Глава 12. Ссылочные типы и динамические переменные

То, что неясно, следует выяснить. То, что трудно, следует делать с великой настойчивостью.

Конфуций, V век до н.э.

Все типы стандарта Паскаля делятся на три вида: простые (скалярные), сложные (составные, композитные) и ссылочные. Мы с Вами уже изучили простые и сложные типы стандарта Паскаля. Значениями ссылочных типов, как можно догадаться, являются ссылки (грубо говоря, адреса) других переменных Паскаля, а в некоторых языках (и в языке Free Pascal) ссылки можно брать и на подпрограммы.

Вообще говоря, ссылочные типы являются скалярными, т.е. не содержат внутри себя других типов. Однако, в отличие от скалярных, каждый ссылочный тип не является полностью упорядоченным (как говорят математики, не обладает отношением полного порядка). Это означает, что две величины одного ссылочного типа можно сравнивать между собой только на равно и не равно, но нельзя на больше и меньше (так же ведут себя, например, комплексные числа).

Это легко понять, и в обыденной жизни адреса (скажем, ул. Ленина, 25 и ул. Победы, 31) тоже можно ставнивать между собой на равно и неравно, но бессмысленно сравнивать на больше и меньше. Как уже отмечалось, в стандартной реализации на ЭВМ скалярный вещественный тип тоже не является полностью упорядоченным.

В Паскале определена только одна константа ссылочного типа со служебным именем **`nil`**, это ссылка «в никуда», вскоре мы узнаем, зачем она нужна и как применяется. Все ссылочные типы нужно описать перед их использованием. Сначала синтаксис:

<ссылочный тип> ::= ↑ <имя типа>;

Например, `x,y:↑integer;` описывает переменные `x` и `y`, значениями которых будут ссылки на целочисленные переменные. К сожалению, на клавиатуре нет клавиши для ввода символа `↑`, поэтому во всех реализациях Паскаля его заменяют маловыразительным символом `^` («крышечка»). В этой книге мы, однако, будем, вслед за автором Паскаля Н. Виртом, использовать красивый символ `↑`. [1]

Как Вы уже знаете, `↑integer` является безымянным типом, его использование ограничено, например, переменную такого типа нельзя передавать в подпрограммы. Рекомендуется всем ссылочным типам давать имена в разделе типов, например:

```
type Ri=↑integer;
```

В языке Free Pascal для удобства есть *стандартные* ссылочные типы для всех скалярных типов данных, их имена начинаются с буквы `P` (Pointer – ссылка, указатель), например

`Pbyte=↑byte; Pchar=↑char; Pint64=↑int64;` и т.д.

Предусмотрены и ссылки на ссылку, например

`PPbyte=↑Pbyte; PPchar=↑Pchar; PPint64=↑Pint64;` и т.д.

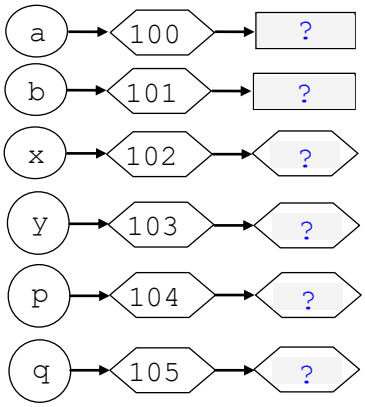
Семантику ссылочного типа изучим на конкретном примере. Пусть есть описания:

```
type Ri=↑integer; RRi=↑Ri;
var a,b: integer; x,y: Ri; p,q: RRi;
```

Итак, у нас описаны два разных ссылочных типа с именами `Ri` и `RRi`. Как Вы уже знаете, встретив раздел переменных, Паскаль-машина начинает порождать эти переменные. Покажем это на картинке ниже слева, в шестиугольниках показаны адреса переменных, а в прямоугольниках – значения этих переменных.

---

[1] Сам Вирт предлагал в качестве альтернативы вместо символа `↑` использовать символ `@`, но вы вышло. В языке Free Pascal символ `@` используется как операция взятия адреса.



Как видим, переменные порождаются в условных ячейках памяти с адресами 100, 101 и т.д., как и положено в стандарте Паскаля, с неопределёнными значения.[1] Ссылочные переменные отличаются от «обычных» только тем, что их значениями являются ссылки, значения которых мы изображаем не в прямоугольниках, а в шестиугольниках.

«Обычные» переменные чаще всего получают значения при выполнения операторов присваивания, например:

```
a:=1; b:=2;
```

Для ссылочных переменных тоже можно применять оператор присваивания, например, переменные `x` и `y` одного типа, следовательно совместимы по присваиванию: `y:=x`, но особого смысла в этом нет, так как переменная `y` сейчас тоже имеет неопределённое значение.

Во многих языках разрешено присваивать ссылочным переменным ссылки (адреса) «обычных» переменных (т.е. статического и автоматического классов памяти), в частности, в языке Free Pascal для этого предназначена одноместная операция взятия ссылки `@`. Тогда можно, например, присвоить `x:=@a` (то же самое в языке C: `x=&a`). В стандарте Паскаля это, однако, запрещено, причину этого мы вскоре обсудим. Впервые ссылки на все классы переменных появились в языке PL/I.

Присвоить значение ссылочной переменной можно и с помощью стандартной процедуры Паскаля с именем `new`, например:

```
new(x)
```

Этот оператор порождает новую безымянную целочисленную переменную динамического класса памяти, ссылка на эту динамическую переменную помещается в параметр `x` процедуры `new`.[2]

В языке Free Pascal можно сделать то же самое более явно:

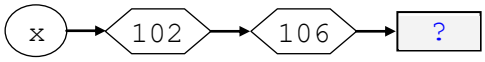
```
GetMem(x,sizeof(x↑))
{ по GetMem(x,4), как и по new(x) будет выделено 16 байт,
  участки меньшей длины в динамической памяти не выделяются }
GetMem(x,100); { x↑ в первых байтах из 7*16=112, зачем? 🙄 }
x:=Alloc(100); { x:=адрес 7*16=112 байт, очищенных нулями }
Reallocate(x,200);
{ сначала выделяются новые 13*16=208 байт, затем «прежние» 112
  байт копируются в новую переменную, потом старая
  переменная уничтожается }
```

Предполагается, что есть достаточный объём свободной памяти, иначе в стандарте Паскаля результат не определён, а в языке Free Pascal будет исключительная ситуация. Можно, однако, присвоить стандартной переменной с длинным, но осмысленным именем значение **`true`**:

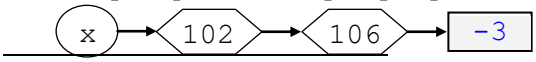
```
ReturnNilIfGrowHeapFails:=true
```

Тогда при нехватки памяти процедуры `new` и `GetMem` будут не вызывать исключительную ситуацию, а возвращать в ссылочной переменной значение **`nil`**.

Например, пусть после `new(x)` для хранения этой целой переменной отведена ячейка с адресом 106, тогда получается такая картинка (порождённая целочисленная динамическая переменная в прямоугольной рамке, у неё неопределённое значение):



Доступ к динамической переменной производится с помощью указывающей на неё ссылочной переменной, для этого в Паскале предназначен знак ↑ разыменование (dereference), т.е. «пойти по ссылке», например, после оператора присваивания `x↑:=-3`, будет:



[1] Как уже говорилось, в языке Free Pascal только для переменных статического класса *числовые* порождаются с нулевыми значениями, а *ссылочные* со значением пустой ссылки **`nil`**.

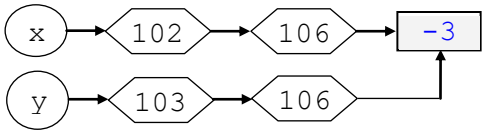
[2] Можно порождать сразу несколько динамических переменных, например, `new(x,y,z,...)`.

Некоторые языки используют для работы с динамическими переменными как ссылки (reference) так и указатели (pointer). И те, и другие хранят адреса, но между ними есть некоторое различие: для указателей явно требуется операция разыменования, а для ссылок эта операция выполняется *автоматически* (производится *неявное* разыменование). Например, для языка C:

```
int X;       // целая переменная X
int *Y=&X;   // Y это указатель на переменную X
int &Z=X;    // Z это ссылка на переменную X
    *Y=3;    // это X:=3, это Паскалевское Y↑:=3
    Z=3;     // это X:=3, разыменование * не нужно
```

Учтите, что при этом ссылочная «переменная» `Z` имеет статус *константы*, её нельзя менять.

Обязательно поймите, что `x↑` является (безымянной и динамической) переменной! А вот теперь, когда (статическая и ссылочная) переменная с именем `x` имеет конкретное значение 106, можно использовать это имя в операторе присваивания, например, после `y:=x` получится такая картинка:



В стиле шпионских романов динамическую переменную можно сравнить с разведчиком, действующим в тылу врага, его (настоящее) имя никто не знает. Местонахождение разведчика известно только его резиденту. Когда связному из Центра надо передать разведчику донесение (записать значение в динамическую переменную) или получить от разведчика сведения (считать значение из динамической переменной), то он приходит к резиденту, который и направляет (`↑` 🙂) связного по конкретному адресу. В нашем последнем примере два «резидента» `x` и `y` знают адрес 106 одного и того же «разведчика». Обратного случая (когда один резидент знает адреса двух и более разведчиков) у нас не бывает. Правда, можно сделать резидента `z` массивом (руководителем агентурной сети́ 🙂):

```
type RiMas=array[1..10] of Ri;
var  z: RiMas;
```

Когда динамическая переменная программисту больше не нужна, можно уничтожить её (освободить занимаемую ей память), для этого в Паскале предназначена стандартная процедура `dispose`:

`dispose(`<ссылочная переменная>`)`

В других языках аналогичная процедура обычно называется более «прямолинейно»: `free` (освободить), `delete` (удалить), `erase` (стереть) и т.д. Н. Вирт выбрал более подходящее по смыслу слово `dispose` (сделать недействительным). Фактически динамическая переменная остаётся на своём месте в памяти и её значение (по крайней мере ещё некоторое время) сохраняется, но ссылка на неё становится *недействительной*, ходить но ней нельзя. Впрочем, в языке Free Pascal (по аналогии с языком C) для уничтожения динамической переменной можно использовать и процедуру `FreeMem`.

В качестве параметра указывается та ссылочная переменная, которая в данный момент связана с уничтожаемой динамической переменной. Память, занимаемая динамической переменной, освобождается, и в дальнейшем может использоваться Вашей программой для размещения новых (вновь порождаемых по процедуре `new`) динамических переменных. Два раза освобождать выделенную область памяти нельзя:

`dispose(x); dispose(x);` { ОШИБКА }

Как обычно, для стандарта Паскаля такое действие не определено, а в языке Free Pascal вызывает «ExitCode `204`. Invalid pointer operation (Неправильная операция с указателем)».

Итак, например, по оператору `dispose(x)` уничтожается динамическая переменная в ячейке 106, а сама ячейка объявляется свободной (как уже говорилось, она включается в список свободной памяти). Учтите, что значение ссылочной переменной `Cx=106` при этом не меняется! Это может приводить к тяжёлым семантическим ошибкам, например:

`dispose(x); ... x↑:=-777`

Здесь производится запись в 106 ячейку, которая может уже быть отдана другой динамической переменной. Эта ошибка носит название висячая ссылка (dangling reference), и программист никак не может проверить, что этот случай имеет место. Более редкая разновидность такой ошибки заключается в повторное освобождение (double free) динамической переменной (т.е. два `dispose` на один `new`). Обычно механизм работы с динамической памятью достаточно хорошо защищён от таких ошибок, но иногда это может вызвать уничтожение полезных данных, поэтому и здесь присвоение ссылки значения **`nil`** (см. далее) решит проблему.

В шпионских романах это случается, когда связной передаёт разведчику приказ, что его срочно отзывают в Центр, а до резидента эта информация почему-то не дошла. И вот, когда к резиденту прибывает очередной связной, резидент «со спокойной совестью» посылает его по адресу, где раньше проживал разведчик. Ну, а там уже гестапо (ну, или ФБР) … 🙁.

Для предотвращения таких ошибок программисту рекомендуется после каждого уничтожения динамической переменной присваивать соответствующей ссылочной переменной значение пустой ссылки **`nil`** (т.е. обязательно сообщать об отзыве разведчика резиденту 🙂). В этом случае при попытке доступа будет происходить ошибка времени выполнения (null reference), например:

```
dispose(x); x:=nil; … x↑:=1; { ОШИБКА }
```

Спрашивается, а почему же в стандарте Паскаля не требуется, чтобы после уничтожения динамической переменной процедура `dispose` сама (автоматически) присваивала соответствующей ссылочной переменной значение **`nil`**? В основном это сделано для того, чтобы не внушать программисту ложное чувство безопасности. Действительно, в нашем примере после этого оператор `x↑:=1` вызовет АВОСТ, а вот оператор `y↑:=1` уже нет, хотя и произошла та же ошибка. Образно говоря, резиденту `x` сообщили об отзыве разведчика, а резиденту `y` забыли. Исходя из этого, программисту рекомендуется после уничтожения динамической переменной тщательно просмотреть свою программу, и самому присвоить **`nil`** всем соответствующим ссылочным переменным, в нашем случае это будет `x:=nil; y:=nil`. Об автоматической борьбе с висячими ссылками [i] [см. сноску в конце главы].

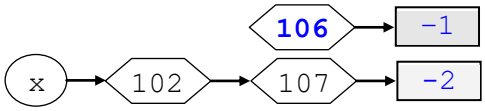
Поймите, что для переменных остальных двух классов памяти (статического и автоматического) такая ошибка доступа к уничтоженной переменной в стандарте Паскаля возникнуть не может. Действительно, переменная статического класса уничтожается только в конце программы. Переменная автоматического класса памяти уничтожается при выходе из подпрограммы, где она описана, однако при этом она становится невидимой, и обратиться к ней нельзя!

Для «менее надёжных» языков (C, C++, Free Pascal и других) можно «ухитриться» обратиться к уничтоженной переменной автоматического класса. В теории программирования это называется *проблемой Фунарга* [см. сноску **`iii`** в конце главы 8].

Рассмотрим теперь другую характерную ошибку при работе с динамическими переменными. Пусть есть операторы

```
new(x); x↑:=-1; … new(x); x↑:=-2;
```

Ниже показана картинка, возникающая при этом в памяти ЭВМ:



Здесь связь с динамической переменной в ячейке 106 потеряна навсегда, доступ к ней невозможен. Говорят, что произошла утечка памяти (memory leak), а сами «потерянные» переменные иногда образно называют переменными-сиротами (orphan variables) или просто мусором (garbage).

В шпионских романах это означает, что для резидента прислали из Центра нового разведчика, а старый разведчик по каким-то причинам приказ о возврате в Центр не получил (связной погиб или что-то ещё…). Судьба такого разведчика печальна, до конца войны (т.е. до окончания программы) не один связной к нему не придёт 🙁.[ii] [см. сноску в конце главы]

В языке Free Pascal для контроля утечки памяти можно использовать стандартную запись с именем `THeapStatus`. В каждый момент времени она содержит необходимую информацию о текущем состоянии динамической памяти. Там много полей, нас будут интересовать только следующие:

```
THeapStatus=record
  TotalAddrSpace: Longword;  { Общий объём доступной памяти }
  TotalFree: Longword;       { Общий объём свободной памяти }
  TotalAllocated: Longword;  { Общий объём выделенной памяти }
  ...
end {THeapStatus};
```

Поле `TotalAddrSpace` само по себе мало информативно, так как Free Pascal берёт у операционной системы динамическую память кусками по 64 Кб по мере необходимости, так что значение этого поля может увеличиваться по мере счёта программы. Самое важное для нас поле `TotalAllocated`, разность значений этого поля в начале и конце работы программы и покажет утечку памяти. Для доступа к записи `THeapStatus` используется стандартная функция без параметров

```
function getHeapStatus: THeapStatus;
```

Как видно, функция выдаёт (безымянную) копию текущего значения записи `THeapStatus`, пример определения утечки динамической памяти:

```
var NByte: Longword;
begin { Начало программы }
  NByte:=getHeapStatus.TotalAllocated;
    { Наш алгоритм }
{ Конец программы }
  NByte:=NByte-getHeapStatus.TotalAllocated;
  Writeln('Утечка памяти=',NByte,' байт')
end.
```

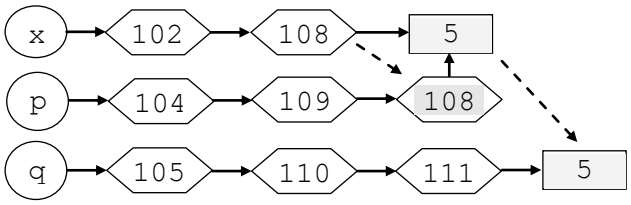
Утечка памяти становится по-настоящему опасной, когда она происходит много раз во время счёта программы (например, в цикле). В этом случае свободная память на компьютере может закончиться, и программа завершиться аварийно (unsufficient memory). Такие ошибки бывает трудно найти, они часто встречаются даже в продуктах известных программистских фирм.

В то же время нет никакого смысла освобождать память непосредственно перед выходом из программы (по **end.**), так как операционная системы сама «отберёт» всю занимаемую программой память, как только программа завершается. Здесь необходимо учесть и такой тонкий момент: даже при освобождении по `dispose` (`free` и т.д.) *больших* участков памяти, они, как правило, не возвращаются операционной системе и, следовательно, не могут быть отданы другим программам 🙁.

Пойдём теперь дальше, рассмотрим результат выполнения операторов:

```
new(x); x↑:=5; new(p); p↑:=x;
new(q); new(q↑); q↑↑:=p↑↑;
```

Обязательно поймите, как присвоены значения всем ссылочным переменным и что здесь у нас три динамических переменных:



Во многих языках программирования значения ссылок можно сравнивать между собой на равно и неравно. В стандарте Паскаля, однако, накладывается ограничение, что на равно и не равно можно сравнивать только ссылки одного типа, например:

`x=y` { или } `p<>q` { Правильно } `x=p` { ОШИБКА! }

С точки зрения здравого смысла это вполне естественно, так как ссылки есть, по сути, адреса. Например, два адреса школ (хранящиеся в `x` и `y`) логично сравнивать между собой на равно и неравно. В то же время адреса разных объектов, скажем, школ и ресторанов (`x` и `p`) сравнивать бессмысленно, они (надеюсь!) никогда не совпадут между собой.

Отметим, что даже «демократичный» язык C, хотя и разрешает сравнивать между собой ссылки разных типов, но объявляет результат такого сравнения неопределённым. Впрочем, ссылки одного типа языки C и Free Pascal позволяют сравнивать также на больше и меньше (по их адресам в памяти, большого смысла это не имеет).

В языке Free Pascal «строгие» ссылки называются типизированными, но предусмотрен и тип не типизированных ссылок (обычно называемых указателями) с типом `Pointer`. Ссылочной переменной типа `Pointer` можно присваивать ссылочное значение любого типа (и наоборот). Когда у операции отношения хотя бы один операнд бестиповый, то адреса можно сравнивать не только на равно и неравно, но и на больше и меньше:

```
type Pi=↑integer; Pc=↑char;
var x: Pi; y: Pc; z: Pointer;[1] b: boolean;
    a: integer; c: char;
begin x:=y; { ОШИБКА } b:=x=y; { ОШИБКА }
      z:=x; b:=z=y; b:=z>y; { Всё хорошо }
```

Как уже упоминалось, в языке Free Pascal есть и операция взятия ссылки @ (и функция взятия адреса `Addr`). Обычно эти операции строго типизированы, например:

```
x:=@a; { Всё хорошо } x:=@c; { ОШИБКА }
```

Можно, однако, выключить этот контроль с помощью директивы `{$T-}`.

Как уже упоминалось, операция взятия @ в стандарте Паскаля запрещена, причину этого легко понять. Во-первых, что делать, если после оператора `x:=@a` вызвать процедуру `dispose(x)` ☺. Во-вторых, если взять адрес у переменной *автоматического* класса памяти, и эта переменная будет уничтожена при выходе из соответствующей подпрограммы, то куда будет указывать ссылочная переменная?

Как и в языке C, со значениями ссылочного типа `Pointer` по умолчанию (при включённом режиме работы `{$PointerMath on}`) можно выполнять арифметические операции сложения и вычитания, например:

```
{$PointerMath on}
type Arr=array[1..8] of smallint; {по 2 байта}
     PArr=↑Arr;
var x: arr=(1,2,3,4,5,6,7,8); y: smallint=-1;
    p: PArr; q: Pointer;
begin
  p:=@x; Write(p↑[1],p↑[5]:2); {x[1]=1,x[5]=5}
  q:=p; p:=p+1; {p+sizeof(x)=p+16; p→x[9] ⚠}
  write(p↑[1]:2); {x[9]=y=-1}
  q:=q+4; {q→x[3]}
  writeln(smallint(q↑):2); {x[3]=3}
  writeln(smallint((q+2)↑):2); {x[4]=4}
  q:=q+1; {q→x[?]=x[3.5]?}
  writeln(smallint(q↑):2); {1024 ⚠}
{ объяснение в курсе по архитектурам ЭВМ }
```

Как видим, такие операции очень опасны и могут вызвать трудно находимые ошибки, поэтому их можно запретить (глобально, для всей программы или всего модуля), поставить в начале директиву `{$PointerMath off}` или `{$PointerMath-}`.

Итак, динамические переменные располагаются где-то в памяти и связываются ссылками, причём сами динамические переменные могут, в свою очередь, быть ссылочными. Теперь поймём, что ссылочная переменная может входить в состав другой переменной, например, быть элементом массива или полем записи. И вот тогда динамические переменные, связанные ссылками, могут образо-

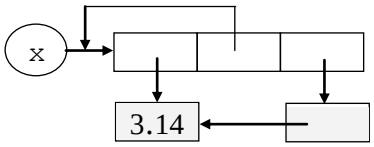
---

[1] В языке C это будет `void *z;`, т.е. указатели языка Free Pascal не являются указателями языка C 😕.

вывать в памяти ЭВМ структуры данных, называемые ориентированными графами, причём любой сложности.

Конкретные адреса расположения переменных в памяти программисту обычно не интересны, они не влияют на алгоритм. В дальнейшем шестиугольники со значением адреса, где расположена переменная, рисовать не будем, а все переменные (числовые и ссылочные) изображать прямоугольниками.

В качестве примера рассмотрим такую задачу. Необходимо на стандарте Паскаля построить показанную ниже структуру данных (как договорились, шестиугольников с адресами больше не рисуем).



Осознание задачи. Стрелки выходят из ссылочных переменных. Переменная из трёх частей, все три поля которой разных (ссылочных) типов (поймите, что типы разные), может быть только записью. Имена полей мы может выбирать произвольно, они локализованы внутри записи и снаружи без селектора-точки не видны. Второе поле ссылается на запись, в которую оно входит, в стандарте Паскаля это может быть только тогда, когда сама запись является динамической переменной (ссылаться на статические и автоматические нельзя). Таким образом, `x` является ссылочной переменной. Первое поле имеет тип ссылки на вещественную переменную, а третье – ссылки на ссылку на вещественную переменную. Приступим к описанию нужных типов и переменных.

```
type RefReal=↑real; RefRefReal=↑RefReal;
     RefRec=↑Rec;
     Rec=record
       a: RefReal; b: RefRec; c: RefRefReal;
     end;
var x: RefRec;
```

Приступим теперь к построению нужной структуры данных, обратите внимание, что запись является безымянной динамической переменной:

```
new(x); new(x↑.a); { Для 3.14 }
new(x↑.c); { Для ссылки на 3.14 }
x↑.a↑:=3.14; x↑.c↑:=x↑.a; [1]
x↑.b:=x; { Ссылка на себя }
```

Необходимо обязательно тщательно изучить этот пример, и убедить себя, что Вы всё понимаете!

Обратите внимание на такую тонкость. Паскаль (и большинство других строго типизированных языков) требует, чтобы каждое имя перед использованием было описано выше по тексту программы. В нашем примере, однако, это сделать невозможно: второе поле записи использует имя `RefRec`, которое описывается ниже по тексту.

Мы уже сталкивались с такой ситуацией при перекрёстной ссылке двух подпрограмм друг на друга, там мы вышли из положения, введя понятие объявления подпрограммы. Здесь такой приём не сработает, и во всех строго типизированных языках, скрипя сердцем 🙁, приняли решения считать это допустимым (но только для ссылочных типов и только внутри одного раздела **`type`** !).

Рассмотрим теперь, почему динамические переменные так важны в программировании. Дело в том, что во многих задачах необходимо обрабатывать входные данные заранее неизвестного объёма. Для таких данных невозможно заранее создать в памяти переменные нужного размера. Можно, однако, по мере необходимости, порождать новые динамические переменные, для хранения вновь введённых (или порождаемых) порций данных. Для хранения такой информации переменного объёма и используются динамические структуры данных, меняющие свой размер в процессе счёта программы. Эти структуры данных мы и начнём изучать в следующей главе.

---

[1] Для продвинутых читателей. В Паскале доступ к полю с именем `a` безымянной записи обозначается как `x↑.a`, в языке C как `x*.a`, в языке C++ как `x -> a`.

В языке Free Pascal работа со ссылочными переменными расширена, так что их возможности практически такие же, как и у указателей языка C. Многим из Вас придётся работать в этом языке и Вы увидите, какой это универсальный, но и очень ненадёжный механизм, лишённый практически всего контроля за своим использованием. Например, в языке Free Pascal определены операции сложения и вычитания ссылочных переменных с константами:

```
type Pi=↑word;
var  x: Pi;
begin
  new(x); { x= ↑ на безымянную переменную }
  x↑:=1;  { послать 1 в эту переменную }
  x:=x+3; { x:=x+3*sizeof(word)
            адрес x увеличивается на 6 байт ⚠ }
  x↑:=2;  { послать 2 «на деревню дедушке» }
```

В качестве ещё одного примера использования динамических переменных рассмотрим задачу, в которой обрабатывается файл комплексных чисел:

```
type Complex=record re,im: real end;
     FComp=file of Complex;
```

В задаче требуется вычислить сумму первого и последнего из комплексных чисел файла. Так как функции в стандарте Паскаля не могут возвращать значения сложных типов, то сначала в голову приходит идея реализовать операцию в виде процедуры:

```
procedure SumFirstLast(var F: FComp; var Rez: Complex);
```

Проводя спецификации задачи решаем для файла из одной компоненты возвращать удвоенное значение этой первой компоненты. Дальше, однако, возникает трудность: что записывать в результат `Rez` для пустого файла? Ведь любое комплексное число может быть «настоящим» ответом. Можно предложить такой выход: реализовать операцию в виде функции, которая возвращает ссылку на результат. В этом случае для пустого файла можно возвращать пустую ссылку:

```
type RefComplex=↑Complex;
function SumFirstLast(var F: FComp): RefComplex;
  var xF,xL: Complex; y: RefComplex;
begin
  Reset(F); y:=nil; { для пустого файла }
  if not eof(F) then begin
    read(F,xF); xL:=xF;
    while not eof(F) do read(F,xL);
    xF.re:=xF.re+xL.re;
    xF.im:=xF.im+xL.im; { First+Last }
    new(y); y↑:=xF
  end;
  SumFirstLast:=y
end;
```

Итак, функция возвращает или пустую ссылку **nil**, или ссылку на (безымянную) динамическую переменную с нужной суммой компонент файла. Разумеется, когда (после возврата из функции) необходимость в этой переменной отпадёт, надо не забыть уничтожить её процедурой `dispose`, например:

```
  var z: RefComplex; x: FComp;
begin
  Assign(x,'MyFile.comp'); z:=SumFirstLast(x);
  if z=nil then write('Файл пуст!') else begin
     write(sqrt(sqr(z↑.re)+sqr(z↑.im))); dispose(z) {⚠}
  end
end.
```

Вопросы и упражнения

Ученик – это не сосуд, который надо наполнить, а факел, который надо зажечь.
Плутарх 46-127 н.э.

1. Чем ссылочные типы данных отличаются от скалярных?
2. Почему в стандарте Паскаля запрещено присваивать ссылочным переменным ссылки на переменные статического и автоматического классов памяти?
3. Что такое «разыменование» и как оно выполняется?
4. Объясните, почему `x↑` является переменной.
5. Почему процедура `dispose` не присваивает своей ссылочной переменной значение **`nil`**?
6. Что такое утечка памяти и чем она опасна?
7. Почему в стандарте Паскаля на равно и не равно нельзя сравнивать значения ссылочных переменных разных типов?
8. Можно ли при описании ссылочного типа данных использовать тип данных, описанный ниже по тексту программы?

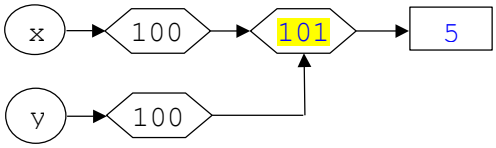
---

[i] Для продвинутых читателей. Во-первых, чтобы немного снизить опасность работы с висячими ссылками, в языки высокого уровня при описании ссылочных переменных (указателей) может вводится классификатор **`restrict`**. Таким образом программист гарантирует компилятору, что на эту область памяти будет ссылаться не более одного указателя («…а я тебе обещаю, что…»). Это позволит компилятору (если он поверит программисту 🙂) сгенерировать более эффективный код. Впрочем, если компилятор увидит, что на одну область памяти явно ссылаются два указателя, он нам всё равно «не поверит». С другой стороны, многие компиляторы позволяют программисту и явно приказать не проводить подобные проверки, т.е. безоговорочно «верить» программисту. Например, для языка C:

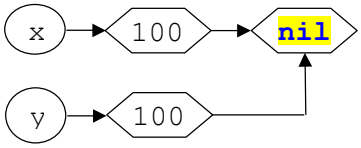
```
int * restrict P;
```

Во-вторых, можно вместе с каждой динамической переменной порождать вспомогательную динамическую переменную, называемую «надгробием» (tombstones) для первой переменной 🙂. Впервые этот механизм контроля висячих ссылок предложил Д. Ломет (D. Lomet) в 1975 году. При этом ссылочная переменная содержит адрес «надгробия», а уже «надгробие» содержит адрес динамической переменной:

```
var x,y: ↑integer;
    newTombstone(x); x↑:=5; y:=x;
```



Процедура `newTombstone` порождает как саму целочисленную динамическую переменную в ячейке `101`, так и её «надгробие» в ячейке `100`. После оператора `y:=x` переменные `x` и `y` (где они сами находятся нам не важно) хранят ссылку `100` на «надгробие». Теперь у нас операция разыменования `x↑` дважды переходит по ссылке (второй раз автоматически), чтобы попасть в динамическую переменную. После операции `dispose(x)` динамическая переменная (со значением `5`) объявляется свободной, а в «надгробие» записывается **`nil`** (…покойся с миром, больше тебя никто не сможет побеспокоить… 🙂):



И вот теперь присваивать переменным `x` и `y` значение **`nil`** не надо, и оператор `y↑:=5`, как и оператор `x↑:=5` тоже будет давать ошибку! Платой за повышение надёжности, естественно, является дополнительный расход памяти.

[ii] Для продвинутых читателей. Проблема утечки памяти очень серьёзна. Кардинальное решение этой проблемы возможно (да и то не полностью) только путём так называемой *автоматической* сборки мусора (garbage collection). Имеется в виду, что в исполнитель алгоритма встроен специальный механизм, который автоматически уничтожает динамические переменные, как только в них отпадёт необходимость (т.е. на них в

программе больше не будет ссылок). С этой целью с каждой динамической переменной связывается специальное поле – счётчик размещения (reference counter), которое считает, сколько ссылочных переменных могут иметь доступ к этой динамической переменной. Счётчик размещения увеличивается, когда на динамическую переменную добавляется ссылка, а также, когда эта переменная в свою очередь является ссылочной на другую динамическую переменную (например, это элемент списка).

Как только значение этого поля становится равным нулю, эту динамическую переменную можно автоматически уничтожить. Рассмотрим пример (счётчик ссылок показан в сером квадратике синим цветом):

```
type Rreal=↑real; RRreal=↑Rreal;
var x,y: Rreal; z: Rrreal;
  new(x); x↑:=1.2; { x→[1;1.2] }
  y:=x;            { x→[2;1.2]  y→[2;1.2] }
  new(x); x↑:=3.4; { x→[1;3.4]  y→[1;1.2] }
  y:=nil;          { x→[1;3.4]   →[0;1.2] }
{ Переменную [0;1.2] можно уничтожить }
  new(x);          { x→[1;?.?]   →  [0;3.4] }
{ Переменную [0;3.4] можно уничтожить }
  new(z); new(z↑); z↑↑:=5.6; { z→[2;ref] → [1;5.6] }
  z↑↑:=nil;        { z→[1;ref] → [0;5.6] }
```

Некоторые сложности возникают, когда у нас кольцевая списковая структура, скоро такой пример будет в тексте 12 главы.

Ещё пример с процедурой:

```
procedure P;
  var x: ↑real;
begin new(x); x↑:=5.6; { x→[1;5.6] }
end; { При выходе из процедуры { →[0;5.6] }
```

Иногда говорят (в языках C++, Rust и других), что используются умные указатели (smart pointer) на области динамической памяти. С такими указателями, кроме автоматического удаления ненужных динамических переменных, могут связываться и другие свойства, способствующие безопасному программированию (например, проверку на выход за границы выделенного участка памяти и т.д.).

Впервые автоматическая сборка мусора была предложена в 1956 году Джоном Маккарти в языке Lisp. Многие современные языки программирования (Java, Python, C#, Haskell и другие), делают только автоматическую сборку мусора, в таких языках у программиста просто нет средств для ручного уничтожения переменных (нет `dispose`, `Freemem` и т.д.). В этих языках нет и ручного порождения динамических переменных (нет `new`, `Getmem` и т.д.), а динамические переменные порождаются автоматически, когда в них возникает необходимость. Этот механизм существенно упрощает разработку и отладку программного обеспечения, но часто может внушать программисту ложное чувство безопасности, так как полностью устранить утечку памяти не удаётся. Далее, связанные с этим усложнения исполнителя и накладные расходы могут снижать скорость работы алгоритма. Кроме того, включаясь в непредсказуемые моменты времени, сборка мусора может нарушить работу критически важных по времени выполнения участков программы. Например, только программа собралась послать команду на запуск ракеты, а тут включилась сборка мусора 😬.

Исходя из этого, в некоторых языках механизм автоматической сборки мусора реализован частично. Например, в языке Free Pascal он работает только для динамических массивов и динамических строк символов типа `Ansistring`. Для этой цели перед каждой такой переменной располагаются 8 байт (два целых числа типа `longint`) служебной информации. Со смещением `-4` хранится значение `high(x)`, т.е. число элементов в массиве (`-1..N-1`), при этом –1 указывает на пустой массив. Со смещением `-8` располагается счётчик ссылок на эту переменную. При желании значение этого счётчика и текущую длину массива можно в программе читать (и, к сожалению, даже писать туда), например:

```
type AL=array of Longint;
var X: array of real; Ф: AL;
begin ...
Writeln('Число ссылок на X=',AL(X)[-1],
        'Длина массива X=',AL(X)[-2]+1);
AL(X)[-1]:=333; В
```

При достижении счётчиком ссылок значения нуля 0, переменная может быть автоматически уничтожена. Отметим, что при передаче переменной типа динамического массива или `Pchar` в процедуру по значению тоже производится увеличения счётчика ссылок, а при выходе из процедуры – уменьшение.

Другим подходом к управлению динамическими переменными является использование так называемых *деструкторов*, операторов, уничтожающих ненужные переменные (в основном это объекты классов, см. главу 16). Кроме уничтожения переменных, деструкторы могут закрывать файлы, разрывать сетевые соединения и выполнять другую полезную работу.

В основном, деструктор должен вызываться программистом, однако, в языках часто предусмотрены и средства *автоматического* вызова деструкторов, когда переменная выходит из зоны видимости и существования. В языке Free Pascal для этого предназначены так называемые интерфейсы, можно посмотреть здесь [https://softwaremaniacs.org/blog/2005/05/10/interface-memory-management/].