

Низкоуровневые средства C++ для работы с памятью

Курс «Разработка ПО систем управления»

Кафедра управления и информатики НИУ «МЭИ»

Весна 2017 г.

Динамическое выделение памяти

- Выделение блока памяти под 10 целых.

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

- Обращение к элементам блока (массива):

```
*xs == xs[0]
```

```
*(xs + 5) == xs[5]
```

- Освобождение блока:

- **delete[]** xs;

- Выделенное **new** освобождают **delete**, **new[]** — **delete[]**.

- **sizeof(xs) == sizeof(int*) == 8** // или 4

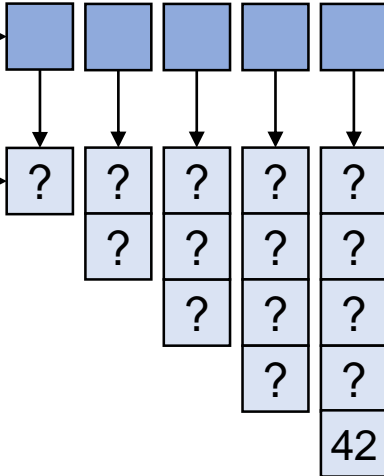
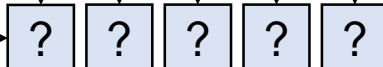
- xs[42] // undefined behavior, но компилируется

Адресная арифметика

- `int xs[10]; // sizeof(int) == 4`
- `xs == &xs == &xs + 0 == &xs[0] == 0 + &xs = 0[xs]`
- `&xs[10] - &xs[5] == (xs + 10) - (xs + 5) == 5`
 - Вычитание указателей на `Type` дает количество элементов типа `Type` между ними.
 - Не количество байт!
- `xs + 1 == &xs[0] + 1 == (&xs + 0) + 1 == &xs[1]`
 - Сложение указателя на `Type` и числа дает указатель, смещенный на размер `Type` (на один `Type`).
 - Не на один байт!

«Рваные» массивы (jagged arrays)

- `int** jagged_array = new int*[5];` `jagged_array` → 

- `for (size_t i = 0; i < 5; ++i) {`
 `jagged_array[i] = new int[i + 1];`
 `}` `jagged_array` → 
 `jagged_array[0]` → 

- На каждый `new[]` нужно `delete[]`:

- Освобождение памяти под каждый элемент:

```
for (size_t i = 0; i < 5; ++i) {  
    delete[] jagged_array[i];  
}
```

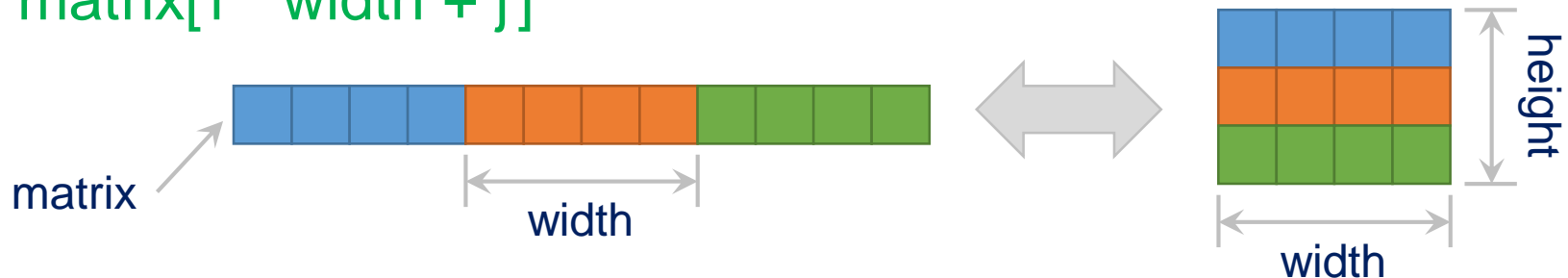
`jagged_array[4][4] = 42`

- Освобождение памяти под массив:

```
delete[] jagged_array;
```

N-мерные дин. массивы

- `size_t width = ...;`
`size_t height = ...;`
`double* matrix = new double[width * height];`
 - `std::vector<double> matrix(width * height);`
- `matrix[i, j]`, `matrix[i][j]` — **неправильно!**
- `matrix[i * width + j]`



- Можно расположить элементы в памяти иначе.
Эффективнее обращаться к памяти последовательно.
 - Например, если обработка идет по столбцам, стоит группировать элементы по ним (column-major).

Встроенные массивы

- **double** data[42];
double table[7][6];
- Размер задается при компиляции и не меняется. Индексация с нуля:
 - data[0]
 - table[0][0] // table[0, 0] — неправильно!
- количество элементов = $\frac{\text{размер всего массива}}{\text{размер одного элемента}}$:
size_t const size = **sizeof**(data) / **sizeof**(data[0]);
- Преобразуются к указателям:
double* start_item_pointer = data;
 - Не копируются:
double mean = get_mean(data, size);
// **double** get_mean(**double*** data, **size_t** size);
 - Массив в составе структуры копируется вместе со структурой.

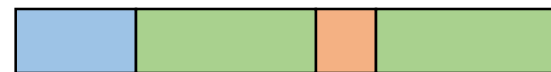
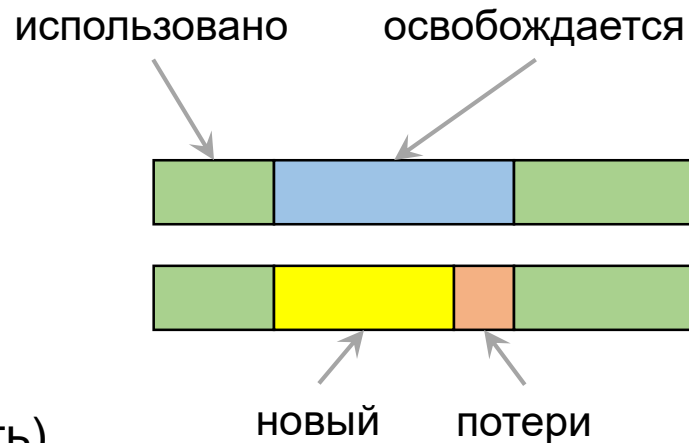
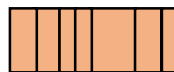
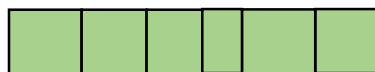
Класс-массив `std::array<T, N>`

- Удобная «обертка» (wrapper) для встроенных массивов:
 - создание объекта не занимает времени
 - (создание вектора требует выделения памяти);
 - поддерживает копирование;
 - можно передавать по ссылке, когда не нужно;
 - можно получить указатель как для массива методом `data()`;
 - поддерживается присваивание;
 - позволяет получить размер методом `size()`;
 - итераторы, проверка индексов, поэлементное сравнение.
- Резюме:
 - «вектор фиксированного размера»;
 - замена простым массивам почти всюду.
- `array<double, 42> data { 1, 2, 3 };`
`cout << data[data.size() / 2];`

Проблемы использования динамической памяти

- Использование адресов
 - переместить объект непросто.
- Время выделения памяти:
 - крайне непредсказуемо;
 - зависит от состояния памяти (нужно найти подходящую область).
- Фрагментация памяти →

Уровень потерь сопоставим с объемом памяти.



Размер типов данных (1)

- Оператор **sizeof** определяет размер в байтах:

```
int value;
```

```
sizeof ( value ) == sizeof ( int ) == 4 // байта
```

- Работает во время компиляции:

- размер объекта-вектора (указатель на данные и число-длина):

```
vector < int > data(10);
```

```
sizeof ( data ) == 8 // возможно
```

- способ определить размер данных в векторе:

```
data . size() * sizeof ( int )
```

Размер типов данных (2)

- Бывает нужно задавать размер точно, обычно когда формат данных задан наперед.
- Есть специальные типы данных (`<stdint>`):
 - `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
- Размер зависит от компилятора и платформы:
 - `sizeof(long int) == 4` // 32 бита (вероятно!)
 - `sizeof(long int) == 8` // 64 бита
- Полагаться на размер чревато ошибками:
 - `0xFFFFFFFF == 0b'11111111'11111111'11111111'11111111`
 - `unsigned long int maximum = 0xFFFFFFFF;`
 - Максимальное возможное значение при 32 битах.
 - При 64 битах — нет (максимальное в 4 млрд. раз больше).
 - `<limits>`, `std::numeric_limits`

Выравнивание (alignment)

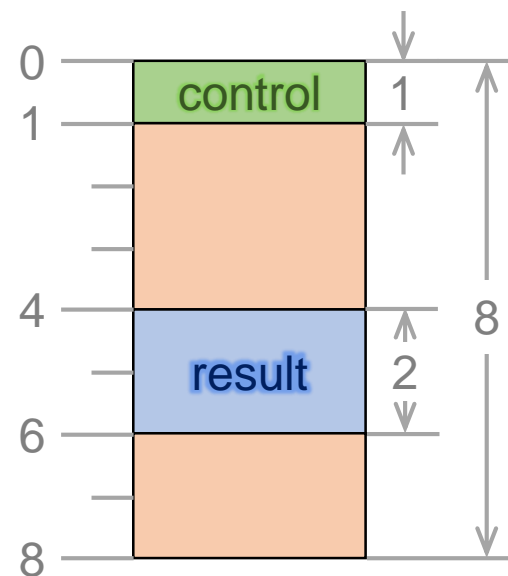
- Явление:

sizeof (uint8_t) == 1

sizeof (int16_t) == 2

sizeof (Device) == 8

- Компилятор располагает данные по адресам, кратным 4 (например); часть памяти не используется.
 - Иногда это работает быстрее (x86).
 - Иногда это необходимо (ARM).
- Иногда это недопустимо!
 - Когда расположение данных (layout) диктуется извне (как для **Device**).
 - В любой компилятор встроены способы отказаться от выравнивания.



#pragma pack (push, 1)

struct Device { ... };

#pragma pack (pop)

Порядок байт (endianness) в представлении целых типов

- $1234_{10} = 04D2_{16}$, $FF_{16} < 04D2_{16} < FFFF_{16} \Rightarrow 2$ байта
- Мы пишем от старших разрядов (04_{16}) к младшим ($D2_{16}$).
- Какой байт в памяти расположен первым? Есть варианты:
 - от младших к старшим (*little-endian*, *LE*, Intel): $D2\ 04$,
 - от старших к младшим (*big-endian*, *BE*, «сетевой»): $04\ 0D$,
 - смешанный (экзотика): $0x12345678 \rightarrow 34\ 12\ 78\ 56$.
- Встречается:
 - процессоры Intel и AMD (ПК, обычные серверы): *little-endian*.
 - процессоры ARM (мобильные устройства):
могут переключать во время работы, обычно *big-endian*.
 - серверы IBM, крупные серверы HP: *big-endian* (обычно).
- При работе с двоичными данными нужно знать **endianness**.
- **число в LE + число в BE = бессмысленное значение**

Оператор `reinterpret_cast`

- Устройство представляется в памяти как набор переменных:

```
struct Device
{
    uint8_t control;
    int16_t result;
};
```

Измерение начинается
при записи в этот байт.

Результат измерения
появляется здесь.



- Известно, что такая структура находится по адресу `0x0300`.

```
Device* device = reinterpret_cast < Device* > ( 0x0300 );
device->control = 1;
double voltage = device->result / 32768.0 * 5; // -5...+5 В
```

- Курс «Технические средства автоматизации и управления» весной.

Побитовые операции

&	И	a	1	0	1	0	1	0	1	0	0xAA
	ИЛИ	b	0	0	0	0	1	1	1	1	0x0F
^	исключающее ИЛИ	a & b	0	0	0	0	1	0	1	0	0x0A
<<	сдвиг влево	a b	1	0	1	0	1	1	1	1	0xAF
>>	сдвиг вправо	a ^ b	1	0	1	0	0	1	0	1	0xA5
~	НЕ	a << 1	0	1	0	1	0	1	0	0	0x54
		b >> 2	0	0	0	0	0	0	1	1	0x03
		~b	1	1	1	1	0	0	0	0	0xF0

©

По мотивам [слайдов](#) Бьярне Страуструпа.



По мотивам [слайдов](#)
Бьярне Страуструпа.

Битовые флаги

Если установлен этот бит, файл можно читать.

- `uint8_t constexpr CAN_READ = 04; // 0b'100`
`uint8_t constexpr CAN_WRITE = 02; // 0b'010`
`uint8_t constexpr CAN_EXECUTE = 01; // 0b'001`
- 
- Разные биты!

- Задание набора флагов логическим «ИЛИ»:

```
uint8_t CAN_EVERYTHING =  
    CAN_READ | CAN_WRITE | CAN_EXECUTE;  
// == 04 | 02 | 01 == 0b'100 | 0b'010 | 0b'001 == 0b'111 == 07
```

- Проверка наличия флага логическим «И»:

```
uint8_t permissions = 05;  
if (permissions & CAN_READ) { ... }  
// 05 & 04 == 0b'101 & 0b'100 == 0b'100 != 0 → true
```

БИТОВЫЕ МАСКИ И СДВИГИ

- Задача: получить биты 4...15 из `uint32_t`.

✓Решение:

- сдвинуть нужные биты к началу числа (в 0...11);
- `full << 4`
- оставить только нужные биты (остальные обнулить).
- `(full << 4) & 0b'1111'1111'1111'0000 // 0xFFFF0`

- Задача: установить 7-й бит в `value`.

✓Решение: `value = value | (1 >> 7);`

- `std::vector<bool>`
- `std::bitset<314>`

Числа с плавающей запятой (floating-point numbers)

- Представлены в памяти нетривиально.
 - IEEE 754: $x = M \cdot 2^E$, M и E целые, и **есть исключения**.
 - Некоторые «простые» десятичные дроби (0,1) нельзя представить точной двоичной дробью.
- Имеют конечную точность.
 - Математически равные результаты, вычисленные по-разному, могут не быть точно (побитово) равны.
 - При операциях над числами разного порядка возможна потеря точности:
 - $1000000.0f + 0.01f == 1000000.0f$
// Копейка рубль бережет, а миллион копейку — нет :-)
 - Практика: не годятся для представления денежных сумм.

Сравнение чисел с плавающей запятой

- Проверка на равенство:

```
float x = 0.3333333f;
```

```
float y = 1.0f / 3.0f;
```

```
if (x == y) // false из-за ошибки округления
```

```
if (abs(x - y) < N * EPS) // Корректно; но что такое N и EPS?
```

- EPS — «машинное эпсилон»,
1.0f + EPS == 1.0f из-за конечной точности.
 - FLT_EPSILON, DBL_EPSILON в <float>.
 - См. курс вычислительной математики (ВМ-2).
- N зависит от способа вычисления x и y,
но на практике выбирают, например, N = 16.

Числа с фиксированной запятой (fixed-point numbers)

- Дробные величины представляют целыми числами (пример: не 1 р. 50 к., а 150 к.).
- Нет потерь точности (у всех чисел она равна).
- Высокая производительность.
- Ограничен диапазон (в т. ч. снизу).
- Пример:
 - `using Money = uint16_t; // Деньги в копейках.`
 - `Money price = 20050; // 200 р. 50 к.`
 - Диапазон: {0, 1 к., ..., 655 р. 35 к.}

Строки C (C-style strings)

Строка C — массив символов, завершающийся нулевым символом `\0`.

```
char greeting[] = "Hello!";
```

- Размер определится автоматически (работает для любых встроенных массивов).
- Длина строки — 6 символов.
- **sizeof**(greeting) == 7
- `// char greeting[7] { 'H', 'e', 'l', 'l', 'o', '!', '\0' };`

```
const char* farewell = "Goodbye!";
```

- **sizeof**(farewell) == 4 // размер указателя
- Длина строки — 8 символов, где-то в памяти их 9.

Обработка строк C

```
size_t get_string_length ( const char* symbols )
```

```
{
```

```
    size_t length = 0;
```

```
    while ( *symbols ) {
```

```
        ++length;
```

```
        ++symbols;
```

```
    }
```

```
    return length;
```

```
}
```

Разыменование дает символ,
на который указывает *symbols*.
Если это `'\0'`, условие ложно.

Смещение указателя
к адресу очередного символа.

- × Если `symbols == nullptr`, нельзя делать `*symbols`.
- × $O(\text{length})$

Копирование строк C

```
void copy_string(char* to, const char* from)
{
    while (*from) {
        *to = *from;
        ++to;
        ++from;
    }
    *to = *from;
    // while (*to++ = *from++);
}
```

1) Пока есть символ для копирования,
2) копировать его
3) и перейти к следующей ячейке для копии,
4) а также к следующему исходному символу.
5) Скопировать нулевой символ.

Предполагается, что массив, на который указывает `to`, достаточно велик, чтобы вместить символы из `from`.

Проверить это в `copy_string()` **нельзя**.

Робота со строками

Клас `std::string`

```
string name, message;  
const string greeting = "Hello";  
getline ( cin, name );
```

```
message = greeting;  
message += ", " + name + "!";
```

```
cout << message << '\n';
```

Строки C (`<cstring>`)

```
char name[32], message[32];  
const char* greeting = "Hello";  
fgets ( name, sizeof(name), stdin );
```

// gets() небезопасна!

```
strcpy ( message, greeting );
```

```
strcat ( message, ", " );  
strcat ( message, name );  
strcat ( message, "!" );
```

```
puts ( message ); // cout << ...
```

48

Литература к лекции

- *Programming Principles and Practices Using C++:*
 - глава 25 — тема лекции;
 - раздел 27.5 — строки C;
 - аналогичная [презентация](#) (скорее, наоборот :-).
- *C++ Primer:*
 - разделы 3.5 и 3.6 — подробно о массивах.
- *Сайт «C++ Reference»:*
 - функции для работы с памятью и строками;
 - ограничения типов с плавающей запятой;
 - описание `std::array`, `std::vector <bool>`, `std::bitset`.
- [Статья](#) о плавающей запятой.