# 

[**Лекция 1 (11.09.18)**](#_ad17p0ra6swd) **2**

[Языки и парадигмы программирования](#_f3petcmgpkug) 2

[**Лекция 2 (18.09.18)**](#_wlx5jcp4noew) **3**

[Функциональное программирование](#_750t66jggqcj) 3

[Логическая парадигма](#_o47kgirduxs6) 5

[**Лекция 3 (25.09.18)**](#_kt64w1jq8054) **6**

[Основные понятия языков программирования](#_sh6jghd7omt4) 7

[Данные, операции](#_dldsgmdzcajx) 7

[Сlosures (замыкания):](#_jd3nkd6br9c2) 8

[**Лекция 4 (02.10.18)**](#_hgnlm2omy2r6) **8**

[Объектно-императивные языки](#_x2uy58ndg5ge) 9

[Арифметические типы данных:](#_t2ane3ft1312) 9

[**Лекция 5 (09.10.18)**](#_zetm3ivf1koe) **11**

[**Лекция 6 (16.10.2018)**](#_8329byxq98rv) **13**

[Строки](#_e6hzxr61k4qq) 13

[Перечисления](#_uvnvuxosra2q) 14

[**Лекция 7 (23.10.2018)**](#_x9nmghvwf0wo) **16**

[Указатели: (не то что ссылка)](#_6imb0k7iwblm) 16

[Ссылки](#_yv4bl36oilc9) 17

[Составные типы данных](#_d49kwmocg74q) 17

[**Лекция 8 (30.10.2018)**](#_ieppx69bg179) **18**

[Структурные базисные типы](#_7bgp3v1i15g3) 18

[Кортежи](#_66zpv8vjlyu9) 19

[Структуры](#_gvmywl320cgd) 20

[**Лекция 9 (06.11.2018)**](#_q514vgwrlcou) **20**

[Структуры](#_rsnngp7gj1rc) 20

[Словарь](#_lkxup6fk28s1) 23

[Операторный базис](#_aa99v17z0w2e) 23

[**Лекция 10 (13.11.18)**](#_8lkz014i4dm7) **24**

[Statements](#_n786kirylbfl) 24

[Операторы-оболочки](#_q9fdbmgnhrkl) 26

[Средства развития](#_tonk8x9g2sbo) 27

[**Лекция 11 (20.11.18)**](#_8y7gc3o1o17) **28**

[Сопрограммы](#_5bw5lrfsiw2y) 28

[Программы](#_romezz6qac9j) 32

[**Лекция 12 (27.11.18)**](#_4mcmrtl2ov2x) **32**

[**Лекция 13 (11.12.18)**](#_onoiw5owwerd) **34**

[Обобщенное программирование (generics)](#_gr9lylx7swoe) 34

[**Лекция 14 (18.12.18)**](#_6rkhvs2lcu8j) **40**

# Лекция 1 (11.09.18)

## Языки и парадигмы программирования

С – чистый императивный язык («Экологическая ниша языка»)

В самом первом варианте языка Си:

Type void()\*;

В Си из стека выкидывает caller, в Паскале called. в C#, Java нет автоматической свёртки стека, в них нет деструктора, есть финализатор, который вызывается в случайный момент – сборщик мусора.

Finally – для освобождения других ресурсов

RAD – Rapid Applications Development

Объектно-ориентированная парадигма

Объект:

* Состояние
* Поведение

Отлично уживаются с состояниями. Обработчик изменяется наследованием. VES (Virtual Execution System - is a run-time system of the Common Language Infrastructure CLI which provides an environment for executing managed code):

* CLR (Common Language Runtime – общеязыковая исполняющая среда) – Windows
* Mono – Linux

Реверс на языке C#:

public class Program{

public static void Main(String[] args){

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

Char[] arr = s.ToCharArray();

Array.Reverse(arr); => for (int i=a.Length-1; i>=0; i--)

System.Console.Write(arr); => System.Console.Write(s[i]);

}

}

В С# и Java нет глобальных переменных.

В С# есть статический класс – контейнеры.



package main

import (“fmt”, ”ioutils” “os”, “strings”)

func main(){

b, err := ioutils.readAll(os.Stdin) ← строка – последовательность байтов

Исключение плохих при многопоточности

if err!=nil {

panic(“bad input”)

}

s:= string(b)

x:= strings.split(s, “”)

for i:=len(x)-1; i>=0; i—{

fmt.Print(x[i])

}

}

На Python:

import sys

raw = sys.stdin.read()

l = [c for c in raw]

l.reverse()

print(“”.join(l))

# Лекция 2 (18.09.18)

С++ - единственный язык, который наиболее обще может выразить reverse строки. STL контейнеры, алгоритмы (не знают друг о друге, посредник – итератор)

container(*list*) c;

std::copy(it1,it2, back\_insert(c));

std::copy(istream\_iterator<char>(cin), istream\_iterator(), back\_insert(c));

std::copy(c.rbegin(), c.rend(), std::osteam\_iterator<char>(count))

В С++ практически все компиляторы поддерживают предкомпиляторные header’ы, чтобы ускорить компиляцию.

## Функциональное программирование

Lisp

Базис(средства) – типы, операции, нет операндов

Тип данных(атом):

* Символы (+ foo bar\*)
* Число (-25, 648)

S – выражение (точечная пара: a.b)



Список:

() – это то же самое, что и nil

a.b.c.nil → (a b c) 





(car s) – левое поддерево (в общем случае список не даёт)

(cdr s) – правое поддерево (всегда список)

Итак атом

(cons a b) символ => значение число => число

(car s) REPL пустой список => пустой список

(cdr s) > x (← символ, значение)\*

С символом может быть связана ф-ция – форма

(+ 1 2) => 3 (1, 2 – аргументы соотв ф-ции)

(if B S S1)

B - условие

S - выражение

S1 – else-выражение

(cond Моделирование if cond’ом (cond

(B1 S1) =========> (B S)

(B2 S2) (T S1)

… )

)

(null s) – пустой список?

(atom a) – явл-ся ли атомом?

(eval s) – выполнить s

(defun символ список\_арг список\_тела)

Пример:

(defun Plus1(x) (+ x 1))

(Plus1 4) => (5)

(lambda(x) (+ x 1))

((lambda(x) (+ x 1)) 5) => (6)

(let (x 5) s) - x=5 – const для s

(set x 5) - вычислить x

(quote s) - не вычислять s

(set (quote x) 5)

(setq s (quote (a b c)))

(append S1 S2)

S1: S2:



Reverse на Lisp

(print (reverse (read())))

(a) = > (a); (abc) => (abc); (a b c) => (c b a)

(defun reverse1(s)

(if (null s)

()

(append (revrse1(cdr s)) (cons (cdr s) nil))

)

)

S1 → S2 - Элементы не копируются, а перебрасывается указатель

STL + компилятор

Некоторые компиляторы не могут скомпилировать STL, поэтому они писали свои. В Swift, ObjectiveC два типа строк: изменяемые и неизменяемые.

(defun shift(l r)

(if (null l) (r)

(append ((cons (car l) nil) shift(cdr l)))

)

)

(defun reverse2 (s) (shift s c)))

(print (reverse2 (read())))

REFAL (не совсем функциональный язык)

s.i – символ o = - любой паттерн

expr1 = expr2

patt11 = patt21

patt12 = patt22

$ENTRGO

{

<Pront<reverse<card>>>

}

Reverse

{

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

=;

}

Палиндром

s.1 e.1 s.1 = <Palindrom e.1>

s.1 = s.2

=;

## Логическая парадигма

Prolog

* факты
* формулы

Факты:

MAN(SOKRAT)

MAN(X) -: MORTAL(X).

-: MORTAL(SOKRAT)

[a b c]

[x | l]

x=[a], l=[b, c]

x=[a,b], l=[c]

x=[a,b,c], l=[]

append(l1, l2, l3) ← l3 - конкатенация l1 и l2 => true

-: append([a], [b,c], [a,b,c]) => true

-: append(X, [b,c], [a,b,c]) X=[a]

-: append([x|l], [b,c], [a,b,c]) X=a(не пусто), l=[]

reverse(l1, l2) reverse([1,2], [2,1]) => true

reverse([], [])

reverse([X, Q], Z) := append(Y, X, Z), reverse(Y, Q)

Можно сделать с shift

Для Паскаля существует декларативное преобразование операций

{In(x)} Ta1…an

Prog

a1<= a2<= … <= an

Out(x)

Из пролога появилась идея DCG(definite clause grammar):

A -> B C D ← предикаты

A(x) -> B(x) C(x) D(x)

С одной стороны – КС-грамматика, с дрогой – программа на Prolog

Модели – слишком тяжелы для вычислений.\

# Лекция 3 (25.09.18)

Lisp:

(defun shift(l r)

(if (null l) r)

(shift (cdr l) (append (cons (car l) nil) r))

))

(defun reverse1(x)

(shift x ())

)

Prolog:

reverse1(X, Y) :- shift(X, [], Y)

main :- reverse1([1 2 3 4], Z)

shift([], X, X).

shift([H|T], R, Y) :- append([H], R, Q), shift(T, Q, Y).

Кстати:

f(x)ifelse

/ \ ← хвостовая рекурсия

result f(val)

В Prolog порядок по теории\* не должен иметь значения, но он имеет из-за 10\*(io) и cnt (cnt используется во многих реализациях на Prolog). Поэтому пролог не взлетел.

Modula2:

ADDRESS

POINTER TOT;

ALLOCATE

(VAR A: ADDRESS,

N: CARDINAL)

## Основные понятия языков программирования

Данные, операции

/ \

тип данных значение

Данные и операции – дуализм(атрибуты). Очень важны в современных языках (\*\*\*)

string => len

c => stren-операция

с++ => str.size()-операция

pascal => len(s)

Op:Dn\*…\*Dn -> D

Stack:

Pop Push(s, x)

Push &

isEmpty Pop(s, r)

r ≡ x

ATD - Abstract Data Type

Свойства (property)

C#:

class X{

private string\_name; 

public string\_names;

set {\_name = value;}

get{return Name;}

}

Кстати, объекты динамической памяти имени не имеют. lambda-функция тоже. Языки – референциальные, объекты создаются динамически.

int[] a;

a = new int [100];

int [][]a;

a = new int [][20];

for (int i=0; i<20; ++i)

a[i] = new int [i];

## Данные, операции

«Универсальные» атрибуты

* значение
  + сигнатура(имя) + тело
* имя ←→ объект данных, ф-ция
  + область видимости(scope),
  + иерархичны
  + область имени
  + определяющие вхождения
  + использующие вхождения

catch (exctype \_ )

exctype – определяющее вхождение(динамическое связывание)

динамический поиск – поиск имени по стеку

статический поиск – поиск определения

Ada → package D

T

Function “+” (x, y:T) return T; (\*\*\*)

C++ → namespace D(T)

operator+(….)

“+”(a, b) ~ a+b

C++: namespace std{

osteam

ostream& operator<<(ostream a)

#include <iostream>

std::cout<<”Hello!”;

operator<<(std::cout<<”hello”; … ←

namespace X{

class T;

void f(\*T);

}

X:T a;

f(a) ← свяжется\* здесь в классе

Область действия (extent) ←→ объекты

⇑

⇓

время жизни

У динамических объектов область действия – вся программа.

## Сlosures (замыкания):

В Java есть аналогичные классы(локальные).

База lambda в С++ область действия может выходить за область действия блока.

Данные, операции, связывание(время связывания) (binding)

Статическое связывание – во время компиляции, загрузка в память, компоновка

Динамическое связывание – во время выполнения

/ \

квазистатическое чисто динамическое

(связывается внутри блока)

Статический тип: Haskell Слабо тип: JavaScript

Динамический тип: Lisp Строго тип: Python

# Лекция 4 (02.10.18)

«Динамические» ЯП

«Комп»/ «Интерпретатор»

Отличие в связывании: статическое, квазистатическое(в обоих), динамическое

Объект данных ⇔ тип данных

CGI(Common Gateway Interface - стандарт интерфейса, используемого для связи внешней программы с веб-сервером) помог динамическим языкам.

## Объектно-императивные языки

Базис

Ср-ва развития

Ср-ва защиты

Базисные типы данных:

* скалярные(примитивные) – с точки зрения языка нет структуры
  1. арифметические
  2. логические
  3. символьные
  4. указатель, ссылки
  5. перечислимые типы
  6. диапазоны
  7. datetime («экзотип»)
  8. currency («экзотип»)
* структурные

Transpiling - преобразование в JS, TypeScript, CoffeeScript. При транспайлинге получаются огромные объекты кода + библиотеки. (Вернулись к старой схеме компиляции???)

Факт: в Паскале 69ого года все типы 1-6) были

Почему в современных языках исчезли диапазоны?

- В современных языках индексация с нуля и тип индекса integer.

## Арифметические типы данных:

1. вещественные
   * с плавающей точкой (универсальные)
   * с фиксированной точкой (есть в Ada)

dettaH\*(rangel\_R)

Когда считывается сигнал не имеет смысла мерить с произвольной точностью

Кстати: в С# есть decimal (удобно выводить, неудобно считывать)

Фиксированная точность нужна в валютных операциях

1. целочисленные(integer)
   * универсальность
   * эффективность
   * «переносимость» с одной архитектуры на другую
   * надёжность (при смене знака обязательно контроль переполнения)

Можно ли смешивать знаковые и беззнаковые?

Ошибка:

while((i=getchar())!= EOF) {}

- Универсальность – если тип появился в машине, значит он для чего-то нужен.

- Эффективность – в реализации задачи.

JavaScript – переносимость, Number(на основе double)

Python: integer – переносимость. Надёжность

Java: переносимость, ±универсальность, надёжность (остаётся только byte из беззнаковых)

С++: нужен конструктор преобразования для универсальных.

Страуструп хотел запретить символьные signed и unsigned.

C: эффективность

Формат BMP: C99:

DWORD size stdint.h

LONG width <cstdint>

LONG height int\_8\_t ← Для обеспечения переносимости

WORD unint\_32\_t …



int\_list\_xx\_t

unint\_list\_xx\_t ← Для обеспечения эффективности

int\_fast\_xx\_t

Наплевать на надёжность

С#: char sbyte8 sbyte

short short16  unshort

int int32  uint ← Это система типов CLR

long long64 unlong

int16 uint16

int8 uint8

Go: int88

uint816

Swift: int8

uint8

uintptr

intptr

При арифметической переполненности:

* Swift – ошибка
* Go – нет ошибки
* C# - блок проверки:

checked{

i+j

}

В С:

LP32 ILP32 ILP64 **LP64** LLP64

char 8 8 8 8 8

short 16 16 16 16 16

int 16 32 64где32? 32 32

long 32 32 64 64 32

pointer 32 32 64 64 64

IA-32 IA-64 X86-64



Переносимость – int8 – сфикс.

Эффективность – int, long, …

Если нет беззнаковых, то что делать со сдвигом?

>>

В Java >>> - беззнаковый сдвиг

# Лекция 5 (09.10.18)

UTF32 – самая простая кодировка(неэкономная)

UTF-7, UTF-1 – были одними из самых первых(кодировки с переменной длинной) ← экзотика

На практике UTF32/UTF16/UTF8

i18n\* = internationalization

1. Локализация
2. глобализованные приложения - способные работать на разных языках

UTF16 ← не прямого доступа, избыточная

C U+D800 – приватный сегмент

V+D800 V+DFFF



code point ≤ 216

0 ≤ code point = 216 < 220 U+D800 – U+DBFF

U+DC00 – V+DFFF

UTF-8 (с переменной длиной, избыточна, не прямого доступа)

Разрабатывалась для Plan9

code point UTF-8



0-127

кириллица

128-2047 ← сюда влезает



2048-65535

UTF-8 выигрывает у UTF-16, так как пробелы, запятые => 1 байт

utf8everywhere.org

Microsoft Unicode = UTF16

Достижение UTF8 strcmp работает, работ. и другие

Функции\*(но не все – is\_alpha\*)



В UTF32, UTF8 есть проблема byte-order 

Big-endian

=> придумал BOM 0xFEFF

Little endian

В С++ wchar\_t Unicode 16

Работа в С, С++ с Unicode очень сложная

Java char <-> UTF16

character <-> codepoint

e’ <= U+00E9

я’

o’

глиф - то, как выглядит

Графичный кластер\* - одно знакоместо

Символ. – модификатор U+B01 =>

“cafe U+301” len=5 0x301=U+0301 - это знак надстрочный вот такой **́**

café ??? (тут написано что за)

“caf U+E9” len=4 А U+00E9=0xe9 - это é (целиковый знак, а не склейка)

Суть предыдущих строк - у них é в первом случае - склейка двух знаков, в третьем - целиковый знак, поэтому длина слова café - 5 в первом случае и 4 в последнем, а длина слова cafe - 4

Строки разные, но выглядят по-разному => сущ-ет нормализация

UTF16 – худший вариант

Golang:

s string rune = int32

s = “lime”

s[0] – число

s = “я”

“unicode/utf8”

for:. runeval := range

fmt.Print(“позиция руниг %u = %d”, runeval, i);

val,width8, DecodeRuneInString(l)

Swift самый честный

# Лекция 6 (16.10.2018)

Осталось поговорить:

1. Строки и представление текста (character)
2. Перечислимые типы
3. Указатели

4-бесконечность - экзотипы

Структурные типы:

1. Массивы
2. Структуры
3. Кортежи
4. map

## Строки

В С/С++, Python2 -> legacy; char, <cstring>; strcpy, strcmp;

А Unicode? wchar\_t (двухбайтный символ)!!! |Unicode|=2^20+2^16-2^11 символов - факт

Для новых Unicode-строк - wcsxxx функции

В Microsoft сделали непереносимую библиотеку

str |

| tcsxxx, wtof, tctof

wcs |

В стандартах компиляторов кодировка исходного текста не специфицирована!

Ю тоже неудачно

fopen

wfopen(wchar\_t x)

**В UTF-8 BOM не нужен**

Факт:

0xFEFF (в сети)

[FE][FF] - big endian

[FF][FE] - little endian

Сильно переделанные языки из-за кодировок: python3, php5

Python3: UTF32, UTF8, UTF16 адаптация к тексту

В Python3 сначала выкинули codecs => file.read() читает в unicode, а на диске могло быть cp1251

Правильно - в C# - при открытии файла заставлять указывать кодировку (файл - поток байтов)

В Go тоже, что и в C#, только внутри все UTF8

Кстати:

2 байта

Go: “cafe{0x301}” len-6

“caf{0xE9}” len=5

2 байта

Swift: символ - это то, что отображается - у них один глиф

Тип данных character

“café”, “cafe{0x301}” - везде len=4

Кстати:

let s = “café”

s[0] - error! нет прямой индексации строк

Есть тип string.index

В строке есть s.startIndex

s[s.startIndex]

В Swift есть views:

s := “café”

s.utf8 - массив байтов utf8

s.utf16

s.unicodeScalars - utf32

## Перечисления

1969 - Паскаль

type CompassPass = (North, East, South, West)

ord(enum) => int

value(T, i) => константа из T

В Си:

enum CompassPass {}

enum CompassPass cp;

cp = -5; => перечисление есть быстрое задание констант !?!?

#define RED 0xFF0000 - просто с enum удобнее

1988 - Оберон

Язык был минимальным

Enum:

1. засорение пространства имен: имена неявно импортируются в область видимости
2. противоречат концепции расширения типов: задавая в базовом классе enum => в дочернем нельзя расширить

1995 - Java

Не было перечислимого типа до 2005 года

В C#:

enum CompassPass {}

Тип данных Enum

Ведь все есть объект

Enum::toString()

Enum::parse() (\*не уверена) -> в тип

В результате: CompassPass.North не засоряет область видимости

(но здесь и в плюсах теряем возможность использовать как флаги, open(“...”, 01)

C++:

enum class CompassPass {

North;

…

};

auto cp = CompassPass::North;

В Java:

enum полноценный класс специального виды

enum Planets {

Mercury(3000\_

Earth(6400)

Planets(int Diam)

private:

…

};

В Go:

const (

North:=iota,

East,

West,

South

)

В Swift:

enum CompassPass {

case North, <- какие-то значения. Какие? (\*)

}

enum CompassPass:[string] {

case smile = ‘😀’;

case dog = ‘(тут нарисован собак)’;

}

enum CompassPass:int {

case North = 0;

case East;

}

enum CompassPass:case iterable {

case North,East,South,Weat;

}

for x in CompassPass:Allcases

…

enum BarCode {

case upc (int,int,int,int)...

case qrcode(string)

}

var Barcode x = .upc(1,2,3,4)

var y = Barcode.qrcode(“12…”)

switch(y) {

case.upc(let numberSystem, let ProductManuf, let Product, let Sys);

Перечисление => запись с вариантной записью

# Лекция 7 (23.10.2018)

На прошлой лекции: перечисления

* ограничение области видимости |новый enum
* наличие “оберток” для конвертации типов |хороший

Факт: в Swift есть как обычный enum, так и экзотический

## **Указатели**: (не то что ссылка)

Указатель -> address

Языки, у которых нет указателей:

1. Динамические языки, тип данных в них заранее неизвестен (в таких языках реализована референциальная модель объекта): объекты/массивы/интерфейсы - их экземпляры представляются ссылками
2. Современные ООП языки: C#, Java, Swift - референциальная модель

Объявления в C++ и в таких языках - это по сущности разные объявления:

X a;

В C++ - это статич. смещения frame pointer в стеке (до C++11, там ввели variable length arrays, с ними еще нужен один регистр динамического смещения)

ФАКТ

В C++ указатель на член-данные-класса - это не настоящий указатель, это смещение через this.

Но, если член-данные - static, то это просто указатель

В референциальных моделях все объекты анонимны и создаются в динамической памяти

В динамических языках переменная - ссылка на что-то.

Указатели не нужны, так как все управление памятью берет на себя транслятор.

Сборщик мусора

1. пометить ненужные объекты - быстро
2. ужимание памяти - долго

=> из-за ужимания памяти работа с указателями в языках с GC сильно ограничена

=> системы реального времени не написаны на языках с GC

Факт: В Паскале указатели нужны только для описания рекурсивной структуры

Поэтому в Паскале сделать GC можно (язык чистый), а в Си/C++ сделать GC нельзя - нужен специальный указатель!

В C# есть unsafe код, где конструкция fixed(pointer) говорит GC не менять адрес указателя. Через unsafe можно вызывать стандартные функции языка C#.

В Java есть JavaNativeInterface (в C# есть похожая штука), с помощью него можно вызывать платформо-зависимые API => теряется смысл Java-WORA, зато можно повысить эффективность программы.

В Go нет референциальной модели!!!

В Go есть указатели, но нет ссылок. Для чего нужен указатель? Для передачи параметров по ссылке.

## Ссылки

Во многих языках есть ссылки и референциальная модель.

Но ссылок на простые типы может и не быть (Java)

В C# есть ref и out f(ref int\*) и f(out int\*)

Два типа ссылок:

1. Ссылки в языке C++ |различные
2. Ссылки в других языках |подходы

C++: ссылки нужны:

1. Для передачи параметров по ссылке
2. переопределение операторов присваивания и других

Ссылка - новое имя

T& a = b;

## Составные типы данных

1. массивы
2. структуры/записи
3. множества (в старых языках огр. по размеру)
4. файлы (в старых языках, в Паскале)
5. таблицы/словари
6. кортежи

Оффтоп: в JavaScript все есть словарь

obj[“foo”] и obj.foo - синонимы

^можно вычислить динамически

obj[0]=1 - преобразование в строку “0”

Массивы: (однородный массив ячеек)

Для массивов должна быть операция copy. Нужна операция индексации с возвращаемым значением D&. В массиве все идет подряд и однородно: [ ][ ][ ][ ][ ]

В современных языках индекс - это int

У массивов есть длина length, которая либо фиксирована, либо динамическая

# 

# Лекция 8 (30.10.2018)

## Структурные базисные типы

однородность (проблема в том, что современные архитектуры отходят от однородной памяти, адрес должен быть машинным словом)

Массив всегда характеризуется однородностью

С массивами проблема при обращении за границу массива:

1. Только в языке Си нет квазистатического контроля
2. В C++ в vector вышли из положения (т.к. проверки неэффективны): arr.at[i]

В массивах интересна длина: len(A) - статич? динамич?

**статическое распределение памяти** (эффективно, обращение к сегменту данных по константному адресу)

В Си - VLA массивы VLA расположены в стеке,

int n = atoi(argv[1]); они не динамичны!!!

char line[n];

Не совсем динамические.

Факт: в Pascal длина 0 свойство типа

type tarr = array[1,...N] of integer

type tarr2 = array[1...N] of integer

procedure P(var x:tarr) <- a вот с tarr2 работать нельзя

В Ада решили эту проблему, введя типы <-> подтипы

неограниченный тип (массивы) type TARR is array (index range<>) of T

Можно определить переменную TARR, только ее уточнив

Подтипы заменяемы!

Как в modula2 решили проблему

array of T

(\*\*\*)procedure scal(var a: array of real) :rcal; <- массив любой длины

(Есть функция HIGH(A) - длина массива)

var s:real; i:integer;

begin

s:=0.0;

for i:=0 to HIGH(A)-1 do

s:=s+A[i];

end;

RETURN s;

end scal;

В Обероне похоже:

type tarr = array N of real;

**динамическое (с 90-ых)**

В C#, Java вообще отказались от статического распределения памяти

Кстати, в C#, Java отсутствуют глобальные переменные, зачем C#, Java использовать сегмент данных? Ответ - static члены класса

byte[] b, b1;

b1=b;

byte[] b = new byte[N];

В Swift:

var Array<Int> = (1,2,3,5); <- массив тоже динамический. Но a=a1 копируются значения!

В Алгол68 хотели описать семантику, т.к. в Алгол60 уже был описан синтаксис. В Алгол68: w-грамматики. Программисты не поняли, что такое Алгол68. Алгол68 - принцип ортогональности

a:=b оператор имеет значение

Слишком сложный язык!

В C#, Java

int[]b = new int[k+1] - квазидинамический, размер не меняется

В Python:

l=[1,2,3,4] Чисто динамический массив

l.append(5)

del l[0]

Но в C#, Java

Object [] b = new Object[n] <- можно так

b[0] = 5;

JavaScript:

var a=[1,2,3,4]

a[1000] = “string”

В C#, Python, Java, JavaScript только в динамич.

Что для C#, Java странно, становится обидно

В Go инновации:

var a[10] int <- статическая память под массив

var pa \*[10] int <- в стеке

pa = new[10]int <- динамич

slice - окошко, через которое смотрим на массив

var sa []int

sa = a[2:4]

sa[0] = -1

…

s := make([]int,5)

## Кортежи

a,b = b,a

В большинстве языков tuple - понятие времени комп., в Python - отдельный тип.

Python - read-only объекты - те, которые имеют неизменяемый хэш. Кортежи могут служить ключами словаря

Go, Swift: кортежи используются для возврата значений из функции

C++11,14: ввели tuple

В С++17Ж

std:tuple<int,int> b;

int x,y;

auto &[x,y]=b;

auto [x,y] = b;

x = 1 модиф. b

В Swift: именованные кортежи, где значения имеют имена

## Структуры

Отличие записи от структуры: для любой записи . - чисто статическая операция

ref R.[имя\_поля]

В Pascal для записей есть оператор with, который экспортирует поля записи в область видимости.

В Java: понятия структуры нет (есть класс)

В Python: нет структур (есть кортеж)

В PHP:

a = Array()

a[0] = 1

a[“property”] = 1

Роль структуры играет массив

# Лекция 9 (06.11.2018)

## Структуры

В Java нет структур (потому что авторы задумывали его по принципу чемодана, наличие структур излишне, т.к. их обобщает класс)

Принципы создания языков:

1. Принцип сундучка (берем все, что может понадобиться - Ада, например)
2. Принцип чемоданчика (берем по минимуму) - здесь добавляем то, что необходимо, чтобы занять нишу. Оберон - ООП для системного программирования, Golang

Типы:

* Референц.
* Значения

C++ - единственный язык, в котором структуры - то же самое, что и класс

Если в языке есть понятие класс и класс является референциальным типом данных (не в C++), то структуры будут типом-значением.

X a; если X - класс => a - ссылка, если X - структура => a содержит все данные

Структуры сделаны для экономии.

class Point {

int x, y;

method1()

…

methodk()

}

Тогда много точек:

Point [] parr;

parr = new Point[1000];

[][][]...[][][][]

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

parr = new Point

C#:

Point[] parr;

parr = new Point[1000]

Новых объектов в цикле не будет

Часто структуры нельзя наследовать. В динамических языках нет структур, там их заменяют другие вещи. В динамическом языке типы не важны.

Фишка структур, вернувшаяся из прошлого:

Объединение как частный вид структур

Паскаль:

record(

постоянная часть

вариативная часть

end;

case[name:] тип of (\*)

value 1: (список полей1)

...

Напоминает наследование. Именно поэтому их нет в современных языках.

ООП нужно для GUI, для событий. Как на не ООП языке описать тип Event

type Event = record

time: time\_type;

case: EvType:EventType {

KBDEvent:

(up:boolean,

move: word;

)

MouseBtnEvent(...)

end

procedure EventHandler(var E:Event);

begin:

case EE.EvType of:

…

end;

Минус такого подхода: написание больших свитчей, при добавлении новых виджетов - добавление их свитч

Если case~~[name:]~~ тип of:

case boolean of:

true:(i:intepr)

false:(bits:packed array[0..47] of boolean;)

В Си - нет записей с вариантами

union tags {

T1 m1;

T2 m2;

…

TN mN;

}

В них нет дискриминанта

Экзотика Swift

Перечисление есть запись с вариантами

enum CompassValue {

case North;

case East;

}

enum BarCode {

case upc (int,int,int,int)

case qrcode(string)

}

var product\_code = Barcode.upc(8,21,32,62)

prodcode = Barcode.qrcode(“....”)

switch product {

case let.upc(numbersystem, num2, num3, num4):

print(“upc”)

case let.qrcode(prodcode)

print(...)

}

Нет наследования, все круто

Моделирование деревьев на enum Swift

indirect enum Expr {

case number(int)

case add(Expr,Expr)

case mult(Expr,Expr)

}

let two = Expr.number(2)

let five Expr.number(5)

let four …

let sum = Expr.add(five, four)

let prod = Expr.mult(sum,two)

func evaluate(e:Expr)=>Int{

switch(c) {

case let.number(value):

return value

case let.add(e,n):

return evaluate(e)+evaluate(n)

case let.mult(e,n):

…

}

}

Записи с вариантами - альтернатива наследованию

Вернемся к массивам

1. референц (C#, Java) = проблема прямоугольных массивов
2. значения (Go, Swift)

Недостаток референц - неэффективно по доступу

Сейчас в C# есть и массивы-значения

Кстати в Go есть слайсы, в Swift есть inout - модификатор, то компилятор сам выбирает тип передачи

## Словарь

В Golang есть map = референциальный тип (создается через make)

В Go словари типизированы

Императивное программирование: программирование с побочным эффектом (?)

Функция может менять глобальные переменные и формировать параметры

## Операторный базис

Оператор присваивания v:=e или v=e

Операторы инкр/декр a++, ++a

Оператор I/O

Составной оператор: (+=)

1. с составным оператором (Си, C++, Паскаль)
2. без составного оператора (Алгол68, Питон)

В алгол60 были проблемы такого рода if B if B2 else B3 - к чему относится else?

if B

S1

elseif B2

S2

elseif B3

end;end;end;

Вводятся elif, elseif...

# Лекция 10 (13.11.18)

## Statements

Побочный эффект

* Control-flow операторы
* Операторы-обертки

В Фортране 5 типов переходов

В Алголе goto и switch

Чем switch лучше if? В goto очень просто транслир. комп. (\*\*\*). Это очень эффективно.

В 1967 “Goto statements are considered harmful” (Дейкстра) - один вход, один выход

К 1967 году было доказано, что любую программу можно представить как:

s1;s2 и while B do S

Дейкстра в 1975 году выпустил книгу с хорошими идеями, как программировать

1969 Structured Programming: в третьей части книги рассказали про корутины

В 1969 появился Паскаль: в нем были while, if, case \_ end (записи с вариантами, моделирование наследования), repeat \_ until B, for i:=Init to Fin do S

Тогда же появился и Си: if B S1 else S2; while (B) S; do {S} while (B); for (e1; e2; e3) S;

В 1974 году Кнут заметил, что некоторые алгоритмы плохо вписываются в только control-flow операторы Си:



start:

prepare();

if (ready) {

process;

goto start;

}

prepare;

while (ready) {

process;

prepare;

}

Ввели break, continue, return (=> совсем нивелировали нужду в goto)

for (ij) {

prepare;

if (!ready) break;

process;

}

В Модула2 в 80-х:

* цикл с конечным числом шагов for
* loop  
   ...  
   exit <- только в нем можно использовать exit  
   end

goto в нем не было!

Возникает вопрос о чистке ресурсов!!!

(Вот зачем goto в Си)

(В С++ goto не нужен - есть RAII)

Факт: сборка мусора  
Отсутствует гарантия выполнения деструктора  
 try {

} finally { <- всегда есть в таких языках

S

}

Языки разделились на 2 класса: есть goto и его нет

Есть goto: Паскаль, С/C++, Ада, С#, Go

Нет: М-2/оберон, Java, JavaScript, Python, Swift

Поэтому уже с 90-х годов операторные части языков были похожи.

Разным мог быть switch

switch (e) {

case M1:

case M2:

...

}

switch e of

0\_13, 24, \_ : S

else

end

Сравнение объектов на равенство с референциальной семантикой:

В JavaScript: == равенство объектов, === равенство ссылок

В Go, Swift, Java остались метки:

L: while (B1) {

…

while (B2) {

break L;

}

}

После 90-х интересно поступили с циклом for.

Еще в Обероне сначала избавились от for (потом снова добавили)

Появилась foreach коллекция (вроде не в Обероне, а в языках после 90-х)

Можно итерировать по объектам, поддерживающим специальный интерфейс

Enumeration it = call.getEnumeration();

while (it.moveNext()) {

T x= (T) it.current; (\*\*\*)

}

В Java в 2005 такое появилось. В С++ с 11 стандарта.

Отдельно Go, Swift

Swift:

while B {

repeat {

for v in {

...

}

} while B;

}

switch e {

case 0:

...

case :

...

}

в case необязательно константа

В Go в case могут быть кортежи (x, y), которые матчатся с e

## Операторы-оболочки

С++ for(v:coll)

C#:

using (var x = new X()) {

...

}

если x поддерживает IDisposable

это синтаксический сахар для следующего кода

var x = new X()

try {

...

} finally {

(IDisposable).Dispose();

}

look(obj) {

...

}

object lock\_obj;

try {

bool entered = false;

lock\_obj -= system.Threading

Monitor.Enter(

obj, ret entered);

} finally {

if (entered)

System.Threading.monitor. … (lock\_obj);

}

Java:

Интерфейс Serialisable

Мы разрешаем в поток выводить его бинарное представление, чтобы потом его обратно

(\*\*\*) (обратно что?)

## Средства развития

Подпрограммы:

* передача управления (control-flow)  
   подпрограммы vs сопрограммы
* передача значений параметров
* функции первого порядка  
   (ФТД, функторы, lambda)

Передача управления



1 вход, 1 выход так было не всегда

Фортран 77:

subroutine S(x, y, z, \*, \*, \*)

...

entry S1(a, b)

…

entry S2(x, z)

...



return 1

return 2

return i

call(1.0, 2.0, 3.0, 100 200, 300)

выход по меткам - плохо! а несколько входов? тоже



сопрограммы - хорошо подходят для реализации компьютеров



реализуют кооперативную многозначность

# Лекция 11 (20.11.18)

## Сопрограммы



Появились в 1958(63)г

Для реализации сопрограмм, вообще говоря (без хаков), нельзя. Поэтому в С/С++ их сделать трудно.

Нужно два стека (мы говорим про императивные языки).

Факт: классические сопрограммы не параллельны (с точки зрения процессора 1 поток вычислений)

В языках: Симула 67, Модула-2

Модула-2: coroutine, address

procedure newprocess(P: proc; N: cardinal; var C: address);

N - размер области, для сохранения состояния сопрограммы

procedure resume(var from, to address);

в 80-х - 90-х корутины были забыты

а в Kotlin появились. сейчас корутины напоминают легковесные потоки (вспомнили Erlang, опередил свое время)

signal:

type signal <- вообще говоря указатель на список сопрограмм (контекстов)

procedure send(signal)

procedure wait(var s: signal)

в 90-х корутины казались ненужными из-за потоков Не системные потоки, потоки

Ada Task

Java Threads

языка, часто слишком тяжеловесные (поэтому в Java протащили JNI)

потом корутины стали возвращаться в языки в виде генераторов

в Java: тератор держит указатель на текущий элемент в контейнере, moveNext() перемещает указатель на следующий элемент

Причем здесь генераторы?

range(L, R) в Python2 [0,1,2,...,99] - неэффективно

xrange(L, R)

Заметили, что итератор легко моделируется сопрограммой

(итератор - частный случай сопрограммы)

def fib():

x, y = 1, 1

while true:

yield x

y, x = x, x+y

в С# 2:

yield return выражение

yield break

class MyCollection: IEnumerable {

private x[] arr;

public IEnumerator

Get Enumerator() {

for (int i = 0; i < arr.length; i++) {

yield return arr[i];

}

}

в Python появились сопрограммы

(хотя Головин говорит, что генератор тоже надо называть сопрограммой)

def sum():

s = 0

while true:

x = (yield)

if x == None:

break

s += x

yield s

s = sum()

s.send(None)

s.send(1)

s.send(N)

s.send(None)

def sum():

s = 0

while True:

s += (yield)

yield s

s.send(None)

s.send(1)

x = s.send(None)

s.send(5)

…..

в С++ 20: появились сопрограммы

generator<int> get \_smth(int start=0, int step=1) {

for (int i ...)

co\_yield

}

У потоков проблемы с синхронизацией и тяжеловесны.

Сопрограммы - программист пишет асинхронный код последовательно

NodeJS action(callback(x, y, err) { … })

поток управления один, но все параллельно

Boost:: asio тот же подход



“Лапша код”, колбэки висят, в каком порядке выполняются?



Как разобраться,что как вызывается, когда перед тобой простыня функций?



Выход-сопрограммы!!!

async/await - парадигма современных сопрограмм

fork, pipe - проблемы производительности

Например, сервера с fork работают очень медленно (здесь рассказ про CGI)

Программировать было легко, но медленно

Кстати, существуют два подхода к написанию параллельных программ:

1. сначала пишем параллельную программу (страдает программист) - MPI
2. система распараллеливает - OpenMP

сопрограммы - это пункт 2

С# программы:

static async void ReadWriteAsync() {

string s = “...”

using(streamwriter writer = var …,

stream\_writer(“hello.txt”, false) {

await writer.WriteLineAsync();

}

)

using (...

await reader.ReadLineAsync();

)

}

static void Fib(int N) {

int x,y;

for(int i = 1; i < N; i++) {

system.Console.Writeln(“Fib = {x}”)

}

}

static async void FibAsync() {

…

await Task.Run(Fib(30))

}

Golang goroutines:

func f(s string) {

fmt.Println(s);

}

go f(“hello”)

go f(“world”)

Tun chan

make (chan int)

func fib(n int, c chan int) {

x,y := 1,1

for (i := 0; i < n; i++) {

c <- x

fmt.Println(“number”)

x, y = y, x+y

}

close(c);

}

func main() {

c := make(chan int)

go fib(10, с)

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

x <- c

fmt.Println(x)

}

}

вариант 2:

func fib(c, quit chan int) {

x,y = 1,1

for {

select {

case c <- x:

x, y = y, x+y

case <- quit:

return

}

}

}

func main() {

c := make(chan int)

quit := make(chan int)

go func() {

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

fmt.Println(<-)

}

quit <- 1

}

fib(c, quit)

}

еще примеры:

func main() {

tick := time.Tick(100 \* time.Millisecond)

boom := time.After(500 \* time.Millisecond)

for {

select {

case <- tick:

fmt.print(“Tick”)

case <- boom:

fmt.Print(“Boom”)

return

default:

fmt.Println(“.”)

time.sleep(50 \* time.Millisecond)

}

}}

## Программы

Control flow - обсудили

Data flow

Способы передачи параметров:

In/Out семантика передачи параметров

1980 Ада:

procedure P(имя: семантика тип)

семантика - in, out, in/out (чтение запись)

в С#:

void f(int x) {

}

вместо int - любой тип, по значению

void f(out int x) {

x = -1

}

void f(ref int x) {

x = -1

}

out, ref - по ссылке

здесь ref - inout

здесь out - out

int x

f(ref x); - ошибка, x - не инициализирован

f(out x); - статуса определенности для x не надо

# Лекция 12 (27.11.18)

Способы передачи параметров:

1. по значению - in
2. по результату - out
3. по значению/результату - inout
4. по ссылке/адресу

1,4 понятно для чего, а вот 2 и 3?

2 и 3 были в устаревших языках

в фортране (т.к. он примитивный) были только по ссылке

по значению/результату - копирование в начале, а потом один раз копирование обратно

минус - косвенная адресация, если по ссылке. если переменная активно используется, то будет медленно работать.

поэтому в некоторых устаревших языках есть по значению/результату

в Ада компилятор сам оптимизировал передачу параметров. но в Ада 95 отказались от этого. программа могла на разных компиляторах давать разные результаты:

procedure P(x,y: inout T) is

begin

x := value1;

if B then raise Error;

y := value2;

end P;

a:T

P(a,a)

неприятная ситуация

Кстати, примеры С++ программ с разными результатами на разных реализациях: в каком порядке будут вызваны конструкторы глобальных статичных объектов

Но в Ада ситуация явно хуже. В Ада 95 приняли решение: inout по ссылке, in по ссылке только для некоторых типов данных

Если в языке у классов нет константных методов, то можно изменять объект (передавать по ссылке)

способ со \*

5) способ передачи параметров (Алгол, Ring)

“передача по имени”

int function f(x, y);

value integer x; (если без value, то по имени)

integer y;

procedure swap(x, y):

integer x;

integer y;

integer tmp;

begin:

tmp = x;

x := y;

y := tmp;

end

integer a[1:10]

integer b;

integer x,y;

swap(x,y);

swap(a,b);

swap(i, a[i]);

swap(a[i], i);

При каждом обращении к параметру по имени вычисляется его адрес и подставляется

(то есть swap написать невозможно)

Передача параметров:

1. позиционная
2. по ключевым словам

В Swift экзотика f(имя1 имя2: тип)

имя1 - имя извне

имя2 - имя внутри

Вообще говоря передача по ключевым словам - не очень, т.к. у функции параметров должно быть немного и их именовать не нужно

# Лекция 13 (11.12.18)

## Обобщенное программирование (generics)

Это статическая параметризация. Т.е. это нужно для языков со статическими типами данных.

Впервые появилось в Ада (там они были сильно влиты в язык, без них прогать было невозможно)

Далее говорим о С++, С#, Java

Описание абстракции

Конкретизация абстракции

Снова про Ада:

generic  procedure sort(x: inout Arr)

Arr, T, Index, Compare

Index - тип индекса



Все, что можно сделать с помощью статической параметризации,   
можно сделать динамически (наследованием)

В С++ добавлен generics для удобства и эффективности

В С# тоже, но в меньшей степени

В Java generics появились для гарантий типа безопасности.

Прироста эффективности в Java не дает

С++:



Оттранслировать отдельно от конкретизации нельзя!

(не как в Ада)

[.h]

[.cpp]

интерфейсы реализации

=> .o

для шаблонов .о файлов быть не может, без конкретизации => нет .cpp файлов

как мы распространяем библиотеку шаблонов?

только в исходных кодах, т.к. нужна конкретизация!

в С++ есть частичная и полная специализация (нет в С#, Java)

Факт: как убирать дубликаты функций

Транспонирование без генерации кода

Линкер => ошибка при обращении к переменной

Транспонирование с генерацией кода

частичная спецификация есть для классов

(у функций есть перегрузка)

template<class x, class y> class T{...}

template<class x, class y> class T<Y\*> …

частичная специализация позволяет писать более эффективный код

Задача:

template <int N>

struct Fun{...}

(сумма квадратов первых 10 чисел)

cout << Fun<10>::res << endl;

Решение:

template <int N> struct Fun {

enum {res = N \* N + Fun<N-1>::res};

};

template <> struct Fun<0> {

enum {res = 0};

}

Факт: double и плавающие вычисления в шаблонах нельзя, т.к. host system != target system

Задача:

static int arr[] = {1,2, ...}

Написать

template <int N> struct Arrsum {

enum {res = ?};

};

cout ArrSum<5>::res << endl;

появилось constexpr => два языка: компилир. С++ и интерпрет-ти во время компиляции.

(\*\*\*)

C#: добавлены в 2.0

class stack <T> {

T[] body;

int top = 0;

public void push(T x) {

...

}

public stack(int n) { body = new T[n]; }

}

А может быть и шаблонный метод у класса.

Можем собрать с абстракцией отдельно от конкретизации.

Генерация кода происходит во время выполнения строки (не компиляции)

Сгенерированный код попадает в кэш кодов.

Т.к. генерация происходит динамически => generics нужны для типа безопасности.

class container<T1> {

public void sort {

x<y

}

}

Проверка корректности происходящего во время компиляции.

Как и в Ада надо сообщать дополнительную информацию

class container<Twhere: > {

struct

new T()

а чаще просто интерфейс указывается

}

Повышается эффективность: статическая проверка на поддержку интерфейса

С++ частичная и полная специализация

template <typename T>

class Matrix {...};

template <typename Real>

class Matrix<complex<Real>> частичная специализация

{...}

template<>

class matrix<bool>

{}

Java:

В C# с 2002г. можно было писать:

List<int> generate List() {...}

int s = 0;

foreach(int x in generate())

s += x

В Java в это время:

List GenerateList() {}

List l = generateList();

for (int i = 0; i < l.length; i++) {

if (l[i] insanceof integer) {

s += (int) l[i];

}

}

class Stack<T> {

T[] body;

public stack(T[] b) {

body = b;

}

}

делать new T в generics Java нельзя!

type erasure:

stack<Integer> s;

Long

Object

Все преобразуется в object!

List<integer> l;

for (int x: l) {

s += x;

}

т.е. генерится только одна функция для object!

class container<T extends comparable<T>> {

public void sort() {x.compareTo(y)}

}

еще о достоинствах generics:

Interface I int<T> {

T produce();

}

interface JConsumer<T> {

void apply(T x);

}

Ковариантность и контрвариантность обобщенных типов

X ковариантна к Y, если X  Y - подтип



X контрвариантна к Y, если Y  X

X инвариантна к Y, если X  Y

Y  X

X = Y (X - контрвариантна, Y - ковариантна)

f(X)

f(Y)



Base

T f(x, y, z, ...)

Derived: Base

T f(x,y,z, ...)

в С++ есть override, плохо, что необязательно

Самое интересное с T\*, T&

Для обобщенных типов:

X(A) ковариантна к Y(B) X(A)  Y(B)

X(A) контрвариантна BY(B) (\*\*\*) Y(B)  X(A)

Animal

Human

Animal [] animals;

Human [] mankind;

animals = mankind; (можно ли так писать?)



В С++:

Animal animals[];

Human mankind[];

animals = mankind;

В С++ типы инвариантны, массивы присваивать нельзя.

В С#, Java они условно-ковариантны. Все дело в том, что они (\*\*\*)



animals[0] = new Dog();



Т.е. ошибка!

ArrayStoreException - Java

ArrayTypeMisMatch - C#

Массив read only - все ок с ковариантными

write only - все ок с контрвариантными

(\*\*\*) пометка слева

Обобщенно:

interface J1 <T> {

T gen();

}

J1<Animal> a;

J1<Human> b;

a = b; ничего страшного

b = a; нельзя

ковариантность!

J1(X) => J1(Y)

interface J2 <T> {

void apply(T x);

}



a = b нельзя

b = a ок

контрвариантность

C#:

interface ISample <out X, in Y> {

X f();

void g(Y y);

}

ISample <Human, Human> a;

ISample <Animal, Animal> b;



# Лекция 14 (18.12.18)

Java:

class F<T> {

public T gen() {...}

public void apply(T x) {...}

}

class X {}

class Y extends X {}

F<X> f1;

F<Y> f2;

F<? extends x> fa

fa = f1;

fa = f2;

F<? super Y> fb;

fb = f1;

fb = f2;

Y или суперкласс Y