1. Основные принципы алгоритмизации и программирования.

1. Основные принципы алгоритмизации и программирования.

1.1. Алгоритмы и программы.

Понятие алгоритма. Понятие алгоритма является одним из основных понятий современной математики. Еще на самых ранних ступенях развития математики (Древний Египет, Вавилон, Греция) в ней стали возникать различные вычислительные процессы чисто механического характера. С их помощью искомые величины ряда задач вычислялись последовательно из исходных величин по определенным правилам и инструкциям. Со временем все такие процессы в математике получили название алгоритмов (алгорифмов).

Термин **алгоритм** происходит от имени средневекового узбекского математика Аль-Хорезми, который еще в IX в. (825) дал правила выполнения четырех арифметических действий в десятичной системе счисления. Процесс выполнения арифметических действий был назван алгоризмом.

С 1747 г. вместо слова **алгоризм** стали употреблять алгорисмус, смысл которого состоял в комбинировании четырех операций арифметического исчисления — сложения, вычитания, умножения, деления.

К 1950 г. **алгорисмус** стал **алгорифмом**. Смысл алгорифма чаще всего связывался с алгорифмами Евклида — процессами нахождения наибольшего общего делителя двух натуральных чисел, наибольшей общей меры двух отрезков и т.п.

Под алгоритмом понимали конечную последовательность точно сформулированных правил, которые позволяют решать те или иные классы задач. Такое определение алгоритма не является строго математическим, так как в нем не содержится точной характеристики того, что следует понимать под классом задач и под правилами их решения.

Первоначально для записи алгоритмов пользовались средствами обычного языка (словесное представление алгоритмов).

Уточним понятие словесного представления алгоритма на примере нахождения произведения n натуральных чисел — факториал числа n (c=n!), т.е. вычисления по формуле c = 1*2*3*4*...•n.

Этот процесс может быть записан в виде следующей системы последовательных указаний (пунктов):

1. Полагаем *c* равным единице и переходим к следующему пункту.
2. Полагаем i равным единице и переходим к следующему пункту.
3. Полагаем *c =i*c* и переходим к следующему пункту.
4. Проверяем, равно ли i числу n. Если i = n, то вычисления прекращаем. Если i <n, то увеличиваем i на единицу и переходим к пункту 3.

Рассмотрим еще один пример алгоритма — нахождение наименьшего числа M в последовательности из n чисел a_1, a_2, ..., a_n (n ≠ 0). Прежде чем записать словесный алгоритм данного примера, детально рассмотрим сам процесс поиска наименьшего числа. Будем считать, что процесс поиска осуществляется следующим образом. Первоначально в качестве числа M принимается a_1, т. е. полагаем M = a_1, после чего M сравниваем с последующими числами последовательности, начиная с a_2. Если M≤ a_2, то M сравнивается с a_3, если M ≤ a_3, то M сравнивается с a_4, и так до тех пор, пока найдется число a_i < M. Тогда полагаем M = a_i, и продолжаем сравнение с M последующих чисел из последовательности, начиная с a_{i+1},. и так до тех пор, пока не будут просмотрены все n чисел. В результате просмотра всех чисел M будет иметь значение, равное наименьшему числу из последовательности (I— текущий номер числа). Этот процесс может быть записан в виде следующей системы последовательных указаний:

1. Полагаем i = 1 и переходим к следующему пункту.
2. Полагаем M = a1,- и переходим к следующему пункту.
3. Сравниваем i с n; если i < n, переходим к 4 пункту, если i = n, процесс поиска останавливаем.
4. Увеличиваем i на 1 и переходим к следующему пункту.
5. Сравниваем ai,- с M. Если M ≤ ai, то переходим к пункту 3, иначе (M > ai) переходим к пункту 2.

В первом алгоритме в качестве элементарных операций используются простейшие арифметические операции умножения, которые могли бы быть разложены на еще более элементарные операции. Мы такого разбиения не делаем в силу простоты и привычности арифметических правил.

Алгоритмы, в соответствии с которыми решение поставленных задач сводится к арифметическим действиям, называются численными алгоритмами.

Алгоритмы, в соответствии с которыми решение поставленных задач сводится к логическим действиям, называются логическими алгоритмами.

Примерами логических алгоритмов могут служить алгоритмы поиска минимального числа, поиска пути на графе, поиска пути в лабиринте и др.

Алгоритмом, таким образом, называется система четких однозначных указаний, которая определяет последовательность действий над некоторыми объектами и после конечного числа шагов приводит к получению требуемого результата.

Свойства алгоритмов. Каждое указание алгоритма предписывает исполнителю выполнить одно конкретное законченное действие. Исполнитель не может перейти к выполнению следующей операции,не закончив полностью выполнения предыдущей. Предписания алгоритма надо выполнять последовательно одно за другим, в соответствии с указанным порядком их записи. Выполнение всех предписаний гарантирует правильное решение задачи.

Поочередное выполнение команд алгоритма за конечное число шагов приводит к решению задачи, к достижению цели. Разделение выполнения решения задачи на отдельные операции (выполняемые исполнителем по определенным командам) — важное свойство алгоритмов, называемое **дискретностью.**

Анализ примеров различных алгоритмов показывает, что запись алгоритма распадается на отдельные указания исполнителю выполнить некоторое законченное действие. Каждое такое указание называется **командой.** Команды алгоритма выполняются одна за другой. После каждого шага исполнения алгоритма точно известно, какая команда должна выполняться следующей. Алгоритм представляет собой последовательность команд (также инструкций, директив), определяющих действия исполнителя (субъекта или управляемого объекта).

Таким образом, выполняя алгоритм, исполнитель может не вникать в смысл того, что он делает, и вместе с тем получать нужный результат. В этом случае говорят, что исполнитель действует формально, т.е. отвлекается от содержания поставленной задачи и только строго выполняет некоторые правила, инструкции.

Это очень важная особенность алгоритмов. Наличие алгоритма формализовало процесс, исключило рассуждения. Если обратиться к другим примерам алгоритмов, то можно увидеть, что и они позволяют исполнителю действовать формально. Таким образом, создание алгоритма дает возможность решать задачу формально, механически исполняя команды алгоритма в указанной последовательности.

Построение алгоритма для решения задачи из какой-либо области требует от человека глубоких знаний в этой области, бывает связано с тщательным анализом поставленной задачи, сложными, иногда очень громоздкими рассуждениями. На поиски алгоритма решения некоторых задач ученые затрачивают многие годы. Но когда алгоритм создан, решение задачи по готовому алгоритму уже не требует каких-либо рассуждений и сводится только к строгому выполнению команд алгоритма.

Всякий алгоритм составляется в расчете на конкретного исполнителя с учетом его возможностей. Для того чтобы алгоритм мог быть выполнен, нельзя включать в него команды, которые исполнитель не в состоянии выполнить. Нельзя повару поручать работу токаря, какая бы подробная инструкция ему не давалась. У каждого исполнителя имеется свой перечень команд, которые он может исполнить. Совокупность команд, которые могут быть выполнены исполнителем, называется **системой команд исполнителя.** Каждая команда алгоритма должна определять однозначно действие исполнителя. Такое свойство алгоритмов называется **определенностью (или точностью) алгоритма**.

Алгоритм, составленный для конкретного исполнителя, должен включать только те команды, которые входят в его систему команд. Это свойство алгоритма называется **понятностью**. Алгоритм не должен быть рассчитан на принятие каких-либо самостоятельных решений исполнителем, не предусмотренных составленным алгоритмом.

Еще одно важное требование, предъявляемое к алгоритмам, — **результативность (или конечность)** алгоритма. Оно означает, что исполнение алгоритма должно закончиться за конечное число шагов.

Разработка алгоритмов — процесс творческий, требующий умственных усилий и затрат времени. Поэтому предпочтительно разрабатывать алгоритмы, обеспечивающие решения всего класса задач данного типа. Например, если составляется алгоритм решения кубического уравнения $ax^3 + bx^2 + cx + d = 0$, то он должен быть вариативен, т.е. обеспечивать возможность решения для любых допустимых исходных значений коэффициентов a, Ь, c, d. Про такой алгоритм говорят, что он удовлетворяет требованию **массовости**. Свойство массовости не является необходимым свойством алгоритма. Оно скорее определяет качество алгоритма; в то же время свойства дискретности, точности, понятности и конечности являются необходимыми (иначе это не алгоритм).

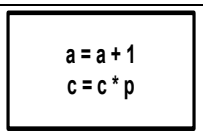
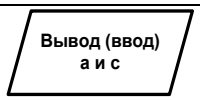
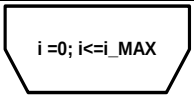
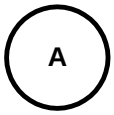
Формы записи алгоритмов. Алгоритмы можно записывать по-разному. Форма записи, состав и количество операций алгоритма зависят от того, кто будет исполнителем этого алгоритма. Если задача решается с помощью ЭВМ, алгоритм решения задачи должен быть записан в понятной для машины форме, т. е. в виде программы. Всякий алгоритм может быть:

- записан на естественном языке (примеры записи алгоритма на естественном языке приведены при определении понятия алгоритма);
- изображен в виде блок-схемы;
- записан на алгоритмическом языке.

Запись алгоритмов в виде блок-схем. Схема алгоритма — графическое представление алгоритма. Каждый пункт алгоритма отображается на схеме некоторой геометрической фигурой — блоком — и

дополняется элементами словесной записи. Правила выполнения схем алгоритмов регламентирует ГОСТ 19.002—80 и 19.003-80 от 01.07.1981 (см. табл.). Эти ГОСТы заменены ГОСТом 19.701-90 (ИСО 5807-85).

Блоки на схемах соединяются линиями потоков информации. **Основное направление потока информации идет сверху вниз и слева направо (стрелки могут не указываться), снизу вверх и справа налево — стрелка обязательна.** Количество входящих линий для блока не ограничено. Выходящая линия должна быть одна (исключение составляет логический блок).

№	Символ	Наименование	Содержание
1		Блок вычислений	Вычислительные действия или последовательность действий
2		Логический блок	Выбор направления выполнения алгоритма в зависимости от некоторого условия
3		Блоки ввода-вывода данных	1 . Общие обозначения ввода (вывода) данных (вне зависимости от физического носителя) 2. Вывод данных, носителем которых является документ
4		Начало (конец)	Начало или конец алгоритма, вход или выход в программу
5		Процесс пользователя (подпрограмма)	Вычисление по стандартной программе или подпрограмме
6		Блок модификации	Функция выполняет действия, изменяющие пункты (например, заголовок цикла) алгоритма
7		Соединитель	Указание связи прерванными линиями между потоками информации в пределах одного листа.
8		Межстраничные соединения	Указание связи между информацией на разных листах

Приведем запись алгоритма нахождения минимального числа M в последовательности из n чисел a_1, a_2, ..., a_n ($n \neq 0$) в виде блок-схемы (рис. 1).

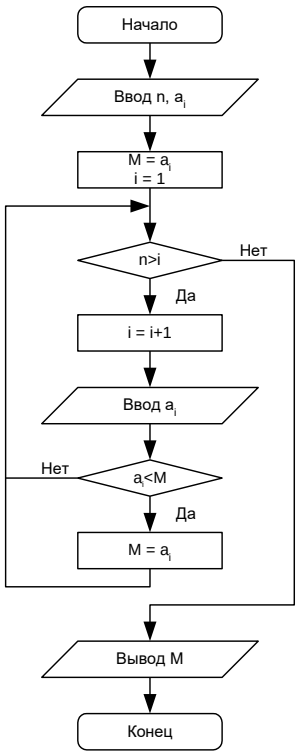


Рис. 1

Базовые структуры алгоритмов — это определенный набор блоков и стандартных способов их соединения для выполнения типичных последовательностей действий. К основным структурам относятся следующие: линейные (рис. 2 а), разветвляющиеся (рис. 2, б), циклические (рис. 2, в, г).

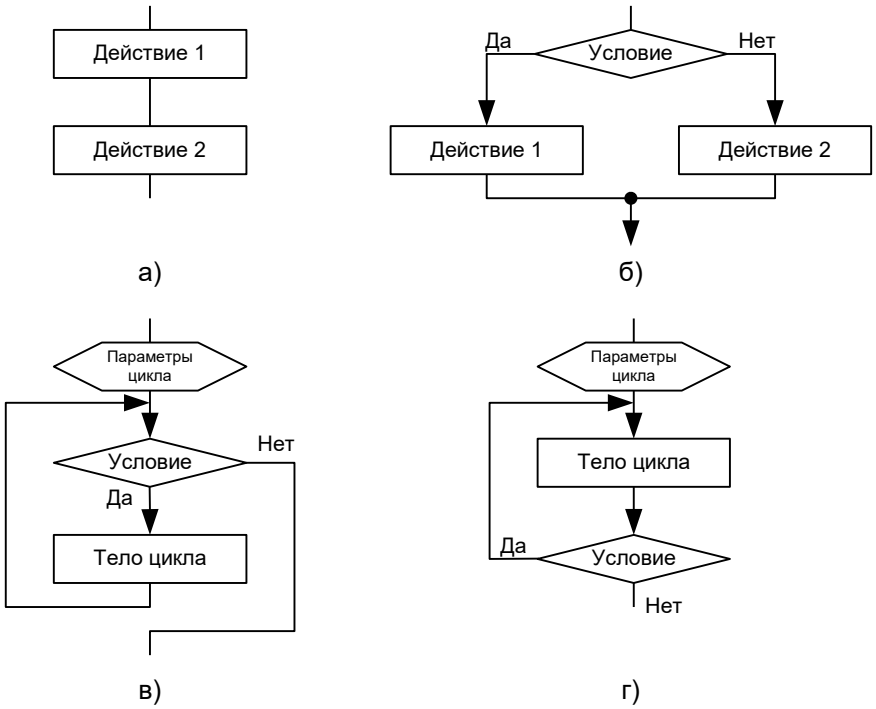


Рис. 2

Линейными называются алгоритмы, в которых действия осуществляются последовательно друг за другом. Стандартная блок-схема линейного алгоритма приводится на рис. 3 а (вычисление произведения двух чисел — A и B).

Разветвляющимся называется алгоритм, в котором действие выполняется по одной из возможных ветвей решения задачи, в зависимости от выполнения условий. В отличие от линейных алгоритмов, в которых команды выполняются последовательно одна за другой, в разветвляющиеся алгоритмы входит условие, в зависимости от выполнения или невыполнения которого выполняется та или иная последовательность команд (действий).

В качестве условия в разветвляющемся алгоритме может быть использовано любое понятное исполнителю утверждение, которое может соблюдаться (быть истинно) или не соблюдаться (быть ложно). Такое утверждение может быть выражено как словами, так и формулой. Таким образом, алгоритм ветвления состоит из условия и двух последовательностей команд.

Примером может являться разветвляющийся алгоритм, изображенный на рис. 3, б. Аргументами этого алгоритма являются числа A и B, а результатом — число X. Если условие A > = B истинно, то выполняется операция умножения чисел (X = A * B), в противном случае выполняется операция сложения (X = A + B). В результате печатается то значение X, которое получается в результате выполнения одного из действий.

Циклическим называется алгоритм, в котором некоторая часть операций (тело цикла — последовательность команд) выполняется многократно. Однако слово «многократно» не значит «до бесконечности». Организация циклов, никогда не приводящая к остановке в выполнении алгоритма, является нарушением требования его результативности — получения результата за конечное число шагов.

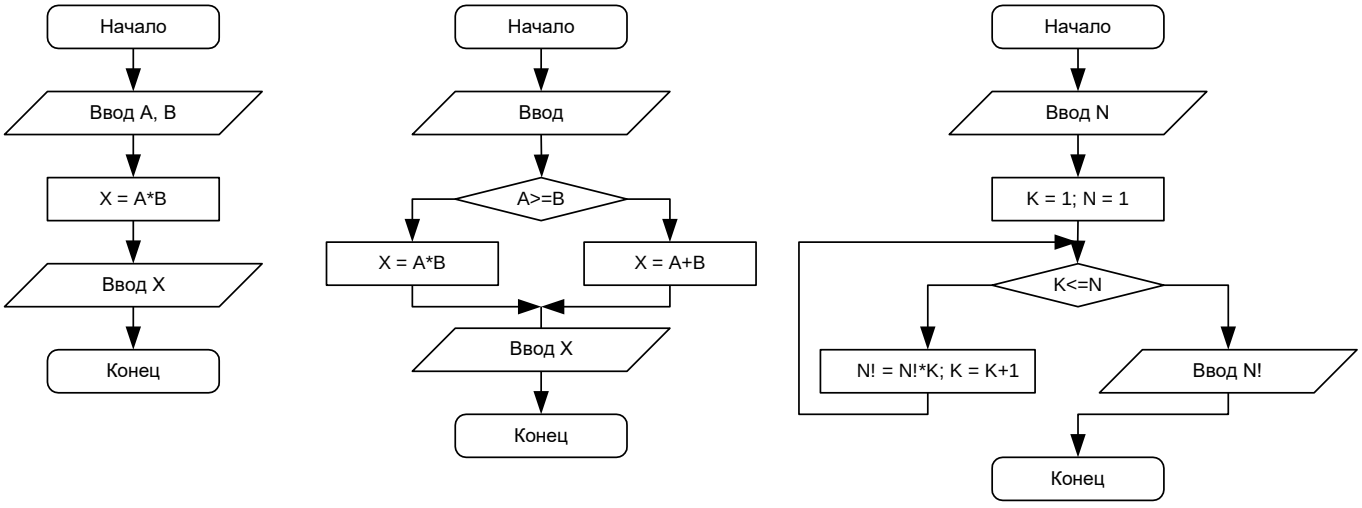
Перед операцией цикла осуществляются операции присвоения начальных значений тем объектам, которые используются в теле цикла. В цикл входят в качестве базовых следующие структуры: блок проверки условия и блок, называемый **телом цикла**. Если тело цикла расположено после проверки условий (рис. 2, в — цикл с предусловием), то может случиться, что при определенных условиях тело цикла не выполнится ни разу. Такой вариант организации цикла, управляемый предусловием, называется **циклом типа пока** (здесь условие — на продолжение цикла).

Возможен другой случай, когда тело цикла выполняется, по крайней мере, один раз и будет повторяться до тех пор, пока не станет истинным условие. Такая организация цикла, когда его тело расположено перед проверкой условия, носит название цикла с постусловием, или **цикла типа до** (рис. 2, г). Истинность условия в этом случае — условие окончания цикла. Отметим, что возможна ситуация с постусловием и при организации цикла - пока. Итак, цикл-до завершается, когда условие становится истинным, а цикл - пока — когда условие становится ложным.

Современные языки программирования имеют достаточный набор операторов, реализующих как цикл - пока, так и цикл - до.

Рассмотрим циклический алгоритм типа пока (рис. 2, в) на примере алгоритма вычисления факториала. N— число, факториал которого вычисляется. Начальное значение N! принимается равным 1. K будет меняться от 1 до N и вначале также равно 1. Цикл будет выполняться, пока справедливо условие N > K. Тело цикла состоит из двух операций N!=N! и K = K+1 (рис. 3, в).

Циклические алгоритмы, в которых тело цикла выполняется заданное число раз, реализуются с помощью цикла со счетчиком. Цикл со счетчиком реализуется с помощью рекурсивного увеличения значения счетчика в теле цикла (K = K + 1 в алгоритме вычисления факториала).



а) б) г)

Рис 3.

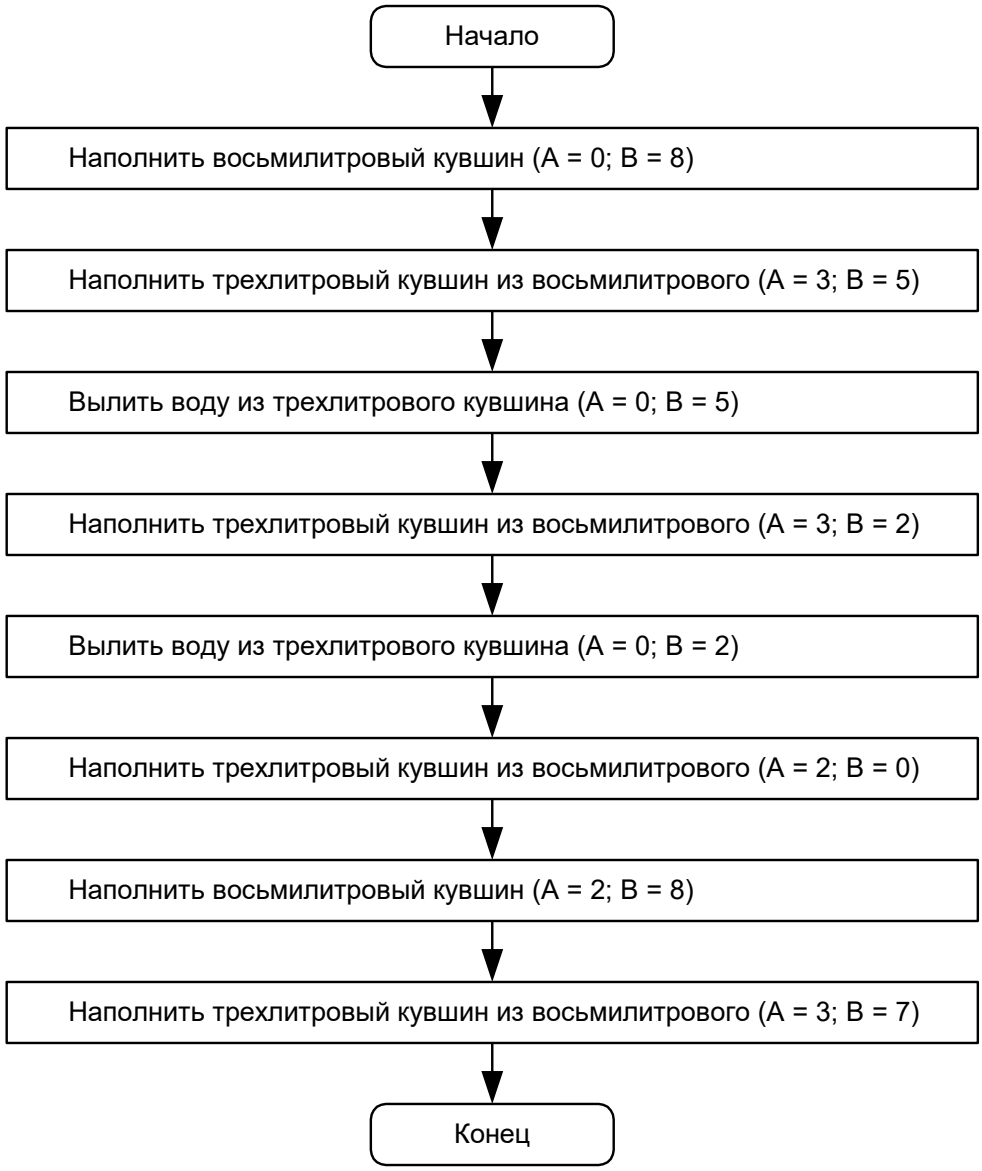
Процесс решения сложной задачи довольно часто сводится к решению нескольких более простых подзадач. Соответственно процесс разработки сложного алгоритма может разбиваться на этапы составления отдельных алгоритмов, которые называются вспомогательными. Каждый такой вспомогательный алгоритм описывает решение какой-либо подзадачи.

Процесс построения алгоритма методом последовательной детализации состоит в следующем. Сначала алгоритм формулируется в «крупных» блоках (командах), которые могут быть непонятны исполнителю (не входят в его систему команд) и записываются как вызовы вспомогательных алгоритмов. Затем происходит детализация, и все вспомогательные алгоритмы подробно расписываются с использованием команд, понятных исполнителю.

Примеры и решения

1.Рассмотрим следующую известную задачу: имеются два кувшина емкостью 3 и 8 л. Необходимо составить алгоритм, с помощью которого, пользуясь только этими двумя кувшинами, можно набрать 7 л воды.

Можно предположить, что кувшин емкостью 3 л необходимо использовать для того, чтобы отлить в него 1 л из полного кувшина емкостью 8 л. Таким образом, решение задачи сводится к поиску возможности поместить, например, 2 л воды в трехлитровый кувшин, затем наполнить восьмилитровый и перелить из него воду в трехлитровый кувшин, в котором до полного заполнения не хватает ровно 1 л. Задача реализуется следующим линейным алгоритмом (A — количество воды в трехлитровом кувшине, B — количество воды в восьмилитровом кувшине):



2. Рассмотрим задачу вычисления наибольшего общего делителя (НОД) двух натуральных чисел A и B с применением алгоритма Евклида. Основная идея алгоритма в том, что НОД A и B есть также и НОД (A - B), т.е. последовательное вычитание из большего числа меньшего до тех пор, пока числа не сравняются, должно привести к искомому значению НОД.

Приведем сначала запись алгоритма Евклида на естественном языке:

Рассмотреть A как первое число, B как второе. Перейти к пункту 2.

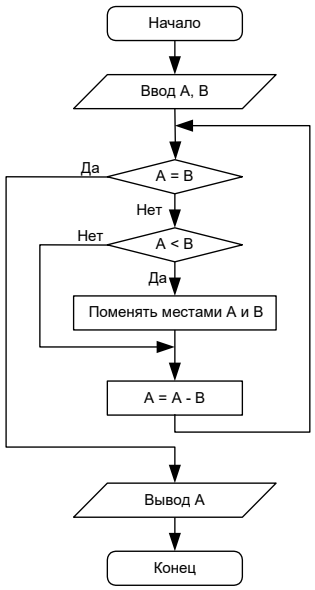
Сравнить первое и второе число. Если они равны, перейти к пункту 5. Если нет, перейти к пункту 3.

Если первое число меньше второго, то поменять их местами. Перейти к пункту 4.

Вычесть из первого числа второе и рассмотреть полученную разность как первое число. Перейти к пункту 2.

Рассмотреть первое число как результат. Закончить выполнение алгоритма.

Представленный алгоритм имеет разветвляющуюся структуру и в виде блок-схемы изображается следующим образом:

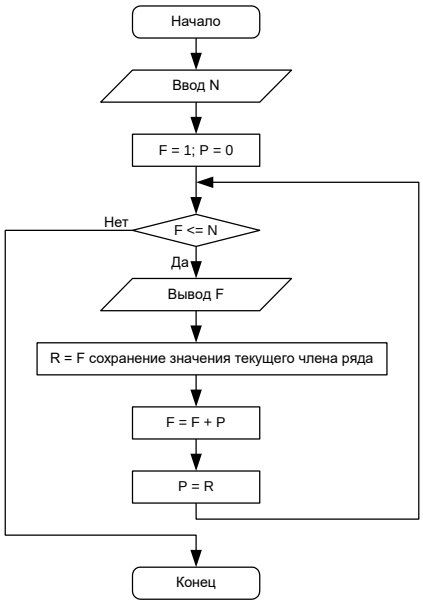


3. Пусть необходимо вывести на печать все числа ряда Фибоначчи (1, 1, 2, 3, 5, 8 ...) до заданного натурального N.

Очередной член ряда F, определяется как сумма двух предыдущих ($F_j = F_{j-1} + F_{j-2}$). Первые два члена ряда равны 1.

Представим блок-схему алгоритма решения задачи с использованием циклической структуры с предусловием (F — очередной член ряда, P — предыдущий член ряда).

Алгоритм решения этой задачи с использованием циклической структуры с постусловием предлагается разработать самостоятельно.



Вопросы и задания.

1. Охарактеризуйте базовые структуры алгоритмов.
2. Дано число X и последовательность действий:
   - умножить полученное число на 2;
   - сообщить результат;
   - разделить X на 3;
   - вычесть из полученного числа 5;
   - прибавить к полученному числу 7.

   Сколько и каких различных алгоритмов можно составить из этой последовательности действий? Какие функции от X при этом вычисляются?
3. Приведите примеры задач, для реализации которых применимы: а) линейные алгоритмы; б) разветвляющиеся алгоритмы; в) циклические алгоритмы.
4. Охарактеризуйте разницу между циклом типа до и циклом типа пока.
5. Приведите примеры задач, для реализации которых целесообразно применять циклические структуры: а) с постусловием; б) с предусловием.
6. Изобразите блок-схему алгоритма определения максимального числа в последовательности из n

произвольных чисел.

7. Нарисуйте блок-схему алгоритма получения произведения n произвольных чисел (n = 15).
8. Нарисуйте блок-схему алгоритма вычисления суммы квадратов первых n чисел натурального ряда.
9. Модифицируйте алгоритм вычисления суммы квадратов первых n чисел натурального ряда для вычисления суммы квадратов: а) только четных чисел (до n); б) только нечетных чисел (до n).
10. Изобразите блок-схему простого диалогового алгоритма, который обращается к пользователю с просьбой ввести сначала строку имя, а затем строку настроение. В результате диалога может появиться следующий общий совместный текст

    Программа> Здравствуйте! Как Ваше имя?
    Пользователь> Гаврик
    Программа> Доброе утро, Гаврик! Как настроение?
    Пользователель> так себе
    Программа> У меня тоже так себе, Гаврик!

1.2. Данные. Понятие типа данных.

Алгоритм, реализующий решение некоторой конкретной задачи, всегда работает с данными. **Данные** — это любая информация, представленная в формализованном виде и пригодная для обработки алгоритмом.

Данные, известные перед выполнением алгоритма, являются начальными, исходными данными. Результат решения задачи — это конечные, выходные данные. В задачах нахождения максимума из последовательности чисел и их произведения исходными данными являются числа, а результатами (выходными данными) — соответственно c и M.

Данные делятся на переменные и константы.

Переменные — это такие данные, значения которых могут изменяться в процессе выполнения алгоритма.

Например, для алгоритма вычисления площади круга необходимо объявить две переменные: переменную R, в которую будет заноситься значение радиуса окружности, и переменную S для вычисления площади круга по формуле

$$S = \pi R^2.$$

Константы — это данные, значения которых не меняются в процессе выполнения алгоритма. В примере, описанном выше, константой является число π.

Каждая переменная и константа должна иметь свое уникальное имя. Имена переменных и констант задаются идентификаторами. **Идентификатор** (по определению) представляет собой последовательность букв и цифр, начинающаяся с буквы.

С данными тесно связано понятие типа данных. Любой константе, переменной, выражению (с точки зрения обработки на ЭВМ) всегда сопоставляется некоторый тип. **Тип данных** характеризует множество значений, к которым относится константа и которые может принимать переменная или выражение. Например, если переменная i в некотором алгоритме может принимать только значения из множества целых чисел, то ей ставится в соответствие целый тип данных.

Типы данных принято делить на простые (**базовые**) и **структурированные**.

К основным базовым типам относятся:

- целый (INTEGER) — определяет подмножество допустимых значений из множества целых чисел;
- вещественный (REAL) — определяет подмножество допустимых значений из множества вещественных чисел;
- логический (BOOLEAN) — множество допустимых значений — истина и ложь;
- символьный (CHAR) — цифры, буквы, знаки препинания и пр.

Тип INTEGER задает подмножество целых чисел, мощность которого зависит от такой характеристики ЭВМ, как размер машинного слова. Если для представления целых чисел в машине используется n разрядов (причем один из них отводится под указание знака числа), то допустимые числа должны удовлетворять условию $-2^{n-1} < x < 2^{n-1}$. Считается, что все операции над данными этого типа выполняются точно и соответствуют обычным правилам арифметики. Если результат выходит за пределы допустимых значений, то вычисления будут прерваны. Такое событие называется **переполнением**. Четыре арифметические операции считаются стандартными: сложение (+), вычитание (-), умножение (*) и деление (/).

Для целых чисел может быть определен дополнительный стандартный тип — целое без знака, задающий подмножество целых положительных чисел. Если для представления целых без знака используется n разрядов, то переменным такого типа можно присваивать значения, удовлетворяющие условию $0 < x < 2^n$.

Тип REAL обозначает подмножество вещественных чисел, границы изменения которых также определяются характеристиками конкретной ЭВМ. И если считается, что арифметика с данными типа INTEGER дает точный результат, то допускается, что аналогичные действия со значениями типа REAL могут быть неточными, в пределах ошибок округлений, вызванных вычислениями с конечным числом цифр. Это принципиальное различие между типами REAL и INTEGER.

Два значения базового типа BOOLEAN обозначаются идентификаторами TRUE FALSE. Операции над данными этого типа и правила их выполнения будут рассмотрены дополнительно.

В базовый тип CHAR входит множество печатаемых символов и символов-разделителей в соответствии с кодом ASCII.

Приведем пример представления числовой информации в различных типах данных применительно к ЭВМ с 16-разрядным машинным словом.

Пусть задано число 12345.

Если этой константе поставлен в соответствие тип INTEGER, то внутренняя стандартная форма представления в памяти (для обработки в двоичной арифметике) — 0011000000111001. Объем — 2 байта, 16 двоичных разрядов.

Если для константы определен тип REAL, то число представляется в форме с плавающей точкой и занимает объем 4 байта, 32 двоичных разряда.

И, наконец, если число рассматривается как последовательность символов типа CHAR (символьное представление), то каждая его цифра представляется байтом в соответствии с кодом ASCII. Представление 12345 есть —00110001 0011001000110011 0011010000110101. Объем — 5 байтов.

Переменные и типы данных вводятся для того, чтобы использовать их в различных алгоритмах обработки. Следовательно, нужно иметь еще и множество операций, которые можно применять к данным того или иного типа. Так, к переменным типа REAL и INTEGER применимы арифметические операции, к переменным типа BOOLEAN — логические, к символьным переменным применима операция конкатенации — «соединения» символов. Операции отношения или сравнения применяются к данным любого типа, но правила их применения различны. Рассмотрим, например, результат применения некоторых операций к значениям 123 и 45 в зависимости от их типа:

Знак	Операция	Запись	Тип данных	Результат
+	Сложение	123+45	INTEGER	168
	Конкатенация	«123» + «45»	CHAR	«12345»
>	Больше	123 > 45	INTEGER	Истина (TRUE)
		«123» > «45»	CHAR	Ложь (FALSE)

Таким образом, тип данных — это такая характеристика данных, которая, с одной стороны, задает множество значений для возможного изменения данных и, с другой стороны, определяет множество операций, которые можно к этим данным применять, и правила выполнения этих операций.

До сих пор мы говорили о переменных, хранящих только одно значение, и рассматривали возможности различного представления и использования этого значения при решении конкретных задач.

На самом деле, огромное количество алгоритмов требует одновременного хранения в памяти целых наборов однородных объектов, причем длина этих наборов может быть заранее неизвестна.

Например, пусть необходимо обрабатывать данные о среднесуточной температуре за год для вычисления максимальной и минимальной температур, среднемесячной и среднегодовой температуры и т.п. Для реализации таких алгоритмов необходимо обеспечить хранение каждого отдельного значения среднесуточной температуры. Если иметь при этом в виду переменные базового типа Real, то таких переменных потребовалось бы 365.

Рассмотрим другой пример. Пусть решение некоторой задачи требует вводить и обрабатывать следующие данные о студентах: фамилия, имя, отчество, дата рождения, год поступления в вуз, номер студенческого билета. При этом алгоритм должен обеспечить возможность ввода произвольного количества данных. Здесь речь идет об обработке однородных объектов, которые можно условно назвать «Информация о студенте», представляющих собой совокупность разнородных данных или атрибутов (фамилия, имя и т.д.).

Для решения подобных задач применяются структуры данных, поддерживаемые множеством структурированных типов данных.

Структурированные типы описывают наборы однотипных или разнотипных данных, с которыми алгоритм должен работать как с одной именованной переменной.

Наиболее широко известная структура данных — массив. **Массив** представляет собой упорядоченную структуру однотипных данных, которые называются элементами массива. Структура данных типа массив подходит, например, для решения задачи обработки среднесуточных температур, описанной выше.

Доступ к каждому отдельному элементу массива осуществляется с помощью индекса — в общем случае порядкового номера элемента в массиве.

Массивы могут быть как одномерными (адрес каждого элемента определяется значением одного индекса), так и многомерными (адрес каждого элемента определяется значением нескольких индексов).

Рассмотрим задачу сортировки (расположения) по возрастанию N Целых чисел. Для ее решения, во-первых, необходимо обеспечить ввод всех N чисел, а затем применить один из известных методов

сортировки. Любой метод сортировки предполагает неоднократный проход всех или части чисел, поэтому числа целесообразно организовать в массив.

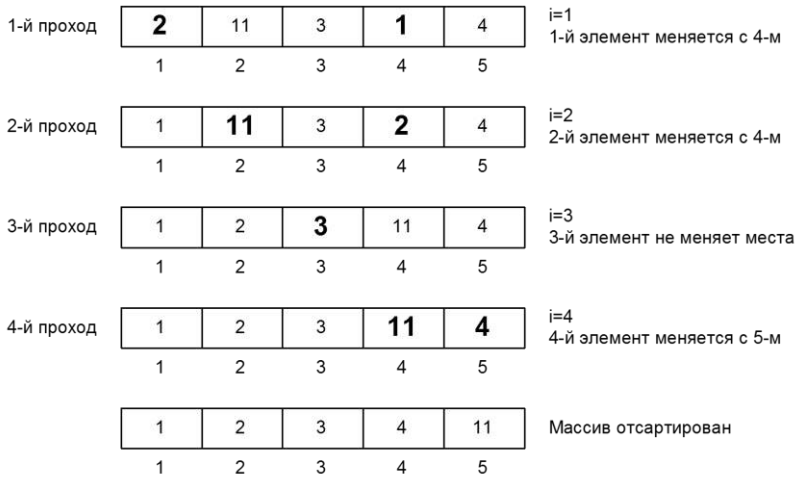


Рис. 4

Метод сортировки посредством простого выбора предполагает циклический просмотр элементов массива, начиная с i-го (i = 1,2,...,N-1), поиск минимального элемента и перестановку найденного минимального элемента с i-м. За N-1 проход по массиву (элемент с номером N останется на своем месте) числа будут отсортированы (рис. 4).

Подобный алгоритм, очевидно, состоит из трех этапов:

- ввод элементов массива;
- цикл обработки массива;
- вывод отсортированных чисел из массива.

Цикл обработки массива представляет собой совокупность двух «вложенных» циклов: первый цикл (с переменной цикла i) обеспечивает «внешний» проход по элементам массива, начиная с 1-го и кончая элементом с номером N - 1, который должен заканчиваться на каждом шаге перестановкой элементов; второй цикл (с переменной цикла j) реализует поиск минимального элемента среди элементов с номерами от i + 1 до N (т.е. j меняется от i + 1 до N).

Блок-схема предлагаемого алгоритма сортировки представлена на рис. 5.

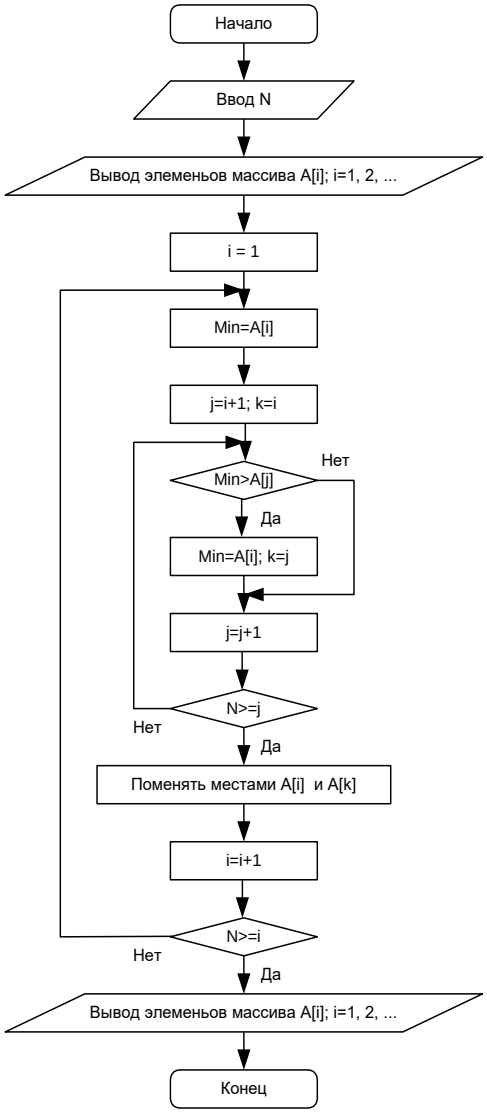


Рис. 5.

Запись. Наиболее общий метод получения структурных типов заключается в объединении элементов произвольных типов. Причем сами эти элементы могут быть в свою очередь структурными. Примером данных такого типа может быть объект «Информация о студенте», рассмотренный выше. В математике это могут быть комплексные числа, состоящие из двух вещественных чисел, или координаты точки, определяемые двумя или более (в зависимости от размерности пространства) числами.

Множество значений, определенное таким структурным типом, состоит из всех возможных комбинаций значений, относящихся к каждому из множеств значений элементов структуры. Таким образом, число таких комбинаций равно произведению числа элементов в каждом из составляющих множеств.

При обработке данных структурные типы, описывающие объекты, обычно встречаются в файлах или базах данных, поэтому к данным такой природы стало широко применяться слово **запись**. Элементы записи называются **полями**.

Величины типа запись могут быть представлены графически. На рис. 6 изображены следующие переменные:

z типа Complex — объект, описывающий комплексное число, с полями re и im типа Real;

d типа Date — объект, описывающий дату, с полями day, month и year со значениями из подмножеств типа Integer (например, day может быть целым числом в интервале от 1 до 31);

p типа Person — объект «Информация о студенте» с полями family, firstname, secondname, Date, year, num.

Complex z

1.0
-1.0

Date d

14
4
2002

Person p

Иванов		
Иван		
Иванович		
1	4	1985
2002		
235/02		

Рис. 6

Ранние языки программирования (ЯП) — Фортран, Алгол — будучи ориентированы исключительно на вычисления, не содержали развитых систем типов и структур данных.

В Алголе, например, были определены два типа структур: элементарные данные и массивы (векторы, матрицы, состоящие из арифметических или логических переменных), а символьные величины и переменные вообще не предусматривались. Основным нововведением, появившимся первоначально в языке Cobol, (затем PL/1, Pascal и пр.) является символьный тип (цифры, буквы, знаки препинания и пр.), а также записи, структуры (синоним записи). В табл. приведены для сравнения типы и структуры данных некоторых языков программирования.

Тип Данных	Наименование	Язык программирования			
		Algol	**Pascal**	**Basic**	**C**
	Целое короткое (2 байта)		Integer Smallint	Integer	Short
	Целое норм. (4 байта)	Integer	Longint Integer	Long	Long
	Действительное норм. (4 байта)	Real	Single	Single	Float
	Действ, двойное (8 байт)		Real	Double Currency	Long float
	Логическое	Boolean	Boolean	Boolean	Boolean
	Символьное (1 байт)		Char	String	Char
	Массивы	Array	Array	Dim	[]
	Записи (структуры)		Record		Struct

Примеры и решения.

1. Рассмотрим алгоритм обработки двумерного массива. Аналогом двумерного массива в математике является матрица. Общий вид матрицы A размерности M*N следующий:

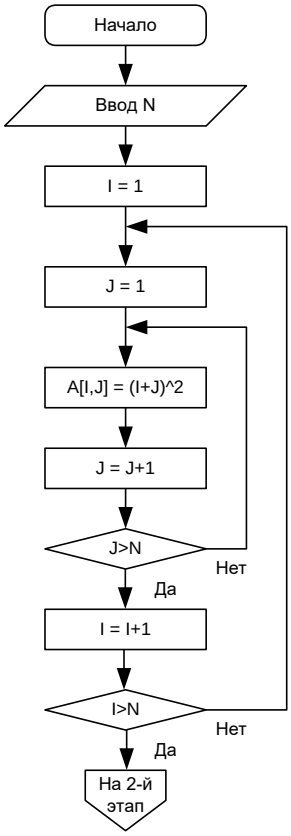
$$A = \{a_{ji}, i \equiv \overline{1...M}, j = \overline{1...N}\} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \dots & \dots & \dots & \dots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{pmatrix}$$

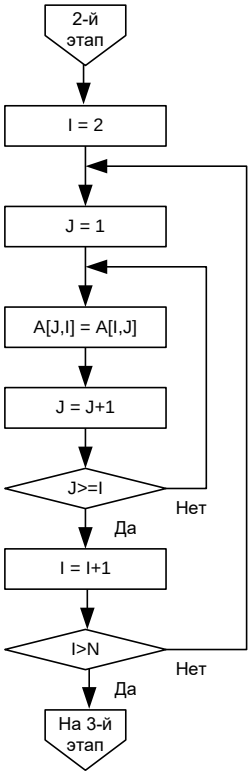
Для матрицы размерностью N*N (квадратной матрицы) определены понятия главной и побочной диагоналей. Элементы главной диагонали — {a_{ii}, i=1…N}, а элементы побочной диагонали — {$a_{i(N-i+1)}$, i=1…N}

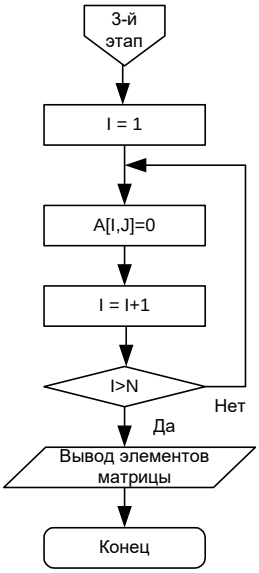
Пусть необходимо построить матрицу, элемент которой a_{ij},- вычисляется как $(i + j)^2$, а затем зеркально отобразить ее относительно главной диагонали. Элементам главной диагонали при этом присвоить значение 0.

Для решения поставленной задачи воспользуемся структурой данных — двумерный массив целых чисел. На первом этапе построим вложенный цикл расчета элементов и заполнения массива. На втором этапе проведем «зеркальное отражение» элементов, находящихся под главной диагональю. И, наконец, на третьем этапе обнулим элементы главной диагонали.

Блок-схема алгоритма приведена на рисунке (N — размерность массива, I и J — индексы)







2. Рассмотрим задачу, решение которой основано на преобразовании типов данных.

Пусть необходимо построить алгоритм сложения очень больших целых чисел, для представления которых не подходит тип данных INTEGER, имеющий ограничения на максимально возможные хранимые значения.

Используем для ввода в алгоритм сложения символьные представления чисел, т.е. рассмотрим каждое число как последовательность символов (например, A = «1234567890» и B = «98765432199»). Для каждого числа посчитаем длину его символьного представления (N — длина числа A, M — длина числа B) и присвоим некоторой переменной L значение максимальной длины.

Следующим шагом разместим 3 массива байтов размерностью L + 1 (такова будет максимальная длина в символах полученного результата) и заполним их элементы числовым значением 0. В первый и второй массивы перешлем (начиная с конца) посимвольно числа A и B, преобразуя каждый символ в числовое представление (в соответствии с таблицей ASCII, это можно сделать, например, вычтя из числового значения кода цифры значение кода символа «0»). При этом в первом и втором массивах получим так называемые двоично-десятичные представления чисел (т.е. когда каждый разряд десятичного числа представляется двоичной комбинацией):

A:	0	0	1	2	3	4	5	6	7	8	9	0
B:	0	9	8	7	6	5	4	3	2	1	9	9
C:	0	0	0	0	0	0	0	0	0	0	0	0

Теперь, складывая побайтно (от элемента с номером L+1 до элемента с номером 1) числовые значения в массивах A и B и помещая результат в соответствующий байт массива C, получим двоично-десятичный результат сложения двух чисел:

A:	0	0	1	2	3	4	5	6	7	8	9	0
B:	0	9	8	7	6	5	4	3	2	1	9	9

1-я итерация

C:	0	0	0	0	0	0	0	0	0	0	0	9

2-я итерация

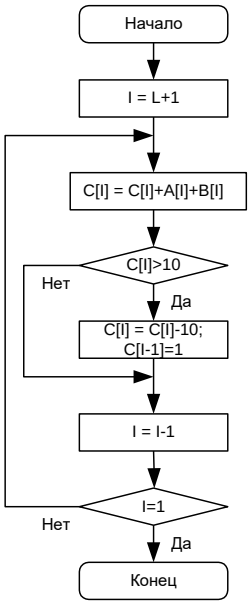
C:	0	0	0	0	0	0	0	0	0	1	8	9

.....

Результат

C:	1	0	0	0	0	0	0	0	0	0	8	9

Перенос разряда при сложении может быть осуществлен так, как показано на блок-схеме.

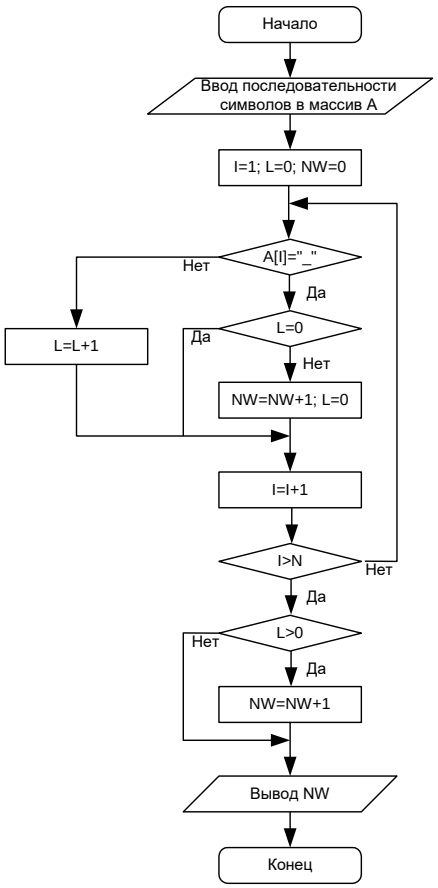


Последним шагом решения задачи должно быть преобразование C из двоично-десятичного представления в символьное для вывода результата.

3. Пусть требуется посчитать количество слов в некоторой последовательности символов. Определим слово как часть последовательности (ненулевой длины), ограниченную с двух сторон (либо только справа или слева) символом «пробел».

Для решения задачи разместим обрабатываемую последовательность в массиве символов размерностью N (где N — длина последовательности). При построении алгоритма необходимо учесть, что последовательность может начинаться или заканчиваться пробелами и может встречаться несколько пробелов подряд.

Блок-схема алгоритма приведена на рис. (L — длина очередного слова, NW — количество слов в последовательности, знак пробела обозначен как «_»).



Вопросы и задания.

1. Какое максимальное (минимальное) целое значение без знака (со знаком) можно записать в 12 битов?
2. Сколько байтов потребуется для записи Вашей фамилии, имени и отчества?
3. Сколько записей «Информация о студенте» можно разместить в 1 мегабайте памяти?
4. Охарактеризуйте различные способы представления числовой информации.
5. Охарактеризуйте разницу между простым и структурированным типом данных.
6. Изобразите в виде блок-схемы алгоритм нахождения максимального (минимального) элемента в двумерном массиве.
7. Постройте алгоритм зеркального отображения элементов двумерного массива размерностью N*N относительно побочной диагонали.
8. Постройте алгоритм формирования одномерного массива, каждый элемент которого представляет собой сумму элементов строки некоторого двумерного массива.
9. Постройте алгоритм вычитания больших целых чисел.
10. Постройте алгоритм ввода данных в структуру «Информация о студенте».
11. Опишите в виде массива записей информацию о студентах группы и постройте алгоритм подсчета среднего возраста студентов.

1.3. Логические основы алгоритмизации.

Важной составляющей алгоритмов являются логические условия. Вычисление значений логических условий происходит в соответствии с аксиомами алгебры логики.

Начало исследований в области формальной логики было положено работами Аристотеля в IV в. до н. э. Однако математические подходы к этим вопросам впервые были указаны Джорджем Булем, который положил в основу математической логики алгебру логики (в честь ее создателя алгебру логики называют булевой алгеброй, а логические значения — булевыми). Алгебра логики используется при построении основных узлов ЭВМ — шифратора, дешифратора, сумматора.

Алгебра логики оперирует с высказываниями. Под высказыванием понимают повествовательное предложение, относительно которого имеет смысл говорить, истинно оно или ложно. Например, выражение «Расстояние от Москвы до Киева больше, чем от Москвы до Тулы» истинно, а выражение «4 < 3» — ложно.

Высказывания принято обозначать большими буквами латинского алфавита: A, B, C ... X, Y и т.д. Если высказывание C истинно, то пишут C = 1 (C = t, true), а если оно ложно, то C = 0 (C = f, false).

В алгебре высказываний над высказываниями можно производить определенные логические операции, в результате которых получаются новые высказывания. Истинность полученных высказываний зависит от истинности исходных высказываний и использованных для их преобразования логических операций.

Для образования новых высказываний наиболее часто используются логические операции, выражаемые словами «не», «и», «или».

Конъюнкция (логическое умножение). Соединение двух (или нескольких) высказываний в одно с помощью союза И (OR) называется операцией, логического умножения, или конъюнкцией. Эту операцию принято обозначать знаками «^, &» или знаком умножения «*». Сложное высказывание A & B истинно только в том случае, когда истинны оба входящих в него высказывания. Истинность такого высказывания задается следующей таблицей

A	B	A&B
false	false	false
false	true	false
true	false	false
true	true	true

Объединение двух (или нескольких) высказываний с помощью союза ИЛИ (OR) называется операцией логического сложения, или дизъюнкцией. Эту операцию обозначают знаками « | , v» или знаком сложения «+». Сложное высказывание A v B истинно, если истинно хотя бы одно из входящих в него высказываний. Таблица истинности для логической суммы высказываний имеет вид:

A	B	AvB	A xor B
false	false	false	false
false	true	true	true
true	false	true	true
true	true	true	false

В последнем столбце данной таблицы размещены результаты модифицированной операции ИЛИ — ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR), отличающееся от обычного ИЛИ последней строкой.

Присоединение частицы НЕ (NOT) к данному высказыванию называется операцией отрицания (инверсии). Она обозначается Ā (или ¬A)и читается не A. Если высказывание A истинно, то B ложно, и наоборот. Таблица истинности в этом случае имеет вид

A	¬A
false	true
true	false

Помимо операций И, ИЛИ, НЕ в алгебре высказываний существует ряд других операций. Например, операция эквиваленции (эквивалентности) A ~ B (или A Ξ B, A eqv B), которая имеет следующую таблицу истинности:

A	B	A~B
false	false	true
false	true	false
true	false	false
true	true	true

Другим примером может служить логическая операция импликации или логического следования (A → B, A imp B), объединяющая высказывания словами «если..., то» и имеющая следующую таблицу истинности:

A	B	A→B
false	false	true
false	true	true
true	false	false
true	true	true

Высказывания, образованные с помощью логических операций, называются сложными. Истинность сложных высказываний можно установить, используя таблицы истинности. Например, истинность сложного высказывания ¬A & B определяется следующей таблицей:

A	B	¬A	B	¬A&B
false	false	true	true	true
false	true	true	false	false
true	false	false	true	false
true	true	false	false	false

Высказывания, у которых таблицы истинности совпадают, называются равносильными. Для обозначения равносильных высказываний используют знак «=» (A = B). Рассмотрим сложное высказывание (A & B) | (¬A & ¬B) и запишем таблицу истинности этого высказывания:

A	B	¬A	¬B	A&B	¬A&¬B	(A&B)|(¬A&¬B)
false	false	true	true	false	true	true
false	true	true	false	false	false	false
true	false	false	true	false	false	false
true	true	false	false	true	false	true

Если сравнить эту таблицу с таблицей истинности операции эквивалентности высказываний A и B, то можно увидеть, что высказывания (A&B)|(¬A&¬B) и A ~ B тождественны, т.е. A~ B = (A&B)|(¬A&¬B).

В алгебре высказываний можно проводить тождественные преобразования, заменяя одни высказывания равносильными им другими высказываниями.

Исходя из определений дизъюнкции, конъюнкции и отрицания, устанавливаются свойства этих операций и взаимные распределительные свойства. Приведем примеры некоторых из этих свойств:

Коммутативность (перестановочность):

A^B = B^A

AvB = BvA

Закон идемпотентности

A&A = A

AvA = A

Сочетательные законы

Av(BvC) = (AvB)vC = AvBvC

A^(B^C) = (A^B)^C = A^B^C

Распределительные (дистрибутивные) законы

A^(BvC) = (A^B)v(A^C)

Av(B^C) = (AvB)^(AvC)

Законы де Моргана

1. ¬ (A^B) = ¬Av¬B (условно его можно назвать 1-й);
2. ¬ (AvB) = ¬A^¬B (2-й) — описывают результаты отрицания переменных, связанных операциями И, ИЛИ.

В некоторых ЯП (например, Visual Basic) для расширения применимости логических выражений на те случаи, когда значение одного или нескольких логических аргументов неизвестны или не определены, вводится значение NULL (в дополнение к false и true). Как правило, такое значение присваивается компилятором логической переменной по умолчанию.

С учетом значения null таблицы истинности основных логических операций приобретают вид, представленный в таблицах.

A	¬A
false	true
true	false
null	null

A	B	A&B	AvB	A→B	A~B	A xor B
false	false	false	false	true	true	false
false	true	false	true	true	false	true
true	false	false	true	false	false	true
true	true	true	true	true	true	false
false	null	false	false	true	null	null
true	null	null	true	null	null	null
null	false	false	null	null	null	null
null	true	null	true	true	null	null
null	null	null	null	null	null	null

В набор операций некоторых современных ЯП включены операции **побитового сравнения** содержимого машинных слов (которые могут содержать числовые, строчные и др. данные) при этом каждый бит результата образуется в соответствии с таблицей для бинарных операций. Унарная операция отрицания (not) в данном случае реализует очевидную замену 1 на 0 и наоборот.

X	Y	X and Y	X or Y	X imp Y	X eqv Y	X xor Y

0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	0	0	1
1	1	1	1	1	1	0

Примеры и решения.

1. Рассмотрим следующую задачу. Пусть требуется сформулировать логическое выражение для отбора среди записей «Информация о студенте» тех, у которых в дате рождения стоит декабрь 1984 г. или январь 1985 г. Построим сначала простые высказывания, затем объединим их в логическое выражение и приведем для него таблицу истинности:

высказывание A — «Месяц рождения декабрь»;
высказывание C — «Месяц рождения январь»;
высказывание B — «Год рождения 1984»;
высказывание D — «Год рождения 1985».
Логическое выражение на базе простых высказываний имеет следующий вид:

(A&B)|(C& D).

Построим таблицу истинности (1 — true, 0 — false, жирным шрифтом выделены наборы, на которых логическое выражение истинно):

A	B	C	D	A&B	C&D	(A&B)|(C&D)
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	**0**	**1**	**1**	**0**	**1**	**1**
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	**1**	**1**	**1**	**0**	**1**	**1**
1	0	0	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	**0**	**1**	**1**	**0**	**1**	**1**
1	**1**	**0**	**0**	**1**	**0**	**1**
1	**1**	**0**	**1**	**1**	**0**	**1**
1	**1**	**1**	**0**	**1**	**0**	**1**
1	**1**	**1**	**1**	**1**	**1**	**1**

2. Известен следующий закон свертки логического выражения:

(A&B)|(¬A&C)|(B&C) = (A&B)|(¬A&C)

т.е. третье слагаемое логической суммы в левой части выражения не влияет на конечный результат. Проиллюстрируем этот закон с помощью таблиц истинности

A	B	C	A&B	¬A&C	B&C	(A&B)|(¬A&C)|(B&C)	(A&B)|(¬A&C)
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	1
0	1	0	0	0	0	0	0
0	1	1	0	1	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	1	0	1	1	1

Вопросы и задания.

1. Составьте таблицу истинности для (A & B) |) | (¬A & ¬B) с учетом значения null.
2. Составьте таблицы истинности для левого (¬(A ^ B)) и правого (¬A v ¬B) выражений 1 -го закона де Моргана. Проверьте их на соответствие.
3. Составьте таблицы истинности для левого (¬(A v B)) и правого (¬A ^ ¬B) выражений 2-го закона де Моргана. Проверьте их на соответствие.
4. Проверьте выполнимость законов де Моргана с учетом значения null.
5. Последний столбец таблицы истинности для двуместных операций, очевидно, может содержать 16 = 24

различных сочетаний 1 и 0. Следовательно, всего может быть определено 16 логических операций над 2 переменными, из которых нами рассмотрены только 5. Составьте таблицу истинности для одной из 9 оставшихся вне рассмотрения функций и попытайтесь построить логическое выражение для этой функции.

1.4. Псевдокод для профи

Псевдокодом называют неформальную нотацию на естественном языке, описывающую работу алгоритма, метода, класса или программы. «Процесс программирования с псевдокодом» (ППП, Pseudocode Programming Process) относится к конкретной методике применения псевдокода для эффективного создания кода методов.

Поскольку псевдокод напоминает естественный язык, разумно предположить, что любая перефразировка ваших мыслей, выполненная на нем, будет иметь одинаковый эффект. На практике же вы обнаружите, что некоторые стили псевдокода предпочтительней других.

- Применяйте формулировки, в точности описывающие отдельные действия.
- Избегайте синтаксических элементов языков программирования. Псевдокод позволяет проектировать на несколько более высоком уровне, чем код. Применяя конструкции языка программирования, вы мыслите на более низком уровне и теряете преимущества проектирования на высоком уровне, загружая себя ненужными синтаксическими ограничениями.
- Пишите псевдокод на уровне намерений. Описывайте назначение подхода, а не то, как этот подход нужно реализовать на выбранном языке программирования.
- Пишите псевдокод на достаточно низком уровне, так чтобы код из него генерировался практически автоматически. Если псевдокод написан на слишком высоком уровне, могут возникнуть проблемы кодирования. Детализируйте псевдокод до тех пор, пока вам не покажется, что проще было бы написать код.

Написав псевдокод, вы окружаете его кодом, а псевдокод превращаете в комментарии программы. Если вы руководствуетесь перечисленными правилами создания псевдокода, комментарии в вашей программе будут полными и ясными.

Вот пример псевдокода, в котором нарушены практически все перечисленные правила:

Пример плохого псевдокода

```
увеличить номер ресурса на 1
выделить структуру dlg посредством malloc
если malloc() возвращает NULL вернуть 1
вызвать OSrsrc_init для инициализации ресурса
*hRsrcPtr = номер ресурса
вернуть 0
```

Какие намерения описывает этот блок псевдокода? Трудно сказать, поскольку написан он плохо. Этот так называемый псевдокод плох потому, что включает конкретику целевого языка программирования: *hRsrcPtr (описание указателя, специфичное для языка C) и malloc() (функция C). Этот блок псевдокода показывает, как будет написан код, а не описывает его назначение. Он вдается в излишние подробности: вернет ли процедура 1 или 0. Если посмотреть, можно ли превратить этот псевдокод в нормальные комментарии, видно, что толку от него мало.

А вот описание тех же действий на гораздо лучшем псевдокоде:

Пример хорошего псевдокода

```
Отслеживать текущее число используемых ресурсов
Если другой ресурс доступен
    Выделить структуру для диалогового окна
    Если структура для диалогового окна может быть выделена
        Учесть, что используется еще один ресурс
        Инициализировать ресурс
        Хранить номер ресурса в вызывающей программе
    Конец «если»
Конец «если»
Вернуть true, если новый ресурс был создан; иначе вернуть false
```

Этот псевдокод лучше предыдущего, так как полностью написан на естественном языке и не использует специфических конструкций целевого языка программирования. В первом примере псевдокод подлежал реализации только на C. Во втором же псевдокод не накладывает ограничений на используемый язык. Второй блок также написан на уровне описания намерений. О чем речь во втором блоке? Наверное, это легче понять, чем в первом блоке.

Хотя второй блок написан на понятном естественном языке, он достаточно точен и подробен, чтобы быть основой программы. Когда предложения этого псевдокода будут преобразованы в комментарии, они станут хорошим пояснением назначения кода.

Вот выгоды, которые вы получите, применяя такой стиль псевдокода.

- Псевдокод упрощает пересмотр конструкции — вам не потребуется вникать в исходный код.
- Псевдокод поддерживает идею итеративного усовершенствования. Вы начинаете с высокоуровневой конструкции, уточняете ее до псевдокода, который в свою очередь преобразуете в исходный код. Такое последовательное усовершенствование, осуществляемое шаг за шагом, позволяет проверять свои проектные решения по мере перехода на более низкие уровни. В результате вы обнаруживаете высокоуровневые ошибки на самом верхнем уровне, среднеуровневые — на среднем, а низкоуровневые — на самом нижнем, прежде чем они создадут проблемы.
- Псевдокод упрощает внесение изменений. Что проще: исправить линию на чертеже или снести стену и сдвинуть ее на метр в сторону? В программировании эффект не столь драматичен в плане физических усилий, но идея та же: несколько строк псевдокода легче исправить, чем страницу кода. Одна из основ успеха проекта — отловить ошибку на «наименее значимой стадии», когда для ее исправления требуется минимум усилий. Поиск ошибки на стадии псевдокода требует гораздо меньше затрат, чем после полного кодирования, тестирования и отладки, так что есть экономический стимул обнаружить ошибку как можно раньше.
- Псевдокод упрощает комментирование программ. В типичной ситуации вы сначала пишете код, а затем добавляете комментарии. В ППП предложения псевдокода становятся комментариями, так что на самом деле их проще оставить, чем удалить.
- Псевдокод сопровождать проще, чем другие виды проектной документации. При других подходах проектная документация отделена от кода, и внесение в нее изменений порождает несоответствие. В ППП предложения псевдокода становятся комментариями программы. Внося изменения в комментарии, вы, таким образом, поддерживаете в корректном состоянии проектную документацию.

Псевдокод как инструмент проектирования трудно переоценить. Исследования показали, что программисты предпочитают псевдокод за его возможности упрощать создание программных конструкций, помощь в определении некорректных проектных решений, простоту документирования и внесения изменений. Псевдокод — не единственный инструмент детального проектирования, но наряду с ППП — полезная вещь в инструментарии программиста. Попробуйте его в деле.

Пример псевдокода метода

Этот метод выводит сообщение об ошибке на основании кода ошибки, получаемого от вызывающей программы. Способ вывода сообщения зависит от режима работы, который он определяет сам. Он возвращает значение, указывающее на успешное завершение или сбой.

```
Установить статус по умолчанию в “сбой”.
Найти сообщение, соответствующее коду ошибки.

Если код ошибки корректен
    Если работа в интерактивном режиме, вывести сообщение
    и указать успешный статус.

    Если работа в режиме командной строки, запротоколировать
    сообщение об ошибке и указать успешный статус.

Если код ошибки некорректен, информировать пользователя
об обнаружении внутренней ошибки.

Вернуть статус
```

Еще раз: этот псевдокод достаточно высокого уровня, он не содержит конструкций языка программирования, а объясняет последовательность действий на естественном языке.

Проверьте псевдокод. Написав псевдокод и спроектировав данные, уделите минутку просмотру написанного. Задумайтесь, как бы вы объяснили это кому-то другому.

Попросите кого-нибудь прочитать написанное или выслушать ваше объяснение. Вам может показаться глупым просить коллегу посмотреть на какие-то 11 строк псевдокода, но результат вас удивит. Псевдокод более явно обозначит ваши ошибочные намерения, чем код на языке программирования. К тому же люди охотней просматривают несколько строк псевдокода своих коллег, чем 35 строк программы на C++ или Java.

Убедитесь, что вы имеете четкое представление о том, что и как делает метод. Если вы не понимаете его концептуально, на уровне псевдокода, какой же тогда у вас шанс разобраться в нем на уровне языка программирования? Если его не понимаете вы, кто его поймет?

Опишите несколько идей псевдокодом и выберите лучшую (пройдите по циклу) Прежде чем кодировать, реализуйте как можно больше своих идей в псевдокоде. Приступив к кодированию, вы эмоционально вовлекаетесь в этот процесс, и вам труднее отказаться от плохого проекта и начать заново.

Общая идея: раз за разом проходиться по псевдокоду, пока каждое его предложение не станет настолько простым, что под ним можно будет вставить строку программы, а псевдокод оставить в качестве документации. Часть псевдокода, написанного при первых проходах, может оказаться достаточно высокоуровневой и потребовать дальнейшей декомпозиции. Не забывайте это сделать. Если вам не понятно, как закодировать какой-то фрагмент, продолжайте работать с псевдокодом, пока это не прояснится. Продолжайте уточнение и декомпозицию, пока это не будет выглядеть как напрасная трата времени по сравнению с написанием настоящего кода.

Кодирование метода

Спроектировав метод, приступайте к его конструированию. Конструирование можно производить в стандартном порядке, а при необходимости отступить от него

Пример законченного метода, созданного посредством Процесса Программирования Псевдокода (C++)

```
/* Этот метод выводит сообщение об ошибке на основании кода ошибки, получаемого от
вызывающей программы. Способ вывода сообщения зависит от режима работы, который он
определяет сам. Он возвращает значение, указывающее на успешное завершение или сбой.
*/
Status ReportErrorMessage(
      ErrorCode errorToReport
      ) {
      // Установить статус по умолчанию в "сбой".
      Status errorMessageStatus = Status_Failure;

      // Найти сообщение, соответствующее коду ошибки.
      Message errorMessage = LookupErrorMessage( errorToReport );

      // Если код ошибки корректен.
      //Отсюда начинаем добавлять код для каждого комментария.
      if ( errorMessage.ValidCode() ) {
            // Определяем метод обработки.
            ProcessingMethod errorProcessingMethod = CurrentProcessingMethod();

            // Если работа в интерактивном режиме, вывести сообщение
            // и указать успешный статус.
```

```
            if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
                  DisplayInteractiveMessage( errorMessage.Text() );
                  errorMessageStatus = Status_Success;
            }
            // Если работа в режиме командной строки, запротоколировать
            // сообщение об ошибке и указать успешный статус.

            //Этот код — хороший кандидат стать новым методом:
            //DisplayCommandLineMessage().
            else if ( errorProcessingMethod == ProcessingMethod_CommandLine ) {
                  CommandLine messageLog;
                  if ( messageLog.Status() == CommandLineStatus_Ok ) {
                        messageLog.AddToMessageQueue( errorMessage.Text() );
                        messageLog.FlushMessageQueue();
                        errorMessageStatus = Status_Success;
                  }

            //Эти код и комментарий новые и
            //являются результатом развертывания оператора if.
            else {
                  // Не можем ничего делать, так как процедура
                  // сама занимается обработкой ошибки.
            }

            //Это тоже новый код и комментарий.
            else {
                  // Не можем ничего делать, так как процедура
                  // сама занимается обработкой ошибки.
            }
      }
      // Если код ошибки некорректен, извещаем пользователя
      // об обнаружении внутренней ошибки.
      else {
            DisplayInteractiveMessage(
                  "Internal Error: Invalid error code in ReportErrorMessage()"
            );
      }
      // Вернуть статус.
      return errorMessageStatus;
}
```

К каждому комментарию добавлена одна или несколько строк кода. Каждый блок кода выражает некоторое намерение, описанное комментариями. Все переменные объявлены и определены рядом с местом их первого использования. Каждый комментарий обычно разворачивается в 2–10 строк кода.

Теперь вернемся к спецификации и псевдокоду. Первоначальная спецификация из пяти предложений превратилась в 15 строк псевдокода, которые в свою очередь развернуты в метод размером в страницу. Хотя спецификация и была достаточно подробной, создание метода потребовало проектировочных работ при написании псевдокода и кодировании. Это низкоуровневое проектирование и есть одна из причин, по которой «кодирование» является нетривиальной задачей.

1.5. Языки программирования: эволюция, классификация.

В развитии инструментального программного обеспечения (т.е. программного обеспечения, служащего для создания программных средств в любой проблемной области) рассматривают пять поколений языков программирования (ЯП). Языки программирования как средство общения человека с ЭВМ от поколения к поколению улучшали свои характеристики, становясь все более доступными в освоении непрофессионалам.

Первые три поколения ЯП характеризовались более сложным набором зарезервированных слов и синтаксисом. Языки четвертого поколения все еще требуют соблюдения определенного синтаксиса при написании программ, но он значительно легче для освоения. Естественные ЯП, разрабатываемые в настоящее время, составят пятое поколение и позволят определять необходимые процедуры обработки информации, используя предложения языка, весьма близкого к естественному и не требующего соблюдения особого синтаксиса.

Поколения	Языки программирования	Характеристика

Первое	Машинные	Ориентированы на использование в конкретной ЭВМ, сложны в освоении, требуют хорошего знания архитектуры ЭВМ
Второе	Ассемблеры, Макроассемблеры	Более удобны для использования, но по-прежнему машинно-зависимы
Третье	Языки высокого уровня	Мобильные, человеко-ориентированные, проще в освоении
Четвертое	Непроцедурные, объектно-ориентированные, языки запросов, параллельные	Ориентированы на непрофессионального пользователя и на ЭВМ с параллельной архитектурой
Пятое	Языки искусственного интеллекта, экспертных систем и баз знаний, естественные языки	Ориентированы на повышение интеллектуального уровня ЭВМ и интерфейса с языками

ЯП первого поколения представляли собой набор машинных команд в двоичном (бинарном) или восьмеричном формате, который определялся архитектурой конкретной ЭВМ. Каждый тип ЭВМ имел свой ЯП, программы на котором были пригодны только для данного типа ЭВМ. От программиста при этом требовалось хорошее знание не только машинного языка, но и архитектуры ЭВМ.

Рассмотрим проблему программирования в абсолютных (машинных) адресах.

Вычислительная машина (система), независимо от типа и поколения, состоит из двух основных типов устройств:

- центральное устройство (ЦУ), включающее в себя центральный процессор (ЦП) и оперативную память (ОП);
- периферийные (внешние) устройства (ВУ).

Несмотря на то, что постоянно идут исследования в направлении разработки новых принципов структур и архитектур ЭВМ, в подавляющем большинстве современных машин реализованы так называемые принципы Фон Неймана:

- ОП организована как совокупность машинных слов (МС) фиксированной длины или разрядности (имеется в виду количество двоичных единиц или бит, содержащихся в каждом МС). Например, ранние ПЭВМ имели разрядность 8, затем появились 16-разрядные, а в последнее время — 32- и 64-разрядные машины. В свое время существовали также 45-разрядные (М-20, М-220), 35-разрядные (Минск-22, Минск-32) и др. машины.
- ОП образует единое адресное пространство, адреса МС возрастают от младших к старшим;
- в ОП размещаются как данные, так и программы, причем в области данных одно слово, как правило, соответствует одному числу, а в области программы — одной команде (машинной иструкции — минимальному и неделимому элементу программы);
- команды выполняются в естественной последовательности (по возрастанию адресов в ОП), если/пока не встретится команда управления (условного/безусловного перехода), в результате которой естественная последовательность нарушится;
- ЦП может произвольно обращаться к любым адресам в ОП для выборки и/или записи в МС чисел или команд.

Обобщенный порядок функционирования некоторого абстрактного (упрощенного) ЦУ следующий:

- извлечение из ОП очередной команды. Типичная команда содержит: код операции (КОП), действия (сложение, вычитание чисел; сравнение строк; передача управления, обращение к ВУ и пр.); адреса операндов А1, А2 и т.д., участвующих в выполнении команды (чисел, строк, других команд программы);
- расшифровка КОП;
- выборка адресов А1, А2 и пр. в регистры адреса;
- выполнение операции (арифметической, логической и пр.) и помещение результата в регистр результата;
- запись результата по одному из адресов (если необходимо);
- переход к следующей команде.

Таким образом, программирование в машинных адресах требует знания системы команд конкретной ЭВМ и их адресности. При этом реализация даже довольно несложных вычислений требует разложения их на простые операции, что значительно увеличивает общий объем программы и затрудняет ее чтение и отладку.

В качестве примера рассмотрим последовательность реализации вычисления по формуле $y = (a + b)^2 - c/d$.

План последовательности машинных операций, выполнение которой приведет к нужному результату, в данном случае следующий:

1. rl = a + b; —операция сложения
2. r2 = r1*r1; —операция умножения

3. r3 = c/d; —операция деления
4. y = r2-r3; —операция вычитания
5. Стоп —завершение обработки

Здесь количество переменных, необходимых для хранения промежуточных результатов, связано с адресностью системы команд и с тем, разрешено или нет в процессе вычислений изменять значения исходных данных.

Некоторые из приведенных выше операций могут быть в свою очередь разложены на несколько операций. Это обусловливается самой системой команд (например, если операция сложения может складывать только величины, находящиеся в специально отведенных для этой цели регистрах, то потребуется дополнительно использовать операции пересылки данных в эти регистры).

Адресность команд ЦП связана с разрядностью ОП. Типичная команда состоит из фиксированной части (КОП) и адресной части, в которой указаны адреса операндов. Увеличение разрядности позволяет увеличить адресность команды и длину адреса (т.е. объем памяти, доступной данной команде). Увеличение адресности, в свою очередь, приводит к повышению быстродействия обработки (за счет снижения числа требуемых команд). С увеличением разрядности увеличивается также и диапазон значений обрабатываемых чисел и количества информации, извлекаемой за один такт работы машины из ОП, или записываемой в ОП. Ранние ПЭВМ имели разрядность 8, объем ОП 64 Кбайт, последние модели обладают разрядностью 32-64 и объемом ОП 64-512 Мбайт.

Второе поколение ЯП характеризуется созданием языков ассемблерного типа (ассемблеров, макроассемблеров), позволяющих вместо двоичных и других форматов машинных команд использовать их мнемонические символьные обозначения (имена). Являясь существенным шагом вперед, ассемблерные языки все еще оставались машинно-зависимыми, а программист все также должен был быть хорошо знаком с организацией и функционированием аппаратной среды конкретного типа ЭВМ. При этом ассемблерные программы все так же затруднительны для чтения, трудоемки при отладке и требуют больших усилий для переноса на другие типы ЭВМ. Однако и сейчас ассемблерные языки используются при необходимости разработки высокоэффективного программного обеспечения (минимального по объему и с максимальной производительностью).

Текст исходной программы на ассемблере состоит из операторов, каждый из которых занимает отдельную строку этого текста. Различают два типа операторов: инструкции и директивы. Первые при трансляции преобразуются в команды процессора, которые исполняются после загрузки в память загрузочного модуля программы, имеющего расширение. COM или .EXE . Операторы второго типа управляют процессом ассемблирования — преобразования текста исходной программы в коды объектного модуля (расширение .OBJ). Ассемблер интерпретирует и обрабатывает операторы один за другим, генерируя последовательность из команд процессора и байтов данных.

Особо следует остановиться на использовании макрокоманд. При программировании на макроассемблере можно формировать обращение к часто повторяющейся последовательности команд при помощи одного оператора. Этот прием несколько напоминает вызов подпрограмм в языках высокого уровня, но между ними лежит значительное различие, заключающееся в том, что подпрограмма, занимающая некоторый участок памяти, может быть исполнена неограниченное число раз путем передачи ей управления из вызывающей программы, в которую подпрограмма сама затем возвращает управление. В ассемблере используются макровызовы макроопределений. Макроопределение — это последовательность операторов, которые могут содержать формальные параметры. Макроопределение и команда обращения к макроопределению (макровызов) образуют макрокоманду. Макровызов — это оператор вызова макроопределения. Если макроопределение содержит формальные параметры, то макровызов обязан содержать фактические значения этих параметров, которые будут подставлены вместо соответствующих формальных. В результате макровызова формируется реальная последовательность команд — макрорасширение. Макрорасширение вставляется в исходный текст программы на место оператора макровызова. Таким образом, в исходный текст программы макрорасширение одного и того же макроопределения может быть вставлено несколько раз, по числу макровызовов. Каждое макрорасширение после трансляции, естественно, занимает свой участок памяти.

Третье поколение ЯП начинается с появления в 1956 г. первого языка высокого уровня — Fortran, разработанного под руководством Дж. Бэкуса в фирме IBM. За короткое время Fortran становится основным ЯП при решении инженерно-технических и научных задач. Первоначально Fortran обладал весьма ограниченными средствами обеспечения работы с символьной информацией и с системой ввода-вывода. Однако постоянное развитие языка сделало его одним из самых распространенных ЯВУ на ЭВМ всех классов — от микро- до супер ЭВМ, а его версии используются и для вычислительных средств нетрадиционной параллельной архитектуры.

Вскоре после языка Fortran появились такие ныне широко известные языки, как Algol, Cobol, Basic, PL/1, Pascal, APL, ADA, C, Forth, Lisp, Modula и др. В настоящее время насчитывается свыше 2000 различных языков высокого уровня.

Языки четвертого поколения носят ярко выраженный непроцедурный характер, определяемый тем, что программы на таких языках описывают только что, а не как надо сделать. В программах формируются скорее соотношения, а не последовательности шагов выполнения алгоритмов. Типичными примерами непроцедурных языков являются языки, используемые для задач искусственного интеллекта (например,

Prolog, Langin). Так как непроцедурные языки имеют минимальное число синтаксических правил, они значительно более пригодны для применения непрофессионалами в области программирования.

Второй тенденцией развития ЯП четвертого поколения являются объектно-ориентированные языки, базирующиеся на понятии программного объекта, впервые использованного в языке Simula-67 и составившего впоследствии основу известного языка SmallTalk. Программный объект состоит из структур данных и алгоритмов, при этом каждый объект знает, как выполнять операции со своими собственными данными. На самом деле, различные объекты могут пользоваться совершенно разными алгоритмами при выполнении действий, определенных одним и тем же ключевым словом (так называемое свойство полиморфизма). Например, объект с комплексными числами и массивами в качестве данных будет использовать различные алгоритмы для выполнения операции умножения. Такими свойствами обладают объектно-ориентированные Pascal, Basic, C++, SmallTalk, Simula, Actor и ряд других языков программирования.

Третьим направлением развития языков четвертого поколения можно считать языки запросов, позволяющих пользователю получать информацию из баз данных. Языки запросов имеют свой особый синтаксис, который должен соблюдаться, как и в традиционных ЯП третьего поколения, но при этом проще в использовании. Среди языков запросов фактическим стандартом стал язык SQL (Structured Query Language).

И, наконец, четвертым направлением развития являются языки параллельного программирования (модификация ЯВУ Fortran, языки Occam, SISAL, FP и др.), которые ориентированы на создание программного обеспечения для вычислительных средств параллельной архитектуры (многомашинные, мультипроцессорные среды и др.), в отличие от языков третьего поколения, ориентированных на традиционную однопроцессорную архитектуру.

К интенсивно развивающемуся в настоящее время пятому поколению относятся языки искусственного интеллекта, экспертных систем, баз знаний (InterLisp, ExpertLisp, IQLisp, SAIL и др.), а также естественные языки, не требующие освоения какого-либо специального синтаксиса (в настоящее время успешно используются естественные ЯП с ограниченными возможностями — Clout, Q&A, HAL и др.).

Классификация ЯП. Изучение ЯП часто начинают с их классификации. Определяющие факторы классификации обычно жестко не фиксируются. Чтобы продемонстрировать характер типичной классификации, опишем наиболее часто применяемые факторы, дадим им условные названия и приведем примеры ЯП для каждой из классификационных групп.

Элементы языков программирования могут рассматриваться на следующих уровнях:

алфавит — совокупность символов, отображаемых на устройствах печати и экранах и/или вводимых с клавиатуры терминала. Обычно это набор символов Latin-1, с исключением управляющих символов. Иногда в это множество включаются неотображаемые символы, с указанием правил их записи (комбинирование в лексемы);

лексика — совокупность правил образования цепочек символов (лексем), образующих идентификаторы (переменные и метки), операторы, операции и другие лексические компоненты языка. Сюда же включаются зарезервированные (запрещенные, ключевые) слова ЯП, предназначенные для обозначения операторов, встроенных функций и пр.

Фактор	Характеристика	Группы	Примеры ЯП
Уровень ЯП	Степень близости ЯП к архитектуре компьютера	Низкий	Автокод, ассемблер
		Высокий	Fortran, Pascal, ADA, Basic, C и др. ЯВУ
		Сверхвысокий	Сетл
Специали – зация ЯП	Потенциальная или реальная область применения	Общего назначения (универсальные)	Algol, PL/1, Simula, Basic, Pascal
		Специализированные	Fortran (инженерные расчеты), Cobol (коммерческие задачи), Refal, Lisp (символьная обработка), Modula, Ada (программирование в реальном времени)
Алгорит- мичность (процедур ность)	Возможность абстрагироваться от деталей алгоритма решения задачи. Алгоритмичность тем выше, чем точнее приходится планировать порядок выполняемых действии	Непроцедурные	Prolog, Langin
		Процедурные	Ассемблер, Fortran, Basic, Pascal, Ada

Иногда эквивалентные лексемы, в зависимости от ЯП, могут обозначаться как одним символом алфавита, так и несколькими. Например, операция присваивания значения в Си обозначается как «=», а в языке Pascal — «:=». Операторные скобки в Си задаются символами «{» и «}», а в Pascal — BEGIN и END. Граница между лексикой и алфавитом, таким образом, является весьма условной, тем более что компилятор обычно на фазе лексического анализа заменяет распознанные ключевые слова внутренним кодом (например, BEGIN — 512, END — 513) и в дальнейшем рассматривает их как отдельные символы;

синтаксис — совокупность правил образования языковых конструкций, или предложений ЯП — блоков, процедур, составных операторов, условных операторов, операторов цикла и пр. Особенностью синтаксиса является принцип вложенности (рекурсивность) правил построения конструкций. Это значит, что элемент синтаксиса языка в своем определении прямо или косвенно в одной из его частей содержит сам себя. Например, в определении оператора цикла телом цикла является оператор, частным случаем которого является все тот же оператор цикла;

семантика — смысловое содержание конструкций, предложений языка, семантический анализ — это проверка смысловой правильности конструкции. Например, если мы в выражении используем переменную, то она должна быть определена ранее по тексту программы, а из этого определения может быть получен ее тип. Исходя из типа переменной, можно говорит о допустимости операции с данной переменной. Семантические ошибки возникают при недопустимом использовании операций, массивов, функций, операторов и пр.

Исходная программа, как правило, состоит из следующих частей (впервые эти требования были сформулированы в языке Cobol):

- раздел идентификации — область, содержащая наименование программы, а также дополнительную информацию для программистов и/или пользователей;
- раздел связи — фрагмент текста, описывающий внешние переменные, передаваемые вызывающей программой (если таковая имеется), т.е. ту часть исходных данных, которая обязательно поступает на вход программы при ее запуске. Эти переменные часто называют параметрами программы;
- раздел оборудования (среда) — описание типа ЭВМ, процессора, требований к оперативной и внешней памяти, существенных с точки зрения выполнимости программы.

Дело в том, что даже среди семейства однотипных ЭВМ могут существовать отличия в наборе машинных инструкций (команд), средств программирования ввода-вывода, кодированного представления данных, в связи с чем описание среды, приводимое в данном разделе оказывается необходимым транслятору с языка с точки зрения оптимизации выполнения или вообще оценки возможности создания рабочей программы;

- раздел данных — идентификация (декларация, объявление, описание) переменных, используемых в программе, и их типов. Понятие типа позволяет осуществлять проверку данных на совместимость в операциях еще на этапе трансляции программы и отвергнуть недопустимые преобразования;
- раздел процедур — собственно программная часть, содержащая описание процессов обработки данных. Элементами процедуры являются операторы и стандартные функции, входящие в состав соответствующего языка программирования.

К типовым операторам управления вычислительным процессом относятся следующие

- организация циклов (выполняемых до исчерпания списка или до достижения управляющей переменной заданного значения, или пока выполняются некоторые условия);
- ветвление программы — выполнение альтернативных групп операторов при заданных условиях;
- блоки операторов — группы, выполняемые как целое.

Примечание: перечисленные операторы принято называть операторами структурного программирования, языки, в которых других средств управления программой нет, — языками структурного программирования;

операторы перехода — условная или безусловная передача управления на определенный оператор, снабженный меткой, или условный/безусловный выход из цикла или блока.

Важным компонентом управляющих операторов обычно являются логические условия, сопоставляющие значения констант, функций, переменных отношениями типа >=, <=, >, <, ≠ и вырабатывающие решение о продолжении цикла, переходе по ветви, выходе из блока и т.д. Несколько простых условий объединяются в составные посредством логических операций И, ИЛИ, НЕ.

Операторы присваивания значений выполняют следующие функции:

- пересылку значений переменных, констант, функций в принимающую переменную;
- вычисление значений арифметической (числовой) переменной в рамках существующих в языке правил построения арифметических выражений (обычно включающих символы «()» — скобки, «+» — сложение, «-» — вычитание, «/» — деление, «*» — умножение, «**» или «^» — возведение в степень);
- вычисление значений строчной (символьной) переменной путем соединения, пересечения, вычисления строк;
- вычисление логических переменных в рамках правил образования логических выражений (обычно включающих скобки и операции И, ИЛИ, НЕ).

Аналогично могут быть выделены типичные группы функций:

- стандартные алгебраические и арифметические — SIN, COS, SQRT, MIN, MAX и др.;
- стандартные строчные — выделение, вставка, удаление подстроки и т.д.;

- нестандартные функции — описание операций и форматов ввода-вывода данных; преобразование типов данных; описание операций над данными, специфичными для конкретной системы программирования, ОС или типа ЭВМ.

В таблице дается обзор основных операторных возможностей ЯП: Pascal, Basic и C (Си).

Элемент языка	Pascal	Basic	C
Идентификация, связь	PROGRAM Имя (Параметры)		MAIN (Параметры)

Циклы	for...do
	repeat…until
	while...do
		For...Next
		Do...Loop
		While...Wend
			for()
			do...while ()
			while ()

Выход из цикла во вне	break	Exit For Exit Do	break
Переход к заголовку цикла	continue		continue
Блоки, составные операторы	begin...end		{}
Ветвление	if...then...else case	If...Then... Else Select Case	if()...else switch
Переход на метку	goto Label	GoTo Label	goto Label
Логические операции (НЕ, И, ИЛИ)	not, and, or	not, and, or	! && !!
Пересылка, присваивание	:=	=	=
Арифметические операции	+, -, *, /, div, mod	+, -, *, /, \, ^, mod	+, -, *, /, %
Стандартные числовые функции	Abs, Sqrt, Sqr, Sin, Cos, LogN	Abs, Sqr, Sin, Cos, Exp, Log	В стандартном языке — нет
Строчные функции	Pos, Copy, Delete, Insert	Left, Right, Instr, Mid	strcat, strlin
Ввод-вывод	Read, Write, BlockRead, BlockWrite	Print, Input, Write, Get, Put	printf, scanf, get, put
Комментарии	(*текст*) {текст} // текст	REM текст 'текст	/* текст */

Необходимо отметить, что конкретные ЯП могут не требовать наличия всех выше перечисленных разделов исходного модуля. В некоторых случаях описания переменных могут размещаться произвольно в тексте, или опускаются, при этом тип переменной определяется компилятором, исходя из системы умолчаний; есть средства программирования, в которых тип переменной задается в момент присвоения ей значения другой переменной или константы и т.д. Существуют фрагменты описания данных, которые могут быть отнесены как к разделу данных, так и к разделу оборудования (указания на устройство, длину и формат записи, организацию файла и т.п.).

Подпрограммы. При разработке программ на алгоритмических языках широко используется понятие подпрограммы. Подпрограмма — это средство, позволяющее многократно использовать в разных местах основной программы один раз описанный фрагмент алгоритма.

В большинстве ЯП не проводится концептуального различия между такими объектами, как программа и подпрограмма (процедура, функция). В связи с этим всякая подпрограмма может приобретать иерархическую структуру, включая в себя подчиненные (вызываемые) подпрограммы, процедуры и т.д., имеющие стандартный состав.

Объявления (типы, переменные, константы), использующиеся любой подпрограммой, относятся к одной из двух категорий — категории локальных объявлений и категории глобальных объявлений. Локальные объявления принадлежат подпрограмме, описаны внутри нее и могут использоваться только ею. Глобальные объявления принадлежат программе в целом и доступны как самой программе, так и всем ее подпрограммам. Обмен данными между основной программой и ее подпрограммами обычно осуществляется посредством глобальных переменных.

Если имя глобального объявления совпадает с именем локального, то внутри подпрограммы обычно объявление интерпретируется как локальное, и все изменения, вносимые, например, в значение такой переменной, актуальны только в рамках подпрограммы.

Формальные и фактические параметры. Объявление подпрограммы может содержать список параметров, которые называются формальными. Каждый параметр из списка формальных параметров является локальным по отношению к подпрограмме, для которой он объявлен. Это означает, что глобальные переменные, имена которых совпадают с именами формальных параметров, становятся недоступными для использования в подпрограмме. Все формальные параметры можно разбить на две категории:

- параметры, вызываемые подпрограммой по своему значению (т.е. параметры, которые передают в подпрограмму свое значение и не меняются в результате выполнения подпрограммы);
- параметры, вызываемые подпрограммой по наименованию (т.е. параметры, которые становятся доступными для изменения внутри подпрограммы).

Главное различие этих двух категорий — в механизме передачи параметров в подпрограмму. При вызове параметра по значению происходит копирование памяти, занимаемой параметром, в стек и использование в дальнейшем в операторах подпрограммы локальной копии параметра. Основное значение параметра (глобальное по отношению к подпрограмме) при этом остается без изменения. Следует отметить, что использование такого механизма при передаче, например, массивов большой длины может отрицательно влиять на быстродействие программы и заполняет стек лишней информацией.

Замечание. Во многих случаях программе требуется временно запомнить информацию, а затем считывать ее в обратном порядке. Эта проблема в ПК решена посредством реализации стека LIFO («последний пришел — первый ушел»), называемого также стеком включения/извлечения (stack — кипа, например, бумаг). Наиболее важное использование стека связано с процедурами. Стек обычно рассчитан на косвенную адресацию через регистр SP — указатель стека. При включении элементов в стек производится автоматический декремент указателя стека, а при извлечении — инкремент, т. е. стек всегда «растет» в сторону меньших адресов памяти. Адрес последнего включенного в стек элемента называется вершиной стека (TOS).

При вызове параметра по наименованию в подпрограмму передается адрес памяти (глобальной по отношению к подпрограмме), в которой размещено значение параметра, т. е. в качестве локальной переменной выступает ссылка на глобальное размещение параметра, обеспечивающая доступ к самому значению.

При обращении к подпрограмме формальные параметры заменяются на соответствующие по типу и категории фактические параметры вызывающей программы или подпрограммы.

Способы описание языков программирования. Для описания языков программирования будут используются системы описания:

- **нотация Бэкуса** (впервые предложена при описании языка ALGOL);
- **нотация IBM** (разработана фирмой для описания языков COBOL и JCL);
- нотация Бэкуса (использована ниже для описания ЯП Паскаль, который является прямым потомком Алгола) содержит конструкции следующего вида:

<Оператор присваивания> ::= <Переменная> := <Выражение>
<Идентификатор> ::= <Буква>|<Идентификатор><Буква>|
<Идентификатор><Цифра>

Левая часть определения конструкции языка содержит наименование определяемого элемента, взятого в угловые скобки.

Правая часть включает совокупность элементов, соединенных знаком |, который трактуется как «или» и объединяет альтернативы — различные варианты значения определяемого элемента.

Части соединяются оператором ::=, который может трактоваться как есть по определению.

Существенно, что многие определения носят рекурсивный характер, т.е. левая часть содержит правую (как для элемента <Идентификатор> в вышеприведенных примерах), и вместо элементов в правой части определения можно подставлять любые их значения. Например, в определении идентификатора, начиная рассматривать его как букву A, можно затем получать любые комбинации: AA, AO, AAI, AA1B и т.п.

Нотация IBM (используется ниже при описании конструкций ЯП C и Basic) включает следующие конструкции:

< > угловые скобки (или двойные кавычки " ") обозначают элементы программы, определяемые пользователем (<идентификатор>, <список параметров> <условие> и пр. В соответствующих местах реальной программы будет находиться идентификатор переменной и т.д.);

[] квадратные скобки — ограничивающие синтаксическую конструкцию, обозначают ее возможное отсутствие. Например, return [<выражение>]; В этой конструкции <выражение> не обязательно;

— вертикальная черта разделяет список значений обязательных элементов, одно из которых должно быть выбрано;

... — горизонтальное многоточие, следующее после некоторой синтаксической конструкции, обозначает последовательность конструкций той же самой формы, что и предшествующая многоточию конструкция. Например, ={<выражение> [,<выражение>]...} обозначает, что одно или более выражений, разделенных запятыми, может появиться между фигурными скобками.

Например:

[Private | Public | Friend] [Static] Sub <идентификатор>
[(<список параметров>)] [<оператор>]
…
[Exit Sub]

[<оператор>]
…
End Sub

Рекурсивные определения в IBM-нотации не используются.

Вопросы.

1. Охарактеризуйте факторы классификации языков программирования.
2. Перечислите основные группы операторов языков программирования.
3. Опишите операцию ветвления с помощью нотации Бэкуса и с помощью IBM-нотации.
4. Опишите оператор цикла С использованием нотации Бэкуса и IBM-нотации.
5. Опишите оператор присваивания с использованием нотации Бэкуса.
6. Дайте определение арифметической операции для языков Pascal, Basic и С с использованием нотации Бэкуса.
7. Дайте определение логической операции для языков Pascal, Basic и С с использованием нотации Бэкуса.
8. Охарактеризуйте разницу между формальными и фактическими параметрами подпрограммы.
9. Охарактеризуйте разницу между вызовом параметров подпрограммы по значению и по наименованию. Приведите примеры.
10. Охарактеризуйте возможные параметры подпрограмм вычисления следующих функций: cos x, ex, 2πR; loge x.

1.6. Системы программирования.

Рассмотренные выше средства являются важными функциональными компонентами соответствующей системы программирования, т.е. среды окружения программиста, позволяющей ему разрабатывать прикладные программы (программировать приложения, разрабатывать приложения) для соответствующих ЭВМ и операционных систем.

Система программирования представляет собой совокупность средств разработки программ (языки программирования, текстовые редакторы, трансляторы, редакторы связей, библиотеки подпрограмм, утилиты и обслуживающие программы), обеспечивающих автоматизацию составления и отладки программ пользователя.

Системы программирования классифицируются по признакам, приведенным в таблице.

Признак классификации	Типы
Набор исходных языков	Одноязыковые
	Многоязыковые
Возможности расширения	Замкнутые
	Открытые
Трансляция	Компиляция
	Интерпретация

Следует отметить, что:

- отличительной особенностью многоязыковых систем является то, что отдельные части (секции, модули или сегменты) программы могут быть подготовлены на различных языках и объединены во время или перед выполнением в единый модуль;
- в открытую систему можно ввести новый входной язык с транслятором, не требуя изменений в системе;
- в интерпретирующей системе осуществляется покомандная расшифровка и выполнение инструкций входного языка (в среде данной системы программирования); в компилирующей — подготовка результирующего модуля, который может выполняться на ЭВМ практически независимо от среды.

Рассмотрим структуру абстрактной многоязыковой, открытой, компилирующей системы программирования и процесс разработки приложений в данной среде.

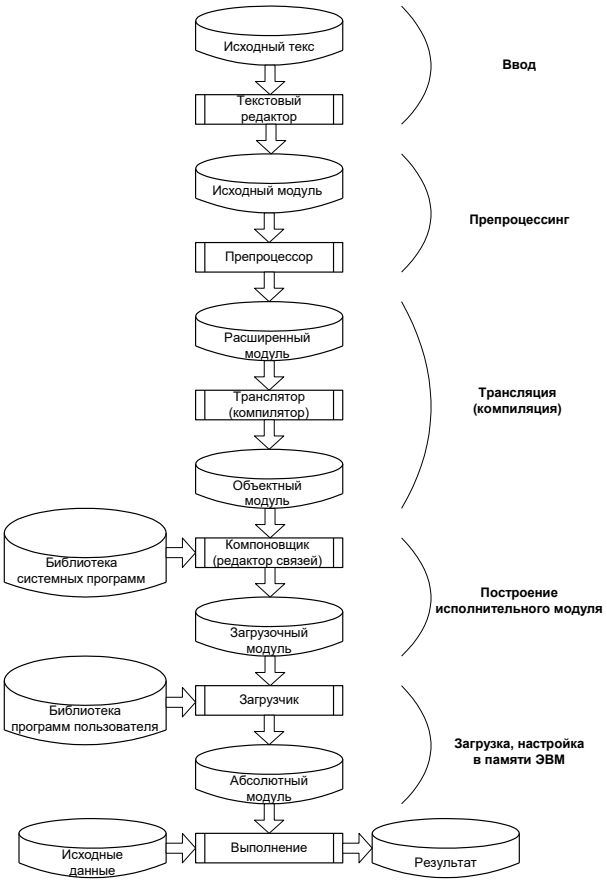


Рис. 7

Ввод. Программа на исходном языке (исходный модуль) готовится с помощью текстовых редакторов и в виде текстового файла или раздела библиотеки поступает на вход транслятора.

Трансляция исходной программы есть процедура преобразования исходного модуля в промежуточную, так называемую объектную форму. Трансляция в общем случае включает в себя препроцессинг (предобработку) и компиляцию.

Препроцессинг — необязательная фаза, состоящая в анализе исходного текста, извлечения из него директив препроцессора и их выполнения.

Директивы препроцессора представляют собой помеченные спецсимволами (обычно %, #, &) строки, содержащие аббревиатуры или другие символические обозначения конструкций, включаемых в состав исходной программы перед ее обработкой компилятором.

Данные для расширения исходного текста могут быть стандартными, определяться пользователем либо содержаться в системных библиотеках ОС.

Компиляция — в общем случае многоступенчатый процесс, включающий следующие фазы:

- синтаксический анализ — проверка правильности конструкций, использованных программистом при подготовке текста;
- семантический анализ — выявление несоответствий типов и структур переменных, функций и процедур;
- генерация объектного кода — завершающая фаза трансляции.

Выполнение трансляции (компиляции) может осуществляться в различных режимах, установка которых производится с помощью ключей, параметров или опций. Может быть, например, потребовано только выполнение фазы синтаксического анализа и т.п.

Объектный модуль представляет собой текст программы на машинном языке, включающий машинные инструкции, словари, служебную информацию.

Объектный модуль не работоспособен, поскольку содержит неразрешенные ссылки на вызываемые подпрограммы библиотеки транслятора (в общем случае — системы программирования), реализующие функции ввода-вывода, обработки числовых и строчных переменных, а также на другие программы пользователей или средства пакетов прикладных программ.

Построение **загрузочного модуля** осуществляется специальными программными средствами — редактором связей, построителем задач, компоновщиком, основной функцией которых является объединение объектных и загрузочных модулей в единый загрузочный модуль с последующей записью в библиотеку или файл. Полученный модуль в дальнейшем может использоваться для сборки других программ и т.д., что создает возможность наращивания программного обеспечения.

Загрузочный модуль после сборки либо помещается в качестве раздела в пользовательскую библиотеку программ, либо в качестве последовательного файла на накопителе на магнитном диске (НМД). Выполнение модуля состоит в загрузке его в оперативную память, настройке по месту в памяти и передаче ему управления. Образ загрузочного модуля в памяти называется абсолютным модулем, поскольку все команды ЭВМ здесь приобретают окончательную форму и получают абсолютные адреса в памяти. Формирование абсолютного модуля может осуществляться как программно, путем обработки командных кодов модуля программой-загрузчиком, так и аппаратно, путем применения индексирования и базирования команд загрузочного модуля и приведения указанных в них относительных адресов к абсолютной форме.

Современные системы программирования позволяют удобно переходить от одного этапа к другому. Это осуществляется в рамках так называемой интегрированной среды программирования, которая содержит в себе текстовый редактор, компилятор, компоновщик, встроенный отладчик и, в зависимости от системы или ее версии, предоставляет программисту дополнительные удобства для написания и отладки программ.

Библиотеки подпрограмм. В предыдущем разделе определено понятие подпрограммы как средства многократного использования одного и того же фрагмента алгоритма в разных местах основной программы (например, вычисление одной и той же арифметической функции при разных значениях аргумента).

Однако довольно распространенной является и такая ситуация, когда один и тот же алгоритм — вычисление значений элементарных функций, перевод чисел из одной системы в другую, преобразование символьной информации к строчному или заглавному представлению и т.д. — используется при решении самых разных задач. Если один из подобных алгоритмов уже был один раз разработан и реализован при решении некоторой задачи, то возникает естественное желание использовать уже готовую подпрограмму как часть любой другой программы. Таким образом, организовав сбор, хранение и удобное использование готовых подпрограмм, можно существенно упростить и ускорить разработку программ различного назначения, записывая лишь обращения к готовым подпрограммам и заново проектируя уже гораздо меньшую часть кода.

В процессе использования готовых подпрограмм возникает ряд проблем, связанных, с одной стороны, с хранением имеющихся подпрограмм и размещением используемых подпрограмм в памяти ЭВМ, и с другой стороны — с организацией их взаимодействия с основной программой. В связи с этим для обеспечения удобства практической работы выбирается определенная система использования подпрограмм, в которой тем или иным образом эти проблемы решены. Такая система всегда предъявляет определенные требования к подпрограммам с точки зрения их организации и оформления. Подпрограммы, которые удовлетворяют всем требованиям выбранной системы, называются стандартными, а организованный соответствующим образом набор таких подпрограмм называется библиотекой подпрограмм.

С точки зрения компоновки и последующего взаимодействия с основным программным кодом библиотеки подпрограмм делятся на библиотеки статического вызова (**статические библиотеки**) и библиотеки динамического вызова (**динамические библиотеки**). Охарактеризуем разницу между статической и динамической компоновкой подпрограмм. В любом случае, когда подпрограмма оказывается недоступной напрямую в исходном файле, компилятор добавляет эту подпрограмму в специальную внутреннюю таблицу, содержащую все внешние идентификаторы. Очевидно, что при этом компилятор должен иметь в своем распоряжении объявление подпрограммы и информацию о ее параметрах.

После компиляции подпрограммы статической библиотеки компоновщик добавляет ее откомпилированный код к исполняемой программе. Получившийся в результате исполнительный модуль содержит код программы и всех используемых подпрограмм.

В случае динамической компоновки компоновщик просто использует информацию о подпрограмме для настройки соответствующих таблиц в исполняемом файле. Когда исполняемый модуль загружается в память, операционная система загружает также все необходимые динамические библиотеки и заполняет внутренние таблицы программы адресами библиотечных подпрограмм в памяти, после чего программа запускается на исполнение.

Схема вызова подпрограмм статической и динамической библиотек изображена на рисунке.

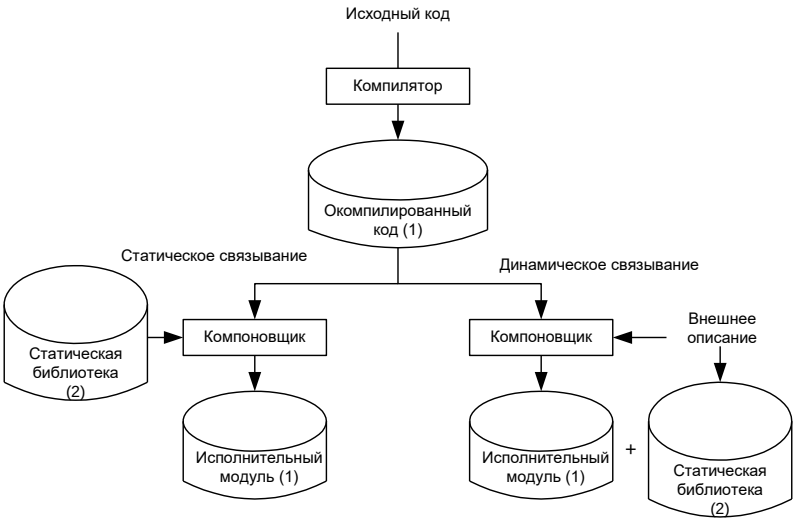


Рис. 8

К преимуществам использования динамических библиотек можно отнести следующие:

- во-первых, если разные программы используют одну и ту же динамическую библиотеку, то библиотека загружается в память только один раз (в отличие от статической компоновки, при которой каждая отдельная программа содержит внутри себя коды всех используемых подпрограмм). Таким образом, экономится системная память;
- во-вторых, при модификации динамической библиотеки можно просто заменить старую версию новой, не перестраивая при этом готовые программы (если, конечно, изменения не коснулись параметров вызова используемых программами подпрограмм).

Эти общие преимущества оказываются полезными в следующих случаях:

- если несколько программ используют один и тот же сложный алгоритм: сохранение его в динамической библиотеке уменьшит размер исполнительного модуля и сэкономит память, когда несколько программ будут запущены одновременно;
- если необходимо постоянно вносить изменения в программу большого объема: разделение программы на исполняемую часть и динамическую библиотеку позволит модифицировать и затем распространять только измененную часть программы.

Обработка исключительных ситуаций. Для обеспечения надежности прикладных программ операционные системы предоставляют программисту специальные средства обработки исключительных ситуаций — аппарат обнаружения и обработки ошибок и граничных состояний. Наличие подобных средств позволяет при написании программного кода сосредоточиться на решении основной задачи и отделить основную работу от рутинной, хотя и необходимой, обработки ошибок.

Рассмотрим действие такого аппарата на примере структурной обработки исключений (SEH — Structured Exeption Handling), введенной фирмой Microsoft в операционную систему Win32.

Основная нагрузка при поддержке SEH ложится не на операционную систему, а на компилятор, который должен сгенерировать специальный код на входах и выходах так называемых блоков исключений и создать таблицы вспомогательных структур данных для поддержки SEH. Разные фирмы по-разному реализуют SEH в своих компиляторах, но большинство фирм-разработчиков компиляторов придерживается синтаксиса, рекомендованного Microsoft.

SEH предоставляет две основные возможности: **обработку завершения и обработку исключений**.

Обработка завершения гарантирует, что некоторый блок кода (обработчик завершения) будет исполнен независимо от того, каким образом произойдет выход из другого блока кода (защищенного блока). Общий синтаксис использования обработки завершения в прикладных программах следующий:

try
<защищенный блок>
finally
<обработчик завершения>

Ключевые слова **try** и **finally** обозначают два программных блока обработки завершения. Благодаря совместным действиям операционной системы и компилятора гарантируется, что код блока finally будет исполнен независимо от того, каким образом произойдет выход из защищенного блока. Порядок исполнения кода при этом следующий:

1. Исполняется код перед блоком try.
2. Исполняется код внутри блока try (защищенный блок).
3. Исполняется код внутри блока fatally (обработчик завершения).

4. Исполняется код после блока, finally.

Рассмотрим пример: пусть необходимо разместить в оперативной памяти и заполнить некоторыми значениями массив целых чисел, провести обработку элементов массива (например, каждый элемент с четным номером возвести в квадрат и посчитать сумму элементов) и освободить оперативную память. Поместив обработку элементов массива в блок try, а освобождение памяти — в блок finally, можно таким образом обеспечить освобождение памяти при любом исходе — даже если в процессе выполнения алгоритма обработки массива произошла какая-нибудь ошибка. Схема решения такой задачи следующая:

<выделение памяти для размещения массива>
<заполнение элементов массива>
try
<выполнение алгоритма обработки>
<вывод результата>
finally
<освобождение занятой памяти>

Обработка исключений. Исключение — это не предполагаемое алгоритмом событие. В хорошо написанной программе обычно не должно быть обращений по недопустимому адресу памяти или деления на нуль, но все же такие ошибки иногда случаются. За перехват попыток обращения по недопустимому адресу и деления на нуль отвечает центральный процессор, возбуждающий исключения в ответ на такие ошибки. Исключение, возбуждаемое процессором, называется аппаратным исключением. Операционная система и прикладная программа способны также вызывать и программные исключения.

При возбуждении аппаратного или программного исключения операционная система дает возможность прикладной программе определить тип исключения и самостоятельно его обработать. Синтаксис обработки исключений следующий:

try
<Защищенный блок>
exept <фильтр исключений>
<обработчик исключений>

Ключевое слово exept отделяет защищенный блок от блока обработки исключений. Фильтр исключений предназначен для задания значений, определяющих типы исключительных ситуаций, возникновение которых предполагается обрабатывать в блоке исключений.

В отличие от обработчиков завершений, фильтры и обработчики исключений выполняются непосредственно операционной системой, нагрузка на компилятор при этом незначительна.

Блок обработки исключений начинает выполняться тогда, когда возникла исключительная ситуация при выполнении защищенного блока. После выполнения блока исключений управление передается на блок, следующий сразу за блоком исключений:

1. Исполняется код перед блоком try.
2. Исполняется код внутри блока try (защищенный блок). Если возникла исключительная ситуация, то переходим к пункту 3, иначе переходим к пункту 4.
3. Исполняется код внутри блока exept (обработчик исключений).
4. Исполняется код после блока exept.

Из приведенной последовательности действий видно, что если при исполнении защищенного блока не произошло ошибки, то блок exept не выполняется. В случае же возникновения ошибки внутри защищенного блока управление передается в обработчик исключений, а после обработки исключительной ситуации не возвращается в защищенный блок. Поясним это следующими примерами.

Пример 1. Пусть защищенный блок содержит одно действие — присваивает целочисленной переменной X значение 0. Тогда предпосылок для возникновения ошибочной ситуации в защищенном блоке нет, и блок исключений никогда не выполняется:

try
X = 0
exept <фильтр исключений>
Сообщение: «Обработка ошибок» — этот код никогда не выполняется

Пример 2. Пусть целочисленной переменной X присваивается значение 0 перед входом в защищенный блок, а внутри защищенного блока выполняется последовательность из двух действий: Y = 10/X и X = X + 5. Тогда получаем следующую ситуацию:

X = 0
try
Y = 10 /X — возбуждается исключение
X = X + 5 — этот код никогда не выполняется
exept <фильтр исключений>
Сообщение: «Деление на 0» — обработка исключения

При программировании обработки исключений необходимо помнить, что за блоком try всегда должен следовать либо блок finally, либо блок exept. Для одного блока try нельзя определить одновременно и блок finally, и блок exept. Нельзя также и указать несколько блоков finally или exept для одного блока try. Однако блоки try-exept и try-finally могут вкладываться друг в друга, т.е. возможны следующие способы использования:

```
try
…
        try
        …
        exept <фильтр исключений>
…
finally
…
```

или

```
try
…
finally
        …
        try
        …
        exept <фильтр исключений>
        …
```

Синтаксис использования аппарата SEH в конкретной системе программирования будет рассмотрен дополнительно.

1.7. Файлы данных.

Ввод-вывод данных в языках программирования осуществляется путем взаимодействия программы с внешними файлами. Внешний файл — это поименованный файл на диске или устройство ввода-вывода (например, клавиатура или дисплей). Во внешних файлах сохраняются результаты работы программы и располагаются данные, служащие источником информации, необходимой для ее функционирования.

Файл можно определить как логически непрерывный именованный набор данных на внешнем носителе.

Понятие файла появляется впервые в операционной системе OS/360 фирмы IBM, причем в ранних версиях системы «настоящим файлом» считался только перфокарточный массив (file = картотека), | данные на МД и МЛ обозначались как DS (Data Set — набор данных). В последующих ОС (RSX, UNIX, MS DOS) файлами становятся именованные организованные наборы данных на любых носителях и устройствах, за сохранность и обновляемость которых (а также передачу в прикладные программы / из прикладных программ) несет ответственность ОС ЭВМ.

Связь программы с внешними файлами осуществляется через специальные переменные. Подробное описание установления связи программных переменных с файлами было впервые сделано в ЯП Cobol (Common Business Oriented Language). Эта проблема была решена следующим образом.

Цикл обработки файла (например, внесение изменений в списки информации о студентах) включает следующие операции:

- открытие файла — занятие устройства, на котором файл размещен (например, МД), создание в ОП управляющего блока, в котором записывается справка о состоянии файла, и буфера (или набора буферов — буферного пула) для хранения текущей, обрабатываемой записи файла;
- организация цикла, управляемого файлом (заканчивается по исчерпании записей файла — наступлении состояния EOF — end-of-file). Цикл должен содержать команду типа READ, GET (ввод записи) или PUT, WRITE (вывод записи), либо REWRITE (обновить запись);
- закрытие файла — выполнение операций по внесению всех окончательных изменений в файл и его реквизиты, освобождение памяти, отведенной под файл, и устройства, на котором он размещался.

Устройства, на которые осуществляется вывод данных из программы, или с которых осуществляется ввод (это может быть одно и то же физическое устройство, как это было в случае ранних терминалов; в современных, так называемых ANSI-терминалах — монитор и клавиатура рассматриваются как два отдельных, независимых устройства) — могут быть подразделены на следующие типы:

- передача информации битовым потоком;
- посимвольный обмен информацией;
- передача информации порциями (записями).

Фактически это как бы «портрет» устройства, каким его «видит» прикладная программа через посредство драйвера устройства и программ операционной системы, ответственных за ввод-вывод информации. Одно и то же устройство может быть представлено как генератор потока символов

(потокоориентированное устройство) или записей (записеориентированное). Поэтому скорее стоит говорить о типе файлов, расположенных на том или ином устройстве.

Могут быть рассмотрены следующие типы файлов:

по типу записей:

- файлы с записями фиксированной длины;
- файлы с записями переменной или неопределенной длины;
- файлы, образующие байтовый или битовый поток;

по способу выборки информации:

- файлы последовательного доступа;
- файлы прямого доступа.

Для всех этих типов файлов возникает проблема идентификации данных, размещенных на носителе (в файле). Каким образом можно правильно сопоставить тем или иным битовым комбинациям, размещенным в файле, те или иные области оперативной памяти, куда они Должны считываться с внешнего носителя для последующей обработки или обновления?

Метод сопоставления, по-видимому, должен выполнять следующие функции:

- «привязывание» данных, размещенных на внешнем носителе (в файле) к тем областям памяти, которые выделены для размещения этих данных (им соответствуют какие-то идентификаторы переменных в обрабатывающей программе);
- обнаружение и обработка ошибок при считывании данных (например, несоответствие типа или длины данного ожидаемому и т.п.).

Файлы последовательного доступа. Рассмотрим вначале файлы последовательного доступа. Для таких файлов характерны операции последовательного чтения и записи в конец файла. При считывании информации из файла (эта функция может быть возложена как на операционную систему, так и на пользовательскую программу или библиотечную процедуру) необходимо уметь:

- определять начало и окончание элементарного данного;
- определять начало и окончание записи файла.

Здесь необходимо отдельно рассмотреть записи фиксированной длины и записи переменной (неопределенной) длины.

В случае работы с записями фиксированной длины данные на носителе (в файле) должны иметь строго ту длину, которая задана в их описании (в прикладной программе).

Для таких записей в буфере считывания выделяется область, равная общей длине записи. Всякое нарушение длины и типа приводит к ошибке считывания.

В случае работы с записями переменной длины они должны быть отделены друг от друга разделителями (ограничителями) записей. Следует отметить, что такой подход действителен для записей, как переменной, так и фиксированной длины.

Возможен также и другой метод идентификации записи: например, первый байт каждой записи может содержать информацию о длине всей записи.

Записи неопределенной длины возникают тогда, когда ограничителем является физическая метка, распознаваемая устройством или файловой системой ОС. Примером файлов неопределенной длины являются текстовые файлы с признаком «возврат каретки — перевод строки» в конце каждой строки.

Файлы прямого доступа. Для файлов прямого доступа характерны операции чтения и записи по произвольному адресу. Такие файлы размещаются на дисках, как устройствах прямого доступа (в истории вычислительной техники, однако, существовали и магнитные ленты прямого доступа для отечественной ЭВМ МИНСК-22(М)).

Наипростейшим способом быстрого нахождения необходимой записи по ее номеру в файле является использование записей с фиксированной длиной. Тогда физический адрес легко вычисляется как некоторое смещение относительно начала файла, пропорциональное номеру выбираемой записи, например:

если в файле F размещено N записей длиной l байт, то:

- общая длина всего файла — Length(F) = N * l байт;
- смещение i-й записи от начала файла — Adr(i) = l•(i—1) байт.

Все вышеперечисленное относится не только к файлам, ориентированным на записи, но и к потоковым файлам. Во многих языках программирования существует понятие двоичного файла, структура которого не ограничена байтами, записями, разделителями и др. Такой файл может интерпретироваться как очень большая битовая строка, любая часть которой может быть передана в пользовательскую программу (и наоборот) путем указания смещения и длины такой «подстроки» в соответствующих функциях или командах обмена с внешними устройствами. Естественно, полную ответственность за обработку этих файлов несет пользовательская программа (а точнее, ее автор).

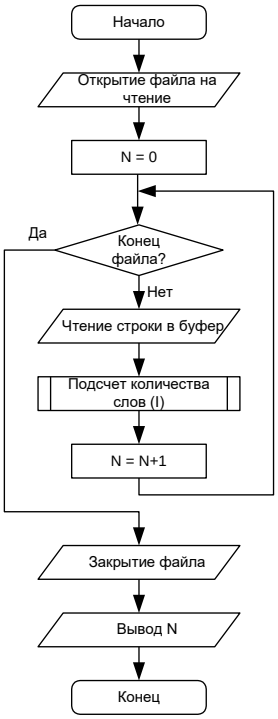
Понятия записей (фиксированной или переменной длины, метода записи и пр.) возникают тогда, когда часть подобной ответственности берет на себя операционная система ЭВМ и/или библиотечные программы соответствующей системы программирования (именно поэтому обычно различаются команды и функции доступа к файлам в различных ЯП).

Примеры и решения.

1. Пусть необходимо подсчитать количество слов в текстовом файле, разделенном на строки с помощью последовательности символов «перевод каретки — возврат строки» (без использования переноса слов). Для решения такой задачи следует:

- открыть файл на чтение;
- обеспечить циклическое чтение строк файла в программный буфер;
- к программному буферу применить подпрограмму подсчета количества слов в очередной строке, накапливая в цикле общее количество слов в файле;
- закрыть файл и вывести результат.

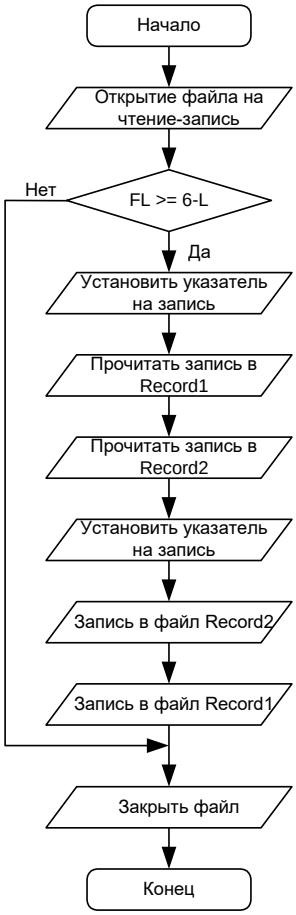
Ниже приведена блок-схема алгоритма (N — количество слов в файле, I — количество слов в очередной строке).



2. Пусть в файле прямого доступа, содержащем записи длиной L, необходимо поменять местами пятую и шестую записи. Порядок решения задачи следующий:

- открыть файл на чтение-запись;
- проверить, содержатся ли в файле записи с номером 5 и 6 (т.е. длина файла должна быть больше либо равна 6 * L);
- определить позицию пятой записи и установить на нее указатель;
- прочитать пятую запись и поместить ее в переменную Record1;
- прочитать следующую (шестую) запись и поместить ее в переменную Record2;
- установить указатель в файле на пятую запись;
- записать содержимое Record2;
- записать содержимое Record1;
- закрыть файл.

Приведем блок-схему алгоритма (FL- длина файла, L – длина записи).



Вопросы и задания.

1. Охарактеризуйте основные типы файлов.
2. Сформулируйте, каковы достоинства и недостатки файлов с фиксированной и переменной длиной записи.
3. Опишите физический состав файла, состоящего из записей фиксированной длины, содержащих «Информацию о студенте».
4. Разработайте алгоритм организации доступа к файлу, содержащему записи «Информация о студенте»:
   - добавление записи в конец файла;
   - поиск записи по фамилии;
   - редактирование найденной записи;
   - удаление найденной записи.

1.8. Объектно-ориентированный подход к программированию.

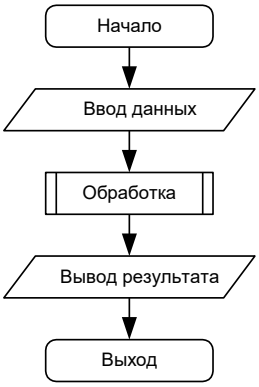
В истории применения компьютеров вычислительная техника все время используется на пределе своих возможностей. Каждое новое достижение в аппаратном либо в программном обеспечении приводит к попыткам расширить сферу применения ЭВМ, что влечет за собой постановку новых задач, для решения которых, в свою очередь, нужны новые вычислительные средства.

Основа для массового промышленного программирования была создана с разработкой методов построения программ. Одной из первых и наиболее широко применяемых технологий программирования стало структурное программирование. Этот метод до сих пор не потерял своего значения для определенного класса задач.

Структурный подход базируется на двух основополагающих принципах:

- использование процедурного стиля программирования;
- последовательная декомпозиция алгоритма решения задачи сверху вниз.

В соответствии с этим подходом задача решается путем выполнения следующей последовательности действий. Первоначально задача формулируется в терминах ввода данных — вывода результата: на вход программы подаются некоторые данные, программа работает и выдает ответ (см. рисунок).



После этого начинается последовательное разложение всей задачи на отдельные более простые действия. При этом на любом этапе декомпозиции программу можно проверить, применяя механизм так называемых «заглушек» — процедур, имитирующих вход и/или выход процедур нижнего уровня. «Заглушки» позволяют проверить логику верхнего уровня до реализации следующего, т.е. на каждом шаге разработки программы можно иметь работающий каркас, который постепенно обрастает деталями.

Структурное программирование ясно определило значение модульного построения программ (т.е. разбиения монолитных программ на группу отдельных модулей) при разработке больших проектов, но в языках программирования единственным способом структуризации программы оставалось составление ее из подпрограмм и функций.

Объектно-ориентированное программирование родилось и получило широкое распространение именно благодаря попыткам разрешения следующих проблем, возникавших в процессе проектирования и разработки программных комплексов.

1. Развитие языков и методов программирования не успевало за все более растущими потребностями в прикладных программах. Единственным реальным способом снизить временные затраты на разработку был метод многократного использования разработанного программного обеспечения, т.е. проектирование новой программной системы на базе разработанных и отлаженных ранее модулей, которые выступают в роли своеобразных «кирпичиков», ложащихся в Фундамент новой разработки.
2. Ускорение разработки программного обеспечения требовало решения проблемы упрощения их сопровождения и модификации.
3. Не все задачи поддаются алгоритмическому описанию по требованиям структурного программирования, поэтому в целях упрощения процесса проектирования необходимо было решить проблему приближения структуры программы к структуре решаемой задачи.

Решение перечисленных проблем в рамках создания объектно-ориентированного подхода к программированию и породило три его основных достоинства: упрощение процесса проектирования программных систем, легкость их сопровождения и модификации и минимизирование времени разработки за счет многократного использования готовых модулей.

Понятие объекта. В отличие от процедурного подхода к программированию, когда описание алгоритма представляет собой последовательность действий, объектно-ориентированный подход предлагает описывать программные системы в виде взаимодействия объектов.

Рассмотрим распространенный пример построения изображения «звездного неба» в двумерной проекции. Задача состоит в размещении и перемещении некоторого количества мерцающих разноцветных точек-звезд и кругов-планет на плоскости экрана.

Пусть звезда в нашем определении — это точка на экране с координатами X, Y , которая имеет цвет, может быть видимой или невидимой и может перемещаться по «звездному небу». Планета же — это цветной круг с координатами центра в точке X, Y, который тоже может быть видимым или невидимым и должен уметь перемещаться.

Все размещаемые звезды (и планеты) в нашем примере однотипны и могут представляться одинаковыми свойствами и процедурами обработки, т.е. для решения задачи необходимо создать так называемые объекты типа «Звезда» и «Планета». Таким образом, **объект** — это понятие, сочетающее в себе совокупность данных и действий над ними. **Свойства** — это характеристики состояния объекта, а действия над данными объекта называются **методами**.

Наследование. Очевидно, что основой изображения любого элемента является его положение (Позиция) на экране (например, значения координат X и Y относительно левого верхнего угла экрана). Опишем объект «Позиция» с координатами X и Y и методом «Назначить координаты»:

Позиция(X , У, НазначитьXУ)

Рассмотрим теперь объект «Звезда», который по нашему определению может быть описан следующим образом:

Звезда(X , Y, Видимость, Цвет, НазначитьXУ, Назначить цвет, Зажечь, Погасить, Переместить)

Обратим внимание на то, что возможности объекта Позиция полностью входят в объект Звезда. Такое свойство объектов называется **наследованием** и позволяет описать объект Звезда с учетом наследования возможностей объекта Позиция:

Звезда (Позиция, Видимость, Цвет, Назначить цвет, Зажечь, Погасить, Переместить)

Наследование позволяет повторно использовать уже созданную часть программного кода в других проектах. Посредством наследования формируются связи между объектами, а для выражения процесса наследования используют понятия «родители» и «потомки». В программировании наследование служит для сокращения избыточности кода, и суть его заключается в том, что уже существующий интерфейс вместе с его программной частью можно использовать для других объектов. При наследовании могут также проводиться изменения интерфейсов.

Инкапсуляция. В дальнейшем при работе с объектом Звезда доступны все возможности объекта Позиция, причем методы объекта Звезда могут использовать данные родительского объекта (Позиция). Объединение в одном месте всех данных и методов объекта (включая данные и методы объектов-предков) называется инкапсуляцией и облегчает понимание работы программы, а также и ее отладку и модификацию, так как только в очень редких случаях разработчика интересует внутренняя реализация объектов — главное, чтобы объект обеспечивал функции, которые он должен предоставить.

Взаимодействие с объектом происходит через интерфейс. Обычно интерфейс определяет единственный способ входа в объект и выхода из него; детали реализации остаются инкапсулированными. При создании собственных объектов необходимо организовать такие же интерфейсы. В объектах интерфейсами являются свойства, методы и события. Только они предоставляются данным объектом в распоряжение других объектов. Благодаря этому предотвращается доступ других объектов к внутренним переменным состояния объекта, которые могут обрабатываться только предусмотренными для этого процедурами. Таким образом, инкапсуляция обеспечивает использование объекта, не зная, как он реализован внутри.

Полиморфизм. Перейдем к описанию объекта Планета. Очевидно, что этот объект должен наследовать все возможности объекта Звезда, но методы Назначить цвет, Зажечь, Погасить, Переместить, по результату действия являясь теми же самыми, фактически не могут быть реализованы одинаковой последовательностью команд в случае со Звездой и Планетой. Эта ситуация разрешается следующим образом: методы объектов имеют одинаковое имя (и, соответственно, одинаковый ожидаемый результат), а внутреннее описание методов будет различно для разных объектов. Такое свойство объектов называется полиморфизмом, а объект Планета имеет следующее описание:

Планета (Звезда, Радиус, Назначить цвет, Зажечь, Погасить, Переместить)

Полиморфизм основывается на возможности включения в данные объекта также и информации о методах обработки этих данных. При этом различные объекты используют одинаковую абстракцию, т.е. могут обладать свойствами и методами с одинаковыми именами. Однако обращение к ним будет вызывать различную реакцию для различных объектов. Большое достоинство полиморфизма состоит в том, что при использовании объекта можно вызывать определенное свойство или метод, не заботясь о том, как объект выполняет задачу.

Классы и объекты. В предыдущем примере на самом деле описаны множества однотипных объектов (Позиция, Звезда, Планета), которые обладают одинаковыми возможностями, т. е. Позиция, Звезда и Планета представляют целые классы объектов.

Класс и объект — два общепринятых термина. Какова же разница между ними?

Термин **класс** объединяет объекты с одинаковыми возможностями (данными и методами). Он описывает общее поведение и характеристики набора аналогичных друг другу объектов. **Объект** — это экземпляр класса или, другими словами, переменная, тип которой задается классом. Объекты в отличие от классов реальны, т. е. существуют и хранятся в памяти во время выполнения программы. Соотношения между объектом и классом аналогичны соотношениям между переменной и типом.

Компоненты. Использование библиотек классов повышает скорость разработки программ, но, с другой стороны, требует определенных усилий для изучения этих библиотек и понимания того, как они устроены. Кроме того, библиотека классов должна быть написана на том же языке программирования, что и разрабатываемая программа. Конечно, существуют способы сопряжения разных языков программирования, но все равно, для того чтобы использовать, например, для программы, написанной на языке Pascal, библиотеку классов C++, необходимо написать программу с вызовами нужных функций или порождением необходимых классов.

Подобные неудобства привели к появлению концепции компонента — программного модуля или объекта, который готов для использования в качестве составного блока программы и которым можно визуально манипулировать во время разработки программы.

Компонент — это объект, объединяющий состояние и интерфейс (способ взаимодействия). Состояние компонента может быть изменено только с помощью изменения его свойств и вызова методов.

У компонента имеются два типа интерфейсов: интерфейс стадии проектирования и интерфейс стадии выполнения.Интерфейс проектирования позволяет включать компоненты в современные среды разработки приложений, а интерфейс выполнения управляет работой компонента во время выполнения программы.

При этом неважно, на каком языке программирования реализован компонент. Он должен просто удовлетворять определенным внешним параметрам и быть нейтрален по отношению к языку программирования, чтобы его можно было использовать в программе на любом языке, поддерживающем компонентную технологию. Так, например, компоненты стандарта ActiveX могут быть одинаково успешно включены в программу, реализованную в среде Visual Basic, и в приложение, разработанное средствами Delphi.

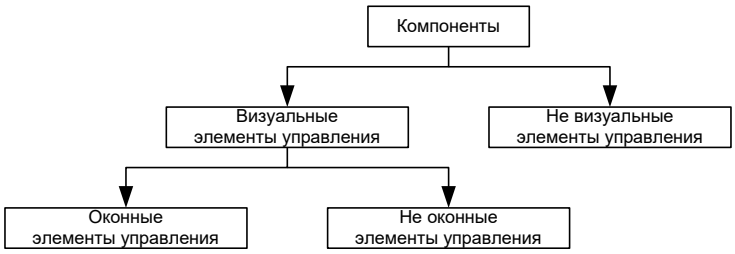
Компоненты графического интерфейса, управляемые событиями, являются основным «строительным» материалом при разработке приложений средствами графических редакторов. Разработка любого приложения состоит из двух взаимосвязанных этапов:

- проектирование и создание функционального интерфейса приложения (т.е. набора визуальных компонентов, которые будут обеспечивать взаимодействие пользователя и вычислительной среды);
- программирование процедур обработки событий, возникающих при работе пользователя с приложением.

На первом этапе (т.е. на этапе проектирования интерфейса — формирования общего вида главного окна при выполнении приложения и способов управления работой приложения) для каждого компонента необходимо определить его внешний вид, размеры, способ и место размещения в области окна приложения (т.е. реализовать интерфейс разработки и интерфейс выполнения).

Компоненты, доступные проектировщику на этапе разработки приложения, разбиты на функциональные подгруппы.

С точки зрения внутренней структуры компоненты разбиваются на три группы. На рисунке представлена графическая интерпретация этого разбиения.



Визуальные компоненты (элементы управления) характеризуются наличием свойств размеров и положения в области окна и на стадии разработки приложения обычно находятся на форме в том же месте, что и во время выполнения приложения (например, кнопки, списки, переключатели, надписи). Визуальные компоненты имеют две разновидности — «оконные» и «неоконные» (графические).

- «Оконные» визуальные компоненты (самая многочисленная группа компонентов) — это компоненты, которые могут получать фокус ввода (т.е. становиться активными для взаимодействия с пользователем) и содержать другие визуальные компоненты.
- «Неоконные» (графические) визуальные компоненты не могут получать фокус и содержать другие визуальные компоненты (например, надписи и графические кнопки).

Невизуальные компоненты на стадии разработки не имеют своего фиксированного местоположения и размеров. Во время выполнения приложения некоторые из них иногда становятся видимыми (например, стандартные диалоговые окна открытия и сохранения файлов), а другие остаются невидимыми всегда (например, таблицы базы данных).

Важной характеристикой компонента являются его свойства. **Свойства компонента** — это атрибуты, определяющие его состояние и поведение. Различают три типа свойств.

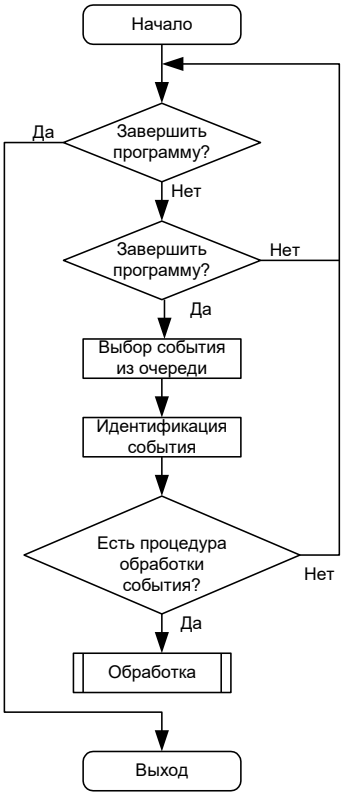
Первые — свойства времени проектирования. Установленные для них значения будут использоваться в момент первого отображения компонента и в дальнейшем могут быть изменены во время выполнения приложения.

Вторые — динамические свойства. Изменением их значений можно управлять только изнутри программного кода (во время выполнения приложения).

Третьи — так называемые свойства только-для-чтения, которые могут быть прочитаны и использованы при выполнении программы, но не могут быть изменены.

Второй этап — непосредственное программирование процедур обработки событий, исходящих от компонентов. Основная задача при разработке таких процедур — запрограммировать реакцию на все возможные изменения состояний объектов.

Общий принцип работы приложения с графическим интерфейсом может быть представлен схемой, изображенной на рисунке.



Работающее приложение находится в состоянии ожидания события, которое возникает в результате взаимодействия пользователя с элементами управления графического интерфейса приложения. При получении события от компонента программа передает управление процедуре обработки этого события (если таковая предусмотрена набором функций приложения).

Вопросы и задания.

1. Спроектируйте объект на базе записи «Информация о студенте». Каковы могут быть свойства и методы подобного объекта?
2. Охарактеризуйте основные преимущества объектно-ориентированного подхода при разработке программ.
3. Приведите примеры объектов, иллюстрирующие свойство наследования.
4. Приведите примеры объектов, иллюстрирующие свойство полиморфизма.
5. Охарактеризуйте понятия «класс» и «объект».
6. Охарактеризуйте основные различия между группами компонентов.
7. В чем, по-вашему, заключаются основные различия в методах «оконных» и «неоконных» компонентов?

1.9. Разработка программного обеспечения (ПО).

Общие принципы разработки ПО. Программное обеспечение различается по назначению, выполняемым функциям, формам реализации. В этом смысле ПО — сложная, достаточно уникальная система. Однако существуют некоторые общие принципы, которые следует использовать при разработке ПО.

Частотный принцип. Основан на выделении в алгоритмах и в обрабатываемых структурах действий и данных по частоте использования. Для действий, которые часто встречаются при работе ПО, обеспечиваются условия их быстрого выполнения. К данным, которым происходит частое обращение, обеспечивается наиболее быстрый доступ, а подобные операции стараются сделать более короткими.

Принцип модульности. Под модулем в общем случае понимают функциональный элемент рассматриваемой системы, имеющий оформление, законченное и выполненное в пределах требований системы, и средства сопряжения с подобными элементами или элементами более высокого уровня данной или другой системы.

Способы обособления составных частей ПО в отдельные модули могут быть существенно различными. Чаще всего разделение происходит по функциональному признаку. В значительной степени разделение системы на модули определяется используемым методом проектирования ПО.

Принцип функциональной избирательности. Этот принцип является логическим продолжением частотного и модульного принципов и, используется при проектировании ПО, объем которого существенно превосходит имеющийся объем оперативной памяти. В ПО выделяется некоторая часть важных модулей, которые постоянно должны быть в состоянии готовности для эффективной организации вычислительного процесса. Эту часть в ПО называют ядром или монитором. При формировании состава монитора требуется удовлетворить два противоречивые требования. В состав монитора, помимо чисто управляющих модулей, должны войти наиболее часто используемые модули. Количество модулей должно быть таким, чтобы объем памяти, занимаемой монитором, был не слишком большим. Программы, входящие в состав

монитора, постоянно находятся в оперативной памяти. Остальные части ПО постоянно хранятся во внешних запоминающих устройствах и загружаются в оперативную память только при необходимости, иногда перекрывая друг друга.

Принцип генерируемости. Данный принцип определяет такой способ исходного представления ПО, который бы позволял осуществлять настройку на конкретную конфигурацию технических средств, круг решаемых проблем, условия работы пользователя.

Принцип функциональной избыточности. Этот принцип учитывает возможность проведения одной и той же работы (функции) различными средствами. Особенно важен учет этого принципа при разработке пользовательского интерфейса для выдачи данных из-за психологических различий в восприятии информации.

Принцип «по умолчанию». Применяется для облегчения организации связей с системой, как на стадии генерации, так и при работе с уже готовым ПО. Принцип основан на хранении в системе некоторых базовых описаний структур, модулей, конфигураций оборудования и данных, определяющих условия работы с ПО. Эту информацию ПО использует в качестве заданной, если пользователь забудет или сознательно не конкретизирует ее. В данном случае ПО само установит соответствующие значения.

Общесистемные принципы. При создании и развитии ПО рекомендуется применять следующие общесистемные принципы:

- принцип включения, который предусматривает, что требования к созданию, функционированию и развитию ПО определяются со стороны более сложной, включающей его в себя системы;
- принцип системного единства, который состоит в том, что на всех стадиях создания, функционирования и развития ПО его целостность будет обеспечиваться связями между подсистемами, а также функционированием подсистемы управления;
- принцип развития, который предусматривает в ПО возможность его наращивания и совершенствования компонентов и связей между ними;
- принцип комплексности, который заключается в том, что ПО обеспечивает связность обработки информации, как отдельных элементов, так и для всего объема данных в целом на всех стадиях обработки;
- принцип информационного единства, определяющий, что во всех подсистемах, средствах обеспечения и компонентах ПО используются единые термины, символы, условные обозначения и способы представления;
- принцип совместимости, который состоит в том, что язык, символы, коды и средства обеспечения ПО согласованы, обеспечивают совместное функционирование всех его подсистем и сохраняют открытой структуру системы в целом;
- принцип инвариантности, который предопределяет, что подсистемы и компоненты ПО инвариантны к обрабатываемой информации, т.е. являются универсальными или типовыми.

Жизненный цикл программного обеспечения. Рассмотрим жизненный цикл программного обеспечения, т.е. процесс его создания и применения от начала до конца. Этот процесс состоит из нескольких стадий: определения требований и спецификаций, проектирования, программирования, отладки и сопровождения.

Первая стадия — определение требований и спецификаций, может быть названа стадией системного анализа. На ней устанавливаются общие требования к ПО: по надежности, технологичности, правильности, универсальности, эффективности, информационной согласованности и т.д. Они дополняются требованиями заказчика, включающими пространственно-временные ограничения, необходимые функции и возможности, режимы функционирования, требования точности и надежности и т.д., т.е. вырабатывается описание системы с точки зрения пользователя. При определении спецификаций производится точное описание функций ПО, разрабатываются и утверждаются входные и промежуточные языки, форма выходной информации для каждой из подсистем, описывается возможное взаимодействие с другими программными комплексами, специфицируются средства расширения и модификации ПО, разрабатываются интерфейсы обслуживающих и основных подсистем, решаются вопросы базы данных, утверждаются основные алгоритмы. Итогом выполнения этого этапа являются эксплуатационные и функциональные спецификации, содержащие конкретное описание ПО. Разработка спецификаций требует тщательной работы системных аналитиков, постоянно контактирующих с заказчиками, которые в большинстве случаев не способны сформулировать четкие и реальные требования.

Эксплуатационные спецификации содержат сведения о быстродействии ПО, затратах памяти, требуемых технических средствах, надежности и т.д.

Функциональные спецификации определяют функции, которые должно выполнять ПО, т.е. в них определяется, что надо делать системе, а не то, как это делать.

Спецификации должны быть полными, точными и ясными. Полнота исключает необходимость получения разработчиками ПО в процессе их работы от заказчиков иных сведений, кроме содержащихся в спецификациях. Точность не позволяет различных толкований. Ясность подразумевает легкость понимания, как заказчиком, так и разработчиком при однозначном толковании.

Значение спецификаций.

1. Спецификации являются заданием на разработку ПО и их выполнение — закон для разработчика.
2. Спецификации используются для проверки готовности ПО.
3. Спецификации являются неотъемлемой частью программной документации, облегчают сопровождение

и модификацию ПО.

Второй стадией является проектирование ПО. На этом этапе:

1. Формируется структура ПО и разрабатываются алгоритмы, задаваемые спецификациями.
2. Устанавливается состав модулей с разделением их на иерархические уровни на основе изучения схем алгоритмов.
3. Выбирается структура информационных массивов.
4. Фиксируются межмодульные интерфейсы.

Цель этапа — иерархическое разбиение сложных задач создания ПО на подзадачи меньшей сложности. Результатом работы на этом этапе являются спецификации на отдельные модули, дальнейшая декомпозиция которых нецелесообразна.

Третья стадия — программирование. На данном этапе производится программирование модулей. Этап менее сложен по сравнению со всеми остальными. Проектные решения, полученные на предыдущей стадии, реализуются в виде программ. Разрабатываются отдельные блоки и подключаются к создаваемой системе. Одна из задач — обоснованный выбор языков программирования. На этой же стадии решаются все вопросы, связанные с особенностями типа ЭВМ.

Четвертая стадия — отладка ПО заключается в проверке выполнения всех требований, всех структурных элементов системы на таком количестве всевозможных комбинаций данных, какое только позволяют здравый смысл и бюджет. Этап предполагает выявление в программах ошибок, проверку работоспособности ПО, а также его соответствие спецификациям.

Пятая стадия — сопровождение, т.е. процесс исправления ошибок, координации всех элементов системы в соответствии с требованиями пользователя, внесения всех необходимых ему исправлений и изменений. Он вызван, как минимум, двумя причинами: во-первых, в ПО остаются ошибки, не выявленные при отладке; во-вторых, пользователи в ходе эксплуатации ПО настаивают на внесении в него изменений и его дальнейшем совершенствовании.

В общем случае, в процессе разработки ПО требуется осуществить несколько итераций (последовательных приближений) к приемлемому варианту системы.

Распределение временных и стоимостных затрат по отдельным стадиям жизненного цикла программного обеспечения представлено в таблицах.

Таблица. Затраты по стадиям жизненного цикла ПО (Сведения из разных источников)

Стадии жизненного цикла программного обеспечения	Гласе Р. Нуазо Р.	Энкар-наччо Ж.	Боэм Б.	Федорчук Чернень-кий	Виш-няков	Среднее
Определение требований и спецификаций	10	15	6	6	6	8,6
Проектирование	10	12	18	5	6	10,2
Программирование	10	12	9	7	6	8,8
Отладка	20	15	17	15	14	16,2
Сопровождение	50	46	50	67	68	56,2
Всего	100	100	100	100	100	100,0

Таблица. Основные параметры стадий жизненного цикла ПО

Стадии жизненного цикла программного обеспечения	Количество ошибок, %	Обнаружение ошибок, %	Затраты на устранение ошибок, %
Определение требований и спецификаций			9
Проектирование	61-64		6
Программирование	36-39		10
Отладка		46	12
Сопровождение		54	63

Исходя из этого, возникает ряд требований к инженерному программированию. Они состоят в необходимости разработки и сопровождения ПО, которое гарантирует, что ВС являются:

- исключительно надежными;
- удобными в использовании;
- естественными;
- проверяемыми;
- труднодоступными для злоупотреблений;
- оставляющими главную роль за человеком, а не за машиной.

Попытка сформулировать общую цель инженерного программирования, достижение которой привело бы к удовлетворению всех указанных требований, обречена на неудачу, так как некоторые требования противоречат друг другу.

Как видно, область инженерного программирования, к счастью (или к несчастью), не столь проста.

Существует большое количество предлагаемых в качестве рекомендаций «стандартов» программирования: «разрабатывайте сверху-вниз», «доказывайте правильность программ», «разрабатывайте дважды», «используйте метод главного программиста», «программируйте структурно», «четко определяйте этапы», «привлекайте к работе пользователя» и т.д. Каждый из этих советов способствует выполнению определенных требований, но никак не учитывает другие и фактически препятствует их удовлетворению.

Вопросы.

1. Охарактеризуйте общие принципы разработки ПО.
2. В чем заключаются общесистемные принципы разработки?
3. Охарактеризуйте стадии жизненного цикла ПО.
4. Какие стадии жизненного цикла требуют наибольших затрат и почему?
5. На каких стадиях жизненного цикла затраты на устранение ошибок максимальны и почему?

Основной текст.

- Список

1. Первый элемент списка.
2. Второй элемент списка

Рис. 1

p p p p p p p p p p p p p p p p p p