HZ BOOKS
华章教育
Pearson

经 典 原 版 书 库

C++语言导学

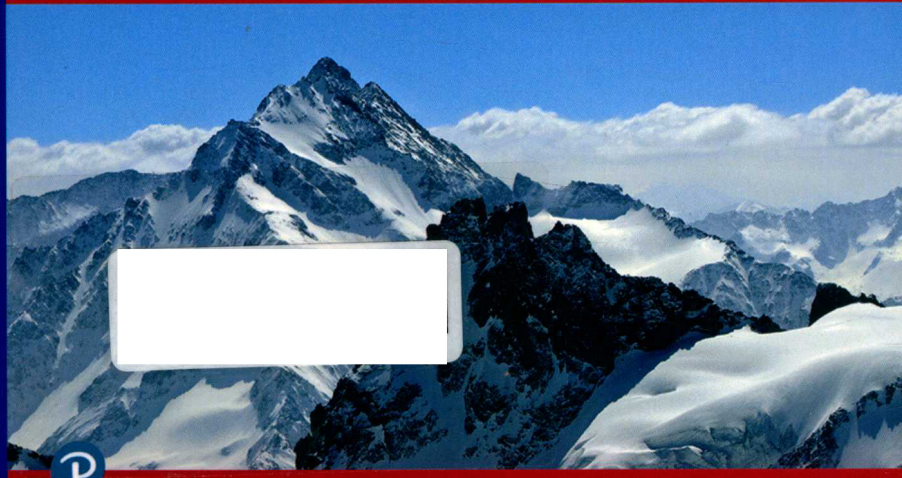
[美] 本贾尼·斯特劳斯特鲁普 著
(Bjarne Stroustrup)

（英文版·第2版）

A Tour of C++ (Second Edition)

A Tour of C++
Second Edition

Bjarne Stroustrup



机械工业出版社
China Machine Press

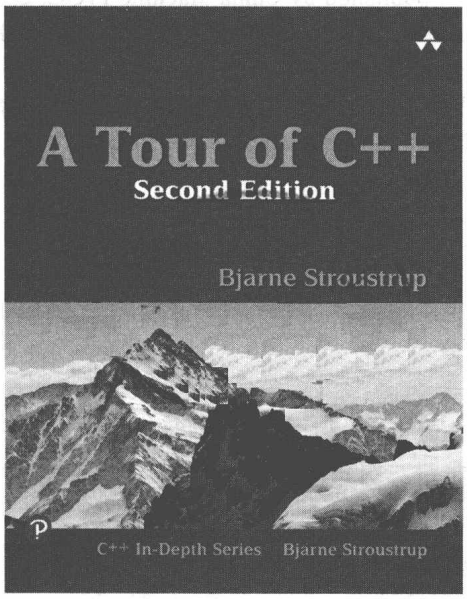
C++ In-Depth Series Bjarne Stroustrup

经 典 原 版 书 库

C++语言导学

（英文版·第2版）

A Tour of C++ (Second Edition)



[美]

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

C++ 语言导学（英文版·第 2 版）/（美）本贾尼·斯特劳斯特鲁普（Bjarne Stroustrup）著．—北京：机械工业出版社，2019.1
（经典原版书库）
书名原文：A Tour of C++, Second Edition

ISBN 978-7-111-61564-4

Ⅰ. C… Ⅱ. 本… Ⅲ. C++ 语言－程序设计－英文 Ⅳ. TP312.8

中国版本图书馆 CIP 数据核字（2019）第 278157 号

本书版权登记号：图字 01-2018-4809

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：张梦玲　　责任校对：殷虹
印　刷：北京瑞德印刷有限公司　　版　次：2019 年 1 月第 1 版第 1 次印刷
开　本：186mm×240mm　1/16　　印　张：15
书　号：ISBN 978-7-111-61564-4　　定　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

出版者的话

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson、McGraw-Hill、Elsevier、MIT、John Wiley & Sons、Cengage 等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出 Andrew S. Tanenbaum、Bjarne Stroustrup、Brian W. Kernighan、Dennis Ritchie、Jim Gray、Afred V. Aho、John E. Hopcroft、Jeffrey D. Ullman、Abraham Silberschatz、William Stallings、Donald E. Knuth、John L. Hennessy、Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：（010）88379604

联系地址：北京市西城区百万庄南街 1 号

邮政编码：100037

邮政编码：100037

华章教育

华章科技图书出版中心

前　言

教而至简，不亦乐乎。

——西塞罗

现在的 C++ 感觉就像是一种新的语言。与 C++98 相比，现在的 C++ 令我能更清晰、更简单、更直接地表达思想。而且，编译器可以更好地检查程序中的错误，程序的运行速度也提高了。

本书给出 C++ 语言的一个概述，这里所说的 C++ 是由当前的 ISO C++ 标准 C++17 所定义的，由主要的 C++ 提供商实现。此外，本书还会介绍概念和模块，它们是由 ISO 技术规范定义的，目前已在使用，但在 C++20 之前尚无计划包含进标准中。

就像其他任何一种现代编程语言一样，C++ 规模庞大且提供非常丰富的库，这是高效编程所需的。这本薄书的目的是让一个有经验的程序员快速了解现代 C++ 语言，因此它覆盖了大多数主要的语言特性和主要的标准库组件。读者花费几个小时就能读完这本书，但显然要想写出漂亮的 C++ 程序绝非一日之功。好在本书的目的并非让读者熟练掌握一切，而只是给出一个概览，给出一些关键的例子，帮助读者开始自己的 C++ 之旅。

假设读者已经拥有了一些编程经验。如果没有，建议你先找一本入门教材学习，比如《Programming: Principles and Practice Using C++》(Second Edition) [Stroustrup, 2009]，然后再来学习本书。即便你曾经编写过程序，你使用的语言或者编写的应用也可能在风格或形式上与本书所介绍的 C++ 相距甚远。

我们用城市观光的例子来说明本书的作用，比方说游览哥本哈根或者纽约。在短短几个小时之内，你可能会匆匆游览几个主要的景点，听一些有趣的传说或故事，然后收到接下来做什么的建议。仅靠这样一段旅程，你无法真正了解这座城市，也无法完全理解听到和看到的东西，更无法熟悉这座城市正式的非正式的生存法则。毕竟想要真正了解一座城市，你必须生活在其中，而且往往需要多年。不过如果幸运的话，此时你已经对城市的概貌有了一些了解，知道了它的某些特殊之处，并且对有些方面产生了兴趣。在这段旅程之后，你就可以开始真正的探索了。

这段旅程会为你介绍 C++ 语言的主要特性，它是按其所支持的程序设计风格来呈现的，例如面向对象编程和泛型编程。本书不准备提供一个详细的、手册式的、逐条特性介绍的 C++ 语言呈现。遵循优秀教科书的传统，我努力在使用每个语言特性之前对其进行解释，但实际情况并不总是允许这样，而且并不是每个人都会严格按顺序阅读本书。因此，我鼓励读者使用交叉引用和索引。

类似地，这段旅程是以示例的方式介绍标准库，而非逐一列举标准库特性。本书没有介绍 ISO 标准之外的库，需要的话，读者可以查阅相关资料，例如文献 [Stroustrup, 2013] 和 [Stroustrup, 2014]，但网络上还有大量的（质量也参差不齐的）其他资料，如文献 [Cppreference]。例如，当我提到一个标准库函数或类时，很容易就能找到它的定义，并且通

过查找其文档，找到很多相关的资料。。

本书力求把 C++ 作为一个整体呈现在读者面前，而非像千层糕一样逐层地介绍。因此，本书不细分某个语言特性是属于 C、C++98 的一部分还是新的 C++11、C++14 或 C++17。这种介绍可在第 16 章中找到。我聚焦基础并力求简洁，但我也未能完全抵抗过度阐述新特性的诱惑。这看起来也满足了很多已经了解旧版本 C++ 的读者的好奇心。

一本程序设计语言参考手册或标准会简单陈述可以做什么，但程序员通常对学习如何用好语言更感兴趣。要达到这个目的，一方面要靠主题的选择，另一方面要靠文字的组织，特别是建议部分。关于优秀的现代 C++ 语言是怎样构成的更多建议可在《C++ Core Guidelines》[Stroustrup, 2015] 一书中找到，对于希望继续深入探索本书介绍的思想的读者，它是一本很好的书。你可能注意到了，《C++ Core Guidelines》和本书在建议的呈现上甚至建议的编号方式上都惊人相似。其中一个原因是本书第一版是最初的《C++ Core Guidelines》的主要内容资源。

致谢

本书的一些内容源自 TC++PL4 [Stroustrup, 2013]，因此要感谢帮助我完成 TC++PL4 的所有同仁。

感谢帮助我完成并校对本书第一版的所有同仁。

感谢摩根·斯坦利给予我时间进行本书的写作。感谢哥伦比亚大学 2018 春季课程“使用 C++ 设计程序”的所有学生找出了本书最初草稿中的很多拼写问题和错误并给出了很多建设性的意见。

感谢保罗·安德森、查克·埃利森、彼得·哥特史林、威廉·蒙斯、查理·威尔逊和谢尔盖·祖布科夫审阅了本书并给出了很多改进建议。

Bjarne Stroustrup

曼哈顿，纽约

目　录

Contents

1

The Basics

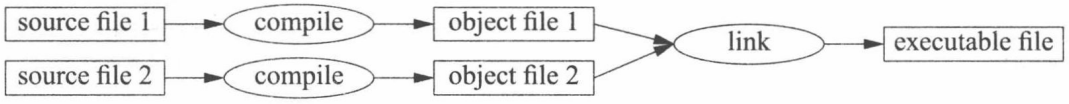
The first thing we do, let's
kill all the language lawyers.
– Henry VI, Part II

1.1 Introduction

This chapter informally presents the notation of C++, C++'s model of memory and computation, and the basic mechanisms for organizing code into a program. These are the language facilities supporting the styles most often seen in C and sometimes called *procedural programming*.

1.2 Programs

C++ is a compiled language. For a program to run, its source text has to be processed by a compiler, producing object files, which are combined by a linker yielding an executable program. A C++ program typically consists of many source code files (usually simply called *source files*).



An executable program is created for a specific hardware/system combination; it is not portable, say, from a Mac to a Windows PC. When we talk about portability of C++ programs, we usually mean portability of source code; that is, the source code can be successfully compiled and run on a variety of systems.

The ISO C++ standard defines two kinds of entities:

- *Core language features*, such as built-in types (e.g., **char** and **int**) and loops (e.g., **for**-statements and **while**-statements)
- *Standard-library components*, such as containers (e.g., **vector** and **map**) and I/O operations (e.g., **<<** and **getline()**)

The standard-library components are perfectly ordinary C++ code provided by every C++ implementation. That is, the C++ standard library can be implemented in C++ itself and is (with very minor uses of machine code for things such as thread context switching). This implies that C++ is sufficiently expressive and efficient for the most demanding systems programming tasks.

C++ is a statically typed language. That is, the type of every entity (e.g., object, value, name, and expression) must be known to the compiler at its point of use. The type of an object determines the set of operations applicable to it.

1.2.1 Hello, World!

The minimal C++ program is

```
int main() { }          // the minimal C++ program
```

This defines a function called **main**, which takes no arguments and does nothing.

Curly braces, **{ }**, express grouping in C++. Here, they indicate the start and end of the function body. The double slash, *//*, begins a comment that extends to the end of the line. A comment is for the human reader; the compiler ignores comments.

Every C++ program must have exactly one global function named **main()**. The program starts by executing that function. The **int** integer value returned by **main()**, if any, is the program's return value to "the system." If no value is returned, the system will receive a value indicating successful completion. A nonzero value from **main()** indicates failure. Not every operating system and execution environment make use of that return value: Linux/Unix-based environments do, but Windows-based environments rarely do.

Typically, a program produces some output. Here is a program that writes **Hello, World!**:

```
#include <iostream>

int main()
{
    std::cout << "Hello, World!\n";
}
```

The line **#include <iostream>** instructs the compiler to *include* the declarations of the standard stream I/O facilities as found in **iostream**. Without these declarations, the expression

```
std::cout << "Hello, World!\n"
```

would make no sense. The operator **<<** ("put to") writes its second argument onto its first. In this case, the string literal **"Hello, World!\n"** is written onto the standard output stream **std::cout**. A string literal is a sequence of characters surrounded by double quotes. In a string literal, the backslash character **** followed by another character denotes a single "special character." In this case, **\n** is the newline character, so that the characters written are **Hello, World!** followed by a newline.

The **std::** specifies that the name **cout** is to be found in the standard-library namespace (§3.4). I usually leave out the **std::** when discussing standard features; §3.4 shows how to make names from a namespace visible without explicit qualification.

Essentially all executable code is placed in functions and called directly or indirectly from **main()**. For example:

```
#include <iostream>          // include ("import") the declarations for the I/O stream library

using namespace std;         // make names from std visible without std:: (§3.4)

double square(double x)      // square a double precision floating-point number
{
    return x*x;
}

void print_square(double x)
{
    cout << "the square of " << x << " is " << square(x) << "\n";
}

int main()
{
    print_square(1.234);     // print: the square of 1.234 is 1.52276
}
```

A "return type" **void** indicates that a function does not return a value.

1.3 Functions

The main way of getting something done in a C++ program is to call a function to do it. Defining a function is the way you specify how an operation is to be done. A function cannot be called unless it has been previously declared.

A function declaration gives the name of the function, the type of the value returned (if any), and the number and types of the arguments that must be supplied in a call. For example:

```
Elem* next_elem();      // no argument; return a pointer to Elem (an Elem*)
void exit(int);         // int argument; return nothing
double sqrt(double);    // double argument; return a double
```

In a function declaration, the return type comes before the name of the function and the argument types come after the name enclosed in parentheses.

The semantics of argument passing are identical to the semantics of initialization (§3.6.1). That is, argument types are checked and implicit argument type conversion takes place when necessary (§1.4). For example:

```
double s2 = sqrt(2);          // call sqrt() with the argument double{2}
double s3 = sqrt("three");    // error: sqrt() requires an argument of type double
```

The value of such compile-time checking and type conversion should not be underestimated.

A function declaration may contain argument names. This can be a help to the reader of a program, but unless the declaration is also a function definition, the compiler simply ignores such names. For example:

```
double sqrt(double d);    // return the square root of d
double square(double);    // return the square of the argument
```

The type of a function consists of its return type and the sequence of its argument types. For example:

```
double get(const vector<double>& vec, int index);    // type: double(const vector<double>&,int)
```

A function can be the member of a class (§2.3, §4.2.1). For such a *member function*, the name of its class is also part of the function type. For example:

```
char& String::operator[](int index);    // type: char& String::(int)
```

We want our code to be comprehensible, because that is the first step on the way to maintainability. The first step to comprehensibility is to break computational tasks into meaningful chunks (represented as functions and classes) and name those. Such functions then provide the basic vocabulary of computation, just as the types (built-in and user-defined) provide the basic vocabulary of data. The C++ standard algorithms (e.g., **find**, **sort**, and **iota**) provide a good start (Chapter 12). Next, we can compose functions representing common or specialized tasks into larger computations.

The number of errors in code correlates strongly with the amount of code and the complexity of the code. Both problems can be addressed by using more and shorter functions. Using a function to do a specific task often saves us from writing a specific piece of code in the middle of other code; making it a function forces us to name the activity and document its dependencies.

If two functions are defined with the same name, but with different argument types, the compiler will choose the most appropriate function to invoke for each call. For example: