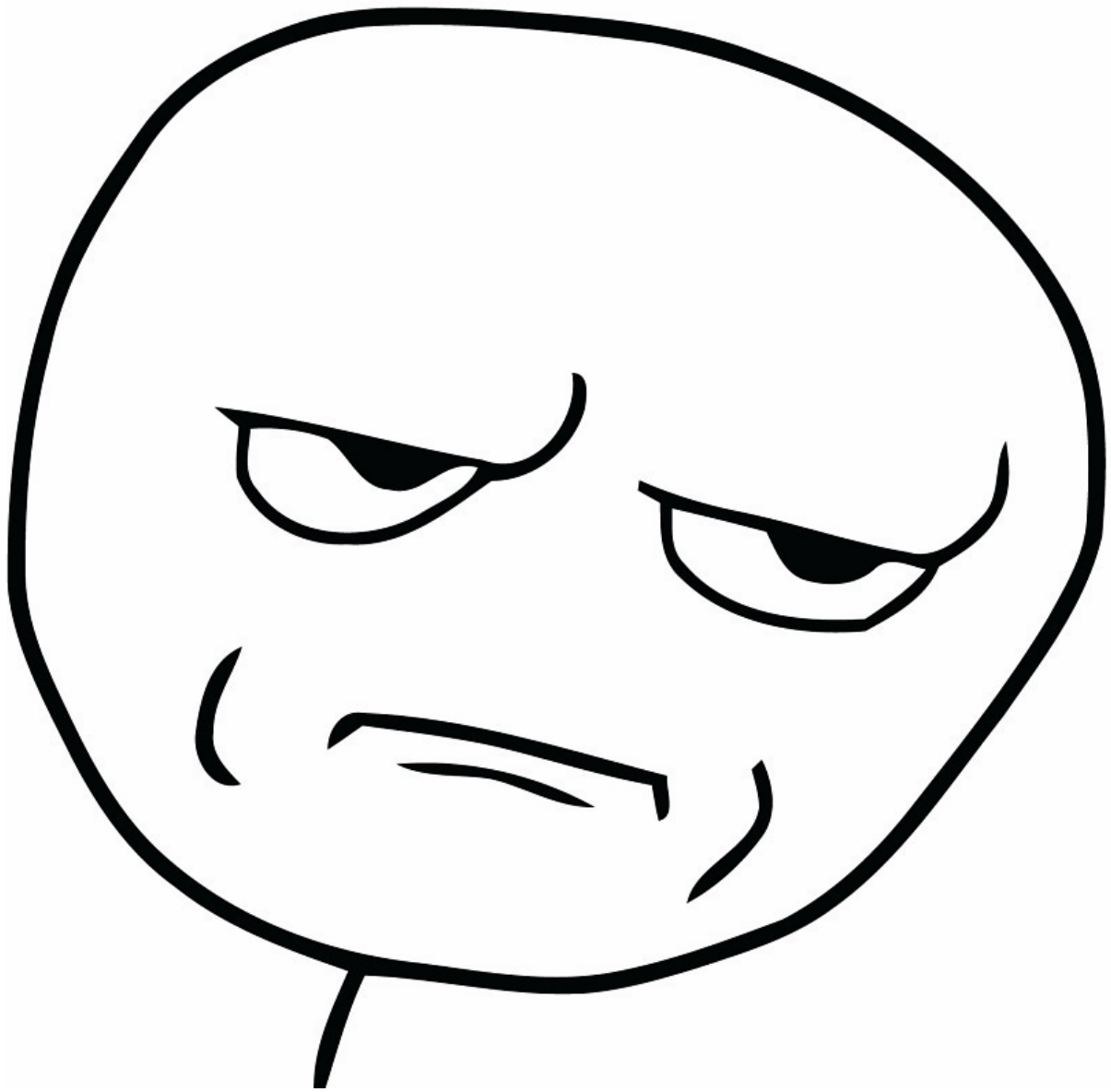
Массивы и строки

(в стиле языка C)



Массив

Последовательность элементов **одного типа**, расположенных **непрерывно** в памяти, к которым имеется **доступ по индексу** через некоторый **уникальный идентификатор** (cplusplus.com)

```
int array[10];
```

Вопросы

```
int array[10];
```

- Что такое **array**?

Вопросы

```
int array[10];
```

- **array** - это ~~массив~~, ~~указатель~~ идентификатор (имя переменной).
- Какой тип у переменной **array**?

Ответы

```
int array[10];
```

- **array** - это ~~указатель~~ идентификатор (имя переменной)
- Тип переменной **array** - ~~указатель на int~~, ~~константный указатель на int~~, массив int'ов размера 10.

Доказательства

```
std::cout << sizeof(array)       << ' '    // размер массива в байтах
          << sizeof(&array[0])   << ' '    // размер указателя в байтах
          << sizeof(int[10])     << ' '    // размер массива в байтах
          << sizeof(int* const) << '\n';  // размер указателя в байтах
```

```
40 8 40 8
```

.....

```
array * 4;  // CE: недопустимая операция
```

```
error: invalid operands of types 'int [10]' and 'int' to binary 'operator*’
```

Преобразование массив → указатель

Массивы почти всегда неявно приводятся к указателям (на нулевой элемент).

```
*array, array + 4, array != nullptr;
int f(int array[10]) {}  // type(array) == int*
```

```
main.cpp: In function 'int f(int*)':
main.cpp:4:23: warning: no return statement in function returning non-void
```

Преобразование массив → указатель

Массивы почти всегда неявно приводятся к указателям (на нулевой элемент).

Исключения: `sizeof`, `&`, строковый литерал в правой части присваивания

```
sizeof(array) != sizeof(int* const);  // 1
(int*)&array == array;                // 2
char str[] = "string";                // 3
```

1. `sizeof(array)` возвращает размер массива в байтах
2. Указатель на массив совпадает с указателем на нулевой элемент массива
3. Массив `str` инициализируется элементами массива `"string"`

Передача массивов в функции

Передача массивов по значению не дает ожидаемого результата.

```
void f(int array[50]);  // <=> void f(int* array)

int main() {
  int normal[50];
  f(normal);  // Ok

  int large[100];
  f(large);   // Ok (беда?)

  int small[10];
  f(small);   // Ok (беда!)
}
```

Передача массивов в функции

Решение: можно передавать указатель на массив.

```
void f(int (*array_ptr)[50]);

int main() {
  int normal[50];
  f(&normal);  // Ok

  int large[100];
  f(&large);   // Compilation error

  int small[10];
  f(&small);   // Compilation error
}
```

```
error: cannot convert 'int (*)[10]' to 'int (*)[50]'
error: cannot convert 'int (*)[100]' to 'int (*)[50]'
```

Передача массивов в функции

Аналогично можно передавать массив по ссылке.

```
void f(int (&array_ref)[50]) {}

int main() {
  int normal[50];
  f(normal);  // Ok

  int large[100];
  f(large);   // Compilation error

  int small[10];
  f(small);   // Compilation error
}
```

```
error: invalid initialization of reference 'int (&)[50]' from expression of type 'int (&)[10]'
error: invalid initialization of reference 'int (&)[50]' from expression of type 'int (&)[100]'
```

Правила работы с массивами

1. Нельзя создавать массивы ссылок и массивы функций, но можно создавать массивы указателей и массивы указателей на функции.

```
int& a[10];       // Compilation error
int b[20](int);  // Compilation error
```

```
error: declaration of 'a' as array of references
error: declaration of 'b' as array of functions
```

```
int* c[30];         // Ok (массив указателей)
int (*d[40])(int); // Ok (массив указателей на функции)
```

Правила работы с массивами

2. Нельзя создать массив с неизвестным числом элементов, но можно его объявить.

```
int a[];                // Compilation error (определение)

int b[] = {1, 2, 3};    // Ok: int[3]

extern int c[];         // Ok (объявление)
```

```
error: storage size of 'a' isn't known
```

Правила работы с массивами

3. При сравнении массивов сравниваются адреса нулевых элементов (не значения!). Это значит, что результат сравнения на равенство для разных массивов всегда `false`.

```
int a[3]{1, 2, 3};

int b[3]{1, 2, 3};

std::cout << (a == b) << ' ' << (a == a);
```

```
0 1
```

Правила работы с массивами

4. Массивы нельзя присваивать друг другу (исключение - строки при инициализации).

```
int a[3];
int b[3]{1, 2, 3};
a = b;  // Compilation error
```

5. Лайфхак: если массив - это поле структуры или класса, то чудесным образом присваивание начинает работать

```
struct S {
    int array[3];
};

S c{1, 2, 3};
S d{4, 5, 6};
c = d;  // Ok (c.array == {4, 5, 6})
```

Динамические массивы

Динамические массивы

Создаются с помощью оператора `new[]`, который возвращает указатель на нулевой элемент массива.

```
int* array = new int[10];
```

Так как результат - указатель на элемент (не на массив!), его нельзя использовать в предыдущих контекстах

```
sizeof(new int[10]) == sizeof(int*);  // 8
sizeof(*array) == sizeof(int);        // true
(int*)&array == array;                // false
```

Динамические массивы

Размер стекового массива, в отличие от динамического обязан быть константой времени компиляции!

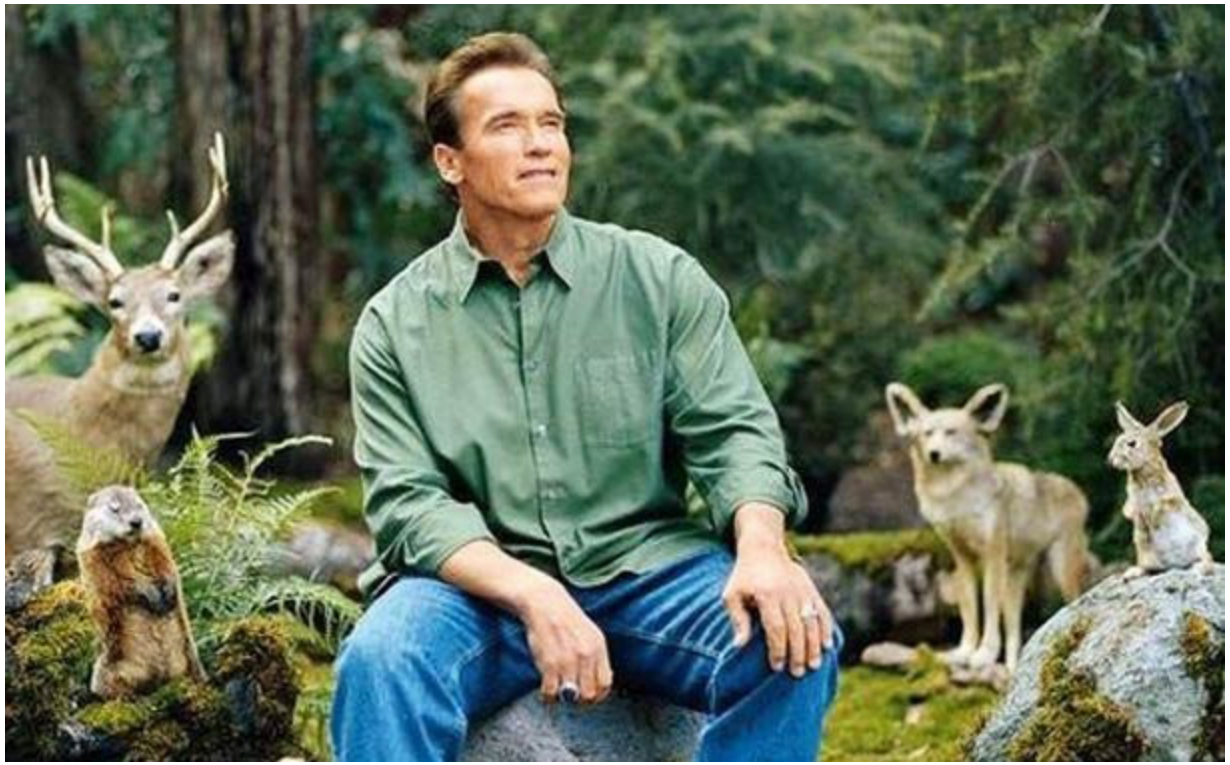
```
int n = 10;

// Работаем с n ...

int a[n]{1, 2, 3};  // стандартом запрещено, но компиляторы позволяют

int* b = new int[n]{1, 2, 3};  // Ok
```

```
warning: ISO C++ forbids variable length array 'a'
```

Динамические массивы

И не забывайте убирать за собой!

```
int* array = new int[10];
// ...
delete[] array;
```



Строки

```
// это код на C++, ок да?
"string";
```

- Что такое `"string"`?
- Какой тип у `"string"`?

Строки

- Что такое `"string"`?

~~const char*~~, ~~строка~~, строковый литерал

- Какой тип у `"string"`?

~~const char*~~, ~~строка~~, const char[7]

Доказательства

```
std::cout << sizeof("string") << ' '
          << sizeof(const char[7]) << ' '
          << sizeof(const char*);
```

```
7 7 8
```

.....

```
"string" * 4;
```

```
error: invalid operands of types ‘const char [7]' and 'int' to binary 'operator*'
```

Строковые литералы

1. Представляют собой массивы символов с `'\0'` на конце

```
std::cout << "ab"[0]  // 'a'
          << "ab"[1]  // 'b'
          << ("ab"[2] == '\0');  // true
```

2. Могут быть скопированы при инициализации (исключение из общего правила)

```
char str[4] = "kek";  // Ok
// str = "lol";          CE (не инициализация)
```

Строковые литералы

3(?). Могут сравниваться (?!)

```
std::cout << ("kek" == "kek" ? "Ну нет..." : "false") << '\n';
std::cout << ("kek" != "lol" ? "Да не может быть..." : "false") << '\n';
```

```
Ну нет...
Да не может быть...
```

Вы же не забыли, что массивы сравниваются не поэлементно?

Сравниваются адреса первых элементов.

Строковые литералы: объяснение

- Массив, с которым связан строковый литерал, лежит в статической (глобальной) области памяти (в таблице строковых литералов).
- Компилятор, анализируя исходный код, помещает каждый попавшийся строковый литерал в отдельный буфер (отдельная строка таблицы).
- Как правило, одинаковые литералы ссылаются на одну и ту же область памяти, поэтому их сравнение путем сравнения указателей может давать верный результат (но это не гарантировано стандартом!).

Строки

- Строка - часть массива элементов `char`, ограниченная символом `\0`
- Правила работы со строками такие же как и с обычными массивами

Строки

- Как создать строку?

```
const char* static_string = "lol";
char stack_string[4] = "kek";
char yet_another_string[] = {'k', 'e', 'k'};
char* heap_string = new char[9];

std::cin >> stack_string; // UB if input is > 3 symbols
std::cin >> heap_string;  // UB if input is > 8 symbols
```

- Где подвох?

Строки

```
const char* static_string = "lol";
char stack_string[4] = "kek";
char yet_another_string[] = {'k', 'e', 'k'}; // не строка! (массив размера 3)
char* heap_string = new char[9];
```

- В строке 3 в отличие от строки 2 нет завершающего нуля!

```
char yet_another_string[] = {'k', 'e', 'k', '\0'}; // теперь строка
```

Строки

Для строк есть удобный интерфейс из большого количества функций для работы с ними, который всем нравится.

`<cstring>`

`std::strcpy` `std::strncpy` `std::strcat` `std::strncat` `std::strxfrm`
`std::strlen` `std::strcmp` `std::strncmp` `std::strcoll` `std::strchr`
`std::strrchr` `std::strspn` `std::strcspn` `std::strpbrk` `std::strstr`
`std::strtok`

Задача

Посчитать количество вхождений символа в строку

```
size_t CountSymbol(const char* str, char symbol) {
    ...
}
```

Задача

Посчитать количество вхождений символа в строку

```
size_t CountSymbol(const char* str, char symbol) {
    size_t counter = 0;
    for (size_t i = 0; i < std::strlen(str); ++i) {
        if (str[i] == symbol) {
            ++counter;
        }
    }
    return counter;
}
```

Задача

Посчитать количество вхождений символа в строку

```
size_t CountSymbol(const char* str, char symbol) {
    size_t counter = 0;
    for (size_t i = 0; i < std::strlen(str); ++i) {  // O(n^2)
        if (str[i] == symbol) {
            ++counter;
        }
    }
    return counter;
}
```

Задача

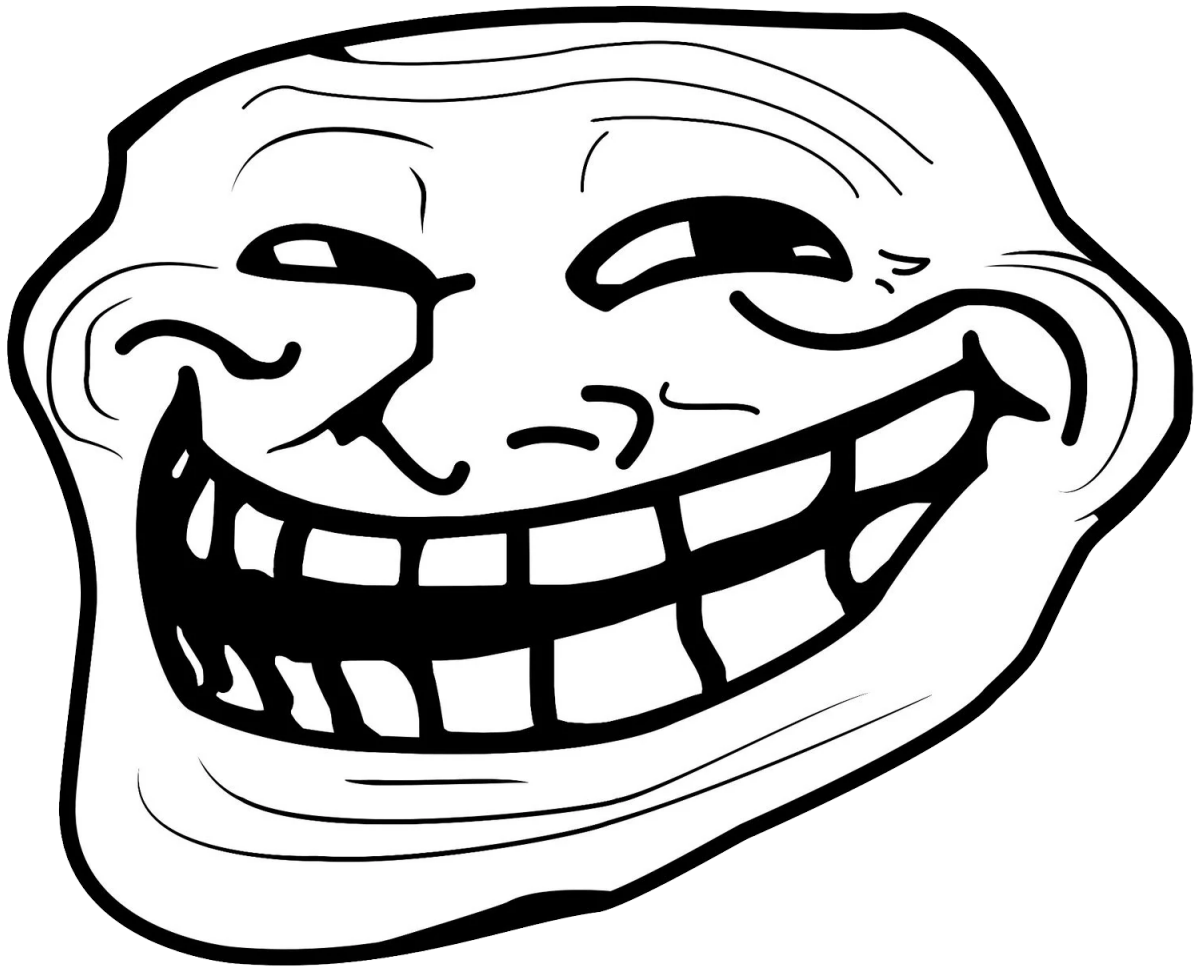
Посчитать количество вхождений символа в строку

```
size_t CountSymbol(const char* str, char symbol) {
    size_t counter = 0;
    for (; *str != '\0'; ++str) {  // O(n)
        if (*str == symbol) {
            ++counter;
        }
    }
    return counter;
}
```

Резюме

- Массивы - отдельный самостоятельный тип данных.
- В большинстве ситуаций низводятся до указателей.
- Строковый литерал - псевдоним статического массива символов.
- Массив `char` $\neq$ строка (у строки обязателен `\0`).
- Работа с C-style строками и массивами доставляет боль, которую, однако, можно преодолеть с помощью ООП...

...но ООП тоже боль.



ВСЁ!