# Скалярные базисные типы данных в ОИЯП

Классификация скалярных (примитивных, простых) типов данных

- Арифметические
  - целочисленные
  - вещественные
- Логические
- Символьные
- Порядковые
  - перечисления
  - диапазоны
- Указатели/ссылки

# Скалярные типы - указатели/ссылки

Указатель - абстракция адреса в ЯА.

Насколько абстрактно это понятие?

Адрес - атрибут переменной в любом императивном ЯП.

Ада: V'ADDRESS

C - T v; .... &v (операция взятия адреса, T - абсолютно произвольный ТД).

Ключевой момент - что можно делать с этим атрибутом, насколько он привязан к базисным типам?

Ада - привязка к конкретным адресам памяти (например, страница ввода/вывода) и т.п., что сильно зависит от реализации => не используется широко в большинстве программ

C - &v => возвращает тип T *. Где может использоваться тип T*?

Везде, где может использоваться тип T (то есть везде)

# Скалярные типы - указатели/ссылки

Указатель - абстракция адреса в ЯА.

Насколько абстрактная?

- "Нестрогие указатели" - в ЯП, где указательный тип используется для ссылок на объекты (почти) любых ТД (C/C++)
  T x; T * p = &x; ..... p = (T*) malloc (sizeof *p); .... free(p);
  p - это "ящичек", где хранится адрес памяти. Изменить его можно только явным образом (присвоить другое значение) - очень "тонкий" слой абстракции
- "Строгие указатели" - в ЯП, где указательный тип используется (исключительно) для ссылок на (анонимные) объекты, размещаемые в динамической памяти (Ада, Паскаль, Модула-2, Оберон, Go,....)
  TYPE PT = POINTER TO T;
  TYPE T = ......;
  VAR X : T; P: PT; .....
  NEW(P); /* размещение в динамической памяти объекта типа T */
  Возможна автоматическая сборка мусора(Оберон, Go, Ада - опционально). P - может быть как "ящичком"  в стиле C, так и ссылкой на функцию, возвращающую реальный адрес (может меняться), или что-нибудь не менее абстрактное.

# Скалярные типы - указатели/ссылки

Указатель - абстракция адреса в ЯА.

"Строгие указатели" - в ЯП, где указательный тип используется (исключительно) для ссылок на (анонимные) объекты, размещаемые в динамической памяти (Ада....)

"Строгие указатели" могут быть реализованы как в стиле указателей C (пример - Турбо Паскаль и ряд других реализаций - GetMem(P, N), FreeMem(P, N)), так и на более высоком уровне.

Оберон, Ада => указатель даже синтаксически похож на ссылку.
TYPE PT = POINTER TO T;

T = RECORD X,Y: INTEGER ..... END;

VAR P: PT; tt: T;

NEW(P); ....P.X := Foo(0); P.Y := P.X + 1; /* автоматическое "разыменование" */

Чего не хватает? Операции доступа к объекту в ДП "целиком"

tt := P^ /* ^ - операция разыменования */

# Скалярные типы - указатели/ссылки

Указатель - абстракция адреса в ЯА.

Оберон, Ада => указатель даже синтаксически похож на ссылку.

TYPE PT = POINTER TO T;

T = RECORD X,Y: INTEGER ..... END;

VAR P: PT; tt: T;

NEW(P); ....P.X := Foo(0); P.Y := P.X + 1; /* автоматическое "разыменование" */

Чего не хватает? Операции доступа к объекту в ДП "целиком"

tt := P^ /* ^ - операция разыменования */

Ада - практически так же, как и в Обероне (не всегда есть сборка мусора)

type T is record X,Y: integer; .... end record;

type PT is access T;

P: PT, tt: T;

P := new T; P.X := Foo(0); P.Y := P.X + 1;

tt := P'all;

# Скалярные типы - указатели/ссылки

Указатель - абстракция адреса в ЯА.

```
generic
   type Object(<>) is limited private;
   type Name   is access  Object;
procedure Unchecked_Deallocation(X : in out Name);
procedure Free is new Unchecked_Deallocation(T, PT);
Free(P); // явное освобождение памяти
```

# Скалярные типы - указатели/ссылки

Указатель - абстракция адреса в ЯА.

"Строгие указатели" - в ЯП, где указательный тип используется (исключительно) для ссылок на (анонимные) объекты, размещаемые в динамической памяти (Ада....)

Интересна эволюция понятия указателя в ЯП Ада.

Ада 83 - чистая строгая модель. Допускается сборка мусора. Надежно (более-менее)

Ада 95 - уже есть указатели на *произвольные* объекты (как в ДП, так и другие). То есть проблемы как в С? Зачем они в 1985, если в 1983 указатели на произвольный объект не нужны? Что изменилось?

# Скалярные типы - указатели/ссылки

Указатель - абстракция адреса в ЯА.

"Строгие указатели" - в ЯП, где указательный тип используется (исключительно) для ссылок на (анонимные) объекты, размещаемые в динамической памяти (Ада....)

Интересна эволюция понятия указателя в ЯП Ада.

Ада 83 - чистая строгая модель. Допускается сборка мусора. Надежно (более-менее)

Ада 95 - уже есть указатели на *произвольные* объекты (как в ДП, так и другие). То есть проблемы как в С? Зачем они в 1995, если в 1983 указатели на произвольный объект не нужны? Что изменилось?

Изменились условия функционирования ЯП => появились новые КРИТИЧНЫЕ технологические потребности. Какие?

# Скалярные типы - указатели/ссылки

Эволюция понятия указателя в ЯП Ада.

Ада 83 => Ада 95

Изменились условия функционирования ЯП => появились новые КРИТИЧНЫЕ технологические потребности. Какие?

От концепции "везде одна Ада" к концепции - "Ада в разных окружениях" Окружение == ОС => критично обращение к API ОС.

Внимание - вопрос: на каком языке написаны современные ОС?

# Скалярные типы - указатели/ссылки

Эволюция понятия указателя в ЯП Ада.

Ада 83 => Ада 95

Изменились условия функционирования ЯП => появились новые КРИТИЧНЫЕ технологические потребности. Какие?

От концепции "везде одна Ада" к концепции - "Ада в разных окружениях" Окружение == ОС => критично обращение к API ОС.

Внимание - вопрос: на каком языке написаны современные ОС?

Правильно - это C/C++. => обращение к API современной ОС (системный вызов) - это обращение к функции, написанной на С и использующей СД языка С. И как тут обойтись без понятия указателя на ЛЮБОЙ объект? НИКАК.

# Скалярные типы - указатели/ссылки

Эволюция понятия указателя в ЯП Ада.

Ада 83 => Ада 95

Как сделать совместимо, сохраняя надежность (хот как-то)?

В общем случае - нетривиальная задача.

Два вида указательных типов

"Старый" - только для манипуляции с объектами в ДП.

type T1 is record .... end record;

type PT1 is access T1;

t1: T1; p1: PT1;

p1 := new T1;  t1 := p1.all;

"Новый"  - для возможности передавать адреса любых объектов

type PTInt is access all integer; type PT1ALL is access all T1;

I,J: integer; pp: PT1ALL;

swap_c(I'access, J'access); -- сравни на C: swap(&i, &j);

pp := p1; -- допустимо - обратное присваивание - нет.

# Скалярные типы - указатели/ссылки

Ссылки - есть понятие ссылки (reference) в ЯП с референциальными типами, есть понятие ссылки в C++.

Часто переплетается (и "путается") с понятием передачи параметров по "ссылке".

Паскаль, Модула-2, Оберон - параметр-переменная. Явное понятие ссылочного типа отсутствует. "Тонкая обертка" адреса параметра.

ЯП с референциальными типами.

Значения референциальных ТД размещаются в динамической памяти (=>анонимны), доступ к ним осуществляется посредством ссылок (именованными могут быть ТОЛЬКО ссылки, но не сами объекты).

# Скалярные типы - указатели/ссылки

ЯП с референциальными типами.

Есть типы-значения (value types) и референциальные типы (классы, массивы, интерфейсы)

Динамические ЯП - все имеют референциальные типы (Python, JS, .....). Имя "объекта" - всегда ссылка на реальный объект из ДП.

Почему? - как сделать универсальное понятие переменной? - через понятие ссылки.

Статические ЯП. Есть C++, а есть другие ООЯП (Smalltalk, Java, C#, Delphi, Objective C, Swift, .....). В "других" - классы - референциальные типы.

C#, Java - классы, массивы, интерфейсы

```
class X{.....}

X a; // это ссылка!

a = new X(); // размещение объекта в ДП

X b = a; // две ссылки на ОДИН И ТОТ ЖЕ объект

int [] arr = new int[128]; // ссылка на массив

int [] arr2 = arr; // две ссылки на ОДИН И ТОТ ЖЕ объект
```

# Скалярные типы - указатели/ссылки

ЯП с референциальными типами.

Есть типы-значения (value types) и референциальные типы (классы, массивы, интерфейсы)

Python:

```
a = [1,2,3,4]
b = a # две ссылки на ОДИН И ТОТ ЖЕ объект
a[0] = ['q',-1]
print(b) # [['q',-1], 2, 3, 4]
```

# Скалярные типы - указатели/ссылки

ЯП с референциальными типами.

Есть типы-значения (value types) и референциальные типы (классы, массивы, интерфейсы)

Отличия типов-значений от референциальных (ссылочных) типов.

- имя связывается с объектом, и изменить эту ассоциацию нельзя
- размещение в памяти
- семантика присваивания

# Скалярные типы - указатели/ссылки

ЯП с референциальными типами.

Есть типы-значения (value types) и референциальные типы (классы, массивы, интерфейсы)

Типы-значения. Зачем? В Smalltalk - нет ("все" есть объект)

- эффективность
- совместимость

В Java - нет такого "явного" понятия (типы-значения). В C# - есть.

Java - "примитивные" типы (их уже все рассмотрели) + реф. типы (классы, массивы, интерфейсы). Примитивные - и есть типы-значения. Вопрос: перечисления - это типы-значения или ...?

C# - ТД Java + добавлены перечисления и структуры.

Перечисления - "чистые" типы-значения.

Структуры - классы с ограниченной функциональностью, не имеющие референциальной семантики. Зачем? Для эффективности.

Аналогичный подход (но скорее с точки зрения совместимости) -  в Delphi.

# Скалярные типы - указатели/ссылки

ЯП с референциальными типами.

Есть типы-значения (value types) и референциальные типы (классы, массивы, интерфейсы)

Типы-значения. Зачем? В Smalltalk - нет ("все - есть объект")

- эффективность
- совместимость

Если в "чистом" ООЯП есть отдельно классы и отдельно типы-значения, то как быть с главной идеей ООП ("все - есть объект")?

Ответ - классы-обертки (wrapper classes) - C#, Java, JavaScript,....

Вопрос - есть ли в Go классы-обертки?

# Скалярные типы - указатели/ссылки

Ссылки C++: отличаются от ссылок в референциальных ЯП

Страуструп: "ссылка - имя объекта".

int i;

int& ri = i; // теперь ri и i - полные взаимозаменяемые синонимы

int &rii = *new int(-1); // анонимный объект получил имя - rii.

!-я операция, применимая к ссылкам в C++ - инициализация

Любая операция, примененная к ссылкам C++ после инициализации, - это операция над объектом, именем которого является ссылка.

# Скалярные типы - указатели/ссылки

Контексты инициализации

```
int i;
int& ri = i;
void foo(int& ri);   // при вызове f(i); - ссылки ЯП Паскаль
void cfoo(const int& ri);   // при вызове f(i); - Паскаль - аналога нет
int& bar(int a[], int n) {  .... return a[k]; ..../* при возврате */ ...}
```

Для ссылок-членов класса - особый синтаксис:

```
class X {
  int& _ri;
public:
  X(int& ri) : _ri(ri) {} // для инициализации ссылок и вызовов конструктора базы
};
X x(i);
```

# Скалярные типы - указатели/ссылки

Итог: есть указатели, ссылки рефЯП и ссылки C++

В чем отличия?

# Скалярные типы - указатели/ссылки

Итог: есть указатели, ссылки рефЯП и ссылки C++

В чем отличия?

В наборе операций!!!

Указатели:

- адресная арифметика - экзотика - очень низкий уровень - фактически только C!
- инициализация и присваивание
- сравнение - ==, != (другие - только для адресной арифметики)
- *разыменование*

Ссылки:

- никакой адресной арифметики и в помине нет!
- инициализация и присваивание (очень похоже на указатели!) - но проблема копирования - это ПРОБЛЕМА
- разыменования нет (всегда неявно подразумевается)
- сравнение - отдельная тема (что такое a == b?) - так ссылка тождественна объекту или нет?
    - либо все 6 операций, которые означают сравнение самих обозначаемых объектов, тогда появляются === и !==
    - либо как для указателей (массивы)
- куча других операций НАД самим объектом (как и сравнение) - они подразумевают неявное разыменование.

Ссылки C++: инициализация ссылки. Все остальные операции - над самим объектом (то есть определяются статически по типу ссылки).

# Скалярные типы - указатели/ссылки

Есть ли референциальные ЯП с указателями? Реально нет. Есть "вкрапления" в ЯП "чуждых" понятий, которые практически не смешиваются.

C# - unmanaged (unsafe) code

```
byte [] b = new byte[1024]; ......
fixed (byte * pb = b) {
   byte * pFin = pb + 1024;
   while (pb < pFin) { *pb |= 0x80; pb++; }
   // здесь можно обращаться к C-RTL - включая malloc, free....
   // почему fixed
}
```

unsafe - атрибут - налагает много ограничений на использование сборок (assemblies)

C++ - есть .NET диалект C++ - "managed C++" или CLI/C++

Два языка в одном - C++ и C#.

gc-ссылки, C++-указатели и C++-ссылки - "смешать, но не взбалтывать" - несовместимы между собой

# Скалярные типы - указатели/ссылки

C++ - есть .NET диалект C++ - "managed C++" или CLI/C++

Два языка в одном - C++ и C#.

gc-ссылки, C++-указатели и C++-ссылки - "смешать, но не взбалтывать" - несовместимы между собой

class ref DotNetClass { // может использоваться в сборках .NET

.....

};

Object ^ obj = gcnew Object();  // object obj = new Object();

Как изучать? Смотрим на строчку из C# и то, как она реализуется на C++/CLI:

string [] arr = new string[100]; // C#

array<String^> ^arr = gcnew array<String^>(100); // C++/CLI

Нечто подобное наблюдали в языке Objective C: это C + SmallTalk (только без GC)

# Скалярные типы - указатели/ссылки

Сборка мусора (garbage collection).

Автоматическая сборка возможна в ЯП, где есть понятие "строгого" указателя (близко к понятию ссылки) или референциальных ЯП.

В чем проблема?

2 этапа:

- найти, пометить все  "ненужные" объекты и выполнить их уничтожение (вызов деструктора, финализатора и т.п.)
- дефрагментировать память - самое неприятное

ЯП со сборкой мусора - все динамические ЯП, C#, Java, Оберон, Go, Swift, Haskell...

ЯП с возможностью сборки мусора - Ада...

ЯП без сборки мусора - C/C++, Rust, Delphi

# Структурные базисные типы данных в ОИЯП

Классификация структурных (составных) типов данных

- Массивы
- Записи (структуры)
  - объединения типов (размеченные и неразмеченные)
- Кортежи
- Словари (таблицы) и множества
- Файлы
- Строки (динамические последовательности символов)

Снова сравним с ЯП Паскаль....

# Структурные базисные типы - массивы

Главная абстракция памяти фон-неймановской модели

Однородная непрерывная последовательность однотипных переменных.

Последовательность - важна однородность и непрерывность.

Однотипность - только для статических ЯП. В динамических ЯП - просто однородная непрерывная последовательность переменных

В общем случае:

Array<ElemType, IndexType> (ElemType - любой, IndexType - порядковый)

Операции(A: Array<ElemType, IndexType>, I: IndexType)

[]: A,I -> ref ElemType  Синтаксис: A[i]. Редко: A(i)

Очень важно: ***ref*** ElemType. Следовательно значения переменных-элементов массива можно менять (в общем случае).  A[i] := A[j]

Len: A-> integer

FirstIndex: A->IndexType

LastIndex: A->IndexType

Есть много вспомогательных, но эти - главные

# Структурные базисные типы - массивы

В общем случае:

Array<ElemType, IndexType> (ElemType - любой, IndexType - порядковый)

Операции(A: Array<ElemType, IndexType>, I: IndexType)

[]: A,I -> ref ElemType  Синтаксис: A[i] = A[j]. Редко: A(i) = A(j)

Len: A-> integer

FirstIndex: A->IndexType

LastIndex: A->IndexType

Очень важно: из однородности и непрерывности вытекает, что A[j] - время вычисления ***не зависит от значения j***

Уже обсуждали, что в современных ЯП все несколько упростилось:

IndexType === integer

FirstIndex === 0 => LastIndex === Len(A) - 1

Отсюда **ГЛАВНЫЙ** вопрос реализации массивов в (современных) ЯП:

Len: A-> integer  -- время связывания этой операции

# Структурные базисные типы - массивы

**ГЛАВНЫЙ** вопрос реализации массивов в (современных) ЯП:

Len: A-> integer  -- время связывания этой операции

Динамические ЯП: ***длина - всегда динамическое свойство***

Основные следствия:

- массивы в динамических ЯП - референциальные типы
- в динамических ЯП имеется большой набор операций по слиянию массивов, добавлению (в массив) и удалению (из массива) как отдельных элементов, так последовательностей

# Структурные базисные типы - массивы

Пример 1 - язык Питон (списки - lists, len(a) - функция-длина a)
a = [1,2, 'line', ['sub', 'list', None], True]
Индексы - от 0 до len(a)-1 => a[len(a)-1]
Интересная особенность:
a[-i] === a[len(a)-i] для отрицательных значений индекса
***Контроль индекса***:
a[-10] или a[25] - выбрасывает исключение
***Референциальность***:
a = [1,[2,3],4]
b = a
a[1] = 'not list at all'

# Структурные базисные типы - массивы

Пример 1 - язык Питон (списки - lists, len(a) - функция-длина a)
a = [1,2, 'line', ['sub', 'list', None], True]
Индексы - от 0 до len(a)-1 => a[len(a)-1]
Интересная особенность:
a[-i] === a[len(a)-i] для отрицательных значений индекса
***Контроль индекса***:
a[-10] или a[25] - выбрасывает исключение
***Референциальность***:
a = [1,[2,3],4]
b = a
a[1] = 'not list at all'
***Создание массива(списка)***:
a = list('foo') # из любой последовательности
l = a + [1,2,3] # конкатенация списков

# Структурные базисные типы - массивы

Пример 1 - язык Питон

***Изменяемость***:

a = [1,2,3]

append: a.append(arg)

a.append(1)

a.append([1,2]) # что-то непохоже на Лисп и Пролог?

extend: a.extend(arg) => a += arg

a.extend(1) # ошибка !???

a.extend([1,2]) # а вот теперь похоже на Лисп и Пролог - добавление списка

a.extend('foo') # а теперь похоже на питон

a.extend([1,[2,3],4]) # добавление списка не линеаризует дерево

reverse: a.reverse()

pop: x = a.pop(arg)

a.pop()

a.pop(2)

clear: a.clear()

insert: a.insert(pos, el)

remove и del: a.remove(el)  del a[1] # del - особая операция - удаляет переменную из ОВ

remove(el) - удаляет первый элемент со значением el

# Структурные базисные типы - массивы

Пример 1 - язык Питон

***Массивы и строки***:

string => list

extend: a.extend('foo')

либо list('foo')

Обратная операция

join: list (of strings) => string

','.join(a)

''.join(a) => обратная к list(string)

# Структурные базисные типы - массивы

Пример 2 - язык JavaScript

a = new Array(1,2,3,'line', true) // общая форма

"Синтаксический сахар": a = [1,2,3,'line', true] // удобная форма

Индексы - от 0 до a.length-1 => a[a.length-1]

***Контроль индекса - его нет!!!***:

a[-10] или a[25] - undefined

Интересная и фундаментальная особенность (но не как в Питоне):

При чтении из массива:  (x = a[index])

a[index] === undefined при index вне диапазона от 0 до a.length-1

При записи в массив:  (a[index] = val) - "чудеса"

Если index в диапазоне от 0 до a.length-1 a[index], то значение элемента с индексом index заменяется на val

Если index >= a.length, то массив "расширяется" до длины index + 1 (new a.length === index + 1).

a[index] = val; a[i] = undefined для всех i = old a.length ... index-1

Самое "интересное":  если index < 0 (или если индекс не преобразуется в целое (!важно!) число в диапазоне 0..a.length)

a[-1] = "negative index"

a[1.05] = "float index"

a["not an index at all"] = "strange index"

В этом случае массив начинает вести себя как обычный JavaScript-объект

# Структурные базисные типы - массивы

Пример 2 - язык JavaScript

***Референциальность***:

```
a = [1,[2,3],4]
b = a
a[1] = 'not list at all'
// как в Python
```

# Структурные базисные типы - массивы

Пример 2 - язык JavaScript

***Изменяемость***:

a = [1,2,3]

push: n = a.push(args) // работает и как append, и как extend

n = a.push(1)

n = a.push(1,2)

n = a.push([1,2])

n = a.push([1,2,[3,4]]) // # добавление списка не линеаризует дерево

reverse: b = a.reverse()

pop: x = a.pop() // только ОДИН элемент с конца

x = a.pop()

x = a.pop(-10000)

x = a.pop(1,2,3,4,5,6, -1000.00009)

shift: x = shift()  // только ОДИН элемент с начала

.....

unshift: a.unshift(args) // как push - только в начало

....

delete: delete a[1] # delete - особая операция - удаляет переменную из ОВ

# Структурные базисные типы - массивы

Пример 2 - язык JavaScript

***Массивы и строки***:

string => array

'foo'.split('')

'foo'.split()  // все как в Python

Обратная операция

join: array => string

Отличия от Python

- метод массива, а не строки
- любой массив!!!
- линеаризация, но странная (как и весь язык)

a.join(sep)

([1,2,"line!", true]).join(',')

([1,[2,3,[4,5],6], 7, [], 8]).join(';')

([1,[2,3,[4,5],6], 7, [], 8]).join('')

# Структурные базисные типы - массивы

Пример 2 - язык JavaScript

***Большое число других встроенных операций***:

splice (манипуляции - вставка, замена, удаление)

concat - аналог + в Python (а что по поводу + в JavaScript?)

Поиск: indexOf / lastIndexOf / includes

Map-методы (применяют функцию-аргумент поэлементно):

filter, find и много других

.....

Больше, чем в Python. Почему?

В JS - массив - основное средство абстракции для последовательностей

Python - список - один из частных случаев последовательности

filter(predicate, l) => объект-последовательность

l = list(l) => сделать список из любой последовательности