

# C++

Наташа Мурашкина

7 января 2017 г.

## Содержание

<b>1. Лекция 14.10.16</b>	<b>1</b>
1.1 Стандартная библиотека . . . . .	1
1.2 (3) String.h . . . . .	1
<b>2. Практика 14.10.16</b>	<b>3</b>
2.1 Структура XML . . . . .	3
2.2 Библиотека Expat . . . . .	3
<b>3. ООП. 21.10.16</b>	<b>4</b>
3.1 C++ . . . . .	4
3.2 Три взгляда на ООП . . . . .	4
3.2.1 Исторический . . . . .	5
3.2.2 Лингвистический . . . . .	5
3.2.3 Практический . . . . .	5
<b>4. Ссылки</b>	<b>7</b>
4.1 Практика 21.10.16 . . . . .	7
<b>5. Лекция 28.10.16</b>	<b>8</b>
5.1 Const . . . . .	10
<b>6. Лекция 11.10.16</b>	<b>13</b>
6.1 Вспомним про const . . . . .	13
6.2 Перегрузка операторов . . . . .	13
<b>7. Лекция 18.11.16</b>	<b>17</b>
7.1 Перегрузка операторов — продолжение . . . . .	17
7.2 Будет в тесте . . . . .	18
7.3 Smart pointer . . . . .	19
7.4 Unique ptr, auto ptr . . . . .	20
7.5 Shared pointer . . . . .	20

<b>8. Лекция 25.11.16</b>	<b>22</b>
8.1 Наследование . . . . .	22
8.2 Будет в тесте . . . . .	23
8.3 Практика . . . . .	23
<b>9. Лекция 02.12.16</b>	<b>25</b>
9.1 Наследование — продолжение. Protected . . . . .	25
9.2 Динамическое связывание . . . . .	25
9.3 Таблица виртуальных методов . . . . .	26
9.4 Пример из жизни. Бухгалтер . . . . .	27
9.5 Практика . . . . .	29
<b>10. Лекция 09.12.16</b>	<b>30</b>
10.1 Зависимости . . . . .	30
10.2 Model-View . . . . .	30
10.3 Доп. ДЗ на CursesView . . . . .	31
10.4 Статические переменные. Ключевое слово static . . . . .	32
10.5 Unit-тестирование, автотесты . . . . .	32
<b>11. Лекция 16.12.16</b>	<b>35</b>
11.1 Static . . . . .	35
11.2 Пример: шаблон проектирования Singleton (Mayer) . . . . .	35
11.3 Разрешение имён . . . . .	36
11.4 Ключевое слово inline . . . . .	37
11.5 Классы inline . . . . .	38
11.6 Immutable . . . . .	38
<b>12. Модификаторы доступа</b>	<b>40</b>
<b>13. Лекция 23.12.16</b>	<b>41</b>
13.1 Стандартная библиотека C++ . . . . .	41
13.2 Конструкция namespace . . . . .	41
13.3 iostream . . . . .	41
13.4 Разбор теста на Gaussian Number . . . . .	43
13.5 Про экзамен . . . . .	43

# 1. Лекция 14.10.16

## 1.1. Стандартная библиотека

Рассмотрим интересную функцию.

```
1 | char *pend;
2 | int i = strtol(s, &pend, 10); // string to long, 10 - base, &pend - указатель, где
   | закончилось успешно
3 | if (pend == s) // Если преобразования не случилось
```

Почему указатель передаётся так странно? Т.е. передаётся его адрес.

```
1 | void get_odd(int *a, size_t s, int **b, size_t * new_size) {
2 |     // Осторожно, код может быть нанутан std::sort
3 |     int count = 0;
4 |     for(i=0;i<s;i++) {
5 |         if(a[i]%2==1)
6 |             count++;
7 |     }
8 |     *b = malloc(sizeof(int) * count);
9 |     *new_size = count;
10 |    count = 0;
11 |    for(i=0;i<s;i++) {
12 |        if(a[i]%2==1)
13 |            (*b)[count++] = a[i];
14 |    }
15 | }
16 |
17 | int a[] = {1,10,15,13,12,1,18,3};
18 | int *b;
19 | size_t ns;
20 | get_odd(a, 7 * sizeof(int), &b, &ns); // Хотим записать в a все нечётные числа
21 | free(b);
```

Какие у нас проблемы, если мы передаём просто указатель `b` и просто `ns`? Ни `b`, ни `ns` не перезаписываются как нужно (только локально в функции) и ещё ворох проблем. Поэтому передаём указатель на `b` и указатель на `ns`.

## 1.2. (3) String.h

Обсудили:

```
1 | memcpu
2 | strcpy
3 | strncpy
4 | strcat
5 | strchr
6 | strstr
7 | strlen
```

И ещё всё что пожелаете. Читайте на `cplusplus`, `cplusplusreference`.

```
1 | char * strtok(char *str, const char *delim); // Возвращает указатель на очередной
   | токен.
2 | "Hello world" // Пробел - символ-разделитель (delimiter)
3 |
```

```
4 | char str[] = "Hello world word";
5 | char *pch = strtok(str, " ");
6 | while(pch != NULL) {
7 |     printf("%s\n", pch);
8 |     pch = strtok(NULL, " "); // strtok продолжает работать с запомненной строкой.
9 | }
10 | // Вывод: "Hello
    |         nworld
    |         word"
```

При вызовах: pch на начале строки, на первом символе второго токена (перед ним ставится нолик, т.е. '0'), etc. Где-то должна быть глобальная переменная!

```
1 | strtok...()
2 | char *start; // где мы закончили в предыдущем вызове, чтобы с него начать
```

## 2. Практика 14.10.16

### 2.1. Структура XML

Поговорили о структуре XML на примере ([https://github.com/HFX-TA/cpp/blob/master/lab\\_06/pbook.xml](https://github.com/HFX-TA/cpp/blob/master/lab_06/pbook.xml)).

### 2.2. Библиотека Expat

Разобрали функцию `parse_xml` ([https://github.com/HFX-TA/cpp/blob/master/lab\\_06/expat\\_example.c#L28](https://github.com/HFX-TA/cpp/blob/master/lab_06/expat_example.c#L28)).

## 3. ООП. 21.10.16

В след. семестре: обобщённое программирование (шаблоны) и что-то ещё.

### 3.1. C++

C++, 198X, Bell Labs, Bjarne Stroustrup

Компьютеры можно было покупать обычным компаниям. Компьютеры использовались для автоматизации (в частности, документооборота). Нужно перенести много информации с бумаги в электронные версии, каждый кусок информации — это сущность (например, товар) с множеством полей: название, артикул, итд.

```
1 || g++
```

В C++ приведение явнее, чем в Си.

```
1 || int *pi;
2 || char *pc;
3 || pc = pi; // Нельзя (?)
4 || pc = (char*)pi;
```

Что ещё нового?

```
1 || // overloading (перезгрузка)
2 || int min(int a, int b)
3 || int min(int a, int b, int c)
4 || double min(double a, double b)
5 || // В Си не могут существовать три функции с одинаковым названием.
6 || // mangling - сигнатура преобразуется
7 ||
8 || // В Си функции идентифицируются по именам: min, min, min
9 || // В C++ - по имени+типам аргументов: min_int_int, min_int_int_int, min_double_double

1 || // В Си
2 || int *a = malloc(sizeof(int) * 100);
3 || free(a);
4 ||
5 || // В C++
6 || int *a = new int[100];
7 || ...
8 || delete []a;
```

Динамическая (python, perl) vs статическая (Си, C++) типизации.

С дин-типизированным языком проще разобраться, но

В C++ ключевое слово для дин. выделения памяти стало частью языка (new).

Обычно в ячейке слева от начала массива new записывает количество элементов в массиве, а delete считывает эту информацию. (Объяснение на след. лекции.)

### 3.2. Три взгляда на ООП

Первое название C++: Си с классами.

Таблица 1: Бла

Процедурные	ООП
нарисуй дом	дом нарисуйся
покрась дом	дом покрась себя
перенести дом	дом перенестись

### 3.2.1. Исторический

bin code → asm → if, for, int a; → func() → struct → class

asm: move, add, load, store, регистры, адреса в памяти, цикл: if + goto

```

1 | // class
2 | int x, y, z;
3 | func1() {}
4 | func2() {}

```

Каждое слово языка высокого уровня — это комбинирование asm-операций. Следующий шаг: комбинируем в функции. Следующий шаг: комбинируем переменные вместе. Следующий шаг: комбинируем функции вместе (те, которые работают с одними и теми же переменными).

Жаргон. Класс — это «тип». Переменная этого класса (этого типа) — объект.

### 3.2.2. Лингвистический

Си — процедурный

Когда всё перейдёт в двоичный код, ничего не отличить ООП-программу от процедурной. Поговорим о том, как

### 3.2.3. Практический

Две роли: автор-программист класса и пользователь-программист класса.

**Инкапсуляция.** Чтобы пользоваться микроволновкой, не нужно знать, как она работает. Пользователю доступны две ручки, остальное закрыто защитным кожухом. В классе две части: private и public.

**Наследование.** Расширить, дописать класс. В машинах были только печки, потом добавили кондиционер, потом климат-контроль. Можно купить готовую библиотеку и уметь приклеить к ней что-то сбоку.

**Полиморфизм.** Заменить один блок на другой. Соглашение о возможности заменить блок — это интерфейс. В одну розетку можно вставлять разные устройства (утюг, чайник).

(Лекции по C++ — время поупражняться в перерисовывании блоков с ушками с доски в тех.)

Перерыв.

```

1 | // возможные ошибки
2 | void () {
3 |     int *a = new mt[10]; // 1) забыть создать
4 |     a[i] = s; // 2) выйти за пределы
5 |     delete []a; // 3) забыть удалить
6 | }

```

Хотим предотвратить совершение этих ошибок программистом-пользователем.

```

1  // my_array.h
2  class my_array {
3  private:
4      int *array;
5      size_t size;
6  public:
7      my_array(size_t s); // constructor, решим ошибку 1
8      ~my_array(); // destructor, решим ошибку 3
9      int get(size_t index);
10     void set(size_t index, int value);
11 };
12
13 my_array arr(10);
14 arr.array; // не скомпилился, т.к. array - приватное поле
15 arr.size; // не скомпилился
16 arr.set(i, 5); // ок

1  // my_array.cpp
2  #include "my_array.h"
3  my_array::my_array(size_t s) {
4      size_s;
5      array = new int [size];
6
7  }
8
9  my_array::~~my_array() {
10     delete []
11 }
12
13 int my_array::get(size_t index) {
14     if (i >= 0 && i < size) {
15         return array[i];
16     }
17     else {
18         return -1; // в массиве лежал -1 или это ошибка? Поговорим в след. семестре про
19                    // exception, а сейчас закроем глаза на такую некрасивость.
20     }
21 }
22 // + set
23
24 // + get_size

```

За такую красоту мы заплатили вызовами функций.

```

1  void f() {
2      my_array arr()
3
4  }

```



## 4. Ссылки

```
1 swap(int *a, int *b) {
2     int t = *a;
3     *a = *b;
4     *b =
5 }
6
7 int c = 3;
8 int d = 4;
9 swap(&c, &d);
10
11 // То же самое с ссылками (это синт. сахар)
12 void swap(int &a, int &b) {
13     int t = a;
14     a = b;
15     b = t;
16 }
17
18 int c = 3;
19 int d = 4;
20 swap(c, d);
```

Вопрос будет в тесте. В языке Си писать или не писать в заголовочном файле объявление функций — дело пользователя. В C++ это обязательно, в связи с появлением ссылок. Вопрос почему?

### 4.1. Практика 21.10.16

Инкапсуляция, наследование, полиморфизм (, абстракция).

Инкапсуляцию можно обойти. (Злоумышленник (или энтузиаст) может получить доступ к приват-полям.)

C++ = Си+ язык символов (??). (Немножко притянута за уши.)

В Си тоже возможно реализовать инкапсуляцию, ораque pointer.

## 5. Лекция 28.10.16

Нет общей темы, будем закрывать дырки.

```

1  int
2  short
3  long
4
5  stdint.h // Набор typedef'ов
6  int8_t // INT8_MIN - мин. значение переменной такого типа
7  uint8_t // UINT8_MIN
8  int16_t
9  uint16_t
10 // Если на платформе физически отсутствует возможность хранить такой размер (из-за длины
    // регистра), то прога не скомпилируется.
11
12 ctype.h
13 int isalpha (char c) // Является ли символ c буквой
14 int isdigit (char c)
15 tolower()
16
17 assert.h // Будет отдельная лекция про обработку ошибок
18 // 2 типа ошибок
19 // 1. По вине программиста, runtime error
20 int a[10];
21 a[10] = 3;
22
23 char *p = NULL;
24 strlen(p);
25
26 // 2. По вине окружения
27 // не можем выделить память с помощью malloc, т.к. закончилась память
28 // не открыть файл, т.к. его нет
29
30 // Поговорим об ошибках типа 1.
31
32 size_t strlen(char *s) {
33     assert(s != NULL); // Если выражение false, вызовется abort()
34     ...
35 }
36
37 #ifdef NDEBUG
38     ; // все асерты заменяются на точку с запятой
39 #else
40     ; // какая-то проверка
41     abort()
42 #endif
43
44 // При компиляции продакшн-кода (уже для пользователя) добавить ключ -D со значением NDEBUG
45 // gcc -DNDEBUG
46
47 FILE *f = fopen("m.txt", "r");
48 assert(f != NULL);
49 // Эта проверка исчезнет, когда скомпилим на продакшн
50
51 // 4)
52 math.h
53 cos
54 sin
55 sqrt

```

```

56
57 // 5)
58 time.h
59 // Время хранится как количество секунд, прошедших с 1 января 1970
60 time_t
61 time(time_t *t);
62 time_t a;
63 a = time(NULL); // или
64 time(&a);
65
66 // Как замерить время работы программы?
67 time_t start = time(NULL);
68 f();
69 time_t end = time(NULL);
70 // Разрешающая способность time() - секунда, поэтому прогу нужно запустить несколько раз в
    // цикле, чтобы вывести среднее время одного запуска.
71
72 // clock() мерит время в тактах
73 // Подходит только для однопоточных программ.
74 clock_t start = clock();
75 f();
76 clock_t end = clock();
77
78 // Если прога многопоточная, на одном процессоре будет  $t_1$  такт, на другом -  $t_2$ 
79 // Реально прошло  $\max(t_1, t_2)$ , а clock() посчитает сумму:  $t_1 + t_2$ 
80
81 // C++
82 // 1)
83 typedef struct {
84     int *array;
85     size_t size;
86 } my_array_t;
87
88 // 2)
89 my_array_t a;
90 set(&a, 0, 5);
91
92 // 3)
93 void set(my_array_t *int, size_t idx, int value);
94
95 // C++
96 // 1)
97 class my_array {
98 private:
99     int * array;
100    size_t size;
101 public:
102    void set(size_t inx, int value);
103 }
104
105 // 2)
106 my_array a;
107 a.set(0, 5);
108
109 // 3)
110 void my_array::set(size_t idx, int value);
111
112 // Компилятор преобразует первое во второе:
113 a.set(0, 5) → my_array_set_...(&a, 0, 5);
114
115 // Как однозначно определяются типы, если они подчеркнуты? См. утилиту objdump

```

```

116 // См. name mangling из пред. лекции
117 void my_array::set(size_t, int) → my_array_set_my_array*_size_t_int(my_array *this
    , size_t idx, int value);
118
119 // В объектном коде не остаётся никакого ООП.
120
121
122 // Помогаем компилятору aka выстреливаем себе в ногу aka делаем что-то ужасное.
123 // На уровне бин. кода или инструкций процессора нет приватности, вот как её обойти:
124 my_array a;
125 char *pa = (char *)&a;
126 int s = *(pa + 8);

```

## 5.1. Const

```

1 // По стандарту 2001 года
2
3 // 1)
4 const double pi = 3.14; // По стандарту нужно double const pi, но так тоже можно.
5 class A {
6     const double e;
7 }
8 A::A() {
9     e = 2.7;
10 }
11 pi = 4; // Ошибка во время компиляции
12
13 // Зачем? Самому подстраховаться + документация кода.

```

```

1 // Указатели и константы.
2
3 char s1[] = "Hello";
4 char s2[] = "World";
5
6 // По стандарту, const защищает то, после чего он находится
7 char const *p1 = s1; // <=> const char *p1;
8 p1[0] = 'A'; // Error
9 p1 = s2; // Ok
10
11 char * const p2 = s1;
12 p2[0] = 'A'; // Ok
13 p2 = s2; // Error
14
15 char const * const p3 = s1;
16 // Error
17 // Error
18
19 // Мы даём гарантию, что не разрушим строку.
20 size_t strlen(const char *s1) {
21     s1 = 333; // Ok
22     s1[1] = 'a'; // Error
23     char s1[] = "Hello"; // Всё ещё ок
24 }
25
26 main () {
27     char s[] = "Hello";
28     strlen(s);
29 }
30

```

```

31 // А здесь строка будет разрушена
32 str_tok(char *s);
33
34
35 char *sz = (char *) s1;
36 sz[0] = 'A'; // ok
37 s1[0] = 'A'; // Error
38
39 // Преобразовывать типы - это как сломать об коленку.
40
41 // Одна интересная особенность
42 char *s = "Hello"; // Warning, нужно препенднуть const
43 char s[] = "Hello";
44 s[0] == 'H';
45 s[1] == 'e';
46
47 char *ss = (char *) s;
48 ss[0] = 3; // Падение проги
49
50 // va (virtual address) ra (real address) pid rw
51
52 // Для обычного массива так, кажется, нельзя. Можно попробовать дома.
53
54 // 4)
55 // C
56 swap(int *a, int *b) {
57     int t = *a;
58     *a = *b;
59     *b = t;
60 }
61
62 // C++
63 swap(int &a, int &b) {
64     int t = a;
65     a = b;
66     b = a;
67 }
68
69
70 // 5)
71 class my_array {
72     private:
73         int *array;
74         size_t size;
75     public:
76         my_array(size_t s);
77         void set(size_t idx, int value);
78         int get(size_t idx) const; // Не меняет полей класса
79         size_t get_size() const;
80 }
81
82 // Если объект передан по константной ссылке, можно вызывать только константные методы.
83 void print (const my_array & a) {
84     a.get(i); // ok
85     a.set(0, 3); // Error
86 }

```

В тесте (скорей всего) будет: если в одной функции параметр инт, а в перегрузке — указатель на инт (Илья спросил на занятии). Будет ошибка или нет?

```
1 | #include <cstdio>
```

```
2
3 void do_magic(int number) {
4     printf("by value: %d\n", number);
5     return;
6 }
7
8 void do_magic(int & reference) {
9     printf("by reference: %d\n", reference);
10    return;
11 }
12
13 int main() {
14     int num = 5;
15
16     do_magic(5); // ok: prints "by value: 5"
17     do_magic(num); // error: call of overloaded 'do_magic(int&)' is ambiguous
18
19     return 0;
20 }
```

Далее — оператор присваивания, конструктор копи.

## 6. Лекция 11.10.16

Далее — оператор присваивания, конструктор копи.

### 6.1. Вспомним про const

```
1 || void f(const person *p); // передаём по указателю, чтобы сэкономить память, не копируя
    // дату
2 || void f(const person &p); // ?? Из ссылки получается указатель?
```

### 6.2. Перегрузка операторов

Мотивация. Есть парадигма ООП.

Вспомним перегрузку функций.

Перегрузка — возможность написать (в C++) функции с одинаковым названием, но с разными параметрами.

В Си такое было невозможно, а в C++ — да, т.к. компилятор именуется функции.

```
1 || // 1) Функции
2 || f(int);
3 || f(int, double);
4 ||
5 || // 2) Операторы
6 || // * +; &&; &; ==, <>
7 || // (type) = конструктор Сору
8 || // [] *p ->
9 || // Оператор точка . перегрузить нельзя
10 ||
11 || BigInt
12 || // Обычная операция + задана для int + int
13 || // Хотим переопределить + для складывания BigInt + BigInt
14 ||
15 || // [6][5][5][3][0][ ] - память, в которой храниться BigInt: один десятичный разряд в каждой
    // ячейке. (Это пример для иллюстрации, вообще эффективнее было бы хранить в системе по
    // основанию размера ячейки).
16 ||
17 || BigInt
18 ||     int * array;
19 ||     size_t size;
20 ||
21 || // 1)
22 || BigInt a(30);
23 || BigInt b(a); // Хотим создать объект b на основе уже существующего объекта a
24 ||
25 || // 2) f(a);
26 ||
27 || void f(BigInt obj) {
28 ||     BigInt a(30);
29 ||     BigInt b(a);
30 || }
31 || // Был объект a:
32 || // array -> [ ][ ][ ][ ][ ][ ]
33 || // size
34 || // Создалась полная копия a:
35 || // array (указывает на тот же array, что и a)
36 || // size
```

```

37
38 // Эта программа упадёт при вызове f, т.к. деструктор. При выходе из f вызовется деструктор a,
    // удалит память под массив. При выходе из main Вызовется b, захочет также удалить память под
    // массив, а её-то уже нет!
39
40 // С point будет ок
41 class point {
42     int x;
43     int y;
44 }
45
46 // В BigInt проблемы, т.к. у класса есть ресурсы.
47
48 // Исправляемся.
49
50 class BigInt {
51     public:
52     BigInt (const BigInt & obj) {
53         // Можно и без this писать, он добавлен для наглядности
54         this->size = obj.size;
55         this->array = new int [this->size];
56         for(int i = 0; i < this->size; i++) {
57             this->array[i] = obj.array[i];
58         }
59     }
60 }
61
62 // Если бы не было †, компилятор сделал бы сам так:
63
64 BigInt(const BigInt & obj) {
65     size = obj.size;
66     array = obj.array;
67 }
68
69 // Если нет конструктора или деструктора, скомпилируются такие по умолчанию:
70 BigInt() {}
71 ~BigInt() {}
72
73 // Уточнение. Поля были приватные. Но объекты одного и того же класса могут получать доступ к
    // приватным полям друг друга.
74
75 // b. конструктор_копии(a);
76 // Обычно при создании копии изменять оригинал не надо. (На сл. занятии рассмотрим случаи,
    // когда надо.)

```

### Приведение типов

```

1 // 1) Приведение из int в BigInt
2 BigInt a = 3;
3 BigInt a = (BigInt)3;
4
5 // 2) Приведение из BigInt в int
6 BigInt a(30);
7 int b = a;
8
9 // Разберём 1)
10
11 BigInt(int)
12 // Построится временный объект tmp типа BigInt
13 // Потом вызовется конструктор копирования tmp в a (но компилятор оптимизирует и на самом деле
    // вызовется только a(3))
14

```



```

15 | // Теперь какая-то жест непонятная про конструкторы и приведение с ацкими нелнейными
    | // примерами на доске, я такое не могу конспектировать
16 |
17 | BigInt b(3); // ок
18 | BigInt b = 3; // не ок
19 |
20 | SquareMatrix a = 3; // Запутывает: делаем матрицу 3x3, а ожидается матрица, заполненная
    | // тройками.
21 |
22 | //
23 | f() {
24 |     BigInt a(30);
25 |     a.set(0, 5);
26 |     a.set(1, 6);
27 |     BigInt b(50);
28 |     b.set(0, 7);
29 |     b.set(1, 8);
30 |     b = a;
31 | }
32 |
33 | // Было раньше:
34 | BigInt a(30);
35 | BigInt b(a);
36 |
37 | // Сейчас
38 | // Если не написать доп. код, компилятор просто копирует поля.
39 | // Нужно перегрузить оператор присваивания.
40 |
41 | // b.operator=(a);
42 | // По умолчанию, такое же, как у копирования.
43 | // Пишем наш. 1) Удалить (свою) старую память. 2) Выделить новую память. 3) Скопировать
44 | BigInt& operator=(const BigInt & obj) {
45 |     if (&obj != this) {
46 |         delete [] array;
47 |         size = obj.size;
48 |         array = new int [size];
49 |         for (size_t i = 0; i < size; i++) {
50 |             array[i] = obj.array[i];
51 |         }
52 |     }
53 |     return *this;
54 | }
55 | // Идиома swap: можно было бы сделать delete, потом конструктор. (Но нам нужен ещё какой-то
    | // промежуточный шаг, обсудим позже)
56 |
57 | // Всё сломается, если присвоить себя себе
58 | a = a;
59 |
60 | // Вторая проблема, тройное присваивание
61 | a = b = c;
62 | // Означает:
63 | a.operator=(b.operator=(c));
64 | // У оператора присваивания должно быть возвращаемое значение
65 |
66 | // Неприятное следствие
67 | if (a = 5) {} // эквивалентно
68 | if (5) {}

```

Возврат объектов из функций.

```

1 | // 1) Функция возвращает BigInt, заполненный 100 единичками.
2 | BigInt getOnes() {

```

```
3 |     BigInt a(100);
4 |     for () {
5 |         a.set(i, 1);
6 |     }
7 |     return a;
8 | }
9 |
10 | main () {
11 |     BigInt p = getOnes();
12 |     p.get(3);
13 | }
14 |
15 | // Будет некорректно, как с указателями, так и со ссылками (ссылки - это просто сахар к
16 | // указателям), т.к. вызовется деструктор по выходе из функции getOnes.
17 | // [RA][RV][----]
18 | // RV - return value
```

То, что делает компилятор, тоже ок. Правило трёх. Если вашему классу нужен деструктор, значит нужен конструктор присваивания и конструктор копирования.

## 7. Лекция 18.11.16

В следующий раз будет тест + разбор старого теста.

Закончим перегрузку операторов, обсудим умные указатели (smart pointer).

### 7.1. Перегрузка операторов — продолжение

Мы хотим сделать наш чёрный ящик как можно более похожим на обычную переменную.

```

1 // 1)
2 BigInt a;
3 f(a); f(BigInt br);
4
5 // Виды ресурсов:
6 int **;
7 FILE *f;
8
9 BigInt b(a);
10
11 // 2
12 BigInt a;
13 BigInt b;
14 a = b; // эквивалентно a.operator=(b);
15
16 // 3
17 BigInt a;
18 BigInte b;
19 a += b; // А в Java, например, перегрузка операторов отсутствует.
20 BigInt c = a + b;
```

Начнём с плюсиков.

```

1 BigInt& operator+=(const BigInt& o) {
2     for () {
3         this->array[i] += o.array[i];
4         // сделать переносы, бла-бла-бла
5     }
6     return *this;
7 }
8
9 // Вариант 1
10 BigInt operator+(const BigInt& o) {
11     BigInt t(o);
12     t += (*this);
13     return t;
14 }
15
16 // Вариант 2
17 BigInt operator+(BigInt o) {
18     return o += (*this);
19 }
20
21 // Примечание: оба кода выше не работают с коммутативными операциями
22
23 BigInt c = b + 3; // 3 переходит в BigInt(3);
24 BigInt c |= 3 + b; // b типа BigInt не может перейти в int
25
26 // Можно определить оператор вне класса.
```

```

27
28 BigInt operator+(const BigInt& o1, const BigInt& o2) {
29     BigInt t(o);
30     t += o2;
31     return t;
32 }

```

Оператор ++

```

1  int a = 3;
2  int d = a++; // d == 3
3  // или
4  int e = ++a; // e == 4
5
6
7  BigInt d = a++;
8  BigInt e = ++a;
9
10
11 class BigInt {
12     BigInt(int r);
13     BigInt operator++() { // ++ префиксный
14         *this += 1;
15         return *this;
16     }
17     BigInt operator++(size_t n) { // постфиксный ++
18         BigInt t(*this);
19         *this += 1; // или ++ *this;
20         return t;
21     }
22 }

```

Логические операторы. < > == != <= >=

```

1  // Можно выразить все операторы через один оператор, например, <.
2
3  bool operator<(const BigInt &o) {
4      for ()
5          if(this->array[i](o.array[i])) blah;
6  }
7
8  bool operator>(const BigInt &o) {
9      return o < *this;
10 }
11
12 // Аналогично
13 !(o < *this) && !(*this < o) // изящно, но медленнее

```

Оператор []

```

1  int operator[](size_t index) {
2      return array[index];
3  }

```

Далее: библиотека algorithm в stl, перегрузка круглых скобок ().

## 7.2. Будет в тесте

```

1  int c = b[0];
2  matrix m(3, 5);
3  m[0][0] = 1;

```

Но нет оператора `[][]`. Как это реализовать? (Возможно, понадобится вспомогательный класс.)

Решение. Перегрузить `operator[]`, чтобы он возвращал объект, к которому можно снова применить `operator[]`, чтобы получить результат.

(<http://stackoverflow.com/questions/6969881/operator-overload>)

```

1 | class ExternalArray {
2 | public:
3 |     ExternalArray() {
4 |         _external = new int*[3];
5 |         _external[0] = new int[42];
6 |         ...
7 |     }
8 |
9 |     class Proxy {
10 | public:
11 |     Proxy(int* _array) {
12 |         _array = _array
13 |     }
14 |
15 |     int operator [] (int index) {
16 |         return _array[index];
17 |     }
18 | private:
19 |     int* _array;
20 | };
21 |
22 | Proxy operator [] (int index) {
23 |     return &Proxy(_external[index]);
24 | }
25 |
26 | private:
27 |     int ** _external;
28 | };
29 |
30 |
31 | ExternalArray ea;
32 | ea[3][5];

```

### 7.3. Smart pointer

```

1 | Person {
2 |     char name [256];
3 |     char * phone [20] [25];
4 |     int age;
5 |     char encl [256];
6 | }
7 |
8 | Person *p = new Person();
9 | Person *p1 = new Person(*something);
10 |
11 | delete p;
12 |
13 | // 2
14 | Person *p = new Person [105]; // для такого нужно иметь конструктор по умолчанию
15 | delete [] p;
16 |
17 | // Объекты Person раскиданы по памяти
18 | Person ** p = new Person* [100];
19 | for (i = 0; i < 100; i++) {
20 |     p[i] = new Person(i);

```

```

21 }
22
23 // Про память. Prefetch. Конвейер. Cache.
24
25 class scoped_ptr {
26     private:
27         Person * p;
28     public:
29         scoped_ptr(Person *p) { this->p = p; }
30         ~scoped_ptr() { delete this->p; }
31         Person* ptr() { return p; }
32         Person* operator->() { return p; }
33         Person& operator*() { return *p; }
34     private:
35         scoped_ptr(const scoped_ptr p) // для запрещения копирования
36         operator=
37 }
38
39 // Person: bool hasBirthday()
40
41 if () {
42     scoped_ptr p (new Person(Vasya));
43     p.ptr()->hasBirthday();
44     // или с перезагрузкой:
45     p->hasBirthday(); // p.operator->()->hasBirthday();
46
47     *p.hasBirthday();
48
49     scoped_ptr p1 = p; // Дважды вызовется деструктор, нужно запретить такое
50
51     printPerson(p);
52 }

```

## 7.4. Unique ptr, auto ptr

```

1 unique_ptr {
2     public:
3         unique_ptr(unique_ptr & o) {
4             this->p=o->p;
5             o->p=NULL;
6         }
7         // можно подстраховаться в деструкторе, чтобы не сделать delete NULL: if(p) delete p;
8     }
9
10    if () {
11        unique_ptr p(new Person("Vasya"));
12        unique_ptr p1(p);
13        unique_ptr p2(p1);
14    }
15
16    f(unique_ptr & p); // Функция, которую можно вызвать два раза

```

## 7.5. Shared pointer

Стратегия reference count. Считаем, сколько указателей на объект существует. Когда выполняется деструктор одного из указателей, счётчик указателя уменьшается на 1. Когда счётчик равен нулю, вызывается деструктор.

```

1
2 shared_ptr

```

```
3 | storage * stor;  
4 |  
5 | if () {  
6 |     shared_ptr p1(new Person("Vasya"));  
7 |     shared_ptr p2(p1);  
8 | }
```

## 8. Лекция 25.11.16

### 8.1. Наследование

Мотивация. В 2015 году мы написали некий класс. В 2016 году нам понадобился такой же класс, но с перламутровыми пуговичками. Мы хотим дописать уже существующий, а не

Чтобы сохранить место на доске, две особенности: 1) не бывает пустым, 2) хитрый деструктор.

```

1 // 1
2 struct linked_list
3     node *head
4
5 // 2
6 struct node
7     int value
8     node *next
9
10 class list {
11     private:
12         int value
13         list *next
14     public:
15         list (int v);
16         ~list (); // ?? написать дома
17         size_t length();
18         void add(int v);
19     private:
20         list (const list&)
21         list operator=(const list &)
22 }
23
24 list::list (int v) {
25     value = v;
26     next = NULL;
27 }
28 void list::add(int v) {
29     list *cur = *this;
30     while (cur->next != NULL) {
31         cur = cur->next;
32     }
33     cur->next = new list(v);
34 }
35 size_t list::length() {
36     // «Встать на голову, завести переменную счётчик»
37     list *cur = this;
38     size_t count = 0;
39     while (cur->next != NULL) {
40         cur = cur->next;
41         count++;
42     }
43     return count;
44 }
```

Настал 2016 год, хотим надстроить класс list. Хотим дописать только новое, изме.

list — базовый класс (base) или суперкласс (superclass). double\_list — производный (derived) или наследник ().



overload — перегрузка (функций) — было раньше override — перекрытие (методов)

```

1 | int value;
2 | list *next;
3 | list *prev;
4 |
5 | class double_list:public list {
6 |     private:
7 |         list *prev;
8 |     public:
9 |         double_list(int v);
10 |         ~double_list();
11 |         void add(int v);
12 |     private:
13 |         double_list(const double_list &&);
14 |         double_list& operator= () {...}
15 | }
16 |
17 | double_list::double_list(int v):list(v) { // вызываем конструктор суперкласса
18 |     prev = NULL;
19 | }
20 |
21 | void double_list::add(int v) {
22 |     double_list *cur = this;
23 |     while (cur->next != NULL) {
24 |         cur = cur->next;
25 |     }
26 |     // Туны: int* u double_list*
27 |     cur->next = new double_list(v);
28 |     // Tun cur->next - list*
29 |     cur->next->prev = cur;
30 | }
31 |
32 | main() {
33 |     double_list dl;
34 | }

```

При конструировании производного класса сначала проинициализируются поля базового класса, а потом нужно доинициализировать новые поля.

Проблемы.

1) Наследник обращается к приватным полям суперкласса. 2) Для перекрытия есть специальное слово `virtual`.

## 8.2. Будет в тесте

Реализовать нормальный деструктор для класса `list`. От лектора: «Цикл не цикл, рекурсия не рекурсия».

## 8.3. Практика

Идиома RAII — Resource acquisition is initialization

Пример: shared pointer, mutex

```

1 | class foo {
2 |     ~Foo() {}
3 |     std::shared_ptr<baz> baz;
4 |     int * bar; // Освободить руками

```

5 || }

Что плохого можно сделать с `shared_ptr`.

$A \leftrightarrow B$  — объекты `A` и `B` ссылаются только друг на друга, и поэтому не смогут удалиться.  
Возможные решения: сборщики мусора и слабые ссылки.

Слабая ссылка (`weak_ptr`). Ссылаемся на объект, но не владеем им.

Шаблон проектирования `observer`.

## 9. Лекция 02.12.16

### 9.1. Наследование — продолжение. Protected

```

1 | list
2 |     protected:
3 |         int value
4 |         list * next
5 |         add_value
6 |
7 | double_list
8 |     add_value(int v) {
9 |         xxx;
10 |    }
```

Protected — модификатор доступа для полей класса, позволяющий доступ не только объектам класса, но и наследникам.

```

1 | list* l = new list(...);
2 | l->value = 35; // ok
3 | l->head = NULL; // комп. error, несовпадение типов
4 | l->length();
5 | l->remove_value(5);
```

Приведение типов.

```

1 | int * a = new ...
2 | char * b = new ...
3 | a = (int*)b;
4 | // или
5 | b = (char*)a;

1 | // next - типа list*
2 | cur->next = new double_list(..)

1 | // Поля list: value, next
2 | // double_list - наследник list, добавлено поле: prev
3 |
4 | list* l = new list(..);
5 | double_list * dl = new double_list(..);
6 |
7 | l = dl; // ok
8 | l->next
9 | l->value
10 |
11 | dl = l; // error, опасное неявное приведение типов
12 | dl->prev
13 | dl->next
14 | dl->value
```

Объект производного класса является объектом исходного класса. Производный объект можно использовать так, как будто это объект суперкласса.

### 9.2. Динамическое связывание

```

1 | // 2015
2 |
3 | void fill(list *l, int v, int num) {
```

```

4 |     for () {
5 |         l->add_value(v); // в момент компиляции нужно проставить адрес на функцию: call
   |         list_add_value
6 |     }
7 | }
8 |
9 | // 2016
10 | main() {
11 |     double_list dl(3);
12 |     list l(5);
13 |     fill(&dl, 6, 10);
14 |     fill(&l, 6, 10);
15 |
16 |
17 | }

```

Чтобы использовать дин. связывание, необходимо использовать слово `virtual`. Оно распространяется на всех потомков.

```

1 | list
2 |     protected:
3 |         int value
4 |         list * next
5 |         virtual add_value // появилось virtual
6 |
7 | double_list
8 |     add_value(int v) {
9 |         xxx;
10 |    }
11 |
12 | list *l = new list(..)
13 | double_list *dl = new double_list(..)
14 | l->add_value(..)
15 | dl->add_value(..)
16 |
17 | list *l1 = dl;
18 | l1->add_value(..) // хотим, чтобы вызвалась функция l, а не dl

```

Перерыв.

### 9.3. Таблица виртуальных методов

```

1 | list *l1 = new ..
2 | list *l2 = new ..
3 | double_list *dl = new ..

```

Рассмотрим такой код:

```

1 | void fill() {
2 |     l->add_value(v); // здесь произойдёт динамическое связывание
3 | }

```

Если у класса есть виртуальная функция, в начале класса будет автоматически добавлена *таблица виртуальных методов*, с которой компилятор будет консультироваться. Проставляется такая инструкция: во время выполнения возьми адрес из начала объекта, по нему расположена таблица, возьми из неё адрес нужной версии функции `add_value` и вызови её по этому адресу.

При статическом связывании, проставляется одна инструкция: вызови функцию с этим конкретным именем. При динамическом связывании проставляется три инструкции: 1. найти в таблице, 2. найти функцию, 3. вызови функцию.

Если у функция `add_value` не будет задано ключевое слово `virtual` в классе `list`, в примере ниже всегда будет вызываться метод `add_value` из `list`.

```

1 | list *l;
2 | stand(time(NULL));
3 | int n = rand();
4 | if (n > 0) {
5 |     l = new list(5);
6 | }
7 | else {
8 |     l = new double_list(6);
9 | }
10 | l->add_value(7);

```

## 9.4. Пример из жизни. Бухгалтер

Для бухгалтера нужно написать программу, считающую зарплату для разных типов сотрудников в зависимости от специфичных для них параметров. Класс должен быть готов к расширению.

### 1. Developer

- salary
- level
- isRelease
- bonus

### 2. Seller

- price
- num
- percent

### 3. Tester

### 4. HR

### 5. Marketing

```

1 | Worker
2 | Worker () {
3 |     char * name = new ..;
4 | }
5 | ~Worker() {
6 |     delete [] name;
7 | }
8 | virtual int get_salary() = 0; // чисто виртуальная

```

Хотим явно сказать компилятору, что функция `get_salary` должна быть перекрыта. Это называется чисто виртуальная функция. Если наследник не перекроет эту функцию, он не скомпилируется.

```

1 | WorkerDataBase db;
2 | Developer *d = new Developer("Masha", 1000, 5, False, 300);
3 | Seller *p = new Seller("Petya", "Windows", 0.03, 10000);

```

```

1 | WorkerDatabase {
2 |     Worker* workers[100];
3 |
4 |     void add(worker *w) {
5 |         ..
6 |         workers[size] = w;
7 |         ..
8 |     }
9 |
10 |    int getTotalSalary() {
11 |        int salary = 0;
12 |        for (i = 0; i < size; ++i) {
13 |            salary += worker[i]->get_salary(); // у каждого типа сотрудника будет вызываться
14 |                свой метод
15 |        }
16 |        return salary;
17 |    }
18 | }

```

Рассмотрим класс Developer.

```

1 | class Developer:public Worker {
2 |     int salary;
3 |     int level;
4 |     bool isRelease;
5 |     int bonus;
6 |
7 |     Developer(..):Worker(name) {...}
8 |
9 |     get_salary() {
10 |         int ret = salary * level;
11 |         if (isRelease) ret += bonus;
12 |         return ret;
13 |     }
14 | }

```

Рассмотрим класс Seller. Поговорим о деструкторе.

```

1 | class Seller:public Worker {
2 |     char * product;
3 |     int price;
4 |     double percent;
5 |
6 |     Seller(p):Worker(name) {
7 |         product = new char [strlen(product) + 1];
8 |         strcpy(product, p);
9 |     }
10 |    ~Seller() {
11 |        delete [] product;
12 |    }
13 |    int get_salary() {
14 |        return price * percent * num;
15 |    }
16 |
17 | }

```

Поговорим о деструкторе.

```

1 | ~WorkerDataBase() {
2 |     for () {
3 |         delete workers[i];
4 |     }

```

```
5 || }
```

Чтобы при удалении worker-ов вызывался деструктор нужного подкласса (для продавца, для программиста и т.д.), нужно так же сделать его виртуальным.

```
1 || Worker {  
2 ||     virtual ~Worker() {...}  
3 || }
```

## 9.5. Практика

Таблица виртуальных методов.

Наследование интерфейса, наследование реализации.

Ключевое слово `override`. Перегрузка, перекрывание.

На следующей недел будет домашка. Паттерн MV(C). Игра в клеточки.

# 10. Лекция 09.12.16

## 10.1. Зависимости

Project 1.

$A \rightarrow B$ . Класс A ссылается на класс B (A depends on B).

```

1 | class A
2 |     1) B obj
3 |     2) B *pObj
4 |     3) B & rObj

```

Project 2.

Хотим использовать только A. Если скопируем только A, проект 2 не скомпилируется, так как будут нарушены зависимости. Нужно также перенести класс B (и все остальные зависимости, если есть).

Рассмотрим взаимную ссылку (цикл в графе зависимостей,  $A \rightarrow B$ ,  $B \rightarrow A$ ). (Пример: генеалогическое дерево, родитель ссылается на ребёнка, ребёнок на родителя.)

Часто можно избежать циклических зависимостей, лучше это делать.

## 10.2. Model-View

Книга Design Patterns («Шаблоны проектирования») — каталог типичных примеров, идей, например, как написать undo, описывает архитектурные решения. Must-read для программистов.

Паттерны программирования нельзя использовать бездумно, иначе

Рассмотрим задачу: крестики-нолики  $10 \times 10$ , для выигрыша нужно поставить 5 в ряд. Текстовый интерфейс.

(См. также модель MVC.)

### 1. Model (class Board)

- хранение в памяти
- правила

### 2. View (class TextView)

- вывод игрового поля
- ввод от пользователя

Когда нам может понадобиться переиспользовать Board? При переносе приложения на другие платформы: GUI, Mobile, Web.

TextView  $\rightarrow$  Board.

По-хорошему, размер доски должен быть параметром класса, но для упрощения мы будем всегда считать, что размер  $10 \times 10$ .



```

1 | class Board {
2 |     private:
3 |
4 |     public:
5 |         Board();
6 |         void move(int x, int y, char c);
7 |         bool canMove(int x, int y, char c) const;
8 |         int isWin(); // 4 состояния: выиграл 1, выиграл 2, ничья, игра в процессе
9 |         int getCell(int x, int y); // Привет, епит, ты где
10 |        int getWidth() const;
11 |        int getHeight() const;
12 |    }

```

isWin(): можно создать структуру enum (разберём на практике), чтобы зафиксировать 4 значения и предотвратить случайное возвращение какого-нибудь пятого значения.

```

1 | main () {
2 |     Board b(..);
3 |     TextView tv(b);
4 |     tv.startGameCycle();
5 |     return 0;
6 | }

```

```

1 | class TextView {
2 |     private:
3 |         Board & b;
4 |     public:
5 |         TextView(Board & board) : b(board) { /* конструкция list initialization,
6 |             отчасти аналогично b = board, см. в следующих сериях */ }
7 |         void StartGameCycle() {
8 |             while(b.isWin() == 4) {
9 |                 // Если бы создали структуру епит, можно было бы константу назвать GAME_IN_PROCESS
10 |                1. Считагы ход игрока : x y
11 |                2. b.move(x, y, 'o');
12 |                3. showBoard();
13 |                4. проверка isWin();
14 |                5. 1,2,3,4 для 'x'
15 |            }
16 |        }
17 |        void showBoard() const {
18 |            for (i= ..)
19 |                for (j= ..)
20 |                    printf(getCell(i, j));
21 |        }

```

### 10.3. Доп. ДЗ на CursesView

Будет доп. задание CursesView:

Рассмотрим пример: stdio 1 3 отрисовать доску 2 3 отрисовать доску

Хотим использовать стрелочки, хотим обновлять доску на одном и том же месте. stdio не позволяет это сделать, но можно с ncurses.

Перерыв.

Model ← View1 [] // Храним текущую игру

Model ← View2 [] // Храним историю ходов

Хотим отмотать игру на несколько ходов. Тогда необходимо связать View1 и View2. Нужно использовать шаблон listener, разберём в следующем семестре.

## 10.4. Статические переменные. Ключевое слово `static`

Ключевое слово `static`. Встречали в си-функции `strtok` — хотим каждый раз получать новый токен: сначала `vasya`, потом `petya` и т.д.

`vasya 123 petya`

`token, delimiter, token`

```

1 | char* strtok(char *s, const char * delim);
2 | static char * saved_s;
3 |
4 | char *token;
5 | token = strtok(s, "");
6 | while(token != NULL) {
7 |     token = strtok(NULL, "");
8 |
9 | }
```

`static` — глобальная переменная, видная только внутри функции, где определена.

Обычные, не-статические переменные — это плохо, так как может произойти коллизия пространства имён, а также сложны в обращении: встретили в коде `foo = 3`, а в какой инклюд лезть, чтобы найти объявление, не понятно.

```

1 | // h
2 | Rectangle {
3 |     static int num;
4 |     int w;
5 |     Rectangle() {
6 |         num++;
7 |     }
8 | }
9 |
10 | // cpp
11 | int Rectangle::num = 0;
12 |
13 | main() {
14 |     Rectangle r1, r2;
15 |     r1.num // == 2
16 |     Rectangle::num = 3;
17 |     r1.w;
18 |     r2.w; // r1.w и r2.w - разные переменные
19 | }
```

Статическая функция. Не привязана к конкретному классу.

```

1 | static bool cmp(Rectangle & r1, Rectangle & r2) {
2 |     w = 3; // compilation error
3 |     r1.w // ok
4 |     r2.w // ok
5 | }
```

## 10.5. Unit-тестирование, автотесты

Цели тестирования:

1. повторимость ошибок
2. документация кода

```

1 | src
2 |   Board.{cpp, h}
3 |   View.{cpp, h}
4 |   main.cpp -> main
5 | test
6 |   BoardTest.{cpp, h}
7 |   test.cpp -> main
8 | Makefile
9 |
10 | make test (Board, BoardTest, test)

```

```

1 | #include "Board.h"
2 | class BoardTest {
3 |     testCanMove() {
4 |         Board b;
5 |         b.move(1, 1, 'x');
6 |         bool res = b.canMove(1, 1, '0');
7 |         if (res) {
8 |             printf("Test Failed");
9 |         }
10 |         // позже заменим иф на check:
11 |         check(res == false, __FUNC__, __FILE__, __LINE__)
12 |         // __FUNC__, __FILE__, __LINE__ - препроцессор подставит значения
13 |     }
14 |
15 |     testWin() {
16 |         Board b;
17 |         b.move(..);
18 |         ..
19 |         int res = b.isWin();
20 |         ..
21 |     }
22 | }

```

```

1 | test.cpp
2 |
3 | main() {
4 |     BoardTest bt;
5 |     bt.testCanMove();
6 |     bt.testIsWin1();
7 |     bt.testIsWin2();
8 | }

```

Хотим статистику по тестам: из 10 тестов 8 ok, 2 failed. Воспользуемся статическими переменными.

```

1 | class Test
2 |     static int total;
3 |     static int failed;
4 |     static void showStat(); // вывести статистику
5 |     static void check(bool expr, const char *func, const char* file, int ) {
6 |         if (!expr) {
7 |             failed++;
8 |             printf("%s in %s:%d", func, file, line);
9 |         }
10 |         total++;
11 |     }

```

В ДЗ будем реализовывать самостоятельно, но есть библиотеки, которыми нужно пользоваться всегда, кроме этого ДЗ: google test, crrunit.

Unit Test — тестирование отдельных функций / классов / компонентов.

Test Driven Development — методология разработки, при которой сначала пишутся тесты, а потом код.

# 11. Лекция 16.12.16

Не будем начинать новые темы, будем закрывать дырки по старым.

## 11.1. Static

```

1 |   strtok
2 |   strtok char * stred_str = NULL;
3 |
4 |   strtok(s, ...);
5 |
6 |   while () {
7 |       strtok(NULL, );
8 |   }

1 |   // Test.h
2 |
3 |   Test
4 |       static int field;
5 |
6 |   // Test.cpp
7 |   int Test::field = 0;
8 |
9 |   Test a;
10 |  Test b;
11 |  Test::field = 3;

1 |   Point {
2 |       int x, y;
3 |       int distance(const point &p); // интуитивно неправильно
4 |       static int distance(const Point &p1, const Point &p2); // логично
5 |   };
6 |
7 |   main()
8 |       Point p1(1, 3), p2(3, 4);
9 |       int d = p1.distance(p2); // неестественный синтаксис
10 |      int d = Point::distance(p1, p2);

```

## 11.2. Пример: шаблон проектирования Singleton (Mayer)

Программе нужны следующие классы:

- Config — настройки: размер шрифта, адрес сервера, и т. д. Пусть конфиг хранится в файле в виде строк типа «board\_size::34», «color:83».

```

1 |   hashable h;
2 |   congfig(const char * key);
3 |   int lookup

```

- logger — журнал событий: «12:45 — подключились к БД», «13:10 — взорвали жёсткий диск».

```

1 |   log()

```

```

1 |   Board
2 |       Config g;
3 |   View
4 |       Config g;

```

Однако мы хотим иметь по одному объекту классов `config` и `logger`. Тогда хотим запретить создавать объект, то есть хотим, чтобы следующий код не компилировался:

```
1 | Board
2 |     Config cfg(); // cfg
3 | View
4 |     Config cfg1();
```

Для этого нужно сделать конструктор приватным.

```
1 | class Config {
2 | private:
3 |     hashable int;
4 |     Config();
5 | public:
6 |     int lookup(const char * key) { }
7 |
8 |     static Config * get_instance() {
9 |         static Config cfg();
10 |        return &cfg;
11 |    }
12 |
13 |    // Или такая реализация. Не забудьте, нужен метод destroy (вызываемый вручную деструктор)
14 |    static Config* get_instance() {
15 |        if (!ptr) ptr = new Config();
16 |        return ptr;
17 |    }
18 |
19 |
20 |    Config() {
21 |        FILE* f = fopen(..)
22 |        while(!fend(f)) {
23 |            ht.insert(..);
24 |        }
25 |    }
26 | }
```

Рассмотрим, как это будет использоваться.

```
1 | Board() {
2 |     int sz = Config::get_instance()->lookup("board-size");
3 |
4 | }
5 | View(...) {
6 |     int clr = Config::get_instance()->lookup("color");
7 | }
```

Также эта техника называется ленивой инициализацией.

Нужно понимать, что не нужно реализовывать Singleton в каждом классе. У каждого шаблона проектирования своё назначение, и при реализации какой-либо конкретной программы нужно учитывать её особенности и принимать те или иные архитектурные решения.

### 11.3. Разрешение имён

Имена глобальных переменных нужно заменить на их адреса. Для этого нужно два прохода: пройти по файлу и выписать в таблицу пары имя-адрес, а потом пройти и расставить адреса.

Внутренняя линковка — разрешение имён через несколько таблиц, один файл, одна таблица. (Нельзя вызывать функции из других файлов.)

Внешняя линковка — разрешение имён через одну таблицу для всего проекта.

Рассмотрим конфликт имён. Без ключевого слова `static` случился бы конфликт. Для переменных `b` будет использована внутренняя линковка.

```

1 // a.cpp - написал Вася
2 static int b = 31;
3 f2() { b; }
4
5 static int min() {} // видна только внутри файла
6
7 // b.cpp - написал Петя
8 static int b = 41;
9 f3() { b; }

```

## 11.4. Ключевое слово `inline`

Рассмотрим две функции: максимум из двух элементов и

```

1 // maxes.cpp
2
3 int max(int a, int b) {
4     if (a > b) return a;
5     return b;
6 }
7
8 int find_max(int *array, size_t size) {
9     int m = 0;
10    for () {
11        m = max(m, a[i]);
12    }
13    return m;
14 }

```

Вызов функции сопряжён с дополнительными расходами: сложить переменные на стек, перейти к адресу функции, перейти обратно.

В данном случае компилятор сгенерирует оптимизированный код, подставив определение функции под вызов:

```

1 // maxes.cpp
2
3 int find_max(int *array, size_t size) {
4     for () {
5         if (m > a[i]) {
6             m = m;
7         }
8         else {
9             m = a[i];
10        }
11    }
12    return m;
13 }

```

Теперь предположим, что функция `max` находится в файле `max.cpp`, а `find_max` — в файле `maxes_array.cpp`. В таком случае оптимизация не произойдёт, так как компилятор оптимизирует внутри одного файла.

Чтобы решить эту проблему, объявим `max` в заголовочном файле `max.h` и включим его в `maxes_array.cpp`. Тогда оптимизация произойдёт. Однако могут возникнуть проблемы при лин-

ковке, так как будут два разных файла с определением одной и той же функции.

Ключевое слово `inline`. Если встретилось несколько определений функции, компилятор включает одну любую, например, первую и отбросит остальные.

## 11.5. Классы `inline`

К реализациям функций в h-файлах по умолчанию дописывается `inline`.

```

1 | Board.h
2 |
3 | class Board() {
4 |     int move(int x, int y) { // дописалось inline
5 |         ..
6 |     }
7 | }
8 |
9 | // Board.cpp
10 |
11 | inline Board::can_move(int x, int y) {
12 |     ..
13 | }
```

Если объявить в двух h-файлах одну и ту же функцию, к ним по умолчанию допишется `inline`, компилятор включит случайную из них, и удачной отладки.

## 11.6. Immutable

Минутка линейной алгебры. Определитель (детерминант) матрицы — некоторая характеристика матрицы, показывающая можно ли найти обратную к ней матрицу. Вычисление определителя — ресурсоёмкая операция.

Предположим, мы работаем с большими матрицами, 1000x1000. Если матрица не изменяется, вычислять определитель каждый раз нерационально. Воспользуемся методом кэширования. Заметим, что метод `determ` перестал быть `static`, так как ему нужно обновить флаг `is_fresh`, однако мы не хотим лишиться защиты от случайного изменения матрицы. Для этого пометим флаги с помощью ключевого слова `mutable`, разрешающего менять переменные внутри статического метода.

```

1 | class Matrix {
2 |     void set(int x, int y, int num);
3 |     int get(int x, int y) const;
4 |     int determ() const;
5 |     mutable int d;
6 |     mutable int is_fresh;
7 | }
8 |
9 | int determ() {
10 |     if (is_fresh) {
11 |         return d;
12 |     }
13 |     is_fresh = true;
14 |     calculate d; // O(n^b)
15 |     return d;

```



```
16 }
17 int set(..) {
18     is_fresh = false;
19 }
20
21 f(matrix & m) {
22     ..
23     int d = m.determ();
24     ..
25 }
26
27 g(matrix & m) {
28     int d = m.determ();
29 }
```

## 12. Модификаторы доступа

Нотация UML.

Класс А

public + x() — можно вызвать откуда угодно.

private - y() — можно вызвать только из класса.

protected # z() — можно вызвать из класса или из наследника.

```
1 | class A {}
2 | class B : public A {}
3 | class C : protected A {}
4 | class D : private A {}
5 |
6 | A a;
7 | a.x()
8 | a.y()
9 | a.z()
10 |
11 | B b;
12 | b.x() // public
13 | b.y() // protected
14 | b.z() // нет доступа
```

Все знают, что В наследник А. Только класс С и его наследники знают, что он наследник А. Только класс D знает, что он наследник А.

# 13. Лекция 23.12.16

## 13.1. Стандартная библиотека C++

## 13.2. Конструкция namespace

```

1 | namespace model {
2 |     class Vector { ... }
3 | }
4 |
5 | namespace view {
6 |     class Vector { ... }
7 | }
8 |
9 | model::Vector v1;
10| model::Vector v2;

```

Всё содержимое стандартной библиотеки находится в пространстве имён `std`, то есть нужно использовать префикс `std`.

```

1 | a.h
2 |
3 | class A {
4 |     std::vector<int> v;
5 | }
6 |

```

Чтобы сократить такую запись, воспользуемся следующей конструкцией.

```

1 | using namespace std;

```

Однако если внутри двух пространств имён

```

1 | using namespace model;
2 | using namespace vector;
3 |
4 | vector v1; // проблема, откуда брать вектор: из model или view?

```

Поэтому рекомендуется всегда использовать префиксы. В заголовочных файлах настойчиво рекомендуется всегда использовать префиксы, так как при подключении заголовочного файла всё содержимое `cpp`-файла станет использовать то же пространство имён, что и заголовочный файл, что чревато негативными последствиями.

## 13.3. iostream

`stream` — поток. Поток — это класс, оборачивающий понятие файла.

В Си `string.h`, в C++ `string`. Чтобы не путаться, каждому заголовочному файлу стандартной библиотеки Си (например, `string.h`) в C++ создали пару с аналогичным названием: (`cstring.h`).

```

1 | #include <iostream>

```

Иерархия классов. (Нарисуйте табличку.)

```

ios ← istream
ios ← ostream
istream ← ifstream

```

```
ostream ← ifstream
istream, ostream ← iostream
iostream ← stringstream, fstream
```

Рассмотрим стандартные потоки ввода вывода.

```
1 |   stdin  std::cin
2 |   stdout std::cout
3 |   stderr std::cerr
```

Особенность cerr — это небуферизируемый поток для вывода ошибок. Хотим напечатать какой-то дебаг-вывод даже если программа упадёт.

```
1 |   ostream& operator<<(ostream& o, int v) {
2 |       ...
3 |       return o;
4 |   }
5 |
6 |   int n = 4;
7 |   std::cout << n;
8 |   opeartor<<(std::coun, n);
9 |
10 | (std::cout << n) << ' ';
11 | // эквивалентно
12 | operator<<(operator<<(std::cout, n), ' ');
```

Рассмотрим определение оператора оператор« для написанного нами класса гауссовых чисел. Доступ к приватным полям класса GN можно получить через геттеры/сеттеры, а можно через дружественную функцию.

```
1 |   class GN
2 |   private:
3 |       int im;
4 |       int re;
5 |       getIm()
6 |       getRe()
7 |
8 |   ostream& operator<<(ostream& o, const GN& gn) {
9 |       o << gn.re << ' ' << gn.im;
10 |       return o;
11 |   }
```

Ещё пример.

```
1 |   GN gn(1, 1);
2 |   std::cout << gn << std::endl;
3 |   std::ofstream of("out.txt");
4 |   of<<gn;
5 |   std::stringstream ss;
6 |   ss << gn;
```

Немного о вводе.

cin определяет, какой тип вводить в переменную по типу аргумента (тип переменной n в примере ниже).

```
1 |   std::cin >> n;
```

Ещё пример. Установим формат вывода в шестнадцатеричном формате. (Аналог printf("

```
1 |   std::cout.setf(std::ios.hes);
2 |   std::cout << n; // вывели n в 16-ом формате, флаг сбросился
3 |   std::cout << n; // сейчас n выведется в 10-ом формате.
```

Есть и другие функции кроме установки флага `setf: width` — ширина вывода, а также другие. Все эти функции аналогичны формату `printf` из Си.

Можно написать то же самое покороче.

```

1 | std::cout << std::ios::hex << N;
2 | // эквивалентно
3 | operator<<(operator<<(std::cout, &hex), N);
4 |
5 | ostream& hex(ostream& os) {
6 |     os.setf(...);
7 |     return os;
8 | }
9 |
10 | ostream& opearor<<(ostream &o, ostream&(*pf)(ostream&)) {
11 |     pf(o);
12 |     return o;
13 | }
```

В Си были `ferror()`, `feof()`. Аналоги в C++: `eof()`, `fail()` — логическая ошибка (не можем считать элемент типа `int`), `bad()` — физическая ошибка (файл не найден).

```

1 | stdio
2 | int r = fscanf(f, "%d", &n);
3 | if (r != 1) { ... } // произошла ли ошибка?
4 | if (feof()) { ... } // произошла ли ошибка?
```

Ещё пример.

```

1 | ofstream of;
2 | of.open("out.txt");
3 | of.put
4 | of.write(char *f, int size)
5 | seekp
6 | tellp
```

## 13.4. Разбор теста на Gaussian Number

Заметим, что в классе нет ресурсов, поэтому не нужно писать деструктор. Также не нужен оператор присваивания.

## 13.5. Про экзамен

Будет 1-2 вопроса в билете, 10-12 билетов. Будет консультация за день до экзамена. Чтобы подтвердить свою оценку, нужно рассказать 2 теоретических вопроса. Чтобы повысить оценку на балл, нужно рассказать два вопроса и написать какой-то код.