3. Подготовка компьютерной программы, алгоритмы.

Подготовка компьютерной программы для конкретной прикладной задачи выполняется за несколько этапов. При этом составление кода осуществляется на последней стадии. Основные этапы можно сформулировать следующим образом:

I. Изучить специфику и цели решаемой проблемы и формализовать ее, т.е. ввести обозначения для объектов и параметров, в этих обозначениях сформулировать условие и определить цель задачи. До того, как начать писать непосредственно программу, надо удостовериться, что все начальные данные определены полностью и совместимы. Например, если цель – «Написать программу для решения уравнений», то это явно неполная постановка, и сразу возникают вопросы: Какого типа уравнений? Сколько уравнений? С какой точностью? и т.д.

II. Проанализировать проблему и решить ее. На этом этапе надо выбрать способ, метод решения задачи, детально описать процесс решения введенных обозначений. Такой метод решения называют *алгоритмом.*

III. Схематизировать решение задачи, т.е. описание решения разделить на простейшие шаги и определить последовательность соответствующих инструкций, каждую из которых можно было бы реализовать на компьютере.

IV. Разработанный на предыдущем этапе алгоритм перевести на подходящий язык программирования высокого уровня: ввести обозначения в соответствии с синтаксисом и выразить инструкции с помощью операторов языка. Такую Текстовую Копию программы называют *исходной программой или исходным кодом.* На этом этапе программу надо проверить вручную на корректность и на то, выполняет ли она поставленные задачи. Программист задает различные начальные данные и, выполняя каждую инструкцию вручную, проверяет истинность полученного результата. После этого текст программы набирается на компьютере с помощью Редактора.

V. Тестировка программы:

- (а) Откомпилировать программу на машинный язык. Программа на машинном языке называется *объектным кодом (object code).* На этом этапе компилятор может обнаружить в программе *синтаксические ошибки,* т.е. ошибки в грамматике языка программирования. Например, Паскаль, или C++ требуют, чтобы каждый оператор завершался символом «;». Если Вы случайно опустили этот знак, то компилятор сообщит о синтаксической ошибке. Когда все синтаксические ошибки исправлены, компилятор генерирует запускаемую программу.
- (б) Объектный код, генерированный компилятором, затем присоединяется к различным системным функциональным библиотекам с помощью программы *редактора связи (linker).* В результате создается *связный объектный код (linked object code),* который затем загружается в память с помощью программы *загрузчика (loader).*
- (в) Запустить откомпилированную, связную и загруженную программу на выполнение с контрольными начальными данными для различных начальных значений объектов и параметров и проверить корректность результата. Здесь в программе могут выявиться логические ошибки — т.е. ошибки в методе решения. Это означает, что синтаксически корректно записанный оператор выполняет неверные действия в контексте решения задачи, например, сложение двух чисел вместо вычитания. Частным случаем логической ошибки является ошибка прогона (run-time error) программы. Эти ошибки вызывают остановку программы, так как компьютер не может выполнить данную инструкцию. Типичный случай — попытка деления числа на величину с нулевым значением, или попытка доступа к несуществующему файлу. При тестировке, чтобы охватить полную область определения параметров, нужно задавать такие значения, для которых интуитивно задача имеет качественно различные решения (например, при решении квадратного уравнения — вариант: коэффициент при квадратичном члене равен нулю).

➢ (г) Если полученный результат не совпадает с ожидаемым, нужно вернуться сначала к четвертому этапу и проверить корректность программы на синтаксические и семантические ошибки. Если после этого не достигается желаемого результата, надо вернуться последовательно к третьему (возможно допущена ошибка при схематизировании) и ко второму (неправильный или неадекватный метод решения задачи) этапам, до тех пор, пока не получен удовлетворительный результат.

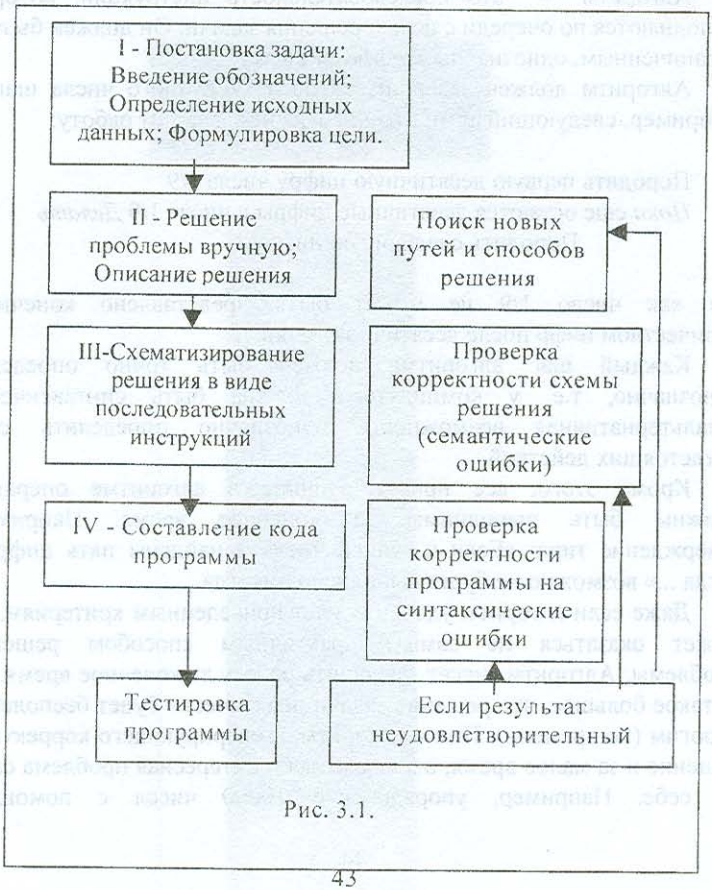

Рис. 3.1.

Теперь программу можно предложить на общее пользование, хотя если тестировка не была полна, в дальнейшем могут опять возникнуть другие логические ошибки. Именно для таких случаев очень важно иметь хорошую и полную документацию описания программы, особенно если после его составления прошло достаточно много времени.

Схема этапов решения проблемы показана на рис. 3.1.

3.1. Алгоритмы.

Алгоритм — это последовательность инструкций, которые выполняются по очереди с целью решения задачи. Он должен быть: ограниченным, однозначным, эффективным.

Алгоритм должен завершиться после конечного числа шагов. Например, следующий алгоритм никогда не завершит работу:

Породить первую десятичную цифру числа 1/9
Пока еще остаются десятичные цифры в числе 1/9 *Делать*
Породить следующую цифру

так как число 1/9 не может быть представлено конечным количеством цифр после десятичной точки.

Каждый шаг алгоритма должен быть точно определен однозначно, т.е. у компилятора должна быть синтаксически безальтернативная возможность однозначно определить суть предстоящих действий.

Кроме этого, все присутствующие в алгоритме операции должны быть выполнимы за конечное время. Например, утверждение типа: «Если в записи числа π найдены пять цифр 5, тогда ...» возможно не будет выполнено никогда.

Даже если алгоритм удовлетворяет приведенным критериям, он может оказаться не самым практичным способом решения проблемы. Алгоритм может завершить работу за конечное время, но за такое большое, что решение задачи практически будет бесполезно дорогим (по времени). Поиск алгоритма, генерирующего корректное решение и за малое время, очень важная и интересная проблема сама по себе. Например, упорядочение 10000 чисел с помощью

«хорошего» алгоритма занимает меньше секунды, тогда как «плохому» алгоритму может понадобиться больше 10 минут.

3.2. Описание алгоритма.

Для наглядности, как формулировать алгоритм в подходящем для компьютера виде, рассмотрим простой пример:

Написать программу, которая считывает несколько чисел и выдает их среднее значение.

Такая постановка задачи весьма неформальна для компьютерного решения и недостаточно полна для определения того, что же программа должна делать конкретно. Возникает несколько вопросов:

Например, откуда будут поступать данные — вводиться пользователем с клавиатуры или считываться из файла?

Очевидно, чтобы найти среднее значение последовательности чисел, сначала надо найти их сумму, а затем разделить на их полное количество. Как же будет определяться количество вводимых чисел? Эта величина может быть введена как часть начальных данных, или может быть вычислена программой по мере поступления чисел. Последний вариант подразумевает, что программа способна определить, на каком месте заканчивается ввод чисел, например, когда достигается конец при чтении файла.

Важно также знать, что должна программа сделать, если не поступило ни одного числа. В этом случае бессмысленно выводить на печать значение средней величины, и, следовательно, этот вариант надо учесть как частное решение задачи.

Теперь можно сформулировать задачу более точно:

Написать программу, которая считывает последовательность чисел и выдает их среднее значение. Все числа поступают из файла данных. Ввод завершается при достижении маркера "конец файла". Если файл пустой, соответствующее сообщение должно появиться на экране.

Алгоритм решения (вручную) очевидно можно описать так:

«Прочитать числа из файла, для вычисления среднего значения, сложить их и разделить на полное количество»

Но на практике такое описание недостаточно детализировано для реализации компьютерного алгоритма, так как компьютер воспринимает инструкции типа: «прочитать число», «сложить два числа» и т.д. И, следовательно, алгоритм должен быть описан в терминах в виде инструкций, которые могут быть реализованы на компьютере. Рассмотрим такую версию алгоритма для решения вышеприведенной задачи:

1. Выполнить предварительную инициализацию;
2. *Пока* не достигнут Конец файла *Делать*
3. {
4. Прочитать очередное число;
5. Добавить число к аккумулируемой сумме;
6. Увеличить «счетчик» вводимых чисел;
7. }
8. Вычислить среднее значение.

Нумерация шагов необязательна, но удобна для ссылок.

Обычно в компьютерных алгоритмах всегда требуется провести определенную предварительную подготовку, инициализацию до начала процесса основной обработки. Этот фактор учтен в пункте 1. Он может быть уточнен и расширен уже после составления основной части алгоритма.

Пункт 2 описывает *цикл.* Он позволяет непрерывно выполнять операторы, содержащиеся в фигурных скобках (3—7), до тех пор, пока *условие* «не достигнут конец файла» остается истинным. Скобки указывают, что утверждения между ними (4—6) воспринимаются и запускаются как одно целое, формируя т. н. *составное утверждение.* Если бы скобок не было, в цикле каждый раз выполнялось бы только утверждение 4. Приведенный способ не единственный для формирования циклов или повторов в программе.

Другие методы будут рассмотрены ниже. В *теле цикла* (4—6) каждая инструкция является *простой* выполняемой инструкцией.

Как только будет достигнут конец файла, программа передает управление следующей по порядку инструкции — вычисление средней величины (пункт 8). Этот пункт требует дальнейшего расширения, так как надо учесть вариант, когда данные не поступают вовсе (файл пуст) и среднее значение не может быть вычислено. Расширение можно представить так:

8а. Если данные не поступили
8б. Тогда вывести «Нет входных данных!»
8в. Иначе {
8г. Среднему значению присвоить величину
аккуммулируемой суммы, разделенной на
количество чисел. Напечатать результат.
8д. }

Здесь используется т.н. *условное* утверждение с условием «данные не поступили». Если это условие истинно, тогда выполняется следующее за *Тогда* утверждение, но если условие ложно, то выполняется утверждение, следующее за *Иначе*. Этот процесс называется процессом *Выбора*.

Легко заметить, что алгоритм следует способу, каким человек решил бы данную задачу, если бы ему на листке бумаги дали несколько чисел и попросили бы найти их среднее значение. Человек проведет пальцем по числам, слагая их, пока не останется ни одного числа, а затем, посчитав их полное количество, разделит полученную сумму на количество чисел. При этом, подразумевается, что начальные значения для суммы и счетчика чисел установлены на ноль. Но это вовсе не так же очевидно для компьютера, и следовательно, должно быть учтено в алгоритме при инициализации соответствующих переменных до начала обработки. Также надо удостовериться, что чтение данных осуществляется с начала файла. Таким образом, пункт 1 должен быть расширен:

1а. Присвоить аккумулируемой сумме 0;
1б. Присвоить счетчику чисел значение 0;

1в. Открыть файл для чтения первого числа;

Заметим, что если файл пуст, то первым элементом будет маркер конца файла. Следовательно, условие цикла *Пока* будет ложным с самого начала, оно не выполнится, значения суммы и счетчика останутся нулевыми, поэтому условие пункта 8а будет истинным и соответствующее сообщение появится на экране дисплея.

Полный алгоритм теперь будет выглядеть так:

Присвоить аккумулируемой сумме 0;
Присвоить счетчику чисел значение 0;
Открыть файл для чтения первого числа;
Пока не достигнут конец файла *Делать*
{
Прочитать очередное число;
Добавить число к аккумулируемой сумме;
Увеличить «счетчик» вводимых чисел;
}
Если данные не поступили
Тогда вывести сообщение «Нет входных данных!»
Иначе {
Среднему значению присвоить величину аккумулируемой суммы, разделенной на количество чисел;
Напечатать результат.
}

3.3. Утверждения, используемые для описания алгоритмов.

В приведенном примере для описания алгоритма использовалось очень мало утверждений. Фактически этих утверждений достаточно, чтобы описать все компьютерные алгоритмы. На практике могут использоваться другие виды циклов, но все они могут быть реализованы с помощью цикла *Пока.* Важно подчеркнуть основные принципы и концепты, составляющие алгоритм:

- 1. *Последовательность* — утверждения выполняются по порядку их написания.
- 2. *Составной оператор* — конструкция, которая позволяет обращаться к группе утверждений, как будто это одно утверждение с помощью т.н. программных скобок.
- 3. *Цикл* — конструкция, которая позволяет повторное выполнение одного или составных утверждений несколько раз.
- 4. Выбор — конструкция, которая позволяет выполнить или отказаться от очередного утверждения в зависимости от значения определенного условия.
- 5. *Присвоение* — способность присвоить величине определенное значение.

В языках высокого уровня присутствуют подобные же конструкции, что весьма облегчает перевод алгоритма в компьютерную программу. Фактически, каждому утверждению алгоритма соответствует оператор языка программирования. Например, на Турбо-Паскале вышеприведенный алгоритм переводится следующим образом:

```
Assigh (f,'filename');
Reset(f);
sum := 0;
count := 0;
While not (EoF(f)) Do
Begin
   Read(f,number);
   sum := sum + number;
   Inc(count);
End;
If count = 0 {Данные не поступили }
  Then Write ln ('Нет данных!!!')
  Else Begin
              average := sum/count;
              Writeln (' Среднее значение = ', average)
       End;
Close(f);
```

Здесь приведены только выполняемые операторы программы. Заметим, что данный фрагмент не является полным. Для того чтобы программа была воспринята компилятором синтаксически корректной, требуются дополнительные объявления всех использующихся элементов.

3.4. Алгоритм нахождения минимума и максимума из последовательности чисел.

Рассмотрим следующую постановку задачи:

Пользователю задается список чисел. Надо найти максимальное и минимальное среди них. Написать программу, которая позволит пользователю вводить числа с клавиатуры и вычислять максимальное и минимальное из них. Пусть полное количество чисел определяется также пользователем.

Первая версия алгоритма может выглядеть так:

Инициализация;
Определить полное количество вводимых чисел;
Ввести числа и вычислить максимум и минимум;
Напечатать результаты вычислений.
}

Алгоритм сформулирован в общем виде и не учтен случай, если данные не поступили вовсе. Хотя это может показаться странным, но пользователь, после запуска программы, может передумать вводить числа, и программа должна учитывать вариант, если в качестве количества чисел будет задан ноль. Принимая это во внимание, получим следующий вариант алгоритма:

Инициализация;
Определить полное *количество* вводимых чисел;
Если количество = 0

Тогда Завершить программу
Иначе {
Ввести числа и вычислить *максимум и минимум;*
Напечатать результаты вычислений.
}

После определения количества чисел надо образовать цикл для повторного ввода, считывая каждый раз число и обрабатывая его, найти максимум и минимум среди вводимых величин. В этом цикле количество повторов известно наперед и равняется количеству чисел. Таким образом, получаем новый тип цикла (для — For):

Цикл n—раз
{Тело цикла}

Эта операция, которая повторно выполняет некоторый набор инструкций (тело цикла) фиксированное количество раз, часто используется в структурах алгоритмов. Заметим, что фигурные скобки для тела цикла используются для формирования составного оператора. Каждый раз выполнение цикла означает последовательное выполнение инструкции между скобок. С учетом этих уточнений получаем новую версию алгоритма:

Инициализация;
Определить полное *количество* вводимых чисел;
Если количество = 0
Тогда Завершить программу
Иначе {
Цикл количество — раз
{
Ввести число;
Обработать число;
}
Напечатать результаты вычислений.
}

Здесь еще не уточняется, как вычислить максимальное и минимальные числа, а подразумевается в пункте «Обработать число».

Какой же может быть методический механизм для нахождения минимальной величины из списка чисел? Один из способов мог быть следующий: начиная систематический просмотр списка, запоминать наименьшее к этому моменту число, если очередное число окажется меньше, то запомнить это новое число. Очевидно, что к моменту начала просмотра списка наименьшим (да и наибольшим тоже) будет самое первое число. Таким образом, после завершения просмотра у нас будет запомнено наименьшее число. Аналогично можно найти и наибольшее число. В результате, уточнение пункта «Обработать число» дает следующее расширение алгоритма:

Инициализация;
Определить полное *количество* вводимых чисел;
Если количество = 0
Тогда Завершить программу
Иначе {
Ввести первое число;
Минимуму присвоить первое число;
Максимуму присвоить первое число;
Цикл (количество— 1) раз
{
Ввести *число*
Если число меньше *минимума*
Тогда минимуму присвоить число
Если число больше максимума
Тогда максимуму присвоить число,
}
Напечатать минимум и максимум.
}

В этом алгоритме предварительные инициализации требуются только для *минимума* и *максимума,* и их начальные значения определяются первым числом, введенным пользователем. Аналогично, *количество* вводится пользователем и не требует инициализации.

3.4.1. Проверка алгоритма на корректность.

Перед тем, как реализовать алгоритм в программу, надо его проверить на корректность. Для тестировки надо задавать примерные начальные данные и проследить вручную результат выполнения каждой инструкции. Надо также задавать предельные варианты для начальных данных, например, когда числа не вводятся вовсе.

Для данного примера тестировку можно провести следующим образом. Если пользователь введет нулевое значение для *количества* чисел, программа завершится сразу. Иначе, до начала цикла вводится одно число, а затем, при каждом запуске цикла (всего *количество*—1 раз) вводится очередное число для обработки. Это означает, что всего будет введено

$$1 + (\textit{количество} - 1) = \textit{количество}$$

чисел, так что все числа будут учтены. Начальные значения для *минимума* и *максимума* определяются первым из введенных чисел, и они обновляются при каждом запуске цикла, если введенное очередное число меньше или больше запомненного значения *минимума* и *максимума.*

В алгоритме учтен граничный вариант, когда количество вводимых чисел равняется нулю, но здесь может возникнуть также другой граничный вариант, когда это число равняется единице. В этом случае *минимум* и *максимум* принимают значение первого же введенного числа, а цикл выполнится 0 раз, т.е. не выполнится вовсе. Следовательно, и в этом случае получаем правильный результат.

Можно проверить алгоритм для конкретных данных, например, когда *количество* вводимых чисел равняется пяти, а сами числа — 3, 5,1,7,2. С этой целью составляют таблицу для промежуточных поэтапных значений каждого параметра:

кол	*число*	*макс*	*мин*	Этап
—	—	—	—	Начало
5	—	—	—	Ввод *количества*
5	3	3	3	Ввод первого числа
5	5	5	3	Первый запуск цикла
5	1	5	1	Второй запуск цикла
5	7	7	1	Третий запуск цикла
5	2	7	1	Последний запуск цикла

Символ «—» используется для указания, что значение соответствующей величины на данном этапе неизвестно. В конце алгоритма минимум и максимум содержат корректные значения для этих величин.

Здесь же упомянем еще один тип цикла, который может использоваться при построении алгоритмов — *повторять* (Repeat):

Повторять
    Утверждение 1.
    Утверждение 2.
    ...
    Утверждение n.
Пока – не условие

и который выполняет утверждения между *Повторять* и *Пока—не* (Until) до тех пор, пока условие (которое является *логическим выражением)* не станет истинным. Например, чтобы дать пользователю возможность повторного запуска программы с новыми начальными данными, можно использовать следующий цикл:

Повторять
    Ввести и обработать данные;
    Запросить пользователя –
        надо ли обработать новые данные.
    Считать ответ.
Пока – не ответ = нет.

3.5. Алгоритм Эвклида для нахождения наибольшего общего делителя двух целых чисел.

Пусть заданы два целых числа и требуется найти их наибольший общий делитель. Алгоритм Эвклида заключается в следующем: пусть *первое* число больше *второго*. Делим *первое* на *второе* и запоминаем *остаток*. Если *остаток* отличен от нуля, тогда надо *второе* разделить на *остаток* и запомнить новый остаток, т.е. сейчас в роли *первого* числа выступает *второе* число, а в роли *второго* — *остаток*. Таким образом, на каждом этапе мы рассматриваем три числа: *первое* (делимое), *второе* (делитель) и *остаток*. Процесс деления повторяется до тех пор, пока в остатке не получим ноль. Таким образом, наибольшим общим делителем (НОДелитель) будет препоследний (т.е. последний ненулевой) остаток.

Здесь повторно выполняется операция деления и запоминания остатка, переопределяя при этом каждый раз значения чисел *первое* и *второе*. Подобная техника обновления значения одной и той же переменной очень часто используется при циклическом повторе. Т.е. в теле цикла участвует набор переменных, по мере развития «событий» (при каждом новом циклическом заходе) их значения обновляются.

При формировании алгоритма в конечном виде нужно учесть и граничные случаи, например, когда общих делителей нет, или одно число делится на другое без остатка, и проверить на корректность.

Теперь можно записать алгоритм в явном виде:

```
Присвоить: НОДелитель = 1;
Считать первое число;
Если первое <= 1
Тогда {
    Вывести: «Нелегальный ввод!»
    Завершить программу;
      }
Считать число второе, Если второе <= 1
Тогда {
```

Вывести: «Нелегальный ввод!»
Завершить программу;
}
Если первое < второго
Тогда Поменять числа местами;
Разделить *первое на. второе* и определить *остаток;*
Если остаток не равен нулю
Тогда
{
Пока остаток не равен нулю *Делать*
}
Присвоить числу *первое — второе;*
Присвоить числу *второе — остаток;*
Делим *первое* на *второе,* запоминаем *остаток;*
{
Присвоить: *НОДелитель = второе,*
Если НОДелитель =1
Тогда Вывести: «Общих делителей нет»
Иначе Вывести значение числа *НОДелитель;*
}
Иначе Вывести: «Число *второе* — общий делитель».

Здесь при вводе чисел учтены случаи, когда в качестве чисел вводятся единицы или нули с тем, чтобы обеспечить наиболее информативный вывод, а также избежать инструкции деления на ноль.

В алгоритме используется инструкция «Поменять числа местами;», которую можно реализовать с помощью дополнительной величины *«резерв».* Дело в том, что, учитывая механизм хранения данных в компьютере (см. в следующих главах), мы не можем просто написать:

первое = второе,
второе = первое,

так как после первой инструкции значение величины *первое* потеряется. Для реализации такой задачи используют

вспомогательную величину, с помощью которой расширение инструкции запишется в виде:

Поменять числа местами:
резерв = первое;
первое = второе;
второе = резерв.

Проведем проверку алгоритма на нескольких примерах:

#	Первое число	Второе число	Остаток	НОДелитель
1.	81	36	9	
	36	9	0	
				9
2.	15	5	0	
				5
3.	1290	155	50	
	155	50	5	
	50	5	0	
				5
4.	27	8	3	
	8	3	2	
	3	2	1	
	2	1	0	
				-