Кафедра вычислительных систем

ПРОГРАММИРОВАНИЕ

# Динамические структуры данных

Преподаватель:

Старший преподаватель Кафедры ВС

**Перышкова Евгения Николаевна**

# Кардинальные числа типов данных

Общим свойством структур данных, изученных к настоящему моменту:

1) скалярных переменных;
2) статических массивов;
3) структур;

является то, что их кардинальное число (количество элементов) конечно.

Представление таких структур в памяти компьютера является достаточно простым.

Большинство *усложненных* структур :

1) последовательности;
2) деревья;
3) графы;

характеризуются тем, что их кардинальные числа бесконечны.

# Динамические структуры данных

Структуры данных, имеющие нефиксированное кардинальное число называются динамическими. Для их реализации используется динамическая память.

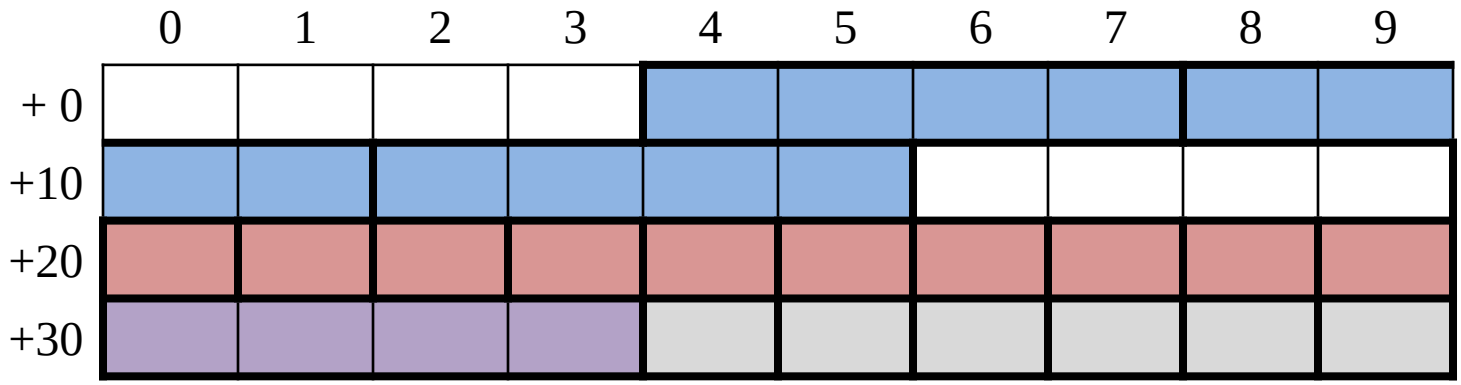
В рамках данной лекции будут рассмотрены только динамические структуры данных, позволяющие хранить последовательности однотипных элементов. Такие структуры можно рассматривать как альтернативу статическим массивам, изученным ранее.

Рассматриваемые структуры:

- Динамически-расширяемые массивы
- Односвязные списки

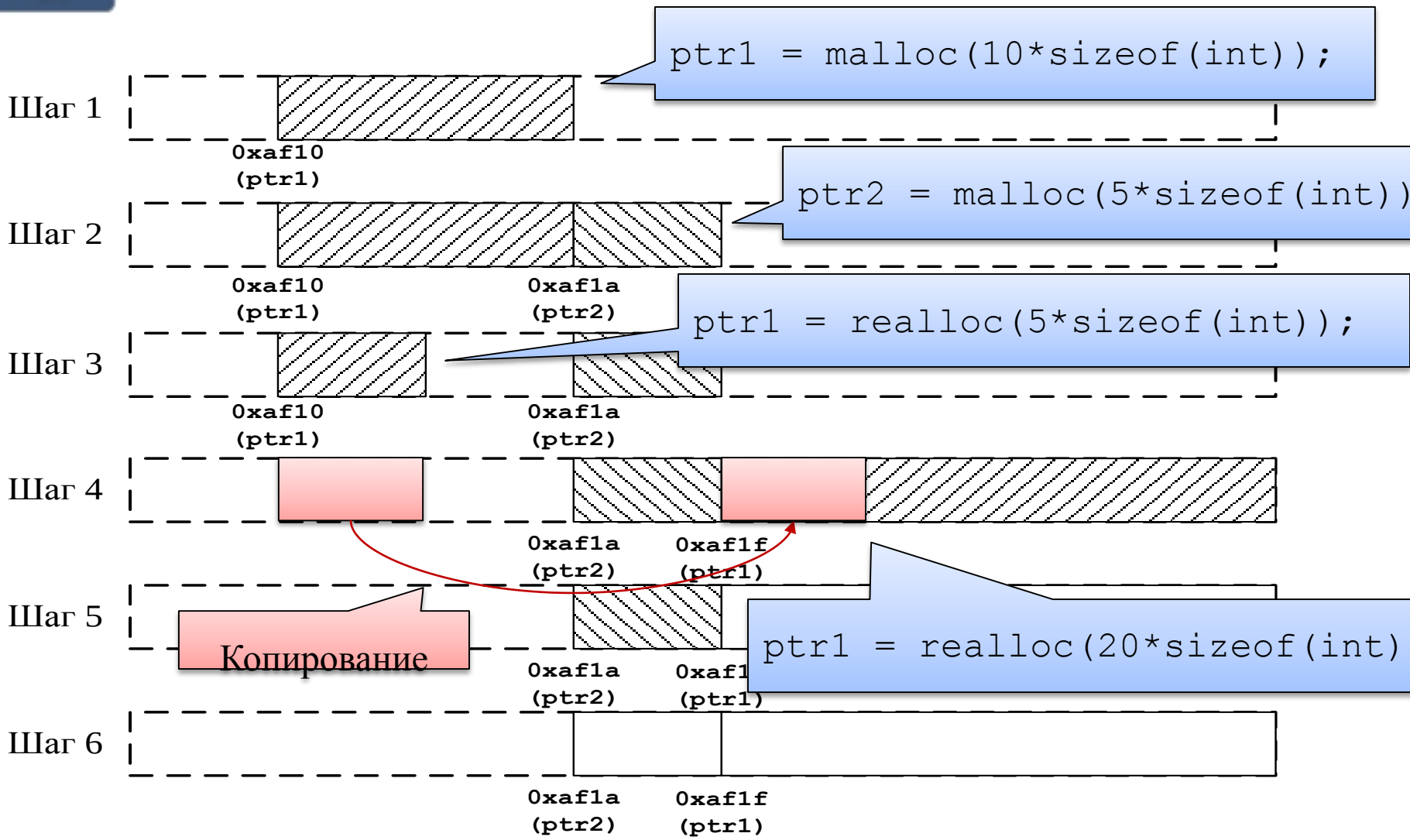
# Динамическая память

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Смещение | **4** | **20** | **30** | **34** | 0 | 0 | 0 | 0 | 0 |
| Размер | **12** | **10** | **4** | **6** | 0 | 0 | 0 | 0 | 0 |




- Инструменты работы с динамической памятью позволяют обеспечить изменение размера выделенной памяти с сохранением ее содержимого.
- Для этого используется функция realloc.
- Изменение размера динамически выделенной области может приводить к значительным накладным расходам
- Частое применение операции `realloc` нежелательно

# Накладные расходы **realloc**



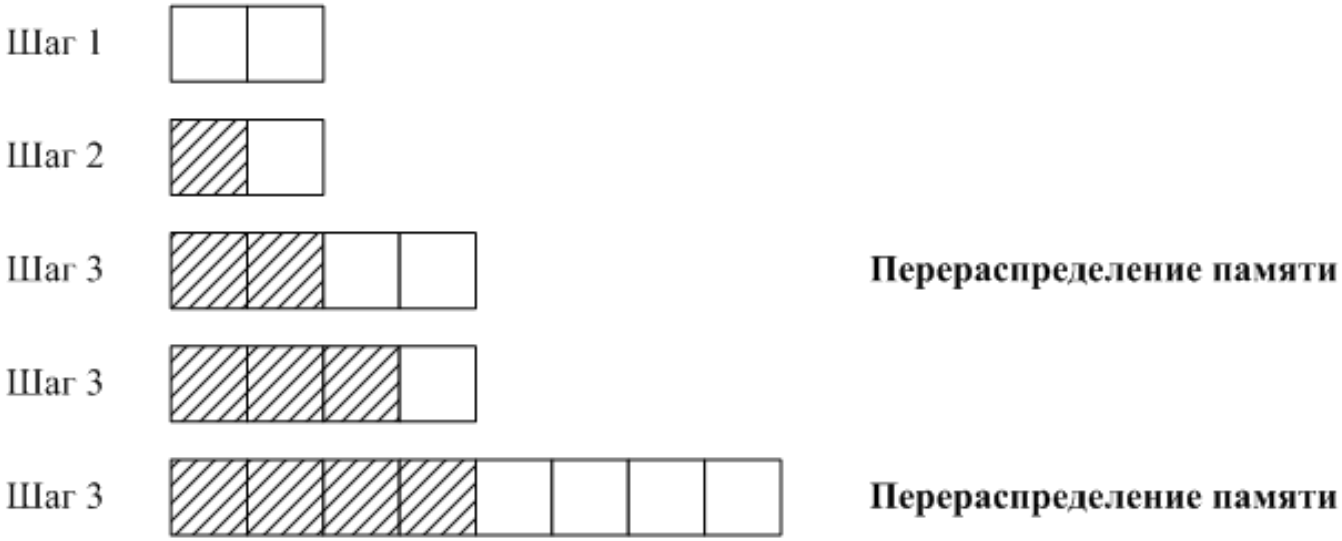
# Динамически-расширяемый массив

Для уменьшения количества вызовов функции `realloc` может быть эффективно применена следующая стратегия:

1. Изначально выделяется некоторый начальный объем памяти *X*.

2. Если при добавлении очередного элемента текущего объема памяти не достаточно, то размер увеличивается в **два раза:** $X = X*2$.

3. Если при удалении очередного элемента количество использованной памяти $X/4$, то размер памяти сокращается в два раза: $X = X/2$.



# **Описание** и **инициализация** динамически-расширяемого массива

```
struct darray {
    <type> *ptr; // указатель на начало
                 // динамической области
    unsigned int size; // размер области
    unsigned int used; // использовано элементов
};
```

```
#define INIT_SIZE 16
int dinit(striuct darray *da)
{
    da->size = INIT_SIZE; // начальный размер массива
    // Выделение памяти для хранения массива
    da->ptr = malloc(INIT_SIZE * sizeof(<type>));
    if( da->ptr == NULL )
        return -1;
    da->used = 0;
    return 0;
}
```

# Изменение размера
## динамически-расширяемого массива

```
int dexpand(struct darray *ptr, int cnt)
{
    if( da->used + cnt < da->size){
        // Памяти достаточно
        da->used += cnt;
    }else{
       int k;
       for(k=1; da->used + cnt > k*da->size; k=k*2);
       // k - количество раз, в которое необходимо увеличить
       // размер массива.
       da->ptr = realloc(da->ptr,k*da->size*sizeof(<type>));
       if( da->ptr == NULL )
           return -1;
       da->size *= k;
    }
    return 0;
}
```

# **Размер/доступ/очистка**
## динамически-расширяемого массива

```
int *dcount(struct darray *da)
{
    return da->used;
}

<type> *dptr(struct darray *da)
{
    return da->ptr;
}

void dfree(struct darray *da)
{
    free(da->ptr);
    da->size = da->used = 0;
}
```

# Демонстрационная программа

```
int main()
{
    struct darray da;
    int i;
    dinit(&da);
    for(i=0;i<80; i++){
        if( dexpand(&da,1) == 0 ){
            dptr(&da)[i] = i+1;
        }
    }

    for(i=0;i<dcount(&da); i++){
        printf("arr[%d] = %d\n",i, dptr(&da)[i]);
    }

    dfree(&da);
    return 0;
}
```

# Недостатки динамически-расширяемых массивов

1. Использование (пусть и не регулярное) функции `realloc`, которая характеризуется значительными накладными расходами (может приводить к копированию всей последовательности).

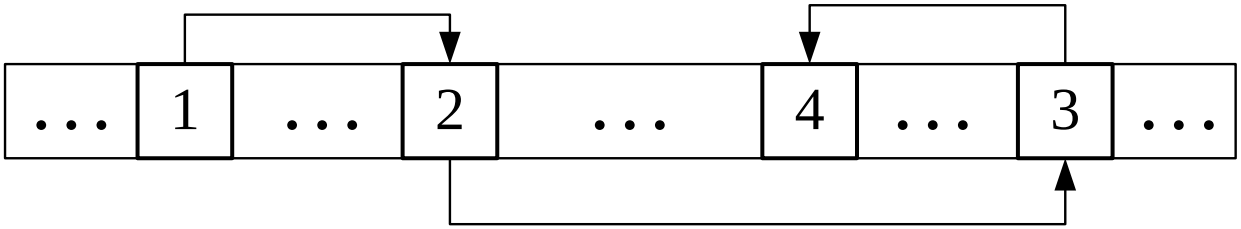
2. Вставка или удаление элементов массива требует перемещения существующих элементов. Это может приводить к перемещению больших объемов данных. Например, сдвига большей части элементов последовательности вправо при удалении.

3. Наличие неиспользуемых ячеек, количество которых может достигать половины от общего числа ячеек.

# Подходы к размещению элементов последовательности в памяти

Непрерывное размещение


Произвольное размещение



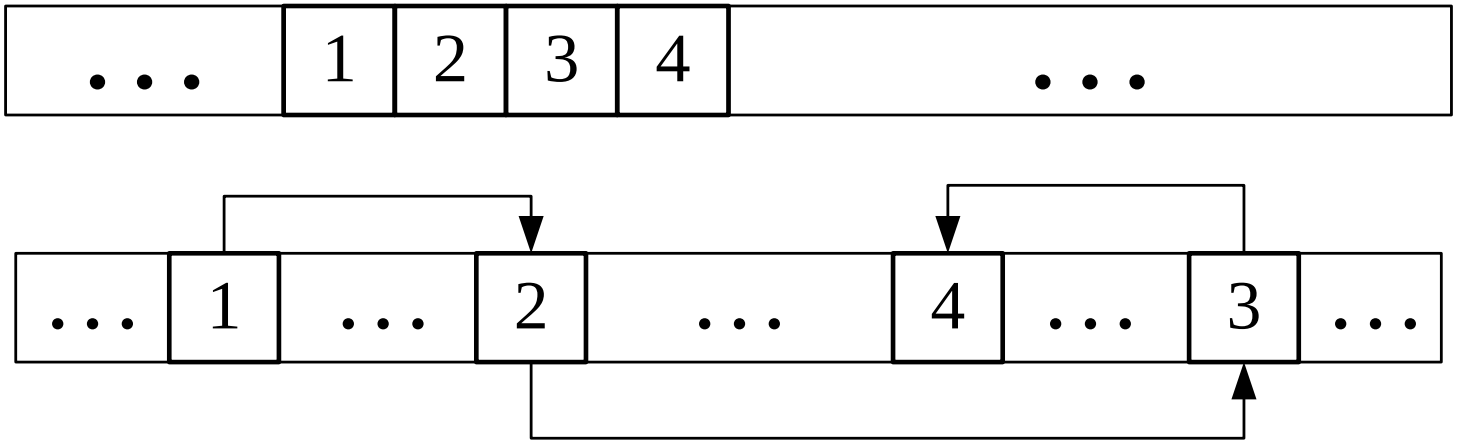

Произвольное размещение элементов в памяти позволяет:

1. **Избавиться от** необходимости **перемещения элементов** при их вставке/удалении в последовательность.

2. Позволяет выделять **только необходимый** объем памяти для хранения последовательности.

3. Обеспечивает выделение/освобождение небольших объемов динамической памяти (под каждый элемент в отдельности).

# Реализация последовательностей с произвольным размещением элементов



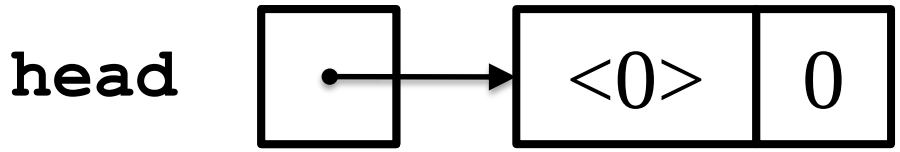

1. Необходим механизм обеспечения связи между элементами. В языке СИ для этого подходят указатели. Обеспечение связности является задачей программиста.

2. В отличие от массивов для доступа к произвольному элементу требуется произвести доступ к элементам расположенным перед ним в последовательности.

# **Описание** и **инициализация** списка

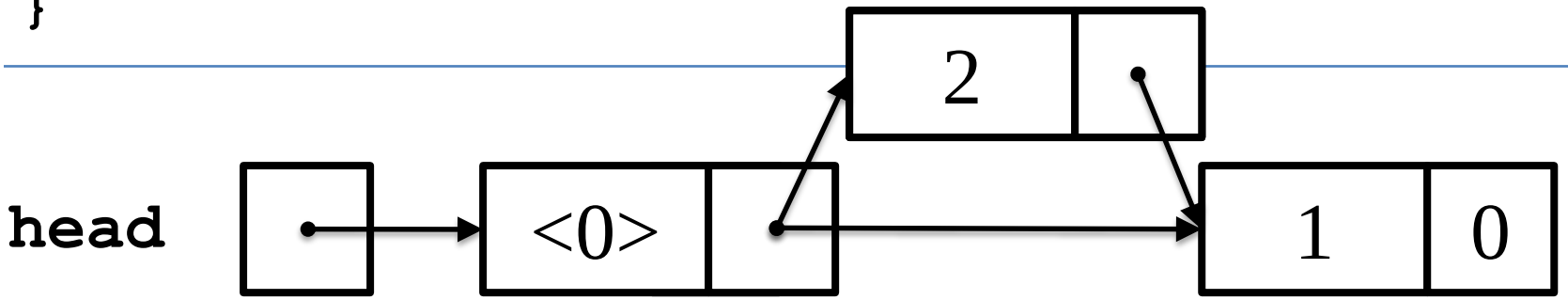
```
struct list {
    <type> val; // Информационное поле
    struct list *next; // Указатель на след. элемент
}
```

```
struct list *linit()
{
    struct list *head = malloc(sizeof(struct list));
    if( head == NULL )
        return head;
    head->next = NULL;
    return head;
}
```



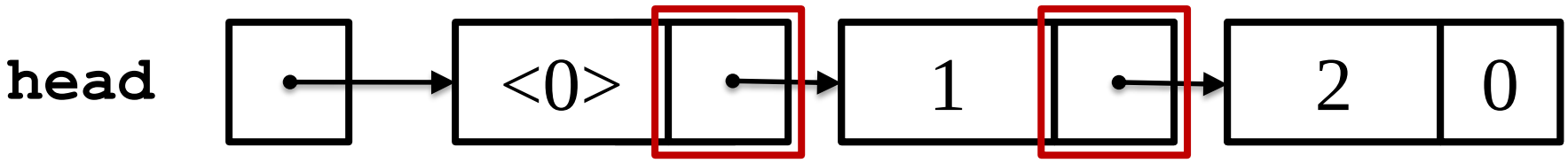
# **Включение** элемента **в начало** списка

```
int linsfirst(struct list *head, int elem)
{
    struct list *tmp;
    tmp->next = malloc( sizeof(struct list));
    if( tmp->next == NULL )
        return -1;
    tmp->next = head->next;
    tmp->val = elem;
    head->next = tmp;
    return 0;
}
```



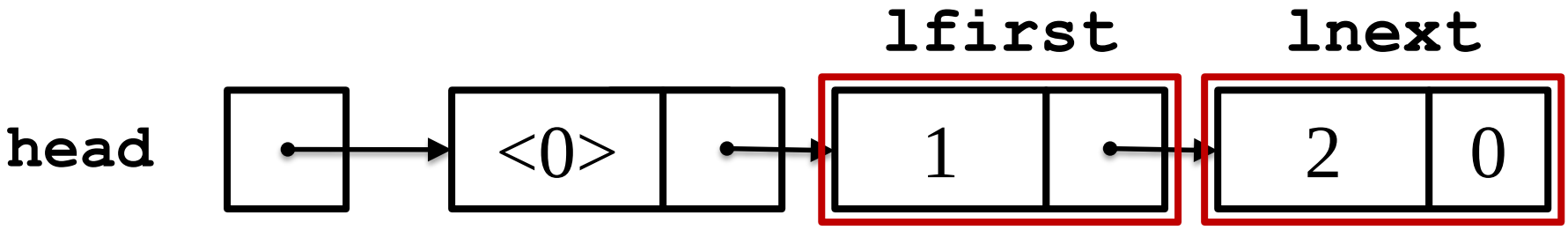
# **Включение** элемента **в конец** списка

```
int linslast(struct list *head, int elem)
{
    struct list *tmp = head;
    for( ; tmp->next != NULL; tmp = tmp->next);
    tmp->next = malloc( sizeof(struct list));
    if( tmp->next == NULL )
        return -1;
    tmp = tmp->next;
    tmp->next = NULL;
    tmp->val = elem;
    return 0;
}
```



# Доступ к элементам списка

```
struct list *lfirst(struct list *head)
{
    return head->next;
}
struct list *lnext(struct list *elem)
{
    if( elem != NULL )
        return elem->next;
    else
        return NULL;
}
```



# **Пример работы** со списками

```
int main()
{
    struct list *head, *elem;
    int i;
    head = list_init();
    if( head == NULL ){
        return 0;
    }
    for(i=0;i<80; i++){
        if( linslast(head, i+1) ){
            printf("Cannot add new element: i=%d\n",i);
        }
    }
     for(elem=lfirst(head); elem != NULL; elem=lnext(head)){
        printf("arr[%d] = %d\n",i, elem->val);
    }
    list_free(&da);
    return 0;
}
```

# Литература

1. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ, М.:МЦНМО, 2002, 960 с.

2. Кнут, Д.Э. Искусство программирования. Том 1. Основные алгоритмы. – Вильямс, 2010. – (Серия: Искусство программирования). – ISBN 978-5-8459-0080-7.