Broadview®
www.broadview.com.cn

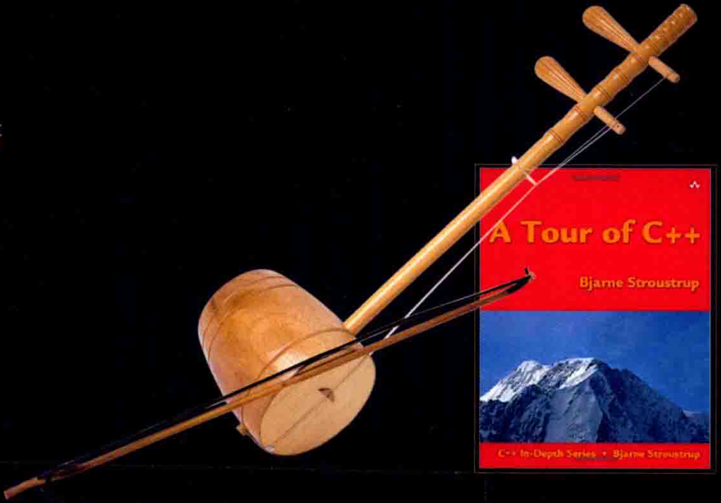
PEARSON

· 原味精品书系 ·

C++之旅（英文版）

A Tour of C++

[美] Bjarne Stroustrup 著



中国工信出版集团 電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

·原味精品书系·

C++之旅（英文版）

A Tour of C++

[美] Bjarne Stroustrup 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书作者是 C++ 语言的设计者和最初实现者，本书的写作目的是让有经验的程序员快速了解 C++ 现代语言。书中几乎介绍了 C++ 语言的全部核心功能和重要的标准库组件，以很短的篇幅将 C++ 语言的主要特性呈现在读者面前，并给出一些关键示例，让读者在很短的时间内就能对现代 C++ 的概貌有一个清晰的了解，尤其是关于面向对象编程和泛型编程的知识。本书没有涉及太多 C++ 语言的细节，非常适合想熟悉 C++ 语言最新特性的 C/C++ 程序设计人员，以及精通其他高级语言而想了解 C++ 语言特性和优点的人员。

图书在版编目（CIP）数据

C++ 之旅 =A Tour of C++：英文 /（美）斯特劳斯特卢普（Stroustrup,B.）著. — 北京：电子工业出版社，2016.4
（原味精品书系）
ISBN 978-7-121-27330-8
I. ① C… II. ① 斯… III. ① C 语言 — 程序设计 — 英文 IV. ① TP312

中国版本图书馆 CIP 数据核字（2015）第 234578 号

责任编辑：张　玲
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980 1/16　　印张：12　　字数：235 千字
版　　次：2016 年 4 月第 1 版
印　　次：2016 年 4 月第 1 次印刷
定　　价：55.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

前言

教而至简，不亦乐乎。

——西塞罗

现在的C++仿佛进化成了一种新的语言。与C++98相比，C++11更易于我们清晰、简洁、直观地表达思想。而且编译器可以将程序中的错误更好地检查出来，程序的运行速度也越来越快。

与其他任何一种现代编程语言相同，C++ 的规模非常庞大，且提供的库也异常丰富，这些都值得程序员认真学习以便高效地利用。本书的目的是让有经验的程序员快速地了解现代 C++ 语言，因此，本书几乎介绍了 C++ 的全部核心功能和重要的标准库组件。读者只需花费几个小时就能读完本书，但是想必所有人都清楚，要想写出漂亮的 C++ 程序绝非一日之功。本书的目的并非让读者熟练掌握一切，而只是介绍语言的概貌，给出一些经典的例子，然后帮助读者开始自己的 C++ 之旅。如果读者希望深入了解 C++ 语言，请阅读我的另一本著作 *The C++ Programming Language*，*Fourth Edition*（简称 *TC++PL4*）。实际上，本书正是 *TC++PL4* 第 2 章 ~ 第 5 章的扩充版，只是出于完整性和独立性的考虑，我们稍微增加了一些内容。本书的篇章结构与 *TC++PL4* 保持一致，读者如果对细节感兴趣，可以在 *TC++PL4* 中进一步寻找答案。同样，在我的个人主页（www.stroustrup.com）上有一些为 *TC++PL4* 编写的习题，也可以用于本书。

我们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一下，比如 *Programming: Principles and Practice Using C++* [Stroustrup, 2009]。即使你曾经编写过程序，你所使用的语言或者编写的应用在风格或形式上也可能与本书相距甚远。

我们用城市观光的例子来比喻本书的作用，比方说参观哥本哈根或者纽约。在短短几个小时之内，你可能会匆匆游览几个主要的景点，听到一些有趣的传说或故事，然后被告知接下来应该参观哪里。但是仅靠这样一段旅程，你无法真正了解这座城市，对听到和看到的东西也是一知半解，更别提熟悉这座城市的生存法则。毕竟要想认识并融入一座城市，需要在这里生活很多年。不过幸运的是，此时你已经对城市的总体情况有了一些了解，知道了它的某些特殊之处，并且对有些方面产生了兴趣。接下来，你就有机会开始真正的探索之旅了。

本书介绍 C++ 语言的主要功能，尤其是关于面向对象编程和泛型编程的知识。在写作时，我们没有涉及太多细节，更不想把本书写成参考手册。对于标准库也尽量去繁就简，用生动的例子进行讲解。本书没有介绍 ISO 标准之外的库，读者需要的话可以自行查阅相

关资料。如果我们提到了某个标准库函数或类，那么读者很容易就能在头文件中找到它的定义，还可以在互联网上搜集到更多与之有关的信息。

本书力求把 C++ 作为一个整体呈现在读者面前，而非逐层地介绍。因此，在这里我们不细分到某项语言特性是归属于 C、C++98 还是 C++11，这些与语言沿革有关的信息在第 14 章可以找到。

致谢

本书的大多数内容源自 *TC++PL4* [Stroustrup, 2012]，因此，首先感谢协助我完成 *TC++PL4* 的所有同仁。还要感谢 Addison-Wesley 的编辑 Peter Gordon，是他建议作者把 *TC++PL4* 的部分章节扩展成本书的。

Bjarne Stroustrup

目录

1

The Basics

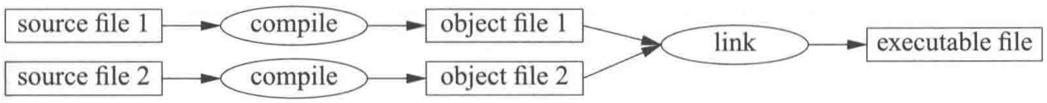
The first thing we do, let's
kill all the language lawyers.
– Henry VI, Part II

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.

1.3 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 often 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 (§3.3). I usually leave out the **std::** when discussing standard features; §3.3 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:: (§3.3)

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.

1.4 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 after the name enclosed in parentheses.

The semantics of argument passing are identical to the semantics of copy initialization. That is, argument types are checked and implicit argument type conversion takes place when necessary (§1.5). 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 the return type and the argument types. For class member functions (§2.3, §4.2.1), the name of the class is also part of the function type. For example:

```
double get(const vector<double>& vec, int index);     // type: double(const vector<double>&,int)
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 comprehensible 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 10). 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:

```
void print(int);        // takes an integer argument
void print(double);     // takes a floating-point argument
void print(string);     // takes a string argument

void user()
{
      print(42);                    // calls print(int)
      print(9.65);                  // calls print(double)
      print("D is for Digital");    // calls print(string)
}
```

If two alternative functions could be called, but neither is better than the other, the call is deemed ambiguous and the compiler gives an error. For example:

```
void print(int,double);
void print(double,int);

void user2()
{
        print(0,0);        // error: ambiguous
}
```

This is known as function overloading and is one of the essential parts of generic programming (§5.4). When a function is overloaded, each function of the same name should implement the same semantics. The **print()** functions are an example of this; each **print()** prints its argument.

1.5 Types, Variables, and Arithmetic

Every name and every expression has a type that determines the operations that may be performed on it. For example, the declaration

```
int inch;
```

specifies that **inch** is of type **int**; that is, **inch** is an integer variable.

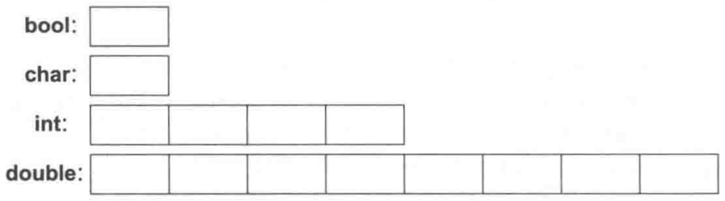
A *declaration* is a statement that introduces a name into the program. It specifies a type for the named entity:

- A *type* defines a set of possible values and a set of operations (for an object).
- An *object* is some memory that holds a value of some type.
- A *value* is a set of bits interpreted according to a type.
- A *variable* is a named object.

C++ offers a variety of fundamental types. For example:

```
bool          // Boolean, possible values are true and false
char          // character, for example, 'a', 'z', and '9'
int           // integer, for example, -273, 42, and 1066
double        // double-precision floating-point number, for example, -273.15, 3.14, and 299793.0
unsigned      // non-negative integer, for example, 0, 1, and 999
```

Each fundamental type corresponds directly to hardware facilities and has a fixed size that determines the range of values that can be stored in it:



A **char** variable is of the natural size to hold a character on a given machine (typically an 8-bit byte), and the sizes of other types are quoted in multiples of the size of a **char**. The size of a type is implementation-defined (i.e., it can vary among different machines) and can be obtained by the

sizeof operator; for example, **sizeof(char)** equals **1** and **sizeof(int)** is often **4**.

The arithmetic operators can be used for appropriate combinations of these types:

```
x+y       // plus
+x        // unary plus
x−y       // minus
−x        // unary minus
x∗y       // multiply
x/y       // divide
x%y       // remainder (modulus) for integers
```

So can the comparison operators:

```
x==y      // equal
x!=y      // not equal
x<y       // less than
x>y       // greater than
x<=y      // less than or equal
x>=y      // greater than or equal
```

Furthermore, logical operators are provided:

```
x&y       // bitwise and
x|y       // bitwise or
x^y       // bitwise exclusive or
~x        // bitwise complement
x&&y      // logical and
x||y      // logical or
```

A bitwise logical operator yields a result of the operand type for which the operation has been performed on each bit. The logical operators **&&** and **||** simply return **true** or **false** depending on the values of their operands.

In assignments and in arithmetic operations, C++ performs all meaningful conversions between the basic types so that they can be mixed freely:

```
void some_function()      // function that doesn't return a value
{
     double d = 2.2;      // initialize floating-point number
     int i = 7;           // initialize integer
     d = d+i;             // assign sum to d
     i = d∗i;             // assign product to i (truncating the double d*i to an int)
}
```

The conversions used in expressions are called *the usual arithmetic conversions* and aim to ensure that expressions are computed at the highest precision of its operands. For example, an addition of a **double** and an **int** is calculated using double-precision floating-point arithmetic.

Note that **=** is the assignment operator and **==** tests equality.

C++ offers a variety of notations for expressing initialization, such as the **=** used above, and a universal form based on curly-brace-delimited initializer lists:

```
double d1 = 2.3;          // initialize d1 to 2.3
double d2 {2.3};          // initialize d2 to 2.3
```

```
complex<double> z = 1;              // a complex number with double-precision floating-point scalars
complex<double> z2 {d1,d2};
complex<double> z3 = {1,2};         // the = is optional with { ... }

vector<int> v {1,2,3,4,5,6};        // a vector of ints
```

The = form is traditional and dates back to C, but if in doubt, use the general {}-list form. If nothing else, it saves you from conversions that lose information:

```
int i1 = 7.8;       // i1 becomes 7 (surprise?)
int i2 {7.8};       // error: floating-point to integer conversion
int i3 = {7.8};     // error: floating-point to integer conversion (the = is redundant)
```

Unfortunately, conversions that lose information, *narrowing conversions*, such as **double** to **int** and **int** to **char** are allowed and implicitly applied. The problems caused by implicit narrowing conversions is a price paid for C compatibility (§14.3).

A constant (§1.7) cannot be left uninitialized and a variable should only be left uninitialized in extremely rare circumstances. Don't introduce a name until you have a suitable value for it. User-defined types (such as **string**, **vector**, **Matrix**, **Motor_controller**, and **Orc_warrior**) can be defined to be implicitly initialized (§4.2.1).

When defining a variable, you don't actually need to state its type explicitly when it can be deduced from the initializer:

```
auto b = true;        // a bool
auto ch = 'x';        // a char
auto i = 123;         // an int
auto d = 1.2;         // a double
auto z = sqrt(y);     // z has the type of whatever sqrt(y) returns
```

With **auto**, we use the = because there is no potentially troublesome type conversion involved.

We use **auto** where we don't have a specific reason to mention the type explicitly. "Specific reasons" include:

- The definition is in a large scope where we want to make the type clearly visible to readers of our code.
- We want to be explicit about a variable's range or precision (e.g., **double** rather than **float**).

Using **auto**, we avoid redundancy and writing long type names. This is especially important in generic programming where the exact type of an object can be hard for the programmer to know and the type names can be quite long (§10.2).

In addition to the conventional arithmetic and logical operators, C++ offers more specific operations for modifying a variable:

```
x+=y      // x = x+y
++x       // increment: x = x+1
x-=y      // x = x-y
--x       // decrement: x = x-1
x*=y      // scaling: x = x*y
x/=y      // scaling: x = x/y
x%=y      // x = x%y
```

These operators are concise, convenient, and very frequently used.

1.6 Scope and Lifetime

A declaration introduces its name into a scope:

- *Local scope*: A name declared in a function (§1.4) or lambda (§5.5) is called a *local name*. Its scope extends from its point of declaration to the end of the block in which its declaration occurs. A *block* is delimited by a **{ }** pair. Function argument names are considered local names.
- *Class scope*: A name is called a *member name* (or a *class member name*) if it is defined in a class (§2.2, §2.3, Chapter 4), outside any function (§1.4), lambda (§5.5), or **enum class** (§2.5). Its scope extends from the opening **{** of its enclosing declaration to the end of that declaration.
- *Namespace scope*: A name is called a *namespace member name* if it is defined in a namespace (§3.3) outside any function, lambda (§5.5), class (§2.2, §2.3, Chapter 4), or **enum class** (§2.5). Its scope extends from the point of declaration to the end of its namespace.

A name not declared inside any other construct is called a *global name* and is said to be in the *global namespace*.

In addition, we can have objects without names, such as temporaries and objects created using **new** (§4.2.2). For example:

```
vector<int> vec;        // vec is global (a global vector of integers)

struct Record {
        string name;    // name is a member (a string member)
        // ...
};

void fct(int arg)       // fct is global (a global function)
                        // arg is local (an integer argument)
{
        string motto {"Who dares wins"};    // motto is local
        auto p = new Record{"Hume"};        // p points to an unnamed Record (created by new)
        // ...
}
```

An object must be constructed (initialized) before it is used and will be destroyed at the end of its scope. For a namespace object the point of destruction is the end of the program. For a member, the point of destruction is determined by the point of destruction of the object of which it is a member. An object created by **new** "lives" until destroyed by **delete** (§4.2.2).

1.7 Constants

C++ supports two notions of immutability:

- **const**: meaning roughly "I promise not to change this value." This is used primarily to specify interfaces, so that data can be passed to functions without fear of it being modified. The compiler enforces the promise made by **const**.