

# **Низкоуровневые средства С++ для работы с памятью**

Курс «Разработка ПО систем управления»

Кафедра управления и информатики НИУ «МЭИ»

Весна 2018 г.

# Модель памяти C++

Независимо от физического исполнения ОЗУ ЭВМ и от организации памяти, предлагаемой ОС, с точки зрения языка C++ память ЭВМ всегда описывается одинаково.

Это описание называется *моделью памяти C++*

Наименьшая единица памяти в C++ — байт. Память, доступная программе на C++, состоит из последовательностей байт (ненулевой длины), называемых областями памяти (memory location). У каждого байта памяти есть уникальный адрес в памяти, который принято записывать как шестнадцатеричное число с префиксом «0x»: 0xFA23BC12

# Стек и куча (динамическая память)

*Область стека (stack)* - хранение значений переменных, использование компилятором при вызове функций и для промежуточных вычислений.

Размер каждой области памяти, отводимый под переменную на стеке, известен на этапе компиляции. Количество памяти стека ограничено компилятором (порядка одного или нескольких Мб). Работает по принципу обоймы: последний зашел - первый вышел (Last In First Out - LIFO).

*Куча (динамическая память, heap)* - это область, части которой могут быть зарезервированы (выделены, allocated) программой во время выполнения для размещения данных, а также удалены (освобождены, deallocated), когда они больше не требуются.

Размер областей памяти, выделяемых в куче, может быть известен только на этапе выполнения.

Фактически, куча – это вся оперативная память.

# Указатели

Указатель – переменная, значением которой является адрес ячейки памяти

```
int var = 123; // инициализация переменной var числом 123
int *ptrvar = &var; // указатель на переменную var (присвоили адрес
переменной указателю)
cout << "&var  = " << &var << endl; // адрес переменной var содержащийся
в памяти, извлечённый операцией взятия адреса
cout << "ptrvar = " << ptrvar << endl; // адрес переменной var, является
значением указателя ptrvar
cout << "var    = " << var << endl; // значение в переменной var
cout << "*ptrvar = " << *ptrvar << endl; // вывод значения содержащегося в
переменной var через указатель, операцией разыменования указателя int
*nptr = nullptr; // нулевой указатель, который не соответствует никакой
ячейке памяти
```

```
&var = 0x22FF08
ptrvar = 0x22FF08
var = 123
*ptrvar = 123
```

# Ссылки

Ссылки – особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разименовывается

```
int var = 123; // инициализация переменной var числом 123
int &refvar = var; // ссылка на переменную var
cout << "var   = " << var << endl;
cout << "refvar = " << refvar << endl;
refvar += 877; // Изменяем переменную посредством ссылки
cout << "var   = " << var << endl; // новое значение в переменной var
cout << "refvar = " << refvar << endl;
```

```
var = 123
refvar = 123
var = 1000
refvar = 1000
```

# Отличия указателей от ссылок

Использование указателей и ссылок схоже. Указатели предоставляют больше возможностей для косвенной адресации, но ссылки использовать часто проще и безопаснее.

Отличия указателей от ссылок:

- 1) Не существует «нулевой ссылки», подобной нулевому указателю.
- 2) Не существует ссылок на ссылки, указателей на ссылки и массивов ссылок
- 3) Не бывает переменных типа `void`, поэтому не бывает и ссылок на `void`.
- 4) Разыменование ссылки не имеет смысла (ссылка — не адрес, а псевдоним).
- 5) У ссылки может не быть собственного адреса, поэтому операция взятия адреса ссылки срабатывает для переменной, на которую ссылаются.
- 6) Ссылки отсутствуют в языке C.

# 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

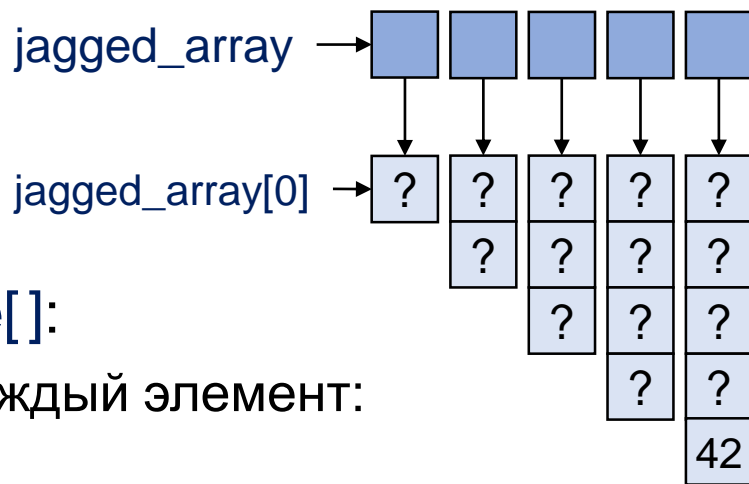
# Адресная арифметика

- `int xs[10]; // sizeof(int) == 4`
- `xs == &xs == &xs + 0 == &xs[0] == 0 + &xs`
- `&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];`  
    `}`    

- На каждый `new[]` нужно `delete[]`:

- Освобождение памяти под каждый элемент:

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

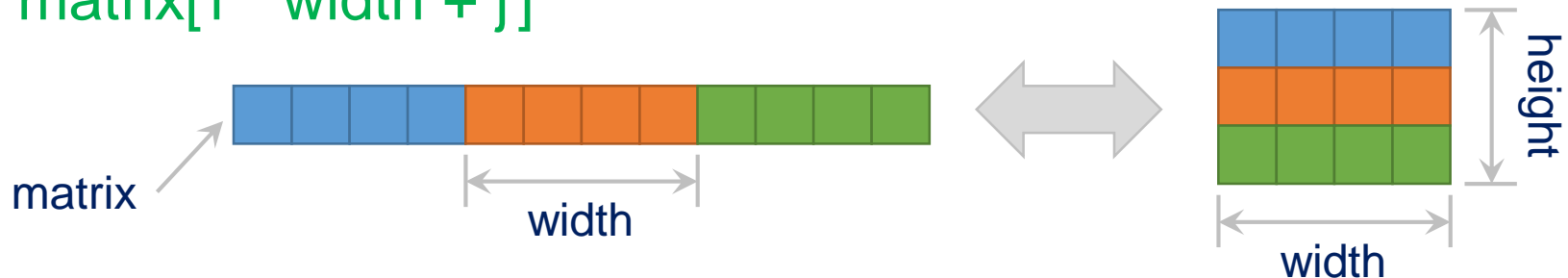
`jagged_array[4][4] = 42`

- 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.

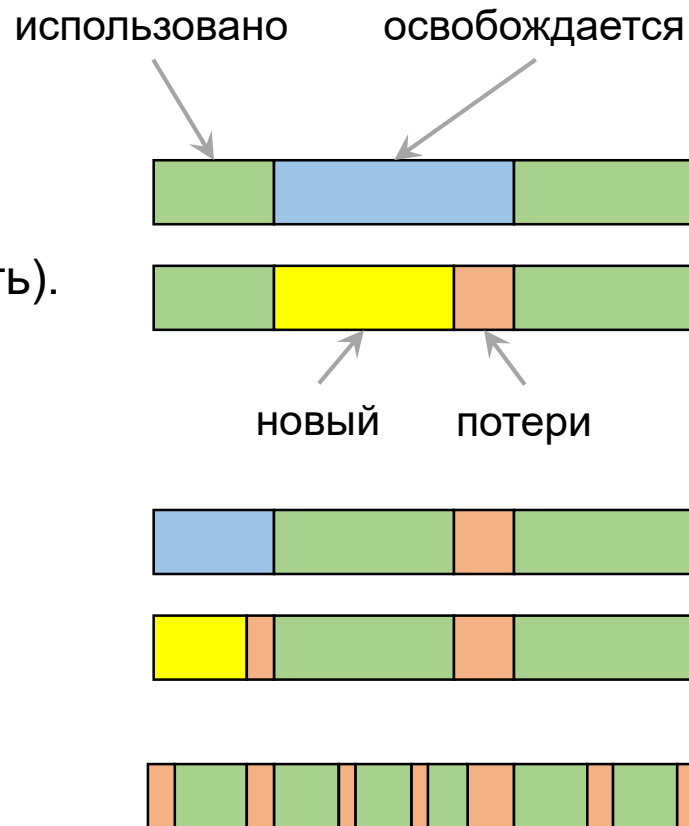
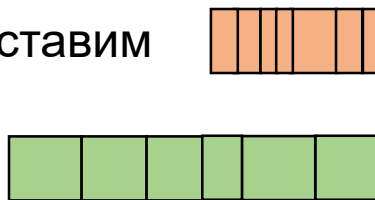
# Класс-массив `std::array<T, N>`

- 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

- 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 ) == 12 // возможно 8
```

- 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).

# Выравнивание (alignment)

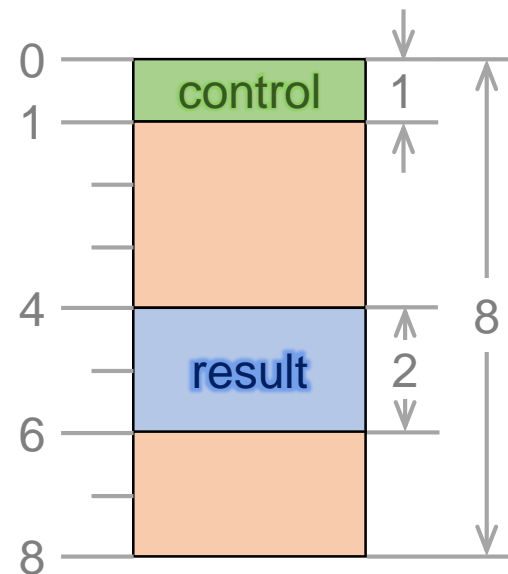
- Явление:

**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 {  
    uint8_t control;  
    uint16_t result;};
```

```
#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»):  $04\ D2$ ,
  - 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**