Глава 9. Записи (структуры)

Структура (struct) – композитный тип данных, инкапсулирующий без сокрытия 🙄 *набор значений различных типов.*

Википедия

По сути, я утверждаю, что разница между плохим программистом и хорошим заключается в том, считает ли он более важным свой код или свои структуры данных. Плохие программисты беспокоятся о коде. Хорошие программисты беспокоятся о структурах данных и их взаимоотношениях.

Луис Торвальдс.
Создатель ОС Linux

Пришло время познакомиться с новой сложной структурой данных, в Паскале переменные этого типа называются записями (**record**), а в большинстве других языков – структурами (**structure**).[1] Сначала рассмотрим записи как абстрактную структуру данных, а потом расскажем, как она реализована в Паскале. Полезно будет сравнить записи и массивы (см. таблицу 9.1).

Таблица 9.1. Сравнение массивов и записей

Массив	Запись
Является *переменной*	Является *переменной*
Состоит из *элементов*	Состоит из *полей* (fields)
Каждый элемент переменная	Каждое поле переменная
Все элементы одного типа	Поля могут быть разных типов
Элементы не имеют имён	Все поля имеют имена
Элементы в массиве упорядочены	Поля в записи не упорядочены

Массивы и записи имеют в программе разное назначение (прагматику). Когда программист хочет упорядоченный набор одинаковых безымянных переменных мыслить как единую (сложную) переменную, он должен использовать массив. А вот когда как единое целое он хочет рассматривать набор (не упорядоченное множество) переменных разных типов, то он должен использовать запись.

Описание типа записи:

```
<тип записи> ::=
    record <секция полей>;{<секция полей>;}…
           [<вариантная часть>]
    end;
<секция полей> ::= <имя поля>{,<имя поля>}…:<тип>;
```

Запись состоит из основной части и необязательной *вариантной* части. Изучим сначала записи без вариантной части. Использование записей безымянных типов в Паскале ограничено например, как мы уже знаем, их нельзя передавать как параметры в подпрограммы, поэтому типу записи лучше присвоить имя в разделе типов. Семантику записей рассмотрим на примерах.

```
type rec=record
           a,❶b:integer; x:char;
           y:array[1..10] of integer;
         end;
var x,z:rec; ❷b,c:integer;
begin x:=z; x.a:=z.b; z.y[5]:=1;
  a:=1; { ОШИБКА, имя a здесь не видно́}
```

---

[1] Термин «структура» как тип данных широко используется, но не совсем удачен, так как путается с общим понятием «структура данных». Впервые среди широко распространённых языков записи появились в языке Cobol, а позже всех, наверное, в языке Fortran-90.

```
b:=0; { это ❷ b:=0 }
z.b:=0; { это ❶ b:=0 в записи z }
z.c:=0; { ОШИБКА, поля c нет в записи }
```

Первое, что смущает учащихся, это совпадающие имена в разделе описания переменных блока и полей записи. Здесь всё дело в том, что внутри записи имена полей, как и имена в блоке, образуют собственное *пространство имён*, т.е. снаружи невидимы. Другими словами, имена полей можно выбирать как угодно, не заботясь о том, что они могут совпасть с другими именами в программе.

Как и для других сложных типов данных, перед описанием записи в языке Free Pascal можно ставить служебное слово **packed**. Это указание не выравнивать поля внутри записи в памяти компьютера, обычно такое выравнивание полей на границу 16-ти байт делается для ускорения доступа к памяти ЭВМ. В языке Free Pascal есть также служебное слово **bitpacked**, оно предписывает (по возможности) провести также упаковку элементов, для которых можно отвести меньше одного байта памяти (для типа `boolean` достаточно одного бита).

Как мы знаем, для доступа к частям сложной структуры данных используется конструкция под названием селектор, например, для массива это индекс в квадратных скобках `x[i]`. Селектором записи является символ точки, за которым следует имя поля, например, `x.b` есть целочисленная переменная с именем `b` – поле записи `x`.

Пространство имён внутри записи, отличается от пространства имён внутри блока (подпрограммы). Снаружи блока все имена внутри полностью инкапсулированы (скрыты, а переменные и не существуют!), а вот имена внутри записи, как сказано в Википедии «инкапсулированы без сокрытия», т.е. эти переменные существуют и к ним возможен доступ, но только через селектор точку.

Записи с именами `x` и `z` являются (сложными) переменными, причём совместимыми по присваиванию (объясните, почему ?), так что допустим оператор `x:=z`. Поля записи тоже являются переменными, так что можно писать `x.a:=z.b`. В то же время, находясь снаружи записи, имена полей внутри записи без селектора не видны, так что оператор `a:=1` будет ошибочен. Далее находясь внутри записи, имена всех внешних констант и переменных с селектором точкой тоже не видны, так что оператор `z.c:=0` будет ошибочным. Образно можно сравнить запись с коробкой, у которой стенки полностью непрозрачны. Сравним это с блоком, стенки которого непрозрачны *снаружи*, но прозрачны *изнутри*. Отметим, что запись изнутри непрозрачна только для имён переменных, имена констант и типов видны (если не закрываются одноимёнными полями внутри записи), например, для приведённого ниже примера это имена ❶ `N` и ❷ `Mas`:

```
const ❶ N=100;
type
❷ Mas=array[1..`10] of char;
  Rec=record
        x: char; y: ❷ Mas; z: 1..❶ N;
  end;
```

Теперь можно ответить на вопрос, сколько одинаковых имён (скажем, `X`) может быть в программе на стандарте Паскаля. Максимальное количество разных объектов, названных одним именем равно числу блоков в программе (это подпрограммы плюс вся программа, плюс объемлющий блок `System`), плюс число описанных записей. Обратите также внимание, что при изменении порядка описания полей внутри записи, например:

```
rec=record
   y: array[1..10] of integer;
   b,a: integer; x: char;
end;
```

остальная программа не меняется. Это и означает, что поля внутри записи *не упорядочены* (могут описываться в любой последовательности).

Итак, записи являются совокупностью именованных полей разного типа. Некоторые языки (ML, F#, Python и другие) предоставляют возможность работать и с совокуностью безымянных полей раз-

ного типа, обычно такие структуры данных называются кортежами. Например, вот описание кортежа (**tuple**) в языке Python:

```
x=tuple('ABC',100,3.14)
```

Доступ к элементам кортежа, как и в массиве, производится по индексам:

```
x[0] → 'ABC'    x[2] → 3.14      x[1:2] → (100,3.14)
```

На работу с кортежами обычно накладываются ограничения, например, в языке Python кортежи рассматриваются как константы, просто так их менять нельзя.

Записи широко используются в задачах обработки данных. В качестве простого примера рассмотрим описание фрагмента записи `Student` в информационной системе некоторого вуза (будем использовать язык Free Pascal).

```
type
  Student=record
    Fio: string[40]; { ФИО }
    Pol: (F,M);[1]
    Grup: 101..649; { № студенческой группы }
{ экзамены по алгебре, матанализу и программированию }
    Ex: record Al,An,Pr: 1..5 end;
  end;
  Gruppa=array[1..24] of Student;[2]
{ далее можно описать курс, факультет и т.д. }
```

«Особая» неудовлетворительная оценка «единица» ставится, если студент по какой-либо причине (уважительной или нет) не явился на экзамен. В принципе, можно было бы сделать экзамены массивом, но тогда придётся обращаться к ним `Ex[1]`, `Ex[2]` и `Ex[3]`, а для программиста удобней присвоить экзаменам *имена*. Рассмотрим реализацию некоторой операции в такой информационной системе. Пусть после завершения экзаменов инспектору учебной части понадобилось для некоторой группы распечатать список студентов, имеющих неудовлетворительную оценку хотя бы за один экзамен (это так называемые «задолжники»).

Ясно, что эту операцию надо реализовывать в виде процедуры, единственным параметром которой является студенческая группа. Наша процедура не меняет свой параметр, поэтому его надо было бы передавать *по значению*, чтобы случайно в процедуре не изменить. Параметр, однако, большой по размеру, поэтому, как мы обосновали ранее, его всё же следует передавать *по ссылке*., однако с использованием служебного слова **const**.

Далее сделаем спецификацию этой операции. Ясно, что не в каждой группе ровно по 24 студента, их может быть и меньше. Более того, если студент удалён из середины группы (например, отчислен или переведён), то в массиве образуется «пустое место». Договоримся, что ФИО таких студентов будет начинаться с символа `'-'` (таких «настоящих» фамилий не бывает). Скажем, при удалении студента из группы выполняется операция `Fio.c:='-'+Fio`. Тогда получится такая процедура.

```
procedure Zadoljniki(const x: Gruppa);
  var i: integer;
begin Writeln('Задолжники ',x[1].Grup,' группы:');
  for i:=1 to 24 do
    if x[i].Fio[1]<>'-' then { студент есть в группе }
      if (x[i].Ex.Al<3) or (x[i].Ex.An<3) or
         (x[i].Ex.Pr<3) then
           Writeln(x[i].Fio)
```

---

[1] Это традиционные ♀ и ♂. Сначала «у них» для этих целей было необходимо шесть значков  , а сейчас уже где-то полтора десятка значков.

[2] Исторически в группе на ф-те ВМК МГУ не более 24 студентов. Когда-то давно, в учебных компьютерных классах было всего по 12 машин. С тех пор на занятиях по практикуму на ЭВМ студенческую группу делят на две подгруппы и выделяют двух преподавателей, чего нет на других предметах (кроме иностранного языка).

```
end;
```

При программировании важно видеть структуру программы, сколько в ней операторов, где начинается и заканчивается каждый из них. В качестве примера на рис. 9.1 приведена структура раздела операторов процедуры `Zadoljniki`, там всего два оператора: процедура вывода ❶ и цикла ❷.

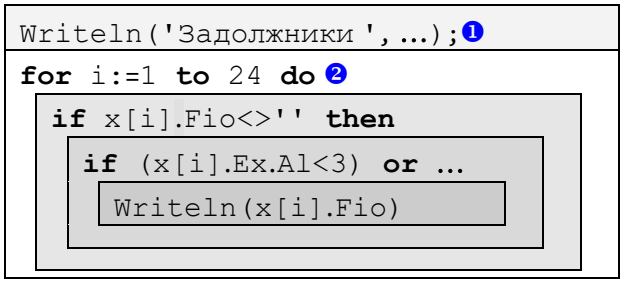

Рис. 9.1. Структура процедуры `Zadoljniki`.

Ещё два тонких момента. Во-первых, надо добавить спецификацию, что у первого студента `x[1]`, даже если его в группе нет, всё равно задано правильное значение поля `x[1].Grup`. Во-вторых, у нас допущена ошибка в интерфейсе: при отсутствии в группе задолжников всё равно выводится заголовок списка «плохих» студентов. Исправьте это, пусть выводится «Задолжников нет».

9.1. Оператор присоединения

Программирование – такое же сложное искусство, как и литературное творчество. И там и здесь предпочтение отдаётся краткости.

Ричард Уэсли Хэмминг,
«Тьюринговская лекция».

Теперь рассмотрим последний сложный оператор Паскаля – оператор присоединения. Этот оператор уникален, первоначально он был реализован только в Паскале.[1] Сначала, как обычно, синтаксис:

<оператор присоединения> ::= **with** <список переменных> **do** <оператор>
<список переменных> ::= <переменная-запись>{,<переменная-запись>}...

Итак, после ключевого слова **with** идёт список переменных-записей, а в теле находится ровно один оператор. Оператор присоединения можно изобразить так (рис. 9.2).

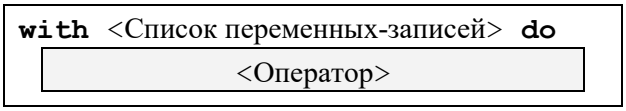

Рис. 9.2. Структура оператора присоединения.

Для изучения семантики оператора присоединения, изменим последнюю процедуру, использовав при её написании оператор присоединения:

```
procedure Zadoljniki(const x: Gruppa);
  var i: integer;
begin Writeln('Задолжники ',x[1].Grup,' группы:');
  for i:=1 to 24 do
    with x[i],Ex do
      if ❶ Fio[1]<>'-' then
        if (Al<3) or (❷ An<3) or (Pr<3) then
           Writeln(Fio)
end;
```

Оператор присоединения меняет трактовку всех имён переменных в своём теле, пытаясь приписать впереди (через точку) переменные-записи из своего списка переменных-записей. Список этих

[1] Сейчас он широко используется во многих, особенно объектно-ориентированных языках, см. главу 14.

переменных для присоединения начинает просматриваться с конца. Например, для ❶ `Fio<>''` сначала рассматривается переменная `Ex.Fio`, однако в области видимости поле с именем `Fio` не обнаруживается. Тогда исполнитель пробует `x[i].Ex.Fio` и опять неудача. Далее пробуется `x[i].Fio` и здесь нас ожидает успех. И, наконец, если бы и последняя попытка была бы неудачной, исполнитель попробовал бы увидеть просто переменную `Fio`. Проще всего понять семантику работы оператора присоединения, если знать, что по определению оператор

```
with x[i],Ex do S
```

выполняется как оператор

```
with x[i] do with Ex do S
```

Таким образом, просматривая переменные-записи из списка **with** в обратном порядке, для ❷ `An<3` исполнитель пытается увидеть имя `An` в такой последовательности:

`Ex.An` → `x[i].Ex.An` → `x[i].An` → `An`

Можно сказать, что внутри оператора **with** установлена своя собственная, весьма специфическая *область видимости*. Главное назначение оператора присоединения состоит в более компактной записи алгоритма, однако здесь кроется и возможность семантических ошибок при неправильном понимании последовательности разрешения областей видимости имён. Например, пусть в записи `Student` добавилось новое поле

```
Student=record
  Fio: string[40]; { ФИО }
  Pol: (F,M);
  Pr: integer; { категория призыва }
  Grup: 101..649; { студенческая группа }
  Ex: record Al,An,Pr: 1..5 end;
{ экзамены по алгебре, матанализу и программированию }
end;
```

Теперь при вычислении `Pr<3` получится интересный результат 🙂.

9.2. Записи с вариантами

Самая катастрофическая вещь, которую вы когда-либо изучали – это ваш первый язык программирования.

Алан Кэй, автор первого объектно-ориентированного языка Smalltalk.

Перейдём теперь к рассмотрению необязательной вариантной части записи, сначала поймём её назначение. В жёстко типизированных языках программирования (к ним относится и Паскаль) с каждой (существующей) переменной связана область памяти и тип. Исполнитель контролирует, чтобы в эту область памяти можно было записать величины только данного типа. Например, в память вещественной переменной невозможно записать целое или символьное значение. Во-первых, переменные этих типов обычно имеют разную длину,[1] а во вторых, разное машинное (битовое) представление. Поэтому, например, писать в некоторую область памяти вещественное значение, а потом читать из этой области целое число обычно строго запрещено.

Возможны различные нарушения такого закона строгой типизации. Например, в языке Python переменной (скажем с именем `X`) можно присваивать значение разных типов, скажем:

`X=123; X=-456.78; X="ABCDEF"` и т.д.

В этом языке, однако, другая *концепция* переменной. Считается, что имя переменной не определяет область памяти, в которой хранится эта переменная, а является только своеобразным *ярлыком* (label), который можно «повесить» на самые разные области памяти. Так, после оператора присваивания `X=123` в некоторой области памяти размещается величина `123` и на эту область «вешается»

[1] Длины переменных разных типов могут и совпадать. Например, переменная типа `char` и переменная типа `boolean` обычно имеют одинаковую длину 1 байт, но в Паскале они не совместимы по присваиванию.

имя-ярлык X. После оператора `X=-456.78` область памяти, занимаемая целой величиной 123 уничтожается, выделяется новая область памяти, достаточная для хранения вещественной величины -456.78, затем ярлык X перевешивается на эту новую область памяти и т.д. По существу, оператор присваивания в Паскале меняет значение переменной на новое, а в языке Python этот оператор меняет у переменной её ссылку (на новую ссылку, указывающую на другую область памяти).

В языке Free Pascal тоже есть очень похожий *вариативный* тип variant, переменным этого типа тоже можно присваивать значения почти любых типов, например:

```
var X: variant;
begin X:=123; X:=-456.67; X:='A'; X:="ABC" и т.д.
```

Ясно, что тип такой переменной определяется по последнему присваиванию, а область памяти приходится выделять заново, как только новое значение будет занимать больший объём памяти, чем старое.

В некоторых задачах, однако, представляется полезным уметь хранить в одной, единожды выделенной, области памяти значения разных типов. Например, в записи для хранения атрибутов автомобиля необходимы целое поле «число мест» для легковых автомобилей и вещественное поле «грузоподъёмность» для грузовых. Если описать в записи оба эти поля, то для каждого автомобиля одно из них будет пустовать и зря занимать память. Для решения этой проблемы можно выделить в записи область памяти, в которую будет входить самая «большая» из этих переменных. Именно для таких задач и предназначена вариантная часть записи в Паскале. Сначала синтаксис:

```
<вариантная часть> ::= case [<селектор>:]<дискретный тип> of
                           <вариант>{;<вариант>}...
<селектор> ::= <имя>
<вариант> ::= <константа>{,<константа>}...:({<секция полей>}...);
```

Каждый вариант определяет свою запись, все такие варианты будут храниться в одной и той же области памяти, накладываясь друг на друга. Допускаются два вида вариантной части. В первом из них вариантная часть «знает», какой именно вариант хранится сейчас в записи, для этого в этой записи предусмотрено служебное поле селектор:

```
case <селектор>:<дискретный тип> of
```

Рассмотрим пример, в нём для наглядности используются имена из русских букв:

```
type
  Тип=(Грузовой,Легковой);
  Авто=record Цена: integer;
         case t:Тип of { t - поле-селектор варианта }
           Грузовой:(Груз: real; Высота: integer);
           Легковой:(Места,Двери,Цвет: integer)
       end;
var X,Y: Авто;
begin {Ввод X}
  if X.t=Грузовой then
    Writeln('Грузовой автомобиль, ',X.Груз:6:2,' тонн')
  else
    Writeln('Легковой автомобиль, число мест= ',X.Места)
end.
```

Во втором случае вариантная часть «не знает», какой именно вариант хранится в конкретной записи, это позволяет не хранить в каждой записи служебное поле-селектор

```
case <дискретный тип> of
```

Например:

```
type
  Тип=(Грузовой,Легковой);
  Авто=record Цена: integer;
         case Тип of
           Грузовой:(Груз: real; Высота: integer);
```

```
            Легковой: (Места,Двери,Цвет: integer)
        end;
var X,Y: Авто;
begin {Ввод X}{ Пусть нам известно, что X – грузовой }
  write('Грузовой автомобиль ',X.Груз:6:2,' тонн')
{ Если автомобиль будет легковым, вывод будет забавный 🙂 }
end.
```

На рис. 9.3 показан вид такой «без селекторной» записи (пусть целый тип занимает 2 байта, а вещественный 8 байт).

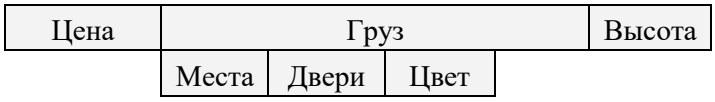

Рис. 9.3. Наложение вариантов записи друг на друга.

В других языках (например, в C, Haskell и Ассемблере MASM) вместо вариантной части записей используется отдельный тип данных под названием «объединение» (**union**), он может входить в описание структуры как самостоятельное поле.

Использование записей с вариантами и объединений, хотя и экономит память, но представляет опасность при небрежном программировании. Например, можно записать поле `Груз`, а потом читать поле `Места`, поведение программы при этом не определено, но явно, что ничего хорошего не будет. Исходя из этого языки, позицирующие себя как «надёжные» (например, Java и C#) такие конструкции не применяют.

Отметим что, как мы уже упоминали, язык Free Pascal допускает описывать и явное наложение переменных друг на друга в памяти ЭВМ, например:

```
var X: real; Y: integer absolute X;
```

Здесь модификатор переменной `absolute` (это *стандартное*, а не служебное имя) [1] требует, чтобы переменные `X` (8 байт) и `Y` (2 байта) начинались с одного и того же адреса памяти ЭВМ, таким образом накладываясь друг на друга. В принципе, можно указать и конкретный адрес ячейка памяти ЭВМ, по которому должна находиться описанная переменная, например

```
var Z: integer absolute $100000;{ 100000₁₆ }
```

Впрочем, так как узнать, что будет находится по конкретному адресу памяти во время счёта программы весьма трудно, то это не имеет большого смысла.

Записи в Паскале (или структуры в других языках), как и блоки, обеспечивают свою область видимости имён переменных-полей. При этом, однако, блоками они не являются, так как в них можно описывать только переменные, но нельзя описывать константы, типы и подпрограммы. При развитии языка программирования, добавилась возможность описывать внутри записи константы, типы и подпрограммы, они стали называться *расширенными записями* (advanced records).

Например, это можно делать в языке Free Pascal, если включить такую возможность директивой:

```
{$modeswitch AdvancedRecords}
type
  MyRec=record {Расширенная запись}
    const N=10;
          S='Запись MyRec=';
    type int=-100..100;
    var x,y: int;
    function PSum(a: integer): integer;
    procedure Print(a: integer);
  end; { MyRec }

function PSum(a: integer): integer;
```

[1] Модификаторы языка Free Pascal (их много: `absolute`, `public`, `vectorcall`, `nostackframe` и другие) являются «странными» стандартными именами, одновременно (в одной области видимости) их можно использовать и как модификаторы (имеющие заданный смысл для компилятора), и как «обычные» имена пользователя.

```
begin PSum:=a*(x+y) end;

procedure MyRec.Print(a: integer);
begin Writeln(S,PSum(a)) end;

var p,q: MyRec;
begin
  with p do begin
    x:=1; y:=2; Print(N)
  end;
  q:=p; Writeln(q.N+q.x)
end;
```

Как видно, внутри самой записи подпрограммы только *объявляются* (хотя ключевое слово **forward** и не используется), а *описание* этих подпрограмм производится позже. В дальнейшем такие записи превратились в новые сущности, которые обычно называются объектами, а сами такие языки стали называть объектно-ориентированными. Самое элементарное введение в такие языки описано в главе 14.