Баула В.Г., Мещеряков Д.К.

Учебное пособие по написанию программ при выполнении работ практикума на ЭВМ

Введение

Настоящее пособие имеет своей целью помочь студентам в написании текстов программ, реализуемых ими в рамках занятий по практикуму на ЭВМ, что в дальнейшем, несомненно, поможет и при написании курсовых и дипломных работ. В пособии разъясняется суть процесса написания исходных текстов программ и показываются простые способы повышения эффективности такого написания.

Следует прежде всего заметить, что собственно написание текста программы (он же часто называется исходным кодом) всегда явно или неявно предваряется начальными этапами реализации решения любой задачи на ЭВМ. Эти должны быть следующие этапы:

1. **Осознание задачи**. Это исключительно важный этап. Преподавателям многократно приходилось сталкиваться с фактом, что студент решил не ту или не совсем ту задачу, которая перед ним ставилась (в таком случае решение не принимается, т.е. ошибка на данном начальном этапе обходится очень дорого в смысле затрат времени и нервов, а часто и конечного результата, поскольку такое неверное решение не всегда возможно быстро исправить).
2. **Спецификация задачи (формирование требований)**. На этом этапе требуется рассмотреть все случаи, в которых необходимо решить задачу, и выделить те из них, для которых решение имеет какую-либо специфику. Для всех случаев следует строго определить, какие данные программа получает на вход и что она должна выдать в результате.

   В учебном процессе большинство требований обычно непосредственно следуют из постановки задачи (например, из текста задачи в задачнике). Необходимо также учитывать, что при решении задачи на ЭВМ (в отличие от задач на зачётах и экзаменах), также обязательно требуется, чтобы программа корректно реагировала на неверные входные данные (например, пользователь ввёл буквенную строку в ответ на запрос программы ввести число) и выдавала в таком случае осмысленное сообщение об ошибке (лучше на русском языке), а не прерывала свою работу, и уж тем более не продолжала работу с неверно введенными данными. Следует также понимать, что надо учитывать и требования, накладываемые различными практическими ограничениями (например, диапазоном представления чисел в конкретной ЭВМ, средствами используемого языка и т.д.).

   По-существу, этот этап состоит в формализации (уточнении) постановки задачи, осознанной на предыдущем этапе, поэтому необходимо отнестись к нему ответственно, т.к. неверно сформированные требования приводят к написанию программы, решающей не ту задачу, или не удовлетворяющей требования, поставленные в условии задачи.
3. **Разбиение на подзадачи**. На этом этапе исходную задачу, если она сложна для немедленной реализации, разбивают на обозримые и более простые в реализации подзадачи, совокупное решение которых приводит к решению исходной задачи целиком (этот процесс в литературе называют пошаговой детализацией, программированием сверху-вниз и другими сходными терминами).

   Здесь также не следует торопиться, поскольку неудачное разбиение на подзадачи часто приводит к проблемам с записью программы на языке программирования. При этом значительно возрастает объём исходного кода, увеличивается числа используемых функций и модулей, а также связей между ними в программе. В итоге получается плохая программа.
4. **Разработка алгоритмов и их запись на языке программирования**. Здесь и начинается то, что традиционно считается программированием, – запись решения подзадач на выбранном языке программирования (конечно, в случае практикума на ЭВМ выбор этого языка не зависит от воли студента). Этому этапу в основном и посвящено данное пособие, поэтому рассмотрим его последующих разделах более подробно.

Суть процесса написания текстов программ

Процесс написания исходных текстов («кода») программного обеспечения включает в себя следующие три основных процесса:

1. разработка собственно алгоритма решения задачи;

2. выбор конкретных средств из арсенала используемого языка программирования;
3. запись алгоритма на языке программирования с использованием выбранных средств.

Эти три процесса находятся в непрерывной и тесной взаимосвязи. В частности, зачастую приходится несколько модифицировать уже разработанный алгоритм просто для того, чтобы его запись на выбранном языке была, например, менее громоздкой.

Существенно, что первый и второй этапы являются в значительной степени творческими, в то время как третий – в основном формальным. Тем не менее, именно на этом формальном третьем этапе у студентов часто и возникают значительные трудности. Структурная сложность алгоритма, перенесенная в исходный текст, дополняется полной или значительной нечитаемостью последнего, что делает задачу разработки и последующей отладки программы более трудоемкой.

Человек не компилятор. Компилятор с некоторого языка с усердием скомпилирует любой синтаксически правильный исходный текст (синтаксическая правильность является чисто формальным понятием). Если полученная при этом программа не запустится на выполнение, не сможет отработать до конца из-за возникшего сбоя или, что чаще, будет работать не так, как хотелось бы программисту, то анализировать сложившуюся ситуацию однозначно придется самому автору программы (типичный вопрос к коллегам и преподавателям: «А почему она не работает/не так работает, я ведь всё сделал правильно?»). Поэтому исходный текст должен быть не только пригодным для машинной компиляции (чтобы компилятор давал на выходе сообщение вида «Ноль ошибок»), но и пригодным для чтения и анализа человеком (и не только непосредственным автором этой программы).

К счастью, руководствуясь несложными правилами и здравым смыслом, можно получать исходный текст, относительно легко читаемый человеком. Именно поэтому в наиболее успешных компаниях – разработчиках программного обеспечения существуют утвержденные правила оформления исходных текстов. Поскольку «университет не готовит коммерческих программистов»[1], этим компаниям часто приходится нанимать выпускников и студентов старших курсов стажерами, а те за 1-3 месяца учатся писать исходные тексты в соответствии с такими правилами.

Разберемся сначала, зачем компании используют такие строгие правила. На это есть ряд причин:

1. Удобно читаемый исходный текст легче анализировать самому разработчику – следовательно, его легче отлаживать и модифицировать.
2. Удобно читаемый код легче анализировать и другим разработчикам. Часто сотруднику достается работа по небольшой доработке написанного ранее другим разработчиком программного кода. Реже, но тоже часто из-за текучки кадров код программы целиком переходит другому разработчику (как для завершения разработки, так и для добавления новой функциональности или исправления найденных ошибок). За счет более высокой читаемости кода экономится время, требуемое на понимание того, как работает данный программный код и какие изменения в него, возможно, необходимо внести.

Заметим здесь еще раз, что компилятор прекрасно компилирует код, едва только тот станет соответствующим требованиям используемого языка. Следует с большой осторожностью относиться к присущему многим начинающим разработчикам интуитивному представлению, что «машина умная», и раз она сказала, что ошибок нет, то всё на самом деле хорошо.

Теперь разберемся, почему строгие правила записи исходных текстов удостоились внимания в университетском методическом пособии, если «университет не готовит коммерческих программистов».

Такие правила используются компаниями для повышения эффективности работы как коллективов, так и каждого разработчика в отдельности. Это простые в использовании методики, позволяющие существенно экономить время на разработке и сопровождении программ, которые могут быть с успехом использованы и в учебном процессе. Следование таким методикам в исследовательских работах (курсовые и дипломные проекты) и при выполнении типовых заданий практикума позволит студентам уделять больше внимания творческим процессам и меньше – формальным.

Замечено, что именно неумение писать читаемый код мешает начинающим разработчикам повышать свою квалификацию и браться за более сложные и объёмные задания. С некоторого объема исходного кода именно низкая читаемость текста, а отнюдь не сложность задачи или большое число связей по данным и/или управлению мешают эффективной разработке программы.

---

[1] Да, это так.

Процесс выполнения задания по практикуму

Прежде чем сдавать задание по практикуму, его, вообще говоря, необходимо выполнить. Высокая читаемость исходного кода позволяет значительно экономить время на отладке такой программы. Заметим, что до приобретения опыта в разработке программного обеспечения с трудом верится в то, что «всего-навсего» неудачная запись исходного текста в недрах объёмной программы может спровоцировать ошибки, на обнаружение которых опытный программист тратит по нескольку дней своего рабочего времени.

Иногда квалификации студента, выполняющего задание, недостаточно для полностью самостоятельного выполнения работы – тогда он зачастую обращается за помощью к более квалифицированному коллеге. За счет высокой читаемости кода можно сэкономить коллеге немало времени и меньше отвлекать его от выполнения собственного задания, не говоря уже о том, что в плохо читаемом коде ошибку, возможно, не удастся найти и с помощью более опытного коллеги.

Если задание все же, по мнению студента, выполнено, его надлежит сдать преподавателю, то есть, в частности, объяснить, как работает программа, человеку, который видит текст этой программы впервые. Замечено, что низкая читаемость исходного кода приводит к особо тщательному и «проникновенному» анализу кода преподавателем, а это, в свою очередь, почему-то обычно приводит к обнаружению значительного числа ошибок, которые студенту затем приходится исправлять.

Данные методические указания призваны помочь начинающим разработчикам в освоении именно процесса написания хорошо читаемого кода. Необходимость данного навыка подробно объяснена выше. Можно предположить, что опыт компаний, выпускающих и поддерживающих продукты объемом, например, 700 файлов в среднем по 300 строк кода в каждом следует при этом принять во внимание.

В следующих разделах приводятся важнейшие приемы написания хорошо читаемого кода в порядке понижения их важности. Многие указания непосредственно сопровождаются примерами. Также в качестве иллюстрации используются две полных программы, исходные тексты которых приведены в последнем разделе.

Выбор имён

Имена констант, типов, переменных и функций/процедур следует выбирать, исходя из смысловой нагрузки соответствующих им сущностей. Имя процедуры (функции) должно говорить о том, какие действия выполняет данная функция, и соответствовать повелительному наклонению. При этом

1. Категорически не следует использовать сокращения. Например, `NOFiles` – явно неудачное обозначение для переменной, хранящей число файлов. В данном случае следует использовать имя вида `numberOfFiles` (как вариант – `filesNumber`).
2. При формировании имени из нескольких слов естественного языка следует каждый символ, соответствующий началу слова, выбирать заглавной буквой, остальные символы – строчными буквами, как это проиллюстрировано в примере пункта 1. Имена процедур/функций следует начинать с заглавной буквы, переменных – со строчной.
3. Язык Паскаль не различает заглавные и строчные буквы, однако следует всегда использовать одно и то же написание каждого идентификатора – это повышает читаемость кода и облегчает переход на языки, отличающие в именах заглавные буквы от строчных (в частности, Си и Си++).
4. Категорически не следует использовать идентификаторы из небольшого числа символов, в частности, из одного символа. Такие идентификаторы, как `a`, `dft`, `tt` прекрасно обозначают переменные в языке программирования, но не сущности алгоритма, запись которого должна быть понятна человеку. Кроме того, не составляет труда допустить опечатку при наборе одного из подобных имён, в результате реализовав в коде ошибку, поиск которой является крайне трудоемким. В качестве единственного исключения следует использовать идентификаторы `i`, `j`, `k` в качестве счетчиков в циклах и индексов элементов массивов (в языках Си и Паскаль).
5. Не следует использовать символ подчеркивания в идентификаторах, поскольку он может подсознательно восприниматься как пробельный символ и затруднять чтение.
6. Категорически не следует пользоваться префиксными или иными символьными последовательностями для обозначения типа переменной, в том числе и так называемой «венгерской записью», когда в начале имени переменной располагается символьная последовательность, указывающая тип переменной, например `bIsNormalized` для переменной типа Boolean. Поскольку префикс – это просто последовательность символов, подробно раскритикованная в пункте 3, использование та-

кой записи явно допускает наличие двух и более переменных, имена которых отличаются лишь такими префиксами, что создает богатую почву для опечаток и затрудняет чтение текста программы.

Сбалансированная плотность кода

Язык программирования обычно допускает многочисленные вольности в использовании переносов текста программы с одной строки на другую. Например, в языке Паскаль перенос можно ставить между любыми двумя лексемами и число переносов также может быть любым. Реально такими переносами следует пользоваться исключительно аккуратно, руководствуясь при этом следующими правилами:

1. Следует начинать каждый оператор (описание, объявление, раздел) с новой строки. При таком расположении подряд идущие операторы однозначно воспринимаются как последовательная композиция (следующий неявно получает на вход переменные, измененные всеми предыдущими).
2. Категорически не следует использовать более одной пустой строки подряд. Большое число идущих подряд пустых строк подсознательно воспринимается как разделение операторов на несвязанные между собой и не относящиеся к одному алгоритму группы.
3. Следует всегда разделять определения функций (и подобных им сущностей) пустой строкой. Функции и процедуры являются по сути совокупностью операторов, объединенных для решения конкретной подзадачи, и выделение их с помощью пустых строк помогает воспринимать их именно в таком качестве. Кроме того, при этом легче искать, например, для оператора процедуры соответствующее описание этой процедуры.
4. Пустые строки также следует использовать для группировки тесно связанных друг с другом по смыслу операторов. Такой прием значительно упрощает разбиение крупного фрагмента кода, реализующего сложный алгоритм, на более мелкие, выполняющие более простые, составляющие его части.
5. Во всех остальных случаях подряд идущие операторы должны располагаться на подряд идущих строках.

Отступы

В литературе подчас используется изображение алгоритма в виде блок-схемы. Это наглядный способ объяснения работы алгоритма, он позволяет легко определять, например, какие операторы находятся в ветвях условного оператора, а какие вовсе этим условным оператором не контролируются. Процесс разработки программного обеспечения в парадигме структурного программирования представляет собой конструирование более сложного алгоритма из более простых, объединенных различными способами. Поэтому возможность легко идентифицировать пути передачи управления в коде программы является одной из важнейших на пути к её быстрой и эффективной разработке. Правильное использование отступов позволяет придать исходному тексту только что описанное свойство структурности блок-схемы.

В использовании отступов следует придерживаться следующих основных правил:

1. Операторы, связанные последовательной композицией, начинаются с одной и той же позиции своих строк.

   Например:

```
alphaCoefficient := shiftRatio * compressionRatio;
PerformTransformation( aplhaCoefficient );
```

2. Операторы, являющиеся вложенными по отношению к какому-либо оператору (условному, цикла, составному) начинаются с позиции, смещенной вправо на некоторое число отступов. Это число должно быть всегда одним и тем же. Для смещения читаемой части строки вправо используют пробелы или символы табуляции. Обычно выбирают либо от 2 до 4 пробелов или один символ табуляции.

   Например:

```
if alphaCoefficient < 0 then begin
   betaCoefficient := Sin( alphaCoefficient );
```

```
    PerformTransformation( alphaCoefficient );
end;
```

3. Для вложенных операторов также применяются приведенные только что пункты 1-2 – чем глубже уровень вложенности, тем на большее число символов вправо смещаются начала строк.

Разбиение длинных строк

В случае, если запись какой-либо конструкции не умещается целиком на одной строке так, чтобы всю строку можно было видеть в рабочем поле текстового редактора, не прибегая к прокрутке, то такую строку следует разбивать на несколько, используя переносы строк (естественно, не нарушая при этом синтаксической правильности конструкции). При этом все строки, начиная со второй, следует располагать со смещением вправо относительно первой строки.

Например:

```
betaCoefficient := alphaCoefficient * shiftRatio +
   correctionParameter * compressionRatio;
```

В случае, если таким образом разбивается вызов функции, то имя функции и открывающая скобка должны располагаться на одной и той же строке.

Например:

```
PerformImageTransformationWithSmoothingAndCorrection(
   imageSmoothingCoefficient, imageCorrectionCoefficient );
```

Операторные скобки

В языках программирования используются так называемые операторные скобки, позволяющие компоновать вместе несколько операторов. В разных языках такие операторные скобки могут иметь различный вид. Напомним, что для языка Паскаль это пары ключевых слов **begin** – **end** и **repeat** – **until**.

С точки зрения синтаксиса языка, операторные скобки являются указаниями на принадлежность заключенных между ними операторов к одной последовательной композиции, которая, возможно, входит в состав какого-либо другого оператора (условного, цикла и т.д.). Именно для подчеркивания принадлежности операторов к такому включающемц оператору и используются унифицированные правила размещения операторных скобок. В совокупности с удачным использованием отступов такие правила позволяют гораздо эффективнее анализировать структуру[1] записанного в исходном тексте алгоритма.

Исходя из только что сказанного, следует придерживаться следующих правил использования операторных скобок

1. В случае определения тела функции или подобной ей сущности открывающая и закрывающая операторные скобки размещаются с тем же отступом слева, что и описание функции.

   Например:

```
function MaximumOfIntegers( firstValue: Integer;
  secondValue: Integer ): Integer;
begin
   if firstValue > secondValue then begin
     MaximumOfTwoIntegers := firstValue;
```

[1] Многие об этом забывают, но слово «структура» в сочетании «структурное программирование» несет глубокий смысл.

```
    end else begin
      MaximumOfTwoIntegers := secondValue;
    end;
  end;
```

2. В случае составного оператора открывающая скобка **begin** размещается на той же строке, что и запись служебного слова включающего оператора (**then, else, do** и т.д.). Поскольку все операторы внутри скобок будут смещены вправо, при таком расположении открывающая скобка не занимает лишней строки. Подходящий пример приведен в пункте 1.
3. В случае, если составной оператор включается в контролирующий его оператор так, что соответствующая часть этого контролирующего оператора не умещается на одной строке, т.е. используется смещение второй и последующих строк вправо, открывающая операторная скобка располагается на следующей строке и с тем же смещением слева, что и первая строка записи контролирующего оператора. Поскольку и все операторы внутри скобок смещены так же, как и часть строк в записи контролирующего оператора, такое размещение позволяет однозначно определять, где начинается запись последовательности, являющейся вложенным оператором.
   Например:

```
if IsAtEndOfFile or totalRecordsRead >= requiredRecordsRead or
  shouldStopReading then
begin
  PerformCleanup;
end;
```

4. Закрывающая операторная скобка размещается на отдельной строке сразу после последнего оператора ограниченной данной скобкой последовательности и с тем же отступом, что и первая строка записи служебного слова соответствующего контролирующего оператора.
5. В том случае, если вложенная последовательность операторов состоит только из одного оператора, все равно следует расставлять операторные скобки по вышеприведенным правилам для однозначного указания на длину и состав этой последовательности.[1]

Примеры, приведенные ниже, содержат достаточно иллюстраций для почти всех составных операторов языка Паскаль. Для оператора цикла **for** следует, очевидно, руководствоваться теми же правилами, что и для оператора цикла **while**. Поскольку запись оператора цикла **repeat** – **until** сама служит операторными скобками, то не следует использовать дополнительную пару операторных скобок для вложенной последовательности оператора, т.е. следует писать

```
repeat
   ReadNextRecord;
   Inc( recordsCount );
until EndOfFile or recordsCount > recordsToRead;
```

Запись оператора множественного выбора **case** следует осуществлять по аналогии со следующим примером:

```
case numberOfFiles of
  105: begin
    PerformGenericTransformation;
```

[1] Заметим, что для единообразия формы записи операторов каждый из них заканчивается точкой с запятой (что в Паскале, как известно, не является необходимым), так что формально внутри каждого составного оператора в этих примера находятся по два оператора, один из которых пустой.

```
      PerformCleanup;
    end;
    106, 107: begin
      PerformCustomTransformation;
    end;
  else
    IndicateError;
  end;
```

Использование пробелов и иных разделителей

1. Категорически не следует использовать более одного пробельного символа подряд. Использование более одного пробельного символа подряд приводит к разреженному коду, который, как и код с большим числом подряд идущих пустых строк, хуже читается.
2. Как явно следует из пункта 1, категорически не следует использовать так называемое декоративное форматирование, т.е. выстраивание выражений и идентификаторов в столбцы за счет добавления табуляций или пробелов. За исключением редчайших случаев сложных численных алгоритмов, такое выстраивание создает впечатление наличия вертикальной структуры в исходном тексте, а такой структуры, как правило, реально нет.

   Пример такого ненужного декоративного форматирования:

```
PerformTransformation( 10     , 100000 );
PerformTransformation( 1000000,     10 );
```

   Повторим, декоративное форматирование использовать категорически не рекомендуется.
3. Следует обязательно ставить по одном пробельному символу в следующих случаях:
   3.1 между открывающей скобкой и следующим за ней выражением,
   3.2 между закрывающей скобкой и следующим перед ней выражением, например

```
returnValue := ( shiftFactor - compressFactor ) * alphaCoefficient;
```

   3.3 между знаком двухместной арифметической операции или оператора присваивания и находящимися по сторонам от них выражениями (подходящий пример приведен в пункте 3.2),
   3.4 между запятой и следующим за ней выражением, например

```
PerformTransformation( 10, 10 );
```

4. Не следует ставить пробел перед запятой или точкой с запятой.

Использование именованных констант

Возможность приписывать константам имена является мощным и исключительно полезным средством, присутствующим во многих языках программирования, включая языки низкого уровня (ассемблеры и макроассемблеры). Реализация механизма именованных констант может различаться – это могут быть закрытые на запись переменные с начальным значением, а может быть и просто подстановка значения на место имени константы. В обоих случаях использование имён констант вместо конкретных значений часто позволяет повысить читаемость кода и избежать крайне тяжелых ошибок.

Объявление константы – это пара из имени этой константы и связанного с ней значения. С точки зрения компилятора, нет существенной разницы между использованием имени константы и её значения. Однако разница становится очевидной при рассмотрении процесса анализа и модификации исходного текста программы.

Рассмотрим типичную ситуацию. Фрагмент кода перебирает элементы массива с первого до последнего. Есть непосредственно оператор цикла или условный оператор, задающий ограничение на то, в каких пределах должно находиться значение индекса элемента. Также есть и фрагмент, где либо объявлен массив, либо выделяется динамическая память под него. Пусть в обоих фрагментах число элементов прописано явно.

```
var
  dataArray: array[1..452] of Integer;
  arrayIndex: Integer;
. . .
begin
  for arrayIndex := 1 to 452 do begin
    WriteLn( dataArray[arrayIndex] );
  end;
end;
```

Тогда при необходимости изменить число элементов (как при изменении требований к программе, так и просто при отладке) необходимо заменить все вхождения в исходный текст старого значения на новые. Необходимость такого изменения может возникать многократно (например, это обычное явление при отладке). При этом каждый раз нужно, во-первых, обязательно заменить все вхождения, и, во-вторых, не изменить ни одного значения, не имеющего отношения к данному изменению кода.

Например, часть кода обрабатывает массив из 452 элементов, а другая часть – получает для каких-то целей остаток от деления по модулю 452 – тогда изменять те вхождения числа 452, которые используются в записи деления, скорее всего, не нужно (можно привести и примеры, когда эти значения взаимосвязаны).

Несоблюдение только что указанного требования немедленно провоцирует тяжелые ошибки. Например, размер массива в его объявлении уменьшен, а значение в операторе, проверяющем значение индекса, оставлено прежним – программа либо портит данные других объектов, либо работает с неинициализированными значениями, либо (это лучший случай) просто вызывает аварийный останов. Аналогично при изменении значения в соседнем участке кода неожиданно меняется поведение части программы, которую, на первый взгляд, и вовсе не изменяли.

Использование механизма констант позволяет избежать описанных выше проблем. Всюду в коде программы используется имя константы, а при изменении ее значения в объявлении константы компилятор при следующей же компиляции расставляет во всех нужных местах программы новое значение. Необходимость ручного поиска и исправления отпадает.

Аналогично и при чтении кода непросто бывает понять, является ли, например, в вышеупомянутом примере значение в объявлении массива связанным со значением в записи получения остатка от деления[1].

Таким образом, в случае, если какое-либо числовое значение используется в коде более одного раза, следует всегда обозначать его именованной константой. Исключением можно считать числовые значения 0 и 1 и только в случаях, когда они используются в качестве значений индексов массивов или заполнителей областей памяти.

Использование комментариев

Руководствуясь приведенными выше правилами, можно получать исходный код, который читается легко и не требует детального комментирования каждого оператора. Использование осмысленных имен для переменных и функций приближает язык программирования к естественному языку, делая исходный текст просто описанием того, что делает алгоритм, записанный в данном исходном тексте.

Безусловно, не следует вовсе отказываться от комментариев. Но однозначно нет смысла в комментариях вида

[1] Например, получение остатка от деления может использоваться для реализации хеш-таблицы поверх массива.

```
arrayIndex := arrayIndex + 1; {прибавляем единицу к arrayIndex}
```

Обычно следует комментировать функции целиком – описывать условия, предъявляемые к параметрам, и работу реализованных в функциях алгоритмов. Следует комментировать последовательности операторов в тех случаях, когда некоторые свойства реализованного алгоритма не вполне очевидны при простом прочтении исходного кода.

Комментарии следует располагать либо перед соответствующим текстом, соблюдая общие правила использования отступов, либо, если комментарии коротки, - справа от соответствующего текста.

Код от использования которого программист отказывается, не следует заносить в комментарии. Это может ввести в заблуждение читающего программу человека.

Простой и короткий пример

Данная программа запрашивает у пользователя целые числа до тех пор, пока пользователь не введет число, кратное 5. Ошибки ввода программой не обрабатываются.

```
program Mod5( input, output);
{ Программа последовательно вводит целые числа,
  пока не будет введено число, кратное 5.
  Контроль правильности входных данных не производится}
var
  inputValue: Integer;
  endCondition: Boolean;

begin
  repeat
    WriteLn( 'Введите число, кратное 5' );
    ReadLn( inputValue );
    endCondition := inputValue mod 5 = 0;
    If endCondition then begin
       WriteLn( 'Введено число, кратное 5.' );
    end else begin
      WriteLn( 'Введено число, не кратное 5.' );
    end;
  until endCondition;
end.
```

Значительно более сложный пример

Этот пример следует изучить, когда Вы решитесь реализовать в качестве задания практикума относительно сложную задачу. Данная программа решает вполне определенную содержательную задачу. Дан файл, полученный путем копирования части компакт-диска. Он содержит просто последовательность блоков данных с диска. Требуется извлечь из него файлы, содержавшиеся на исходном диске.

Естественно, такая задача в общей постановке непроста. Поэтому будем считать, что на диске содержатся только файлы GIF и JPEG (и только определенных версий этих форматов). Такие файлы содержат в определенных позициях последовательности символов “GIF” или “JFIF” соответственно. Файловая система (алгоритм размещения данных), используемая для компакт-дисков, размещает файлы 2048-байтными блоками, поэтому файл внутри последовательности блоков может начинаться только со смещения, кратного 2048, относительно начала.

Поскольку задача сама по себе редкая (в реальной жизни данная программа была использована один раз), то пути к файлам прописаны прямо в исходном тексте. Программа не обрабатывает ошибки ввода-вывода.

```
program Expand03;
const
  {Число блоков, участвующих в одной операции с файлом}
  OneCluster = 1;
  ClusterLength = 2048; {размер блока в файловой системе}

type
  TDataCluster = array [1..ClusterLength] of Byte;

var
  DataBuffer: TDataCluster;
  InFile: file;
  OutFile: file;
  { номер обрабатываемого файла - используется для генерации
    имен файлов по правилу resultNNNN }
  FileNumber: Integer;

{Функция определяет, является ли блок первым блоком файла,
  проверяя, содержатся ли конкретные последовательности
  байт в наперед заданных позициях}
function SectorContainsBorder( var Buffer: TDataCluster ) : Boolean;
begin
  {символы 'GIF' в позициях 1..3}
  if ( Buffer[1] = $047 ) and ( Buffer[2] = $049 ) and
  ( Buffer[3] = $046 ) ) then
  begin
    SectorContainsBorder := True;
  end else begin
    {символы 'JFIF' в позициях 7..10}
    if ( Buffer[7] = $04A ) and ( Buffer[8] = $046 ) and
      ( Buffer[9] = $049 ) and ( Buffer[10] = $046 ) then
    begin
      SectorContainsBorder := True;
    end else begin
      SectorContainsBorder := False;
    end;
  end;
end;

{ Функция выдает имя очередного выходного файла в виде
```

```
  resultNNNN, используя передаваемый ей по ссылке номер файла,
  затем сама увеличивает номер файла на единицу }
function GetNextFilePath( var FileNumber: Integer ) : String;
var
  TempString: String;
begin
  Str( FileNumber, TempString );
  while Length( TempString ) < 4 do begin
    TempString := '0' + TempString;
  end;
  TempString := 'c:\windows\temp\result' + TempString;
  Inc( FileNumber );
  GetNextFilePath := TempString;
end;

begin
  {инициализируем номер выходного, открываем входной и первый
    выходной файлы}
  FileNumber := 1;
  Assign( InFile, 'c:\windows\temp\data.dat' );
  Reset( InFile, ClusterLength );
  Assign( OutFile, GetNextFilePath( FileNumber ) );
  Rewrite( OutFile, ClusterLength);
  while not Eof( InFile) do begin
    {читаем очередной блок, выясняем, является ли он первым
      блоком файла, при необходимости закрываем прежний
      выходной файл и открываем новый}
    BlockRead( InFile, DataBuffer, OneCluster );
    if SectorContainsBorder( DataBuffer ) then begin
      Close( OutFile );
      Assign( OutFile, GetNextFilePath( FileNumber ) );
      Rewrite( OutFile, ClusterLength );
    end;
    BlockWrite( OutFile, DataBuffer, OneCluster );
  end;
  Close( InFile );
  Close( OutFile );
end.
```