ТУБ №36:
Дополнительные операторы языка Delphi

Появление структурного программирования стало важным шагом на пути к пониманию того, как разрабатывать надёжные программы, т.е. такие, которые не будут ломаться от простейших правок или даже одного только взгляда пользователя.

Сама теория структурного программирования, разумеется, не была привязана к конкретным языкам программирования или их операторам (на то она и теория). Тем не менее, прямым следствием из неё стали вполне определённые *рекомендации по разработке программ*. Так в рамках структурного программирования предполагается, что все действия, выполняемые программой, должны представлять собой:

- исполняемые в линейном порядке выражения;
- вызовы подпрограмм (обращения к замкнутым участком кода с одним входом и одним выходом);
- вложенные на произвольную глубину операторы `if…then…else`;
- циклические операторы `while`.

Наблюдательный читатель заметит, что это достаточно скромный набор инструментов и совершенно непонятно, зачем же тогда в языках программирования все остальные операторы. И действительно, на практике помимо непосредственно разрешённых структурным программированием операторов допускается *расширение набора базовых конструкций*, а именно:

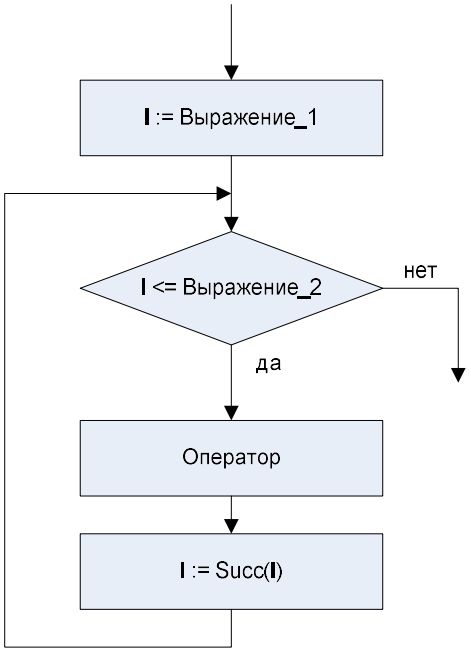
- использование дополнительных конструкций для организации циклов (операторы `for` и `repeat…until`);
- оператор case как расширение оператора if;
- подпрограммы с несколькими входами или выходами;
- оператор goto с жёсткими ограничениями.

Дело в том, что предложенный Бомом и Джакопини набор базовых конструкций — это лишь необходимый минимум, позволяющий описать любой алгоритм, но при этом не позволяющий построить неструктурированную программу. Однако «описать любой алгоритм» — это ещё не значит «описать любой алгоритм просто», поэтому в сложных программах часто применяются операторы, позволяющие записывать типовые фрагменты алгоритмов более компактно и наглядно, чем с использованием базовых операторов.

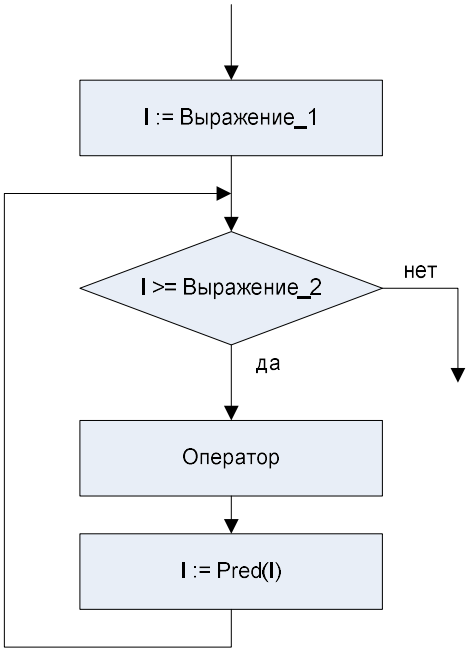
Операторы for и repeat…until в структурном программировании

Начнём с того, что операторы `for` и `repeat…until` хоть и не реализуют конструкцию обобщённого цикла в чистом виде (в отличие от оператора `while`), тем не менее могут быть легко к ней сведены.

Логика работы оператора `for` иллюстрируется одной из следующих схем алгоритма:



Если диапазон задан через **to**

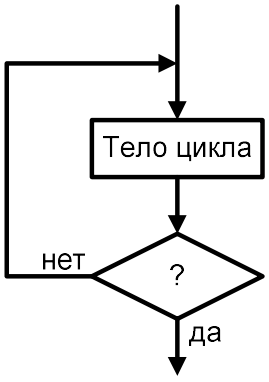


Если диапазон задан через **downto**

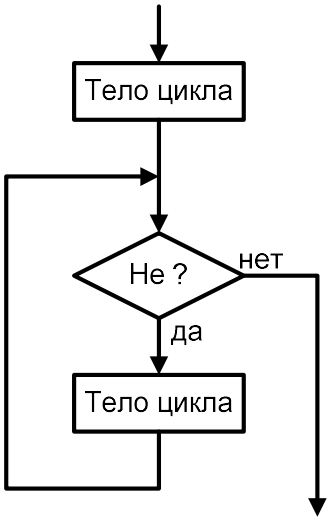
Легко заметить, что от обычного обобщённого цикла (цикла с предусловием) тот цикл, который реализует оператор `for`, отличается лишь наличием двух функциональных блоков в теле цикла и одним функциональным блоком, предшествующим циклу. При этом два блока в теле цикла образуют конструкцию следования.

Опираясь на эти наблюдения, нетрудно показать, что конструкция, соответствующая оператору `for` легко приводится преобразованиями Бома-Джакопини к одному функциональному блоку, а это означает, что такая конструкция также не нарушает структурированности алгоритма или программы.

С оператором `repeat…until` это чуть менее очевидно, однако также легко доказуемо. Этот оператор позволяет реализовать следующую конструкцию:



Нетрудно заметить, что от обобщённого цикла эта конструкция отличается только порядком выполнения тела цикла и проверки условия, а также смыслом самого условия (условие входа или условие выхода). Кроме того, точно такую же логику можно реализовать следующим образом:



Другими словами, оператор `repeat…until` эквивалентен оператору `while`, перед которым продублировано тело цикла. Нетрудно заметить, что такая конструкция также может быть приведена к единственному функциональному блоку с помощью преобразований Бома-Джакопини, а значит, также не нарушает структурированности алгоритма или программы.

Таким образом, все циклические операторы в Delphi можно использовать, не нарушая принципов структурного программирования.

Оператор case

Часто в программах возникает необходимость выполнить в зависимости от значения какой-либо величины различные действия. Попробуем, например, вывести название поры года, к которой относится заданный месяц — значение перечислимого типа:

```
type
  TMonth = (monJan, monFeb, monMar, monApr, monMay, monJun,
            monJul, monAug, monSep, monOct, monNov, monDec);

var
  Month: TMonth;

...

if (Month = monJan) or (Month = monFeb) or (Month = monDec) then
  WriteLn('Зима')
else
  if (Month >= monMar) and (Month <= monMay) then
    WriteLn('Весна')
  else
    if (Month >= monJun) and (Month <= monAug) then
      WriteLn('Лето')
    else
      WriteLn('Осень');
```

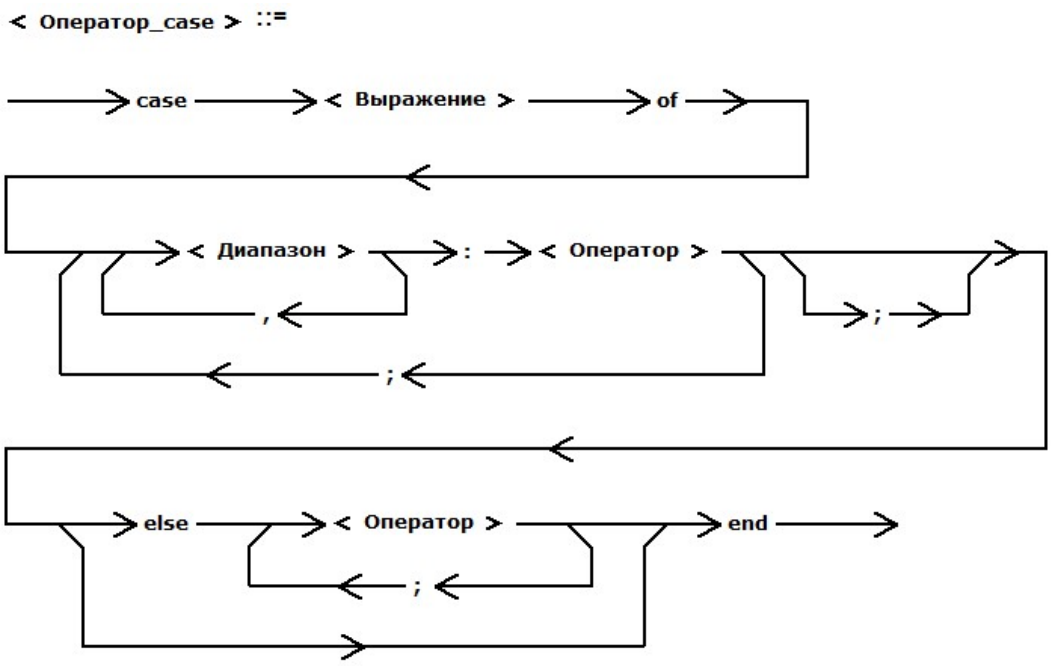
Согласитесь, не очень-то наглядно. Как проверить, все ли случаи учтены? Придётся внимательно анализировать этот фрагмент кода и убеждаться в этом. И главное: случайно пропущенный символ (например, клавиша «=» не нажалась) — и всё, в программе ошибка, которую в такой громоздкой записи ещё надо заметить. Не говоря уж о том, что в одном случае операция `and`, в другом — `or`, расслабиться при анализе кода просто негде.

Оператор `case` позволяет решить эту же задачу намного нагляднее:

```
case Month of
  monJan, monFeb, monDec:
    WriteLn('Зима');
  monMar..monMay:
    WriteLn('Весна');
  monJun..monAug:
    WriteLn('Лето');
  monSep..monNov:
    WriteLn('Осень');
end;
```

Ну вот, так значительно лучше! И вложенность куда меньше, и текста куда меньше, и отследить, все ли случаи проверены, намного проще.

Для начала разберёмся с самим оператором. Синтаксическая диаграмма, описывающая правила его записи, имеет следующий вид:



Выражение, которое указывается после ключевого слова `case`, может быть любого перенумерованного типа. После ключевого слова `of` начинается перечисление ветвей.

Каждая ветвь помечается *селектором* — одним или несколькими диапазонами, задающими условие выполнения этой ветви. Каждый диапазон может быть либо двумя константными выражениями, разделёнными двумя точками (как в объявлении типа диапазон), либо одним константным выражением, например:

```
case Month of
  monFeb, monJul..monAug:
    WriteLn('Сейчас могут быть каникулы.');
  else
    WriteLn('Идёт учебный год.');
end;
```

В данном примере первая ветвь будет выполняться в тех случаях, когда переменная `Month` содержит одно из значений `monFeb`, `monJul` или `monAug`.

Значения, которыми помечаются ветви, не могут пересекаться. Другими словами, одному возможному значению анализируемого выражения должна соответствовать не более чем она ветвь. Попытки нарушить это требование приводят к ошибке компиляции.

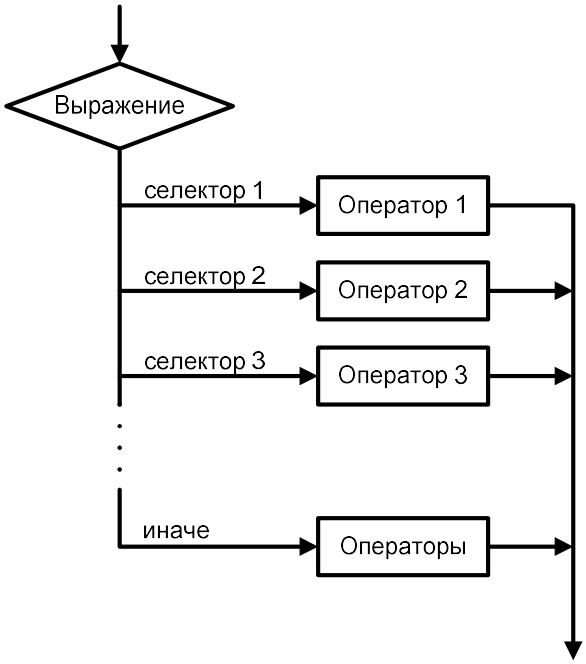
В зависимости от значения выражения оператор `case` выбирает ветвь, которая должна быть выполнена. Если подходящей ветви не найдено, возможны два варианта: при наличии ветви `else` выполняться будет именно она, а при её отсутствии управление сразу будет передано дальше по тексту программы.

Ветвь `else` оператора `case` — ещё одно из немногочисленных мест в программах на Delphi, где можно записывать несколько операторов без использования составного оператора:

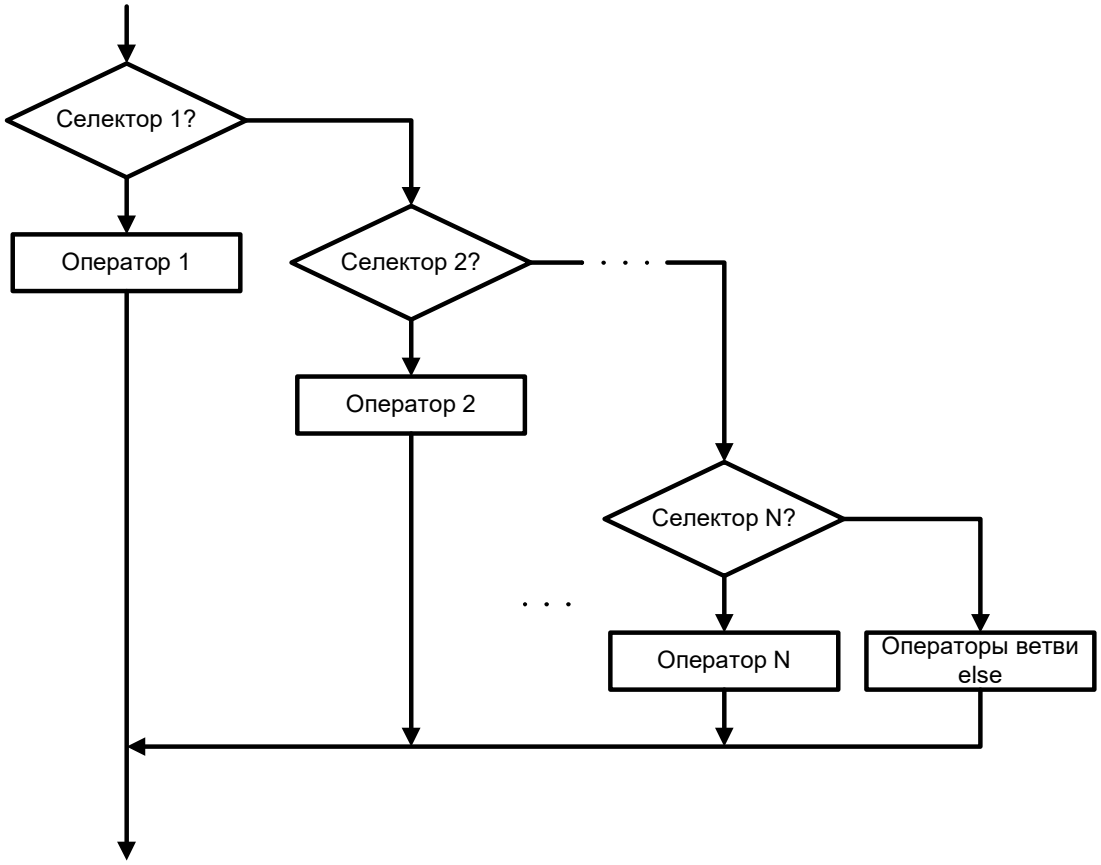
```
case Month of
  monFeb, monJul..monAug:
    WriteLn('Сейчас могут быть каникулы.');
  else
    WriteLn('Сейчас идёт');
    WriteLn('учебный год.');
end;
```

Как можно видеть из предыдущих примеров, ещё одна синтаксическая вольность, допускаемая Delphi, — необязательность точки с запятой перед `else` (в отличие от оператора `if`, где точка с запятой перед else не ставится никогда). Несмотря на кажущуюся нелогичность, такая постановка этого знака препинания приближает текст программы на языке Delphi к записи алгоритма на естественном языке.

Схема алгоритма, соответствующая оператору case в общем случае выглядит так:



Логически эквивалентная схема алгоритма выглядит следующим образом:



Начиная с блока «Решение» с самой большой вложенностью, можно применить преобразования Бома-Джакопини: на каждом шаге преобразований на наибольшем уровне вложенности будет оказываться конструкция принятия двоичного решения. Таким образом, как и операторы цикла `for` и `repeat…until`, оператор `case` не нарушает структурированности программы и может использоваться в качестве расширения стандартного набора конструкций.

Следует однако иметь в виду, что не во всех языках программирования это так! Например, в C-подобных языках программирования аналогичный оператор — `switch` — в общем случае нарушает структурированность программы, т.к. может иметь более сложную схему передачи управления, чем в Pascal/Delphi. Поэтому этот оператор рекомендуется использовать с осторожностью.

Оператор goto

Без преувеличения самый критикуемый оператор в большинстве языков программирования — это оператор `goto`, и критика эта связана как раз с тем, что этот оператор позволяет легко нарушать требования структурного программирования.

Чаще всего при упоминании о вреде оператора `goto` ссылаются на статью Эдсгера Дейкстры «Go To Statement Considered Harmful», опубликованную в 1968 г. При этом утверждается, что в этой статье, якобы, написано, что оператор `goto` — это практически вселенское зло и его использование приличными программистами попросту недопустимо.

В действительности же те, кто делает такую отсылку, саму статью-то и не читали, как, впрочем, и другие статьи Дейкстры. Мысль, сформулированная в статье, была не

настолько радикальной, да и сам он вполне успешно использовал этот оператор в своих публикациях. Дословно мысль звучала следующим образом:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce

---

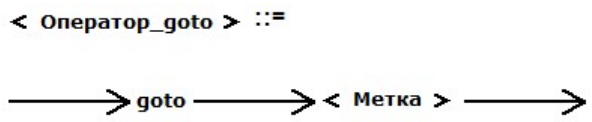
Таким образом, Дейкстра не утверждал, что от оператора `goto` следует отказаться полностью раз и навсегда, однако отметил, что чем реже используется этот оператор, тем выше обычно оказывается качество программы (и её автора).

Более того, даже в новых языках программирования остаётся место для этого оператора. Правда, иногда создатели этих языков старательно делают вид, что это не `goto`, или слегка урезают его возможности, делая проблематичным его неоправданное применение. Так поступили, например, в языке Java: оператор `goto` в нём отсутствует, зато есть оператор `break`, который в отличие от очень ограниченного оператора `break` в других языках программирования позволяет делать всё то же, что и классический `goto`, но чуть менее удобно. В общем, классическая ситуация из песенки о том, что мышечный орган пониже спины есть, а слова нет.

Чем же вызвано такое отношение к оператору `goto`?

Дело в том, что он позволяет передавать управление в программе из одного места в другое достаточно произвольно, тем самым давая возможность (преимущественно начинающим) программистам писать неструктурированные программы.

В языках Pascal и Delphi этот оператор есть и имеет следующий синтаксис:



В коде это выглядит примерно так:

```
label 1, 2;

...

  I := 1;
1:
  if I > 10 then
    goto 2;
  WriteLn(I:7);
  Inc(I);
  goto 1;
2:
  WriteLn('Mission complete.');
```

Оператор `goto` передаёт управление на указанную после ключевого слова метку. Таким образом, приведённый пример представляет собой запутанный способ записи цикла с предусловием, выводящего числа от 1 до 10.

Основная проблема с использованием оператора `goto` — это возможность превратить код программы в так называемый *спагетти-код*. Так называют код, в котором передача

управления между его частями происходит очень запутанным образом, не поддающимся отслеживанию. Примерно так же сложно проследить, где начинается и где заканчивается каждая макаронина в тарелке спагетти:

На схеме алгоритма оператор `goto` превращается в линию, которая может идти из любой точки в любую. В результате почти всегда такая схема не поддаётся сворачиванию в единственный функциональный блок преобразованиями Бома-Джакопини, а значит, соответствующая программа оказывается неструктурированной.

На практике в большинстве языков программирования оператор `goto` имеет ряд ограничений по передаче управления. Чаще всего они связаны со сложностью технической реализации. В частности, обычно с его помощью невозможно:

- переходить внутрь производных операторов (составного, `for`, `while`, `repeat…until`, `if`, `case` и т.п.), не содержащих данный оператор `goto`;
- переходить из одной альтернативы в другую в выбирающих операторах (`if`, `case`);
- входить в подпрограмму или выходить из неё.

В соответствии с принципом Бома-Джакопини для написания программы любой сложности достаточно трёх базовых конструкций, ни одна из которых не требует наличия оператора `goto`, поэтому обычно от его использования отказываются полностью, либо применяют с жёсткими ограничениями.

Break u Continue

В большинстве языков программирования есть операторы `break` и `continue`, предназначенные для использования вместе с циклическими операторами. Единственное отличие Delphi заключается в том, что здесь это не операторы, а встроенные процедуры.

Процедура `Break` осуществляет досрочный выход из цикла. Например:

```
for I := 1 to 1000 do
begin
  ...
  if X > 0 then
    Break;
  ...
end;
```

В данном примере выход из цикла может произойти в двух случаях: либо переменная `I` достигнет конечного значения, либо на одной из итераций значение переменной `X` окажется больше 0.

Процедура Break передаёт управление на оператор, следующий сразу за оператором цикла. Используется только внутри тела цикла с любым из операторов цикла: `for`, `while`, `repeat…until`. Строго говоря, `Break` представляет собой `goto`, для которого жёстко задано место, куда будет передаваться управление, т.е. это `goto` с жёстким ограничением. Эквивалентный код с `goto` выглядел бы так:

```
  for I := 1 to 1000 do
  begin
    ...
    if X > 0 then
      goto EndLoop;
    ...
  end;

EndLoop:
```

Нетрудно заметить, что получающиеся при использовании `Break` циклы будут иметь два и более выхода: один — по основному условию цикла, остальные — по тем условиям, при которых выполняется `Break`. А конструкция с несколькими выходами прямо противоречит принципам структурного программирования.

Процедура Continue завершает выполнение текущей итерации:

```
for I := 1 to 1000 do
begin
  ...
  if X > 0 then
    Continue;
  ...
end;
```

Эквивалентный код с `goto` выглядел бы следующим образом:

```
  for I := 1 to 1000 do
  begin
    ...
    if X > 0 then
      goto EndIteration;
    ...

EndIteration:
  end;
```

Таким образом, как и `Break`, процедура `Continue` представляет собой сильно ограниченный оператор `goto`. При её использовании фрагмент с двумя выходами образуется в теле цикла, что также нарушает принципы структурного программирования.

На практике эти две процедуры в отличие от оператора `goto` достаточно распространены. Разумеется, любая программа может быть написана без их использования, однако иногда они позволяют записать текст программы более лаконично и понятно. Кроме того, иногда, когда сформулировать условие цикла сразу проблематично, часто используют конструкцию следующего вида:

```
while True do
begin
  ...
  if ... then Break;
  ...
  if ... then Break;
  ...
  if ... then Break;
  ...
end;
```

По своей сути это бесконечный цикл, однако процедура `Break` позволяет организовать выход из него. Использовать подобные конструкции в программах нежелательно, однако как промежуточный способ записи алгоритма, чтобы понять, какие условия будут фигурировать в цикле и как их объединить в общее условие цикла, может применяться.

Когда можно использовать goto?

Несмотря на то, что использование оператора `goto` считается крайне нежелательным и всячески избегается, есть ряд задач, в которых его использование оказывается оправданным.

Всего можно выделить три основных *случая оправданного использования оператора goto*:

- выход из нескольких вложенных циклов;
- сложная обработка ошибок;
- автоматически сгенерированный код.

Случай выхода из нескольких вложенных циклов — самый распространённый. Как правило, его можно избежать грамотным проектированием программы, однако в редких случаях вложенные циклы с необходимостью досрочно завершать сразу несколько из них всё же встречаются. Выглядит это примерно так:

```
  while A > B do
  begin
    for I := 1 to N do
      for J := 1 to M do
      begin
        if Matrix[I, J] = Value then
        begin
          WriteLn(I, J);
          goto ExitLabel;
        end
        else
          Matrix[I, J] := Matrix[I, J] + A;
      end;
    A := A - 1;
  end;

ExitLabel:
```

Логика этого фрагмента кода примерно такова: есть матрица размера N×M, которая подвергается модификации, пока не будет найден элемент с заданным значением. Процесс останавливается либо при достижении определённого порога изменений (определяется значением `B`), либо при нахождении элемента, который принял искомое значение.

Реализовать то же самое без goto вполне возможно, однако при этом пострадает либо читаемость кода, либо производительность программы.

В некоторых языках программирования для тех же целей оператор `break` позволяет указать количество вложенных циклов, из которых нужно выйти за один раз, однако разработчики Pascal/Delphi предпочли следовать принципу

Все сложные и потенциально опасные действия должны выглядеть сложно и потенциально опасно

Вместо того, чтобы дать возможность построить сложный цикл с несколькими выходами — сложную и потенциально опасную в плане ошибок конструкцию — простой записью вида

```
break 3;
```

языки Pascal и Delphi заставляют программиста подтвердить свои намерения. «Случайно» написать подобный выход из цикла не получится: придётся ведь ещё объявлять метку. Причём само наличие объявления метки уже заставляет того, кто читает код, насторожиться и внимательнее проверять написанное.

Второй случай оправданного использования `goto` — сложная обработка ошибок. Опять же, при грамотном проектировании программы почти всегда удаётся избежать этой ситуации, однако иногда это случается.

Предположим, программе необходимо выполнить сложную операцию, состоящую из целого ряда шагов, на каждом из которых может возникнуть ошибка. При этом, если ошибка возникла, необходимо будет отменять часть уже выполненных действий, а если всё прошло успешно, освобождать часть выделенных во время работы ресурсов (например, закрыть временные сетевые соединения и т.п.). В этом случае структурированный код окажется слишком громоздким, чтобы быть приемлемым:

```
  ...
  if ErrorCondition1 then goto Error1;
  ...
  if ErrorCondition2 then
  begin
    ...
    goto Error2;
  end;
  ...
  if ErrorCondition3 then goto Error3;
  ...
  goto FreeResources
Error1:
  ...
Error2:
  ...
  goto FreeResources
Error3:
  ...
FreeResources:
  ...
```

В данном примере на месте всех многоточий предполагается выполнение сложных действий, в том числе выделение и освобождение ресурсов. Как правило, в действительно сложных случаях (а они очень редки!) попытки написания структурированного кода только усугубляют ситуацию, как, впрочем, и другие механизмы (например, исключения). В таких ситуациях использование `goto` может быть оправданным.

Наконец третий случай — это автоматически сгенерированный код. Здесь всё просто: если код генерируется автоматически какой-то программой и не предназначен для чтения человеком, то необходимость поддерживать его структурированность отпадает, т.к. компилятор справится и с таким кодом.

Наиболее типичный пример — инструменты, позволяющие по грамматическому описанию языка программирования сгенерировать код, позволяющий разобрать текст программы на этом языке. Иногда эти инструменты ещё называют компиляторами компиляторов: в первую очередь это генераторы лексических и синтаксических анализаторов. Зачастую код с `goto`, выдаваемый этими инструментами, оказывается даже более эффективным, чем структурированная реализация.

И всё же, несмотря на перечисленные случаи, когда `goto` могут простить, в программах языках программирования высокого уровня его использования следует по возможности избегать.

Дополнительные вопросы

1. Как организовать досрочный выход из цикла в рамках структурного программирования (без использования Break и goto)?

2. Найдите ошибку в приведённом фрагменте программы:

```
case Month of
  monJan:
    WriteLn('Январь');
  monApr:
    WriteLn('Апрель');
  monDec:
    WriteLn('Декабрь');
  monFeb..monMay:
    WriteLn('Весенний семестр');
end;
```

3. Дана переменная `C` типа `Char`. Запишите оператор `case`, который определяет, к какой из следующих групп относится записанный в эту переменную символ: латинские буквы, арабские цифры, знаки препинания, другие символы.