

# C++

Наташа Мурашкина

25 ноября 2016 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

# 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`, `cppreference`.

```
1 | char * strtok(char *str, const char *delim); // Возвращает указатель на очередной
   | токен.
2 | "Hello word 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 скомпилируется, т.к. array - приватное поле
15 arr.size; // не скомпилируется
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`.