

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

Физфак МГУ. 2006/7 уч. год.

## Лекция 9

### Вместо метафизики

Из письма Л.Н.Чернышова (МАИ; компиляторы ...)

"... В дискуссиях о языках программирования я привожу цитату из недавно вышедшей книги Стефана К.Дьюхэрста "Скользящие места C++. Как избежать проблем при проектировании и компиляции ваших программ".

Цитата из предисловия:

"Разумеется, вовсе не обязательно - и даже не рекомендуется - читать эту книгу подряд, от "Совета 1" к "Совету 99". Прием лекарства в таких дозах может навеки отвратить вас от программирования на C++."

**Я всем рекомендую читать эту книгу именно подряд. ..."**

**Задача\*\*\*** Даны VAR x, y: INTEGER; Выяснить, будет ли при сложении иметь место переполнение.

К следующему разу попытаться сделать.

**Была задача:** генерить пары простых чисел, отличающихся на 2 ("близнецы"). Смотрим **решение**.

MODULE Kurs2006Example33; (\* Иллюстрация к теме "генератор": генератор пар простых чисел, отличающихся на 2.

**NB** два примера фильтра:

числа --> подходящие числа (простые);

пары простых --> подходящие пары простых (отличающиеся на 2).

\*)

IMPORT Log := StdLog, Math, In := FVTSysIn (\* Epse21SysIn \*);

TYPE

Primes = RECORD

n: INTEGER;

eos: BOOLEAN;

END;

Pairs = RECORD

p: Primes;

n0, n1: INTEGER;

eos: BOOLEAN

END;

Близнецы = RECORD

p: Pairs;

n0, n1: INTEGER;

eos: BOOLEAN

END;

VAR близ: Близнецы;

```
45 PROCEDURE IsPrime ( n: INTEGER ): BOOLEAN;
46   VAR кандидат: INTEGER; корень: REAL;
47 BEGIN  ASSERT( n > 1, 20 );
48   корень := Math.Sqrt( n ); (* любой делитель не превосходит сего *)
49   (* поиск делителя: *)
50   кандидат := 2;
51   WHILE (кандидат <= корень) & ~( n MOD кандидат = 0 ) DO
52     кандидат := кандидат + 1
53   END;
54   RETURN ~(кандидат <= корень) (* ~"нашли" *)
55 END IsPrime;
```

```
56 PROCEDURE ( VAR p: Primes ) Search, NEW;
57 BEGIN
58   WHILE ( p.n < MAX( INTEGER ) ) & ~IsPrime( p.n ) DO
59     INC( p.n )
60   END;
61   IF p.n < MAX( INTEGER ) THEN
62     p.eos := FALSE
63   ELSE
64     p.eos := ~IsPrime( p.n )
65   END
66 END Search;
```

```
67 PROCEDURE ( VAR p: Primes ) Init ( n: INTEGER ), NEW;
68 BEGIN  ASSERT( n > 1, 20 );
69   p.n := n;
70   p.Search;
71 END Init;
```

```
72 PROCEDURE ( VAR p: Primes ) Next, NEW;
73 BEGIN  ASSERT( ~p.eos, 20 );
74   IF p.n < MAX( INTEGER ) THEN
75     INC( p.n );
76     p.Search
77   ELSE
78     p.eos := TRUE
79   END;
80 END Next;
```

```
81 PROCEDURE ( VAR p: Pairs ) Init ( n: INTEGER ), NEW;
82 BEGIN  ASSERT( n > 1, 20 );
83   p.p.Init( n );
84   p.eos := p.p.eos;
85   IF ~p.eos THEN
86     p.n0 := p.p.n;
87     p.p.Next;
88     p.eos := p.p.eos;
89     IF ~p.eos THEN
90       p.n1 := p.p.n
91     END
92   END
93 END Init;
```

```

95  PROCEDURE ( VAR p: Pairs ) Next, NEW;
96  BEGIN  ASSERT( ~p.eos, 20 );
97      p.p.Next;
98      p.eos := p.p.eos;
99      p.n0 := p.n1;
100     p.n1 := p.p.n; (* новое значение *)
101  END Next;
102
103  PROCEDURE ( VAR 6: Близнецы ) Invariant (): BOOLEAN, NEW;
104  BEGIN
105      RETURN 6.eos OR ( IsPrime( 6.n0 ) & IsPrime( 6.n1 ) & ( 6.n1 = 6.n0 +
106  2 ) )
107  END Invariant;
108
109  PROCEDURE ( VAR 6: Близнецы ) Search, NEW;
110  BEGIN
111      WHILE ~6.p.eos & ~( 6.p.n1 = 6.p.n0 + 2 ) DO
112          6.p.Next
113      END;
114      IF ~6.p.eos THEN
115          6.n1 := 6.p.n1;
116          6.n0 := 6.p.n0;
117          6.eos := FALSE
118      ELSE
119          6.eos := TRUE
120      END
121  END Search;
122
123  PROCEDURE ( VAR 6: Близнецы ) Init ( n: INTEGER ), NEW;
124  BEGIN
125      6.p.Init( n );
126      6.Search;
127      ASSERT( 6.Invariant(), 60 ); (* всегда можно выбросить *)
128  END Init;
129
130  PROCEDURE ( VAR 6: Близнецы ) Next, NEW;
131  BEGIN  ASSERT( ~6.eos, 20 );
132      6.p.Next;
133      6.Search;
134      ASSERT( 6.Invariant(), 60 ); (* всегда можно выбросить *)
135  END Next;
136
137  PROCEDURE Init* ( n: INTEGER );
138  BEGIN  ASSERT( n > 1, 20 );
139      близ. Init( n ); (* находим первую такую пару >= n, либо близ.eos =
140  TRUE *)
141      (* исключительно для иллюстрации: *)
142      ASSERT( близ.Invariant() );
143  END Init;
144
145  PROCEDURE Next* ( i: INTEGER );
146  BEGIN
147      WHILE ( i > 0 ) & ~близ.eos DO  ASSERT( близ.Invariant() );
148          Log.Int( близ.n0 ); Log.Int( близ.n1 ); Log.Ln;

```

```

145      DEC( i ); близ.Next;
146  END;
147  (* простой цикл — т.к. [близ.]Init подразумевает близ.Next *)
148  END Next;
149  BEGIN близ.Init( 2 ) (* страховка *)
150  END Kurs2006Example33.
151
152  ! "Kurs2006Example33.Init(2)" --искать будем, начиная с 2
153  ! "Kurs2006Example33.Next(10)" --найти десять очередных пар
154  ! "Kurs2006Example33.Init(2147480897)" --искать будем, начиная с этого
155  ! "Kurs2006Example33.Init(2147480647)" --MAX(INTEGER) = 2147483647
156
157  3 5
158  5 7
159  11 13
160  17 19
161  29 31
162  41 43
163  59 61
164  71 73
165  101 103
166  107 109
167  2147480897 2147480899
168  2147480969 2147480971
169  2147481899 2147481901
170  2147482091 2147482093
171  2147482661 2147482663
172  2147482817 2147482819
173  2147482949 2147482951
174
175  Правильно сработало на первом запуске.
176  Как было сделано (вещи простые, но описывать на бумаге трудно, и читать-
177  понимать трудно... общая проблема в обучении прог-ю).
178
179  Во-первых, сразу можно написать процедуру определения простоты числа, т.к.
180  она полностью автономна (IsPrime). Не останавливаемся (--> модуль
181  ПошаговаяПростые).
182
183  NB Решаем задачу с конца, с цели (принцип Дейкстры; "программирование
184  есть целенаправленная деятельность").
185  Размышляем о задаче. Простое и универсальное решение — генератор (мы же
186  грамотные программисты).
187  Для любого генератора:
188  инициализация?
189  переход к очередному эл-ту?
190  сигнал окончания?
191  доступ к очередному эл-ту?
192
193  Организуем в виде типа Близнецы, нужно решить эти вопросы (усилием
194  разумной воли).
195  (NB Хотя рассказ долгий, но само дело быстрое, лишь бы в голове ясность...)
196
197  Сразу пишем программки — use cases (примеры использования).
198  Принимаем решения:

```

```

193 Инициализация: начать генерить с первой пары, у которой меньший элемент
194 не меньше заданного числа:
195   PROCEDURE ( VAR 6: Близнецы ) Init ( n: INTEGER );
196   нужно проверять n, пусть n > 1.
197 Сигнал окончания:
198   Когда? квази-естественная граница MAX(INTEGER).
199   Как? поле eos: BOOLEAN;
200       а могли бы и
201       функцию Близнецы.Eos(): BOOLEAN;
202       или через OUT параметр в Next:
203       Близнецы.Next( OUT eos: BOOLEAN );
204 Доступ к очередной паре:
205   поля n0, n1: INTEGER;
206   или вариант: GetTweens( OUT n0, n1: INTEGER );
207 Переход к очередной паре:
208   Близнецы.Next;
209   или вариант:
210   Next( OUT n0, n1: INTEGER; OUT eos: BOOLEAN );
211
212 Здесь уже можно фиксировать синтаксис:
213   TYPE
214     Близнецы = RECORD
215       n0, n1: INTEGER;
216       eos: BOOLEAN
217     END;
218
219   PROCEDURE ( VAR 6: Близнецы ) Init ( n: INTEGER ), NEW;
220   BEGIN
221     ASSERT( n > 1, 20 );
222     HALT(126)
223   END Init;
224
225   PROCEDURE ( VAR 6: Близнецы ) Next, NEW;
226   BEGIN
227     HALT(126)
228   END Next;
229
230 Сравним: In.Open; In.Done; In.Int( OUT n: INTEGER );
231
232 Теперь фиксируем семантику (разделение условное).
233 Разрешенные последовательности использования.
234 0) ясно, что до вызова Init все поля неопределены (NB проблема!).
235 Поэтому для Init условия поставить невозможно (а иногда хотелось бы... напр.,
236 запретить повторный Init)
237 1) ясно, что после Next:
238   б.eos OR ( IsPrime( б.n0 ) & IsPrime( б.n1 ) & ( б.n1 = б.n0 + 2 ) )
239   причем: б.eos — охрана (guard) для доступа к б.n*.
240 сразу оформляем в виде функции
241   PROCEDURE ( VAR 6: Близнецы ) Invariant (): BOOLEAN, NEW; ...
242   Назвали Invariant, т.к.
243   NB генератор — брат цикла, там тоже есть "инвариант", который мы еще не
244   смотрели.
245   2) что после Init, но до Next?
246   если как в In, то б.eos = FALSE, но поля неопределены, т.е. инвариант-
247   охрана не выполняется.

```

```

243 Для единообразия требуем, чтобы и после Init.
244 По-человечески: Init должен установить поля в первую пару близнецов -- или
245 сообщить eos.
246 NB полное единообразие. Хорошо! т.е. легко думать.
247 3) Будем ли запрещать вызывать Next после eos? Видимо, это будет у клиента
248 ошибка. Другого не видим — запретим -- пусть пораньше узнает о своей ошибке.
249
250 Итого:
251   PROCEDURE ( VAR 6: Близнецы ) Invariant (): BOOLEAN, NEW;
252   BEGIN
253     RETURN б.eos OR ( IsPrime( б.n0 ) & IsPrime( б.n1 ) & ( б.n1 = б.n0 +
254     2 ) )
255   END Invariant;
256
257   PROCEDURE ( VAR 6: Близнецы ) Init ( n: INTEGER ), NEW;
258   BEGIN
259     ASSERT( n > 1, 20 );
260     HALT(126);
261     ASSERT( б.Invariant(), 60 ); (* выбросить можно всегда *)
262   END Init;
263
264   PROCEDURE ( VAR 6: Близнецы ) Next, NEW;
265   BEGIN
266     ASSERT( ~б.eos, 20 );
267     HALT(126);
268     ASSERT( б.Invariant(), 60 ); (* выбросить можно всегда *)
269   END Next;
270
271 NB Все должно компилироваться!
272
273 Если бы делали "на экспорт", то
274   TYPE
275     Близнецы* = RECORD
276       n0-, n1-: INTEGER;
277       eos-: BOOLEAN
278     END;
279   PROCEDURE IsPrime* ( n: INTEGER ): BOOLEAN;
280   PROCEDURE ( VAR 6: Близнецы ) Init* ( n: INTEGER ), NEW;
281   PROCEDURE ( VAR 6: Близнецы ) Next*, NEW;
282
283 NB Экспортировать Invariant нет нужды.
284
285 Тут же для тестов делаем глобальную переменную типа Близнецы и две
286 команды, выясняем, как всё "компонуется". Если обнаруживаем шероховатости
287 -- исправляем интерфейсы и семантику.
288 NB Единственный слабый пункт: до Init в полях мусор, инвариант не выполнен.
289 В остальном — полная герметичность и гладкость.
290
291 Следующий шаг: наполняем содержимым.
292 Не спеша: сводим к генератору пар простых.
293
294 Кто понял жизнь, спешить не будет
295
296 Назидательное упр Исключить этот шаг, т.е. сразу свести к генератору
297 простых чисел.

```

288 Вводим тип **Pairs** -- точно такой же по внешней организации, но инвариант  
 289 проще.  
 290 Погружаем переменную p: Pairs внутрь записи Близнецы.  
 291 Реализуем Близнецы.Next как обычный линейный поиск в парах, получаемых  
 292 генератором пар.  
 293 Реализуем Init, тоже поиск, замечаем общий код, выносим в общую процедуру  
 294 (**не** экспортируем).

295 Потом реализуем Pairs, сведя его к генератору **Primes**.

296 Реализуем Primes.

297 Особенность: корректная обработка "последних" значений (MAX(INTEGER)).

298 -----

299 **NB** Схема (pattern) интерфейса единообразная, по структуре очень общая.

300 **Упр** Почему в эту схему не укладывается In?

301 -----

302 **NB**

303 Близнецы — **фильтр** для Pairs;

304 Primes — **фильтр** для последовательности целых.

305 **NB** Дважды использован механизм **композиции** — использование  
 306 другого объекта как "кубика", содержащегося целиком внутри строяемого.

307 **NB** Генератор целых выражается непосредственно средствами языка:

308 n := ...; INC( n ); eos --> (квази) n <= MAX(INTEGER)

309 -----

310 **Назидательное упр** Повторить всё с другой семантикой вызовов для всех  
 311 генераторов: требуя (как в In), чтобы обязательно после \*.Init нужно было  
 312 вызывать \*.Next.

313 -----

314 "Послойная утюжка" логики:

315 — вариация на тему "пошаговое уточнение"

316 — не нужен никакой "пошаговый отладчик"

317

## 318 **Динамические записи и указательн. типы**

319 Технически: размещение данных "на куче"

320 (allocation, on the heap, pointer types)

321 До сих пор мы размещали данные **на стеке** (кроме глоб. переменных).

322 Фунд. проблема: далеко не всегда не только заранее не известен объем данных,  
 323 которые надо обработать, но и объем промежуточных данных неизвестен (т.наз.  
 324 комбинаторный взрыв в компьютерной алгебре; оптимальное разбиение  
 325 области в адаптивных алгоритмах многомерного интегрирования, и т.д. и т.п.).

326 Простой пример: модули.

327 Решение: особая область памяти — **куча (heap)**.

328 — В этой области по заказу (псевдо-процедура NEW) в любой момент находится  
 329 и выделяется свободный contiguous memory block требуемого размера.

330 — В программе (т.е. на стеке) выделяется место для "номерка" — "указателя" =  
 331 адреса выделенного блока памяти. Имеется механизм доступа к блоку (к  
 332 "пальту") с помощью этого "номерка".

333 — В отличие от переменных на стеке, переменные на куче не обязательно  
 334 прекращают существование после окончания процедуры, в которой они были

335 созданы ("пальто" продолжает висеть в гардеробе, даже если потерять  
 336 "номерок").

337 — "Номерки"-указатели = особая категория типов переменных, но все соотв.  
 338 переменные имеют один и тот же фиксированный размер, равный машинному  
 339 слову — 32 или 64 бита (размер "номерка" не зависит от размера "пальта").

340 TYPE

341 Complex = **POINTER TO** ComplexDesc;

342 ComplexDesc = RECORD

343 re, im: REAL

344 END;

345 Euclid = **POINTER TO** EuclidDesc;

346 EuclidDesc = ARRAY 3 OF REAL;

347 Пары: тип-база ↔ тип-указатель.

348 (тип-база = "фасон верхней одежды" пальто, куртка, доха...)

349 Суффикс Desc в названии типа-базы — соглашение.

350 В процедуре или на уровне модуля:

351 VAR c, d: Complex; e, f: Euclid;

352 каждая такая переменная занимает 4 байта (или 8 байт на 64-битных CPU).

353 **NB** Каждая такая переменная в начале своего существования скрытно от нас  
 354 инициализируется в (т.е. ей присваивается) специальное значение **NIL** (все  
 355 биты в нуль) — несуществующий адрес ("наш адрес Советский Союз").

356 **Q** Что значит "в начале своего существования"?

357 Как и с любыми типами, можно объявлять записи с полями таких типов, а также  
 358 массивы указателей.

359 Можно даже объявить так:

360 TYPE

361 Link = POINTER TO LinDesc; (\* Link = звено \*)

362 LinkDesc = RECORD

363 x, y: REAL;

364 next: Link (\* next = следующее|ая|... \*)

365 END;

366 Никаких проблем: |x: 8 байтов|y: 8 байтов|next: 4 байта| — всего 20 байтов.

367 Для номерков-указателей допустимы:

368 операции сравнения вида c = d и d # NIL, но не c = e.

369 **NB** сравнивать "номерки" можно, хотя сравнение "польт" запрещено  
 370 присваивания типа c := NIL и c := d, но не c := e.

371 **NB** это копирование "номерка", а не "пальта"

372 **NB** Компилятор знает (благодаря устройству языка) тип каждой переменной  
 373 (включая тип каждого поля, каждой записи и тип элементов каждого массива),  
 374 и поэтому всегда может отследить совместимость типов в такого рода  
 375 сравнениях и присваиваниях.

376 Но как возникает самое первое не-NIL-ное значение у указателей типа Complex?

377 **NEW( c ); ASSERT( c # NIL );**

378 "За кулисами" происходит обращение к одной из процедур модуля Kernel,  
 379 действие которой можно описать сл. *логической моделью*:

380 — в общем случае "куча" представляет собой набор непересекающихся блоков  
 381 сводобной (неиспользуемой) памяти разных размеров; информация об этих

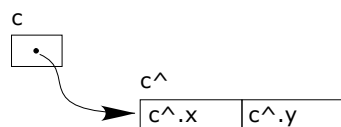
382 блоках (их список с указанием адресов и размеров) хранится, скажем, в неких  
 383 внутренних таблицах;  
 384 — процедура, получив размер записей типа ComplexDesc, ищет на куче  
 385 достаточно большой свободный блок памяти В, в который бы поместилась такая  
 386 запись;  
 387 — "откусывает" от В ровно столько, сколько нужно для ComplexDesc (C);  
 388 — изменяет внутренние таблицы так, чтобы вместо В там фигурировал В'=В\С;  
 389 — записывает адрес "откушенного" куска С в с (VAR-параметр);  
 390 — возвращает управление.

391 На самом деле возможны разные способы организации информации о  
 392 свободных блоках на куче, разные тактики выбора свободных блоков, и т.д. (см.  
 393 Кнут, т.1, разд.2.5). Но общий смысл — как описано выше.

394 **NB** "Возвращать" системе более не нужную память (DISPOSE и т.п.) -- не надо:  
 395 работает **"сбор мусора" = "автоматическое управление памятью"**  
 396 (подробнее ниже).

397 **Правило:** просто не думаем о том, нужно ли освобождать данный кусок памяти.

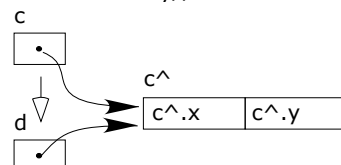
398 Итак, после NEW( с ):



399 **NB** Привыкнуть к таким картинкам.

400 Стрелочка означает, что в с хранится адрес того блока памяти, на который  
 401 стрелочка указывает.

402 После с := d будет



403  
 404 Четко усвоить вышеописанное на концептуальном уровне (пусть не сразу).

407 Итак, сделали NEW( с ). Где-то на куче размещена запись типа ComplexDesc.

408 Как к ней обратиться? — **Нужно имя.** Имя этой записи — **с^** ("с со шляпой").

409 Это имя можно ставить всюду, где требуется переменная-запись  
 410 типа ComplexDesc.

411 Например, с^.x — имя поля-переменной типа REAL и т.д.

412 Допустимо чисто синтаксическое сокращение: **с^.x → с.x** (недоразумений не  
 413 возникает).

414 Аналогично после NEW( е ):

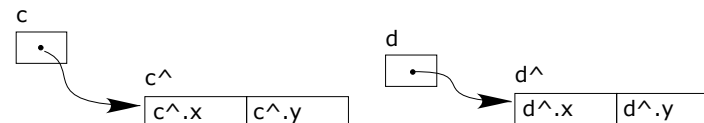
415 **е^** именуется некий EuclidDesc = ARRAY 3 OF REAL, к элементам которого можно  
 416 обращаться как **е^[ i ]** или сокращенно **е[ i ]**.

417 Эта операция называется **разыменование**.

418 Концептуально очень важна (еще бы... "номерок" вместо "пальта" не оденешь),  
 419 хотя на практике чаще всего "сокращается".

420 Применение разыменования — получение другой переменной ("пальта с  
 421 карманами"), для которой исходная была всего лишь "номерком".

422 **Упр** Уяснить смысл присваивания с^ := d^ после NEW( с ); NEW( d ).



423 **NB** При выполнении NEW( с ) все биты блока памяти с^ обнуляются из  
 424 соображений безопасности (Level C2 ...):

425 числовые поля в нули,  
 426 BOOLEAN в FALSE,  
 427 CHAR в 0X,  
 428

429 **все указатели в NIL.**

430 **Неявное разыменование**

431 Вспомним: 2/3 подразумевает преобразование 2 и 3 к типу REAL (меняется  
 432 внутреннее представление в битах) и затем выполнение операции  
 433 вещественного деления, с вещественным результатом.

434 Это называется автоматическое/неявное **приведение типа (cast)**.

435 Вполне конкретная машинная операция с битами. Нередко дается возможность  
 436 и явного приведения, например, ENTIER: REAL → LONGINT и SHORT: LONGING  
 437 → INTEGER. Правда, нет для INTEGER → REAL (в О/КП совсем не нужна).

438 Обероны/Компонентный Паскаль **предельно скупы** на разрешения  
 439 автоматического приведения типов — из соображений безопасности.

440 В С на каждый случай есть неявное преобразование. Откуда жуткие ошибки.

441 Теперь:

442 с^ := d (справа d без шляпы)

443 Здесь автоматическое разыменование в правой части.

444 Равно как и при подстановке d вместо параметра, описанного как ComplexDesc.

445 Тип-база и соотв. указательный тип "ходят парой". Связаны "разыменованием",  
 446 которое работает в одном направлении: от указателя к записи/массиву.

447 **Q** Можно ли получить указатель на произвольную переменную-запись типа  
 448 ComplexDesc? (скажем, объявили на стеке — среди локальных переменных  
 449 процедуры)

450 **A Нет.**

451 Вот:

452 VAR C: ComplexDesc; — объект **размещен/allocated**/аллоцирован **на стеке**.  
 453 VAR c: Complex; ... NEW( с ); — объект размещен **на куче**.

454 **NB** "объект" — это, в сущности, всего лишь запись.

455 **К указателям применимы все правила передачи параметров.**

456 Кроме того, т.к. указательные типы относятся к категории "основных"  
 457 ("элементарных"), то

458 **соотв. значения можно возвращать в качестве результата**  
 459 **функций.**

460 **Упр** Написать модуль комплексной арифметики (алгебры евклидовых векторов,  
 461 алгебры кватернионов) так, чтобы все записи-комплексные числа размещались  
 462 на куче, а все арифметические операции были бы реализованы в виде функций.

463

## 464 Динамические массивы

465 т.е. неизвестной длины (неизвестной статически, т.е. компилятору).

466 TYPE

467 Array = POINTER TO ArrayDesc;

468 ArrayDesc = ARRAY (\*длина отсутствует\*) OF REAL;

469 и затем где-нибудь объявление:

470 VAR a: ARRAY;

471 и затем:

472 **NEW( a, n );**

473 где n — любое выражение типа INTEGER.

474 После этого можно обращаться к массиву a<sup>^</sup>: LEN( a ), a[ i ] := ..., и т.д.

475 Сокращение: если ArrayDesc нигде не используется, то можно написать

476 TYPE Array = POINTER TO ARRAY OF REAL;

477 Если есть, скажем, только пара таких переменных в одной процедуре, то можно вообще написать просто

479 VAR a, b: POINTER TO ARRAY OF REAL;

480 т.е. не вводить имя типа явно.

481 **Упр** TYPE Полином = POINTER TO ARRAY OF INTEGER или REAL;

482  $P(x) = \sum_{i \leq n} c_i x^i \rightarrow c[i]$ . Здесь n = степень полинома.

483 (полином = polynomial; степень = degree)

484 Обсудить варианты представления, реализацию функции "степень".

485 Написать процедуру сложения двух полиномов. Учесть возможность

486 сокращений в двух случаях.

487 Написать процедуру умножения двух полиномов.

488 **Пример** Блэкбоксковский модуль Integers:

489 DEFINITION Integers;

490 TYPE

491 Integer = POINTER TO IntegerDesc;

492 (\* сам IntegerDesc скрыт -- но это POINTER TO ARRAY OF SHORTREAL.

493 Элементы массива -- цифры по основанию 10000 + служеб. информация \*)

494 PROCEDURE Abs (x: Integer): Integer;

495 PROCEDURE Sign (x: Integer): INTEGER;

496 PROCEDURE Compare (x, y: Integer): INTEGER;

497 PROCEDURE Sum (x, y: Integer): Integer;

498 PROCEDURE Difference (x, y: Integer): Integer;

499 PROCEDURE Product (x, y: Integer): Integer;

500 PROCEDURE QuoRem (x, y: Integer; OUT quo, rem: Integer);

501 PROCEDURE Quotient (x, y: Integer): Integer;

502 PROCEDURE Remainder (x, y: Integer): Integer;

503 PROCEDURE GCD (x, y: Integer): Integer;

504 PROCEDURE Power (x: Integer; exp: INTEGER): Integer;

505 PROCEDURE ConvertFromString (IN s: ARRAY OF CHAR; OUT x: Integer);

506 PROCEDURE ConvertToString (x: Integer; OUT s: ARRAY OF CHAR);

507 PROCEDURE Entier (x: REAL): Integer;

508 PROCEDURE Float (x: Integer): REAL;

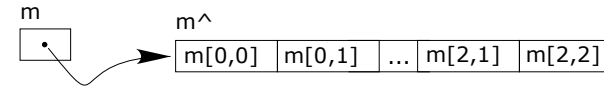
509 PROCEDURE Long (x: LONGINT): Integer;

510 PROCEDURE Short (x: Integer): LONGINT;

511 END Integers.

512 **Двумерный** набор чисел можно описать сл. (существенно разными!) способами:

513 1) TYPE Matrix = POINTER TO ARRAY OF ARRAY OF REAL;



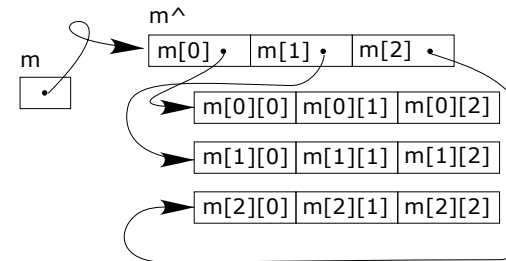
514

515 Создание: **NEW( m, 26, 11 );**

516 2) TYPE

517 Row = POINTER TO ARRAY OF REAL; (\* "ряд" \*)

518 Matrix = POINTER TO ARRAY OF Row;



519

520 **Упр** Для каждого из случаев создать матрицу размерности 10\*6 и заполнить значениями m[i,j] := (-1)^(i+j)

522 **Разница в функциональности**

523 а) В первом случае, чтобы переставить ряды (в вариантах алгоритма Гаусса), нужно копировать все элементы (при размерности 1000, пересылать туда-сюда порядка 3\*1000\*8 байтов).

526 Во втором случае достаточно переставить указатели (пересылка всего порядка 3\*4 байтов).

528 б) Во втором случае все элементы массива Matrix могут иметь разную длину!

529 **Упр** Для каждого из случаев написать процедуру, переставляющую iй и jй ряды.

530 **+Упр** Написать функцию "порядок" и процедуры сложения и умножения для полиномов TYPE Полином = POINTER TO ARRAY OF Integers.Integer;

532 **+Упр Есть:** Во входном потоке дан набор пар вещественных чисел x, y (например, какая-то процедура распечатала в Log, и мы взяли числа оттуда).

534 **Нужно:** Модуль, который умеет "всасывать" такой набор чисел

535 ("инициализация"), а потом, интерпретируя этот набор как узлы кусочно-линейной функции, вычислять значения этой функции для любых (?) x.

537 **Обсудить устно: интерфейсы; структуру данных; реализацию.**

538 **+Упр** То же самое, но интерполяция должна быть с помощью полинома

539 Ньютона (по Калиткину). Обдумать, как добавлять новый узел интерполяции.

540 **Упр\*** То же самое, но с помощью кубических сплайнов.

541 Тут можно было бы поделаться обещанное LU разложение...

## 542 Простые (односвязные линейные)

### 543 СПИСКИ

544 Простые списки **нужно уметь**. В любое время суток, при любой погоде, в  
 545 любом *состоянии* ..  
 546 В лиспе вообще только простые списки. В функц. языках -- основная структура  
 547 данных. Одна из причин, почему считается, что полезно поучиться  
 548 программировать на функц. языках -- чтобы избавиться от "массивно-FOR-ного  
 549 мышления", вколачиваемого устаревшими курсами, и научиться работать со  
 550 списками и думать в соотв. терминах.

551 **Вопрос #1:** когда нужны списки?

552 **Ответ:** Когда не хотим или не можем знать, сколько может быть элементов,  
 553 когда количество, порядок и т.п. все время меняются непредсказуемым образом  
 554 ("**динамическая структура данных**"). Ведь из массива выбросить элемент  
 555 или наоборот, вставить (типа новый узел интерполяции), затруднительно.

```
556 MODULE Kurs2006Example34Lists0;
557   IMPORT StdLog, Math, In := FVTSysIn (* Info21SysIn *);
558   TYPE
559     Link = POINTER TO LinkDesc;
560     LinkDesc = RECORD
561       next: Link;
562       n: INTEGER;
563     END;
564   VAR counter: INTEGER; first: Link;
565   (* сюда будем вставлять процедуры *)
```

```
566 BEGIN counter := 0; first := NIL
567 END Kurs2006Example34Lists0. (* полная версия *)
```

568 После загрузки модуля и выполнения секции BEGIN: first = NIL:

569 **NB** Это КП такой хороший, что все глобалы инициализируются в  
 570 NIL. Поэтому для надежности и понятности — делать явно.

571 Пусть есть такая процедура:

```
572 PROCEDURE Do0*;
573 BEGIN
574   INC( counter );
575   NEW( first ); first.n := counter
576 END Do0;
```

577 При выполнении NEW( first ); где-то ("на куче") будет размещена/allocated  
 578 запись типа LinkDesc (блок памяти соотв.  
 579 размера);

580 а в переменную first будет записан ее адрес:

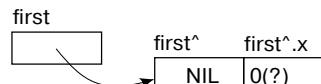
581 **NB** Это КП такой хороший, что все поля  
 582 обнуляются (US Federal Security Requirements  
 583 Level C2). Где-то может быть и не так!

584 Далее

585 first.x := counter; — то же, что и first^.x :=  
 586 counter;

587 Получим (в первый раз):

588



589 Выполним, выделим мышкой имя модуля, Info, Global Variables:

Kurs2006Example34Lists0 [Update](#)

590 .first Kurs2006Example34Lists0.Link [010BEF10H] ◆

591 Видим, что наша новая запись размещена по некоему 16-ричному адресу (у вас  
 592 будет другой).

593 **Упр** Прикинуть десятичное значение.

594 Кликнем по ромбику и увидим:

Kurs2006Example34Lists0.first^ ◆

[010BEF10H] Kurs2006Example34Lists0.LinkDesc ⇨

.next Kurs2006Example34Lists0.Link NIL

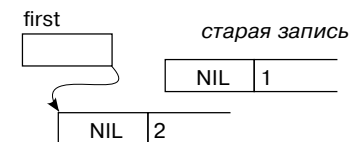
.n INTEGER 1 ⇐

595

596 Почти как на картинке, только информация более подробная.

597 (**NB** Клик по верхнему ромбу на последней картинке возвращает нас на  
 598 предыдущую.)

599 Выполним Do0 еще раз. NEW найдет "на  
 600 куче" незанятый кусок памяти и  
 601 "отдаст" его нам, записав адрес в first.



602 **Упр** Посмотреть, что first действительно  
 603 хранит новый адрес, а поле n — двойку.

604 Куда девается старая запись? Что с ней происходит?

605 Остается на месте. Ничего не происходит.

606 Как к ней получить доступ? (**Q** Что это значит?)

607 Здесь -- никак, она превратилась для нас (и для системы) в "мусор", "garbage" -  
 608 - именно потому, что к ней нет доступа.

609 **Упр** Написать и выполнить несколько раз процедуру DoMany, которая  
 610 выполняет Do0 сто миллионов раз.

611 Проверить значение first.n.

612 Сколько памяти нужно, чтобы разместить все эти записи? А сколько в вашем  
 613 компе RAM'у?

614 Включить Task Manager (Ctrl+Alt+Delete) и, наблюдая графики времени CPU и  
 615 занятой (виртуальной) памяти, снова выполнить DoMany.

616 **Q** Почему Task Manager не показывает рост занятой памяти?

617 **A** Потому что работает "**сбор мусора**" — "**автоматическое**  
 618 **управление памятью**".

619 "Управление памятью" = нахождение свободных блоков памяти на куче, а  
 620 также "узнавание" более не нужных блоков памяти ("мусора") и его утилизация  
 621 -- т.е. перевод в разряд свободных блоков.

622 Стратегия размещения памяти: пока много свободной памяти, просто отдаем  
 623 свободные блоки. Как исчерпали -- ищем "мусор" и утилизируем его.

624 В данном случае: доступ только по first. Все остальное -- "мусор".

625 **Создаем список**

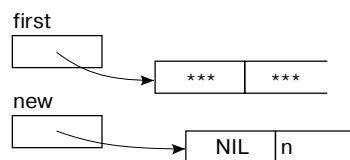
```

626 PROCEDURE Push* ( n: INTEGER );
627   VAR new: Link;
628   BEGIN ASSERT( new = NIL );
629     NEW( new ); new.n := n;
630     new.next := first;
631     first := new;
632   END Push;

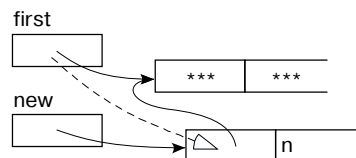
```

633 new: Link; -- 4 байта для адреса в 32-битной машине  
 634 Свежий указатель всегда хранит NIL. (Q Как это достигается?)

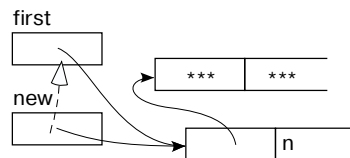
635 После NEW( new ); new.n := n; имеем картинку:  
 636 Звездочки в полях first^ — чтобы подчеркнуть, что их содержимое не будет затронуто дальнейшими действиями.



640 После new.next := first; имеем:



first и next ссылаются, куда и ссылались, но теперь из поля new.next стрелочка

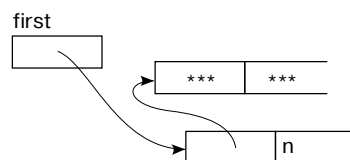


643 ведет к first^.

644 После first := new; получим:

645 Т.е. в first скопируется содержимое new.

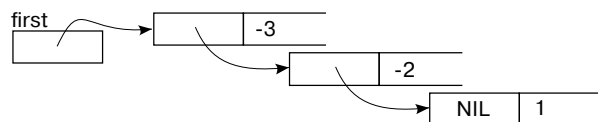
647 После выхода из процедуры new прекратит существование (станет недоступной, содержимое соотв. байтов будет "затерто" другой информацией при любом следующем вызове), и останется такая картина:



652 **NB** Поскольку содержимое поля, отмеченного звездочками, не менялось, там могла быть ссылка на другие записи LinkDesc, которые, в свою очередь... Т.е. можно несколько раз вызывать Push, и цепочка ссылок (список) будет наращиваться.

656 Например, выполним:

657 ① "Kurs2006Example34Lists0.Do0;Kurs2006Example34Lists0.Push(-2);Kurs2006Example34Lists0.Push(-3)"



659 **NB** Понятно, почему называли Push ("затолкнуть") — потому что "заталкиваем" очередное число в список как патрон в магазинную обойму.  
 661 (Push|Pop -- стандартная пара терминов для подобных ситуаций.)

663 Глобальные переменные модуля:

Kurs2006Example34Lists0

[Update](#)

```

        .counter      INTEGER      100000001
664      .first        Kurs2006Example34Lists0.Link [01086C10H]

```

665 Кликнем на синий ромбик справа:

Kurs2006Example34Lists0.first^ ◆

[01086C10H] Kurs2006Example34Lists0.LinkDesc ⇒

```

        .next          Kurs2006Example34Lists0.Link [01086030H]

```

666 ◆

667 Кликнем еще раз:

Kurs2006Example34Lists0.first^.next^ ◆

[01086030H] Kurs2006Example34Lists0.LinkDesc ⇒

```

        .next          Kurs2006Example34Lists0.Link [010824F0H]

```

668 ◆

669 И еще раз:

Kurs2006Example34Lists0.first^(.next^)^2 ◆

[010824F0H] Kurs2006Example34Lists0.LinkDesc ⇒

```

        .next          Kurs2006Example34Lists0.Link NIL
        .n              INTEGER      100000001

```

670

671 Конец списка -- в next стоит NIL.

672 Чтобы вернуться назад, кликать по верхнему ромбику.

673 **Задача для зачета:** Нарисовать картинку списка после выполнения

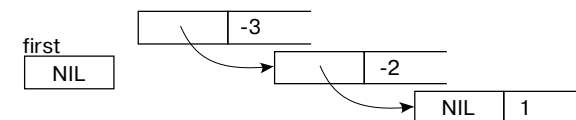
```

674 PROCEDURE Do*;
675   VAR i: INTEGER;
676   BEGIN
677     FOR i := 0 TO 1000 DO Push( i ) END
678   END Do;

```

679 **Как очистить список?**

680 Просто **first := NIL;** и все!



681 **Повторенье — мать ученья**

682 Куда денутся старые размещенные записи? К ним больше нет доступа (если мы никуда ссылок на них не копируем). "Мусор"/garbage. ББ их найдет и "соберет"/collect для повторного использования, когда решит, что памяти стало мало (при каком-то обращении к NEW, возможно, из других модулей ББ — управляющих окнами, текстами и т.п.).

688 **Как заставить ББ собрать мусор сию же секунду?** Команда Services.Collect

689 Т.е. если хотите показать, что вы крутые программисты, поставьте после first := NIL еще и комментарий (\* Services.Collect \*) — типа "я зна-а-аю, что можно форсировать сбор мусора, но закомментил, т.к. я у-у-умный и понима-а-аю, что нужды-то в этом на самом деле нет, ну разве что в каких-то совсем особых случаях".

## Как создать несколько списков одновременно?

Достаточно иметь на каждый список указатель на соотв. первый элемент.  
Процедуру Push удобно изменить, явно указав в параметрах, с какой переменной работаем:

```

698  PROCEDURE Push ( VAR f: Link; x: INTEGER );
699      VAR new: Link;
700  BEGIN
701      NEW( new ); new.x := x;
702      new.next := f;
703      f := new
704  END Push;

```

**Упр** Почему VAR?

**Упр (фильтрация, классификация)** Числа из входного потока собрать в два списка: положительных и отрицательных. Нули игнорировать.

**\*\*\*Проблема** А вдруг f окажется элементом в середине какого-то списка? (Кто ж его, клиента, знает...) **Как защититца от ошибки клиента?** Запомним ...

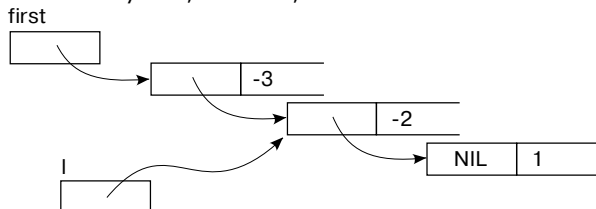
## Цикл полного прохода по списку

```

711  PROCEDURE Log*;
712      VAR l: Link;
713  BEGIN
714      IF first = NIL THEN
715          StdLog.String('список пуст'); StdLog.Ln
716      ELSE
717          l := first;
718          WHILE l # NIL DO
719              StdLog.Real( l.n );
720              l := l.next
721          END
722      END;
723      StdLog.Ln
724  END Log;

```

Здесь l — "бегунок", "runner", "rider".



**Упр** Что происходит на картинке, когда выполняется l := l.next?

Еще раз схема:

```

729      l := <приглашаем первого клиента>;
730      WHILE l # NIL DO <есть кто живой>
731          <следственные действия>
732          l := l.next <кто следующий?>
733      END;

```

**+Упр** Как сработает цикл, если охрану сделать l.next # NIL? Убедиться примером.

**+Упр** Как сработает цикл, если поменять местами два оператора в теле цикла? Убедиться примером.

**+Упр** Сравнить проход по списку с проходом по массиву:

```

739      i := 0;
740      WHILE i < LEN( a ) DO
741          StdLog.Int( a[ i ] );
742          i := i + 1
743      END;

```

**NB** В отличие от массива, по списку можем ходить только в одном направлении.

**Упр** Создать список, считывая числа из входного потока.

Посмотреть список через Info, Global Variables.

Посчитать числа в списке.

Подсчитать кол-во нечетных/отрицательных.

Вычислить арифметическое среднее проходом по списку.

**Пример** Создать копию списка (порядок элементов не важен).

```

752      VAR second: Link; (* глобальная *)
753      ...
754      PROCEDURE MakeCopy*;
755          VAR l: Link;
756      BEGIN
757          second := NIL;
758          l := first;
759          WHILE l # NIL DO
760              Push( second, l.n );
761              l := l.next;
762          END;
763      END MakeCopy;

```

## Вариант: фильтрация

```

767      l := <приглашаем первого пассажира>;
768      WHILE l # NIL DO <есть кто живой>
769          <проверяем паспорт>
770          IF <у пассажира есть виза> THEN
771              <впускаем в страну>
772          ELSE
773              <отправляем обратно>
774          END;
775          l := l.next <кто следующий?>
776      END;

```

**Упр** Скопировать во второй список все четные числа из первого (первый список должен остаться intact; порядок чисел в новом списке не важен).

## Вариант: полный проход по парам

(Ср.: поиск в тексте комбинации "(\*").)

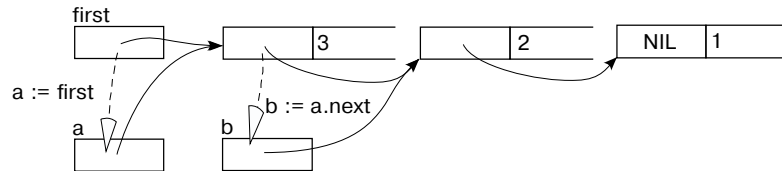
Напр., в списке числа, нужно распечатать все пары, отличающиеся на 2.

**Q** Что значит, "пройти по парам"?

```

783 PROCEDURE DoPairs*;
784   VAR l: Link; a, b: Link;
785 BEGIN
786   IF first = NIL THEN RETURN END; (**)
787
788   (* пара a, b — наш "композиционный бегунок": *)
789   a := first; b := a.next;
790   (* пар больше нет — b = NIL *)
791   WHILE b # NIL DO
792     (* следственные действия: *)
793     IF ABS( a.n - b.n ) = 2 THEN
794       StdLog.Int( a.n );
795       StdLog.Int( b.n );
796       StdLog.Ln
797     END;
798     (* передвигаем бегунок: *)
799     a := b; b := b.next;
800   END;
801 END DoPairs;

```



802  
803

## Линейный поиск в списке

804 **Пример** Ищем семерку:

```

806 PROCEDURE Find7*;
807   VAR l: Link;
808 BEGIN
809   l := first;
810   WHILE ( l # NIL ) & ( l.n # 7 ) DO l := l.next END;
811   IF l # NIL THEN
812     ASSERT( l.n = 7 ); (* сугубо для иллюстрации *)
813     HALT(0); (* чтобы насладиться зрелищем *)
814   ELSE
815     ASSERT( l = NIL ); (* сугубо для иллюстрации *)
816     (* прошли весь список, но нуля не нашли *)
817   END
818 END Find7;

```

819 Общая схема:

```

820   l := <приглашаем первого клиента>;
821   WHILE ( l # NIL ) & ~(отпечаток пальца совпадает) DO
822     (* извиняемся, выдаем справку и т.п.; *)
823     l := l.next (* "Следующий!" *)
824   END;
825   (* обработка окончания: *)

```

```

826   IF l # NIL THEN (* преступник найден: *)
827     (* ASSERT( отпечаток пальца совпадает ) *)
828   ELSE (* преступник не найден *)
829     END;

```

830 **NB** Порядок важен. Отклоняться от схемы **не надо**.  
831 В частности за WHILE всегда IF — или присваивание логич. переменной...

832 **NB Самое главное:** условие поиска — проверка "отпечатка пальца".  
833 **NB** Упорядоченность элементов не важна.

834 **Упр** Понять до конца и выучить наизусть.

835 Обычно "найти" — значит получить указатель на соотв. эл-т:

```

836 PROCEDURE Find7* (l): Link;
837 .....
838   (* в обоих случаях вернем l:
839     l = NIL — удобный "сигнал" неудачи поиска *)
840   RETURN l
841 END Find7;

```

842 В подобном случае обработку окончания цикла можно не делать.

## Найти элемент в заданной позиции

844 Что значит **позиция**? У научных нас всегда **смещение** (кол-во эл-тов до  
845 нужного).

846 "Отпечаток пальца": до искомого элемента в списке стоит pos эл-тов.

```

847 PROCEDURE Find ( pos: INTEGER ): Link;
848   VAR l: Link; offset: INTEGER;
849 BEGIN
850   l := first; offset := 0;
851   WHILE ( l # NIL ) & ~( offset = pos ) DO
852     offset := offset + 1;
853     l := l.next
854   END;
855   RETURN l
856 END Find;

```

857 **NB** Если сомневаетесь ("ошибка +/-1") — проверяйте граничный и  
858 околограничный случаи (pos = 0, 1).

859 **NB Еще раз** Удобно всегда работать со смещениями/offset — поэтому в компьютер.  
860 науке (в отличие от VB) **нумерации с нуля**.

## Поиск последнего элемента:

862 "Отпечаток пальца": l.next = NIL.

```

863 PROCEDURE Last (): Link;
864   VAR l: Link;
865 BEGIN
866   l := first;
867   WHILE ( l # NIL ) & ~( l.next = NIL ) DO l := l.next END;
868   RETURN l
869 END Last;

```