Московский государственный университет

имени М.В. Ломоносова

В.Г. Баула

Основы разработки программного обеспечения

Учебно-методическое пособие

Москва, 2016

ББК 22.18

О-75

УДК 681.142.2

Рецензенты:
А.В. Гуляев – доцент;
А.Н. Сальников – ведущий научный сотрудник, к.ф.м.н.

Баула В.Г.

Основы разработки программного обеспечения: учебно-методическое пособие. – М.: Издательский отдел факультета ВМК МГУ им. М.В. Ломоносова; МАХ Пресс, 2016.

ISBN 978-5-89407-561-7
ISBN

Настоящее пособие имеет своей целью помочь студентам в разработке программного обеспечения, начиная от простых программ в рамках занятий по практикуму на ЭВМ на первых курсах, а далее и при написании курсовых и дипломных работ. Разъясняется суть процесса реализации программного обеспечения на большинстве этапов его разработки. Пособие предназначено для студентов программистских факультетов, а также для преподавателей, ведущих практические занятия по программированию.

ISBN 978-5-89407-561-7
ISBN

В.Г. Баула

Основы разработки программного обеспечения

Введение

Подготовка специалистов в области разработки программного обеспечения является сложным многоэтапным процессом. Имеет большое значение и хорошая математическая подготовка, и достаточно высокий общенаучный уровень знания. Но, конечно, главным всё же является обучение в области собственно программирования.

Как среди преподавателей по программированию, так и среди самих студентов бытует мнение, что уже на самом первом этапе этого обучения необходимо иметь как можно больше практики в написании различных программ. Обычно речь идёт о написании нескольких работающих программ каждую неделю. Такая практика приводит к тому, что пишется по определённым шаблонам большое количество упрощённых, сугубо учебных программ, имеющих весьма малое отношение к настоящему программному обеспечению. Предполагается, что лишь на следующем этапе студенты должны вовлекаться в полноценные программные проекты. Этот подход к обучению представляется не совсем правильным, так как игнорирование важных этапов в написании программного обеспечения в начальный период обучения приводит к существенным пробелам в профессиональном образовании студентов. Здесь представляется целесообразным не забывать о старом полезном принципе "лучше меньше, да лучше".

Настоящее пособие имеет своей целью помочь студентам в поэтапной разработке программного обеспечения, начиная от простых программ в рамках занятий по практикуму на ЭВМ на первых курсах, а далее и при написании курсовых и дипломных работ. Разъясняется суть процесса реализации программного обеспечения на большинстве этапах его разработки. Иллюстрация этого процесса проводится на простых примерах, написанных на Турбо-Паскале [1].

1. Этапы решения задачи

Традиционно при разработке программного обеспечения выделяют несколько стадий или этапов. Прежде всего, следует заметить, что при этом собственно написание текста программы (он же часто называется исходным или программным кодом) всегда явно или неявно предваряется некоторыми начальными этапами. Принято считать, что разработка и реализация программного обеспечения должны содержать следующие этапы.

1.1. Осознание поставленной задачи

Это исключительно важный этап, он предполагает ясное понимание того, что является входными данными задачи, и что она должна выдать в качестве своего результата. Преподавателям многократно приходилось сталкиваться с фактом, что студент решил не ту или не совсем ту задачу, которая перед ним ставилась. Ошибка на данном начальном этапе обходится очень дорого в смысле затрат времени и нервов, а часто и конечного результата, поскольку неверное решение не всегда возможно быстро исправить так, чтобы получилось верное.

В качестве примера рассмотрим следующую простую задачу. Необходимо ввести из стандартного входного потока целое число в целочисленную переменную `x` и вывести сумму всех цифр в десятичной записи этого числа. Часто приводится следующее решение этой задачи на Турбо-Паскале.

```
var x,sum: integer;
begin
  read(x); sum:=0;
  repeat sum:=sum+x mod 10; x:=x div 10
  until x=0;
  writeln(x)
end.
```

В этом решении не осознано, что вводимое число может быть отрицательным. При этом в большинстве конкретных реализаций языка Паскаль выражение `x mod 10` будет отрицательным (именно так вычисляют остаток целочисленного деления большинство ЭВМ), а цифры в нашей десятичной

системе счисления по определению имеют положительные значения. В результате получиться *отрицательная* сумма цифр, что лишено математического смысла.

Для исправления этой ошибки можно попытаться перед циклом вычислить абсолютную величину числа `x:=abs(x)`, это может решить проблему, но не всегда полностью. Здесь нужно осознать, что для некоторых целых чисел могут не существовать их абсолютные величины, именно так и обстоит дело в наиболее распространенной на ЭВМ целочисленной системе счисления с так называемым дополнительным кодом. Таким образом, при правильном осознании задачи для верного решения абсолютную величину надо брать от значения каждой цифры числа:

```
sum:=sum+abs(x mod 10);
```

Осознанию поставленной задачи в значительной степени мешают особенности компьютерной арифметики. Та математика, которая реализована в ЭВМ, называется *дискретной* математикой и значительно отличается от традиционной, привычной для нас со школы математикой. Главное отличие заключается в том, что как для целых, так и для вещественных величин в ЭВМ существует только конечное число допустимых значений. Таким образом, в компьютере существуют минимальное и максимальное целые и вещественные, а вне этих диапазонов чисел нет. Заметим, что в большинстве реализаций языков программирования высокого уровня для этих величин предусмотрены имена, например, в Турбо-Паскале наименьшая (отрицательная) и наибольшая целые константы имеют стандартные имена `MinInt` и `MaxInt` соответственно.[1]

При выполнении арифметических операций (сложения, вычитания и т.д.) с такими величинами, правильный результат получается только в том случае, если этот результат (сумма, разность и т.д.) не выходит за допустимый диапазон представимых величин соответствующего типа (целого, вещественного и т.д.). А что же будет, если, скажем, результат сложения выходит за допустимый диапазон? В описании стандарта Паскаля сказано, что в этом случае результат операции *не определён*. Это, конечно, не означает, что он принимает случайное значение. Просто, когда в стандарте языка говорится, что результат некоторого действие не определён, это означает, что в каждой конкретной реализации этого языка в данном случае должно быть чётко сказано, что будет делаться. Например, в Турбо-Паскале при выходе целочисленного значения за допустимый диапазон, результат будет зависеть от того, в каком режиме будет работать исполнитель (в каком режиме будет выполняться компиляция программы).

Все режимы компилятора задаются комментариями особого вида, интересующий нас режим задаётся директивами `{$R+}` (контроль выхода за допустимый диапазон включён) и `{$R-}` (контроль выключен). При включённом контроле в случае выхода целочисленного значения за допустимый диапазон происходит аварийное завершение выполнения программы с выдачей соответствующей диагностики. При выключенном контроле результат операции будет неправильным, но выполнение программы продолжится. Например, при выполнении программы

```
var x: integer; {$R-}
begin
  x:=30000; x:=x+x; writeln('x=',x)
end.
```

будет выдан ответ `x=-5536`.

Резонно задать вопрос, зачем выдавать неправильный результат и продолжать бессмысленный дальнейший счёт программы? Здесь всё дело в том, что при включённом контроле на каждую полезную машинную операция (например, сложения) будет тратиться дополнительная машинная операция контроля (операция условного перехода). Следовательно, можно сформулировать такую стратегию работы: на этапе отладки программы включаем контроль, а после отладки контроль выключаем, уменьшая тем самым размер программы и увеличивая скорость её работы.

Как уже говорилось, дискретная математика, в которой работает ЭВМ, существенно отличается от обычной. В частности, если коммутативный закон по сложению и умножению выполняется, т.е. всегда `a + b = b + a`, ассоциативный и дистрибутивный законы не выполняются, т.е. в общем случае `(a + b) + c ≠ a + (b + c)` и `a*(b + c) ≠ a*b + a*c`. Легко понять, что это происходит из-за того, что результат любой операции может выйти за допустимый диапазон. Кроме того, все действия с вещественными величинами производятся *приближённо*, т.е. результат операции округляется до ближайшего представимого вещественного значения.

---

[1] К сожалению, в стандарте Паскаля [11] определено только имя `MaxInt`, что иногда приводит к трудностям в программировании.

Для того, чтобы более наглядно представить себе отличие дискретной математики от обычной, рассмотрим на вещественной прямой решение уравнения

```
x + A = A
```

В обычной математике это уравнение имеет единственное решение `x=0`, а вот в дискретной могут существовать и отличные от нуля решения. Например, для типа `real` в Турбо-Паскале для $A=10^{10}$ это будет корень `x=0.156`, для $A=10^{15}$ – корень `x=1024.0`, а для $A=10^{20}$ – уже корень $x=10^6$. Так происходит потому, что после прибавления относительно маленького корня `x` к большой константе `A` результат округляется снова в `A`. Легко понять, что и все числа, меньше чем `x`, тоже будут корнями этого уравнение

К сожалению, во многих учебниках по программированию на такие "ненужные тонкости" вообще не обращают внимания, что приводит в реальных программах к семантическим ошибкам, которые трудно выявляются при отладке.

1.2. Спецификация задачи

На этом этапе требуется рассмотреть все случаи входных данных, для которых необходимо решить задачу, и выделить те из них, для которых решение имеет какую-либо специфику. Для всех таких особых случаев следует строго определить, какие данные программа получает в качестве входных и что она должна выдать в результате своей работы.

Рассмотрим один курьёзный пример плохой спецификации программы. В конце прошлого века на общественном транспорте стали широко применяться проездные билеты с магнитной полосой, на которой записывалась все необходимые данные (число оставшихся поездок, дата окончания действия и т.д.). Для программы обработки этих данных во входных транспортных турникетах была составлена примерно такая спецификация.

Так как при неаккуратном обращении с карточкой магнитная полоса может портиться, то она делиться на две одинаковые половины с идентичными данными, которые программа должна обрабатывать следующим образом. После чтения информации с магнитной полосы сначала анализируется первая половина данных. Если информация считана правильно (совпала так называемая контрольная сумма), то из положительного числа поездок вычитается единица, и обновлённые данные записываются на обе половинки магнитной полосы. Если же первую половину магнитной полосы правильно считать не удалось, то анализируется содержимое второй половины этой полосы, и, если там правильные данные, то из числа поездок вычитается единица, и обновлённые данные записываются на обе половинки магнитной полосы (в надежде, что данные на первой половине полосы исправятся после перезаписи).

В этой спецификации была допущена грубая ошибка. Рассмотрим, что будет, если пассажир заклеит первую половину полосы, например, скотчем, что будет препятствовать как чтению, так и записи на неё данных.[1] При использовании такой карты, когда будет исчерпано число поездок на второй половине магнитной полосы, можно (на один раз) снять наклейку с первой половина, потом снова её заклеить и т.д. Легко подсчитать, что для допустимого числа поездок `N` таким хитрым способом можно проехать `N*(N+1)/2` раз.

Правильная спецификация этой программы должна быть примерно следующей. В случае, если читается только *одна* из двух половинок магнитной полосы, то надо действовать, как и в предыдущей спецификации, а если правильно прочитаны *обе* половины, то надо взять минимум из оставшегося числа поездок, вычесть из него единицу и записать изменённые данные на обе половины полосы.[2] Такая спецификация позволяет уменьшить число поездок при хитроумном заклеивании половинок магнитной полосы до `2*N` раз. Уменьшение числа допустимых поездок до `N` раз требует существенного изменения структуры магнитной карты.

В качестве следующей иллюстрации важности спецификации рассмотрим следующую простую задачу для реализации на Паскале. Надо вводить из входного потока целые числа, пока не будет введено число ноль (признак конца ввода), после чего следует вывести максимальное чётное число из всех введённых чисел. На этапе осознания задачи следует понять, что вводимые числа могут быть как

---

[1] Метод вычисления контрольной суммы таков, что, если данные совсем не читаются (например, магнитная полоса отсутствует), то будет тоже зафиксирована ошибка, как и при повреждении магнитного носителя.

[2] Такая ошибка была допущена в программе турникетов с магнитными проездными билетами Московского метрополитена, информация об этом быстро распространилась по Интернету. Исправление ошибки, как легко догадаться, заняло достаточно продолжительное время, чем немедленно и воспользовались нехорошие люди.

положительные, так и отрицательные, что важно при поиске максимума. Кроме того, следует уяснить, что признак конца ввода ноль тоже является чётным числом, но оно при поиске максимума учитываться не должно. Далее, количество вводимых чисел неизвестно, поэтому нельзя, например, предварительно ввести их в некоторый массив (обычно реализации с записью вводимых чисел в файл или некоторую динамическую структуру данных, например, список, совсем не рассматривается, как крайне не эффективные для этой задачи).

Теперь надо зафиксировать все особые случаю, и для каждого из них строго определить, что наша программа будет выводить в качестве результата. Очевидно, что особых случаев всего два: входная последовательность пуста (сразу введён признак конца ввода ноль), и в последовательности все числа нечётные. Заметим, что часто при формулировке текста задачи в учебнике, а часто и в билете на зачёте и экзамене, вообще ничего не говориться, что в этом случае делать. Когда на это обращают внимание, то говорят, что задача не доопределена, задана не полностью (или не строго сформулирована) и т.д. Вообще говоря, эта ситуация типичная, так как в реальной жизни задача перед программистом ставится заказчиком, который и сам часто не понимает, что в этой задаче есть какие-то особые случаи, и что при этом надо делать. В учебном процессе большинство требований обычно (но далеко не всегда, а часто неявно) следуют из постановки задачи, например, из текста задачи в задачнике, или задания для практического решения на практикуме работы на ЭВМ.

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, если пользователь ввёл буквенную строку в ответ на запрос программы ввести число, программа должна выдавать 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.). Всё это делает спецификацию задачи ещё более сложной.

По существу, спецификация задачи 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 требованиям, поставленные в условии задачи.

Итак, надо специфицировать нашу задачу поиска максимального чётного числа в последовательности вводимых чисел. Например, в случае пустой последовательности будем выдавать сообщение "Чисел нет", а при отсутствии в последовательности чётных чисел (кроме, естественно, последнего нуля) выдавать "Нет чётных". Тогда можно предложить следующее решение.

```
var x,max: integer;
begin
  read(x);
  if x=0 then Writeln('Чисел нет') else
  begin
    max:=1;{Пока чётных не было}
    repeat {здесь обязательно x<>0}
      if not odd(x) then {чётное x}
        if (max=1) or (x>max) then max:=x;
      read(x)
    until x=0;
    if max=1 then Writeln('Нет чётных')
             else Writeln('Max.чётное=',max)
  end
end.
```

А теперь модифицируем постановку задачи. Пусть ввод чисел до нуля и вычисление максимального чётного числа необходимо реализовать в виде функции. Другими словами, это подзадача какой-то более общей задачи, главная программа[1] вызовет эту функцию, а потом будет что-то делать с полученным результатом её работы.

На этапе осознания задачи важно понять, что наша функция не должна (как это часто делают студенты) выполнять операторы вроде `Writeln('Нет чётных')`. Ясно, что вызвавшая эту функцию

[1] В общем случае в роли главной программы может выступать также и другая процедура или функция.

главная программа, вероятно, ничего не читает из стандартного выходного потока output,[1] и не сможет "увидеть" эту диагностику, так что надо сделать для нашей функции какую-то другую спецификацию для особых случаев входных данных. Например, пусть при отсутствии во входном потоке чисел, кроме конечного нуля, функция выдаёт значение −1, а при отсутствии чётных чисел выдаёт −3, так как эти значения нечётные и не могут получаться при "нормальной" работе функции. В программистской литературе такие особые значения обычно называются *кодом возврата*. Заметим, что по правилам Паскаля при любом вызове функция *обязана* возвратить какой-то результат своей работы, так что нельзя, как иногда пытаются сделать студенты, "ничего не вернуть". Тогда получится, например, следующее описание нужной функции.

```
function MaxEven: integer;
  var x,max: integer;
begin
  read(x);
  if x=0 then MaxEven:=-1 else
  begin
    max:=1; {Пока чётных не было}
    repeat {здесь x<>0}
      if not odd(x) then
        if (max=1) or (x>max) then max:=x;
      read(x)
    until x=0;
    if max=1 then MaxEven:=-3 else MaxEven:=max
  end
end {MaxEven};
```

Итак, спецификация задачи выделяет все особые случаи в её входных данных, и фиксирует, что должно быть в этих случаях получено в качестве выходных данных (результатов решения задачи).

В качестве следующего примера рассмотрим совсем простую задачу. Пусть необходимо ввести одно целое число в переменную x оператором read(x), затем присвоит переменной y значение x+1 и вывести полученное значение y. Ясно, что чаще всего для этого на Паскале будет написана такая программа.

```
  var x,y: integer;
begin
   read(x); y:=x+1; Writeln(y)
end.
```

А теперь зададимся естественным вопросом, когда эта программа будет давать правильный ответ? Ясно, что для того, чтобы эта программа дала правильный ответ, необходимо наложить определённые ограничения на её входные данные. Эти ограничения можно записать в виде некоторого логического условия от входных данных, в теории программирования это логическое условие называется *предусловием* программы.[2] Какое же предусловие будет в нашем случае?

Во-первых, необходимо потребовать, чтобы в стандартном входном потоке данных input, к которому обращается оператор read, находилась правильная лексема целого числа, причём после преобразования её во внутреннее представление, это целое число принадлежало к диапазону представимых в нашей Паскаль-машине целых чисел. Обозначим этот диапазон буквой $\underline{Z}$, по аналогии с обозначением в математике множества всех целых чисел как Z, тогда это требование предусловия можно записать как $x \in \underline{Z}$. Далее, надо потребовать, чтобы величина x была меньшей, чем самое большое представимое в Паскале целое число, которое, как известно, имеет стандартное имя MaxInt. Теперь можно написать предусловие нашей программы в виде

$$Pred=\{x \in \underline{Z} \wedge x<MaxInt\}$$

---

[1] Для программы прочесть собственный стандартный выходной поток достаточно сложная проблема. Её можно пытаться решить, например, путём перенаправления выходного потока перед началом счёта в текстовый файл. Однако из-за проблем с буферизацией и блокировкой файла при попытке множественного доступа там тоже может быть всё не так просто.

[2] Более точно оно называется *слабейшим* предусловием [8].

Будем, как часто пишется в математических текстах, знак ∧ использовать для операции Паскаля **and**, а знак ∨ – для операции **or** .При истинности предусловия программа должна выдать правильный ответ, что можно записать в виде логического *постусловия* программы.

```
Post={y∈Z ∧ y=x+1}
```

Дадим теперь определение того, когда (не имеющая синтаксических ошибок) программа является (семантически) *правильной*. Сначала поймём, что правильность или неправильность программы зависит от её предусловия. Например, приведённая выше программа для предусловия `Pred={x∈Z}` будет *неправильной*, так как для введённого `x=MaxInt` правильного ответа не получиться, т.е. постусловие не будет выполняться. Предусловие, сама программа и постусловие носят название *триада Хоара*.[1]

Итак, будем называть программу **правильной** *в некотором предусловии*, если при истинности этого предусловия программа завершится и обязательно обеспечит истинность нужного нам постусловия.[2] Здесь можно провести аналогию с правильностью работы завода: при соответствии входного сырья заданным стандартам, правильно работающий завод должен обеспечит выпуск продукции, тоже удовлетворяющей заданным стандартам, иначе завод работает **неправильно**.

В том случае, если предусловие не выполнено, программа, вообще говоря, не обязана обеспечить правильность постусловия. Другими словами, в этом случае программа может зациклиться, аварийно остановиться (ошибка времени выполнения – Run Time Error, по-русски АВОСТ), или же выдать неверный результат. По аналогии с заводом, при получении им плохого сырья он не обязан производить хорошую продукцию, а может выдать брак или повести себя и совсем нехорошо (например, взорваться ☺).

Как Вы догадываетесь, такое неправильное поведение программы не обрадует заказчика, который поручил программисту написать эту программу. Исходя из этого, с точки зрения нашего заказчика, лучшим предусловием программы было бы предусловие

```
Pred=True
```

Такое предусловие следует известному принципу хорошего обслуживания "Клиент всегда прав". В этом предусловии написанная выше программа, конечно же, будет *неправильной*. Разберёмся сначала, а какое постусловие должно быть у нашей программы в таком, тождественно истинном, предусловии. Естественно, оно тоже должно быть тождественно истинным, т.к. все входные данные по определению "хорошие". Можно, например, для этого случая предложить следующее постусловие

```
Post={x∈Z ∧ x<MaxInt ⇒ y∈Z ∧ y=x+1 ∨
      x∉Z ⇒ 'Плохое x' ∨
      x∈Z ∧ x=MaxInt ⇒ 'Большое x'}
```

Как Вы можете убедиться, данное постусловие является тождественно истинным, так как для любого значения, введённого из входного потока, одно из трёх логических слагаемых обязательно будет истинным, что, как Вы знаете, повлечёт и истинность всего выражения, которое является логической суммой трех логических слагаемых.[3]

Как можно догадаться, написать программу для решения задачи в тождественно истинном предусловии будет совсем не так просто. В то же время, очевидно, что "настоящие" программы должны быть написаны именно в таком предусловии (как уже говорилось, "Пользователь всегда прав"). У хороших программ не должно быть зацикливаний, выдачи непонятных диагностик и "синего экрана смерти". В конце этого пособия мы попытаемся написать программу для решения этой задачи в тождественно истинном предусловии.

В качестве следующего примера рассмотрим такую простую программу.

```
  var a,b,c: real;
begin
```

---

[1] Используются элементы аксиоматической семантики Т.Хоара. Рассматривается применение предусловий и постусловий только ко всей программе целиком, а не к составляющим её операторам. Более подробно о предусловиях и постусловиях можно почитать в книгах [2,12].

[2] В научной литературе в этом случае говорят о *полной* правильности программы. *Частичная* правильность подразумевает, что если программа завершится, то постусловие будет выполнено.

[3] В математической логике тождественно истинные формулы называются *общезначимыми* или *тавтологиями*.

```
  read(a,b); c:=sqrt(sqr(a)+sqr(b)); Write(c)
end.
```

Можно предположить, что программа решает следующую задачу: ввести длины двух катетов a и b, затем вычислить и вывести длину гипотенузы c прямоугольного треугольника. Правильная ли эта программа? Теперь Вы уже должны понимать, что этот вопрос не так прост и имеет смысл только для конкретного предусловия данной задачи. Синтаксически верная программа может быть правильной в одном постусловии и неправильной в другом. Для написания предусловия, в котором написанная выше программа будет правильной, сначала, естественно, придётся осознать поставленную задачу и выполнить её спецификацию.

Для введённых *отрицательных* значений длин катетов естественно предположить, что треугольник не существует и программа должна давать соответствующую диагностику. Сложнее обстоит дело с нулевыми значениями длин катетов. В классической математике такие треугольники, конечно, не существуют (вырождаются в отрезок или точку), но Вы уже должны знать, что программы "живут" в мире *дискретной* математики. В частности, вводимые из "внешнего мира" вещественные числа округляются до ближайшей точки на оси вещественных чисел Паскаль машины.[1] При этом введённая ненулевая лексема вещественного числа, после преобразования её во внутреннее представление может округлиться до вещественного машинного нуля. Исходя из этого, лучше будет считать (это спецификация), что у таких *вырожденных* треугольников гипотенуза существует и вычисляется по той же самой формуле, т.е. будет совпадать с другим ненулевым катетом или вообще обращаться в ноль.

Попытаемся написать предусловие для задачи вычисления длины гипотенузы. По аналогии с классической математикой, где R обозначает множество всех вещественных чисел, обозначим через $\underline{R}$ конечное множество всех вещественных чисел на оси Паскаль-машины, а внутренние представления длин катетов a и b как $\underline{a}$ и $\underline{b}$, а гипотенузы – как $\underline{c}$. Тогда можно написать такое хорошее для заказчика программы предусловие задачи:

$$Pred=\{\underline{a},\underline{b}\in\underline{R} \wedge \underline{a},\underline{b}\geq 0 \wedge \underline{c}=sqrt(\underline{a}^2+\underline{b}^2)\in\underline{R}\}$$

с таким постусловием

$$Post=\{\underline{a},\underline{b}\in\underline{R} \wedge \underline{a},\underline{b}\geq 0 \wedge sqrt(\underline{a}^2+\underline{b}^2)\in\underline{R} \Rightarrow \underline{c}\in\underline{R} \wedge \underline{c}=sqrt(\underline{a}^2+\underline{b}^2) \vee$$
$$\underline{a},\underline{b}\in\underline{R} \wedge (\underline{a}<0 \vee \underline{b}<0) \Rightarrow \text{'Плохой катет'}\}$$

Другими словами, предполагается, что, если для введённых значений катетов гипотенуза представима (существует) на оси Паскаль-машины, то она должна быть вычислена и выведена программой. В этом, естественном для заказчика, предусловии написанная выше программа *неправильная*! Действительно, даже если длина гипотенузы $\underline{c}\in\underline{R}$, это не значит, что она будет получена программой, т.к. значения $\underline{a}^2$, $\underline{b}^2$ или $\underline{a}^2+\underline{b}^2$ могут не принадлежать $\underline{R}$. Предусловие для написанной выше программы, в котором она правильная, следующее

$$Pred=\{\underline{a},\underline{b}\in\underline{R} \wedge \underline{a},\underline{b}\geq 0 \wedge \underline{a}^2+\underline{b}^2\in\underline{R}\}$$

Это может не удовлетворить заказчика, т.к. многие *представимые* в Паскале значения гипотенуз для *представимых* значений катетов не будут получены программой, которая при этом будет аварийно завершаться с не очень понятной для пользователя диагностикой типа Real Overflow. В то же время следует понять, что можно написать правильную программу и для "хорошего" предусловия. Это, например, такая программа:

```
var a,b,c,max: real;
begin
  read(a,b);
  if (a>=0.0) and (b>=0.0) then begin
    if a>b then max:=a else max:=b;
    if max=0.0 then c:=0.0
               else c:=max*sqrt(sqr(a/max)+sqr(b/max));
    Write(c)
  end else writeln('Плохой катет')
end.
```

---

[1] Это не обязательно вещественные числа самой ЭВМ, на которой выполняется программа. Например, в Турбо-Паскале числа типа real имеют длину 6 байт, а машинные вещественные числа обычно имеют длину 4 (тип single), 8 (тип double) или бóльшее число байт.

В этой программе из-под квадратного корня вынесена длина бóльшего катета, так что величина подкоренного выражения будет не больше двух. Таким образом, если гипотенуза представима в этом типе данных, то она будет этой программой получена. Разумеется, новая программа существенно сложнее старой версии. Такие программы, которые всегда пытаются получить ответ, если он представим, называются в программистской литературе робастными (robust), буквально "крепкими", в смысле устойчивыми к "плохим" входным данным. Как Вы догадываетесь, хорошо написанные программы (программный продукт) должны быть робастными, хотя это и не так просто сделать.

В качестве еще одного примера рассмотрим такую задачу. Необходимо ввести 100 целых чисел и вывести их сумму. Обычно студенты осознают, что для решения этой задачи не надо описывать в программе массив для хранения вводимых чисел и предлагают такой вариант программы для решения задачи (эта же программа приведена и в большинстве учебников по программированию):

```
  const N=100;
  var x,sum,i: integer;
begin
  sum:=0;
  for i:=1 to N do begin
    read(x); sum:=sum+x
  end;
  Writeln(sum)
end.
```

Когда (т.е. в каком предусловии) эта программа является правильной? На первый взгляд, можно предложит такое предусловие для этой задачи, в котором написанная программа будет правильной:

$Pred=\{\forall i\in 1..N\ x_i\in \underline{Z} \wedge \Sigma x_i(i\in 1..N)\in \underline{Z}\}$

Необходимо осознать, что в таком предусловии написанная программа *неправильная*. Действительно, в процессе суммирования *частичная сумма* может выйти за допустимый диапазон представимых целых чисел, и правильного ответа не получиться. При выполненном предусловии это может случиться, например, если сначала вводятся в основном положительные числа, а затем отрицательные, так что общая сумма принадлежит $\underline{Z}$, но не будет получена программой! [1]

На самом деле написанная выше программа верна в таком предусловии:

$Pred=\{\forall x_i\ i\in 1..N\ x_i\in \underline{Z} \wedge \forall j\le N\ \Sigma x_i(i\in 1..j)\in \underline{Z}\}$

Другими словами, допустимой должна быть любая частичная сумма. Это предусловие, конечно, для заказчика не очень хорошее, так как многие *представимые* суммы программой не будут получены. Как же, однако, написать такую программу в хорошем первоначальном предусловии? Необходимо понять, что без использования массива, в котором будут храниться все введённые числа, при этом уже не обойтись.[2]

Новая программа сначала должна будет ввести все числа в некоторый массив, а затем упорядочить его по не убыванию. После этого надо организовать "встречное" суммирование, двигаясь одновременно от начала к концу и от конца к началу массива. Ниже приведена возможная программа (она не будет выдавать диагностику при выходе конечной суммы за допустимый диапазон).

```
  const N=100;
  type Mas=array[1..N] of integer;
```

---

[1] Для продвинутых читателей. Как ни странно, но для целочисленной знаковой системы счисления с дополнительным кодом, реализованной на большинстве ЭВМ, программа в этом предусловии будет *правильной*. Дело в том, что в этой системе счисления при выполнении машинной команды сложении целых чисел, если сумма выходит за допустимый диапазон, то аварийного завершения программы не происходит. Вместо этого выдаётся *неправильный* ответ и поднимается так называемый флаг переполнения, который, при желании, программист на Ассемблере может проанализировать после команды сложения. Так вот, если не обращать на переполнение внимания, и сумма выйдет за допустимый диапазон, но при окончании суммировании снова попадёт в этот допустимый диапазон, то ответ будет правильным! Можно считать это одним из достоинств системы счисления с дополнительным кодом, хотя у этой системы есть и свои недостатки. При программировании на Турбо-Паскале это можно использовать, если не задавать директиву `{$R+}` для контроля выхода величин за диапазон их допустимых значений. Стоит, однако, заметить, что для вещественных чисел такой трюк уже не работает, и при переполнении будет аварийное завершение программы.

[2] Если вводимых чисел не очень много, и они помещаются в массив, то использования для хранения этих чисел других структур данных (файлов, списков и т.д.) не рассматривается, как крайне неэффективное.

```
  var sum,i,j,temp: integer; x: Mas;
begin
  for i:=1 to N do Read(x[I]);
{простейшее упорядочивание массива по не убыванию}
  for i:=1 to N-1 do for j:=i+1 to N do
    if x[j]<x[i] then begin
      temp:=x[i]; x[i]:=x[j]; x[j]:=temp
    end;
{Встречное суммирование}
  sum:=0; i:=1; j:=N;
  while i<=j do
    if sum>0 then begin sum:=sum+x[i]; inc(i) end
             else begin sum:=sum+x[j]; dec(j) end;
  Writeln(sum)
end.
```

Еще одной широко распространенной задачей является нахождение среднего арифметического числа в массиве, для которой во многих учебниках приводится примерно такое решение.

```
  type Mas=array[1..N] of real;
  var srarr: real; i: integer; x: Mas;
begin
  { ввод массива } srarr:=0.0;
  for i:=1 to N do srarr:=srarr+x[i];
  srarr:=srarr/N; Writeln(srarr)
end.
```

Ясно, что предусловие для этой задачи, в котором написанная программа будет правильной, будет следующим

$$Pred=\{\forall i\in 1..100\ x_i\in \underline{R} \wedge \Sigma x_i(i\in 1..N)\in \underline{R}\}$$

Таким образом, для подавляющего большинства введённых векторов написанная программа *неправильная*. Правильной в "хорошем" предусловии, например, будет такая программа.

```
  type Mas=array[1..N] of real;
  var srarr: real; i: integer; x: Mas;
begin
  { ввод массива } srarr:=x[1];
  for i:=2 to N do srarr:=(srarr*(i-1)+x[i])/i;
  Writeln(srarr)
end.
```

Большинство языков программирования не позволяют записывать в программе предусловие и постусловие иначе, чем в виде комментария. В некоторых языках, однако, можно использовать специальные логические утверждения, истинность которых должна проверяться во время счета программы. Например, можно упомянуть язык Эйфель [10], в котором предусловие и постусловие являются директивами для соответствующего исполнителя и могут учитываться при выполнении программы. Из достаточно широко используемых языков можно назвать Pyton [13] с директивами-утверждениями **assert**.

Рассмотрим теперь пример, который часто встречается в задачниках по программированию. Требуется обменять значения двух целочисленных переменных без использования третьей (вспомогательной) переменной. Обычно предполагается следующее решение этой проблемы:

```
  var x,y: integer;
        ...
    y := y-x; x := x+y; y := x-y;
```

При проведении этой операции в физическом мире сразу возникают проблемы. Например, предложение обменять содержимое двух вагонов, в одном из которых песок, а в другом уголь, причём без использования третьего вагона, обычно сразу отвергается, как невыполнимое. Немного подумав, можно, однако, предложить решение этой задачи. Например, если песок занимает не весь вагон, то можно сгрести его в одну сторону и установить в вагоне перегородку. После этого уголь из второго вагона пересыпать в свободную часть вагона с песком, а песок затем перегрузить в освободившийся от угля вагон. Теперь осталось только убрать ненужную больше перегородку.

Так можно ли обменять значения двух переменных без использования третьей? Поразмыслив над таким решением, можно сообразить, что в качестве предусловия этой задачи необходимо потребовать, чтобы сумма и разность значений переменных принадлежали множеству Z. Другими словами, только в половине случаев предложенное выше решение будет давать правильный ответ!.

Как уже упоминалось выше, при реализации этой задачи на ЭВМ, в которой реализована целочисленная арифметика с дополнительный кодом, ответ всегда будет правильным. Кроме того, во многих языках программирования высокого уровня реализованы битовые операции над целочисленными аргументами, в частности, операция **xor** (неэквивалентность, сложение по модулю два). С помощью этой операции также можно обменять значения двух целочисленных переменных без использования третьей переменной. Например, в Турбо-Паскале:

```
var x,y: integer;
   ...
  y := y xor x; x := x xor y; y := x xor y;
```

Для обмена значениями переменных других типов (вещественных, строковых и т.д.) можно использовать записи с вариантами. Приведенный пример заставляет задуматься над вопросом, как это соотносится с приведенной выше операцией обмена содержимым двух вагонов. Противоречия здесь нет, потому что *на самом деле*, на уровне машинных команд самой ЭВМ, в операции обмена с помощью команд **xor** компилятором с языка высокого уровня неявно используется вспомогательная переменная на регистре, например на Ассемблере с помощью регистра ax

```
x   db   ?;
y   db   ?; var x,y: integer;
   ...
    mov  ax,x
    xor  y,ax; y:=y xor x;
    xor  ax,y
    mov  x,ax; x:=x xor y
    xor  y,ax; y:=y xor x;
```

Впрочем, на языке Ассемблера операцию обмена значениями двух переменных с использованием регистра ax можно записать и более компактно, например

```
x   db   ?;
y   db   ?; var x,y: integer;
   ...
    mov   ax,x; ax:=x
    xchg  y,ax; y:=ax; ax:=y
    mov   x,ax; x:= y
```

Может быть, однако, если переменные *уже* хранятся на регистрах, то можно обойтись без вспомогательной переменной? На первый взгляд это можно сделать с помощью команды **xchg**:

```
x   equ  ax
y   equ  bx;
   ...
    xchg x,y
```

Однако, если рассмотреть команду обмена на уровне микроархитектуры ЭВМ, то оказывается, что она выполняется по правилу:

```
R1:=ax; ax:=bx; bx:=R1 ,
```

где R1 – не адресуемый (служебный, невидимый программисту) регистр ЭВМ, т.е. по сути та же вспомогательная переменная.

Так что же, полная безнадёжность и без вспомогательной переменной не обойтись? Немного подумав, однако, можно предложить такой способ обмена содержимом, например, вагонов с рисом и пшеницей без использования промежуточного вагона. Можно брать по одному зерну риса и пшеницы и менять их местами, продолжая так до тех пор, пока все зерна не поменяются местами, здесь вспомогательная "переменная" совсем маленькая, равная одному зерну. Итак, задача решена! Алгоритм, правда, получился очень затратный по проведённым операциям.

С точки зрения архитектуры ЭВМ таким "зерном" является один бит данных. Можно, например, предложить следующий алгоритм обмена значениями двух регистров по одному биту за раз:

```
mov  cx,16; цикл на обмен 16 битами
shl  ax,1; CF:=ax[15]
```

```
L: rcl  bx,1; bx[0]:=CF; CF:=bx[15]
   rcl  ax,1; ax[0]:=CF; CF:=ax[15]
   loop L
```

Приведённые примеры должны помочь Вам проводить правильную спецификацию задачи. При решении задач на самостоятельных и контрольных работах, зачетах и экзаменах, вполне можно позволить себе "щадящее" предусловие вида "все входные данные хорошие, иначе пусть будет неправильный ответ или стандартная аварийная диагностика". При решении же задачи на ЭВМ этого уже делать не стоит, спецификация должна быть согласована с преподавателем (он считается заказчиком программы ☺).

Итак, можно сказать, что сутью спецификация является четкая постановка задачи. В некоторых книгах по программированию говорится, что необходимо построить *математическую модель* решаемой задачи, однако очевидно, что это относится в основном к физическим и техническим задачам.

В качестве примера построения математической модели рассмотрим такую простую задачу из школьного учебника физики. Необходимо вычислить, на какую максимальную высоту h поднимется тело, если его бросить вертикально вверх с начальной скоростью v. При выполнении осознания задачи сначала надо определить, что будет пониматься под термином "тело". Вероятно, это не может быть что-то очень легкое, вроде пушинки или песчинки, так как сопротивление воздуха не позволит нам бросить его достаточно высоко ни при какой начальной скорости. Итак, или тело бросается в пустоте, или сопротивление воздуха не учитывается, т.е. это должно быть достаточно плотное и увесистое тело, вроде камня, пули или металлического ядра.

С другой стороны, тело не должно быть и уж слишком массивным, например, весить многие миллионы и миллиарды тонн, т.к. тогда оно само будет ощутимо притягивать Землю, а это уже совсем другая задача (по крайней мере, интуитивно понятно, что такое тело при заданной скорости будет подниматься на *меньшую* высоту). При таком осознании задачи, используя знания из школьного учебника физики, математическая модель представима в виде формулы (g – ускорение свободного падения на поверхности Земли)

$$h = v^2/2g$$

Далее, важно понять, что была сделана и ещё одна спецификация задачи, которая заключается в том, что "v не слишком велико". Действительно, если начальная скорость тела превысит первую космическую скорость, то это тело уже не упадет назад на Землю. Величина первой космической скорости около 8 километров в секунду, так что уже при стрельбе вверх из артиллерийского орудия приведенная выше математическая модель становится малопригодной.

Построение новой математической модели для достаточно больших скоростей v уже несколько выходит за рамки школьной математики и приводит к следующей формуле (R – радиус Земли)

$$h = Rv^2/(2gR-v^2)$$

Кроме того, здесь предполагается, что Земля не вращается, иначе сила Кориолиса внесёт свои поправки в результат. Эта математическая модель, в свою очередь, применима лишь для скоростей, не превышающих вторую космическую скорость, выше которой поставленная задача теряет смысл, т.к. брошенное вверх тело уже не вернётся на Землю.

1.3. Разбиение на подзадачи

На этом этапе исходную задачу, если она слишком сложна для немедленной реализации, разбивают на обозримые и более простые подзадачи, совокупное решение которых приводит к решению исходной задачи целиком. В программистской литературе этот процесс 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 может возрасти объём исходного кода, увеличивается числа используемых функций и модулей, а также связей между ними. В итоге получается неэффективная программа.

В качестве примера рассмотрим типичную задачу из сборника задач [3]. Необходимо ввести три вектора A, B и C, в каждом по 100 вещественных чисел и вычислить величину S=(X,X)-(Y,Z), где запись (P,Q) обозначает скалярное произведение векторов P и Q. В качестве каждого из векторов X, Y и Z выступает один из введенных векторов A, B или C, причем конкретное значение для вектора X определяется по следующему правилу. В каждом векторе находится максимальное значение, обозначим эти значения как MaxA, MaxB и MaxC, далее за вектор X берется тот из векторов A, B или C, для кото-

рого этот максимум самый маленький. В качестве векторов `Y` и `Z` берутся два оставшихся вектора, а так как скалярное произведение коммутативно, то неважно, какой из них будет `Y`, а какой `Z`.

В качестве предусловия программы будем предполагать, что при вводе данных "все будет хорошо", т.е. в стандартном входном потоке находятся 300 правильных лексем вещественных чисел, а при вычислении скалярных произведений и их разности не произойдет выход за границу допустимых вещественных значений.

Далее необходимо выполнить спецификацию задачи. Ясно, что особым будет случай, когда сразу два или три введённых вектора имеют одинаковый максимум. Давайте в этом случае отдавать предпочтение вектору, имя которого в алфавите меньше, т.е. `A` предпочтительнее, чем `B` и т.д.

На следующем этапе необходимо разбить задачу на подзадачи, каждая из которых является законченным этапом алгоритма. Это можно сделать по-разному. Например, в качестве таких подзадач можно выбрать ввод вектора, вычисление максимальной величины вектора и вычисление итоговой разности `S`. В последней подзадаче при её дальнейшей детализации можно было бы выделить подзадачу вычисления скалярного произведения, но для простоты не будем этого делать. Теперь перейдём к следующему этапу разработки программы.

1.4. Разработка алгоритмов подзадач и их запись на языке программирования

Здесь и начинается то, что традиционно считается программированием, т.е. запись решения подзадач на выбранном языке программирования (конечно, в случае практикума на ЭВМ выбор этого языка обычно не зависит от воли студента). Будем использовать для написания программы язык Турбо-Паскаль.

Теперь необходимо решить, как подзадачи будут описаны на языке программирования. В большинстве языков программирования для детализации алгоритма (разбиение его на подзадачи) используются три уровня. На верхнем уровне вся задача разбивается на подзадачи, которые называются *модулями* (языки программирования, обеспечивающие такую возможность, называются модульными языками). Модули, в свою очередь, реализуют дальнейшую детализацию алгоритма в виде процедур и функций, а каждая процедура и функция описывает алгоритм в виде *операторов* языка программирования.[1]

Процедуры и функции в модуле обычно объединяет общее назначение, например, работа с графикой, работа с устройствами ввода/вывода и т.д. Один из модулей является головным (главным), с него начинается счет программы. В Паскале это модуль с заголовком **`program`**, он оканчивается **`end.`**, в языке С это тот модуль, который содержит функцию с именем `main` и т.д. Для рассматриваемой простой задачи будет написан только один (головной) модуль, а подзадачи будут реализованы как процедуры и функции.

Любую подзадачу на Паскале можно реализовать как в виде процедуры, так и в виде функции, но у этих способов детализации алгоритма разная *прагматика*. Функция имеет один главный результат своей работы, который и возвращается в точку вызова этой функции. Все остальные возможные результаты (изменение глобальных переменных или переменных, переданных по ссылке), называются *побочным эффектом* функции. Как известно [4], использование побочных эффектов функции нежелательно. Таким образом, функция должна иметь *один* результат своей работы. Из-за трудностей, связанных с передачей этого результата из места его вычисления (тело функции) в место использования (точка вызова функции) этот результат в большинстве языков невелик по объему занимаемой памяти (одно число, один символ, ссылка и т.д.). В процедуре все ограничения на количество, тип и размер результатов снимаются.

Первая подзадача ввода вектора возвращает введённый массив, поэтому должна быть реализована в виде процедуры.[2] Вторая и третья подзадачи, наоборот, имеют один результат – вещественное число, поэтому их следует реализовывать в виде функций. Ниже приводится текст программы для решения этой задачи.

```
program P(input,output);
  const N=100;
```

---

[1] Некоторые языки программирования (например, Паскаль), допускают, чтобы внутри описаний процедур и функций содержались описания других процедур и функций, а другие языки (например, С) этого не допускают.

[2] В языке С функция может возвращать в виде своего главного результата массив, но это самообман, так как под массивом в этом случае имеется ввиду *ссылка* на начала этого массива.

```
  type Vec=array[1..N] of real;
  var A,B,C: Vec; MaxA,MaxB,MaxC,S: real;
procedure Vvod(var X: Vec; N: word);
  var i: word;
begin for i:=1 to N do Read(X[i]) end;
function Max(var X: Vec; N: word):real;
  var i: word; M: real;
begin M:=X[1];
  for i:=2 to N do if X[i] > M then M:=X[i];
  Max:=M
end;
function RSP(var X,Y,Z: Vec; N: word):real;
  var i: word; Rez: real;
begin Rez:=0.0;
  for i:=1 to N do Rez:=Rez+X[i]*X[i]-Y[i]*Z[i];
  RSP:=Rez
end;
begin {раздел операторов главной программы}
  Vvod(X,N); Vvod(Y,N); Vvod(Z,N);
  MaxA:=Max(A,N); MaxB:=Max(B,N); MaxC:=Max(C,N);
  if (MaxA<=MaxB)and (MaxA<=MaxC)
    then { X это A } S:=RSP(A,B,C,N)
    else if (MaxB<=MaxC)
           then { X это B } S:=RSP(B,B,C,N)
           else { X это C } S:=RSP(C,A,B,N);
  writeln ('S=',S)
end.
```

Требуется обсудить эту программу. Первым обычно возникает вопрос, зачем в процедуру и функции передавать параметр `N`, задающий длину вектора. Действительно, если этот параметр в описании опустить, то внутри процедуры и функций становится видимой одноимённая глобальная константа `N`, и все будет работать по-прежнему. Для того чтобы понять желательность этого параметра, необходимо сначала сформулировать один из основных принципов в разработке подзадач, оформленных в виде процедур и функций.

Как известно, в большинстве языков программирования высокого уровня, тела процедур и функция являются блоками, так что извне всё внутри процедур и функций, включая имена формальных параметров, становится невидимым. Это одно из проявлений принципа *инкапсуляции*, который в данном случае заключается в том, что внутренняя реализация процедур и функций невидима извне и может быть изменена при изменении реализации подзадачи. Для хорошо написанной процедуры и функции, однако, должно быть верным и обратное: они не должны сами использовать в своей работе ничего из "внешнего мира", т.е. все свои входные данные они получают в качестве параметров, и все результаты возвращают тоже через параметры (и значение функции). Это позволяет минимизировать и стандартизировать *связи* между подзадачей и остальной программой.

Для приведённого примера процедуры и функции не должны знать, как в основной программе задаётся длина вектора. Это позволяет, например, свободно изменять в программе имя константы, задающей длину вектора, или вызывать процедуру ввода в виде `Vvod(X,N div 2)`, чтобы ввести только первую половину вектора и т.д. При необходимости можно пойти ещё дальше и передавать в подзадачу начальный и конечный индексы вектора

```
procedure Vvod(var X:Vec; K,N: word);
```

и записать цикл ввода в виде

```
for i:=K to N do Read(X[i])
```
[1]

Это позволит осуществить ввод данных в любую часть вектора, причем сама процедура не должна знать, что это на самом деле только часть, в не весь вектор. Запись подзадач в таком обобщенном виде

---

[1] Здесь студенты часто делают характерную ошибку, используя вместо оператора `Read(X[i])` оператор `Readln(X[i])`. При этом для задачи делается дополнительная спецификация, что в потоке `input` каждое вводимое число располагается в отдельной строке, что, конечно, ниоткуда не следует и в большинстве задач не выполняется.

позволит использовать их и в других программах, что называется повторным использованием (reuse) реализованного программного обеспечения.

На этапе записи алгоритма на языке программирование возможно проникновение в программный текст различных семантических ошибок. Это чаще происходит для языков программирования, которые не обладают хорошим (надежным) синтаксисом. Пожалуй, самым известным примером такой ошибки была запись символа точки вместо символа запятой в программе на Фортране. Эта программа управляла 22 июля 1962 года ракетой-носителем Атлас для запуска на Венеру космического корабля Маринер-1. Вместо заголовка оператора цикла `DO 50 I = 12,525` после замены запятой на точку и учитывая, что в Фортране пробелы внутри имён допускаются и игнорируются, получился оператор присваивания для неописанной вещественной переменной `DO50I = 12.525` [9]. В результате через 5 минут после старта ракету пришлось взорвать. Ущерб составил 18,5 миллионов долларов. Видно, что язык Фортран игнорирует пробелы в записи имён и чисел, позволяет не описывать переменные, а также допускает использование служебных слов не по их прямому назначению. Всё это свидетельствует о ненадёжности его синтаксиса.

1.5. Отладка, тестирование и верификация

Общеизвестно, что отладка является важным этапом разработки программ. На этом этапе автор программы (программист) составляет набор особых (отладочных) входных данных. Для каждого такого набора программист заранее знает правильный результат, который должна выдать программа. Весь такой набор входных данных, в соответствие со спецификацией задачи, должен покрывать все "особые" входные данные. Кроме того, во время отладки должны по возможности вызываться все подзадач (процедуры и функции) программы, а также выполняться все наиболее важные ветви вычислительного алгоритма.

В качестве примера рассмотрим отладку программы приближенного вычисления значения определённого интеграла на заданном отрезке методом Симпсона (парабол). В качестве отладочной подынтегральной функции нельзя брать, например, квадратичную функцию, так как в этом случае решение будет получено за одну итерацию, и соответствующий цикл не будет проверен.

Наряду с отладкой применяется и *тестирование* программного обеспечения. У отладки и тестирования программ разные цели. Целью отладки является исправление найденных ошибок и достижение определённой уверенности, что больше ошибок нет. Так как отладка – неформальный процесс, то для достаточно сложной программы практически невозможно составить набор отладочных входных данных , проверяющих все логические ветви программы. Остаётся только надеется, что какой-то важный случай не упущен, здесь должна помочь хорошая и полная спецификация программы.

Целью тестирования является нахождения таких тестовых входных данных, которые выявят хотя бы одну ошибку в программе. Для больших программных систем, чтобы не допустить конфликта интересов, тестированием занимаются люди, не являющиеся программистами, написавшими данную программу. Выявленные в процессе тестирования ошибки должны быть затем исправлены авторами программного обеспечения на дополнительной отладке. Часто фирма-разработчик программного продукта выпускает для свободного использования так называемую *бета-версию*, предлагая всем желающим пользователям принять участие в тестировании. Так иногда удаётся выявить и исправить большое количество ошибок перед выпуском так называемой *финальной* версии программной системы.

Для некоторых программ проводится и их *верификация*. Целью верификации является проверка того, что программа удовлетворяет заявленным в её описании свойствам. Например, если было заявлено, что программа способна обрабатывать входные файлы объёмом до 32 Гб, то запускается тесты, проверяющие, что это именно так.

1.6. Оптимизация программы

После проведения отладки может выясниться, что, несмотря на правильную работу, реализованная программа не удовлетворяет необходимым требованиям. Например, она занимает слишком много места в памяти ЭВМ, или работает неоправданно долго. Здесь можно вспомнить известную шутку о том, что у метеорологов есть программа, которая даёт абсолютно правильный прогноз погоды на завтра, правда, выдаёт его только послезавтра.

Оптимизация программы призвана улучшить эффективность по времени счёта и занимаемой памяти.

2. 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 при этом, что первый и второй этапы являются в значительной степени творческими, в то время как третий этап – в основном формальным процессом. Тем не менее, именно на этом формальном третьем этапе у студентов часто и возникают значительные трудности. Структурная сложность алгоритма, перенесенная в текст программы, дополняется полной или значительной не читаемостью этого текста, что делает задачу разработки и последующей отладки программы более трудоёмкой.

Человек – не компилятор. Компилятор с некоторого языка с усердием скомпилирует любой синтаксически правильный исходный текст, т.к. синтаксическая правильность является чисто формальным понятием. Часто полученная при этом программа не запустится на выполнение, не сможет отработать до конца из-за возникшего сбоя или, что вероятнее, будет работать не так, как хотелось бы программисту. При этом чаще всего анализировать сложившуюся ситуацию придется самому автору программы. У такого автора возникает типичный вопрос к коллегам и преподавателям: "А почему программа не работает (не так работает), ведь всё написано 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).

Для написания удобочитаемых текстов программ в большинстве программистских фирм приняты достаточно жёсткие *правила* записи текстов на языках программирования. Желательно уже при написании первых простейших программ выдать такие правила студентам и требовать от них следовать таким правилам [5]. На факультете ВМК МГУ для студентов также существуют такие правила.

3. Основные компоненты программы

Каждая программа обычно содержит следующие основные части (компоненты):

1. алгоритм решения задачи;
2. контроль входных данных на допустимость;
3. диалоговый интерфейс с пользователем (для диалоговых программ).

До сих пор мы в основном рассматривали правила разработки алгоритма решения задач. Рассмотрим теперь две остальные компоненты программы. Для примера рассмотрим приведённый ранее пример простой программы

```
  var x,y: integer;
begin
  read(x); y:=x+1; Writeln(y)
end.
```

Теперь, однако, попробуем переписать этот алгоритм в тождественно истинном предусловии

```
  Pred=True
```

Постусловие программы при этом тоже будет тождественно истинным

```
  Post={x∈Z ∧ x<MaxInt ⇒ y∈Z ∧ y=x+1 ∨
        x∉Z ⇒ 'Плохое x' ∨
        x∈Z ∧ x=MaxInt ⇒ 'Большое x'}
```

Сначала рассмотрим проблему контроля входных данных. Логическое слагаемое

```
x∉Z ⇒ 'Плохое x'
```

предполагает, что некорректная или выходящая за допустимые значения вводимая величина вызывает соответствующую диагностику. Чтобы понять, как это происходит, рассмотрим схему работы процедуры стандартного ввода `read(x)`. Сначала эта процедура читает из стандартного входного потока Паскаля `input` символ за символом, в попытке собрать из этих символов правильную лексему целого

числа, эта лексема заканчивается символом пробела или концом строки.[1] При поступлении неправильной лексемы целого числа фиксируется ошибка времени выполнения `Input/Output Error` и выполнение программы завершается. Примеры неправильных лексем:

`+21+4` `ABC` `123,` `130.7` и т.д.

Ясно, что вместо диагностики `Input/Output Error` нам бы хотелось продолжить выполнение программы и выдать свою диагностику "`Плохое x`". Для обеспечения этой и многих других возможностей по управлению вводом/выводом, в Турбо-Паскале пользователю предоставляется набор новых стандартных имён процедур, функций, переменных и констант. Все эти процедуры, функции, переменные и константы описаны в модуле с именем `Crt`, а чтобы сделать эти новые стандартные имена доступными, в начале программы следует поместить предложение **`Uses`** `Crt;`

Для того, чтобы для плохой лексемы не выдавать фатальную диагностику `Input/Output Error`, где-то перед процедурой ввода `read(x)` следует записать директиву `{$I-}`. При этом в случае поступления плохой лексемы аварийная диагностика не выдаётся, а выполнение программы продолжается с оператора, следующего за оператором `read(x)`. Переменная `x` при этом, естественно, не получает никакого значения. Таким образом, директива `{$I-}` при ошибке ввода/вывода *блокирует* прекращение счёта программы и выдачу стандартной аварийной диагностики.

Как же теперь, однако, узнать, прошла ли операция ввода/вывода нормально, или же нет? Для этого в модуле `Crt` предусмотрена функция без параметров с именем `IOResult`. При обращении к этой функции она выдает целое число, которое называется *кодом возврата* для последней операции ввода/вывода. Нулевое значение кода возврата означает успешное окончание последней операции ввода/вывода, а остальные (отрицательные) значения свидетельствуют о какой-либо ошибке. Например, если поток `Input` подключён к клавиатуре, то различные ненулевые значения кода возврата могут означать поступление плохой лексемы, неисправность клавиатуры, исчерпание (закрытие) входного потока[2] и т.д. Функция `IOResult` имеет одну особенность: выдача кода возврата происходит с его очисткой, так что спросить, произошла ли ошибка, можно только один раз. При повторном вызове этой функции после ошибки она уже выдаст нулевой код возврата, поэтому лучше сразу записать этот код в какую-нибудь целочисленную переменную, например: `kv:=IOResult`.

Для правильной введённой лексемы процедура `Read` пытается преобразовать эту лексему во внутреннее представление целого числа. Так как этот этап работает для ввода в целочисленные переменные любого типа, то процедура сначала строит внутреннее представление целого числа самого большого в Турбо-Паскале типа `Longint` в диапазоне от -2^{31} до $+2^{31}-1$.[3] Затем это значение присваивается вводимой переменной, в нашем случае переменной `x` типа `integer`, причем при выходе введённого значения за диапазон типа `integer`, поведение процедуры `read` зависит от значения параметра в директиве компилятора Турбо-Паскаля `{$R<параметр>}`. При отсутствии этой директивы, или её задания в виде `{$R-}` вводимой переменной присваивается *неправильное* значение, а при задании директивы в виде `{$R+}` происходит ошибка времени выполнения `Range Checking Error`. Исходя из изложенного алгоритма работы процедуры `read` ясно, что сначала нужно вводит целое число не в переменную `x` типа `integer`, а во вспомогательную переменную `temp` типа `Longint`, что мы и будем делать в нашей программе.

Рассмотрим теперь основные правила организации диалогового интерфейса с пользователем. Будем рассматривать простейший случай, когда ввод данных производится с клавиатуры, а вывод – на дисплей в текстовом режиме. Диалог ведётся между программой и пользователем[4], причём ведущей в этом диалоге является программа. Именно программа выдаёт приглашения к вводу данных (в том числе выбору из списка действий из меню), сообщения об ошибках и т.д.

---

[1] Иногда говорят, что лексема может заканчиваться также символом табуляции, однако при вводе с контролем символ табуляции поступает из входного потока `input` уже в виде последовательности пробелов.

[2] Чтобы закрыть подключённый клавиатуре поток можно ввести служебный символ `Ctrl-Z` или вызвать в программе процедуру `Close(input)`, после чего ввод из стандартного потока будет невозможен.

[3] На самом деле это диапазон от -2^{31} до $+2^{32}-1$, так как сюда включаются и значения типа `Longword`, но для нашего рассмотрения это не важно, так как внутреннее представление числа от этого не меняется.

[4] Предполагается, что пользователь является человеком, случай, когда диалог ведётся с другой программой, не рассматривается.

Рассматриваемый дисплей в основном текстовом режиме обладает следующими характеристиками. Он состоит из 25 строк, пронумерованных от 1 до 25 (по возрастанию координаты `y`), в каждой строке содержаться ровно 80 символов, пронумерованных от 1 до 80 (по возрастанию координаты `x`, см. рис 1). Без крайней необходимости на начальном этапе обучения не рекомендуется переключать дисплей в другие текстовые режимы, содержащие большее или меньшее число строк и столбцов, так как текст становится плохо читаемым.

На пересечении строки и столбца находится *знакоместо*, в которое может быть выведен один символ. У каждого знакоместа есть несколько свойств или *атрибутов*, важнейшими из которых являются цвет символа и цвет фона, на котором изображается этот символ. Множество этих цветов образует цветовую палитру. Минимально возможное число цветов, естественно, два (обычно это светло-серые символы на чёрном фоне). Далее следуют палитры из `16`, `256`, 2^{16} и 2^{32} различных цветов, бóльшее количество цветов обычно не имеет смысла, так как человеческий глаз перестаёт их различать.

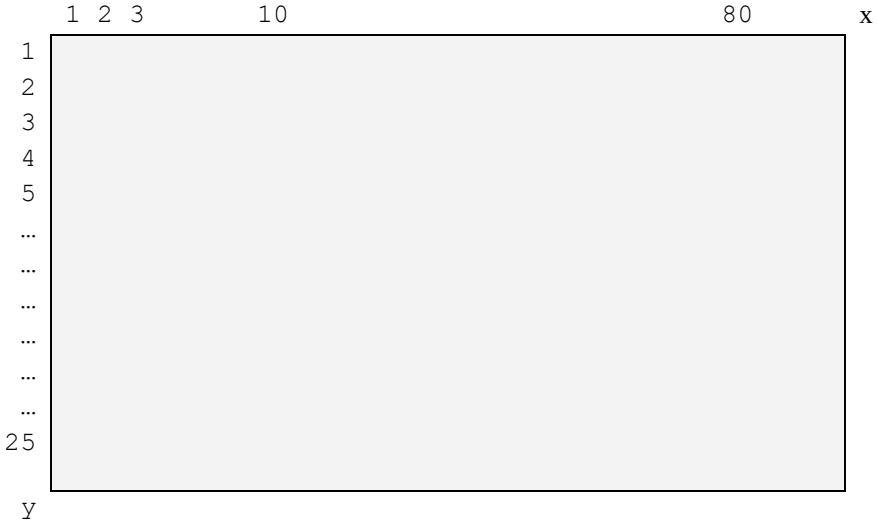

Рис. 1. Вид дисплея в стандартном текстовом режиме.

В качестве примера рассмотрим, как работать в палитре из 16 цветов, которые пронумерованы целыми числами от `0` до `15`. Можно не заучивать, какой номер соответствует какому цвету, так как в модуле `Crt` все цвета описаны в виде стандартных имён констант:

```
Const Black=0; Blue=1; …
```

Для того, чтобы увидеть описания этих констант достаточно набрать в редакторе Турбо-Паскаля имя одной из них, например, `Blue`, подвести курсор к этому имени и нажать клавиши `Ctrl-F1`. Это соответствует запросу к информационной подсистеме помощи Турбо-Паскаля: "Что известно об этом имени?". Такие же вопросы можно задавать и об остальных служебных и стандартных именах языка Паскаль.

В каждый момент времени определены текущий цвет фона и текущий цвет символа, именно этими цветами будет выводить данные стандартная процедура `Write`. Разумеется, в модуле `Crt` описаны стандартные процедуры для независимой смены цвета фона и цвета символа, но часто необходимо одновременно сменить *оба* эти цвета, для чего можно использовать такой приём. В модуле `Crt` описана целочисленная переменная со стандартным именем `TextAttr`, в которой, в частности, хранится информация о текущих цветах. Эта информация хранится в виде двух шестнадцатеричных цифр в формате

```
16*<цвет фона> + <цвет букв>
```

Для одновременной смены этих цветов достаточно присвоить этой переменной новое значение, например

```
TextAttr := 16*Blue + Yellow
```

После этого вывод на экран будет производиться жёлтыми буквами на синем фоне. *Стирание* экрана, т.е. вывод во все его позиции символа пробела текущим цветом фона, производится вызовом стандартной процедуры модуля `Crt` с именем `ClrScr`. В текущей позиции экрана находится *курсор*, именно с этой позиции производит вывод текста стандартная процедура `Write`. Текущую позицию курсора можно получить, вызвав стандартные функции без параметров `WhereX` и `WhereY`. Для уста-

новки курсора в нужную позицию экрана используется стандартная процедура `GotoXY(x,y)`. Например, вызов процедуры `GotoXY(10,5)` поставит курсор в 10 колонку 5 строки экрана. Вот, теперь у нас есть минимальные средства для организации диалогового интерфейса с пользователем.

Хорошо организованный интерфейс должен обладать, по крайней мере, тремя свойствами. В каждый момент диалога

1) на экране должно быть всё необходимое, чтобы пользователь знал, что от него требуется со стороны программы;
2) на экране не должно быть ничего лишнего, не относящегося к данному этапу решения задачи;
3) у пользователя должен быть выбор, по крайней мере, из трёх возможностей:
   - исправить ошибку, если выдана (не фатальная) диагностика об ошибке;
   - начать выполнения программы с начала;
   - завершить выполнения программы.

Ясно, что пока наша программа не обладает такими свойствами.

Для организации диалогового интерфейса надо тщательно продумать сценарий (алгоритм) диалога с пользователем. Сначала надо очистить экран от ненужной информации и выдать пользователю приглашение к вводу данных. Человек лучше воспринимает информацию, выведенную ближе к центру экрана, чем в углах или на краях, поэтому выведем приглашение к вводу целого числа, например, начиная с 8 колонки 5 строки:

```
GotoXY(8,5); Write('Введите X= ');
```

После этого программа переходит в состояние ожидания конца ввода, т.е. нажатия клавиши ENTER. Обратите внимание, что курсор сейчас находится после символа равно, а не в начале следующей строки, поэтому неверно использовать, как это часто делают начинающие программисты, вместо процедуры `Write` процедуру `WriteLn`. Далее необходимо решить, что делать при вводе неправильных данных. Отведём, например, 25 строку экрана для выдачи аварийных сообщений (рис. 2).

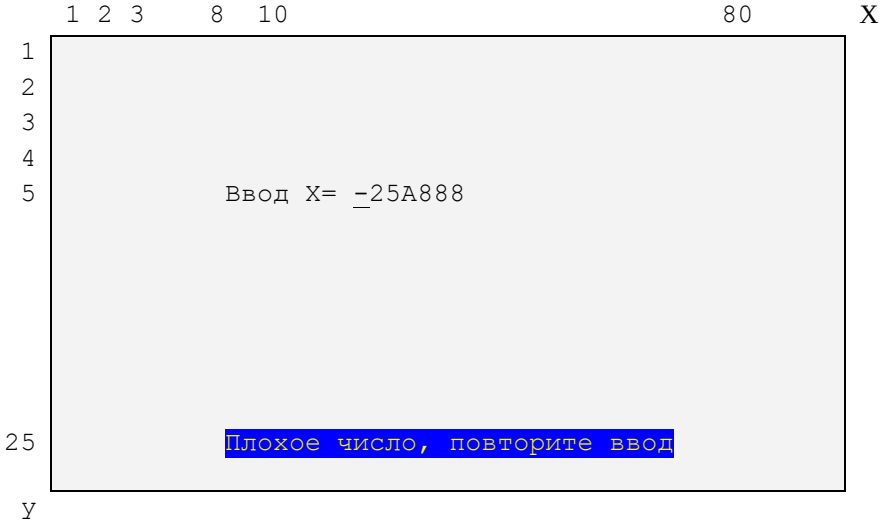

Рис. 2. Выдача аварийного сообщения.

Для привлечения внимания пользователя аварийная диагностика выведена бросающимися в глаза цветами (в данном случае жёлтыми буквами на синем фоне). После выдачи диагностики курсор снова находится в позиции ввода, причём неверно введённые данные *сохранены* на экране, чтобы пользователь видел допущенную при вводе ошибку. Далее следует решить вопрос, что делать, если пользователь ошибается при вводе снова и снова. Например, как и при вводе пин-кода в банкомате, будем после третьей неудачной попытки заканчивать программу с выдачей соответствующей аварийной диагностики.

Исходя из рассмотренных соображений, напишем первую версию программы, удовлетворяющую предусловию `Pred=`**`True`**.

```
  uses Crt; {$I-} {Блокируем ошибку ввода}
  var x,y,OldColors,ReturnCode,InpNum: integer;
      Temp: Longint; Sym: char;
begin
repeat {Главный цикл}
```

```
  ClrScr; {Очистка экрана} InpNum:=0; {Число попыток ввода}
  GotoXY(8,2);{Заголовок программы}
  Write('Ввод целого X, вывод значения Y=X+1');
  repeat {Цикл ввода числа}
    GotoXY(8,5); Write('Введите X = '); ReadLn(Temp);
    ReturnCode:=IOResult; {Получения кода возврата}
    InpNum:=InpNum+1;
    if ReturnCode<>0 then begin {Temp∉ }
      OldColors:=TextAttr; {Запоминаем текущие цвета}
      TextAttr:=16*Blue+Yellow; {Цвета диагностики}
      GotoXY(8,25); Write('Плохое число, повторите ввод');
      TextAttr:=OldColors; {Восстановим текущие цвета}
      {1}
    end else
    if (Temp<-MaxInt-1) or (Temp>=MaxInt) then begin
      OldColors:=TextAttr; TextAttr:=16*Black+LightRed;
      GotoXY(8,25); Write('Большое число, повторите ввод');
      TextAttr:=OldColors; ReturnCode:=1; {Был плохой ввод}
      {1}
    end
  until (ReturnCode=0) or (InpNum=3);
  if (ReturnCode<>0) then begin
    OldColors:=TextAttr; TextAttr:=16*Blue+White;
    GotoXY(8,25); Write('Превышено число попыток ввода');
    TextAttr:=OldColors
  end else begin {Всё хорошо !}
    x:=Temp; y:=x+1;
    GotoXY(8,10); Write('Ответ Y = ',y);
    {2}
    GotoXY(8,25); Write('Повторить Y/N ?');
    {1}
    Read(Sym)
  end
until (ReturnCode<>0) or (Sym<>'Y') and (Sym<>'y')
end.
```

В этой первой версии программы специально оставлено несколько ошибок в алгоритме организации диалога с пользователем, предполагается, что они выявятся в процессе отладки. Во-первых, диагностики об ошибках и запрос повторного выполнении программы выводятся в одно место экрана, они имеют разную длину и цвета. В результате более длинный текст может "торчать" из-под более короткого текста. Во-вторых, после вывода правильного результата не удаляется выведенная ранее диагностика об ошибке, что, конечно, неправильно. И, наконец, не фиксируется исправление пользователем ошибка при вводе числа. Например, если сначала пользователь ошибся, ввёл

```
Введите X = 1234*
```

и получил диагностику

```
Плохое число, повторите ввод
```

затем исправил ошибку, введя правильное число 56

```
Введите X = 5634*
```

В этом случае программа выдаст

```
Ответ Y: = 57
```

что выглядит весьма странно.

Для исправления этих ошибок в организации диалога можно использовать стандартную процедуру без параметров `ClrEol` из модуля `Crt`. Эта процедура, как можно догадаться из её имени, стирает все символы от позиции курсора до конца строки, при этом сам курсор не сдвигается. Места вставки этой процедуры помечены в предыдущей программе комментариями `{1}`, их надо заменить на вызов процедуры `ClrEol;`.

Для исправления введённого число вместо комментария `{2}` надо вставить строку

```
  GotoXY(8,5); Write('Введите X = ',x); ClrEol;
```

Не надо думать, что теперь в нашей программе не осталось ошибок в организации диалога с пользователем, но эти ошибки более тонкие. Например, что будет, если перед вводом числа несколько раз нажать клавишу ENTER? Кроме того, ошибка может возникнуть и при введении двух неправильных чисел подряд. Попробуйте исправить эти ошибки, предварительно хорошо изучив язык Турбо-Паскаль.

При реализации этой программы использовалась естественная концепция ввода данных пользователем. Эта концепция предполагает, что, в ответ на приглашение к вводу данных, пользователь может нажимать на любые символы, а также многократно стирать введённое и набирать заново. Другими словами, пользователь является полным хозяином положения до тех пор, пока не нажмёт клавишу ENTER, и только после этого введённые данные поступают на обработку в программу. Как было показано, при таком способе ввода возможны многочисленные ошибки.

Существует, однако, и альтернативная концепция ввода данных, иногда её называют методом "навязчивого сервиса". Суть этого метода ввода в том, чтобы не давать пользователю возможности вводить неверные данные. Для реализации такого ввода используется метод ввода символов без буферизации, без эха и без контроля. При таком вводе код нажатого на клавиатуре символа немедленно (не ожидая нажатия клавиши ENTER) передаётся в программу, при этом он не анализируется на принадлежность к служебным (управляющим) символам и не выводится на экран автоматически.

В Турбо-Паскале для такого ввода предназначена функция без параметров с именем `ReadKey`, которая возвращает символ, как только он будет нажат клавиатуре. Здесь, однако, нужно сказать, что в Турбо-Паскале, кроме основного набора символов, есть и вспомогательный набор (так как все символы в один основной 256-х символьный алфавит не поместились). Например, во вспомогательный набор входят символы стрелочек, функциональных клавиш `F1-F12`, символы с комбинациями служебных клавиш `ALT` и `CTRL` и другие. При вводе символов из дополнительного алфавита они поступают во входной поток `Input` в виде сразу двух символов, первым из которых является нулевой символ алфавита, а вторым – собственно символ из дополнительного алфавита. Например, при нажатии клавиши F1 в программу сначала поступают символ `chr(0)`, который в Турбо-Паскале удобно записывать как `#0`, а затем и символ, соответствующий клавише `F1` в дополнительном алфавите.

Итак, будем вводить из потока `Input` символы, непрерывно проверяя введённую лексему целого числа на синтаксическую правильность и сами преобразовывать число во внутреннее машинное представление. По-прежнему заблокируем выдачу диагностики о плохом вводе директивой `{$I-}`, однако теперь, так как вводятся только символы, могут возникнуть лишь такие ошибки, как закрытый для ввода поток `Input` и физическая неисправность устройства ввода. Кроме того, если поток `Input` подключён к текстовому файлу,[1] то возможны также такие ошибки, как отсутствие соответствующего файла или невозможность читать из него. При возникновении всех таких ошибок будем выдавать универсальную диагностику "Ошибка ввода" и завершать выполнение программы.

Ниже показана возможная программа, реализующая новую стратегию ввода данных. Заметим, что в Турбо-Паскале в правильном целом числе типа `integer` не более 5 десятичных цифр.

```
  uses Crt; {$I-} {Блокируем ошибку ввода}
  const Enter=#13; BS=#8; Beep=#7; Esc=#27;
  var x,y,OldColors,ReturnCode,InpNum,NumSym,PosX: integer;
      Temp: Longint; Sym,Sign: char;
      BigNumber: boolean;
begin
repeat {Главный цикл программы}
  ClrScr; {Очистка экрана} InpNum:=0; {Число попыток ввода}
  Sign:='+'; Temp:=0; BigNumber:=false;
  GotoXY(8,2);{Заголовок программы}
  Write('Ввод целого X, вывод значения Y=X+1');
  GotoXY(8,3); Write('Esc - выход из программы');
  repeat {Цикл попыток ввода числа}
    InpNum:=InpNum+1; GotoXY(8,5); Write('Введите X = ');
    Temp:=0; NumSym:=0; {Число введённых символов}
```

[1] Для подключения потока `Input` к текстовому файлу необходимо запускать программу на исполнение из командной строки в виде `C:>My_program.exe <My_File.txt`

```
    repeat {Цикл ввода символов числа}
      Sym:=ReadKey; ReturnCode:=IOResult; {Код возврата}
      if ReturnCode<>0 then begin {Ошибка ввода}
        OldColors:=TextAttr; {Запоминаем текущие цвета}
        TextAttr:=16*Blue+Yellow; {Цвета диагностики}
        GotoXY(8,25); Write('Ошибка ввода, конец программы');
        TextAttr:=OldColors; {Восстановим текущие цвета}
        ClrEol
      end else
      if (Sym in ['-','+']) and (NumSym=0) then begin {Знак числа}
        NumSym:=NumSym+1; Sign:=Sym; Write(Sym); {Эхо знака}
      end else
      if (Sym in ['0'..'9']) and (NumSym<5) then begin {Цифра}
        NumSym:=NumSym+1; Write(Sym); {Эхо цифры}
        Temp:=10*Temp+ord(Sym)-ord('0')
      end else
      if (Sym=BS) and (NumSym>0) then begin {Удалить символ}
        if NumSym=1 then Sign:='+'; {Возможно, знак числа}
        Temp:=Temp div 10; {Удаление цифры или знака числа}
        Write(BS,' ',BS); {Эхо удаления символа}
        NumSym:=NumSym-1
      end else
      if Sym=Esc then begin {Выход из программы}
        GotoXY(8,25); Write('Выход из программы');
        ReturnCode:=1; Sym:=ReadKey; {Задержка выхода}
      end else
      if (Sym<>Enter) or (NumSym=0) then begin {Плохой символ}
        if Sym=#0 then Sym:=ReadKey; {Дополнительный алфавит}
        PosX:=WhereX; {Позиция курсора}
        OldColors:=TextAttr; TextAttr:=16*Black+LightRed;
        GotoXY(8,25); Write('Плохой символ');
        Write(Beep); {Звуковой сигнал}
        TextAttr:=OldColors; ClrEol; GotoXY(PosX,5)
      end
    until (Sym=Enter) and (NumSym>0) or (ReturnCode<>0);
    if (ReturnCode=0) then begin {Проверка числа}
      if (Temp<-MaxInt-1) or (Temp>=MaxInt) then begin
        OldColors:=TextAttr; TextAttr:=16*Black+LightRed;
        GotoXY(8,25); Write('Большое число, повторите ввод');
        TextAttr:=OldColors; ClrEol; BigNumber:=True
      end
    end
  until (ReturnCode<>0) or (InpNum=3) or not BigNumber;
  if (ReturnCode=0) and (InpNum=3) then begin
    OldColors:=TextAttr; TextAttr:=16*Blue+White;
    GotoXY(8,25); Write('Превышено число попыток ввода');
    TextAttr:=OldColors
  end else
  if (ReturnCode=0) then begin {Всё хорошо !}
    x:=Temp; if Sign='-' then x:=-x;
    y:=x+1; GotoXY(8,10); Write('Ответ Y = ',y);
    GotoXY(8,5); Write('Введите X = ',x); ClrEol
    GotoXY(8,25); Write('Повторить Y/N ?'); ClrEol;
    Sym:=ReadKey
  end
until (ReturnCode<>0) or (Sym<>'Y') and (Sym<>'y')
end.
```

В приведённой программе специально допущена ошибка в алгоритме ввода числа, попытайтесь найти её и исправить.

Из приведённых примеров качественной реализации даже простейшей программы видно, насколько трудоёмок этот процесс. Принято считать, что, если для реализации одного только алгоритма решения задачи потрачена одна условная единица времени работы программиста, то для реализации полной программ потребуется, по крайней мере, в четыре раза больше времени. Это показывает, насколько трудно написать *качественное* программное обеспечение.

5. Список литературы

1. Поляков Д.Б., Круглов И.Ю. Программирование в среде Турбо Паскаль (версия 5.5). – М.: Изд-во МАИ, 1992, 576 с.
2. Грис Д. Наука программирования. – М., Мир, 1984, 416 с.
3. Пильщиков В.Н. Сборник упражнений по языку Паскаль: учебное пособие для вузов. – М.: Наука, 1989, 160 с.
4. Абрамов В. Г., Трифонов Н. П., Трифонова Г. Н. Введение в язык Паскаль. – Наука, 1988, 320 с.
5. Баула В.Г., Волканов Д.Ю., Мещеряков Д.К. Методические указания по написанию текстов программ как неотъемлемая часть преподавания практикума на ЭВМ. //Образовательные технологии. Научно-технический журнал. ISSN 1815-6851 №2(24) 2007. Воронеж 2007. С. 5-8.
6. Вирт Н. Систематическое программирование. – М., Мир, 1977, 183 с.
7. Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М., Мир, 1975, 248 с.
8. Дейкстра Э. Дисциплина программирования. – М., Мир, 1978, 275 с.
9. G.J. Myers, Software Reliability: Principles and Practices, New York, John Wiley, 1976, p. 275.
10. Бертран Мейер. Объектно-ориентированное конструирование программных систем. – Русская редакция, 2005, 1204 с.
11. Pascal. ISO 7185 :1990
12. Хоар Ч. Взаимодействующие последовательные процессы (Communicating Sequential Processes). – М., Мир, 1989, 264 с.
13. Россум Г., Дрейк Ф.Л.Дж., Откидач Д.С. Язык программирования Python. – 2001, 454 с.