

С/к Введение в современное программирование (v.5.5)

Физфак МГУ. 2006/7 уч. год.

Лекция 11

Еще о постановке базовой техники: *".. When software developers are in their teens or early twenties, their priorities are typically focused on learning and working with the latest technology. They describe themselves as PERL hackers, Linux experts, Enterprise JavaBeans developers or .NET programmers.*

But because the technology is constantly changing, younger developers tend to just barely learn a technology .. and then start over again, either learning something completely new or absorbing the latest incarnation of what they worked with previously.

The problem is that they keep learning slightly different flavors of the same low-level, fundamental skills over and over again .. developers start to understand that many of the fundamentals, which they may or may not have been taught in school, remain the same regardless of the technology employed. .."

Еще о стадных эффектах: *".. many people tend to overvalue popularity. It's important, but dismissing a language just because it's "unpopular" is a rather juvenile thing to do. At the end of the day, you've got to make the following considerations:*

1) Does language X have the libraries I need?

2) Does language X have the kind of commercial support I need?

3) Does language X have enough programmers to hire to work on the project?

... (2) and (3) are very close to non-issues for all but the most high-profile projects, yet, people tend to overvalue popularity anyway. As a result, they let themselves miss out on some very real technical advantages. .."

Еще о языках и библиотеках (2006-11-14): *".. In fact, my experience has been, using a much better language often cuts development time so much, that it offsets the time that is needed to be spent writing libraries and plumbing. Shocking? Impossible? No way. I have proof ..*

In fact, sometimes writing a library from scratch in a great language, custom suited to your needs, is often quicker than attempting to write "glue code" in a poor language for a "one size fits all" library. .."

Еще о параноидальных заклинаниях насчет простоты и надежности (2006-09-08): *".. Programming is mentally overwhelming, even for the smartest of us. The more I program, the harder I realize it is. .."*

Еще об автоматическом управлении памятью (2006-07-22): *"... VS.dotnet и дельфи которые мне приходится выключать по 3, 4 раза в день из за memory violation..."*

"Good design, done right, is timeless." ср.: "Оберон -- Дао программирования" (www.chernyshov.com).

Пример использования средств SYSTEM:

Речь идет об арифметике беззнаковых 32- и 64-разрядных чисел.
Целое next равномерно распределено во всем диапазоне INTEGER.

(George Marsaglia's MWC (multiply with carry) random number generator posted in sci.stat.math, sci.math (30-Jan-98) as x86 assembler routine. For references do a DejaNews search (www.dejanews.com) on "A very fast and very good Random Number Generator". This implementation is based on a one by B.Treutwein, corrected by FVT. *)*

```
MODULE Kurs2006Ex35unsigned;
IMPORT Log := StdLog, SYSTEM;

VAR next, carry: INTEGER; (* global *)

PROCEDURE Next*(): REAL; (* вся зависимость от SYSTEM здесь *)
CONST
  a = 2083801278; (* from the list of constants recommended by
Marsaglia *)
  x = 0.5 - MIN( INTEGER ); (* REAL *)
  y = ( 1.0 - MIN( INTEGER ) ) + MAX( INTEGER ); (* REAL *)
TYPE
  Aux = RECORD [union]
    l: LONGINT;
    i: RECORD i0, i1: INTEGER END;
  END;
  VAR t, X, C: Aux;
BEGIN ASSERT( ( next # 0 ) OR ( carry # 0 ), 20 );
  X.i.i0 := next; X.i.i1 := 0;
  C.i.i0 := carry; C.i.i1 := 0;

  (* основная "рандомизирующая" операция: *)
  t.l := a * X.l + C.l;

  next := t.i.i0; carry := t.i.i1;
  ASSERT( ( next # 0 ) OR ( carry # 0 ), 100 );
  RETURN ( x + next ) / y
END Next;

PROCEDURE Do*;
CONST N = 10000000; VAR i: INTEGER; x, s: REAL;
BEGIN s := 0;
  FOR i := 1 TO N DO
    x := Next(); s := s + x + x*x
  END;
  Log.Real( s / N - 1/2 - 1/3 ); Log.Ln
END Do;

BEGIN next := 314159; carry := 271828
END !Kurs2006Ex35unsigned.Do -- три клика:
5.324581208861109E-5
1.804711961902553E-4
-2.185371279047195E-5

next, carry -- просто хранилища для 32 бит.
Q Где они "интерпретируются" как 32/64-битные положительные целые?
Q В чем прокол (первого автора) в присваиваниях X.l := next; C.l := carry?
```

Варианты устройства списка

Фиктивные элементы-"стражи" -- в начале и/или в конце. Для упрощения кода (исключение спец. случаев -- IF; картинка). Всегда присутствуют, должны создаваться сразу -- инициализация списка.

Страж в конце может быть общим для группы списков (некий nil -- псевдо-NIL). Страж может быть один, но список -- кольцевой.

Или кольцевой список без всякого стража.

(Картинки-----)

-----)
-----)

Проблема: не перепутать стража с регулярными элементами! (особое значение? ...)

Много вариантов работы:

— вставлять в начало, в конец, в середину (упорядоченность); всяко

— вставлять по одному, по нескольку

— "съедать" с начала, из середины, с конца (**Q** что здесь подразумевается?)

— нужно много ходить или нет; по одному или парами

— как все эти действия "замешаны"

Придумать общую библиотеку, равно пригодную для всех случаев жизни, еще ни у кого не получилось.

Примеры

— Обработка данных: списки создаются и тихо живут (полные проходы) до конца -- уничтожаются целиком. Возможное преобразование -- фильтрация.

— Специализир. алгебраич. система: вставки только в короткие списки, в остальном постоянно поддерживается упорядоченность списков, львиная доля - списки, которые создаются один раз и "съедаются"; списки могут представлять часть последовательностей, хранимых во вне.

Проблема Абстрагироваться от технических деталей

Два аспекта -- скрыть манипуляции с полями вроде next; -- скрыть технич. детали вроде второго указателя last. Начнем со второго.

Из архитектуры: каждому концепту — тип

Концепт "списка" и концепт "элемент списка" — два разных концепта. "Логика"/"функциональность" списка — отлична от "логики"/"функциональности" элемента списка.

Общее правило:

Каждому концепту — по типу

В кр. случае, по записи (если принципиально одиночн.).

Не без исключений, *конечно*.

Мы раньше работали с одним списком, "якорили" в глобальной переменной.

Что если много? (напр., сортировка слияниями)

Если динамика заведомо предсказуема, размещаем на стеке:

List* = RECORD

first: Link

END;

Если динамика **не** предсказуема, размещаем на куче:

List* = POINTER TO ListDesc;

ListDesc = RECORD

first: Link

END;

NB Суффикс Desc (description) -- традиция. Следовать! (для читабельности)

Синтаксическое сокращение (если **только** на куче; ср. Java):

List* = POINTER TO RECORD

first: Link

END;

NB ББ внутри себя все равно дает имя соотв. типу записей -- "List^"

NB List может содержать другие поля, относящиеся до списка в целом:

List* = RECORD

first: Link;

last: Link;

len: INTEGER;

END;

Процедуры должны работать с конкретно задаваемым списком-List, поэтому он должен быть параметром.

Два варианта:

0) Классический Оберон:

добавить к процедурам модуля параметр-список:

для List: RECORD: PROCEDURE Push* (VAR l: List; x: REAL);

для List: POINTER: PROCEDURE Push* (l: List; x: REAL);

1) Компонентный Паскаль/Оберон-2:

привязать процедуры к этому типу:

для List: RECORD: PROCEDURE (VAR l: List) Push* (x: REAL), NEW;

для List: POINTER: PROCEDURE (l: List) Push* (x: REAL), NEW;

Тогда вызов l.Push(x); вместо ИмяМодуля.Push(l, x);

Не просто синтаксическое удобство — но орг. средство, допускающее дальнейшее развитие (об этом напоминает атрибут NEW в заголовке процедур - компилятор сам напомнит про NEW).

В обоих случаях мы полностью абстрагируемся от внутреннего устройства списка. Может даже оказаться массивом... но мы сейчас фокусируемся на списках как таковых.

186 Инициализация/создание сложного объекта

187 Под таким объектом понимаем, например, List.

188 Эта проблема — всегда в голове для каждой переменной и для каждого типа.

189 **Вариант:** размещается на стеке: List = RECORD.

190 Если у него есть поле len: INTEGER (длина списка), то в ней будет случайное
191 число — нельзя начинать работу без инициализации.

192 PROCEDURE (VAR l: List) Init*, NEW;

193 и в ней

194 ASSERT(l.first = NIL); (* страховка *)

195 l.len := 0;

196 **Задача** Как заставить клиента выполнять Init для произвольного типа записей,
197 который мы хотим спроектировать?

198 Проблема: можно забыть!

199 Способ гарантировать правильную инициализацию:

200 TYPE

201 **List*** = POINTER TO LIMITED RECORD

202 len-: INTEGER;

203 first-: Звено

204 END;

205 ...

206 PROCEDURE **NewList*** (): List;

207 VAR new: List;

208 BEGIN

209 NEW(new); new.len := 0; new.first := NIL; **RETURN** new

210 END NewList;

211

212 **0)** LIMITED запрещает модулям-клиентам размещать как на стеке, так и на куче
213 (NEW).

214 **1)** Зато они могут обратиться к NewList — пример того, что называется

215 фабричная процедура = factory function

216 Таковая может иметь параметры для инициализации (часто).

217

218 Можем полностью "законопатить" все щели -- лишить клиента всякой
219 возможности (кроме лома-SYSTEM...) что-то испортить:

220

221 MODULE Kurs2006Ex**36**Lists1; (* сугубая иллюстрация средств *)

222 IMPORT StdLog, Math, In := FVTsysIn (* Epse21SysIn *);

223

224 TYPE

225 **List*** = POINTER TO LIMITED RECORD

226 first: Link

227 END;

228

229 **Link*** = POINTER TO LIMITED RECORD

230 next-: Link;

231 x-: REAL

232 END;

233

234 PROCEDURE **NewList*** (): List;

235 VAR new: List;

236 BEGIN

237 NEW(new); **RETURN** new

238 END NewList;

239 PROCEDURE (l: List) **IsEmpty*** (): BOOLEAN, NEW;

240 BEGIN

241 **RETURN** l.first = NIL

242 END IsEmpty;

243 PROCEDURE (l: List) **First*** (): Link, NEW;

244 BEGIN **RETURN** l.first

245 END First;

246 PROCEDURE (l: List) **Push*** (x: REAL), NEW;

247 VAR new: Link;

248 BEGIN

249 NEW(new); new.x := x;

250 (* вставка звена new в начало списка: *)

251 new.next := l.first; l.first := new

252 END Push;

253 PROCEDURE (l: List) **Pop*** (): REAL, NEW;

254 VAR res: REAL;

255 BEGIN

256 ASSERT(l.first # NIL, 20);

257 (* удаление первого звена из начала списка: *)

258 res := l.first.x;

259 l.first := l.first.next;

260 **RETURN** res

261 END Pop;

262 (* вместо PROCEDURE **Clear***; делать просто l := NIL *)

263 PROCEDURE **Do***; (* может быть в любом модуле *)

264 VAR l: List; x: REAL; z: Link;

265 BEGIN

266 l := NewList();

267 In.Open; ASSERT(In.Done, 20);

268 In.Real(x);

269 WHILE In.Done DO

270 l.Push(x);

271 In.Real(x)

272 END;

273 IF l.IsEmpty() THEN StdLog.String('список пуст'); StdLog.Ln

274 ELSE

275 z := l.First();

276 WHILE z # NIL DO

277 StdLog.Real(z.x);

278 z := z.next

279 END

280 END;

281 StdLog.Ln

282 END Do;

283 END Kurs2006Ex36Lists1.

!Kurs2006Ex36Lists1.Do 1 2 3 4 5!

```

284 DEFINITION Kurs2006Ex36Lists1;
285     TYPE
286         Link = POINTER TO LIMITED RECORD
287             next-: Link;
288             x-: REAL
289         END;
290
291     List = POINTER TO LIMITED RECORD
292         (l: List) First (:): Link, NEW;
293         (l: List) IsEmpty (:): BOOLEAN, NEW;
294         (l: List) Pop (:): REAL, NEW;
295         (l: List) Push (x: REAL), NEW
296     END;
297
298 PROCEDURE Do;
299 PROCEDURE NewList (:): List;
300
301 END Kurs2006Ex36Lists1.

```

NB Доступ к List.first — спрятали в Link.First(),
но доступ к Link.next — оставили.
Можно было бы сделать оба одинаково.

NB У клиентов нет никакого способа создать Link — и вообще испортить List.

Security, блин 🙄

Упр Переделать модуль так, чтобы список всегда был отсортирован по
неубыванию.

Здесь спектр возможностей: слияния-сортировки; ООП; разные алгоритмы.
Все обязательные.
Но без ООП невозможно въехать в организацию библиотек ББ.

310 Полиморфизм и наследование

311 polymorphism, inheritance

312 Две базисных мотивировки:
313 — работа со списком не зависит от полей-содержимого элементов;
314 — если моделируем функции записями, хотим, чтобы некоторые процедуры
315 (напр., интегрирования) работали со многими разными функциями, у которых
316 совершенно разные процедуры вычисления -- но одинаковый интерфейс.
317 Первая мотивировка должна быть понятна, до второй дойдем.

318 Историч. отступление

319 New York, February 5, 2002. The Association for Computing Machinery (ACM) has
320 presented the 2001 A.M. Turing Award, considered the "Nobel Prize of Computing,"
321 to Ole-Johan **Dahl** and Kristen **Nygaard** of Norway for their role in the invention of
322 object-oriented programming ...
323
324 Norwegian Computing Center
325 Nygaard пришел из военной проблематики (1949-1960), занимался задачами
326 discrete event simulation, в 1961 г. замыслил спец. ЯП на основе Алгола-60,
327 пригласил Дала в качестве программиста для написания компилятора -->
328 Simula I.

330 At the NCC there was no initial enthusiasm for Simula. Dahl and Nygaard were
331 told variously that:
332 "1. There would be no use for such a language as Simula.
333 2. There would be use, but it had been done before.
334 3. Our ideas were not good enough, and we lacked in general the competence
335 needed to embark upon such a project, which for these reasons never would
336 be completed.
337 4. Work of this nature should be done in countries with large resources, and not
338 in small and unimportant countries like Norway."
339 [K.Nygaard and O.-J.Dahl, The development of the SIMULA language, ACM
340 SIGPLAN Notices, 13(6), 1978, pp.245-272]
341 (**NB** Вдуматца в возражения -- они всегда одинаковые.)
342 Simula I начал использоваться людьми вокруг, они вдохновились и стали думать
343 дальше.
344 В 1967 додумались до простой схемы = "полиморфизм" + "наследование" -->
345 **Simula-67**.
346
347 ... led the way for software programmers to build software systems **in layers**
348 **of abstraction**. With this approach, each layer of a system relies on a platform
349 implemented by the lower layers. Their approach has resulted in programming
350 that is both accessible and available <?> to the entire research <?> community.
351
352 The work of Drs. Dahl and Nygaard has been instrumental in developing a
353 remarkably responsive programming model that is both flexible and agile when
354 it is applied to complex software design and implementation," said John R.
355 White, executive director and CEO of ACM ...
356
357 (**NB** Элемент of **hype** (from hyperbole) — **всегда** в речах о награждении,
358 предисловиях к книгам и т.п. Торжественная риторика развита как особый жанр
359 на Западе, особенно во Франции! Всегда воспринимать такие речи **cum**
360 **grano salis**.)
361
362 В дальнейшем:
363 Dahl участвовал в разработке методов верификации/структурного прог-я (Дал,
364 Дейкстра, Хоор, Структурное программирование). "Упертый тип" подчеркивал
365 "строгие методы".
366 Nygaard — левый либерал, был активен в политике вообще, связан с
367 профсоюзами, был озабочен вовлечением работников в управление
368 ("скандинавский социализм"), считал, что инфо технологии тут сильно помогут,
369 пропагандировал в профсоюзах широкое обучение инфо технологиям;
370 фактически предтеча нашего А.П.Ершова с его "Программирование — вторая
371 грамотность".
372
373 **0) Очень естеств. расширение** процедурно-модульного прог-
374 я (объект=запись + поведение).
375 **1) Весьма полезно, но не настолько, как часто думают.**
376 **2) По-настоящему работает только при наличии автоматич.**
377 **упр. памятью.**
378
379 Займемся пристальным рассматриванием (очень похоже на Nygaard, Dahl).
380 В модуле, реализующем работу со списком:

```

381
382 TYPE
383   Link* = POINTER TO LinkDesc;
384   LinkDesc* = RECORD
385     next:- Link; (* менять только в данном модуле *)
386     x:- REAL;
387   END;

388 VAR first:- Link

389 PROCEDURE Push* ( new: Link ); (* раньше было x: REAL *)
390 BEGIN
391   ASSERT( new.next = NIL, 20 );
392   (* вставка звена new в начало списка: *)
393   new.next := first;
394   first := new
395 END Push;

396 NB Поставили x после next >>> смещение поля next относительно начала
397 записи LinkDesc не зависит от типа поля x >>> код, работающий с next,
398 совершенно не изменится, если x: REAL заменить на x: INTEGER или x: ARRAY 3
399 OF REAL и т.п.

400 Мы специально иллюстрировали манипуляции со списками так, чтобы было
401 видно, кто сии манипуляции не зависят от содержимого полей после next: Link.
402 Замещение next в contiguous memory block для записи не зависит от прочих
403 полей, стоящих после next.
404 Совершенно неважно, что там дальше в записях и какой
405 они длины.
406
407 Глупо (!?) повторно писать точно такой же модуль для x: INTEGER, раз уж даже
408 после компиляции ничего не изменится.
409 (!?) В простом случае действительно глупо, т.к. с простым списком проще
410 работать непосредственно.
411 Но бывают гораздо более сложные случаи с большим количеством связей
412 и с гораздо большей функциональностью.
413 Поэтому "абстрагируемся" от конкретного содержимого эл-тов списка в модуле,
414 обслуживающем список:
415
416 TYPE
417   Link* = POINTER TO LinkDesc;
418   LinkDesc* = ABSTRACT RECORD (* NB см. о терминологии *)
419     next:- Link
420   END;
421
422 или
423   Link* = POINTER TO ABSTRACT RECORD
424
425 NB Терминология: "абстрактный тип данных" -- до-ОО термин. Есть родство со
426 смыслом ABSTRACT, но не тождественность. Но говорить ABSTRACT-тип
427 неудобно. В контексте ББ и КП -- "абстрактный тип" = тип с атрибутом
428 ABSTRACT. Помнить об этом!
429 полный текст Kurs2006Ex37Lists
430

```

```

431 В модулях-клиентах разрешим доопределять ("расширять") тип Link:

432 MODULE Kurs2006Ex37ListsClient;
433   IMPORT Lists := Kurs2006Ex37Lists, Log := StdLog, In := FVTsysIn (*
434   Epse21SysIn *);

435   TYPE
436     (* типы-потомки *)
437     Link = POINTER TO LinkDesc;
438     LinkDesc = RECORD ( Lists.LinkDesc ) (* в скобках тип-предок *)
439       x: REAL
440     END;

441   PROCEDURE Create;
442     VAR x: REAL; l: Link;
443   BEGIN
444     Lists.Clear;
445     In.Open; ASSERT( In.Done, 20 );
446     In.Real( x );
447     WHILE In.Done DO
448       NEW( l ); l.x := x; (* должны, конечно, сами создать; можно
449 сделать "фабричную процедуру" *)
450       Lists.Push( l ); (* "Lists.Link" := "Client.Link" *)
451       In.Real( x )
452     END
453   END Create;

454   PROCEDURE PrintLog;
455     VAR l: Lists.Link; ll: Link;
456   BEGIN
457     IF Lists.first = NIL THEN
458       Log.String('список пуст'); Log.Ln
459     ELSE
460       l := Lists.first;
461       WHILE l # NIL DO
462         ll := l( Link ); (* l = NIL --> Trap: "NIL dereference" *)
463         Log.Real( ll.x ); (* Log.Real( l( Link ).x ) *)
464         l := l.next
465       END
466     END;
467     Log.Ln
468   END PrintLog;

469   PROCEDURE Do*;
470   BEGIN Create; PrintLog
471   END Do;

472 END !Kurs2006Ex37ListsClient.Do 1 2 3 4 5 6 7 !
473 compiling "Kurs2006Ex37ListsClient"
474   new symbol file 252 0
475   7.0 6.0 5.0 4.0 3.0 2.0 1.0
476

```

477 Говорим: типы-потомки **наследуют (inherit from)** своим типам-предкам.
 478 Механизм называется **наследование (inheritance)**.
 479 **NB Терминология:** метафора сия — неудачная.
 480 Type extension = Расширение типов.
 481 Впрочем, в теор. множественном смысле — наоборот, сужение.
 482
 483 **NB Математическая аналогия:** "группа" -- ABSTRACT, и любая группа должна
 484 иметь перечисленный в аксиомах групп набор свойств. Конкретные группы
 485 (группа целых чисел относительно сложения; группа матриц 3x3 с ненулевым
 486 детерминантом, относительно умножения, и т.п.) имеют и другие свойства.
 487 **NB** Конструкция мотивирована общими логическими "ходами" конструктивного
 488 мышления: отвлекаемся от несущественных в данном контексте деталей,
 489 работаем на основе только заявленных свойств.
 490 **NB Стандартная ошибка:** "Дайте посмотреть, что там у модуля ундре!"
 491 **NB** Как и с процедурами-функциями: не ставится задача "общего
 492 моделирования" данного понятия: держимся специфики конкретной задачи --
 493 программирования.
 494
 495 **NB** В разных, независимо написанных модулях, могут быть разные
 496 конкретизации абстрактного типа L.Link -- автоматическое, необходимое
 497 следствие.
 498
 499 **NB** Lists.Push(I) подразумевает **присваивание** параметру:
 500 VAR I: ***Client.Link; I0: Lists.Link;
 501 I0 := I;
 502 иначе с расширенными типами ничего сделать нельзя будет с помощью "общих"
 503 процедур.
 504
 505 Поэтому общее правило ("**полиморфизм**") : указательная переменная
 506 типа-предка может реально указывать на запись *любого* ("poly-") типа-потомка.
 507 **NB** VAR/IN/OUT параметры — это фактически указательные переменные.
 508 Поэтому для них такие же правила.
 509
 510 Пусть
 511 VAR I: Lists.Link; (* абстрактный, общий *)
 512 II: Link; (* конкретный вариант *)
 513 Тогда
 514 I := II; — ok, дальнейшие манипуляции с I должны быть применимы к любым
 515 конкретным вариантам;
 516 II := I; — ~ok (ошибка компиляции), т.к. I мог ссылаться на другой
 517 конкретный вариант;
 518
 519 "**Статический тип**" переменной: то, что в объявлении (см. PrintLog).
 520 "**Динамический тип**" — тип той фактической записи, на которую в данный
 521 момент ссылается указатель.
 522
 523 Мы можем заранее знать, что в данной точке I ссылается на здешний
 524 конкретный Link, тогда:
 525 II := I(**Link**); — охрана типа/type guard;
 526 компилятор такое пропустит, но вставит проверку типа I, и случится
 527 **TRAP**, если динамический тип I на момент присваивания **не** Link.
 528

529 Оператор/инструкция WITH

530 Для явной проверки типа подобно CASE:
 531 WITH папа: Внук DO
 532 (* **NB** здесь папа интерпретируется как имеющий стат. тип Внук *)
 533 | папа: Сын DO
 534 ...
 535 | папа: Папа DO
 536 ...
 537 [ELSE
 538 ...]
 539 END;

540 В отсутствие ELSE облом, если управление попадает туда.

541 **NB** Почему облом? Для безопасности.

542 **Q** Каков смысл проверки папа: Сын, если Сын -- ABSTRACT?

543 **NB** Механизм расширения типов заменяет механизм вариантных
 544 записей старого паскаля.

545 Но: вариантные записи не позволяют организовать сбор мусора, а расширение
 546 типов -- позволяет.

548 Как ББ узнает, на что ссылается указатель?

549 TYPE
 550 **Папа** = POINTER TO ABSTRACT RECORD END;
 551 **Сын** = POINTER TO ABSTRACT RECORD (ПапаDesc) END;
 552 **Внук** = POINTER TO RECORD (СынDesc) END;

553 внук := папа(Внук);

554 По-грубому:

555 — для каждого типа — дескриптор типа: скрытая запись специального вида:

556 TypeDescriptor = RECORD
 557 name: ARRAY ... OF CHAR;
 558 предок: POINTER TO TypeDescriptor;
 559 ...
 560 END;

561 — для всех типов, появляющихся в модуле, ссылки на дескрипторы среди
 562 глобальных переменных модуля:

563 HIDDENVAR дескПапа, дескСын, дескВнук: POINTER TO TypeDescriptor;

564 — у каждой записи на

565 куче — скрытое поле-

566 ссылка на дескриптор

567 (.td: POINTER TO

568 TypeDescriptor);

569 SIZE(Link) его **не**

570 учитывает (т.к.

571 implementation

572 dependent = зависит от

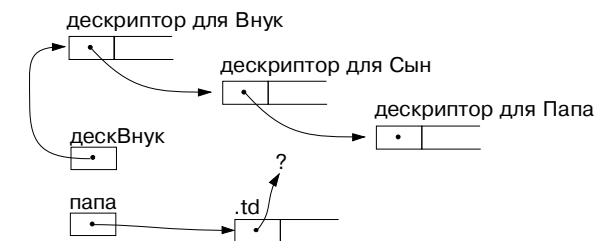
573 реализации [напр.,

574 32/64 бита]).

575

576 Достаточно "пробежать" по цепочке ссылок на дескрипторы, чтобы определить,

577 есть ли там ссылка на нужный дескриптор:



```

578   t := папа.тd; (* TRAP if папа = NIL — NIL dereference *)
579   WHILE ( t # NIL ) & ~( t = дескВнук ) DO
580     t := t.предок
581   END;
582   IF t # NIL THEN
583     (* в цепочке ссылок нашелся дескВнук;
584       продолжать копирование адреса *)
585   ELSE
586     HALT(...)
587   END;

588 NB Реализация может быть и другой (дескриптор хранит ссылки на всю цепочку
589 ссылок; компилятор видит уровень расширения; может сразу найти и
590 сравниться с нужным дескриптором, без поиска по списку).
591
592 NB Скрытое поле отсутствует, если записи собраны в массив (даже если массив
593 на куче) — хитрости реализации; надо думать, скрытое поле в дескрипторе
594 массива ...
595
596 Еще пример (немножко silly):
597 Полезный, т.к. понятный. Но некий человек, поняв здесь, перестал понимать
598 версию со списком. Тем более полезно!
599 Арифметика комплексных. Вот как будет снаружи:

600 DEFINITION Kurs2006Ex38Complex2;

601   TYPE Complex = POINTER TO ComplexDesc;

602   PROCEDURE NewC ( x, y: REAL ): Complex;
603   PROCEDURE NewP ( rho, phi: REAL ): Complex;
604   PROCEDURE Prod ( a, b: Complex ): Complex;
605   PROCEDURE Sum ( a, b: Complex ): Complex;

606 END Kurs2006Ex38Complex2.

607 Две фабричные функции -- для удобства.
608 Могли бы реализовать простейшим образом: ComplexDesc = RECORD x, y: REAL
609 END;
610 Предположим, что написали приложения, проанализировали типичный ход
611 вычислений, и увидели, что полезно менять представление результата (x, y или
612 ρ, φ) в зависимости от операндов и операции. Например, результат Prod --
613 всегда в полярном представлении, а Sum -- в декартовом.
614 Тогда, не меняя приложений, можно заменить модуль так:
615
616 MODULE Kurs2006Ex38Complex2;
617   IMPORT Log := StdLog, Math;

618   TYPE
619     Complex* = POINTER TO ComplexDesc;
620     ComplexDesc = ABSTRACT RECORD END;

621     ComplexC = POINTER TO ComplexCDesc; (* конкретный тип *)
622     ComplexCDesc = RECORD ( ComplexDesc )
623       x, y: REAL
624     END;

```

```

625     ComplexP = POINTER TO ComplexPDesc; (* конкретный тип *)
626     ComplexPDesc = RECORD ( ComplexDesc )
627       rho, phi: REAL
628     END;

629 PROCEDURE NewC* ( x, y: REAL ): Complex;
630   VAR c: ComplexC;
631 BEGIN
632   NEW( c ); c.x := x; c.y := y; RETURN c
633 END NewC;

634 PROCEDURE NewP* ( rho, phi: REAL ): Complex;
635   VAR p: ComplexP;
636 BEGIN
637   NEW( p ); p.rho := rho; p.phi := phi; RETURN p
638 END NewP;

639 PROCEDURE ConvertToC ( a: ComplexP ): ComplexC;
640   VAR res: ComplexC;
641 BEGIN
642   NEW( res );
643   res.x := a.rho * Math.Cos( a.phi );
644   res.y := a.rho * Math.Sin( a.phi );
645   RETURN res
646 END ConvertToC;

647 PROCEDURE ConvertToP ( a: ComplexC ): ComplexP;
648   VAR res: ComplexP;
649 BEGIN
650   NEW( res );
651   res.rho := Math.Sqrt( a.x * a.x + a.y * a.y );
652   res.phi := Math.ArcTan2( a.y, a.x );
653   RETURN res
654 END ConvertToP;

655 PROCEDURE Sum* ( a, b: Complex ): Complex;
656   VAR ac, bc, res: ComplexC;
657 BEGIN
658   WITH a: ComplexC DO
659     ac := a
660   | a: ComplexP DO
661     ac := ConvertToC( a )
662   END;

663   WITH b: ComplexC DO
664     bc := b
665   | b: ComplexP DO
666     bc := ConvertToC( b )
667   END;

668   NEW( res );
669   res.x := ac.x + bc.x;
670   res.y := ac.y + bc.y;
671   RETURN res
672 END Sum;

```

```

673  PROCEDURE Prod* ( a, b: Complex ): Complex;
674      VAR ap, bp, res: ComplexP;
675  BEGIN
676      HALT(126);
677      RETURN res
678  END Prod;
679  END Kurs2006Ex38Complex2.
680  NB Здесь, конечно, в обоих случаях два поля REAL, только интерпретируются
681  по-разному, но даже здесь польза — контроль представления: ошибиться
682  невозможно (почти Q?).
683  NB Идея данного модуля может быть более разумной, если операции
684  реализовывать по стековой дисциплине.
685  Упр Дописать Prod.
686  Упр Избавиться от NEW в Convert (обращение к куче without good reasons в
687  вычислительных задачах будет немного "тормозить"; зависит от того, сколько
688  вычислений per allocation).

689  NB Возможность подменить модуль -- благодаря изначальному
690  сокрытию всех деталей.
691  NB Не так просто решить, что нужно экспортировать, а что нет.
692  В условиях неопределенности и агрессивной среды не
693  раскрываться!
694  Тщательно обдумывать, что открывать.
695  Если открывать, то по минимуму (- вместо *).
696  (Как в бизнесе... как в жизни.)

697  ОО слэнг
698  класс = тип ...;
699  объект (класса) = запись (типа), обычно на куче;
700  О/КП допускают объекты на стеке (Java -- нет), что полезно для эффективности,
701  особ. во встроенных системах (sic);
702  метод = процедура, ассоциированная с типом.
703
704  Синтаксическое удобство
705  Если только на куче, то имена *Desc нигде не будут встречаться, и тогда КП
706  разрешает такое удобство:
707      Complex* = POINTER TO ABSTRACT RECORD END;
708      ComplexC = POINTER TO RECORD ( Complex )
709          x, y: REAL
710      END;
711      ComplexP = POINTER TO RECORD ( Complex )
712          rho, phi: REAL
713      END;
714  NB Тогда для внутренних нужд ББ заведет имена для типов записей:
715  "ComplexC^", "ComplexP^". Об этом полезно помнить (в дальнейшем средства
716  модуля Meta).
717

```

718 Методы

```

719  Mod y Complex: вычисляется по-разному, в зависимости от представления.
720  Можно написать Mod( c: Complex ): REAL; и внутри проверки. Но этого
721  недостаточно.
722
723  Процедуры, ассоциированные с типом:
724
725      PROCEDURE ( c: Complex ) Mod* (): REAL, NEW, ABSTRACT;
726
727  и затем
728
729      PROCEDURE ( c: ComplexC ) Mod (): REAL;
730      BEGIN RETURN Math.Sqrt( c.x * c.x + c.y * c.y )
731      END Mod;

732      PROCEDURE ( c: ComplexP ) Mod (): REAL;
733      BEGIN RETURN c.rho
734      END Mod;

735  Варианты
736      PROCEDURE ( VAR c: ComplexDesc ) Mod* (): REAL, NEW, ABSTRACT;
737
738      PROCEDURE ( VAR c: ComplexCDesc ) Mod (): REAL; ...

739      PROCEDURE ( VAR c: ComplexPDesc ) Mod (): REAL; ...

739  Смысл ABSTRACT: процедура должна быть реализована в конкретном типе-
740  потомке.
741  Смысл NEW: чтобы из-за ошибки в имени процедуры (нередко при
742  переделках ...) не разрушить иерархию наследования.
743  NB Переопределяющие методы должны иметь точно такие же сигнатуры, за
744  исключением функций, рез-т которых может быть сужен ("ковариантность").
745  Как реализован выбор: В дескрипторе типа есть массив процедурных
746  переменных. Место каждого метода фиксировано порядком определения (NEW).
747  Если метод в типе-потомке не переопределяется, то копируется указатель из
748  дескриптора типа-предка. Иначе туда вписывается новый указатель.
749  -----
750  NB Все сие называется single inheritance — наследование от одного предка.
751  Множественное наследование (multiple inheritance) в О/КП не реализовано
752  (80/20).

753  Упр Обдумать, к каким сложным алгоритмам и структурам данных это приведет.

```