVolume(10);
16
17    // specify length and width; default height
18    cout << "\n\nThe volume of a box with length 10,\n"
19       << "width 5 and height 1 is: " << boxVolume(10, 5);
20
21    // specify all arguments
22    cout << "\n\nThe volume of a box with length 10,\n"
23       << "width 5 and height 2 is: " << boxVolume(10, 5, 2)
24       << '\n';
25 }
26
27 // function boxVolume calculates the volume of a box
28 int boxVolume(int length, int width, int height) {
29    return length * width * height;
30 }
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50
```

```
The volume of a box with length 10,
width 5 and height 2 is: 100
```

图 5.9 使用默认参数

对 boxVolume 的第一次调用（第 11 行）没有指定任何参数，因此使用全部三个默认值 1。第二次调用（第 15 行）只传递一个 length 实参，因此 width 和 height 参数使用默认值 1。第三次调用（第 19 行）只传递 length 和 width 实参，因此 height 参数使用默认值 1。最后一次调用（第 23 行）同时传递了长宽高，所以没有使用任何默认值。任何明确传给函数的实参都会从左到右依次赋给函数的参数（形参）。所以，当 boxVolume 收到一个实参时，函数将该它的值赋给 length 参数（即参数列表最左边的参数）。当 boxVolume 收到两个实参时，函数将这些值依次赋给它的 length 和 width 参数。最后，当 boxVolume 收到全部三个参数时，函数将这些值依次赋给 length、width 和 height 参数。

默认参数注意事项

默认参数必须是函数参数列表最右边（靠近尾部）的参数。调用具有两个或多个默认参数的函数时，如果一个被省略的实参不是最右边的那个，该实参右边的所有实参也必须省略。默认参数必须在函数名首次出现时指定，这通常是在函数原型中。如果省略了函数原型，那么因为函数定义也可以作为原型使用，所以默认参数必须在函数头中指定。默认值可为任意表达式，包括常量、全局变量或其他函数调用。默认参数也可与内联函数一起使用。使用默认参数可以简化函数调用的编写。但是，有的程序员认为，为所有参数显式提供参数值会更清晰。

5.14 一元作用域解析操作符

C++支持**一元作用域解析操作符**（unary scope resolution operator），即::。如果当前作用域有一个和全局变量同名的局部变量，就可以用该操作符访问变量的全局版本。注意，不能用它访问外层块中的同名局部变量。当然，如果全局变量的名字与作用域内的局部变量的名字不一样，那么可以直接访问全局变量，而不必使用一元作用域解析操作符。

图 5.10 展示了一元作用域解析操作符与同名的局部和全局变量（第 6 行和第 9 行）。为了强调 number 变量的局部和全局版本是不同的，程序将一个变量声明为 int 类型，另一个声明为 double 类型。一般来说，应避免在程序中将同名的变量用于不同的目的。虽然许多时候都可以这样做，但很容易引起错误。一般来说，我们不鼓励使用全局变量。如果使用了全局变量，一定要使用一元作用域解析操作符（::）来引用它（即使和局部变量的名称没有冲突）。这样可以清楚地表明你是在访问一个全局变量。这也使程序更容易修改，因为它减少了与非全局变量名称冲突的风险，并消除了局部变量将全局变量“隐藏”时可能发生的逻辑错误。

错误

```
1  // fig05_10.cpp
2  // Unary scope resolution operator.
3  #include <iostream>
4  using namespace std;
5
6  const int number{7}; // global variable named number
7
8  int main() {
9     const double number{10.5}; // local variable named number
10
11    // display values of local and global variables
12    cout << "Local double value of number = " << number
13       << "\nGlobal int value of number = " << ::number << '\n';
14 }
```

```
Local double value of number = 10.5
Global int value of number = 7
```

图 5.10 一元作用域解析操作符

5.15 函数重载

C++允许定义同名函数，只要它们有不同的签名即可，这称为**函数重载**（function overloading）。C++编译器检查调用时传递的实参的数量、类型和顺序来选择适当的函数进行调用。利用函数重载，我们可以创建几个执行相似任务、但对不同类型的数据进行处理的同名函数。例如，数学库中的许多函数都进行了重载，能处理不同的数值类型。5.3 节介绍的数学库函数就提供了 `float`、`double` 和 `long double` 的重载版本。对执行密切相关任务的函数进行重载，可以使程序更清晰。

重载的 square 函数

图 5.11 使用两个重载的 `square` 函数来分别计算 `int` 值的平方（第 7 行~第 10 行）和 `double` 值的平方（第 13 行~第 16 行）。第 19 行传递字面值 `7` 来调用 `int` 版本，C++将整数字面值视为 `int` 类型。类似地，第 21 行传递字面值 `7.5` 来调用 `double` 版本，C++将带小数点的字面值视为 `double` 类型。在每种情况下，编译器都会根据实参的类型选择适当的函数来调用。输出证明在每种情况下都调用了正确的函数。

```cpp
// fig05_11.cpp
// Overloaded square functions.
#include <iostream>
using namespace std;

// function square for int values
int square(int x) {
   cout << "square of integer " << x << " is ";
   return x * x;
}

// function square for double values
double square(double y) {
   cout << "square of double " << y << " is ";
   return y * y;
}

int main() {
   cout << square(7); // calls int version
   cout << '\n';
   cout << square(7.5); // calls double version
   cout << '\n';
}
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

编译器如何区分不同的重载版本？

重载函数是通过其签名来区分的。**签名**（signature）是函数名称及其参数类型（按顺序）的组合。**类型安全连接**（type-safe linkage）确保调用的是正确的函数，并且实参的类型与参数（形参）的类型一致。为了实现类型安全连接，编译器会在内部用参数的类型对每个函数标识符进行编码——这一过程称为**名称改编**（name mangling）[①]。具体编码方式因编译器而异。因此，所有最终需要链接到一起，从而为给定平台创建可执行文件的东西都必须使用该平台上的同一个编译器进行编译。图 5.12 是用 GNU C++编译的[②]。我们没有像平是那样显示程序的输出，而是显示 GNU C++在汇编语言中生成的、改编后的函数名称[③]。

[①] 译注：各种说法都有，包括但不限于：名称粉碎、名称修饰、名称重整。本书采用“名称改编”。

[②] 主体为空的 `main` 函数确保在编译这些代码时不会出现链接器错误。

[③] 用 `g++ -S fig05_12.cpp` 命令来生成汇编语言文件 fig05_12.s。

```
1  // fig05_12.cpp
2  // Name mangling to enable type-safe linkage.
3
4  // function square for int values
5  int square(int x) {
6     return x * x;
7  }
8
9  // function square for double values
10 double square(double y) {
11    return y * y;
12 }
13
14 // function that receives arguments of types
15 // int, float, char and int&
16 void nothing1(int a, float b, char c, int& d) { }
17
18 // function that receives arguments of types
19 // char, int, float& and double&
20 int nothing2(char a, int b, float& c, double& d) {
21    return 0;
22 }
23
24 int main() { }
```

```
_Z6squarei
_Z6squared
_Z8nothing1ifcRi
_Z8nothing2ciRfRd
main
```

图 5.12 通过名称改编来实现类型安全连接

对于 GNU C++来说，每个改编后的名称（除 main 外）都以下划线（_）开始，后面是字母 Z、一个数字和函数名。Z 后面的数字指定了函数名中有多少个字符。例如，square 函数的名称中有 6 个字符，所以它的改编名称的前缀是_Z6。函数名后面是其参数列表的编码：

- 对于接收一个 int 的 square 函数（第 5 行），i 代表 int，如输出的第一行所示。
- 对于接收一个 double 的 square 函数（第 10 行），d 代表 double，如输出的第二行所示。
- 对于 nothing1 函数（第 16 行），i 代表 int，f 代表 float，c 代表 char，而 Ri 代表 int&（即对 int 的引用），如输出的第三行所示。
- 对于 nothing2 函数（第 20 行），c 代表 char，i 代表 int，Rf 代表 float&，而 Rd 代表 double&。

编译器通过参数列表来区分两个 square 函数——一个用 i 代表 int，另一个用 d 代表 double。注意，函数返回类型不是签名的一部分，所以没有在改编中名称中反映。重载的函数可以有不同的返回类型，但还是必须有不同的参数列表。具体如何改编函数名称是由

编译器决定的。例如，Visual C++为第 5 行的 `square` 函数生成的改编名称是 `square@@YAHH@Z`。GNU C++编译器不对 `main` 这个名称进行任何处理，但有些编译器会。例如，Visual C++会改编成`_main`。

同一个参数列表，不同的返回类型，像这样创建重载函数会造成编译错误。编译器只根据参数列表来区分重载函数。重载函数不需要有相同数量的参数。

有默认参数的函数在调用时可能与另一个重载函数发生冲突；这也会造成编译错误。例如，假定程序中有一个函数显式地不获取任何实参，另一个同名函数则全部包含默认参数，那么一旦调用这个名字的函数，同时不传递任何实参，两者就会产生歧义，因为编译器无法判断应该调用函数的哪个版本。

重载解析详情

如果想详尽了解编译器如何解析重载的函数调用，请参考 C++标准的“Overload Resolution”一节：

https://timsong-cpp.github.io/cppwp/n4861/over.match

同时参考 cppreference.com 的“重载决议”一节：

https://zh.cppreference.com/w/cpp/language/overload_resolution

重载的操作符

第 11 章会讨论如何重载操作符，从而定义它们应该如何操作用户自定义数据类型的对象。（事实上，之前已经用到了许多重载操作符，其中包括流插入操作符<<和流提取操作符>>）。它们为所有 C++基本类型进行了重载。第 11 章将详细介绍如何重载<<和>>，使其能处理用户自定义类型的对象。

5.16 函数模板

重载的函数通常需要对不同的数据类型进行相似的操作。如果针对每种数据类型的程序逻辑和操作都一样，可考虑使用**函数模板**（function templates）来更精简、更方便地进行重载。在这种情况下，写单一的函数模板定义就可以了。根据调用此函数时提供的实参类型，C++会自动生成单独的**函数模板特化**（function template specializations）——也称为**模板实例**（template instantiations）——以适当地处理每种类型的调用。所以，定义单独一个函数模板，就相当于定义了全套重载函数。模板编程也称为**泛型编程**（generic programming）。本节将定义一个自定义函数模板。在以后的章节中，还会用到许多现成的 C++标准库模板。第 15 章将详细讨论定义自定义模板，其中包括 C++20“缩写函数模板”（abbreviated

function templates）和 C++20“概念”（Concepts）。

maximum 函数模板

图 5.13 定义了 maximum 函数模板，它判断三个值中哪一个最大。所有函数模板定义都以 template 关键字开始（第 3 行），后跟一个用尖括号（<和>）括起来的模板参数列表。**模板参数列表**（template parameter list）中的每个参数前面都有关键字 typename 或关键字 class（它们在这里是同义词）。**类型参数**（type parameters）是基本类型或用户自定义类型的占位符。这些占位符——本例是 T——被用来指定函数参数的类型（第 4 行），指定函数的返回类型（第 4 行），并在函数定义的主体中声明变量（第 5 行）。函数模板的定义和普通函数没有太多区别，只是使用类型参数作为实际数据类型的占位符。

```
1  // Fig. 5.13: maximum.h
2  // Function template maximum header.
3  template <typename T>  // or template <class T>
4  T maximum(T value1, T value2, T value3) {
5     T maximumValue{value1}; // assume value1 is maximum
6
7     // determine whether value2 is greater than maximumValue
8     if (value2 > maximumValue) {
9        maximumValue = value2;
10    }
11
12    // determine whether value3 is greater than maximumValue
13    if (value3 > maximumValue) {
14       maximumValue = value3;
15    }
16
17    return maximumValue;
18 }
```

图 5.13 maximum 函数模板定义

maximum 函数模板的类型参数 T（第 3 行）是一个占位符，代表该函数要处理的数据类型。在一个特定的模板参数列表中，所有类型参数名称都必须是唯一的。编译器在程序源代码中遇到对 maximum 的调用时，会在整个模板定义中将调用 maximum 时传递的实参类型替换为 T，从而创建一个完整的“函数模板特化”，以判断指定类型的三个值中的最大值。注意，这些值必须具有相同的类型，因为本例只使用了一个类型参数。然后，编译器对新创建的函数进行编译。从中可以看出，模板只是生成代码的一种手段。第 13 章会到需要多个类型参数的 C++标准库模板。

使用 maximum 函数模板

图 5.14 使用 maximum 函数模板分别判断三个 int 值、三个 double 值和三个 char 值的最

大值（第 15 行、第 24 行和第 33 行）。每个调用都传递了不同类型的实参，所以编译器在“幕后”为每个调用都创建了单独的函数定义：一个期望三个 int，一个期望三个 double，一个期望三个 char。

```
1  // fig05_14.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  #include "maximum.h" // include definition of function template maximum
5  using namespace std;
6
7  int main() {
8     // demonstrate maximum with int values
9     cout << "Input three integer values: ";
10    int int1, int2, int3;
11    cin >> int1 >> int2 >> int3;
12
13    // invoke int version of maximum
14    cout << "The maximum integer value is: "
15       << maximum(int1, int2, int3);
16
17    // demonstrate maximum with double values
18    cout << "\n\nInput three double values: ";
19    double double1, double2, double3;
20    cin >> double1 >> double2 >> double3;
21
22    // invoke double version of maximum
23    cout << "The maximum double value is: "
24       << maximum(double1, double2, double3);
25
26    // demonstrate maximum with char values
27    cout << "\n\nInput three characters: ";
28    char char1, char2, char3;
29    cin >> char1 >> char2 >> char3;
30
31    // invoke char version of maximum
32    cout << "The maximum character value is: "
33       << maximum(char1, char2, char3) << '\n';
34 }
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

图 5.14 maximum 函数模板测试程序

针对 int 类型的 maximum 函数模板特化

为 int 类型创建的函数模板特化将 T 的每一个实例都替换为 int，如下所示：

```
int maximum(int value1, int value2, int value3) {
   int maximumValue{value1}; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if (value2 > maximumValue) {
      maximumValue = value2;
   }

   // determine whether value3 is greater than maximumValue
   if (value3 > maximumValue) {
      maximumValue = value3;
   }

   return maximumValue;
}
```

5.17 递归

对于某些问题，让函数调用自己是非常有用。**递归函数**（recursive function）是一个能直接或间接（通过另一个函数）调用自己的函数。本节和下一节将介绍简单的递归例子。高级计算机科学课程中会更深入地讨论递归。

递归的概念

本节首先介绍递归的概念，然后讨论了使用递归函数的程序。递归解决方案有几个共同的要素。我们调用递归函数来解决一个问题，但只知道如何解决该问题最简单的情况，或称**基本情况**（base case）。如果针对的是基本情况来调用函数，那么函数能直接返回一个结果。但是，如果调用函数时针对的是更复杂的情况，那么它通常会将问题分成两个概念性的部分：一个是函数知道如何做的部分，另一个是它不知道如何做的部分。为了实现递归调用，后一部分必须类似于原始问题，但又是该问题的一个稍微简单或者更小的版本。这个新问题看起来像原始问题，所以函数调用它自身的一个拷贝来处理较小的问题。这称为**递归调用**（recursive call），也称为**递归步骤**（recursion step）。递归步骤通常包括关键字 return，因其结果会和该函数知道如何解决的那部分问题结合起来，以形成传递给原始调用者（可能是 main）的结果。

错误

如果遗漏基本情况，或者错误地编写了递归步骤，造成其无法收敛于基本情况，那么会导致无限递归错误，这通常会造成**栈溢出**（stack overflow）。这相当于一个循环（非递归）解决方案中的无限循环。如果一个没有设计成递归的函数意外地调用了自己，那么无论是直接调用还是通过另一个函数间接调用，也会发生无限递归。

执行递归步骤时，对函数的原始调用仍然“开着”（open），这意味着它尚未完成执行。递归步骤可能会导致更多这样的递归调用，因为函数不断地将调用该函数时针对的每个新的

子问题划分为两个概念性的部分（一个知道怎么做，另一个不知道怎么做）。递归最终要想终止，函数每次都会针对原始问题的一个稍微简单的版本来调用自己，这个越来越小的问题序列最终必须收敛于“基本情况”。在这个时候，函数识别出了基本情况，并向函数的上一个拷贝返回结果。然后，通过一连串的返回，直到最初的调用将最终结果返回给调用者[1]。与我们到目前为止所使用的问题解决方式相比，这一过程听起来相当奇特。为了理解递归的概念，让我们写一个递归程序来执行一个流行的数学计算。

阶乘

非负整数 *n* 的阶乘记为 *n*!（读作“*n* 的阶乘”），其公式是：

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

其中，$1! = 1$，且$0! = 1$。例如，$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$。

以循环方式求阶乘

对于一个大于或等于 `0` 的整数，其阶乘可用 `for` 语句以循环方式（非递归）进行计算，如下所示：

```
int factorial{1};
for (int counter{number}; counter >= 1; --counter) {
   factorial *= counter;
}
```

以递归方式求阶乘

递归阶乘的定义是通过观察到以下代数关系来得出的：

```
n! = n · (n – 1)!
```

例如，`5!`显然等于 `5 * 4!`，如下所示：

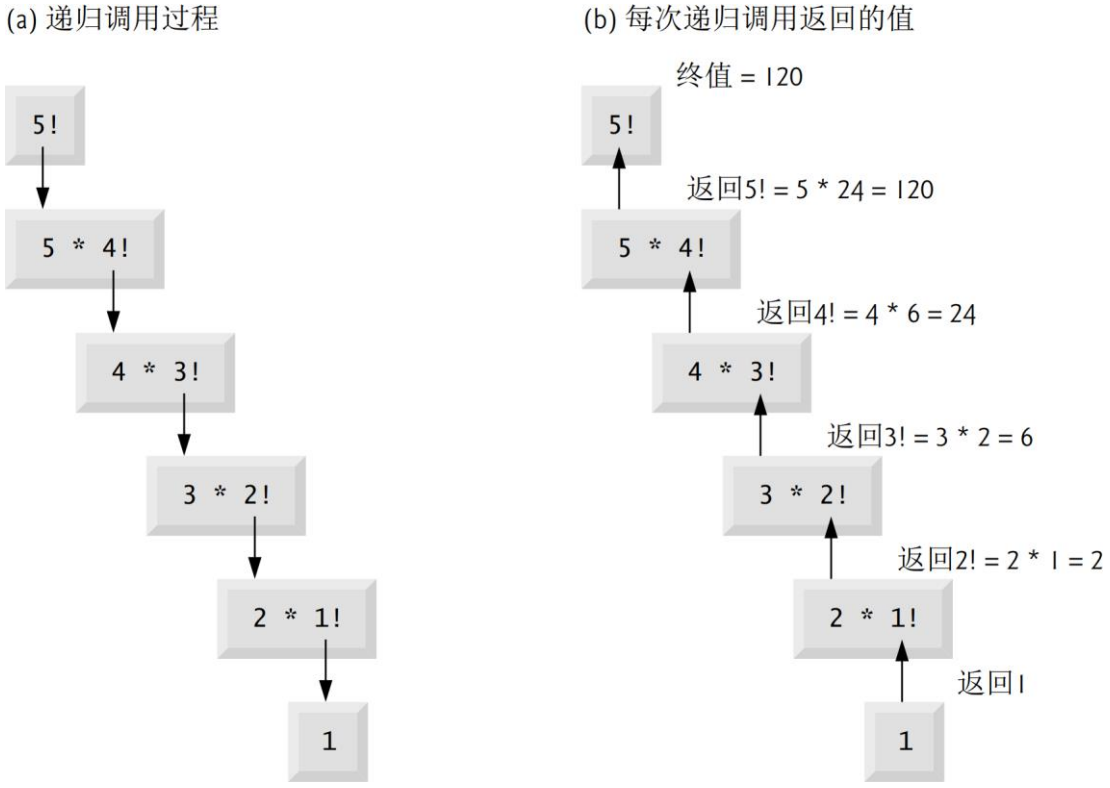
```
5! = 5 · 4 · 3 · 2 · 1
5! = 5 · (4 · 3 · 2 · 1)
5! = 5 · (4!)
```

---

[1] C++标准指出，`main` 不应该在程序内调用，也不能以递归方式调用。它唯一的目的就是成为程序执行的入口点。https://timsong-cpp.github.io/cppwp/n4861/basic.start.main 和 https://timsong-cpp.github.io/cppwp/n4861/expr.call。

求值 5!

下图展示了 5!的求值过程。其中，(a)部分展示了如何连续进行递归调用，直到 1!被求值为 1（基本情况），从而终止递归。(b)部分展示了每次递归调用返回给调用者的值，直到计算出终值并将其返回。



递归调用 factorial 函数来计算阶乘

图 5.15 利用递归算法来计算并打印 0~10 的整数的阶乘。递归函数 factorial（第 18 行~第 25 行）首先判断终止条件 number <= 1（即基本情况；第 19 行）是否为真。如果 number 小于或等于 1，就符合基本情况，所以 factorial 函数返回 1（第 20 行），此时没必要进一步递归，函数终止。如果 number 大于 1，第 23 行将问题表达为 number 乘以求“number-1”阶乘的一个递归 factorial 调用，后者比原始计算 factorial(number)稍微简单一些。

```
1  // fig05_15.cpp
2  // Recursive function factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  long factorial(int number); // function prototype
8
9  int main() {
10    // calculate the factorials of 0 through 10
11    for (int counter{0}; counter <= 10; ++counter) {
12       cout << setw(2) << counter << "! = " << factorial(counter)
13          << '\n';
14    }
15 }
16
17 // recursive definition of function factorial
18 long factorial(int number) {
19    if (number <= 1) { // test for base case
20       return 1; // base cases: 0! = 1 and 1! = 1
21    }
22    else { // recursion step
23       return number * factorial(number - 1);
24    }
25 }
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

图 5.15 递归版本的 factorial 函数

阶乘值快速增长

factorial 函数接收一个 int 类型的参数并返回一个 long 类型的结果。通常，long 至少用 4 个字节（32 位）来存储；这样变量可以容纳–2 147 483 648 到 2 147483 647 范围内的值。但遗憾的是，factorial 函数生成大数的速度非常快，计算不了多少阶乘值，就会达到 long 的最大值。要容纳更大的整数值，可以使用 long long 类型（3.11 节）或一个能表示任意大小整数的类（例如 3.12 节介绍的开源 BigNumber 类）。

5.18 递归示例：斐波那契数列

斐波那契数列从 0 和 1 开始，后续每个斐波那契数都是前两个之和，如下所示：

0, 1, 1, 2, 3, 5, 8, 13, 21, …

连续斐波那契数的比率收敛于一个常数 1.618…，它由<numbers>头文件中的常量::phi 表示。这个数字经常出现在自然界，被称为黄金比例或黄金分割。人类倾向于认为黄金分割线具有美感。建筑师经常设计出长度和宽度与黄金分割比例一致的窗户、房间和建筑物。明信片的设计也经常采用黄金分割的长/宽比例。在网上搜索“自然界中的斐波那契”会发现许多有趣的例子，包括花瓣、贝壳、螺旋星系、飓风等等。

斐波那契的递归定义

斐波那契的递归定义如下所示：

$$fibonacci(0) = 0$$

$$fibonacci(1) = 1$$

$$fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)$$

图 5.16 的程序使用 fibonacci 函数递归地计算第 *n* 个斐波那契数。虽然比阶乘慢得多，但斐波那契数也会迅速变大。图 5.16 展示了程序的执行情况，显示了几个数的斐波那契值。

```
1  // fig05_16.cpp
2  // Recursive function fibonacci.
3  #include <iostream>
4  using namespace std;
5
6  long fibonacci(long number); // function prototype
7
8  int main() {
9     // calculate the fibonacci values of 0 through 10
10    for (int counter{0}; counter <= 10; ++counter)
11       cout << "fibonacci(" << counter << ") = "
12          << fibonacci(counter) << '\n';
13
14    // display higher fibonacci values
15    cout << "\nfibonacci(20) = " << fibonacci(20) << '\n';
16    cout << "fibonacci(30) = " << fibonacci(30) << '\n';
17    cout << "fibonacci(35) = " << fibonacci(35) << '\n';
18 }
19
20 // recursive function fibonacci
21 long fibonacci(long number) {
22    if ((0 == number) || (1 == number)) { // base cases
23       return number;
24    }
25    else { // recursion step
26       return fibonacci(number - 1) + fibonacci(number - 2);
27    }
28 }
```

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8
fibonacci(7) = 13
fibonacci(8) = 21
fibonacci(9) = 34
fibonacci(10) = 55

fibonacci(20) = 6765
fibonacci(30) = 832040
fibonacci(35) = 9227465
```

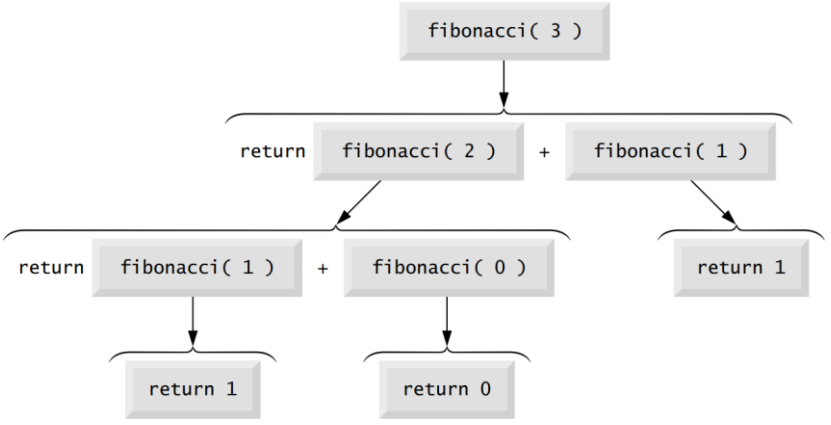
图 5.16 递归函数 `fibonacci`

该应用程序以一个循环开始，计算并显示整数 0~10 的斐波那契值，随后通过三个调用来计算整数 20、30 和 35 的斐波那契值（第 15 行~第 17 行）。`main` 中对 `fibonacci` 的调用（第 12 行和第 15 行~第 17 行）不是递归调用，但第 26 行的 `fibonacci` 调用是。每次程序调用 `fibonacci` 时（第 21 行~第 28 行），该函数立即测试基本情况，判断 `number` 是否等于 `0` 或 `1`（第 22 行）。如果为真，第 23 行返回 `number`。有趣的是，如果 `number` 大于 `1`，递归

步骤（第26行）会产生两个递归调用，每个调用解决的问题都比最初调用 fibonacci 时要解决的问题小一点。

求值 fibonacci(3)

下图展示了 fibonacci 函数如何求值 fibonacci(3)。关于 C++编译器对操作符的操作数进行求值的顺序，这张图提出了一个有趣的问题。这是一个独立的问题，它有别于操作符应用于其操作数的顺序，后者是由操作符优先级和分组规则决定的。下图显示，求值 fibonacci(3)导致两个递归调用，即 fibonacci(2)和 fibonacci(1)。这些调用是按什么顺序进行的？



操作数的求值顺序

大多数程序员简单地认为操作数从左到右求值，在一些编程语言中确实如此。但是，C++没有规定许多操作符（包括+）的操作数求值顺序。因此，对这些调用的执行顺序不能做任何预设。事实上，这些调用可能先执行 fibonacci(2)，再执行 fibonacci(1)，也可能相反。在这个程序和其他大多数程序中，最终结果都是一样的。但在一些程序中，操作数的求值可能会产生副作用（输入/输出操作或数据值的改变），从而影响表达式的最终结果。

指定了求值顺序的操作符

在 C++17 之前，C++只为&&、||、逗号（,）和?:等操作符规定了操作数的求值顺序。前三个是二元操作符，其两个操作数保证从左到右求值。最后一个操作符是 C++唯一的三元操作符。它最左边的操作数总是先被求值。如果求值结果为真，中间的操作数将被求值，最后的操作数会被忽略。如果最左边的操作数被求值为假，第三个操作数将被求值，而中间的操作数会被忽略。

从 C++17 开始，还为其他多种操作符规定了操作数的求值顺序。对于[]（第 6 章）、->（第 7 章）、（函数调用的）圆括号、<<、>>、.*和->*操作符，编译器会从左到右求值操作数。对于一个函数调用的圆括号来说，这意味着编译器会在实参之前先求值函数名。编译器对赋值操作符的操作数从右到左进行求值。

如程序依赖于操作符的操作数的求值顺序，那么可能导致逻辑错误。为了确保副作用以正确的顺序应用，请将复杂表达式分解成单独的语句。以前讲过，&&和||操作符使用了短路求值技术。如果一个具有副作用的表达式总是应该求值，那么将该表达式放在&&或||操作符右侧就是一种逻辑错误。

指数级复杂度

对于本节用于生成斐波那契数的递归程序，有一点需要注意。`fibonacci` 函数中的每一级递归都会倍增函数调用数量；换言之，为了计算第 n 个斐波那契数，所需的递归调用在在2^n数量级上。这很快就会失去控制。仅计算第 20 个斐波那契数就需要2^{20}次或大约一百万次调用，计算第 30 个斐波那契数则需要2^{30}次或大约十亿次调用，以此类推。计算机科学家将此称为**指数级的复杂度**（exponential complexity）。当 n 变得更大时，这种性质的问题甚至会让世界上最强大的计算机都感到羞愧。一般来说，复杂度问题，特别是指数级复杂度，会在通常称为“算法”的高级计算机科学课程中详细讨论。在实际工作中要避免写斐波那契式的递归程序，这种程序会导致指数级的调用“爆炸”。

5.19 对比递归和循环

前两节研究了两个递归函数，它们也可以用简单的循环结构来实现。本节对这两种方法进行了比较，并讨论了为什么在特定情况下应该青睐其中一种而不是另一种。

- 循环和递归都基于一个控制语句。循环使用一个循环语句，递归使用一个选择语句。
- 循环和递归都涉及迭代。循环显式使用某个循环语句进行迭代。递归则通过调用自身的函数来实现迭代。
- 循环和递归都有一个终止测试。当循环持续条件失败时，循环就会终止。递归则在识别到一个基本情况时终止。
- 由计数器控制的循环和递归各自逐渐接近终止。循环会修改一个计数器，直到该计数器的值使循环继续条件失败。递归不断产生原始问题的更简单版本，直至到达基本情况。
- 无论循环还是递归都可能无限地发生。如果循环继续测试永远不会变成假，就会发生无限循环。如果递归步骤在每次递归调用中没有以收敛于基本情况的方式减小问题，则会发生无限递归。

⊗错误

阶乘的循环版本

让我们用计算阶乘的循环版本（图 5.17）来体会循环和递归之间的区别。第 22 行~第 24 行使用的是一个循环语句，而不是递归解决方案中的选择语句（图 5.15 的第 19 行~第 24 行）。在图 5.15 的递归版本中，第 19 行测试基本情况以终止递归。在图 5.17 的循环版本中，第 22 行测试循环继续条件——如果测试失败，则循环终止。此外，循环解决方案不是每次迭代都生成原始问题的一个更简单的版本。相反，它使用了一个计数器，该计数器被一直修改，直到循环继续条件变为假。

```
1  // fig05_17.cpp
2  // Iterative function factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  long factorial(int number); // function prototype
8
9  int main() {
10    // calculate the factorials of 0 through 10
11    for (int counter{0}; counter <= 10; ++counter) {
12       cout << setw(2) << counter << "! = " << factorial(counter)
13          << '\n';
14    }
15 }
16
17 // iterative function factorial
18 long factorial(int number) {
19    long result{1};
20
21    // iterative factorial calculation
22    for (int i{number}; i >= 1; --i) {
23       result *= i;
24    }
25
26    return result;
27 }
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

图 5.17 循环版本的 `factorial` 函数

递归的缺点

性能

递归重复生成函数调用，会造成不小的开销，会在处理器时间和内存空间上付出昂贵代价。每次递归调用都会创建函数变量的另一份拷贝，这可能会消耗大量的内存。而循环通常一个函数内发生，所以节省了重复函数调用和额外内存分配的开销。那么，为什么有时还是要选择递归呢？

递归和循环的选择时机

性能

任何能用递归方法解决的问题也能用循环（非递归）方法解决。如果递归方法能更自然地反映出问题，并导致程序更容易理解和调试，通常可以考虑递归方法。选择递归方法的另一个原因是，当递归方法明显时，循环方法可能并不明显。如有可能，避免在追求性能的场合使用递归。递归调用需要时间并消耗额外的内存。

5.20 Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz

你肯定已经注意到了，本章“学习目标”的最后一项、5.1 节的最后一句话以及本节的标题都看起来像是胡言乱语。这不是错误！本节将继续我们的“对象自然”方法，将创建现有类的一个对象来加密和解密信息，该类实现了维吉尼亚密码（Vigenère secret key cipher）[①]。

在之前的“对象自然”小节中，你已经创建了 C++标准库 `string` 类的对象以及来自开源库的类的对象。有的时候，你会使用自己的组织或团队成员为内部使用或在特定项目中使用而创建的类。我们已经为本例编写了自己的 Cipher 类（在头文件`"cipher.h"`中）并提供给你。在第 9 章中，你将创建你自己的自定义类。

密码学

安全性

密码学已经有几千年的历史[②③]，在互联网时代更是至关重要。每一天，密码学都会在幕后使用，以确保你基于互联网的通信是私有和安全的。例如，大多数网站现在使用 HTTPS

[①] https://zh.wikipedia.org/zh/维吉尼亚密码

[②] https://zh.wikipedia.org/wiki/密码学

[③] Binance Academy，“History of Cryptography”，Binance Academy，https://www.binance.vision/security/history-of-cryptography。

协议来加密和解密你的 Web 交互。

凯撒密码

凯撒大帝用一种简单的替换加密技术来加密军事通信[①]。后被称为**凯撒密码**的这个技术是将通信中的每个字母替换为字母表中后数第三位的字母。因此，A 替换成 D，B 替换成 E，C 替换成 F，……，X 替换成 A，Y 替换成 B，Z 替换成 C。所以以下明文：

```
Caesar Cipher
```

会被加密成：

```
Fdhvdu Flskhu
```

明文加密后就成了**密文**（ciphertext）。

以下网站提供了体验凯撒密码和其他许多密码技术的一种有趣的方式：

https://cryptii.com/pipes/caesar-cipher

它其实是开源 cryptii 项目的一个网上实现：

https://github.com/cryptii/cryptii

维吉尼亚密码

像凯撒密码这样简单的替换密码很容易解密。例如，字母“e”是英语中使用频率最高的字母。所以，可以研究用凯撒密码加密的密文，其中出现频率最高的字符极可能就是“e”。

本例使用维吉尼亚密码，它是一种密钥替换密码。这种密码是用 26 个凯撒密码实现的，字母表中的每个字母都对应一个。维吉尼亚密码使用来自明文和秘钥中的字母来查找各个凯撒密码中的替换字符。可以访问以下网址来更具体地了解它是如何实现的：

https://zh.wikipedia.org/zh/维吉尼亚密码

在这种密码系统中，秘钥必须由字母组成。和你上网时使用的各种密码一样，秘钥不应该被人轻松猜到。我们使用的是随机选择的 11 个字符：

```
XMWUJBVYHXZ
```

[①] https://zh.wikipedia.org/wiki/凯撒密码

在秘钥中使用的字符数没有限制。然而，对密文进行解密的人必须知道这个密钥[1]。通常，你会事先向对方提供这个密钥，例如以面对面的方式。

使用现成的 Cipher 类

图 5.18 的例子将使用我们已经创建好的 Cipher 类，它实现了维吉尼亚密码。本章 ch05 示例文件夹中的头文件"cipher.h"（第 3 行）定义了这个类。你不需要阅读和理解这个类的代码，就能直接用加密和解密。只需创建一个 Cipher 类的对象，然后调用它的 encrypt 和 decrypt 成员函数来分别加密和解密文本。

```
1  // fig15_18.cpp
2  // Encrypting and decrypting text with a Vigenère cipher.
3  #include "cipher.h"
4  #include <iostream>
5  #include <string>
6  using namespace std;
7
8  int main() {
9     string plainText;
10    cout << "Enter the text to encrypt:\n";
11    getline(cin, plainText);
12
13    string secretKey;
14    cout << "\nEnter the secret key:\n";
15    getline(cin, secretKey);
16
17    Cipher cipher;
18
19    // encrypt plainText using secretKey
20    string cipherText{cipher.encrypt(plainText, secretKey)};
21    cout << "\nEncrypted:\n   " << cipherText << '\n';
22
23    // decrypt cipherText
24    cout << "\nDecrypted:\n   "
25       << cipher.decrypt(cipherText, secretKey) << '\n';
26
27    // decrypt ciphertext entered by the user
28    cout << "\nEnter the ciphertext to decipher:\n";
29    getline(cin, cipherText);
30    cout << "\nDecrypted:\n   "
31       << cipher.decrypt(cipherText, secretKey) << '\n';
32 }
```

[1] 有许多网站提供了维吉尼亚密码解码器，试图在没有原始密钥的情况下解密密文。我们尝试了几个，但没有一个能恢复我们的原始文本。

```
Enter the text to encrypt:
Welcome to Modern C++ application development with C++20!

Enter the secret key:
XMWUJBVYHXZ

Encrypted:
   Tqhwxnz rv Jnaqnh L++ bknsfbxfeiw eztlinmyahc xdro Z++20!

Decrypted:
   Welcome to Modern C++ application development with C++20!

Enter the ciphertext to decipher:
Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz

Decrypted:
   Objects Natural Case Study: Encryption and Decryption
```

图 5.18 用维吉尼亚密码加密和解密寺、

Cipher 类的成员函数

这个类提供了两个关键的成员函数：

- `encrypt` 接收代表明文和密钥的字符串，使用维吉尼亚密码对文本进行加密，然后返回一个包含密文的 `string`。
- `decrypt` 接收代表密文和密钥的字符串，反转维吉尼亚密码以解密文本，然后返回一个包含明文的 `string`。

程序首先要求输入待加密的文本和一个密钥。第 17 行创建 `Cipher` 对象。第 20 行~第 21 行对输入的文本进行加密并显示密文。然后，第 24 行~第 25 行解密文本，显示你之前输入的明文字符串。

虽然本章“学习目标”的最后一项和本节的标题看起来像是胡言乱语，但它们都是我们用 `Cipher` 类和密钥 `XMWUJBVYHXZ` 创建的密文。

第 28 行~第 29 行提示并输入现有的密文，然后第 30 行~第 31 行解密密文并显示我们加密的原始明文。

5.21 小结

本章介绍了函数的特性，包括函数原型、函数签名、函数头和函数主体。我们概述了数学库函数以及 C++20、C++17 和 C++11 新增的数学函数和数学常数。

你了解了实参强制类型转换——使实参的类型顺应形参所指定的类型。我们概述了 C++标

准库的头文件，演示了如何生成 C++11 非确定性随机数，通过“有作用域的 enum”定义常量集合，并介绍了 C++20 的 using enum 声明。

本节讨论了了变量的作用域，并讨论了向函数传递实参的两种方法：传值和传引用。我们展示了如何实现内联函数和接收默认参数的函数。你知道了重载函数具有相同的名称但有不同的签名。这种函数可以用来执行相同或类似的任务，但使用不同的类型或不同的参数数量。我们演示了如何使用函数模板来方便地生成一套重载函数。然后，我们讨论了递归，即函数调用自己来解决一个问题。最后的“对象自然”案例研究探讨了用于加密和解密文本的密钥替换密码。

第 6 章将学习如何用数组和面向对象的向量来维护数据的列表和表格。你会看到掷骰子应用程序的一个更优雅的、基于数组的实现。

附录 A 操作符优先级和分组

以下操作符按优先级从上到下降序排列。

操作符	用途	分组
::	作用域解析	从左到右
()	分组圆括号	
()	函数调用	从左到右
[]	数组下标	
.	通过对象选择成员	
->	通过指针选择成员	
++	一元后递增	
--	一元后递减	
typeid	运行时类型信息	
dynamic_cast<*类型*>	运行时类型检查强制类型转换	
static_cast<*类型*>	编译时类型检查强制类型转换	
reinterpret_cast<*类型*>	针对非标准转换的强制类型转换	
const_cast<*类型*>	去掉 const 性质的强制类型转换	
++	一元前递增	从右到左
--	一元前递减	
+	一元加法	
-	一元减法	
!	一元逻辑取反	
~	一元按位求补	
sizeof	判断大小(单位：字节)	
&	取址	

`*`
`new`
`new[]`
`delete`
`delete[]`
`co_await`
`(`*类型*`)`
	解引用
	动态内存分配
	动态数组分配
	动态内存取消分配
	动态数组取消分配
	等待协程完成
	C 风格的一元强制类型转换	

`.*` `->*`	通过对象对指向成员的指针进行解引用 通过指针对指向成员的指针进行解引用	从左到右

`*`
`/`
`%`
	乘法
	除法
	求余
		从左到右

`+` `-`	加法 减法	从左到右
`<<` `>>`	按位左移位 按位右移位	从左到右
`<=>`	三路比较	从左到右

`<`
`<=`
`>`
`>=`
	关系小于
	关系小于或等于
	关系大于
	关系大于或等于
		从左到右

`==` `!=`	关系等于 关系不等于	从左到右
`&`	按位 AND	从左到右

^	按位 XOR(异或)	从左到右
|	按位 OR	从左到右
&&	逻辑 AND	从左到右
||	逻辑 OR	从左到右
?:	三元条件	从右到左
=	赋值	
+=	加法赋值	
-=	减法赋值	
*=	乘法赋值	
/=	除法赋值	
%=	求余赋值	
&=	按位 AND 赋值	
^=	按位 XOR 赋值	
|=	按位 OR 赋值	
<<=	按位左移位赋值	
>>=	按位右移位赋值	
co_yield	暂停协程并返回一个值	
throw	抛出异常	
,	逗号	从左到右

附录 B 字符集

ASCII字符集

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

表格左侧的数字是字符十进制编码（0~127）左边的数字，表格上方的数字是编码右边的数字。例如，'F'的字符编码是 70，'&'的字符编码则是 38。