30. Динамический расширяемый массив.
План ответа:
	1. функция realloc и особенности ее использования;
	2. описание типа;
	3. добавление и удаление элементов.

1. функция realloc и особенности ее использования;
	void* realloc(void *ptr, size_t size);

	ptr == NULL && size != 0	=>	Выделение памяти (как malloc)

	 ptr != NULL && size == 0	=>	Освобождение памяти аналогично fr
ee().

	ptr != NULL && size != 0	=>	Перевыделение памяти.
		В худшем случае:
			выделить новую область
			скопировать данные из старой области в новую
			освободить старую область

	int *p = malloc(10 * sizeof(int));

	p = realloc(p, 20 * sizeof(int));// НЕВЕРНО А если realloc вернула NULL?

Првильно

	int *tmp = realloc(p, 20 * sizeof(int));
	if (tmp)
	    p = tmp;
	else
	    // обработка ошибки

2. описание типа;
	Для уменьшение потерь при распределении памяти изменение размера должно п
роисходить относительно крупными блоками.
	Для простоты реализации указатель на выделенную память должен храниться в
месте со всей информацией, необходимой для управления динамическим массивом.
	struct dyn_array
	{
	    int len;
	    int allocated;
	    int step;
	    int *data;
	};

	#define INIT_SIZE 1

	void init_dyn_array(struct dyn_array *d)
	{
	    d->len = 0;
	    d->allocated = 0;
	    d->step = 2;
	    d->data = NULL;
	}

3. добавление и удаление элементов.

	int append(struct dyn_array *d, int item)
	{
	    if (!d->data)
	    {
	        d->data = malloc(INIT_SIZE * sizeof(int));
	        if (!d->data)
	            return -1;
	        d->allocated = INIT_SIZE;
	    }
	    else
	        if (d->len >= d->allocated)
	        {

```
                int *tmp = realloc(d->data,
                                   d->allocated * d->step * sizeof(int));
                if (!tmp)
                    return -1;
                d->data = tmp;
                d->allocated *= d->step;
            }
        d->data[d->len] = item;
        d->len++;
        return 0;
    }
```

Удвоение размера массива при каждом вызове realloc сохраняет средние «ожидаемые» затраты на копирование элемента.

Поскольку адрес массива может измениться, программа должна обращаться к элементами массива по индексам.

Благодаря маленькому начальному размеру массива, программа сразу же «проверяет» код, реализующий выделение памяти.

```
    int delete(struct dyn_array *d, int index)
    {
        if (index < 0 || index >= d->len)
            return -1;

        memmove(d->data + index, d->data + index + 1,
                           (d->len - index - 1) * sizeof(int));
        d->len--;

        return 0;
    }
```

«+»

- Простота использования.
- Константное время доступа к любому элементу.
- Не тратят лишние ресурсы.
- Хорошо сочетаются с двоичным поиском.

«-»

- Хранение меняющегося набора значений.