

# Практикум на ЭВМ

## Указатели. Динамическое выделение памяти.

Баев А.Ж.

Казахстанский филиал МГУ

24 сентября 2019

# Строки

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

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

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

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

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

## Указатели. Взятие адреса.

& — взятие адреса

```
1  int a = 5;  
2  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.

```
1  int *int_pointer;  
2  char *string;  
3  void *ptr;
```

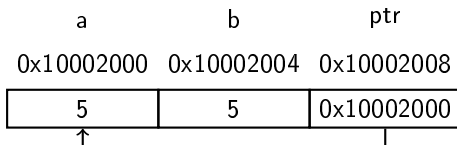
Указатель

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

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

## Указатели. «Косвенное» обращение к переменной

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

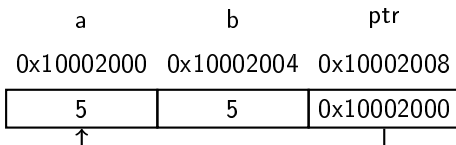


## Указатели. Зачем нужно указывать тип указателя №1

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

## Указатели. «Косвенное» изменение переменной

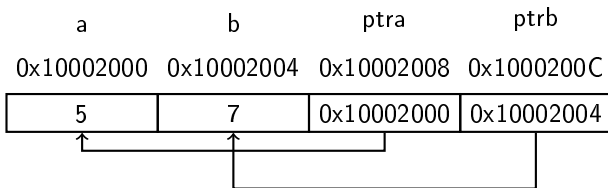
```
1  int a = 5;  
2  int *ptr = &a;    //ptr = 0x10002000  
3  *ptr = b;         //a = 5
```



## Указатели. Приоритет

Приоритет выше арифметических операторов:

```
1   int a = 5;  
2   int b = 7;  
3   int *ptrb = &a, *ptrb = &b;  
4   *ptrb = *ptrb + *ptrb;      // a = a + b  
5   printf("%d_ %d\n", a, b);  // 12 7
```





# Указатели. Приоритет

Приоритет ниже инкремента.

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

Что будет?

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

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

```
1   *ptr++;
```

## Указатели. Самая любимая ошибка

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

```
1  int *ptr;  
2  *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`.

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

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

```
1      int a = 5;
2      int *ptr = NULL;
3      if (ptr != NULL)           // False
4          printf("%d\n", *ptr);
5      ptr = &a;
6      if (ptr != NULL)           // True
7          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:

```
1      int a = 10;  
2      int *ptr = &a;  
3      ptr++;  
4      --ptr;  
5      ptr += 4;  
6      ptr = ptr - 4;
```

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

# Указатели на массивы

Пример на массивах.

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

Сколько здесь использовано памяти? Где она находится?

## Указатели на массивах

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

Чтобы изменить

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

## Указатели. Еще одно тождество

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

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

Данное соотношение приводит еще к одному интересному соотношению:

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



## Указатели. Полезное применение

Для считывания массива:

```
1  int a[5], i;  
2  for (i = 0; i < 5; i++) {  
3      scanf("%d", a + i);  
4  }
```

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

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

## Указатели. Бесполезное применение

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

Как следствие

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

# Динамическая память.

Статическая память — стек.

Динамическая память — куча.

## Динамическая память.

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

Выделение непрерывной области памяти заданного размера  $n$  байт:

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

Освобождение памяти:

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

Золотое правило — кто создает, тот и чистит.

## Динамическая память. Один байт.

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

### Статическая

Переменная

Адрес

Память

ptr			
0x10002000	0x10002001	0x10002002	0x10002003
0x70000010			

### Динамическая

Адрес

Память

0x70000010
?

## Динамическая память. Много байт.

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

## Динамическая память. Плоский код

Сгенерировать массив из 10 целых чисел от 1 до 10.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int n = 10, i;
6      int *ptr = malloc(n * sizeof(int));
7      for (i = 0; i < n; i++)
8          ptr[i] = i + 1;
9      for (i = 0; i < n; i++)
10         printf("%d ", ptr[i]);
11     free(ptr)
12     return 0;
13 }
```

## Динамическая память. Процедурный код

```
1  int *generate(int n) {
2      int i, *ptr = malloc(n * sizeof(int));
3      for (i = 0; i < n; i++)
4          ptr[i] = i;
5      return ptr;
6  }
7  void print(int n, int *ptr) {
8      int i;
9      for (i = 0; i < n; i++)
10         printf("%d_", ptr[i]);
11 }
12 int main() {
13     int *ptr = generate(5);
14     print(5, ptr);
15     free(ptr);
16     return 0;
17 }
```



## Динамическая память. Нужно больше функций

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

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

```
1 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?

```
1     char *ptr = NULL;  
2     for (size_t n = 1; n <= 1000000; n++) {  
3         ptr = realloc(ptr, n);  
4     }
```

# 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.

```
1  int main() {
2      int* ans = NULL;
3      int m = 0, i, n, size;
4      scanf("%d", &n);
5      for (i = 2; i <= n; i++)
6          if (isprime(i)) {
7              size = (m + 1) * sizeof(int);
8              ans = realloc(ans, size);
9              ans[m] = i;
10             m++;
11         }
12     printf("%d\n", m);
13     for (i = 0; i < m; i++)
14         printf("%d_", ans[i]);
15     putchar('\n');
16
17     if (ans != NULL)
18         free(ans);
19     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.

```
1  int main() {
2      int *numbers, *even_numbers;
3      int number_size, even_size;
4      scan_array(&number_size, &numbers);
5      even_filter(number_size, numbers,
6                  &even_size, &even_numbers);
7      free(numbers);
8      print_array(even_size, even_numbers);
9      if (even_numbers != NULL)
10         free(even_numbers);
11     return 0;
12 }
```

## Пример. Считать массив.

```
1 void scan_array(int *size_ptr, int **array_ptr) {  
2     int *array = NULL, size, i;  
3     scanf("%d", &size);  
4     array = malloc(size * sizeof(int));  
5     for (i = 0; i < size; i++)  
6         scanf("%d", &array[i]);  
7  
8     *size_ptr = size;  
9     *array_ptr = array;  
10 }
```

## 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.

```
1 void even_filter(int src_size,
2                 int *src_array,
3                 int *dst_size_ptr,
4                 int **dst_array_ptr) {
5     int i;
6     int *array = NULL;
7     int size = 0, bytes;
8     for (i = 0; i < src_size; i++)
9         if (src_array[i] % 2 == 0) {
10             bytes = (size + 1) * sizeof(int);
11             array = realloc(array, bytes);
12             array[size] = src_array[i];
13             size++;
14         }
15     *dst_size_ptr = size;
16     *dst_array_ptr = array;
17     return;
18 }
```

## Пример. Полезный

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

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

## Пример. Полезный

```
1 char *get_word(char *last_char_ptr) {
2     char delimiter = ' ', final = '\n';
3     char *word = NULL, ch;
4     int n = 0, bytes;
5
6     ch = getchar();
7     while (ch != delimiter && ch != final) {
8         bytes = (n + 1) * sizeof(char);
9         word = realloc(word, bytes);
10        word[n] = ch;
11        n++;
12        ch = getchar();
13    }
14    bytes = (n + 1) * sizeof(char);
15    word = realloc(word, bytes);
16    word[n] = '\0';
17    *last_char_ptr = ch;
18    return word;
19 }
```

## Пример. Полезный

```
1  int main() {
2      char last_char = '\0';
3      char *word = NULL;
4      word = get_word(&last_char);
5      while (last_char != '\n') {
6          puts(word);
7          free(word);
8          word = get_word(&last_char);
9      }
10     return 0;
11 }
```

## Динамическая память. Еще раз об ошибке сегментации

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

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

## Динамическая память. Еще раз об ошибке сегментации

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

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

## Динамическая память. Совсем плохо

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

# Функции работы со строками

Библиотека «stdio.h»

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

Введем "the math"

```
1   int scanf("%s", s); //unsaved
2   t h e \0 ? ? ? ? ?
3
4   char *gets(char *s); //unsaved
5   t h e m a t h \0 ?
6
7   char *fgets(char *s, int size, FILE *stream);
8   t h e m a t h \n \0
```

Функции парные.



# Функции работы со строками

## Библиотека «ctype.h»

```
1      int isdigit(int c);  
2      int isalpha(int c);  
3      int islower(int c);  
4      int isupper(int c);  
5      int isalnum(int c);  
6      int isgraph(int c);  
7      int isprint(int c);  
8      int ispunct(int c);  
9      int isspace(int c);  
10     int tolower(int c);  
11     int toupper(int c);
```

# Функции работы со строками

## Библиотека «string.h»

```
1    size_t strlen(const char *s);
2    int strcmp(const char *s1,
3              const char *s2);
4    int strncmp(const char *s1,
5               const char *s2,
6               size_t n);
7    char *strcpy(char *dest,
8                 const char *src);
9    char *strncpy(char *dest,
10                 const char *src,
11                 size_t n);
12    char *strstr(const char *haystack,
13                 const char *needle);
14    void *memset(void *s, int c, size_t n);
```

# Функции работы со строками

Что будет напечатано?

```
1  char a[] = "mechanics";  
2  char b[] = "mathematics";  
3  puts(strncpy(strstr(a, "a"), b, 4));  
4  puts(a);  
5  puts(b);  
6  if (strcmp(a + 4, b) > 0)  
7      puts("OK")  
8  else  
9      puts("NO");
```

# Отладка в gdb

Компилируем

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

Запускаем отладчик

```
1 gdb prog
```

Посмотрим код

```
1 list 1
```

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

```
1 break 6
```

Запускаем

```
1 run
```

Добавляем переменную наблюдения (можно несколько переменных)

```
1 display n  
2 display ans
```

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

```
1 next
```

Выход

```
1 quit
```

## Динамическая память. Санитайзер

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

### Makefile

```
1 CC=gcc
2 CFLAGS=-Wall -Werror -g -fmax-errors=2 \
3 -fsanitize=address,undefined \
4 -fno-sanitize-recover=all \
5 -fsanitize-undefined-trap-on-error
6
7 all:
8     echo "run: make A"
9
10 %: %.c ~/cstyle.py
11     $(CC) $@.c -o $@ $(CFLAGS)
12     ~/cstyle.py $@.c
```