4. Тестирование и отладка программы

Необходим строгий контроль правильности каждого этапа решения задачи на компьютере ввиду возможных ошибок в алгоритме, его программной реализации, в исходных данных, в их записи на машинный носитель, в работе аппаратуры, действиях пользователя и т. д.

Как бы тщательно ни проектировалась программа, она практически всегда содержит ошибки. *Отладка* (обнаружение, поиск и устранение ошибок) - самый трудоемкий и наименее изученный этап разработки программы.

Не рассчитывайте на отсутствие ошибок в программе! Планируйте ее отладку с самого начала разработки! Познакомимся с некоторыми основными принципами и приемами тестирования и отладки программ.

Чем раньше сделана и позже обнаружена ошибка, тем больше от нее вреда. Особое внимание поэтому следует уделять начальным этапам разработки, когда принимаются наиболее важные и ответственные решения.

Написав программу, просмотрите ее и проверьте, нет ли в ней типичных ошибок, например, по следующему *контрольному списку*.

1. Неправильный тип переменной.
2. Неверная запись имени.
3. Использована переменная без присваивания значения.
4. Тип присваиваемого значения не соответствует типу переменной.
5. Деление на ноль и т. п.
6. Индексы выходят за допустимые границы.
7. Нецелые индексы;
8. *Ошибка "плюс-минус один"* в индексах и циклах (индекс или число повторений цикла на единицу больше или меньше, чем нужно).
9. Бесконечный цикл.
10. Путаница между операциями < и <=, > и >=, == и =, == и !=, И и ИЛИ.
11. Ошибки в условиях: A<B<C вместо *(A<B) && (B<C);* A > B || C вместо *(A>B) || (A>C);* A==B==C вместо *(A==B) && (A==C).*
12. Ошибка в расположении и количестве операторных скобок { и }.
13. Неверное количество и типы параметров вызова подпрограммы.
14. В определении и вызовах подпрограммы (функции) не отражено, что выходные параметры передаются по ссылке (в языке С это указывается с помощью операций * и &).

Большинство из этих ошибок не обнаруживается транслятором. Дополните приведенный контрольный список характерными для себя ошибками.

Основным методом обнаружения ошибок в постановке задачи, алгоритмизации и программировании на практике является *тестирование*, т. е. выполнение алгоритма или программы со специально подобранными тестами вручную либо на компьютере. *Тест* состоит из исходных данных и ожидаемого правильного результата их обработки или какого-либо иного средства проверки правильности получаемого результата.

Проверьте правильность работы программы (хотя бы наиболее сложных ее частей) ручным тестированием с составлением трассировочной таблицы, как для алгоритма.

Целью тестирования является обнаружение максимального количества из имеющихся в программе ошибок.

Исчерпывающее тестирование на всех комбинациях исходных данных невозможно даже для простой программы ввиду астрономически большого числа таких комбинаций. Поэтому необходимо подобрать не слишком большое и в то же время достаточно представительное множество тестов, обнаруживающее возможно больше ошибок, что является неформальной и нелегкой творческой задачей.

Взгляд на тестирование, как на демонстрацию правильной работы программы, является психологически неверным, так как он нацелен, во-первых, на совершенно недостижимый результат, а, во-вторых, не на выявление ошибок, а на их сокрытие. Тестирование может доказать наличие ошибок, но не их отсутствие! Поэтому хорошим следует считать тест с высокой вероятностью обнаружения ошибок, а не такой тест, который программа заведомо отработает правильно.

Разработка тестов, как и любой творческий процесс, требует искусства и опыта. Необходим также критический, придирчивый подход к программе, чего трудно ожидать от ее автора. Поэтому при коллективной работе нежелательно, чтобы тесты для программы составлял ее автор.

Полезным дополнением к тестированию являются активно разрабатываемые сейчас математические методы *верификации* (доказательства правильности) программ, которые, однако, тоже не гарантируют отсутствия ошибок.

4.1. Проектирование тестов

Используйте в качестве тестов систематически подобранные, а не случайные входные данные! При разработке программ любая, даже не очень удачная, система лучше бессистемного подхода. Исходными данными для разработки тестов служат спецификация программы и логика программы.

Спецификация описывает входные данные, выходные данные и функции (“поведение”) программы – что она делает. *Логика* программы описывает, как программа выполняет свои функции, ее внутреннее устройство (алгоритм), и определяется схемой или текстом программы.

При разработке тестов для небольшой, например, учебной программы рекомендуется сначала использовать только спецификацию программы: программа рассматривается как черный ящик (*методы черного ящика*), а затем проанализировать логику программы для получения дополнительных тестов (*методы белого ящика*), чтобы весь набор тестов покрывал основные пути выполнения алгоритма (все пути перебрать невозможно).

Черным ящиком в кибернетике называют систему, рассматриваемую без учета ее внутреннего устройства, только с точки зрения ее внешнего

поведения, т. е. зависимости между входными и выходными данными. *Белый ящик* (в противоположность черному ящику) – это система, внутреннее устройство которой известно и учитывается при ее анализе.

В этом случае разработка тестов состоит из двух этапов.

1. Тесты черного ящика. К этой группе относятся методы разработки тестов, при использовании которых на основе спецификации программы готовится тест для каждой возможной ситуации (комбинации условий) во входных и выходных данных. Здесь учитывается каждая из областей допустимых и недопустимых значений входных данных и областей изменения выходных данных программы.

В отличие от большинства учебных студенческих программ, в реальной программе должны предусматриваться разумные действия для любых, в том числе неправильных или недопустимых, входных данных. Поэтому необходимы тесты для каждой области недопустимых значений входных данных. Такие *"неправильные" тесты* зачастую позволяют более эффективно выявлять ошибки в программе, чем *"правильные" тесты*. "Правильный" тест может охватить несколько тестовых ситуаций. "Неправильный" тест должен содержать не более одной ошибки в данных, т. к., обнаружив первую ошибку, программа изменяет режим своей работы, может не найти другие ошибки, и правильность реакции на них останется не проверенной.

Одним из методов черного ящика является *метод эквивалентного разбиения*, заключающийся в следующем. Совокупность значений входных данных (например, соответствующих определенному тесту) можно рассматривать как вектор или "точку" некоторого многомерного *пространства входных данных*, а соответствующий результат исполнения программы – как "точку" *пространства выходных данных* программы. Аналогичным образом можно говорить о *пространстве тестов*.

Два или несколько тестов считаются *эквивалентными*, если с их помощью обнаруживаются одинаковые ошибки и не обнаруживаются одинаковые ошибки (это понятие не является формальным: полной эквивалентности разных тестов не бывает). Желательно разделить все пространство тестов на *области эквивалентности* - множества эквивалентных между собой тестов. Тогда достаточно использовать по одному тесту из каждой области эквивалентности.

Метод эквивалентного разбиения удобно сочетать с *методом граничных значений*, идея которого в том, что ошибки чаще обнаруживаются не внутри, а на границах областей значений входных или выходных данных. В этом случае используются *тесты граничных условий*, расположенные на границах областей эквивалентности и с незначительным выходом за эти границы. Следует также попытаться составить тесты, заставляющие программу выйти из области возможных значений выходных данных.

Здесь проявляется один из общих законов природы – «*краевой эффект*» - возникновение аномальных, чаще всего, неприятных явлений на границах разных областей: вулканы и землетрясения на границах плит земной коры, искривление

поверхности жидкости у стенок сосуда, ухудшение дорог на границах административных районов и т. п. Как образно поётся в одной песне, «На границе тучи ходят хмуро!». В программировании краевой эффект необходимо учитывать при разработке программ и тестов для их отладки.

Можно использовать также *метод предположений об ошибке*, идея которого заключается в том, чтобы, исходя из некоторого предположения о возможной ошибке, создать такую тестовую ситуацию для отлаживаемой программы, которая могла не учитываться разработчиком программы.

Проектирование тестов методами черного ящика полезно еще до разработки алгоритмов на этапе составления внешней спецификации программы. Это способствует более четкой постановке решаемой задачи и служит критерием понимания задачи и качества спецификации. Если разработчик затрудняется определить реакцию программы на какие-либо исходные данные (результат ее работы для этого случая), то он не до конца понимает задачу и должен уточнить спецификацию.

Тесты служат не только для отладки программы, но и для оценки ее характеристик: времени работы, требуемого объема памяти и др. Для этого в определенной степени полезны и тесты со случайными данными, которые могут также иногда создать ситуации, упущенные при систематической разработке тестов.

Пример Разработка тестов. Как пример, рассмотрим построение тестов для программы решения следующей задачи. Дано количество входных чисел n, за которым следуют n вещественных чисел X_1, X_2, … , X_n. Числа разделены пробелами и/или символами перехода к новой строке. Требуется найти максимальное входное число max.

Сначала удобно разрабатывать тесты методом эквивалентного разбиения. Рассмотрим каждую величину из входных и выходных данных, как один из элементов (одну из координат) многомерного пространства данных и выделим границы областей эквивалентности по этой координате (см. таблицу).

Таблица Области эквивалентности для тестов (тестовые ситуации)

Входное или выходное значение или условие	***«Правильные» классы эквивалентности***	***«Неправильные» классы эквивалентности***
Заданное количество чисел n	$1 \le n \le$ MAXINT (1) , n = 0 (2)	n < 0 (3), n > MAXINT (4)
Фактическое количество входных чисел	количество чисел = n (5)	количество чисел < n (6), количество чисел > n (7)
Входные числа	нулевые (8), отрицательные (9), положительные (10), разных знаков (11), целые (12), нецелые (13)	отсутствуют (14), > MAXINT (15)
Значение max	Max < 0 (16), max = 0	-

	(17), max > 0 (18)	
Сообщения программы	Значение max (19), “Нет чисел” (20)	“Недопустимое число” (21), “Неверное количество чисел” (22)

Каждую область эквивалентности можно рассматривать как некоторую тестовую ситуацию, которую необходимо создать для отлаживаемой программы хотя бы одним тестом. Номера тестовых ситуаций указаны в скобках в таблице выше и в тексте. Константа MAXINT обозначает максимальное значение целого числа (того типа, который используется в программе).

“Неправильные” классы эквивалентности, соответствующие “неправильным” тестам, выделены в таблице в отдельный столбец, поскольку каждый тест может создавать несколько “правильных” тестовых ситуаций, но не более одной “неправильной”.

Первой входной величиной является количество чисел n, которое должно находиться в диапазоне от 0 до MAXINT. Для этой величины имеются четыре области эквивалентности - две “правильные”: $1 \le n \le$ MAXINT (1) и n = 0 (2) и две “неправильные”: n < 0 (3), и n>MAXINT (4). Область n = 0 выделена отдельно, т. к. в этом случае программа не может определить максимальное число и должна выдать сообщение об отсутствии чисел. Аналогично выявляются тестовые ситуации с номерами до 18.

Каждое предусмотренное в программе сообщение должно хотя бы один раз выводиться при ее отладке и поэтому образует самостоятельную тестовую ситуацию (от 19 до 22), причем сообщениям об ошибках соответствуют “неправильные” ситуации (21, 22).

Желательно, кроме других значений, тестировать программу для граничных значений: n = -1 (23), 0 (24), 1 (25), MAXINT (26), MAXINT+1 (27).

Исходя из предположения о возможных ошибках, желательно проверить программу в ситуациях, когда максимальное число расположено в начале последовательности (28), внутри нее (29) и в конце (30).

Затем разрабатывается по возможности минимальный набор тестов, обеспечивающий создание каждой из выявленных тестовых ситуаций (таблица ниже).

Таблица Тесты, разработанные методами черного ящика

Тест	Входные данные: n X_1 X_2 … X_n	Ожидаемый результат	Охватываемые ситуации
1	2 0 0	0	1, 5, 8, 12, 16, 18
2	3 -1 -2 -3	-1	1, 5, 9, 12, 15, 19, 28
3	3 2 1 3	3	1, 5,10, 12, 18, 19, 30
4	3 0 2 -1	2	1, 5, 11, 12, 17, 19, 29
5	1 2.1	2.1	1, 5, 10, 13, 18, 19, 25

6	1 MAXINT	MAXINT	1, 5, 10, 12, 26
7	0	Нет чисел	2, 5, 20, 24
8	Пустой файл	Нет чисел	14, 19
9	-1 2	Недопустимое число	3, 20, 22, 23
10	2 1	Неверно кол. чисел	1, 6, 22
11	1 2 3	Неверно кол. чисел	1, 7, 22, 25
12	1 MAXINT+1	Недопустимое число	1, 5, 15
13	MAXINT+1	Недопустимое число	4, 21, 22, 27

После составления теста определяются охватываемые им тестовые ситуации, которые указаны в правом столбце таблицы. Затем выявляются не охваченные им ситуации и подбираются тесты для этих ситуаций до тех пор, пока не будут хотя бы по одному разу охвачены все ситуации.

После этого методами белого ящика проверяется полнота разработанного набора тестов.

2. Тесты белого ящика. В этой группе методов набор тестов строится таким образом, чтобы он удовлетворял выбранному критерию покрытия алгоритма, т. е. в достаточной мере охватывал логику программы – возможные пути выполнения алгоритма.

Критерий *покрытия операторов* требует, чтобы каждый оператор программы выполнялся при ее отладке хотя бы один раз.

Более сильным критерием является *покрытие решений* (или *покрытие переходов*), т.е. выполнение во время отладки программы каждого условного перехода (разветвления алгоритма) в каждом возможном направлении. Он включает в себя покрытие операторов.

Критерий *комбинаторного покрытия условий* требует, чтобы для составных условий в циклах и ветвлениях во время отладки тестировались все возможные комбинации значений. Этот критерий включает оба предыдущих критерия.

В данном примере отсутствует текст программы или алгоритм ее работы, необходимые для составления тестов белого ящика. Поэтому рассмотрим разработку таких тестов лишь в общих чертах.

Если, например, в отлаживаемой программе имеется фрагмент

```
for (vn=v[k]; vn<n && (g[j][vn]==0 || k>1 && vn==v[k-2]); vn++) ;
if (vn<n && k<*dcmin)                                          */
```

содержащий шесть элементарных условий:

vn<n, g[j][vn]==0, k>1, vn==v[k-2], vn<n, k<*dcmin,

то каждое из них удобно выписать в отдельную строку таблицы вида:

Элементарное условие	Номера тестов	
	Ложь	Истина

vn<n	1, 3, 5	1, 2, 3, 7
. . .	. . .	. . .

Для каждого условия в таблице указаны номера тестов, при пропуске которых это условие принимает значения “истина” и “ложь”. При некоторых тестах условие может принимать оба этих значения в разные моменты. Необходимо убедиться в том, что при отладке каждое элементарное условие программы принимает оба возможных значения хотя бы по одному разу.

Тесты должны охватывать основные пути выполнения алгоритма. Например, желательно (если это возможно), чтобы при тестировании число повторений каждого цикл равнялось нулю, единице и максимальному значению.

Желательно проверить чувствительность алгоритма к особым значениям входных данных (например, нулевым, отрицательным, пустому файлу и т. п.), а также выход за верхние и нижние границы индексов и диапазона числовых значений. При необходимости следует разработать для этого дополнительные тесты.

4.2. Поиск и устранение ошибок

Для тестировании программы необходимо иметь достаточно информации о входных, промежуточных и выходных данных при ее выполнении, необходимой для выявления и локализации (поиска) ошибок. Для обнаружения ошибок тщательно проверяйте результаты программы! Ищите малейшие отклонения от ожидаемого результата, которые служат симптомами ошибок. Отклонения могут вызываться ошибками не только в программе, но и в тестах. Выявив симптомы, можно приступать к поиску ошибки, т. е. определению ее причины и местоположения в программе.

Ищите ошибку, прежде всего, методами логических рассуждений! Метод *индукции* подразумевает выявление закономерностей в симптомах, выдвижение на их основе гипотезы об ошибке и объяснение симптомов с помощью этой гипотезы. В процессе *дедукции*, наоборот, сначала выдвигаются несколько гипотез о возможных причинах ошибки, а затем путем анализа симптомов исключаются неверные гипотезы.

Эффективным методом локализации ошибки является *обратная прокрутка* программы от первой точки, где был получен неверный результат, назад к тому месту программы, где она еще работала правильно и где сбилась в первый раз. Для получения дополнительной информации иногда приходится придумывать специальные тесты.

Если логический анализ не помог найти ошибку, вставьте в программу отладочный вывод дополнительных промежуточных данных и снова пропустите тот же тест. Позаботьтесь о наглядности вывода. Менее эффективны автоматические отладочные средства, например, пошаговое исполнение программы или контрольные точки. Их эффект будет выше, если

предварительно проводилось ручное тестирование программы, и поэтому разработчик хорошо знаком с деталями ее функционирования.

При исправлении ошибки остерегайтесь внести новые ошибки! Измененную программу необходимо снова тщательно тестировать!

Результаты тестирования и разработанные отладочные средства: тесты, заглушки, драйверы, операторы отладочного вывода, вспомогательные файлы и т. п., необходимо сохранять и после окончания отладки, чтобы облегчить сопровождение программы!

Стиль программирования

Разные люди по-разному – в различном стиле - напишут программу решения одной и той же задачи, даже если будут использовать одинаковые методы и алгоритмы. *Стиль программирования* – это совокупность приемов и особенностей использования средств языка для оформления программы. Стиль определяется характером языка и зависит от квалификации и вкусов программиста, а также от характера самой программы.

Тем не менее, можно сформулировать ряд общих правил хорошего стиля программирования, которых придерживается большинство квалифицированных программистов. Эти правила призваны обеспечить наглядность (читабельность) программы, что улучшает ее понимание и разработку, уменьшает количество ошибок и облегчает отладку; а также упростить изменения, модернизацию программы и ее использование в других разработках.

Эти правила касаются, прежде всего, выбора имен для функций, величин и других объектов программы; форматирования программы (использования пробелов, пустых строк, отступов в начале строки и других приемов повышения наглядности текста) и использования комментариев.

1. В программировании любая, даже не очень удачная система лучше, чем бессистемный подход. Поэтому выработайте для себя определенные стандарты записи программы и всегда в одной программе поступайте одинаково в сходных ситуациях!

Для начала придерживайтесь, например, стиля программ, приведенных в данной книге. Необходимо, однако, заметить, что этот стиль в определенной степени рассчитан на начинающих программистов. Поэтому некоторые комментарии используются, например, для пояснения использованных средств языка. Квалифицированным программистам подобные пояснения, как правило, не нужны и даже вредны – отвлекают от чтения программы.

В учебнике используются разные приемлемые варианты стиля, сравните их по наглядности и простоте изменений!

2. Используйте только осмысленные имена! Избегайте слишком коротких или длинных имен, оптимальна средняя длина от 3 до 12 символов (более длинные имена нужны для больших программ). Избегайте сходных имен, чтобы не было путаницы.

Обозначайте, например, величину сокращенным русским наименованием, записывая его латинскими буквами стандартным способом в соответствии с таблицей ниже. При сокращении учитывайте, что согласные важнее гласных, начало важнее конца.

Таблица Телеграфная запись русских букв латинскими буквами

а	a	з	z	п	p	ч	ch
б	b	и	i	р	r	ш	sh
в	v	й	j (i)	с	s	щ	sc
г	q (g)	к	k	т	t	ь	y (пусто)
д	d	л	l	у	u	ы	i (y)
е	e	м	m	ф	f	ъ	y (пусто)
ё	e (jo)	н	n	х	h (kh)	э	e
ж	g (zh)	о	o	ц	c (ts)	ю	ju (yu)
						я	ja (ya)

Примечание: В скобках приведен американский вариант.

Нежелательно использовать в именах иностранные слова, особенно бессистемно, эпизодически. Это имеет смысл лишь для слов, вошедших в русскую терминологию, или в тех случаях, когда во всей программе, включая комментарии, использован иностранный язык.

В именах, составленных из нескольких слов, слова можно начинать с большой буквы либо разделять подчеркиванием, например PoiskMinChisla или poisk_min_chisla.

3. Явно описывайте тип переменной или значения функции и т. п., избегайте использования правил умолчания!

4. Не используйте переменную для нескольких целей, не экономьте на лишней переменной!

5. Не используйте особые значения переменной с особым смыслом! Пример неудачной переменной: "dlina - длина введенного текста, причем dlina=0 обозначает ошибку ввода". Для кода ошибки лучше использовать отдельную переменную.

6. Используйте символические (именованные) константы! Понимание и изменение программы упрощается, если важной константе дать имя. В языке С имена символических констант по традиции принято писать заглавными буквами, например:

```
#define NMAX 20                    /* максимальное количество чисел */
```

7. Соблюдайте правила структурного программирования! Записывайте ветвления и циклы с помощью операторов **if**, **while**, **do-while**, **switch**, **break**, **continue** и т. п. без использования оператора **goto**.

8. Используйте ступенчатую запись программы («лесенку») для подчеркивания вложенности операторов! Каждый оператор начинайте с

новой строки. Операторы одного уровня вложенности начинайте с одинаковой позиции строки. Вложенный оператор начинайте на одну ступень правее охватывающего оператора. Возможны различающиеся в деталях системы ступенчатой записи. Выбирайте какую-нибудь одну и всегда ее используйте.

9. Избегайте запутанных приемов, "трюков"! Не жертвуйте легкостью чтения программы ради экономии на мелочах. Избегайте запутанных преобразований типов данных, особенно неявных. Не изменяйте значение параметра в теле цикла, не используйте его значение после выхода из цикла.

10. Комментируйте программу! Данные - ключ к пониманию программы! Обязательно поясняйте смысл каждой переменной, назначение каждой самостоятельной части программы: подпрограммы, блока, цикла, ветви условного оператора и т. п. В начале программы и каждого ее модуля полезен *вводный комментарий*: программист, дата, решаемая задача, используемый метод, перечень используемых подпрограмм, имеющиеся отклонения от принятых стандартов, укрупненный алгоритм на псевдокоде и т. п. Полезными комментариями являются соблюдающиеся в определенных точках программы условия, инварианты циклов. Четко отделяйте комментарий от основного текста, чтобы он не мешал читать программу.