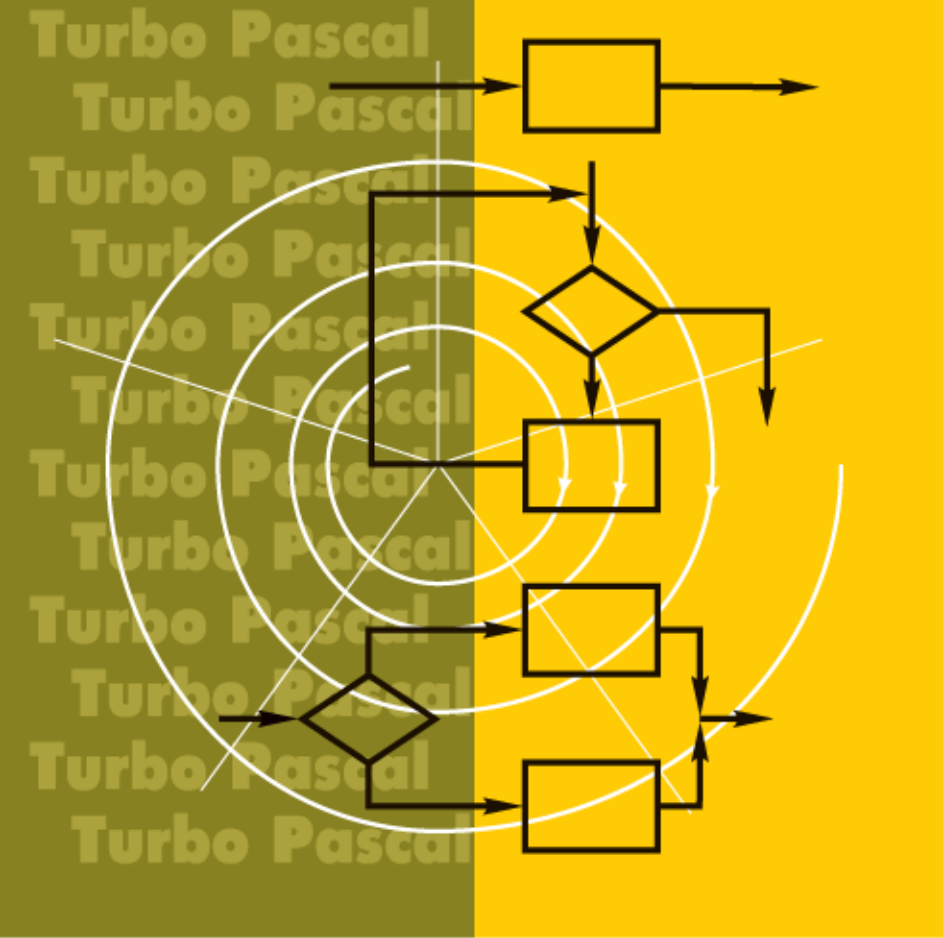
М. ПЛАКСИН

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММ

ДЛЯ ПРОФЕССИОНАЛОВ БУДУЩИХ И НАСТОЯЩИХ



М. ПЛАКСИН

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММ

ДЛЯ ПРОФЕССИОНАЛОВ БУДУЩИХ И НАСТОЯЩИХ

3-е издание (электронное)

Москва
БИНОМ. Лаборатория знаний
2015

УДК 004.42
ББК 32.973-018
П37

Плаксин М. А.

П37 Тестирование и отладка программ для профессионалов будущих и настоящих [Электронный ресурс] / М. А. Плаксин. — 3-е изд. (эл.). — Электрон. текстовые дан. (1 файл pdf : 170 с.). — М. : БИНОМ. Лаборатория знаний, 2015. — Систем. требования: Adobe Reader XI ; экран 10".

ISBN 978-5-9963-3007-2

Изложена теория тестирования и отладки программ, причем рассматриваются как вопросы, интересные начинающим программистам, так и вопросы, полезные профессионалам, например вероятностные модели оценки количества ошибок в программе и количества необходимых тестов. Описание простой в использовании высокотехнологичной методики тестирования учебных программ подкрепляется примерами создания программ, в которых тестирование выступает как неотъемлемый аспект разработки программы. Отдельная глава посвящена подробному описанию отладочных средств системы Турбо Паскаль, широко используемой в школах и вузах для обучения программированию.

Для тех, кто изучает и учит программированию: старшеклассников, студентов, преподавателей вузов, учителей; также полезна и для профессиональных программистов.

УДК 004.42
ББК 32.973-018

Деривативное электронное издание на основе печатного аналога: Тестирование и отладка программ для профессионалов будущих и настоящих / М. А. Плаксин. — М. : БИНОМ. Лаборатория знаний, 2007. — 167 с. : ил. — ISBN 978-5-94774-458-3.

ISBN 978-5-9963-3007-2

Оглавление

Введение

Учебные планы программистских факультетов большинства вузов подразумевают, что студенты-первокурсники уже умеют программировать. Однако тот курс программирования, который входит в школьную программу, недостаточен. В частности, в нем почти не уделяется внимания таким важным вопросам, как тестирование и отладка. Профессионалы знают, что затраты на тестирование и отладку оцениваются в 50–60% всех затрат на разработку программы. Но для подавляющего большинства школьников, да и многих студентов «написать программу» означает написать некий текст на языке программирования и, в лучшем случае, добиться того, чтобы в нем не было ошибок трансляции. О проверке соответствия программы поставленной задаче, поиске и исправлении смысловых ошибок, как правило, речь даже не заходит. Причиной этого в большой степени является отсутствие наглядной и доходчивой методики тестирования и отладки программ.

В данной книге изложена методика тестирования учебных программ, основанная на классических научных исследованиях и опыте преподавания начального курса программирования в старших классах средней школы и младших курсах университета. Подробно рассматриваются отладочные средства популярной системы Турбо Паскаль. В главах 14 и 15 описываются статистические модели оценки количества ошибок в программе и количества тестов, необходимых для их обнаружения. Изучение этого материала требует определенной математической подготовки. Данные главы отмечены звездочкой.

Классик жанра Гленфорд Майерс свою легендарную книгу «Искусство тестирования программ» начал с того, что предложил читателям сформировать набор тестов для проверки простенькой программы. Программа читала 3 целых числа (у Майерса — с перфокарт, мы скажем — из файла; впрочем, кто не умеет вводить числа из файла, может вводить их с терминала), рассматривала их как длины сторон треугольника и печатала сообщение о том, является ли этот треугольник разносторонним, равносторонним или равнобедренным. Далее набор тестов, предложенный читателем, надо было сравнить с набором, предложенным самим Майерсом.

Мы также используем майерсову задачу, но порядок работы несколько изменим. Читателю предлагается составить набор тестов для задачи про треугольники. Но обсуждать полученный набор немедленно мы не будем. Вместо этого мы еще несколько раз вернемся к этой задаче по ходу изучения темы с тем, чтобы читатель мог на практике применить получаемые знания и оценить свой прогресс в этой области. И только после этого мы обсудим полученный результат.

Итак, перед дальнейшим чтением вам предлагается самостоятельно составить набор тестов для вышеуказанной программы.

Готово? Тогда — продолжим.

Глава 1

В каком случае программа содержит ошибку?

Понятия тестирования и отладки связаны с процессом поиска и исправления ошибок в программе. Поэтому первый вопрос, на который надо ответить, будет звучать так: в каком случае в программе есть ошибка?

Программа содержит ошибку, если она ведет себя неразумно с точки зрения пользователя.

Это утверждение повергает новичков в замешательство: откуда же я знаю, какое поведение программы пользователь сочтет разумным? Но если вы не знаете, какое поведение программы разумно с точки зрения вашего заказчика, значит, вы не понимаете, какую задачу решаете. Как можно писать программу, не понимая, что она должна делать?

Как определить разумность поведения программы?

Во-первых, естественно, программа должна быть верна синтаксически, т. е. при ее трансляции не должно быть ошибок. Текст, содержащий синтаксические ошибки, вообще не имеет права называться программой. Такая «программа» вообще никак себя не ведет.

Во-вторых, программа должна правильно решать поставленную перед ней задачу. То есть при вводе в нее корректных исходных данных она должна выдавать правильный результат. Какой именно результат считать правильным, надо уточнить у заказчика. Например, если вам заказывают программу для вычисления квадратного корня, то должна ли она выдавать оба корня (положительный и отрицательный) или только арифметический? Корень из нуля — это ±0 или просто 0? Какова требуемая точность? Она всегда должна быть одна и та же или может меняться? Если меняться, то каким образом и по чьей инициативе? Надо ли выявлять периодические дроби? Как программа должна реагировать на отрицательное подкоренное выражение? А на предложение извлечь корень из a^2? Вы можете предложить свои ответы на все эти вопросы. Но в данном случае важно не то, что думаете по этому поводу вы, а то, что думает по этому поводу ваш заказчик. Ваша задача — выявить и сформулировать все эти вопросы, а не придумывать на них свои собственные ответы. Для выявления и формулировки вопросов полезно не хвататься за клавиатуру сразу же, как толь-

ко вам дадут задание на разработку программы, а сначала немножко подумать. Все эти вопросы все равно возникнут в процессе разработки программы. Но если их не оговорить заранее, ответы на них вы будете придумывать сами, и нет никакой гарантии, что они покажутся заказчику разумными.

В-третьих, программа не должна делать ничего лишнего. При вычислении квадратного корня не надо менять настройки операционной системы и исполнять вашу любимую мелодию.

В-четвертых, результат должен быть получен через разумное время при разумных затратах других ресурсов. Если программа при вычислении квадратного корня из числа выдает десятистраничную распечатку, в ней, наверное, что-то не так.

И наконец, в-пятых, программа должна разумно реагировать на ввод некорректных входных данных и на непредусмотренные заранее ситуации. Например, если программа вычисляет квадратный корень, а пользователь вместо подкоренного выражения ввел свою фамилию, то программа должна распознать, что введенная строка — не число[1], и сообщить об этом пользователю на его родном языке, а не заканчивать работу аварийно. Если вы разработали программу для управления аэропортом, рассчитанную на то, что в небе могут быть одновременно не более 20 самолетов, а в какой-то момент их оказалось 21, ваша программа должна отреагировать на это неким разумным образом. Например, сообщить о чрезвычайной ситуации диспетчеру и экипажу и передать этот борт диспетчеру для ручного ведения. Вариант, когда появление 21-го самолета приведет к тому, что программа потеряет один из ранее ведомых самолетов или просто отключится, совершенно недопустим.

В поддержку данного выше утверждения об ошибках в программе в [2] упоминается следующая история. При разработке системы противоракетной обороны США военные заказчики потребовали от программистов, чтобы система подавала сигнал тревоги, если обнаружит в небе враждебный летательный объект. Программисты спросили, какой объект считать враждебным. Военные ответили: любой, который не будет опознан как дружественный. Программисты добросовестно заложили в систему полученные от военных правила распознавания дружественных летательных объектов. В ночь испытания системы над горизонтом взошла Луна... Была ли в данном случае выдача сигнала к началу III Мировой войны верным поведением программы, или в ней все-таки была ошибка?

---

[1] В Турбо Паскале для преобразования последовательности литер, представляющей собой изображение числа, в число (и проверки правильности записи числа) используется процедура `Val`. Для организации собственной диагностики любое значение следует вводить как строковое и преобразовывать в число с помощью процедуры `Val`.

Глава 2

Минимальные требования к программе: функциональность и удобство использования

Минимальные требования к любой программе — это обеспечение требуемой функциональности (реализация требуемых функций) и удобство эксплуатации. Именно они должны быть проверены в первую очередь.

Почти вся оставшаяся часть книги будет посвящена тестированию функциональности. Удобство взаимодействия человека с программой останется вне нашего внимания. Относительно него ограничимся только одним замечанием. Лозунг современного интерфейса: «Не пользователь должен приспосабливаться к программе, а программа к пользователю». Даже самая простая программа должна обеспечить пользователю некоторый уровень комфорта. Например, совершенно недопустима ситуация, когда после запуска программы перед пользователем возникает пустой экран с мигающим курсором и больше ничего. Пользователю предлагается самому догадаться, что за программу он запустил и что он должен делать дальше. Как минимум, программа должна сообщить о своей функциональности и объяснить свое поведение (выдать осмысленное приглашение к вводу или сообщить, почему возникла пауза).

Глава 3

Понятия тестирования и отладки

Тестирование — это выполнение программы с целью обнаружения факта наличия в программе ошибки.

Отладка — определение места ошибки и внесение исправлений в программу.

В русском языке словосочетание «обнаружить ошибку» может иметь, по крайней мере, два смысла: «обнаружить факт наличия ошибки» и «обнаружить место, где допущена ошибка». Под тестированием понимается обнаружение ошибки в первом смысле, под отладкой — во втором.

Как правило, эти два процесса тесно взаимосвязаны, но могут быть и разделены. Например, в процессе приемки системы заказчик занимается только тестированием — поиском несоответствий между заданием на разработку программы (спецификацией программы) и разработанной программой. Если его поиски увенчаются успехом, программисту придется заниматься отладкой (определением места допущенных ошибок и исправлением программы).

Цель тестирования — обнаружение ошибок в программе.

Отметим важный психологический момент: цель тестирования — не доказать правильность программы, а обнаружить в ней ошибки! Если вы ставите себе задачей показать, что программа правильная и ошибок не содержит, то (на подсознательном уровне) и тесты будете подбирать такие, которые ошибок в программе не обнаружат.

Отсюда следующее неожиданное рассуждение. Зададим вопрос: какой тест считать удачным? Если цель тестирования — найти в программе ошибку, то удачным должен считаться тест, который обнаруживает наличие в ней ошибки. Это положение противоречит интуитивному представлению о тестировании и требует сознательного психологического настроя.

Глава 4

Принципы тестирования

1. *Ошибки в программе есть.*

Необходимо исходить из того, что ошибки в программе есть. Иначе тестирование не будет иметь для вас никакого смысла, и отношение к нему будет соответственное.

2. *Тест — это совокупность исходных данных и ожидаемых результатов.*

Очень частая ошибка заключается в том, что на вход программе подаются данные, для которых заранее не известны правильные результаты. Здесь в дело опять вступает психология. Человеческая психика устроена так, что наши глаза очень часто видят не то, что есть на самом деле, а то, что нам хочется видеть. Если заранее не зафиксировать ожидаемый результат, то всегда возникает искус объявить, что полученные результаты — это и есть то, что должно было получиться.

3. *Тестовые данные должны быть достаточно просты для проверки.*

Прямое следствие предыдущего принципа.

4. *Тесты готовятся заранее, до выхода на машину.*

Это касается как исходных данных, так и ожидаемых результатов. Реально в подавляющем большинстве случаев тесты придумываются на ходу, причем только исходные данные.

5. *Первые тесты разрабатываются после получения задания на разработку программы до написания программного кода.*

Самые первые тесты следует продумать сразу же после постановки задачи до того, как начали писать программный код. Ранняя разработка тестов позволяет правильно понять поставленную задачу. Даже в самых простых и, на первый взгляд, очевидных заданиях часто встречаются тонкости, которые сразу не видны, но становятся заметны, когда вы пытаетесь определить результат, соответствующий конкретным входным данным. (Пример уточнений, которых потребовала программа для извлечения квадратного корня, приведен в главе 1 «В каком

случае программа содержит ошибку?».) Цель ранней разработки тестов — уточнить постановку задачи, выявить тонкие места. Без этого вы рискуете написать программу, которая будет решать какую-то иную задачу, а не ту, которая была перед вами поставлена. О методике разработки тестов до написания программы речь пойдет ниже в главе 6 «Критерии черного ящика».

6. *Перед началом тестирования следует сформулировать цели, которые должны быть достигнуты в ходе тестирования.*

В частности, набор тестов должен быть полон с точки зрения выбранных критериев полноты тестирования. О критериях полноты тестирования речь пойдет в главе 6 «Критерии черного ящика», седьмой главе «Критерии белого ящика», главе 8 «Минимально грубое тестирование». Независимо от применяемых критериев разработка тестов должна вестись систематически, по определенной методике.

7. *В процессе тестирования необходимо фиксировать выполненные тесты и реально полученные результаты.*

К сожалению, обычной является ситуация, когда студент, получив задание, сразу же садится за компьютер, вводит некоторый программный текст, после нескольких перетрансляций избавляется от синтаксических ошибок, после чего запускает полученную программу, на ходу придумывает и подает на вход программы некие исходные данные, получает результаты, вводит новые данные, опять получает результаты и т. д. Тестовые данные и результаты нигде не фиксируются. Для любых нетривиальных программ подобное тестирование почти бесполезно, поскольку не позволяет отделить проверенные участки от непроверенных, не позволяет оценить количество ошибок в программе, не дает информации для принятия решения об окончании тестирования.

8. *Тесты должны быть одинаково тщательны как для правильных, так и для неправильных входных данных.*

На практике часто ограничиваются тестированием правильных входных данных, забывая о неправильных.

9. *Необходимо проверить два момента: программа делает то, что должна делать; программа не делает того, чего делать не должна.*

Особенно это важно для изменений в глобальной среде. Если программа выдает правильные результаты, но при этом затирает половину винчестера, то едва ли ее можно признать правильной.

10. *Результаты теста необходимо изучать досконально и объяснять полностью.*

11. *Недопустимо ради упрощения тестирования изменять программу.*

Тестировать после этого вы будете уже другую программу.

12. *После исправления программы необходимо повторное тестирование.*

Для того чтобы исправить обнаруженную ошибку, мы вносим изменения в программу. Но кто может гарантировать, что, исправив одну ошибку, мы не внесем другую? Увы, никто! Вероятность внесения новой ошибки при исправлении старой оценивается в 20–50%. Для особо сложных систем эта вероятность может быть значительно выше. Так, в знаменитой в свое время ОС IBM/360 количество ошибок считалось постоянным и оценивалось примерно в 1000. Система была настолько сложна и запутанна, что считалось невозможным «починить» ее в одном месте и при этом не «сломать» в другом.

Итог: после внесения изменений в программу необходимо заново прогнать весь пакет ранее выполненных тестов.

13. *Ошибки кучкуются.*

Чем больше ошибок обнаружено в модуле, тем больше вероятность, что там есть еще. Так, в одной из версий системы 370 (преемника IBM/360) 47% обнаруженных ошибок пришлось на 4% модулей [1].

На первый взгляд это утверждение противоречит здравому смыслу. «Здравый смысл» неявно исходит из предположения о равномерном распределении ошибок по всему тексту программы. На чем основано такое предположение? На взгляде на программу как на некую однородную сущность, все части которой обладают примерно одинаковыми свойствами. Реально программа устроена гораздо более сложно. В ней есть фрагменты простые и фрагменты сложные. Есть функции, которые были хорошо специфицированы, и функции, для которых спецификации были сформулированы нечетко. Есть модули, которые были спроектированы добротно, и модули, которые были спроектированы небрежно. Есть части, которые писали опытные программисты, и части, которые писали новички. Естественно, что большая часть ошибок окажется в тех частях, которые более сложны, хуже специфицированы, хуже спроектированы, написаны новичками. С учетом этих условий кучкование ошибок уже не выглядит странным.

Данный принцип имеет одно неприятное последствие. Если следовать ему строго, то количество ошибок в программе должно возрастать до бесконечности. Ведь нахождение каждой следующей ошибки увеличивает вероятность существования других еще ненайденных ошибок. К счастью, это не так. Подробнее мы поговорим об этом в главе 14 «Оценка количества ошибок в программе».

14. *Окончательное тестирование программы лучше проводить не ее автору, а другому человеку.*

Тестирование программы — процесс разрушительный. Цель его — выявление дефектов в программе. Психологически любой создатель всегда склонен в большей или меньшей степени отождествлять себя со своим созданием, «вкладывать в него душу». И чем больше усилий потрачено на работу, тем больше степень отождествления. Программа начинает восприниматься программистом как продолжение его самого. В восприятии автора обнаружение ошибок в программе приобретает трагический оттенок: «Программа — это продолжение меня. Программа — дефектна. Следовательно, я дефектен!». Если вы не склонны к мазохизму, такая ситуация вам вряд ли понравится. Выходов из нее два. Либо передать окончательное тестирование вашей программы другому человеку. Либо четко отделить себя от программы, перестать воспринимать ее как часть себя (не «моя программа», а «программа, написанная мной»). Это, может быть, проще сделать в программистском коллективе, в котором все программы являются результатом коллективной работы и коллективной собственностью. Но то же самое необходимо и при одиночной работе. Не отделив себя от своего детища, вы не сможете объективно оценить его достоинства и недостатки.

Естественно, что человек, проверяющий вашу программу, должен быть достаточно подготовлен, готов потратить на эту нужное количество времени и усилий. Это не сможет сделать первый встречный. Это нельзя сделать мимоходом.

Глава 5

Понятие полноты тестирования

Цель тестирования — обнаружить ситуацию, когда результаты работы программы не соответствуют входным данным. Самый простой способ сделать это — перебрать все возможные варианты входных данных и проверить правильность получаемых результатов. К сожалению, воспользоваться этим способом почти никогда не удается. Даже для простейших программ количество вариантов входных данных оказывается астрономическим.

Например. Пусть наша программа решает очень простую задачу. Пользователь вводит с клавиатуры два целых числа, а программа выдает их сумму. Зададим дополнительные ограничения. Будем рассматривать только шестнадцатибитные числа.

Сколько вариантов работы программы нам надо проверить? Первых слагаемых у нас будет 2^{16}. Вторых — столько же. Их комбинаций: $2^{16} \cdot 2^{16} = 2^{32}$. Поскольку $2^{10} \approx 1000$, получаем $2^{32} \approx 4\,000\,000\,000$.

Пусть на проверку одного варианта потребуется одна секунда. (Реально не любые два числа можно ввести за одну секунду, а тем более проверить полученный результат. Но для простоты пусть будет так.) Тогда для выполнения всех вариантов потребуется более 4 000 000 000 с. = 66 666 666,(6) мин. = 1 111 111,(1) ч. = 46 296,(296) суток ≈ 126,75 лет. Тестирование вашей программы (для сложения двух шестнадцатиразрядных чисел!) закончат только ваши правнуки! (Хотя откуда же им взяться, ведь 126 лет потребуется при непрерывной работе по 24 часа в сутки.)

А если программа будет складывать не два числа, а три? А ведь большинство самых элементарных программ выполняет гораздо более сложные вычисления. Не говоря уже о том, что наличие циклов `while` или `repeat` делает количество вычислений в программе потенциально бесконечным.

Отсюда печальный вывод: ***исчерпывающее тестирование*** (т. е. перебор всех возможных вариантов выполнения) для любой нетривиальной программы ***невозможно***.

Но проверять программы все-таки нужно. Как быть? Как проверить все варианты, перебрав не все? Как решить, какие из вариантов проверять надо, а какие нет? И сколько всего вариантов должно быть проверено?

Для уменьшения количества проверяемых вариантов все множество возможных вариантов выполнения программы делят на подмножества. Все варианты, попавшие в одно подмножество, считаются в некотором смысле равнозначными. Считается, что, проверив один из вариантов данного подмножества, мы тем самым проверили и все остальные варианты данного подмножества.

В системологии деление множества объектов на подмножества с одинаковыми свойствами называется **классификацией**, сами такие подмножества — **классами**, признак, по которому один класс отличается от другого, — **основанием классификации.** Например, множество людей можно поделить на мужчин и женщин по основанию «пол», на молодых, зрелых и старых по основанию «возраст» и т. д.

При переводе на язык системологии: для уменьшения количества тестов нам необходимо классифицировать все возможные варианты выполнения программы, разбить их на классы, эквивалентные с точки зрения проверки правильности программы. Нам необходимы основания для классификации возможных вариантов выполнения программы, необходимы критерии, по которым мы будем объявлять те или иные варианты выполнения равнозначными друг другу. Если такие критерии выбраны, если классификация проведена, достаточно взять по одному тесту из каждого класса. Если все они окажутся неудачными (т. е. не смогут обнаружить в программе ошибки), тестирование можно заканчивать.

Критерии, по которым проводится классификация всех возможных вариантов выполнения программы с точки зрения проверки ее правильности, называются **критериями полноты тестирования.**

После того как критерий полноты тестирования выбран, каждый тест должен анализироваться с двух точек зрения:

1) Удачен ли данный тест (удалось ли с его помощью обнаружить ошибки)?
2) Достаточен ли набор тестов с точки зрения применяемого критерия или нужны еще какие-то тесты? Если нужны, то какие? Что еще следует проверить?

Очевидно, что допущение об эквивалентности всех объектов одного класса требует большой аккуратности при проведении классификации. Для повышения надежности проводят несколько классификаций, делят множество вариантов выполнения на классы несколькими разными способами.

Существует два подхода к формулированию критериев полноты тестирования: критерии «черного ящика» и критерии «белого ящика». **Критерии черного ящика** описывают тестирование с точки зрения поставленной задачи без учета внутреннего устройства программы. **Критерии белого ящика** учитывают структуру программы.

Термин «черный ящик» позаимствован из кибернетики. В кибернетике представление системы в виде черного ящика означает описание ее входов и выходов и связей между входными воздействиями и результатами работы системы. При этом структура системы, ее внутреннее устройство никак не затрагиваются и считаются неизвестными. В качестве примера описания систем в виде черных ящиков могут служить инструкции по использованию бытовой техники.

Термин «белый ящик» появился чисто формально как противовес «черному». (Содержательно антонимом «черному ящику» должен был бы выступить «прозрачный ящик».)

При проектировании тестов начинать следует с критериев черного ящика. Собственно, это единственный возможный способ, если вы начинаете придумывать тесты до того, как написали программу (как было сказано в четвертой главе «Принципы тестирования»). После подготовки текста программы тесты, разработанные исходя из критериев черного ящика, примеряются на структуру программы. Если их оказывается недостаточно, они дополняются тестами, разработанными исходя из критериев белого ящика.

Глава 6

Критерии черного ящика

Существуют следующие критерии черного ящика:

1) тестирование функций;
2) тестирование классов входных данных;
3) тестирование классов выходных данных;
4) тестирование области допустимых значений (тестирование границ класса);
5) тестирование длины набора данных;
6) тестирование упорядоченности набора данных.

Критерий **тестирования функций** актуален для многофункциональных программ. Он требует подобрать такой набор тестов, чтобы был выполнен хотя бы один тест для каждой из функций, реализуемых программой.

Критерий **тестирования классов входных данных** требует классифицировать входные данные, разделить их на классы таким образом, чтобы все данные из одного класса были равнозначны с точки зрения проверки правильности программы. Считается, что если программа работает правильно на одном наборе входных данных из этого класса, то она будет правильно работать на любом другом наборе данных из этого же класса. Критерий требует выполнения хотя бы одного теста для каждого класса входных данных.

Критерий **тестирования классов выходных данных** выглядит аналогично предыдущему критерию, только проверяются не входные данные, а выходные.

Надо отметить, что часто эти три критерия хорошо согласуются друг с другом. При применении одного из них остальные будут удовлетворены автоматически. Если программа реализует несколько функций, то вполне естественно, что каждой из этих функций будет соответствовать свой класс входных и свой класс выходных данных. Часто существует соответствие между классами входных и выходных данных. Например, для упоминавшейся ранее программы извлечения квадратного корня выходными классами можно считать вещественные числа, комплексные числа, аналитические выражения. Соответствующими входными классами будут неотрицательные числа, отрицательные числа, аналитические выражения.

Рассмотрим программу для учета кадров предприятия. Скорее всего, она будет иметь следующие функции:

— принять на работу,
— уволить с работы,
— перевести с одной должности на другую,
— выдать кадровую сводку.

Классы входных данных:

— приказ о приеме,
— приказ об увольнении,
— приказ о переводе,
— заявка на кадровую сводку.

Классы выходных данных:

— запись о приеме,
— запись об увольнении,
— запись о переводе,
— кадровая сводка.

Этот пример хорошо демонстрирует соответствие между функциями, классами входных и выходных данных.

Тестирование области допустимых значений (тестирование границ класса). Если область допустимых значений переменной представляет собой простое перечисление (например, ноты, цвет, пол, диагноз и т. п.), надо проверить, что программа правильно понимает все эти значения и не принимает вместо них никаких иных значений. Например, как программа отреагирует на попытку ввести несуществующую ноту или пол.

Если класс допустимых значений представляет собой числовой диапазон, то понадобится более серьезная проверка. В этом случае выделяются:

1) нормальные условия (в середине класса);
2) граничные (экстремальные) условия;
3) исключительные условия (выход за границу класса).

Например, пусть программа предназначена для обработки деканатом информации об одной студенческой группе. Нормальное число студентов в группе — 20–25. Но на младших курсах их часто бывает больше, а на старших — наоборот. Пусть максимальное число студентов в группе ограничено 30. В этом случае имеет смысл проверить правильность работы программы с группой из 20 студентов (нормальные условия), с группой из 30 человек и с группой из одного человека (экстремальные условия). Необходимо проверить, как программа отреагирует на приказ

о зачислении в группу 31-го студента и об отчислении последнего остававшегося в группе студента (исключительные условия).

Особенно интересна бывает проверка минимального выхода за границу класса. Например, если значение некоторой переменной x должно лежать в промежутке [0,999; 1,999], стоит проверить, что будет, если $x = 0{,}998$ или $x = 1{,}9991$.

Иногда требуется более тонкая градация. Возможна ситуация, когда вся область допустимых значений делится на отдельные подобласти, требующие от программы разной реакции. Например, пусть программа готовит сводный отчет о работе нескольких филиалов некоторой фирмы. Известно, что каждый из филиалов ежедневно продает продукции примерно на 10 тыс. руб. Как должна реагировать программа на сообщение о том, что некий филиал за день получил 100 руб. или, наоборот, 1 000 000 руб.? Теоретически можно допустить, что этот филиал сработал настолько плохо или, наоборот, потрясающе хорошо. Однако не логичней ли предположить, что в этих случаях при вводе данных возникли проблемы с клавиатурой? В результате вся область допустимых значений делится на 3 подобласти: подобласть нормальных значений, подобласть подозрительно больших значений и подобласть подозрительно маленьких значений. Нормальные данные программа должна просто принимать к обработке. Подозрительно большие и подозрительно малые значения, в принципе, допустимы. Однако при их получении имеет смысл затребовать подтверждения, действительно ли они так выглядят.

Впрочем, вместо деления области допустимых значений на подобласти можно было бы просто выделить 3 класса входных данных: нормальные, слишком маленькие, слишком большие.

Следующие два критерия являются частными случаями, уточняющими ранее названные критерии. Но в силу их важности и частой применимости, они заслуживают отдельного упоминания.

Тестирование длины набора данных можно считать частным случаем тестирования области допустимых значений. В данном случае речь пойдет о допустимом количестве элементов в наборе. Если программа последовательно обрабатывает элементы некоторого набора данных, имеет смысл проверить следующие ситуации:

1) пустой набор (не содержит ни одного элемента);
2) единичный набор (состоит из одного-единственного элемента);
3) слишком короткий набор (если предусмотрена минимально допустимая длина);

4) набор минимально возможной длины (если такая предусмотрена);
5) нормальный набор (состоит из нескольких элементов);
6) набор из нескольких частей (если такое возможно. Например, если программа читает литеры из текстового файла или печатает текст, то как она отнесется к переходу на следующую строку? На следующую страницу?);
7) набор максимально возможной длины (если такая предусмотрена);
8) слишком длинный набор (с длиной больше максимально допустимой).

Тестирование упорядоченности входных данных важно для задач сортировки и поиска экстремумов. В этом случае имеет смысл проверить следующие ситуации (классы входных данных):

1) данные неупорядочены;
2) данные упорядочены в прямом порядке;
3) данные упорядочены в обратном порядке;
4) в наборе имеются повторяющиеся значения;
5) экстремальное значение находится в середине набора;
6) экстремальное значение находится в начале набора;
7) экстремальное значение находится в конце набора;
8) в наборе несколько совпадающих экстремальных значений.

А сейчас сделаем паузу и вернемся к задаче тестирования программы, анализирующей треугольники. Попробуйте посмотреть на составленный вами набор тестов с точки зрения критериев черного ящика. Дополните его, если в этом возникла необходимость.

Глава 7

Критерии белого ящика

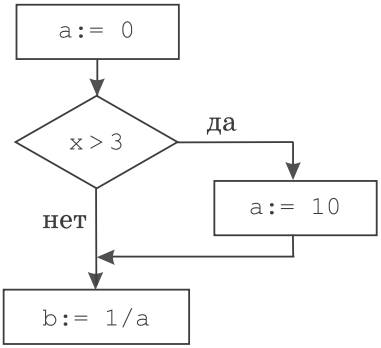
Первый из критериев белого ящика — критерий **покрытия операторов.** Он требует подобрать такой набор тестов, чтобы каждый оператор в программе был выполнен хотя бы один раз. В качестве примера рассмотрим следующий фрагмент Паскаль-программы:

Пример 1

```
a:= 0;
if x>3 then a:= 10;
b:= 1/a;
```

Для того чтобы удовлетворить критерию покрытия операторов, достаточно одного выполнения. Такого, чтобы x был больше 3. Очевидно, что ошибка в программе этим тестом обнаружена не будет. Она проявится как раз в том случае, когда $x \le 3$. Но такого теста критерий покрытия операторов от нас не требует.

Итак, мы имеем программу, оттестированную с точки зрения критерия покрытия операторов и при этом содержащую ошибку. Попробуем разобраться, в чем дело. Для наглядности перейдем с Паскаля на язык блок-схем.



Теперь причина видна сразу. Следуя критерию покрытия операторов, мы проверили только положительную ветвь раз-

вилки, но не затронули отрицательную. Сокращенная форма условного оператора в Паскале этому весьма способствует.

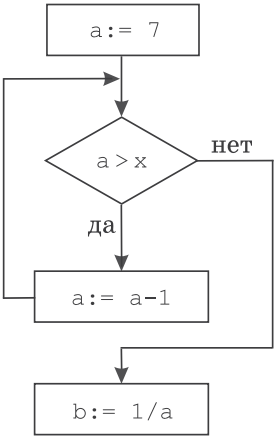
Чтобы избавиться от указанного недостатка, введем второй критерий белого ящика — критерий **покрытия ветвей** (иначе его называют критерием **покрытия решений**). Он требует подобрать такой набор тестов, чтобы каждая ветвь в программе была выполнена хотя бы один раз. Тестирование с точки зрения этого критерия обнаружит ошибку в предыдущем примере.

Рассмотрим другой пример. На Паскале он будет выглядеть так:

Пример 2

```
a:= 7;
while a>x do a:= a-1;
b:= 1/a;
```

Мы уже знаем, что паскалевская запись может служить провокатором ошибок. Поэтому сразу составим блок-схему:



Для того чтобы удовлетворить критерию покрытия ветвей, в данном случае достаточно одного теста. Например такого, чтобы x был равен 6 или 5. Все ветви программы будут пройдены (при x = 5 одна из ветвей — тело цикла — даже 2 раза). Но ошибка в программе обнаружена так и не будет! Она проявится в одном-единственном случае, когда x = 0. Но такого теста от нас критерий покрытия ветвей не потребовал.

Итак, мы имеем программу, оттестированную с точки зрения критерия покрытия ветвей и при этом содержащую ошиб-

ку. Причина в том, что некоторые ветви в программе могут быть пройдены несколько раз, и результат выполнения зависит от количества проходов. Для того чтобы учесть этот факт, введем третий критерий белого ящика — критерий **покрытия путей.** Он требует подобрать такой набор тестов, чтобы каждый путь в программе был выполнен хотя бы один раз. Тестирование с точки зрения этого критерия обнаружило бы ошибку в примере 2. Но из этого же примера виден принципиальный недостаток данного критерия. Сколько всего путей возможно в примере 2? Бесконечно много! Проверить их все невозможно. Значит, как только в программе появляются циклы с пред- или постусловием или цикл со счетчиком, но с вычисляемыми границами, количество путей в программе становится потенциально бесконечным, и критерий покрытия путей становится неприменимым. Необходим какой-то компромиссный критерий. Более жесткий, чем покрытие ветвей, но менее жесткий, чем покрытие путей. О нем мы поговорим в следующей главе.

Кроме проблем с проверкой циклов существенные проблемы связаны с проверкой сложных условий — логических выражений, содержащих знаки дизъюнкции и/или конъюнкции. Например:

```
if (a<b) or (c=0) then …
while (i<=n) and (x>eps) do …
```

И в том, и в другом операторе можно пройти по обеим ветвям, изменяя значение только одного из простых условий. Пусть $c \neq 0$. Меняя значение переменных a и b, можно пройти и по ветви «то», и по ветви «иначе». При этом ситуация, когда $c = 0$, останется непроверенной. Аналогично, пусть $i \leq n$. Меняя значения переменных x и eps, можно управлять выполнением цикла while, не проверив поведение программы при $i > n$.

Для того чтобы учесть подобные ситуации, были предложены следующие критерии:

— критерий покрытия условий;

— критерий покрытия решений/условий;

— критерий комбинаторного покрытия условий.

Критерий **покрытия условий** требует подобрать такой набор тестов, чтобы каждое простое условие (слагаемое в дизъюнкции и сомножитель в конъюнкции) получило и значение «истина», и значение «ложь» хотя бы один раз.

Критерий пытается «в лоб» исправить вышеуказанный недостаток в тестировании сложных условий. Однако сам оказывается

весьма слаб. Дело в том, что выполнение критерия покрытия условий не гарантирует покрытие ветвей. Пусть сложное условие представляет собой дизъюнкцию двух слагаемых. Например,

```
if (a<b) or (c=0) then d:= 1 else d:= 1/c;
```

При первом выполнении первое слагаемое истинно, второе ложно, вся дизъюнкция в целом истинна. При втором выполнении первое слагаемое ложно, второе истинно, вся дизъюнкция в целом истинна. Критерий покрытия условий выполнен, критерий покрытия ветвей — нет! Ошибка в программе не обнаружена.

Аналогичная ситуация возможна для конъюнкции. Например,

```
if (a<b) and (c=0) then d:= 1/c else d:= 1;
```

Чтобы исправить этот недостаток, критерии покрытия ветвей (решений) и условий объединяют в единый критерий **покрытия решений/условий.** Он требует подобрать такой набор тестов, чтобы каждая ветвь в программе была пройдена хотя бы один раз и чтобы каждое простое условие (слагаемое в дизъюнкции и сомножитель в конъюнкции) получило и значение «истина», и значение «ложь» хотя бы один раз. Критерий надежнее, чем простое покрытие ветвей, но сохраняет его принципиальный недостаток: плохую проверку циклов. Приведенный выше пример 2, «ломающий» критерий покрытия ветвей, «сломает» и критерий покрытия решений/условий. Ошибка в данном случае проявится только при фиксированном количестве повторений цикла (в примере 2 — семикратном), а критерий покрытия решений/условий не гарантирует, что повторений будет именно столько.

Совмещение критериев покрытия ветвей и покрытия условий не решает также всех проблем, порождаемых сложными условиями. Ошибки могут быть связаны не со значением того или иного простого условия, а с их комбинацией. Например, в фрагменте:

Пример 3

```
if (a=0) or (b=0) or (c=0)
   then d:= 1/(a+b)
   else d:= 1;
```

ошибка будет выявлена только при одновременном равенстве нулю двух переменных: `a` и `b`. Критерий покрытия решений/условий не гарантирует проверки такой ситуации.

Для решения этой проблемы был предложен критерий **комбинаторного покрытия условий**, который требует подобрать такой набор тестов, чтобы хотя бы один раз выполнялась любая комбинация простых условий. Критерий значительно более надежен, чем покрытие решений/условий, но обладает двумя существенными недостатками. Во-первых, он может потребовать очень большого числа тестов. Количество тестов, необходимых для проверки комбинации n простых условий, равно 2^n. Комбинация двух условий потребует четырех тестов, трех условий — восьми, четырех условий — шестнадцати тестов и т. д. Во-вторых, даже комбинаторное покрытие условий не гарантирует надежную проверку циклов. Тот же самый пример 2, который демонстрировал недостатки покрытия ветвей и покрытия решений/условий, покажет и недостаток комбинаторного покрытия условий.

Критерии белого ящика требуют знания текста программы. Текста программы, анализирующей треугольники, у нас нет. Тем не менее читателю предлагается еще раз вернуться к сформированному ранее набору тестов. Не возникнет ли желания его модифицировать?

Глава 8

Минимально грубое тестирование

Обзор критериев белого ящика, данный в предыдущей главе, подводит к необходимости построения некоторого компромиссного критерия, который, с одной стороны, обеспечивал бы достаточную надежность тестирования, а с другой — имел приемлемую сложность. В качестве такого компромисса был предложен критерий **минимально грубого тестирования** (МГТ). МГТ представляет собой критерий покрытия решений/условий, усиленный дополнительными требованиями по проверке циклов. Проверка циклов организуется по следующим правилам:

1) для каждого цикла с предусловием должна быть проверена правильность при нулькратном, однократном и многократном повторении тела цикла. Многократным считается любое повторение более одного раза;
2) для каждого цикла с постусловием должна быть проверена правильность при однократном и многократном повторении тела цикла;
3) проверка цикла со счетчиком зависит от того, фиксированы ли границы изменения счетчика или вычисляются. Вообще говоря, как и для цикла с предусловием, требуется проверка при нулькратном, одно- и многократном повторении тела цикла.

Одно из преимуществ критерия минимально грубого тестирования состоит в том, что для него существует очень удобное представление в форме таблицы, которое позволяет контролировать степень выполнения критерия и придумывать тесты прицельно для проверки именно нужных частей программы. Строится таблица МГТ следующим образом.

Строки таблицы соответствуют проверяемым условиям, графы — тестам. Для каждого условного оператора в таблице МГТ создаются 2 строки: для ветви «то» и ветви «иначе».

		Тест 1	Тест 2	Тест 3	Тест 4	Тест 5
`if a>b`	+					
	—					

Для каждого цикла с предусловием — 3 строки: для нулькратного, однократного и многократного повторения тела цикла.

		Тест 1	Тест 2	Тест 3	Тест 4	Тест 5
`while a>b`	=0					
	=1					
	>1					

Для каждого цикла с постусловием — 2 строки: для однократного и многократного повторения тела цикла.

		Тест 1	Тест 2	Тест 3	Тест 4	Тест 5
`repeat until a>b`	=1					
	>1					

Для каждого цикла со счетчиком — 3 строки: для нулькратного, однократного и многократного повторения тела цикла.

		Тест 1	Тест 2	Тест 3	Тест 4	Тест 5
`for k:=1 to n`	=0					
	=1					
	>1					

Если какая-либо из этих строк для данной программы не имеет смысла (например, по условиям задачи тело цикла обязательно повторяется хотя бы один раз), соответствующая строка все равно помещается в таблицу, но в ней записывается пояснение, что такая ситуация невозможна.

Для каждого оператора выбора записывается столько строк, сколько он имеет ветвей (включая ветвь «иначе»):

		Тест 1	Тест 2	Тест 3	Тест 4	Тест 5
`case color`	red					
	green					
	blue					
	else					

Если в условном операторе, в цикле с пред- или постусловием стоит сложное условие, то к строкам, соответствующим самому оператору, добавляются по 2 строки на каждое простое условие:

		Тест 1	Тест 2	Тест 3	Тест 4	Тест 5
`if (a>b) and (x=y)`	+					
	—					
`a>b`	+					
	—					
`x=y`	+					
	—					

		Тест 1	Тест 2	Тест 3	Тест 4	Тест 5
`while (c>d) or (z=f)`	=0					
	=1					
	>1					
`c>d`	+					
	—					
`z=f`	+					
	—					

Для удобства ссылок все условия в таблице можно перенумеровать.

Каждая строка таблицы МГТ соответствует одной из ситуаций, которую надо проверить в процессе тестирования. Проверка может быть произведена одним или несколькими тестами. Отметим в таблице те тесты, которые проверяют данную ситуацию. Для этого поставим плюс в ячейку, находящуюся на пересечении соответствующих строки и столбца.

			Т1	Т2	Т3	Т4	Т5
У1	`if (a>b) and (x=y)`	+	+				
		—		+	+		
У2	`a>b`	+	+	+			
		—			+		
У3	`x=y`	+	+		+		
		—		+			
У4	`while (a>b) or (x=y)`	=0					
		=1	+				
		>1		+	+		
У5	`a>b`	+	+		+		
		—	+	+	+		
У6	`x=y`	+		+	+		
		—	+	+	+		

Такая запись означает, что при выполнении теста Т1 в развилке У1 выполнялась ветвь «то». При этом, естественно, условия У2 и У3 были истинны. Тело цикла У4 было повторено 1 раз. Условие У5 сначала было истинно, затем стало ложно. Условие У6 было ложно все время.

При выполнении теста Т2 в развилке У1 выполнялась ветвь «иначе». Произошло это за счет того, что условие У3 стало ложным, хотя условие У2 осталось истинным. Цикл У4 повторялся более одного раза. Условие У5 при этом все время было ложно. Условие У6 сначала было истинно, а затем стало ложным.

Тест Т3 придумывался уже прицельно, для того чтобы «закрыть» строку, соответствующую ложному значению условия У2. Во всем остальном он перепроверяет то, что уже было проверено раньше.

Из таблицы видно, что плюс отсутствует в единственной строке — строке, соответствующей нулькратному повторению цикла У4. Это та ситуация, которая осталась непроверенной. Значит, именно на проверку этой ситуации должен быть нацелен очередной тест.

Возможна ситуация, когда некоторая развилка данным тестом не затрагивается вообще. Например, в случае вложенных условных операторов:

```
if a>b then … else if c=d then …
```

В случае выполнения внешнего оператора по ветви «то», вложенный оператор выполняться не будет вообще. В этом случае в МГТ-таблице просто остаются пустые ячейки:

			Т1	Т2	Т3	Т4	Т5
У1	`if a>b`	+	+				
		—		+	+		
У2	`if c=d`	+		+			
		—			+		

В случае разбиения программы на процедуры, модули и т. п. МГТ-таблицы строятся для каждой процедуры, модуля и пр. Соответственно, тестирование проводится прицельно для данной процедуры, для данного модуля.

Критерий МГТ не идеален, как и все предыдущие. Он не гарантирует проверку комбинации всех простых условий. Он не гарантирует надежную проверку циклов. Приведенный в преды-

дущей главе пример 3, который «ломал» критерий решений/условий, «сломает» и критерий МГТ. Выполнение требований минимально грубого тестирования не гарантирует проверку всех возможных комбинаций простых условий. Ошибка в примере 2, выявление которой не гарантировали критерии покрытия ветвей, решений/условий и комбинаторного покрытия условий, может быть пропущена и критерием МГТ. Тем не менее критерий МГТ представляется достаточно разумным компромиссом между надежностью и сложностью тестирования.

Применение критерия МГТ связано с вопросом, который повергает в ужас новичков-программистов. Это вопрос о количестве «писанины», требуемой критерием. Неужели так уж необходимо строить и заполнять все эти таблицы? Ответ: да. Особенно для новичков. Задача новичка добиться, чтобы соответствующие критерии перешли «в подкорку», «на уровень подсознания», начали использоваться автоматически. А для этого необходимо достаточно много раз применить их «врукопашную». Благо, первые учебные программы, с которых начинается курс программирования, достаточно просты, и МГТ-таблички для них совсем невелики. Кроме того, не возбраняется для ведения МГТ-таблиц использовать компьютер, который способен существенно упростить многие рутинные операции.

Как и в двух предыдущих разделах, с учетом полученных знаний читателю еще раз предлагается вернуться к проверке программы, анализирующей треугольники. Вдруг МГТ натолкнет на какие-то новые мысли по этому поводу.

Глава 9

Ошибкоопасные ситуации

Ни один из критериев полноты тестирования, описанных в предыдущих главах и применимых на практике, не дает стопроцентной гарантии отсутствия ошибок. Поэтому в дополнение к критериям рекомендуется еще один способ поиска ошибок в программе. Это просмотр текста программы для выявления и проверки ошибкоопасных ситуаций. Далее приводится список таких ситуаций, на которые стоит обращать внимание при просмотре программы. Список, к сожалению, не исчерпывающий. Каждый может продолжить его сам в соответствии с собственным опытом.

9.1. Обращение к данным

1. Использование значения переменной.

Опасность: Неинициализированная переменная. (Переменная используется до того, как ей было присвоено значение.)

Очень частая ошибка начинающих программистов. Признаком ее служит появление у переменных неожиданных «мусорных» значений (очень больших чисел, чисел с громоздкой дробной частью, строк, забитых «мусором», и пр.). К сожалению, обнаруживать эту ошибку автоматически умеют очень немногие вычислительные системы.

2. Автоматическая инициализация переменных.

Опасность: Неверная инициализация.

Некоторые системы программирования автоматически заполняют выделяемую память стандартными значениями, чаще всего нулями. Всегда или при включении соответствующего режима. Проверьте, делает ли это ваша система. Использует ли она нужное вам значение? Можно ли управлять процессом инициализации? Если возможно, от данной услуги лучше отказаться. Сэкономите вы на ней мизер, зато рискуете не заметить ситуацию, когда была необходима ручная инициализация, а вы о ней забыли.

Турбо Паскаль обнуляет память под переменные, описанные в главной программе. Переменные, описанные в подпрограммах, не инициализируются и получают изначально некоторые «мусорные» значения.

3. Индексация массива.

Опасность: Выход индекса за границу измерения.

Как правило, существует возможность автоматического контроля выхода индекса за границу. Правда, начинающие программисты, уверенные в собственной гениальности, любят этот контроль отключать. Ведь это позволит им сэкономить несколько микросекунд при выполнении программы, для которой время выполнения вообще не играет никакой роли. Зато поиск соответствующей ошибки превращается в увлекательную и, часто, достаточно длительную процедуру.

В Турбо Паскале контроль индексации массивов регулируется двумя способами:

1) с помощью управляющих комментариев {$R+} и {$R–}. Первый из них контроль включает, второй — выключает;
2) переключением опции компилятора **Options/Compiler.../Range checking** (см. рис. 9.1).

По умолчанию контроль, к сожалению, выключен.

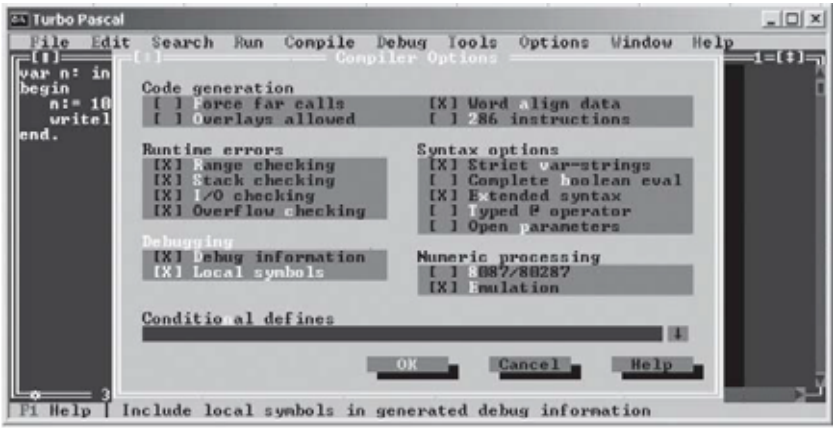


Рис. 9.1. Опции компилятора Турбо Паскаль

4. Изменение переменной внутри блока.

Опасность: Побочный эффект (изменение глобальной переменной при выполнении подпрограммы).

В языках с блочной структурой это не является ошибкой, но требует повышенного внимания. Ведь по внешнему виду вызова процедуры или функции никак нельзя сказать, какие глобальные переменные будут изменены при ее выполнении.

5. Использование значения ссылочной переменной.

Опасность: «Висячая ссылка».

Переменная ссылочного типа указывает на область памяти, которая уже возвращена системе. Особенно неприятно, если эта область уже будет перераспределена под другую переменную.

6. Схожие имена переменных.

Само по себе это не ошибка. Но может быть причиной описки. Соответственно требует повышенного внимания.

7. Использование записи с вариантами.

Опасность: Несколько полей записи с вариантами используются для обращения к одной и той же области памяти при отсутствии контроля за правильной типизацией.

Само по себе это не ошибка. Но может оказаться, что эти поля имеют разные типы. В этом случае величина, записанная как значение одного типа, может быть прочитана или откорректирована как значение другого типа. Пример см. на рис. 9.2.

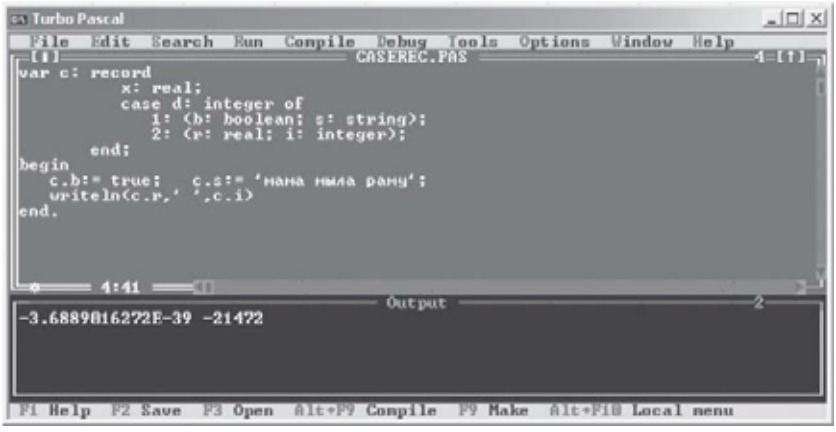


Рис. 9.2. Нарушение типового контроля при обращении к записи с вариантами

8. Использование нетипизированного указателя.

Опасность: Еще одна возможность обойти типовый контроль. Динамическая переменная, на которую ссылается данный указатель, в разных местах программы может трактоваться как переменная разных типов.

9. Обращение к одной и той же области памяти из разных модулей.

Само по себе это не ошибка. Но имеет смысл проверить, все ли модули рассматривают переменную в одном и том же смысле.

10. Использование статических переменных. Использование типизированных констант в Турбо Паскале.

Опасность: Путаница между статическими и динамическими переменными.

Статические переменные создаются в единственном экземпляре и сохраняют значения между вызовами процедуры. Динамические переменные создаются при каждом входе в процедуру и уничтожаются при выходе из нее. Дело осложняется тем, что в Турбо Паскале статические переменные официально отсутствуют, «спрятаны» под видом типизированных констант. Пример см. на рис. 9.3.

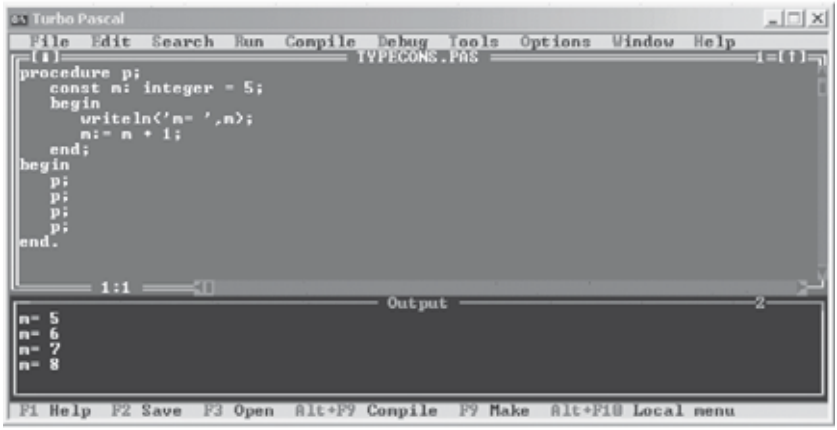


Рис. 9.3. Типизированные константы (статические переменные) в Турбо Паскале

11. Использование переменных, не имеющих явного описания.

Ряд языков не требует явного описания переменных. В любой точке программы вы можете вставить в текст программы новое имя и тем самым создать новую переменную. Подобное неявное описание требует повышенного внимания по трем причинам. Во-первых, оно опасно с точки зрения описок. Неверно введенное имя будет просто воспринято как неявное описание новой переменной. Во-вторых, имеет смысл уточнить, какой

тип приписывается по умолчанию автоматически создаваемой переменной (если приписывается). В-третьих, стоит уточнить, какова область действия (область видимости) автоматически созданной переменной.

К счастью, Паскаль неявных описаний не допускает.

9.2. Вычисления

1. Выражение.

Опасность: Неверный порядок вычисления операций в выражении.

Как вы думаете, в каком порядке будет вычисляться выражение `m<n or i=k`? Математик будет ожидать дизъюнкции двух сравнений. А в Паскале сначала будет выполнена дизъюнкция `n or i`, а потом ее результат будет сравниваться со значением переменной `m`. К этому надо добавить, что в Турбо Паскале (не в стандартном!) дизъюнкция применима к целым числам.

Особенно опасно непонимание порядка выполнения операций в сочетании с отсутствием строгого типового контроля.

2. Логическое выражение.

Опасность: Путаница между полной и краткой схемами вычисления логических выражений.

Одно из отличий логических вычислений от арифметических в том, что возможных результатов всего два — «истина» и «ложь». Поэтому очень часто результат выражения становится ясен без выполнения всех вычислений. Если первое слагаемое в дизъюнкции истинно или первый сомножитель в конъюнкции ложен, остальные можно уже не вычислять. С точки зрения математики совершенно неважно, прервем мы вычисления после первого слагаемого (сомножителя) или будем продолжать дальше. Но в программировании это может быть важно. Например, если второй операнд — это вызов функции, которая изменяет значения глобальных переменных или записывает значения в файл: `b and f(x)`. Если второй операнд будет вычисляться, глобальная переменная или файл будут изменены, не будет — останутся прежними.

Краткая схема вычисления логических выражений означает, что вычисления будут прерваны, как только будет ясен результат. Полная — что выражение вычисляется до конца, независимо от промежуточных результатов. Схема вычисления

может быть зафиксирована в языке, а может регулироваться в системе программирования.

В Турбо Паскале схема вычисления логических выражений регулируется двумя способами:

1) с помощью управляющих комментариев {$B+} и {$B–}. Первый из них включает полную схему вычисления логических выражений, второй — краткую;
2) переключением опции компилятора **Options/Compiler…/ Complete Boolean eval** (см. рис. 9.1).

По умолчанию устанавливается краткая схема.

К сожалению, переключение опции компилятора никак не отражается в тексте программы. В результате в Турбо Паскале появляется уникальная возможность писать программы, по внешнему виду которых нельзя сказать, как они будут выполняться. Пример приведен на рис. 9.4 и 9.5. В логическом выражении `(1 =1) or f` первое слагаемое всегда истинно. Значит, дизъюнкция в целом будет истинна, независимо от второго слагаемого. При вычислении по краткой схеме оно вычисляться не будет, при полной — будет. Но второе слагаемое — это вызов логической функции, которая выдает сообщение («Вызов функции f») и меняет значение глобальной переменной `m`. В результате при краткой схеме сообщение выдано не будет, переменная `m` сохранит нулевое значение. При полной схеме сообщение будет выдано и `m` станет равно единице. Выполнение одной и той же программы дает разные результаты, хотя никаких изменений в тексте программы нет.

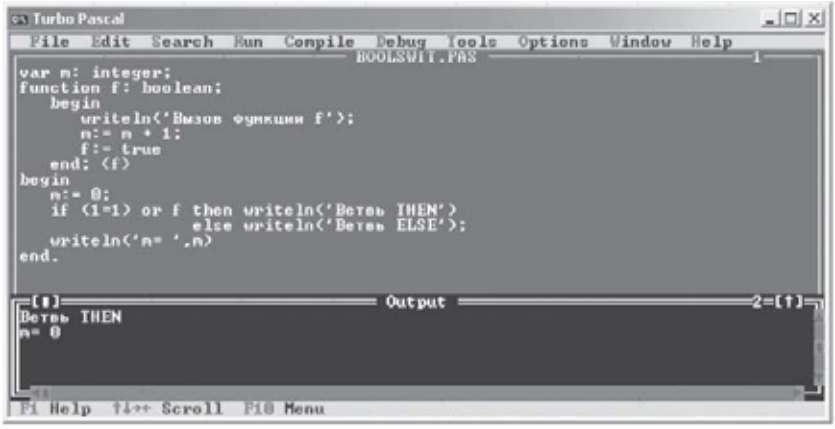


Рис. 9.4. Вычисление логического выражения по краткой схеме

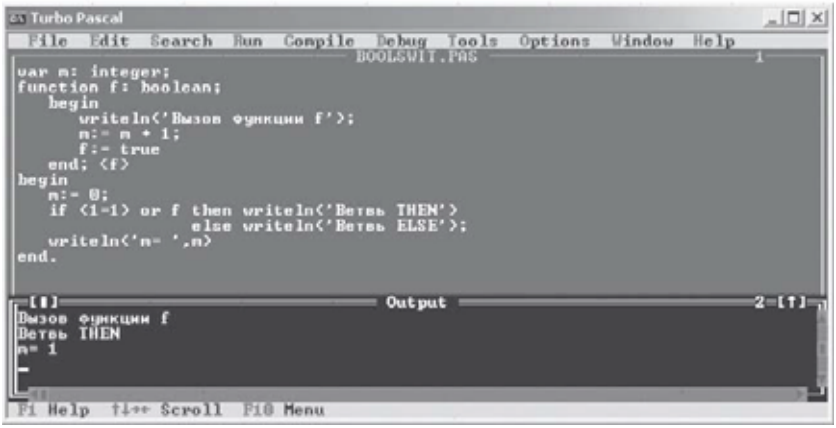


Рис. 9.5. Вычисление логического выражения по полной схеме

3. Сравнения $>$ / $<$.

Опасность: Потеря третьего результата операции сравнения.

В математике сравнение двух чисел может иметь 3 исхода: больше, меньше, равно. В языках программирования операции сравнения — логические, т. е. имеющие только 2 исхода. Третий исход зачастую оказывается потерянным. Для «полноценного» сравнения требуется не один условный оператор, а два вложенных:

```
if a>b then … else if a=b then … else {a<b} …
```

4. Сравнения $>$ и $\geq$, $<$ и $\leq$.

Опасность: Ошибки типа «± 1».

Частая ошибка — путаница сравнений на строгое и нестрогое неравенство: $>$ и $\geq$, $<$ и $\leq$. Если сравнение стоит в заголовке цикла, то он будет повторяться на один раз больше или меньше требуемого при неправильном выборе знака сравнения. Отсюда и название типа ошибки.

5. Деление.

Опасность: Деление на нуль.

Убедитесь (приведите правдоподобные рассуждения в пользу того), что выражение в знаменателе не может оказаться равным нулю.

6. Извлечение квадратного корня.

Опасность: Корень из отрицательного числа.

Убедитесь, что аргумент не может получить недопустимое значение.

7. Взятие логарифма.

Опасность: Логарифм неположительного числа.

Убедитесь, что аргумент не может получить недопустимое значение.

8. Использование в вычислениях данных «не того» типа.

Опасность: Неверное приведение типов данных.

Например, использование литерных значений в арифметических операциях или целочисленных значений — в логических. Строго типизированные языки такое запрещают. Но не все языки строго типизированы. В слабо типизированных языках это не обязательно ошибка. Но имеет смысл уточнить, как именно будут выполняться операции. Например, какой результат даст сумма `1 + '1'`? Чему будет равна литерная единица: целочисленной единице или коду литеры '1'?

Обратите внимание на то, что Турбо Паскаль (в отличие от стандартного Паскаля) распространил логические операции на целые числа. Попробуйте угадать, чему равно `724 or -5308`.

9. Присваивание целой переменной вещественного значения.

Опасность: Способ преобразования вещественного числа в целое.

Само по себе присваивание целой переменной вещественного значения может языком допускаться. При этом проводится автоматическое преобразование вещественного числа в целое. Проводиться оно может двумя путями: округлением или обрубанием дробной части. Какой способ будет использован в вашем случае? В зависимости от ответа на этот вопрос, следующий фрагмент даст разные результаты.

```
a: real;
n: integer;
...
a:= 0.9999;
n:= a;
...
```

В случае округления `n` получит значение 1, в случае обрубания — 0.

К счастью, Паскаль такое присваивание вообще запрещает.

10. Вычисления с плавающей точкой (вещественная арифметика).

Опасность 1: Погрешности округления.

Машинные вычисления не всегда совпадают с арифметическими. В первую очередь это относится к вещественным значениям. Причин тому две. Во-первых, в машинной памяти числа хранятся не в десятичном формате, а в двоичном. Преобразование выполняется с некоторым округлением. Во-вторых, точность представления чисел в памяти ограничена. В результате вещественная единица вполне может оказаться равна 0,9999999999, а может — 1,0000000001 (количество знаков после запятой зависит от точности представления). В качестве примера приведем пару присваиваний:

```
a:= 1/3;
b:= a*3;
```

В математике `b` будет равно 1. В программировании для начала окажется

$$a = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \frac{1}{1024} + \frac{1}{4096} + \frac{1}{16384} + \frac{1}{65536}$$

и т. д. в зависимости от точности представления. Это не совсем одна треть. В конце концов, наберется сумма, которая с заданной точностью даст `a` = 0,3333333333. Но тогда `b` = 0,9999999999, а не 1.

Если таких значений в выражении будет несколько или выражение многократно перевычисляется в цикле, ошибка округления будет копиться. Автор этих строк сам как-то столкнулся с ситуацией, когда сумма трех слагаемых, каждое из которых было равно 0,1, оказалась больше 0,3! В программе на языке Reduce тело цикла

```
for k:= 0 step 0.1 until 0.3 do …
```

было повторено только 3 раза. К счастью, в большинстве языков счетчик цикла не может быть вещественным. Но поскольку в таких конструкциях есть объективная необходимость, его приходится моделировать. Поэтому о накоплении погрешностей надо помнить при любых вещественных вычислениях. Пример см. на рис. 9.6.

Опасность 2: Потеря значимости (получение чисел с очень маленькой мантиссой).

Слишком маленькая мантисса в машинной арифметике может превратиться в нуль. Пример см. на рис. 9.7.

Рис. 9.6. Накопление погрешности при вещественных вычислениях

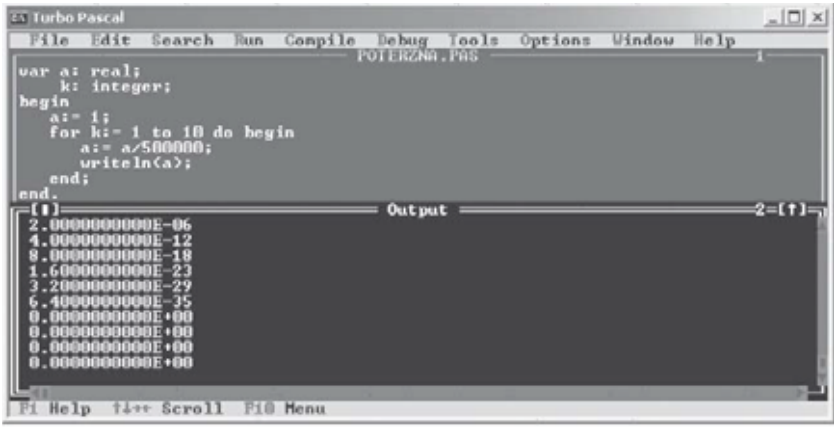


Рис. 9.7. Потеря значимости

11. Сравнение вещественных чисел.

Опасность: Погрешности округления.

Погрешности округления чреваты ошибками не только в арифметических операциях (о чем уже было сказано). В операциях сравнения они могут привести к тому, что «лобовое сравнение» вещественных чисел даст неверный результат. Пусть вещественные переменные `q` и `r` обе равны единице. Но в одном случае вещественная единица будет представлена как

0,9999999999, а в другом — как 1,0000000001. В этом случае сравнение `q = r` вполне может дать значение «ложь». Лучше сравнивать модуль разности с некоторым ε: `abs(q – r) < eps`.

12. Арифметические вычисления (как вещественные, так и целые).

Опасность 1: Переполнение (получение очень больших чисел).

Еще одна особенность машинной арифметики: слишком большие числа ей противопоказаны. В лучшем случае произойдет аппаратное прерывание, но возможны более неприятные случаи, когда сумма двух больших положительных целых чисел окажется числом отрицательным или положительным, но маленьким.

В Турбо Паскале вещественное переполнение всегда вызывает аппаратное прерывание. Что касается целочисленного переполнения, то контроль за ним регулируется. Делается это, как обычно в Турбо Паскале, двумя способами:

1) с помощью управляющих комментариев {$Q+} и {$Q–}. Первый из них включает контроль целочисленного переполнения, второй — отключает;
2) переключением опции компилятора **Options/Compiler…/ Overflow checking** (см. рис. 9.1).

По умолчанию контроль целочисленного переполнения отключен.

Примеры переполнений приведены на рис. 9.8–9.10. На рис. 9.8 при вычислении цикла произошло прерывание из-за вещественного переполнения.

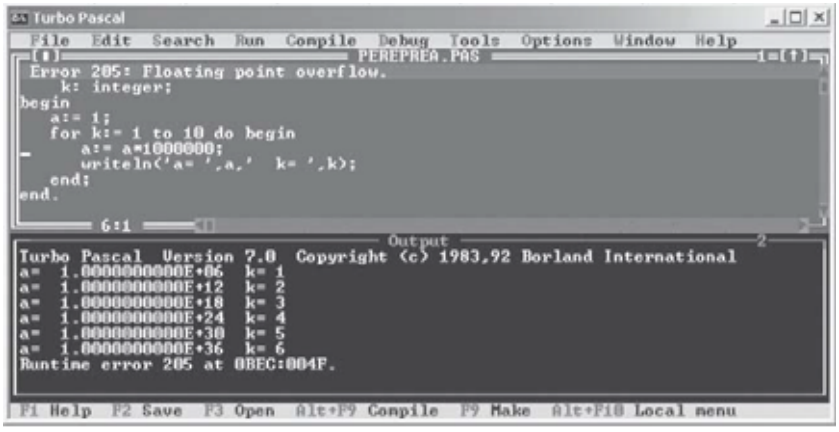


Рис. 9.8. Вещественное переполнение

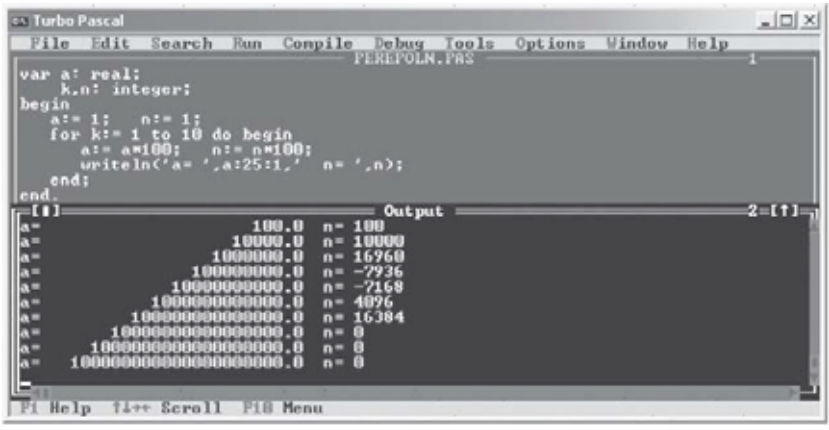


Рис. 9.9. Целочисленное переполнение при отключенном контроле

На рис. 9.9 программа выполняется при отключенном контроле целочисленного переполнения. Переполнение происходит при вычислении переменной `n`. Вычислительная система на него никак не реагирует. На примере хорошо видно, как меняется значение целочисленной переменной `n` в результате многократного умножения:

$$100 \cdot 100 = 10\,000,$$
$$10\,000 \cdot 100 = 16\,960,$$
$$16\,960 \cdot 100 = -7\,936,$$
$$-7\,936 \cdot 100 = -7\,168$$

и т. д.

На рис. 9.10 та же самая программа выполняется при включенном контроле. В этом случае целочисленное переполнение приводит к аппаратному прерыванию так же, как и вещественное.

Надо подчеркнуть, что диапазон значений типа `integer` в Турбо Паскале весьма невелик: от –32 768 до 32 767. Но в Турбо Паскале (в отличие от стандартного Паскаля) существует еще несколько целых типов, как большего, так и меньшего размера. В частности, тип `longint` охватывает значения от –2 147 483 648 до 2 147 483 647.

Опасность 2: Переполнение или потеря значимости в промежуточных вычислениях.

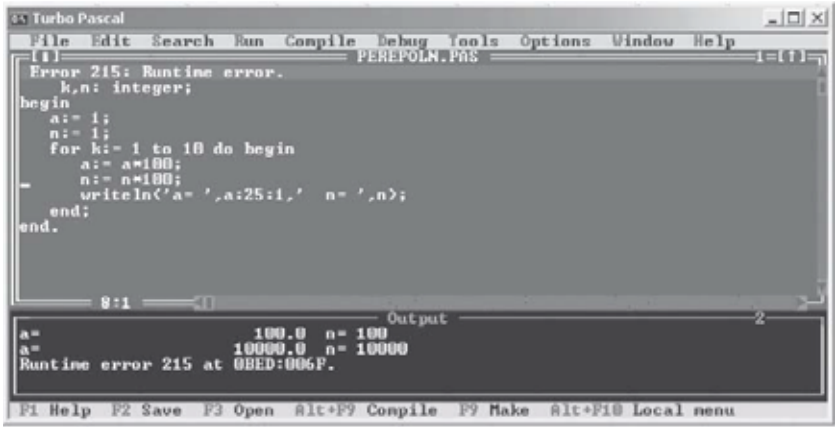


Рис. 9.10. Целочисленное переполнение при включенном контроле

Конечный результат может иметь нормальный вид, но промежуточные могут оказаться слишком большими или слишком маленькими. Естественно, о правильности конечного результата в данном случае говорить не приходится. Для машинной арифметики порядок выполнения операций может оказаться весьма существенным. Это в математике `a*b/c=a/c*b`. В программировании — не всегда. Пример см. на рис. 9.11.

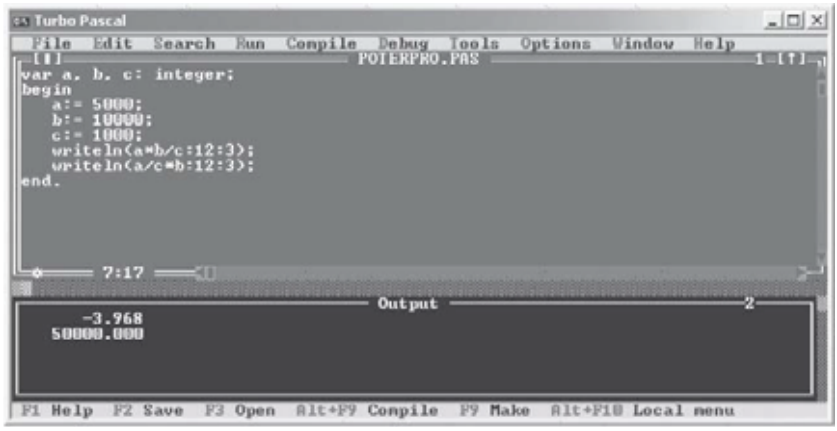


Рис. 9.11. Переполнение при промежуточных вычислениях

9.3. Передача управления

1. Развилки.

Опасность: Пропущена ветвь «иначе».

Этот момент особенно важен для оператора выбора. Что будет делать ваша программа, если значение выражения после `case` не соответствует ни одному из перечисленных вариантов?

2. Вложенные условные операторы.

Опасность: Сочетание `if-then-if-then-else`.

Как понимать запись: `if C1 then if C2 then S1 else S2`? Возможны два варианта:

1) `if C1 then (if C2 then S1 else S2)`
2) `if C1 then (if C2 then S1) else S2`

или в структурной записи:

```
1) if C1 then
       if C2 then S1
             else S2
2) if C1 then
       if C2 then S1
   else S2
```

В первом случае внешний оператор является сокращенным: `if C1 then`, а вложенный — полным: `if C2 then S1 else S2`.

Во втором случае внешний оператор является полным: `if C1 then … else S2`, а вложенный — сокращенным: `if C2 then S1`.

Как будет трактовать эту запись ваш язык программирования? Паскаль трактует ее первым способом: приписывает `else` к ближайшему предшествующему `if`.

Отметим, что в данном случае плохую услугу может оказать неправильная структурная запись. Для Паскаля абзацные отступы и пробелы роли не играют, для человека — играют. Запись

```
if C1 then
   if C2 then S1
else S2
```

у человека создает впечатление, что ветвь `else S2` относится к оператору `if C1 then`. Но Паскаль-то отнесет ее к оператору `if C2 then S1`!

Дополнительного внимания требует корректировка программы. Пусть изначально был записан абсолютно правильный текст:

```
if C1 then
    if C2 then S1
    else S3
else S2
```

Потом в процессе корректировки программы ветвь `else S3` была исключена. Получилось:

```
if C1 then
    if C2 then S1
else S2
```

При этом Паскаль автоматически переставит ветвь `else S2` из оператора `if C1 then` в оператор `if C2 then S1`.

Чтобы избежать подобных неприятностей, не следует после `then` писать сокращенных условных операторов. Если это необходимо, то либо такие операторы должны быть преобразованы в полные, приписыванием пустой части «иначе»:

```
if C1 then
   if C2 then S1
   else {not C2 – ничего}
else S2
```

либо сокращенный условный оператор должен быть заключен в операторные скобки `begin—end`:

```
if C1 then begin
   if C2 then S1
   end
else S2
```

3. Циклы.

Опасность: Зацикливание.

Убедитесь, что, в конце концов, каждый из циклов будет завершен. Если это цикл с пред- или постусловием, то при выполнении тела цикла должна меняться хотя бы одна из переменных, входящих в условие. Соблюдение этого требования не гарантирует выхода из цикла, но без него зацикливание неизбежно. Если это цикл со счетчиком, не меняйте счетчик цикла «вручную» при вычислении тела цикла, даже если язык программирования и позволяет это сделать.

Обратите внимание на то, что Турбо Паскаль позволяет «вручную» менять счетчик цикла в теле цикла. В результате появляется возможность зацикливания для циклов, в которых границы заданы константами. Например, так:

```
for k:= 1 to 5 do k:= 2;
```

Остановить зациклившуюся программу в Турбо Паскале можно нажатием комбинации клавиш **Ctrl+Break**.

9.4. Подпрограммы

1. Вызов подпрограммы.

Опасность 1: Неверное количество параметров.

Убедитесь, что количество фактических параметров соответствует количеству формальных. Неприятности могут возникнуть в случае раздельной трансляции.

Опасность 2: Неверные типы параметров.

Убедитесь, что типы фактических параметров согласуются с типами соответствующих формальных параметров. Неприятности могут возникнуть в случае раздельной трансляции.

Опасность 3: Неверный порядок следования параметров.

Почти во всех языках программирования соответствие между фактическими и формальными параметрами устанавливается по номеру параметра (первый фактический соответствует первому формальному, второй фактический — второму формальному и т. д.). Убедитесь, что порядок следования фактических параметров соответствует порядку следования формальных. Повышенного внимания требует ситуация, когда подряд идут несколько параметров одного типа. В этом случае вам не сможет помочь типовый контроль.

2. Использование параметров внутри подпрограммы.

Опасность: Неверные единицы измерения параметров.

Убедитесь, что во всех программных модулях для одного и того же параметра используется одна и та же единица измерения (углы могут измеряться в градусах, а могут в радианах, размеры — в сантиметрах и дюймах и т. д.). Сложность в том, что почти во всех языках программирования единица измерения не записывается явно в тексте программы, а подразумевается.

3. Использование формального параметра внутри подпрограммы для изменения значения соответствующего фактического параметра.

Опасность: Неверный способ передачи параметров.

В различных языках программирования используется около десятка способов передачи параметров в процедуру. Однако большинство языков ограничивается одним-двумя способами. Самые ходовые — это передача параметров по значению и по ссылке (в Паскале это называется «параметры-константы» и «параметры-переменные» соответственно). Формальный параметр, передаваемый по значению, представляет собой локальную переменную, которая получает начальное значение из фактического параметра. После этого всякая связь между формальным и фактическим параметрами разрывается. Формальный параметр может быть изменен внутри процедуры (с этой точки зрения паскалевский термин «параметр-константа» неудачен), но это изменение никак не повлияет на соответствующий фактический параметр. Если формальный параметр передается по ссылке, соответствующий фактический параметр обязательно должен быть переменной. Формальный параметр в этом случае представляет собой еще одно имя этой переменной. Все действия, задаваемые для формального параметра, на самом деле выполняются с соответствующим фактическим параметром. В частности, все изменения формального параметра являются изменениями параметра фактического.

Если вы хотите, чтобы изменения, произведенные в процедуре с формальным параметром, отразились вне процедуры на фактическом параметре, такой параметр надо передавать по ссылке. Наоборот, если изменения формального параметра не должны выйти наружу, такой параметр надо передавать по значению.

Еще одна опасность связана с тем, что некоторые языки программирования позволяют передавать по ссылке константы. Вообще говоря, изменение такого формального параметра должно приводить к изменению соответствующей константы. Следующая программа должна напечатать число 8!

```
procedure p(var x: integer);
   begin x:= x + 1; end; {p}
begin p(7); write(7) end.
```

К счастью, в большинстве языков (в том числе, в Паскале) такое запрещено.

4. Использование внутри подпрограммы формальных параметров, передаваемых по ссылке.

Опасность: Одна и та же переменная может быть указана как фактический параметр одновременно для нескольких формальных параметров, передаваемых по ссылке.

Поскольку формальный параметр, передаваемый по ссылке, — это просто другое имя для переменной, являющейся фактическим параметром, возникает ситуация, когда к одной и той же переменной процедура будет обращаться сразу по нескольким именам. Все изменения, выполняемые с одним из формальных параметров, будут реально выполняться сразу со всеми.

Само по себе это не ошибка, но требует повышенного внимания.

В качестве примера рассмотрим следующую ситуацию. Пусть у нас есть процедура `mulmatr(a,b,c)` с тремя параметрами-матрицами, которая занимается умножением своего первого параметра на свой второй параметр. Результат заносится в третий параметр. Пусть из соображений экономии места и времени все три параметра передаются по ссылке. И пусть, наконец, мы решили воспользоваться этой процедурой для возведения в квадрат некоторой матрицы `X`. Старое ее значение нам больше не нужно, поэтому новую матрицу записываем прямо поверх старой. Если бы `X` было числом, мы написали бы оператор присваивания `X:=X*X`. В случае матриц естественным кажется вызов процедуры `mulmatr`, в котором всем трем параметрам соответствует матрица `X`:

```
mulmatr(X, X, X).
```

Представьте себе, как будет выполняться умножение. Сначала через первую строку параметра `a` и первый столбец параметра `b` будет посчитан элемент c_{11}. Затем через первую строку параметра `a` и второй столбец параметра `b` начнется подсчет элемента c_{12}. Но дело в том, что элемент c_{11} является в то же время и элементом a_{11}, и элементом b_{11}. То есть, изменяя результирующую матрицу `C`, мы изменим и матрицы-сомножители. Понятно, что остальные элементы матрицы `C` будут подсчитаны неверно.

5. Использование внутри подпрограммы глобальной переменной и параметра, передаваемого по ссылке.

Опасность: Одновременный доступ к одной и той же переменной через параметр, передаваемый по ссылке, и через глобал.

Пусть некоторая переменная из объемлющего блока доступна внутри процедуры как глобальная. Пусть она изменяется

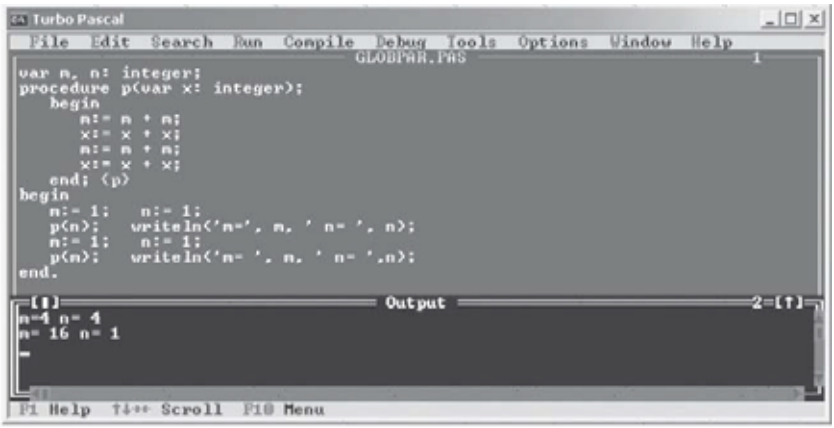


Рис. 9.12. Доступ к одной и той же переменной через параметр, передаваемый по ссылке, и как к глобальной переменной

внутри этой процедуры. Пусть у этой процедуры есть параметр, передаваемый по ссылке. Пусть при вызове данной процедуры в качестве фактического параметра для передачи по ссылке указана та самая переменная, которая является глобальной. Тогда возникает ситуация, когда к одной и той же переменной мы обращаемся одновременно по двум разным именам: по имени глобальной переменной и по имени формального параметра. Пример см. на рис. 9.12.

После первого вызова процедуры `p` обе переменные получат значение 4. Внутри процедуры переменная `m` была доступна как глобал, а переменная `n` как параметр, передаваемый по ссылке. После второго вызова переменная `n` сохранит значение 1 (ее вызов процедуры вообще не затрагивает), переменная `m` станет равна 16. Внутри процедуры `m` и `x` — это два имени одного и того же объекта. Любое изменение `m` является в то же время изменением `x`, а любое изменение `x` — изменением `m`.

Само по себе такое «двухименное» обращение к переменной не ошибка, но требует повышенного внимания.

9.5. Файлы

1. Запись/чтение двоичных файлов.

Опасность: Путаница между файлами разных типов.

Паскалевские файлы (и файлы во многих других языках) могут быть текстовые (специфицированы стандартным типом

TEXT) и двоичные (описаны как file of T). Каждой файловой переменной должен быть поставлен в соответствие набор данных (файл) на диске. Обратите внимание на то, что в файле на диске не сохраняется информация о том, какого типа данные в нем записаны. Поэтому вы можете сначала поставить в соответствие этому файлу файловую переменную типа file of real, записать в него несколько вещественных чисел, закрыть, а потом опять открыть, но уже поставив ему в соответствие файловую переменную типа file of integer. С этим файлом можно будет выполнять любые операции, но его прежнее содержимое — двоичный код нескольких вещественных чисел — будет рассматриваться как код целых чисел. Программа, демонстрирующая этот эффект, приведена ниже.

```
var freal: file of real;
    fint:  file of integer;
    r: real;
    k, i: integer;
begin
   assign(freal,'f.dat');
   rewrite(freal);
   r:= 3.1415926;
   write(freal,r)
   reset(freal);
   while not eof(freal) do begin
      read(freal,r);
      writeln(r:12:8)
   end;
   close(freal);
   assign(fint,'f.dat');
   reset(fint);
   while not eof(fint) do begin
      read(fint,i);
      writeln(i)
   end;
end.
```

Результат ее работы будет выглядеть следующим образом:

```
  3.14159260
-26750
-9624
18703
```

2. Создание файлов с данными с помощью текстового редактора. Просмотр файлов с данными с помощью текстового редактора.

Опасность: Путаница текстовых и двоичных файлов.

Файл, создаваемый с помощью текстового редактора (например, в среде Турбо Паскаль), будет файлом текстовым. В Паскале он соответствует стандартному типу `TEXT`. Даже если вы записали туда последовательность целых чисел, открывать его как `file of integer` не следует. В текстовом файле будет храниться последовательность литер, с помощью которых вы изобразили целые числа, а в `file of integer` должно храниться их внутреннее представление, соответствующий двоичный код.

И наоборот. Если файл записан из программы как двоичный, корректный просмотр его с помощью текстового редактора невозможен. В лучшем случае вы увидите некоторый набор самых разных символов (отнюдь не цифр). В худшем вы или сам текстовый редактор что-нибудь в нем измените, после чего прежнее содержимое файла окажется испорченным.

3. Запись/чтение нетипизированных файлов.

Опасность: Еще одна возможность обойти контроль типов. Данные могут быть записаны в файл как значения одного типа, а прочитаны как значения другого.

4. Запись в файл.

Опасность: Отсутствие явного закрытия файлов.

Если вы не закрыли явно файл перед окончанием работы программы, есть шанс, что содержимое последнего заполненного буфера не будет вытолкнуто на диск. Это значит, что результаты последних операций записи могут в файл не попасть.

Не возникло ли у читателя желания еще раз вернуться к задаче о проверке программы для анализа треугольников? С учетом понимания тех ошибок, которые могут быть допущены при составлении этой программы.

Глава 10

Безмашинное тестирование

Для большинства начинающих программистов тестирование подразумевает непременный прогон программы на компьютере. И напрасно. На большинство людей экран терминала оказывает гипнотическое воздействие. Человек, сидящий перед монитором, соображает с трудом. А при тестировании и отладке соображать совершенно необходимо!

Особенно важно безмашинное тестирование для новичков. Только ручная прокрутка программы позволит вам понять, как же работает ваша программа.

Но разве можно вести тестирование без помощи компьютера? Можно. И весьма плодотворно. Как показывает опыт, безмашинное тестирование обнаруживает от 30 до 80% ошибок в профессиональных программах и до 100% в учебных. Существуют следующие варианты безмашинного тестирования.

1. «Сухая прокрутка».

Вы пытаетесь вручную смоделировать работу машины при выполнении программы. Вам при этом достается роль центрального процессора. А роль оперативной памяти будет играть трассировочная таблица. В этой таблице будет своя графа для каждой переменной, а порядок присваивания значений будет отображаться выравниванием значений по вертикали. Например, для следующей программы

```
program P;
var k, n: integer;
    sum: integer;
begin
  writeln('Я считаю сумму квадратов чисел от 1 до 5');
  n:= 5;
  sum:= 0;
  for k:= 1 to n do
     sum:= sum + k*k;
  writeln('Сумма квадратов чисел от 1 до ',n,' = ',sum)
end. {p}
```

трассировочная таблица будет выглядеть так:

n	*sum*	*k*
5		
	0	
		1
	1	
		2
	5	
		3
	14	
		4
	30	
		5
	55	

Замечание. Сдвиг по вертикали в разных графах трассировочной таблицы удобней делать на полстроки.

2. «Символическая прокрутка».

На этот раз вам не надо подставлять конкретные значения и вести трассировку переменных. Вместо этого необходимо шаг за шагом анализировать ход программы. Каждый раз, когда переменная получает значение, вы должны убедиться, что это значение соответствует смыслу переменной (для этого, как минимум, надо точно знать смысл каждой переменной), найти те условия, при которых вычисления дадут сбой (см. главу 9 «Ошибкоопасные ситуации»), убедиться, что в программе отражены все возможные особые случаи.

3. Объяснение коллеге.

Расчет в данном случае делается не на то, что коллега много умнее вас, а на то, что объяснение другому человеку отличается от рассуждений про себя. В ваших рассуждениях о программе обязательно присутствуют некоторые допущения и предположения. Когда вы рассуждаете о программе сами с собой, многие из них остаются не проговоренными и явно не сформулированными. Когда вы то же самое начинаете вслух объяснять постороннему человеку, вы вынуждены явно сформулиро-

вать все допущения. И очень может быть, что в процессе этой формулировки сами и обнаружите неточность[2].

4. Передача программы коллеге для самостоятельного изучения.

В данном случае опять расчет не на то, что коллега много умнее вас. Мы уже упоминали особенность человеческой психики, которая заключается в том, что человеческий глаз видит не то, что есть на самом деле, а то, что человек хочет. Соответственно возможно, что при изучении своей программы вы видите не то, что она делает на самом деле, а то, что вам хотелось бы, чтобы она делала. Есть надежда, что ваш коллега подобным пристрастиям будет не подвержен и увидит в тексте программы то, что там на самом деле написано.

5. Просмотр текста программы с анализом ошибкоопасных мест.

Об этом речь шла в предыдущей главе.

6. Искусственное внесение ошибок в программу.

Естественно, вносить ошибки должен посторонний человек (или программа), и место ошибок должно держаться в секрете от человека, занимающегося тестированием.

Очень мощный прием. Прежде всего, с психологической точки зрения. Мы уже говорили, что многие программисты склонны отождествлять себя со своей программой и поэтому не склонны выискивать в этой программе огрехи. После внесения в программу искусственных ошибок этот психологический стопор снимается. Ошибки-то не мои! Более того, настроение меняется на прямо противоположное. У меня была такая прекрасная программа, а эти злодеи ее испортили своими ошибками! Ну, я им покажу, как портить мои программы!

[2] Вообще говоря, с этой точки зрения примерно все равно, кому объяснять программу: соседу по комнате или кошке.

Глава 11

Пример тестирования несложной программы

Проиллюстрируем все вышесказанное достаточно простым примером. Пусть требуется написать программу, которая считает среднее арифметическое последовательности целых чисел, заканчивающейся нулем.

Для начала отметим, чего делать ни в коем случае не следует. Не надо, еще не дослушав задание, бросаться к монитору компьютера и быстро-быстро набирать что-нибудь вроде:

```
Program p;
var a, b, c: integer:
begin
   repeat
      read(a);
      b:= b + 1;
      c:= c + a
   until a=0;
   writeln(c/b)
end.
```

Сначала всегда полезно подумать.

Согласно принципу 6, перед началом тестирования надо сформулировать цели тестирования, выбрать критерии полноты. Зафиксируем в качестве цели: провести тестирование, полное с точки зрения критериев черного ящика и критерия МГТ.

Согласно принципу 5, разработка тестов должна предшествовать написанию программы и первоначально должна быть направлена на уяснение поставленной задачи. Начнем, естественно, с критериев черного ящика (критерии белого ящика пока неприменимы за отсутствием текста программы).

Первый критерий — тестирование функций — нам в данном случае неинтересен, поскольку программа у нас однофункциональная. Эту единственную функцию мы и будем постоянно тестировать.

Далее на очереди тестирование классов входных и выходных данных, областей допустимых значений, длины последовательности, упорядоченности набора данных. Выделение классов входных и выходных данных в нашей задаче не очевидно. Об-

ластей допустимых значений — пока тоже. Упорядоченность нам не важна. А вот проверка длины последовательности имеет к нам непосредственное отношение.

Пойдем по порядку.

1. Пустой набор (не содержит ни одного элемента).

Входные данные? Стоп. Вопрос на уточнение: как ввести пустую последовательность? Вы можете сами придумать ответ на этот вопрос. Но вы не должны этого делать! Ваша цель не додумывать самому непонятные места, а понять, чего хочет от вас заказчик. Поэтому этот вопрос следует задать постановщику задачи.

Ответ: пустая последовательность — это последовательность, состоящая из одного нуля. Нуль элементом последовательности не является. Количество элементов в такой последовательности считается равным нулю.

С входными данными теперь ясно: 0.

Выходные данные? Опять стоп. Что должна выдавать программа в ответ на пустую последовательность? Опять же, вы можете сами придумать ответ на этот вопрос. Но вы не должны этого делать! Это опять вопрос постановщику задачи. Ответ: в случае пустой последовательности результатом должна быть фраза «Последовательность пуста».

Вот теперь мы можем сформулировать первый тест.

Вход: 0.

Ожидаемый выход: Последовательность пуста.

2. Единичный набор (состоит из одного-единственного элемента).

Вход: 4, 0.

Ожидаемый выход: 4.

3. Слишком короткий набор.

Для нас такого понятия нет.

4. Набор минимально возможной длины.

Для нас такого понятия нет.

5. Нормальный набор (состоит из нескольких элементов).

Для определенности возьмем набор из 3 элементов.

Вход: 1, 3, 4, 0.

Выход: Стоп!

В математике $8/3 = 2,(6)$. Оказывается, среднее арифметическое последовательности целых чисел может быть числом вещественным. На всякий случай уточним у заказчика, должны

ли мы выдавать вещественный результат или должны преобразовывать его к целому. Если преобразовывать, то как: округлять или обрубать? Если вещественный, то с какой точностью? Повторим, что вы можете сами придумать ответы на все эти вопросы, но не должны делать это. Пусть отвечает заказчик. Его ответ: среднее арифметическое выдается как вещественное число с той точностью, которую обеспечит ваш компьютер. Что ж. Ответ для нас весьма удобный. Для определенности, пусть будет 6 знаков после запятой. Итак:

Вход: 1, 3, 4, 0.

Ожидаемый выход: 2.666667.

6. Набор из нескольких частей (если такое возможно).

В нашем случае это невозможно.

7. Набор максимально возможной длины (если такая предусмотрена).

Не предусмотрена.

8. Слишком длинный набор (с длиной больше максимально допустимой).

И это для нас невозможно.

Итак, что мы получили? Во-первых, мы построили 3 теста. Во-вторых, мы уточнили несколько важных деталей (что такое пустая последовательность, как на нее реагировать, какого типа должен быть результат, с какой точностью он должен выдаваться). Наши тесты покрывают классы выходных данных («Последовательность пуста» и число, являющееся средним арифметическим) и классы входных (3 вида последовательностей, различающихся длиной).

Запишем 3 наших теста в таблицу следующего формата:

Тест	Вход	Ожидаемый результат	Результат «сухой прокрутки»	+/—	Результат выполнения на компьютере	+/—
T1	0	Последовательность пуста				
T2	4, 0	4				
T3	1, 3, 4, 0	2.666667				

Пока что мы заполним в ней только первые строки и в каждой из них — первые графы. Строки нам, возможно, придется

добавить, исходя из критериев белого ящика и анализа ошибкоопасных мест. Две средние графы заполним по результатам безмашинного тестирования, две последние — по результатам выполнения на компьютере. В графах «+/−» будем отмечать совпадение или несовпадение реального результата с ожидаемым.

Теперь можно переходить к составлению текста программы. Идея проста. Среднее арифметическое — это частное от деления суммы элементов на их количество. Значит надо ввести элементы, просуммировать их, посчитать их количество, после чего напечатать частное. Например, так:

Программа на псевдокоде	Примечание

begin
  выдать начальное приветствие;
  ввести 1-й элемент (tek);
  while последовательность не кончена do begin
  увеличить сумму (sum);
  увеличить количество (kol)
  end;
  вывести среднее арифметическое (sum/kol)
end.
	var tek: integer;
	
	var sum: integer;
	var kol: integer;

Упомянутые в проекте фрагменты раскроем следующим образом:

```
Ввести 1-й элемент
  writeln('Введите, пожалуйста, 1-й элемент');
  read(tek);

Последовательность не кончена
  tek<>0

Увеличить сумму
  sum:= sum + tek;

Увеличить количество
  kol:= kol + 1;
```

Первый вариант программы будет выглядеть так:

```
program SrednArifm;
var tek, kol, sum: integer;
begin
  writeln('Я считаю среднее арифметическое' +
    'последовательности чисел, кончающейся нулем');
```

```
   writeln('Введите, пожалуйста, 1-й элемент');
   read(tek);
   while tek<>0 {последовательность не кончена} do begin
      sum:= sum + tek;  {увеличить сумму элементов}
      kol:= kol + 1;    {увеличить кол-во элементов}
   end;
   if kol=0 then writeln('Последовательность пуста')
         else writeln('Среднее арифметическое=',sum/kol)
end.
```

Заметим, что худо-бедно, но мы позаботились не только о функциональности программы, но и об удобстве пользователя. Это, конечно, не windows-интерфейс, но, во всяком случае, пользователю не придется сидеть перед пустым черным экраном и гадать, что же за программу он запустил и что ему делать дальше.

Кроме того, первым пользователем любой программы является ее разработчик. Значит, надо позаботиться о его удобстве. Здесь существует некоторое противоречие. Разработчику приходится выполнять с текстом программы два противоположных действия: писать и читать. Для удобства записи текст должен быть кратким, для удобства чтения — подробным. Что выбрать? Текст записывается только один раз, а читается много. Значит, решение надо принимать в пользу чтения. Для упрощения чтения могут использоваться осмысленные имена переменных, абзацные отступы (запись лесенкой), комментарии.

Продолжим составление тестов. На этот раз с использованием критериев белого ящика, конкретней — МГТ. Строим таблицу минимально грубого тестирования. В нашей программе всего две развилки: цикл с предусловием и ветвление. В обоих случаях используются простые условия. Значит, МГТ-таблица будет содержать всего 5 строк:

			T1	T2	T3
У1	`while tek<>0`	=0			
		=1			
		>1			
У2	`if kol=0`	+			
		—			

Мы уже имеем 3 теста, построенные исходя из критериев черного ящика. Посмотрим, как они поведут себя на нашей программе и как отразятся в таблице МГТ. Для этого вручную выполним первый тест и заполним трассировочную таблицу. Вход первого теста состоит из единственного числа — нуль. Оно и будет введено как значение переменной `tek`. Занесем его в трассировочную таблицу:

tek	*kol*	*sum*
0		

Тело цикла `while` не выполняется ни разу. Сразу переходим на условный оператор. Равна ли переменная `kol` нулю? Моделью оперативной памяти для нас является трассировочная таблица. Там и ищем ответ. Но... В трассировочной таблице для переменной `kol` не указано вообще никакого значения! Мы забыли ее инициализировать! `kol` — это количество элементов последовательности. Значит, изначально, пока никакой последовательности еще нет, `kol` должна быть равна нулю. Вставим соответствующий оператор перед вводом первого элемента последовательности. Тело программы приобретет вид:

```
begin
   writeln('Я считаю …');
   kol:= 0;
   writeln('Введите, пожалуйста, 1-й элемент');
   read(tek);
   while tek<>0 do begin
      sum:= sum + tek;
      kol:= kol + 1;
   end;
   if kol=0 then writeln('Последовательность пуста')
         else writeln('Среднее арифметическое = ', sum/kol)
end.
```

Повторим первый тест. Теперь его удастся выполнить до конца. Трассировочная таблица будет иметь вид:

tek	*kol*	*sum*
	0	
0		

В таблице тестов отметим результат выполнения первого теста:

Тест	Вход	Ожидаемый результат	Результат «сухой прокрутки»	+/–	Результат выполнения на компьютере	+/–
Т1	0	Последовательность пуста	Последовательность пуста	+		
Т2	4, 0	4				
Т3	1, 3, 4, 0	2.666667				

В таблице МГТ отметим проверенные строки:

			Т1	Т2	Т3
У1	`while tek<>0`	=0	+		
		=1			
		>1			
У2	`if kol=0`	+	+		
		–			

Перейдем ко второму тесту. Выполняем программу и заполняем трассировочную таблицу.

tek	*kol*	*sum*
	0	
4		

После входа в цикл спотыкаемся на том же, на чем уже споткнулись в предыдущий раз. Переменная `sum` не имеет начального значения. Ее, так же как и переменную `kol`, необходимо инициализировать нулем перед входом в цикл. Получаем следующий текст тела программы:

```
begin
   writeln('Я считаю …');
   kol:= 0;
   sum:= 0;
   writeln('Введите, пожалуйста, 1-й элемент');
   read(tek);
   while tek<>0 do begin
      sum:= sum + tek;
      kol:= kol + 1;
   end;
   if kol=0 then writeln('Последовательность пуста')
            else writeln('Среднее арифметическое=',sum/kol)
end.
```

Теперь выполнение второго теста пойдет веселее. По крайней мере, поначалу.

tek	*kol*	*sum*
	0	
		0
4		
		4
	1	
		8
	2	
		12
	3	

Уже можно остановиться. Понятно, что что-то не так. Что именно? Воспользуемся перечнем ошибкоопасных ситуаций. У нас проблемы с циклом, произошло именно то, о чем нас и предупреждали: зацикливание. Чтобы его избежать, в теле цикла, как минимум, должна меняться хотя бы одна переменная, входящая в условие цикла. У нас такая переменная всего одна, и она в теле цикла не меняется! Мы забыли ввести остальные элементы последовательности! Исправим ошибку. Теперь тело программы приобретет вид:

```
begin
   writeln('Я считаю …');
   kol:= 0;
   sum:= 0;
   writeln('Введите, пожалуйста, 1-й элемент');
   read(tek);
   while tek<>0 do begin
      sum:= sum + tek;
      kol:= kol + 1;
      writeln('Введите, пожалуйста, элемент № ',kol);
      read(tek);
   end;
   if kol=0 then writeln('Последовательность пуста')
      else writeln('Среднее арифметическое = ',sum/kol)
end.
```

Заметим, что мы опять в меру сил позаботились об удобстве пользователя. Во-первых, он получит приглашение к вводу очередного элемента, во-вторых, это приглашение будет содержать номер элемента.

Выполняем еще раз второй тест и наконец-то доходим до конца.

tek	*kol*	*sum*
	0	
		0
4		
		4
	1	
0		

По ходу выполнения выясняем, что мы допустили еще одну маленькую ошибку. Правда, не в вычислениях, а в интерфейсе. Программа второй раз попросила нас ввести элемент № 1. Приглашение в теле цикла должно выглядеть так:

```
writeln('Введите, пожалуйста, элемент № ',kol+1);
```

Внесем в текст соответствующие изменения, на результатах выполнения это не скажется.

Сделаем отметку в таблице тестов.

Тест	**Вход**	**Ожидаемый результат**	**Результат «сухой прокрутки»**	**+/–**	**Результат выполнения на компьютере**	**+/–**
Т1	0	Последовательность пуста	Последовательность пуста	+		
Т2	4, 0	4	4	+		
Т3	1, 3, 4, 0	2.666667				

Заполним соответствующий столбец в таблице МГТ.

			Т1	**Т2**	**Т3**
У1	`while tek<>0`	=0	+		
		=1		+	
		>1			
У2	`if kol=0`	+	+		
		–		+	

С тестом 2 вроде бы все. Но поскольку мы внесли в программу исправления, необходимо повторить все ранее прогнанные тесты. Нам это просто, поскольку такой тест был всего один.

При его повторе результаты получим прежние, хотя трассировочная таблица несколько изменится:

tek	*kol*	*sum*
	0	
		0
0		

Добрались до теста 3. Строим трассировку.

tek	*kol*	*sum*
	0	
		0
1		
		1
	1	
3		
		4
	2	
4		
		8
	3	
0		

Заполняем таблицы тестов и МГТ.

Тест	**Вход**	**Ожидаемый результат**	**Результат «сухой прокрутки»**	**+/–**	**Результат выполнения на компьютере**	**+/–**
T1	0	Последовательность пуста	Последовательность пуста	+		
T2	4, 0	4	4	+		
T3	1, 3, 4, 0	2.666667	2.666667	+		

			T1	**T2**	**T3**
У1	`while tek<>0`	=0	+		
		=1		+	
		>1			+
У2	`if kol=0`	+	+		
		–		+	+

Третий тест оказался неудачен — никакой ошибки не выявил. Зато оказалось, что все строки таблицы МГТ уже заполнены. Значит, мы построили и прогнали набор тестов, полный и с точки зрения критериев черного ящика, и с точки зрения минимально грубого тестирования. Значит, ручное тестирование пора заканчивать.

Теперь можно выходить на компьютер и прогнать еще раз те же тесты уже через него. Никаких неожиданностей мы при этом не встретим и сможем заполнить крайние правые графы в таблице тестов:

Тест	Вход	Ожидаемый результат	Результат «сухой прокрутки»	+/–	Результат выполнения на компьютере	+/–
T1	0	Последовательность пуста	Последовательность пуста	+	Последовательность пуста	+
T2	4, 0	4	4	+	4	+
T3	1, 3, 4, 0	2.666667	2.666667	+	2.666667	+

Перед окончанием данной главы сделаем 3 замечания:

1. Тестов, построенных исходя из критериев черного ящика, оказалось достаточно, чтобы удовлетворить критерию МГТ.
2. При этом тесты эти (построенные в тот момент, когда текста программы еще не существовало) оказались настолько хороши, что первые 2 из них обнаружили в нашей программе 3 ошибки. И только последний третий тест оказался неудачен, ошибок не выявил.
3. Легко заметить, что все допущенные нами ошибки упоминались в перечне ошибкоопасных ситуаций. Значит, если бы мы вместо того, чтобы сразу же начинать тестовые прогоны, сначала проанализировали нашу программу с точки зрения этого перечня, ошибки удалось бы обнаружить гораздо быстрей и с меньшими усилиями. Это урок на будущее!

Глава 12

Порядок работы над программой

Из примера, разобранного в предыдущей главе, виден порядок работы над программой.

1. Начинаем с критериев черного ящика. Соответствующие тесты составляем *до* написания текста программы. Тесты заносим в таблицу тестов, в которой кроме входных данных и ожидаемых результатов предусмотрена графа для реальных результатов и отметки о совпадении их с ожидаемыми.
2. Пишем текст программы.
3. Проверяем его, исходя из списка ошибкоопасных конструкций и ситуаций.
4. Составляем таблицу МГТ.
5. Прогоняем тесты, составленные исходя из критериев черного ящика. В процессе прогона строим трассировочные таблицы.
6. При необходимости корректируем программу. Проверяем места корректировок на ошибкоопасность.
7. После корректировки программы проводим повторное тестирование: повторяем заново все ранее прогнанные тесты.
8. Заполняем последние графы таблицы тестов и таблицу МГТ.
9. Если таблица МГТ еще неполна, добавляем тесты для покрытия незаполненных строк таблицы МГТ. Заносим их в таблицу тестов.
10. Прогоняем их через программу. Строим трассировки.
11. При необходимости корректируем программу. Проверяем места корректировок на ошибкоопасность.
12. Заполняем последние графы таблицы тестов и таблицу МГТ.
13. После корректировки программы проводим повторное тестирование: повторяем заново все ранее прогнанные тесты.

Глава 13

Нисходящее тестирование

Программирование методом пошаговой детализации (программирование сверху вниз) подразумевает написание программы на псевдокоде и постепенный перевод ее на язык программирования. При этом программа может достаточно долго включать в себя фрагменты на псевдокоде. С точки зрения системы программирования, такую программу исполнять нельзя. Можно ли такую программу тестировать? Да, можно. Прежде всего, можно (и нужно) использовать безмашинные методы, о которых шла речь выше. Но кроме того, можно организовать выполнение такой программы на компьютере. Для этого используются так называемые *заглушки*[3]. Заглушка представляет собой имитатор еще не разработанной части программы. Она может быть подставлена вместо подпрограммы или фрагмента, написанного на псевдокоде. В качестве заглушек могут использоваться:

— пустые операторы;
— операторы печати;
— упрощенные варианты имитируемых частей (например, вместо выполнения сложных вычислений заглушка процедуры может присваивать выходным параметрам некоторые стандартные значения).

Пустой оператор в качестве заглушки требует минимальных усилий, но и пользу приносит минимальную. Заглушка в виде оператора печати усилий требует ненамного больших, а для слежения за работой программы гораздо полезнее. В качестве печатаемого текста удобно выдавать название имитируемого фрагмента. Если имитируется работа подпрограммы, полезно распечатать значения ее параметров. Надо позаботиться, чтобы операторы заглушки по внешнему виду отличались от «нормальных» операторов программы, а сообщения, выдаваемые заглушкой, — от «нормальных» сообщений программы. Например, операторы заглушки будем помечать специальным комментарием, включающим в себя два диеза (##). Любое сообщение, выдаваемое заглушкой, будем начинать со специального маркера: двух диезов (##).

---

[3] В английском языке используется термин «stub» — обрубок.

В качестве примера использования заглушек приведем следующий текст:

```
procedure Obrabotka(m, n: integer; var c: char; var x: real);
begin
  {##} c:= '+';   x:= 0.5;  {константы вместо вычислений}
  {##} writeln('##Obrabotka. m=',m,' n=',n,' c=',c,' x=',x);
end; {Obrabotka}
begin
   writeln('## Инициализация данных');
   writeln('## Ввод данных');
   Obrabotka(7, 4, s, z);
   Obrabotka(14, -3, d, y);
   writeln('## Завершение');
end.
```

При выполнении такой программы каждый из выполняемых фрагментов сообщит о том, что он проработал. Процедура `Obrabotka`, кроме того, присвоит выходным параметрам некоторые стандартные значения и распечатает значения входных и выходных параметров.

Нисходящее тестирование обладает рядом достоинств и недостатков.

Достоинства нисходящего тестирования:

1) возможность раннего начала тестирования;
2) ранняя проверка сопряжений между модулями;
3) более тщательная проверка модулей верхнего уровня.

Недостатки нисходящего тестирования:

1) трудно тестировать модули нижнего уровня. На выполнение подается программа целиком. Вход/выход для модуля нижнего уровня опосредован работой модулей верхнего уровня. Приходится таким образом подбирать значения вводимых в программу переменных, чтобы тестируемая процедура была вызвана именно с теми значениями параметров, которые нужны для тестирования;
2) необходимо тратить силы на написание заглушек;
3) заглушка — это все-таки имитатор, упрощенный вариант. Поэтому тестирование на заглушках оказывается неполным. После замены их реальным программным текстом тестирование приходится повторять;
4) заглушки можно подставить только вместо процедур и операторов. Нет заглушек для переменных и типов.

Первый недостаток можно преодолеть, если отказаться от строго нисходящего порядка работы. Для отладки модулей нижнего уровня пишут специальные модули верхнего уровня, так называемые *драйверы*[4]. Назначение этих модулей:

1) установка нужного для тестирования контекста;
2) загрузка исходных данных теста;
3) вызов тестируемых подпрограмм;
4) обработка результатов тестирования.

Для языков с независимой трансляцией подпрограмм драйвер оформляется в виде отдельной программной единицы. Для языков с блочной структурой дело обстоит сложнее. Для тестирования вложенных подпрограмм приходится менять тело главной программы и объемлющих блоков.

[4] Сейчас термин «драйвер» прочно ассоциируется с управлением внешними устройствами. Но исторически впервые он появился именно в тестировании.

Глава 14

*Оценка количества ошибок в программе[5]

Сколько ошибок в программе? Это вопрос, который волнует и каждого программиста, и каждого заказчика. Особую актуальность придает ему принцип «кучкования ошибок», согласно которому нахождение в некотором модуле ошибки увеличивает вероятность того, что в этом модуле есть и другие ошибки.

Точного ответа на вопрос о количестве ошибок в программе очень часто дать невозможно, а вот построить некоторую оценку — можно. Для этого существуют нескольких статистических моделей, которые мы и рассмотрим в данной главе.

Заметим, что даже приблизительное знание количества ошибок в программе может быть очень полезно для формулирования цели тестирования и составления планов тестирования.

14.1. Модель Миллса

В 1972 г. суперпрограммист фирмы IBM Харлан Миллс предложил следующий способ оценки количества ошибок в программе.

Пусть у нас есть программа. Предположим, что в ней N ошибок. Назовем их «естественными». Внесем в нее дополнительно M «искусственных» ошибок. Проведем тестирование программы. Пусть в ходе тестирования было обнаружено n естественных ошибок и m искусственных. Предположим, что вероятность обнаружения для естественных и искусственных ошибок одинакова. Тогда выполняется соотношение:

$$\frac{n}{N} = \frac{m}{M}.$$

Мы нашли один и тот же процент естественных и искусственных ошибок. Отсюда количество ошибок в программе $N = n \cdot \frac{M}{m}$. Количество необнаруженных ошибок равно $N - n$.

---

[5] **Глава отмечена звездочкой. Это означает, что некоторые ее части могут показаться недостаточно подготовленному читателю слишком сложными. Можете пропустить такие части (если это допускается вашей учебной программой). Чтение последующих глав не требует обязательного знания материала данного раздела.**

Например, пусть в программу было внесено 20 искусственных ошибок, в ходе тестирования было обнаружено 12 искусственных и 7 естественных ошибок. Получим следующую оценку количества ошибок в программе:

$$N = 7 \cdot \frac{20}{12} = \frac{70}{6} \approx 12.$$

Количество необнаруженных ошибок равно $N - n = 12 - 7 = 5$.

Сам Миллс предлагал по ходу тестирования постоянно строить график для оценки количества ошибок в программе.

Легко заметить, что в описанном выше способе Миллса есть один существенный недостаток. Если мы найдем 100% искусственных ошибок, это будет означать, что и естественных ошибок мы нашли 100%. Но чем меньше мы внесем искусственных ошибок, тем больше вероятность того, что мы найдем их все. Внесем единственную искусственную ошибку, найдем ее и на этом основании объявим, что нашли все естественные ошибки! Чтобы противостоять этому искусу, Миллс добавил вторую часть модели, предназначенную для проверки гипотезы о величине N. Выглядит она так.

Предположим, что в программе N естественных ошибок. Внесем в нее еще M искусственных ошибок. Будем тестировать программу до тех пор, пока не найдем все искусственные ошибки. Пусть к этому моменту найдено n естественных ошибок. На основании этих чисел вычислим величину

$$C = \begin{cases} 1 & \text{при } n > N, \\ \dfrac{M}{M + N + 1} & \text{при } n \le N. \end{cases}$$

Величина C выражает меру доверия к модели. Это вероятность того, что модель будет правильно отклонять ложное предположение. Например, пусть мы считаем, что естественных ошибок в программе нет ($N = 0$). Внесем в программу 4 искусственные ошибки. Будем тестировать программу, пока не обнаружим все искусственные ошибки. Пусть при этом мы не обнаружим ни одной естественной ошибки. В этом случае мера доверия нашему предположению (об отсутствии ошибок в программе) будет равна 80% (4 / (4 + 0 + 1)). Для того чтобы довести ее до 90%, количество искусственных ошибок придется поднять до 9. Следующие 5% уверенности в отсутствии естествен-

ных ошибок обойдутся нам в 10 дополнительных искусственных ошибок. M придется довести до 19.

Если мы предположим, что в программе не более 3 естественных ошибок ($N = 3$), внесем в нее 6 искусственных ($M = 6$), найдем все искусственные и одну, две или три (но не больше!) естественных, то мера доверия к модели будет 60% (6 / (6 + 3 + 1)).

Далее в таблице 1 приведены значения функции C для различных значений N и M от 0 до 20 с шагом 1, в таблице 2 — для N и M от 0 до 100 с шагом 5.

Модель допускает динамическую подстройку. Например, пусть в предыдущем примере была найдена только одна естественная ошибка. Тогда удобно понизить N с 3 до 1. В результате степень доверия к модели возрастет до 75%. С другой стороны, если в процессе тестирования было найдено 4 естественные ошибки, гипотезу об $N = 3$ придется заменить на гипотезу об $N = 4$. А это сразу же понизит меру доверия к модели с 60% до 55%.

Верхнюю строку в определении величины C можно интерпретировать следующим образом. Если мы предположили, что в программе не более N ошибок, а обнаружили их больше N, то с вероятностью 100% мы опровергнем гипотезу о том, что ошибок в программе не более N.

Из формулы для вычисления меры доверия легко получить формулу для вычисления количества искусственных ошибок, которые необходимо внести в программу для получения нужной уверенности в полученной оценке:

$$M = \frac{C(N + 1)}{1 - C}.$$

Далее в таблице 3 приведены значения функции M для различных значений N от 0 до 20 с шагом 1 и C от 0 до 99 с шагом 5 (C выражено в процентах), в таблице 4 — для N от 0 до 100 с шагом 5 и C от 0 до 99 с шагом 5. Поскольку при $C = 100\%$ функция M невычислима, последнее значение C берется равным 99%.

Обращают на себя внимание резкие скачки при переходе C от 90% к 95% и, особенно, от 95% к 99%.

Модель Миллса достаточно проста. Ее слабое место — предположение о равновероятности нахождения ошибок. Чтобы это предположение оправдалось, процедура внесения искусственных ошибок должна обладать определенной степенью «интеллекта».

Таблица 1. Мера доверия к миллсовой модели в зависимости от гипотезы о количестве естественных ошибок (*N*) и количества внесенных искусственных ошибок (*M*) для *N* и *M* от 0 до 20 с шагом 1, %

M	*N*																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	50	33	25	20	17	14	13	11	10	9	8	8	7	7	6	6	6	5	5	5	5
2	67	50	40	33	29	25	22	20	18	17	15	14	13	13	12	11	11	10	10	9	9
3	75	60	50	43	38	33	30	27	25	23	21	20	19	18	17	16	15	14	14	13	13
4	80	67	57	50	44	40	36	33	31	29	27	25	24	22	21	20	19	18	17	17	16
5	83	71	63	56	50	45	42	38	36	33	31	29	28	26	25	24	23	22	21	20	19
6	86	75	67	60	55	50	46	43	40	38	35	33	32	30	29	27	26	26	24	23	22
7	88	78	70	64	58	54	50	47	44	41	39	37	35	33	32	30	29	28	27	26	25
8	89	80	73	67	62	57	53	50	47	44	42	40	38	36	35	33	32	31	30	29	28
9	90	82	75	69	64	60	56	53	50	47	45	43	41	39	38	36	35	33	32	31	30
10	91	83	77	71	67	63	59	56	53	50	48	45	43	42	40	38	37	36	34	33	32
11	92	85	79	73	69	65	61	58	55	52	50	48	46	44	42	41	39	38	37	35	34
12	92	86	80	75	71	67	63	60	57	55	52	50	48	46	44	43	41	40	39	38	36
13	93	87	81	76	72	68	65	62	59	57	54	52	50	48	46	45	43	42	41	39	38
14	93	88	82	78	74	70	67	64	61	58	56	54	52	50	48	47	45	44	42	41	40
15	94	88	83	79	75	71	68	65	63	60	58	56	54	52	50	48	47	45	44	43	42
16	94	89	84	80	76	73	70	67	64	62	59	57	55	53	52	50	48	47	46	44	43
17	94	89	85	81	77	74	71	68	65	63	61	59	57	55	53	52	50	49	47	46	45
18	95	90	86	82	78	75	72	69	67	64	62	60	58	56	55	53	51	50	49	47	46
19	95	90	86	83	79	76	73	70	68	66	63	61	59	58	56	54	53	51	50	49	48
20	95	91	87	83	80	77	74	71	69	67	65	63	61	59	57	56	54	53	51	50	49

Таблица 2. Мера доверия к миллсовой модели в зависимости от гипотезы о количестве естественных ошибок (*N*) и количества внесенных искусственных ошибок (*M*) для *N* и *M* от 0 до 100 с шагом 5, %

M	*N*																				
	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	83	45	31	24	19	16	14	12	11	10	9	8	8	7	7	6	6	5	5	5	5
10	91	63	48	38	32	28	24	22	20	18	16	15	14	13	12	12	11	10	10	9	9
15	94	71	58	48	42	37	33	29	27	25	23	21	20	19	17	16	16	15	14	14	13
20	95	77	65	56	49	43	39	36	33	30	28	26	25	23	22	21	20	19	18	17	17
25	96	81	69	61	54	49	45	41	38	35	33	31	29	27	26	25	24	23	22	21	20
30	97	83	73	65	59	54	49	45	42	39	37	35	33	31	30	28	27	26	25	24	23
35	97	85	76	69	63	57	53	49	46	43	41	38	36	35	33	32	30	29	28	27	26
40	98	87	78	71	66	61	56	53	49	47	44	42	40	38	36	34	33	32	31	29	28
45	98	88	80	74	68	63	59	56	52	49	47	45	42	41	39	37	36	34	33	32	31
50	98	89	82	76	70	66	62	58	55	52	50	47	45	43	41	40	38	37	35	34	33
55	98	90	83	77	72	68	64	60	57	54	52	50	47	45	44	42	40	39	38	36	35
60	98	91	85	79	74	70	66	63	59	57	54	52	50	48	46	44	43	41	40	38	37
65	98	92	86	80	76	71	68	64	61	59	56	54	52	50	48	46	45	43	42	40	39
70	99	92	86	81	77	73	69	66	63	60	58	56	53	51	50	48	46	45	43	42	41
75	99	93	87	82	78	74	71	68	65	62	60	57	55	53	51	50	48	47	45	44	43
80	99	93	88	83	79	75	72	69	66	63	61	59	57	55	53	51	50	48	47	45	44
85	99	93	89	84	80	77	73	70	67	65	63	60	58	56	54	53	51	50	48	47	46
90	99	94	89	85	81	78	74	71	69	66	64	62	60	58	56	54	53	51	50	48	47
95	99	94	90	86	82	79	75	73	70	67	65	63	61	59	57	56	54	52	51	50	48
100	99	94	90	86	83	79	76	74	71	68	66	64	62	60	58	57	55	54	52	51	50

Таблица 3. Количество искусственных ошибок, которые необходимо внести в программу для достижения нужной меры доверия к миллсовой модели в зависимости от гипотезы о количестве естественных ошибок (*N*) для *N* от 0 до 20 с шагом 1

C, %	*N*																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
10	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
15	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	4	4
20	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5
25	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6	6	6	7	7
30	0	1	1	2	2	3	3	3	4	4	5	5	6	6	6	7	7	8	8	9	9
35	1	1	2	2	3	3	4	4	5	5	6	6	7	8	8	9	9	10	10	11	11
40	1	1	2	3	3	4	5	5	6	7	7	8	9	9	10	11	11	12	13	13	14
45	1	2	2	3	4	5	6	7	7	8	9	10	11	11	12	13	14	15	16	16	17
50	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
55	1	2	4	5	6	7	9	10	11	12	13	15	16	17	18	20	21	22	23	24	26
60	2	3	5	6	8	9	11	12	14	15	17	18	20	21	23	24	26	27	29	30	32
65	2	4	6	7	9	11	13	15	17	19	20	22	24	26	28	30	32	33	35	37	39
70	2	5	7	9	12	14	16	19	21	23	26	28	30	33	35	37	40	42	44	47	49
75	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60	63
80	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84
85	6	11	17	23	28	34	40	45	51	57	62	68	74	79	85	91	96	102	108	113	119
90	9	18	27	36	45	54	63	72	81	90	99	108	117	126	135	144	153	162	171	180	189
95	19	38	57	76	95	114	133	152	171	190	209	228	247	266	285	304	323	342	361	380	399
99	99	198	297	396	495	594	693	792	891	990	1089	1188	1287	1386	1485	1584	1683	1782	1881	1980	2079

Таблица 4. Количество искусственных ошибок, которые необходимо внести в программу для достижения нужной меры доверия к миллсовой модели в зависимости от гипотезы о количестве естественных ошибок (*N*) для *N* от 0 до 100 с шагом 5

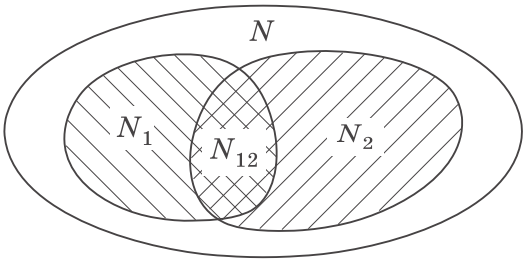
C, %	*N*																				
	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	5	5	5	5
10	0	1	1	2	2	3	3	4	5	5	6	6	7	7	8	8	9	10	10	11	11
15	0	1	2	3	4	5	5	6	7	8	9	10	11	12	13	13	14	15	16	17	18
20	0	2	3	4	5	7	8	9	10	12	13	14	15	17	18	19	20	22	23	24	25
25	0	2	4	5	7	9	10	12	14	15	17	19	20	22	24	25	27	29	30	32	34
30	0	3	5	7	9	11	13	15	18	20	22	24	26	28	30	33	35	37	39	41	43
35	1	3	6	9	11	14	17	19	22	25	27	30	33	36	38	41	44	46	49	52	54
40	1	4	7	11	14	17	21	24	27	31	34	37	41	44	47	51	54	57	61	64	67
45	1	5	9	13	17	21	25	29	34	38	42	46	50	54	58	62	66	70	74	79	83
50	1	6	11	16	21	26	31	36	41	46	51	56	61	66	71	76	81	86	91	96	101
55	1	7	13	20	26	32	38	44	50	56	62	68	75	81	87	93	99	105	111	117	123
60	2	9	17	24	32	39	47	54	62	69	77	84	92	99	107	114	122	129	137	144	152
65	2	11	20	30	39	48	58	67	76	85	95	104	113	123	132	141	150	160	169	178	188
70	2	14	26	37	49	61	72	84	96	107	119	131	142	154	166	177	189	201	212	224	236
75	3	18	33	48	63	78	93	108	123	138	153	168	183	198	213	228	243	258	273	288	303
80	4	24	44	64	84	104	124	144	164	184	204	224	244	264	284	304	324	344	364	384	404
85	6	34	62	91	119	147	176	204	232	261	289	317	346	374	402	431	459	487	516	544	572
90	9	54	99	144	189	234	279	324	369	414	459	504	549	594	639	684	729	774	819	864	909
95	19	114	209	304	399	494	589	684	779	874	969	1064	1159	1254	1349	1444	1539	1634	1729	1824	1919
99	99	594	1089	1584	2079	2574	3069	3564	4059	4554	5049	5544	6039	6534	7029	7524	8019	8514	9009	9504	9999

Еще одно слабое место — это требование второй части миллсовой модели отыскать непременно все искусственные ошибки. А этого может не произойти долго, может быть и никогда. Существует модификация модели, позволяющая учесть нахождение только части искусственных ошибок. Вычисление C там проводится по более громоздкой формуле, и рассматривать ее здесь мы не будем.

14.2. «Парная» оценка

Следующая модель требует тестирования программы двумя независимыми специалистами (или группами специалистов). Зато не требует искусственного внесения ошибок в программу.

Пусть программу тестируют независимо друг от друга две группы специалистов. Предположим, что в программе содержится N ошибок. Пусть первая группа нашла N_1 ошибок, а вторая — N_2. Часть ошибок обнаружена обеими группами. Пусть таких ошибок N_{12}. Изобразим соответствующие множества на схеме.



Эффективность работы групп оценим через процент обнаруженных ими ошибок:

$$E_1 = \frac{N_1}{N}, \quad E_2 = \frac{N_2}{N}.$$

Обнаружение всех ошибок считаем равновероятным. Тогда верно следующее рассуждение (оно довольно громоздко, и, возможно, его придется, не торопясь, перечитать пару раз).

В силу равновероятности нахождения любой из ошибок любое случайным образом выбранное подмножество из N можно рассматривать как аппроксимацию всего множества N. Это зна-

чит, что если первая группа обнаружила 10% всех ошибок, она должна обнаружить примерно 10% ошибок из любого случайным образом выбранного подмножества. В качестве такового случайным образом выбранного подмножества возьмем множество ошибок, найденных второй группой. Доля всех ошибок, найденных первой группой, равна $\frac{N_1}{N}$. Доля ошибок, найденных первой группой среди тех ошибок, которые были найдены второй группой, равна $\frac{N_{12}}{N_2}$. Согласно нашим рассуждениям, эти две величины должны быть равны:

$$\frac{N_1}{N} = \frac{N_{12}}{N_2}.$$

Отсюда количество ошибок в программе: $N = \frac{N_1 \cdot N_2}{N_{12}}$.

Количество ненайденных ошибок равно $(N - N_1 - N_2 + N_{12})$.

Например, пусть первая группа нашла 8 ошибок, вторая — 9. Обеими группами были найдены 3 ошибки. Тогда количество ошибок в программе $N = \frac{8 \cdot 9}{3} = 24$. Из них уже найдено $(8 + 9 - 3) = 14$. Осталось найти еще 10.

14.3. Исторический опыт

Еще один способ оценить количество ошибок в программе — использовать опыт предыдущих программ. Оценка эта не очень надежная, однако, больше, чем ничего. В литературе встречаются следующие оценки. После написания текста программы в ней содержится (в среднем) 4–8 ошибок на 100 операторов. После окончания автономной отладки модулей в программе остается 1 ошибка на 100 операторов. Последняя оценка дается со ссылкой на фирму IBM. Заметим, что в фирме IBM работают профессионалы. Для новичков такая оценка может оказаться слишком оптимистичной.

Для крупных длительных проектов некоторые фирмы-разработчики строят специальные модели, отражающие специфику именно этого проекта. Так, в процессе работы над уже упоминавшейся ОС/360 фирма IBM использовала для оценки числа ошибок в системе формулу:

$$N = 2 \cdot ИМ + 23 \cdot МИМ.$$

Здесь

N — полное число исправлений из-за ошибок,
ИМ — число исправляемых модулей,
МИМ — число многократно исправляемых модулей.

Многократно исправляемыми считались модули, которые потребовали 10 или более исправлений. *ИМ* оценивалось как 90% новых модулей и 15% старых, *МИМ* — как 15% новых модулей и 6% старых. При подстановке этих оценок формула получает вид

$$N_{\text{испр.}} = 2 \cdot (0{,}9 \cdot N_{\text{нов.мод.}} + 0{,}15 \cdot N_{\text{стар.мод.}}) + \\ + 23 \cdot (0{,}15 \cdot N_{\text{нов.мод.}} + 0{,}06 \cdot N_{\text{стар.мод.}}).$$

Таким образом, если в вашей системе уже содержится 140 модулей и в процессе обновления предстоит добавить еще 20, то количество ошибок, которое при этом будет обнаружено, оценивается следующим образом:

$$2 \cdot (0{,}9 \cdot 20 + 0{,}15 \cdot 140) + 23 \cdot (0{,}15 \cdot 20 + 0{,}06 \cdot 140) = \\ = 2 \cdot (18 + 21) + 23 \cdot (3 + 8{,}4) = \\ = 2 \cdot 39 + 23 \cdot 11{,}4 = 78 + 262{,}2 = 340{,}2.$$

Можно ожидать 340 ошибок. В 18 из 20 новых модулей придется внести хотя бы одно исправление, в 3 модуля — не менее десятка. Из старых модулей хотя бы один раз придется исправить 21 модуль, а 8 или 9 модулей — не менее 10 раз.

Стоит сделать два замечания. Во-первых, очевидно, что описанная модель годится только для достаточно крупных проектов. Во-вторых, разрабатывалась она для конкретной системы. Не факт, что ее можно напрямую применять в других случаях.

Глава 15

*Оценка количества необходимых тестов[6]

В предыдущей главе мы рассмотрели модели, которые позволяют оценить количество ошибок в программе. А сколько тестов понадобится, чтобы обнаружить эти ошибки?

В [3] для ответа на этот вопрос предлагается следующая модель. Утверждается, что вероятность успешности очередного теста зависит от процента оставшихся ошибок. Последние ошибки обнаружить сложнее. (Заметим, что данное утверждение вполне соответствует опыту.)

Пусть T_n — среднее количество тестов, необходимых для обнаружения n ошибок, N — число ошибок в программе. Для оценки T_n предлагается следующая формула:

$$T_n = \alpha \sum_{k=0}^{n-1} \frac{1}{N-k}.$$

Здесь α — некий коэффициент, значение которого надо определять экспериментально. Но мы заниматься этим не будем, поскольку формула для вычисления T_n будет интересовать нас не сама по себе, а лишь как база для последующих рассуждений.

Мы хотим оценить количество тестов, которое потребуется для нахождения всех N ошибок. Посчитать напрямую T_N мы не можем, поскольку не знаем коэффициента α. Чтобы исключить его из вычислений, перейдем от абсолютных величин к относительным. Попробуем оценить, какую часть от всех необходимых тестов придется выполнить для того, чтобы найти первые n ошибок.

$$P = \frac{T_n}{T_N} = \frac{\alpha \sum_{k=0}^{n-1} \frac{1}{N-k}}{\alpha \sum_{k=0}^{N-1} \frac{1}{N-k}} = \frac{\sum_{k=0}^{n-1} \frac{1}{N-k}}{\sum_{k=0}^{N-1} \frac{1}{N-k}}.$$

Теперь осталось подставить в формулу для P конкретные значения N и n. Результаты — очень интересные — представлены в таблицах 5 и 6. В таблице 5 N изменяется от 10 до 100, в таб-

---

[6] **Глава отмечена звездочкой. Это означает, что некоторые ее части могут показаться недостаточно подготовленному читателю слишком сложными. Можете пропустить такие части (если это допускается вашей учебной программой). Чтение последующих глав не требует обязательного знания материала данного раздела.**

лице 6 — от 1 до 10. Последняя таблица построена специально для новичков, программы которых настолько малы по размерам, что в них просто не найдется места для сотни ошибок.

Таблица 5. Средний процент тестов, необходимых для обнаружения заданного процента ошибок (в зависимости от общего числа ошибок в программе)

Процент найденных ошибок (*n*/*N*)	Общее число ошибок в программе (*N*)									
	10	20	30	40	50	60	70	80	90	100
10	3	3	3	2	2	2	2	2	2	2
20	7	6	5	5	5	5	5	4	4	4
30	11	10	9	8	8	8	7	7	7	7
40	16	14	13	13	11	11	10	10	10	10
50	22	19	17	16	15	15	14	14	14	13
60	29	24	22	21	20	19	19	18	18	18
70	37	32	29	27	26	25	25	24	23	23
80	49	42	39	36	35	34	33	32	31	31
90	66	58	54	51	49	48	46	45	44	44
100	100	100	100	100	100	100	100	100	100	100

Оказывается, что при $N = 10$ первые 22% тестов обнаружат половину всех ошибок (5 штук). Для того чтобы обнаружить две следующие ошибки, количество тестов придется увеличить в 1,7 раза — до 37%. Поиск следующих двух ошибок потребует увеличения количества тестов еще в 1,8 раза — до 66%. И наконец, поиск последней ошибки потребует оставшихся 34% тестов.

Чем больше ошибок в программе, тем дороже обойдется поиск последних ошибок. При $N = 50$ для обнаружения 50% ошибок достаточно 15% тестов, 70% ошибок будут найдены с помощью 26%, 90% ошибок — с помощью 49% тестов. Поиск последних 10% ошибок будет стоить дороже, чем поиск первых 90%!

При увеличении количества ошибок с 10 до 100 стоимость поиска последних 10% ошибок возрастает с 34% тестов до 66%.

Заметим, что речь идет не об абсолютном числе тестов, а о их доле. Поскольку программа, содержащая 100 естественных ошибок, скорее всего, сложнее программы, в которой ошибок только 10, есть основания полагать, что общее число тестов также будет возрастать. То есть придется брать больший процент от большего числа тестов.

В таблице 6 N изменяется от 1 до 10. В этом случае говорить о процентах найденных ошибок смысла нет. Поэтому в боковике записаны не относительные, а абсолютные значения. Поскольку указанное количество ошибок вполне реально для учебных программ, интересно было сравнить модельные данные, приведенные в таблице, с реальными данными из практики. Надо только помнить, во-первых, что речь идет о средних значениях. А во-вторых, что данные в таблице получены из статистической модели. А статистика любит большие числа.

Таблица 6. Средний процент тестов, необходимых для обнаружения заданного количества ошибок (в зависимости от общего числа ошибок в программе)

Количество найденных ошибок (n)	Общее число ошибок в программе (N)									
	1	2	3	4	5	6	7	8	9	10
1	100	33	18	12	9	7	6	5	4	3
2	—	100	45	28	20	15	12	10	8	7
3	—	—	100	52	34	35	20	16	13	11
4	—	—	—	100	56	39	29	23	19	16
5	—	—	—	—	100	59	42	33	26	22
6	—	—	—	—	—	100	61	45	35	29
7	—	—	—	—	—	—	100	63	47	37
8	—	—	—	—	—	—	—	100	65	49
9	—	—	—	—	—	—	—	—	100	66
10	—	—	—	—	—	—	—	—	—	100

Имея такие оценки относительного количества тестов, в конкретном проекте можно перейти к абсолютным величинам. Поскольку нам известно, сколько тестов нам понадобилось для нахождения n ошибок, легко оценить, сколько понадобится для нахождения оставшихся. Если количество ошибок в программе оценено в 10 и для обнаружения первых пяти потребовалось, например, 7 тестов, то для поиска двух следующих ошибок количество тестов придется довести до 12 (5 дополнительных тестов на 2 ошибки), для поиска следующих двух — до 21 (9 дополнительных тестов на 2 ошибки). Все 10 ошибок есть надежда найти за 32 теста.

Подобные оценки можно использовать для планирования времени и ресурсов, выделяемых для процесса тестирования.

Глава 16

Отладка

Выше мы определили отладку[7] как процесс локализации места ошибки и внесения изменений в программу. В этой главе мы обсудим некоторые вопросы, связанные с определением места ошибки.

16.1. Место проявления ошибки и место нахождения ошибки

Прежде всего, необходимо различать **место проявления ошибки** и **место нахождения ошибки.** Вернемся к примеру 1 из главы 7 «Критерии белого ящика»:

```
a:= 0;
if x>3 then a:= 10;
b:= 1/a;
```

Место проявления ошибки в данном случае — третий оператор фрагмента: `b:= 1/a`. Но место нахождения ошибки указать так определенно мы не можем. Возможно, что ошибка в третьем операторе, и в знаменателе должна стоять не переменная `a`, а какое-то иное выражение. Возможно, что ошибка в первом операторе, и переменной `a` должно было быть присвоено иное ненулевое значение. Возможно, что ошибка во втором операторе, и там потеряна ветвь «иначе», при выполнении которой переменная `a` должна была поменять свое значение на ненулевое.

---

[7] В английском языке используется термин «debug», который является буквальным аналогом принятого в русском языке (но не имеющего отношения к computer science) термина «инсектикация» (борьба с насекомыми). Причины появления англоязычного термина исторические. Согласно апокрифу, во время одного из сеансов работы «Эниака» (машины, которую принято считать первой американской ЭВМ, хотя она и не имела памяти команд) произошел аварийный останов, вызванный тем, что под одну из клемм одного из многочисленных устройств попал мотылек (честно говоря, не совсем понятно, как). На поиск места неисправности и ее устранение потребовалось время. Виновник был найден и приклеен в журнал для регистрации работы ЭВМ. Но в это время последовал телефонный звонок от начальства с вопросом, почему машина не работает и чем там вообще занимается обслуживающий персонал. На что Грейс Мюррей Хоппер — руководитель группы кодировщиков (слова «программист» тогда еще не существовало) ответила: «Ловим насекомых».

По каким-то внешним признакам мы можем найти только место проявления ошибки. Место нахождения ошибки можно определить только путем содержательного анализа текста программы.

16.2. Отладочные операторы

После обнаружения факта наличия в программе ошибки уточнение ее места нахождения может потребовать сбора дополнительной информации о ходе выполнения программы. Большинство систем программирования включает в себя те или иные отладочные средства. (В следующей главе будут рассмотрены отладочные средства популярной системы Турбо Паскаль.) Кроме встроенных отладочных средств для сбора данных о работе программы можно использовать специальные **отладочные операторы**, добавляемые в ее текст.

Так же, как и в случае с заглушками, отладочные операторы должны отличаться от «нормальных» операторов программы, а отладочные сообщения — от «нормальных» сообщений. В главе 13 «Нисходящее тестирование» для этого предлагалось использовать в качестве маркера строку из двух диезов: «##». Все отладочные операторы будут помечаться комментарием, содержащим эту пометку, а все отладочные сообщения — начинаться с этой комбинации.

Как правило, отладочные операторы — это операторы печати, которые выполняют полную или частичную трассировку значений нужных переменных. Трасса может выдаваться на экран или записываться в файл. При этом надо учитывать следующие факторы. Информация, выводимая на экран, может очень быстро «убежать» с него, вытесненная новыми данными. Поэтому если трассируемые значения выводятся на экран, может быть полезно после выдачи очередной порции информации сделать паузу. В некоторых языках есть специальные операторы `pause`. Там, где их нет, для приостановки выполнения программы может использоваться оператор ввода, который ждет ввода конца строки (т. е. нажатия клавиши **Enter**). В Турбо Паскале для этого достаточно вызова процедуры `readln` без параметров.

Главный недостаток выдачи информации на экран — невозможность ее дальнейшего анализа. Для сохранения трассировочных данных их можно записать в файл. При этом программиста ждет другая опасность. Велик искус начать записывать в файл «все подряд» по принципу «авось пригодится». В результате трассировочные файлы достигают совершенно неверо-

ятных размеров, и разобраться в их содержимом становится практически невозможно. Не случайно подобный метод сбора информации называют методом «грубой силы».

Полезно, чтобы по отладочному сообщению можно было однозначно определить, какой именно отладочный оператор его выдал. Для этого в выдаваемый текст вдобавок к отладочной информации надо включать указание на точку выдачи.

Удобно, когда в отладочном сообщении выдаются не только значения трассируемых переменных, но и их имена. Это существенно упрощает анализ трассировки.

С учетом всего сказанного отладочная печать может выглядеть, например, так:

```
{##} writeln('##Точка QR7: a=', a, 'b=', b);
{##} readln;
```

Как только в программе появляются отладочные операторы, возникает желание управлять ими (подключать или отключать при очередном выполнении программы). Физическое удаление отладочных операторов из текста программы — решение слабое. А вдруг они еще понадобятся? Удобным приемом для отключения отладочных операторов является превращение их в комментарий, заключение в «комментаторные скобки». Заметим, что Турбо Паскаль поощряет использование этого приема тем, что имеет две пары скобок для записи комментариев. Если в стандартном Паскале комментарий обязательно заключается в фигурные скобки: {комментарий}, то в Турбо Паскале комментарий может быть заключен либо в фигурные скобки, либо в «звездчатые скобки», составленные из пары «круглая скобка — звездочка»: (*комментарий*). Причем скобки должны быть обязательно парными! Комментарий, открытый фигурной скобкой, закрывается только фигурной скобкой. Звездчатые скобки внутри него игнорируются. Аналогично комментарий, открытый звездчатой скобкой, закрывается только звездчатой скобкой. Фигурные скобки внутри него игнорируются. Это дает возможность отделить обычные комментарии от отключенных отладочных операторов. Для этого достаточно для выделения обычных комментариев использовать один вид скобок (например, фигурные), а для отключения отладочных операторов — другой (звездчатые). Тогда отключение отладочных операторов будет иметь, например, такой вид:

```
a:= b;
{это оператор программы с обычным комментарием}
(*{##} writeln('##Точка SD3: a= ',a);
{это отладочная печать}*)
```

Еще один полезный инструмент отладки — встраивание в текст программы утверждений о том, каким ограничениям должны удовлетворять те или иные переменные. Если известно, что переменная a всегда должна быть больше нуля, а переменная b в какой-то точке программы должна быть меньше переменной c, в соответствующих местах программы ставятся утверждения assert(a>0), assert(b<c). Мысль явно формулировать и указывать подобные утверждения была предложена в 70-е гг. XX в. при разработке идей формального доказательства правильности программ. В некоторых языках программирования даже появились соответствующие конструкции. Нарушение указанного в утверждении условия должно было приводить к аварийному останову программы. Впрочем, отсутствие специальных конструкций в языке не является серьезным препятствием для использования утверждений. На любом языке легко написать процедуру следующего вида:

```
procedure Assert(cond: Boolean; message: string;
                 loc: string);
 begin
   if not cond then begin
     writeln('Нарушено утверждение',message,'в точке',loc);
     halt {А может останов и не нужен? Пусть считает дальше?}
   end {if}
 end; {Assert}
```

В этом случае вызов типа

```
Assert(p>0,'Вес человека > 0','P1.q');
```

потребует от программиста усилий не намного больших, чем стандартная конструкция, а сработает ничуть не хуже.

Резко усилить мощность отладочных средств можно, сделав их условными. Делается это следующим образом. В программе описываются одна или несколько специальных переменных для управления отладкой. Будем называть их отладочными флагами. Переменные эти, как правило, логические, хотя могут быть и других типов. Отладочные операторы помещаются внутрь условных операторов, выполнение которых зависит от отладочных флагов. Например, так:

```
{##} var DebugFlag: Boolean;
…
{##} DebugFlag:= true; {DebugFlag:= false;}
…
{##} if DebugFlag then writeln('##Точка P1:x=',x,'y=',y);
…
```

Теперь можно включать и выключать отладочные действия без изменения текста программы. Для этого достаточно изменить значение отладочных флагов. Причем управление может быть достаточно «хитрым». Отладочные операторы могут быть включены в одной части программы и отключены в другой. Могут существовать наборы отладочных операторов, расположенных в разных частях программы, но управляемых одним и тем же отладочным флагом и, соответственно, включающихся/выключающихся одновременно. Это имеет смысл, если все операторы такого набора отслеживают, например, одну и ту же переменную.

Менять значения отладочных флагов (и, соответственно, включать/выключать отладочные действия) можно динамически по ходу выполнения программы.

Можно ввести понятие «уровень отладки». Меняя значение соответствующей управляющей переменной (она должна быть числом), можно регулировать подробность трассировочной информации.

Начальное значение отладочных флагов может быть задано в тексте программы, а может загружаться из файла. В последнем случае включать/выключать трассировки можно будет, даже не перетранслируя текст программы. Например, так:

```
{##} var debug1, debug2: integer;
{##} var DebugFile: text;
…
{##} assign(DebugFile,'dbg.txt');
{##} reset(DebugFile);
{##} read(DebugFile, debug1, debug2);
…
{##} if debug1>0 then writeln('##Точка FDR z= ',z);
```

В принципе условные отладочные операторы вообще необязательно удалять из текста программы. Они могут оставаться там (естественно, в отключенном виде) и по окончании отладки. Особенно если в перспективе возможна дальнейшая модификация программы. Достаточно вспомнить приведенную в главе 14 «Оценка количества ошибок в программе» IBM-овскую формулу для оценки количества исправлений в старых модулях системы при добавлении в систему новых модулей. Для больших систем модификация (а значит, и отладка) могут стать процессом непрерывным.

Платой за наличие отладочных операторов в тексте программы станет некоторое повышение требований к машинным ресурсам. Отладочные операторы увеличивают размер исходной программы, время трансляции, размер кода, время выполнения. Избежать всего этого можно, если использовать механизм условной трансляции, который поддерживают многие системы программирования. Отладочные операторы можно сделать условно транслируемыми. Если версия программы предназначена для отладки, трансляция отладочных операторов включается, и они работают. Если отладка закончена и пора готовить систему к эксплуатации, надо перетранслировать программу, отключив трансляцию отладочных средств. В результате в окончательной версии программы их уже не окажется.

В Турбо Паскале установка/сброс символов для управления условной трансляцией производится в опциях компилятора **Options/Compiler.../Conditional Defines** (см. рис. 9.1 на стр. 33) либо директивами DEFINE и UNDEF. Управление условной трансляцией осуществляется с помощью директив: IFDEF, IFNDEF, IFOPT, ELSE, ENDIF.

Последнее замечание по поводу отладочных операторов. Помните, что добавление/удаление отладочных операторов — это изменение текста программы. Независимо от того, каким образом оно выполнялось: вручную или автоматически с помощью условной трансляции. И — как любое изменение — оно может привести к внесению в программу ошибок. Поэтому после удаления из программы отладочных операторов тестирование программы необходимо повторить.

16.3. Индуктивный и дедуктивный методы поиска ошибки. Ретроанализ

К поиску ошибки существует два подхода: индуктивный и дедуктивный.

Индуктивный подход означает движение от частного к общему. От того, какие данные говорят об ошибке, к тому, как их можно объяснить.

Для сбора информации о работе программы могут быть полезны дополнительные тесты, спроектированные специально для нужд отладки. Такие тесты отличаются от тестов, предназначенных для выявления факта наличия ошибки. Тесты для выявления факта наличия ошибки должны быть как можно более «охватными». Чем большую часть программы прове-

рит такой тест, тем лучше. К отладочным тестам требования противоположные. От них требуется не широта охвата, а прицельность, сбор как можно более точной и подробной информации о том конкретном месте программы, в котором есть основания подозревать ошибку.

Схема индуктивных рассуждений:

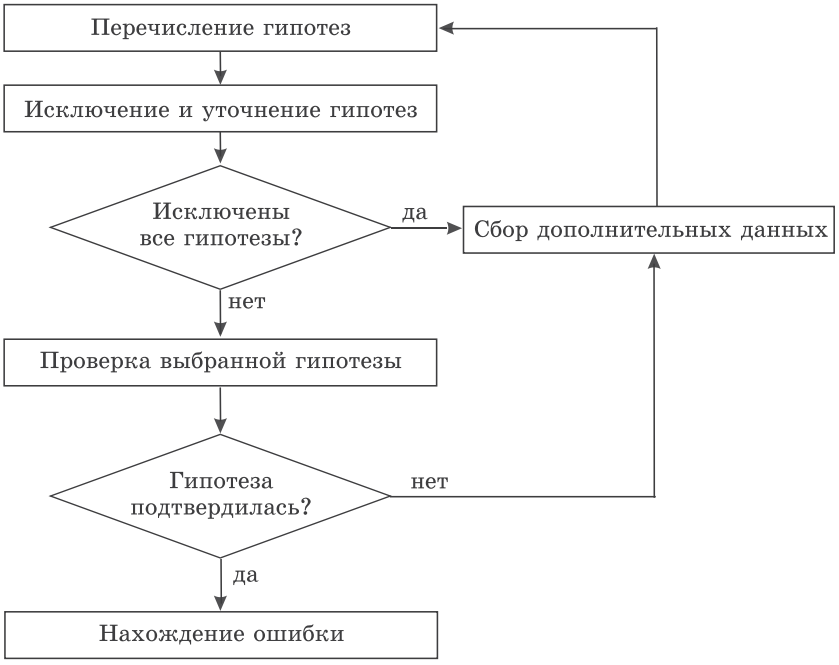
Дедуктивный подход означает движение от общего к частному. Что в принципе могло произойти? Исходя из некоторых общих соображений формируется множество гипотез. Затем оно уточняется: какие-то гипотезы будут исключены как несоответствующие имеющимся признакам ошибки, какие-то уточнены за счет дополнительной информации.

Например: программа нормально заканчивает вычисления, но выдает странные результаты. На выходе ожидались два числа, близких к единице с одним знаком после запятой. Получили одно число очень большое, а второе — со странной непериодической дробной частью. Известно, что «странные» выходные данные могут получиться, если в вычислениях участвовала неини-

циализированная переменная, либо при вводе из файла «не тех» значений (например, файл готовился как текстовый, а читался как типизированный), либо при неверной передаче значения параметра из подпрограммы, либо при неправильных вычислениях (например, требующих слишком больших или слишком маленьких промежуточных результатов). Значит, надо проверить, все ли переменные, использованные при вычислении «странного» результата, были явно инициализированы. Не использовались ли для вычисления этих результатов значения, введенные из файла? Если да, то что за данные лежат в этом файле. Не было ли передачи параметра из подпрограммы? Если да, то передается ли этот параметр по ссылке? Результатом каких именно вычислений являются «странные» значения? Не могло ли по ходу этих вычислений возникнуть слишком больших или слишком маленьких промежуточных результатов?

Надо отметить, что именно дедуктивный подход (основанный на накопленном опыте) часто позволяет преподавателям быстро найти ошибку в программе студента. Ошибки-то повторяются!

Схема дедуктивных рассуждений:



Ретроанализ

Четверть века назад в шахматных журналах были очень популярны задачи на так называемый ретроанализ. Выглядели они так. Дается ситуация, при которой белые ставят мат в один ход и черные ставят мат в один ход. Вопрос: кто выиграет? То есть надо разобраться, чей сейчас ход. А для этого «открутить» игру назад и понять, каким образом сложилась описанная в задаче ситуация.

Как ретроанализ выглядит применительно к отладке? Находим место ошибки. Ошибка связана с тем, что некоторые переменные имеют определенные значения. Проследим назад по программе, откуда эти значения взялись. Они были вычислены через некоторые другие переменные. А откуда взялись значения этих других переменных? И так далее. Например:

```
read(a);
b:= 7;
c:= a + b;
d:= 1/c;
```

При вычислении `d` возникает деление на нуль. Значит, переменная `c` в этом операторе равна нулю. Возможны два варианта. Либо ошибочно выражение, записанное в знаменателе (знаменатель должен быть отличен от `c`), либо переменная `c` имеет неверное значение. Если выражение в знаменателе правильно, значит, переменная `c` имеет неверное значение. Откуда взялось значение переменной `c`? Поднимаемся вверх по программе. Переменная `c` получила значение из выражения `a + b`. Значит, либо в этой позиции должно стоять другое выражение, либо что-то не так со значениями переменных `a` и `b`. Откуда взялись значения этих переменных? Опять поднимаемся вверх по программе. И так далее. Не всегда картина бывает такой ясной. Разбираться приходится и с развилками, и с циклами, и с подпрограммами. Но принципиальная схема именно такова.

16.4. Принципы отладки

1. Думай! Средства отладки играют только вспомогательную роль. В этом отношении процесс отладки напоминает работу детектива. Вспомните «В августе 44-го...» В. Богомолова. Массовая отладочная печать напоминает войсковую операцию. Внешне она, может быть, и эффектна. Но добиться с ее помощью «момента истины» сложно.
2. Избегай экспериментирования. Работа по принципу «я не знаю, в чем ошибка, но сейчас исправлю вот это место и посмотрю, к чему это приведет» — совершенно недопустима!

3. Исправляй поочередно. Одновременное внесение в программу нескольких исправлений существенно затрудняет анализ последствий каждого из них.
4. Необходимо найти ошибку, которая бы объясняла все 100% симптомов.
5. Там, где есть ошибка, может быть еще.
6. Исправление может внести новую ошибку.

16.5. Анализ обнаруженной ошибки

Раз уж мы допустили ошибку в программе, хочется, по возможности, обратить вред в пользу и извлечь из допущенной ошибки максимум пользы на будущее. Для этого надо проанализировать обнаруженную ошибку по следующему плану:

1. Когда была сделана ошибка?

На каком этапе работы над программой допущена ошибка: при постановке задачи, при проектировании программы, при написании текста на языке программирования и т. д.

2. Почему была сделана ошибка?

Что именно было непонятно? Была ли причиной ошибки неточность в формулировке задачи, или недопонимание какой-то языковой конструкции, или неудачный выбор имени переменной (который сделал описку ошибкой), или что-либо еще.

3. Как можно было предотвратить ошибку?

Есть хороший методический принцип: «Никогда не спрашивай человека, чему он научился. Спрашивай, что он в следующий раз будет делать по-другому». Именно этот вопрос мы сейчас и задаем. Что в следующий раз надо делать по-другому, чтобы подобные ошибки не повторялись?

4. Почему ошибку не обнаружили раньше?

Если ошибку удалось обнаружить сейчас, то почему ее не удалось обнаружить раньше, на более ранних этапах?

5. Как можно было обнаружить раньше?

Как нашли ошибку? Почему именно этот тест оказался удачным? Нельзя ли таким же образом найти аналогичные ошибки в этой или других программах?

Результаты анализа полезно фиксировать в «дневнике отладки». Ошибки и их исправление — процесс творческий. Свой собственный опыт в этой области заменить трудно. А вот собирать и анализировать его очень полезно.

Глава 17

Отладочные средства системы Турбо Паскаль

17.1. Перечень отладочных средств Турбо Паскаля

В данной главе описаны отладочные средства, предоставляемые системой программирования Турбо Паскаль версии 7.0. В других версиях они могут несколько отличаться, но отличия эти непринципиальны.

Для поиска ошибок в программе, а также для лучшего понимания того, как именно она работает, можно использовать следующие возможности системы Турбо Паскаль:

1) пошаговое исполнение программы с заходом в процедуры и без захода;
2) исполнение «до курсора»;
3) установка контрольных точек, в том числе условных и со счетчиком;
4) наблюдение за значениями переменных;
5) вычисление выражений и изменение значений переменных во время приостанова программы;
6) наблюдение за состоянием стека вызванных подпрограмм;
7) динамическое управление отладочными средствами.

Рис. 17.1. Вертикальное меню **Run**

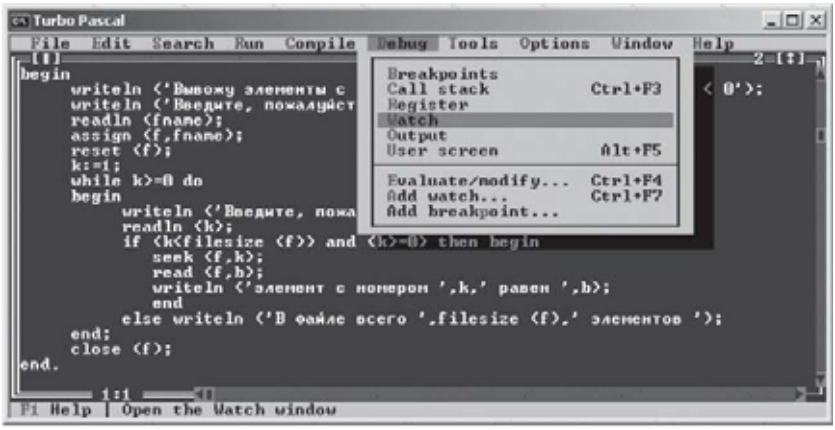


Рис. 17.2. Вертикальное меню **Debug**

В верхнем меню они сосредоточены в пунктах **Run** и **Debug** (см. рис. 17.1, 17.2).

Команды из вертикального меню пункта **Run** обеспечивают управление способом выполнения программы, из пункта **Debug** — управление контрольными точками, наблюдение за значением переменных и вызовами подпрограмм, вычисление выражений и переменных. Кроме того, часть наиболее ходовых команд продублирована в локальном меню окна редактирования программы. Это меню вызывается комбинацией клавиш **Alt+F10** (см. рис. 17.3).

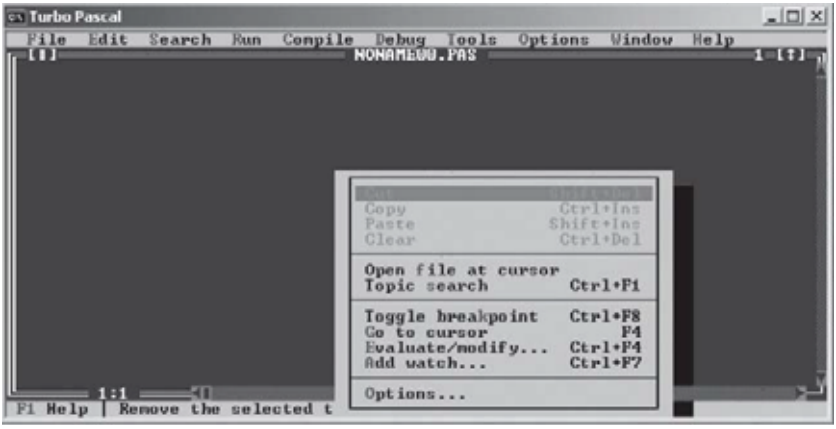


Рис. 17.3. Локальное меню в окне редактирования текста программы

Рис. 17.4. Сообщение об ошибке при отключенных опциях сбора отладочной информации

Надо подчеркнуть, что для нормальной работы отладочных средств необходимо, чтобы были включены две опции компилятора: **Options/Compiler.../Debug information** и **Options/Compiler.../Local symbols** (см. рис. 9.1 на стр. 33). Включение этих опций обеспечивает сбор отладочной информации и позволит далее вести отладку в терминах языка высокого уровня (использовать имена переменных и процедур, ссылаться на конкретные строки текста программы). При выключенных опциях вы можете рассчитывать только на сообщения типа «Runtime error» (см. рис. 17.4).

17.2. Пошаговое выполнение программы

Программа в Турбо Паскале может выполняться безостановочно или пошагово.

Безостановочное выполнение запускается командой **Run** в вертикальном меню **Run** (горячая клавиша **Ctrl+F9**). В простейшем (с точки зрения отладки) случае выполнение начинается с начала программы и продолжается до конца. В более сложных случаях внутри программы могут быть установлены точки приостанова. В этом случае по команде **Run** выполнение идет от одной точки до другой. Подробнее об этом будет рассказано в следующем пункте.

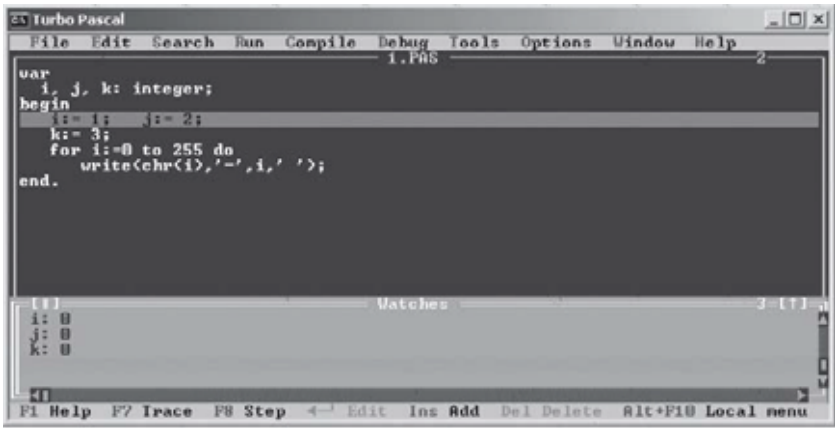


Рис. 17.5. Пошаговое выполнение программы. Состояние до выполнения присваивания значений переменным *i* и *j*

Второй возможный режим выполнения программы — выполнение пошаговое (построчное). Перед выполнением очередного шага соответствующая строка в программе выделяется бирюзовой полосой. Если в строке записаны несколько операторов, все они будут выполнены за один шаг (см. рис. 17.5, 17.6).

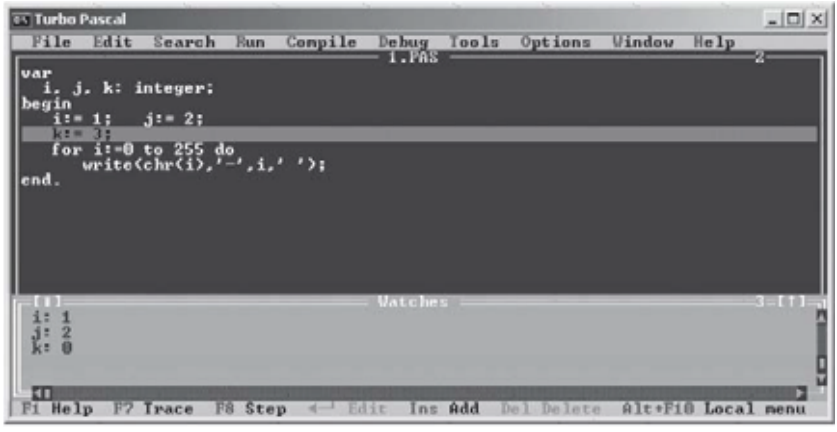


Рис. 17.6. Пошаговое выполнение программы. Состояние после выполнения присваивания значений переменным *i* и *j*. Присваивание значений обеим переменным выполнено за один шаг

Для выполнения шага могут использоваться функциональные клавиши **F8** или **F7** либо соответствующие команды **Step over** или **Trace into** из вертикального меню **Run**. Если выполняемая программа не имеет подпрограмм (процедур и функций), обе команды действуют совершенно одинаково: по нажатию горячей клавиши **F8** или **F7** выполнится одна строка. Указатель следующего шага (бирюзовая полоса) переместится на соответствующую строку текста программы.

Разница между командами проявляется при попытке выполнить вызов процедуры.

Команда **Step over** (горячая клавиша **F8**, буквальный перевод «Шагать через») рассматривает вызов подпрограммы как один оператор. Порядок выполнения тела подпрограммы ее не интересует. Вся подпрограмма будет выполнена за один шаг. По нажатию клавиши **F8** указатель следующего выполняемого шага (бирюзовая полоса) переместится на соответствующую строку текста программы.

Команда **Trace into** (горячая клавиша **F7**, буквальный перевод «Следить внутрь») выполняет заход внутрь тела вызываемой подпрограммы и ее пошаговое выполнение. По нажатию клавиши **F7** на экран будет выдано тело вызванной подпрограммы и указатель следующего выполняемого шага (бирюзовая полоса) переместится на ее первый оператор.

17.3. Контрольные точки

Пошаговое выполнение дает максимально подробную информацию о ходе вычислений. Но для больших программ оно может быть слишком медленно и утомительно. Часто нас интересует подробная информация о выполнении не всей программы, а только каких-то ее участков. Нам бы хотелось безостановочно дойти до интересующего нас участка программы, там остановить выполнение и заняться исследованием (просмотром значений переменных и стека подпрограмм, пошаговым выполнением интересного участка и т. п.). Затем опять быстро (безостановочно) пройти неинтересную часть программы до следующего интересного участка, где опять заняться исследованиями.

Приостанов выполнения программы с возможностью его возобновления реализуется в Турбо Паскале с помощью **контрольных точек** (точек приостанова, англ. breakpoint). Аппарат контрольных точек в Турбо Паскале весьма развит. Они могут быть условными (т. е. выполнение программы будет или не будет останавливаться в зависимости от значения некоторого

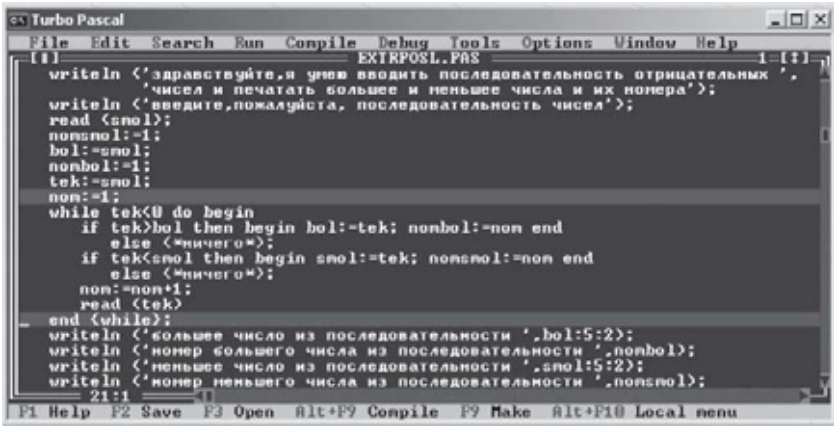


Рис. 17.7. Контрольные точки

условия). Они могут быть связаны со счетчиком (выполнение будет останавливаться не каждый раз, а только после заданного числа проходов). Они могут устанавливаться и сниматься динамически прямо во время выполнения программы. После контрольной точки выполнение может быть продолжено далее или начато опять с начала программы.

На экране контрольные точки изображаются красной полосой. Пример контрольных точек см. на рис. 17.7. Программа будет приостановлена в первый раз перед инициализацией переменной `nom`, а затем будет приостанавливаться каждый раз после выполнения тела цикла.

Добавление новой контрольной точки проводится с помощью команды **Add breakpoint...** в вертикальном меню **Debug** (существует горячая клавиша **Ctrl+F8**, но в вертикальном меню она не указана). Воспользоваться ей можно как до начала выполнения программы, так и во время выполнения (после выполнения очередного шага при пошаговом выполнении или в момент приостанова в контрольной точке). На рис. 17.8 показано назначение новой контрольной точки во время выполнения программы (бирюзовая полоса на заголовке цикла `while tek<0` указывает, что именно эта строка будет выполняться следующей).

Имя файла (поле **File name**) и номер строки (поле **Line number**), в которой ставится контрольная точка, могут быть введены вручную. По умолчанию берется текущая строка текущего файла. Поэтому перед тем, как дать команду **Add breakpoint...**, удобно поставить курсор на нужную строку.

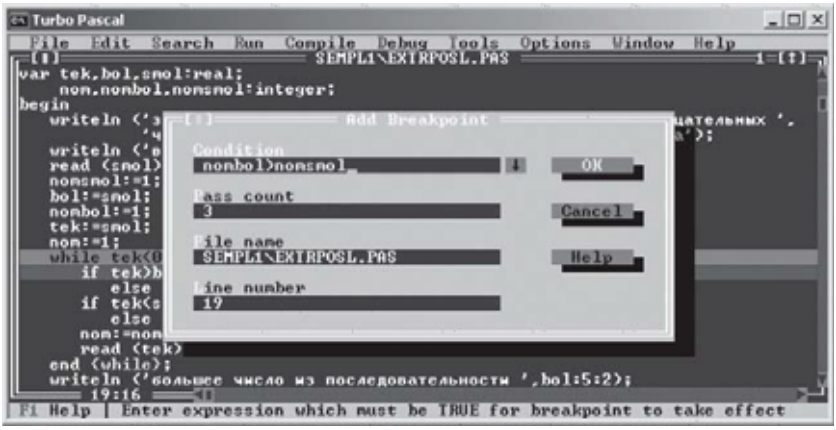


Рис. 17.8. Назначение новой контрольной точки (условной и со счетчиком)

Контрольная точка на рис. 17.8 назначается сразу с двумя дополнительными ограничениями, которые должна контролировать система. Во-первых, в поле **Condition** указано условие приостанова. Выполнение в данной контрольной точке будет приостановлено только в том случае, если истинно заданное в поле **Condition** логическое выражение. Во-вторых, в поле **Pass count** указан номер прохода, при котором должен происходить приостанов. В данной контрольной точке выполнение будет приостанавливаться через два раза на третий.

Если условие и счетчик заданы одновременно, то приостанов программы происходит только при одновременном выполнении обоих ограничений: условие должно быть истинно, а номер повтора кратен указанному счетчику.

Для быстрой установки/снятия контрольных точек можно воспользоваться горячей клавишей **Ctrl+F8.** Она ставит только простые контрольные точки (не позволяет задавать ограничения в виде условия или счетчика), зато работает как переключатель в обе стороны: и ставит контрольную точку, и снимает ее (по-английски это называется **Toggle breakpoint**). При необходимости ограничения (условие и/или счетчик) могут быть добавлены к описанию контрольной точки, поставленной с помощью **Ctrl+F8**, позже при просмотре списка установленных контрольных точек.

Просмотр описаний контрольных точек, их корректировка (например, изменение условия или счетчика) и удаление выполняются по команде **Breakpoints** меню **Debug.** По этой команде

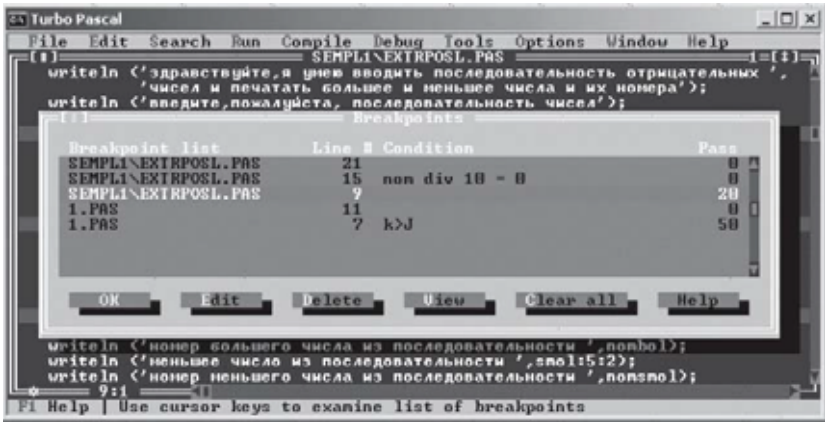


Рис. 17.9. Список контрольных точек

выдается список установленных в данный момент контрольных точек. Список выдается единый для всех открытых в данный момент файлов. Пример списка см. на рис. 17.9.

С помощью меню, расположенного в нижней части окна **Breakpoints**, можно удалить какую-то одну контрольную точку (позиция **Delete**) или сразу все (позиция **Clear all**). Можно перейти к просмотру файла и того его участка, где расположена указанная контрольная точка (позиция **View**). Можно закрыть окно (позиция **Ok**). Наконец, можно изменить описание контрольной точки (позиция **Edit**). Пример корректировки контрольной точки представлен на рис. 17.10. Кнопка **New** позволяет со-

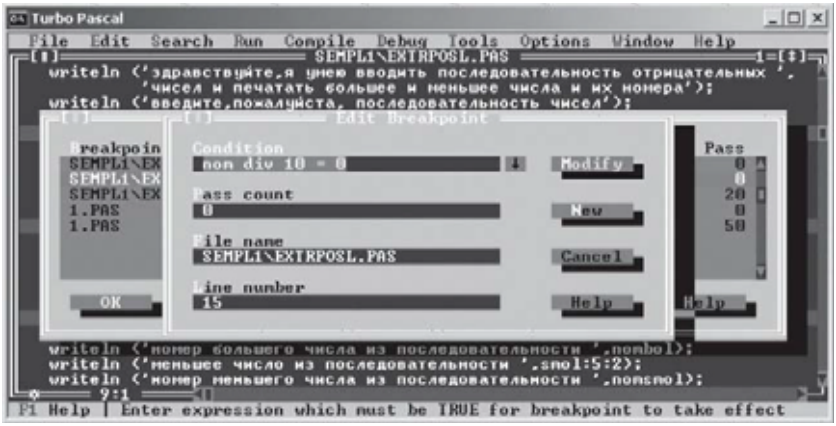


Рис. 17.10. Редактирование описания контрольной точки

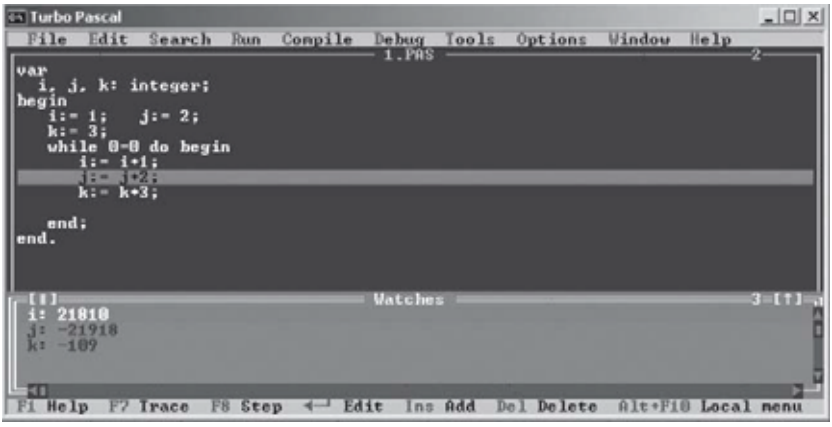


Рис. 17.11. Зацикленная программа, приостановленная с помощью комбинации **Ctrl+Break**

хранить результаты редактирования описания контрольной точки как новую контрольную точку, не затирая старую. Таким образом, упрощается создание новых контрольных точек на базе старых.

Если в момент после приостанова в контрольной точке дать команду выполнения **(Run, Step over, Trace into** или **Go to cursor)**, работа программы будет продолжена дальше с точки приостанова. Если нам это уже не нужно и мы бы хотели начать выполнение с начала программы, следует «переустановить» программу с помощью команды **Program reset** из вертикального меню **Run** (горячая клавиша **Ctrl+F2)**. После этого команда выполнения запустит программу сначала.

Возможна разовая быстрая установка одной контрольной точки без использования команд **Breakpoints** и **Add breakpoint.** Такая контрольная точка указывается курсором. Курсор ставится на то место программы, где должен произойти приостанов. Выполнение программы в этом случае запускается командой **Go to cursor** из вертикального меню **Run** (функциональная клавиша **F4)**. Команда **Go to cursor** позволяет очень просто и быстро указывать системе, в каком месте она должна приостановить выполнение программы.

Еще одна контрольная точка автоматически создается по нажатию комбинации клавиш **Ctrl+Break** во время выполнения программы. Это позволяет бороться с зацикливаниями. На рис. 17.11 показан пример приостанова специально зацикленной программы с помощью комбинации **Ctrl+Break.**

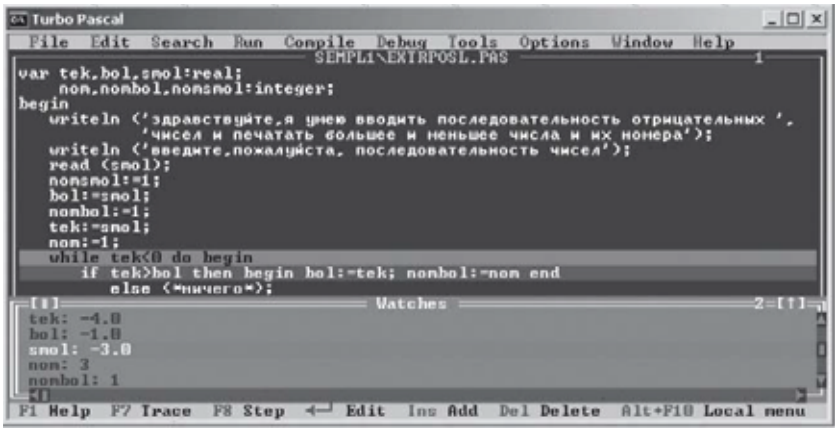


Рис. 17.12. Просмотр значений переменных в окне **Watches**

Кстати, на этом примере хорошо видны результаты целочисленного переполнения. В результате слишком длительного увеличения положительного числа мы получим отрицательное число, затем — опять положительное, затем — опять отрицательное и т. д.

17.4. Просмотр и вычисление значений переменных и выражений

Для просмотра значений переменных и выражений во время выполнения программы предназначено специальное окно **Watches** (см. рис. 17.5, 17.6, 17.11, 17.12). Открывается оно по команде **Watch** вертикального меню **Debug**.

В простейшем случае в окне **Watches** размещаются простые переменные. Но это могут быть и выражения (см. рис. 17.13), и структуры данных (см. рис. 17.14).

На рис. 17.14 интересно проследить за тем, какие значения получают неинициализированные поля записи (`a.x`) и элементов массива (`b[2].d` и `b[3].c`). Выше уже упоминалось, что Турбо Паскаль обнуляет память, отводимую под переменные главной программы. Но поскольку в данном случае мы имеем дело с переменными разных типов, то и нулевое заполнение памяти трактуется по-разному.

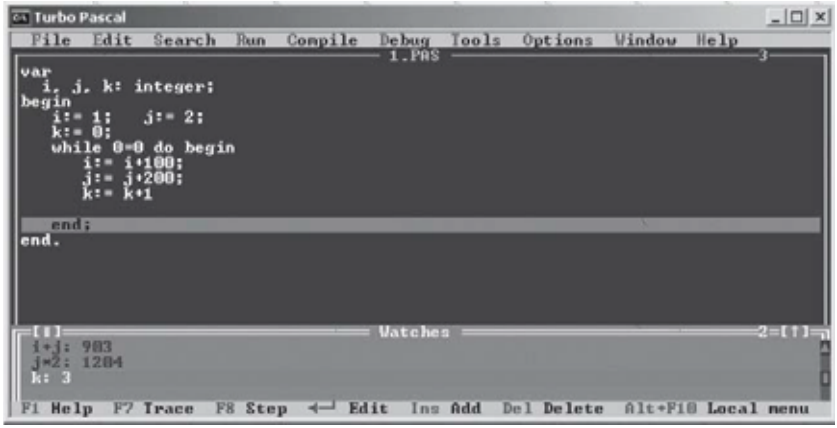


Рис. 17.13. Выражения в окне **Watches**

Для добавления в окно **Watches** новых объектов (переменных и выражений) предназначена команда **Add watch...** из вертикального меню **Debug** (горячая клавиша **Ctrl+F7**). По команде **Add watch...** открывается окно для ввода наблюдаемого выражения (см. рис. 17.15). По умолчанию в это окно помещается тот идентификатор (или то значение), на котором в данный момент стоит курсор. Поэтому перед нажатием комбинации **Ctrl+F7** бывает удобно указать курсором ту переменную, за значением которой вы хотите понаблюдать.

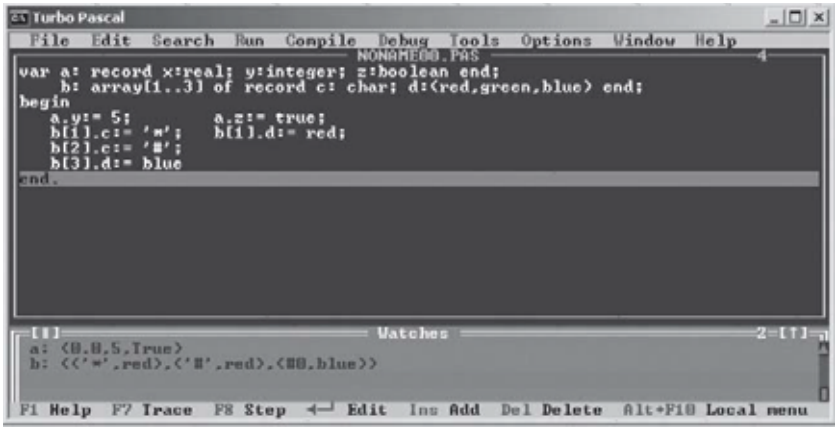


Рис. 17.14. Структуры данных в окне **Watches**

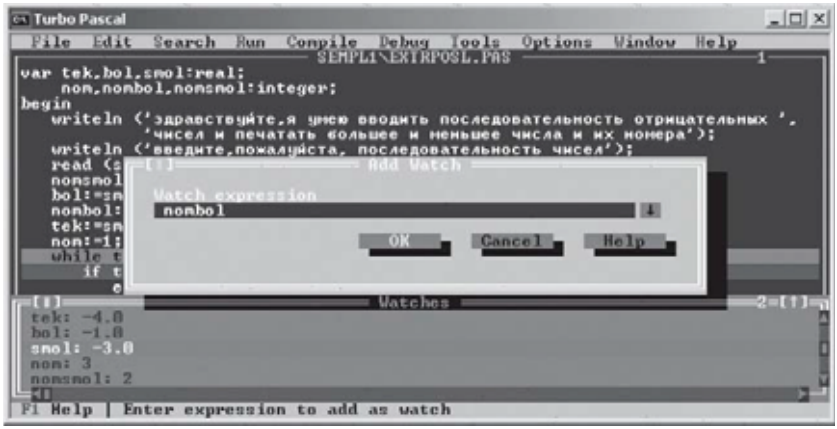


Рис. 17.15. Добавление переменной в окно **Watches**

Если вы находитесь в окне **Watches**, вы можете использовать для управления списком наблюдаемых объектов следующие клавиши: **Insert** — добавка в окно наблюдения нового объекта; **Delete** — удаление наблюдаемого объекта из окна **Watches**; **Enter** — ручное редактирование имени переменной или выражения.

Кроме того, в окне **Watches** (так же, как и в окне с текстом программы) существует специальное локальное меню. Вызывается оно по комбинации клавиш **Alt+F10** или по щелчку правой кнопки мыши (см. рис. 17.16). Кроме уже названных команд добавления, удаления и редактирования в локальном

Рис. 17.16. Локальное меню для управления окном **Watches**

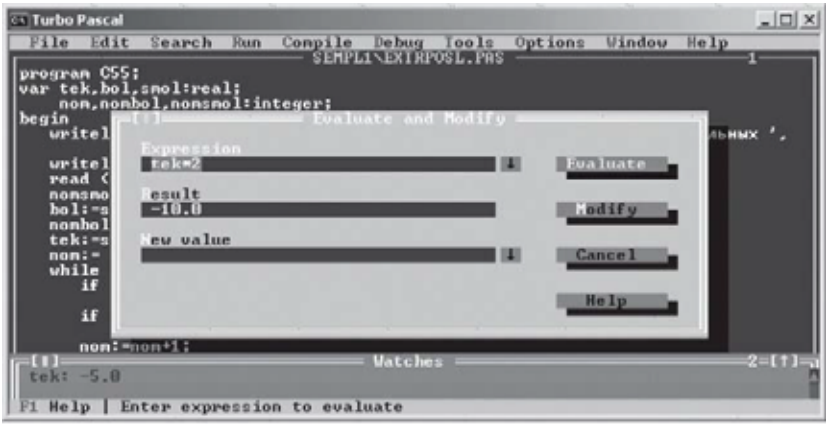


Рис. 17.17. Вычисление выражения командой **Evaluate/modify**

меню находится команда **Clear all** для удаления сразу всех наблюдаемых величин и пара команд **Enable/Disable**, которые позволяют временно отключить вычисление наблюдаемого выражения **(Disable)**, не удаляя его из окна наблюдений, а потом подключить его обратно **(Enable)**.

Значения переменных можно не только наблюдать, но и изменять. Прямо во время вычисления программы. Точнее, во время ее приостанова. Для этого служит команда **Evaluate/modify** из вертикального меню **Debug** (горячая клавиша **Ctrl+F4**). Примеры ее исполнения показаны на рис. 17.17 и 17.18.

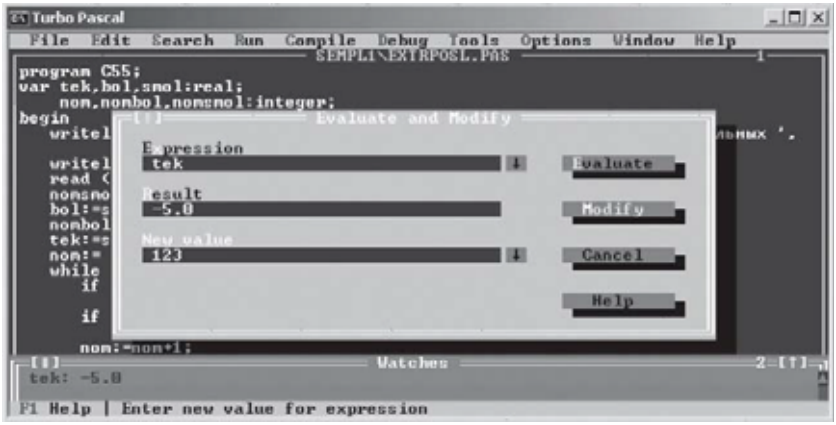


Рис. 17.18. Изменение значения переменной командой **Evaluate/modify**

В поле **Expression** вводится выражение. По нажатию кнопки **Evaluate** оно вычисляется, и его значение помещается в поле **Result**. Если в поле **Expression** стоит переменная, можно вести ее новое значение в поле **New value**. В случае нажатия кнопки **Modify** это значение будет присвоено переменной. После этого выполнение программы можно будет продолжить уже с измененной переменной.

Для учебных программ возможность произвольно менять значения переменных во время выполнения программы представляется излишней. Но ничего не поделаешь. Изначально Турбо Паскаль предназначался для производства программ, а не для обучения. Соответственно, возможность изменять значения переменных в приостановленной программе предназначалась для упрощения экспериментов с программой.

17.5. Наблюдение за стеком вызванных подпрограмм

Турбо Паскаль позволяет организовать просмотр стека вызванных в настоящий момент подпрограмм. Делается это с помощью команды **Call stack** вертикального меню **Debug** (горячая клавиша **Ctrl+F3**). По этой команде открывается специальное окно **Call stack**, в котором в стековом порядке выдается перечень исполняемых в данный момент подпрограмм и значения их параметров (вершина стека — вверху). Закрыть это окно можно, нажав клавишу **Enter**.

На рис. 17.19 демонстрируется просмотр стека вызовов при вычислении факториала строго по математическому определению. (Вычислять его таким образом на практике ни в коем случае не нужно!) Контрольная точка установлена на закрывающей скобке `end` тела функции. Запуск программы командой **Run** привел к ее выполнению до первой встречи с контрольной точкой. Произошло это при выходе из самого последнего вызова функции при вычислении 1!. Если теперь продолжить выполнение, то стек будет постепенно освобождаться. Из него будет вытолкнуто сначала `f(1)`, потом `f(2)` и т. д.

Среди параметров подпрограммы могут быть и выходные параметры, вычисляемые в теле подпрограммы. При слежении за такими параметрами надо иметь в виду, что локальные переменные подпрограмм Турбо Паскалем не инициализируются и при создании получают некоторые «мусорные» значения. Надо быть готовым, что начальные значения выходных параметров будут выглядеть совершенно неожидан-

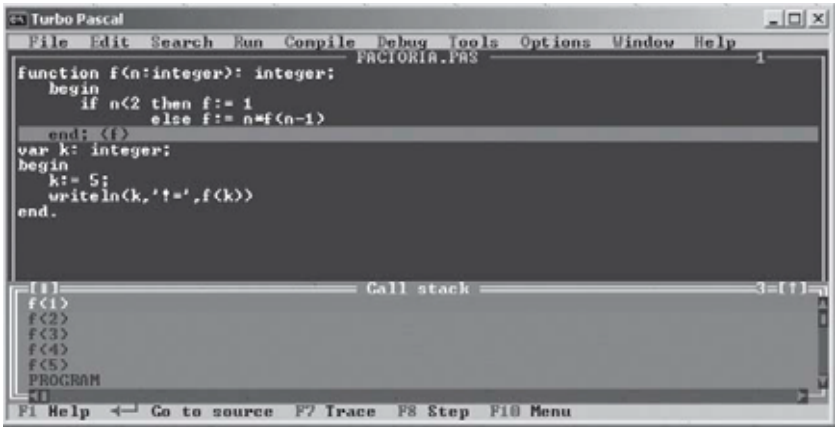


Рис. 17.19. Просмотр списка вызванных в данный момент подпрограмм

но. На рис. 17.20 приведенная выше функция для вычисления факториала переделана в процедуру (что уж точно не имеет никакого практического смысла!). Вычисляемый факториал возвращается через выходной параметр `f`. Состояние стека показано в момент возврата из того рекурсивного вызова, который вычислил 4!. В стеке еще лежат вызовы с входными параметрами 5, 6 и 7. При этом выходной параметр для

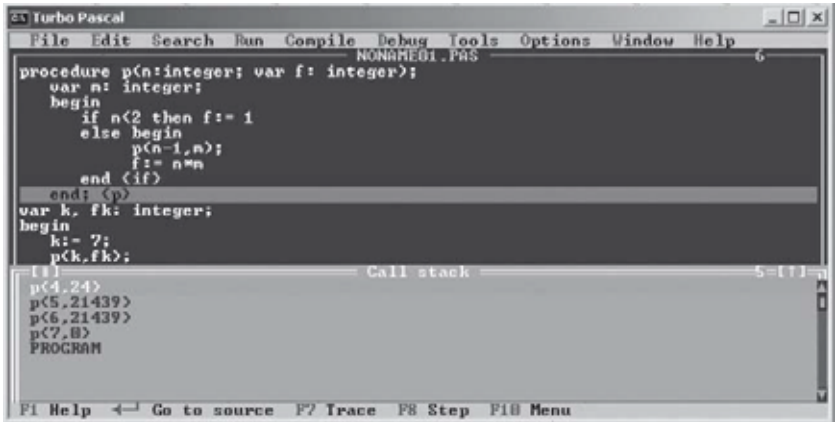


Рис. 17.20. Просмотр списка вызванных подпрограмм с выходными параметрами

вызовов 5 и 6 имеет «мусорное» значение. Причина в том, что соответствующие параметры являются локальными переменными процедуры, в отличие от параметра для вызова 7, который описан в главной программе[8].

17.6. Локальное меню окна редактирования программы

До сих пор мы говорили о доступе к отладочным средствам через горячие клавиши и вертикальные меню команд **Run** и **Debug**. Но есть еще один путь, который упоминался в начале данной главы. В окне редактирования текста программы существует собственное локальное меню (см. рис. 17.3 на стр. 95). Вызов его осуществляется комбинацией клавиш **Alt+F10** или правой кнопкой мыши. В этом меню дублируются отладочные команды **Toggle breakpoint (Ctrl+F8)**, **Go to cursor**, **Evaluate/modify**, **Add watch**.

[8] Еще раз повторим: использовать тот факт, что Турбо Паскаль обнуляет память главной программы, — это плохой стиль работы. Так же, как использование любой другой инициализации по умолчанию.

Глава 18

Еще один пример тестирования программы

Цель данной главы — повторение и закрепление изученного материала. Ничего нового здесь не будет. Поэтому старательному читателю предлагается сначала самому решить поставленную задачу (целиком или частично), а потом сверить свой результат с материалом данного раздела. Сверку тоже можно делать целиком или по частям.

Требуется: написать программу, которая вводит число N — длину числовой последовательности, а затем еще N чисел. Результат программы — сообщение о том, что в последовательности встречается раньше: минимальное положительное или максимальное отрицательное число.

18.1. Построение тестов для критериев черного ящика

По предыдущему опыту мы уже знаем, что начинать надо не с составления программы, а с составления тестов «черного ящика», и что первая цель этих тестов — понять поставленную задачу. Перебираем критерии черного ящика.

Программа однофункциональна. Значит, критерий покрытия функций будет выполнен гарантированно. Что касается остальных критериев — покрытия классов входных данных, классов выходных данных, области допустимых значений — специфика задачи позволяет сразу сосредоточиться на двух частных критериях: тестирование длины набора данных и тестирование упорядоченности набора данных.

1. Тестирование длины начнем с пустого набора. Как его задать и какова должна быть реакция программы? Для задания пустого набора, очевидно, надо ввести $N = 0$. Требуемую реакцию программы на такой ввод надо уточнить у заказчика. Пусть его ответ будет тот же, что и в примере главы 11: «Последовательность пуста». Имеем первый тест:

T1: Вход: 0.

Ожидаемый выход: Последовательность пуста.

2. Далее канон предлагает нам ввести единичный набор данных. Задать его достаточно просто. $N = 1$, а затем еще одно число. Но каков должен быть ответ программы? Ведь нам надо сравнить номера двух элементов последовательности, а элемент всего один. (Вообще говоря, выявляется еще одна проблема. Нам нужны не просто два числа, а два числа разных знаков. Эту проблему мы себе заприметим, но пока отложим на будущее. Давайте сначала разберемся с длиной последовательности.) Как быть в этом случае, решить может только заказчик. Спросим у него. Решено, что программа должна выдать одно из двух сообщений: «Положительных чисел нет» или «Отрицательных чисел нет».

Заметим, что заказчик сам заговорил о проблеме разнознаковости чисел в последовательности, предложил анализировать эту ситуацию и выдавать два разных выходных сообщения. Значит, у нас оказывается 2 класса выходных данных. Для их покрытия понадобятся, по крайней мере, 2 теста.

T2: Вход: 1, 5.

Ожидаемый выход: Отрицательных чисел нет.

T3: Вход: 1, −5.

Ожидаемый выход: Положительных чисел нет.

3. Следующий тест должен проверять работу со слишком коротким набором данных. Мы это уже сделали. Для нас таким набором стал единичный набор.

4. Проверяем набор минимально возможной длины. Поскольку никаких ограничений на длину последовательности у нас нет, минимальным будем считать единичный набор. Но этот случай мы уже проверили.

5. Нормальный набор. Ввод нескольких чисел, среди которых есть и положительные, и отрицательные. Стоп! А, собственно, почему мы в этом так уверены? Кто сказал, что в последовательности обязательно должны быть и положительные, и отрицательные числа? Только что у нас была последовательность единичной длины, в которой либо положительных, либо отрицательных чисел не было вовсе. Почему не может быть последовательности из нескольких чисел, которые все будут либо положительными, либо отрицательными? Вообще говоря, наша входная последовательность делится на две подпоследовательности[9], каждая из которых имеет право быть пустой.

---

[9] Один умный человек заметил как-то: «Все знают, что значит “будем последовательны”. А что значит “будем подпоследовательны”?».

Возникают следующие тесты:

T4: Вход: 3, 1, 2, 3.

Ожидаемый выход: Отрицательных чисел нет.

T5: Вход: 2, −1, −2.

Ожидаемый выход: Положительных чисел нет.

Теперь проверим случай, когда есть и те, и другие:

T6: Вход: 5, −1, 2, −3, 4, 7.

Ожидаемый выход: Максимальное отрицательное стоит перед минимальным положительным.

Но возможен и другой исход сравнения (есть еще один класс выходных данных). Поэтому добавим:

T7: Вход: 4, 2, −4, −1, 7.

Ожидаемый выход: Минимальное положительное стоит перед максимальным отрицательным.

Кажется, что тестирование длины набора данных завершено. Но навевает некоторые сомнения формулировка, которая у нас только что появилась, когда мы рассуждали про подпоследовательности положительных и отрицательных чисел. Во-первых, оказывается, входные данные у нас организованы более сложно, чем нам показалось сначала. Мы решили, что на входе у нас одна последовательность чисел, а оказывается, что там две подпоследовательности. То, что их элементы могут идти в произвольном порядке, только усложняет дело. И для каждой из подпоследовательностей надо бы также проверить все, что касается ее длины. Как поведет себя программа, если уже не вся последовательность, а та или иная подпоследовательность будет иметь длину нулевую, единичную или нормальную? Слишком короткие и слишком длинные подпоследовательности можно не проверять (слишком короткие сводятся к нулевым, а сверху длина не ограничена). А вот подпоследовательности, состоящие из нескольких частей, проверить стоит. Должны ли все положительные (отрицательные) числа идти подряд или могут быть перемешаны?

Во-вторых, мы только что сказали, что каждая из подпоследовательностей может быть пустой. А могут ли быть пусты сразу обе подпоследовательности? Почему нет? Ведь кроме чисел положительных и отрицательных у нас есть нуль. Значит, мы можем ввести непустую последовательность, состоящую из

N нулей. И что в этом случае должна делать наша программа? Не стоит придумывать ответ самим. Гораздо разумней спросить об этом заказчика. (Не потому, что мы не в состоянии сами придумать ответ на этот вопрос, а потому, что мы не должны его придумывать. Не наше мнение в данном случае важно, а мнение заказчика.) Ответ заказчика: «Раньше программа сообщала об отсутствии положительных или отрицательных чисел. Теперь пусть сообщит об отсутствии и тех, и других».

Теперь от рассуждений перейдем к конструированию тестов. Судя по длине рассуждений, потрудиться нам предстоит изрядно. Но, взглянув на уже имеющийся набор тестов, мы обнаружим, что дело не так уж плохо. Ситуацию с нулевой длиной подпоследовательностей мы уже проверили тестами T2, T3, T4, T5. Ситуацию с неединичной длиной — тестами T6, T7. Остается проверить только единичные подпоследовательности.

Формально у нас уже были единичные подпоследовательности в тестах T2 и T3. Но считать, что эти тесты проверяют правильность определения экстремума в наборе из одного числа, нельзя. Из их результатов вообще не ясно, был поиск экстремума или нет. У этих тестов другая цель. Исходя из этих соображений, добавим еще два теста:

T8: Вход: 3, −2, 4, −3.

Ожидаемый выход: Максимальное отрицательное стоит перед минимальным положительным.

T9: Вход: 4, 1, 7, 9, −4.

Ожидаемый выход: Минимальное положительное стоит перед максимальным отрицательным.

(Можно было бы обойтись одним тестом (например: 2, 2, −2), но какой-то он подозрительный.)

Тесты T6 и T8 проверили «перемешанные» подпоследовательности, тест T7 — подряд идущую отрицательную подпоследовательность, тест T9 — подряд идущую положительную.

Осталось проверить случай, когда обе подпоследовательности имеют нулевую длину, а последовательность в целом не пуста. Для этого добавим еще один тест. Вход его очевиден, а вот выход... Должны ли мы использовать уже имеющиеся фразы или выдать какую-то новую? Лучше новую, так будет точнее. Получим:

T10: Вход: 3, 0, 0, 0.

Ожидаемый выход: Последовательность состоит из одних нулей.

Для того чтобы упорядочить результаты проделанной работы, свяжем предложенные тесты и проверяемые ситуации с помощью таблицы:

Проверяемая ситуация	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Характеристики длины набора и поднаборов										
Длина всей последовательности равна нулю	+									
Длина всей последовательности равна 1		+	+							
Длина всей последовательности больше единицы				+	+	+	+	+	+	+
Длина положительной подпоследовательности равна нулю, а отрицательной — не равна			+		+					
Длина положительной подпоследовательности равна 1								+		
Длина положительной подпоследовательности больше единицы						+	+		+	
Длина отрицательной подпоследовательности равна нулю, а положительной — не равна		+		+						
Длина отрицательной подпоследовательности равна 1									+	
Длина отрицательной подпоследовательности больше единицы						+	+	+		
Сразу обе подпоследовательности имеют нулевую длину										+
Элементы подпоследовательностей перемешаны						+		+		
Положительная подпоследовательность идет подряд									+	
Отрицательная подпоследовательность идет подряд							+			
Классы выходных данных										
Последовательность пуста	+									
Положительных чисел нет			+		+					
Отрицательных чисел нет		+		+						
Максимальное отрицательное стоит перед минимальным положительным						+		+		
Минимальное положительное стоит перед максимальным отрицательным							+		+	
Последовательность состоит из одних нулей										+

Теперь перейдем к следующему критерию — тестированию упорядоченности набора данных. Собственно, нас интересует не порядок чисел, а экстремумы. Значит, нам необходимо проверить следующие ситуации:

1) экстремальное значение находится в середине набора;
2) экстремальное значение находится в начале набора;
3) экстремальное значение находится в конце набора;
4) в наборе несколько совпадающих экстремальных значений.

Эти ситуации должны быть проверены как для положительной, так и для отрицательной подпоследовательности. Всего получается 8 случаев. Посмотрим, какие из них закрыты уже существующими тестами.

Проверяемая ситуация	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Характеристика экстремума										
Минимальное положительное в середине подпоследовательности										
Минимальное положительное в начале подпоследовательности						+	+		+	
Минимальное положительное в конце подпоследовательности										
В положительной подпоследовательности несколько минимумов										
Максимальное отрицательное в середине подпоследовательности										
Максимальное отрицательное в начале подпоследовательности						+		+		
Максимальное отрицательное в конце подпоследовательности							+			
В отрицательной подпоследовательности несколько максимумов										

Можно попробовать переработать некоторые старые тесты для того, чтобы закрыть большее количество строк. Но это потребует повторного анализа предыдущих критериев. Чтобы не делать этого, проще добавить новые тесты.

В тесте T11 минимальное положительное в середине подпоследовательности, максимальное отрицательное в середине подпоследовательности. Кроме того, длина всей последовательности больше 1, длина каждой подпоследовательности больше 1, обе подпоследовательности идут подряд.

T11: Вход: 6, 5, 2, 7, –6, –3, –8.

Ожидаемый выход: Минимальное положительное стоит перед максимальным отрицательным.

В тесте T12 минимальное положительное в конце подпоследовательности, в отрицательной подпоследовательности несколько максимумов. Кроме того, длина всей последовательности больше 1, длина каждой подпоследовательности больше 1, обе подпоследовательности идут подряд.

T12: Вход: 7, –6, –3, –8, –3, 5, 7, 2.

Ожидаемый выход: Максимальное отрицательное стоит перед минимальным положительным.

В тесте T13 в положительной подпоследовательности несколько минимумов, в отрицательной подпоследовательности несколько максимумов. Кроме того, длина всей последовательности больше 1, длина каждой подпоследовательности больше 1, элементы подпоследовательностей перемешаны.

T13: Вход: 8, –6, 2, –8, –3, 5, 7, 2, –3.

Ожидаемый выход: Стоп!

Какой, собственно говоря, выход должен быть в данной ситуации? Элементы подпоследовательностей перемешаны. В каждой из них несколько экстремумов. Некоторые из положительных минимумов стоят перед некоторыми отрицательными максимумами. Некоторые из положительных минимумов стоят после некоторых отрицательных максимумов. Какой результат должна выдать программа в этом случае? Ответить на этот вопрос может только заказчик. Оказывается, мы до сих пор не до конца понимаем, какую же задачу мы решаем! Но лучше поздно, чем никогда. Итак, обращаемся к заказчику. Его ответ: «В каждой подпоследовательности надо брать последний экстремум». В тесте T13 это окажутся два последних числа: последняя двойка как минимальное положительное и последнее –3 как максимальное отрицательное. Теперь мы можем записать тест.

T13: Вход: 8, –6, 2, –8, –3, 5, 7, 2, –3.

Ожидаемый выход: Минимальное положительное стоит перед максимальным отрицательным.

Здесь следует сделать важное, но печальное замечание. Надо признать, что на необходимость последнего уточнения задачи мы натолкнулись совершенно случайно. Не было никакого формального повода совмещать три требования разных критериев: наличие нескольких экстремумов в каждой из последовательностей и перемешивание элементов подпоследовательностей. Более того, даже совмещение в одном тесте всех этих требований не гарантировало того, что проблемная ситуация будет обнаружена. Вполне могло оказаться, что даже при перемешивании элементов разных подпоследовательностей все положительные минимумы оказались бы раньше или позже всех отрицательных максимумов. Ведь для обнаружения данной проблемной ситуации совсем необязательно иметь по нескольку экстремумов в каждой из подпоследовательностей. Достаточно нескольких экстремумов хотя бы в одной подпоследовательности. Например, так, как это сделано в тесте T12. Но в тесте T12 мы

разместили все отрицательные числа перед всеми положительными. В результате коллизии просто не возникло. Увы! Приходится признать, что мы имели реальный шанс взяться за написание программы, не совсем понимая, что она должна делать.

Запишем сводную таблицу используемых критериев черного ящика полноты тестирования и покрывающих их тестов.

Таблица критериев черного ящика

Проверяемая ситуация	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
Характеристики длины набора и поднаборов													
Длина всей последовательности равна нулю	+												
Длина всей последовательности равна 1		+	+										
Длина всей последовательности больше единицы				+	+	+	+	+	+	+	+	+	+
Длина положительной подпоследовательности равна нулю, а отрицательной — не равна			+		+								
Длина положительной подпоследовательности равна 1								+					
Длина положительной подпоследовательности больше единицы						+	+		+		+	+	+
Длина отрицательной подпоследовательности равна нулю, а положительной — не равна		+		+									
Длина отрицательной подпоследовательности равна 1									+				
Длина отрицательной подпоследовательности больше единицы						+	+	+			+	+	+
Сразу обе подпоследовательности имеют нулевую длину										+			
Элементы подпоследовательностей перемешаны						+		+					+
Положительная подпоследовательность идет подряд									+		+	+	
Отрицательная подпоследовательность идет подряд							+				+	+	

Проверяемая ситуация	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
Классы выходных данных													
Последовательность пуста	+												
Положительных чисел нет			+		+								
Отрицательных чисел нет		+		+									
Максимальное отрицательное стоит перед минимальным положительным						+		+				+	
Минимальное положительное стоит перед максимальным отрицательным							+		+		+		+
Последовательность состоит из одних нулей										+			
Характеристика экстремума													
Минимальное положительное в середине подпоследовательности											+		
Минимальное положительное в начале подпоследовательности						+	+		+				
Минимальное положительное в конце подпоследовательности												+	
В положительной подпоследовательности несколько минимумов													+
Максимальное отрицательное в середине подпоследовательности											+		
Максимальное отрицательное в начале подпоследовательности						+		+					
Максимальное отрицательное в конце подпоследовательности							+						
В отрицательной подпоследовательности несколько максимумов												+	+

Предварительная подготовка тестов — с точки зрения критериев черного ящика — закончена. Можно переходить к написанию текста программы. Заметим, что разработка этих тестов существенно улучшила наше понимание решаемой задачи.

18.2. Написание текста программы

Идея программы:

1. Вводим последовательно числа.
2. Отделяем положительные от отрицательных.
3. Для положительных проверяем, не является ли очередное число минимальным. Если является, запоминаем его номер.

4. Для отрицательных проверяем, не является ли очередное число максимальным. Если является, запоминаем его номер.
5. По окончании последовательности сравниваем номера минимального положительного и максимального отрицательного элементов.

На верхнем уровне алгоритм будет выглядеть так:

Текст на псевдокоде	Примечание

begin
  выдать начальное приветствие;
  инициализировать данные;
  ввести длину последовательности (n);
  for k:= 1 to n do begin
  ввести очередной элемент (tek);
  if tek>0 then
  обработать положительный элемент
  else обработать отрицательный элемент;
  end; {do}
  выдать результат
end.
	пока не понятно, какие
	var n: integer;
	var k: integer;
	var tek: integer;

Уточняем недоопределенные фрагменты:	

Обработать положительный элемент
 if tek<MinPol then begin
  MinPol:= tek;
  NomMinPol:= k
 end
	var MinPol: integer;
	var NomMinPol: integer;

Обработать отрицательный элемент
 if tek>MaxOtr then begin
  MaxOtr:= tek;
  NomMaxOtr:= k
 end
	var MaxOtr: integer;
	var NomMaxOtr: integer;

При написании программы не забудем шишки, которые мы уже успели набить на первом примере. По большому счету, их было две. Во-первых, мы хорошо запомнили, что все данные должны быть инициализированы. Во-вторых, чтобы не зациклиться, надо, как минимум, менять хотя бы одну переменную в условии цикла.

Разберемся с инициализацией. У нас 7 переменных: n, k, tek, MinPol, NomMinPol, MaxOtr, NomMaxOtr. Какие из них надо инициализировать, какие нет? Первая группа: n, k, tek. Значение переменной n будет введено сразу после начала выполнения программы. Собственно, это первое содержательное действие нашей программы. Переменная k — счетчик цикла, она получит значение в операторе for. Значение переменной tek будет введено. Значит, эти три переменные в инициализации не нуждаются.

Далее появляются две пары переменных: MinPol, NomMinPol и MaxOtr, NomMaxOtr. Каждая пара должна рассматриваться не изолированно, а именно как пара. Смысл этих переменных: минимальное положительное значение и его номер в последовательности, максимальное отрицательное значение и его номер в последовательности. Возникали они у нас на уровне уточнения текста программы, при раскрытии недоопределенных фрагментов верхнего уровня. Две из них — MinPol и MaxOtr — возникли сразу в вычислениях как переменные, уже имеющие осмысленные значения. Значит, их инициализировать нужно обязательно. А поскольку NomMinPol и NomMaxOtr с ними неразрывно связаны, лучше инициализировать сразу все четыре. Чем? Будем исходить из смысла этих переменных. Что такое MinPol и MaxOtr? Минимальное положительное и максимальное отрицательное значения. В момент инициализации у нас еще нет ни того, ни другого. Значит, при инициализации переменные должны получить значения, по которым было бы видно, что эти значения «не настоящие», получены не в ходе содержательных вычислений. Очевидно, что в качестве таких значений удобно использовать нуль. Он не является ни положительным, ни отрицательным. Поэтому не может быть «настоящим» значением ни для MinPol, ни для MaxOtr. Аналогично можно рассуждать про NomMinPol и NomMaxOtr. В момент инициализации отсутствуют минимальное положительное и максимальное отрицательное, стало быть, отсутствуют и их номера. Значит, NomMinPol и NomMaxOtr надо инициализировать значениями, которые не могут быть «настоящими» номерами. И опять в качестве такого значения удобно взять нуль. Значит, эти четыре переменные инициализировать надо, и в качестве начального значения всем им присвоим нули.

Что касается цикла, то легко заметить, что условия у нас изменились. Раньше мы не знали заранее, сколько раз должен повторяться цикл. Это выяснялось только в ходе его выполнения. Поэтому необходимо было использовать либо цикл while,

либо цикл repeat, в зависимости от минимального количества повторов цикла. Сейчас мы знаем еще до входа в цикл, что он должен отработать n раз. Значит, вместо цикла с предусловием нам удобней воспользоваться циклом со счетчиком: for k:= 1 to n do. Правда, у нас возможна пустая последовательность (n=0), но это не страшно. В Паскале минимальное количество повторений цикла со счетчиком равно нулю.

Первый вариант программы будет иметь вид:

```
program PolOtrMinMax;
var n, k, tek: integer;
    MinPol, NomMinPol: integer;
    MaxOtr, NomMaxOtr: integer;
begin
 {выдать начальное приветствие}
  writeln('Функция программы:');
  writeln(' ввести кол-во чисел в последовательности,');
  writeln(' ввести указанное число чисел,');
  writeln(' сообщить, что идет раньше: минимальное');
  writeln(' положительное или максимальное отрицательное.');
 {инициализировать данные}
  MinPol:= 0;   NomMinPol:= 0;
  MaxOtr:= 0;   NomMaxOtr:= 0;
 {ввести длину последовательности}
  writeln('Введите кол-во элементов последовательности');
  read(n);
 for k:= 1 to n do begin
   {ввести очередной элемент}
     writeln('Введите элемент №',k);
     read(tek);
   if tek>0 then
     {обработать положительный элемент}
       if tek<MinPol then begin
         MinPol:= tek;
         NomMinPol:= k
       end
   else
     {обработать отрицательный элемент}
       if tek>MaxOtr then begin
         MaxOtr:= tek;
         NomMaxOtr:= k
       end;
 end; {do}
 {выдать результат}
  if n=0 then writeln('Последовательность пуста')
```

```
   else if NomMinPol=0 then
              writeln('Положительных чисел нет')
   else if NomMaxOtr=0 then
              writeln('Отрицательных чисел нет')
   else if NomMinPol=0 and NomMaxOtr=0 then
        writeln('Последовательность состоит из одних нулей')
   else if NomMaxOtr<NomMinPol then
        writeln('Максимальное отрицательное стоит перед '+
                'минимальным положительным')
   else writeln('Минимальное положительное стоит перед '+
                'максимальным отрицательным')
end.
```

18.3. Подготовка к тестированию по критериям белого ящика

Строим таблицу МГТ:

			Т1	Т2	Т3	Т4	Т5	Т6	Т7	Т8	Т9	Т10	Т11	Т12	Т13
У1	for k:=1 to n	=0													
		=1													
		>1													
У2	if tek>0	+													
		—													
У3	if tek<MinPol	+													
		—													
У4	if tek>MaxOtr	+													
		—													
У5	if n=0	+													
		—													
У6	if NomMinPol=0	+													
		—													
У7	if NomMaxOtr=0	+													
		—													
У8	if NomMinPol=0 and NomMaxOtr=0	+													
		—													
У9	NomMinPol=0	+													
		—													
У10	NomMaxOtr=0	+													
		—													
У11	if NomMaxOtr< NomMinPol	+													
		—													

18.4. «Сухая прокрутка»

Приступим к прокрутке. Для теста T1 трассировочная таблица будет предельно проста:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
0	1					

Результат: Последовательность пуста.

Отметим результаты в таблице тестов:

Тест	**Вход**	**Ожидаемый результат**	**Результат «сухой прокрутки»**	+/–	**Результат выполнения на компьютере**	+/–
T1	0	Последовательность пуста	Последовательность пуста	+		
...	...	...				

Тест T1 оказался неудачен. Перейдем к тесту T2. Трассировка будет иметь вид:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
1	1	5				

Результат: Положительных чисел нет.

То есть как? Мы только что ввели положительное число 5, а программа заявляет, что положительных чисел нет. В чем дело? Для поиска ошибки применим ретроанализ.

Программа выдала сообщение «Положительных чисел нет». Такое сообщение может быть выдано только из одной точки программы: из оператора

```
if NomMinPol=0 then writeln('Положительных чисел нет')
```

Значит, при выполнении этого оператора условие было истинно, т. е. переменная `NomMinPol` = 0. Почему переменная `NomMinPol` имеет значение 0? Нулевое значение переменная получила при инициализации. И это было правильно. В момент инициализации не введено ни одного элемента последователь-

ности. Значит, нет и минимального положительного. А значит, нет и его номера. Признаком отсутствия номера удобно сделать нулевой номер. Но это все говорится о моменте инициализации. Сравнение `if NomMinPol=0` выполняется в конце программы. К этому времени мы уже ввели положительный элемент последовательности. Значит, нулевое значение `NomMinPol` уже должно быть заменено. Сделать это должен был фрагмент:

```
MinPol:= tek;
NomMinPol:= k
```

Не сделал. Значит, этот фрагмент не выполнялся. Почему? Он «скрыт» двумя условиями:

```
if tek>0 then
   if tek<MinPol then
```

и выполняется только в том случае, если оба этих условия истинны. В нашем случае `tek=5`, значит, условие `tek>0` — истинно. Через первое условие проходим. Проверяем второе: `tek=5`, `MinPol=0`, `tek<MinPol` — ложь. Значит, ветвь `then` выполняться не будет, переменные `MinPol` и `NomMinPol` не изменятся. Ошибка: они должны измениться!

Почему условие `tek<MinPol` оказалось ложным? Переменная `tek` имеет свое «законное» значение, она равна текущему элементу последовательности. Неверное значение имеет переменная `MinPol`. Почему она равна нулю? Нулевое значение `MinPol` получила при инициализации, оно означает, что переменная еще не имеет никакого осмысленного значения. Такое решение казалось логичным. Значение `MinPol` (минимального положительного) по смыслу может быть только положительным. Нуль таковым не является, поэтому его и использовали как «пустышку». Решение оказалось неудачным. В тексте программы переменная `MinPol` получает новое значение в том случае, если она больше текущего положительного элемента последовательности. Но для нуля это условие не выполнится никогда. Если мы хотим, чтобы оно выполнилось, начальное значение `MinPol` должно быть заведомо больше (во всяком случае, не меньше) любого возможного положительного элемента последовательности. Значит, надо изменить начальное значение `MinPol`, которое она получает при инициализации. Инициализация должна выглядеть так:

```
MinPol:= Максимальное возможное положительное;
NomMinPol:= 0;
```

Есть ли у нас значение, которое больше любого другого положительного числа? Математика ответит: «Нет»[10]. Но мы-то сейчас занимаемся не математикой, а программированием. Мы уже говорили про ограниченность представимых в машине чисел. В Паскале даже есть встроенная константа `maxint`, выдающая максимально допустимое целое число. В стандартном Паскале можно было бы использовать ее. Но в Турбо Паскале существует несколько целых типов, в том числе тип `LongInt`, диапазон значений которого во много раз больше, чем у `integer`. На какой из них ориентироваться?

Ответ на этот вопрос можно получить только из постановки задачи. Но там он не очевиден. Что надо делать в этом случае, мы уже знаем. Не надо ничего придумывать самим. Надо задать вопрос заказчику. Придется побеспокоить заказчика еще раз.

Диалог:

Мы: Извините, что перекладываем на Вас некоторые наши *компьютерные* проблемы. Но нам нужна Ваша помощь для принятия некоторых решений. Нет ли в задаче ограничений на максимальное значение положительных чисел? Это наши внутримашинные заморочки, но не устроит ли Вас максимальное значение в 32 767? Или в 2 147 483 647? Или этого тоже мало?

Заказчик: Я как-то не думал на эту тему. Вообще-то, по условию задачи никаких ограничений нет. Хотя я понимаю, что машина не резиновая и туда не все может поместиться. Но Вы говорите про ограничения на **целые** числа. А для **дробных** они действовать не будут? Тогда давайте к большим целым приписывать нулевую дробную часть. Это, конечно, лишние хлопоты при вводе. Но не такие большие: запятая и нуль. Я готов на них согласиться. Начиная с какого значения, надо будет приписывать дробную часть?

Мы: (*Непонимающе*) Погодите! Какую дробную часть?

Заказчик: (*Непонимающе*) Как какую? После целой части ставится запятая, а потом записывается дробная часть числа. Ах, извините, я слышал, что компьютерщики пишут на англо-американский манер: вместо запятой, как в Европе, ставят точку, как в Англии. Хорошо, пусть будет точка, Бог с ней.

---

[10] Как говорили древние греки: «Куда бы ни стал воин, он может протянуть свое копье еще дальше». Заметим, что это не единственный возможный взгляд на числа. Древние индусы строили свою математику, исходя из предположения о существовании «числа Будды», наибольшего среди всех чисел.

Мы: Так значит, у чисел может быть дробная часть?

Заказчик: Ну, разумеется.

Мы: А почему Вы нам об этом раньше не говорили?

Заказчик: То есть как не говорил? Я везде говорил про **числа**. А разве дробные числа — не числа? Разве я когда-то говорил, что числа будут только целые?

Мы: ...!!!...

(*Немая сцена*)

Конец диалога.

Вот это да! Действительно. Нигде в постановке задачи слово «целые» не употреблялось. Мы его сами домыслили. И даже не потрудились уточнить у заказчика. Для нас же было очевидно, что числа — целые. Еще понять бы, почему это стало для нас очевидно?

Урок на будущее! Решать надо строго ту задачу, которую ставит заказчик. А не ту, которую мы придумали «по мотивам» задачи, поставленной заказчиком. Если приходится что-то додумывать самим, то все (все!!!) уточнения необходимо согласовывать с заказчиком[11].

Интересно, как нас угораздило до сих пор не обнаружить ошибочность своего предположения? Дело в том, что в задаче не используются какие-либо специфические свойства, которыми обладают вещественные числа, но не обладают целые. Требуется только отделить положительные числа от отрицательных и сравнить, какое из двух чисел больше, какое — меньше. А эти свойства одинаковы как для целых, так и для вещественных чисел.

Хорошо, что хоть и случайно, но ошибку обнаружили. И обойдется она нам достаточно дешево. Понадобится всего лишь изменить описания переменных.

```
var n, k: integer;
    tek, MinPol, MaxOtr: real;
    NomMinPol, NomMaxOtr: integer;
```

В теле программы никаких изменений не потребуется.

---

[11] Очень характерно наше возмущение: а почему заказчик нам не указал явно, что числа должны быть вещественные? А ему и в голову не пришло, что для нас это важно! Он же — не компьютерщик. И понятия не имеет про то, что мир можно воспринимать в терминах `integer/real`. Более того, он, похоже, и не математик. Во всяком случае, слов «вещественные числа» он, похоже, не знает, предпочитая именовать их «дробными».

Все тесты, которые были сконструированы для проверки той или иной ситуации, можно использовать. Поскольку целые числа в математике — частный случай вещественных, обобщение целого числа до вещественного (в момент ввода) Паскаль выполнит сам. Но, естественно, нужны тесты, содержащие именно вещественные числа. Чтобы не увеличивать общее число тестов, подправим немного некоторые из уже имеющихся тестов. Причем так, чтобы на входе были числа вещественные, а результат прогона не изменился. Например, так:

T2: Вход: 1, 5.6.

Выход: Отрицательных чисел нет.

T6: Вход: 5, −1.3, 2.4, −3.1, 4, 7.03.

Выход: Максимальное отрицательное стоит перед минимальным положительным.

T7: Вход: 4, 2.1, −4.2, −1.3, 7.4.

Выход: Минимальное положительное стоит перед максимальным отрицательным.

С этой внезапно всплывшей проблемой, кажется, разобрались. Теперь вернемся к обнаруженной нами ошибке и к поиску способов ее исправления. Как переход к вещественным числам скажется на решении возникшей перед нами проблемы — определения минимального положительного значения? Скажется плохо! Мы собирались инициировать переменную `MinPol` (минимальное положительное) значением, которое заведомо больше любого положительного числа. Теперь этой идей воспользоваться не удастся. Константы `maxreal` в Паскале нет. Надо искать другие пути[12].

Что еще можно использовать для инициализации? Найти нужное начальное значение вне последовательности не удалось. Может быть, поискать его в самой последовательности? Проще всего взять первый элемент. Идея: ввести первый элемент и именно его использовать как начальное значение `MinPol`. Это вполне логично. В момент, когда введен единственный элемент, именно он и является минимальным. Правда, для такой инициализации надо быть уверенным, что первый элемент существует. То есть случай пустой последовательности надо будет отфильтровывать заранее. Прикинем, как будет выглядеть соответствующий фрагмент программы.

---

[12] Надо отметить, что для предложенного варианта программы даже в случае целых чисел инициализация переменной `MinPol` константой `maxint` была бы ошибочной. О причинах этого мы поговорим позже в конце подраздела «Сухая прокрутка».

```
{ввести длину последовательности}
    writeln('Введите кол-во элементов последовательности');
    read(n);
if n=0 then writeln('Последовательность пуста')
else begin
    writeln('Введите элемент №1');
    read(tek);
   {инициализировать данные}
     MinPol:= tek;   NomMinPol:= 1;
     MaxOtr:= tek;   NomMaxOtr:= 1;
```

Стоп! Последняя строка нуждается в более пристальном внимании. Насколько корректно инициализировать `MinPol` и `MaxOtr` одним и тем же значением? `MinPol` — это элемент положительной подпоследовательности, а `MaxOtr` — отрицательной. Для инициализации `MinPol` нам нужно первое положительное значение. Его мы будем в дальнейшем сравнивать с другими положительными значениями, чтобы найти минимальное. Если эту же идею распространить на отрицательные числа, то для инициализации `MaxOtr` нужно первое отрицательное значение. Его мы в дальнейшем будем сравнивать с другими отрицательными, чтобы найти максимальное. Но первый элемент последовательности не может быть сразу и положительным, и отрицательным!

Если мы хотим инициализировать `MinPol` и `MaxOtr` первыми элементами соответствующих подпоследовательностей, эти элементы еще надо найти. Мы уже говорили, что только на формальном уровне мы имеем одну последовательность чисел. Содержательно мы должны эту единую последовательность разделить на две подпоследовательности и обрабатывать каждую из них отдельно. Если бы нам надо было искать максимум и минимум всей последовательности, то предложенный нами вариант — инициализировать максимум и минимум первым элементом последовательности — был бы вполне логичен. В момент, когда введено только одно значение, именно оно является одновременно и минимальным, и максимальным. Но, увы! Для нашей задачи применение этого варианта «в лоб» невозможно. Его надо модифицировать таким образом, чтобы в `MinPol` попало первое положительное значение, а в `MaxOtr` — первое отрицательное. Но никаких правил следования положительных и отрицательных чисел у нас не предусмотрено. Они могут идти в любом порядке и в любом количестве. Поэтому поиск первого элемента каждой из подпоследовательностей становится самостоятельной задачей. Соответствующие проверки придется раз-

мещать не перед основным циклом программы, а внутри него. Назвать это инициализацией данных уже нельзя.

Кстати, обратим внимание на то, что кое-какие положительные результаты мы все же получили. Оказывается, проблемы с инициализацией переменной MinPol симметрично отображаются на переменную MaxOtr, и мы столкнулись бы с ними при прогоне теста Т3. Но поскольку проблема уже обнаружена, один тестовый прогон мы сэкономим. Утешение слабое, но больше, чем ничего.

Найти подходящее начальное значение для MinPol не удается. Может быть, стоит посмотреть повнимательней, а что значит «подходящее»? Проблема возникает при проверке условия if tek<MinPol then. Отметим, что сравнение работает верно во всех случаях, кроме первого. До сих пор мы пытались найти такой способ инициализации переменой MinPol, чтобы это сравнение верно сработало и в первом случае. Чем отличается первый случай от всех остальных? Первый из положительных элементов должен попасть в MinPol обязательно, а остальные — только по результатам сравнения. Значит, нам нужно условие, которое будет звучать так:

```
if  Это первый положительный элемент
   or  tek<MinPol then …
```

Слово «положительный» в данном случае даже лишнее, поскольку положительность элемента обеспечена вышестоящим условием if tek>0 then.

Есть ли какой-либо признак, по которому мы можем определить, что текущий элемент является первым положительным? Сами положительные числа ничем друг от друга не отличаются. Первое выглядит так же, как и все последующие. Может быть, его обработка выглядит как-то иначе и оставляет какие-то особые следы? Да, и мы только что об этом говорили. Первый элемент обязательно попадает в MinPol. Значит, до него MinPol имеет начальное «ненастоящее» значение, а после него — первое настоящее. Можем ли мы отличить начальное ненастоящее значение MinPol от прочих ее значений? Да. Именно исходя из этих соображений, мы и подбирали для нее начальное значение нуль. Тогда проверка приобретет следующий вид:

```
if  MinPol=0  or  tek<MinPol then …
```

По сути, мы реализуем таким образом предлагавшуюся ранее идею поиска первого элемента положительной подпоследовательности. Но делаем это (как и было сказано выше) не в инициализации данных, а в основном цикле программы.

Поскольку мы уже заметили, что аналогичные изменения нужны для отрицательной подпоследовательности, сразу же внесем и их. Тело программы приобретет вид:

```
begin
   {выдать начальное приветствие}
     ...
   {инициализировать данные}
     MinPol:= 0;   NomMinPol:= 0;
     MaxOtr:= 0;   NomMaxOtr:= 0;
   {ввести длину последовательности}
     writeln('Введите кол-во элементов'+
             'последовательности');
     read(n);
   for k:= 1 to n do begin
     {ввести очередной элемент}
       writeln('Введите элемент №',k);
       read(tek);
     if tek>0 then
       {обработать положительный элемент}
          if MinPol=0  or  tek<MinPol then begin
             MinPol:= tek;
             NomMinPol:= k
          end
     else
       {обработать отрицательный элемент}
          if MaxOtr=0  or  tek>MaxOtr then begin
             MaxOtr:= tek;
             NomMaxOtr:= k
          end;
   end; {do}
   {выдать результат}
     if n=0 then writeln('Последовательность пуста')
     else if NomMinPol=0 then
               writeln('Положительных чисел нет')
     else if NomMaxOtr=0 then
               writeln('Отрицательных чисел нет')
     else if NomMinPol=0 and NomMaxOtr=0 then
           writeln('Последовательность состоит из'+
                    'одних нулей')
     else if NomMaxOtr<NomMinPol then
          writeln('Максимальное отрицательное стоит'+
                  'перед минимальным положительным')
     else writeln('Минимальное положительное стоит'+
                  'перед максимальным отрицательным')
end.
```

Поскольку изменились условия, придется откорректировать таблицу МГТ. В развилке У3 теперь стоит сложное условие. Поэтому кроме двух сток для этой развилки добавим по две строки для каждого из простых условий. Аналогично поступим с развилкой У4. Чтобы не сбивать прежнюю нумерацию условий, номера новых строк сделаем составными.

			Т1	Т2	Т3	Т4	Т5	Т6	Т7	Т8	Т9	Т10	Т11	Т12	Т13
У3	`if MinPol=0 or tek<MinPol`	+													
		—													
У3.1	`MinPol=0`	+													
		—													
У3.2	`tek<MinPol`	+													
		—													
У4	`if MaxOtr=0 or tek>MaxOtr`	+													
		—													
У4.1	`MaxOtr=0`	+													
		—													
У4.2	`tek>MaxOtr`	+													
		—													

Запускаем заново тест Т2. Теперь трассировка выглядит так:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
1	1	5.6				
			5.6	1		

Результат: Отрицательных чисел нет.

Полученный результат соответствует ожидаемому. Делаем соответствующие пометки в таблице тестов. Отмечаем в таблице МГТ пройденные условия.

В программу были внесены изменения, значит, необходимо повторное тестирование. Благо, нам оно пока обойдется дешево. Надо повторить всего лишь один тест. Никаких изменений в его прогоне не произойдет.

Продолжим тестирование. Тест Т3 даст следующую трассировку:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
1	1	−5				
					−5	1

Результат: Положительных чисел нет.

Полученный результат соответствует ожидаемому. Делаем соответствующие пометки в таблице тестов. Отмечаем в таблице МГТ пройденные условия.

Аналогично поступаем с тестами Т4 и Т5. Получим следующие трассировки.

Для Т4:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
3	1	1				
			1	1		
	2	2				
	3	3				

Результат: Отрицательных чисел нет.

Для Т5:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
2	1	−1				
					−1	1
	2	−2				

Результат: Положительных чисел нет.

Таблицы тестов и МГТ приобретут вид:

Таблица тестов

Тест	**Вход**	**Ожидаемый результат**	**Результат «сухой прокрутки»**	+/−	**Результат выполнения на компьютере**	+/−
Т1	0	Последовательность пуста	Последовательность пуста	+		
Т2	1, 5.6	Отрицательных чисел нет	Отрицательных чисел нет	+		
Т3	1, −5	Положительных чисел нет	Положительных чисел нет	+		
Т4	3, 1, 2, 3	Отрицательных чисел нет	Отрицательных чисел нет	+		
Т5	2, −1, −2	Положительных чисел нет	Положительных чисел нет	+		
...	...	...				

Таблица МГТ

			T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
У1	for k:=1 to n	=0	+												
		=1		+	+										
		>1				+	+								
У2	if tek>0	+		+		+									
		—			+		+								
У3	if MinPol=0 or tek<MinPol	+		+		+									
		—				+									
У3.1	MinPol=0	+		+		+									
		—				+									
У3.2	tek<MinPol	+													
		—		+		+									
У4	if MaxOtr=0 or tek>MaxOtr	+			+		+								
		—					+								
У4.1	MaxOtr=0	+			+		+								
		—					+								
У4.2	tek>MaxOtr	+													
		—			+		+								
У5	if n=0	+	+												
		—		+	+	+	+								
У6	if NomMinPol=0	+			+		+								
		—		+		+									
У7	if NomMaxOtr=0	+		+		+									
		—													
У8	if NomMinPol=0 and NomMaxOtr=0	+													
		—													
У9	NomMinPol=0	+													
		—													
У10	NomMaxOtr=0	+													
		—													
У11	if NomMaxOtr< NomMinPol	+													
		—													

Добавляем тесты T6, T7, T8, T9. Имеем следующие трассировки.

Для T6:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
5	1	−1.23				
					−1.23	1
	2	2.4				
			2.4	2		
	3	−3.1				
	4	4				
	5	7.03				

Результат: Максимальное отрицательное стоит перед минимальным положительным.

Проверим по трассировке: минимальное положительное (2.4) во второй позиции, максимальное отрицательное (−1.23) в первой позиции. Первая позиция раньше второй.

Для T7:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
4	1	2.1				
			2.1	1		
	2	−4.2				
					−4.2	2
	3	−1.3				
					−1.3	3
	4	7.4				

Результат: Минимальное положительное стоит перед максимальным отрицательным.

Проверим по трассировке: минимальное положительное (2.1) в первой позиции, максимальное отрицательное (−1.3) в третьей позиции. Первая позиция раньше третьей.

Для T8:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
3	1	−2				
					−2	1
	2	4				
			4	2		
	3	−3				

Результат: Максимальное отрицательное стоит перед минимальным положительным.

Проверим по трассировке: минимальное положительное (4) во второй позиции, максимальное отрицательное (−2) в первой позиции. Первая позиция раньше второй.

Для T9:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
4	1	1				
			1	1		
	2	7				
	3	9				
	4	−4				
					−4	4
	4	7.4				

Результат: Минимальное положительное стоит перед максимальным отрицательным.

Проверим по трассировке: минимальное положительное (1) в первой позиции, максимальное отрицательное (−4) в четвертой позиции. Первая позиция раньше четвертой.

Таблицы тестов и МГТ будут выглядеть так:

Таблица тестов

Тест	Вход	Ожидаемый результат	Результат «сухой прокрутки»	+/–	Результат выполнения на компьютере	+/–
T1	0	Последовательность пуста	Последовательность пуста	+		
T2	1, 5.6	Отрицательных чисел нет	Отрицательных чисел нет	+		
T3	1, –5	Положительных чисел нет	Положительных чисел нет	+		
T4	3, 1, 2, 3	Отрицательных чисел нет	Отрицательных чисел нет	+		
T5	2, –1, –2	Положительных чисел нет	Положительных чисел нет	+		
T6	5, –1.3, 2.4, –3.1, 4, 7.03	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+		
T7	4, 2.1, –4.2, –3.1, 4, 7.03	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+		
T8	3, –2, 4, –3	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+		
T9	4, 1, 7, 9, –4	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+		
…	…	…				

Таблица МГТ

			T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
У1	for k:=1 to n	=0	+												
		=1		+	+										
		>1				+	+	+	+	+	+				
У2	if tek>0	+		+		+		+	+	+	+				
		—			+		+	+	+	+	+				
У3	if MinPol=0 or tek<MinPol	+		+		+		+	+	+	+				
		—				+		+	+		+				
У3.1	MinPol=0	+		+		+		+	+	+	+				
		—				+		+	+		+				
У3.2	tek<MinPol	+													
		—		+		+		+	+	+	+				
У4	if MaxOtr=0 or tek>MaxOtr	+			+		+	+	+	+	+				
		—					+	+		+					
У4.1	MaxOtr=0	+			+		+	+	+	+	+				
		—					+	+	+	+					
У4.2	tek>MaxOtr	+							+						
		—			+		+	+	+	+	+				
У5	if n=0	+	+												
		—		+	+	+	+	+	+	+	+				
У6	if NomMinPol=0	+			+		+								
		—		+		+		+	+	+	+				
У7	if NomMaxOtr=0	+		+		+									
		—						+	+	+	+				
У8	if NomMinPol=0 and NomMaxOtr=0	+													
		—						+	+	+	+				
У9	NomMinPol=0	+													
		—						+	+	+	+				
У10	NomMaxOtr=0	+													
		—						+	+	+	+				
У11	if NomMaxOtr< NomMinPol	+						+		+					
		—							+		+				

Тесты, разработанные исходя из критериев черного ящика, неплохо ведут себя и с точки зрения критериев белого ящика. Незакрытыми остались условие У3.2+ и группа условий У8+, У9+, У10+, которые могут быть выполнены только все вместе. Тесты Т8 и Т9 с точки зрения МГТ вообще избыточны. Ни одной новой строки в таблице они не закрыли. Идущий далее тест Т10 должен закрыть 3 из четырех пока еще не закрытых строк.

Его трассировка:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
3	1	0				
					0	1
	2	0				
					0	2
	3	0				
					0	3

Результат: Положительных чисел нет.

Стоп! Это совсем не то, на что мы рассчитывали. Тест — удачен! В программе обнаружено наличие еще одной ошибки! Начинаем разбираться.

Прежде всего, в трассировочной таблице бросается в глаза то, что мы каждый раз (точнее, для каждого элемента последовательности) перевычисляли `MaxOtr` и `NomMaxOtr`, хотя не должны были делать этого ни разу. При этом `MinPol` и `NomMinPol`, как и положено, ни разу не перевычислялись. Почему предшествующие условия предохранили от перевычисления `MinPol` и `NomMinPol`, но не предохранили `MaxOtr` и `NomMaxOtr`?

Условие, предохраняющее `MinPol` и `NomMinPol`: `if tek>0 then`. Оно отфильтровывает только положительные числа. А вот условие, отфильтровывающее только отрицательные числа, в программе отсутствует. Мы допустили стандартную ошибку: потеряли третий результат операции сравнения. Все числа, которые «НЕ больше нуля», мы объявили отрицательными. Но ведь в эту категорию попадает и сам нуль. Значит, на ветке «иначе» вышеуказанного оператора `if tek>0 then` должна появиться еще одна проверка: `if tek<0 then`. Основной цикл программы приобретет вид:

```
for k:= 1 to n do begin
   {ввести очередной элемент}
      writeln('Введите элемент №',k);
      read(tek);
   if tek>0 then
      {обработать положительный элемент}
        if MinPol=0  or  tek<MinPol then begin
           MinPol:= tek;
           NomMinPol:= k
        end
```

```
    else if tek<0 then
      {обработать отрицательный элемент}
        if MaxOtr=0  or  tek>MaxOtr then begin
          MaxOtr:= tek;
          NomMaxOtr:= k
        end
    else {tek=0 - ничего не делать}
  end; {do}
```

Программа исправлена. Повторяем тест Т10.

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
3	1	0				
	2	0				
	3	0				

Результат: Положительных чисел нет.

Опять? Мы же исправили ошибку! Почему еще что-то не в порядке?

Сделаем маленькую паузу. В главе 4 «Принципы тестирования» был сформулирован принцип 10: результаты теста необходимо изучать досконально и объяснять полностью. Сделали мы это в предыдущий раз? Нет. У нас было 2 признака ошибки: «странная» трасса и неверный результат. На «странную» трассу мы отреагировали, а на неверный результат вообще не обратили внимания. И это обойдется нам, как минимум, в лишний тестовый прогон. Для нашей игрушечной программы это не страшно. А вот для реальной программы может быть весьма затратно.

Пауза закончена. Запускаем ретроанализ.

Сообщение «Положительных чисел нет» выдается при выполнении условия `NomMinPol=0`. И при прогоне теста Т10 это условие действительно должно быть истинно. Но сообщение должно выдаваться не это, а «Последовательность состоит из одних нулей». Это сообщение в программе предусмотрено. Почему же оно не выдается? Сообщение прикрыто верным условием: `NomMinPol=0 and NomMaxOtr=0`. Но стоит оно слишком далеко. Равенство нулю переменных `NomMinPol` и `NomMaxOtr` проверяется предстоящими операторами:

```
else if NomMinPol=0 then writeln('Положительных чисел нет')
else if NomMaxOtr=0 then writeln('Отрицательных чисел нет')
```

Необходимо «поднять» проверку `if NomMinPol=0 and NomMaxOtr=0 then` выше по тексту. Выдача результата будет иметь вид:

```
{выдать результат}
  if n=0 then writeln('Последовательность пуста')
  else if NomMinPol=0 and NomMaxOtr=0 then
    writeln('Последовательность состоит из одних нулей')
  else if NomMinPol=0 then writeln('Положительных чисел нет')
  else if NomMaxOtr=0 then writeln('Отрицательных чисел нет')
  else if NomMaxOtr<NomMinPol then
    writeln('Максимальное отрицательное стоит перед'+
            'минимальным положительным')
  else writeln('Минимальное положительное стоит перед'+
               'максимальным отрицательным')
end.
```

Условие `NomMinPol=0 and NomMaxOtr=0` истинно только тогда, когда истинны оба простых условия. Если хотя бы одна из переменных ненулевая (т. е. в соответствующей подпоследовательности были элементы), программа пройдет через конъюнкцию и начнет — в следующих операторах — проверять каждое из простых условий. Если одно из них истинно, будет выдано соответствующее сообщение.

Посмотрим, нет ли еще каких-либо признаков ошибки. Вроде бы нет. Тогда повторяем тест Т10 еще раз:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
3	1	0				
	2	0				
	3	0				

Результат: Последовательность состоит из одних нулей.

Результат совпадает с ожидаемым. Трассировка также не вызывает вопросов.

В программу были внесены исправления, значит, необходимо повторное тестирование. Перспектива повторного прогона девяти тестов не радует. Но на практике картина оказывается далеко не такой мрачной. Внесенные нами изменения узко направлены и не оказывают влияния ни на один из ранее выполненных тестов. (Нулей-то в этих тестах не было!) Все трассировочные таблицы сохранят свой прежний вид. Откорректировать придется только

таблицу МГТ. Да и то корректировка коснется только четырех условий: добавленного У3.3 и переставленных У8, У9, У10 (их помещаем перед условиями У6 и У7).

Таблица МГТ

			Т1	Т2	Т3	Т4	Т5	Т6	Т7	Т8	Т9	Т10	Т11	Т12	Т13
У1	for k:=1 to n	=0	+												
		=1		+	+										
		>1				+	+	+	+	+	+	+			
У2	if tek>0	+		+		+		+	+	+	+				
		—			+		+	+	+	+	+	+			
У3	if MinPol=0 or tek<MinPol	+		+		+		+	+	+	+				
		—				+		+	+		+				
У3.1	MinPol=0	+		+		+		+	+	+	+				
		—				+		+	+		+				
У3.2	tek<MinPol	+													
		—		+		+		+	+	+	+				
У3.3	if tek<0	+			+		+	+	+	+	+				
		—										+			
У4	if MaxOtr=0 or tek>MaxOtr	+			+		+	+	+	+	+				
		—					+	+		+					
У4.1	MaxOtr=0	+			+		+	+	+	+	+				
		—					+	+	+	+					
У4.2	tek>MaxOtr	+							+						
		—			+		+	+	+	+	+				
У5	if n=0	+	+												
		—		+	+	+	+	+	+	+	+	+			
У8	if NomMinPol=0 and NomMaxOtr=0	+										+			
		—		+	+	+	+	+	+	+	+				
У9	NomMinPol=0	+			+		+					+			
		—		+		+		+	+	+	+				
У10	NomMaxOtr=0	+		+		+						+			
		—			+		+	+	+	+	+				
У6	if NomMinPol=0	+			+		+								
		—		+		+		+	+	+	+				
У7	if NomMaxOtr=0	+		+		+									
		—						+	+	+	+				
У11	if NomMaxOtr< NomMinPol	+						+		+					
		—							+		+				

Отметим, что неприкрытой осталась единственная строка: У3.2+.

Осталось прогнать три теста: Т11, Т12, Т13. Начнем.

T11:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
6	1	5				
			5	1		
	2	2				
			2	2		
	3	7				
	4	−6				
					−6	4
	5	−3				
					−3	5
	6	−8				

Результат: Минимальное положительное стоит перед максимальным отрицательным.

Результат совпадает с ожидаемым. Проверим по трассировке: минимальное положительное (2) во второй позиции, максимальное отрицательное (−3) в пятой позиции. Вторая позиция раньше пятой.

Делаем пометки в таблицах тестов и МГТ. Ура! Мы закрыли последнюю строку в МГТ. Тестирование с точки зрения принятого критерия белого ящика выполнено. Оставшиеся тесты интересуют нас постольку, поскольку они были запланированы с точки зрения черного ящика. С точки зрения белого ящика они избыточны.

T12:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
7	1	−6				
					−6	1
	2	−3				
					−3	2
	3	−8				
	4	−3				
	5	5				
			5	5		
	6	7				
	7	2				
			2	7		

Результат: Максимальное отрицательное стоит перед минимальным положительным.

Результат совпадает с ожидаемым. Проверим по трассировке: минимальное положительное (2) в седьмой позиции, максимальное отрицательное (–3) во второй позиции. Вторая позиция раньше седьмой.

Делаем пометки в таблицах тестов и МГТ.

T13 (Последний!):

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
7	1	–6				
					–6	1
	2	2				
			2	2		
	3	–8				
	4	–3				
					–3	4
	5	5				
	6	7				
	7	2				
	8	–3				

Результат: Минимальное положительное стоит перед максимальным отрицательным.

Результат совпадает с ожидаемым. Проверим по трассировке: минимальное положительное (2) во второй позиции, максимальное отрицательное (–3) в четвертой позиции. Вторая позиция раньше четвертой. Ура!

Стоп! Это же тест T13. Что-то мы о нем уже говорили... Была какая-то заминка, когда мы его еще только придумывали. Даже к заказчику, помнится, пришлось обращаться. О чем же мы тогда говорили? Ах, да! Тест проверяет случай многих экстремумов. Мы тогда еще запутались в этих многочисленных экстремумах, поскольку положительные минимумы шли вперемешку с отрицательными максимумами. Был вопрос, какие из них сопоставлять. И как же на него тогда ответили? Заказчик сказал, что сравнивать надо позиции последних экстрему-

мов. В данном тесте это два последних числа. Но наша-то программа сравнивала не последние экстремумы, а первые! Похоже, в программе ошибка. Несмотря на верный результат. В тесте T13 первый положительный минимум стоит перед первым отрицательным максимумом, и последний положительный минимум стоит перед последним отрицательным максимумом. Вот у нас и получается верный результат. А если их в одном случае переставить? Пусть первый положительный минимум по-прежнему стоит перед первым отрицательным максимумом, а вот последний положительный минимум переставим после последнего отрицательного максимума. Конструируем тест T14:

> Вход: 8, −6, 2, −8, −3, 5, 7, −3, 2.
>
> Ожидаемый выход: Максимальное отрицательное стоит перед минимальным положительным.

Проверяем:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
7	1	−6				
					−6	1
	2	2				
			2	2		
	3	−8				
	4	−3				
					−3	4
	5	5				
	6	7				
	7	−3				
	8	2				

> Результат: Минимальное положительное стоит перед максимальным отрицательным.

Есть! Результат не совпадает с ожидаемым. В программе действительно ошибка! Запоминается номер не последнего, а первого экстремума. Это касается как положительного минимума, так и отрицательного максимума.

Разбираемся почему. Запоминание экстремумов прикрыто условиями:

```
if MinPol=0  or  tek<MinPol then
if MaxOtr=0  or  tek>MaxOtr then
```

Текущее значение считается экстремальным либо если оно первое (`MinPol`= 0, `MaxOtr`= 0), либо если оно строго меньше (строго больше) существующего экстремума. Но если речь идет о нескольких одинаковых значениях, то последующий экстремум НЕ меньше (НЕ больше) предыдущего, а равен ему. Условия должны выглядеть так:

```
if MinPol=0  or  tek<=MinPol then
if MaxOtr=0  or  tek>=MaxOtr then
```

Тогда трассировка теста Т14 приобретет вид:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
7	1	−6				
					−6	1
	2	2				
			2	2		
	3	−8				
	4	−3				
					−3	4
	5	5				
	6	7				
	7	−3				
					−3	7
	8	2				
			2	8		

Результат: Максимальное отрицательное стоит перед минимальным положительным.

Результат совпадает с ожидаемым. Проверим по трассировке: минимальное положительное (2) в восьмой позиции, максимальное отрицательное (−3) в седьмой позиции. Седьмая позиция раньше восьмой.

Повторяем тест T13, на котором и заметили ошибку:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
7	1	−6				
					−6	1
	2	2				
			2	2		
	3	−8				
	4	−3				
					−3	4
	5	5				
	6	7				
	7	2				
			2	7		
	8	−3				
					−3	8

Результат: Минимальное положительное стоит перед максимальным отрицательным.

Вот теперь не только результат совпадает с ожидаемым, но и трассировка подтверждает, что сравнивались именно нужные элементы. Минимальное положительное (2) в седьмой позиции, максимальное отрицательное (−3) в восьмой позиции. Седьмая позиция раньше восьмой.

С этой ситуацией разобрались. И разобрались, почему T13 не «ткнул нас носом» в ошибку при первом прогоне. А остальные тесты? Они почему эту ошибку пропустили? Потому что в них во всех только по одному экстремуму было. А тест T12? Когда мы придумывали тесты под критерий упорядоченности последовательности, то первый тест, в котором было несколько экстремумов (отрицательных), был тест T12. Почему же он не выявил ошибки? Взглянем еще раз на сам тест и на его трассировочную таблицу. С тестом все ясно, все отрицательные числа стоят раньше положительных. Значит, и максимальное тоже. Никакого отличия, влияющего на результат работы программы, между первым и последним экстремумами здесь нет. А вот по трассировке заметить нелады было бы можно. В трассировочной таблице видно, что в качестве отрицательного максиму-

ма сохранилось первое −3, а не второе. Почему мы этого не заметили? По невнимательности! Увидели, что окончательный результат совпадает с ожидаемым, и анализировать промежуточные результаты, показанные трассировочной таблицей, не стали.

И ведь было такое уже на тесте T10. Только там наоборот. На ошибки в трассировке внимание обратили, а на неверный результат — нет. Может быть, уже пора поверить в принцип 10: результаты теста необходимо изучать досконально и объяснять полностью?

Повторяем T12 в новых условиях:

n	*k*	*tek*	*MinPol*	*NomMinPol*	*MaxOtr*	*NomMaxOtr*
			0	0	0	0
7	1	−6				
					−6	1
	2	−3				
					−3	2
	3	−8				
	4	−3				
					−3	4
	5	5				
			5	5		
	6	7				
	7	2				
			2	7		

Результат: Максимальное отрицательное стоит перед минимальным положительным.

Результат по-прежнему совпадает с ожидаемым. Но теперь в трассировке видно, что в качестве отрицательного максимума взято последнее −3.

Если вы заметили, мы уже перешли к повторному тестированию. Для остальных тестов оно пройдет очень быстро, поскольку, как уже было раньше, изменения в программе узко направлены и большинство тестов не заденут.

Заносим результаты в таблицу тестов. Для тестов T12–T14 проверяем еще раз таблицу МГТ. Не произошло ли в ней каких-либо изменений? Теперь таблицы будут выглядеть так:

Таблица тестов

Тест	Вход	Ожидаемый результат	Результат «сухой прокрутки»	+/–	Результат выполнения на компьютере	+/–
T1	0	Последовательность пуста	Последовательность пуста	+		
T2	1, 5.6	Отрицательных чисел нет	Отрицательных чисел нет	+		
T3	1, –5	Положительных чисел нет	Положительных чисел нет	+		
T4	3, 1, 2, 3	Отрицательных чисел нет	Отрицательных чисел нет	+		
T5	2, –1, –2	Положительных чисел нет	Положительных чисел нет	+		
T6	5, –1.3, 2.4, –3.1, 4, 7.03	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+		
T7	4, 2.1, –4.2, –3.1, 4, 7.03	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+		
T8	3, –2, 4, –3	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+		
T9	4, 1, 7, 9, –4	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+		
T10	3, 0, 0, 0	Последовательность состоит из одних нулей	Последовательность состоит из одних нулей	+		
T11	6, 5, 2, 7, –6, –3, –8	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+		
T12	7, –6, –3, –8, –3, 5, 7, 2	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+		
T13	8, –6, 2, –8, –3, 5, 7, 2, –3	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+		
T14	8, –6, 2, –8, –3, 5, 7, –3, 2	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+		

Таблица МГТ

			T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14
У1	for k:=1 to n	=0	+													
		=1		+	+											
		>1				+	+	+	+	+	+	+	+	+	+	+
У2	if tek>0	+		+		+		+	+	+	+		+	+	+	+
		—			+		+	+	+	+	+	+	+	+	+	+
У3	if MinPol=0 or tek<=MinPol	+		+		+		+	+	+	+		+	+	+	+
		—				+		+	+		+		+	+	+	+
У3.1	MinPol=0	+		+		+		+	+	+	+		+	+	+	+
		—				+		+	+		+		+	+	+	+
У3.2	tek<=MinPol	+											+	+	+	+
		—		+		+		+	+	+	+		+	+	+	+
У3.3	if tek<0	+			+		+	+	+	+	+		+	+	+	+
		—										+				
У4	if MaxOtr=0 or tek>=MaxOtr	+			+		+	+	+	+	+		+	+	+	+
		—					+	+		+			+	+	+	+
У4.1	MaxOtr=0	+			+		+	+	+	+	+		+	+	+	+
		—					+	+	+	+			+	+	+	+
У4.2	tek>=MaxOtr	+							+				+	+	+	+
		—			+		+	+	+	+	+		+	+	+	+
У5	if n=0	+	+													
		—		+	+	+	+	+	+	+	+	+	+	+	+	+
У8	if NomMinPol=0 and NomMaxOtr=0	+										+				
		—		+	+	+	+	+	+	+	+		+	+	+	+
У9	NomMinPol=0	+			+		+					+				
		—		+		+		+	+	+	+		+	+	+	+
У10	NomMaxOtr=0	+		+		+						+				
		—			+		+	+	+	+	+		+	+	+	+
У6	if NomMinPol=0	+			+		+									
		—		+		+		+	+	+	+		+	+	+	+
У7	if NomMaxOtr=0	+		+		+										
		—						+	+	+	+		+	+	+	+
У11	if NomMaxOtr< NomMinPol	+						+		+				+		+
		—							+		+		+		+	

Перед окончанием раздела вернемся к вопросу об инициализации переменных `MinPol` и `MaxOtr`. Среди обсуждавшихся вариантов было предложение об инициализации `MinPol` максимальным допустимым целым числом. Мы отказались от этого предложения, поскольку переменная `MinPol` оказалась вещественной. Но при этом сделали замечание, что для того варианта программы даже в случае целых чисел инициализация кон-

стантой maxint была бы ошибочной. Сейчас самое время разобраться с этим замечанием.

Ошибочность инициализации переменной MinPol с помощью константы maxint была связана с двумя факторами:

1) необходимостью зафиксировать не только само минимальное значение, но и его номер;
2) строгими сравнениями tek<MinPol и tek>MaxOtr, стоявшими тогда в программе.

Представьте, к чему привела бы в этих условиях инициализация MinPol:= maxint.

Для того чтобы при такой инициализации программа сработала правильно, требовалось, чтобы среди положительных элементов оказался хотя бы один, меньший maxint. В этом случае условие if tek<MinPol оказалось бы истинным, и мы бы правильно запомнили и MinPol, и NomMinPol. Однако если бы все положительные элементы были равны maxint, условие if tek<MinPol никогда не стало бы истинным. В этом случае переменная MinPol сохранила бы свое начальное значение. Поскольку это значение равнялось бы значению всех положительных элементов, оно равнялась бы и минимальному положительному элементу. То есть переменная MinPol имела бы правильное значение. Но вместе с ней сохранила бы свое начальное — нулевое — значение и переменная NomMinPol. А это было бы уже неверно.

Для переменной MaxOtr и константы -maxint ситуация была строго аналогична.

18.5. Отладка на компьютере

Безмашинное тестирование закончено. Пора на компьютер. Вводим программу и начинаем прогоны. При этом можно будет опробовать на практике отладочные средства Турбо Паскаля, о которых говорилось в предыдущей главе.

В этом месте читателю настоятельно рекомендуется действительно пересесть за компьютер и проделать все нижеописанные действия.

Итак, набираем текст и нажимаем **Ctrl+F9** (команда **Run**).

Первое, что нас ждет, это сообщение об ошибке, показанное на рис. 18.1. На рисунке только не видно, что курсор мигает в строке

```
if MinPol=0  or  tek<=MinPol  then begin
```

на знаке <.

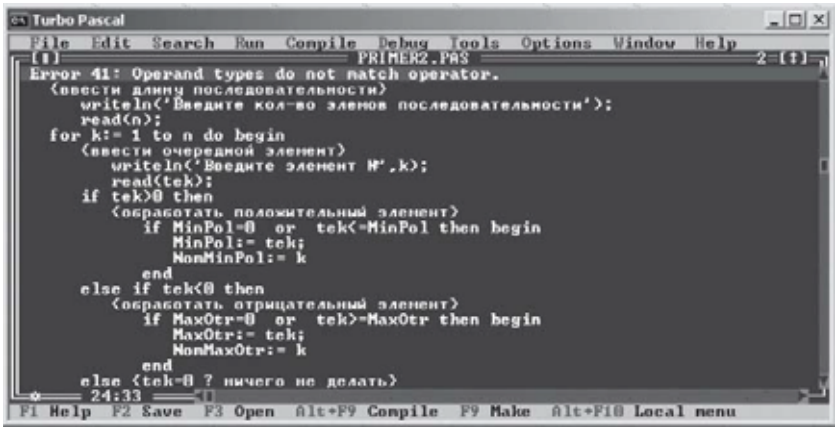


Рис. 18.1. Ошибка при трансляции сложного условия

Тип операнда не соответствует операции. Операнд, видимо, `tek`. А операция <= ? Непонятно. Переменная `tek` — вещественная. Почему вещественная переменная не может быть операндом операции сравнения? Может быть, речь идет о какой-то другой операции? К чему еще может быть отнесена переменная `tek`? Знак сравнения стоит от него справа, а слева — знак дизъюнкции `or`. Может быть, дело в нем? Но какое отношение имеет `tek` к дизъюнкции? Логически складываться должны результаты двух сравнений: `MinPol=0` и `tek<=MinPol`.

И тем не менее проблема именно в дизъюнкции. Более того, об этой проблеме нас предупреждали в главе 9 «Ошибкоопасные ситуации». Мы опять допустили стандартную ошибку: не учли старшинство операций. В Паскале приоритет логического сложения выше приоритета операций сравнения. Поэтому в сложном условии Паскаль сначала попробует логически сложить нуль и `tek`, потом уже результат будет сравнивать на равенство с `MinPol`.

Чтобы исправить ошибку, надо все простые условия заключить в скобки:

```
if (MinPol=0)  or  (tek<=MinPol)  then begin
```

С ошибкой мы разобрались. Почему она была допущена? Оказывается, мы не совсем правильно понимали, как в Паскале вычисляются сложные условия. Но если мы этого не понимали в данном операторе, то, скорее всего, мы не понимали этого и в других аналогичных случаях. Это значит, что такая же ошибка могла быть допущена при записи других сложных условий. Име-

ет смысл проверить иные сложные условия, используемые в нашей программе. Таких мест всего два: в обработке отрицательного элемента и выдаче результата в случае последовательности, состоящей из одних нулей. Мы оказались правы. В обоих местах допущена та же самая ошибка: в сложном условии отсутствуют скобки вокруг операций сравнения. Добавим их:

```
if (MaxOtr=0)  or  (tek>=MaxOtr) then begin
else if (NomMinPol=0) and (NomMaxOtr=0) then
```

После этих исправлений у транслятора претензий больше нет, и наша программа начинает выполняться. Запускаем тест T1. Его машинные результаты совпадают с ожидаемыми (см. рис. 18.2).

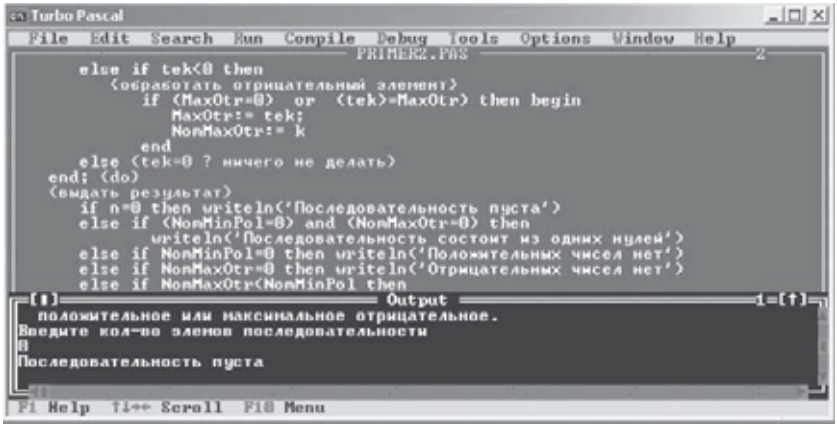


Рис. 18.2. Компьютерный прогон теста T1

Заносим их в таблицу тестов.

Тест	Вход	Ожидаемый результат	Результат «сухой прокрутки»	+/–	Результат выполнения на компьютере	+/–
T1	0	Последовательность пуста	Последовательность пуста	+	Последовательность пуста	+
...	...	...	...			

Следующий тест — T2. Результаты опять совпадают с ожидаемыми (см. рис. 18.3). Делаем отметку в таблице тестов.

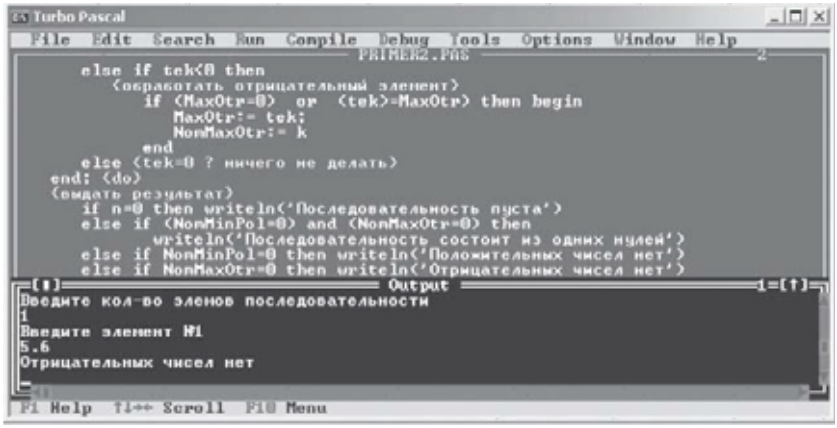


Рис. 18.3. Компьютерный прогон теста Т2

Запускаем тест Т3. Программа правильно отработала отсутствие отрицательных чисел. А положительные чем хуже? Посмотрим результат на рис. 18.4.

Результат неожиданный. Программа отказалась признать –5 отрицательным числом, объявила его нулем! Но ведь при «сухой прокрутке» результат был иным. Получается, что мы и Паскаль понимаем нашу программу по-разному. Ну что ж, давайте выясним, в чем у нас разногласия. Воспользуемся отладочными

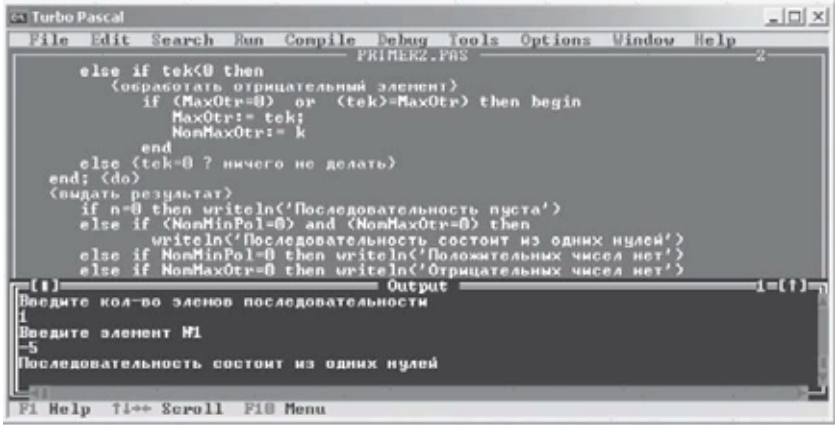


Рис. 18.4. Компьютерный прогон теста Т3

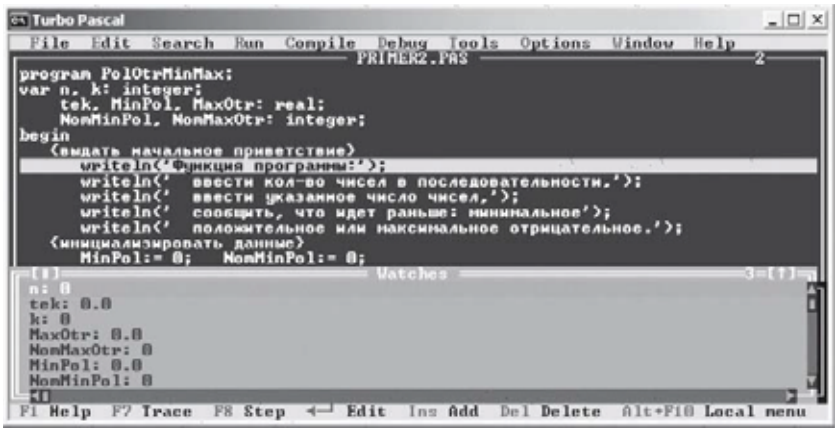


Рис. 18.5. Начало пошагового исполнения программы

средствами Турбо Паскаля. Откроем окно **Watches** (в верхнем меню позиция **Debug**), загрузим в него наши переменные (благо их немного) и начнем трассировку (клавиши **F7** или **F8**, в нашей ситуации все равно). После второго нажатия клавиши получим экран следующего вида (см. рис. 18.5). Данные инициализированы самим Паскалем.

Динамическая развертка нашей программы будет выглядеть так:

```
1. begin
2. writeln('Функция программы:');
3. writeln(' ввести кол-во чисел в последовательности,');
4. writeln(' ввести указанное число чисел,');
5. writeln(' сообщить, что идет раньше: минимальное');
6. writeln(' положительное или максимальное отрицательное.');
7. MinPol:= 0;   NomMinPol:= 0;
8. MaxOtr:= 0;   NomMaxOtr:= 0;
9. writeln('Введите кол-во элементов последовательности');
10. read(n);
11. for k:= 1 to n do begin
12. writeln('Введите элемент №',k);
13. read(tek);
14. if tek>0 then
15. end; {do}
16. if n=0 then writeln('Последовательность пуста')
17. else if (NomMinPol=0) and (NomMaxOtr=0) then
18. writeln('Последовательность состоит из одних нулей')
19. end.
```

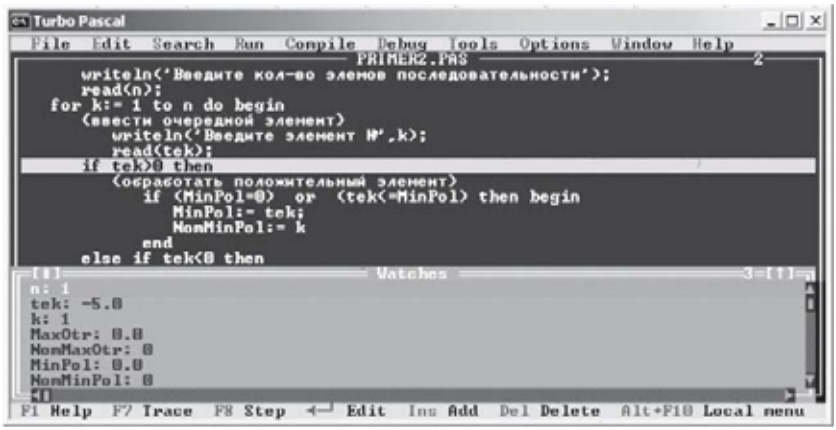


Рис. 18.6. Продолжение пошагового исполнения программы

Попробуем разобраться, что мы сделали не так. Данные инициализировали верно (`MaxOtr`, `NomMaxOtr`, `MinPol`, `NomMinPol` — все нули). Длину последовательности загрузили верно (`n` = 1). Цикл выполнился верно: от 1 до 1. Элемент последовательности загрузили верно (`tek` = −5). А вот после шага 14 — сбой. Мы должны были пойти на ветвь `else if tek<0 then`, а вместо этого вообще выскочили из условного оператора и оказались в конце тела цикла. Получается, что `else if tek<0 then` не является ветвью «иначе» для оператора `if tek>0 then`. Но ведь ошибок трансляции не было. Значит, структура условных операторов с формальной точки зрения верна. Значит, каждая ветвь `else` присоединена к какому-то `if`.

Кроме `if tek>0 then` перед `else if tek<0` стоит еще один `if`: `if (MinPol=0) or (tek<=MinPol)`. Значит, ветвь `else if tek<0` может быть отнесена только к нему. То есть структуру этого оператора Паскаль понимает так:

```
if tek>0 then
   {обработать положительный элемент}
      if (MinPol=0)  or  (tek<=MinPol) then begin
          MinPol:= tek;
          NomMinPol:= k
        end
      else if tek<0 then
```

В очередной раз мы допустили стандартную ошибку, описанную в перечне ошибкоопасных ситуаций. На этот раз мы запутались с вложенными условными операторами, забыли, что каждый else Паскаль автоматически присоединяет к ближайшему предшествующему if. Рекомендации по исправлению этой ошибки также даны в главе 9 «Ошибкоопасные ситуации». Либо после then надо писать полную форму условного оператора, либо сокращенные условные операторы, стоящие после then, заключать в операторные скобки begin–end. Получим два варианта.

Первый:

```
if tek>0 then
   {обработать положительный элемент}
      if (MinPol=0)  or  (tek<=MinPol) then begin
          MinPol:= tek;
          NomMinPol:= k
        end
      else {ничего}
else if tek<0 then
```

Второй:

```
if tek>0 then
   {обработать положительный элемент}
   begin
      if (MinPol=0)  or  (tek<=MinPol) then begin
         MinPol:= tek;
         NomMinPol:= k
      end
   end
else if tek<0 then
```

Какой вариант выбрать, все равно. Пусть будет первый.

С ошибкой разобрались. В чем была ее причина? Мы неверно записали сокращенный условный оператор внутри другого условного оператора. В результате была нарушена структура условных операторов, часть «иначе» оказалась присоединена не к тому оператору.

Мы уже выучили правило: исправили ошибку — проверьте, нет ли других таких же. Есть ли еще в программе места, где

аналогичным образом используются вложенные условные операторы? Да, есть. При обработке отрицательного элемента. Проверить это место есть еще одна причина: мы уже привыкли, что положительный и отрицательный экстремумы обрабатываются аналогично. И действительно, там нас ждет точно такая же картина. Сейчас мы имеем дело со следующей трактовкой наших вложенных условных операторов:

```
else if tek<0 then
   {обработать отрицательный элемент}
      if (MaxOtr=0)  or  (tek>=MaxOtr) then begin
          MaxOtr:= tek;
          NomMaxOtr:= k
        end
      else {tek=0 - ничего не делать}
```

Часть `else {tek=0 - ничего не делать}` присоединена не к оператору `if tek<0`, а к оператору `if (MaxOtr=0) or (tek>=MaxOtr)`. Для исправления ошибки надо либо сделать оператор `if(MaxOtr=0) or (tek>=MaxOtr)` полным, либо заключить его в операторные скобки. В предыдущем случае — при обработке положительного минимума — был выбран первый вариант. Логично так же поступить и сейчас. Получим:

```
else if tek<0 then
   {обработать отрицательный элемент}
      if (MaxOtr=0)  or  (tek>=MaxOtr) then begin
          MaxOtr:= tek;
          NomMaxOtr:= k
        end
      else {ничего}
else {tek=0 - ничего не делать}
```

Интересно отметить, что если отвлечься от комментариев, стоящих после `else`, последняя ошибка, строго говоря, ошибкой не является. И в операторе `if tek<0`, и в операторе `if(MaxOtr=0) or (tek>=MaxOtr)` части «иначе» пусты. Поэтому к какому из операторов будет приписан единственный `else`, в данном случае не важно.

Новых развилок в программе не появилось, прежние — не изменились. Поэтому в таблице МГТ никаких изменений не произойдет.

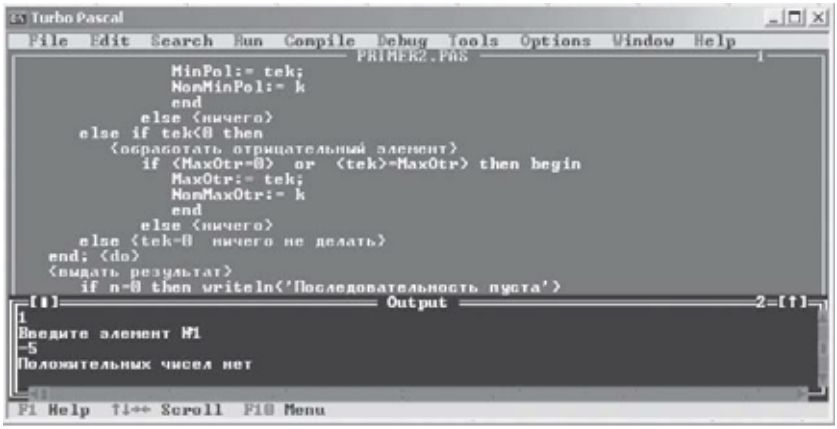


Рис. 18.7. Повторный прогон теста Т3

Продолжим тестирование. Прежде всего, повторим тест Т3. Теперь результаты машинного выполнения совпадут с результатами сухой прокрутки (см. рис. 18.7).

Повторяем тесты Т1 и Т2. Результаты совпадают с теми, которые уже есть в таблице тестов:

Тест	Вход	Ожидаемый результат	Результат «сухой прокрутки»	+/–	Результат выполнения на компьютере	+/–
Т1	0	Последовательность пуста	Последовательность пуста	+	Последовательность пуста	+
Т2	1, 5.6	Отрицательных чисел нет	Отрицательных чисел нет	+	Отрицательных чисел нет	+
Т3	1, –5	Положительных чисел нет	Положительных чисел нет	+	Положительных чисел нет	+
…	…	…	…			

Продолжим. Результаты выполнения тестов Т4–Т14 на компьютере совпадают с результатами «сухой прокрутки».

Таблица тестов

Тест	Вход	Ожидаемый результат	Результат «сухой прокрутки»	+/–	Результат выполнения на компьютере	+/–
T1	0	Последовательность пуста	Последовательность пуста	+	Последовательность пуста	+
T2	1, 5.6	Отрицательных чисел нет	Отрицательных чисел нет	+	Отрицательных чисел нет	+
T3	1, –5	Положительных чисел нет	Положительных чисел нет	+	Положительных чисел нет	+
T4	3, 1, 2, 3	Отрицательных чисел нет	Отрицательных чисел нет	+	Отрицательных чисел нет	+
T5	2 , –1, –2	Положительных чисел нет	Положительных чисел нет	+	Положительных чисел нет	+
T6	5, –1.3, 2.4, –3.1, 4, 7.03	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+	Максимальное отрицательное стоит перед минимальным положительным	+
T7	4, 2.1, –4.2, –3.1, 4, 7.03	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+	Минимальное положительное стоит перед максимальным отрицательным	+
T8	3, –2, 4, –3	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+	Максимальное отрицательное стоит перед минимальным положительным	+
T9	4, 1, 7, 9, –4	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+	Минимальное положительное стоит перед максимальным отрицательным	+
T10	3, 0, 0, 0	Последовательность состоит из одних нулей	Последовательность состоит из одних нулей	+	Последовательность состоит из одних нулей	+
T11	6, 5, 2, 7, –6, –3, –8	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+	Минимальное положительное стоит перед максимальным отрицательным	+
T12	7, –6, –3, –8, –3, 5, 7, 2	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+	Максимальное отрицательное стоит перед минимальным положительным	+
T13	8, –6, 2, –8, –3, 5, 7, 2, –3	Минимальное положительное стоит перед максимальным отрицательным	Минимальное положительное стоит перед максимальным отрицательным	+	Минимальное положительное стоит перед максимальным отрицательным	+
T14	8, –6, 2, –8, –3, 5, 7, –3, 2	Максимальное отрицательное стоит перед минимальным положительным	Максимальное отрицательное стоит перед минимальным положительным	+	Максимальное отрицательное стоит перед минимальным положительным	+

Тестирование закончено. Мы прощаемся с нашей программой. В заключение сделаем одно замечание, не имеющее отношения к процессу тестирования.

При разработке программы мы несколько раз сталкивались с необходимостью установить наличие или отсутствие элементов в положительной или отрицательной подпоследовательности. Вообще говоря, у нас есть для этого два индикатора: `MinPol` (`MaxOtr`) и `NomMinPol` (`NomMaxOtr`). Пока нет ни одного элемента в положительной подпоследовательности, переменные `MinPol` и `NomMinPol` равны нулю. Как только появляется первый элемент, их значения становятся ненулевыми. Аналогично ведут себя `MaxOtr` и `NomMaxOtr` для отрицательной подпоследовательности.

Если посмотреть на получившийся в конце концов текст программы, окажется, что в разных местах программы мы для выявления одного и того же свойства (отсутствие положительных или отрицательных чисел в последовательности) использовали разные признаки. Внутри цикла — равенство нулю переменных `MinPol` или `MaxOtr`. После цикла — равенство нулю `NomMinPol` или `NomMaxOtr`. Это не ошибка, но такое разночтение сбивает с толку, вызывает лишние вопросы: почему в одном случае один признак, а в другом — другой? Причина этой неаккуратности в том, что изначально в первом варианте программы стояли проверки `if tek<MinPol` и `if tek>MaxOtr`. Именно их пришлось исправлять. Условия изменили, но переменные «по инерции» оставили эти же. Для пользователя это не имеет никакого значения. А вот с точки зрения программиста лучше стремиться к единообразию.

18.6. Уроки данного примера

Этот раздел предназначается для повторения и закрепления изученного материала. Сделаем несколько заключительных замечаний.

1. Пример показал, что хорошо проработанные тесты черного ящика вполне могут оказаться не только достаточны с точки зрения критериев белого ящика, но даже избыточны.
2. В процессе отладки можно добавлять новые тесты прицельно для проверки интересующих нас ситуаций.
3. Набор тестов, полный как с точки зрения черного, так и с точки зрения белого ящика, обнаружил НЕ все ошибки в программе. Признаки части ошибок удалось обнаружить только в трассировочных таблицах. И только позже подтвердить их специально сконструированным тестом.

4. 30 лет назад голландский ученый «отец структурного программирования» Эдсер Дейкстра сформулировал утверждение: «Тестирование может показать наличие ошибок в программе, но никогда не докажет их отсутствие». Мы получили еще одно подтверждение правильности этой мысли.
5. Мы получили аргумент в пользу безмашинного тестирования. Признаки части ошибок удалось обнаружить только в трассировочных таблицах. Прогон тех же самых тестов на машине их бы не обнаружил.
6. Анализ по списку ошибкоопасных ситуаций может реально снизить потребность в тестовых прогонах.
7. Повторное тестирование — это не так уж страшно. Разные тесты нацелены на проверку разных частей программы. Исправления, как правило, затрагивают немногие части программы и, соответственно, сказываются на небольшом числе тестов.

Глава 19

Что еще можно проверить в программе?

До сих пор мы говорили главным образом об одном аспекте программы: о ее функциональности. Нас интересовал вопрос: правильно ли программа вычисляет требуемую от нее функцию? От учебных программ, в большинстве случаев, больше ничего и не требуется. Но если вы решили написать что-нибудь практически значимое, кроме функциональности и удобства придется проверить и другие аспекты программы. Потребуются следующие категории тестов:

1. Тестирование функциональности.

2. Тестирование удобства эксплуатации.

3. Тестирование на предельных объемах.

Проверьте, как ваша программа будет работать с очень большими объемами данных. Например, если вы пишете редактор, загрузите в него очень большой документ. Если программа нумерует кадры в электронной фотокамере и под номер кадра отведено 4 позиции, попробуйте сделать более 10 000 снимков. Если ваша программа руководит доставкой товара покупателю, сделайте такую покупку, которая не поместится в одном грузовике.

4. Тестирование на предельных нагрузках.

Такое тестирование имеет смысл для интерактивных программ и программ, работающих в режиме реального времени, например управляющих технологическими процессами. Подразумевается, что программа должна одновременно реагировать на большое число сигналов. Например, к базе данных одновременно обратились несколько десятков пользователей с удаленных терминалов (ситуация вполне реальна, если речь идет о продаже билетов, а за удаленными терминалами работают кассиры). Программе-авиатренажеру поступило сразу несколько команд (обучаемый вместо того, чтобы щелкать тумблерами по очереди, перебросил их все сразу ладонью). Тестировать следует даже такие ситуации, которых «никогда не будет». Например, одновременное нажатие пилотом сразу двадцати кнопок, расположенных в двух метрах друг от друга.

5. Тестирование защиты.

Тема по нынешним временам архипопулярная.

6. Тестирование производительности.

Докажите, что ваша программа не обеспечивает требуемого времени отклика, пропускной способности и т. п.

7. Тестирование требований к памяти.

Найдите условия, при которых ваша программа превысит допустимый расход оперативной или дисковой памяти, объем буферов и т. п.

8. Тестирование конфигураций.

Способна ли ваша программа работать на разных платформах, под управлением разных операционных систем и т. п.?

9. Тестирование совместимости.

Совместима ли новая версия программы с предыдущей версией или с той программой, которую она заменяет?

10. Тестирование удобства установки (настройки).

11. Тестирование надежности.

Сюда относится:

— проверка возможности обнаруживать ошибки пользователя;
— проверка возможности обнаруживать сбои аппаратуры;
— проверка возможности восстанавливать работоспособность системы после ошибок пользователя и сбоев аппаратуры;
— проверка среднего времени между отказами системы;
— проверка среднего времени восстановления системы после отказа;
— проверка количества ошибок в программе;
— проверка последствий отказов системы;
— проверка допустимости объема данных, теряемых в случае отказа;
— проверка того, что информация, которая не должна быть потеряна при отказе, действительно сохраняется;
— наличие и работоспособность функций, необходимых для обнаружения ошибок;
— наличие и работоспособность функций, необходимых для исправления ошибок;
— наличие и работоспособность функций, необходимых для обеспечения устойчивости к ошибкам.

12. Тестирование удобства обслуживания.

Сложная программа часто требует определенных действий по поддержанию своей работоспособности. Эти действия необходимо проверить.

13. Тестирование документации. Насколько она понятна, точна, однозначна и пр.

14. Тестирование процедур, которые должны выполняться человеком.

Часто разрабатываемая программа является компонентом автоматизированной системы, включающей в себя действия не только компьютера, но и обслуживающего персонала. Необходима проверка процедур, которые возлагаются на человека.

Заключение

В заключение рассмотрим набор тестов для задачи про треугольники, поставленной во введении. Проверьте, все ли нижеперечисленные ситуации вы учли (в скобках приводятся наборы входных данных, которые моли бы использоваться для проверки указанной ситуации).

1. Тест на правильный разносторонний треугольник (4, 7, 9).
2. Тест на правильный равносторонний треугольник (3, 3, 3).
3. Тест на правильный равнобедренный треугольник (4, 3, 3).
4. Еще два теста на правильный равнобедренный треугольник, в которых равные стороны вводятся в разных позициях (3, 4, 3), (3, 3, 4).
5. Тест, в котором сумма двух сторон меньше третьей (3, 4, 8).
6. Еще два теста, в которых сумма двух сторон меньше третьей и максимальная сторона стоит в разных позициях (8, 3, 4), (4, 8, 3).
7. Тест, в котором сумма двух сторон равна третьей (проверить сравнение: $<$ или $\leq$) (3, 4, 7).
8. Еще два теста, в которых сумма двух сторон равна третьей и максимальная сторона стоит в разных позициях (7, 3, 4), (4, 7, 3).
9. Три теста, в которых одна из сторон равна нулю (4, 3, 0), (0, 4, 3), (3, 0, 4).
10. Тест, в котором все стороны равны нулю (0, 0, 0).
11. Тест, в котором все стороны равны между собой, но все — отрицательны (−1, −1, −1).
12. Три теста, в которых одна из сторон отрицательна (4, 3, −2), (−2, 4, 3), (3, −2, 4).
13. По крайней мере один тест с вещественными значениями вместо целых (2,3; 3,5; 4,1).
14. По крайней мере один тест с символьными значениями вместо числовых (‘мама мыла раму’).
15. Тест из двух чисел вместо трех (3, 4).
16. Тест из четырех чисел вместо трех (3, 4, 5, 5).

Записали ли вы заранее для каждого теста не только входные значения, но и ожидаемый результат?

Последний вопрос зачастую приводит к тому, что количество набранных баллов падает до нуля. Несмотря на категорическое требование фиксировать заранее не только входы, но и ожидаемые выходы, делать это новички не любят.

Но если отвлечься от последнего вопроса, то каковы ваши конечные результаты? И как они менялись по ходу чтения этой книги? Попробуйте оценить собственный прогресс в области «тестостроения».

Напоследок три замечания.

1. Предложенная задача не содержит самую сложную для проверки конструкцию — циклы. Попробуйте оценить, каковы были бы ваши результаты, если бы программа должна была обрабатывать не одну тройку чисел, а их последовательность.
2. При желании можно написать программу, которая выдаст правильные результаты на всех вышеприведенных тестах, но будет при этом содержать ошибку.
3. На этом примере хорошо видны сравнительные затраты на проверку правильных и неправильных входных данных.

Что читать дальше?

Классика жанра до сих пор непревзойденная:

1. *Майерс Г.* Искусство тестирования программ. — М.: Финансы и статистика, 1982. — 176 с.
2. *Майерс Г.* Надежность программного обеспечения. — М.: Мир, 1980. — 360 с.

Примерно в это же время главы про тестирование и отладку появились в книгах по технологии программирования:

3. *Зиглер К.* Методы проектирования программных систем. — М.: Мир, 1985. — 328 с.
4. *Глас Р.* Руководство по надежному программированию. — М.: Финансы и статистика, 1982. — 256 с.
5. Средства отладки больших систем. — М.: Статистика, 1977. — 135 с.
6. *Хьюз Дж., Мичтом Дж.* Структурный подход к программированию. — М.: Мир, 1980. — 280 с.
7. *Ван Тассел Д.* Стиль, разработка, эффективность, отладка и испытание программ. — М.: Мир, 1985. — 332 с.

Затем в публикациях по данной тематике наступила длительная пауза. Computer science переключилась на «переваривание» феномена микроЭВМ. Остальные вопросы надолго отошли на второй план. Только через 20 лет тема тестирования и отладки вернулась в книги по технологии программирования. Вот некоторые из них:

8. *Орлов С. А.* Технологии разработки программного обеспечения. — СПб.: Питер, 2002. — 464 с.
9. *Соммервил И.* Инженерия программного обеспечения. — М.: Издательский дом «Вильямс», 2002. — 624 с.
10. *Калбертсон Р., Браун К., Кобб Г.* Быстрое тестирование. — М.: Издательский дом «Вильямс», 2002. — 384 с.
11. *Тамре Л.* Введение в тестирование программного обеспечения. — М.: Издательский дом «Вильямс», 2003. — 368 с.
12. *Канет С., Фолк Дж., Нгуен Е. К.* Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений. — Киев: ДиаСофт, 2001. — 544 с.
13. *Котляров В. П., Коликова Т. В.* Основы тестирования программного обеспечения. — М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006. — 285 с.

Минимальные системные требования определяются соответствующими требованиями программы Adobe Reader версии не ниже 11-й для платформ Windows, Mac OS, Android, iOS, Windows Phone и BlackBerry; экран 10"

Учебное электронное издание

Плаксин Михаил Александрович

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММ ДЛЯ ПРОФЕССИОНАЛОВ БУДУЩИХ И НАСТОЯЩИХ

Ведущий редактор *Е. Костомарова*
Художник *Н. Лозинская*
Корректор *Е. Клитина*
Компьютерная верстка: *С. Янковая*

Подписано к использованию 19.03.15.
Формат 125×200 мм

Издательство «БИНОМ. Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499)157-5272
e-mail: info@pilotLZ.ru, http://www.pilotLZ.ru

Плаксин Михаил Александрович — доцент кафедры математического обеспечения вычислительных систем Пермского государственного университета, к.ф.-м.н., член-корр. Академии информатизации образования РФ, специалист по ТРИЗ 3-го уровня. Автор более полутора сотен научных и методических публикаций.

Область профессиональных интересов — методика преподавания пропедевтического курса информатики, теория решения изобретательских задач (ТРИЗ), системный анализ, управление проектами, консалтинг.

Одно из главных отличий программиста-профессионала от новичка-«чайника» — понимание значимости и сложности процесса тестирования и отладки. Для новичка написание программы — это написание ее текста, для профессионала же — только 15% всей работы. Именно тестирование и отладка превращают «текст на языке программирования» в программу.

Но осознать это — мало. Надо еще научиться тестированию и отладке, дать конкретную методику, продемонстрировать эту методику на практике, показать, какие средства конкретной системы программирования могут помочь в этом процессе.

Именно об этом предлагаемая книга.

Основные ее читатели — «чайники» и те, кто превращает «чайника» в профессионала: студенты, старшеклассники, преподаватели вузов, учителя. Однако некоторые главы будут интересны и профессионалам.