C++

Наташа Мурашкина

16 декабря 2016 г.

Содержание

1. Лекция 14.10.16

1.1. Стандартная библотека

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 напуган 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  memcpy
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 (delimeter)
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).

2.2. Библиотека Expat

Разобрали функцию parse_xml (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.

```
g++
```

В C++ приведение явнее, чем в Си.

```
int *pi;
char *pc;
pc = pi; // Нельзя (?)
pc = (char*)pi;
```

Что ещё нового?

```
// overloading (перегрузка)
int min(int a, int b)
int min(int a, int b, inc c)
double min(double a, double b)
// В Си не могут существоать три функции с одинаковым названием.
// mangling - сигнатура преобразуется

// В Си функции идентифицируются по именам: min, min, min
// В C++ - по имени+типам аргументов: min_int_int, min_int_int_int, min_double_double
```

```
// В Си
int *a = malloc(sizeof(int) * 100);
free(a);

// В C++
int *a = new int[100];
...
delete []a;
```

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

С дин-типизированным языком проще разобраться, но

В C++ ключевое слово для дин. выделение 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

```
// class
int x, y, z;
func1() {}
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. Пользоватулю 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.

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

Хотим предотвратить совершение этих ошибок программистом-пользователем.

```
// my_array.h
class my_array {
private:
  int *array;
  size_t size;
public:
  my_array(size_t s); // constructor, решим ошибку 1
  ~my_array(); // destructor, решим ошибку 3
  int get(size_t index);
  void set(size_t index, int value);
};

my_array arr(10);
arr.array; // не скомпилится, т.к. array - приватное поле
arr.size; // не скомпилится
arr.set(i, 5); // ок
```

```
// my_array.cpp
#include "my_array.h"
my_array::my_array(size_t s) {
  size_s;
  array = new int [size];

}

my_array::~my_array() {
  delete []
}

int my_array::get(size_t index) {
  if (i >= 0 && i < size) {
    return array[i];
  }
  else {
    return -1; // в массиве лежал -1 или это ошибка? Поговрим в след. семестре про
        exception, а сейчас закроем глаза на такую некрасивость.
  }
}

// + set

// + get_size
```

За такую красоту мы заплатили вызовами функций.

```
void f() {
  my_array arr()

}
```

4. Ссылки

```
swap(int *a, int *b) {
  mt += *a;
  *a = *b;
  *b =
}

int c = 3;
int d = 4;
swap(&c, &d);

// То же самое с ссылками (это синт. сахар)
void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t;
}

int c = 3;
int d = 4;
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 притянуто за уши.)

В Си тоже возможно реализовать инкапсуляцию, opaque pointer.

5. Лекция 28.10.16

Нет общей темы, будем закрывать дырки.

```
int
short
long

stdint.h // Набор typedef'ов
int8_t // INT8_MIN - мин. значение переменной такого типа
uint8_t // UINT8_MIN
int16_t
uint16_t
// Если на платформе физически отсутствует возможность хранить такой размер (из-за длины
    регистра), то прога не скомпилится.

ctype.h
int isalpha (char c) // Является ли символ с буквой
int isdigit (char c)
tolower()

assert.h // Будет отдельная лекция про обработку ошибок
// 2 типа ошибок
// 1. По вине программиста, runtime error
int a[10];
a[10] = 3;

char *p = NULL;
strlen(p);

// 2. По вине окружения
// не можем выделить память с помощью malloc, т.к. закончилась память
// не открыть файл, т.к. его нет

// Поговорим об ошибках типа 1.

size_t strlen(char *s) {
  assert(s != NULL); // Если выражение false, вызовется abort()
  ...
}

#ifdef NDEBUG
  ; // все асерты заменяются на точку с запятой
#else
    ; // какая-то проверка
    abort()
#endif

// При компиляции продакшн-кода (уже для пользователя) добавить ключ -D со значением NDEBUG
// gcc -DNDEBUG

FILE *f = fopen(''m.txt'', ''r'');
assert(f != NULL);
// Эта проверка исчезнет, когда скомпилим на продакшн

// 4)
math.h
cos
sin
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 t1 такт, на другом - t2
79  // Реально прошло max(t1, t2), а clock() посчитает сумму: t1 + t2
80
81  // Си
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 ака выстреливаем себе в ногу ака 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 препендуть 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 именует функции.

```
1  // 1) Функции
2  f(int);
3  f(int, double);
4
5  // 2) Операторы
6  // * +; &&; &; ==, <>
7  // (type) = конструктор Copy
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  // C 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 ↑, копилятор сделал бы сам так:
63
64  BigInt(const BigInt & obj) {
65    size = obj.size;
66    array = obj.array;
67  }
68
69  // Если нет конструткора 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 временные объект tmp типа BigInt
13 // Потом вызовется конструтор копирования tmp в a (но компилятор соптимизирует и на самом деле
       вызовется только a(3))
14
```

```
15  // Теперь какая-то жесть непонятная про конктруторы и приведение с ацкими нелинейными
        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, заполненая
        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 скопирует 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 вовращает BigInt, заполненый 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
          указателям), т.к. вызовется деструктор по выходе из функции getOnes.
16
17    // [RA][RV][----]
18    // RV - return value
```

То, что делает компилятор, тоже ок. Правило трёх. Если вашему классу нужен деструтор, 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)
  BigInt a;
  f(a); f(BigInt br);

  // Виды ресурсов:
  int **;
  FILE *f;

  BigInt b(a);

  // 2
  BigInt a;
  BigInt b;
  a = b; // эквивалентно a.operator=(b);

  // 3
  BigInt a;
  BigInte b;
  a += b; // А в Java, например, перегрузка операторов отсутствует.
  BigInt c = a + b;
```

Начнём с плюсиков.

```
BigInt& operator+=(const BigInt& o) {
    for () {
      this->array[i] += o.array[i];
      // сделать переносы, бла-бла-бла
    }
    return *this;
}

// Вариант 1
BigInt operator+(const BigInt& o) {
    BigInt t(o);
    t += (*this);
    return t;
}

// Вариант 2
BigInt operator+(BigInt o) {
    return o += (*this);
}

// Примечание: оба кода выше не работают с коммутативными операциями

BigInt c = b + 3; // 3 переходит в BigInt(3);
BigInt c |= 3 + b; // b типа BigInt не может перейти в int

// 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)

```
class ExternalArray {
public:
    ExternalArray() {
        _external = new int*[3];
        _external[0] = new int[42];
        ...
    }

    class Proxy {
    public:
        Proxy(int* _array) {
          _array = _array
        }

        int operator[](int index) {
            return _array[index];
        }
    private:
        int* _array;
    };

    Proxy operator[](int index) {
        return &Proxy(_external[index]);
    }

private:
    int ** _external;
};


ExternalArray ea;
ea[3][5];
```

7.3. Smart pointer

```
  Person {
    char name[256];
    char * phone[20][25];
    int age;
    char encl[256];
  }

  Person *p = new Person();
  Person *p1 = new Person(*something);

  delete p;

  // 2
  Person *p = new Person [105]; // для такого нужно иметь конструктор по умолчанию
  delete [] p;

  // Объекты Person раскиданы по памяти
  Person ** p = new Person* [100];
  for (i = 0; i < 100; i++) {
    p[i] = new Person(i);
```

```
21  }
22
23  // Про память. Prefetch. Конвеер. 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& opeartor*() { 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 перегрузкой:
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 существуеющий, 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 — перекрузка (функций) — было раньше override — прекрытие (методов)

```
  int value;
  list *next;
  list *prev;

  class double_list:public list {
    private:
      list *prev;
    public:
      double_list(int v);
      ~double_list();
      void add(int v);
    private:
      double_list(const double_list &&);
      double_list& operator= () {...}
  }

  double_list::double_list(int v):list(v) { // вызываем конструтор суперкласса
    prev = NULL;
  }

  void double_list::add(int v) {
    double_list *cur = this;
    while (cur->next != NULL) {
      cur = cur->next;
    }
    // Типы: int* и double_list*
    cur->next = new double_list(v);
    // Тип cur->next - list*
    cur->next->prev = cur;
  }

  main() {
    double_list dl;
  }
```

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 деструтор для класса list. От лектора: «Цикл не цикл, рекурсия не рекурсия».

8.3. Практика

Идиома RAII — Resource acquisition is initialization

Пример: shared pointer, mutex

```
  class foo {
    ~Foo() {}
    std::shared_ptr<baz> baz;
    int * bar; // Освободить руками
```

```
5 ||  }
```

Что плохого можно сделать с shared_ptr.

A $\leftarrow\rightarrow$ 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

```
list
  protected:
    int value
    list * next
    add_value

double_list
  add_value(int v) {
    xxx;
  }
```

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

```
list* l = new list(...);
l->value = 35; // ok
l->head = NULL; // comp. error, несовпадение типов
l->length();
l->remove_value(5);
```

Приведение типов.

```
int * a = new ...
char * b = new ...
a = (int*)b;
// или
b = (char*)a;
```

```
// next - типа list*
cur->next = new double_list(..)
```

```
// Поля list: value, next
// double_list - наследник list, добавлено поле: prev

list* l = new list(..);
double_list * dl = new  double_list(..);

l = dl; // ok
l->next
l->value

dl = l; // error, опасное неявное приведение типов
dl->prev
dl->next
dl->value
```

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

9.2. Динамическое связывание

```
// 2015

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      }
```

```
1     list *l = new list(..)
2     double_list *dl = new double_list(..)
3     l->add_value(..)
4     dl->add_value(..)
5
6     list *l1 = dl;
7     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 начала 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.

```
list *1;
stand(time(NULL));
int n = rand();
if (n > 0) {
  l = new list(5);
}
else {
  l = new double_list(6);
}
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

```
Worker
  Worker () {
    char * name = new ..;
  }
  ~Worker() {
    delete [] name;
  }
  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.

```
WorkerDataBase db;
Developer *d = new Developer(''Masha'', 1000, 5, False, 300);
Seller *p = new Seller(''Petya'', ''Windows'', 0.03, 10000);
```

```
WorkerDatabase {
  Worker* workers[100];

  void add(worker *w) {
    ..
    workers[size] = w;
    ..
  }

  int getTotalSalary() {
    int salary = 0;
    for (i = 0; i < size; ++i) {
      salary += worker[i]->get_salary(); // у каждого типа сотрудника будет вызываться свой метод
    }
    return salary;
  }

}
```

Рассмотрим класс Developer.

```
class Developer:public Worker {
  int salary;
  int level;
  bool isRelease;
  int bonus;

  Developer(..):Worker(name) {..}

  get_salary() {
    int ret = salary * level;
    if (isRelease) ret += bonus;
    return ret;
  }
}
```

Рассмотрим класс Seller. Поговорим о деструторе.

```
class Seller:public Worker {
  char * product;
  int price;
  double percent;

  Seller(p):Worker(name) {
    product = new char [strlen(product) + 1];
    strcpy(product, p);
  }
  ~Seller() {
    delete [] product;
  }
  int get_salary() {
    return price * percent * num;
  }

}
```

Поговорим о деструкторе.

```
~WorkerDataBase() {
  for () {
    delete workers[i];
  }
```

```
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 интерефейса, 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 → 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 → B, B → 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 10x10, 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 → Board.

По-хорошему, размер доски должен быть параметром класса, но для упрощения мы будем всегда считать, что размер 10x10.

```
class Board {
  private:

  public:
    Board();
    void move(int x, int y, char c);
    bool canMove(int x, int y, char c) const;
    int isWin(); // 4 состояния: выиграл 1, выиграл 2, ничья, игра в процессе
    int getCell(int x, int y); // Привет, enum, ты где
    int getWidth() const;
    int getHeight() const;
}
```

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

```
main () {
  Board b(..);
  TextView tv(b);
  tv.startGameCycle();
  return 0;
}
```

```
class TextView {
  private:
    Board & b;
  public:
    TextView(Board & board) : b(board) { /* конструкция list initialization,
        отчастианалогично  b = board, см. вследующихсериях  */ }
    void StartGameCycle() {
      while(b.isWin() == 4) {
        // Если бы создали структуру enum, можно было бы константу назвать GAME_IN_PROCESS
        1. Считатьходигрока  : x y
        2. b.move(x, y, 'o');
        3. showBoard();
        4. проверка isWin();
        5. 1,2,3,4 для 'x'
      }
    }
    void showBoard() const {
      for (i= ..)
        for (j= ..)
          printf(getCell(i, j));
    }
}
```

10.3. Доп. ДЗ на CursesView

Будет доп. задание CursesView:

Рассмотрим пример: stdio 1 3 отрисовать доску 2 3 отисовать 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, delimeter, token

```
char* strtok(char *s, const char * delim);
static char * saved_s;

char *token;
token = strtok(s, "");
while(token != NULL) {
  token = strtok(NULL, "");

}
```

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.

```
// h
Rectangle {
  static int num;
  int w;
  Rectangle() {
    num++;
  }
}

// cpp
int Rectangle::num = 0;

main() {
  Rectangle r1, r2;
  r1.num // == 2
  Rectangle::num = 3;
  r1.w;
  r2.w; // r1.w и r2.w - разные переменные
}
```

Статическая функция. Не привязана к конкретному классу.

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

10.5. Unit-тестирование, автотесты

Цели тестировния:

1. повторимость ошибок
2. документация кода

```
src
  Board.{cpp, h}
  View.{cpp, h}
  main.cpp -> main
test
  BoardTest.{cpp, h}
  test.cpp -> main
Makefile

make test (Board, BoardTest, test)
```

```
#include "Board.h"
class BoardTest {
  testCanMove() {
    Board b;
    b.move(1, 1, 'x');
    bool res = b.canMove(1, 1, '0');
    if (res) {
      printf("Test Failed");
    }
    // позже заменим иф на check:
    check(res == false, __FUNC__, __FILE__, __LINE__)
    // __FUNC__, __FILE__, __LINE__ - препроцессор подставит значения
  }

  testWin() {
    Board b;
    b.move(..);
    ..
    int res = b.isWin();
    ..
  }
}
```

```
test.cpp

main() {
  BoardTest bt;
  bt.testCanMove();
  bt.testIsWin1();
  bt.testIsWin2();
}
```

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

```
class Test
  static int total;
  static int failed;
  static void showStat(); // вывести статистику
  static void check(bool expr, cnst char *func, const char* file, int ) {
    if (!expr) {
      failed++;
      printf("%s in %s:%d", func, file, line);
    }
    total++;
  }
```

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

Unit Test — тестирование отдельный 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

```
strtok
  strtok char * stred_str = NULL;

strtok(s, ...);

  while () {
    strtok(NULL, );
  }
```

```
// Test.h

Test
  static int field;

// Test.cpp
int Test::field = 0;

Test a;
Test b;
Test::field = 3;
```

```
Point {
  int x, y;
  int distance(const point &p); // интуитивно неправильно
  static int distance(const Point &p1, const Point &p2); // логично
};

main()
  Point p1(1, 3), p2(3, 4);
  int d = p1.distance(p2); // неестественный синтаксис
  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».

```
  hashable h;
  congfig(const char * key);
  int lookup
```

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

```
  log()
```

```
Board
  Config g;
View
  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:

```
Board
  Config cfg(); // cfg
View
  Config cfg1();
```

Для этого нужно сделать конструктор приватным.

```
class Config {
private:
  hashable int;
  Config();
public:
  int lookup(const char * key) { }

  static Config * get_instance() {
    static Config cfg();
    return &cfg;
  }

  // Или такая реализация. Не забудьте, нужен метод destroy (вызываемый вручную деструктор)
  static Config* get_instance() {
    if (!ptr) ptr = new Config();
    return ptr;
  }


  Config() {
    FILE* f = fopen(..)
    while(!fend(f)) {
      ht.insert(..);
    }
  }
}
```

Рассмотрим, как это будет использоваться.

```
Board() {
  int sz = Config::get_instance()->lookup("board-size");

}
View(...) {
  int clr = Config::get_instance()->lookup("color");
}
```

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 имёт 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 maxes.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 включит 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(ine 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    }
```