

# Введение в язык программирования С.

## Лекция 6

Память. Указатели. Динамическая память.

Сергей Леонидович Бабичев

# Указатели и массивы

- Настало время подробнее понять, что происходит, когда мы пишем, например, `int a[100];`.
- Мы называли такие массивы *простыми*.
- Под него выделяется память на момент его создания.
- Оказывается, имя массива (в данном случае `a`) не что иное, как указатель на выделенную системой память.
- Так `a` — массив или указатель?
- Если `a` — указатель, то как с ним работать?
- Что у нас имеется кроме операции `*` для обращения к той памяти, на которую указатель показывает?
- В обращении к элементам массива нам помогает *адресная арифметика*.

# Адресная арифметика

- Пусть начальным адресом нашего массива  $a$  будет 10000.
- Тогда обращение  $*a$  разрешено и это будет нулевым элементом массива.
- К нулевому элементу массива можно обратиться и как  $a[0]$ .
- Давайте запишем первое выражение немного по-другому:  $*(a+0)$ . Тогда  $a[0]$  есть синоним к  $*(a+0)$ .
- Память под массивы гарантированно выделяется *непрерывным куском*, поэтому элемент под номером 1,  $a[1]$  будет находиться по адресу 10004, если размер одного элемента типа `int` равен четырём.
- $a[1]$  есть синоним к  $*(a+1)$ ,  $a[i]$  — для  $*(a+i)$ .
- Арифметика становится не совсем обычной: к указателю, равному 10000 прибавили единицу и он стал равен 10004.
- Для всех ли типов данных подобное верно?
- Для всех с одной поправкой: степень увеличения адреса зависит от размера элемента массива.

# Адресная арифметика

Указатели допускают следующие операции:

- сложение указателя с целым  $\text{ptr} + i$ . Результат показывает на элемент того же типа, отстоящий на  $i$  элементов дальше  $\text{ptr}$ ;
- вычитание из указателя целого  $\text{ptr} - i$ . Результат показывает на элемент того же типа, находящийся за  $i$  элементов перед  $\text{ptr}$ ;
- вычитание указателей одного типа  $\text{ptr2} - \text{ptr1}$ . Результат — количество элементов этого типа между адресами.

# Массивы и функции

- Можно ли передавать массивы в функции?
- Да, конечно!
- Для совсем наивных можно описывать аргументы функций как массивы:

```
int sum_elems(int array[100]) {  
    int i, sum = 0;  
    for (i = 0; i < 100; i++) {  
        sum += array[i];  
    }  
    return sum;  
}
```

# Массивы и функции

- Почему «для наивных»?
- Потому, что в Си массив как целое передать в функцию невозможно.
- Имя массива — просто указатель на нечто, который нельзя изменить.
- Пока мы в области определения простого массива, видимости точки его создания, компилятор знает его размер.
- После того, как массив, точнее, указатель на него, переданы в функцию, никто внутри функции не способен определить его размер.

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

```
void bar(int b[100]) {  
    int sizebar = sizeof b / sizeof b[0];  
    printf("sizebar = %d\n", sizebar);  
}
```

```
void func() {  
    int arr[100];  
    int sizearr = sizeof arr / sizeof arr[0];  
    printf("sizearr = %d\n", sizearr);  
    bar(arr);  
}
```

```
int main() {  
    func();  
}
```

```
sizearr = 100  
sizebar = 2
```

# Массивы и функции

- В функцию передан указатель на массив, размер указателя здесь 8 байт, размер типа `int` здесь четырем байтам, отсюда `sizeof = 2`.
- Обман? Зачем всё это?
- Чтобы не отпугнуть тех, кто хочет мигрировать с FORTRAN на Си.
- Функция не может отличить указатель на отдельную переменную от указателя на кусок памяти (массив).
- Число в квадратных скобках, которое, вроде бы, должно показать нам количество элементов массива, для одномерных массивов игнорируется.
- Следующие три объявления функции эквивалентны:

```
int add_elems(int arr[10000]);  
int add_elems(int arr[]);  
int add_elems(int *arr);
```



# Массивы и функции

- Теперь у нас есть универсальная функция, которая готова вычислить сумму элементов любого целочисленного массива любой длины.
- Нужно только, чтобы она знала количество элементов.
- В классическом Pascal такую функцию написать было невозможно.
- Математикам такие функции позарез необходимы, для линейной алгебры, для векторов и матриц.

# Массивы и функции

- Наша функция `add_elems` пока не знает количество элементов массива для суммирования.
- Нужно добавить в неё ещё один аргумент.
- Обычно в Си его добавляют после имени массива.
- А какой у этого аргумента тип?
- Это тот тип, который может содержать любые размеры для массивов.
- Это не `int`, появилось бы ограничение на размеры массивов.
- В современных компьютерах памяти много и массив может содержать триллионы элементов.
- Для размеров массивов и индексов лучше применять тип `size_t`.
- Он описан в файле `<stddef.h>`, или `<stdlib.h>` или `<stdio.h>`
- Рядом имеется тип `ptrdiff_t`, который может иметь знак, в отличие от `size_t`.

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

void add_elems(int *b, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; i++) {
        sum += b[i];
    }
    return sum;
}

int main() {
    int arr[] = {3, 1, 4, 1, 5, 9, 2, 6};
    int sum_arr = add_elems(arr, sizeof arr / sizeof arr[0]);
}
```

# Инициализация простых массивов

Простой массив можно объявить с его инициализацией. Его размер будет известен при компиляции по количеству инициализаторов.

```
int x[] = {1,2,3,4,5,6,7,8,9,10}; // OK
int y[10] = {1,2,3,4,5,6,7,8,9,10}; // OK
int z[20] = {1}; // OK, z[1..19] = 0
int t[5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Error.
```

```
#include <stdio.h>
int sum_matrix(int m[3][3]) {
    int sum = 0;
    for (size_t i = 0; i < 3; i++) {
        for (size_t j = 0; j < 3; j++) {
            sum += m[i][j];
        }
    }
    printf("sizeof m=%zu\n", sizeof m);
    return sum;
}
int main() {
    int matrix[3][3];
    for (size_t i = 0; i < 3; i++) {
        for (size_t j = 0; j < 3; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
    int s = sum_matrix(matrix);
    printf("sizeof matrix=%zu sum=%d\n", sizeof matrix, s);
}
```

# Простые двумерные массивы

- Размер `m` в `sum_matrix` равен 8, а `matrix` в `main` равен 36.
- Таким образом, `m` — указатель.
- Но попытка передать его явным образом

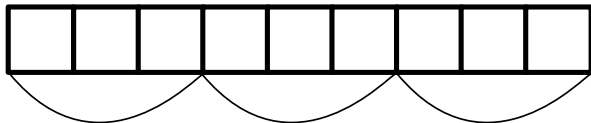
```
int sum_matrix(int *m) { ...
```

даст ошибку: что тогда означает `m[i][j]`, если `m[i]` — `int`?

- Приходится сохранять *двумерность* синтаксически.
  - Можно писать так:
- ```
int sum_matrix(int m[][3]) { ...
```
- Это читается так: мы будем использовать двойную индексацию и строка состоит из трёх элементов.

## Простые двумерные массивы

m



$m[0][0..2]$   $m[1][0..2]$   $m[2][0..2]$

- Теперь компилятор знает, как получить доступ к нужному элементу.
- Это позволяет вычислять адрес любого из элементов массива  $m[i][j]$  по формуле  $m+i*3+j$ . Фактически компилятор превращает нашу функцию в

```
int sum_matrix(int *m) {  
    int sum = 0;  
    for (size_t i = 0; i < 3; i++) {  
        for (size_t j = 0; j < 3; j++) {  
            sum += m[i*3+j];  
        }  
    }  
    return sum;  
}
```

# Простые многомерные массивы

Простые многомерные массивы хранятся в Си в *линеаризованном виде*, строка за строкой. В функцию передаётся адрес занятой памяти. Разрешается (только в аргументах функции!) оставить пустые [] для самого левого измерения массива, например, вместо `double x[5][5][5]` писать `double x[][5][5]`.  
Заметьте, что принимаемые в функции двумерные и многомерные массивы должны содержать *константные выражения* в каждой из квадратных скобок.



# Куча

- Пока мы знаем массивы с временем жизни, равным времени жизни всей программы (статические и глобальные), или время жизни, равное времени жизни блока, а котором они описаны (автоматические или стековые).
- Можно создавать и такие объекты, которые не существуют при входе в функцию и сохраняются при выходе из неё.
- Это не статические массивы — они существуют и *до* входа в функцию.
- Это не автоматические массивы тоже не подходят — они уничтожаются *после* выхода из функции.
- Это — *динамические* массивы, которые создаются в области памяти под названием *куча* или по английски *heap*.

## Куча. Тип `void *`

- Запрос памяти из кучи ещё называют «заказ памяти»).
- Функция должна где-то выделить нужное место и вернуть на него указатель.
- Специальный тип указателя, который может хранить любой адрес — `void *`.
- `int *` может оказаться или адресом отдельной переменной, или массивом типа `int`, адрес чего даёт указатель `void *`?
- Массивов и переменных такого типа не бывает. Нельзя написать:

```
int f(void *arg) {  
    return arg[0];  
}
```

Ценность этого указателя в том, что указатель такого типа можно присвоить указателю нужного типа — и далее работать с тем, как обычно.

```
int f(void *arg) {  
    int *a = arg;  
    return a;  
}
```

## Функции, выделяющие память

- Прототипы таких функций находятся в `<stdlib.h>`.
- Самая простая функция — `malloc`.
- Один аргумент — количество запрашиваемых байтов.
- Возвращаемое значение — указатель на выделенную память.
- При неуспехе вернётся `NULL`.

```
int *a = (int *)malloc(n * sizeof(int));
```

- Теперь указатель `a` хранит адрес места, где можно разместить `n` переменных типа `int`, что соответствует массиву из `n` элементов `int`.

## Выделение памяти — ответственность

- Тот, кто получил этот указатель, несёт ответственность за жизнь этого указателя — этот указатель — единственный способ обратиться к памяти из кучи.
- Потеряем указатель — потеряем эту память.
- Возникнет ситуация, известная, как *утечка памяти* или *memory leak*.
- Она крайне неприятна и в ряде случаев приводит к ситуациям, когда программа становится неработоспособной.
- Потерять память — проще простого:

```
int foo(int n) {  
    int *bar = (int *)malloc(n * sizeof (int));  
    return 0;  
}
```

- Переменная `bar`, содержащая адрес свежewedенного системой куска памяти, после достижения закрывающей фигурной скобки исчезает.
- Вся выделенная память внезапно становится недоступной. Она как бы есть, но её как бы и нет.

# Выделение памяти и её освобождение

- Освободить память, то есть вернуть её в систему, можно функцией `free`.

```
int foo(int n) {  
    int *bar = (int *)malloc(n * sizeof (int));  
    // ....  
    free(bar);  
    return 0;  
}
```

В корректных программах каждый заказ памяти по `malloc/calloc/realloc` должен иметь парный вызов `free`.

## Другие способы выделения памяти

- Память, выделенная `malloc`, неинициализирована. Чтобы получить инициализированный нулями кусок памяти, используйте `calloc`. Он требует два аргумента: количество заказываемых элементов и размер единичного элемента: `int *s = (int *)calloc(n, sizeof(int));`
- Количество заказанной памяти можно изменить функцией `realloc`.

```
int *bar = (int *)malloc(n * sizeof (int));  
// ....  
bar = realloc(bar, 2*n*sizeof(int));  
return 0;  
}
```

- Мы расширили массив в два раза, скопировав старые элементы.

## Куча: двумерные массивы

- Все `alloc` функции возвращают адрес куска памяти.
- Но массивы бывают и многомерными.
- Как работать с многомерными массивами?
- Мы знаем, что простые двумерные массивы (`m[3][3]`) в функцию передаются *линеаризованными*.
- Передаётся указатель на область непрерывной памяти, в которой помещается 9 элементов.
- Повторим такой же путь создания.
- Будем работать с двумерным массивом с  $n$  строками, в каждой из которых по  $m$  элементов, или, другими словами, с матрицей  $n \times m$ .

## Куча: двумерные массивы

- Закажем память один раз сразу под все элементы матрицы:

```
double *matr = (double *)calloc(n*m, sizeof(double));
```

- Вызовом `calloc` мы обеспечили, что все элементы матрицы стали нулевыми.
- Теперь мы *обязаны* помнить про этот указатель всё: сколько в матрице, которая представлена этим указателем, строк и сколько столбцов.
- В другую функцию нужно будет передавать всё, что мы о ней знаем.
- Получим сумму элементов  $k$ -го столбца матрицы:

```
double column_sum(double *matr, size_t n, size_t m, size_t k) {  
    double sum = 0.;  
    for (size_t i = 0; i < m; i++) {  
        sum += matr[i*m+k];  
    }  
    return sum;  
}
```

- Доступ к элементу матрицы  $matr[i, j]$ , обязан иметь вид `matr[i*m+j]`.
- Если это неудобно, можно использовать другой метод хранения.



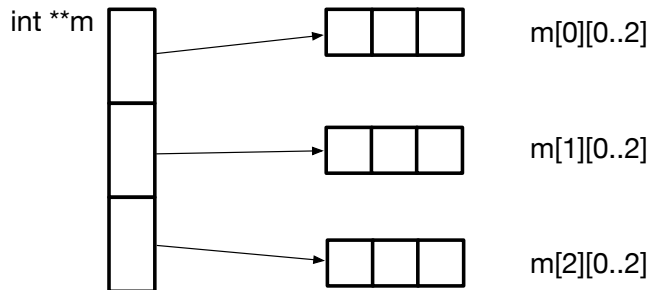
# Указатели указателей: разреженное представление матриц

- Указатели могут показывать на произвольные объекты.
- Указатели могут показывать на другие указатели.
- Для строк матрицы создадим много одномерных массивов.
- Каждый массив будет представлен указателем.
- Все такие указатели свяжем в массив указателей.

```
double **matr = (double **)calloc(n, sizeof(double *));
for (size_t i = 0; i < n; i++) {
    matr[i] = (double *)calloc(m, sizeof(double));
}
```

## Указатели указателей: разреженное представление матриц

- Как это выглядит в памяти для матрицы  $m \times 3$ :



- Что стало удобнее?

```
double column_sum(double **matr, size_t n, size_t m, size_t) {  
    double sum = 0.;  
    for (size_t i = 0; i < m; i++) {  
        sum += matr[i][k];  
    }  
    return sum;  
}
```

# Указатели указателей: разреженное представление матриц

- Операция `matr[i][k]` расшифровывается так: сначала операция `matr[i]` даёт нам какой-то указатель, который мы трактуем как массив, так как знаем, что он показывает на память, достаточную для хранения  $m$  элементов типа `double`.
- Ну а раз это массив, индексируем его и получаем результат.

## Указатели указателей: разреженное представление матриц

- Что удобнее, линейризованное или разреженное представление?
- Это зависит и от того, кто реализует набор функций под конкретную задачу, и от особенностей самой задачи.
- В первом приближении они равнозначны.

Не забывайте освобождать память после того, как поработали с матрицей и подобными объектами, возникшими на этапе исполнения программы.

- Освобождаем память для разреженного представления.

```
for (int i = 0; i < n; i++) {  
    free(matr[i]);  
}  
free(matr);
```

- Попытка освободить сначала `matr`, а потом `matr[i]` приведёт к катастрофе.