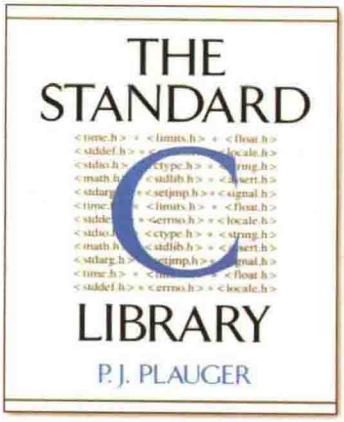
PEARSON

C和C++实务精选
品味岁月积淀，读享技术菁华

# C标准库（英文版）

[美] P.J. Plauger 著

*The Standard C Libarary*




- C标准库圣经
- 提供完整源代码，全面深入阐述库函数的实现与运用
- C程序员必备参考书

人民邮电出版社
POSTS & TELECOM PRESS

PEARSON

# C标准库（英文版）

[美] P.J. Plauger 著

人民邮电出版社
北京

**图书在版编目（CIP）数据**

C标准库 : 英文 / (美) 普劳戈 (Plauger, P. J.) 著
-- 北京 : 人民邮电出版社, 2014.4
ISBN 978-7-115-34422-9

Ⅰ. ①C… Ⅱ. ①普… Ⅲ. ①C语言－程序设计－英文
Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第031230号

## 版权声明



◆ 著　　[美] P. J. Plauger
责任编辑　傅道坤
责任印制　程彦红　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：32
字数：670 千字　　2014 年 4 月第 1 版
印数：1-2 500 册　　2014 年 4 月北京第 1 次印刷
著作权合同登记号　图字：01-2013-8788 号

定价：79.00 元

**读者服务热线：(010)81055410　印装质量热线：(010)81055316**
**反盗版热线：(010)81055315**
**广告经营许可证：京崇工商广字第 0021 号**

# 内容提要

本书是由世界级C语言专家编写的C标准库经典著作，影响了几代程序员。

本书集中讨论了C标准库，全面介绍了ANSI/ISO C语言标准的所有库函数。书中通过引用ISO C标准的相关部分，详细讲解了每一个库函数的使用方法，并通过示例描述了其实现细节，且给出了实现和测试这些函数的完整代码。此外，每章结尾附有不同难度的习题，帮助读者巩固和提高。通过此书，读者将会更好地使用C标准库，并学会如何设计和实现库。

本书结构清晰，内容权威，阐述精辟，对于各层次C程序员和相关专业高校师生而言都是一本优秀的参考书。

# 前言[1]

本书将告诉你如何使用符合C语言的ANSI/ISO 标准的库函数。因为已经有很多书出色地讲解了C语言本身，所以本书只专注于“库”这个话题。本书还会告诉你C标准库是如何实现的。本书提供了大约9 000行测试过的可实际工作的代码。我相信，看了C标准库的实现细节后，你能更好地理解如何使用它。

库函数的实现代码尽可能地使用标准C，这样做有3个设计目的：首先，它使代码具有可读性和示范性；其次，它使代码在各种计算机体系结构间具有高度可移植性；最后，它能使编写的代码兼顾正确性、性能和规模各方面。

教你如何编写C程序并不是本书的目的。本书假定你能读懂简单的C程序，对于那些稍有难度的代码，我会向你解释其中的难点和技巧。

## C标准库

C标准库是非常强大的，它在多种不同的环境下提供了相当多的功能：它允许用户和实现者使用明确定义的名字空间；它对其所提供的数学函数的健壮性和精确性有非常严格的要求；它率先对适应不同文化习惯的代码提供支持，包括那些拥有很大字符集的文化习惯。

为了能有效利用标准库所提供的强大功能，用户应该了解其实现上的很多隐晦细节。库的实现者必须向用户提供这些细节，以使他们更好地使用标准库。C标准中并没

[1] 注：需要提及的是，本书英文原版于 1992 年 7 月出版，由于年代久远，Pearson Education 无法向我们人民邮电出版社提供该书的 PDF 文档。不得已，我们只好将英文原版纸质图书拆开逐页扫描。鉴于纸质图书的清晰度也较为一般，我们虽然尽最大努力进行调优处理，但是最终效果只能说是差强人意。还请购买本书的各位读者多理解和谅解。在此，也向购买本书的读者表示感谢。

有把这些隐晦的实现细节都很好地描述清楚，因为制定标准的主要目的并不是给库的实现者提供指导。与ANSI C标准一起发布的Rationale也没有对这些细节做出很好的解释。Rationale要服务的对象范围很广，而关注这些细节的标准实现者只是众多服务对象的一部分。

在C的传统实现中并不能找到上面提到的新特性。现在的实现已经可以支持国际化开发中的区域设置（locale）概念。每个区域设置都对应于专属的某个国家、某种语言或者某个职业的特定习惯，一个C程序可以通过修改和查询区域设置来动态地适应多种文化。现在的实现也能支持很大的字符集，如字符数量众多的汉字。C程序能把它们作为多字节字符（multibyte character）或者宽字节字符（wide character）处理。它也能在这两种形式之间转换。在迅速加剧的市场竞争中，这就使得程序的编写更加简单和标准化。

因为以前对这些新特性几乎不存在相应的编程艺术，所以即使是最有经验的C程序员，在使用区域设置、多字节字符和宽字节字符的时候也需要一些指导。所以，这些主题在这里给予了特殊的关注。

## 细节

本书向用户和实现者解释了库的设计用意和可能用法。通过提供C标准库中所有库函数的实际实现，本书用例子告诉你怎样处理它的细节。在那些没有明确是最好实现方法的地方，它还讨论了可供选择和折中的办法。

一个涉及细节的例子是函数getchar。头文件(stdio.h)原则上可以用下面的宏来屏蔽函数的声明：

```
#define getchar() fgetc(stdin) /* NOT WISE! */
```

然而，它却不应该这样做，一个合理（即使没有用）的C程序是

```
#include (stdio.h)#undeffgetc
int main(void) { intfgetc = getchar();/* PRODUCES A MYSTERIOUS ERROR */
return (0);
}
```

当然，这个例子有点极端，但它却阐述了即使是一个很好的程序员也可能犯的错误。

用户有权要求尽可能少地出现这类奇怪的错误，所以设计者就有义务避免出现这些奇怪的错误。

我最终确定的getchar宏的形式是：

```
#define getchar () (_Files[0]-)_Next ( _Files[0]-)_Rend \? *_Files[0]-)_Next++ : (getchar) ())
```

它和上面第一次给出的那个显而易见（而且更具可读性）的形式大不相同。第12章会解释其中的原因。

## 库的设计

本书还有一个目的，那就是从一般角度教授程序员怎样设计和实现库。从本质上讲，程序设计语言提供的库是一个混合的“袋子”。库的实现者要具备非常广泛的技巧才能处理这个“袋子”中各种各样的内容。仅仅是一个有能力的数值分析员，或者能熟练高效地操作字符串，或者懂得很多操作系统接口方面的知识，都是不够的。编写库不仅需要具备以上所有能力，还需要掌握更多的知识。

已经有很多好书告诉你怎样编写数学函数，也有很多书专门介绍某种特定用途的库。这些都是告诉你怎样使用现有的库。有些书甚至证明某个库的各种设计方案抉择的正确性。很少有书致力于告诉读者全面地构建一个库要求具备的技能。

## 可复用性

很多书都介绍了设计和实现软件的一般性原则。这些书中提到的方法有结构化分析、结构化设计、面向对象设计和结构化编程等。这些书中的大部分例子只考虑了为某个应用编写的程序。然而，这些原则和方法也同样适用于可复用的库的实现。

但可复用性的目标进一步提高了要求。从结构化设计的角度来看，如果一个库函数没有很高的内聚性，它就不太可能有新的用途。同样，如果它不具备低耦合性，它就会更难使用。简单地说，库函数一定要隐藏它的实现细节并且提供完整的功能。否则，从面向对象的角度看，它们不能实现可复用的数据抽象。

所以本书的最终目的是讨论构建库所特有的设计和实现问题。C标准库的设计是固定的。然而，它从很多方面来看都是一个好的设计，值得讨论。C标准库的实现也可以有很

多选择。任何一个选择都要遵守一定的原则，例如正确性和可维护性等。其他的一些选择还要遵循某个项目特定的优先级，例如高性能、可移植性或者小规模等。这些选择和原则也值得讨论。

## 本书的结构

本书的结构基本对应C标准库本身。标准库中15个头文件声明或定义了库中所有的名字。本书中每一章讲述一个头文件。大部分头文件都有比较紧凑的内容，书中对此也作了紧凑的讨论。然而，也有一两个很笼统，它们相应的章节自然也会涉及更多的内容。

在本书的每一章中，我都摘录了ISO C标准的相关部分。（除了格式细节，ISO和ANSI的C标准是完全相同的。）这些摘录补充说明了库的每一部分通常是怎样使用的。它们也使本书成为一个更完整的参考手册（当然本书比单独阅读C标准要容易理解）。本书也给出了实现那些部分和测试这些实现所需的代码。

每一章最后都附有参考文献和配套习题。假如本书作为大学的教材使用，这些习题可以当成作业。很多习题都只是简单的代码改写，它们可以把某个知识点讲得更清楚或者能够说明在实现上可以做一些合理的修改。那些更具有挑战性的题目都做了标记，可以将它们作为那些长期项目的基础，而自学者则可以把这些习题当作深入思考的入口点。

## 代码

本书中出现的代码已经在Borland、GNU项目和VAX ULTRIX的C编译器上进行了测试。它通过了广泛使用的Plum Hall Validation Suite的库函数的测试，也通过了那些专门为C实现设计的、发现C实现的缺陷的公共域程序。虽然我尽最大努力使错误减到最少，但仍然不能保证代码完全正确。

同时也请注意本书的代码是受版权保护的。它没有放置在公共域中，也不是共享软件。和那些自由软件基金会（GNU项目）发布的代码不同，它不受“copyleft”协议的保护。我保留本书所有的权利。

## 合理使用

你可以把书中的代码转变为电子版的形式，供个人使用。你也可以从堪萨斯州劳伦斯市的C用户组那里购买机器可识别的代码。无论哪种情况，你对代码的使用都要遵守版权法关于“合理使用”的规定。合理使用不允许你以任何形式发布这些代码的副本，不论是打印件还是电子版，也不论是免费的还是收费的。

除了以上谈到的，我允许合理使用范围之外的一种重要用途。你可以编译库中的部分并且把生成的二进制目标模块和你自己的代码连接生成可执行文件。无限发布这样的可执行文件是允许的。对这样的副本我不要求任何权利。但是，我强烈要求你声明库（在你的文件中）的存在，不论你用了多少，也不论是修改过的还是没有修改的。请在可执行文件的某个地方把下面的文字包含进去：“本作品的部分代码源自 The Standard C Library，copyright (c) 1992 by P. J. Plauger, PrenticeHall 出版社出版，这些代码已获授权使用。”随可执行文件一同发布的文档中也要在适当的地方显著地标注这些内容。如果你遗漏了其中的任何内容，将被视为侵权。

## 许可

你也可以被授权做更多的事。你可以以二进制目标模块的形式发布整个库，甚至可以发布本书中的源文件的副本，不论是否修改过。简单地说，你可以把整个库合并到一个让人们使用它来编写可执行程序的产品中。但是，所有的这些都需要许可证。你可以购买许可证。如果你想购买许可证而且想不断地获得库的支持，请和夏威夷州卡姆艾拉市的Plum Hall公司联系。

虽然上面的几段有很浓的商业色彩，但我的主要目的不是推销一个商业产品。我非常信任C标准，也已经很努力地参与了它的制定。我花了很大的精力制定C标准库规范。我想证明我们创建了一个良好的语言标准。编写这个实现和这本书就是为了说明这个简单但很重要的事实。

## 致谢

马萨诸塞州韦克菲尔德市的Compass公司在本书还没完成的时候就很信任我。他们

是我的库代码的第一批使用者，他们帮助我测试、调试，并且在Intel 860 编译器上使用的过程中对这些库代码进行了改进。特别是Ian Wells，愿意忍受我的延迟和修改，体现了他良好的专业精神。Don Anderson先生为了使这个库的结构更合理，牺牲了很多休息时间给我发电子邮件。我衷心地感谢Compass公司的每个同事的真诚和耐心。

Prentice Hall的出版人Paul Becker对本书也很有信心。他的温和与不懈的激励是本书完成的一个保证。他请了一些匿名的审稿人帮助我，使我的观点更加明确，同时也减少了那些绝对化的语句。Paul的职业精神使我明白了Prentice Hall能在技术出版领域辉煌这么多年的原因。

写作过程中我搬到了澳大利亚，这个项目面临着重重困难。我的好朋友，澳大利亚Whitesmiths公司的商业伙伴 John O'Brien经常帮助我。他是一个“把荆棘变成玫瑰”的好手。他对我的帮助很大，不是友谊这个词能概括的。

感谢Prentice Hall的发行经理Andrew Binnie为我提供了完成这本书所用的激光打印机，他乐于在很多方面帮助我。感谢新南威尔士大学计算机科学系给我提供时间和场所，尽管他们也需要这些来完成他们自己的计划。

Tom Plum一直告诫我们要深刻理解C的基础。我也多次从和他对本书中涉及的话题进行的讨论中受益匪浅。Dave Prosser也与我分享了他对C的工作方式的深刻见解。作为ANSI 和ISO C标准的编辑，Dave提供了本书中大量电子版的文本资料。日本东京的高级数据控制公司（率先对C提供日文中的汉字的支持）的两个主要负责人Hiroshi Fukutomi和Takashi Kawahara在我了解日本程序员的技术需求的工作中，给了我很多帮助。

这里的大部分材料原来都在The C Users Journal上发表过。Robert Ward是一个很容易相处的出版人，我很感谢他能让我重新使用这些材料。也同样感谢Jim Brodie允许我使用我们合著的Standard C一书中的内容。

阅读技术类书稿是一项很枯燥的任务。John O'Brien和Tom Plum审阅了本书的部分内容并且反馈了很有价值的信息。发现第一次印刷版本中（只是一小部分）错误的人有：Nelson H. F. Beebe、Peter Chubb、Stephen D. Clamage、Steven Pemberton、Tom Plum和Ian Lance Taylor。

最后，我要感谢我的家人对我的支持。我的儿子Geoffrey在本书的排版和页面设计方面做了很多工作。我的妻子Tana在这么长的时间里给我提供了巨大的精神支持和后勤保障。他们的参与使我的写作过程充满乐趣。

P. J. Plauger

**邦迪，新南威尔士**

# Contents


**Chapter 0: Introduction** 1

Background 1
What the C Standard Says 3
Using the Library 7
Implementing the Library 9
Testing the Library 13
References 15
Exercises 15

**Chapter 1: `<assert.h>`** 17

Background 17
What the C Standard Says 18
Using `<assert.h>` 18
Implementing `<assert.h>` 20
Testing `<assert.h>` 22
References 22
Exercises 24

**Chapter 2: `<ctype.h>`** 25

Background 25
What the C Standard Says 28
Using `<ctype.h>` 30
Implementing `<ctype.h>` 34
Testing `<ctype.h>` 42
References 43
Exercises 43

**Chapter 3: `<errno.h>`** 47

Background 47
What the C Standard Says 50
Using `<errno.h>` 50
Implementing `<errno.h>` 51
Testing `<errno.h>` 55
References 55
Exercises 55

**Chapter 4: <float.h> 57**
Background 57
What the C Standard Says 59
Using <float.h> 62
Implementing <float.h> 64
Testing <float.h> 69
References 71
Exercises 72

**Chapter 5: <limits.h> 73**
Background 73
What the C Standard Says 74
Using <limits.h> 75
Implementing <limits.h> 77
Testing <limits.h> 79
References 80
Exercises 80

**Chapter 6: <locale.h> 81**
Background 81
What the C Standard Says 84
Using <locale.h> 87
Implementing <locale.h> 94
Testing <locale.h> 123
References 123
Exercises 123

**Chapter 7: <math.h> 127**
Background 127
What the C Standard Says 130
Using <math.h> 135
Implementing <math.h> 137
Testing <math.h> 171
References 177
Exercises 177

**Chapter 8: <setjmp.h> 181**
Background 181
What the C Standard Says 184
Using <setjmp.h> 185
Implementing <setjmp.h> 187
Testing <setjmp.h> 191
References 192
Exercises 192

**Chapter 9: <signal.h> 193**
Background 193
What the C Standard Says 195
Using <signal.h> 197
Implementing <signal.h> 199
Testing <signal.h> 203
References 203
Exercises 203

**Chapter 10: <stdarg.h> 205**
Background 205
What the C Standard Says 207
Using <stdarg.h> 208
Implementing <stdarg.h> 210
Testing <stdarg.h> 212
References 212
Exercises 212

**Chapter 11: <stddef.h> 215**
Background 215
What the C Standard Says 217
Using <stddef.h> 217
Implementing <stddef.h> 222
Testing <stddef.h> 223
References 223
Exercises 223

**Chapter 12: <stdio.h> 225**
Background 225
What the C Standard Says 233
Using <stdio.h> 252
Implementing <stdio.h> 274
Testing <stdio.h> 325
References 327
Exercises 329

**Chapter 13: <stdlib.h> 333**
Background 333
What the C Standard Says 334
Using <stdlib.h> 344
Implementing <stdlib.h> 353
Testing <stdlib.h> 381
References 381
Exercises 384

**Chapter 14: `<string.h>`** **387**
Background 387
What the C Standard Says 388
Using `<string.h>` 394
Implementing `<string.h>` 398
Testing `<string.h>` 411
References 411
Exercises 411

**Chapter 15: `<time.h>`** **415**
Background 415
What the C Standard Says 416
Using `<time.h>` 420
Implementing `<time.h>` 425
Testing `<time.h>` 442
References 443
Exercises 443

**Appendix A: Interfaces** **445**

**Appendix B: Names** **453**

**Appendix C: Terms** **463**

**Index** **475**

# Chapter 0: Introduction

## Background

A *library* is a collection of program components that can be reused in many programs. Most programming languages include some form of library. The programming language C is no exception. It began accreting useful *functions* right from the start. These functions help you classify characters, manipulate character strings, read input, and write output — to name just a few categories of services.

**a few definitions**

You must *declare* a typical function before you use it in a program. The easiest way to do so is to incorporate into the program a *header* that declares all the library functions in a given category. A header can also define any associated *type definitions* and *macros*. A header is as much a part of the library as the functions themselves. Most often, a header is a *text file* just like the you write to make a program.

You use the `#include` directive in a C source file to make a header part of the *translation unit*. For example, the header `<stdio.h>` declares functions that perform input and output. A program that prints a simple message with the function `printf` consists of the single C source file:

```
/* a simple test program */
#include <stdio.h>

int main(void)
    {   /* say hello */
    printf("Hello\n");
    return (0);
    }
```

A *translator* converts each translation unit to an *object module*, a form suitable for use with a given *computer architecture* (or *machine*). A *linker* combines all the object modules that make up a program. It incorporates any object modules you use from the C library as well. The most popular form of translator is a *compiler*. It produces an *executable file*. Ideally at least, an executable file contains only those object modules from the library that contain functions actually used by the program. That way, the program suffers no size penalty as the C library grows more extensive. (Another form of translator is an *interpreter*. It may include the entire C library as part of the program that interprets your program.)

**making a library**

You can construct your own libraries. A typical C compiler has a *librarian*, a program that assembles a library from the object modules you specify. The linker knows to select from *any* library only the object modules used by the program. The C library is not a special case.

You can write part or all of a library in C. The translation unit you write to make a library object module is not that unusual:

- A library object module should contain no definition of the function `main` with external linkage. A programmer is unlikely to reuse code that insists on taking control at program startup.
- The object module should contain only functions that are easy to declare and use. Provide a header that declares the functions and defines any associated types and macros.
- Most important, a library object module should be usable in a variety of contexts. Writing code that is highly reusable is a skill you develop only with practice and by studying successful libraries.

After you have read this book, you should be comfortable designing, writing, and constructing specialized libraries in C.

**the C library in C**

The C library itself is typically written in C. That is often *not* the case with other programming languages. Earlier languages had libraries written in *assembly language*. Different computer architectures have different assembly languages. To move the library to another computer architecture, you had to rewrite it completely. C lets you write powerful and efficient code that is also highly *portable*. You can move portable code simply by translating it with a different C translator.

Here, for example, is the library function `strlen`, declared in `<string.h>`. The function returns the length of a null-terminated string. Its pointer argument points to the first element of the string:

```
/* strlen function */
#include <string.h>

size_t (strlen)(const char *s)
    {   /* find length of s[] */
    const char *sc;

    for (sc = s; *sc != '\0'; ++sc)
        ;
    return (sc - s);
    }
```

`strlen` is a small function, one fairly easy to write. It is also fairly easy to write incorrectly in many small ways. `strlen` is widely used. You might want to provide a special version tuned to a given computer architecture. But you don't have to. This version is correct, portable, and reasonably efficient.

Other contemporary languages cannot be used to write significant portions of their own libraries. You cannot, for example, write the Pascal library function `writeln` in portable Pascal. By contrast, you *can* write the