## Тема 2.3. Основные понятия алгоритмизации и императивного программирования

[**2.3.3. Разработка программы**](#_2.3.1._Этапы_решения)

[**2.3.2. Базовые алгоритмические структуры**](#_2.3.2._Базовые_алгоритмические)

[**2.3.1. Этапы решения задач**](#_2.3.3._Разработка_программы)

**[2.3.4. Контрольные вопросы по теме «Основные понятия алгоритмизации и](#_2.3.4._Контрольные_вопросы)**

**[императивного программирования»](#_2.3.4._Контрольные_вопросы)**

**[2.3.5. Тестовые задания по теме «Основные понятия алгоритмизации и](#_2.3.5._Тестовые_задания)**

**[императивного программирования»](#_2.3.5._Тестовые_задания)**

### **2.3.1. Этапы решения задач**

  Общая схема решение задач с помощью компьютера включает в себя следующие основные этапы, часть из которых осуществляются до использования компьютера:

1. **Постановка задачи**. Этап включает в себя: сбор информации о задаче; определение конечных целей решения задачи; определение формы выдачи результатов; описание данных.
2. **Анализ и исследование задачи**. На этом этапе анализируются существующие аналогичные задачи; производится подбор технических и программных средств; разрабатывается математическая модель задачи; осуществляется формализация; определяются структуры данных.
3. **Разработка алгоритма**. Этап заключается в выборе формы записи алгоритма и в последующем процессе разработки алгоритма.
4. **Программирование.** На этом этапе вначале осуществляется выбор алгоритмического языка и уточнение способов организации данных, а затем разрабатывается код программы, описывающий разработанный алгоритм.
5. **Тестирование и отладка**. При тестировании и отладке выявляют синтаксические, семантические (смысловые) и логические ошибки, допущенные при разработке алгоритма и программировании. Анализ результатов тестирования позволяет устранить все выявленные семантические и логические ошибки.
6. **Анализ результатов решения задачи**. На этом этапе осуществляется прогон программы при реальных исходных данных. В результате анализа результатов расчета возможно уточнение математической модели и повторение этапов 2-5.
7. **Сопровождение программы**. Этот этап относится к программе, находящейся в рабочей эксплуатации. При передаче программы в эксплуатацию проводится составление документации, включающей описание задачи, ее математическую модель, алгоритм и программу. Также здесь приводятся наборы тестов и инструкций по использованию.

Одними из самых трудоемких и ответственных этапов являются этапы алгоритмизации и программирования. Процесс алгоритмизации заключается в опи­сании необходимой последовательности действий, с помощью которой можно однозначно реализовать выбранный метод решения задачи. На практике только очень простые задачи представляются в виде известной последовательности арифметических или логических действий. Для боль­шинства задач перед написанием программы требуется разработать соответствующую последовательность действий, приводящую к решению задачи, то есть алгоритм ее решения. Причем, при разработке алгоритма сложной задачи це­лесообразно провести декомпозицию вычислительного процесса, составить укрупненную схему алгоритма с це­лью выявления типовых участков алгоритма и использования для их реализации стандартных или ранее разработанных алгоритмов (**процедурное программирование**). Заметим, что время, потраченное на разработку вначале укрупненного, а затем детального алгоритма, полно­стью окупается при программировании и отладке программы.

**Алгоритм** можно определить как точное предписание (действие, группу действий), определяющее **процесс преобразования исходных данных в результат.** Из определения алгоритма вытекают и его основ­ные свойства:

* **детерминированность** - однозначность получения результата при одних и тех же исходных данных;
* **результативность** – обязательность получения искомого ре­зультата за конечное число шагов;
* **массовость** – возможность получения результата при различ­ных исходных данных рассматриваемого класса задач;
* **дискретность** – возможность разбиения алгоритма на отдель­ные элементарные действия, позволяющие рассматривать ал­горитм с различным уровнем детализации.

Существуют различные способы описания алгоритмов. На прак­тике наиболее распространены следующие формы представления алгоритмов:

* [**словесная**](http://www.tomsk.ru/Books/informatica/theory/chapter7/1_7_5.html)**-** последовательность действий, описанная на естественном языке;
* [**графическа*я***](http://www.tomsk.ru/Books/informatica/theory/chapter7/1_7_6.html) **-** изображение в виде схемы, содержащей функциональные общепринятые **графические блоки** алгоритма;
* [**псевдокодов**](http://www.tomsk.ru/Books/informatica/theory/chapter7/1_7_7.html) **-** полуформализованное описание алгоритма на условном алгоритмическом языке, включающее в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения;
* [**программная**](http://www.tomsk.ru/Books/informatica/theory/chapter7/1_7_12.html) - текст программы на языке программирования.

Рассмотрим перечисленные формы представления алгоритмов более подробно.

***Словесный способ записи алгоритмов***, представляет собой описание последовательных этапов обработки данных и имеет произвольную форму изложения на естественном языке.

**Пример 2.3.1-1.** Запись алгоритма нахождения наибольшего общего делителя (**НОД**) двух натуральных чисел (алгоритм Эвклида) может выглядеть следующим образом (рис.2.3.1-1):

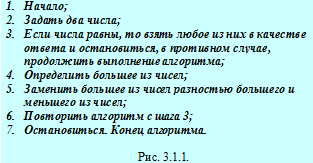


Рис. 2.3.1-1

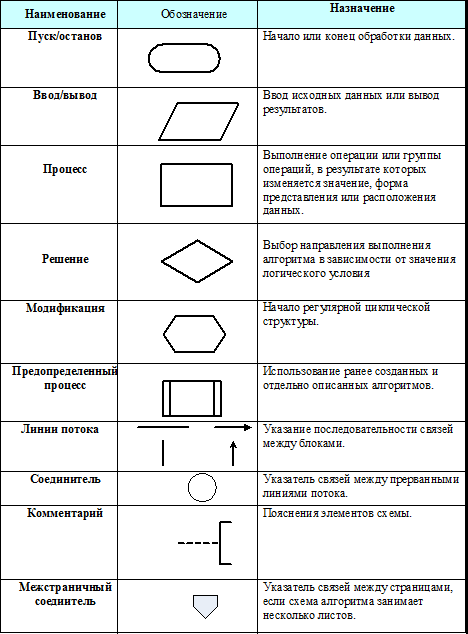
Описанный алгоритм применим к любым натуральным числам и должен приводить к решению поставленной задачи. Убедитесь в этом, определив самостоятельно с помощью этого алгоритма наибольший общий делитель чисел **125** и **75**.

Словесный способ не имеет широкого распространения, поскольку такие описания:

* строго не формализуются;
* страдают многословностью записей;
* допускают неоднозначность толкования отдельных предписаний.

***Графический способ представления алгоритмов*** является более компактным и наглядным по сравнению со словесным представлением. При графическом представлении алгоритм изображается в виде последовательности связанных между собой **функциональных блоков**, каждый из которых соответствует выполнению одного или нескольких действий. Такое графическое представление называется **схемой алгоритма**. В схеме алгоритма каждому типу действий (вводу исходных данных, вычислению значений выражений, проверке условий, управлению повторением действий, окончанию обработки и т.п.) соответствует геометрическая фигура, представленная в виде **блочного символа**. В соответствии с логикой решения конкретной задачи, блоки связы­вают друг с другом линиями, которые называются **линиями потока**. В табл.2.3.1-1 приведен перечень основных обозначений, принятых в схемах алгоритмов.

Таблица 2.3.1-1



**Пример 2.3.1-2.Рассмотрим запись алгоритма нахождения НОД двух натуральных чисел (алгоритм Эвклида) с помощью схемы алгоритма:**

|  |
| --- |
| Рис-3-1-2 |

Рис 2.3.1-2.

***Псевдокод*** представляет собой систему обозначений и правил, предназначенную для единообразной записи алгоритмов**.** Псевдокод занимает промежуточное место между естественным и формальным языками. С одной стороны, он близок к обычному естественному языку, поэтому алгоритмы, записанные на нем, могут читаться как обычный текст. С другой стороны, в псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи.

В псевдокоде не приняты строгие синтаксические правила для записи команд, присущие формальным языкам, что облегчает запись алгоритма на стадии его проектирования и дает возможность использовать более широкий набор команд, рассчитанный на абстрактного исполнителя. Однако в псевдокоде обычно имеются некоторые конструкции, присущие формальным языкам, что облегчает переход от записи на псевдокоде к записи алгоритма на формальном языке. В частности, в псевдокоде, так же, как и в формальных языках, есть служебные слова**,** смысл которых определен раз и навсегда. Они выделяются в печатном тексте жирным шрифтом, а в рукописном - подчеркиваются.

Из рассмотренных выше способов описания алгоритмов самым распространенным и наиболее наглядным является графический способ, поэтому в дальнейшем все алгоритмы будут изображаться графически, в виде схем алгоритмов.

### **2.3.2. Базовые алгоритмические структуры**

***Базовыми алгоритмическими структурами*** принято называть подмножество алгоритмических структур, которые позволяют составить алгоритм решения сколь угодно сложной задачи. Эти структуры могут быть вы­браны разработчиком программы в зависимости от специфики решаемой задачи. К простейшим базовым алгоритмическим структурам относятся:

* последовательные структуры;
* разветв­ляющиеся структуры;
* циклические структуры регулярного типа;
* циклические структуры итеративного типа.

В свою очередь из базовых алгоритмических струк­тур могут быть составлены алгоритмы решения простейших типовых за­дач, часто встречающихся в качестве составляющих при решении мно­гих инженерных задач. Такие типовые алгоритмы будем в дальнейшем называть базовыми алгоритмами.

***Последовательными*** называются такие алгоритмические струк­туры, в которых функциональные элементарные блоки вы­полняются в том порядке, в котором они записаны или изо­бражены на схеме алгоритма. Такая структура может быть составлена из совокупности блоков «**Пуск/останов**», «**Ввод/вывод**», а также блоков «**Процесс**», используемых для проведения вычислений (рис.2.3.2-1).

|  |
| --- |
| Рис-3-1-3 |

Рис. 2.3.2-1

***Разветвляющимися*** называются такие алгоритмические струк­туры, в которых порядок выполнения функциональных блоков определяется значе­ниями логических выражений, использующихся для организации разветвления. Разветвляющийся алгоритм может состоять из нескольких ветвей, каждая из которых может содержать любую, сколь угодно сложную, алгоритми­ческую структуру. В процессе работы алгоритма в первую очередь вычисляются **логиче­ские выражения**. Если логическое выражение принимает значение «**Истина**», то выполняется часть алгоритма, расположенная по ветви «**Да**», если принимает значение «**Ложь**», то – часть алгоритма по ветви «**Нет**».

Анализ разветвляющихся алгоритмов, применяемых в практиче­ских задачах, показывает, что наиболее часто используются три типа разветвлений:

**Стандартное разветвление** содержит функциональные блоки как в ветви «**Да**», так и в ветви «**Нет**» (рис.2.3.2-2).

|  |
| --- |
| Рис-3-1-4 |

Рис.2.3.2-2

**Усеченное разветвление** содержит функциональные блоки только в ветви «**Да**» или только в ветви «**Нет**» (рис.2.3.2-3, 2.3.2-4).

|  |  |
| --- | --- |
| Рис-3-1-5  Рис. 2.3.2-3 | Рис-3-1-6  Рис.2.3.2-4 |

**Вложенное разветвление** содержит одно или несколько дополнительных разветвлений (рис.2.3.2-5).

|  |
| --- |
| Рис-3-1-7 |

Рис. 2.3.2-5

В соответствии с основным принципом **структурного программиро­вания**, все разветвляющиеся структуры, как и все другие алгоритмические структуры, должны иметь **один вход** и **один выход**.

**Циклическими** называются структуры, в которых предусмотрена возможность многократного повторения выполнения участка алгоритма. Этот участок называется **телом цикла**. Различают циклические структуры двух видов: с заранее известным и с заранее неизвестным числом повторений цикла.

Циклические структуры, в которых число повторений цикла заранее известно или может быть определено до начала цикла, называются **регулярными циклическими структурами** (рис.2.3.2-6).

В блоке организации цикла используется специальная пере­менная, которая предназначена для определения условия останова цикла (i). Эта переменная называется **параметром цикла**. Блоки, следующие за заголовком цикла, составляют **тело цикла**. Тело цикла выполняется для всех значений параметра цикла i, начинаю­щегося со значения m1 и изменяющегося с шагом m3 до значения m2.

 Если из условия задачи следует, что число повторений цикла за­ранее не определено, а вычисляется в процессе выполнения алгоритма, то условие выхода из цикла должно быть определено в процессе его вы­полнения. При этом важно, чтобы в условие выхода из цикла входила перемен­ная, значение которой изменялось бы в теле цикла, иначе выполнение цикла будет бесконечным.

 Циклическая структура, в которой число повторений цикла заранее неизвестно, а определяется только в процессе выполнения алгоритма, называется **итеративной циклической структурой**.

В зависимости от места расположения условия продолжения цикла (или выхода из цикла) **итеративные циклические алгоритмы** подразделяются на два вида: **с предусловием** и **с постусловием**.

|  |
| --- |
| Рис-3-1-8 |

Рис.2.3.2-6

При организации цикла с предусловием (рис.2.3.2-7) блоки тела цикла, следующие за блоком, в котором проверяется условие выхода из цикла, выполняются всякий раз, когда условие L принимает значение «**Истина**». При первом невыполнении этого условия происходит выход из цикла. Таким образом, возможен случай, когда тело цикла не будет выполнено ни разу. Поэтому циклические структуры с известным числом повторений (регулярные циклы) относятся к числу циклических алгоритмов с предусловием.

При организации циклов с постусловием, для которых условие выхода из цикла (или повторения тела цикла) проверяется после выполнения цикла (рис.2.3.2-8), цикл всегда выпол­няется хотя бы один раз, независимо от значения L, и только после его выполнения принимается решение – продолжать выполнение цикла или выйти из него.

|  |
| --- |
| Рис-3-1-9 |

Рис. 2.3.2-7

|  |
| --- |
| Рис-3-1-10 |

Рис.2.3.2-8

### **2.3.3. Разработка программы**

Средства и приемы разработки программ будут рассмотрены в последующих темах, здесь рассмотрим некоторые проблемы, возникающие при разработке простейших программ.

Во-первых, предполагается, что при разработке программ используется технологии **императи́вного программи́рования.**

***Императи́вное программи́рование***  – это [парадигма программирования](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), которая описывает процесс вычисления в виде [инструкций](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), изменяющих состояние программы. Императивная программа очень похожа на приказы, выражаемые повелительным наклонением в [естественных языках](http://ru.wikipedia.org/wiki/%D0%95%D1%81%D1%82%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA), то есть это последовательность команд, которые должен выполнить [компьютер](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80).

[Императивные языки программирования](http://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) противопоставляются [функциональным](http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) и [логическим](http://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) языкам программирования. Функциональные языки не представляют собой последовательность инструкций и не имеют глобального состояния. Логические языки программирования обычно определяют ***что*** надо вычислить, а не ***как*** это надо делать.

***Паради́гма программи́рования***  – это совокупность идей и понятий, определяющих стиль написания программ. [Парадигма](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0) в первую очередь определяется базовой программной единицей. Кроме того, парадигма программирования определяет и то, в каких терминах [программист](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82) описывает логику программы. Например, в [императивном программировании](http://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) программа описывается как последовательность действий, а в [функциональном программировании](http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) представляется в виде выражения и множества определений функций. В [объектно-ориентированном программировании](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (в дальнейшем ООП) программу принято рассматривать как набор взаимодействующих объектов. ООП, в основном, есть по сути императивное программирование, дополненное принципом [инкапсуляции](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) данных и методов в объект (принцип модульности) и наследованием (принципом повторного использования разработанного функционала).

Императивное программирование – это исторически первая методология программирования, которой пользовался ***каждый*** программист, программирующий на любом из «массовых» языков программирования – Basic, Pascal, C.Оно ориентировано на классическую фон Неймановскую модель, остававшуюся долгое время единственной аппаратной архитектурой.

Методология императивного программирования характеризуется принципом ***последовательного изменения состояния вычислителя пошаговым образом***. При этом управление изменениями полностью определено и полностью контролируемо. Если под вычислителем понимать современный компьютер, то состоянием такого вычислителя будут значения всех ячеек памяти, состояние процессора (в том числе – указатель текущей команды) и всех сопряженных устройств. И выполнение программы, написанной на императивном языке программирования, сводится к изменению состояния таких ячеек памяти – изменению состояния компьютера.

В качестве ***математической модели*** императивное программирование использует машину Тьюринга-Поста – абстрактное вычислительное устройство, предложенное на заре компьютерной эры для описания алгоритмов.

На всякий случай напоминаем, что вычислительным системам, основанным на архитектуре, предложенной Джоном фон Нейманом (JohnvonNeumann) в 40-х годах ХХ века присущи следующие характеристики:

* ***Принцип программного управления***. Из него следует, что программа состоит из набора команд, которые выполняются процессором автоматически друг за другом в определенной последовательности. Последовательность выполнения команд определяется *алгоритмом* работы и связанной с ним программой на языке программирования.
* ***Принцип однородности памяти***. Программы и данные хранятся в одной и той же памяти. Поэтому компьютер не различает, что хранится в данной ячейке памяти — число, текст или команда
* ***Принцип адресности***. Структурно основная память состоит из перенумерованных ячеек; процессору в произвольный момент времени доступна любая ячейка. Отсюда следует возможность давать имена областям памяти, так, чтобы к запомненным в них значениям можно было впоследствии обращаться или менять их в процессе выполнения программ с использованием присвоенных имен.

Процесс разработки программы можно выразить следующей формулой:

**Разработка=Изготовление + Доказательство правильности**

Эта простая формула указывает на то, что после того, как программа написана, мы должны убедиться в ее правильности. Следует заметить, что наличие ошибок в только что разработанной программе - это вполне нормальное и закономерное явление. Практически невозможно составить реальную, достаточно сложную программу без ошибок. Кроме того, нельзя делать вывод, что программа правильна, лишь на том основании, что она выполняется, и выдает результаты, поскольку эти результаты еще не обязательно правильные. В программе может оставаться большое количество логических ошибок. Ответственные участки программы рекомендуется проверять отдельно при помощи тестов, ориентированных на проверку конкретного участка программы.

Проконтролировать программу можно еще до ввода в компьютер, то есть за столом, с помощью просмотра, проверки и прокрутки.

**Просмотр** текста программы предусматривает **обнаружение описок и расхождений с алгоритмом**. Просматривается **организация всех циклов** с тем, чтобы убедиться в правильности операторов, задающих кратности циклов. Полезно посмотреть еще раз **условия в условных операторах, аргументы в обращениях к подпрограммам** и т.п.

При **проверке программы** программист по тексту программы мысленно воспроизводит тот вычислительный процесс, который определяет программа, после чего сверяет его с требуемым процессом. На время проверки нужно "забыть", что должна делать программа, и "узнавать" об этом по ходу её проверки. Только после окончания проверки программы можно "вспомнить" о том, что она должна делать и сравнить реальные действия программы с требуемыми.

Основой **прокрутки** является **имитация выполнения программы.**  Для выполнения прокрутки используют простейшие исходные данные и над ними производят все необходимые вычисления, следуя тексту программы. **Прокрутка —** это трудоемкий процесс, поэтому ее следует применять лишь для контроля логически сложных участков программ.

Следующим этапом контроля правильности программы является **отладка** и **тестирование** на компьютере.

**Отладка программы —** это процесс поиска и устранения ошибок в программе, производимый по результатам её прогона на компьютере, а **тестирование —** это испытание, проверка правильности работы программы в целом, либо её составных частей. Отладка и тестирование — это два четко различимых и непохожих друг на друга этапа, поскольку при **отладке** происходит локализация и устранение синтаксических ошибок и явных ошибок кодирования, а в процессе **тестирования** проверяется работоспособность программы, не содержащей явных ошибок. Таким образом, тестирование устанавливает факт наличия ошибок, а отладка выясняет ее причину.

В современных программных системах отладка осуществляется часто с использованием специальных программных средств, называемых отладчиками. **Программа-отладчик** обычно обеспечивает следующие возможности:

* **пошаговое исполнение программы** с остановкой на каждой команде(операторе);
* **просмотр текущего значения любой переменной или нахождение значения любого выражения**, в том числе, с использованием стандартных функций; при необходимости можно установить новое значение переменной;
* **установку в программе "контрольных точек"**, т.е. точек, в которых программа временно прекращает свое выполнение, так что можно оценить промежуточные результаты, и др.

Но даже если вы не используете средства программы-отладчика, при отладке программ важно помнить что:

* в начале процесса отладки следует использовать **простые тестовые данные**;
* возникающие затруднения необходимо четко **разделять** и **устранять строго поочередно**;
* **не нужно считать причиной ошибок компьютер**, поскольку современные компьютеры и трансляторы обладают чрезвычайно высокой надежностью.

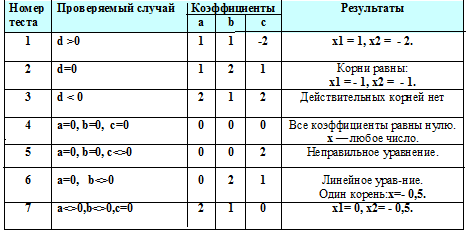
Как бы ни была тщательно отлажена программа, решающим этапом, устанавливающим ее пригодность для работы, является контроль программы по результатам ее выполнения на системе тестов.

Под **тестом** понимается некоторая совокупность исходных данных для программы и точное описание результатов, которые должна выработать программа при этих данных, в том виде, котором эти результаты должны быть выданы программой. Если для выбранной системы тестовых исходных данных программа дает правильные результаты, то программу условно можно считать правильной, поскольку тестирование может показать лишь наличие ошибок, но не их отсутствие. Нередки случаи, когда новые входные данные вызывают "ошибку" или получение неверных результатов работы программы**,** которая уже считалась полностью отлаженной. Для реализации метода тестов должны быть изготовлены или заранее известны эталонные результаты.

Тестовые данные должны обеспечить проверку всех возможных условий возникновения ошибок. При проведении тестирования руководствуются следующим:

* первый тест должен быть **максимально прост**, чтобы проверить, работает ли программа вообще;
* должна быть испытана **каждая ветвь** алгоритма;
* очередной тестовый прогон должен контролировать нечто такое, что еще **не было проверено** на предыдущих прогонах;
* арифметические операции в тестах должны **предельно упрощаться** для уменьшения объема вычислений;
* в тестовых примерах количество элементов последовательностей, точность для итерационных вычислений, количество проходов цикла должны задаваться из соображений **сокращения объема вычислений;**
* тестирование должно быть **целенаправленным и систематизированным**, поскольку случайный выбор исходных данных приводит к трудностям в проверке ожидаемых результатов, а также при случайном выборе тестовых данных многие ситуации могут оказаться непроверенными;
* усложнение тестовых данных должно происходить постепенно.

**Пример 2.3.3-1.** Разработать систему тестов для задачи нахождения корней квадратного уравнения  При нахождении корней квадратного уравнения возможны следующие случаи:



Процесс тестирования можно разделить на три этапа:

**1.** Проверка **в нормальных условиях.** Предполагает тестирование на основе данных, которые характерны для реальных условий функционирования программы.

**2.** Проверка **в экстремальных условиях.** Тестовые данные включают **граничные  значения** области изменения входных переменных, которые должны восприниматься программой как правильные данные. Типичными примерами таких значений являются очень маленькие или очень большие числа и отсутствие данных. Еще один тип экстремальных условий — это **граничные объемы** данных, когда массивы состоят из слишком малого или слишком большого числа элементов.

**3. Проверка в исключительных ситуациях.** Проводится с использованием данных, значения которых лежат **за пределами допустимой области изменений.** Известно, что все программы разрабатываются в расчете на обработку ограниченного набора данных.

При тестировании, например, важно получить ответ на следующие вопросы:

* что произойдет, если программе, не рассчитанной на обработку отрицательных и нулевых значений переменных, в результате какой-либо ошибки придется иметь дело как раз с такими данными?
* как будет вести себя программа, работающая с массивами, если количество их элементов повысит величину, указанную в объявлении массива?
* что произойдет, если числа будут слишком малыми или слишком большими?

Наихудшая ситуация складывается тогда, когда программа воспринимает неверные данные как правильные и выдает неверный, но правдоподобный результат. Программа должна сама отвергать любые данные, которые она не в состоянии обрабатывать правильно. При этом ошибки могут быть допущены на всех этапах решения задачи — от ее постановки до оформления.

Обычно синтаксические ошибки выявляются на этапе трансляции. К синтаксическим ошибкам, например, относятся:

* пропуск знака пунктуации;
* несогласованность скобок;
* неправильное формирование оператора;
* неверное образование имен переменных;
* неверное написание служебных слов;
* отсутствие условий окончания цикла;
* отсутствие описания массива и т.п.

Существует множество ошибок, которые транслятор выявить не в состоянии. Ниже приведены некоторые типы ошибок, не выявляемые транслятором.

**Логические ошибки:**

* неверное указание ветви алгоритма после проверки некоторого условия;
* неполный учет возможных условий;
* отсутствие в программе одного или более блоков алгоритма.

**Ошибки в циклах:**

* неправильное указание начала цикла;
* неправильное указание условий окончания цикла;
* неправильное указание числа повторений цикла;
* бесконечный цикл.

**Ошибки ввода-вывода:**

* неправильное задание типа данных;
* организация считывания меньшего или большего объёма данных, чем требуется;
* неправильное редактирование данных при выводе.

**Ошибки в использовании переменных:**

* использование переменных без указания их начальных значений;
* ошибочное указание одной переменной вместо другой.

**Ошибки при работе с массивами:**

* массив неправильно описан;
* массив предварительно не обнулен;
* индексы следуют в неправильном порядке.

**Ошибки в арифметических операциях:**

* неверное указание типа переменной (например, целочисленный тип вместо вещественного);
* неверное определение порядка действий;
* деление на нуль;
* извлечение квадратного корня из отрицательного числа;
* потеря значащих разрядов числа.

В данном перечне, приведена только малая часть характерных ошибок. Полный перечень ошибок, как правило, приводится в литературе, посвященной полному описанию языка программирования.

### **2.3.4. Контрольные вопросы по теме**

### **«Основные понятия алгоритмизации и императивного программирования»**

1. Какие основные этапы включает в себя решение задач на компьютере?
2. Какие этапы компьютерного решения задач осуществляются без участия компьютера?
3. Что называют математической моделью объекта или явления?
4. Почему невозможно точное исследование поведения объектов или явлений?
5. Какие способы моделирования осуществляются с помощью компьютера?
6. Из каких последовательных действий состоит процесс разработки программы?
7. Что называется алгоритмом?
8. Какими основными свойствами должен обладать алгоритм?
9. Какие существуют способы описания алгоритмов?
10. Какими графическими символами принято изображать в схемах алгоритма?
11. В чем отличие циклической структуры с предусловием от циклической структуры с постусловием?
12. Что такое параметр цикла?
13. В чем отличие регулярной циклической структуры от итеративной?
14. Доказывает ли получение правдоподобного результата правильность программы?
15. Какие ошибки могут остаться не выявленными, если не провести проверку (просмотр, прокрутку) программы?
16. Чем тестирование программы отличается от её отладки?
17. Можно ли с помощью тестирования доказать правильность программы?
18. На какой стадии работы над программой вычисляются эталонные результаты тестов?
19. Назовите основные этапы процесса тестирования.
20. В чём заключается отличие синтаксических ошибок от семантических?
21. О чём свидетельствует отсутствие сообщений машины о синтаксических ошибках?
22. Какие разновидности ошибок транслятор не в состоянии обнаружить?

### **2.3.5. Тестовые задания по теме «Основные понятия алгоритмизации и императивного программирования»**

1. **Алгоритм – это**
2. понятное и точное предписание исполнителю совершить последовательность действий, направленных на решение поставленной задачи
3. некоторые истинные высказывания, которые должны быть направлены на достижение поставленной цели
4. отражение предметного мира с помощью знаков и сигналов, предназначенное для конкретного исполнителя
5. инструкция по технике безопасности
6. **Понятное и точное предписание исполнителю выполнить конечную последовательность команд, приводящую от исходных данных к искомому результату, называется**
7. алгоритмом
8. моделью
9. системой
10. технологией
11. **Свойствами алгоритма являются**
12. дискретность, результативность, детерминированность, массовость, понятность
13. новизна, понятность, массовость, дискретность, результативность
14. массовость, понятность, условность, четкость, однозначность
15. четкость, однозначность, массовость, дискретность, результативность
16. **Расчлененность алгоритма на отдельные шаги, возможность выполнения которых исполнителем не вызывает сомнений, отражена в свойстве алгоритма, которое называется**
17. дискретностью
18. однозначностью
19. результативностью
20. понятностью
21. **Основное свойство алгоритма, характерное только для решения задач на ЭВМ, это**
22. массовость
23. дискретность
24. понятность
25. точность
26. **Свойство алгоритма – дискретность – обозначает**
27. разбиение алгоритма на конечное число простых шагов
28. что команды должны следовать последовательно друг за другом
29. что каждая команда должны быть описана в расчете на конкретного исполнителя
30. нет верного ответа
31. **Графическое задание алгоритма – это**
32. способ представления алгоритма с помощью геометрических фигур
33. представление алгоритма в форме таблиц и расчетных формул
34. система обозначения правил для единообразной и точной записи алгоритмов и их исполнения
35. схематичное изображение в произвольной форме
36. **Выбор метода решения должен стоять перед**
37. разработкой алгоритма
38. построением математической модели
39. анализом и уточнением результатов
40. тестированием и отладкой
41. **Свойство алгоритма – массовость – обозначает**
42. что алгоритм должен обеспечивать возможность его применения для решения однотипных задач
43. что каждая команда должна быть описана в расчете на конкретного исполнителя
44. разбиение алгоритма на конечное число простых шагов
45. использование любым исполнителем
46. **Линейный алгоритм – это**
47. набор команд, которые выполняются последовательно друг за другом
48. способ представления алгоритма с помощью геометрических фигур
49. строгое движение как вверх, так и вниз
50. все ответы верны
51. **Последним этапом в решении задач на ЭВМ является**
52. анализ и уточнение результатов
53. выбор метода решения
54. постановка задачи
55. тестирование и отладка
56. **Алгоритм, который должен быть выбран при решении квадратного уравнения – это**
57. разветвляющийся
58. линейный
59. циклический регулярный
60. циклический итеративный
61. **Запись алгоритма в виде последовательности команд компьютера называется**
62. программной
63. вербальной
64. графической
65. словесной
66. **На этапе тестирования и отладки происходит**
67. обнаружение и исправление синтаксических ошибок, и доведение программы до рабочего состояния
68. получение результата
69. перевод алгоритма на алгоритмический язык
70. представление задачи в виде последовательности математических формул
71. **Второй этап решения задач на ЭВМ – это**
72. построение математической модели
73. разработка алгоритма
74. постановка задачи
75. тестирование и отладка
76. **Компьютер может отследить**
77. синтаксические ошибки
78. логические ошибки
79. фактические ошибки в формулах
80. любые ошибки
81. **Циклический алгоритм – это**
82. алгоритм, содержащий многократное повторение некоторых операторов
83. способ представления алгоритма с помощью геометрических фигур
84. алгоритм, содержащий условия
85. представление алгоритма в форме таблиц и расчетных формул
86. **Разветвляющийся алгоритм – это**
87. присутствие в алгоритме хотя бы одного условия
88. набор команд, которые выполняются друг за другом
89. многократное исполнение одних и тех же действий
90. нет верного ответа
91. Алгоритмом можно назвать
92. инструкцию по приготовлению пищи
93. правило техники безопасности
94. расписание движения поездов
95. нет верного ответа
96. Для вычисления площади треугольника по трем сторонам используется
97. линейный алгоритм
98. разветвляющийся алгоритм
99. циклический алгоритм
100. любой алгоритм
101. **Завершаемость алгоритма за конечное число шагов отражена в свойстве, которое называется**
102. результативностью
103. однозначностью
104. понятностью
105. дискретностью
106. **Алгоритмическая структура, предполагающая выполнение либо одного, либо другого действия в зависимости от истинности или ложности условия, называется**
107. разветвляющейся
108. линейной
109. повторяющейся
110. рекурсивной
111. **Когда некоторые этапы алгоритма повторяются многократно, алгоритмическая структура называется**
112. циклической
113. рекурсивной
114. разветвляющейся
115. нет верного ответа

Императивное программирование наиболее пригодно для реализации небольших подзадач, где очень важна скорость исполнения на современных компьютерах. Кроме этого, работа с внешними устройствами, как правило, описывается в терминах последовательного исполнения операций («открыть кран, набрать воды»), что делает такие задачи идеальными кандидатами на императивную реализацию.