

# **Массивы**

## **Адресная арифметика**

Основы языка C, лекция 6

# sizeof

- Размер
- `char c;`  
`int x;`  
`float z;`  
`char * p;`  
`double * pz;`
- **sizeof** (char)     // 1  
**sizeof** (x)
- **size\_t** n = **sizeof** (x);
- `printf ( "%zu", n);`

# Задачи (нужно запомнить)

- Дано число N. Далее N чисел через пробел.  
Дважды напечатайте введенную последовательность.

Input:

5

7 -2 11 0 -32

Output:

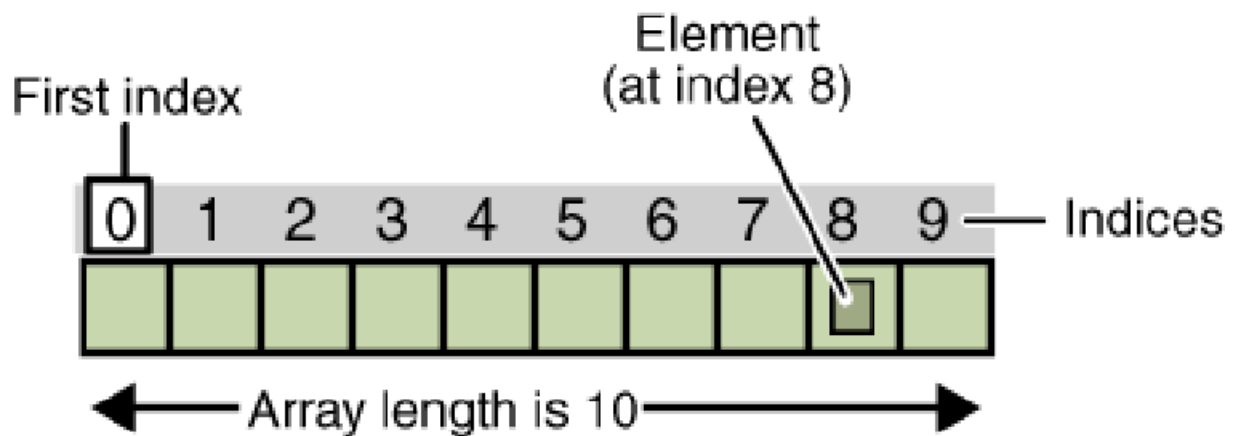
7 -2 11 0 -32

7 -2 11 0 -32

- Дано число N. Далее N чисел через пробел.  
Напечатать только числа превышающие среднее арифметическое этой последовательности.

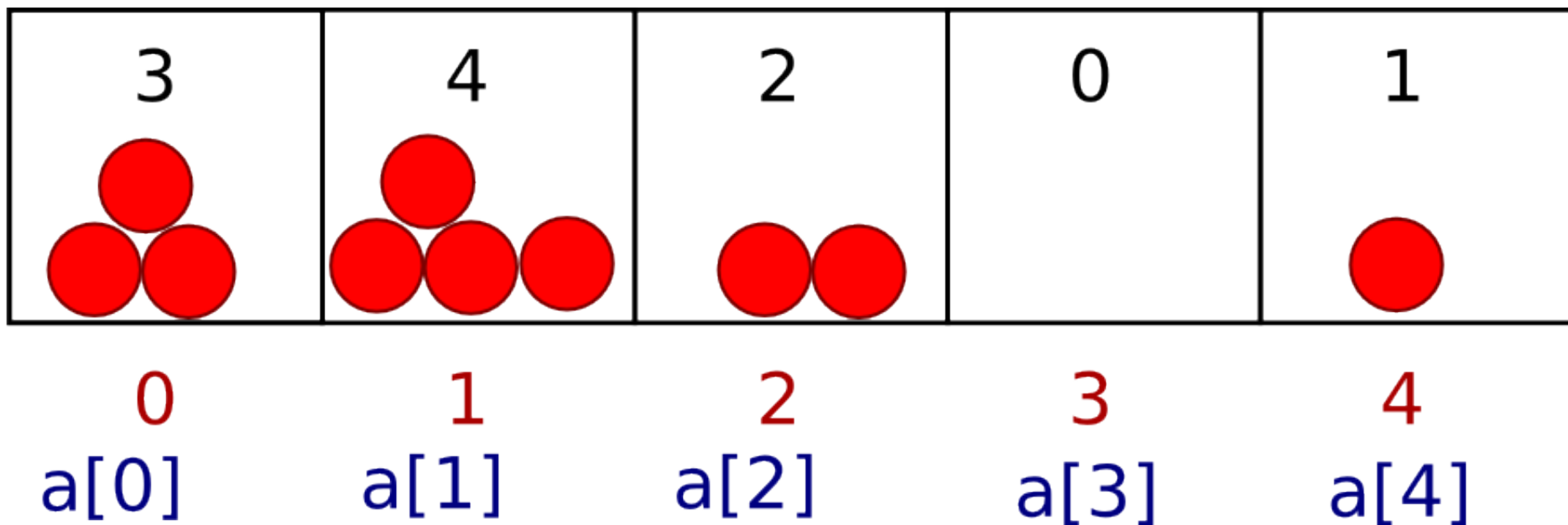
# Массив

- Массив (array) — набор **однотипных** элементов **фиксированной** длины.
- Объявление массива:  
**тип** **имя** **[размер]**;
- **int** **a** **[10]**;
- **a[8]** - обращение к элементу массива с индексом **i**  
 $a[0] = -17 + a[3]$ ;
- Нумерация элементов начинается с **0**.



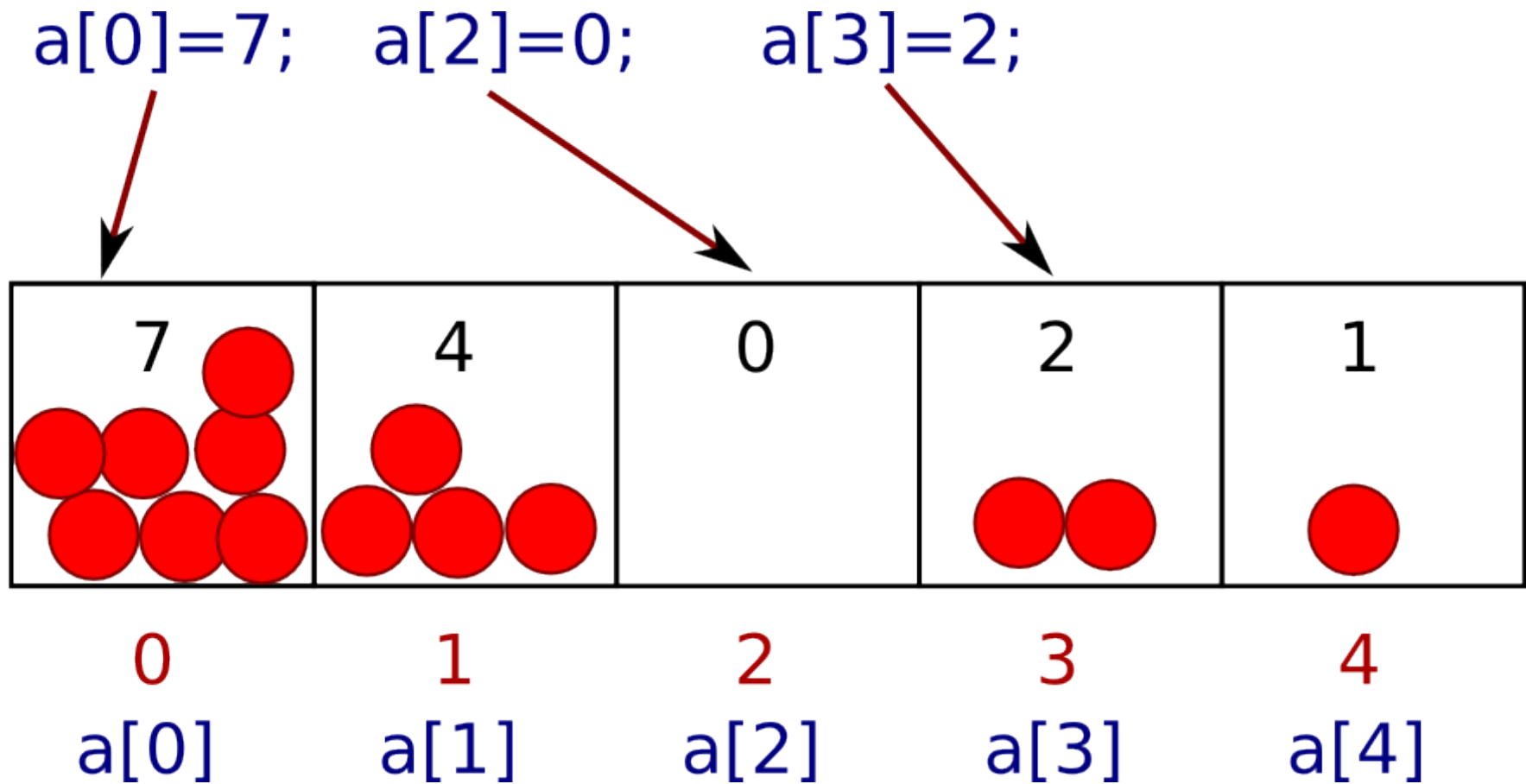
# На пальцах

- Набор одинаковых коробок.  
У каждой коробки свой номер (индекс).



# Изменение значений

a



# Перебор всех элементов

- `int a [3];`  
`int i;`  
`for ( i = 0; i < 3; i++)`  
`a[i] = 0;`
- C99 - забанен в проверяющей системе  
`for ( int i = 0; i < 3; i++) { .. }`  
принцип "где нужен, там и делаем"
- `int matrix[3][4], i, j;`  
`for (i = 0; i < 3; i++)`  
`for (j = 0; j < 4; j++)`  
`matrix [ i ] [ j ] = 0;`

# Явная инициализация

- `int a [ ] = {23, 7, 144};`      `// a[3]`
- `int a [ 3 ] = {23, 7, 144};`      `// ok`
- `int a [ 3 ] = {23, 7};`      `// остальное нулями`
- `int a [ 3 ] = {0};`      `// все 0`
- `int a [ 2 ] = {23, 7, 144};`      `// ERROR`
- `int a[10] = {[5]= -7, [1]=100};`      `// C99+`
- `char str1[ ] = {'H', 'e', 'l', 'l', 'o', '\0'};`  
  `char str2[ ] = "Hello";`
- `size_t n = strlen (str1);`      `// 5 (без '\0')`  
  `sizeof(str1)`      `// 6 (место в памяти)`



# Явная инициализация 2D

- `int matrix [3][4] = {  
    { 13, -7, 8, 5} ,  
    {-7, -1, 14, 3} ,  
    {1, -4, 2, 11}  
};`

# Ошибки

- Использование неинициализированных значений

```
int a[3];  
int x = a[2];
```

- Выход за границы массива

```
int a[3];  
// Сообщений об ошибках нет  
a[3] = 5;  
x = a[-2];
```

- C99

```
int n;  
scanf("%d", &n);  
int a[n];
```

# Избегайте магических чисел

- ```
int main ( ) {  
    int a[10], i;  
    for (i = 0; i < 10; i++)  
        a[i] = 10 * i;  
    return 0;  
}
```
- Эти 10 — одна характеристика или независимые числа?

# Лучше

- #define **MAX\_SIZE** 10  
#define **BASE** 10

```
int main ( ) {  
    int a[MAX_SIZE], i;  
    for (i = 0; i < MAX_SIZE; i++)  
        a[i] = BASE * i;  
    return 0;  
}
```

- Легче перейти к восьмеричной системе или более длинным массивам.

# sizeof

- #define MAX\_SIZE 10  
#define BASE 10

```
int main ( ) {  
    int a[MAX_SIZE], i;  
    for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)  
        a[i] = BASE * i;  
    return 0;  
}
```

- Вычислим длину массива

# Указатели

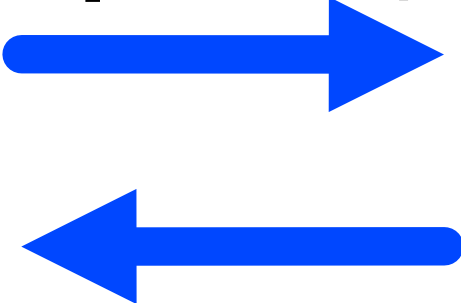
**&** операция взятия указателя

`int x = 1;`

`p = &x;`

`int * p;`

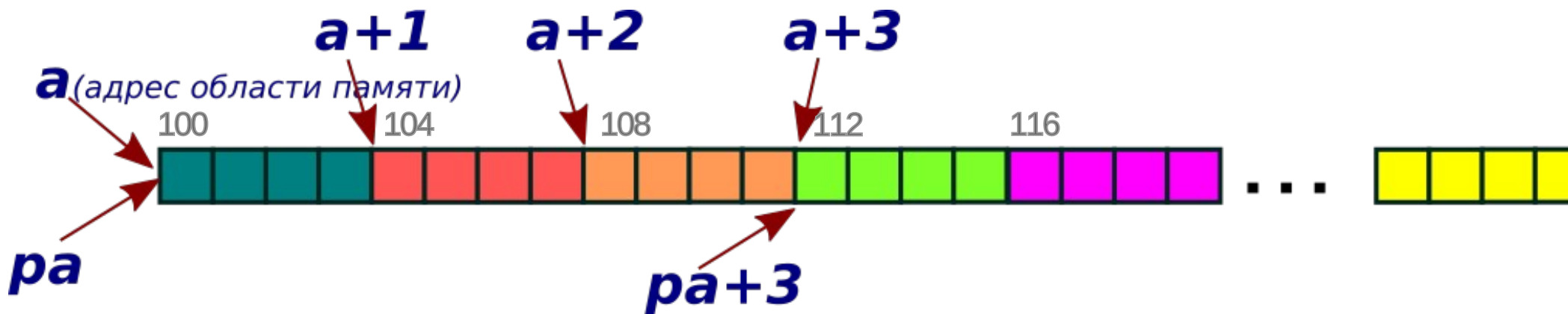
`*p = *p + 3`



**\*** операция чтения/записи в память по указателю

**int \*** тип данных

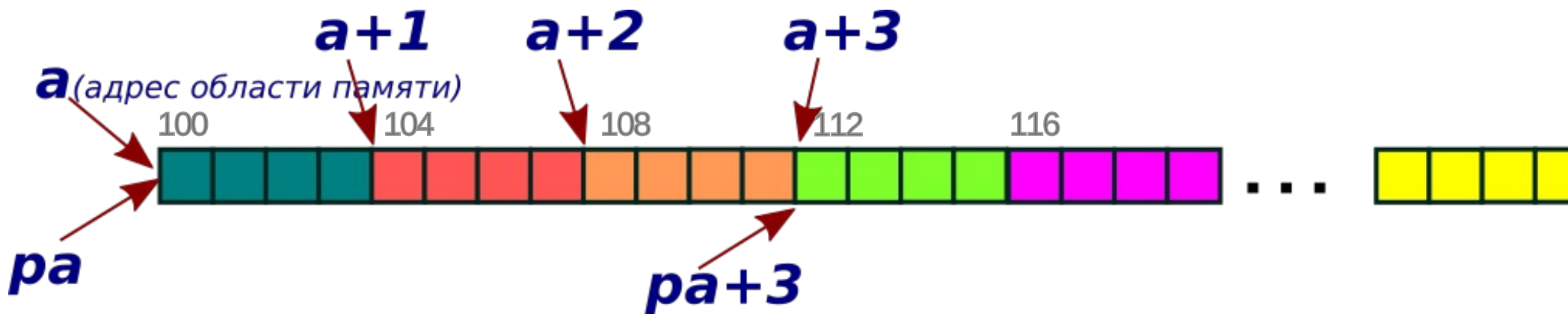
# Адресная арифметика



- Пусть `int` занимает 4 байта. **`sizeof(int)`**  
Тогда, если `a` начинается с адреса 100, его ячейки имеют адреса 100, 104, 108, 112 и далее
- Найдем адрес `a[0]` и `a[3]`  

```
int a[10];  
int * p0 = &a[0];  
int * p3 = &a[3];  
printf("%p\n%p\n", p0, p3);
```

# Адресная арифметика



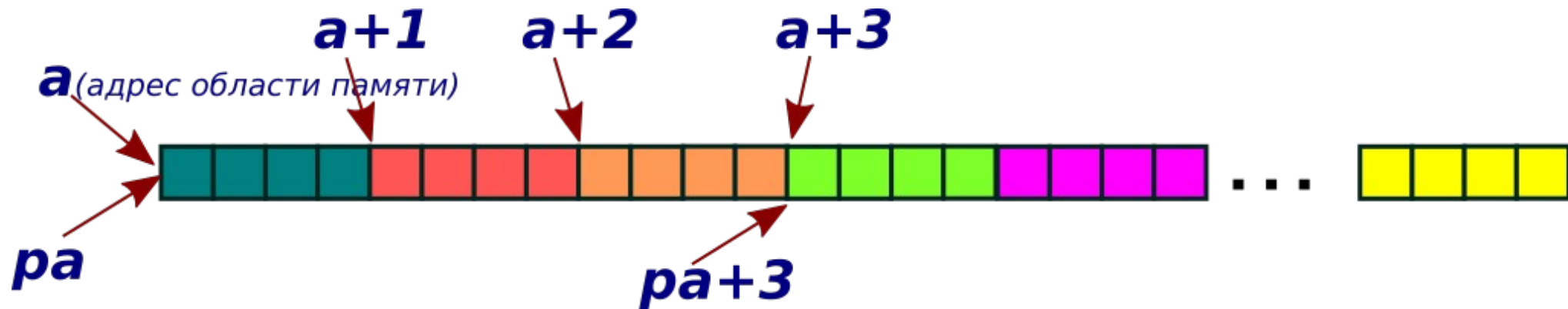
- Переход от  $p_0$  (100) к  $p_3$  (112), если `sizeof(int)` 4 с точки зрения математики:  
сдвигаемся на  $3 * \text{sizeof(int)}$  // громоздко
- Для многомерных массивов еще неудобнее.
- Можно ли из  $p_0$  извлечь информацию, что сдвигаться надо на `int`?

**int \***  $p_3$ ;

$p_3 = p_0 + 3$ ;



# Указатели и массивы



- `int a [10], b[10] = {1};`  
`int * pa;`  
`pa = a;      // имя массива — адрес его начала`
- `*(a + 3) = 7;      a[3] = 7;      &a[3]`  
`*(pa + 3) = 7;      pa[3] = 7;      a + 3`
- Можно: `*(3 + a) = 7` или `3[a] = 7`    `"012345"[i]`
- Нельзя: `a = pa;      a = b;`

# Адресная арифметика

- указатель + целое = указатель
- указатель — указатель = целое
- указатель + указатель // ошибка
- Сравнить указатели
  - указатель == указатель
  - указатель != указатель
  - указатель < указатель // машинно-зависимо
  - указатель > указатель // машинно-зависимо
  - указатель — указатель < 0 // ок

# Тип `void *`

- `void *` `pv`;  
`int *` `pint` = `a`;  
`char *` `pchar` = `b`;  
`float *` `pfloat` = `m`;
- Преобразования адресов в / из `void *` неявно  
`pv` = `pint`;                `pint` = `pv`;  
`pv` = `pchar`;              `pchar` = `pv`;  
`pv` = `pfloat`;              `pfloat` = `pv`;
- В C++ надо писать явное приведение типа  
`pint` = (`int *`) `pv`;
- `pv + 3`        // ошибка (на сколько байт сдвигать?)

# NULL

- **NULL** — адрес, которого не существует  
#include <stdio.h> // stddef.h  
intptr — C99 и выше
- Обычно  
**(void \*) 0**
- Запись по адресу NULL запрещена
- Чтение по адресу NULL запрещено (Linux, UNIX, ..)

# Передача массива в функцию

```
void foo10(int a[10]) { printf("foo10: %zu\n", sizeof(a)); }  
void foo (int a[ ])   { printf("foo   : %zu\n", sizeof(a)); }  
void foop (int * a)   { printf("foo10: %zu\n", sizeof(a)); }
```

```
int main() {  
    int a[10];  
    printf("main : %zu\n", sizeof(a));  
    printf("int*  : %zu\n", sizeof(int*));  
    foo10(a);  
    foo(a);  
    foop(a);  
    return 0;  
}
```

```
main   : 40  
int*   : 8  
foo10  : 8  
foo    : 8  
foop   : 8
```

# Передача массива в функцию

- Массив — очень много данных.  
Копировать его в функцию тяжело.
- Поэтому массивы передаются в функцию не копией, а указателем (адрес)
- Как узнать длину массива в функции, если `sizeof(a) / sizeof(a[0])` не работает?
  - поместить в конце данных специальный символ (строки - `'\0'`)
  - передать длину данных в аргументе  
`void print_arr (int a[ ], int n);`

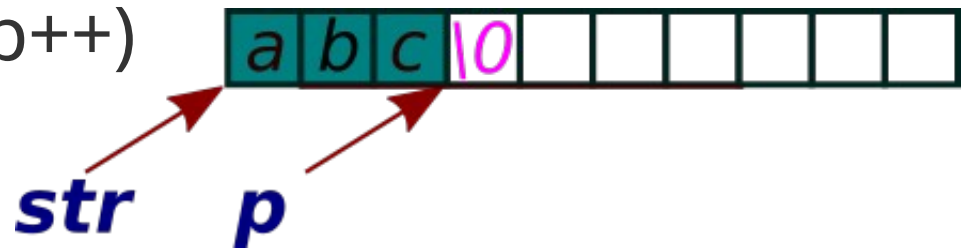
# Указатели - удобно

- ```
void print_arr (int a [ ], int n) {  
    for (int i = 0; i < n; i++)  
        printf("%d ", a[ i ]);  
    printf("\n");  
}  
  
int main() {  
    int b [10] = {1, -2, 3, -4, 5, -6, 7, -8, 9, 10};  
    print_arr (b, 10);        // весь массив  
    print_arr (b, 7);         // первые 7  
    print_arr (b+3, 10-3);    // последние 7  
    print_arr (b+3, 4);       // от -4 до 7 включительно  
    return 0;  
}
```

# i++ vs p++

- ```
int my_strlen (char str[ ] ) {  
    int n;  
    for (n = 0; str[n] != '\0'; n++)  
        ;  
    return n;  
}
```

- ```
int my_strlen (char str[ ] ) {  
    char * p;  
    for (p = str; *p != '\0'; p++)  
        ;  
    return p - str;  
}
```





# Треугольник Паскаля

- $(a + b)^0 = 1$

$$(a + b)^1 = a + b$$

$$(a + b)^2 = a^2 + 2ab + b^2$$

$$(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

...

$$(a + b)^n = \sum C_n^k * a^k b^{n-k}$$

- Почему не посчитать по формуле

- $C_n^k = n! / (k! * (n-k)!) )$

- 1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

.....

Треугольник Паскаля

0:						1														$(a+b)^n =$
1:						1	1													$= \sum_{k=0}^n \binom{n}{k} a^k b^{n-k}$
2:						1	2	1												
3:						1	3	3	1											
4:						1	4	6	4	1										
5:						1	5	10	10	5	1									
6:						1	6	15	20	15	6	1								
7:						1	7	21	35	35	21	7	1							
8:						1	8	28	56	70	56	28	8	1						
9:						1	9	36	84	126	126	84	36	9	1					
10:						1	10	45	120	210	252	210	120	45	10	1				
11:						1	11	55	165	330	462	462	330	165	55	11	1			
12:						1	12	66	220	495	792	924	792	495	220	66	12	1		
13:						1	13	78	286	715	1287	1716	1716	1287	715	286	78	13	1	

# Реализация — 1

- Для подсчета до N ряда включительно  
`int pas [ N + 1 ] [ N + 1 ];`
- `pas[ i ][ 0 ]` и диагональ содержат 1  
Заполняем, начиная с `pas[0][0]`  
$$\text{pas}[ i ][ j ] = \text{pas}[ i ][ j - 1 ] + \text{pas}[ i - 1 ][ j ]$$
- - Половина памяти не используется,  
+ хранятся уже вычисленные

0	1
1	1 1
2	1 2 1
3	1 3 3 1
4	1 4 6 4 1

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		

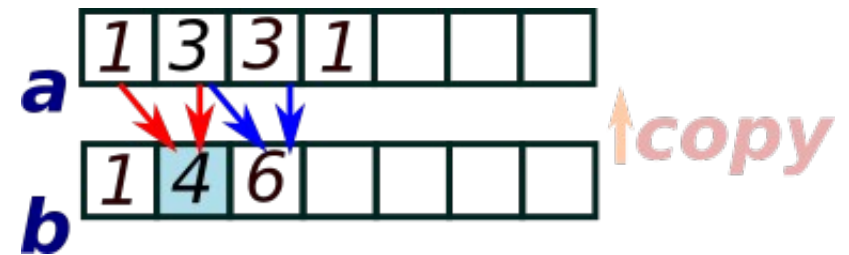
# Реализация — 2

- Если не нужно хранить все результаты
- Для вычисления ряда  $i$  нужен только ряд  $i - 1$

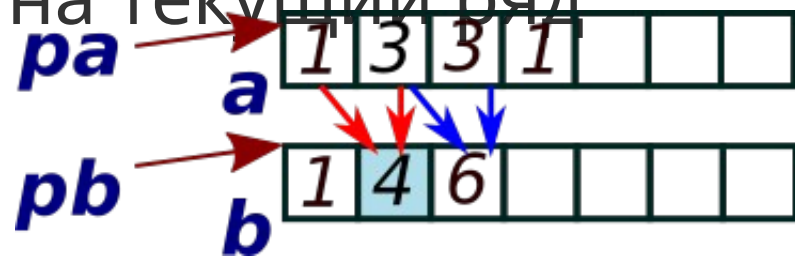
- `int a [ N + 1 ], b [ N + 1 ];`

- "Школьный" подход:

- По `a[ ]` вычислим `b[ ]`
- откопируем данные из `b` в `a`



- `int a [ N + 1 ],`  
`b [N + 1];`
- `int * pa;    // указатель на "предыдущий" ряд`  
`int * pb;        // указатель на текущий ряд`  
`// по pa вычислим pb`



`int * t = pa; pa = pb; pb = t;    // поменять ряды`

Diagram illustrating the state of two arrays, `a` and `b`, after a swap operation. Array `a` now contains the values `1, 5, 3, 1` and array `b` contains the values `1, 4, 6`. Red arrows indicate pointers: `pa` points to the first element of `a`, and `pb` points to the first element of `b`. Blue arrows show the calculation of `pb` from `pa`: the second element of `a` (5) points to the second element of `b` (4), and the third element of `a` (3) points to the third element of `b` (6). A red 'X' is drawn over the pointer labels `pa` and `pb`, indicating they have been swapped.

# Реализация - 3

- Если подумать, можно обойтись 1 одномерным массивом  
`int a [N + 1];`
- Идем сзади, ставим 1 и вычисляем остальные  
 $a[j] = a[j] + a[j-1]$



# Решето Эратосфена

- Находим простые числа
- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
- Выписать подряд все целые числа от двух до  $n$ .
- Пусть переменная  $p$  изначально равна двум — первому простому числу.
- Зачеркнуть в списке числа кратные  $p$
- Идем дальше. Если число зачеркнуто, оно не простое, идем дальше.
- Если число не зачеркнуто, то повторяем алгоритм для  $p =$  этому числу
- До каких пор?