А. А. Вылиток

Язык Си. Реализация списков с помощью цепочек динамических объектов

В языке Си нет встроенных типов данных и операций для работы со списками. Программируя на языке Паскаль (в котором также нет стандартных типов для списков), мы представляли списки цепочками динамических объектов. Аналогичная реализация возможна и в языке Си.

1. Стандартные функции для работы с динамической памятью

Для работы с динамическим объектами в Си есть аналоги паскалевских процедур *new* и *dispose* — это функции стандартной библиотеки *malloc* и *free*. Для их использования нужно включить в программу заголовочный файл ‹*stdlib.h*›. В этом файле также вводится обозначение *NULL* для пустого (нулевого) указателя.

***void** *malloc* (*size_t size*)

malloc возвращает указатель на место в памяти для объекта размера *size*. Выделенная память не инициализируется. Если память отвести не удалось, то результат работы функции — *NULL*. (Тип *size_t* — это беззнаковый целый тип, определяемый в файле ‹*stddef.h*›, результат операции *sizeof* имеет тип *size_t*). Как правило, обобщенный указатель, возвращаемый этой функцией, явно приводится к указателю на тип данных. Например, создать динамическую переменную типа *double* и присвоить значение, возвращаемое *malloc*, переменной *dp* — указателю на *double*, можно с помощью выражения

dp = (***double****) *malloc* (***sizeof*** (***double***)).

Созданная динамическая переменная существует вплоть до завершения работы программы, или до момента, когда она явно уничтожается с помощью функции *free*.

void** free* (void** *p*)

free освобождает область памяти, на которую указывает *p*; если *p* равно *NULL*, то функция ничего не делает. Значение *p* должно указывать на область памяти, ранее выделенную с помощью функций *malloc* или *calloc* После освобождения памяти результат разыменования указателя *p* непредсказуем; результат также непредсказуем при попытке повторного обращения к *free* с этим же указателем.

Приведем описание еще одной функции распределения памяти в Си. Ею удобно пользоваться, когда нужно разместить массив в динамической памяти.

***void** *calloc* (*size_t nobj, size_t size*)

calloc возвращает указатель на место в памяти, отведенное для массива *nobj* объектов, каждый из которых имеет размер *size*. Выделенная область памяти побитово обнуляется. (Заметим, что это не всегда равнозначно присваиванию нулевых значений соответствующим элементам массива. В некоторых реализациях в побитовом представлении нулевого указателя или значения 0.0 с

плавающей точкой могут быть ненулевые биты). Если память отвести не удалось, то результат работы функции — *NULL*.

2. Представление списков цепочками звеньев

Для хранения отдельного элемента списка создается динамический объект — структура с двумя членами, называемая *звеном*. В одном из членов (*информационном*) располагается сам элемент, а другой член содержит указатель на звено, содержащее следующий элемент списка, или пустой указатель, если следующего элемента нет. С помощью указателей звенья как бы сцеплены в *цепочку*. Зная указатель на первое звено можно добраться и до остальных звеньев, то есть указатель на первое звено задаёт весь список. Пустой список представляется пустым указателем.

Приведём соответствующие описания на языке Си. В качестве типа элемента списка выбран тип ***char***. Списки с элементами других типов описываются аналогично.

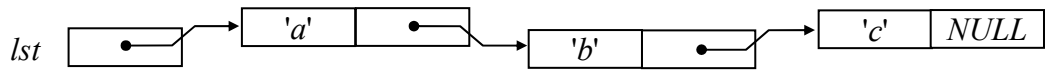
```
#include <stdio.h>
#include <stdlib.h>

typedef  struct Node *link;        /*  указатель на звено                */
typedef  char  elemtype;           /*  тип элемента списка               */

typedef struct Node {              /*  звено состоит из двух полей:      */
        elemtype elem;             /*    – элемент списка,               */
             link next;            /*    – указатель на следующее звено  */
        } node;

typedef link list;                 /*  список задается указателем на звено */

list  lst;                         /*  переменная типа список            */
```

Переменная *lst* представляет в программе список.

Обратите внимание на то, что в описании типа *link* используется незавершенный тип *struct Node*. Описание указателей на незавершенный тип допустимо в Си. Тип *struct Node* становится завершенным при описании типа *node*. Тип *list* объявляется синонимом типа *link*.

В примерах мы будем обозначать обсуждаемые списки в виде кортежей, как это принято в математике. Так, конструкция ⟨'*a*', '*b*', '*c*'⟩ означает список из трех элементов. Первый элемент в этом списке — '*a*', второй — '*b*', третий — '*c*'. Пустой список выглядит так ⟨ ⟩.

Пример. Список символов ⟨'*a*', '*b*', '*c*'⟩, представленный цепочкой звеньев, изображается следующим образом (в переменной *lst* — указатель на первое звено):



При этом в программе выражение *lst* означает указатель на первое звено в цепочке; **lst* означает само первое звено, (**lst*).*elem* — первый элемент списка. По-другому первый элемент обозначается с помощью операции доступа к члену структуры через указатель: *lst−>elem*. Выражение *lst−>next* означает указатель на второе звено. Далее,

**lst−>next*	— само второе звено,
lst−>next−>elem	— второй элемент списка,
lst−>next−>next	— указатель на третье звено,
**lst–>next−>next*	— само третье звено,
lst−>next−>next−>elem	— третий элемент списка,
lst−>next−>next−>next	— пустой указатель (конец списка).

Выражение *lst* имеет и другой смысл. Оно обозначает список в программе, поскольку, зная указатель на первое звено, мы имеем доступ ко всем остальным звеньям, т.е. «знаем» весь список. С этой точки зрения выражение *lst−>next* в нашем примере обозначает список[1] ⟨'*b*', '*c*'⟩, а выражение *lst−>next−>next−>next* — пустой список.

Заметим, что соседние звенья цепочки располагаются в оперативной памяти произвольно относительно друг друга, в отличие от соседних компонент массива, всегда занимающих смежные участки памяти. Такое расположение звеньев облегчает операции вставки и удаления, так как нет необходимости перемещать элементы, как это было бы в случае реализации списков массивами. Поясним это на примерах.

Пусть из списка ⟨'*a*', '*b*', '*c*'⟩, представленного в программе переменной *lst*, требуется удалить второй элемент. Для этого достаточно исключить из цепочки второе звено – «носитель» второго элемента. Изменим указатель в поле *next* первого звена:

$$lst{->}next = lst{->}next{->}next.$$

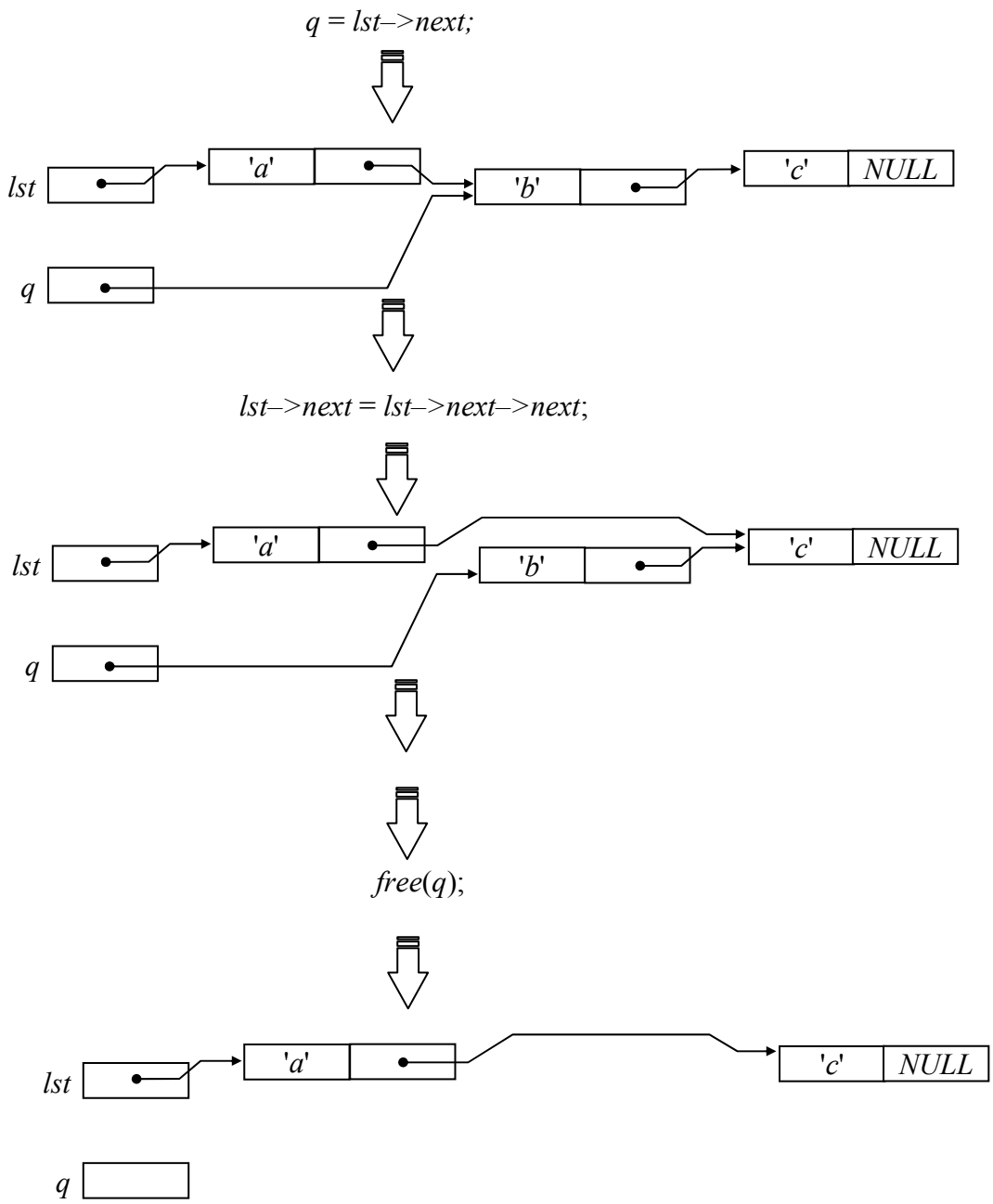
Теперь после первого звена в цепочке идёт звено, содержащее элемент '*c*'. Получился список ⟨'*a*', '*c*'⟩. Чтобы исключённое звено не занимало места в памяти, его следует уничтожить с помощью функции *free*, предварительно запомнив указатель на него во вспомогательной переменной *q*. Итак, окончательное решение таково:

$$q = lst{->}next;\ lst{->}next = lst{->}next{->}next;\ free(q);$$

Покажем на рисунке происходящие после каждого действия изменения.

---

[1] По правилам языка Си имена *link* и *list* обозначают один и тот же тип, и мы вправе рассматривать *lst−>next* как переменную типа *list*, означающую список.



Пусть теперь требуется вставить '*d*' после первого элемента списка ⟨'*a*', '*c*'⟩. Решение состоит из двух этапов. Во-первых, необходимо создать «носитель» — звено для хранения вставляемого элемента, и занести этот элемент в поле *elem* «носителя». Во-вторых, путём изменения указателей включить созданное звено в цепочку после первого звена. Первый этап реализуется фрагментом

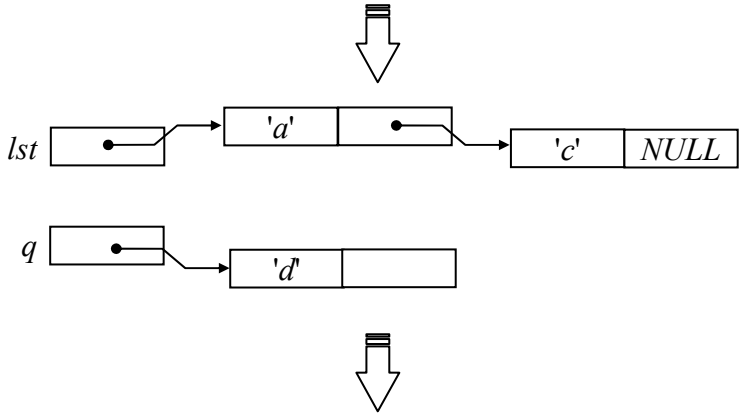
$$q = (link)\ malloc(\textbf{sizeof}\ (node));\ q{-}{>}elem = 'd';$$

где *q* — вспомогательная переменная типа *link*. Фрагмент

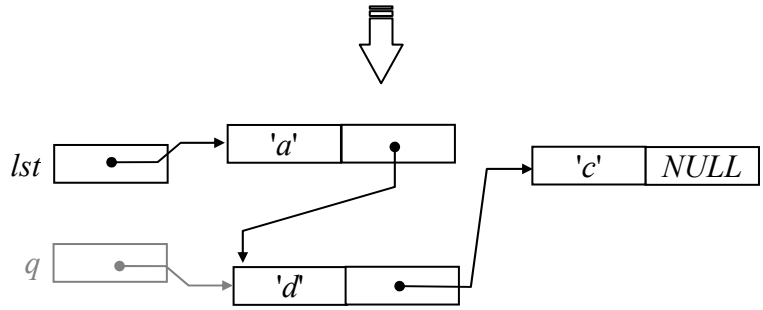
$$q{-}{>}next = lst{-}{>}next;\ lst{-}{>}next = q;$$

осуществляет второй этап вставки. Следующий рисунок иллюстрирует этапы вставки.

q = (*link*) *malloc*(*sizeof* (*node*)); *q*−>*elem* = '*d*';



q–>*next* = *lst*–>*next*; *lst*–>*next* = *q*;



Из примеров видно, что длина цепочки (количество звеньев в ней) может динамически изменяться, т.е. изменяться в процессе выполнения программы. Подобно цепочкам можно сконструировать и более сложные структуры, в которых объекты связаны между собой с помощью указателей. Такого рода структуры данных называются *динамическими*.

3. Некоторые операции со списками

Приведём описание нескольких функций для работы со списками.

Создание списка

Функция *create* (*s*) создает и возвращает в качестве результата список из символов параметра-строки *s*.

```
list create (char *s)
{
    int  i;
    link  cur;                  /* указатель на текущее звено списка      */
    list   res;                 /*  возвращаемый список                   */
    if ( *s == '\0' ) return NULL;  /* если строка пуста, то возвращаем
                                       пустой список
                                       иначе строим непустой список        */
    res = cur = ( link ) malloc ( sizeof (node)); /* создаем первое звено списка  */
    cur−>elem = *s++;           /* заполняем поле elem и переходим к
```

```
                                        следующему элементу строки          */
    while ( *s != '\0' ) {              /* пока не конец строки                */
        cur = cur->next = (list) malloc( sizeof (node));  /* присоединяем в конец
                                        списка новое звено                   */
        cur->elem = *s++;               /* помещаем в новое звено очередной элемент
                                        и переходим к следующему элементу
                                        строки                               */
    }
    cur->next = NULL;                   /* последнее звено должно иметь нулевой
                                        указатель                            */
    return res;                         /* возвращаем  список (указатель на первое
                                        звено)                               */
}
```

Условие *s != '\0' можно заменить на *s, а *s == '\0' заменить на !*s .

Печать элементов списка

```
/* print: печатает в стандартный выходной поток элементы списка        */
void  print ( list  p )
{
    while ( p != NULL ) {       /*   пока не конец списка,              */
        putchar ( p->elem );    /*   печатаем очередной элемент         */
        p = p->next;            /*   и переходим к следующему звену     */
    }
    putchar ( '\n' );
}
```

Заголовок *while*-цикла можно было записать как *while* (*p*). Однако, выражение *p* != *NULL* более информативно — по нему можно догадаться, что *p* скорее всего является указателем, даже не видя его описания.

Опишем функцию печати, используя цикл *for*.

```
/* print:   печать в стандартный выходной поток элементов списка
        версия с циклом for                                              */
void print ( list p )
{
    for ( ; p ; p = p->next )   /* пока не конец списка, т.е. p!=NULL   */
    putchar ( p->elem );        /* печатаем очередной элемент
                                   и переходим к следующему звену       */
    putchar( '\n' ) ;
}
```

Здесь мы использовали *p* в качестве условия продолжения цикла, а не *p* != *NULL*, так как рядом стоящее *p –>next* (в третьем выражении заголовка *for*), говорит о том, что *p* — указатель.

Вычисление свойств и характеристик списков

```
/*  is_empty: возвращает 1, если список пуст, иначе —  0                    */
int is_empty ( list ls)
{
        return ls == NULL;
}

/*  count:  подсчитывает число вхождений элемента elem в список ls          */
int count ( list ls, elemtype elem)
{
        int c = 0;   /* счетчик вхождений                                   */
        for ( ; ls ; ls = ls−>next )
                if ( ls−>elem == elem ) c++ ;
        return c;
}
```

Можно обойтись без условного оператора в теле цикла for, записав вместо него оператор-выражение:

```
c += ( ls−>elem == elem);
```

```
/*  length:   вычисляет длину списка ls                                     */
int length ( list ls )
{
        int c;
        for ( c = 0; lst ; lst = lst−>next, c++)
                ;
        return c;
}
```

Теперь опишем рекурсивную версию функции *length*. Для этого список удобно представлять себе как рекурсивную структуру данных: он либо пуст, либо состоит из «головы» (первый элемент) и «хвоста» (все, кроме первого). Хвост, в свою очередь, также является списком.

```
/* length: рекурсивно вычисляет длину списка ls;
        длина пустого списка равна нулю,
        длина непустого списка на единицу больше длины его хвоста          */
int length ( list ls)
{
        return ( ls ) ? length ( ls−>next) + 1 : 0;
}
```

Вставка элемента

```
/* insfront: вставляет элемент elem в начало списка, переданного через
        указатель lp                                                    */
void insfront ( elemtype elem, list * lp)
{
        list cur = ( list ) malloc ( sizeof ( list ) );
        cur->elem = elem;
        cur->next = * lp;
        * lp = cur;
}
```

Удаление списка

После того, как список в программе стал не нужен, следует удалить его, освободив память, занимаемую звеньями.

```
/* destruct : удаляет список, освобождая занимаемую им  память        */
void destruct ( list ls )
{
        link q;
        while ( ls != NULL ) {
                q = ls ;
                ls = ls->next;
                free (q);
        }
}
```

Задача. Ввести последовательность символов (не более 80). Добавить в начало последовательности символ '*a*', если длина последовательности меньше 10 и в ней есть хотя бы один символ '*b*'. Напечатать результат.

Решение. Представим последовательность в виде списка, используя описанные выше глобальную переменную *lst* и функции работы со списками.

```
int main ( )
{
        char buf [81];
        scanf  ( "%80s", buf );
        lst = create ( buf );
        if ( length ( lst ) < 10 && count ( lst, 'b' ) )
                insfront ( 'a', &lst );
        print ( lst );
        destruct ( lst );
 }
```