Языки программирования, 2017г

[**Лекция 1:**](#_mpu44ou02jb1) **5**

[Введение](#_h9of971866h2) 6

[Парадигмы программирования](#_ql462xvr1l3) 6

[1. Императивная парадигма](#_dcyq64hwcjv1) 6

[2. Функциональная парадигма](#_8np1ffcn4xn7) 6

[3. Логическое программирование](#_sr4wllpyulgh) 6

[4. ООП](#_8yvu0nlnpp2d) 7

[**Лекция 2:**](#_welw2lydw4a7) **7**

[Деятельность программирования (делится на 3 части)](#_iwrx1xkrmdc) 7

[Всякие разные ЯПы](#_bu1vv6rjj2ut) 7

[Основные выводы](#_ws1mn5mvxt5i) 7

[Задача для индустриального ЯП](#_805t1ym32m1x) 7

[**Лекция 3:**](#_kgofa83npz18) **8**

[**Лекция 4:**](#_rquha0k99hry) **9**

[Язык С++](#_j1b8nm2pwtyk) 9

[Функциональное программирование](#_czsmx6hokmp1) 10

[Идея байт-кода](#_gn2kohhvxnay) 11

[**Лекция 5:**](#_hcjutjrluqc6) **12**

[REFAL](#_qabilua1xv2) 12

[Rust](#_629kthka307s) 13

[Web-программирование](#_uqye97gp1x5h) 13

[Логическое программирование](#_gfhwalz1puam) 13

[Prolog](#_cc0g0u6mw67d) 13

[Основные понятия ЯП](#_44wvkbx2nnla) 14

[Геттеры/Сеттеры](#_8ja0aflxwqy1) 14

[Механизм создания новых ТД](#_24b0m9s957bo) 14

[**Лекция 6**](#_sklwg2mcmw6i) **15**

[Связывание (binding)](#_8tbhyocvsrne) 15

[Динамическое](#_pid47w6iz1u2) 15

[Статическое](#_awphjuclzedx) 15

[**Лекции 7, 8:**](#_934jdigrf38) **16**

[Простые типы данных, операции над ними](#_srtrxypgghtk) 16

[Объектно-процедурная парадигма](#_y55exl5307or) 17

[**Лекция 9:**](#_74kordo4i0bc) **18**

[Oпять классификация типов данных](#_endg7ibi5xt6) 18

[Перечислимые типы](#_7z79biie0stu) 18

[Диапазоны](#_zgu14hadf597) 19

[Ссылки и указатели](#_1my0d17u9qoj) 20

[Ссылки](#_d1p3kys82btw) 20

[Символьные типы данных](#_bm3loblif7sq) 20

[**Лекция 10**](#_10ruk25ei8bm) **21**

[Кодировка](#_yl92p56zgqtg) 21

[ASCII-7 (7 бит)](#_mk3nasem06rp) 21

[UNICODE - стандарт](#_uackj8p102k9) 21

[UTF- Unicode Transformational Format (1991)](#_ug2np5c446hk) 21

[UCS-4](#_dvoioop1ziha) 22

[Проблемы UTF-16](#_9enk462bh4ah) 22

[UTF-32 - фактически QBCS](#_yt0yzej5egho) 22

[**Лекция 11.**](#_h1hb7i2jr1ey) **23**

[Тут пусто](#_h0qzv6696gcd) 23

[**Лекция 12.**](#_b5chs78k8nne) **23**

[Кодировки](#_yfpnjs37nbtg) 23

[Структурный базис языков программирования](#_12rld2dq3g36) 24

[Структурный тип - имеет структуру с точки зрения ЯП](#_p3417orrns2i) 24

[Множества](#_h2wv0lkff1hl) 24

[**Лекция 13.**](#_in7vmbqu07gx) **25**

[Базис ПООЯП (Процедурн. объектн. ор. ЯП)](#_qy8i1nicv7nr) 25

[Статически-типизируемые ЯП](#_zajd8ipum8xs) 25

[Динамически типизируемые](#_7hel3ltse8h6) 25

[Снова про кодировки](#_bexg0pjnmr7c) 25

[**Лекция 14**](#_1bvee5xnvlkm) **28**

[Массивы в статических ЯП](#_7djuj1iymqy5) 28

[Modula, Modula2, Pascal (Turbo), …](#_ww0pv3p2eu04) 28

[Oberon](#_os1twdrl6rhc) 29

[Java, C#](#_tcnyq09sb9ce) 29

[C](#_tb2yd14lh0k2) 29

[Golang](#_4uzmswu8q9pg) 29

[Массивы в динамических ЯП](#_tc59iee5ac7p) 30

[Python](#_e8dhuey7xnfv) 30

[Comprehensions](#_chz3n6v7sdxg) 31

[Кортежи](#_fkz4phv8ljji) 31

[Таблицы](#_gg16ork7p9ol) 31

[.NET](#_yjbbcas4k3ng) 32

[PHP](#_kl4eyi91idgd) 32

[**Лекция 15**](#_ez2q2sap0kla) **32**

[Немного о ТД: таблицы, массивы, записи, классы](#_2c5fp07gin8l) 33

[Коротко о ТД в динамических ЯП: Python, PHP, JS](#_94x3akymscx) 35

[**Лекция 16**](#_rae53sf7pclr) **36**

[Структурированное (структурное) программирование](#_r0yg9gdcmclw) 37

[ЯП с терминаторами](#_p0kxi9nvm8ie) 39

[Конструкция for each](#_z7blbzqb1kwc) 39

[**Лекция 17**](#_far61tr6xuev) **40**

[Средства развития ЯП](#_qwq0keb8lnbc) 40

[Подпрограммы с точки зрения передачи управления](#_m3awezjkyue) 41

[**Лекция 18**](#_o3fipu7a3yyp) **43**

[Go routines (Корутины в ГО)](#_rknhvxp4mnkx) 43

[Concurrency (взаимодействие между потоками)](#_trgp697ojs60) 43

[Многопоточное суммирование массива на ГО](#_jz507riungi6) 44

[Оператор Select-case](#_uaklk4ccrsyw) 44

[Timeout через ГОрутины](#_5y80aktqlwt0) 44

[Деревья на ГО](#_u2njo5432c9) 45

[Передача информации](#_vq98ng10stng) 45

[Способы](#_6ncjks26y0u7) 45

[in-out семантика](#_dpnu5e9lnd4a) 45

[Пример ADA](#_yvgfkuxq73d6) 46

[В С#](#_ub8m45pgajf9) 46

[Пример передачи параметров по имени ALGOL60](#_p8dbk7k1w39) 46

[**Лекция 19**](#_apxp1wvim2rc) **47**

[f(1, 2, 3) - позиционный способ вызова](#_kyfojdbztv5k) 47

[“Ключевой” способ именования параметров](#_hoka64kxbh92) 47

[Переменное число параметров](#_hyvbidbyrjcn) 47

[Перегрузка (Overloading)](#_ub4oowiouru4) 48

[Поиск подходящей функции](#_yham56ncmwux) 49

[**Лекция 20**](#_jxo99kuzat43) **50**

[Подпрограммный тип](#_a606o3xzb5dt) 50

[Анонимные функции](#_b55wzrxrsdpa) 51

[Замыкание: (closure)](#_rp1gzzcusc6x) 51

[В новом JavaScript](#_8cycsx1kwm5v) 51

[Python](#_tvsobjq4io2m) 51

[Появление лямбд](#_eyroxs2e5gae) 52

[lambda-функции в C++](#_bcbgnn3u9x1b) 52

[**Лекция 21**](#_khl0i2ye0o98) **52**

[Функции как объекты первого порядка](#_6sl3q6lh4lpy) 52

[Синтаксический сахар в c#](#_d1ucgp8htwhj) 53

[События](#_goigz1bmw44w) 53

[В 2003 году появились обобщения](#_mmnsbr89h3ut) 55

[2003 год - появились анонимные делегаты](#_aaovafnplttd) 55

[В 2008 году появились -выражения](#_shike667s7v) 55

[LINQ](#_1hu467aunaku) 55

[**Лекция 22**](#_q74co7av8slr) **56**

[Вложенные классы в Java](#_rt7lus2ffo9p) 56

[Анонимные классы](#_8ggpwpsj34d8) 56

[Обобщения (2005 г.)](#_feekxuwq9rkp) 56

[Ссылки на методы](#_5hic8pladebm) 57

[**Лекция 23**](#_t9xwugzb6dpx) **58**

[Модульность. Раздельная трансляция. Управление видимостью](#_hx9dxg1m1vx4) 58

[**Лекция 24**](#_w5r0tzjshbaa) **61**

[Видимости и все такое](#_vwxx3k5vxc1r) 61

[Односторонняя связь](#_2a1f8bjdb0ed) 61

[Общая схема](#_2us6o1gq40b) 61

[Ada](#_amncevzdnejx) 62

[Импорт в Java](#_arpk9xt9m8wn) 63

[**Лекция 25**](#_7pissolxq582) **64**

[Средства инкапсуляции](#_ozi4rd5pzemz) 64

[Разделение интерфейса и реализациии в Ада/Модула2](#_1llma0gyioj3) 64

[в Java есть пакеты](#_cufulihsnk1e) 65

[Расширение типа](#_bq4v8ms4g61) 66

[Приватные виртуальные функции в С+](#_u4it17eylmuk) 67

[**Лекция 26**](#_s3wl2mwer00w) **67**

[Мультиметоды](#_csb4aat8bxx9) 67

[Модификаторы доступа](#_91k1ipjb1uuc) 68

[Оберон](#_el5kucmqv991) 68

[Ада 95](#_fla5xquh4d7t) 68

[C++](#_l3fv193qrq8b) 68

[Друзья](#_c497v12sy93b) 68

[C#](#_xskhaxjqyv92) 69

[Java](#_jjcw7tum1fqv) 69

[**Лекция 27**](#_3i6t3eo86uls) **71**

[Обработка ошибок: исключения](#_4q8ogjal3mqw) 71

[Обобщенное программирование](#_ck8q7wyj9di3) 71

[Ада](#_egszibx5d33b) 71

[С++](#_px7a1fso31yv) 72

[Родовые абстракции](#_j2q0ulkrswa4) 72

[Специализация шаблона](#_k13okj459c1n) 72

[Полная (явная) специализация](#_ounivee8225s) 73

[Частичная специализация](#_wjkgleat7lp8) 73

[move - семантика](#_4l7xal69dnrv) 73

[**Лекция 28**](#_gts7m6uu6jyf) **74**

[Программирование на шаблонах](#_z2adl1iio3fz) 74

[Const expression (C++11)](#_qfbnpidymszi) 75

[Вариадические шаблоны](#_z26so3sifsvm) 75

[fold expressions (выражения - свертки)](#_wp54tfotwvxq) 76

[Обобщенные функторы](#_yi82i2cqfmqw) 77

[std::bind](#_k1bary6h2q0c) 77

[**Лекция 29**](#_pqurj57bzxz2) **78**

[Обобщенное программирование: C# и Java](#_j5jol695tt6n) 78

[C#](#_mn0i28hc6656) 78

[Механизмы вариации обобщенных классов](#_8lzn0qei8sib) 80

[ковариантность и контравариантность](#_mv5o7chlhnqw) 80

[Ковариантность](#_przuvxkvyze7) 80

[Контрвариантность](#_ptnggf31g883) 80

[Инвариантность](#_8z5emieu8g9q) 81

# Лекция 1:

Курс начал читаться в 1983 г. Роль ЯП в скиллах программиста падает (мнение лектора может не совпадать с реальностью, авторский текст сохранен).

Первый ЯП высокого уровня - FORTRAN (formula translator, 1954-1957). FORTRAN позволил избавиться от привязки программы к конкретной машине.

Что нужно знать программисту кроме C++?

* Компилятор
* Текстовый редактор
* Отладчик
* Стандартная библиотека

Дополнительные инструменты:

* Системы контроля версий (git, SVN)
* JIRA - Система управления проектом (система контроля за проектом), подробнее - [jira](https://ru.atlassian.com/software/jira)
* CI - (Continuous Integration) -- практика разработки ПО, в которой члены команды проводят интеграцию не реже чем раз в день. Результаты интеграции проверяются автоматически, обычно используя автотесты и статический анализ кода.
* SCRUM - спринты (демонстрация работы команды) каждые 1-2 недели, подробнее - [scrum](https://ru.atlassian.com/agile/scrum)

## Введение

### Парадигмы программирования:

1. Императивная (Фон-Неймановская)
2. Функциональная
3. Логическая
4. ООП

### 1. Императивная парадигма

Принципы Фон-Неймана:

Память - линейная, однородная

Состояние (программы?) определяется памятью

: Оператор меняет состояние, операция вычисляет выражение

Языки высокого уровня структурируют данные и вводят операции над ними.  
Основной оператор в этой парадигме - присваивание ( )

В языке ассемблера mov -- классическая и самая основная инструкция присваивания.

Кроме присваивания существует еще некоторый набор операций.

### 2. Функциональная парадигма

Основа -- вычисление функций -- f(a, b) .

У них (функций?) нет состояний.

Доказывать свойство программы просто

где - предикат входных данных и предикат результата. (?) Подробнее - [wikibooks](https://ru.wikibooks.org/wiki/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F/%D0%94%D0%BE%D0%BA%D0%B0%D0%B7%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE_%D1%81%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B9), [еще источник](http://www.dissercat.com/content/vyyavlenie-i-dokazatelstvo-svoistv-funktsionalnykh-programm-metodami-superkompilyatsii)

**FP** - первый функциональный ЯП (ну хоть его мы не учим)

Фон-Неймановские (императивные) программы тяжело распараллеливать (возможно, ФП - легко, не уверен)

тк функции не меняют внешнего состояния. В ФП автоматический анализ функций и поиск подходящих кандидатов для распараллеливания — это тривиальнейшая задача, как автоматический inline! В этом смысле функциональный стиль программирования соответствует требованиям завтрашнего дня.

### 3. Логическое программирование

См. [внезапно, здесь](http://mk.cs.msu.ru/index.php/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0_%D0%B8_%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(3-%D0%B9_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA))

### 4. ООП

ООП - объектно-ориентированное программирование, это надстройка над (1) - Императивной парадигмой. (содержательно, согласен)

Основа ООП - объект.  
Объект - экземпляр некоторого класса.  
У объекта скрыта внутренняя реализация (черный ящик, сокрытие деталей).  
Он может посылать сообщения другим объектам и определенным образом реагировать на сообщения сам.

Больше - в [wiki](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

# Лекция 2:

Парадигма программирования Язык программирования

## Деятельность программирования (делится на 3 части) :

1. Игровое программирование (прогр-е для себя). В ЯП главное, чтобы программисту нравилось (?). Пример - **BASIC** (???)
2. “Научное” программирование (прогр-е для себя и узкого круга коллег). Главное - результат. Пример ЯП - **Perl** (изначально создан для этого, потом стал промышленным). Средства - sed, awk (ну так сюда хоть bash пихай, хз зачем это тут)
3. Индустриальное программирование (написание программ для последующего их использования в отрыве от программиста)  
   ТП - технологическая потребность  
   КТП - критическая ТП  
   Модульность - это КТП для ЯП здесь.  
   Пример - **FORTRAN**.

## Всякие разные ЯПы:

**PL/1** (IBM), **Algol 68** (на его основе был создан **С** (но это не точно)), **Pascal**.

**Ada** - был признан военным стандартом. ЯП для систем реального времени (в боевых действиях). Требования - надежность, устойчивость (к отказам), ...

**Modula-2** - был нацелен на системное ПО; сейчас эту нишу занял **С.**

**GNAT** - GNU NYU Ada Translator

Курс изначально был основан на языке **Ada.**

## Основные выводы

ЯП не внедряются, а выживают. Предсказать это нельзя. Выживают, как правило, те ЯП, которые придумывает один человек.

Язык должен быть достаточно прост (при обучении первые программы должны появляться на 1-2 день)

Нет и не будет никогда единого ЯП. Лектор: выживает не наилучший ЯП, а первый, занявший нишу.

## Задача для индустриального ЯП:

<stdin> => [reverse] => <stdout>.

**C:**

#include <stdio.h>

#define MAX\_LEN 1024;

char buffer[MAX\_LEN];

int size = 0;

int main()

{

int ch;

while ((ch = getchar()) != EOF)

{

if (size >= MAX\_LEN)

{

fprintf(stderr, “too large\n”);

exit(1);

}

buffer[size++] = ch;

}

for (int i = size - 1; i > -1; i--)

putchar(buffer[i]);

}

Недостаток этой программы - статическое распределение памяти, писать не очень удобно.

# Лекция 3:

Беда использования realloc() - фрагментация памяти.  
Работа с динамической памятью - тонкая вещь.

**Modula-2** -> **Oberon** - 10 стр. описание языка (?).

ЯП был нацелен на системное программирование (СП). Первый СП-ЯП со сборкой мусора. Компилятор ЯП Oberon на Oberon - 4k строк.

Принципы выбора функциональности ЯП:

* Принцип сундука - берем все, что нужно (что может понадобиться)
* Принцип чемодана - берем все, без чего ЯП (наверное) не имеет смысла.

**Go** - язык с динамической сборкой мусора.

Объявления в Go: var <name> <type> [ = <value> ]  
Можно иначе, без явного указания типа <name> **:=** <value>

Вспомним задачу <stdin> => [reverse] => <stdout>.

Решение на **Go:**

import {

“put”,

“OS”,

“io/ioutil”,

“string }

func main() {

rdr := os.stdin // объявление переменной rdr

b, err := ioutil.ReadAll(rdr) //b - массив байт, ReadAll()

// возвращает кортеж

if err != nil {

panic (“Bad Input”)

}

b := string(b)

// альтернатива - s := string.Split(b, “”), “” - пустая строка

// в s будет срез (slice) массива

for (i:= len(b)-1; i>=0; --i {

fmt.print(b[i]) // могут быть проблемы с кодировками

}

}

*^ сверху какая-то дичь с кучей ошибок*

package main

import (

"os"

"io/ioutil"

"fmt"

"unicode/utf8")

func main() {

rdr := os.Stdin

b, err := ioutil.ReadAll(rdr)

if err != nil {

panic ("Bad Input")

}

str\_b := string(b)

fmt.Println(reverse1(str\_b))

fmt.Println(reverse2(str\_b))

}

func reverse1(s string) string {

output := ""

for i:= len(s); i>0; i-- {

output += s[i-1:i]

}

return string(output);

}

func reverse2(s string) string {

o := make([]rune, utf8.RuneCountInString(s));

i := len(o);

for \_, c := range s {

i--;

o[i] = c;

}

return string(o);

}

в го есть byte и rune

в reverse1 строка делится на byte и т к юникодные символы некоторые состоят из 2х и более получается каша

в reverse2 rune - отдельный символ utf-8

Решение на **C#**:

class Program

{

static void Main(String[] args)

{

var s = System.Console.In.ReadToEnd();

for (int i = s.Length - 1; i > -1; i--)

System.Console.write(s[i]);

}

}

Другой способ (в лекциях как “более объектный способ”):

class Program

{

static void Main(String[] args)

{

var s = System.Console.In.ReadToEnd();

var a = s.ToCharArray();

System.Console.Write(a.Reverse());

}

}

Решение на **Python**:

import sys

print(sys.stdin.read()[::-1])

# Лекция 4:

## Язык **С++**

generic (обобщенное) программирование - шаблоны (параметрический полиморфизм)

в **Python**, **JS** его нет и быть не может

Решение задачи на **C++:**

#include <algorithm>

#include <vector>

#include <iostream>

#include <iterator>

using namespace std;

int main()

{

vector<char> v;

copy (istream\_iterator<char>(cin), istream\_iterator<char>(),

back\_inserter(v));

copy (v.rbegin(), v.rend(), ostream\_iterator<char>(cout));

return 0;

}

Степень общности (абстракции) очень большая. При этом не теряется эффективность.

## Функциональное программирование

**Точка рассмотрения ЯП:**

|  |  |
| --- | --- |
| **Базис** | |
| **Скалярный базис** | **Структурный базис** |
| Встроенные типы и операции, операторы, выражения | Составные типы данных, массивы, структуры, составные операторы |

**LISP** ( “Чистый” LISP). **LISP** - List Processing.

(LISP не знаю, переписываю с лекций, чекните & поправьте если что)

У **LISP** Крайне простой базис (правила).

Базис **LISP’a**:

Типы данных: (атом) -> (символ, т.е. идентификатор ) | (целое число)

Средство развития: (S - выражение) -> (голова).(хвост)

Шаг вычисления: число вычисляется само в себя.

Список - частный случай S-выражения

Иллюстрации:

(something).nil - список  
() или nil - пустой список

(a.(b.(c.nil))) ~ (a b c) - средство облегчения нотации.

( (1 a) (2) 3 b ) - тоже список

Замечание: вообще правильнее рисовать так:



Основная структура данных **LISP** - список.

Программа - тоже список:  
(+ 5 3) #res = 8

(+ 5 3 8) #res = 16

(+ (- 3 1) 8) #res = 10

Определение своих функций:

(defun name (args) (body))

Встроенные функции:

* CAR (head) - рез-т атом или список, по сути это голова S-выражения
* CDR (tail) - рез-т всегда список (может быть пустым), по сути это хвост S-выражения
* (CONS a b) - конструирование точечной пары (a - голова, b - хвост)
* (NULL S) - является ли S пустым списком?
* (Atom X) - является ли X атомом
* T - символ TRUE
* () - пустой список, подразумевает FALSE
* ( COND (S11 S12) # как только S(N)1 дало истину - вычисляем S(N)2  
   (S21 S22) # аналог switch() { case S11: S12; break; case S21: ... }  
   … )
* ( IF (S) E1  
   E2  
  ) # if S then E1 else E2

Вспомним задачу <stdin> => [reverse] => <stdout>.

Решение на **LISP:**

( print (reverse (read)))

reverse может быть встроенный, а можно реализовать самому.

( defun reverse1 (S)

( IF (NULL S) ()

( append (reverse1 # в лекциях - просто reverse (?)

# добавили цифру в название, чтобы отличить от встроенной ф-ии

(cdr S)) (cons (car S)())

)

)

)

Диалектов у LISP много, самый популярный - Common Lisp.

<еще какая-то дичь типа reverse2, shift >

Отсутствует понятие состояния. Ф-я одного и того же аргумента возвращает один и тот же результат. Естественный параллелизм.

Недостаток: нет эффективных компиляторов.

(SET (QUOTE A) 2)

(SETQ A 2) # что-то вроде A:=2

DO (цикл?)

Теперь язык стал мультипарадигмальным, но этого нет в чистом LISP.

## Идея байт-кода:

построчная интерпретация

**Java**, **SCALA**, **KOTLIN** (альтернатива Java), **CLOSURE**

* дают код для JVM (Java Virtual Machine, виртуальная машина Java)

Python (оюн тут к чему?)

**.NET** -> **MSIL** (Microsoft immediate language) -- assembly -> JIT-компилятор (Just In Time, компилируется когда нужно)

GAC (Global Assembly Cache)

**Basic** -> **VBA** (Visual Basic for Applications)

# Лекция 5:

## REFAL

Язык **REFAL** - разработан в СССР в середине 1960-х, ответвление от Функционального программирования

-Аппликативное программирование

Есть поле ввода (им может быть, например, stdin)

Рефал работает примерно так: внутри блока идет последовательность предложений вида:

хрень1 = хрень2;

Хрень1 - шаблон входа функции, хрень2 - то, что функция должна выдавать, если ее вход соответствует этому шаблону. Например

=; - пустое выражение, если на вход подается пустая строка, то вернуть пустую строку

е.1 = False; - если на вход подается один символ, то выдать False

Хрень1 может состоять из символов, фигурных скобок и свободных переменных (это как раз e.1 в примере) Хрень2 помимо этого может содержать вызовы функций вида <Func arg1 … argn >. Допускается вложенный вызов функций.

Теперь подробнее о том, что такое свободные переменные: это штука вида

TYPE.ID

TYPE = e | s | t

s - 1 какой-либо символ

t - любой терм (символ либо выражение в скобках)

e - любое выражение

Вообще вот туть все есть понятненьким языком: http://refal.net/rf5\_frm.htm

Образец = преобразованный образец, фильтры

.

.

.  
Напоминает нормальные алгоритмы Маркова.

Образец записи, где s.1 - произв. литер, e.1 - произв. выражение:

{

s.1 e.1 s.1 = <palindrome, e.1> - вызов функции palindrome

s.1 = ;

=; - пустое правило

} /// я не знаю наверняка, но кажется так не будет работать (точнее строка просто сотрется или типа того. Но я не спец по рефалу, поэтому призываю сюда аяшников

Я так понимаю, что здесь имеется в виду просто проверка на палиндром, тогда её можно записать вот так:

Palindrom { s.1 e.2 s.1 = <Palindrom e.2> ;

s.1 = 'yes' ;

= 'yes' ;

e.1 = 'no' ; }

Решение нашей задачи с reverse:

$ENTRY GO  
{  
 < Prout <reverse <card>>>;   
}

reverse

{

s.1 e.1 = <reverse e.1> s.1;

=;

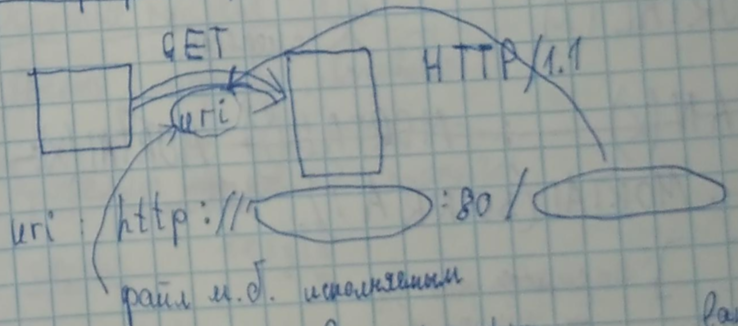
}

-- наиболее естественная форма записи алгоритма reverse

## Rust

Предназначен для низкоуровневого СП. Ниша языка С. Язык С - ненадежный.

## Web-программирование

Взаимодействие:  


**Perl** - Practical Extraction and Report Language

Для написания генерации веб-страниц.

**LAMP** - Linux Apache MySQL Perl (в дальнейшем - PHP)

## Логическое программирование

### Prolog

**Prolog** (1971) - алгоритм, проверяющий истинность любого утверждения, записанного на спец. языке.  
Хорновский дизъюнкт:

X - вектор переменных

FALSE => Q(X) док-во за O(n)

X <-> значения термов

Значения термов могут быть строками и числами

Пример:  
MAN(SOKRAT) == Сократ - человек

MORTAL(X) == Х - смертен

MORTAL(X) :- MAN(X) == если Х - человек: то Х - смертен.

-: MORTAL(SOKRAT)

[ ] -список

[x, y, z]

Сопоставление образов:  
[ x | z ]

^ ^---- tail

head

append(x, y, z) - предикат, z = xy

Наша задача на **Prolog:**

reverse( [ ], [ ] ). // reverse(empty\_list) == этот самый список.

reverse( [ X | Y ], Z) :- append(W, [X], Z), reverse(Y, W).

:- reverse([1, 2, 3], L). выдаст true, L = [3, 2, 1].

**SWI-Prolog** - реализация в Unix

## Основные понятия ЯП

Данные, операции, связывание:  
Программа обрабатывает некие **данные** с помощью **операций**

Рассмотрим строку

Ее длина (length) - операция или данные?  
В Turbo Pascal - данные.

Существует дуализм: данные операции

Свойства:

-set

-get

**Smalltalk**

Переменные:

* Класса (в С++ называются static члены класса)
* Экземпляра

Object

доступ к данным - всегда операция

### Геттеры/Сеттеры:

**C#**:

class X{

private int \_p;

public int p{

get { return \_p; }

set { \_p = value; }

}

}

Достоинство такого подхода - легко менять класс изнутри.

Тип Данных:

* Данные
* Операции

ТД = Мн-во Данных + Мн-во Операций

Структуры Данных - СД (?)

В абстрактных типах данных абстрагируемся от множества данных.

### 

### Механизм создания новых ТД

Модуль:

Интерфейс;

Реализация

Модуль здесь - это механизм инкапсуляции (сокрытия внутренностей)

Классовый подход:

class

public:

…

private:

…

protected:

....

Мультиметоды - легко реализуются в Модульных ЯП.

**Модульное программирование** - это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определенным правилам

**Мультиметод** - механизм, позволяющий выбрать одну из нескольких функций в зависимости от динамических типов или значений аргументов.

# Лекция 6

## Связывание (binding)

связывание: Сущность (конструкция) ЯП набор атрибутов

Время связывания:

* динамическое
* статическое

### Динамическое:

* в любой точке программы
* квазистатическое: при входе в блок; исчезает при выходе из блока

### Статическое:

* по выбору программиста ( переменная -- ее ТД)
* по выбору транслятора ( локальная переменная -- ее относительный адрес на стеке)
* по выбору компоновщика (статическая переменная -- абс. адрес, глобальная функция -- абс. адрес)

Связывание времени реализации:

char → signed | unsigned

Связывание времени создания ЯП:  
ЯП, где большинство критических связываний статические -- компилируемые(?)

Самое критическое связывание - связывание ТД

Объект данных (ОД) ←→ Тип данных

Связывание переменной и ее адреса тоже довольно критично.

Имя - способ ссылки на сущности ЯП.

Имя ⇔ ОД

В С/С++ - статическое, но во многих современных ЯП - динамическое.

ОДК - ОД классов

Статически связываются только ссылки на объект

class X { … }

X a; -- в С++ - объект в памяти стека (или в статической)

Java, C# - в динамической памяти (на стеке - только ссылки)

В С# есть struct, классы-обертки  
Упаковка/распаковка.

# Лекции 7, 8:

## Простые типы данных, операции над ними

Классификация простых типов данных:

1. Арифметические
   1. целые
      1. signed
      2. unsigned
   2. вещественные
      1. floating point
      2. fixed point
2. Символьные
3. Логические
4. Порядковые
   1. перечислимые
   2. диапазоны
5. Ссылочные и указательные
6. Подпрограммные (Их и … )

С++ не фиксирует целый тип ( в разных архитектурах может работать по-разному)

Java, C# полностью регламентируют размер и диапазоны значений всех типов

Ошибка:

char ch;

while ((ch = getchar()) != EOF) { ... }

т.к. int getchar(); то надо int ch; вместо char ch;

(внезапный факт о Go)

go run - интерпретация

go build - компиляция

Вещественные данные - формат с плавающей точкой:

, s - бит знака, М - нормированная мантисса, p - порядок

float - 32 бита; double - 64 бита

Кодировки (внезапно):

**ASCII:**

0..127 - English(Ну и ещё всякие спецсимволы вначале (NUL, CR, LF из известных) + цифры + часто используемые символы (.,;" пробел и т.п.)) ;

128..255 - другие символы, много проблем (Сюда запихивали символы, зависящие от кодировки, т.е., к примеру, в Windows-1251 здесь кириллица, в Windows-1252 здесь некоторые доп. латинские символы типа ÂÃȮ и т.п.)

**Unicode (UTF-16):**

2 байта на символ -- 0..65535(Это только базовая многоязычная плоскость. Сейчас для всего юникода надо 4 байта на код символа.)

char in Java/C# - UTF-16

(внезапный факт)

while (n-m) - в С++ можно (и в Java Script)

while (n-m != 0 ) - так нужно в Java и C#

Порядковые типы данных:

1. Перечислимые
2. Диапазоны

В C# и Java bool не приводится к целому типу.

Имена:

* Предопределенные
  + Базис, вшиты в стандартную библиотеку.
* Пользовательские
  + Определяющее вхождение
  + Использующее вхождение

Некоторые базисные классы предоставляются пользователю как обычные библиотеки (компилятор про них знает)

C#: string - обертка над базовым классом

Java: перегрузка имен / overloading -- полиморфизм

FORTRAN:

DO 5 I = 1, 3 // start of the DO loop

// action

5 CONTINUE // end of the DO loop

Трансляция - сложный JS ⇒ стандартный JS

Статический поиск определяющего вхождения:



throw X(); → статический поиск

catch (X&) → динамический поиск

(тут про поиск соответствующего типа данных, видимо)

## Объектно-процедурная парадигма

Схема рассмотрения:

* Базис
  + Скалярный - простые ТД
  + Структурный - составные ТД
* Средства развития
* Средства защиты

Переменные - ссылки на реальные объекты

ОД - объект данных ( референциальные ТД)

// перечисление типов данных, дано [тут](#_srtrxypgghtk)

Go / C#:

предопределение типов

int8, uint8;

int 16, uint 16;

…

int64, uint 64

в Go -> int, uint, uintptr.

IEEE 754: 52 бита - мантисса, 11 - порядок, 1 - знаковый

JavaScript:

Object, Function, Array | String, Number, Undefined

Битовые сдвиги:

* SAL, SAR - арифм., с учетом знакового биты
* SHL, SHR - логические, без учета знакового бита

# Лекция 9:

## Опять классификация типов данных

1. Порядковые
   1. Перечислимые (1995 - Java; 1999 - C#)
   2. Диапазоны
2. Ссылки / указатели
3. Подпрограммные типы (обсудим позже)
4. Символьный тип данных

### Перечислимые типы

Классические перечислимые типы:

type Enum is (const 1, const 2, …, const n) ⇽ попадают в область видимости

val(T,i)

ord(Const 1) ≡ 0

Из Pascal:

case exp [выражение порядкового типа] of

const 1:

const 2:

…

end;

Перечислимые типы появились в 1969 в Паскале. Исчезли в 1988 - Оберон. Появились снова в 1999 в C#.

type TColor is (Red, Green, Yellow)

type RGB is (Red, Green, Blue) Конфликт имён!

Давайте Red, Green сделаем нуль-местными функциями.

Перегрузка функций (но не в C++)

Только в языках, где у выражений обязательный контекст

Но: procedure P(X: TColor);

procedure P(X: RGB);

P(TColor “Red”);

3 возражения к перечислимому типу:

* Неявный трафик имён
* Нерасширяемость
* Ввод-вывод

**Решение первой проблемы:**

В C#: enum [: базовый тип] TColor { Red, Green, Yellow};

=0 =1 =2

TColor.Red

TColor c; int i;

(int) c; (TColor) i;

enum class TColor : unsigned char {...}

TColor::Red

**Решение третьей проблемы:**

В C# есть неявный класс-оболочка class Enum.

В 2005 в Java появились перечислимые типы:

enum TColor {Red, Green, Yellow};

это класс-потомок Enum

public RGB (int rgb) {

}

enum RGB {Red (0xFF0000), Green (0xFF00), Blue (0xFF)};

Проблема с расширением также решилась.

### Диапазоны:

Дискретный тип - тип, для которого определены succ, pred.



< > =

L .. R - Pascal

[L .. R] - Modula-2

Лучше всего реализовано в Ada.

имя\_базового\_типа range [L .. R]

Большая часть способов использования:

а) объявление массива

б) объявление индексной переменной

a [0 : 50]

array [Диапазонный тип] of базовый\_тип

array [D1, D2] то же самое, что array[...] of

array [...] of базовый\_тип

Нерасширяемость!

Выкинем диапазоны. Тогда объявление массива:

0 .. N-1

ARRAY N of T;

Как в Си.

### Ссылки и указатели

Беда в Си: появилась адресная арифметика, надёжность ухудшилась. В других ЯП отказались от указателей. Появились ссылки вместо них: var X = new X();

Java, C#, Delphi (Объектно-референциальная модель)

В Python, Javascript:



На самом деле в C# есть указатели:

byte [] b = new byte [N];

fixed (byte \*pb = b) {

… ⇽ здесь старый добрый С.

}

весь этот блок нужно поместить в блок unsafe, но при этом сборка станет небезопасной.

Но будем считать, что в C# нет указателей.

В языке Ada указатели строго на динамические структуры данных (Строгий ЯП).

### Ссылки:

В C++ не так, как в других ЯП ‘-;с объектно-реферативной моделью:   
ссылка - аналог имени.

Единственная операция с ссылкой в С++ - инициализация.

### Символьные типы данных

Текст ⇿ компьютер

аспекты:

* хранение
* обработка
* передача

string char

**Решения:**

50-е - 60-е:

1961 г. - IBM/360 - первый компьютер, в котором слово отлично от минимальной адресуемой единицы памяти.



# Лекция 10

## Кодировка

Для кодирования символа решили использовать байт.

### ASCII-7 (7 бит)

0, 1, …, 31, 32, …

0..9 A..Z a..z

В битах 1-31 - управляющие символы.

‘0’ + 1 = ‘1’

‘A’ + 32 = ‘a’

ANSI-8

0-127 - ASCII-7

128-255 - национальные кодировки

ISO-8859/1,2,3,4,... - расширенная латиница

Кириллица - КОИ-8 и еще несколько кодировок

У Microsoft: кодовые страницы (CP)

CP-866 - кириллица

457

CP-1251 - кириллица

1252 - отвечает ISO-8859

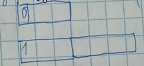
Кодовая страница определяет кодировку:

* charset - алфавит (набор символов) => 0..N ряд чисел
* правило представления в компьютере - code point (кодовая точка)

DBCS - double byte character set

SBCS - single byte character set

MBCS - многобайтовые кодировки (с переменной длиной)



### UNICODE - стандарт:

* универсальный набор символов для разных алфавитов
* категории букв: Lu (letter upper case), Ll - заглавная, строчная и т.д.

из - code point

* алгоритмы нормализации и декомпозиции
* кодировки

### UTF- Unicode Transformational Format (1991)

Первая версия Unicode фактически совпадала с UCS-2.

UCS-2 - universal character set (2 байта)

>60000 китайских иероглифов

### UCS-4

wchar\_t - в языке С

i18n - internatiolization

* глобализация
* локализация

Один и тот же .exe должен работать одинаково по всему миру.

str | cpy

str | cat

… |

wcs|cpy — принимают wchar\_t\*

tcs|xxx через макросы — принимают TCHAR\*, TCHAR задефайнен в виндовом заголовке как CHAR или WCHAR, в зависимости от дефайна UNICODE.

однобайтовые ->(перешли) UCS-2

“line” (char)

L”line” (wchar\_t)

L’ ‘

L” “

U+0400 - U+04FF - диапазон для базовой кириллицы

Фактически, в Windows -> Java -> C# используется UTF-16.

В 1996 г. поняли, что codepoint > U+FFFF.

Плоскость (plane) UNICODE:

0 : 65535

0 - 0xFFFF

BMP - basic multilingual plane (фактически UCS-2 / UTF-16)

Класс Encoding

string => byte[] - нужно указывать кодировку

### Проблемы UTF-16:

* избыточность
* не хватает кодов для всех символов

Плоскости Диапазон

BMP 0 - U+FFFF

1 U+10000 - U+1FFFF

2 2 2 - 16 дополнительных плоскостей

F U+F0000 - U+FFFFF

2^32

### UTF-32 - фактически QBCS

* удобное индексирование
* единственный недостаток - избыточность

В С++ sizeof(wchar\_t) == 4 вообще говоря (не у Microsoft).

(В C++ sizeof(wchar\_t) не определён. Под линуксами часто равен 4, под виндой 2. Но может быть и 1.)

# Лекция 11.

## Тут пусто

Кодировки: UTF-32 - 4 байта на code unit (проблема - big/little endian).

UTF-8 (1992 г.) - переменное число байтов.

0..127 -> 0xxxxxxx = ASCII-7

128..2047 -> 110xxxxx 10yyyyyy

# Лекция 12.

## Кодировки

Кодировки: UTF-32 - 4 байта на code unit (проблема - big/little endian).

UTF-8 (1992 г.) - переменное число байтов.

0..127 -> 0xxxxxxx = ASCII-7

128..2047 -> 110xxxxx 10yyyyyy

2048-65535 -> 1110zzzz 10xxxxxx 10yyyyyy

Кодировки UNICODE

UCS-2 (Microsoft)

UCS-4

UTF-32 (ранние версии Python)

UTF-16

UTF-8

Внутреннее представление текста (оно должно быть единым в ЯП)

UTF-32

Encode =>

Decode =>

2.7 => 3.3 utf-8 => 1251

Манифест UTF-8 везде.

У UTF-8 много плюсов

Недостаток - нет прямой индексации

Нужен byte-order маркер (BOM) (Позволяет определить, используются ли Little или Big endian, а также отличать UTF-8/16/32)

Но прямая индексация при обработке текстов часто не помогает:

é e’

Юникод не стандартизирует все (тонкости иероглифов)

Язык GO (Зафиксированное внутреннее представление текста)

.go - в UTF-8 всегда

len (S) - количество байтов.

S[0] - байт

Есть модуль Unicode/UTF-8 для работы с текстами, символами…

(Rune)

glyph - синоним int32

## Структурный базис языков программирования

### Структурный тип - имеет структуру с точки зрения ЯП

Стандартный Паскаль:

* массивы - последовательность однотипных элементов, непрерывно расположенных в памяти
* записи - последовательность неоднотипных, неоднородных и необязательно непрерывных (tuple)
* множества
* файлы
* строки

Отрываясь от стандарта Паскаля - таблицы(словари, хеш-таблицы)

Понятие записи из ЯП с динамическим связыванием (типизированием) исчезает постепенно (его нет в Python, ..)

файлы ушли из базиса ЯП

### Множества:

в Паскале

set of T

операции:

:=

x in S

Incl(S, X)

excl(S, X)

В Modula-2

BITSET: set of [0 .. N-1] появился для удобства

В Java нет, но есть битовые операции (>>, <<, ...)

Таблицы в статически типизируемых языках перешли в стандартную библиотеку.

add(T, K, V)

remove (T, K)

bool lookup(T, K, out V)

T[K] => N

IDictionary <T>

Dictionary <T>

Но в динамически типизируемых ЯП таблицы есть:

Python, словари:

d = {key1: value1, key2: value2, ...}

# Лекция 13.

## Базис ПООЯП (Процедурн. объектн. ор. ЯП)

**Какие структурные ТД остались:**

### Статически-типизируемые ЯП:

* массивы
* записи (в ООЯП может не быть, заменяется на классы)
* строки (кроме C++, где строки не встроены в ЯП)

(в C#, Java строки представлены как классы из библиотеки, но входят в язык:

С#: system

Java: java.lang .. SE - standart edition

ME

EE - enterprise)

### Динамически типизируемые:

* массивы
* строки
* таблицы/кортежи

ЯП SWIFT: плохой ЯП, строки SWIFT заслуживают внимания

Языки не внедряются - они выживают

Мультипарадигмальные ЯП: MOZART, LEDA

Java -> C# -> Go -> SWIFT каждый из них возник в своей корпорации, за каждым ЯП стоит 1-2 человека (конкретные)

У ЯП должно быть community для выживания

SWIFT еще не стал промышленным ЯП

Мы все-таки в этом курсе рассматриваем индустриальные ЯП

## Снова про кодировки

Проблемы в UNICODE:

Все упирается в понятие символ. Что такое символ?

é U+E6

e U+65 (?)

‘ U+301

Код языка - код страны

RU - RU

EN - US

EN - GB

Коды символов, кодирующих флаг:

(В общем, там есть набор букв от A до Z, который называется REGIONAL INDICATOR A (, B, C, ..., Z), если ставить по две таких "буквы" рядом, то может получиться флаг страны (например, RU = флаг РФ). Но в юникоде не стандартизировали, какие именно комбинации дают флаги, чтобы не вдаваться в политику (некоторые страны не признаются некоторыми другими странами и т.п.))

1XXXX

1F1XX

|

F5 - P

F7 - R

FA - U

F8 - S

U+1F1F7 - R

U+1F1FA - U

Если флага нет, рисуется “PR”

В SWIFT:

String

Character - это то, что может быть нарисовано в знакоместе

“U{1F1F7}U{1F1FA}” - символьный литерал

“eU{301}” - é (здесь два юникодовских символа) - это декомпозированная форма “U{E6}”

какое внутренне представление строк в SWIFT - неизвестно.

Если в ЯП нет понимания, как устроена строка = это плохо

S.utf8

S.utf16

S.unicodeScalars (похоже на utf-32)

S = “U{1F1F7}U{1F1FA}”

S.count = 1

S.utf8.count = 8

S.utf16.count = 4

S.unicodeScalars = 2

8 байт

var s = “Cafe”

print(s.count) = 4 (и если “КАФЕ”)

S+= “u{301}”

print(s.count) = 4

print(Array(S)) (до - “C”, “a”, “f”, “e”, после - “C”, “a”, “f”, “e”, “u{301}”)

S.utf8.count = 6

S.utf16.count = 5

S.unicodeScalars = 5

10 байт

let s1 = “Cafeu{301}”

let s = “Café”

if s1 == s

print(“Same”)

else

print(“Different”);

Будет напечатано Same (Эти строки считаются эквивалентными)

Возникает понятие индексации. Но не целым типом

let s = “Hello”

s[s.startIndex] //”H”

s[s.endIndex] //Error

s[index(before: s.endIndex)] - параметр (before) - обязательно должен быть хотя бы один

s[index(after: s.startIndex, offset: 3)]

Все эти навороты из-за проблем в графематике.

Массивы

Dx...xD - n штук

n - длина массива

Статические ЯП:

Одномерный массив - последовательность однотипных элементов, расположенных в памяти непрерывно

Требования выравнивания - примитивные типы данных стараются делать без дыр

В динамических ЯП однородность может исчезнуть

У массива всегда есть длина (Length)

Операции:

:= копирование массивов

[ ] операция индексирования - двухместная

[ ]: A, I -> Value

-> D& (в терминах C++)

Length: N

К какому типу принадлежит I?

До 90-х в этом вопросе нет конкретики:

с 1-цы, с 0 ; [-50:50]

В Pascal появилось понятие дискретного типа данных

Но в любом случае: [-50: 50] => 0..N-1

В ЯП, в которых дозволено использовать произвольные диапазоны, также есть операции(кроме длины):

A’LENGTH

A’RANGE

A’LOW

A’HIGH

(for i in A’RANGE)

Если Length чисто динамическая функция, возникает вопрос когда определяется размер?

Си:

нет := , поэтому в Си массив не объект первого порядка

char \*strcpy (char s1[], char s2[])

или char \*strcpy (char \*s1, char \*s2)

Здесь массив - это фактически шаблон статического размещения элементов в памяти.

D arr[];

D\* const arr

почти эквивалентны(особенно если слева стоит extrn)/

D arr[10]

Задача (типичная на знание Си)

char m[10][20];

T \*p = m;

T?

T = char[20] \* (хотелось бы, но так нельзя)

typedef char T[20]

такой подход позволяет эффективно управлять памятью

VLA - массивы (Массивы переменной длины):

void p(int n) {

char vla[n];

}

адреса локальных переменных становится вычислять сложнее -падает эффективность

Pascal

array[IndexT] of T

длина массива - свойство типа

Недостаток - нельзя написать универсальной процедуры работы с массивами

В ЯП Ada:

Есть неограниченные типы данных

и есть подтипы

type ARR is array(Integer range<>) of INTEGER

procedure P(A: ARR)

begin

for i in A’RANGE loop … A(i)

end loop

end

A1: ARR range 1..N

подтипы можно передавать в процедуру P

# Лекция 14

17.10.2017

## Массивы в статических ЯП

Length => статическое свойство типа

Гибкость <—?!—> Эффективность (при работе с процедурами и функциями)

В Ada:

Ограниченный тип статическое

Неограниченный тип динамическое свойство типа

procedure P(X1, X2: ARR)

X1 := X2; — компилятор вставит проверку динамически

i := 0;

X1[i];

Если у компилятора есть возможность проверить статически, он это сделает, иначе динамически.

Это называется квазистатический контроль

### Modula, Modula2, Pascal (Turbo), …

PROCEDURE SUM (VAR A: ARRAY OF INTEGER, :INTEGER)

VAR I, CNT : INTEGER;

BEGIN

…

0..HIGH(A)

### Oberon

(1988г.): выкинуты перечисл. типы, диапазоны…

int 0..N-1 — далее развитие шло так.

### Java, C#

T[] — спецификатор массива

T[] a; // имя ссылки

T[] a, b;

a = b; // копирование ссылки

a = new T[expr];

Длина является квазистатическим свойством массива, её нельзя изменить, но можно аллоцировать новый массив и скопировать туда старые данные.

Строки неизменяемы (немутабельны); для конструирования (манипуляции) строки StringBuilder.

+=

Метод toString (ToString в C#)

Неудобства с двумерными массивами:

int [][]q;

Элементами двумерного массива на самом деле являются ссылки на одномерные массивы.

int [][]q = new int[][10];

for (i = 0; i < 10; i++)

a[i] = new int[i];

Это так называемые ступенчатые массивы

То есть в Java все двумерные массивы ступенчатые.

Только в C#:

int [,]q = new int[10,20];

a[i,j] = 420; // обращение к элементу (i,j)

Это прямоугольные массивы (чистые двумерные массивы)

### C

POD — Plain Old Data — структуры в C (плоские структуры):

struct A {

char adr[104];

int b[4];

}; // в Java и C# такого не существует (?)

struct B {

char[] adr;

int[] b;

}; // это не плоская структура (??? таких типов нет в C)

### Golang

[N]T — длина массива является частью типа

var b [100]int

var c [100]int

b = c

Казалось бы, как Pascal. А где же гибкость?

Было введено понятие slice(срез). (Вместо подтипов)

В FORTRAN90:

I:A range 0..10;

A

появились срезы

В Go срез это, фактически, окно, через которое мы смотрим на массив.



var b [100]int;

var c []int; // срез

c = b[0:10]

**В Python:**

p[1:] # срезы являются способом

p[:] # появления новых массивов

p[1:-1]

p[i:j:s]

p[::-1]

В Go срез — самостоятельный тип.

Процедуры могут работать со срезами.

c = b[0..10]

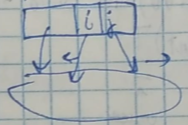
len(c)

c := make([]int, 50) // создаётся массив длины 50 в дин.памяти,

// срез указывает на него

с := make([]int, 50, 100) // --//-- длины 100, срез длины 50

Срез можно расширять.

 — wat

Есть возможность сделать как бы realloc.

Срез это как вектор в с++

## Массивы в динамических ЯП

### Python

Массив это список

l = [] # пустой

l = [1, 2, 3, 4, [“line1”, “line”, 3], 1.0]

l.append(5)

V — объект

dir(V) — выдаёт список методов *(на самом деле, всех имён в данном объекте, или в текущем пространстве имён)*

V = []; dir(V)

del l[-1]

del l[0]

#### Comprehensions

l = [i for i in range(1, 101)] # нормальные люди пишут это как

# l = list(range(1, 101))

или

l = [i + 1 for i in range(100)]

можно добавить условное выражение

l = [i + 1 for i in range(100) if i % 420 == 0]

enumerate(l) # -> [..., (индекс, элемент), ...]

l = [(1, 2), (2, 3), (3, 4)]

l = [(u, v) for (v, u) in l] # l == [(2, 1), (3, 2), (4, 3)]

Недостаток: очень легко загадить память. (видимо если большой цикл будет)

Массивы в Python динамические.

#### Кортежи

(В Golang как отдельный тип данных не существуют, но существуют во время компиляции)

x = …

y = …

x, y = y, x

a = (1, 2, 3)

b = (420,)

Есть индексация (похоже на массив)

Но: []: A,I -> D (а не &D)

0..N-1

^ к этому // што -> (Наверно, имеется в виду, что возвращается новая копия элемента D, а не ссылка на текущий элемент D в массиве. Т.о. нельзя непосредственно изменить элемент массива)

Кортеж — очень накладная структура (три ссылки в кортеже 54 б)

## Таблицы

Встроены в динамические ЯП и в библиотеки статических ЯП

В Perl называются хеши

В Python называются словари

### .NET

IDictionary<K,V>

Dictionary<K,V>

d = {“name1”: 1, “name2”: 3, “name3”: 9}

V = d[“name1”] # V == 1

Если ключа нет, генерируется исключение.

Исключение — авария (катастрофа). В остальных случаях исключения использовать не стоит.

try:

V = d[key]

except:

V = def\_value

*(то же самое пишется как V = d.get(key, def\_value))*

map может быть реализован через:

АВЛ-дерево (Адельсон-Вельский, Ландис — советские учёные)

Красно-чёрное дерево

Хэш таблицу (тратится больше памяти - и нет сортировки)

...

Чёткого аналога словаря в STL нет *(но как же std::map?..) (Думаю, имелось в виду, что ключи могут быть произвольного типа в одном словаре.)*

### PHP

В PHP массивы это хеши (словари)



# Лекция 15

24.10.2017

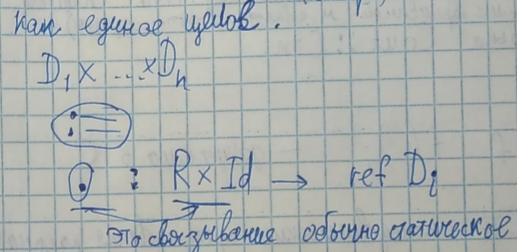
## Немного о ТД: таблицы, массивы, записи, классы

Таблицы отлетают в библиотеку в статических языках и входят в базис динамических языков.

Массивы всегда будут…

Записи (структуры) (ввёл Хоар). Появились в конце 1960-х. В то время запись - строка таблицы в файле.

Запись (структура) - совокупность (множество, набор) переменных, рассматриваемых как единое целое.



“.” нельзя перегрузить в С++.

“->” – можно.

record

x:integer;

z:real;

end record;

“.” – статическая операция в статических ЯП.

Во многих ЯП запись деградировала. Во многих заменена классом.

С++ (совместимость с С): struct \_tag{...}

struct Cout{

struct Cin{

}

} Cout;

В C Cin доступно как имя тега снаружи (имя переменной может совпадать с тегом структуры).

В C++ только Cout::Cin.

struct f f(); – допустимо в С.

В С++ и С# пользовательские преобразования разрешены. Страуструп хотел запретить их, но сломался на том, что при использовании комплексных чисел нужно писать слишком много перегрузок операций:

+(Complex, int)

+(int, Complex)

double, float…

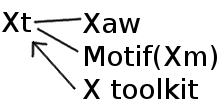
В Go комплексные числа входят в базис.

В Java записей (структур) нет.

В Go нет классов, но есть структуры, их моделирующие.

В C можно писать код “с использованием наследования и объектов...” (в объектном стиле):

Xlib (X Window)

 // *это часть примера*

struct Core {

…

}

struct Button {

struct Core \_corePart;

struct \_Button{...} \_buttonPart; //*в конспекте там buttonPard, имхо, описка*

…

};

В Go наследование реализуется через агрегацию. В C# есть структуры как компромисс класса с объектно-референциальной моделью.

Пример:

Point {

public int x;

public int y;

… //куча методов *(в конспекте написано ровно так)*

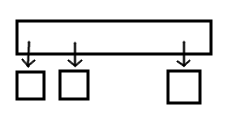
}

массив точек

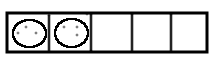
Point [ ] arr = new Point[1000];

for (int i = 0; i < 1000; ++i)

arr[i] = new Point();

 Память расходуется неэффективно.

Структура – “недокласс”.



У неё не может быть виртуальных методов. Они наследуются только от объекта. Их наследовать нельзя. Это чистые POD-структуры. //*для таких же эрудитов, как я: Plain Old Data – тип данных, имеющий жёстко определённое расположение полей в памяти, не требующий ограничения доступа и автоматического управления.*

void Foo(Object o) {...}

Point p;

Foo(p); //упаковка struct -> объект. (автоупаковка типа значения в объект)

Объект -> TD

TO



(Запись с вариантами) Объединение типов 

Паскаль:

record

постоянная часть

case Дискриминант: тип of // тип должен быть дискретным

константа 1: (запись 1);

константа 2: (запись 2);

…

end;



Неразмеченное объединение (как union в C):

type Word =

record

case boolean of

true: (i:integer);

false: (bits: packed array [1..48] of boolean);

end;

В C отсутствуют размеченные объединения.

Проблемы размеченных объединений – отсутствие защиты (надежность)... //*не до конца уверена, это ли имелось в виду. В оригинале “Проблемы размеч. объед. отсут. защиты (надёжность)...”*

WIMP

Windows icon mouse pointer

Событие : время.

Event



### Коротко о ТД в динамических ЯП: Python, PHP, JS

В динамич. ЯП:

Запись - просто набор переменных. Записи отсутствуют вообще.

Python:

v = (x, y, z)

v[0]

v[1]

v[2]

PHP:

$a = [ ];

$a[“name1”] = val1;

$a[“name2”] = val2;

$arr = compact(“x”, “y”, “z”);

$arr[“x”]

extract($arr)

JavaScript:

Object

Array

var O = new Object();

или {}

O.prop = val **[1]**

эквивалентно O[“prop”] = val => в JavaScript каждый объект по сути словарь.

Python:

dict = {}

dict[“prop”] = val **[2]**

У [1] и [2] одна и та же роль.

var O = {}

O[0] = “a”;

O[2] = “b”;

O[3] = “c”;

alert(O[0]) будет выдано a.

var s = “” Undefined undefined //*я не понимаю, к чему это здесь*

for (i = 0; i <= 3; i++)

s += O[i]; “a undefined b c”

В JavaScript объект подобен массиву.

В каждой функции можно:

this.arguments[0]

“0”

# Лекция 16

Операторный базис ЯП: самая главная задача - побочный эффект

:= - главный оператор (оператор присваивания)

Все остальные операторы - операторы передачи управления (control flow)

Операторы ввода/вывода почти ушли в стандартную библиотеку

Fortran:

DO

IF (e) M1, M2, M3

GOTO 5

GOTO M

ASSIGN 10 TO M

GOTO (1, 5, 28, 30), i

SUBROUTINE P(\*, \*, \*)

RETURN i

P(10, 20, 30)

IF (e) S чаще всего IF (e) GOTO L

1960г - статья “О вредности оператора goto” Дейкстры, т.к. очень тяжело разбираться в программах, где есть goto.

## Структурированное (структурное) программирование

Один вход и один выход у каждой управляющей структуры.  
P(x1, x2, … xn) - предикат, верный при входе  
Q(x1, x2, … xn) - предикат, верный при выходе  
Каждая структура преобразует предикат  
Пример: сортировка

Было доказано, что достаточно:

S1; S2 :=

while B do S

Но Дейкстра предложил избыточный набор, но более удобный и понятный:

v := e

S1; S2

if B then

S1

else

S2

while B do S

repeat S1, …, SN until B

for i := L to R do S

case e of:

C1:S1;

C2:S2;

else SN

end

Все они обладают свойством 

Операторы, заменяющие goto:

return;

break;

continue;

В Oberon:

LOOP

…

EXIT

END



while ((ch = getchar()) != EOF) {

//process

}

Если операция “подготовить” емкая?

prepare;

while (B) {

process;

prepare;

}

for (;;) {

prepare;

if (!B) break;

...

}

Oberon:

LOOP

prepare

IF not B THEN EXIT

EXIT

В этих языках есть составной оператор.

Есть также языки с явными терминаторами.

Преимущество языков с составными операторами: лаконичные, структура вложенная, но кажется не вложенной

if (B1)

S1

else if (B2)

S2

else if (B3)

S3

…

else

SN

### ЯП с терминаторами:

IF B1 THEN

S11; S12; S13

ELSE IF B2 THEN

S21; S22; S23

…

ELSE

SN

END END … END

Введено ключевое слово ELIF (ELSIF, ELSEIF)

В Java нет goto, но есть break

В Go:

if [<инициализирующий блок>;] e1

|  |  |
| --- | --- |
| Терминатор | Составной оператор |
| Ада  Модула-2  Алгол-68  Python (двумерный синтаксис) | C  Паскаль  Python (двумерный синтаксис) |

Python

....if B:

....S1

....S2

S3

Отсутствие отступа является терминатором.

### Конструкция for each

std::for\_each(it1, it2, f)

f - функтор ( класс с оператором() )

C#:

a - контейнер (например массив) или коллекция. Коллекцию можно итерировать.

foreach (int v in a) {

v ~ a[0]

v ~ a[1]

…

}

⬇

while (it.MoveNext()) {

it.Current

}

interface IEnumerable {

IEnumerator GetEnumerator();

}

⬇

Object Current;

bool MoveNext();

void Reset();

# Лекция 17

## Средства развития ЯП:

* модульность
* механизм создания новых типов (классы и т.п.)
* **подпрограммы**

Подпрограммы:

* понятие алгоритма в чистом виде
* это главная абстракция

SUBROUTINE P(A, B)

CALL P(X, Y)

FUNCTION EXP(X)

В Java, C# попытались уйти от подпрограммы, заменить их методами… Появился класс Math -- статический класс -- странный класс (Странный класс, потому что он нужен только для объявления статических методов. Т.е. если бы можно было объявлять просто функции без класса, в нём не было бы необходимости.), который играет только роль модуля

Math.exp вместо exp….

План:

* передача управления
  + подпрограммы vs сопрограммы
* передача информации
* подпрограммные типы
  + функции как объекты первого порядка
  + анонимные функции (lambda) функции
  + замыкание

## Подпрограммы с точки зрения передачи управления



В Fortran: (раньше)

ENTRY имя…

несколько входов и выходов (Там суть в том, что в фортране можно было в функции объявлять несколько разных точек входа и передавать в функцию несколько разных точек выхода.)

От этого отказались в современном программировании

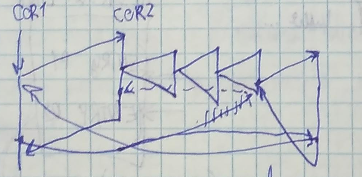
Несимметричность: вызвать можно из любой точки, а передача управления всегда в начало подпрограммы

Стековая дисциплина: размещения лок. переменных в памяти … появилась не сразу.

если сделать симметрично:

сопрограммы (Кнута-Конвея)





здесь у каждой сопрограммы свой стек.

Задача слияния двух файлов. С исп. сопрограмм алгоритм выглядит яснее

Modula-2: (COROUTINE)

PROCEDURE TRANSFER(VAR C1, C2: ADDRESS)

(Кстати: TYPE ADDRESS = POINTER TO ARRAY OF WORD;) (TYPE ADDRESS это аналог void\*)

-передать управление с сопрограммы C1 на сопрограмму C2



DEFINITION MODULE PROCESSES;

TYPE SIGNAL;

PROCEDURE STARTPROCESS(P:PROC; N:CARDINAL);

PROCEDURE SEND(VAR S: SIGNAL);

PROCEDURE WAIT(VAR S: SIGNAL);

END PROCESSES.

Монитор Хоара \_модуль….

MODULE M[4]

…FETCH…

…CONSUME…

IOTRANSFER (VAR C1, C2: ADDRESS; VAR …(прерывание):CARDINAL);

У сопрограмм общий сегмент данных, кода, свой стек у каждой.

Это похоже на потоки.

Но в сопрограммах не возникало проблем с синхронизацией. Сопрограммы - квазипараллельные процессы.

У сопрограмм Кнута-Конвея две особенности:

* RESUME TO (кому)
* свой стек

Частные случаи сопрограмм встречаются в современных ЯП

### Iterator-ы (Enumerator-ы)

C# 2.0, Java 2005

Python : \_\_next\_\_, \_\_iter\_\_

for x in C:

возвращаемый объект либо выбрасывает исключение stopIteration с версии 2.2 (или др.) появился генератор (функция в которой встречается yield):

def fib(n):

x, y = (1, 1)

i = 0

while i < n:

yield x

x, y = (y, x + y)

i += 1

n = fib(10)

type(n) //n - generator

next(n) //будет печататься новое число Фибоначчи

for x in fib(n):

…

Можно сделать так:

def fib()

x, y = (1, 1)

while true:

yield x

x, y = (y, x + y)

for i in fib(): //бесконечный цикл

…

В Python 3 range(n) стал генератором вместо списка из n элементов.

Сопрограммы Python-а.

|  |  |
| --- | --- |
| def sum():  s = 0  while True:  s += (yield)  print(s) | s = sum() // s будет типа генератор  s.send(None)  s.send(3)  3  s.send(5)  8 |

“(yield)” - yield-выражение (для приема данных)

yield <expr> - передать данные.

# Лекция 18

## Go routines (Корутины в ГО):

Горутина это корутина, с ГОшным сборщиком мусора, может быть чисто параллельной (а может и не быть, например, для io-операций)

пример:

go f(x,y,z) (запускает f в отдельной горутине - потоке)

Механизмов передачи управления (типа там yield и др) нету (в отличии от сопрограмм Кнута-Конвея)

Роб Пайк - Работает в Гугл - активно участвует в разработке ГО

### Concurrency (взаимодействие между потоками):

В Го используются каналы. Канал - легковесный типизированный pipe с размером.

пример:

c := make(chan int) // создать канал

c <- 5 // отправить в канал 5

x := <- с // получить из канала данные (5)

make(chan int, 0) == make(chan int) - канал размера 0. Т е   
При записи в канал он заблокируется, пока его кто-то не прочтет.

При чтении из канала он заблокируется, пока в него кто-то не запишет.

Т е пример выше вызовет дедлок.

Решение проблемы - make(chan, int, 1) // размер канала - 1 (capacity)

### Многопоточное суммирование массива на ГО:

a := [1,2,3,4,5,6,7,8,9]

func sum(s []int, c chan int) {

cnt := 0

for \_, val := range(s) {

cnt += val

}

c <- cnt

}

c := make(chan int)

go sum(a[0:len(a)/2], c)

go sum(a[len(a)/2:], c)

x := <- c

y := <- c

fmt.Println(x,y,x+y)

### Оператор Select-case

Смотрит какие из вариантов не залочены и выполняет случайный из доступных.

Select {

case <-c: // варик когда канал не пуст, вероятно

s1

case c <- x: // а это варик когда канал не полон?

default:

...

}

Если вызов любого оператора залочит программу - вызовется default.

### Timeout через ГОрутины

timeout := make(chan bool)

go f(){

time.Sleep(5 \* time.Second)

timeout <- true

}() // тут же объявили и вызвали функцию в другом потоке

for {

select {

case <- timeout:

fmt.Println(“Expired”)

default:

S1

}

}

функция, реализующая это за нас

c := time.After(5 \* time.Second) // func After(d Duration) chan time

### Деревья на ГО

struct Tree {

Left \*Tree

Right \*Tree

Value Int

}

func same(t1, t2 \*Tree) bool // сравнить 2 дерева

func walk(t \*Tree, c chan int) { … }

walk - функция обхода дерева (например в глубину)

same - вызывает walk - получает данные из каналов и сравнивает их

## Передача информации

Передача параметров

call P(arguments)

Фактические параметры -> Формальные параметры

Важно различать:

* формальный параметр — аргумент, указываемый при объявлении или определении [функции](https://www.wikiwand.com/ru/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).
* фактический параметр — аргумент, передаваемый в [функцию](https://www.wikiwand.com/ru/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) при её вызове;

### Способы:

1. По значению
2. По результату
3. По значению/результату (комбинированный подход)
4. По адресу (ссылке)
5. По имени

### in-out семантика

1. IN

Перед Call фактические параметры -> формальные параметры

1. OUT

Перед Return формальные параметры -> фактические параметры (“перед Return” ⇔ до конца функции)

1. IN-OUT

Перед Call фактические параметры -> формальные параметры

Перед Return формальные параметры -> фактические параметры

#### Пример ADA:

(x: **in** integer

y: **in** integer

z: **in out** integer) - от этого отказались, потому что в разных компиляторах - разное поведение и программа выдавала непредсказуемые результаты

procedure P(x, y: in out integer)

x := val1

raise Error

y := val2

P(A, A) - в разных компиляторах разное поведение

В современных ЯП обычно выбирают один-два способа передачи параметров:

1. Си - по значению
2. С++ - по значению и ссылке
3. В объектно-референциальных ЯП (типа питон): по значению (но надо помнить, что тут все значения - ссылки на объекты)

В некоторых языках можно явно указывать in, out, in-out семантику

#### В С#:

void f(ref int a) {

a++;

}

int x;

f(ref x); // ошибка - x неинициализированная переменная

void f(out int a) {

a++;

}

// насколько я понял тут ок

#### Пример передачи параметров по имени ALGOL60:

procedure P(x);

integer x;

P(i) P(a+b) // x будет a+b а не их фактическая сумма

Это похоже на макроподстановку

На этом языке нельзя (из-за такой передачи параметров) реализовать SWAP.

procedure P(x);

integer x,y; integer tmp;

begin

tmp := x; x := y; y := tmp

end

integer i;

integer A[1:10];

i=5;

swap(A[i], i); // не будет работать так как параметр не вычислится

swap(i, A[i]); // не будет работать так как параметр не вычислится

**Костыль**

thunk - процедура, которая передается, которая передается вместо параметров (??). Она вычисляет параметры внутри

# Лекция 19

* control flow
* data flow

-> - синтаксис вызова

## f(1, 2, 3) - позиционный способ вызова

void f(t1 x, t2 y, t3 z)

void f(int x, int y, int z)

## “Ключевой” способ именования параметров:

f(x : 1, z : 3, y : 2)

здесь удобно реализовать параметры по умолчанию.

В Python есть оба способа:

def fun(x, y, z)

def fun(\*args, \*\*kwargs)

\*args - кортеж, \*\*kwargs - словарь с ключевыми аргументами

## Переменное число параметров:

printf(“count = %5d\n”, cnt);

cout << “count = ” << cnt << endl;

cout << a[i] << i++;

(вывод не стандартизирован:

operator << (operator << (cout, a[i]), i++);

какой порядок вычисления параметров?)

|  |  |
| --- | --- |
| **Включение модулей:** |  |
| Turbo Pascal | Модула-2 |
| uses M; | IMPORT M;  M.entity  FROM M IMPORT entity |

**В Oberon:**

.Writestring

InOut.Writestring(‘count’);

InOut.WriteInt(int);

InOut.WriteLn

Вывод громоздкий

**В ЯП C/C++:**

…

va\_list

va\_next

в случае ошибки нет безопасности

**В ООЯП:** (чисто ОО ЯП)

Object

↑

Integer(i)

Int32(i)

(Тогда переменное число параметров - это массив из Object-ов)

void f(int x, params Object[] args)

params T[] args

void f(T ...args)

f(x, y, z) => T[] args

## Перегрузка (Overloading)

Область видимости:



Вложенные области видимости

hiding (скрытие)

Это не очень хорошая ситуация

T name (T1 x, T2 y)

T name (T1 x, T2 y)

В классе наследнике Это override (замещение)

Перегрузка стандартных операций на самом деле уже была в Си, Pascal… : +, \*, …

Имена из разных пакетов:



Возвращаемое значение не учитывается при разрешении перегрузки в ЯП C++, Java, C#, …(Т.е. нельзя перегружать функции, изменяя исключительно тип возвращаемого значения.)

Но

В Ada:

function F return Boolean;

function F return Float;

if F then

d := F \* 3.0

Функции в Ada можно вызывать только в контексте выражения.

Перегрузка операций:

C++: нельзя перегружать:

**.**

**.-> (это как вообще?)**

**::**

**?:**

Перегружать можно только для классов.

В Java знаки операций перегружать нельзя.

В C# арифметико-логические операции можно, экзотические - нет.

“=” нельзя, потому что это операция над ссылками

T + T - это обязательная статическая функция static …

### Поиск подходящей функции:

package M is

TYPE T is … ;

function “+” (X, Y: T)

return T;

end M;

A : M.T

A, B, C : M.T

C := A + B (ошибка)

C := M. “+”(A, B) (правильно)

так писать в больших выражениях неудобно.

В ЯП с областями видимости и перегрузкой стандартных знаков операций есть проблема:

#include <iostream>

cout << “Hello, world” << endl; (ошибка)

std::cout << “Hello, world” << std::endl;

<< - ведь это тоже в std лежит! (как мы это увидели!)

ostream & operator <<

Если компилятор видит функцию, он ее ищет сначала в текущей области видимости, затем в том пространстве имен, где лежат типы ее параметров.

namespace M1 {

class X {...};

void f(X);

}

M1::X a;

f(a) //правильно (ошибки нет)

using T = std::cout;

новое имя (переименование)?

# Лекция 20

Javascript

var o = {} //пустой объект

function f(x) { return x + 1 }

o.g = f

o.g(1)

o.g() //будет передан undefined

o.g(1, 2, 3)

function ff(i, j){...}

o.g = ff (“перебивание”)

При описании функции у глобального объекта появляется свойство

Как таковой перегрузки нет.

В Python:

У нас есть { \*args }, и перегрузка не нужна.

{\*\*KWargs}

Настоящая перегрузка -- это перегрузка по типу… статическая типизация

## Подпрограммный тип

В языке Си -- это адрес (как и в ассемблере)

В Oberon, Pascal… : (есть процедурный тип)

TYPE PP = PROCEDURE (VAR: INTEGER; REAL);

аналогично строке на Си:

typedef void (\*pp)(int, double);

PP x;

void foo(int &i, double d) {...}

x = foo;

void (\*signal)(int sig, void (\*h)(int))

(int sig);

Но нужно:

typedef void (\*handler)(int sig);

handler signal(int sig, handler h);

Как передавать функцию в качестве аргумента другой функции?

Если нет процедурного типа, можно использовать generic-и (если они есть). (Так делалось в Ada до 95-го года)

В Ada 95 введен подпрограммный тип, указатель не только на объекты из динамической памяти, потому что нужно общаться с внешним миром… API ОС на ЯП Си

#### Анонимные функции:

var g = function(x, y) {return x + y;}

g(1, 2) -> 3

## Замыкание: (closure)

**Замыкание** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *closure*) в программировании — [функц](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%B3%D0%BE_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0)ия, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами. Говоря другим языком, замыкание — функция, которая ссылается на [свободные переменные](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F) в своём [контексте](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%BE%D0%BD%D1%82%D0%B5%D0%BA%D1%81%D1%82).

Замыкание, так же как и [экземпляр объекта](https://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D0%B7%D0%B5%D0%BC%D0%BF%D0%BB%D1%8F%D1%80_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%B0), есть способ представления функциональности и данных, связанных и упакованных вместе.

Замыкание — это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. Синтаксически это выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции.

В случае замыкания ссылки на переменные внешней функции действительны внутри вложенной функции до тех пор, пока работает вложенная функция, даже если внешняя функция закончила работу, и переменные вышли из области видимости.

Замыкание связывает код функции с её лексическим окружением (местом, в котором она определена в коде). Лексические переменные замыкания отличаются от глобальных переменных тем, что они не занимают глобальное пространство имён. От переменных в объектах они отличаются тем, что привязаны к функциям, а не объектам.

function make\_incr(n) {

function adder(x) {

return x + n;

}

return adder;

}

var x = make\_incr(3);

var y = make\_incr(10);

Можно производить:

function make\_incr(n) {

return function(x) {return x + n;}

}

#### В новом JavaScript:

------||-------

return x => x+n;

------||-------

(список параметров) => блок.

Это механизм lambda-функций.

lambda-исчисление Чёрча: λx.x λx.λy.x+y

(LAMBDA (X Y)(+ X Y)) в Lisp

Это выражение дает результат: анонимная функция

(LAMBDA (X) (+X N))

Переменная n захвачена

В некотором смысле это синтаксический сахар.

#### Python:

def make\_incr(n):

return lambda x: x + n

#### Появление лямбд:

Java - 2014

C# - 2002 (08?)

C++ C++11

Роль анонимных функций в C++ могут играть функторы

size\_t count\_if(it1, it2, f){

auto cnt = 0;

while (it1 < it2)

if (f(\*it1++))++cnt;

return cnt;

}

bool f(int i) {return i >=0;}

vector <int> v;

size\_t c= count\_if(v.begin(), v.end(), f);

class FunctorN {

int \_n;

public:

FunctorN(int N): \_n(N){}

int operator()(int x) {return x + \_n;}

}

FunctorN x(3), y(10);

#### lambda-функции в C++

count\_if(v.begin(), v.end(), [](int i) { return i >= 0; })

[захват](список аргументов)модификаторы -> rettype {блок выраж-в}

rettype - он может автоматически определяться компилятором, но не всегда

Компилятор фактически создает анонимный функтор.

Захват

[список захваченных переменных]

id, &id, this, = , &

/\

|

область действия не расширяется

[a, &] (Такая запись значит захватить a по значению, а всё остальное — по ссылке.)

c++ lambdas: https://habrahabr.ru/post/66021/

# Лекция 21

## Функции как объекты первого порядка.

с# вышел в 1999 году. Java - 2005.

2.0 - 2003. 2014 - появились -функции.

3.0 - 2008.

7.0 - 2017

В 1999 в c# уже были delegate (делегаты).

System.Delegate

System.MulticastDelegate

### Синтаксический сахар в c#:

try {

…

} finally {

…

}

using(var a = new X()) {

…

}

delegate int f(int i);

// теперь f - имя типа

f g;

// g нужно присвоить адрес метода

int k = g(0);

Пример:

class Y {

public static int ff(int i) { … }

public int fff(int i) { … }

}

Y y; // ? = new Y(); ?

g += Y::ff; // ? g += Y.ff; ?

g += y.fff;

Операции над MulticastDelegate:

* () - рассылка // т.е. вызов метода
* = - инициализация
* += - подписка
* -= - отписка

<https://metanit.com/sharp/tutorial/3.13.php>

## События

*Событие*, это не что иное, как ситуация, при возникновении которой произойдет действие или несколько действий. Говоря языком программного моделирования, *Событие* — это именованный делегат, при вызове которого будут запущены все подписавшиеся на момент вызова события методы заданной сигнатуры. Эта трактовка хоть и раскрывает всю суть структуры события, но не только сбивает с толку начинающих «шарп-прогеров», но и не дает возможность рационально представить в программистской голове весь смысл.

Итак, *Событие*, это ситуация, при возникновении которой произойдут некоторые действия. Само событие имеет определенную структуру.

Example:

class ClassCounter *//Это класс - в котором производится счет.*  
 {  
 public delegate void MethodContainer();  
  
 *//Событие OnCount c типом делегата MethodContainer.*  
 public event MethodContainer onCount;  
  
 public void Count()  
 {  
 for (int i = 0; i < 100; i++)  
 {  
 if (i == 71)  
 {

OnCount();  
 }  
 }  
 }  
 }

class Handler\_I *//Это класс, реагирующий на событие (счет равен 71) записью строки в консоли.*  
 {  
 public void Message()  
 {  
 *//Не забудьте using System*   
 *//для вывода в консольном приложении*  
 Console.WriteLine("Пора действовать, ведь уже 71!");   
 }   
 }

class Handler\_II  
 {  
 public void Message()  
 {  
 Console.WriteLine("Точно, уже 71!");  
 }   
 }

class Program  
 {  
 static void Main(string[] args)  
 {  
 ClassCounter Counter = new ClassCounter();  
 Handler\_I Handler1 = new Handler\_I();  
 Handler\_II Handler2 = new Handler\_II();  
  
 *//Подписались на событие*  
 Counter.onCount += Handler1.Message;  
 Counter.onCount += Handler2.Message;  
 }  
 }

Комментарий: опять же - отсюда непонятно, что это за событие - и чем он отличается от делегата (если убрать event - код будет работать, обычные делегаты в шарпе - мультикаст). Кратко - event - это механизм защиты мультикаст делегата, который мы считаем событием. event скрывает исходный делегат (делает его private) и перегружает += -= делая их потокобезопасными. Теперь, везде, кроме класса создателя мы можем только подписаться на событие, но не можем, например, его вызвать

### В 2003 году появились обобщения:

delegate R foo<T,R> (T x);

Что это такое по сути? Это заготовка для генерации объявлений.

В стандартной библиотеке описано:

delegate R Funct<T,R>(T x);

### 2003 год - появились анонимные делегаты:

Func<int,int> f = delegate (int i) { … return …; }

Это параметрические делегаты, доступные из system. Первые n параметров входные значения, последнее - выходное. void - нельзя, для этого есть Action

### В 2008 году появились -выражения:

Func<int,int> f = (x) => x+1; // => { return x+1;}

Обычные формулы можно объявлять так же.

## LINQ

IEnumerable<int> Coll = …;

int n = 2;

c = Coll.Select( (x) => x+n )

Coll.OrderBy( (x) => x)

.Where( (x) => x%2==0)

Coll.Where( (x) => x%2==0 ).OrderBy( x => x);

^трешак, смотреть здесь, все оч просто:

<https://docs.microsoft.com/ru-ru/dotnet/csharp/linq/>

example:

class LINQQueryExpressions  
{  
 static void Main()  
 {  
   
 // Specify the data source.  
 int[] scores = new int[] { 97, 92, 81, 60 };  
  
 // Define the query expression.  
 IEnumerable<int> scoreQuery =  
 from score in scores  
 where score > 80  
 select score;  
  
 // Execute the query.  
 foreach (int i in scoreQuery)  
 {  
 Console.Write(i + " ");  
 }   
 }  
}  
// Output: 97 92 81

# Лекция 22

Java:

double Integrate(double A, double B, double EPS, Integrand (это интерфейс) i)

interface Integrand { double F(double x);}

## Вложенные классы в Java:

Если связь между объектом внутреннего класса и объектом внешнего класса не нужна, можно сделать внутренний класс статическим (static). Такой класс называют вложенным (nested).

Применение статического внутреннего класса означает следующее:

* для создания объекта статического внутреннего класса не нужен объект внешнего класса
* из объекта вложенного класса нельзя обращаться к нестатическим членам внешнего класса

Вложенный класс имеет доступ к членам своего внешнего класса, в том числе и к закрытым членам. Однако, внешний класс не имеет доступа к членам вложенного класса. Вложенный класс при этом является членом внешнего класса.

Статический класс объявляется ключевым словом **static**. При этом класс должен обращаться к нестатическим членам своего внешнего класса при помощи объекта, т.е. он не может обращаться напрямую на нестатические члены своего внешнего класса. На практике подобные классы используются редко.

class Outer {

\*

(class Inner {...} ) - что блин это значит в лекциях?

void foo() {

int n;

class Inner { … n … }

}

…

}

Нормальный пример:

// внешний класс  
class outerClassName {  
   
 private **static** class innerClassName {  
 // тело вложенного класса  
 }  
}

## **Внутренние классы**

Есть вложенные классы (nested) и внутренние классы (inner). nested == inner static. Внутренний класс имеет доступ ко всем переменным и методам своего внешнего класса и может непосредственно ссылаться на них.

Внутренние классы создаются внутри окружающего класса:

// внешний класс  
class Outer {  
 int outer\_x = 9;  
   
 void test() {  
 Inner inner = new Inner();  
 inner.display();  
 }  
   
 // внутренний класс  
 class Inner {  
 void display() {  
 Log.i(TAG, outer\_x);  
 }  
 }  
}  
  
class MainActivity...{  
 // В методе onCreate() активности  
 Outer outer = new Outer();  
 outer.test();  
}

Внутренний класс **Inner** определён в области видимости класса **Outer**. Поэтому любой код в классе **Inner** может непосредственно обращаться к переменной **outer\_x**. Когда мы создаём экземпляр класса **Outer** и вызываем метод **test()**, то создаём также экземпляр класса **Inner** с вызовом метода **display()**.

Кстати говоря, внутренний класс можно определить не только на уровне класса, но и внутри метода или внутри тела цикла.

Если понадобится создать объект внутреннего класса не в статическом методе внешнего класса, тип этого объекта должен задаваться в формате ИмяВнешнегоКласса.ИмяВнутреннегоКласса.

Объект внутреннего класса связан с внешним объектом-создателем и может обращаться к его членам без каких-либо дополнительных описаний. Для внутренних классов доступны все элементы внешнего класса.

Если вам понадобится получить ссылку на объект внешнего класса, запишите имя внешнего класса, за которым следует точка, а затем ключевое слово **this**.

### Анонимные классы:

Button btn = new Button() {

protected void OnClick() { … } }

class Outer {

protected Button Make\_Btn() { return new Button() { … тут захват (вероятно, объекта полученного из new Button() ?) }

...

}

Существует разновидность внутреннего класса, которая называется анонимным классом, так как у него нет имени. Подобные классы очень часто встречаются в примерах на Android. Например, когда вы пишете код для щелчка или других событий.

seekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {  
 @Override  
 public void onProgressChanged(SeekBar seekBar, int i, boolean b) {  
   
 }  
  
 @Override  
 public void onStartTrackingTouch(SeekBar seekBar) {  
  
 }  
  
 @Override  
 public void onStopTrackingTouch(SeekBar seekBar) {  
  
 }  
});

В этом коде вы используете конструкцию **new SeekBar.OnSeekBarChangeListener()**, но обходитесь без создания переменной для этого класса.

### Обобщения (2005 г.):

аннотация:

@FunctionalInterface

interface Foo {

int bar (int i);

}

С 2014 года - появились lambda-выражения:  
(int x) -> x+1 или

(int x) -> {return x+1;}

Для реализации интеграла:

interface Foo {

static double var(double i);

}

double Integrate(double A, double B, double Eps, Foo i);

Integrate(0.0, 1.0, 10e-8, (x)->x\*x) или так:

Integrate(0.0, 1.0, 10e-8, new Foo() { public double bar(double x) {return x\*x;}})

в нашем случае interface Foo - функциональный интерфейс, по сути - обертка какой-то одной функции для передачи ее в качестве аргумента. То есть по факту мы не передаем функцию, а передаем объект, который имплементирует этот интерфейс (а значит и нужную нам функцию)

## Ссылки на методы:

class X {

public static double sqr(x) {return x\*x; }

}

Integrate(0.0, 1.0, 10e-8, X::sqr),

coll.stream().map(x -> x \* x).filter(x -> x % 2 != 0).sort( (a,b) -> a - b).count()

^--------- получить поток

interface Comparator <T> { int compare (T a, T b); }

class Person {

private string \_firstName;

private int \_age;

public int get …

… set …

…

}

// coll это Map например

coll.stream().sort((p1,p2)->p1.GetName().compareTo(p2.GetName()))

coll.stream().sort.comparing(Person::getName\*())  
  
options:  
Person::getAge()  
p->p.getName()

p->p.getAge()

.reversed (у Comparator - это <неразборчиво> по умолчанию)

Реализация по умолчанию методов в интерфейсах (default)

interface … {

default int func() { … }

}

interface IDerived extends IBase { … }

# Лекция 23

## Модульность. Раздельная трансляция. Управление видимостью.

* физические модули
* логические модули

Первые логические модули в истории - подпрограммы. Виды трансляции:

1. Пошагово транслируемые: Basic, bash (в bash модульность не нужна)
2. Инкрементная трансляция. Интерактивные отладчики. Что-то поменяли, перетранслировали небольшой измененный кусок.
3. Цельная трансляция: стандартный Паскаль, изначальный JS

Зачем в JS появились модули (в 2015 г.)?

До этого эту (какую?) проблему решали библиотеки.

В Python, JS модульность можно реализовать средствами самого языка. Можно моделировать модули объектами.

4) Раздельная трансляция

Возникает понятия контекста трансляции (КТ). Это совокупность имен, полное знание о которых необходимо для трансляции (выполнения) данного куска программы. Откуда транслятор берет эту информацию?

Из декларации -- места, где объявляются (определяются) имена.

Единица Компиляции (ЕК) ~ модуль

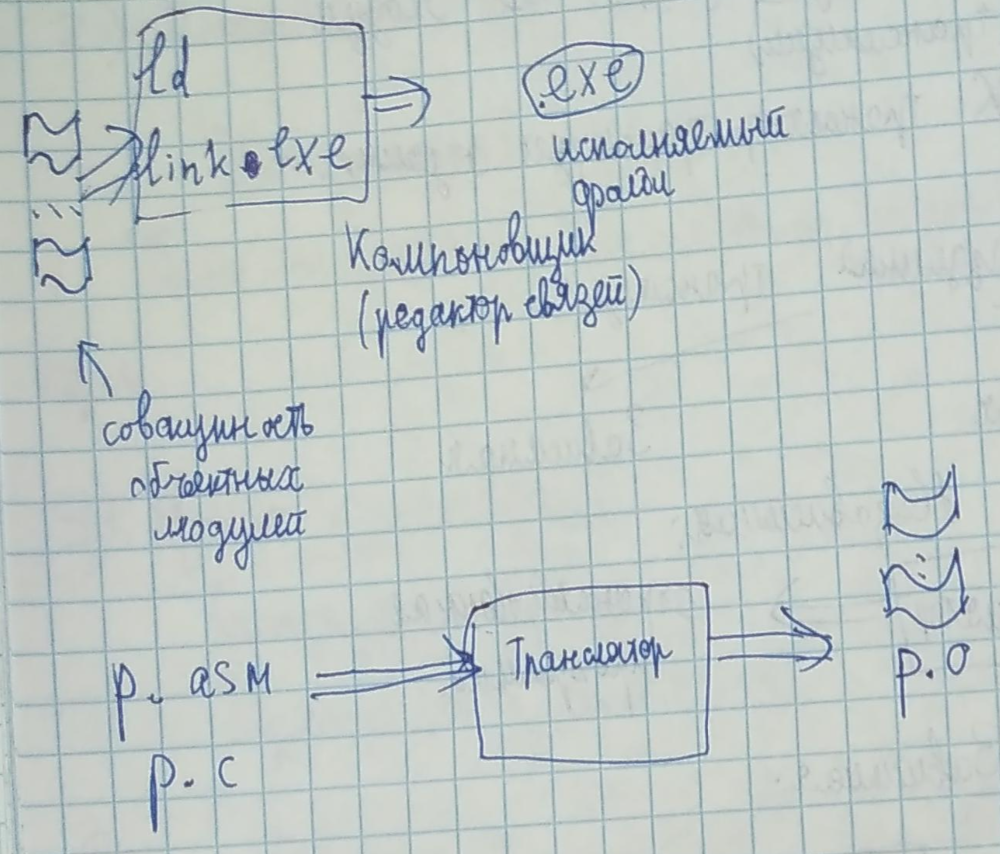
(трансляции -----^ )

Каждую ЕК транслятор транслирует отдельно.  
Раздельная трансляция:

* Независимая: ЕК ⇒ Транслятор ⇒ оттранслированная программа  
  
* Зависимая: ЕК ⇒ Транслятор ⇒ оттранслированная программа  
   ⇔  
   Трансляционная библиотека

Рефлексивность: (во время выполнения известны все свойства объектов из исходного кода)

* нерефлексивные: Asm, C, C++, ADA, MODULA-2
* рефлексивные: C#, Java, ...

   
^--- это пример трансляции, видимо.  
  
В C/C++ - раздельная, независимая трансляция  
КТ (контекст трансляции) должен задаваться в тексте модуля.

В Ассемблере:  
EXTRN имя.тип

WORD

FAR (?)

NEAR (?)

PUBLIC имя (считать имя внешним)

В ЯП С:

**p.c**

extern int X;

**m.c**

int X; // по умолчанию эта переменная будет экспортированна в общее пр-во имен

разрешено стандартом С:

|  |
| --- |
| module\_1:  struct A { int x, y; };  void f (struct A \*pa) { pa->x = 0; pa->y = 0; … } |
| module\_2:  struct B {  int foo, bar;  } b;  extern void f (struct B\* pb);  f(&b);  ... |

C++, перегрузки, ф-ии члены, как их транслировать?

void f(int i);

void f(double d);

class X {

void f(int i);

}

^----------------- X\_f(X \*const this, int i) { … }

Страуструп изобрел схему кодирования имен.  
Это же самое на С++ не будет работать:



Modula 2:

|  |
| --- |
| // здесь могут быть только объявления  DEFINITION MODULE M;  TYPE T = …;  VAR X:T;  PROCEDURE FOO (VAR BAR:T);  END M; |
| // все переменные, объявленные здесь - внутренние  IMPLEMENTATION MODULE M;  VAR PROCEDURE FOO (VAR BAR:T);  BEGIN  BAR p:= …  END FOO;  END M; |
| IMPORT M;  VAR MY:M:T; |
| или:  FROM M IMPORT T;  VAR MY:T; |

В Обероне вместо DEFINITION и IMPLEMENTATION пишется один модуль, а экспортируемые переменные помечаются ‘\*’

В С/С++:

#ifndef (\_\_M\_H\_\_)

#define \_\_M\_H\_\_

....

#endif

Далее поток сознания:

stdafx.h ⇔ stdafx.cpp

экспорт(модули)  
импорт

# Лекция 24

## Видимости и все такое

### Односторонняя связь:

Так - в Oberon, M-2, Delphi

B C/C++ ⇒ M.h; include - аналог импортирующей конструкции



Видимость: потенциальная, непосредственная (через операцию уточнения (квалификаторы (**Квалификатор типа** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *type qualifier*) — одно из [зарезервированных слов](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D1%80%D0%B5%D0%B7%D0%B5%D1%80%D0%B2%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2%D0%BE) const, volatile или restrict в [языках программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) семейства [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)). Мб под квалификатором видимости подразумевается private/public/…? ))

В Delphi, C/C++ имена в глобальном пространстве имен видны непосредственно и иногда это очень плохо

Delphi:

unit M:

interface:

uses M1;

implementation:

…

В Оберон, М-2 видимость потенциальная, после import’а.

### Общая схема:

Экспорт - фактически, broadcast



Импорт - точечный ⇒ связь - односторонняя

#### Ada:

package M1 is

… объявления

…

end M1

^------ единица компиляции

package body M1 is

…

implementation

…

end M1

^------ тоже единица компиляции



with P;

package M2 is

… PT (раздельная трансляция?)

end M2;

package M2 is

…

package M21 is

…

end M21;

end M2;

package body M2 is

…

package body M21 is

…

end M21;

end M2;

**separate:**

**в одной единице компиляции:**

package body M2 is

…

package body M21 is separate // но здесь подсоединяются не только интерфейсная // часть М2, но и body из другой единицы

//компиляции (см. далее)

end M2

**в другой единице компиляции:**

separate (M2)

package body M21 is

…

end M21;

Рефлексия:

C# → IL (Intermediate language) → {JIT - компиляция} → бинарный код

Common **Intermediate Language** (сокращённо CIL) — «высокоуровневый ассемблер» виртуальной машины .NET. Промежуточный язык, разработанный фирмой «Microsoft» для платформы .NET Framework.

Есть библиотека, предоставляющая возможность компиляции “на лету”.

Java, C# - механизм модулей “выкинули на помойку”.

В одном модуле только один класс ⇒ много мелких модулей.

### Импорт в Java

Java: (файл) модуль ←→ класс  
**package** - пространство имен, содержит классы внутри:

package имя;

class …

**import:**

import package\_name;

но это делать необязательно

package M;

import java.utils.\* или  
import java.utils.comparator

(вместо import’a можно просто использовать java.utils.comparator)

static import package\_name.class\_name; → все статические члены класса становятся видимы непосредственно:

static import java.Math;

exp(x); // из java.Math

т.е. import в java - для удобства обращения к именам.

# Лекция 25

## Средства инкапсуляции

то, что делает ЯП объектно-ориентированным.

“Упрятывание” деталей реализации.

АТД - абстрактный тип данных.  
Это почти просто тип данных с операциями

ТД = множество значений + множество операций

АТД = множество операций (а множество значений - это уже детали)

Задавая множество операций, можно задать, например, алгебру

Упрятывание - средство статических ЯП. В динамических ЯП упрятывать нечего.  
АТД есть даже в С: например, FILE \*

Методы:

[ Интерфейс

[ Реализация ← все, что здесь не видно извне.

### Разделение интерфейса и реализациии в Ада/Модула2

|  |  |
| --- | --- |
| Ada | Modula-2 |
| package P is  …  end P; | DEFINITION MODULE P;  TYPE T; //opaque, скрытый ТД (???)  END P.  // P.DEF |
| package P body is  …  end P; | IMPLEMENTATION MODULE P;  ...  END P.  // P.MOD |

P.DEF:

TYPE T;

В этом случае Т должен быть либо указателем: либо целочисленным типом, размером с указатель.

IMPLEMENTATION:

… REC …

TYPE T = POINTER TO REC;

Ada:

type T is private; (в интерфейсе)

WITH P;

package Client is

P.T;

…

end Client;

Сделаем так: пусть компилятор знает, как устроен тип, но нам он не будет доступен.

package P is

type T is private

набор операций над типом Т…

private

type T is record

TOP: INTEGER := 0;

...

end record

end P;

Что язык позволяет делать с АТД:  
:=, =, /=

type T is limited private;

теперь :=, =, /= автоматически работать не будут

DISTANCE

TIME

SPEED

…

package P is

type DISTANCE is separate;

function “+” (T1, T2: DISTANCE) return …;

function “/” (T1: DISTANCE, T2:TIME) return SPEED;

…

end P;

Наследование:

T ====================⇒ S

^--BASE, SUPERCLASSS (тип, суперкласс) ^---------- SUBLCASS (класс, подтип)

В чем проблема наследования при инкапсуляции?

public/private.

NS ::

m.cs ⇒ m.obj

в C# namespaces не имеет ничего общего со сборками (как элементами дистрибуции)

### в Java есть пакеты

вложенные пакеты в Java существуют подкаталогом (но это может быть нарушено)

project/subp1

/subp2

⇒

project.subp1

project.subp2

class T {

члены класса:

* данные (члены структур данных)
* операции (методы)

}

public/private?

В модулях было просто:

MODULE M;

TYPE T\* = RECORD // экспортируется

X:INTEGER

Y\*: INTEGER

END

Z\*:REAL

PROCEDURE OP\* (VAR X:T)

PROCEDURE Helper(X:T)

end M.

### Расширение типа:

в модуле M:

TYPE T1 = RECORD(T)

…

END;

вне модуля:

TYPE ST = RECORD (M.T)

....

END:

здесь нет **protected**

В событийном программировании без этого не обойтись.

class EventEmitter {

protected:

virtual void E1Handler(...);

};

class MyEmitter : public EventEmitter {

protected:

virtual void E1Handler(...) {... }

};

### Приватные виртуальные функции в С++

В ЯП С++ можно переопределять виртуальные приватные функции (но это уже нарушение инкапсуляции).  
В C# - нельзя. Там private function не может быть virtual

В Java:

class EventEmitter {

....

private void foo () {...}

};

class MyEventEmitter extends EventEmitter {

public void foo() {...};

}

EventEmitter x;

x = new MyEventEmitter;

x.foo() // ошибка

# 

# Лекция 26

x : f() Y ⇐X f()

X x = new Y() new X()

X.f() // Y.f() X.f()

? what ?

## Мультиметоды:

T1 T2

r @s.f() // ?

Геометрические фигуры:  
Draw()

Intersect()

s1.Intersect(s2) ← это плохо тк при добавлении новой фигуры эту функцию придется дописывать

B Ada:  
procedure P(X1:T1 ‘class; X2:T2 ‘class)

P(a,b) - выбор нужного метода

Это - **мультиметод**

CLOS - Common Lisp with Object System - там есть мультиметоды

В библиотеках С++ мультиметоды моделируются шаблонами (Статически)

\_\_\_\_\_\_\_\_\_\_

## Модификаторы доступа

### Оберон

В оберон только public/private

### Ада 95

B Ada 95 -- public / protected

package P is

type T is tagged private;

… operations …

private

type T is record … end;

end P;

**В клиентском модуле:**

with P;

package Client is

type ST is

new T with tagged

record … end;

private

private

type ST is …. end record;

end Client;

Child package:

package P.Client is

--- || ---

теперь private-часть Р доступна в private-части пакета Client

т.е. private в P играет роль protected.

### C++

private, public, protected (часто бывают ситуации, когда классы как-то связаны между собой)

#### Друзья

friend прототип функции или функции-члена:

friend void foo(int)

friend void Y::var();

friend class Z;

### C#

(программа состоит из определений классов и интерфейсов)

public (доступно всем, для класс - означает экспорт из пакета)

private (закрыто для всех кроме функций-членов, как в С++)

internal (доступно из той же сборки)

protected (как в С++)

### Java

public

private

protected (!!! для функций членов - доступ всем из данного пакета)

пакетный доступ (если нет ключевого слова)

class X {

protected virtual void f() {...}

}

class Y:X {

protected override void f() {...}

}

class Z:X {

protected override void f();

void foo() {

Y y = new Y();

y.f(); // в C#, C++ доступно

}

В Java так нельзя (можно через ссылку на себя - ?)

\_\_\_\_

Принцип подстановки Лисков  
(T) \any P(T)

⇒

(ST)

Проблемы с множественным наследованием:

A B

f() \ / g()

X

[A f() -- виртуальная

[В g() -- виртуальная

[X - наследник

X x;

x.f(); // this→f ()

x.g() this → х со смещением

A \*pa;

B\* pb;

X x;

pa = &x; pa→ f();

pb = &x; pb→g();

Здесь компилятору нужны две таблицы виртуальных функций:



Виртуальное наследование:



class X: public virtual A { int i; void f() {X::i = 0;} }

class Y: public virtual A { int i; void g() {Y::i = 0;} }

interface IX {

void foo();

int i {get;}

int bar();

}

class X: IX (implements в Java) { //наследование в Java - extends

public void foo() {...}

public int bar () {...}

...

}

Если Х не реализует одну из функций интерфейса, он абстрактный (abstract class X)

X x = new X();

IX y = x;



# Лекция 27

## Обработка ошибок: исключения

throw

raise → в Unix посылка сигнала самому себе

Чем опасны исключения?

* Деструкторы: если они выбросят еще одно исключение

При выбрасывании исключения, стек "разворачивается", и все сконструированные объекты, находящиеся на стеке, разрушаются (вызываются их деструкторы). Соответственно, если из деструктора выбросится ещё одно исключение, то всё очень плохо (в плюсах в данном случае процесс аварийно завершается — abort()).

* Многопоточность

## Обобщенное программирование:

Влезать в само это понятие мы не будем

(**Обобщённое программирование**([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *generic programming*) — [парадигма программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), заключающаяся в таком описании данных и [алгоритмов](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), которое можно применять к различным [типам данных](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), не меняя само это описание. В том или ином виде поддерживается разными [языками программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F). )

Статическая параметризация (типов) ⇒ статическая типизация ЯП

С++ - templates,

C#, Java - generics;

В Java c 2005 - generics;

B C++ классы и функции являются параметризируемыми объектами

В Java классы и их методы (м.б. отдельно)

### Ада

В Ада еще в 1980-х

generic

type T is private

package Stack is

…

end Stack;

package body Stack is

//если нам нужен только размер Т, то отдельно откомпилировать (до всех случаев <...>) можно.

end Stack;

трансляция должна быть раздельной (Одно из требований при разработке Ады было раздельная трансляция)

package INT\_Stack is new

package Stack(INTEGER) // or T ⇒ INTEGER

end INT\_Stack;

можно сделать так же FLOAT\_STACK, STRING STACK, …

Но если теперь нам нужно написать общую функцию sort?

generic

type Arr is array(INDEX) of T

type INDEX is range <>;

type T is private;

function “<”(x,y: T) return ...

boolean is (<>):

4 параметра у нас получилось.

### С++

template <class (or typename) T> class Stack {...};

Stack<int> -- конкретизация шаблона

template <typename T, int size>

class Stack {

private:

T[size] \_body;

public:

Stack(); …

}

S.push(1);

Проблема конкретизации при раздельной трансляции:

template <typename T>

void sort (T\* art, int l, int r)

{

…

if (a[i] < a[j])

std::swap(a[i], a[j]);

…

}

Компилятор в точке конкретизации должен иметь тело функции

Class C {...};

C arr[N];

sort (arr, 0, N-1); //компилятор здесь должен проверить, можно ли конкретизировать Т=С, т.е. определены ли операции “<”, swap(), … )

Создателям C#, Java это не понравилось

### Родовые абстракции

Абстракция ⇒ обобщение (? в лекциях было “об-е”) шаблона

Конкретизация

#### Специализация шаблона:

* Полная - получаем конкретный класс
* Частичная - получаем новый шаблон

template <class T>

class Container {

T x, y;

void sort() { … x<y … }

};

T = const char \*, x < y будет работать не так как мы хотим.

Container<const char \*> c;

#### Полная (явная) специализация:

template <> class Container <const char \*> { sort() {... strcmp(x,y)<0 ...)} };

Это похоже на наследование, но здесь мы можем полностью поменять интерфейс

Container <int> ci; // общая реализация

Container <const char \*> cs; // наша отдельная реализация

#### Частичная специализация

template <typename T, comparer P> class Container <T\*>

{

void sort(Comparer P)

}

container <char\*, std::comparer> cs;

container <int\*, …> ci;

По умолчанию:

template <typename T, class comparer = std::less>; …

это позволяет компилятору выбирать эффективный код

SFINAE

ошибочная подстановка не является ошибкой

Т.е. если мы подставили в шаблон типы и получили ошибку, мы не завершаем компиляцию с ошибкой, а пробуем следующий вариант шаблона.

Частичная специализация для получения информации о типе.

template <class T>

class Stack {

public:

~Stack() { clean(IsPtr<T>::Result();) }

private:

void clean (No) { }

void clean (Yes) { for… delete …}

}

unique\_ptr

#### move-семантика

TT& operator = (TT&& t) {

char \* p = t.\_p;

t.\_p = \_p;

\_p = p;

cout << "Hello, " << \_p << endl;

return \*this;

}

IsPtr<T> :: Result либо

typedef struct{} Yes;

либо

typedef struct{} No;

template <typename T>

struct IsPtr {

typedef Result No; // using Result = No;

enum {r = 0;} ;

};

template <typename T>

struct IsPtr<T\*> {

typedef Result Yes;

enum {r = 1;} ;

};

template <typename T>

struct IsArray {

typedef Result No; // using Result = No;

enum {r = 0;} ;

};

template <typename T, int n>

struct IsArray<T[n]> {

typedef Result Yes;

enum {r = 1;} ;

enum {size = n};

Etype = T; (???)

};

# Лекция 28

## Программирование на шаблонах

Задача:

описать шаблон struct Fib так чтобы

const int n=10;

cout << Fib <n> :: result; выдавалось n-e число Фибонначи

template <int n> struct Fib {

enum {result = Fib<n-1>::result + Fib<n-2>::result };

}; //общий случай

template <> struct Fib <0> {

enum {result=1};

};

template <> struct Fib <1> {

enum {result=1};

};

С факториалом аналогично.

пример шаблона:

std :: tuple <int, int, std::string> t(1,2, “tuple”);

std::get <0> (t) ⇒ 1

std::get <1> (t) ⇒ 2

std::get <2> (t) ⇒ “tuple”

### Const expression (C++11)

constexpr int a[] = {1,2,...,24}; // constexpr ← то, что может входить в константное выражение

constexpr int f(int i) {return i+1;}

const int n = f(5);

template <typename T, int n>

struct Sum {

enum {result =...}; // Дз - реализовать это.

};

constexpr int sum(int a[], int n) {

if (n==1) return a[n-1];

else return sum(a,n-1) + a[n-1];

}

cout << sum(a, sizeof(a)/sizeof(a[0]));

if constexpr

//условие проверяется во время компиляции, и false-ветки вообще не рассматриваются компилятором

…

~Stack() {

if constexpr (IsPtr<T>::r)

{

/\* текс clean (Yes) ...\*/

}

}

typedef char True; // sizeof == 1

typedef struct {char a[2];} False; // sizeof > 1

False IsPtr(...);

template <typename T>

True IsPtr(T\*);

#define is\_ptr(e) (sizeof(IsPtr(e))==1)

### Вариадические шаблоны

Вариадические (variadic) шаблоны - с переменным числом аргументов

template <typename ...Args> [class/function];

...Args - argument pack, его можно раскрыть: Args…

example:

template<class ... Types> struct Tuple {};  
Tuple<> t0; // Types contains no arguments  
Tuple<int> t1; // Types contains one argument: int  
Tuple<int, float> t2; // Types contains two arguments: int and float  
Tuple<0> error; // error: 0 is not a type

pack expansion:

template<class ...Us> void f(Us... pargs) {}  
template<class ...Ts> void g(Ts... args) {  
 f(&args...); // “&args...” is a pack expansion  
 // “&args” is its pattern  
}

g(1, 0.2, "a"); // Ts... args expand to int E1, double E2, const char\* E3  
 // &args... expands to &E1, &E2, &E3  
 // Us... pargs expand to int\* E1, double\* E2, const char\*\* E3

функция суммирующая все свои аргументы:

template <int A, int … Args> int Sum() {

return A + sum<Args …> ();

}

int Sum() {return 0;}

Sum <1,2,3> (); //out 6

Sum <1,2,3>; //out 1

печать аргументов (параметров):

void print() { cout << endl; }

template <typename Arg, … Args>

void print (const Arg& a, const Args& … Args) { cout << a<< endl; print (Args…); }

print (1,2.0, “line”);

out:

1

2.0

line

-std=c++14 (ключ компиляции, можно 11, 17, etc)

C++17:

### fold expressions (выражения - свертки)

(init op … op пачка)

(пачка ор … ор init ) //init - изначальное значение

(пачка ор …)

(... ор пачка)

Example:

**auto** **SumCpp11**(){  
 **return** 0;  
}  
  
**template**<**typename** T1, **typename**... T>  
**auto** **SumCpp11**(T1 s, T... ts){  
 **return** s + SumCpp11(ts...);  
}

And with C++17 we can write much simpler code:  
**template**<**typename** ...Args> **auto** **sum**(Args ...args)   
{   
 **return** (args + ... + 0);   
}  
  
// or even:  
**template**<**typename** ...Args> **auto** **sum2**(Args ...args)   
{   
 **return** (args + ...);  
}

пример Головина:

template <typename … Args>

void print(const Args& … args)

{

string sep (“\n”);

((cout<<args<<sep), …);

} ^--------- унарная свертка относительно “,”

### Обобщенные функторы

std::function < R(Arg1, .., Arg B) >

int (bool, double)

Раньше:

int foo(int);

class X {

public:

int f(int);

static int g(int);

int (X::\*pm) (int);

};

int (\*pf)(int) = X::g;

fg = foo;

X a;

a.\*pm();

Теперь:

std::function<int(int)> func;

func = //теперь можно присваивать все что угодно - и функторы, и указатель на функцию-член…

#### std::bind

int add (int a, int b) { return a+b;}

std::function <int()> /\* или auto \*/ func = std::bind(add,1,9);

cout << func(); //10

int sub (int a, int b) { return a-b;}

auto func = std::bind(sub,9,-1);

<int(int)>

func(2) ⇒ 7

bind (sub, \_2,\_1)(9,1) ⇒ -8 //\_1, \_2 лежат в std::placeholders

<int(int,int)>

struct adder {

int val = 0;

int add(int i, int j) { return val = i+j; }

};

adder myadd;

std::bind (adder:add, myadd, \_1, \_2)(5, 6); //11

cout << myadd << val; //0

тк myadd передавался в bind по значению

template <typename T>

T& ref (T&& t) { return t;}

bind (&adder::add, std::ref(myadd), \_1, \_2)(5,6)

cout << myadd << val; //11

struct Functor() {

int val = 0;

int operator() (int i, int j) { return val = i+j; }

};

Functor f;

bind(&f, 5, -1)

# 

# Лекция 29

## Обобщенное программирование: C# и Java

Вариация обобщенных классов

### C#:

class Generic <T> {...}

var x = new Generic<int> ();

var y = new Generic<String>();

class Generic <T> {

List<T> body = new List<T>();

T Pop() {...}

void Push(T x) {...}

}

var x = new Generic<int> ();

x.Push(1);

int k = x.Pop();

var y = new Generic<String>();

y.Push(”line”);

string s = y.Pop();

x.Push(“Foo”); // compile-time error

int i = y.Pop(); // compile-time error

Что нужно знать о типе Т компилятору при создании Generic?

class Generic <T> {

const int maxsize = 128; // статическая константа

T[] body = new T[maxsize];

…  
}

в С# для описания константного объекта инициализирующегося в runtime используется слово readonly

обобщенными могут быть делегаты.

delegate R Func<T,R> (T x);

Func<R> // делегат без параметров, возвращает тип R

Func <R, T>

System.Func // обобщенный делегат уже есть в библиотеке

System.Action // void функция

Когда происходит генерация кода?

В C#, Java - в момент выполнения (JIT, Just-In-Time) (равно как и любой другой код)

Родовые абстракции поставляются в оттранслированном виде (промеж. код)

class X <T, U, V>

where T: new T(int); // В C# означает, что T должен иметь конструктор от int

// Другие примеры: "where T : class", "where U : struct", "where T : IComparable"

Generic <Integer> v = new Generic<Integer>;

v.Push(1); //~ v.Push(new Integer(1)); ← автоупаковка

Integer i = v.Pop;

int j = i;

// в Java в качестве Т можно передавать только классы.

Что происходит во время вызова метода? (в данном случае - Push())

В родовой абстракции (откомпилированной) стирается тип( Something<T> компилируется в Something<Object>), заменяется на Integer и вызывается

Динамическая проверка типа

Generic g = new Generic();

g.Push(“line”); // runtime error

class Generic <T extends X> { //X = interface/class

можно использовать все операции и свойства X

}

Нельзя создавать массивы объектов обобщенного типа

class Generic < T> {

Generic( T[] of) { body = of;}

private T[] body;

}

Обобщенный метод:

<T,R> R Func(T x);

## Механизмы вариации обобщенных классов

### ковариантность и контравариантность

class Super {... }

class Sub : Super {... }

здесь показан не функционал, а что-то типа кол-ва объектов входящих в этот класс

Sub является ковариантным относительно Super

Super является контравариантным относительно Sub

Ковариантность: Super = Sub

F(T): тип F зависит от типа Т

в каких случаях?

class F<T> { … }

F ⇒ T[]

F ⇒ T\*

### Ковариантность

из A ⇒ подкласс В

следует, что

F(A) ⇒ подкласс F(B)

### Контрвариантность

из A ⇒ подкласс В

следует, что

F(B) ⇒ подкласс F(A)

### Инвариантность:

F(A) и F(B) не имеют отношения друг к другу независимо от того, как сочетаются А и В

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

С++

class B{...}

class A: public B{...}

B arrb [50];

A arra [50];

B \*pb = arrb;

A \*pa = arra;

pa = pb; // нельзя

pb = pa; // можно

⇒ в C++ классы ковариантны

C#

class B{...}

class A:B {...}; class C:B {...}

A[] arra = new A[10];

B[] arrb = new B[10];

arrb = arra; // можно

arra = arrb; // нельзя

arrb[0] = new C(); // массив из разных элементов?!

В runtime будет выброшено исключение, в Java - тоже

Массивы ковариантны в С++, C#, Java

^-----------^ ← ------- readonly !

F(A) - ковариантный =>

A \in B

F<A> x;

F<B> y;

y = x; //ok

x = y; //not ok!

F(B) - контрвариантный =>

A \in B

F<A> x;

F<B> y;

y = x; //not ok

x = y; //ok!

class Product {

int Price() {...}

…  
};

class Producer <T> where T: Product {

T produce() {...}

};

class Consumer <T> where T: Producer {

void consume (T x) {...}

}

class ProdA: Product {...}

class ProdB: Product {...}

Consumer <Product> c = new …;

Producer<Product> p = new …;

Producer<ProdA> pa = new …;

Consumer<ProdB> cb = new …;

c = cb; ?

cb = c; ?

p = pa; ?

pa = p; ?

p=pa;

p.Produce() ⇒ ProdA

Product P= p.Produce();

pa = a; // NOT OK

pa.Produce()

ProdA x = pa.Produce() ⇒ Product

но в С# эти классы инвариантны.

c = cb; // NOT OK

c.Consume (new Prod A); небезопасно

cb = c; // безопасно

здесь могла бы быть контрвариантность

Класс, в котором Т только возвращается - ковариантен

Класс, в котором Т только передается в качестве параметра - контрвариантен

B C# ко/контр вариантными могут быть интерфейсы:

interface IProducer <out T> { T Produce(); } //out -- interface ковариантен

Interface IConsumer <in T> { void Consume(T x); } // in -- interface контрвариантен

IProducer <Prod A> pA = new Producer <Prod A>

Producer <Product> p;

p = pA;

not pA=p;

аналогично для IConsumer:

cB = c;

not c = cB;

В C++ все классы инвариантны

Указатели - ковариантны.

В Java по-другому:

class ProdCons <T extends Product> {

T Produce();

void Consume(T x);

}

ProdCons <? extends Product // использ. как ковариантный класс //> p = new ProdCons<ProdA>();

p.Produce();

ProdCons <? super ProdB> c = new ProdCons<ProdB> //контрвариантность

c.Consume(new Product( ));

Generic<?> P

в 2х словах

C# in контрвариация, out ковариация

java extends ковариация, super контрвариация

-------------------------------------------------------------------------------------------------------------------

Дополнения от Сергея :

Шаблоны похожи на макросы, это действительно просто шаблон, по которому потом генерится код. С точки зрения языка у шаблона нет собственного смысла — это просто кусок токенов (или чего-то такого). Смысл появляется, когда этот шаблон инстанциируется, причём у каждой инстанциации смысл отдельный. Проверяется код, который в результате сгенерится (и каждая инстанциация отдельно). Если ты шаблон ни разу не использовал, можешь написать внутри любой бред — ошибки не будет. Собственно, если вы понимаете, как работают макросы в С, то шаблоны в этом отношении очень похожи.

А дженерики — это просто типы (или функции, или ещё что-то), обобщённые относительно каких-то параметров времени компиляции (обычно типов). Их объявление само по себе компилируется, ошибки появляются именно в нём — причём сразу, независимо от того, где и как эти классы/функции используются. Система типов о них знает и понимает.

//////////

<Про вариацию в Java>: ...наверно, про это нужно думать как-то так:

? extends X кастуется к X, но ничто не кастуется к нему

X кастуется к ? super X, а оно к Object

GenericType<X> кастуется и к GenericType<? extends X> (он read-only, потому что вы не сможете создать ничего типа ? extends X), и к GenericType<? super X> (он почти write-only, потому что вы можете вытащить только object)

GenericType<? extends Derived> кастуется к GenericType<? extends Base> (ковариация), GenericType<? super Base> кастуется к GenericType<? super Derived> (контравариация), GenericType<Base> и GenericType<Derived> между собой не кастуются (инвариация)