

# Практикум на ЭВМ

## Указатели. Динамическое выделение памяти.

Баев А.Ж.

Казахстанский филиал МГУ

03 октября 2018

Строка в языке C — массив из `ascii`-символов, который заканчивается нулевым символом `'\0'`:

```
char s1[] = "math";  
char s2[] = {'m', 'a', 't', 'h', '\0'};  
char s3[] = {109, 97, 116, 104, 0};
```

Константные строки оператором присваивать можно только при инициализации, то есть такая запись будет ошибочной:

```
char s[7];  
s = "math";
```

Работа со строками ничем не отличается от массива с нулевым символом в конце.

# Указатели. Взятие адреса.

& — взятие адреса

```
int a = 5;  
printf("%p\n", &a); //0x10002000
```

# Указатели. Переменная типа адрес.

Размер указателя зависит от ОС: для 32-битной системы будет равен 4 байта, для 64-битной — 8 байт.

```
int *int_pointer;  
char *string;  
void *ptr;
```

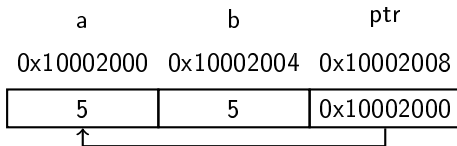
Указатель

```
int a = 5;  
int *ptr = &a;
```

имя	a	ptr
адрес	0x10002000	0x100020004
значение	5	0x10002000

# Указатели. «Косвенное» обращение к переменной

```
int a = 5, b;  
int *ptr = &a; //ptr = 0x10002000  
b = *ptr;      //b = 5
```

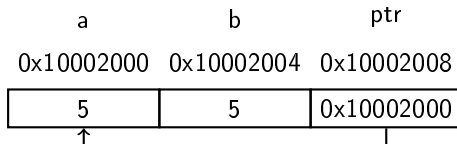


# Указатели. Зачем нужно указывать тип указателя №1

```
char a = '+';  
char *ptr = &a;  
size_t n = sizeof(*ptr);  
size_t k = sizeof(ptr);
```

# Указатели. «Косвенное» изменение переменной

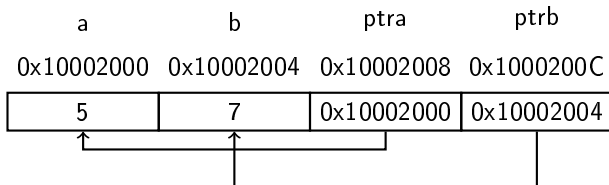
```
int a = 5;  
int *ptr = &a; //ptr = 0x10002000  
*ptr = b;      //a = 5
```



# Указатели. Приоритет

Приоритет выше арифметических операторов:

```
int a = 5;  
int b = 7;  
int *ptra = &a, *ptrb = &b;  
*ptra = *ptra + *ptrb;      // a = a + b  
printf("%d %d\n", a, b);    // 12 7
```





# Указатели. Приоритет

Приоритет ниже инкремента.

```
int a = 5;  
int *ptr = &a;
```

Что будет?

```
(*ptr)++;
```

```
*(ptr++);
```

```
*ptr++;
```

# Указатели. Самая любимая ошибка

Разыменовывать можно только указатели, в которых хранятся адреса существующей памяти (переменных).

```
int *ptr;  
*ptr = 5;
```

Что будет?

# Указатели. Два золотых правила работы с указателями

1) Инициализировать указатели сразу при создании. Если необходимо объявить указатель, который пока никуда не указывает, его следует присвоить нулевому указателю с помощью специальной константы `NULL`.

```
int *ptr = NULL;
```

2) перед попыткой его разыменовать, проверять можно ли так делать.

```
int a = 5;
int *ptr = NULL;
if (ptr != NULL)                // False
    printf("%d\n", *ptr);
ptr = &a;
if (ptr != NULL)                // True
    printf("%d\n", *ptr);
```

# Указатели. Два почти тождества

- $*(&a) == a$  — верно всегда;
- $&(*a) == a$  — верно, только если  $a$  — указатель на существующую переменную.

## Указатели. Зачем нужно указывать тип указателя №2

Смещение на соседние ячейки:

```
int a = 10;  
int *ptr = &a;  
ptr++;  
--ptr;  
ptr += 4;  
ptr = ptr - 4;
```

Величина сдвига определяется из типа указателя (т.е. на sizeof(тип))

Пример на массивах.

```
char s[] = "math";  
char *ptr = s;
```

Сколько здесь использовано памяти? Где она находится?

# Указатели на массивах

```
*ptr = 'p';           // 'p', 'a', 't', 'h', '\0'
```

Чтобы изменить

```
*(ptr + 2) = 'l'; // 'p', 'a', 'l', 'h', '\0'  
*(ptr + 3) = 'm'; // 'p', 'a', 'l', 'm', '\0'
```

## Указатели. Еще одно тождество

В общем виде для массива `a[]` верно следующее соотношение:

```
a[i] == *(a + i);
```

Данное соотношение приводит еще к одному интересному соотношению:

```
&a[i] == &*(a + i) == a + i;
```



# Указатели. Полезное применение

Для считывания массива:

```
int a[5], i;  
for (i = 0; i < 5; i++) {  
    scanf("%d", a + i);  
}
```

С помощью указателя можно распечатать строку без индексирования строки как массива, передвигая указатель:

```
char s[5] = "math";  
char *ptr;  
for (ptr = s; *ptr; ptr++) {  
    putchar(*ptr);  
}
```

```
a[i] == i[a];
```

Как следствие

```
char a[] = "math";  
2[a] = 's';  
puts(a);
```

# Динамическая память.

Статическая память — стек.

Динамическая память — куча.

# Динамическая память.

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

Выделение непрерывной области памяти заданного размера  $n$  байт:

```
void *malloc(size_t n)
```

Освобождение памяти:

```
void free(void *ptr);
```

Золотое правило — кто создает, тот и чистит.

# Динамическая память. Один байт.

```
char *ptr;  
ptr = malloc(1);
```

## Статическая

Переменная

Адрес

Память

ptr			
0x10002000	0x10002001	0x10002002	0x10002003
0x70000010			

## Динамическая

Адрес

Память

0x70000010
?

# Динамическая память. Много байт.

```
int *ptr = malloc(10 * sizeof(int));  
ptr[8] = 8;  
*(ptr + 9) = 9;
```

# Динамическая память. Плоский код

Сгенерировать массив из 10 целых чисел от 1 до 10.

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

int main() {
    int n = 10, i;
    int *ptr = malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        ptr[i] = i + 1;
    for (i = 0; i < n; i++)
        printf("%d_", ptr[i]);
    free(ptr)
    return 0;
}
```

# Динамическая память. Процедурный код

```
int *generate(int n) {
    int i, *ptr = malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        ptr[i] = i;
    return ptr;
}

void print(int n, int *ptr) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", ptr[i]);
}

int main() {
    int *ptr = generate(5);
    print(5, ptr);
    free(ptr);
    return 0;
}
```



# Динамическая память. Нужно больше функций

```
void *calloc(size_t n, size_t size);
```

Выделяет массив из  $n$  элементов, каждый из которых имеет размер  $size$  байт (всего  $n*size$  байт).

```
void *realloc(void *ptr, size_t size);
```

- 1 если можно, то расширить массив, иначе выделить новую память размером  $size$ ;
- 2 при необходимости скопировать элементы из исходного массива во второй (из памяти под указателем  $ptr$  в память под указателем  $new\_ptr$ );
- 3 очистить исходную память (под указателем  $ptr$ ).

# Динамическая память. Скорость работы.

Работает быстрее чем вы думаете! Почему?

```
char *ptr = NULL;
for (size_t n = 1; n <= 1000000; n++) {
    ptr = realloc(ptr, n);
}
```

# Динамическая память vs статическая память.

1. именованные переменные.
2. время жизни.
3. место для памяти.
4. возможность узнать размера массива.
5. размер массива:  $n * \text{sizeof}(\text{type})$  и  $n * \text{sizeof}(\text{type}) + \text{sizeof}(\text{type}^*)$ .
6. представление матриц.

## Пример. Простые числа.

Дано целое число  $n$  от 1 до 100. В динамический массив записать все простые числа, которые не превосходят  $n$ . Вывести количество простых чисел и сами числа по возрастанию.

Ввод	8	2
Вывод	4 2 3 5 7	0

## Пример. Простые числа.

```
int main() {
    int* ans = NULL;
    int m = 0, i, n;
    scanf("%d", &n);
    for (i = 2; i <= n; i++)
        if (isprime(i)) {
            ans = realloc(ans, (m + 1) * sizeof(int));
            ans[m] = i;
            m++;
        }
    printf("%d\n", m);
    for (i = 0; i < m; i++)
        printf("%d_", ans[i]);
    putchar('\n');

    if (ans != NULL)
        free(ans);
    return 0;
}
```

## Пример. Фильтрация массива.

Дано целое положительное  $n$ . Далее  $n$  целых чисел. Вывести только четные числа.

Ввод	3 14 11 12
Вывод	14 12

Описать функции без возвращаемого значения.

```
int main() {
    int *numbers, *even_numbers;
    int number_size, even_size;
    scan_array(&number_size, &numbers);
    even_filter(number_size, numbers,
                &even_size, &even_numbers);
    free(numbers);
    print_array(even_size, even_numbers);
    if (even_numbers != NULL)
        free(even_numbers);
    return 0;
}
```

## Пример. Считать массив.

```
void scan_array(int *size_ptr, int **array_ptr) {  
    int *array = NULL, size, i;  
    scanf("%d", &size);  
    array = malloc(size * sizeof(int));  
    for (i = 0; i < size; i++)  
        scanf("%d", &array[i]);  
  
    *size_ptr = size;  
    *array_ptr = array;  
}
```

# Пример. Считать массив.

**scan\_array()**

Переменная

Значение

size_ptr	array_ptr	size	array
0x10002000	0x10002004	3	0x70000010

**main()**

Переменная

Адрес

Значение

number_size	numbers
0x10002000	0x10002004
?	?

**Динамическая**

Адрес

Память

0x70000010	0x70000014	0x70000018
14	11	12



## Пример. Отфильтровать массив.

```
void even_filter(int src_size, int * src_array,
                int * dst_size_ptr, int ** dst_array_ptr)
{
    int i;
    int *array = NULL;
    int size = 0;
    for (i = 0; i < src_size; i++)
        if (src_array[i] % 2 == 0) {
            array = (int*) realloc(array, (size + 1) * sizeof(int));
            array[size] = src_array[i];
            size++;
        }
    *dst_size_ptr = size;
    *dst_array_ptr = array;
    return;
}
```

## Пример. Полезный

Дана строка из слов. Слова разделены пробелами, каждое слово состоит из печатных символов, отличных от пробела, табуляции и переноса строки. Считать каждое слово в динамический массив. Вывести слова.

```
char *get_word(char *last_char_ptr);
```

## Пример. Полезный

```
char *get_word(char *last_char_ptr) {
    char delimiter = ' ', final = '\n';
    char *word = NULL, ch;
    int n = 0;

    ch = getchar();
    while (ch != delimiter && ch != final) {
        word = realloc(word, (n + 1) * sizeof(char));
        word[n] = ch;
        n++;
        ch = getchar();
    }
    word = realloc(word, (n + 1) * sizeof(char));
    word[n] = '\0';
    *last_char_ptr = ch;
    return word;
}
```

## Пример. Полезный

```
int main() {
    char last_char = '\0';
    char *word = NULL;
    word = get_word(&last_char);
    while (last_char != '\n') {
        puts(word);
        free(word);
        word = get_word(&last_char);
    }
    return 0;
}
```

# Динамическая память. Еще раз об ошибке сегментации

```
int *ptr = malloc(10);  
ptr[9] = 0;
```

Может и не произойти. Массивы выделяются со смещением по степеням двойки.

# Динамическая память. Еще раз об ошибке сегментации

```
int *ptr = malloc(10);  
ptr[9] = 0;
```

Может и не произойти. Массивы выделяются со смещением по степеням двойки. Средство борьбы — санитайзеры.

# Динамическая память. Совсем плохо

```
int a = 5, b = 7;  
int *ptr = &a;  
*(ptr + 1) = 6
```

# Функции работы со строками

## Библиотека «stdio.h»

```
int printf("%s", str);  
int puts(const char *s);  
int fputs(const char *s, FILE *stream);
```

Введем "the math"

```
int scanf("%s", s); //unsave  
t h e \0 ? ? ? ? ?  
  
char *gets(char *s); //unsave  
t h e m a t h \0 ?  
  
char *fgets(char *s, int size, FILE *stream);  
t h e m a t h \n \0
```

Функции парные.



# Функции работы со строками

## Библиотека «ctype.h»

```
int isdigit(int c);  
int isalpha(int c);  
int islower(int c);  
int isupper(int c);  
int isalnum(int c);  
int isgraph(int c);  
int isprint(int c);  
int ispunct(int c);  
int isspace(int c);  
int tolower(int c);  
int toupper(int c);
```

# Функции работы со строками

## Библиотека «string.h»

```
size_t strlen(const char *s);
int strcmp(const char *s1,
           const char *s2);
int strncmp(const char *s1,
            const char *s2,
            size_t n);
char *strcpy(char *dest,
             const char *src);
char *strncpy(char *dest,
              const char *src,
              size_t n);
char *strstr(const char *haystack,
             const char *needle);
void *memset(void *s, int c, size_t n);
```

# Функции работы со строками

Что будет напечатано?

```
char a[] = "mechanics";
char b[] = "mathematics";
puts(strncpy(strstr(a, "a"), b, 4));
puts(a);
puts(b);
if (strcmp(a + 4, b) > 0)
    puts("OK")
else
    puts("NO");
```

# Отладка в gdb

Компилируем

```
gcc prog.c -o prog -Wall -Werror -lm -g
```

Запускаем отладчик

```
gdb prog
```

Посмотрим код

```
list 1
```

Ставим точку останова (до которой программа будет выполняться в обычном режиме). Лучше ставить сразу после ввода.

```
break 6
```

Запускаем

```
run
```

Добавляем переменную наблюдения (можно несколько переменных)

```
display n  
display ans
```

Делаем построчное выполнение (первый раз надо набрать команду целиком, потом просто Enter).

```
next
```

Выход

```
quit
```

# Динамическая память. Санитайзер

```
-fsanitize=address,undefined  
-fno-sanitize-recover=all  
-fsanitize-undefined-trap-on-error
```

## Makefile

```
CC=gcc  
CFLAGS=-Wall -Werror -g -fmax-errors=2 \  
-fsanitize=address,undefined \  
-fno-sanitize-recover=all \  
-fsanitize-undefined-trap-on-error  
  
all:  
    echo "run: _make_A"  
  
%: %.c ~/cstyle.py  
    $(CC) $@.c -o $@ $(CFLAGS)  
    ~/cstyle.py $@.c
```