

Программирование низкоуровневых задач

Курс «Технология программирования»

Кафедра управления и информатики НИУ «МЭИ»

Осень 2015 г.

Низкоуровневые задачи

- Системное программное обеспечение:
 - взаимодействие с ОС;
 - драйверы устройств.
- Встраиваемые системы (embedded):
 - программирование без ОС;
 - написание ОС.
- Взаимодействие с кодом на других языках:
доступны только простые типы данных и указатели.
- Программирование обмена данными:
 - сетевые приложения;
 - разбор двоичных форматов файлов.

Требования и специфика задач

- **Предсказуемость (predictability):**
 - режим реального времени (= известное, ≠ малое)
 - жесткое (hard) — к нужному времени код должен отработать;
 - мягкое (soft) — то же самое, изредка допустимы нарушения.
- **Безопасность (safety):**
 - последствия ошибок катастрофичны;
 - «цена» ошибки >> цены разработки.
- **Надежность (robustness):**
 - корректность работы в любых режимах;
 - возможен отказ оборудования, резервирование.
- **Экономичность (resource management):**
 - ресурсы бывают очень ограниченными;
 - при долгой работе недопустимы утечки.

Уступки при разработке

- Правила написания кода:
 - простота чтения;
 - code review, парное программирование;
 - «страховка» от ошибок:
 - неизменяемость,
 - имена переменных и функций.
- Ограничения на языковые средства:
 - динамическое выделение памяти,
 - исключения,
 - рекурсия.
- Специальные стандарты безопасного кодирования:
 - JFK+
 - MISRA/C

Преимущества C++

- **Предсказуемость:**

- ✓ почти все предсказуемо,
- ✗ кроме **new**
- ✗ и исключений (**throw**).

- **Безопасность:**

- большой опыт безопасного кодирования;
- языковые средства (контроль типов и т. п.)
 - ✗ часть из них небезопасна!

- **Надежность:**

- автоматическое управление ресурсами (RAII);
- ✓ переносимость,
 - как следствие: перекрестная проверка языка и библиотек.

- **Экономичность:**

- ✓ zero-overhead principle;
- доступ к низкоуровневым конструкциям;
- доступ к адресам памяти, работа с указателями.

Динамическая память

- Выделение:

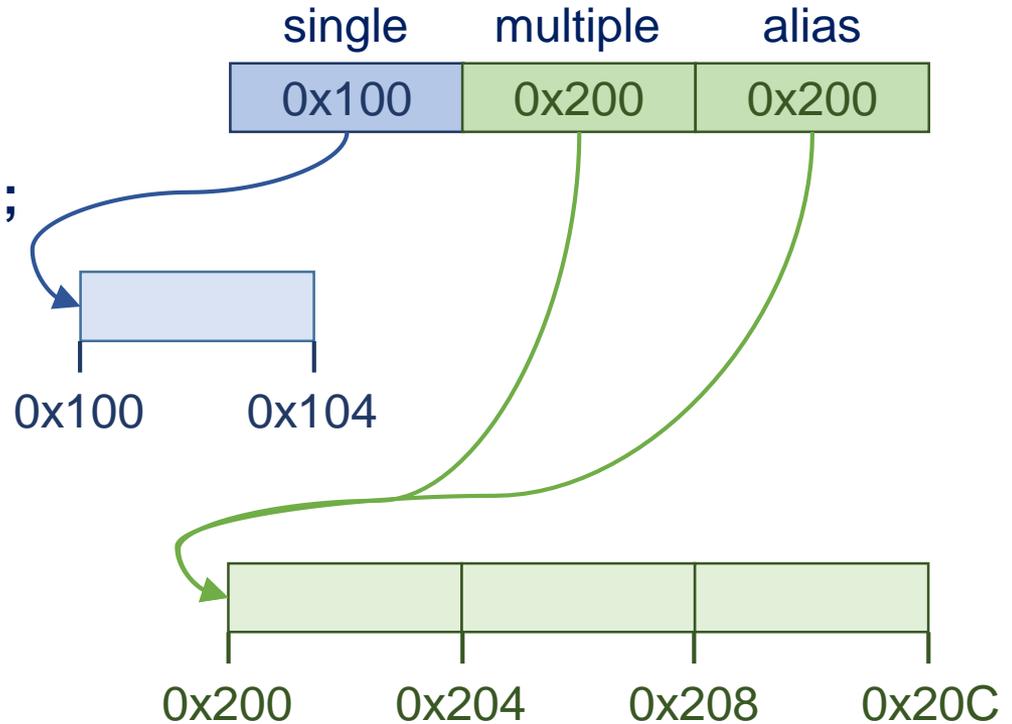
```
int *single = new int;  
int *multiple = new int[3];  
int *alias = multiple;
```

- Освобождение:

```
delete single;  
delete [] multiple;
```



- **delete** для одиночной ячейки;
- **delete []** для массива.
- По указателю нельзя узнать, адрес области какого размера он содержит.



Доступ по указателю

- Как к вектору: `multiple[0]`, `multiple[1]`, `multiple[3]`
 - Выход за границы не проверяется!
- К начальному элементу:
 - `multiple == &multiple[0] == multiple + 0`
 - `*multiple == multiple[0] == *(multiple + 0)`
- Адресная арифметика:
 - `multiple + 0 == &multiple[0] // 0x200`
`multiple + 1 == &multiple[1] // 0x204`
 - `multiple[2] - multiple[0] == 2`
`== (multiple + 2) - (multiple + 0)`
 - `*(single + 42) == single[42] // 0x100 + 4 × 42`
 - Корректность обращения не проверяется!

Пример: устройство вектора

- Создание:
 - `Vector numbers(10);`
 - Без конструктора `data` останется неинициализированным.
- Доступ к элементам:
 - `numbers.data[5] = 42;`
 - `numbers[50]`
- Добавление элементов:
 - Область памяти нельзя расширить, можно только выделить новую.

```
struct Vector {  
    size_t size;  
    double* data;  
    Vector (size_t size);  
};  
  
Vector::Vector (size_t size) {  
    this -> size = size;  
    data = new double [ size ];  
}
```

Добавление элемента в **Vector**

```
void add_to_vector ( Vector& xs, double x ) {
```

1. Выделить новую область памяти:

```
double* new_data = new double [ xs . size + 1 ];
```

2. Скопировать в нее содержимое старой:

```
for ( size_t i = 0; i < xs . size; ++i)  
    new_data [ i ] = xs . data [ i ];
```

3. Дописать в нее же новый элемент:

```
    new_data [ xs . size ] = x;
```

4. Удалить старую область памяти:

```
    delete [ ] xs . data;
```

5. Заменить `data` и `size`:

```
    xs . data = new_data;  
    xs . size++;
```

```
}
```

Управление ресурсами

- **Проблема:** ресурсы нужно освобождать
 - самостоятельно;
 - в правильном порядке (обратном захвату);
 - обязательно,
 - при любом ходе исполнения;
 - даже в случае ошибки или исключения.
- **Цель:** автоматическое освобождение ресурсов
 - в любом случае;
 - в правильном порядке.
- **Что нужно:**
вызов кода освобождения ресурса,
когда он перестает требоваться.

Деструкторы

- Специальные методы у структур: `~Type()`.
- Вызываются, когда структура удаляется из памяти:
 - закрывающая `}` блока или функции, где объявлена переменная;
 - конец программы для глобальных переменных;
 - перед удалением содержащей структуры.

```
struct Inner { };  
struct Outer { Inner inner; };  
Outer outer;
```

 - При уничтожении `outer` сначала уничтожается `outer . inner`.
 - при выходе из блока (функции) по исключению.
- Вызываются в порядке, обратном созданию.

Вызов деструктора

```
struct Example {  
    int id;  
    Example (int id) {  
        this->id = id;  
    }  
    ~Example () {  
        cout << "~Example() "  
            << id << '\n';  
    }  
};
```

```
Example e1 {1};  
  
int main() {  
    Example e2{2};  
    if (...) {  
        Example e3{3};  
    }  
    cout << "main() ends\n";  
}
```

Результат:

```
~Example() 3  
main() ends  
~Example() 2  
~Example() 1
```

Автоматическое управление ресурсами (RAII)

- Resource Acquisition Is Initialization:
 - После занятия (acquisition) ресурс передается во владение объекту прямо в конструктор (initialization).
 - При разрушении объект освобождает ресурс в своем деструкторе.
- Обычно ресурс нельзя копировать.
 - Пример: файл, устройство, сетевое подключение.
 - Достаточно запретить копирование объекта-владельца.
- Объект обязательно будет уничтожен,
 - если он не создан **new** явно
(в современном C++ **new** используется только в «системном» коде),
 - поэтому утечка ресурса невозможна.

С

RAII в действии

```
Receiver* source = get_receiver();  
if (!source)  
    return;  
Transmitter* sink = get_transmitter();  
if (!sink) {  
    shutdown ( source );  
    return;  
}  
Coder* coder = create_coder();  
if (!coder) {  
    shutdown ( source );  
    shutdown ( sink );  
    return;  
}  
shutdown ( source );  
shutdown ( sink );  
shutdown ( coder );
```

Автоматически
вызывает shutdown();
проверяет ресурс (как cin).

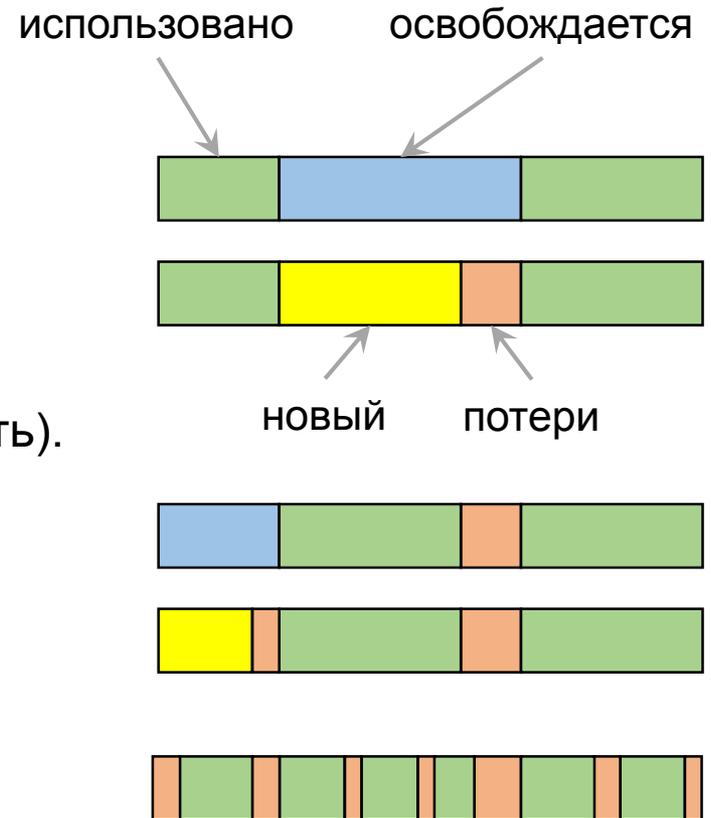
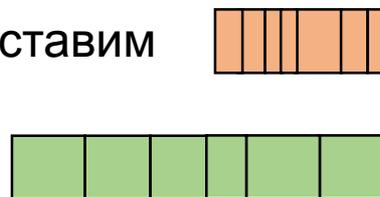
С++

```
Receiver source { get_receiver() };  
if (!source)  
    return;  
Transmitter sink { get_transmitter() };  
if (!sink)  
    return;  
Coder coder { create_coder() };  
if (!coder)  
    return;
```

Проблемы использования динамической памяти

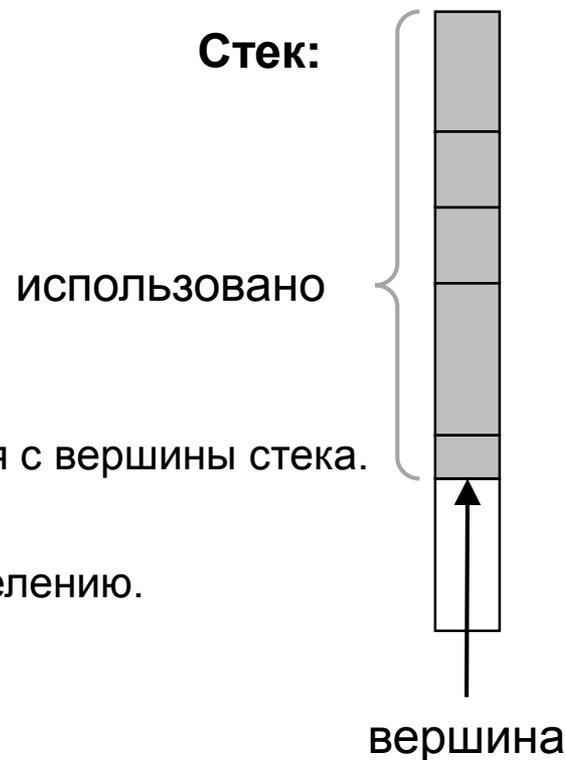
- Использование адресов
 - переместить объект непросто.
- Время выделения памяти:
 - крайне непредсказуемо;
 - зависит от состояния памяти (нужно найти подходящую область).
- Фрагментация памяти →

Уровень потерь сопоставим с объемом памяти.

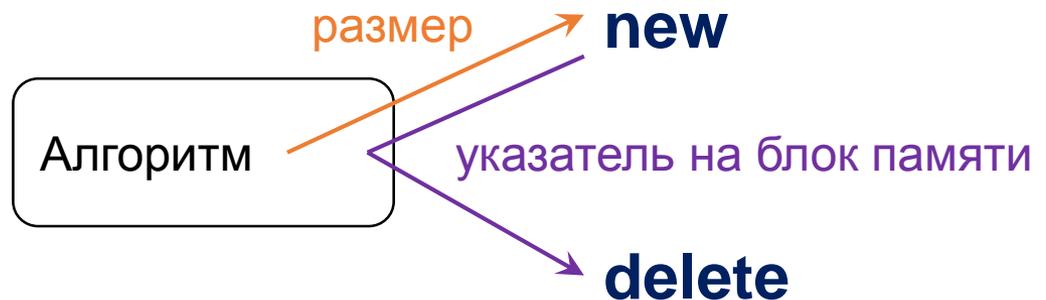


Борьба с фрагментацией памяти

- Общее решение:
 - выделить крупный блок 1 раз,
 - обычно в начале работы;
 - использовать фрагменты блока.
- Примеры:
 - Искусственный стек:
 - Фрагменты выделяются и освобождаются с вершины стека.
 - Размер фрагментов — произвольный.
 - Освобождение в порядке, обратном выделению.
 - Пул объектов (pool):
 - Фрагменты фиксированного размера.
 - Порядок выделения и освобождения — произвольный.



Распределители памяти



Проблемы:

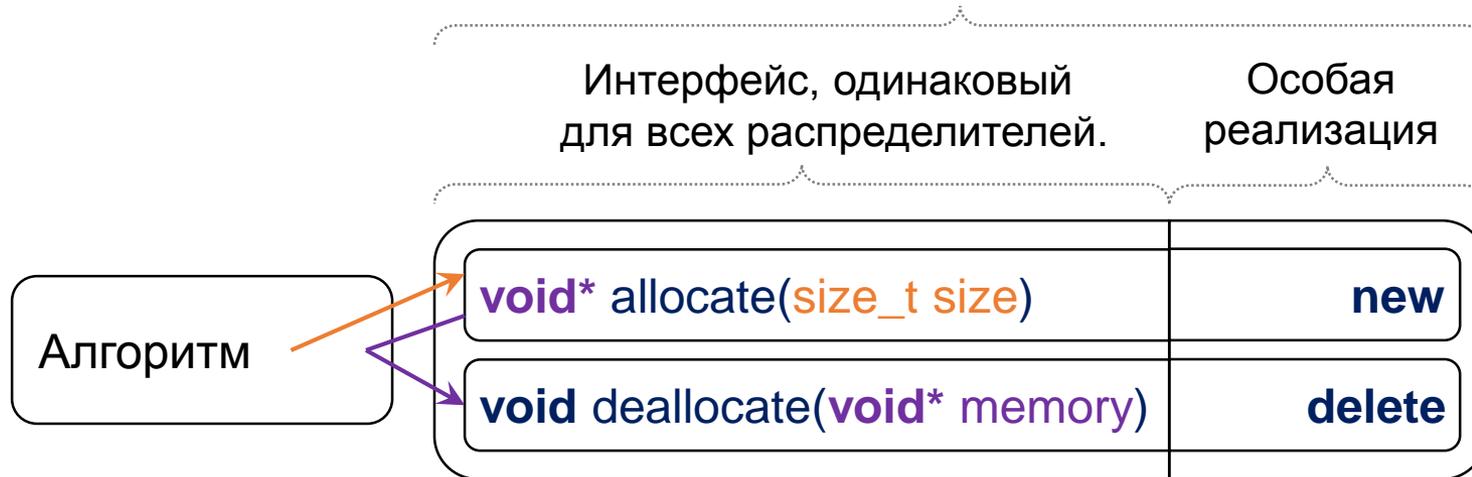
- Алгоритм жестко зависит от **new** и **delete**.
Чтобы использовать пул или стек, нужно менять код алгоритма.
- Чтобы в одном случае использовать **new** и **delete**, а в другом — пул, нужно дублировать алгоритм и изменять копию.
- Код распределения памяти нужно дублировать в каждом алгоритме.

Цель:

- Написать алгоритм независимым от способа распределения памяти.
- Выделить код способа распределения памяти в набор функций.
- В месте использования задавать способ (стратегию) как параметр.

Распределители памяти

Оформленный стандартным образом способ распределения памяти — *распределитель (allocator)*.



- ✓ Способ распределения памяти записан **один раз** в реализации.
- ✓ Можно заменять распределитель, **не изменяя алгоритм.**
 - ✓ Выбирать распределитель можно **в момент привлечения алгоритма.**



Размер типов данных

- 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

- Бывает нужно задавать размер точно, обычно когда формат данных задан наперед.
- Есть специальные типы данных (`<cstdint>`):
 - `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).

Побитовые операции

&	И	a	<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	1	0	1	0	1	0	1	0	0xAA
1	0	1	0	1	0	1	0					
 	ИЛИ	b	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	0	0	0	1	1	1	1	0x0F
0	0	0	0	1	1	1	1					
^	исключающее ИЛИ	a & b	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	0	0	1	0	1	0	0x0A
0	0	0	0	1	0	1	0					
<<	сдвиг влево	a b	<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	0	1	0	1	1	1	1	0xAF
1	0	1	0	1	1	1	1					
>>	сдвиг вправо	a ^ b	<table border="1"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	1	0	1	0	0	1	0	1	0xA5
1	0	1	0	0	1	0	1					
~	НЕ	a << 1	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	0	1	0	1	0	0	0x54
0	1	0	1	0	1	0	0					
		b >> 2	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	0	0	0	0	0	0	1	1	0x03
0	0	0	0	0	0	1	1					
		~b	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	1	1	1	0	0	0	0	0xF0
1	1	1	1	0	0	0	0					

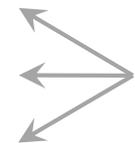


По мотивам [слайдов](#)
Бьярне Страуструпа.

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>`

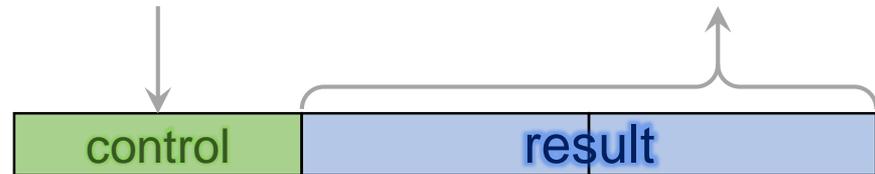
Оператор `reinterpret_cast`

- 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.



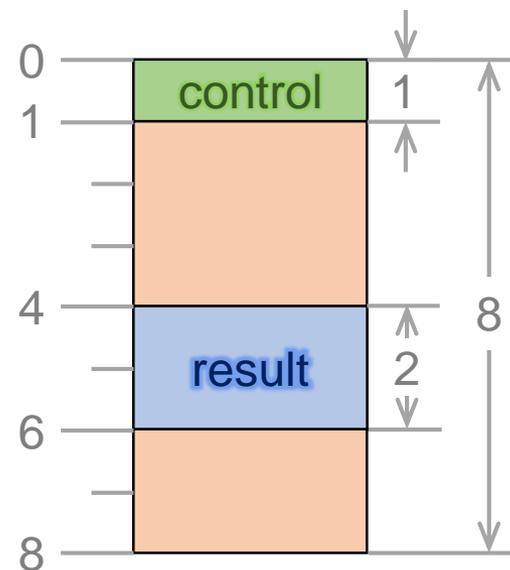
- Известно, что такая структура находится по адресу `0x0300`.

```
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.

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 )
```

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];`

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 `*symbols`.
- × Время работы пропорционально длине строки.

Копирование строк C

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

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

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

Работа со строками

Класс `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";  
  
gets ( name );  
  
strcpy ( message, greeting );  
  
strcat ( message, ", " );  
strcat ( message, name );  
strcat ( message, "!" );  
  
puts ( message ); // cout << ...
```

48

Имеется аналогичный набор функций для копирования памяти и т. д.

Литература к лекции

- *Programming Principles and Practices Using C++:*
 - глава 25 — тема лекции;
 - раздел 27.5 — строки C;
 - аналогичная [презентация](#) (скорее, наоборот :-)
 - [презентация](#) о написании аналога `std::vector<T>`.
- *C++ Primer:*
 - разделы 3.5 и 3.6 — подробно о массивах.
- *Сайт «C++ Reference»:*
 - функции для работы с памятью и строками C;
 - описание `std::array`, `std::vector<bool>`, `std::bitset`;
- *Доклады о низкоуровневом программировании в аэрокосмической отрасли:*
 - [использование стандарта JSF+](#) для бортового ПО самолетов;
 - особенности [написания ПО для марсохода](#).