第 1 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

```
typedef struct node {
    struct node * next;
    ... information ...
} node;
```

而对线性表的操作，我们给出如下表头函数：

```
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。

1.2 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

[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

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 **??** 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：

[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。

```
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

```
#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?");

```

```
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：

```
1   #if ! defined MANY || MANY < 1
2   #define MANY    10
3   #endif
4
5   static int heap [MANY];
6
7   void * new (const void * type, ...)
8   {
9       int * p;        /* & heap[1..] */
10
11      for ( p = heap + 1; p < heap + MANY; ++p )
12          if ( ! *p )
13              break;
14      assert( p < heap + MANY );
15      *p = MANY;
16      return p;
17  }
```

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。?? 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。

```
1 void delete (void * _item)
2 {
3     int * item = _item;
4     if (item)
5     {   assert(item > heap && item < heap + MANY);
6         * item = 0;
7     }
8 }
```

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。

```
1 void * add (void * _set, const void * _element)
2 {
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：

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

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

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。

```
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() 函数依旧要检查要查找的元素是不是指向着合适的集合：

```
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() 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。

```
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;
}
```

现在又新加入一个函数 **count()**，用来计算集合中元素的个数：

```
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 **??** 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：

```
1 ok
2 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：

```
1 int store (const void * object, FILE * fp);
2 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：

```
1 int apply (const void * set,
2     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 2 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 **??** 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 */
7      const void * destroy;    /* locate destructor */
8  };
9  struct Set {
10     ... information ...
11     const void * destroy;
12 };
```

[1]译注：即，需要将数据类型的信息硬编码到 **new()** 中

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 **??** 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。

```
1  void * new (const void * _class, ...)
2  {
3      const struct Class * class = _class;
4      void * p = calloc(1, class -> size);
5      assert(p);
6      * (const struct Class **) p = class;
7      if (class -> ctor)
8      {   va_list ap;
9          va_start(ap, _class);
10         p = class -> ctor(p, & ap);
11         va_end(ap);
12     }
13     return p;
14 }
```

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。**??** 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 **??** 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 **??** 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 **??** 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：

```
1  #include "String.h"
2  #include "new.h"
3  int main ()
4  {
5      void * a = new(String, "a"), * aa = clone(a);
6      void * b = new(String, "b");
7      printf("sizeOf(a) == %u\n", sizeOf(a));
8      if (differ(a, b))
9          puts("ok");
10     if (differ(a, aa))
11         puts("differ?");
12     if (a == aa)
13         puts("clone?");
14     delete(a), delete(aa), delete(b);
15     return 0;
16 }
```

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：

```
1 sizeOf(a) == 8
2 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。

[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

```
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：

```
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 ?? 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?