Основы программирования

Лекция № 3, 17 марта 2016 г.

http://xkcd.ru/371

Массивы

Одномерные массивы

Массив — упорядоченный набор элементов одного типа.

Объявление массива `arr`, состоящего из N элементов произвольного типа T:

```
T arr[N];
```

где N — константа времени компиляции.

Пример объявления и инициализации массива:

```
int xs[4] = {20, 10}; // {20, 10, 0, 0}
int ys[]  = {20, 10}; // {20, 10}
```


Мотивация: для задания квадратного уравнения нужно 3 переменные: a, b, c. А что делать с уравнением 10-ой степени? Правильно, нужен массив!

Без инициализации массив, объявленный в функции, содержит мусор.

Элементы, для которых нет явно заданного значения, инициализируются нулями.

Размер массива может быть вычислен автоматически, что может быть довольно удобным.

Одномерные массивы: доступ к элементам

`T arr[N]`:

Чтение элемента из массива по индексу `i`:

```
T value = arr[i];
```

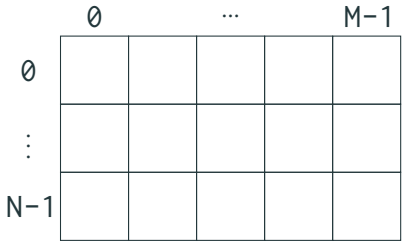
Запись элемента в массив по индексу `i`:

```
arr[i] = new_value;
```

Доступ по недопустимому индексу (-1, N, 2*N) — неопределенное поведение.

Двумерные массивы

`T arr[N][M]`:




```
int m[2][3] = {
  {1, 2, 3},
  {4, 5, 6}};
```

```
int two = m[0][1];
m[1][2] = 66;
```

Первая размерность — количество строк, как в линейной алгебре.

two получит значение 2, а вместо 6 будет записано 66.

Многомерные массивы

`T arr[N1][N2][...][Nk]` — все аналогично.

Передача массивов в функции

Зачастую, размер массива передается отдельным параметром:

```
int sum_values(int values[], size_t size) {
    int sum = 0;
    for (size_t i = 0; i < size; i++) {
        sum += values[i];
    }
    return sum;
}

int sum_pairs(int pairs[][2], size_t size) {
    // ...
}
```

size_t — специальный беззнаковый тип данных, которого гарантированно хватает для хранения размера любого массива на данной архитектуре.

Для многомерных массивов значения всех размерностей, кроме первой фиксируются.

Передача фиксированных массивов в функции

Массивы жестко фиксированных размерностей передаются без дополнительных параметров:

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

Возвращение массивов из функций

```
int[] reverse(int arr[], size_t size) {
  // compile time error
    int result[size]; // compile time error
    // ...
    return result; // run time error
}

int original[] = {1, 2, 3};
int reversed[] = reverse(original, 3);
  // compile time error
```

Формально говоря, функция в C вообще не может иметь массив в качестве возвращаемого значения.

size — не может быть использована для инициализации массива, т. к. не является константой времени компиляции.

Более того, такой массив уничтожится после возвращения из функции, но об этом позже.

Более того, в массивную переменную нельзя ничего «присвоить», что делать с результатом?

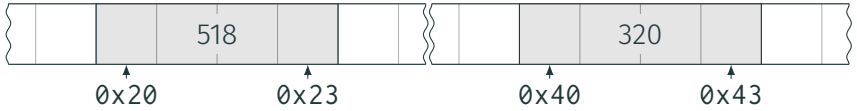
«Возвращение» массивов из функций

```
void reverse(int dst[], int src[], size_t size) {
    for (size_t i = 0; i < size; i++) {
        dst[i] = src[size - i - 1];
    }
}

int original[3] = {1, 2, 3};
int reversed[3];
reverse(reversed, original, 3);
// use reversed
```

Перекладываем ответственность за создание массива на вызывающую функцию.

Указатели

Указатель как адрес



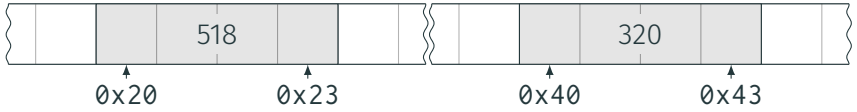

```
int x = 518;
int y = 320;
```

Адресом блока памяти считается номер его самого младшего байта.

int* — указатель на int, т.е. адрес памяти, где лежит int.

Заметьте, у поменяло свое значение, хотя прямых записей в него не было.

Указатель как адрес



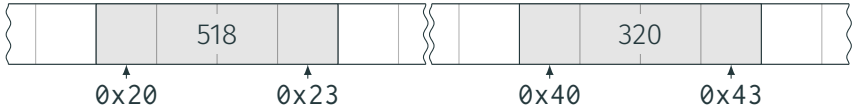

```
int x = 518;      int* ptr;
int y = 320;
```

Адресом блока памяти считается номер его самого младшего байта.

int* — указатель на int, т.е. адрес памяти, где лежит int.

Заметьте, у поменяло свое значение, хотя прямых записей в него не было.

Указатель как адрес




```
int x = 518;
int y = 320;
```

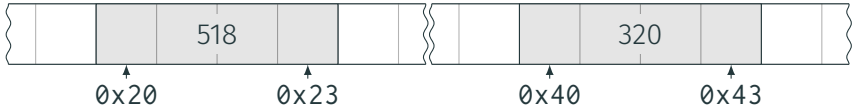
```
int* ptr;
// & - взятие адреса
// * - разыменование указателя
ptr = &x;
printf("%p %d\n", ptr, *ptr); // 0x20 518
ptr = &y;
printf("%p %d\n", ptr, *ptr); // 0x40 320
```

Адресом блока памяти считается номер его самого младшего байта.

int* — указатель на int, т.е. адрес памяти, где лежит int.

Заметьте, у поменяло свое значение, хотя прямых записей в него не было.

Указатель как адрес




```
int x = 518;
int y = 320;
```

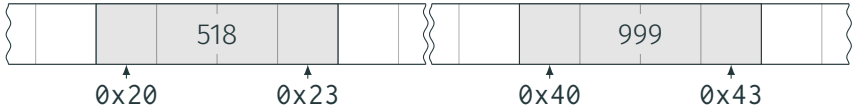
```
int* ptr;
// & - взятие адреса
// * - разыменование указателя
ptr = &x;
printf("%p %d\n", ptr, *ptr); // 0x20 518
ptr = &y;
printf("%p %d\n", ptr, *ptr); // 0x40 320
*ptr = 999;
```

Адресом блока памяти считается номер его самого младшего байта.

int* — указатель на int, т.е. адрес памяти, где лежит int.

Заметьте, у поменяло свое значение, хотя прямых записей в него не было.

Указатель как адрес




```c
int x = 518;
int y = 320;
```

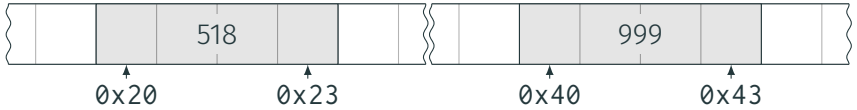
```c
int* ptr;
// & - взятие адреса
// * - разыменование указателя
ptr = &x;
printf("%p %d\n", ptr, *ptr); // 0x20 518
ptr = &y;
printf("%p %d\n", ptr, *ptr); // 0x40 320
*ptr = 999;
```


Адресом блока памяти считается номер его самого младшего байта.

int* — указатель на int, т.е. адрес памяти, где лежит int.

Заметьте, у поменяло свое значение, хотя прямых записей в него не было.

Указатель как адрес




```
int x = 518;
int y = 320;
```

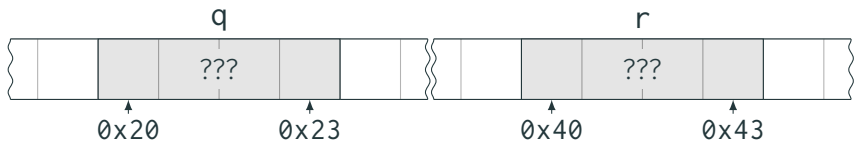
```
int* ptr;
// & - взятие адреса
// * - разыменование указателя
ptr = &x;
printf("%p %d\n", ptr, *ptr); // 0x20 518
ptr = &y;
printf("%p %d\n", ptr, *ptr); // 0x40 320
*ptr = 999;
printf("%d %d\n", x, y); // 518 999
```


Адресом блока памяти считается номер его самого младшего байта.

int* — указатель на int, т.е. адрес памяти, где лежит int.

Заметьте, у поменяло свое значение, хотя прямых записей в него не было.

Пример: возврат двух значений из функции




```
void divide(
    int a, int b,
    int* pq,
    int* pr)
{
  *pq = a / b;
  *pr = a % b;
}
```

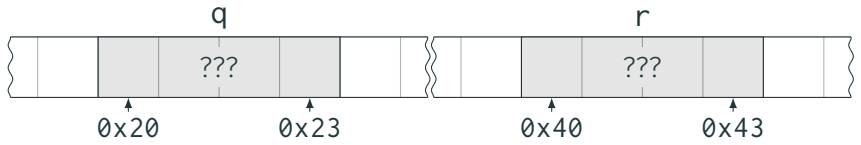
```
int x = 123, y = 10;
int q, r;
```

x, y передаются по значению, q, r передаются по указателю.

Модификация a и b в функции никак не изменит значения x, y.

Однако запись по указателям pq, pr непосредственно меняет значения q, r.

Пример: возврат двух значений из функции



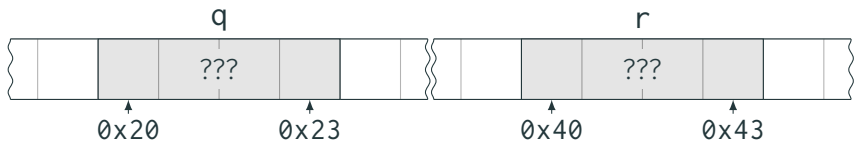

```
void divide(                    int x = 123, y = 10;
    int a, int b, // 123, 10    int q, r;
    int* pq, // 0x20
    int* pr) // 0x40            divide(x, y, &q, &r);
{
  *pq = a / b;
  *pr = a % b;
}
```

x, y передаются по значению, q, r передаются по указателю.

Модификация a и b в функции никак не изменит значения x, y.

Однако запись по указателям pq, pr непосредственно меняет значения q, r.

Пример: возврат двух значений из функции



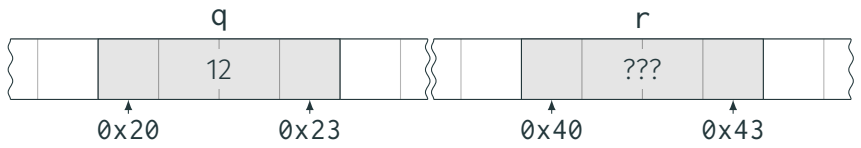

```
void divide(                    int x = 123, y = 10;
    int a, int b, // 123, 10    int q, r;
    int* pq, // 0x20
    int* pr) // 0x40            divide(x, y, &q, &r);
{
  *pq = a / b; // *(0x20) = 12
  *pr = a % b;
}
```

x, y передаются по значению, q, r передаются по указателю.

Модификация a и b в функции никак не изменит значения x, y.

Однако запись по указателям pq, pr непосредственно меняет значения q, r.

Пример: возврат двух значений из функции



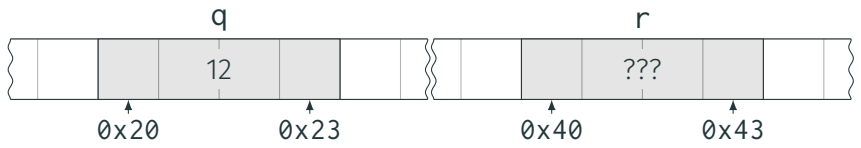

```
void divide(                    int x = 123, y = 10;
    int a, int b, // 123, 10    int q, r;
    int* pq, // 0x20
    int* pr) // 0x40            divide(x, y, &q, &r);
{
  *pq = a / b; // *(0x20) = 12
  *pr = a % b;
}
```

x, y передаются по значению, q, r передаются по указателю.

Модификация a и b в функции никак не изменит значения x, y.

Однако запись по указателям pq, pr непосредственно меняет значения q, r.

Пример: возврат двух значений из функции



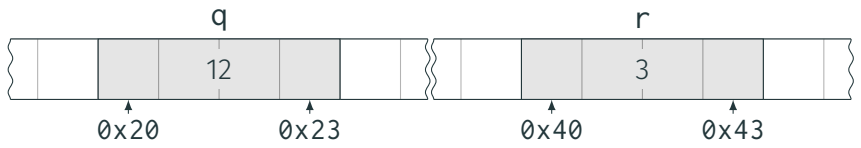

```
void divide(                    int x = 123, y = 10;
    int a, int b, // 123, 10    int q, r;
    int* pq, // 0x20
    int* pr) // 0x40            divide(x, y, &q, &r);
{
  *pq = a / b; // *(0x20) = 12
  *pr = a % b; // *(0x40) = 3
}
```

x, y передаются по значению, q, r передаются по указателю.

Модификация a и b в функции никак не изменит значения x, y.

Однако запись по указателям pq, pr непосредственно меняет значения q, r.

Пример: возврат двух значений из функции



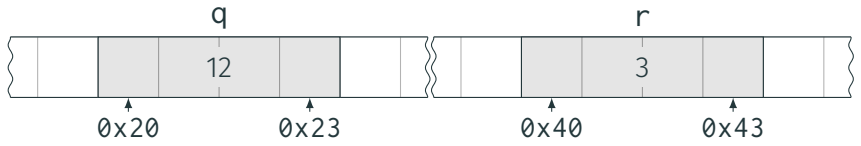

```
void divide(                    int x = 123, y = 10;
    int a, int b, // 123, 10    int q, r;
    int* pq, // 0x20
    int* pr) // 0x40            divide(x, y, &q, &r);
{
  *pq = a / b; // *(0x20) = 12
  *pr = a % b; // *(0x40) = 3
}
```

x, y передаются по значению, q, r передаются по указателю.

Модификация a и b в функции никак не изменит значения x, y.

Однако запись по указателям pq, pr непосредственно меняет значения q, r.

Пример: возврат двух значений из функции




```
void divide(
    int a, int b, // 123, 10
    int* pq, // 0x20
    int* pr) // 0x40
{
  *pq = a / b; // *(0x20) = 12
  *pr = a % b; // *(0x40) = 3
}
```

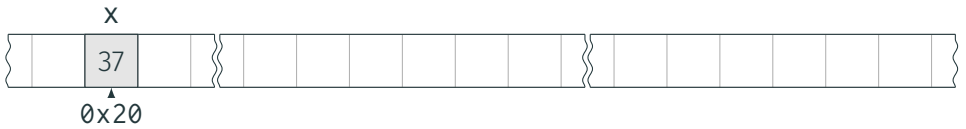
```
int x = 123, y = 10;
int q, r;

divide(x, y, &q, &r);
printf("%d = %d * %d + %d\n",
  x, y, q, r);
// 123 = 10 * 12 + 3
```

x, y передаются по значению, q, r передаются по указателю.

Модификация a и b в функции никак не изменит значения x, y.

Однако запись по указателям pq, pr непосредственно меняет значения q, r.

Представление указателя




```
char x = 37;
```

T* для любого T всегда занимает одинаковое количество байт на одной платформе. Обычно так: 32-битные платформы — 4 байта, 64-битные — 8 байт.

Адрес хранится как беззнаковое число.

Сейчас не будем подробно останавливаться на двойных указателях.

Представление указателя

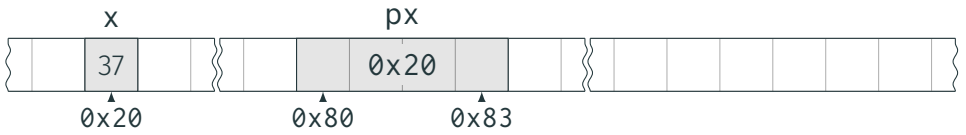

```
char x = 37;
char* px = &x;
```

T* для любого T всегда занимает одинаковое количество байт на одной платформе. Обычно так: 32-битные платформы — 4 байта, 64-битные — 8 байт.

Адрес хранится как беззнаковое число.

Сейчас не будем подробно останавливаться на двойных указателях.

Представление указателя



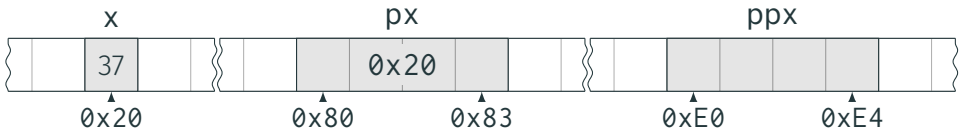

```
char x = 37;
char* px = &x;
```

T* для любого T всегда занимает одинаковое количество байт на одной платформе. Обычно так: 32-битные платформы — 4 байта, 64-битные — 8 байт.

Адрес хранится как беззнаковое число.

Сейчас не будем подробно останавливаться на двойных указателях.

Представление указателя



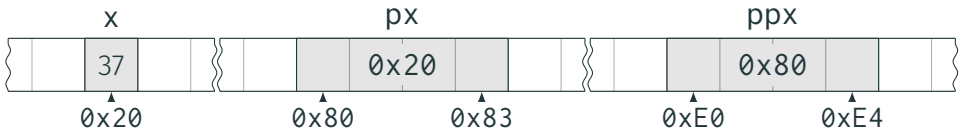

```
char x = 37;
char* px = &x;
char** ppx = &px;
```

T* для любого T всегда занимает одинаковое количество байт на одной платформе. Обычно так: 32-битные платформы — 4 байта, 64-битные — 8 байт.

Адрес хранится как беззнаковое число.

Сейчас не будем подробно останавливаться на двойных указателях.

Представление указателя



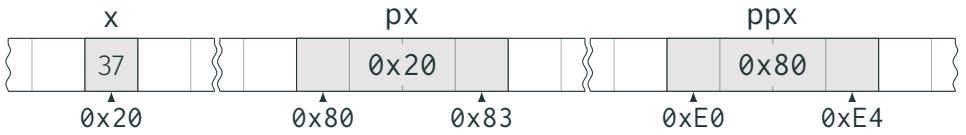

```
char x = 37;
char* px = &x;
char** ppx = &px;
```

T* для любого T всегда занимает одинаковое количество байт на одной платформе. Обычно так: 32-битные платформы — 4 байта, 64-битные — 8 байт.

Адрес хранится как беззнаковое число.

Сейчас не будем подробно останавливаться на двойных указателях.

Представление указателя



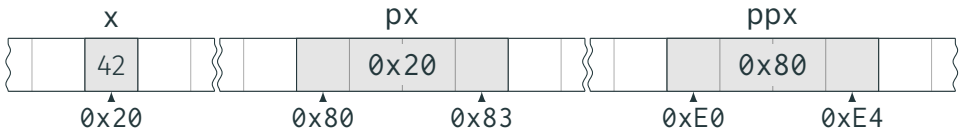

```
char x = 37;
char* px = &x;
char** ppx = &px;

**ppx = 42;
```

T* для любого T всегда занимает одинаковое количество байт на одной платформе. Обычно так: 32-битные платформы — 4 байта, 64-битные — 8 байт.

Адрес хранится как беззнаковое число.

Сейчас не будем подробно останавливаться на двойных указателях.

Представление указателя



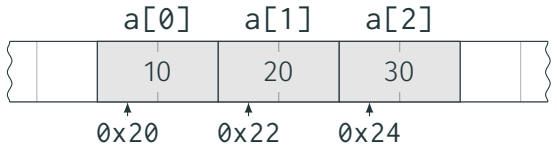

```
char x = 37;
char* px = &x;
char** ppx = &px;

**ppx = 42;
```

T* для любого T всегда занимает одинаковое количество байт на одной платформе. Обычно так: 32-битные платформы — 4 байта, 64-битные — 8 байт.

Адрес хранится как беззнаковое число.

Сейчас не будем подробно останавливаться на двойных указателях.

Представление массива



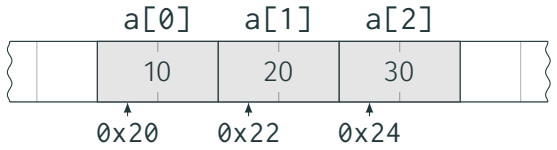

```
short a[3] = {10, 20, 30};
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление массива



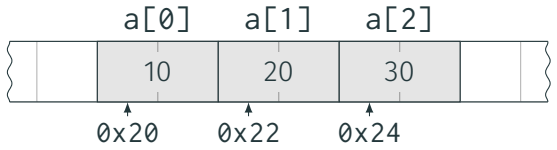

```
short a[3] = {10, 20, 30};

short* p = &(a[0]); // 0x20
short* p1 = p + 1;  // 0x22
short* p2 = p + 2;  // 0x24
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление массива



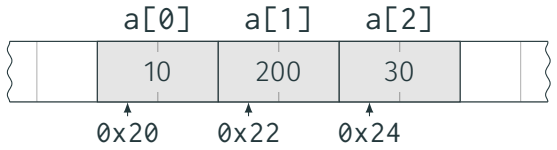

```
short a[3] = {10, 20, 30};          *p1 = 200;

short* p = &(a[0]); // 0x20
short* p1 = p + 1;  // 0x22
short* p2 = p + 2;  // 0x24
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление массива



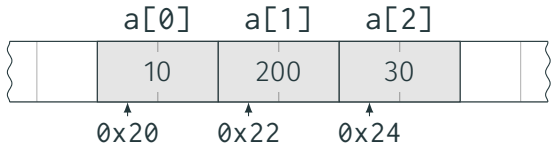

```
short a[3] = {10, 20, 30};          *p1 = 200;

short* p = &(a[0]); // 0x20
short* p1 = p + 1;  // 0x22
short* p2 = p + 2;  // 0x24
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление массива




```
short a[3] = {10, 20, 30};

short* p = &(a[0]); // 0x20
short* p1 = p + 1;  // 0x22
short* p2 = p + 2;  // 0x24
```

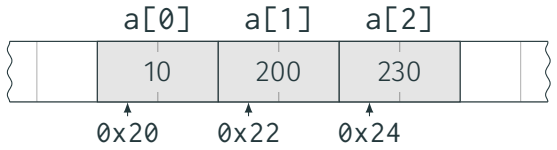
```
*p1 = 200;
*p2 = *p1 + *p2;
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление массива




```
short a[3] = {10, 20, 30};

short* p = &(a[0]); // 0x20
short* p1 = p + 1;  // 0x22
short* p2 = p + 2;  // 0x24
```

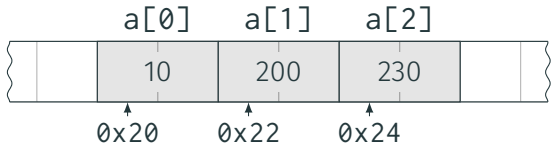
```
*p1 = 200;
*p2 = *p1 + *p2;
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление массива




```
short a[3] = {10, 20, 30};

short* p = &(a[0]); // 0x20
short* p1 = p + 1;  // 0x22
short* p2 = p + 2;  // 0x24
```

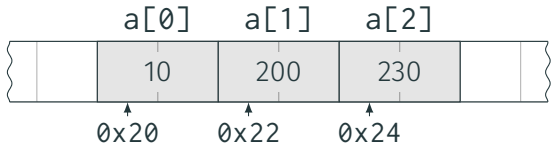
```
*(p+1) = 200;
*(p+2) = *(p+1) + *(p+2);
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление массива




```
short a[3] = {10, 20, 30};

short* p = &(a[0]); // 0x20
short* p1 = p + 1;  // 0x22
short* p2 = p + 2;  // 0x24
```

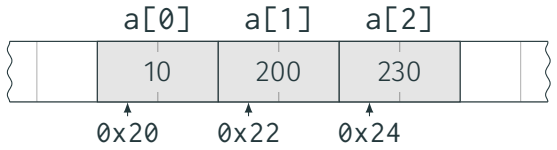
```
*(p+1) = 200;
*(p+2) = *(p+1) + *(p+2);
// *(x + y) == x[y]
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление массива




```
short a[3] = {10, 20, 30};

short* p = &(a[0]); // 0x20
short* p1 = p + 1;  // 0x22
short* p2 = p + 2;  // 0x24
```

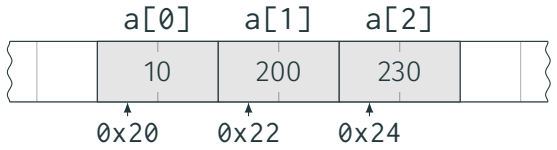
```
p[1] = 200;
p[2] = p[1] + p[2];
// *(x + y) == x[y]
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление массива




```
short a[3] = {10, 20, 30};

short* p = a; // 0x20
short* p1 = p + 1;  // 0x22
short* p2 = p + 2;  // 0x24
```

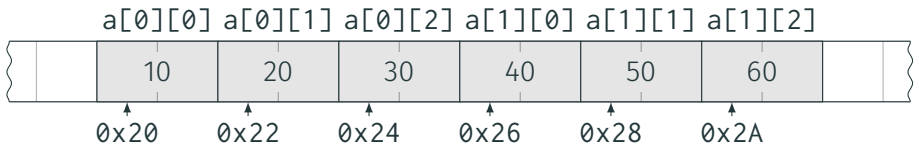
```
p[1] = 200;
p[2] = p[1] + p[2];
// *(x + y) == x[y]
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно.

При прибавлении значения n к указателю, его значение увеличивается на (n * размер типа данных под указателем).

Массивная переменная — всего лишь указатель на начало массива.

Представление двумерного массива




```
short a[2][3] = {
  {10, 20, 30},
  {40, 50, 60}};

a[i][j] == *(a + i*3 + j);
```

```
short* p = a;

a[i][j] == p[i*3 + j];
```

Массив хранится как непрерывный блок памяти, элементы располагаются последовательно: строка за строкой.

3 — это ширина одной строки.

Виды памяти

Статическая память

Примеры:

- глобальные переменные: `char game_field[100][100]`,
- строковые литералы: `"Hello world!"`,
- ...

Свойства:

- выделяется во время старта программы,
- освобождается во время завершения программы,
- инициализируется нулевыми значениями,
- фиксированный размер.

К статической памяти также относятся static-переменные.

Автоматическая память

Примеры:

- локальные переменные: `{...; int tmp; ...}`,
- аргументы функций: `(..., double speed, ...)`,

Свойства:

- выделяется при входе в объемлющий блок,
- освобождается при выходе из объемлющего блока,
- изначально не инициализирована,
- фиксированный размер.

Динамическая память

Свойства:

- выделяется и освобождается по запросу программы,
- изначально не инициализирована,
- размер задается динамически.

Недостатки автоматической и статической: фиксированный размер и недостаточно гибкое время жизни.

Самый гибкий и самый сложный в работе вид памяти.

Динамическая память

Библиотечные функции

Для работы с динамической памятью используется библиотека `stdlib.h`:

- `NULL`,
- `malloc`,
- `calloc`,
- `realloc`,
- `free`.

На слайдах мы будем опускать "#include <stdlib.h>".

Указатель, никуда не указывающий

NULL — специальное значение, символизирующее, что указатель не указывает ни на какой элемент памяти.

```
int* ptr = NULL;
int value = *ptr; // run time error
*ptr = 37; // run time error
```

Разные названия, но суть одна (segmentation fault, segfault, access violation, «Программа выполнила недопустимую операцию...», ...).

Численное значение NULL == 0.

Указатель, указывающий хоть куда

`void*` — специальный тип указателя, который может указывать на любой адрес в памяти. Может быть приведен к любого другому типу указателей и обратно.

```
int x = 37;
int* px = &x;
void* p = px;
int* py = p;
```

Минутка философии: «Какова природа void?» — спросил учитель, ...

Обязательно прочитать рассказ:
http://thecodelesscode.com/case/5?lang=ru

Выделение блока памяти

```
void* malloc(size_t size);
```

Функция выделяет блок памяти размером `size` байт и возвращает указатель на начало блока. В случае, если память выделить не получилось, возвращает NULL.

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

Функция выделяет блок памяти размером `num*size` байт, зануляет его и возвращает указатель на начало. В случае, если память выделить не получилось, возвращает NULL.

malloc = Memory ALLOCate

calloc = ALLOCate and Clear

Освобождение блока памяти

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

Функция освобождает блок памяти. Если `ptr` равен `NULL`, ничего не делает.

После вызова значение указателя `ptr` остается прежним, но разыменовывать его нельзя.

Неиспользуемую память нужно обязательно освобождать, иначе рано или поздно она может кончиться (утечка памяти).

Утечки памяти на серверных приложениях, которые должны работать несколько лет, недопустимы.

Утечки памяти в десктопных приложениях не так критичны, но могут и причинять неудобства (см. современные браузеры).

Пример: массив динамического размера

```
int n = 5;
int* p; // ???
```

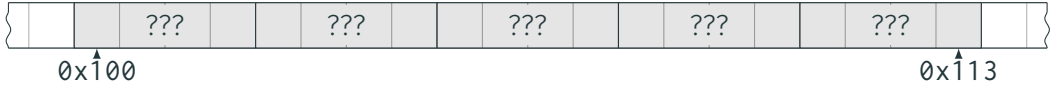

sizeof — оператор, выдающий размер типов данных. Например, на Intel x86 sizeof(int) == 4.

Пример: массив динамического размера

```
int n = 5;
int* p; // ???
p = malloc(n * sizeof(int));
if (p == NULL) { /* error */ }
```


sizeof — оператор, выдающий размер типов данных. Например, на Intel x86 sizeof(int) == 4.

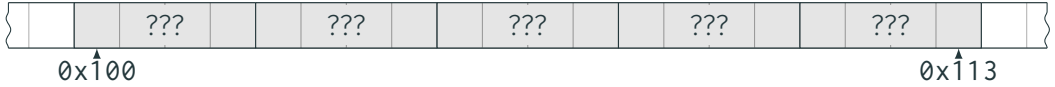
Пример: массив динамического размера




```
int n = 5;
int* p; // 0x100
p = malloc(n * sizeof(int));
if (p == NULL) { /* error */ }
```

sizeof — оператор, выдающий размер типов данных. Например, на Intel x86 sizeof(int) == 4.

Пример: массив динамического размера




```
int n = 5;
int* p; // 0x100
p = malloc(n * sizeof(int));
if (p == NULL) { /* error */ }
p[0] = p[n/2] = p[n-1] = 37;
```

sizeof — оператор, выдающий размер типов данных. Например, на Intel x86 sizeof(int) == 4.

Пример: массив динамического размера


```
int n = 5;
int* p; // 0x100
p = malloc(n * sizeof(int));
if (p == NULL) { /* error */ }
p[0] = p[n/2] = p[n-1] = 37;
```

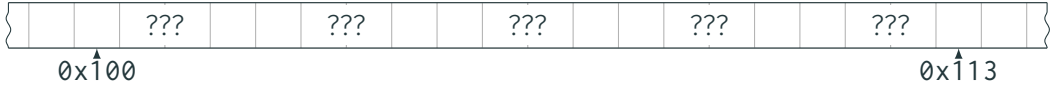
sizeof — оператор, выдающий размер типов данных. Например, на Intel x86 sizeof(int) == 4.

Пример: массив динамического размера


```
int n = 5;
int* p; // 0x100
p = malloc(n * sizeof(int));
if (p == NULL) { /* error */ }
p[0] = p[n/2] = p[n-1] = 37;
free(p);
```

sizeof — оператор, выдающий размер типов данных. Например, на Intel x86 sizeof(int) == 4.

Пример: массив динамического размера




```
int n = 5;
int* p; // 0x100
p = malloc(n * sizeof(int));
if (p == NULL) { /* error */ }
p[0] = p[n/2] = p[n-1] = 37;
free(p);
```

sizeof — оператор, выдающий размер типов данных. Например, на Intel x86 sizeof(int) == 4.

Изменение размера блока памяти

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

Функция изменяет размер блока памяти до `size` байт. В случае успешного изменения размера возвращает указатель на начало блока, иначе `NULL`.

Функция может как уменьшать размер, так и увеличивать. Возможно перемещение содержимого памяти, при этом возвращается указатель на новое месторасположение.

В случае неуспешного изменения размера, изначальный блок памяти не освобождается.

Нельзя писать ”ptr = realloc(ptr, ...)” — утечка памяти.

Возвращение массивов из функций

```
int* reverse(int src[], size_t size) {
    int* dst = malloc(size * sizeof(int));
    if (dst == NULL) { return NULL; }

    for (size_t i = 0; i < size; i++) {
        dst[i] = src[size - i - 1];
    }
    return dst;
}
int original[3] = {1, 2, 3};
int* reversed = reverse(original, 3);
// use reversed
free(reversed);
```


Теперь мы знаем, как мы можем вернуть массив из функции.

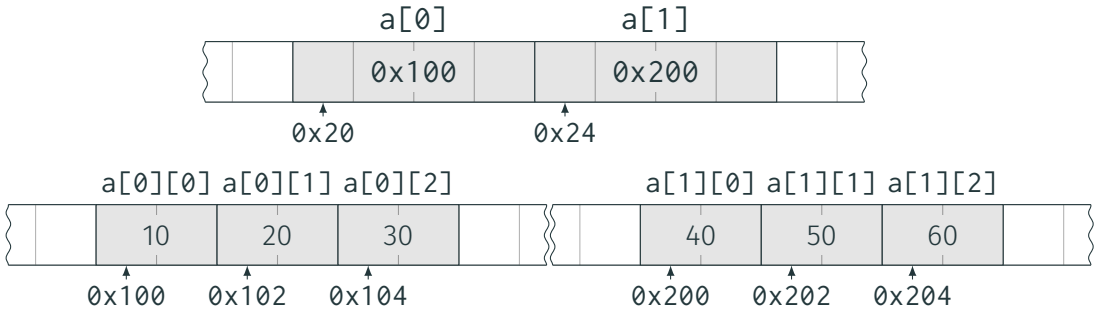
Плоские динамические двумерные массивы

```
int* arr = malloc(n * m * sizeof(int));
if (arr == NULL) { /* error */ }

// arr[i*m + j] = 37;

free(arr);
```

Плюсы: высокая производительность.

Минусы: дополнительная арифметика при доступе к элементам.

Динамические массивы массивов: идея




```
short** a = /* 0x20 */;
// {{10, 20, 30}, {40, 50, 60}}

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

Храним массив указателей на другие массивы.

Плюсы: привычный синтаксис доступа к элементам.

Минусы: большее потребление памяти, лишняя косвенность.

Такой подход используется по умолчанию в некоторых языках программирования для реализации многомерных массивов. Например, в Java.

Динамические двумерные массивы массивов

```
int** arr = malloc(n * sizeof(int*));
if (arr == NULL) { /* error */ }
for (int i = 0; i < n; i++) {
    arr[i] = malloc(m * sizeof(int));
    if (arr[i] == NULL) { /* error */ }
}

// arr[i][j] = 37;

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

Конец третьей лекции