# 5. Указатели и массивы

Владимир Верстов

Б. Керниган, Д. Ритчи. Язык программирования C

# Указатель

- … — это переменная, содержащая адрес другой переменной

- Определенные операции можно выполнить только при помощи указателей

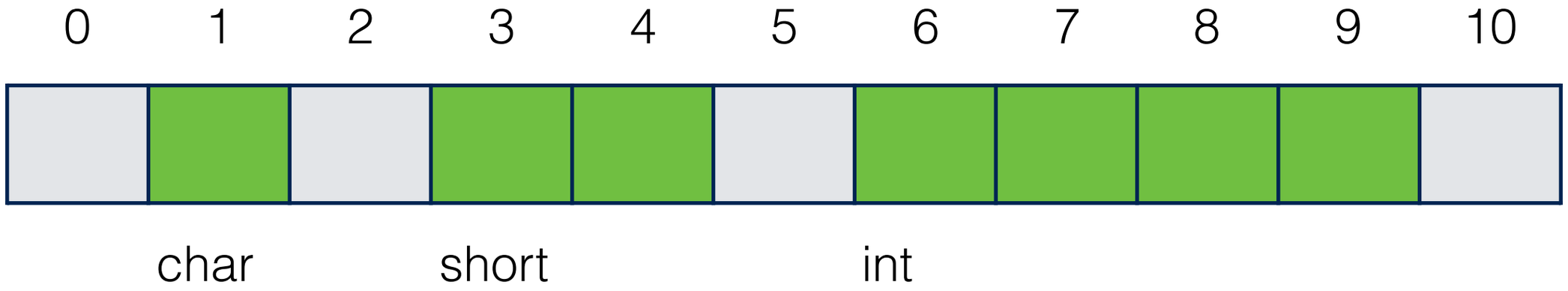
- Указатели и массивы тесно связаны между собой

- Указатели — верный способ запутать программу до невозможности

# Организация памяти

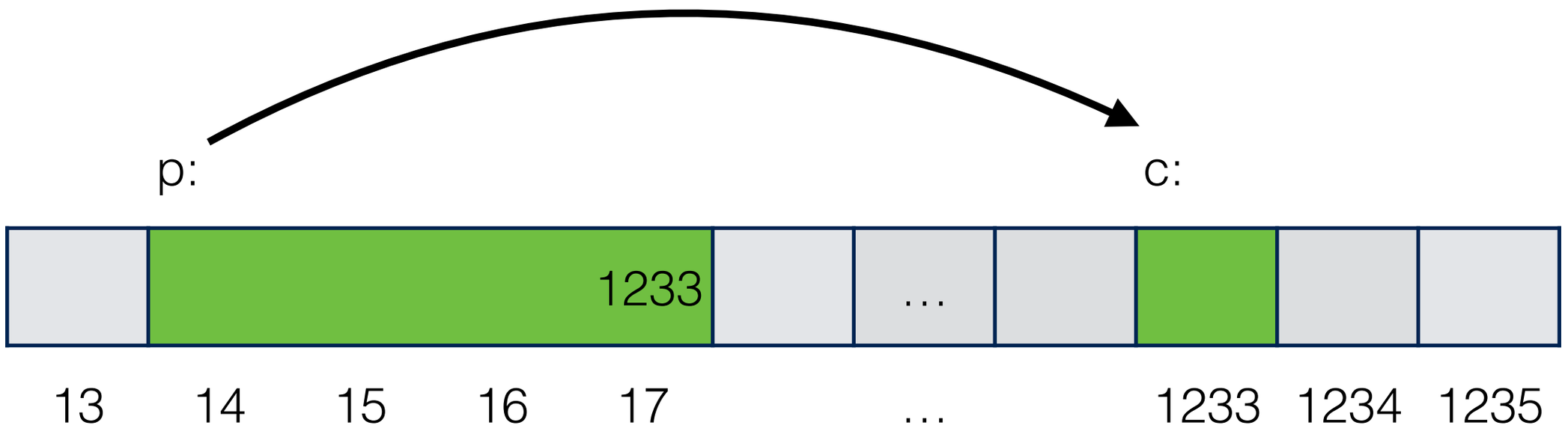
- Типичная система содержит массив последовательно пронумерованных ячеек памяти, с которыми можно работать по отдельности либо целыми непрерывными группами

# Пример организации памяти



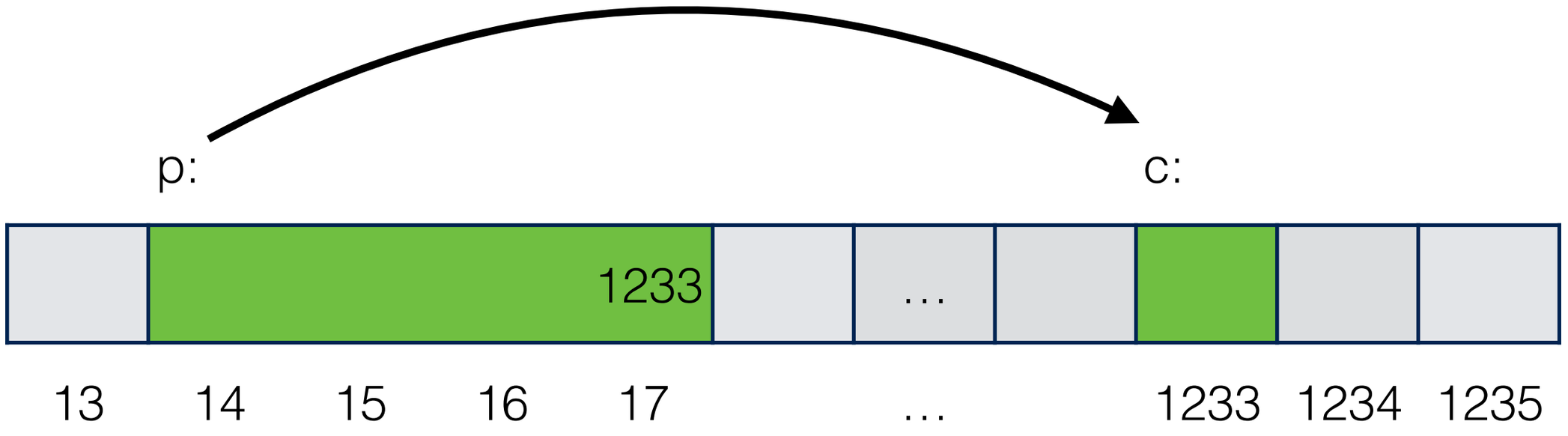
- Указатель занимает чаще всего занимает 4 или 8 байт (ячеек) в памяти
- 4 байта для 32-х разрядной системы
- 8 байт для 64-х разрядной системы

# Пример



# Пример

```
char *p;
char c;

// p указывает на c
p = &c;
```



# Получение адреса

- & — операция для получения адреса объекта
- Операция & применима только к объектам, которые хранятся в оперативной памяти, переменным и массивам
- Операция & не применима к `register`-переменным

# Получение значения

- * — операция ссылки по указателю (indirection) для получение значения объекта, на который указывает указатель

- Операцию * называют разыменованием указателя (dereferencing)

# Примеры

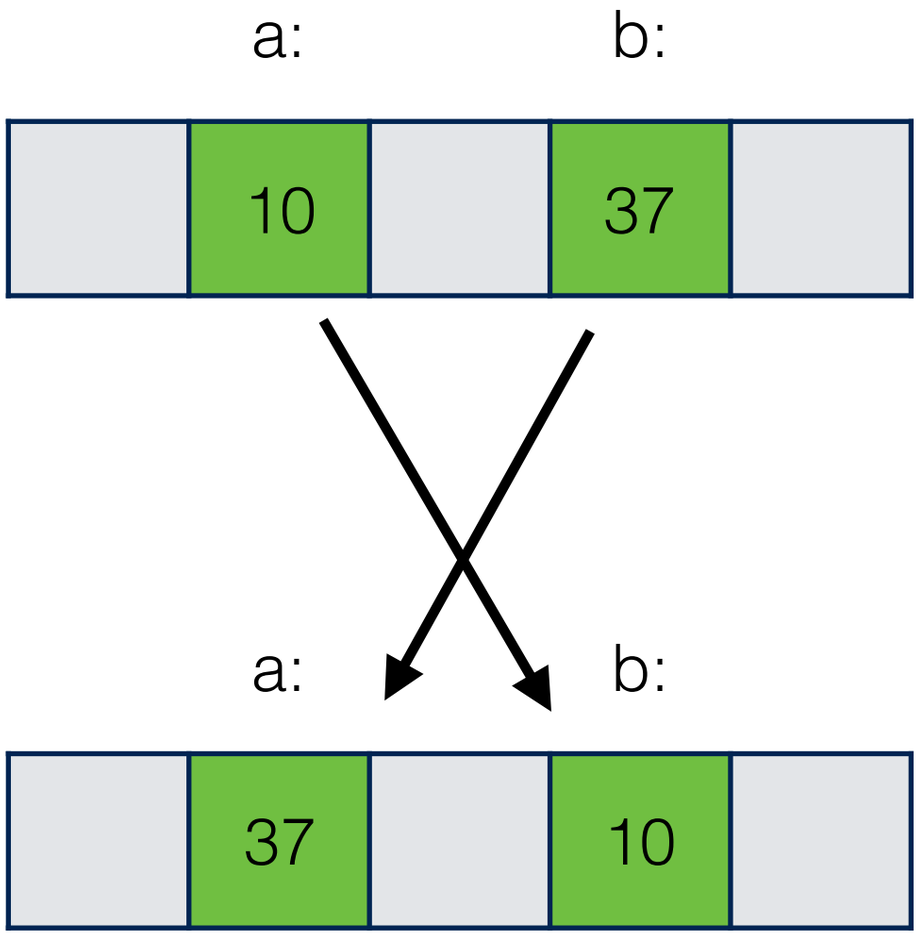
```
int x = 1, y = 2, z[10];
int *ip; /* integer pointer */
ip = &x; /* ip -> x */
y = *ip; /* y = 1 */
*ip =0; /* x = 0 */
ip = &z[0]; /* ip -> z[0] */

*ip = *ip+ 10;
y = *ip + 1;
*ip += 1;
++*ip;
(*ip)++;

int *iq;
iq = ip
```

# Указатели и аргументы функций

- Аргументы передаются по значению — нельзя изменить значение переменной из вызывающей функции

- Однако, если передать указатель на переменную, то ее можно будет изменить

# Функция swap

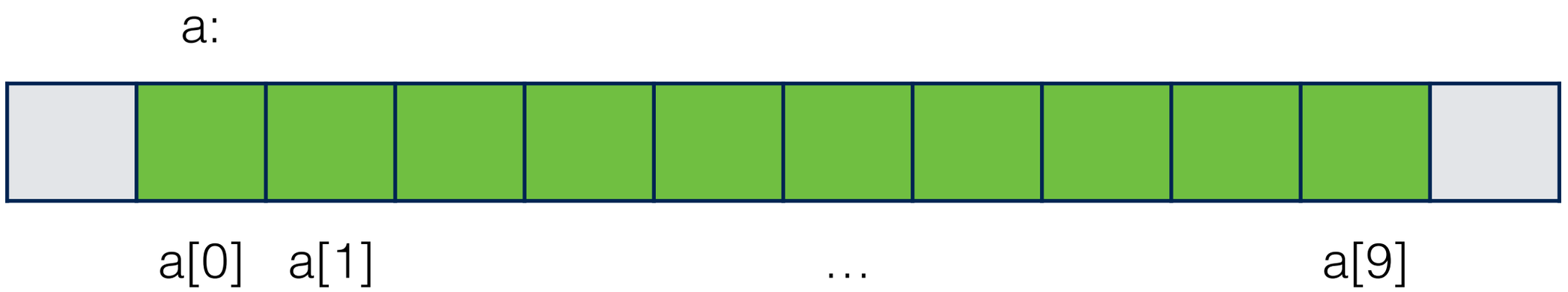



```
void f() {
   int a, b;
   // ...
   swap(&a, &b);
}
```

```
void swap(int *px, int *py) {
     int temp;
     temp = *px;
     *px = *py;
     *py = temp;
}
```

# Массив

```
int a[10];
```



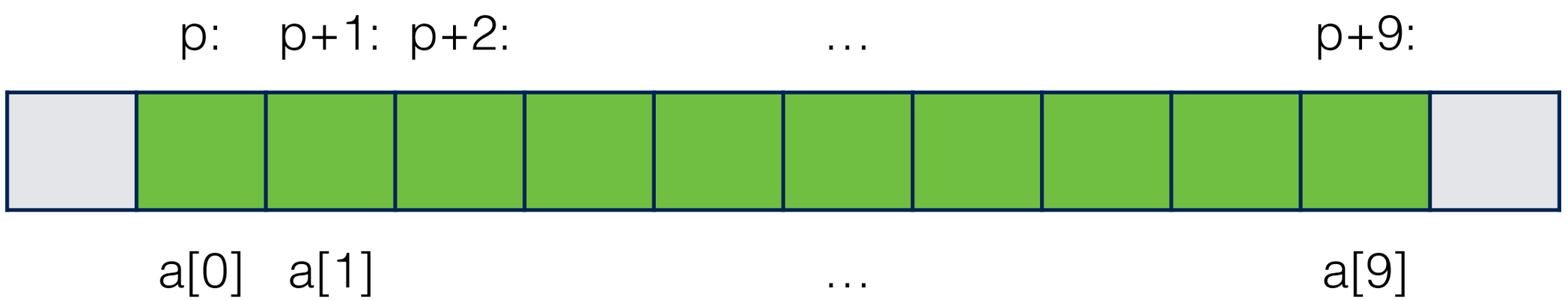
# Массив

- — это блок в памяти из N последовательных элементов одного типа

- Массив `int a[10]` — блок памяти из 10 последовательных целых чисел с именами `a[0], a[1], …, a[9]`

# Массив

```
int a[10];
int *p;

p = &a[0];
```



- Если `p` указывает на какой-либо элемент массива, то по определению:
  - ➡ `p+1` указывает на следующий элемент
  - ➡ `p+i` указывает `i`-й элемент после `p`
  - ➡ `p-i` указывает `i`-й элемент перед `p`
- Если указатель `p` указывает на элемент массива `a[0]`, то:
  - ➡ `p+i` адрес элемента `a[i]`
  - ➡ Значение выражения `*(p+i)` равно `a[i]`
- Это справедливо для массивов любого типа и размера

# Указатели и массивы

- По определению значение переменной или выражения с типом массив является адресом первого элемента массива (указателем на первый элемент массива)

- Если `p` — указатель и `a` — массив, то:

  ➡ Выражения `p=a` и `p=&a[i]` эквивалентны

  ➡ Выражение `a+i` эквивалентно выражению `&a[i]`

  ➡ Выражение `*(a+i)` эквивалентно выражению `a[i]`

  ➡ Компилятор преобразует выражение `a[i]` в выражение `*(a+i)`

  ➡ Выражения `p=a` и `p++` допустимы

  ➡ Выражения `a=p` и `a++` не допустимы

# Длина строки

```
int strlen(char *s) {
    int n;
    for (n = 0; *s != '\0' ; s++)
        n++;
    return n;
}
```

```
int main() {
   char array[100];
   char *ptr;

   // ...

   strlen("Hello, world!");
   strlen(array);
   strlen(ptr);
}
```

# Передача части массива

```
#define A_SIZE 100

// ...

int a[A_SIZE];

// ...

f(int arr[], int n) {...}
f(int *arr, int n) {...}

// ...

f(&a[2], A_SIZE-2);
f(a+2, A_SIZE-2);
```

# Адресная арифметика

| Операция | Пример |
|---|---|
| Присваивание указателей одного типа | `int *p1, *p2, i;`<br>`p1 = &i;`<br>`p2 = p1;` |
| Сложение или вычитание указателя и целого числа | `p++; p--;`<br>`char *s = p + 2;`<br>`s -= s;` |
| Вычитание или сравнение указателей, указывающих на один и тот же массив | `int a[10], *p;`<br>`p = &a[3];`<br>`if (a - p >= 3) {...}` |
| Присваивание NULL (нуль, 0) или сравнение с ним | `int *p = NULL;`<br>`// ...`<br>`if (p != NULL) {...}` |

# Примитивные функции для управления памятью

| Функция | Описание |
| --- | --- |
| `char* alloc(int n)` | Возвращает указатель на `n` последовательно идущих ячеек памяти длиной в 1 символ (байт) |
| `void afree(char* p)` | Освобождает ранее выделенную память |

```c
#define ALLOCSIZE 10000

static char allocbuf[ALLOCSIZE];
static char *allocp = allocbuf;

char *alloc(int n) {
    if (allocbuf + ALLOCSIZE - allocp >= n) {
        allocp += n;
        return allocp - n;
    } else
        return NULL; // 0
}

void afree(char *p) {
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

- `0` или `NULL` никогда не бывает адресом ячейки памяти с данными

- Константа `NULL` определена в `stdio.h`

- Любой указатель можно сравнить с `NULL`

- `Null Pointer Exception (NPE)` и `Segmentation Fault` — самые “популярные” ошибки в программировании

# Длина строки

```
int strlen(char *s) {
    char *p = s;
    while (*p != '\0')
     p++;
    return p - s;
}
```

# Длина строки

- Функция определена в библиотеке string.h
- Сигнатура функции из библиотеки:

```
size_t strlen(const char *str)
```

- Страшный и сложный код из glibc

# Символьные указатели

- Строковые константы — массивы символов, например “`hello, world`”

- Строка всегда заканчивается символ `'\0'`, поэтому в памяти строка на 1 символ длиннее, чем текст в двойных кавычках

- Символ `'\0'`, нужен для того, чтобы функция или программа “знала”, где заканчивается строка в памяти

# Типовой пример

- Функция printf принимает на выход указатель на начало массива символов

- Обращение к строковой константе происходит через указатель на ее первый элемент

```
printf("Hello, world!");
```

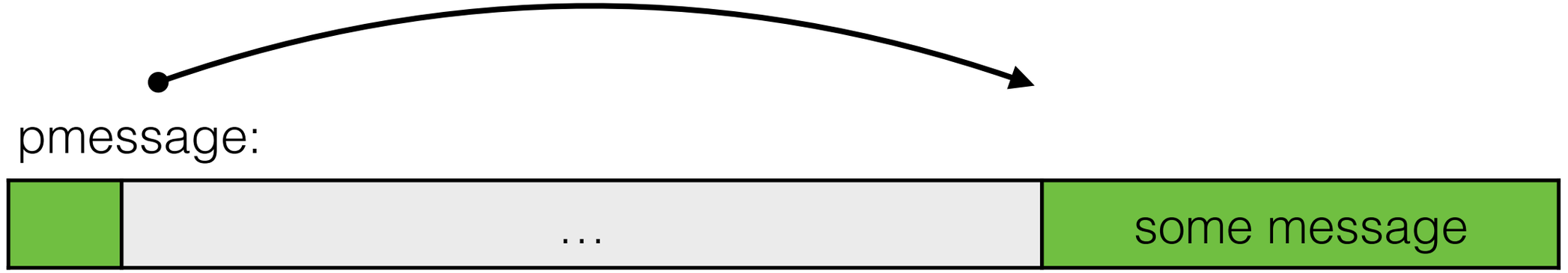
# Еще пример

- Результат выражения в двойных кавычках — указатель на первый символ строки

```
char *pstring;
// ...
pstring = "some text";
```

# И снова пример

```
char amessage[] = "some message";
char *pmessage = "some message";
```



- `amessage` — массив  длиной 13 символов (12 + `'\0'`)

- `pmessage` — указатель на массив из 13 символов

- В `pmessage` можно присвоить другое значение, тогда он будет указывать на другой фрагмент памяти

- `amessage` всегда будет указывать на один и тот же фрагмент памяти

# Копирование строк

```
void strcpy(char *s, char *t) {
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0')
      i++;
}
```

```
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0' ) {
        s++;
      t++;
    }
}
```

```
void strcpy(char *s, char *t) {
    while ((*s++ = *t++) != '\0')
        ;
}
```

```
void strcpy(char *s, char *t) {
    while (*s++ = *t++)
        ;
}
```

# Сравнение строк

```
int strcmp(char *s, char *t) {
    int i;
    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

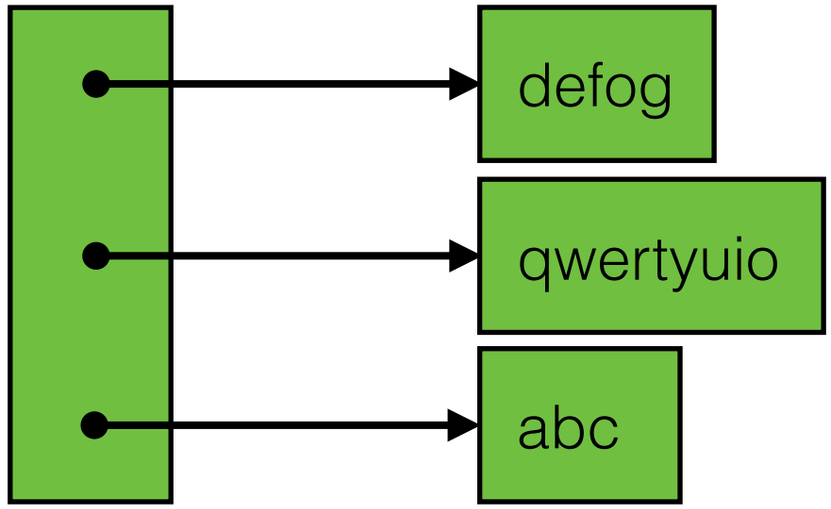
```
int strcmp(char *s, char *t) {
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

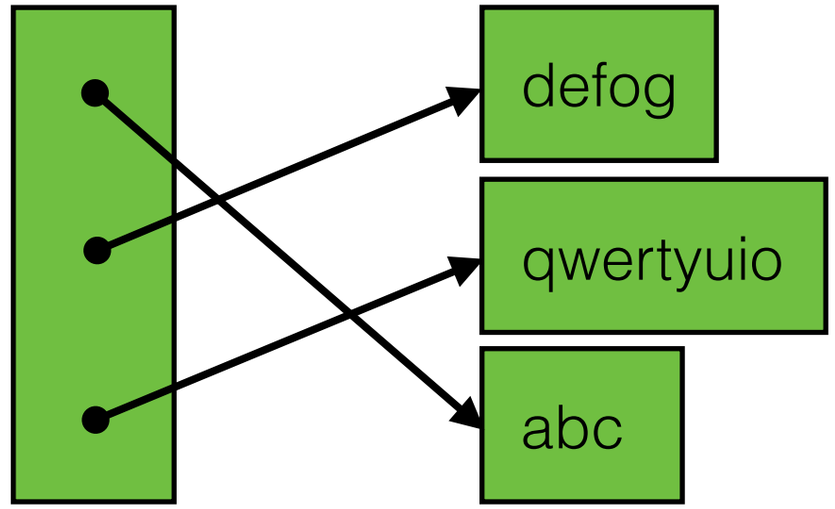
# Массивы указателей и указатели на указатели

# Сортировка строк

```
считать все строки из выходного потока
отсортировать строки
вывести строки в отсортированном порядке
```

# Массив указателей и результат сортировки



```c
#include <stdio.h>
#include <string.h>

#define MAXLINES 5000

char *lineptr[MAXLINES];

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void qsort(char *lineptr[], int left, int right);

int main() {
   int nlines;
   if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
            qsort(lineptr, 0, nlines-1);
            writelines(lineptr, nlines);
            return 0;
        } else {
            printf("error: too many lines to sort\n");
            return 1;
   }
}
```

```c
#include <stdlib.h>

#define MAXLEN 1000

int get_line(char *, int n);

int readlines(char *lineptr[], int maxlines) {
    int len, nlines;
    char *p, line[MAXLEN];
    nlines = 0;
    while ((len = get_line(line, MAXLEN)) > 0)
        if (nlines >= maxlines ||
            (p = (char*) malloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0';
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}
```

```
void writelines(char *lineptr[], int nlines) {
    int i;
    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

```
void writelines(char *lineptr[], int nlines) {
    while (nlines-- > 0)
        printf( "%s\n", *lineptr++);
}
```

```c
void swap(char *v[], int i, int j);

void qsort(char *v[], int left, int right) {
    int i, last;
  if (left >= right)
     return;
  swap(v, left, (left+right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}

void swap(char *v[], int i, int j) {
   char *temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```