ГЛАВА 5

Структурное программирование и операторы языка Object Pascal

Раз дощечка, два дощечка – будет лесенка.
Раз словечко, два словечко – будет песенка.

Слова из песни

В предыдущей главе мы рассмотрели базовые составляющие языка программирования высокого уровня Object Pascal: алфавит, ключевые слова, правила формирования идентификаторов, выражений и операторов, структуру программы. Также обсудили такие важнейшие элементы любой программы, как переменная и оператор присваивания. Без правильного их понимания ни одну сколько-нибудь серьезную программу написать невозможно (да и считаться программистом, на наш взгляд, тоже). Наконец, мы изучили простейшие средства обмена информацией между программой и ее пользователем – операторы ввода и вывода. Используя полученные знания, мы совершили пару маленьких шагов на пути к сияющим вершинам программистского мастерства – написали несколько элементарных программ.

Однако чтобы продвинуться дальше, нам потребуется нечто большее, чем простое изучение элементов языка. Как овладев, пусть даже в совершенстве, английским языком, вы не станете автоматически Уильямом Шекспиром, так и изучив досконально язык программирования, пусть даже такой мощный, как Object Pascal, вы не подниметесь выше уровня написания программ «для себя». Процесс создания сложных, больших программных комплексов включает в себя и радость творчества, и рутину кодирования, и мучительные поиски ошибок, однако все это будет бесполезной тратой сил и времени, если сам процесс не будет должным образом организован. Будучи Левшой, можно «на коленке» создать невероятно красивую, оптимизированную, оригинальную, полезную (подставьте любой эпитет в превосходной степени) программу. Но! Исходный код программных комплексов уровня операционной системы содержит десятки миллионов строк. Никому не под силу не только разработать такую систему в одиночку, но и удержать в голове все необходимые решения, которые будут приняты в процессе создания. Более того, написать код подобного объема не придерживаясь определенных

правил и стандартов, не используя типовые приемы невозможно. Нужны организационные и технологические решения, которые позволили бы справиться с этой фантастической сложностью. Одним из таких решений является концепция структурного программирования, с которой мы познакомимся в настоящей главе. Но прежде обсудим, чт∴ же скрывается за самим понятием *технология программирования*.

1. Программирование как технологический процесс. Понятие технологии программирования

Еще в первой главе мы перечислили ряд вопросов, без получения ответов на которые невозможна разработка сколько-нибудь серьезной программной системы. Вот еще некоторые.

Из чего состоит сам процесс создания сложного программного комплекса? Как организованы работа коллектива программистов, процессы анализа, проектирования, программирования, отладки, модификации? И почему мы употребляем здесь слово процесс? Только ли для того, чтобы подчеркнуть сложность, или по какой-то другой причине?

Кажущаяся легкость написания учебных «игрушечных» программ порождает ложное впечатление о наличии простых ответов на перечисленные вопросы. Однако на поверку выясняется, что простотой здесь и не пахнет. Для того чтобы это продемонстрировать, рассмотрим пример из любой давно существующей отрасли, например судостроения. Давным-давно индейцы Северной Америки не испытывали никаких проблем в этой области. Так, для того чтобы построить пирогу, не требовалось никаких специальных методов и уж, конечно, никакой формальной теории. Казалось бы, в чем проблема? Топор, дерево, немного усилий – и пирога способна принять на борт ваших соплеменников. А вот удастся ли на этой пироге достичь Британии? Что с ней будет в случае возникновения шторма? А что, если в пирогу врежется акула? А если ее подожгут? И это только часть вопросов, за каждым из которых своя проблема, требующая решения.

Поместите теперь мысленно в эту пирогу себя. Вот вы беретесь за весла… Какой такой двигатель? Навигационные приборы? Телевизор? Вы не можете путешествовать без компьютера? Вы чувствуете недостаток сотовой связи? Всего этого не сделать топором. Тут понадобятся некоторые знания и инструменты. Иногда даже понадобится много рабочей силы и оборудования, быть может, целые заводы. Добавьте сюда проектирование, разработку, спуск на воду и ходовые испытания корабля, которые в настоящее время представляют собой очень сложный, но продуманный,

отлаженный и детально расписанный процесс, *технологический* процесс. Для решения большинства производственных задач существуют свои технологии. Давно ушли в прошлое времена, когда человек изготовлял что-либо по наитию. Сегодня уже невозможно представить ситуацию, когда рабочие приходят утром на завод и голосованием решают, кто вырубает топором корпус, кто ставит мачту, кто приделывает парус и, тем более, как именно это делать. Получившееся корыто (простите, корабль!) вряд ли вообще поплывет. Технологические процессы, в которых все четко определено, заняли ведущее место на производстве.

Итак, что же такое технология? *Технология*[1] *– совокупность производственных процессов в определенной отрасли производства, а также научное описание способов производства.*

Вернемся теперь к программированию. В 60-х–70-х годах XX века актуальным считался вопрос о сущности программирования, о том, чт.∴ это такое: наука, техника или искусство? Одна из самых известных книг по программированию, написанная в то время и выдержавшая множество переизданий (настоятельно рекомендуем нашим читателям внести ее в перечень ближайших пополнений своей домашней библиотеки), так и называется: «Искусство программирования» [9, 10]. Уже тогда людям было ясно, что ввиду роста сложности решаемых при помощи компьютера задач неимоверно возрастает стоимость разработки программ. Причем если стоимость аппаратуры растет умеренными темпами, а иногда и вовсе падает, то со стоимостью разработки программ ничего поделать не удается. Именно тогда вопросы о том, чт.∴ такое программирование и как оптимизировать процесс разработки, вышли на первый план.

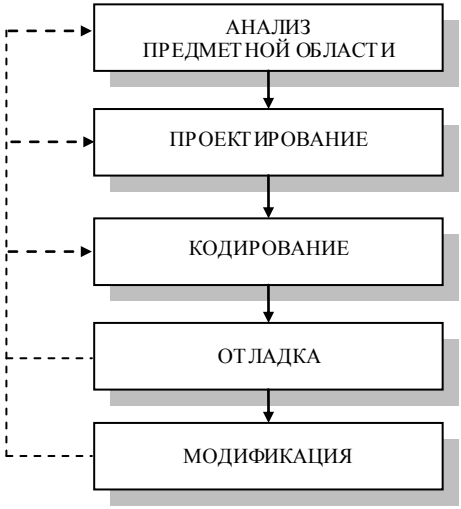


Рис. 5.1. Схема процесса создания программной системы

Для того чтобы ответить на эти вопросы, потребовалось определить, – куда же уходят средства, за счет чего возрастает стоимость разработки программных систем?

[1] Ожегов С.И. Словарь русского языка. – М.: Советская энциклопедия, 1975.

Рассмотрим кратко следующую схему (рис. 5.1).

Данная схема представляет собой последовательность стандартных этапов, которые необходимо выполнить при создании любой программной системы. Именно на этих этапах и возникают существенные финансовые затраты. Для их оптимизации необходимо было понять, что программирование есть обычный *технологический процесс*, по характеру возникающих проблем мало чем отличающийся от, скажем, строительства дома или корабля.

Для сокращения затрат необходимо было конкретизировать схему, упорядочить действия, выполняемые на каждом этапе, разработать методы решения возникающих на разных этапах проблем. В довершение ко всему, схема подразумевает возвраты назад (циклы) в тех случаях, когда обнаруживается ошибка предыдущего этапа.

В результате кропотливой работы большого количества специалистов на каждом этапе и подэтапе возникли и продолжают появляться и совершенствоваться специальные *технологии*, позволяющие решать задачи в заданные сроки с заданным качеством.

Итак, *технология программирования* – совокупность методов, приемов и средств для сокращения стоимости и повышения качества разработки программных систем.

В любой серьезной компании, занимающейся разработкой программного обеспечения, на каждом этапе рассмотренной схемы применяется большое количество разных технологий. Сразу отметим, что та часть организационных моментов, которая относится к обеспечению взаимодействия множества людей в процессе работы над программным комплексом, а также к построению самого процесса разработки программного обеспечения от постановки задачи до получения результата в виде готовой программной системы, выходит за рамки данной книги. Мы уделим этому минимально необходимое внимание, а серьезно изучать будем ту часть технологий программирования, которая непосредственно связана с процессом превращения имеющегося набора алгоритмов в программную реализацию. Положения рассмотренных в книге технологий – структурного, модульного и объектно-ориентированного программирования – стали к настоящему моменту общепринятыми, устоялись, перешли в разряд базовых. Надеемся, что скоро и вы согласитесь с их фундаментальным значением.

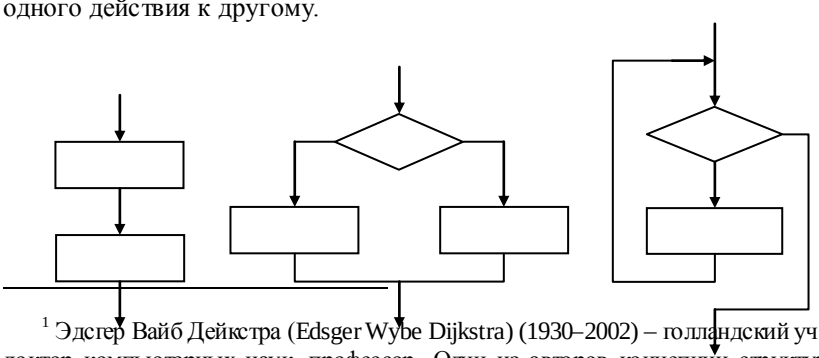
2. Концепция структурного программирования

Возникновение концепции структурного программирования [13, 28, 41] связывается с именем известного голландского ученого Э. Дейкстры[1] – в 60-х годах прошлого века он сформулировал основные ее положения.

Принцип, на котором зиждется *технология структурного программирования*, – фундаментальная научная и техническая идея о выделении множества базисных элементов, с помощью которых можно выразить (из которых можно собрать) любой объект из некоторого широкого набора. Так же как из ограниченного числа деталей детского конструктора можно при наличии некоторой фантазии построить весьма большое количество «изделий», так же как суперпозицией функций из базисного множества можно выразить любую функцию из некоторого пространства (например, все булевы функции могут быть построены на основе полной системы из конъюнкции и отрицания), так и программы, как было доказано, можно создавать, используя лишь небольшое число базисных алгоритмических конструкций.

Итак, основной принцип технологии структурного программирования гласит: для любой *простой* программы можно построить функционально эквивалентную ей *структурную* программу, т.е. программу, сформированную на основе фиксированного базисного множества, включающего *структуру последовательного действия*, *структуру выбора* одного из двух действий и *структуру цикла*, то есть многократного повторения некоторого действия с проверкой условия остановки повторения.

На рисунке 5.2 представлено изображение указанных алгоритмических конструкций в виде блок-схем. Здесь прямоугольник обозначает обобщенное действие, ромб – проверку условия, стрелки – переход от одного действия к другому.



[1] Эдсгер Вайб Дейкстра (Edsger Wybe Dijkstra) (1930–2002) – голландский ученый, доктор компьютерных наук, профессор. Один из авторов концепции структурного программирования. Активно участвовал в разработке языка программирования Algol, автор первого компилятора Algol 60. Лауреат премии Тьюринга. Разработал многие ставшие классическими алгоритмы. Идеи Э.В. Дейкстры оказали огромное влияние на развитие компьютерной индустрии.

Рис. 5.2. Блок-схемы базисных алгоритмических конструкций

Под *простой* программой в данном случае понимается программа, имеющая ровно один вход и один выход по управлению, такая, что через все ее функциональные блоки проходит путь от входа до выхода.

Изложенный принцип представляет собой *теорему о структурировании*. Ее точная формулировка и доказательство не входят в нашу задачу, желающие могут обратиться к монографии [13].

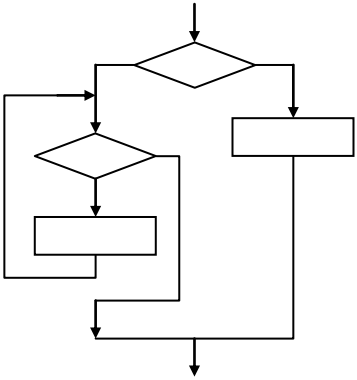


Рис. 5.3. Суперпозиция алгоритмических конструкций

Базисные алгоритмические конструкции обладают важным свойством – они в точности удовлетворяют определению *простой* программы, то есть имеют один вход и один выход, что обеспечивает возможность осуществлять их суперпозицию. Любая из трех структур может быть подставлена в остальные или в саму себя. Например, на рисунке 5.3 произведена подстановка цикла в выбор.

Таким образом, центральный технологический принцип структурного программирования состоит в том, что формулировку алгоритма и его запись в виде программы рекомендуется выполнять на основе базиса из трех алгоритмических конструкций, применяя при необходимости их суперпозицию. Результатом последовательного применения этого принципа будет более ясная структура программы (в особенности если использовать выделение структурных уровней с помощью отступов), что, несомненно, облегчит поиск в ней ошибок и упростит ее модификацию.

Ориентация языка программирования на некоторую технологию означает включение в него соответствующих выразительных средств. Язык Pascal создавался как язык структурного программирования, поэтому в него включены операторы, реализующие рассмотренные структурные примитивы. Более того, для большего удобства эти примитивы имеют несколько вариантов реализации.

Прежде чем перейти к изучению соответствующих операторов языка Pascal, сделаем два важных замечания общего плана.

Во-первых, концепция структурного программирования носит универсальный характер и не связана с конкретным языком. Например, в первом широко известном языке программирования высокого уровня – языке Fortran (точнее, в его ранних версиях) отсутствуют операторы структурного программирования. Но это совсем не значит, что данная

технология не может быть применена для составления программ на языке Fortran. Просто требуется выбрать способ записи базисных структур на основе имеющихся в языке операторов и придерживаться этого способа при разработке программ.

Во-вторых, необходимо иметь в виду: использование технологии структурного программирования (да и любой другой) при разработке программ не самоцель, а средство, применяемое в конечном счете для уменьшения затрат на их создание. Таким образом, на практике поступать следует в точном соответствии с общеизвестной истиной, гласящей, что из любого правила бывают исключения, – если точное следование канонам технологии ухудшает, каким угодно образом, текст программы, каноны нужно нарушить. В структурном программировании таким каноном, прямое следование которому способно в некоторых ситуациях принести больше вреда, чем пользы, является принцип: «один вход, один выход». В дальнейшем мы рассмотрим ситуации, в которых отступление от этого принципа является вполне оправданным.

3. Программирование последовательности действий

Реализация *структуры последовательного действия* чрезвычайно проста. В языке Pascal, как и в большинстве других языков программирования, принято естественное правило выполнения операторов в порядке их физического следования в тексте программы. Все приведенные в предыдущей главе фрагменты программ как раз и являются примерами использования этой алгоритмической конструкции. Как вы уже могли заметить по этим примерам, в языке Pascal операторы отделяются друг от друга точкой с запятой.

И все-таки, несмотря на простоту первой из базисных структур, один важный момент, необходимость и полезность которого в полной мере мы проясним дальше, нужно обсудить именно здесь. Как было указано выше, каждая из трех базисных алгоритмических конструкций может быть подставлена в качестве элемента в любую другую. То есть одна структура последовательного действия может быть элементом другой структуры последовательного действия. Вполне очевидно, что подобное разбиение последовательности действий на элементы является исключительно логическим, однако Pascal содержит средства, позволяющие такое разбиение осуществить в тексте программы явно. Помимо прочего оно является указанием компилятору при осуществлении синтаксического анализа текста программы рассматривать данный блок как один оператор, что часто бывает важно для правильной интерпретации программы.

Итак, для того чтобы оформить структуру последовательного действия в отдельную синтаксическую конструкцию, необходимо использовать так называемые *операторные скобки*: ключевые слова **begin** и **end**. При этом сама такая конструкция называется *составным оператором*.

```
Write('Введите координаты прямоугольника');
begin
   Read(x1, y1);
   Readln(x2, y2)
end;
{ дальнейший код, например рисование }
```

В приведенном примере операторы ввода синтаксически выделены в отдельный составной оператор. Заметим, что после последнего оператора группы, то есть перед ключевым словом **end**, точку с запятой можно не ставить (однако из практических соображений пользоваться этим исключением из правил мы не рекомендуем).

4. Программирование выбора

Структура выбора является, без сомнения, самой важной из базисных алгоритмических конструкций. Лишь чрезвычайно простые программы могут быть написаны без ее использования. *Структура последовательного действия* естественна, как отображение того факта, что любой алгоритм есть последовательность некоторых шагов. А *структура цикла* при наличии в языке программирования оператора безусловного перехода может быть реализована, как будет показано далее, на основе структуры выбора. В данном разделе мы рассмотрим практические ситуации, в которых возникает необходимость использования этой структуры, изучим имеющиеся в языке Pascal *операторы выбора*, реализующие структуру выбора в ее исходном виде или в виде некоторых расширений, повышающих удобство программирования.

4.1. Выбор из двух вариантов

Рассмотрим одну из типичных задач, возникающих при реализации взаимодействия программы с пользователем, – обеспечение контроля вводимых данных. Допустим, что пользователь в ответ на запрос программы должен ввести целое число, обозначающее день месяца. Число

это, естественно, должно находиться в интервале от 1 до 31. Допустим также, что программа написана правильно и, помимо предложения ввести число, выводит на экран подсказку о правильном диапазоне. Очевидно, что даже в этом случае пользователь при вводе может ошибиться и набрать неверное число или вовсе нажать клавиши с символами, отличными от цифр. Таким образом, данные, получаемые от пользователя, необходимо проконтролировать и при наличии ошибки как минимум выдать надлежащее сообщение, а еще лучше предложить ввести число заново.

Решение этой задачи в общем виде требует использования цикла, поэтому пока мы рассмотрим минимальный и не совсем верный с практической точки зрения вариант, в котором возможные ошибки ввода только проверяются. Словесное описание данного алгоритма можно дать в следующей форме (такую запись иногда называют *псевдокодом*):

```
вводим данное
если введенное данное не является целым числом, то
   выдаем сообщение об ошибке,
иначе,
   если введенное число – целое,
     но не входит в интервал от 1 до 31, то
       выдаем сообщение об ошибке,
   иначе
     выполняем необходимые действия
завершаем работу
```

Приведенная последовательность действий содержит характерную алгоритмическую конструкцию вида «если… то… иначе…». Очевидно, что эта конструкция в точности соответствует *структуре выбора*. Именно в таком виде она и реализована в языке Pascal (естественно, на английском языке): «if… then… else…».

Формальная запись *оператора выбора* **if** выглядит следующим образом:

```
if <условное выражение> then
   <оператор1>
else
   <оператор2>;
```

При этом условное выражение, по которому собственно осуществляется выбор, должно иметь *булевский* тип. Если результат вычисления выражения – **True** (истина), то выполняется `оператор1`, в противном случае `оператор2`.

При использовании условного оператора **if** нужно иметь в виду два важных *синтаксических* правила. Во-первых, перед ключевым словом **else**, то есть после оператора, записанного в секции **then**, нельзя ставить

точку с запятой. Компилятор в таком случае выдаст ошибку. Во-вторых, согласно приведенному формату записи условного оператора каждое из двух действий в нем *должно* состоять *ровно* из одного оператора. Что же делать, если по схеме алгоритма действий должно быть много? Решение было указано в предыдущем разделе – необходимо использовать *составной оператор*. В результате получится, например, следующее:

```
if <условное выражение> then
begin
   <оператор1>;
   <оператор2>;
end
else begin
   <оператор3>;
   <оператор4>;
   <оператор5>;
end;
```

Вернемся к поставленной в начале раздела задаче контроля ввода числа. В языке Pascal предусмотрены специальные средства для выполнения автоматического контроля операций ввода и вывода. В частности, контролируется соответствие типа вводимой информации, и при необходимости можно установить контроль за попаданием введенных чисел в некоторый диапазон. Будут средства контроля включены в код исполняемого модуля или нет – зависит от *директив* компилятора. Так, проверка операций ввода и вывода задается директивами `{$I}`, `{$IOCHECKS}`, а проверка на диапазон – директивами `{$R}` и `{$RANGECHECKS}`. По умолчанию проверка ввода-вывода включена, проверка на диапазон выключена.

Использование стандартных средств, однако, приводит к тому, что при возникновении ошибок, во-первых, выдаются сообщения вида «Runtime error 106 at 00403A30», которые мало о чем скажут пользователю программы (а скорее просто напугают его), а во-вторых, автоматически обрывается выполнение программы, что и вовсе является недопустимым. Неверный ввод не является критическим для работоспособности программы и не должен приводить к ее завершению. Грамотно написанная программа должна сама «перехватывать» ошибочный ввод и выдавать сообщения, из которых можно понять, что именно произошло и что в результате делать дальше.

Таким образом, в примере, приведенном ниже, мы не используем директиву `{$R}` и отказываемся от действия директивы `{$I}`. Взамен, для того чтобы узнать, чем закончилась последняя операция ввода-вывода, мы используем функцию `IOResult`. Если последняя операция ввода-вывода

завершилась успешно, функция возвращает нуль, иначе значение, отличное от нуля.

```
{ ================================================== }
{ Пример 5.1 }
{ Контроль вводимой информации }
Program Control;

{$APPTYPE CONSOLE}

var
   Day: Integer;
begin
   {$I-} { отключили автоматическую проверку ввода-вывода }
   ReadLn(Day);
   if (IOResult <> 0) then { ошибка ввода }
    WriteLn('Ошибка 1: Недопустимые символы в целом числе')
   else
    if (Day < 1) or (Day > 31) then
      WriteLn('Ошибка 2: День месяца вне диапазона 1..31')
    else begin
      { Дальнейший текст программы }
    end;
end.
{ ================================================== }
```

На практике часто встречается ситуация, когда содержательные действия необходимы *только* при выполнении некоторого условия, в противном же случае ничего делать не требуется. В этой ситуации в языке Pascal можно использовать так называемый *пустой оператор*, то есть:

```
if <условное выражение> then
   <оператор1>
else
   ; { это пустой оператор, "ничего не делать" }
```

Однако такая конструкция, оправданная синтаксически, является явно избыточной с точки зрения здравого смысла. Поэтому оператор **if** имеет сокращенную форму (иногда ее называют *неполной альтернативой*), в которой секция **else** опускается вместе со своим ключевым словом:

```
if <условное выражение> then
   <оператор>;
```

Заметим, что теперь после оператора в секции **then** точка с запятой необходима.

В качестве примера использования сокращенной формы оператора **if** рассмотрим программу поиска корня уравнения $ax + b = 0$. Известно, что

данное уравнение не имеет корней, если $a = 0$ и $b \neq 0$, бесконечно много корней, если $a = 0$ и $b = 0$, один корень, если $a \neq 0$.

```
{ ================================================ }
{ Пример 5.2 }
{ Решение линейного уравнения }
Program LinearEquation;

{$APPTYPE CONSOLE}

var
   a, b: Real;
begin
   WriteLn('Введите коэффициенты уравнения ax+b=0: ');
   ReadLn(a, b);
   if (a = 0) and (b = 0) then
      WriteLn('Корней бесконечно много');
   if (a = 0) and (b <> 0) then
      WriteLn('Корней нет');
   if (a <> 0) then
      WriteLn('x = ', -b/a);
end.
{ ================================================ }
```

Указанные три варианта можно анализировать в разном порядке, в том числе и с использованием условного оператора в виде полной альтернативы. Соответствующий вариант программы мы предлагаем написать читателю самостоятельно.

Вернемся еще раз к примеру 5.1. В представленном коде была произведена суперпозиция двух структур выбора. Чтобы обеспечить строгое следование правилу структурного программирования «один вход – один выход», весь основной текст программы нам пришлось поместить в составной оператор, что и показывает комментарий. Однако с чисто практической точки зрения это не слишком удобно. Отчасти потому, что при соблюдении правила отступов текст программы постоянно сдвигается вправо, что ухудшает ее «читабельность». Но главное, сильная синтаксическая связь основного текста с интерфейсной частью (часть программы, в которой осуществляется взаимодействие с пользователем) неоправданна, так как они часто модифицируются независимо. Поэтому здесь вполне допустимо отойти от «структурного экстремизма» и допустить «обрывы» по управлению. Поскольку в приведенной реализации выполнение программы прекращается при ошибочном вводе числа, такой «обрыв» управления мы можем реализовать специальной процедурой

останова Halt. Кроме того, условные операторы в этом случае разумно использовать в сокращенной форме. Получится вот что:

```
{ ================================================= }
{ Пример 5.3 }
{ Контроль вводимой информации - вариант 2 }
Program Control2;

{$APPTYPE CONSOLE}

var
   Day: Integer;
begin
   {$I-}
   { отключили автоматическую проверку ввода-вывода }
   ReadLn(Day);
   if (IOResult <> 0) then { ошибка ввода }
   begin
    WriteLn('Ошибка 1: Недопустимые символы в целом числе');
    Halt;
   end;
   if (Day < 1) or (Day > 31) then
   begin
    WriteLn('Ошибка 2: День месяца вне диапазона 1..31');
    Halt;
   end;
   { Дальнейший текст программы }
end.
{ ================================================= }
```

В приведенных в этом разделе примерах мы уже несколько раз использовали операторы выбора, условные выражения в которых содержали логические операции **and** и **or**. Как известно, для получения результата в таких выражениях не обязательно подсчитывать значения обоих операндов. Например, если вычисленный первым операнд операции **or** имеет значение **True**, то вне зависимости от значения второго операнда результат всей операции также равен **True**. Если принять такой подход неполного вычисления булевского выражения, то это не только сократит время работы программы, но и обеспечит ее более компактную и ясную запись.

Рассмотрим пример. Допустим, некоторая программа использует в качестве входной информации целое число, которое должно быть делителем (нацело) другого целого числа (пусть, например, речь идет о числе равноправных потребителей некоторого ресурса). Фрагмент

программы, обеспечивающий контроль вводимого числа (количества потребителей), выглядит следующим образом:

```
{ ================================================= }
{ Пример 5.4 }
{ Неполное вычисление логического выражения }
Progam Calculation;

{$APPTYPE CONSOLE}

const
   Resourses = 28;
var
   Consumers: Integer;
begin
   Writeln('Количество потребителей ресурса: ');
   {$I-}
   Readln(Consumers);
   if (IOResult <> 0) or (Resourses mod Consumers <> 0) or
      (Consumers < 1) then
   begin
     WriteLn('Ошибка 1: Недопустимое число потребителей');
     Halt;
   end;
   { Продолжение программы }
end.
{ ================================================= }
```

Использованный в коде оператор выбора одновременно проверяет синтаксическую корректность введенного числа и ограничения на его значение. Если бы булевское выражение вычислялось полностью, то могла бы возникнуть следующая неприятная ситуация. При вводе синтаксически неправильного числа оператор `ReadLn` состояния переменной `Consumers` не изменит, то есть ее значение будет определяться состоянием отведенного ей в момент запуска программы блока оперативной памяти. Вполне возможно, что это значение будет равно нулю. Следовательно, при попытке вычислить второй операнд условного выражения в операторе выбора весьма вероятна выдача стандартного системного сообщения о делении на ноль, что, конечно, недопустимо.

В языке Pascal по умолчанию действует директива компилятора `{$B-}`, устанавливающая неполное вычисление условных выражений. Таким образом, приведенная программа будет работать корректно. Если по каким-

либо причинам требуется полное вычисление всех операндов условных выражений, необходимо использовать директиву `{$B+}`.

В качестве окончания данного раздела приведем одну практическую рекомендацию по способу записи оператора выбора. Очевидно, что любой условный оператор можно записать в двух вариантах, отличающихся порядком секций **then** и **else**. К примеру, если нам необходимо проверить, не равна ли нулю переменная *a*, то это можно сделать так:

```
if a = 0 then
   <оператор1>
else
   <оператор2>;
```

А можно так →

```
if a <> 0 then
   <оператор2>
else
   <оператор1>;
```

На первый взгляд кажется, что между этими вариантами нет никакой разницы. Тогда немного изменим ситуацию. Пусть если *a* = 0, необходимо выполнить много действий, а в противоположном случае одно. Тогда первый вариант примет следующий вид:

```
if a = 0 then
begin
   <оператор1>
   ...
   <операторN>
end
else
   <операторN+1>;
```

А второй вариант будет выглядеть так:

```
if a <> 0 then
   <операторN+1>
else begin
   <оператор1>
   ...
   <операторN>
end
```

Как вы думаете, какой вариант лучше? Конечно же, второй! В этом случае ключевые слова секций условного оператора располагаются рядом друг с другом, тогда как в первом варианте, чтобы найти секцию **else**, потребуется приложить некоторые усилия. Поверьте, их можно потратить с большей пользой, чем разыскивать «убежавшую» секцию.

4.2. Выбор из нескольких вариантов

Составим программу, реализующую функциональность простейшего калькулятора с операциями +, -, *, /. Программа должна запрашивать два числа, операцию, которую необходимо выполнить, и вычислять результат. При выполнении операции деления необходимо проверить делитель на равенство нулю.

```
{ ================================================ }
{ Пример 5.5 }
{ Простейший калькулятор }
Program Calculator;

{$APPTYPE CONSOLE}

var
   x, y: Real;
   Res: Real;
   Op: Char;
begin
   WriteLn('Введите операцию (+, -, *, /): ');
   ReadLn(Op);
   WriteLn('Введите аргументы: ');
   ReadLn(x, y);
   if (Op = '+') then
     Res := x + y
   else
      if (Op = '-') then
        Res := x - y
      else
         if (Op = '*') then
           Res := x * y
         else
           if (Op = '/') then
           begin
             if (y <> 0) then
               Res := x / y
             else begin
               WriteLn('Деление на ноль');
               Halt;
             end;
           end
           else begin
              WriteLn('Неизвестная операция');
              Halt;
           end;
   WriteLn(x, ' ', Op, ' ', y, ' = ', Res);
end.
```

```
{ ================================================ }
```

Мы использовали рассмотренный в предыдущем разделе оператор выбора в виде полной альтернативы. Чтобы не дублировать вывод результата, в программе пришлось предусмотреть две дополнительные точки выхода с использованием процедуры `Halt`. Достаточно очевидно, что предложенный вариант обладает рядом недостатков. Если потребуется увеличить количество поддерживаемых калькулятором операций, текст программы будет уходить вправо «за горизонт». Постоянный повтор фактически одной и той же строки вида

```
if (Op = 'символ операции') then
```

тоже не доставляет удовольствия. И наконец, количество строк данной программы, в которых выполняются реальные действия, меньше, чем число строк, содержащих чисто служебную информацию (вроде операторных скобок). Чем вызваны все эти неприятности? Мы столкнулись с явным противоречием: алгоритм в данной программе – это, по сути, выбор из нескольких однотипно описываемых альтернатив, а для его записи у нас в распоряжении лишь оператор выбора из двух вариантов. Если попытаться изобразить блок-схему алгоритма, получится примерно следующее.

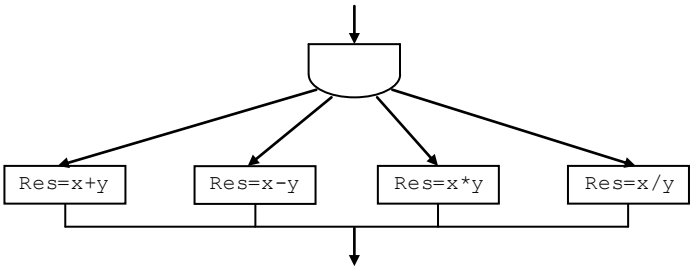


Рис. 5.4. Блок-схема калькулятора

Подобные алгоритмы встречаются весьма часто. Например, по такой же схеме устроен алгоритм обработки команд меню – задача, которую приходится в том или ином виде решать в любой более-менее сложной программе. К счастью, язык Pascal содержит прямую реализацию алгоритма выбора из нескольких вариантов – *оператор множественного выбора*.

Формальная запись *оператора множественного выбора* выглядит следующим образом:

```
case <выражение порядкового типа> of
  <список выбора 1> : <оператор1>;
```

```
    <список выбора 2> : <оператор2>;
     ...
    <список выбора N> : <операторN>;
    [else <операторN+1>;]
end;
```

Здесь «список выбора» – это:

- константа того порядкового типа, который соответствует результату выражения после ключевого слова Case;
- список констант через запятую;
- тип диапазон, образованный из констант соответствующего порядкового типа.

При этом любая константа не может содержаться более чем в одном списке выбора.

Например:

```
var
   a, b: Integer;
   ...
case a of
   1      : b := a + 1;
   3, 4   : b := a - 1;
   5..10  : b := a * a;
   else     b := a;
end;
```

К порядковым типам в языке Pascal относятся целые типы, символьный и булевский типы, перечислимый тип, тип «диапазон»[1].

Оператор множественного выбора выполняется следующим образом. Вычисляется значение выражения. Если это значение содержится в одном из списков выбора, то выполняется соответствующий оператор. Если значение не принадлежит ни одному из списков выбора, тогда если имеется секция **else**, то выполняется оператор этой секции, в противном случае оператор **case** не производит никакого действия.

Используя этот оператор, пример 5.5 можно переписать следующим образом:

```
{ ================================================ }
{ Пример 5.6 }
{ Простейший калькулятор - вариант 2 }
Program Calculator2;
```

[1] Подробнее перечислимый тип и тип «диапазон» будут рассмотрены в следующей главе.

```
{$APPTYPE CONSOLE}

var
   x, y: Real;
   Res: Real;
   Op: Char;
begin
   WriteLn('Введите операцию (+, -, *, /): ');
   ReadLn(Op);
   WriteLn('Введите аргументы: ');
   ReadLn(x, y);
   case Op of
     '+' :Res := x + y;
     '-' :Res := x - y;
     '*' :Res := x - y;
     '/' :if (y <> 0) then
             Res := x / y
          else begin
            WriteLn('Деление на ноль');
            Halt;
          end;
     else begin
       WriteLn('Неизвестная операция');
       Halt;
     end;
   end;
   WriteLn(x, ' ', Op, ' ', y, ' = ', Res);
end.
{ ================================================= }
```

Как видим, текст программы стал заметно короче, понятнее, избавился от лишних элементов. Да и выполнить расширение функциональности калькулятора теперь будет не в пример проще.

Заметим также, что оператор множественного выбора соответствует схеме «один вход – один выход», а значит, может использоваться в качестве элемента в любой из базисных алгоритмических конструкций.

5. Программирование цикла

5.1. Цикл с предусловием

Рассмотрим задачу вычисления факториала целого неотрицательного числа. Согласно определению *факториал* натурального числа N есть произведение всех чисел от 1 до N, то есть

$$N! = 1 \cdot 2 \cdot \ldots \cdot N,$$

$$0! = 1.$$

Факториал нуля по определению полагается равным единице.

Составим алгоритм для решения этой задачи. Первое очевидное действие – задание требуемого числа N (с проверкой его корректности в общем случае). Кажется, что вторым шагом должна идти проверка ситуации $N = 0$, поскольку формула вычисления факториала для нуля не совпадает с формулой для остальных чисел. Вернемся к этому позже. Если $N > 0$, то для получения результата нужно несколько раз (а именно N) повторить одно и то же действие – умножение. Очевидно, нам потребуется алгоритмическая конструкция *цикл*. Однако, хотя действие, которое будет выполняться в цикле, и одно и то же, то, над чем это действие будет осуществляться, каждый раз будет меняться. На первой итерации цикла нужно умножить 1 на 1, на второй – 1 (результат предыдущей итерации) на 2, на третьей – 2 (результат предыдущей итерации) на 3, на четвертой – 6 на 4, на пятой – 24 на 5 и так далее.

Из сказанного можно сделать важный вывод: *алгоритмическая формулировка цикла требует не только выявления повторяемого действия, но и соответствующего однообразного представления обрабатываемых в цикле данных*. Последнее, собственно, и представляет основную «интеллектуальную» проблему программирования цикла.

Итак, в нашей задаче вычисления факториала необходимо предложить однообразное представление сомножителей для каждой итерации. Известно, что формулу расчета факториала можно записать так: $N! = (N - 1)! * N$. Представим себе, что нам осталось выполнить последнюю (с номером N) итерацию цикла, тогда действие на этой итерации будет состоять в следующем: результат предыдущей итерации умножить на номер текущей итерации. Нетрудно заметить, что формулировка действия никак не связана с тем, что эта итерация была последней, и может быть с успехом применена на всех остальных. Для ее реализации нам потребуется переменная

`i` – счетчик числа итераций и переменная `F` – результат предыдущей итерации. Таким образом, действия, повторяемые в цикле (так называемое *тело цикла*), будут иметь вид:

```
   F = F * i,
   i = i + 1.
```

Последний важный вопрос формулировки цикла – установка начальных значений переменных и условия выхода. В нашем случае логично присвоить начальное значение переменной `F = 1` (минимальное значение факториала). Чтобы задать начальное значение счетчика `i`, необходимо определиться с условием останова цикла. Особенность рассматриваемой задачи состоит в том, что если $N = 0$, то умножение производить не надо, так как правильное значение факториала обеспечено начальной установкой переменной `F`. Таким образом, если мы установим начальное значение счетчика $i = 1$, то условие выхода $i \le N$ обеспечит невыполнение цикла при $N = 0$ и правильный подсчет числа умножений при $N \ge 1$.

Итак, все необходимые для построения алгоритма решения приняты. В результате мы можем представить следующий псевдокод:

```
вводим N
   i = 1
   F = 1
пока i ≤ N выполнять
начало
   F = F * i
   i = i + 1
конец
вывод значения F
```

Использованная нами конструкция в точности соответствует последней из базисных алгоритмических структур – *структуре цикла*. Реализация этой структуры в языке Pascal выполнена следующим образом:

```
while <условное выражение> do
   <оператор>;
```

При этом, как и в *операторе выбора*, условное выражение должно иметь *булевский* тип. Пока результат вычисления выражения – **True** (истина), выполняется оператор. При необходимости выполнять в теле цикла несколько действий, нужно использовать *составной оператор*. Обращаем внимание на то, что внутри цикла должны содержаться операторы, изменяющие значение условного выражения так, чтобы оно, в конце концов, стало равно **False**.

Приведем теперь программу вычисления факториала.

```
{ ================================================= }
{ Пример 5.7 }
{ Вычисление факториала }
Program Factorial;
```

```
{$APPTYPE CONSOLE}

var
   N, i: Word;
   F: Longword;
begin
   {$R+}
   Write('Аргумент: ');
   Readln(N);
   i := 1;
   F := 1;
   while i <= N do
   begin
      F := F * i;
      i := i + 1;
   end;
   WriteLn(N, '! = ', F);
   ReadLn;
end.
{ ================================================ }
```

Чтобы не загромождать пример явными проверками корректности, при объявлении переменной *N* мы использовали тип `Word` (неотрицательные числа) и воспользовались директивой `{$R+}`, которая обеспечит вставку в исполняемый модуль команд проверки на принадлежность введенного значения *N* диапазону типа `Word`. Для переменной `F` тип `Word` является не слишком подходящим, поскольку факториал – очень быстро растущая функция (диапазон типа `Word` от 0 до 65 535). Впрочем, и использованный нами тип `LongWord` с диапазоном от 0 до 4 294 967 295 также позволяет корректно вычислять факториал лишь для небольших значений *N*. Попробуйте, например, последовательно вычислить с использованием данной программы факториал для чисел 11, 12 и 13.

Вообще говоря, представленный код является не совсем корректным, поскольку не предусматривает проверки на максимально возможное значение *N*, при котором программа еще способна выдать правильный результат. Однако выполнить такую проверку не так-то просто. Как именно это можно сделать, мы обсудим чуть позднее.

5.2. Цикл с постусловием

Вернемся еще раз к задаче о контроле получаемой от пользователя информации. Как мы уже отмечали выше, ввод данных является потенциально опасным местом с точки зрения выполнения программы – пользователь может случайно (или намеренно, если он – тестер программы) задать неверные данные, на которых программа не сможет продолжать работу. Однако, с точки зрения пользователя, тот факт, что он ошибся при вводе, не может служить основанием для аварийного завершения программы. Налицо явное противоречие. Самый правильный подход в данной ситуации – не дать пользователю ошибиться вообще. Например, при запросе дня месяца программа может показывать на экране календарь и предлагать пользователю *выбрать* нужный день. Понятно, что в этом случае выбрать 32-е число пользователю не удастся при всем желании. К сожалению, такой подход не всегда возможен. Если список альтернатив заранее неизвестен и выбор предоставить нельзя, тогда то, что ввел пользователь, необходимо проверять. Проверять на соответствие типов (требовалось ввести число, а ввели букву) и на допустимость (попадание в диапазон или в некоторый допустимый набор вариантов). Программирование таких проверок мы рассмотрели в предыдущем разделе. Там же было отмечено, что кроме проверки корректности необходимо предоставить пользователю возможность повторить ввод в случае ошибки. Теперь мы можем реализовать такой повтор. Перепишем код примера 5.3 так, чтобы обеспечить правильный ввод данных.

```
{ ================================================== }
{ Пример 5.8 }
{ Контроль вводимой информации – вариант 3 }
Program Control3;

{$APPTYPE CONSOLE}
var
   Day: Integer;
   IORes: Integer;
begin
   {$I-}
   { отключили автоматическую проверку ввода-вывода }
   Write('Введите день месяца: ');
   ReadLn(Day);
   IORes := IOResult;
   while (IORes <> 0) or (Day < 1) or (Day > 31) do
   begin
      if (IORes <> 0) then
        WriteLn('Ошибка 1: Недопустимые символы в целом числе')
      else
        WriteLn('Ошибка 2: День месяца вне диапазона 1..31');
```

```
      Write('Введите день месяца: ');
      ReadLn(Day);
      IORes := IOResult;
   end;
   { Дальнейший текст программы }
end.
{ ================================================= }
```

Проанализируем представленный вариант программы.

Во-первых, исчезли оба досрочных выхода с помощью процедуры `Halt`. Теперь программа добивается от пользователя получения верных данных, вместо завершения работы в случае ошибки.

Во-вторых, после первоначального ввода значения переменной `Day` запускается цикл **`while`**, проверяющий возможные ошибки. Внутри цикла, если ошибка имела место, она классифицируется, выдается соответствующее сообщение, после чего ввод повторяется.

В-третьих, в коде появилась новая переменная `IORes`, в которую сохраняется значение, возвращаемое функцией `IOResult`. Вызвано это тем, что функция `IOResult` не только возвращает состояние последней операции ввода-вывода, но и обнуляет внутренний флаг ошибки. Таким образом, повторный ее вызов, необходимый в условном операторе, сообщающем пользователю вид ошибки, не привел бы к нужному эффекту.

И наконец, представленный вариант обзавелся повтором из трех одинаковых строк:

```
Write('Введите день месяца: ');
ReadLn(Day);
IORes := IOResult;
```

Эти строки выполняются до цикла – первоначальный ввод и внутри цикла – повторный ввод. Неприятность эта – прямое следствие конструкции цикла **`while`**. Цикл **`while`** является *циклом с предусловием* – сначала проверка, потом действие. А условное выражение, на основании значения которого принимается решение об остановке, естественно, не может содержать неизвестных (неопределенных) переменных.

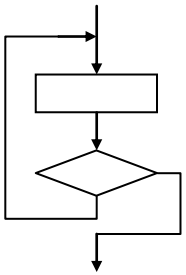


Рис. 5.5. Блок-схема цикла с постусловием

Вместе с тем в данном случае намного логичнее было бы использовать цикл, организованный по противоположному принципу – сначала действие, затем проверка и при

необходимости повтор, то есть *цикл с постусловием*.

Ситуации, в которых требуется подобная схема организации цикла, встречаются не так уж редко, а строк кода, которые необходимо будет продублировать, если использовать для ее программирования цикл **while**, может оказаться много больше, чем три. Таким образом, наличие в языке программирования реализации *цикла с постусловием* является весьма полезным. В языке Pascal такая реализация есть и выглядит она следующим образом:

```
repeat
   <оператор 1>;
   <оператор 2>;
   ...
   <оператор N>;
until <условное выражение>;
```

Дословный перевод с английского этой конструкции: «повторять … пока не …». Таким образом, в отличие от *цикла с предусловием* **while**, цикл **repeat** выполняется, пока результат вычисления условного выражения – **False** (ложь). Кроме того, в теле цикла **repeat** может располагаться несколько операторов – в данном случае в качестве *операторных скобок* выступают ключевые слова **repeat** и **until**.

С использованием цикла с постусловием пример 5.8 может быть переписан следующим образом:

```
{ ================================================ }
{ Пример 5.9 }
{ Контроль вводимой информации - вариант 4 }
Program Control4;

{$APPTYPE CONSOLE}

var
   Day: Integer;
   IORes: Integer;
begin
   {$I-}
   { отключили автоматическую проверку ввода-вывода }
   repeat
    Write('Введите день месяца: ');
    ReadLn(Day);
    IORes := IOResult;
    if (IORes <> 0) then
      WriteLn('Ошибка 1: Недопустимые символы в целом числе')
    else
      if (Day < 1) or (Day > 31) then
        WriteLn('Ошибка 2: День месяца вне диапазона 1..31');
```

```
   until (IORes = 0) and (Day >= 1) and (Day <= 31);
   { Дальнейший текст программы }
end.
{ ================================================= }
```

Итак, после некоторых усилий мы получили, наконец, код, который в точности соответствует требуемой функциональности в задаче о контроле ввода информации: ввели значение; проверили на корректность; если были ошибки, сообщили о них и вернулись к вводу; если не было, вышли из цикла и продолжили работу.

5.3. Цикл с известным числом повторений

Описанные в двух предыдущих разделах операторы цикла **while** и **repeat** являются универсальными и позволяют запрограммировать любой алгоритм, связанный с выполнением повторяющихся действий. Однако среди всех таких алгоритмов существует очень большой класс, в котором циклическая обработка выполняется заранее известное количество раз и чаще всего над элементами некоторого упорядоченного множества.

Все алгоритмы этого класса обладают общим устройством. В них требуется счетчик числа итераций и задание для этого счетчика начального и конечного значений. При этом конечное значение определяет условие останова цикла, а сам счетчик на каждой итерации меняется на одно и то же число, чаще всего равное единице. Одним из примеров такого алгоритма является рассмотренный выше алгоритм вычисления факториала.

Вследствие перечисленных обстоятельств язык Pascal (как и многие другие языки программирования) содержит специальный оператор цикла для программирования алгоритмов подобного вида – оператор *цикла с известным числом* повторений **for**.

Формальная запись оператора **for** выглядит следующим образом:

```
for <счетчик цикла> := <начальное значение>
  to <конечное значение> do <оператор>;
```

При этом счетчик цикла – переменная *порядкового* типа, начальное значение и конечное значение – арифметические выражения, результат каждого из них должен быть того же типа, что и счетчик, или должен быть совместим с ним по присваиванию. В конце каждой итерации счетчик цикла автоматически увеличивается на единицу. Изменение счетчика в теле цикла не допускается[1]. Количество итераций, которое будет выполнено,

[1] Сказанное верно для Object Pascal, более ранние версии языка позволяли менять значение счетчика цикла в теле цикла.

вычисляется до входа в цикл, таким образом, изменения в теле цикла переменных, входящих в начальное значение и конечное значение, не приводят к изменению числа итераций. Более того, число итераций, которое выполняет цикл **for**, всегда[1] равно конечное значение – начальное значение + 1.

Кроме указанной формы цикл **for** имеет второй вариант:

```
for <счетчик цикла> := <начальное значение>
  downto <конечное значение> do <оператор>;
```

В этом варианте в конце каждой итерации счетчик цикла автоматически *уменьшается* на единицу.

Если к моменту входа в цикл начальное значение оказалось больше (во второй форме цикла меньше), чем конечное значение, то цикл не выполняется ни разу.

Попытаемся теперь переписать пример 5.7 с использованием цикла **for**.

```
{ ================================================ }
{ Пример 5.10 }
{ Вычисление факториала - вариант 2 }
Program Factorial2;

{$APPTYPE CONSOLE}

var
   N, i: Word;
   F: Longword;
begin
   {$R+}
   Write('Аргумент: ');
   ReadLn(N);
   F := 1;
   for i := 1 to N do
      F := F * i;
   WriteLn(N, '! = ', F);
   ReadLn;
end.
{ ================================================ }
```

[1] Поскольку в более ранних версиях языка Pascal смена значения счетчика цикла внутри тела цикла разрешалась, то число итераций цикла for можно было изменить. В Object Pascal это стало невозможным.

Как видим, код стал существенно короче – вся работа со счетчиком итераций, которую в примере 5.7 мы программировали явным образом, теперь выполняется автоматически.

Приведем еще один пример не вполне очевидного использования цикла **for**. Пусть мы хотим распечатать коды символов букв английского алфавита от `'A'` до `'Z'`. Фрагмент кода для этой задачи может выглядеть следующим образом:

```
var
   ch: Char;
...
for ch := 'A' to 'Z' do
   WriteLn('For simbol ', ch, ' code is ', Ord(ch));
```

Здесь используется тот факт, что тип `Char` относится к порядковым, а значит, вполне может служить типом счетчика цикла **for**.

5.4. Выход из тела цикла

Еще раз вернемся к задаче о вычислении факториала. Представленные в примерах 5.7 и 5.10 программы допускают лишь однократное выполнение. Однако на практике пользователю может потребоваться вычислить факториал не для одного, а для нескольких чисел. Конечно, можно в самом начале программы спросить, сколько факториалов нужно будет рассчитать, и весь имеющийся расчетный код заключить в цикл с известным числом повторений **for**. Однако это решение не вполне корректно. Общепринятым способом использования программ является вариант, в котором программа работает до тех пор, пока пользователь не решит из нее выйти. Таким образом, обычно тело программы (кроме начальных и финальных действий, которые выполняются однократно) заключается в тело цикла **repeat**, в конце которого задается вопрос вида: «Продолжить (Да/Нет) ?». В данном случае мы можем сэкономить и запрос числа N совместить с вопросом об окончании работы. Поскольку $N = -1$ является недопустимым значением, вопрос на ввод информации может выглядеть так:

```
Write('Аргумент (-1 - для завершения): ');
```

Такое решение приводит к тому, что в теле цикла **repeat** нам потребуется проверка. По соображениям, указанным в разделе «Программирование выбора», оптимальный вариант такой проверки:

```
if N = -1 then
  ...;
```

Осталось только решить, что должно стоять на месте трех точек. Если после цикла **repeat** никаких действий больше делать не надо, то можно использовать такой вариант:

```
if N = -1 then
  Halt;
```

А если надо? Как быть в этом случае? Хотелось бы вместо `Halt` подставить что-то, что позволило бы выйти из цикла досрочно. То есть реализовать следующую схему (рис. 5.6).

К сожалению, в языке Pascal такой оператор отсутствует. Тем не менее указанную схему можно запрограммировать.

Во-первых, для этого можно использовать *оператор безусловного перехода* **goto**.

Формат этого оператора:

```
goto <метка>;
```

где метка – целое число в диапазоне от 0 до 9999 или *идентификатор*. Метки должны быть объявлены в собственной секции объявлений **label**, где они просто перечисляются через запятую.

```
label
  1, 345, EndLoop;
```

Метка может быть поставлена в начале любой строки тела программы, содержащей оператор (в том числе *пустой оператор*). От остальной строки метка отделяется двоеточием.

Оператор **goto** передает управление на строку, отмеченную меткой.

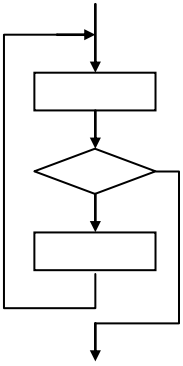


Рис. 5.6. Блок-схема цикла с выходом из тела цикла

Используя этот оператор, мы можем представить следующий код:

```
{ ================================================ }
{ Пример 5.11 }
{ Вычисление факториала – вариант 3 }
```

```
Program Factorial3;

{$APPTYPE CONSOLE}
label l;
var
   i: Word;
   N: Integer;
   F: LongWord;
begin
   {$R+}
   repeat
      Write('Аргумент (-1 - для завершения): ');
      Readln(N);
      If N = -1 then
         goto l;
      F := 1;
      for i := 1 to N do
         F := F * i;
      WriteLn(N, '! = ', F);
   until False;
l: ;
end.
{ ================================================ }
```

Оператор безусловного перехода существует в языках программирования очень давно. Без него невозможно программирование на языке низкого уровня Assembler, также весьма активно он используется в большинстве версий языка Basic. Однако повсеместное применение этого оператора в программах делает их код чрезвычайно запутанным, сложным для прочтения и поиска ошибок и, хотя во многих языках высокого уровня этот оператор существует, использовать его не рекомендуется. Помимо прочего, оператор безусловного перехода противоречит положениям технологии структурного программирования, поскольку из любой алгоритмической конструкции позволяет выйти, что называется «через окно». Таким образом, прежде чем использовать оператор **goto**, убедитесь, что другого способа реализации нужного вам алгоритма нет.

В силу указанных обстоятельств, а также поскольку конструкция цикла, в которой решение о продолжении или выходе требуется принимать в его середине, весьма распространена, в реализацию языка Pascal от фирмы Borland, начиная с версии 7.0, ввели, а в языке Object Pascal сохранили специальные *операторы управления циклом*.

Первый из них – оператор **break** – представляет второй способ реализовать конструкцию, изображенную в виде блок-схемы на рисунке 5.6. Его вызов обрывает выполнение цикла и передает управление на

оператор, стоящий *сразу за* циклом. В этом принципиальное отличие от оператора безусловного перехода **goto**, с помощью которого управление может быть передано в любую точку программы, в том числе стоящую выше по тексту.

Второй оператор управления циклом – **continue** – предназначен для досрочного окончания не всего цикла, а только текущей итерации. При его вызове выполнение итерации прекращается, управление передается на начало цикла и начинается следующая итерация. В цикле **for** при этом, как и положено, происходит изменение счетчика цикла.

Использование оператора **break** вместо **goto** в примере 5.11 позволит избавиться от объявления метки и пустого оператора в конце программы, то есть сократит текст на три строки.

6. Расширенный пример

В конце главы рассмотрим полное решение задачи, с которой мы начали раздел «Программирование цикла».

Все представленные в предыдущих примерах программы вычисления факториала обладают одним существенным недостатком, который никак не связан с расчетным алгоритмом, а вытекает лишь из выбранного способа хранения результата (тип переменной *F*) и того факта, что любой числовой тип данных представляет не все числовое множество (целое или вещественное), а лишь ограниченную его часть.

В данном случае проблема состоит в том, что какой бы тип мы ни выбрали для переменной *F*, рано или поздно мы «упремся» в верхнюю границу значений этого типа, после чего программа перестанет выдавать верные результаты. Решение, которое первым приходит в голову, – в самом начале программы подсчитать максимально допустимое *N*, соответствующее выбранному типу переменной *F*, сообщить об этом пользователю и проверять ввод переменной *N*. Решение правильное со всех точек зрения, осталось лишь понять, как воплотить его в жизнь.

Возникшую задачу можно представить математически. Имеется монотонно возрастающая функция $F(N)$. Дано число FMax. Требуется найти максимальное *N*, при котором $F(N) \le \text{FMax}$. Получилась задача на решение неравенства от одной переменной. Однако в силу сложности формулы $F(N)$ найти решение аналитически будет весьма непросто.

Возможный численный вариант состоит в следующем. Будем последовательно увеличивать *N*, начиная с 1, подсчитывать значение факториала и так до тех пор, пока $F(N)$ не превысит первый раз значения FMax. Все правильно? Математически абсолютно. Вот только программу

по такому принципу составить нельзя. Проблема как раз в том и состоит, что за FMax в типе данных ничего больше нет. Кажется, мы зашли в тупик. Или нет? Напрягаем интеллект и… Эврика! Если нельзя «перейти» через границу FMax, надо остановиться в тот момент, когда следующий шаг сделать будет невозможно. В такой постановке проблема решается: будем на каждой итерации делить FMax на текущее значение *F* нацело и сравнивать частное со следующим *N*. Как только частное станет меньше, чем *N*, пора остановиться.

Наконец, для того чтобы программа была максимально корректной, добавим в нее контроль за вводом числа *N*. В результате получится следующий код.

```
{ ================================================== }
{ Пример 5.12 }
{ Вычисление факториала - вариант 4 }
Program Factorial4;

{$APPTYPE CONSOLE}

var
   N: Integer;
   NMax, i: Word;
   F, FMax: LongWord;
   IORes: Integer;


begin
   {$I-}
   FMax := High(LongWord);
   F := 1;
   i := 1;
   while (FMax div F >= i) do
   begin
      F := F * i;
      Inc(i);
   end;
   NMax := i - 1;
   repeat
      Write('Аргумент (-1 - для завершения): ');
      Readln(N);
      if N = -1 then
         break;
      IORes := IOResult;
      if (IORes <> 0) then
      begin
```

```
        WriteLn('Ошибка 1: Недопустимые символы в целом числе')
        Continue;
      end
      else
        if (N < -1) or (N > NMax) then
        begin
          WriteLn('Ошибка 2: Аргумент вне диапазона 0..', NMax);
          Continue;
        end;
      F := 1;
      for i := 1 to N do
         F := F * i;
      WriteLn(N, '! = ', F);
   until False;
   ReadLn;
end.
{ ================================================== }
```

7. Выводы

В данной главе мы перекинули мостик от написания «игрушечных» программ к проектированию и созданию полноценных программных продуктов, в общих чертах познакомившись с понятием технологии программирования. Нами был получен ответ на важный вопрос о том, что такое разработка программ. С каждым днем становится все больше сторонников того мнения, что разработка программ есть технологический процесс со своими методами, приемами, средствами – технологиями, обеспечивающими повышение производительности труда программистов и способствующими достижению результата. Хорошо известен один из так называемых законов Мерфи: «Любая программа обходится дороже и требует больших затрат времени, чем предполагалось». Наша задача как специалистов в данной области – побороть эту закономерность в максимально возможной степени, и структурное программирование, с основным положениями которого мы познакомились в данной главе, – первый шаг в этом направлении.

Литература

1. Богатырев Р. Летопись языков Паскаль // Мир ПК. –№ 4.–2001.

2. Богатырев Р. От Паскаля к языку Zonnon: реализация новых идей на платформе .NET // Мир ПК.–2003.–№ 9.

3. Большая советская энциклопедия. Издание третье. – М.: Советская энциклопедия, 1970.

4. Брэдли Д. Программирование на языке ассемблера для персональной ЭВМ фирмы IBM.–М.: Радио и связь, 1988.

5. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. – Бином, 1998.

6. Касперски К. Техника оптимизации программ. Эффективное использование памяти.– СПб.: БХВ-Петербург, 2003.

7. Кетков А., Кетков Ю. Практика программирования: Бейсик, Си, Паскаль. Самоучитель.–СПб.: БХВ-Петербург, 2001.

8. Кетков А., Кетков Ю. Практика программирования: Visual Basic, C++ Builder, Delphi. Самоучитель.–СПб.: БХВ-Петербург, 2002.

9. Кнут Д.Э. Искусство программирования. Т. 1. Основные алгоритмы.–М.:Вильямс, 2000.

10. Кнут Д.Э. Искусство программирования. Т. 3. Сортировка и поиск.–М.:Вильямс, 2004.

11. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ.–М.:МЦНМО, 1999.

12. Кэнту М. Delphi 7 для профессионалов.–Спб.:Питер, 2004.

13. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования.–М.: Мир, 1982.

14. Модернизация и ремонт персонального компьютера.– [http://www.allcompinfo.com]

15. Новичков А. Система генерации проектной документации Rational SoDA.– [http://www.citforum.ru/programming/digest/soda.shtml]

16. Олифер В.Г., Олифер Н.А. Сетевые операционные системы.–СПб.: Питер, 2002.

17. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ.– 4-е изд.–СПб.: Питер; М.: Издательско-торговый дом «Русская Редакция», 2001.

18. Служба тематических толковых словарей. – [http://www.glossary.ru/]

19. Стивенс Р. Delphi. Готовые алгоритмы.– Спб.:Питер, 2004.

20. Тейксейра С., Пачеко К. Borland Delphi 6. Руководство разработчика. –Вильямс, 2002.

21. Шень А. Программирование: теоремы и задачи.–М.:МЦНМО, 2004.

22. Юров В. Assembler.–СПб.: Питер, 2002.

23. Cantu M. Essential Delphi.– [www.marcocantu.com/edelphi]

24. Cantu M. Essential Pascal.– [www.marcocantu.com/epascal]

25. Cantu M. Mastering Borland Delphi 2005.–Sybex, 2005.

26. Cantu M. Mastering Delphi 7.–Sybex, 2003.

27. Clark R., Koehler S. The UCSD Pascal Handbook.–Prentice-Hall, 1982.

28. Dahl O., Dijkstra E., Hoare C.A.R. Structured Programming.–London, England: Academic Press, 1972.

29. Gutknecht J., Zueff E. Zonnon Language Report. Draft.–ETH Zurich, June 2003.

30. Gutknecht J., Zueff E. Zonnon for .NET, A Language and Compiler Experiment. // Computer Systems Institute, ETH Zürich, Switzerland. JMLC Conference.–August 2003.

31. Gutknecht J. Zonnon: A .NET Language Beyond C# // Moscow: Microsoft Conference, June 15-17, 2003.

32. Gutknecht J., Wirth N. The Oberon System // Software – Practice and Experience.–Vol.19, № 9.–1989.–p.857-893.

33. Jacobson I., Christerson M., Jonsson P., Overgaard G. Object-oriented Software Engineering.–Workingham, England: Addison-Wesley Publishing Company, 1992.

34. Jensen K., Wirth N. PASCAL – User Manual and Report, ISO Pascal Standard.–1974.

35. Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture.– Intel Corporation: 1997.

36. Meyer B. Object-oriented Software Construction.–Prentice Hall, 1988.

37. Rumbaugh H., Blaha M., Premarlani W., Eddy F., Lorensen W. Object-Oriented Modeling and Design.–Prentice-Hall, 1995.

38. Sedgewick R. Algorithms.–Reading, MA: Addison-Wesley, 1983.

39. Tesler L. Object Pascal Report. // Structured Language World, Vol.9, No.3.– 1985.–p. 10-14.

40. Thiriez H. Modelling of an interactive scheduling system in a complex environment // European Jornal of Operational Research. –1991.– №50.–p.37-47.

41. Wirth N. Program Development by Stepwise Refinement // Communications of the ACM vol.26(1).– January 1983.

42. Wirth N. The Programming Language Pascal // Acta Informatica, 1.– 1971.–p. 35-63.

43. Wirth N. Programming in Modula-2.–Springer-Verlag, 1982.

44. Wirth N. The programming language Oberon // Software – Practice and Experience, Vol.18, № 7.–1988.–p.671-690.

45. Wirth N. Programming in Modula-2.–Springer, 1974.

46. Wirth N. Pascal and its Successors // Conference on Computer Pioneers.–Bonn, 2001.