ANSI-C 面向对象编程

Axel-Tobias Schreiner　著

三无舍人　译

饭朵文化教育
2011 年 4 月

ANSI-C 面向对象编程

Axel-Tobias Schreiner 著

三无舍人 译

2011 年 4 月

版本 0.01, Draft

Fandol Culture & Education

饭朵文化教育 荣誉出品

图书在版编目（CIP）数据

ANSI-C 面向对象编程 / Axel-Tobias Schreiner 著，三无舍人 译 / 青岛：饭朵文化教育出版，2011.5
ISBN 7-04-XXXXXX-X

Info···

编　　辑	LiTuX	**校　　对**	LiTuX
封面设计	LiTuX	**印　　制**	LiTuX

出版发行	饭朵文化教育	**地　　址**	山东省青岛市
开　　本	787×960 1/16	**印　　张**	无实体书
字　　数	XXX 000	**版　　次**	第一版
字　　数	XXX 000	**版　　次**	第一版
字　　数	XXX 000	**版　　次**	第一版
字　　数	XXX 000	**版　　次**	第一版
字　　数	XXX 000	**版　　次**	第一版
字　　数	XXX 000	**版　　次**	第一版
字　　数	XXX 000	**版　　次**	第一版
字　　数	XXX 000	**版　　次**	第一版
字　　数	XXX 000	**版　　次**	第一版
字　　数	XXX 000	**版　　次**	第一版

致谢/献给

路人甲、路人乙，东软，山东科技大学

以及

所有热爱和平的程序猿

译者序

我刚刚做出了一个艰难的决定[1]，我决定翻译这本《ANSI-C 面向对象编程》（Object-Oriented Programming With ANSI-C）。这本书是一位刚刚离职的技术狂同事推荐给我的，细看了一点发现这是一本非常棒的书。但是苦于本人英语不佳，从网上又没有搜到中文译本，于是我决定翻译它，方便我自己的阅读，也提供给诸位爱好编程的技术狂人们。

C 语言虽然是一门结构化的语言，但是它是一个非常有技巧的语言，用 C 可以写出非常优美、非常具有艺术性的代码来。C++ 脱胎于 C，虽然是一门面向对象的语言，但是我不得不说，它的确是一门非常糟糕的语言，它的标准复杂到目前没有任何一款编译器支持 C++ 的所有特性，而且不同的编译器甚至是同一款编译器的不同版本，对 C++ 代码的理解是不一样的，我认为这是一件非常糟糕的事情。由于多继承的引入，虽然 C++ 是面向对象的，但是你需要比 C 更加地了解底层才能在胜任多继承情况下 C++ 代码的调试。

这本书从另一个角度去理解 C，去看 C 是如何实现面向对象的。技术之外，我也更想去思考一下我们与老外思想上和技术上的差距，以及造成这种差距的原因。

本人能力有限，翻译中难免有错误的地方，有感兴趣的童鞋希望不吝指摘。今天先翻译了序言，以后会定期更新。由于工作的原因，我不可能投入太大的精力到本书的翻译，预计每一至两周翻译一章。英文原本的下载地址是 http://www.planetpdf.com/codecuts/pdfs/ooc.pdf 或者通过作者网站下载：http://www.cs.rit.edu/~ats/books/ooc.pdf。

本文已经在 Google Code 上托管了文档翻译项目，欢迎感兴趣的童鞋参与。项目的托管地址是 http://code.google.com/p/ooc，通过 SubVersion 版本管理系统进行版本控制管理，英文电子版也可以在本项目下载得到，同时中文翻译的最新版本也会添加到项目的下载列表中，欢迎提出宝贵意见。

译者，
2010 年 11 月 21 日于沈阳

[1]此处遵循腾讯公司古例，有关“艰难的决定”请 Google 搜索“360 大战 QQ”。

原版序

没有能解决掉所有问题的编程技术。
没有能只产生正确结果的编程语言。
没有每个项目都该从头写的程序员。

面向对象编程已经出现了十多年，它目前仍是解决问题的灵丹妙药。本质上，除了接受了二十多年来的一些好的编程法则外并没有什么新的东西带给我们。C++ 是一门新的语言因为它是面向对象的，如果你不想使用或者不知道如何使用那么你不需要使用它，因为普通的 C 就可以实现面向对象。虽然子程序的思想和计算机一样久远并且好的程序员总是随身携带着他们的工具和库，但是只有面向对象才可以在不同项目间实现代码复用。

这本书不准备推崇面向对象或者批评传统的方式。我们准备以 ANSI-C 来发掘如何实现面向对象，有哪些技术，为什么这些技术能帮我们解决更大的问题，如何利用它的一般性以及更早的捕获异常。虽然我们会接触很多术语，如类、继承、实例、连接、方法、对象、多态等等，但是我们将会剥去其魔幻的外表，使用我们熟悉的事物来表述他们。

我非常有意思的发现了 ANSI-C 其实是一门完全的面向对象的语言。如果想要和我分享这份乐趣你需要非常熟悉它，至少也要对结构、指针、原型和函数指针。通过这本书，你将遇到一个“新语言”——按照 Orwell 和韦氏词典对一门语言的解释，语言的设计目的就是缩减思维的广度——而我会尽力证明，它不仅仅汇合了所有的那些你想汇聚到一起的良好的编程原则。结果，你可以成为一个更熟练的 ANSI-C 程序员。

前六章建立 ANSI-C 做面向对象编程的基础。我们从一个抽象数据类型的信息隐藏技术开始，然后加入基于动态连接的通用函数，再通过谨慎地扩充结构来继承代码。最后，我们将上述所有放进一个类树中，来使代码更容易地维护。

编程需要规范。良好的编程更需要很多的规范、众多原则和标准以及确保正确无误的防范措施。程序员使用工具，而优秀的程序员则制作工具来一劳永逸地处理那些重复的工作。用 ANSI-C 的面向对象的编程需要相当大量的不变的代码——名称可能变化但结构不变。因此，在第 7 章里我们搭建一个小小的预处理器，用来创建所需要的模板。它很像是另一个方言式面向对象的语言。但是它不应该这样被看待，它剔除“方言”中枯燥无味的东西，让我们专注于用更好的技术解决问题的创新。OOC 有非常好的可塑性：我们创造了它，了解它，能够改变它，而且它可以如我们所愿的写 ANSI-C 代码。

余下章节继续深入讨论我们的技术。第 8 章加入动态类型检测来实现错误的早期捕获。第 9 章讲我们通过使用自动初始化来防止另一类软件缺陷。第 10 章引入委托代理，说明类和回调函数如何协作，比如去简化标准主程序的生成这样的常规任务。其他章节专注于用类方法来堵塞内存泄漏，用一致的方法来存储和加载结构数据，和通过嵌套异常处理系统的规范错误的恢复。

在最后一章，我们突破 ANSI-C 的限制，做了一个时髦的鼠标操作的计算器——先是针对 *curses* 然后是针对 X Window 系统。这个例子极好地表明：即使是不得不应对外部库和类树的风格，通过对象和类我们已然可以非常精致地进行设计和实现。

每一章都有总结，这些总结中我试图给随意浏览的读者一个梗概以及它对此后章节的重要性。大多数的章节都有练习题，不过他们并不是正式的阐明性文字，因为我坚定的相信读者应当自己实践。由于该技术是我们从无到有建立起来的，所以尽管有些例子应该能够从中获益，但是我避免建立和使用庞大的类库。如果你想要真正地理解面向对象的编程，首先掌握该技术并且在代码设计阶段考虑你的选择更为重要；而开发中依赖使用他人的库应当在这稍后一点。

本书的一个重要部分是所附源码软盘[2]，——其上有一个 DOS 文件系统，包括一个用来按照章节顺序来创建源码的简单 shell 脚本。还有一个 *ReadMe* 文件——在你执行 *make* 命令前要先查阅这个文件。使用一个工具如 *diff* 并且追踪根类和 OOC 报告在后续章节的演化也是非常有帮助的。

这里展现的技术源自我对 C++ 的失望。当时我需要面向对象技术实现一个交互式编程语言，但我意识到无法用 C++ 建立一个可移植的东西来。于是我转向我所了解的 ANSI-C，并且我完全能够做到要做的事情。我将这个些告诉组里的几个人，然后他们用同样的方法完成了他们的工作。如果不是布赖恩 · 克尼翰（Brian Kernighan）以及我的出版商翰斯 · 尼科拉斯（Hans-Joachim Niclas）、约翰 · 维特（John Wait）鼓励我出版这些笔记（在适当的时候全新的展现一下），这个事情很可能就止于此，我的注解也就是一时的时尚了。我感谢他们和所有帮助并且经历本书不断完善的人。最后但是并非不重要的，感谢我的家庭——面向对象当然绝不可能代替餐桌上的面包。

1993 年 10 月于 Hollage
阿塞尔—托彼亚斯 · 斯莱内尔（Axel-Tobias Schreiner）

[2]由于本书没有出版实体书，故无法附带这个软盘，但相应的资料可以通过本项目的托管网站 Google Code 下载得到。 ——译注。

目　录

第一章

抽象数据类型：信息隐藏

1.1 数据类型

数据类型是每种编程语言不可缺少的一部分。ANSI-C 有 `int`， `double` 和 `char` 这几种。程序员们很少满足于现状，而编程语言通常会提供了从预定义的类型创建新数据类型的机制。一种简单的方法是构造集合如数组，结构体和联合体。被 C. A. R. Hoare 称为“我们永远无法收回的一步”的指针，允许我们表示和操作本质上极其复杂的数据。

到底什么是数据类型？我们可以从几个方面来看。数据类型就是值（value）的集合——`char` 就有 256 种不同的值，`int` 就更多了；两者都有均匀的间隔，或多或少有点像是数学里的自然数或整数。`double` 类型的值就更多了，但是它们实际上却不像数学里的实数。

或者我们可以定义一种数据类型，它是一组值的集合以及作用于该集合之上的操作。一般来说，这些值是计算机能够表示的，而这些操作多少反映了可用的硬件指令。ANSI-C 中的 `int` 在这方面上就做得不太好：值的集合在不同的机器上可能是不同的，并且如算数右移一样的操作也会不同。

更加复杂的例子没有更好的用处。一般我们可以用结构体来定义线性表的元素

```
#include<stdio.h>
typedef struct node {
    struct node * next;
    ... information ...
} node;
```

至于对线性表的操作我们指定像这样的函数头部：

```
node * head(node * elt, const node * tail);
```

但是，这种办法是很松散的。好的编程原则要求我们封装数据项表示，并且只声明可能的操作。

1.2 抽象数据类型

如果我们不打算把具体表示暴露给用户，那么我们称这个数据类型是**抽象的** 。理论上将这要求我们通过涉及到适当操作的数学公理来指定这个数据类型的属性。例如，只有当我们首先经常往一个队列里添加元素时我们才可以删除它，并且我们取回这个元素的顺序与它们被加入的顺序是相同的。

抽象数据类型为程序员提供了巨大的灵活性。既然数据表示不是类型定义的一部分，我们可以自由的选择更加简单或者最有效的方法。如果我们正确区分了必要的信息（数据类型的内部信息），那么数据类型的使用与我们选择实现的方法就完全独立了。

抽象数据类型满足了**信息隐藏**以及使用与实现**分而治之**的良好编程原则。信息（如数据项的表示）只要被提供给需要知道的它的人就可以了：是实现者而不是使用者。通过抽象数据类型我们可以清楚分开实现与使用：我们在把大系统分成小模块的道路上顺利前行。

1.3 一个例子：*Set*

那么我们怎么实现抽象数据类型呢？作为一个例子我们考虑一个带有 *add*、 *find* 和 *drop* 操作的元素集合[1]。它们适用于一个集合、一个元素，并且返回添加的、查找到的或者从集合中删除的元素。*find* 可以被用来实现 *contains*，它告诉我们这个集合中是否已经包含了这个元素。

从这个角度，集合就是一种抽象数据类型。为了声明我们可以对一个集合做什么，我们创建了一个头文件 *Set.h*

```
#ifndef SET_H
#define SET_H

extern const void * Set;

void * add(void * set, const void * element);
void * find(const void * set, const void * element);
void * drop(void * set, const void * element);
int contains(const void * set, const void * element);

#endif
```

预处理语句保护下面的声明：无论我们包含多少次 *Set.h*，C 编译器只会看到它一次。这种保护头文件的技术是非常标准的，GNU C 编译器可以识别它并且当这个保护符号（宏 SET_H）已经被定义了的时候不会再访问 *Set.h*。

[1] 不幸的是，*remove* 是一个用来删除文件的 ANSI-C 库函数。如果我们使用其命名一个 set 的函数，我们就不能再包含 *stdio.h* 了。

Set.h 是完整的，但是它有用么？我们几乎不能暴露或者显示更少的内容了： `Set` 必须以某种方式来实现操作这个集合：`add()` 获得一个元素并且将其添加进集合，返回该元素，无论这个元素是新加入的还是已经存在于集合中； `find()` 在集合中查找一个元素，返回该元素或者空指针；`drop()` 定位一个元素，从集合中将其删除，并且返回我们删除的元素；`contains()` 把 `find()` 的结果转换成一个真值。

我们一直使用通用指针 `void *`。一方面它使得了解一个集合到底有什么内容变得不可能，但另一方面它允许我们向 `add()` 和其他的函数传递任何东西。并使任何东西都会表现的像一个集合或者一个元素——为了信息隐藏我们牺牲了类型安全。然而，我们将在第八章中看到如何才能使这种方式变得实现。

1.4 内存管理

我们可能已经忽略了一些事情：如何获得一个集合呢？`Set` 是一个指针，不是一个使用 `typedef` 定义的类型；所以，我们不能定义一个 `Set` 类型的局部或者全局变量，而只能通过指针来指示集合和元素。并且我们在 *new.h* 中声明所有数据项的资源申请与释放的方法：

```
void * new(const void * type, ...);
void delete(void * item);
```

就像 *Set.h* 一样，这个文件被预处理符号 `NEW_H` 保护。本文只显示了每个新文件中更有意义的部分。源文件软盘包含所有示例的完整代码。

`new()` 接受类似于 `Set` 一样的描述符以及更多的可能使用的初始化参数，它返回一个指向新创建的符合描述参数的数据项的指针。`delete()` 接受一个有 `new()` 创建的指针并且回收相关资源。

`new()` 和 `delete()` 大体上是 ANSI-C 函数 `calloc()` 和 `free()` 的一个（向用户暴露的）前端函数。如果使用它们，描述符至少需要指明需要申请内存的大小。

1.5 对象

如果我们需要收集一个 set 中任何感兴趣的东西，我们就需要在头文件 *Object.h* 中描述另外一个抽象数据类型 `Object`：

```
extern const void * Object;      /* new(Object); */

int differ (const void * a, const void * b);
```

`differ()` 能够比较对象：如果它们不相等就返回 `true`，否则返回 `false`。这样的描述为 `strcmp()` 函数的应用留下空间：对于一些成对的对象我们可以选择返回正值或者负值来确定其次序。

生活中的对象需要更多的功能来完成一些实际的事情。目前，我们把自己约束在成为一个集合中成员这样一个单纯的需求。如果我们创建一个更大的类库，我们会发现集合——事实上任何其他的东西——也是一个对象。在这点上，大量的实际情况也多少给予了我们自由。

1.6　一个应用程序

通过这些定义抽象数据类型的头文件，我们就可以写一个程序 *main.c*

```c
#include <stdio.h>

#include "new.h"
#include "Object.h"
#include "Set.h"

int main ()
{
    void * s = new(Set);
    void * a = add(s, new(Object));
    void * b = add(s, new(Object));
    void * c = new(Object);

    if ( contains(s, a) && contains(s, b) )
        puts("ok");

    if ( contains(s, c) )
        puts("contains?");

    if ( differ(a, add(s, a)) )
        puts("differ?");

    if ( contains(s, drop(s, a)) )
        puts("drop?");

    delete( drop(s, b) );
    delete( drop(s, c) );

    return 0;
}
```

我们建立一个集合并且加入了两个新的对象。如果一切正常，我就能在集合中找到这两个对象，并且诶集合找不到其他的对象。这个程序应该简单的打印出 `ok`。

对于 **differ()** 的调用展示了一个语义：一个数学集合中只能包含一份对象 **a** 的拷贝；如果试图再加入一次就会返回原始对象并且 **differ()** 返回 **false**。类似的，一旦我们删除了这个对象，它就不在这个集合中了。

删除一个不在集合中的元素会产生一个空指针传递给 **delete()**。目前我们坚持 **free()** 的语义并且它是可以接受空指针的。

1.7 一个实现：*Set*

main.c 可以正确编译，但是我们在链接并且执行这个程序之前，我们必须实现这个抽象数据类型并且完成内存管理部分。如果一个对象不存放任何信息并且每个对象至多只属于一个集合，那么我们就可以把每个对象和集合都视为唯一的小整数，那么也就可以作为数组 **heap[]** 的索引。如果一个对象是一个集合的元素，那么该对象的数组元素中存储着其所在集合的整数索引值。这样，对象指向了包含它的集合。

第一个方案是如此的简单以至于我们可以把所有的模块合到一个简单的文件 *Set.c* 中。集合和对象有相同的表示，所以 **new()** 不再考虑类型描述。它只返回一个 **heap[]** 中值为 0 的元素：

```c
#if ! defined MANY || MANY < 1
#define MANY    10
#endif

static int heap [MANY];

void * new (const void * type, ...)
{
    int * p;        /* & heap[1..] */

    for ( p = heap + 1; p < heap + MANY; ++p )
        if ( ! *p )
            break;
    assert( p < heap + MANY );
    *p = MANY;
    return p;
}
```

We use zero to mark available elements of **heap[]**; therefore, we cannot return a reference to **heap[0]** —if it were a set, its elements would contain the index value zero.

Before an object is added to a set, we let it contain the impossible index value **MANY** so that **new()** cannot find it again and we still cannot mistake it as a member of any set.

new() can run out of memory. This is the first of many errors, that “cannot happen”.

We will simply use the ANSI-C macro **assert()** to mark these points. A more realistic implementation should as least print a reasonable error message or use a general function for error handing which the user may overwrite. For our purpose of developing a coding technique, however, we prefer to keep the code uncluttered. In chapter 13 we will look at a general technique for hading exceptions.

delete() has to be careful about null pointers. An element of **heap[]** is recycled by setting it to zero:

```
void delete (void * _item)
{
    int * item - _item;
    if (item)
    {   assert(item > heap && item < heap + MANY);
        * item = 0;
    }
}
```

We need a uniform way to deal with generic pointers; therefore, we prefix their names with an underscore and only use them to initialize local variables with the desired types and with the appropriate names.

A set is represented in its objects: each element points to the set. If an element contains **MANY**, it can be added to the set, otherwise, it should already be in the set because we do not permit an object to belong to more than one set.

```
void * add (void * _set, const void * _element)
{
    int * set = _set;
    const int * element = _element;

    assert(set > heap && set < heap + MANY);
    assert(* set == MANY);
    assert(element > heap && element < heap + MANY);

    if (* element == MANY)
        * (int *) element = set - heap;
    else
        assert(* element == set - heap);

    return (void *) element;
}
```

assert() takes out a bit of insurance: we would only like to deal with pointers into **heap[]**

and the set should not belong to some other set, i.e., its array element value ought to be **MANY**.

The other functions are just as simple. **find()** only looks if its element contains the proper index for the set:

```
void * find (const void * _set, const void * _element)
{
    const int * set = _set;
    const int * element = _element;

    assert(set > heap && set < heap + MANY);
    assert(* set == MANY);
    assert(element > heap && element < heap + MANY);
    assert(* element);

    return * element == set - heap ? (void *) element : 0;
}
```

contains() converts the result of **find()** into a truth value:

```
int contains (const void * _set, const void * _element)
{
    return find(_set, _element) != 0;
}
```

drop() can rely on **find()** to check if the element to be dropped actually belongs to the set. If so, we return it to object status by marking it with **MANY**:

```
void * drop (void * _set, const void * _element)
{
    int * element = find(_set, _element);
    if (element)
        * element = MANY;
    return element;
}
```

If we were pickier, we could insist that the element to be dropped not belong to another set. In this case, however, we would replicate most of the code of **find()** in **drop()**.

Our implementation is quite unconventional. It turns out that we do not need **differ()** to implement a set. We still need to provide it, because our application uses this function.

```
int differ (const void * a, const void * b)
{
    return a != b;
}
```

Objects differ exactly when the array indices representing them differ, i.e., a simple pointer comparison is sufficient.

We are done — for this solution we have not used the descriptors **Set** and **Object** but we have to define them to keep our C compiler happy:

```
const void * Set;
const void * Object;
```

We did use these pointers in **main()** to create new sets and objects.

1.8 另一个实现：*Bag*

Without changing the visible interface in *Set.h* we can change the implementation. This time we use dynamic memory and represent sets and objects as structures:

```
struct Set { unsigned count; };
struct Object { unsigned count; struct Set * in;};
```

count keeps track of the number of elements in a set. For an element, **count** records how many times this element has been added to the set. If we decrement **count** each time the element is passed to **drop()** and only remove the element once **count** is zero, we have a *Bag*, i.e., a set where elements have a reference count.

Since we will use dynamic memory to represent sets and objects, we need to initialize the descriptors **Set** and **Object** so that **new()** can find out how much memory to reserve:

```
static const size_t _Set = sizeof(struct Set);
static const size_t _Object = sizeof(struct Object);

const void * Set = & _Set;
const void * Object = & _Object;
```

new() is now much simpler:

```
void * new (const void * type, ...)
{
    const size_t size = * (const size_t *) type;
    void * p = calloc(1, size);

    assert(p);
    return p;
}
```

`delete()` can pass its argument directly to `free()` — in ANSI-C a null pointer may be passed to `free()`.

`add()` has to more or less believe its pointer arguments. It increments the element's reference counter and the number of elements in the set:

```
void * add (void * _set, const void * _element)
{
    struct Set * set = _set;
    struct Object * element = (void *) _element;
    assert(set);
    assert(element);
    if (! element -> in)
        element -> in = set;
    else
        assert(element -> in == set);
    ++ element -> count, ++ set -> count;
    return element;
}
```

`find()` still checks, if the element points to the appropriate set:

```
void * find (const void * _set, const void * _element)
{
    const struct Object * element = _element;

    assert(_set);
    assert(element);

    return element -> in == _set ? (void *) element : 0;
}
```

`contains()` is based on `find()` and remains unchanged.

If `drop()` finds its element in the set, it decrements the element's reference count and the number of elements in the set. If the reference count reaches zero, the element is removed from the set:

```
void * drop (void * _set, const void * _element)
{
    struct Set * set = _set;
    struct Object * element = find(set, _element);
    if (element)
    {
        if (-- element -> count == 0)
            element -> in = 0;
        -- set -> count;
    }
    return element;
}
```

We can now provide a new function **count()** which returns the number of elements in a set:

```
unsigned count (const void * _set)
{
    const struct Set * set = _set;

    assert(set);
    return set -> count;
}
```

Of course, it would be simpler to let the application read the component **.count** directly, but we insist on not revealing the representation of sets. The overhead of a function call is insignificant compared to the danger of an application being able to overwrite a critical value.

Bags behave differently from sets: an element can be added several times; it will only disappear from the set, once it is dropped as many times as it was added. Our application in section 1.6 added the object **a** twice to the set. After it is dropped from the set once, **contains()** will still find it in the bag. The test program now has the output

```
ok
drop?
```

1.9　小结

For an abstract data type we completely hide all implementation details, such as the representation of data items, from the application code.

The application code can only access a header file where a descriptor pointer represents the data type and where operations on the data type are declared as functions accepting and returning generic pointers.

The descriptor pointer is passed to a general function **new()** to obtain a pointer to a data item, and this pointer is passed to a general function **delete()** to recycle the associated resources.

Normally, each abstract data type is implemented in a single source file. Ideally, it has no access to the representation or other data types. The descriptor pointer normally points at least to a constant **size_t** value indicating the space requirements of a data item.

1.10 练习

If an object can belong to several sets simultaneously, we need a different representation for sets. If we continue to represent objects as small unique integer values, and if we put a ceiling on the number of objects available, we can represent a set as a bitmap stored in a long character string, where a bit selected by the object value is set or cleared depending on the presence of the object in the set.

A more general and more conventional solution represents a set as a linear list of nodes storing the addresses of objects in the set. This imposes no restriction on objects and permits a set to be implemented without knowing the representation of an object.

For debugging it is very helpful to be able to look at individual objects. A reasonably general solution are two functions

```c
int store (const void * object, FILE * fp);
int storev (const void * object, va_list ap);
```

`store()` writes a description of the object to the file pointer, **`storev()`** uses **`va_arg()`** to retrieve the file pointer from the argument list pointed to by **`ap`**. Both functions return the number of characters written, **`storev()`** is practical if we implement the following function for sets:

```c
int apply (const void * set,
    int (* action) (void * object, va_list ap), ...);
```

`apply()` calls **`action()`** for each element it **`set`** and passes the rest of the argument list. **`action()`** must not change **`set`** but it may return zero to terminate **`apply()`** early. **`apply()`** returns true if all elements were processed.

第二章

动态链接：通用函数

<++>

2.1 构造器和析构器

<++>

2.2 方法、消息、类和对象

<++>

2.3 Selectors, Dynamic Linkage, and Polymorphisms

<++>

2.4 一个应用程序

<++>

2.5 一个实现：*String*

<++>

2.6 另一个实现：*Atom*

<++>

2.7 小结

<++>

2.8 练习

<++>

第三章

编程常识：算数表达式

3.1 主循环

<++>

3.2 The Scanner

<++>

3.3 The Recognizer

<++>

3.4 The Processor

<++>

3.5 信息隐藏

<++>

3.6 Dynamic Linkage

<++>

3.7 A Postfix Writer

<++>

3.8 Arithmetic

<++>

3.9 Infix Output

<++>

3.10 小结

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第四章

继承：代码重用和精炼

4.1 一个父类：*Point*

<++>

4.2 父类的实现：*Point*

<++>

4.3 继承：*Circle*

<++>

4.4 联系与继承

<++>

4.5 静态和动态链接

<++>

4.6 Visibility and Access Functions

<++>

4.7 父类的实现：*`Circle`*

<++>

4.8 小结

<++>

4.9 Is It or Has It? —Inheritance VS. Aggregates

<++>

4.10 多继承

<++>

4.11 练习

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第五章

编程常识：符号表

5.1 标识符扫描

<++>

5.2 使用变量

<++>

5.3 The Screener —*Name*

<++>

5.4 父类的实现：*Name*

<++>

5.5 子类的实现：*Var*

<++>

5.6 赋值

<++>

5.7 另一个子类：*Constants*

<++>

5.8 数学函数：*Math*

<++>

5.9 小结

<++>

5.10 练习

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第六章

类层次结构：可维护性

6.1 需求

<++>

6.2 元类

<++>

6.3 Roots —*Object and Class*

<++>

6.4 Subclassing —*Any*

<++>

6.5 实现：*Object*

<++>

6.6 实现：*Class*

<++>

6.7 初始化

<++>

6.8 选择器

<++>

6.9 父类选择器

<++>

6.10 一个新的元类：*PointClass*

<++>

6.11 小结

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第七章

ooc 预处理器：执行一种编码标准

7.1 *Point* Revisited

<++>

7.2 设计

<++>

7.3 预处理

<++>

7.4 实现策略

<++>

7.5 *Object* Revisited

<++>

7.6 讨论

<++>

7.7 一个例子：*列表、队列和堆栈*

<++>

7.8 练习

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第八章

动态类型检查：防错性编程

8.1 Technique

<++>

8.2 一个例子：*列表*

<++>

8.3 实现

<++>

8.4 编码标准

<++>

8.5 避免递归

<++>

8.6 小结

<++>

8.7　练习

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第九章

静态构造：自组织

9.1 初始化

<++>

9.2 Initializer Lists —*munch*

<++>

9.3 对象的函数

<++>

9.4 实现

<++>

9.5 小结

<++>

9.6 练习

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第十章

委派——回调函数

10.1 Callbacks

<++>

10.2 Abstract Base Classes

<++>

10.3 Delegates

<++>

10.4 An Application Framework —*`Filter`*

<++>

10.5 The *`respondsTo`* Method

<++>

10.6 Implementation

<++>

10.7 Anothor Application —*`sort`*

<++>

10.8 Summary

<++>

10.9 Exercises

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第十一章

类方法——内存泄漏封堵

11.1 An Example

<++>

11.2 Class Methods

<++>

11.3 Implementation Class Methods

<++>

11.4 Programming Savvy —A Classy Calculator

<++>

11.5 Summary

<++>

11.6 Exercises

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第十二章

对象持久化——存储、加载数据结构

12.1 An Example

<++>

12.2 Storing Objects —*puto()*

<++>

12.3 Filling Objects —*geto()*

<++>

12.4 Loading Objects —*retrieve()*

<++>

12.5 Attaching Objects —*value* Revisited

<++>

12.6 Summary

<++>

12.7 Exercises

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第十三章

异常——Disciplined 错误恢复

13.1 Strategy

<++>

13.2 Implementation —*`Exception`*

<++>

13.3 Examples

<++>

13.4 Summary

<++>

13.5 Exercises

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

第十四章

Forwarding Messages ——一个图形界面计算器

14.1 The Idea

<++>

14.2 Implementation

<++>

14.3 Object–Oriented Design by Example

<++>

14.4 Implementation —*Ic*

<++>

14.5 A Character–Based Interface —*curses*

<++>

14.6 A Graphical Interface —`Xt`

<++>

14.7 Summary

<++>

14.8 Exercises

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

附录 A
ANSI-C 编程提示

C was originally defined by Dennis Ritchie in the appendix of [K&R78]. The ANSI-C standard [ANSI] appeared about ten years later and introduced certain changes and extensions. The differences are summarized very concisely in appendix C of [K&R88]. Our style of object-oriented programming with ANSI-C relies on some of the extensions. As an aid to classic C programmers, this appendix explains those innovations in ANSI-C which are important for this book. The appendix is certainly not a definition of the ANSI-C programming language.

A.1 变量名和作用域

ANSI-C specifies that names can have almost arbitrary length. Names starting with an underscore are reserved for libraries, i.e., they should not be used in application programs.

Globally defined names can be hidden in a translation unit, i.e., in a source file, by using `static`:

```c
static int f(int x) {...}    // Only visible in source file
int g;                        // visible throughout the program
```

Array names are constant addresses which can be used to initialize pointers even if an array references itself:

```c
struct table{ struct table * tp; }
    v[] = { v, v+1, v+2 };
```

It is not entirely clear how one would code a forward reference to an object which is still to be hidden in a source file. The following appears to be correct:

```c
extern struct x object;      // forward reference
```

```
f() { object = value; }      // using the reference
static struct x object;      // hidden definition
```

A.2 函数

ANSI-C permits —but does not require —that the declaration of a function contains parameter declarations right inside the parameter list. If this is done, the function is declared together with the type of its parameters. Parameter names may be specified as part of the function declaration, but this has no bearing on the parameter names used in the function definition.

```
double sqrt();               // classic version
double sqrt(double);         // ANSI-C
double sqrt(double x);       // ... with parameter names
int getpid(void);            // no parameters, ANSI-C
```

If an ANSI-C function prototype has been introduced, an ANSI-C compiler will try to convert argument values into the types declared for the parameters.

Function definitions may use both variants:

```
double sqrt(double arg)      // ANSI-C
{ ... }
double sqrt(arg)             // classic
    double arg;
{ ... }
```

There are exact rules for the interplay between ANSI-C and classic prototypes and definitions; however, the rules are complicated and error-prone. It is best to stick with ANSI-C prototypes and definitions, only.

With the option `-Wall` the GNU-C compiler warns about calls to functions that have not been declared.

A.3 通用指针：*void **

Every pointer value can be assigned to a pointer variable with type `void *` and vice versa, except for `const` qualifiers. The assignments do not change the pointer value. Effectively, this turns off type checking in the compiler:

```
int iv[] = {1, 2, 3};
int * ip = iv;               // OK, same type
void * vp = ip;              // OK, arbitrary to void *
double * dp = vp;            // OK, void * to arbitrary
```

%p is used as a format specification for **printf()** and (theoretically) for **scanf()** to write and read pointer values. The corresponding argument type is `void *` and thus any pointer type:

```
void * vp;
    printf("%p\n", vp);      // display value
    sanf("%p", & vp);        // read value
```

Arithmetic operations involving `void *` are not permitted:

```
void * p, ** pp;
    p + 1                    // wrong
    pp + 1                   // OK, pointer to pointer
```

The following picture illustrates this situation:

A.4 *const*

`const` is a qualifier indicating that the compiler should not permit an assignment. This is quite different from truly constant values. Initialization is allowed; `const` local variables may even be initialized with variable values:

```
int x = 10;
int f(){ const int xsave = x; ... }
```

One can always use explicit typecast operations to circumvent the compiler checks:

```
const int cx = 10;
    (int) cx = 20;           // wrong
    * (int *) & cx = 20;     // not forbidden
```

These conversions are sometimes necessary when pointer values are assigned:

```
const void * vp;

int * ip;
int * const p = ip;              // OK for local variable

    vp = ip;                     // OK, blocks assignment
    ip = vp;                     // wrong, allows assignment
    ip = (void *) vp;            // OK, brute force
    * (const int **) & ip = vp; // OK, overkill
    p = ip;                      // wrong, pointer is blocked
    * p = 10;                    // OK, target is not blocked
```

`const` normally binds to the left; however, `const` may be specified before the type name in a declaration:

```
int const v[10];             // ten constant elements
const int * const cp = v;    // constant pointer to constant value
```

`const` is used to indicate that one does not want to change a value after initialization or from within a function:

```
char * strcpy(char * target, const char * source);
```

The compiler may place global objects into a write-protected segment if they have been completely protected with `const`. This means, for example, that the components of a structure inherit `const`:

```
const struct { int i; } c;
    c.i = 10;                // wrong
```

This precludes the dynamic initialization of the following pointer, too:

```
void * const String;
```

It is not clear if a function can produce a `const` result. ANSI-C does not permit this. GNU-C assumes that in this case the function does not cause any side effects and only looks at its arguments and neither at global variables nor at values behind pointers. Calls to this kind of a function can be removed during common subexpression elimination in the compilation process.

Because pointer values to `const` objects cannot be assigned to unprotected pointers, ANSI-C has a strange declaration for **`bsearch()`**:

```
void * bsearch(const void * key
    const void * table, size_t nel, size_t width,
    int (* cmp) (const void * key, const void * elt));
```

`table[]` is imported with `const`, i.e., its elements cannot be modified and a constant table can be passed as an argument. However, the result of **`bsearch()`** points to a table element and does not protect it from modification.

As a rule of thumb, the parameters of a function should be pointers to `const` objects exactly if the objects will not be modified by way of the pointers. The same applies to pointer variables The result of a function should (almost) never involve `const`.

A.5 *typedef* 和 *const*

`typedef` does not define macros. `const` may be bound in unexpected ways in the context of a `typedef`:

```
const struct Class { ... } * p; // protects contents of structure
typedef struct Class { ... } * ClassP;
```

```
const ClassP cp;              // contents open, pointer protected
```

How one would protect and pass the elements of a matrix remains a puzzle:

```
main()
{
    typedef int matrix [10][20];
    matrix a;
    int b [10][20];

    int f(const matrix);
    int g(const int [10][20]);

    f(a);
    f(b);
    g(a);
    g(b);
}
```

There are compilers that do not permit any of the calls...

A.6 结构体

Structures collect components of different types. Structures, components, and variables may all have the same name:

```
struct u { int u; double v; } u;
struct v { double u; int v; } * vp;
```

Structures components are selected with a period for structure variables and with an arrow for pointers to structures:

```
u.u = vp -> v;
```

A pointer to a structure can be declared even if the structure itself has not yet been introduced. A structure may be declared without objects being declared:

```
struct w * wp;
struct w { ... };
```

A structure may contain a structure:

```
struct a { int x; };
struct b { ... struct a y; ... } b;
```

The complete sequence of component names is required for access:

```
b.y.x = 10;
```

The first component of a structure starts right at the beginning of the structure; therefore, structures can be lengthened or shortened:

```
struct a { int x; };
struct c { sturct a a; ... } c, * cp = & c;
struct a * ap = & c.a;

    assert( (void *) ap == (void *) cp );
```

ANSI-C permits neither implicit conversions of pointers to different structures nor direct access to the components of an inner structure:

```
ap = cp;                        // wrong
c.x, cp -> x                    // wrong
cp -> a.x                       // OK, fully specified
( (sturct a *) cp ) -> x        // OK, explicit conversion
```

A.7 函数指针

The declaration of a pointer to a function is constructed from the declaration of a function by adding one level of indirection, i.e., a `*` operator, to the function name. Parentheses are used to control precedence:

```
void *      f    (void *);        // function
void * (* fp)    (void *) = f;    // pointer to function
```

These pointers are usually initialized with function names that have been declared earlier. In function calls, function names and pointers are used identically:

```
int x;
    f        (& x);             // using a function name
    fp       (& x);             // using a pointer, ANSI-C
    (* fp)   (& x);             // using a pointer, classic
```

A pointer to a function can be a structure component:

```
struct Class { ...
    void * (* ctor) (void * self, va_list * app);
... } * cp, ** cpp;
```

In a function call, `->` has precedence over the function call, but is has no precedence over dereferencing with `*`, i.e., the parentheses are necessary in the second example:

```
cp -> ctor ( ... );
(* cpp) -> ctor ( ... );
```

A.8 预处理器

ANSI-C no longer expands **#define** recursively; therefore, function calls can be hidden or simplified by macros with the same name:

```
#define malloc(type) (type *) malloc(sizeof(type))
int * [ = malloc(int);
```

If a macro is defined withe parameters, ANSI-C only recognizes its invocation if the macro name appears before a left parentheses; therefore, macro recognition can be suppressed in a function header by surrounding the function name with an extra set of parentheses:

```
#include <stdio.h>                   // defines putchar(ch) as a macro
int (putchar) (int ch) { ... }       // name is not replaced
```

Similarly, the definition of a parametrized macro no longer collides with a variable of the same name:

```
#define x(p) (((const struct Object *)(p)) -> x)
int x = 10;                          // name is not replaced
```

A.9 断言：*assert.h*

<++>

A.10 全局跳转：*setjmp.h*

<++>

A.11 变长参数列表：*stdarg.h*

<++>

A.12 数据类型：*stddef.h*

<++>

A.13 内存管理：*stdlib.h*

<++>

A.14 内存函数：*`string.h`*

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

附录 B

ooc 预处理器：*awk* 编程指南

awk was originally delivered with the Seventh Edition of the UNIX system. Around 1985 the authors extended the language significantly and described the result in [AWK88]. Today, there is a POSIX standard emerging and the new language is available in various implementations, e.g., as *nawk* on System V; as *awk*, adapted from the same sources, with the MKS-Tools for MSDOS; and as *gawk* from the (new) *awk* programming language and provides an overview of the implementation of the *ooc* preprocessor. The implementation uses several features of the POSIX standard, and it has been developed with *gawk*.

B.1 构型

ooc is implemented as a shell script to load and execute an *awk* program. The shell script facilitates passing *ooc* command arguments to the *awk* program and it permits storing the various modules in a central place.

The *awk* program collects a database of information about classes and methods from the class description files, and produces C code from the database for interface and representation files and for method headers, selectors, parameter import, and initialization in the implementation files. The *awk* program is based on two design concepts: modularisation and report generation.

A module contains a number of functions and a **BEGIN** clause defining the global data maintained by the functions. *awk* does not support information hiding, but the modules are kept in separate files to simplify maintenance. The *ooc* command script can use **AWKPATH** to locate the files in a central place.

All work is done under control of **BEGIN** clauses which *awk* will execute in order of appearance. Consequently, *main.awk* must be loaded last, because it processes the *ooc* command line.

Pattern clauses are not used. They cannot be used for all files anyway, because *ooc* consults for each class description all class description files leading up to it. The algorithm to read lines, remove comments, and glue continuation lines together is implemented in a single function `get()` in *io.awk*. If pattern clauses were used, the same algorithm would have to be replicated in pattern clauses.

The database can be inspected if certain debugging modules are loaded as part of the *awk* program. These debugging modules use pattern clauses for control, i.e., debugging starts once the command line processing of *ooc* has been completed. Debugging statements are entered from standard input and they are executed by the pattern clauses.

Regular output is produced only by interpreting reports. The design goal is that the *awk* program contain as little information about the generated code as possible. Code generation should be controlled totally by means of changing the report files. Since the *ooc* command script permits substitution of report files, the application programmer may modify all output, at least theoretically, without having to change the *awk* program.

B.2 文件管理：*io.awk*

<++>

B.3 识别：*parse.awk*

<++>

B.4 数据库

<++>

B.5 报告生成：*report.awk*

<++>

B.6 行计数

<++>

B.7 主程序：*main.awk*

<++>

B.8 报告文件

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

B.9 *ooc* 命令

ooc can load an arbitrary number of reports and descriptions, output several interface and representation files, and suggest or preprocess various implementation files, all in one run. This is a consequence of the modular implementation. However, *ooc* is a genuine filter, i.e., it will read files as directed by the command line, but it will only write to standard output. If several outputs are produced in one run, they would have to be split and written to different files by a postprocessor based on *awk* or *csplit*. Here are some typical invocations of *ooc*:

```
$ ooc -R Object -h > Object.h       # root class
$ ooc -R Object -r > Object.r
$ ooc -R Object Object.dc > Object.c

$ ooc Point -h > Point.h            # other class
$ ooc -M Point Circle >> makefile   # dependencies
$ echo "Point.c: Point.d" >> makefile
$ ooc Circle -dc > Circle.dc        # start an implementation
$ ooc Circle -dc | ooc Circle - > Circle.c  # fake...
```

If *ooc* is called without arguments, it produces the following usage description:

```
$ ooc
usage:  ooc [option ...] [report ...] description target ...
options:     -d         arrange for debugging
             -l         make #line stamps
             -Dnm=val   define val for `nm (one word)
             -M         make dependency for each description
             -R         process root description
             -7 -8 ...  versions for book chapters
report:      report.rep load alternative report file
description: class      load class description file
targets:     -dc        make thunks for last 'class'
             -h         make interface for last 'class'
             -r         make representation for last 'class'
             -          preprocess stdin for last 'class'
             source.dc preprocess source for last 'class'
```

It should be noted that if any report file is loaded, the standard reports are not loaded. The way to replace only a single standard report file is to provide a file by the same name earlier on **OOCPATH**.

The *ooc* command script needs to be reviewed during installation. It contains **AWKPATH**, the path for the *awk* processor to locate the modules, and **OOCPATH** to locate the reports. This last variable is set to look in a standard place as a last resort; if *ooc* is called with **OOCPATH** already defined, this value is prefixed to the standard place.

To speed things up, the command script checks the entire command line and loads only the necessary report files. If *ooc* is not used correctly, the script emits the usage description shown above. Otherwise *awk* is executed by the same process.

附录 C

手册

This appendix contains UNIX manual pages describing the final version of *ooc* and some classes developed in this book.

C.1 命令

C.1.1 munch：生成类列表

```
    nm -p object... archive... | munch
```

munch reads a Berkeley-style *nm(1)* listing from standard input and produces as standard output a C source file defining a null-terminated array **classes[]** with pointers to the class functions found in each *object* and *archive*. The array is sorted by class function names.

A class function is any name that appears with type **T** and, preceded with an underscore, with type **b**, **d**, or **s**.

This is a hack to simplify retrieval programs. The compatible effect of option **-p** in Berkeley and System V *nm* is quite a surprise.

Because HP/UX *nm* does not output static symbols, *munch* is not very useful on this system.

C.1.2 ooc：ANSI C 面向对象编码预处理器

```
    ooc [option ...] [report ...] description target ...
```

ooc is an *awk* program which reads class descriptions and performs the routine coding tasks necessary to do object-oriented coding in ANSI C. Code generated by *ooc* is controlled by reports which may be changed. This manual page describes the effects of the standard reports.

`description` is a class name. *`ooc`* loads a class description file with the name *`description`*`.d` and recursively class description files for all superclasses back to the root class. If `-h` or `-r` is specified as a *`target`*, a C header file for the public interface or the private representation of *`description`* is witten to standard output. If *`source`*`.dc` or `-` is specified as a *`target`*, `#include` statements for the *`description`* header files are written to standard output. If `-dc` is specified as a *`target`*, a source skeleton for *`description`* is written to standard output, which contains all possible methods.

C.1.3 *词法*

<++>

C.1.4 *类描述文件*

<++>

C.1.5 *预处理*

<++>

C.1.6 *标签*

<++>

C.1.7 *报告文件*

<++>

C.1.8 *环境*

<++>

C.2 函数

<++>

C.2.1 retrieve：从文件获取对象

<++>

C.3 根类

<++>

C.3.1 简介：根类入门

<++>

C.3.2 Class Class: Object —root metaclass

<++>

C.3.3 Class Exception: Object —manage a stack of exception handlers

<++>

C.3.4 Class Object —root class

<++>

C.4 GUI 计算器类

<++>

C.4.1 intro —introduction to the calculator application

<++>

C.4.2 lcClass Crt: lc —input/output objects for curses

<++>

C.4.3 Class Event: Object —input item

<++>

C.4.4 lcClass: Class Ic: Object —basic input/output/transput objects

<++>

C.4.5 Class Xt: Object —input/output objects for X11

<++>

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs. A quick brown fox jumps over the lazy dogs.

索　引

参考文献

[ANSI] *American National Standard for Information Systems —Programming Language C* X3.159-1989.

[AWK88] A. V. Aho, B. W. Kernighan and P. J. Weingerger *The awk Programming Language* Addison-Wesley 1988, ISBN 0-201-07981-X.

[Bud91] T. Budd *An Introduction to Object-Oriented Programming* Prentice Hall 1991, ISBN 0-201-54709-0.

[Ker82] B. W. Kernighan "pic — A Language for Typesetting Graphics" *Software — Practice and Experience* January 1982.

[K&P84] B. W. Kernighan and R. Pike *The UNIX Programming Environment* Prentice Hall 1984, ISBN 0-13-937681-X.

[K&R78] B. W. Kernighan and D. M. Ritchie *The C Programming Language* Prentice Hall 1978, ISBN 0-13-110163-3.

[K&R88] B. W. Kernighan and D. M. Ritchie *The C Programming Language* Second Edition, Prentice Hall 1988, ISBN 0-13-110362-8.

[Sch87] A. T. Schreiner *UNIX Sprechstunde* Hanser 1987, ISBN 3-446-14894-9.

[Sch90] A. T. Schreiner *Using C with curses, lex, and yacc* Prentice Hall 1990, ISBN 0-13-932864-5.