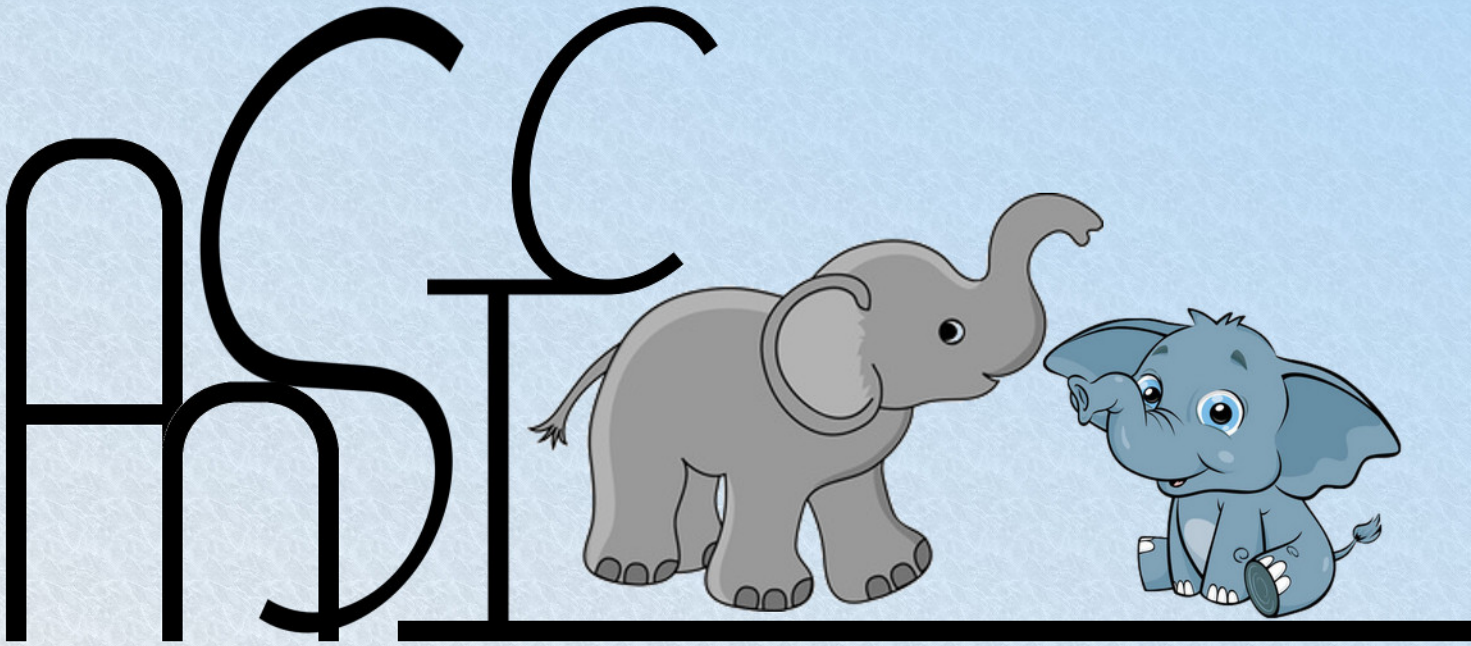
若干技术狂人推荐阅读

ANSI-C 面向对象编程

Axel-Tobias Schreiner 著

LiTuX, github, yuyu391 译



文化教育　中国 · 北京

原译者序

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。

三无舍人，
2010 年 11 月 21 日于沈阳

[1] 此处遵循腾讯公司事例，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或缺的一部分，仅举几例，ANSI-C 就有 `int`，`double` 和 `char`。程序员们极少满足于已有的数据类型，于是编程语言通常都会提供从预定义的类型创建新数据类型的机制，最简单的方法是构造聚合体，比如数组，结构体或是联合体。而指针，依照 C. A. R. Hoare 的话[1]：“……(这种倒退）可能我们永远没办法弥补”，却允许我们表示和操作本质上无限复杂的数据。

到底什么是数据类型呢？我们可以从不同角度来看待这个问题。数据类型就是值（value）的集合——`char` 一般而言有 256 种不同的值，`int` 的取值就多得多；这两者都有均匀的间隔，或多或少像是数学里的自然数或整数。`double` 类型又具有更多数值可取，但是它们却显然不同于数学里的实数。

不同于此，我们还可以定义这样一种数据类型，它包含了一组值的集合以及作用于该集合之上的一些操作。一般来说，这些值应该是计算机能够表示的，而这些操作则多少反映了可用的硬件指令。在这方面，ANSI-C 中的 `int` 就做得不太好：可取值的范围在不同的机器上可能有所不同，并且有些操作例如算数右移的表现也可能有所差异。

考虑到更复杂的例子也没有更大的意义，一般我们会用结构体来定义线性表中的元素

```
1 typedef struct node {
2     struct node * next;
3     ... information ...
4 } node;
```

而对线性表的操作，我们给出如下表头函数：

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

不过，这种处理方式显得非常松散。好的编程原则要求我们封装数据项的表示，并且只声明合适的操作。

[1] Charles Anthony Hoare 的原话为：“Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backwards from which we may never recover”。文中的这句话因没有上下文导致之前的翻译不知所云，是译者的疏忽。——译注

1.2 抽象数据类型

如果我们不把数据表示的细节暴露给用户，那么我们称这个数据类型是**抽象的**。从理论层面上讲，这需要我们利用与可能的操作有关的数学原理来表明这种数据类型的属性。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[2]。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*

```
1  #ifndef SET_H
2  #define SET_H
3
4  extern const void * Set;
5
6  void * add(void * set, const void * element);
7  void * find(const void * set, const void * element);
8  void * drop(void * set, const void * element);
9  int contains(const void * set, const void * element);
10
11 #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*。

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

[2] 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。

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 8 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：

```
1 void * new(const void * type, ...);
2 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**：

```
1 extern const void * Object;      /* new(Object); */
2
3 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*

```
1  #include <stdio.h>
2
3  #include "new.h"
4  #include "Object.h"
5  #include "Set.h"
6
7  int main ()
8  {
9      void * s = new(Set);
10     void * a = add(s, new(Object));
11     void * b = add(s, new(Object));
12     void * c = new(Object);
13
14     if ( contains(s, a) && contains(s, b) )
15         puts("ok");
16
17     if ( contains(s, c) )
18         puts("contains?");
19
20     if ( differ(a, add(s, a)) )
21         puts("differ?");
22
23     if ( contains(s, drop(s, a)) )
24         puts("drop?");
25
26     delete( drop(s, b) );
27     delete( drop(s, c) );
28
29     return 0;
30 }
```

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：

```
#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;
}
```

我们用 0 值来标识 **heap[]** 数组中可用的成员；所以不能返回 **heap[0]** 的引用，——因为如果它是一个集合，那么它所包含的元素可能包括索引值 0。

在向集合中增加对象之前，将一个不可能取到的索引值 **MANY** 赋给该元素对应的数组成员，这样 **new()** 就不可能再次将其分配为新元素，我们也不会将其误认为某个集合的元素。

new() 会用完内存。这是很多“不会发生”的错误的第一个。我们简单的使用 ANSI-C 的 **assert()** 宏来标记这个错误。更为可行的方法应该至少打印出适当的错误信息，或者使用用户可以重写的通用错误处理函数。然而，就增进编程技术而言，我们更倾向于保持代码的干净整洁。13 章中我们会介绍普适的方法来处理异常。

delete() 必须小心处理空指针。当 **heap[]** 数组中的成员被设为 0 时即完成了对其的回收。

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

我们需要一个统一的方式来处理这些通用指针，于是我们用名字前面加下划线的方法来表示这些通用指针，并且只用他们来初始化需要的特定类型和名称的局部变量。

集合是由其对象表示出来的：每个集合中的元素都指向集合。如果一个元素对应的数组成员值为 **MANY**，那么该元素就可以被加入到集合中，不然的话它一定已经在集合中了，因为我们不允许一个对象属于多个集合。

```
void * add (void * _set, const void * _element)
{
```

```
3       int * set = _set;
4       const int * element = _element;
5
6       assert(set > heap && set < heap + MANY);
7       assert(* set == MANY);
8       assert(element > heap && element < heap + MANY);
9
10      if (* element == MANY)
11          * (int *) element = set - heap;
12      else
13          assert(* element == set - heap);
14
15      return (void *) element;
16  }
```

assert() 宏保证了我们不会处理指向 **heap[]** 数组之外的元素，并保证集合不能再属于别的集合，也就是集合对应的数组成员的值应为 **MANY**。

其余的函数也很简单。**find()** 用来检验其 element 参数对应的 **heap[]** 中的数组成员值是不是等于集合对应的数组序号值。

```
1  void * find (const void * _set, const void * _element)
2  {
3       const int * set = _set;
4       const int * element = _element;
5
6       assert(set > heap && set < heap + MANY);
7       assert(* set == MANY);
8       assert(element > heap && element < heap + MANY);
9       assert(* element);
10
11      return * element == set - heap ? (void *) element : 0;
12 }
```

contains() 函数将 **find()** 的结果转化为布尔值：

```
1  int contains (const void * _set, const void * _element)
2  {
3       return find(_set, _element) != 0;
4  }
```

drop() 利用 **find()** 函数来检查要被 drop 的元素是否真的属于该集合。如果是，我们恢复它的状态，标记为 **MANY**，并返回它。

```
1  void * drop (void * _set, const void * _element)
2  {
3       int * element = find(_set, _element);
4       if (element)
5           * element = MANY;
6       return element;
7  }
```

如果我们挑剔一点，我们还得保证要被除去的元素也不能属于别的集合。如果这样，我们拷贝一些 **find()** 中的代码来实现 **drop()**。

我们的实现方法很不寻常。事实上在集合操作中我们甚至不需要 **differ()** 函数。但是我们依旧要写出它，因为我们应用程序的需要。

```
1 int differ (const void * a, const void * b)
2 {
3     return a != b;
4 }
```

元素对象不同也就是他们在数组中对应的序号不同，也就是说，仅仅比较元素对应的指针就足够了。

对于这个方案来说，我们已经完成所有的工作了。至于 **Set** 和 **Object** 描述符其实并不需要，我们这里加上只是使 C 编译器高兴：

```
1 const void * Set;
2 const void * Object;
```

在 **main.c** 中我们就用这两个指针来创造新的集合和元素。

1.8 另一个实现：*Bag*

不需要修改 *Set.h* 中提供的接口我们便可以更改其实现方法。这里我们用动态内存的方法，并且把集合和元素看作结构体：

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

count 用来记录集合中元素的个数，而对于每个元素来说，**count** 记录该元素已经被加入某集合多少次了。如果用 **drop()** 函数从集合中删除该元素的时候我们仅仅对 **count** 减一，直到 **count** 值为 0 时才真正删除该元素，此时的数据结构就叫做 *Bag*，也就是有引用次数记录的元素的集合。

因为我们要用动态存储的方法来表示集合和元素，就需要初始化描述符 **Set** 和 **Object**，这样 **new()** 才知道需要预留多少内存。

```
1 static const size_t _Set = sizeof(struct Set);
2 static const size_t _Object = sizeof(struct Object);
3
4 const void * Set = & _Set;
5 const void * Object = & _Object;
```

new() 函数变得简单多了：

```
1 void * new (const void * type, ...)
2 {
3     const size_t size = * (const size_t *) type;
4     void * p = calloc(1, size);
5
6     assert(p);
7     return p;
8 }
```

delete() 函数也可以将其参数直接传给 **free()** 函数——在 ANSI-C 中空指针也可以传给 **free()** 函数。

add() 必须更多的信任其指针参数了。其将元素引用计数与集合元素数都加一。

```
1  void * add (void * _set, const void * _element)
2  {
3      struct Set * set = _set;
4      struct Object * element = (void *) _element;
5      assert(set);
6      assert(element);
7      if (! element -> in)
8          element -> in = set;
9      else
10         assert(element -> in == set);
11     ++ element -> count, ++ set -> count;
12     return element;
13 }
```

find() 函数依旧要检查要查找的元素是不是指向着合适的集合：

```
1  void * find (const void * _set, const void * _element)
2  {
3      const struct Object * element = _element;
4
5      assert(_set);
6      assert(element);
7
8      return element -> in == _set ? (void *) element : 0;
9  }
```

contains() 函数基于 **find()** 所以没有任何变化。

如果 **drop()** 函数在集合中发现了其参数所指向的元素，便将该元素的引用计数和集合元素数减一。如果该元素的引用计数变为 0，则从集合中删除该元素。

```
1  void * drop (void * _set, const void * _element)
2  {
3      struct Set * set = _set;
4      struct Object * element = find(set, _element);
5      if (element)
6      {
7          if (-- element -> count == 0)
8              element -> in = 0;
9          -- set -> count;
10     }
11     return element;
12 }
```

现在又新加入一个函数 **count()**，用来计算集合中元素的个数：

```
1  unsigned count (const void * _set)
2  {
3      const struct Set * set = _set;
4
5      assert(set);
6      return set -> count;
7  }
```

当然了，如果直接让程序去读取结构体的 **.count** 成员会简单很多，但是我们坚持不应泄漏集合的表示方法。与让程序能够直接修改程序关键数值的危险相比，多调用几次函数的代价真不算什么。

包和集合不同：在包中，任何一个元素可以被多次加入；只有被扔掉的次数和放入包中的次数想同时，它才会从包中消失。在第 1.6 节的程序中，我们将一个元素 **a** 两次加入集合，在将其从包中删除一次后，**contains()** 依旧能够从包中找到该元素。程序会有如下输出：

```
ok
drop?
```

1.9 小结

对于一个抽象数据结构来说，我们完全隐藏了其所有的实现细节，比如数据项的表示方法等等。

程序代码只能访问头文件，该文件仅仅声明数据类型的描述符指针以及作用于该数据类型的操作。

描述符指针传给通用函数 **new()** 来获得一个指向数据项的指针。这个指针传给另一个通用函数 **delete()** 来回收其相关的资源。

一般来说，每个抽象数据类型都在一个单独的代码文件中实现。理想情况下，它也不能看到别的数据类型的具体实现。描述符指针应该至少指向一个常数值 **size_t** 用来指明数据项需要的空间。

1.10 练习

如果一个元素可以同时属于多个集合，集合的实现就必须改变了。如果我们继续用一些惟一的小整数来表示元素，并且限定了可用元素的上限，那么就可以把一个集合表示成一个位图，并以一个长字符串的形式存储。其中某一位如果是选中的就表示其对应的元素在该集合中，否则则不在。

更通用更常见的方法是用线性链表的节点存放集合中元素的地址。这种方法对元素没有任何限制，并且允许在不了解元素的实现方法的情况下实现集合。

能查看单个元素对于调试是很有帮助的。一个合理的通用解决方法是定义两个函数：

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

store() 向文件指针 **fp** 所指文件内写入对元素的描述。**storev()** 则使用 **va_arg()** 取得 **ap** 所指之参数表中的文件指针，然后向该文件指针写入对元素的描述。两个函数都返回写入的字符数。在下列的集合函数中 **storev()** 更实用一些：

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

apply() 对 **set** 中的每一个元素调用 **action()**，并将参数表中其余的参数传递给 **action()**。**action()** 不可以改变 **set**，但可以通过返回 0 而提早终止 **apply()**。若所有元素都得到处理则 **apply()** 返回 **true**。

第二章

动态链接：通用函数

2.1 构造函数和析构函数

让我们先实现一个简单的字符串数据类型，在后面的章节里，我们会把它放入一个集合中。在创建一个新的字符串时，我们分配一块动态的缓冲区来保存它所包含的文本。在删除该字符串时，我们需要回收那块缓冲区。

new() 负责创建一个对象，而 **delete()** 必须回收该对象所占用的资源。**new()** 知道它要创建的对象是什么类型的，因为它的第一个参数为该对象的描述符。依据该参数，我们可以用一系列 **if** 语句来分别处理每一种数据类型的对象的创建。这种做法的的缺点是，对我们所要支持的每一种数据类型，**new()** 中都要显式地包含特定于该数据类型的代码[1]。

然而，**delete()** 所要解决的问题更为棘手。它也必须随着被删除对象的类型的不同而作出不同的动作：若是一个 **String** 对象，则必须释放它的文本缓冲区；若是在第 1 章中用过的那种 **Object** 对象，则只需回收该对象自身；而若是一个 **Set** 对象，则需要考虑它可能已经请求了很多内存块用来储存其元素的引用。

我们可以给 **delete()** 添加一个参数：类型描述符或者做清理工作的函数，但这种方式不仅笨拙，而且容易出错。有一种更为通用更为优雅的方式，即保证每个对象都知道如何去销毁它所占有的资源。可以让每个对象都存有一个指针域，用它可以定位到一个清理函数。我们称这种函数为该对象的析构函数。

现在 **new()** 有一个问题。它负责创建对象并返回一个能传递给 **delete()** 的指针，就是说，**new()** 必须配置每个对象中的析构函数信息。很容易想到的办法，是让指向析构函数的指针成为传递给 **new()** 的类型描述符的一部分。到目前为止，我们需要的东西类似如下声明：

```
1 struct type {
2     size_t size;            /* size of an object */
3     void (* dtor) (void *); /* destructor */
4 };
5 struct String {
6     char * text;            /* dynamic string */
```

[1] 译注：即，需要将数据类型的信息硬编码到 **new()** 中

```
    const void * destroy;   /* locate destructor */
};
struct Set {
    ... information ...
    const void * destroy;
};
```

看起来我们有了另一个问题：需要有人把析构函数的指针 **dtor** 从类型描述符中拷贝到新对象的 **destory** 域，并且该副本在每一类对象中的位置可能还不尽相同。

初始化是 **new()** 工作的一部分，不同的类型有不同的事情要做——**new()** 甚至需要为不同的类型而配备不同的参数列表：

```
new(Set);             /* make a set */
new(String, "text"); /* make a string */
```

对于初始化，我们使用另一种特定于类型（译者注：与类型有绑定性质）的函数，我们称之为构造函数。由于构造函数和析构函数都是特定于类型的，不会改变，我们把他们两个都作为类型描述的一部分传递给 **new()**。

要注意的是，构造函数和析构函数不负责请求和释放该对象自身所需的内存，这是 **new()** 和 **delete()** 的工作。构造函数由 **new()** 调用，只负责初始化 **new()** 分配的内存区域。对于一个字符串来说，构造函数做初始化工作时确实需要申请一块内存来存放文本，但 **struct String** 自身所占空间是由 **new()** 分配的。这块空间最后会被 **delete()** 释放。而首先要做的是，**delete()** 调用析构函数，做与构造函数的初始化相逆的工作，然后才是 **delete()** 回收 **new()** 所分配的内存区域。

2.2 方法、消息、类和对象

delete() 必须能够在不知所给对象类型的情况下定位到析构函数。因此，需要修订第 2.1 节中的声明，对于所有传入 **delete()** 的对象，强调用于定位析构函数的指针必须位于这个对象的头部，而不管这些对象具体是什么类型。

这个指针又应该指向什么呢？如果我们有的只是一个对象的地址，那么这个指针可让我们访问这个对象的类型信息，诸如析构函数。这样看起来我们同样也将很快建立一个其他的类型信息函数，诸如显示对象的函数，或者比较函数 **differ()**，又或者可以创建本对象完整拷贝的 **clone()** 函数。因此我将让这个指针指向一个函数指针表。

如此看来，我们认识到这个指针索引表必须是类型描述的一部分，并且传给 **new()**，并且显而易见的解决方式便是把整个的类型描述作为一个对象，如下所示：

```
struct Class {
    size_t size;
    void * (* ctor) (void * self, va_list * app);
    void * (* dtor) (void * self);
    void * (* clone) (const void * self);
    int (* differ) (const void * self, const void * b);
};
struct String {
    const void * class; /* must be first */
    char * text;
```

```
};
struct Set {
    const void * class; /* must be first */
        ...
};
```

我们的每一个对象开始于一个指向它自身所拥有的类型描述的指针，并且通过这个类型描述，我们能定位这个对象类型描述信息：**.size** 是通过 **new()** 分配的这个对象的长度；**.ctor** 指针指向被 **new()** 函数调用的构造函数，这个构造函数接受被申请的区域和在初始时传递给 **new()** 的其余的参数列表；**.dtor** 指向被 **delete()** 调用的析构函数，用来销毁接受到的对象；**.clone** 指向一个拷贝函数，用来拷贝接受到的对象；**.differ** 指针指向一个用来将这个对象于其他对象进行比较的函数。

大体上看看上面这个函数列表，就能发现每个函数都是通过对象来选择作用于不同的对象的。只有构造函数要处理那些部分初始化的存储区域。我们称这些函数叫做这些对象的方法。调用一个方法就叫做一次消息，我们已经用 **self** 作为函数参数来标记接收该消息的对象。当然这里我们用的是纯 C 函数，所以 **self** 不一定得是函数的头一个参数。

一些对象将共享同样类型描述，就是说，他们需要同样数量的内存和提供同样的方法供使用。我们称所有拥有同样类型描述的对象为一个类；单独的一个对象称作为这个类的实例。到现在为止，一个类、一个抽象数据类型、一些可能的值及其的操作也即一个数据类型，几乎是一样的。

一个对象是一个类的实例，也就是说，在通过 **new()** 为它分配了内存后，它就有了一个状态，并且这个状态可以通过它所属类的方法进行操作。按惯例，一个对象是一个特定数据类型的一个值。

2.3 选择器、动态连接与多态

谁来传递消息？构造函数被 **new()** 调用来处理几乎没初始化的内存区。

```
void * new (const void * _class, ...)
{
    const struct Class * class = _class;
    void * p = calloc(1, class -> size);
    assert(p);
    * (const struct Class **) p = class;
    if (class -> ctor)
    {   va_list ap;
        va_start(ap, _class);
        p = class -> ctor(p, & ap);
        va_end(ap);
    }
    return p;
}
```

把 **struct Class** 指针放在一个对象的开始是非常重要的，这也是为什么初始化这个已经在 **new()** 中的指针。

上图右方的描述数据类型的类是在编译的时候初始化的，对象则是在运行时被创建，此时才将指针安入到其中。在以下的赋值中

```
1 * (const struct Class **) p = class;
```

p 指向该对象的内存区的开始处。我们对 **p** 强制类型转换，使之作为 **struct Class** 的指针，并且把参数 **class** 设置为这个指针的值。

随后，如果在这个类型描述里有构造函数，我们调用它并以它的返回值作为 **new()** 的返回值，即作为 **new** 出的对象。2.6 节会说明一个聪明的构造函数能够对自身的内存进行管理。

要注意的是，只有像 **new()** 那样明确可见的函数能有变参数列表。用文件 *stdarg.h* 里的宏 **va_start()** 对 **va_list** 类型的变量 **ap** 初始化后，就可以通过变量 ap 来访问这个变参列表。**new()**只能把整个列表传递给构造函数。因此，**.ctor** 声明使用一个 **va_list** 参数，而不是它自己的变参列表。由于我们可能在以后想要在几个函数间共享这些原始参数，因此我把 **ap** 的地址传递给构造函数——当它返回后 **ap** 将指向变参列表中第一个没有被构造函数使用的参数。

delete() 假定每个对象，即每个非空指针，指向一个类型描述对象。这是为了在有析构函数时方便调用它。在这里，参数 **self** 扮演着前面图中的指针 **p** 的角色。我们利用本地变量 **cp** 进行强制类型转换，并且非常小心的从 **self** 找到它的类型描述：

```
1 void delete (void * self)
2 {
3     const struct Class ** cp = self;
4     if (self && * cp && (* cp) -> dtor)
5         self = (* cp) -> dtor(self);
6     free(self);
7 }
```

析构函数通过被 **delete()** 调用也获得一个机会来换它自身的指针并传给 **free()**。如果在开始时构造函数想要欺骗，析构函数因而也有机会去更正这个问题，见 2.6 节。如果一个对象不想被删除，它的析构函数可以返回一个空指针。

所有其他存储在类型描述中的方法都以类似的方式被调用。每个方法都有一个参数 **self** 用来接受对象，并且需要通过它的描述符调用方法。

```
1 int differ (const void * self, const void * b)
2 {
3     const struct Class * const * cp = self;
4     assert(self && * cp && (* cp) -> differ);
5     return (* cp) -> differ(self, b);
6 }
```

同样核心部分当然是假定我们能直接通过指针 **self** 找到一个类型描述指针 ***self**。至少现在我们会提防空指针。我们可以是指定一个“幻数”(magic number) 于每个类型描述的开始，或者甚至用 ***self** 与所有已知的类型描述的地址或地址范围进行比较。第 8 章，我们将做更多的严肃的检测。

无论如何，**differ()** 的示例解释为什么这个调用函数的技术被称为动态联接或晚绑定 (late binding)：我们能为任意对象调用 **differ()** 只要他们有一个适当的类型描述指针作为开始，真正工作的函数是尽可能晚得被决定——只有在真正调用时。

我们称 **differ()** 为一个选择函数。这是一个多态函数的例子，即一个函数能接受不同类型的参数，并且根据他们类型表现出不同的行为。一旦我们实现类型描述符中含有 **.differ** 的更多的类，**differ()** 便成为一个能适用于任何对象的通用函数。

我们可以认为选择函数是虽然自身不是动态链接的但仍表现出类似多态函数的方法，这是因为他们让动态链接能作真正的工作。

多态函数通常编译在很多程序语言之中，例如在 Pascal 中 **write()** 函数处理不同的参数类型，C 中的 **+** 操作符能使用在整型、指针或浮点指针上。这叫做重载：参数类型和操作符名共同决定会执行什么样的操作；同样的操作符名能同不同参数类型产生不同的操作效果。

这里没有完全清晰的差别：由于动态链接，至少对于内置的数据类型，**differ()** 的行为如同一个重载函数，并且 C 编译器能产生如同 **+** 产生的那样的多态函数。但是 C 编译器能根据不同的 **+** 操作符返回创建的不同的返回类型，而函数 **differ()** 必须总是有独立于传入参数的返回类型。

方法可以在没有进行动态链接的情况下成为多态的。看如下例子，构造一个函数 **sizeOf()** 返回任何一个对象的大小：

```
1 size_t sizeOf (const void * self)
2 {
3     const struct Class * const * cp = self;
4     assert(self && * cp);
5     return (* cp) -> size;
6 }
```

所有的对象都有他们的描述符，并且我们可以从描述符中获得 **size** 的大小，注意他们的区别：

```
1 void * s = new(String, "text");
2 assert(sizeof s != sizeOf(s));
```

sizeof 是一个 C 的操作符，它被用来在编译时计算并返回他的参数的字节个数。**sizeOf()** 是我们多态函数，它在运行时返回参数所指对象的字节个数。

2.4 一个应用程序

尽管还没有实现字符串，我们仍然能写个简单的测试程序。*String.h* 定义了如下抽象数据类型：

```
1 extern const void * String;
```

我们所有的方法对所有的对象都通用；因此，我们把它们的声明放到 1.4 节介绍的内存管理头文件 *new.h* 中：

```
1 void * clone (const void * self);
2 int differ (const void * self, const void * b);
3 size_t sizeOf (const void * self);
```

前两个原型声明选择函数，它们从 **Class** 结构体中对应的元素中通过简单地从声明函数中去掉一次间接而派生出。以下是应用：

```c
#include "String.h"
#include "new.h"
int main ()
{
    void * a = new(String, "a"), * aa = clone(a);
    void * b = new(String, "b");
    printf("sizeOf(a) == %u\n", sizeOf(a));
    if (differ(a, b))
        puts("ok");
    if (differ(a, aa))
        puts("differ?");
    if (a == aa)
        puts("clone?");
    delete(a), delete(aa), delete(b);
    return 0;
}
```

我们建了两个字符串并复制了其中一个，我们显示了 **String** 对象的大小——不是被对象控制的文本[2]的大小——并且我们确认了两个不同的文本是两个不同的字符串。最后，我们确认了复制的字符串于原来的相等却不相同，然后我们又把它们删掉了。如果一切正常，该程序会打印出：

```
sizeOf(a) == 8
ok
```

2.5 一个实现：*String*

我们用编写需要被加入到 **String** 这个类型描述中去的方法来实现字符串。动态连接有助于明确指定需要编写哪些函数来实现一个新的数据类型。

构造函数的文本从传向 **new()** 的文本中得到，并保存一份动态拷贝于由 **new()** 分配的 **String** 结构体中。

```c
struct String {
    const void * class; /* must be first */
    char * text;
};
static void * String_ctor (void * _self, va_list * app)
{
    struct String * self = _self;
    const char * text = va_arg(* app, const char *);
    self -> text = malloc(strlen(text) + 1);
    assert(self -> text);
    strcpy(self -> text, text);
    return self;
}
```

在构造函数中我们只需要初始化 **.text**，因为 **.class** 已经由 **new()** 设置。

析构函数释放由字符串控制的动态内存。由于只有在 **self** 非空的情况下 **delete()** 才会调用析构函数，因此在这里我们不需要检查其他事情：

[2] 译注：此处的文本指的是 C 字符串，而字符串指的是作者正在实现的对象

```
static void * String_dtor (void * _self)
{
    struct String * self = _self;
    free(self -> text), self -> text = 0;
    return self;
}
```

String_clone() 复制一份字符串。由于之后初始的和复制的字符串都将被传送到 **delete()**，所以我们必须产生一份新的字符串的动态拷贝。这只要简单的调用 **new()** 即可。

```
static void * String_clone (const void * _self)
{
    const struct String * self = _self;
    return new(String, self -> text);
}
```

String_differ() 在比较两个同一的字符串对象时返回 **false**，如果比较的时两个完全不同的对象，则返回 **true**。如果我们比较的是两个不同的字符串，则使用 **strcmp()**：

```
static int String_differ (const void * _self, const void * _b)
{
    const struct String * self = _self;
    const struct String * b = _b;
    if (self == b)
        return 0;
  if (! b || b -> class != String)
      return 1;
  return strcmp(self -> text, b -> text);
}
```

类型描述符是唯一的——在这里我们用这个事实来验证第二个参数是否真的是一个字符串。

这些方法都被设置成静态是因为他们需要被 **new()**，**delete()** 或者其他的选择函数调用。这些方法通过类型描述符使得他们可以被选择函数调用。

```
#include "new.r"
static const struct Class _String = {
    sizeof(struct String),
    String_ctor, String_dtor,
    String_clone, String_differ
};
const void * String = & _String;
```

String.c 包含了 *String.h* 和 *new.h* 中的公共声明。为了能够合适的初始化类型描述符，它也包含了私有头文件 *new.r*，*new.r* 中包括了在 2.2 节中说明的 **struct Class** 的定义。

2.6 另一个实现：*Atom*

为了举例说明我们能使用构造器和析构器接口能做什么，我们实现了 **atoms**。每个 **atom** 是一个唯一的字符串对象；如果两个 **atom** 包含同样的字符串，那么他们是一样

的。**atom** 是非常容易比较的：如果两个参数指针不同则 **differ()** 为 **true**。原子的构造于析构的代价更高：我们为所有的 **atoms** 维护了一个循环队列，并且我们当 **atom** 克隆时就算数量：

```
struct String {
    const void * class;         /* must be first */
    char * text;
    struct String * next;
    unsigned count;
};
static struct String * ring;    /* of all strings */
static void * String_clone (const void * _self)
{
    struct String * self = (void *) _self;
    ++ self -> count;
    return self;
}
```

我们所有的 **atoms** 循环列表是在环中被标志的，通过 **.next** 成员进行扩展，并且被 **string** 的构造函数与析构函数进行维护。在构造器保存一个文本之前，他先遍历表内是否有同样的 **text** 已经存在。下段代码被置于字符串构造函数 **String_ctor()** 之前。

```
if (ring)
{
    struct String * p = ring;
    do
        if (strcmp(p -> text, text) == 0)
        {
            ++ p -> count;
            free(self);
            return p;
        }
    while ((p = p -> next) != ring);
}
else
     ring = self;
self -> next = ring -> next, ring -> next = self;
self -> count = 1;
```

如果我们找到了一个匹配的 **atom**，我们将累加引用计数，并释放新的字符串自身，随后返回 **atom p**。否则我们将新的字符串对象插入到循环队列中，并设置引用计数为 1。

直到 **atom** 的引用计数为 0 时，析构器才释放删除这个 **atom**。以下的代码被置于析构函数 **String_dtor()** 之前：

```
if (-- self -> count > 0)
    return 0;
assert(ring);
if (ring == self)
    ring = self -> next;
if (ring == self)
    ring = 0;
else
```

```
{
    struct String * p = ring;
    while (p -> next != self)
        p = p -> next;
    {
        assert(p != ring);
    }
    p -> next = self -> next;
}
```

如果引用计数是整数，我们返回一个空指针使 **delete()** 不处理我们的对象。否则我们清除循环链表的标志，如果我们的 **string** 是最后一次引用，我们将从链表中删除这个字符串对象。

通过这种方式实现我们的程序，注意克隆出字符串同样是最初的值

```
sizeOf(a) == 16
ok
clone?
```

2.7 小结

给定一个指向一个对象的指针，动态链接让我们找到类型详细而精确的函数：每个对象开始于一个描述符，这个描述符包含一系列指向对象可用的函数。特别是一个描述符包含一个指针指向构造器用来初始化对象的内存区；一个指针指向析构器用来在对象被删除前归还对象的资源。

我们成所有对象共享同样的描述符为一个类。一个对象是类的一个实例，类型详细而精确的函数被称作对象的方法，消息调用这些方法。我们使用选择函数为对象来定位并动态链接调用方法。

通过选择器动态链接不同类的相同的函数名，但会带来不同的结果。这样的函数被称作动态函数。

动态函数是非常有用的。他们提供一个概念上的抽象：**differ()** 将比较任意两个对象，我们不需要记住哪个 **differ()** 的细节标志在具体情况下是适用的。一个低成本并且十分方便的调试工具是多态函数 **store()**，用来显示任何一个在文件描述符上的对象。

2.8 练习

为了查看多态函数的行为，我需要实现对象并通过动态链接进行 Set。对于 Set 这是有难度的，因为我们不再记录这些元素是由谁来设置。(待修改)

对于字符串，应该有更丰富的方法：我们需要知道字符串长度，我们想要指派新的文本内容，我们应该能打印字符串。如果我们想做，将会有很多有趣的事情。

如果我们通过哈希表跟踪原子性的增加是更加有效率的。那么这些值能否被原子的更改?

`String_clone()` 造成了一个敏感的问题：在这个函数 `String` 应该同 `self->class` 用样的值。这样是否会造成我们传递给 `new()` 的参数会有些不同？

第三章

编程常识：算数表达式

动态链接就其本身而言是一种很有用的程序规划技术。相比写比较少的附带许多 `switch` 语句控制许多特殊情况的函数，不如写许多短的函数，这样，在每种情况下，都能很好地动态链接。这也常常简化了我们的程序工作并且它常常使我们的代码容易扩展。

作为例子，我们将要写一个计算由浮点数、圆括号及一些常用的加、减等算术符号组成的算术表达式的程序。一般说来，我们将要通过规定的编译工具和另外的某编译器来生成能计算这个算术运算的目标程序。当然，这本书不是谈论编译器的工作原理，所以，这一次我们将亲自写出程序代码.

3.1 主循环

这个程序的主循环先从标准输入中读取一行，并且根据它进行部分初始化，这样，数字和操作符就能被提取出来，而且空格也被忽略掉了。然后调用一个函数来识别这个正确的算术表达式，并以某种方式存储它，最后处理这些存储的结果。如果出错，那我们可以简单地阅读下一个输入行。下面是主函数:

```
#include <setjmp.h>
static enum tokens token; /* current input symbol */ /*当前输入信号*/
static jmp_buf onError;
int main (void)
{
    volatile int errors = 0;
    char buf [BUFSIZ];

    if (setjmp(onError))
    ++ errors;
    while (gets(buf))
        if (scan(buf))
        {   void * e = sum();
            if (token)
            error("trash after sum");
            process(e);
            delete(e);
```

```
18          }
19      return errors > 0;
20  }
21  void error (const char * fmt, ...)
22  {
23      va_list ap;
24
25      va_start(ap, fmt);
26      vfprintf(stderr, fmt, ap), putc(' \n' , stderr);
27      va_end(ap);
28      longjmp(onError, 1);
29  }
```

错误校正在 **setjmp()** 中定义了。如果 **error()** 函数在程序的某个地方被调用，则伴随着 **setjmp()** 的另一个循环，**longjmp()** 将继续被执行。在这种情况下，结果就就传给了 **longjmp()**，错误也被保存了，同时下一个输入行仍有效。离开当前程序时将报告这段代码是否有错误。

3.2 扫描器

在主循环中，一旦输入数据被传入 **buf[]**，它就会传递至 **scan()** 函数，为每一处调用函数的地方提供下一个输入变量标志。在每行的结束是零标记。

```
1  #include <ctype.h>
2  #include <errno.h>
3  #include <stdlib.h>
4  #include "parse.h" /* defines NUMBER */
5  static double number; /* if NUMBER: numerical value */
6  static enum tokens scan (const char * buf)
7  /* return token = next input symbol */
8  {
9      static const char * bp;
10     if (buf)
11         bp = buf; /* new input line */
12     while (isspace(* bp))
13         ++ bp;
14     if (isdigit(* bp) || * bp == ' .' )
15     {   errno = 0;
16         token = NUMBER, number = strtod(bp, (char **) & bp);
17         if (errno == ERANGE)
18             error("bad value: %s", strerror(errno));
19     }
20     else
21         token = * bp ? * bp ++ : 0;
22     return token;
23 }
```

我们调用 scan() ，可传递输入行缓冲的地址，或传进一个空指针得以继续工作在当前的行。空格被忽略，并且遇到第一个为数字或小数点，我们就是用一个 ANSI-C 的函数 strtod() 开始提取出浮点数字。若为其他的任何字符将被返回，并且我们不会预先在输入缓冲传递一个空字节。

scan() 的结果被存储在全局变量 token ——这样简化了识别程序（识别器）。如果我们侦测出一个数字，我们将返回唯一的值 NUMBER 并使得在全局变量 number 中实际的值有效。

3.3 识别器

在最高水平, 表达式通过函数 sum() 被识别，sum() 函数内部调用 scan() 并返回一个表示，这个表示可通过调用 process() 被处理并通过 delete() 被回收。

如果我们不使用 yacc(是 Unix/Linux 上一个用来生成编译器的编译器（编译器代码生成器)），我们将通过递归下降的方法识别表达式，合乎语义的规则被翻译成等价的 C 函数。例如：一个 sum 是一个产物，接下来被 0 跟随，或更多的组，每个由额外的操作符和另外的产物组成，一个语义规则如下：

sum: product{+|- product}...

其被翻译成 C 函数如下：

```
1  static void * sum (void)
2  {
3      void * result = product();
4      for(;;)
5      {
6          switch (token)
7          {
8              case '+':
9              case '-':
10                 scan(0), product();continue;
11             return;
12         }
13     }
14 }
```

对于每一个语义规则有一个 C 函数，以便于这些规则能够相互调用，这些不同的分支被转换成 switch 或 if 语句，迭代的语法将在 C 中翻译成循环。仅仅一个问题就是我们必须避免无限的递归。

token 总是包含下一个输入的符号。如果我们识别出它，我们必须调用 scan(0)。

3.4 The Processor 处理器

我们如何来处理表达式呢？如果我们仅仅想用一些用数字表示的值执行简单的算术。我们可以扩展识别函数并且一旦识别出操作符和操作码就计算出结果如：sum() 应该会期望从每一个对 product() 的调用期望一个 double 类型的结果，尽可能的执行加或减法，并且返回结果，再次作为一个 double 类型函数的值。

如果我们想要建立一个系统用来处理更加复杂的表达式，我们需要存储表达式以便于后续处理。在这种情况下，我们能够不仅仅执行算术，而且可以允许决定并且有条件

的评估一个表达式的一部分，且可用存储的表达式作为用户的函数包含在其他表达式中。我们所需要的是一个合理通用的方式代表一个表达式。比较常规的技术是使用一个二叉树在每一个节点上存储 token.

```
struct Node{
    enum tokens token;
    struct Node * left, * right;
};
```

然而，这样并不是很灵活。我们需要介绍一个 union 去创建一个节点，在这个节点上我们可存储一个数，并且我们在这些节点代表的一元操作符上浪费了空间。此外，process() 和 delete() 将包含 witch 分支，并 witch 分支会随着我们增加的符号而增多。

3.5 Information Hiding 信息隐藏

应用迄今为止我们学到的，我们绝不去揭示节点结构。相反，我们先在头文件 value.h 中放置一些声明如下：

```
const void * Add;
...
void * new(const void * type, ...);
void process(const void * tree);
void delete(void * tree);
```

现在我们可以编写代码 sum() 如下：

```
#include "value.h"
static void * sum(void)
{
    void * result = product();
    const void * type;

    for (;;)
    {
        switch (token)
        {
            case '+':
                type = Add;
                break;
            case '+':
                type = Sub;
                break;
            default:
                return result;
        }
        scan(0);
        result = new(type, result, product());
    }
}
```

product() 与 sum() 有相同的结构，并且调用一个函数 factor() 去识别数字，符号，且 sum 被赋予了括号：

```
static void * factor(void)
```

```
{
    void * result;

    switch (token)
    {
        case '+':
            scan(0);
            return factor();
        case '-':
            scan(0);
            return factor();
        default:
            error("bad factor: '%c' 0x%x", token, token);
        case NUMBER:
            result = new(Value, number);
            break;
        case '(':
            scan(0);
            result = sum();
            if (token != ')')
                 error("expecting )");
    }
    scan(0);
    return result;
}
```

尤其在 factor() 中，我们需要特别小心的保持扫描器（scanner）是不变的：token 必须总是包含下一个输入的符号。一旦 token 被使用，我们需要调用 scan(0)。

3.6 Dynamic Linkage 动态链接

识别器是完善的。value.h 对于算术表达式完全隐藏了求值程序，且与此同时指定了我们必须所实现的。new() 携带描述符，如 Add 和合适的参数如指针对加的操作且返回一个表示和的指针。

```
struct Type {
    void * (* new) (va_list ap);
    double (* exec) (const void * tree);
    void (* delete) (void * tree);
};

void * new (const void * type, ...)
{
    va_list ap;
    void * result;

    assert(type && ((struct Type *) type) -> new);

    va_start(ap, type);
    result = ((struct Type *) type) -> new(ap);
    * (const struct Type **) result = type;
    va_end(ap);
    return result;
```

```
19 }
```

我们使用动态连接并传递一个对指定节点例程的调用，在例程中的 Add 分支处，必须常见一个节点，并且传进两个指针。

```
1  struct Bin {
2      const void * type;
3      void * left, * right;
4  };
5
6  static void * mkBin (va_list ap)
7  {
8      struct Bin * node = malloc(sizeof(struct Bin));
9
10     assert(node);
11     node -> left = va_arg(ap, void *);
12     node -> right = va_arg(ap, void *);
13     return node;
14 }
```

注意，只有 mkBin() 知道它创建的是什么。所有我们要求的是各个节点对于动态连接是以一个指针开始。这个指针被 new() 传进一遍于 delete() 能够调用到它指定节点的函数：

```
1  void delete (void * tree)
2  {
3      assert(tree && * (struct Type **) tree
4          && (* (struct Type **) tree) -> delete);
5
6      (* (struct Type **) tree) -> delete(tree);
7  }
```

动态连接很优雅的避免了复杂难解的节点。.new() 精确的创建了每个类型描述符的右节点：二元操作符拥有两个子孙。一元操作符拥有一个子孙，且值节点仅仅包含了值。delete() 是一个非常简单的函数因为每个节点处理它自己的销毁过程：二元操作符删除两个子树并且释放他们自己的节点，一元操作符仅仅删除一个子树，且值节点仅仅释放自己。变量和常量甚至可以留到后面——对于 delete() 的回应他们简单的什么也不做。

3.7 A Postfix Writer

到目前为止我们还没有真正的决定 process() 将要真正做什么。如果我们想要发布一个表达式的后缀版，我们将要对 struct Type 增加一个字符串以便于显示出实际的操作符，且 process() 将要安排一个单独的被 tab 键缩进的行：

```
1  void process (const void * tree)
2  {
3      putchar('/t');
4      exec(tree, (* (struct Type **) tree) -> rank, 0);
5      putchar('/n');
6  }
```

exec() 处理动态连接

```
static void exec (const void * tree, int rank, int par)
{
    assert(tree && * (struct Type **) tree
        && (* (struct Type **) tree) -> exec);

    (* (struct Type **) tree) -> exec(tree, rank, par);
}
```

每一个二元操作符被使用如下函数发出：

```
static void doBin(const void *tree)
{
    exec(((struct Bin *) tree) —> left);
    printf(" %s", (* (struct Type **) tree) —> name);
    exec(((struct Bin *) tree) —> right);
}
```

类型描述符如下绑定：

```
static struct Type _Add = { "+", mkBin, doBin, freeBin };
static struct Type _Sub = { "—", mkBin, doBin, freeBin };
const void * Add = & _Add;
const void * Sub = & _Sub;
```

应该很容易猜测一个数值是怎样被实现的。它被代表作为一个结构体携带 double 类型的信：

```
struct Val {
    const void * type;
    double value;
};
static void * mkVal (va_list ap)
{
    struct Val * node = malloc(sizeof(struct Val));
    assert(node);
    node —> value = va_arg(ap, double);
    return node;
}
```

处理组成的打印值：

```
static void doVal (const void * tree)
{
    printf(" %g", ((struct Val *) tree) —> value);
}
```

我们已经做了——没有子树要删除，因此我们可以使用库函数 free() 直接的删除值节点：

```
static struct Type _Value = { "", mkVal, doVal, free };
const void * Value = & _Value;
```

一元操作符如 Minus 将留作练习。

3.8 Arithmetic

如果我们想做算术运算，我们让执行的函数返回一个 double 类型的值，然后让 process() 打印这个值：

```
1 static double exec (const void * tree)
2 {
3     return (* (struct Type **) tree) —> exec(tree);
4 }
5 void process (const void * tree)
6 {
7     printf("/t%g/n", exec(tree));
8 }
```

对于每个节点的类型，我们需要一个执行函数来计算和返回这个节点的值。这里有两个实例：

```
1 static double doVal (const void * tree)
2 {
3     return ((struct Val *) tree) —> value;
4 }
5 static double doAdd (const void * tree)
6 {
7     return exec(((struct Bin *) tree) —> left) +
8     exec(((struct Bin *) tree) —> right);
9 }
10 static struct Type _Add = { mkBin, doAdd, freeBin };
11 static struct Type _Value = { mkVal, doVal, free };
12 const void * Add = & _Add;
13 const void * Value = & _Value;
```

3.9 Infix Output

也许对于处理算术表达式的突出点是带小括号的形式打印。这通常是有点滑稽的，依照谁来负责发出括号。此外对于操作符的名字用于前缀输出，我们增加了两个数值到 struct Type 中。

```
1 struct Type {
2     const char * name; /* node' s name */
3     char rank, rpar;
4     void * (* new) (va_list ap);
5     void (* exec) (const void * tree, int rank, int par);
6     void (* delete) (void * tree);
7 };
```

.rank 是优先的操作符，以 1 开始，此外.rpar 被设置用于操作符，如减操作，此操作如果用于相等的优先级的操作就要求他们的右操作被附上括号。

```
1 $ infix
2 1 + (2 — 3)
3 1 + 2 — 3
4 1 — (2 — 3)
5 1 — (2 — 3)
```

这个证实了我们需要如下的初始化：

```
static struct Type _Add = {"+", 1, 0, mkBin, doBin, freeBin};
static struct Type _Sub = {"—", 1, 1, mkBin, doBin, freeBin};
```

滑稽的部分是对于二元节点得去决定它是否必须要增加括号。一个二元节点如加法，被给予它自己较高的优先级并且一个标记指示在相等的优先级中括号是否是必须的。doBin() 去判别是否使用括号:

```
static void doBin (const void * tree, int rank, int par)
{
    const struct Type * type = * (struct Type **) tree;
    par = type —> rank < rank || (par && type —> rank == rank);

    if (par)
    putchar(' (' );
    exec(((struct Bin *) tree) —> left, type —> rank, 0);
    printf(" %s ", type —> name);
    exec(((struct Bin *) tree) —> right, type —> rank, type —>
        rpar);
    if (par)
    putchar(' )' );
}
```

与高优先级的操作符比若我们有一个较低优先级，或者如果我们被要求在相等的优先级情况下输出括号，我们就打印括号。在任何情况下，如果我们的描述有.rpar 的设置，我们要求仅仅我们的所有操作输出额外的括号如上：

保持打印的实例程序是较容易写的。

3.10 小结

三种不同的处理器证实了信息隐藏的优越性。动态连接帮助我们把一个问题分解成很简单的函数功能点。最终的程序是很容易扩展的——试着去增加 C 语言中的比较和如?: 的操作符吧。

第四章

继承：代码重用和精炼

4.1 A Superclass:*Point*

我们在这个章节里将开始一个基本绘画程序. 我们应该有一个像这样的测试类 (class) 的代码:

```
#include "Point.h"
#include "new.h"

int main(int argc, char ** argv)
{
    void *p;

    while (* ++ argv)
    {
        switch (** argv)
        {
            case 'p':
                p = new(Point, 1, 2);
                break;
            default:
                continue;
        }
        draw(p);
        move(p, 10, 20);
        draw(p);
        delete(p);
    }
    return 0;
}
```

对于每一个命令参数以字符 p 开始，我们获得一个新的绘图的点，移动这个点到某处，从新绘制，并且删除。标准化 C 语言不包含图形化输出标准的函数：然而，如果我们坚持产生一幅图片，我们能够发表文本，对于这个文本 Kernighan 的图片 [Ker82] 能够理解:

```
$ points p
"." at 1,2
"." at 11,22
```

坐标对于测试是无关紧要的——从商业和面向对象的说法解释：“点就是一则消息。”

我们用这个点能做些什么呢？new() 将产生一个点，并且构造器期望着初始化坐标作为进一步的参数传进 new() 。通常，delete() 将回收我们的点并且按照惯例调用析构器。

draw() 安排点被显示出来。由于我们希望与其他图形对象协同工作——因此在测试程序中会有 switch——对于 draw() 我们将提供动态连接。

move() 通过传递一系列参数来改变点的坐标。如果我们实现每一个图形对象，这些对象都与它涉及的点关联，我们将能够通过简单的应用这个点的 move() 方法来移动它。因此，对于 move() 在不需要动态连接的情况下我们应该可以做。

4.2 Superclass Implementation:*Point*

在抽象数据类接口 Point.h 中有:

```
1 extern const void * Point;          /* new(Point, x, y) */
2 void move(void * point, int dx, int dy);
```

我们重复利用第二章 new.? 文件, 我们移除 new 中一下方法并且增添 draw() 到 new.h 中:

```
1 void * new (const void * class, ...);
2 void delete(void * item);
3 void draw(const void * self);
```

在 new.r 中类型描述 struct Class 应该与在 new.h 中声明的方法相关联:

```
1 struct Class {
2     size_t size;
3     void * (* ctor) (void * self, va_list * app);
4     void * (* dtor) (void * self);
5     void (* draw) (const void * self);
6 };
```

选择器 draw() 在 new.c 中实现。它将代替如 differ() 在 2.3 节介绍的选择器，并且以相同的风格编写代码:

```
1 void draw (const void * self)
2 {
3     const struct Class * const * cp = self;
4     assert(self && * cp && (* cp) -> draw);
5     (* cp) -> draw(self);
6 }
```

这些预备工作完成后，我们将转去做真正的工作去写 Point.c，对点的实现。在此，面向对象帮助我们精确的鉴别出我们需要做什么：我们必须对表示式做出决定并实现构造器，析构器，动态链接方法 draw() 和静态链接方法 move() ，这些都是基本的函数。如果我们坚持二维，笛卡尔坐标，我们选择如下明确的表示:

```
1 struct Point {
2     const void * class;
```

```
3     int x, y;                          /* coordinates */
4 };
```

构造器必须初始化坐标.x 和.y ——现在一个绝对的例程如下:

```
1 static void * Point_ctor (void * _self, va_list * app)
2 {
3     struct Point * self = _self;
4     self -> x = va_arg(* app, int);
5     self -> y = va_arg(* app, int);
6     return self;
7 }
```

现在的结果是我们并不需要析构器, 因为在 delete() 之前没有资源呢需要回收. 在**Point_draw()** 函数中, 我们以一种图片能够识别的方式打印当前坐标:

```
1 static void Point_draw (const void * _self)
2 {
3     const struct Point * self = _self;
4     printf("\".\" at %d,%d\n", self -> x, self -> y);
5 }
```

这样照顾到所有的动态连接方法，并且我们能够定义类型描述符，在此一个空的指针代表一个不存在的析构器:

```
1 static const struct Class _Point = {
2     sizeof(struct Point), Point_ctor, 0, Point_draw
3 };
4 const void * Point = & _Point;
```

move() 不是动态连接的，因此我们省略 static 使得它作用域能够超出 Point.c 并且我们不给它加类名前缀 Point ：

```
1 void move (void * _self, int dx, int dy)
2 {
3     struct Point * self = _self;
4     self -> x += dx, self -> y += dy;
5 }
```

与在 new.c 中的动态连接相结合，这就得出了 Point.c 中点的实现。

4.3 继承:*Circle*

一个环形仅仅是一个大的点：此外对于中心坐标它需要一个半径。画法有点不同，但是移动只需要我们改变中心坐标。

这就是我们能够正常的为我们的文本编辑器和演择源代码重用而做好准备的地方。我们对点的实现做一个拷贝并且改变环与点不同的地方。Struct Circle 获取其他额外的组成:

```
1 int rad;
```

这部分组成在构造器中初始化

```
1 self->rad=va_arg(*app,int);
```

并且在**Circle_draw()** 中使用：

```
printf("circle at %d,%d rad %d\n",
    self —> x, self —> y, self —> rad);
```

我们在 move() 中有点迷惑。对于一个点和一个环必要的动作是相同的：对于坐标部分我们需要增加转移参数。然而，在一种情况，move() 工作于 struct Point ，在另外一种情况，它工作与 struct Circle 。如果 move() 是动态连接的，我们需要提供两个不同的函数去做相同的事情。但是，会有更好的方式，考虑一下点和环表示的层：

```
\* 此处缺少图片 *\
```

图片显示每一个环都以一个点开始。如果我们分配一个 struct Circle 通过增加到 struct Point 的结尾，我们可以向 move() 函数中传递一个环，因为表示式的初始化部分看起来仅仅像点，而 move() 方法期望接到收点，并且点仅仅是 move() 方法能够改变的。这里是一个合理的方式确保对环的初始化部分总看起来像点：

```
struct Circle { const struct Point _; int rad; };
```

我们让派生的结构体以一个我们要扩展的基结构体的拷贝而开始。信息隐藏要求我们决不直接的访问基结构体；因此，我们使用几乎不可见的下划线作为它的名字并且把它声明为 const 避开粗心的指派。

这就是简单的继承的全部：一个子类从一个超类（或者基类）继承仅仅通过扩充表示超类的结构体。

由于子类对象（一个环）的表示就像一个超类对象（一个点）的表示一样动身。环总能够伴装成一个点——在一个环的表示的初始化地址处的确是一个点的表示。

向 move() 中传递一个环是完全确定的：子类继承了超类的方法，因为这些方法仅在子类的表示上操作，这些子类的表示和超类的表示是相同的，而这些方法原先就在超类上写好了。传递一个环就像传递一个点意味着把 struct Circle* 转换成 struct Point* 。我们将把这样的操作看成一个从子类到超类的上抛——在标准化 C 语言中，它能够使用明确的转换操作符来实现或者通过中间的 void* 的值。

这通常是不佳的，然而，传递一个点到一个函数专为环如，**Circle_draw()**：如果一个点原先就是一个环，从 struct Point* 转换成 struct Circle* 仅仅是可允许的。我们称这样的从超类到子类的转换为下抛——这也要求明确的转换或 void* 值，并且它仅仅对于指针，对于对象能够使用，指针，对象在子类的开始做转换。

4.4 Linkage and Inheritance

move() 不是动态连接的并且不使用动态连接方法做工作。然而我们能够传递一个指针和环到 move() 中，它的确不是一个多肽的函数：move() 对于不同的对象不会做不同的处理，它总是增加参数到坐标，忽略其他与坐标相依附的。

对于动态连接方法如 draw() ，这种情形是不同的。让我们再次看先前的图片，这次完全明确类型描述符如下：

```
\* 此处缺少图片 *\
```

当我们上抛从一个环到一个点时，我们没有改变环的状态，换句话说，即使我们把环的 struct Circle 表示当成一个点的 struct Point ，我们不会改变它的内容。结果，把环视为点作为一个类型描述符仍然拥有 Circle，因为点在它的.class 部分并没有改变。draw() 是一个选择器函数，即，它将会使用无论传入什么样的参数作为自身，去处理被.class 所指示的类型描述符，并且调用在这里存储的画图方法。

一个子类继承它的超类的静态链接的方法——这些方法操作子类对象的部分，这些子类对象是已经在超类对象上呈现的。一个子类能够选择支持它自己的方法代替它的超类的动态连接方法。如果继承，即，若没有重写，超类动态的连接的方法就像静态连接的方法一样的起作用并且修改子类对象的超类的部分内容。如果重写，子类他自己的动态连接方法的版本访问子类对象所有的表示，即，对于一个环，draw() 将会调用**`Circle_draw()`** 方法，此方法能够考虑到半径当画环的时候。

4.5 Static and Dynamic Linkage

一个子类继承了它的超类的静态连接的方法并且选择性的继承或重写动态连接的方法. 考虑对于 move() 和 draw() 的声明如下:

```
1 void move(void* point, int dx,int dy);
2 void draw(const void* self);
```

我们不能够从这两个声明中发现连接，尽管对于 move() 的实现能够直接的工作，然而 draw() 仅仅是一个选择器函数在运行时跟踪动态连接。不同点就是我们声明一个静态链接方法就像 move() 在 Point.h 中作为抽象数据类型接口的一部分，且我们声明一个动态连接方法就像 draw() 携带内存管理接口在 new.h 中，因为迄今为止我们已经决定在 new.c 中实现数据选择器。

静态链接会更加有效率因为 C 编译器能够使用直接的地址调用子程序，但是对于一个函数如 move() 对于子类不能被重写。动态连接在间接调用的扩展上更加便捷——我们已经对调用选择器函数如 draw() 的额外开销作了决定，检查参数，定位，调用正确的方法。我们丢弃了检查并且使用 macro* 像如下减少了额外开销:

```
1 #define draw(self)  ((*(struct Class**)self)->draw(self));
```

但是如果他们的参数有负面的影响宏会引发问题并且对于宏并没有明确的技术用于操作可变参数列表。此外，宏需要 struct Class 的声明，此 struct Class 到目前为止对于类的实现已经可用而不是对于整个程序。

不幸的是当我们设计超类时，我们还需要决定很多事情。但是函数调用方法是不会改变的，它会占用很多文本编辑，更可能的会在许多类中，把一个函数的定义从静态转换到动态连接，反之亦然。从第七章开始我们将使用一个简单的预处理去简化编码，即使如此连接转换也是极易出错的。

带着这种怀疑，与静态链接相比决定动态连接可能会更好点即使它效率较低。通用函数能提供一个有用的概念性的抽象并且他们倾向于减少我们需要在项目过程中记忆的函数名的数量。如果，实现所有要求的类后，我们发现其实动态连接方法从来没有被

重写，通过其单一的实现去替代它的选择器并且甚至在 struct Class 中浪费它的位置与扩展类型描述和更正所有的初始化相比麻烦会更少。

4.6 Visibility and Access Function

我们现在可以尝试着实现**Circle_draw()**。基于“need to know ”这样的规则信息隐藏要求我们对于每个类使用 3 个文件。Circle.h 包含抽象数据类型接口；对于一个子类它包含了超类的接口文件以便于这样的声明使得继承的方法可用：

```
1 #include "Point.h"
2 extern const void* Circle;  /*new(Circle,x,y,rad)*/
```

接口文件 Circle.h 被应用程序代码所包含并且对于类的实现；它避免了多次包含所引发的错误。

一个环的表示在第二个头文件中声明，Circle.r 。对于子类它包含了超类的表示文件以便于我们能够通过扩展超类派生出子类的表示：

```
1 #include "Point.r"
2 struct Circle{const struct Point _;int rad;};
```

子类需要超类的表示去实现继承：struct Circle 包含了一个 const struct Point。这个点确定不是只读的——move() 将改变它的坐标——但是 const 限定词防止了意外的覆盖它的组成部分。表示文件 Circle.r 仅仅被类的实现所包含；仍然受到多重调用的保护。

最终，对一个环的实现对于类，对于对象管理，被在包含接口和表示文件的原文件 Circle.c 中所定义：

```
1 #include "Circle.h"
2 #include "Circle.r"
3 #include "new.h"
4 #include "new.r"
5
6 static void Circle_draw(const void * _self)
7 {
8     const struct Circle* self=_self;
9     printf("circle at %d rad %d\n",self->_.x,self->_.y,self->rad)
          ;
10 }
```

在**Circle_draw()** 中，对于环我们通过子类部分使用“可见的名字”_. 来读取点部分。从信息隐藏的角度看这并不是一个好的注意。然而读取坐标值不应该产生重大的问题，我们决不能确保在其他情形下，一个子类的实现不去直接的欺骗和修改它的父类的一部分，因此带着其不变量去玩一场浩劫。

效率要求一个子类能直接的访问到其超类的组成部分。信息隐藏和可维护性原则要求一个超类从它的子类上尽可能好的隐藏对它自己的表示。如果我们后面做出选择，我们应该能够提供对这些子类被允许查看超类所有组成部分访问函数，并且对于这些组成部分提供更正函数，即，便要子类去做修改。

访问和修改函数时静态链接的方法。如果我们对于超类在表示文件中声明了他们，超类仅包含在子类的实现中，我们可以使用宏，如果宏使用每个参数仅以此则副作用没有问题。作为一个例子，在 Point.r 中，我们定义了下面的访问宏 macros:*

```
#define x(p)    (((const struct Point*)(p))->x)
#define y(p)    (((const struct Point*)(p))->y)
```

这些宏对于任何以 struct Point 开始对象能够被应用于一个指针，也就是说，对于对象，从我们的点的任何子类。这项技术即为，上抛我们的点到超类并引用我们感兴趣的部分。const 在抛得过程中对结果的分配。如果 const 被忽略

```
#define x(p)    (((struct Point*)(p))->x)
```

一个宏调用 x(p) 产生一个能成为分配的目标的 l-value，一个好点的修改函数最好是一个宏的定义

```
#define set_x(p,v) (((struct Point*)(p))->x=(v))
```

此定义产生一个分配。

在子类实现的外部对于访问和修改函数我们仅仅使用静态链接的方法。我们不能够求助于宏，因为对于宏引用超类的内部表示是不可见的。对于包含进应用程序的信息隐藏并不提供表示文件 Point.r 而实现。

宏定义揭示了，然而，一旦一个类的表示可用，信息隐藏能够被很容易的击败。这里有一个方式更好的隐藏 struct Point 。在超类的实现中，我们使用正常的定义:

```
struct Point{
    const void* class;
    int x,y;
};
```

对于子类的实现我们提供下面的看起来不透明的版本:

```
struct Point{
    const char _[sizeof(struct {const void* class; int x,y;})];
};
```

这个结构体像先前拥有相同的大小，但是我们不能够读取也不能够写它的组成部分因为他们被隐藏在一个匿名的内部结构中。重点是这两种声明必须包含相同的组成部分的声明并且这在没有与处理器的情况下是很难维持的。

4.7 Subclass Implementation:*Circle*

子类的实现——环

我们已经做好了些完整实现的准备，我们可以选择先前部分介绍的我们最喜欢的技术。面向对象规定我们需要一个构造器，可能的话还会有一个析构器，`Circle_draw()`，和类型描述 Circle 都绑定在一起。以便于练习我们的方法，我们包含了 Circle.h 并增加了下面的行在 4.1 部分的程序中做测试:

```
case 'c':
    p=new(Circle,1,2,3);
    break;
```

现在我们能够观察到下面的测试程序的表现：

```
$ circles p c
"." at 1,2
"." at 11,12
circle at 1,2 rad 3
circle at 11,22 rad 3
```

环的构造函数接收 3 个参数：第一个参数为环的点的坐标接下来是半径。初始化点部分是点的构造器的工作。它会处理部分 new() 参数列表的参数。环的构造器从它的初始化半径的地方携带保留的参数列表。

一个子类的构造器首先应该允许超类做部分初始化，这部分初始化把清晰地内存带进超类对象。一旦超类构造器构造完成，子类构造器完成初始化并把超类对象带进子类对象中。

对于环，意味着我们需要调用**Point_ctor()** 。像其他所有动态链接一样，这个函数被声明为 static ，因此隐藏在 Point.c 的内部。然而，我们仍然能够通过在 Circle.c 中可用的类型描述符来 Point 获得此函数。

```
static void * Circle_ctor (void * _self, va_list * app)
{
    struct Circle * self = ((const struct Class *) Point) —>
        ctor(_self, app);
    self —> rad = va_arg(* app, int);
    return self;
}
```

这里应该很清楚为什么我们传递参数的地址 app 列表指针到每个构造器而不是 **va_list** 的值本身：**new()**调用子类的构造器，此构造器调用超类的构造器，等等。最超级的构造器是第一个将去实际的作一些事情，并且会捡起传进**new()** 的最左边的参数列表。保留的参数对于下一个子类是可用的，等等知道最后，最右边的参数被最终的子类所使用，也就是说，被**new()**所直接的调用的构造器所调用。

构造器以严格的相反的次序是最好的组织：delete() 调用子类的析构器。它首先应该销毁它自己的资源接下来调用直接的超类的析构器，这个析构器可直接的销毁下一个资源集等等。构造是先发生在子类之前的父类上的。析构则是相反，子类要先于父类，即，环部分要先于点部分。这里，然而，什么也不需要做。

我们先前已经让**Circle_draw()** 工作了，我们使用可见部分，并且编码表示文件 Point.r 如下：

```
struct Point {
    const void * class;
    int x, y; /* coordinates */
};
#define x(p) (((const struct Point *)(p)) -> x)
#define y(p) (((const struct Point *)(p)) -> y)
```

现在我们可以对于**Circle_draw()** 使用访问宏：

```
static void Circle_draw (const void * _self)
{
    const struct Circle * self = _self;
```

```
4       printf("circle at %d,%d rad %d\n",x(self), y(self), self —>
            rad);
5   }
```

move() 拥有静态链接并且被从点的实现上继承。我们得出结论环的实现是通过定义仅仅全局可见 Circle.c 的部分内容:

```
1   static const struct Class _Circle = {
2       sizeof(struct Circle), Circle_ctor, 0, Circle_draw
3   };
4   const void * Circle = & _Circle;
```

然而，在接口，表示式，实现文件之间似乎我们有一个可行的分配程序文本实现类的策略，点和环的例子还没有显现出一个问题：如果一个动态连接的方法如**Point_draw()**在子类中没有被重写，子类的类型描述符需要指向在父类实现的函数。函数名，然而在这里被定义成 static，因此选择器是不能够被规避的。我们将在第六章看到一个清晰地解决此问题的方法。作为暂时的权衡，我们在这种情况下可以避免对 static 的使用，仅仅在子类的实现文件中声明函数的头，对于子类并且使用函数名去初始化类型描述。

4.8 小结

超类的对象和子类是相似的，但是在表现形式上并不相同。子类正常情况下会有更详尽的陈述更多的方法——他们被超类对象的版本专用指定。

我们使用超类对象的表示的拷贝来作为子类对象表示的开始，即，子类对象通过把它的组成部分增加到超类对象的末尾被表示。

一个子类继承了超类的方法：因为一个子类对象的起始部分看起来像超类对象，我们可以上抛并且看到一个指向子类对象的指针作为一个指向我们能够传递超类方法的超类对象。为了避免显性转换，我们使用 void* 作为通用指针来声明所有方法的参数。

继承可以被看成一个多态机制的根本形式：一个超类方法接受不同类型，它自己的类和所有子类命名的对象。然而因为对象都佯装成超类对象，方法仅仅在每个对象的超类部分起作用，并且它将，因此从不同的类对于对象不会起不同的作用。

动态链接方法能够从一个超类继承或在子类中重写——对于子类通过无论何种函数的指针被放进类型描述符来决定。因此，对于一个对象如果动态链接方法被调用，我们总能够访问属于对象真正的类的方法即使指针上抛到一些超类上。如果动态链接方法被继承，它只能在子类对象的超类部分起作用，因为它的确不知道子类的存在。如果一个方法被重写，子类的版本能够访问整个对象，他甚至可以通过显性的超类的类型描述符的使用来调用它关联的超类的所有方法。

特别注意，对于超类的表示，构造器首先回调超类的构造器直到最终的祖先以便于每个子类的构造器仅仅处理它自己的对类的扩展。每个超类析构器应该先删除它的子类的资源然后调用超类的析构器等等直到最终的祖先。构造器的调用顺序是从祖先到最终的子类，析构器的发生则正好是相反的顺序。

我们的策略还是有点小毛病的：在通常情况下我们不应该从一个构造器中调用动态链接方法，因为对象也许并没有完全被初始化好。在构造器被调用之前 new() 把最终

的类型描述符插入到一个对象中，作为一个构造器在相同的类中是没有必要的访问方法的。安全的技术是在相同的类中对于构造器通过内部的名字来调用方法，也就是说，对于点，我们调用`Points_draw()`而不是 draw() 。

为了鼓励信息隐藏，我们使用了三个文件对类的实现。接口文件包含了抽象的数据类型描述，表示文件包含了对象的结构，实现文件包含了方法和初始化类型描述的代码。一个接口文件包含了超类接口文件并且被实现和任何应用所包含。一个表示文件包含了超类的表示文件并且仅仅被实现所包含。

超类的部分不应该直接的在子类中被引用。相反，对于每个部分我们能够既提供静态链接访问和尽可能的修改方法，也能对于超类的表示文件增加适当的宏。函数符号使得使用文本编辑器或调试器去跟踪可能的信息泄露或不变量的破坏更简单。

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

是或有吗?——继承对集合

作为 struct Circle 我们对环的表示包含了对点的表示：

```
1 struct Circle { const struct Point _; int rad; };
```

但是，我们自然绝冬不去直接的访问者部分。相反，当我们想要继承我们从 Circle 上抛到 Point 并且在这里处理 struct Point 的初始化。

这里有另外一个表示环的方式：它能包含一个点作为一个集合。我们能够仅仅通过指针来处理对象；因此这样的一个环的表示看起来就像如下所示：

```
1 struct Circle2 { struct Point * point; int rad; };
```

这个环一点也不像一个点，也就是说，它不能够从 Point 所继承并且重用它的方法。然而，它能够把点的方法应用到点的部分；它仅仅不能把点的方法用于它自己。

如果一种语言对于继承有明确的符号，差异就会更加明显，相似的表示在 C++ 中会有如下的表示：

```
1 struct Circle:Point{int rad;};  //inheritance
2 struct Circle2{ struct Point point;int rad}; //aggregate
```

在 C++ 中作为一个指针我们是不必要访问对象的。

继承，即，从超类来建立子类，而集合，即，把对象的一部分作为另外一个对象的一部分，提供非常相似的功能。这些应用在特殊的设计中通常被所 is-it-or-has-it? 的测试所决定：如果一个新类的一个对象仅仅像一些其他类的对象，我们应该使用继承来实现新的类；如果一个新类有一个其他类作为它的状态的一部分对象，我们应该建立集合。

到我们的点所关注的，一个环仅仅是一个大的点，这就是为什么我们使用继承来做一个环的原因。一个方形是一个不明确的例子：我们能通过一个参考点和边的长度来描述它，或我们能够使用端点的对角线或甚至三个角来描述。仅仅带参考点是方形的几分花哨点；其他表示通向集合。 在我们的算术表达式中，我们已经使用了继承从单目到双目操作节点，但是这已经充分的违背了测试。

4.10 Multiple Inheritance

多继承, 因为我们使用平凡的标准化 C 语言。我们不能够隐藏这样的事实——继承意味着在另一个结构的开始包含一个结构体。利用上抛是在子类的对象上重复利用超类方法的关键所在。通过投掷一个结构体起始的地址完成一个从环岛到点的上抛；指针的值并没有改变。

如果我们在其他结构中包含两个及以上的结构体，并且如果我们愿意在上抛期间做一些地址的处理，我们可以称这样的结果为多重继承：一个对象能够像它属于几个类一样了表现。优点似乎是我们不必很仔细的设计继承的关系——我们可以很快的把类仍到一起并且继承我们希望继承的任何东西。缺点是，显然，在我们能够重用方法之前我们得有地址处理机制。

事情能够实际的很快让我们感到迷惑。思考一个文本，一个方形，每一个都有一个继承的引用点。我们能够把他们一起扔到一个按钮上——仅仅存在的问题希望这个按钮应该继承一个或两个引用点。

我们使用标准化 C 语言拥有很大的优点：它会使这样的事实很明显，即，继承——多重或其他总是伴随着包含而进行。包含，然而也能作为集合被实现。与复杂化语言定义和增加过量实现相比多重继承对于程序员来说要做的更多，这一点也不清晰。我们将使得事情变得简单兵器只做简单的继承。第 14 章将首要展示多重继承的使用，库的合入能够被集合和消息转换所实现。

4.11 Exercises

Graphics programming offers a lot of opportunities for inheritance: a point and a side length defines a square; a point and a pair of offsets defines a rectangle, a line segment, or an ellipse; a point and an array of offset pairs defines a polygon or even a spline. Before we proceed to all of these classes, we can make smarter points by adding a text, together with a relative position, or by introducing color or other viewing attributes.

Giving move() dynamic linkage is difficult but perhaps interesting: locked objects could decide to keep their point of reference fixed and move only their text portion.

Inheritance can be found in many more areas: sets, bags, and other collections such as lists, stacks, queues, etc. are a family of related data types; strings, atoms, and variables with a name and a value are another family.

Superclasses can be used to package algorithms. If we assume the existence of dynamically linked methods to compare and swap elements of a collection of objects based on some positive index, we can implement a superclass containing a sorting algorithm. Subclasses need to implement comparison and swapping of their objects in some array, but they inherit the ability to be sorted.

第五章

编程常识：符号表

正确的扩展结构体, 并且, 分享基础结构体的函数, 能帮助我们避免用 union 的麻烦. 尤其是在结合动态连接, 我们获得一个制度和完美稳健处理发散信息的方法. 一旦基础的机制确定好位置, 那么新的扩展结构体能很容易的添加到基础机制中并且重复使用.

作为一个示例, 我们将添加关键字, 常量, 变量, 数学表达式到第三章的小计算器中. 所有添加的项目都生存字符号表中并且分享同样的基础名字的搜索机制.

5.1 标识符扫描

在 3.2 章节中, 我们实现了**scan()**函数, 其能从主进程接收每一行输入并且在每一次搜寻中维持一个输入符号. 如果我们想要介绍关键字, 常量等等, 我们需要扩展下**scan()**. 就像浮点数一样, 我们提取字母字符串最为进一步的分析:

```
#define ALNUM   "ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
                "abcdefghijklmnopqrstuvwxyz" \
                "_" "0123456789"
static enum tokens scan (const char * buf)
{
    static const char * bp;
    if (isdigit(* bp || * bp == '.'))
        ...
    else if (isalpha(* bp) * bp == '_')
    {
        char buf [BUFSIZ];
        int len = strspn(bp, ALNUM);

        if (len >= BUFSIZ)
            error("name too long: %—.10s...", bp);
        strncpy(buf, bp, len), buf[len] = '\0', bp += len;
        token = screen(buf);
    }
    ...
}
```

一旦我们有一个标识符, 我们让**screen()**一个新的函数来决定**token**的取值. 如果必要的,**screen()**函数会存储一个描述符到解析器可以检查的全局变量描述符中.

5.2 使用变量

一个变量参与两个操作: 它的值被用作于操作数在一个表达式中, 或者一个表达式的值被分配给它. 第一个操作是一个简单**factor()**函数的延伸, 其在 3.5 章节中作为识别器的一部分.

```
static void * factor (void)
{
    void * result:
    ...
    switch (token) {
    case VAR:
        result = symbol;
        break;
    ...
    }
    ...
}
```

VAR 是一个特殊的值, 当一个合适的描述符被发现时**screen()**替代 token 的值. 额外的关于描述符的信息是被安排于全局变量符号中. 在这个例子中, 符号包含一个代表作为表达式树的叶子变量的节点.**screen()**函数能找到任何一个在描述符表中的变量或者用描述符 Var 去创建一个.

识别某一个分配是一个 bit 变得更加复杂. 我们的计算器是一个非常实用的, 我们允许两种语法声明:

```
asgn : sum
    | VAR = asgn
```

不幸的是,VAR 也能出现也 sum 表达式的左边,i.e., 它不能立即清楚的如何认出 c-style 嵌入式分配用我们的递归下降技术. 这是因为我们想要学习如何去处理关键字不论用任何方法, 我们设定遵循这样的语法规则:

```
stmt : sum
    | LET  VAR = sum
```

下面是遵循这一规则的函数实现:

```
static void * stmt (void)
{
    void * result;
    switch (token)
    {
    case LET:
        if (scan(0) != VAR)
            error("bad assignment");
        result = symbol;
        if (scan(0) != '=')
            error("expecting =");
        scan(0);
        return new(Assign, result, sum());
    default:
        return sum();
    }
```

```
17 }
```

在中程序中, 我们用**stmt()**替代**sum()**并且我们的识别器准备捕捉变量. **Assign**是一个新的数据类型描述符, 其作为一个计算值的节点和分配值到一个变量.

5.3 The Screener —*Name*

分配器遵循这样的语法:

```
1 stmt : sum
2     | LET VAR = sum
```

LET 是一个关键字的例子. 在 building 中 the screener 我们仍然能决定什么标识符将是当时的代表 LET:**scan()**提取出一个标识符从输入行里, 并把它压进**screen()**中,**screen()**看起来像在符号表中其返回值作为 token, 至少是一个变量, 一个符号节点.

识别器丢弃 LET 但是它安装变量作为一个叶子节点到树上. 对于其他符号, 比如算术表达式的名字, 我们将希望应用**new()**不论 screener 返回什么样的符号而得到一个树上的新节点. 因此, 有很多部分组成的符号表条目将会有一样动态连接树节点的函数.

作为关键字, 一个**Name**需要包含输入字符串和 token 值. 此后, 我们想要从 **Name**继承; 因此, 我们定义一个这样的结构体在 Name.r 文件中:

```
1 struct Name {                 /* base structure */
2     const void * type;        /* for dynamic linkage */
3     const char * name;        /* may be malloc-ed */
4     int token;
5 };
```

我们的符号永不消逝: 如果他们的名字包含作为预定义的关键字的字符串, 那也没有关系, 或者作为用户定义的变量的动态存储字符串–我们不会回收他们.

在我们能找到一个符号之前, 我们需要把它输入符号表. 这个不能通过调用 **new(Name,...)**来处理, 因为我们希望支持比 Name 更加复杂化的符号, 我们应该隐藏符号表实现. 替代的，我们提供了一个函数 install()，它接受一个 name 对象并将其插入符号表中。下面是符号表接口文件 Name.h:

```
1 extern void * symbol;   /* -> last Name found by screen() */
2 void install (const void * symbol);
3 int screen (const char * name);
```

识别器必须将关键字如 let 插入符号表中，在筛选器寻找它们之前。这些关键字被定义在结构体的常量表中–它们没有差别用于**install()**。以下函数用于初始化识别:

```
1  #include "Name.h"
2  #include "Name.r"
3
4  static void initNames (void)
5  {
6      static const struct Name names [] = {
7          { 0, "let", LET },
8          { 0 }};
9      const struct Name * np;
10     for (np = names; np -> name; ++np)
```

```
11          install(np);
12  }
```

注意**names[]**，关键字表，不需要排序。要定义**names[]**，我们使用名称的表示形式，即，我们包括 Name.r。因为关键字 let 被丢弃，所以我们没有提供动态链接的方法。

5.4 父类的实现：*Name*

按名称搜索符号是一个基本问题。不幸的是，ANSI 标准没有定义一个合适的库函数来解决这个问题。**bsearch()**–排序表中的二进制搜索–接近了，但是如果我们插入一个新的符号，我们就必须调用**qsord()**来为进一步的搜索做好准备。

Unix 系统可能提供两个或三个功能族来处理不断增长的表。**lsearch()**–对数组进行线性搜索并在末尾（！）添加–并不完全有效。**hsearch()**–一个包含文本和信息指针的结构的哈希表–只维护一个固定大小的表，并且在 entrires 上强加了一个笨拙的结构。**tsearch()**–一种具有任意比较和删除的二叉树–是最普遍的家族，但是如果初始符号是从排序序列中安装的。

在 UNIX 系统上，**tsearch()**可能是最好的折衷方案。带有二进制线程树的可移植实现的源代码可以在 [Sch87] 中找到。然而，如果这个家族不可用，或者如果我们不能保证随机初始化，我们应该寻找一个更简单的工具来实现。事实证明，可以很容易地扩展 bsearch() 的仔细实现，以支持插入到排序数组中：

```
1  void * binary (const void * key,
2      void * _base, size_t * nelp, size_t width,
3      int (* cmp) (const void * key, const void * elt))
4  {   size_t nel = * nelp;
5  #define base (* (char **) & _base)
6      char * lim = base + nel * width, * high;
7
8      if (nel > 0)
9      {   for (high = lim — width; base <= high; nel >>= 1)
10         {   char * mid = base + (nel >> 1) * width;
11             int c = cmp(key, mid);
12
13             if (c < 0)
14                 high = mid — width;
15             else if (c > 0)
16                 base = mid + width, —— nel;
17             else
18                 return (void *) mid;
19         }
```

到目前为止，这是在任意数组中的标准二进制搜索。key 要查找对象的关键点；base 基础是 *nelp 元素表的起始地址，每一个都有 width 宽度字节；cmp 是一个比较函数，用于将键与表元素进行比较。此时，我们已经找到了一个表元素并返回了它的地址，或者基现在是键应该在表中的地址。我们继续这样做：

```
1          memmove(base + width, base, lim — base);
2      }
3      ++ *nelp;
```

```
4       return memcpy(base, key, width);
5   #undef base
6   }
```

memmove()将数组的末尾移开 *，**memcpy()**插入键。我们假设在数组之外还有空间，并通过 nelp 记录，我们添加了一个元素-**binary()**与标准函数**bsearch()**不同，只需要地址，而不是包含表中元素数的变量的值。

考虑到搜索和录入的一般方法，我们可以很容易地管理我们的符号表。首先，我们需要将一个关键字与一个表元素进行比较：

```
1  static int cmp (const void * _key, const void * _elt)
2  {   const char * const * key = _key;
3      const struct Name * const * elt = _elt;
4      return strcmp(* key, (* elt) —> name);
5  }
```

作为键，我们只传递指向输入符号文本的指针的地址。当然，Name 结构体，我们只查看它们的.name 组件。

搜索或输入是通过调用具有适当参数的**binary()**来完成的。由于我们事先不知道符号的数量，所以我们确保表始终有展开的空间：

```
1  static struct Name ** search (const char ** name)
2  {   static const struct Name ** names; /* dynamic table */
3      static size_t used, max;
4
5      if (used >= max)
6      {   names = names
7              ? realloc(names, (max *= 2) * sizeof * names)
8              : malloc((max = NAMES) * sizeof * names);
9          assert(names);
10     }
11     return binary(name, names, & used, sizeof * names, cmp);
12 }
```

NAMES 是一个定义的常量，表条目的初始分配是表项的初始分配；每次我们用完时，我们都会将表的大小翻一番。

search()以一个指向文本的地址被发现并返回该表项的地址。如果文本不能在表中找到，**binary()**已在表中插入键，即仅指向文本的指针，而不是一个结构名。这一战略是为**screen()** 效益，只有建立一个新表元素如果标识符从输入是未知的：

```
1  int screen (const char * name)
2  {   struct Name ** pp = search(& name);
3
4      if (* pp == (void *) name) /* entered name */
5      * pp = new(Var, name);
6      symbol = * pp;
7      return (* pp) —> token;
8  }
```

screen()让**screen()**查找要筛选的输入符号。如果指向符号文本的指针被输入到符号表中，我们需要用描述新标识符的条目替换它。

对于**screen()**，新的标识符必须是变量。我们假设有一个类型描述 Var，它知道如

何构造描述变量的名称结构，然后让**new()**来完成其余的工作。在任何情况下，我们让符号指向符号表条目，然后返回它的.token 值。

```
void install (const void * np)
{   const char * name = ((struct Name *) np) —> name;
    struct Name ** pp = search(& name);

    if (* pp != (void *) name)
        error("cannot install name twice: %s", name);
    * pp = (struct Name *) np;
}
```

install()稍微简单一些。我们接受一个 name 对象，让**search()**在符号表中找到它。**install()**应该只处理新的符号，所以我们应该总是能够输入对象而不是它的名称。否则，如果**search()**真的找到了一个符号，我们就有麻烦了。

5.5 子类的实现：*Var*

screen() 调用 **new()** 来创建一个新的变量符号表, 并将其返回给识别器, 该识别器将其出入表达树中. 因此,VAR 必须创建可以向节点那样工作的符号表条目, 也就是说, 在定义
ccodestruct Var 时, 我们需要扩展一个 struct 名称, 已集成驻留在符号表中的能力, 并且必须支持适用于表达式接待的动态连接函数. 我们在 Var.h 中描述了接口:

```
const void * var;
const void * Assign;
```

变量具有名称和值。如果我们评估算术表达式，我们需要返回.Value 组件。如果删除表达式，则不能删除变量节点，因为它位于符号表中：

```
struct Var {    struct Name _; double value; };
#define value(tree) (((struct Var *) tree) -> value)
static double doVar (const void * tree)
{
    return value(tree);
}
static void freeVar (void * tree){}
```

正如在 4.6 节中所讨论的那样，通过为值提供访问函数来简化代码。

创建变量需要分配一个**struct Var**，插入变量名的动态副本，以及识别器规定的令牌值 VAR：

```
static void * mkVar (va_list ap)
{   struct Var * node = calloc(1, sizeof(struct Var));
    const char * name = va_arg(ap, const char *);
    size_t len = strlen(name);

    assert(node);
    node -> _.name = malloc(len + 1);
    assert(node -> _.name);
    strcpy((void *) node -> _.name, name);
    node -> _.token = VAR;
```

```
11      return node;
12  }
13  static struct Type _Var = { mkVar, doVar, freeVar };
14  const void * Var = &_Var;
```

new()负责在符号返回到**screen()**或任何想使用它的人之前将类型描述 Var 插入到节点中。

从技术上讲，**mkVar()**是 Name 的构造函数。但是，只需要动态地存储变量名称。因为我们决定在计算器中构造函数负责分配对象，所以我们不能让 Var 构造函数调用名称构造函数来维护.name 和.Token 组件–Name 构造函数将分配一个**struct Name**，而不是**struct Var**。

5.6 赋值

赋值是二进制操作。识别器保证有一个变量作为左操作数，和作为右操作数。因此，我们真正需要实现的是实际的赋值操作，即动态链接到类型描述的.exec 组件中的函数：

```
1  #include "value.h"
2  #include "value.r"
3
4  static double doAssign (const void * tree)
5  {
6      return value(left(tree)) = exec(right(tree));
7  }
8
9  static struct Type _Assign = { mkBin, doAssign, freeBin };
10
11 const void * Assign = & _Assign;
```

我们共享 Bin 的构造函数和析构函数，因此在实现算术操作时必须使其成为全局的。我们还共享**struct Bin**和访问函数**Left()**和**right()**。所有这些都是通过接口文件 value.h 和表示文件 value.r 导出的。我们自己的**struct Var** 的访问函数**value()**故意允许修改，这样赋值就可以很好地实现。

5.7 另一个子类：*Constants*

谁喜欢键入 π 或其他数学常量的值? 我们从 Kernighan 和 Pike 的 hoc[KP 84] 中获得线索，并为我们的计算器预定义了一些常数。在初始化识别器时需要调用以下函数：

```
1  void initConst (void)
2  {
3      static const struct Var constants [] = { /* like hoc */
4          { &_Var, "PI", CONST, 3.14159265358979323846 },
5          ...
6          0 };
7      const struct Var * vp;
8      for (vp = constants; vp —> _.name; ++ vp)
9          install(vp);
10 }
```

变量和常量几乎相同：它们都有名称和值，并且都存在于符号表中；都返回它们的值以便在算术表达式中使用；在删除算术表达式时不应该删除它们。但是，我们不应该将其赋值给常量，因此我们需要商定一个新的令牌值 Const，识别程序在 factor() 中接受它，就像 VAR 一样，但是在 stmt() 中赋值的左边不允许这样做。

5.8 数学函数：*Math*

ANSI-C 定义了许多数学函数，如 sin()、sqrt()、exp() 等。作为继承的另一个练习，我们将向计算器中添加具有单个双参数和双重结果的库函数。

这些函数就像一元运算符一样工作。我们可以为每个函数定义一种新的节点类型，并从**Minus**和**Name**类中收集大部分功能，但是有一种更简单的方法。我们将**struct Name**扩展到**struct Math**，如下所示：

```
1 struct Math { struct Name _;
2     double (* funct) (double);
3 };
4
5 #define funct(tree) (((struct Math *) left(tree)) -> funct)
```

除了输入中要使用的函数名和识别标记之外，我们还将像 sin() 这样的库函数的地址存储在符号表条目中。

在初始化过程中，我们调用以下函数将所有函数描述输入符号表：

```
1 #include <math.h>
2 void initMath (void)
3 {
4     static const struct Math functions [] = {
5         { &_Math, "sqrt", MATH, sqrt },
6         ...
7         0 };
8
9     const struct Math * mp;
10
11    for (mp = functions; mp -> _.name; ++ mp)
12        install(mp);
13 }
```

函数调用是一个因素，就像使用**minus**一样。为了获得认可，我们需要扩展语法以了解各种因素：

```
1 factor : NUMBER
2     | - factor
3     | ...
4     | MATH ( sum )
```

MATH是**initMath()**输入的所有函数的通用标记。这将转换为识别器中的**factor()**添加的以下内容：

```
1 static void * factor (void)
2 {
3     void * result;
```

```
    ...
    switch (token) {
    case MATH:
    {
        const struct Name * fp = symbol;

        if (scan(0) != '(')
            error("expecting (");
        scan(0);
        result = new(Math, fp, sum());
        if (token != ')')
            error("expecting )");
        break;
    }
```

symbol首先包含 sin() 等函数的符号表元素。我们通过调用**sum()** 来保存指针并为函数参数构建表达式树。然后，我们使用**Math**，函数的类型描述，并让**new()**为表达式树构建以下节点：

```
\* 缺失图片 *\
```

我们让二进制节点的左边指向函数的符号表元素，并在右边附加参数树。二进制节点以**Math**作为类型描述，即将分别调用**doMath()**和**FreeMath()**方法来执行和删除该节点。

Math节点仍然是用**mkBin()**构造的，因为这个函数不关心它作为后代输入了哪些指针。然而，**FreeMath()**只能删除正确的子树：

```
static void freeMath (void * tree)
{
    delete(right(tree));
    free(tree);
}
```

如果仔细查看图片，我们可以看到**Math**节点的执行是非常容易的。**doMath()**需要调用存储在符号表元素中的任何函数，该元素可作为二进制节点的左后代访问，该二进制节点的左端名为：

```
#include <errno.h>
static double doMath (const void * tree)
{
    double result = exec(right(tree));
    errno = 0;
    result = funct(tree)(result);
    if (errno)
        error("error in %s: %s", ((struct Math *) left(tree)) —>
            _.name,
            strerror(errno));
    return result;
}
```

唯一的问题是通过监视 ANSI-C 头文件 errno.h 中声明的 errno 变量来捕获数值错误。这就完成了计算器数学函数的实现。

5.9 小结

基于用于搜索和插入排序数组的函数**binary()**，我们实现了一个符号表，该表包含具有名称和标记值的结构。继承允许我们将其他结构插入到表中，而无需更改搜索和插入功能。一旦我们考虑到符号表元素的传统定义，这种方法的优雅就显而易见了：

```
1 struct {
2     const char * name;
3     int token;
4     union { /* based on token */
5         double value;
6         double (* funct) (double);
7     } u;
8 };
```

对于关键字，**union**是不必要的。用户自定义的函数需要更详细的描述，而引用**union**的部分是麻烦的。

继承允许我们将符号表功能应用于新条目，而根本不更改现有代码。动态链接在许多方面有助于保持实现的简单性：可以将常量、变量和函数的符号表元素链接到表达式树中，而不必担心我们无意中删除它们；执行函数只关心它自己的节点排列。

5.10 练习

新的关键字是必要的，以实现事情，例如**while**或**report loops**，if 语句，等等。识别是在**stmt()**中处理的，但在大多数情况下，这只是编译器构造的问题，而不是继承问题。一旦我们决定了语句的类型，我们将构建节点类型，比如 When、Repeter、或 IfElse，符号表中的关键字不需要知道它们的存在。

更有趣的是 ANSI-C 的数学库中带有两个参数的函数，如**atan2()**。从符号表的角度来看，函数就像简单的函数一样处理，但是对于表达式树，我们需要发明一个具有三个后代的新的节点类型。

用户自定义的函数提出了一个非常有趣的问题。如果我们用 *Parm*

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

消息转发：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**:

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

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

```
1 struct table{ struct table * tp; }
2     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:

```
1 extern struct x object;     // forward reference
2 f() { object = value; }     // using the reference
3 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()**:

```
1 void * bsearch(const void * key
2     const void * table, size_t nel, size_t width,
3     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**:

```
1 const struct Class { ... } * p; // protects contents of structure
2 typedef struct Class { ... } * ClassP;
3 const ClassP cp;              // contents open, pointer protected
```

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

```
1  main()
2  {
3      typedef int matrix [10][20];
4      matrix a;
5      int b [10][20];
6
7      int f(const matrix);
8      int g(const int [10][20]);
9
10     f(a);
11     f(b);
12     g(a);
13     g(b);
14 }
```

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:

```
1 struct u { int u; double v; } u;
2 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:

```
1 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:

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

A structure may contain a structure:

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

The complete sequence of component names is required for access:

```
1 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:

```
1 struct a { int x; };
2 struct c { sturct a a; ... } c, * cp = & c;
3 struct a * ap = & c.a;
4
5     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:

```
1 ap = cp;                      // wrong
2 c.x, cp -> x                  // wrong
3 cp -> a.x                     // OK, fully specified
4 ( (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:

```
1 void *      f    (void *);       // function
2 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

munch：生成类列表

```
1    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.

ooc：ANSI C 面向对象编码预处理器

```
1    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.

词法

<++>

类描述文件

<++>

预处理

<++>

标签

<++>

报告文件

<++>

环境

<++>

C.2 函数

<++>

retrieve：从文件获取对象

<++>

C.3 根类

<++>

简介：根类入门

<++>

Class Class: Object —root metaclass

<++>

Class Exception: Object —manage a stack of exception handlers

<++>

Class Object —root class

<++>

C.4 GUI 计算器类

<++>

intro —introduction to the calculator application

<++>

lcClass Crt: lc —input/output objects for curses

<++>

Class Event: Object —input item

<++>

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

<++>

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.