Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

О. А. Кравченко, Л. К. Титова

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Электронный аналог печатного издания

Гомель 2015

ВВЕДЕНИЕ

Языки программирования развивались одновременно с развитием ЭВМ. С начала 50-х гг. это были низкоуровневые языки (машинные и ассемблеры). В 1956 г. появился язык Фортран, а в 1960 – Алгол-60. Это языки компилирующего типа, существенно уменьшившие трудоемкость программирования и ориентированные на выполнение математических вычислений. В дальнейшем возникло большое количество различных языков, претендовавших на универсальность (PL/1) или для решения конкретных задач (COBOL – для деловых задач, ЛОГО – для обучения, Пролог – для разработки систем искусственного интеллекта). С середины 60-х до начала 80-х гг. разработаны и получили распространение языки Pascal, Basic, Си, Ада и др.

Принципиально новым этапом в развитии языков программирования стало появление методологии непроцедурного (объектно-ориентированного) программирования. Основные достоинства объектно-ориентированного программирования – быстрота разработки интерфейса программного приложения, возможность наследования свойств программных объектов.

По способу разработки программ можно выделить два подхода:

1) *процедурное программирование* – это программирование, при котором выполнение команд программы определяется их последовательностью, командами перехода, цикла или обращениями к процедурам;

2) *объектно-ориентированное программирование* – программирование, при котором формируются программные объекты, имеющие набор свойств, обладающие набором методов и способные реагировать на события, возникающие как во внешней среде, так и в самом объекте (нажатие мыши, срабатывание таймера, превышение числовой границы и т. д.). Таким образом, выполнение той или иной части программы зависит от событий в программной системе. Объектно-ориентированное программирование не исключает, а охватывает технологию процедурного программирования.

Из универсальных языков программирования наиболее популярны следующие: Basic; Pascal; *C*++; Java. Для разработки программного обеспечения (ПО) на этих и других языках получила широкое распространения методология быстрой разработки приложений RAD (Rapid Application Devolopment).

Для языка Basic существует много версий, реализованных и как интерпретаторы, и как компиляторы. Среда визуального программирования Microsoft Visual Basic используется как программная поддержка приложений MS Office.

Язык Pascal является компилируемым и широко используется как среда для обучения программированию. RAD-средой, наследующей его основные свойства, является среда Borland Delphi.

Для языка *C*++ RAD-средой является Borland *C*++ Builder. Этот компилируемый язык часто используется для разработки программных приложений, в которых необходимо обеспечить быстродействие и экономичность программы.

Язык Java – интерпретируемый язык – позволяет создавать платформно-независимые программные модули, способные работать в компьютерных сетях с различными операционными системами. RAD-средой для него является Symantec Cafe.

Цель курса «Основы алгоритмизации и программирования» для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» состоит в изучении методов алгоритмизации, основ программирования на алгоритмических языках высокого уровня и в использовании полученных навыков при решении инженерных задач.

Данный курс предназначен для приобретения основных теоретических знаний по основам алгоритмизации и программирования и практических навыков подготовки, отладки и решения задач на современных ЭВМ.

Дисциплина «Основы алгоритмизации и программирования» позволяет выровнять уровень подготовки студентов по программированию и научить «свободно владеть» языком программирования как «средством выражения» алгоритмов применительно к традиционному кругу задач – арифметико-логическим, сортировки и поиска, приближенных вычислений, обработки текста.

Постепенный переход к детальному изучению структур данных в памяти и в файлах и алгоритмов работы с ними, к разработке принципиально более сложных программ, алгоритмов и структур данных позволяет сформировать уровень знания языка Си, позволяющий свободно оперировать типами данных и переменных произвольной сложности и модульными алгоритмами их обработки.

Задачи курса:

– формирование базовых знаний по алгоритмизации и программированию – о стиле написания программ, о рациональных методах их разработки и оптимизации, о стратегии отладки и тестирования программ;

– получение базового уровня по программированию на языке Си с использованием простых типов данных: базовых типов данных и массивов;

– изучение структур данных в памяти и в файлах и алгоритмов работы с ними с использованием языка Си;

– знакомство с основными принципами организации хранения и поиска данных, алгоритмами сортировки и поиска;

– приобретение навыков использования базового набора фрагментов и алгоритмов в процессе разработки программ, навыков анализа и «чтения» программ;

– изучение основ технологии программирования и методов решения вычислительных задач и задач обработки символьных данных;

– формирование уровня знания языка, позволяющего свободно оперировать типами данных и переменными произвольной сложности и модульными алгоритмами их обработки.

В результате выполнения учебной программы курса студент должен

• *иметь представление*:

– о структуре программного обеспечения, о видах и областях применения основных программных средств;

– об основных этапах решения задач на ЭВМ, о жизненном цикле программы, о критериях качества программы, о понятии диалоговых и дружественных пользователю программ;

– об общих принципах построения, описания, способах записи и основах доказательства правильности алгоритмов (верификации алгоритмов);

– об общей классификации современных языков программирования, их областях применения и использования;

– о стандартных типах данных и о типах данных, определяемых пользователем;

– о представлении основных структур программирования: итерация, ветвление, повторение, процедуры и функции; о программировании рекурсивных алгоритмов;

– о файлах и об основных видах динамических структур данных и способах их реализации;

– о способах конструирования программ и о понятии модульных программ;

– об использовании дополнительных пакетов и библиотек при программировании;

• *знать*:

– современные методы и средства разработки алгоритмов и программ на языке Си;

– синтаксис и семантику основных конструкций языка Си;

– способы организации сложных структур данных (массивы, структуры), основные методы представления и алгоритмы обработки этих данных;

– особенности работы с файлами в языке Си;

– особенности технологии разработки программ сложной структуры на языке Си;

• *уметь*:

– принимать участие во всех фазах проектирования, разработки, изготовления, отладки и документирования программ;

– разрабатывать модульные программы на Си с использованием базовых типов данных и массивов и сложных иерархических типов данных и переменных;

– разрабатывать собственные программы с использованием стандартных фрагментов алгоритмов;

– использовать указатели, структурированные переменные в разрабатываемых программах;

– применять динамическую память при обработке данных заранее неизвестного объема и размерности;

• *получить навыки*:

– разработки программ, использующих данные в произвольном формате, а также использующих массивы указателей и списки для хранения, упорядочения и поиска данных;

– проектирования программ, использующих двоичные файлы для размещения различных структур данных с полной и поэлементной их загрузкой в память;

– работы в интегрированных средах программирования и с использованием библиотек.

В качестве рабочей среды взята среда программирования Dev-*C*++. Dev-*C*++ – это интегрированная среда для программирования на языках *C* и *C*++, работающая под управлением операционной системы Windows. Среда Dev-*C*++ распространяется свободно с исходными кодами (на Delphi) по лицензии GPL.

Достоинства оболочки Dev-*C*++:

– графический интерфейс;

– русскоязычное меню;

– встроенный отладчик GDB;

– возможность создавать консольные и графические программы.

Ключевые особенности и функции программы:

– простой и практичный интерфейс;

– наличие инструментов для полноценного написания кода на *C*/*C*++;

– наличие встроенного отладчика;

– возможность написания, как консольных приложений, так и проектов использующих Windows API;

– возможность подключения дополнительных библиотек;

– бесплатное распространение программы.

1. ОСНОВЫ АЛГОРИТМИЗАЦИИ

1.1. Понятие об алгоритме

Решение любой задачи осуществляется по определенному плану, называемому алгоритмом.

Алгоритм – понятное и точное предписание исполнителю совершить последовательность действий, направленных на достижение указанной цели или на решение поставленной задачи.

В своей жизни мы постоянно имеем дело с различными правилами – разрешаюшими, запрещающими, предписывающими.

Примерами разрешающих правил могут быть следующие: «Каждый студент может получать повышенную стипендию», «Стоянка разрешена», «На дискотеку вход свободный». Никакое разрешающее правило не является алгоритмом. Это же утверждение справедливо и для запрещающих правил, примерами которых являются: «При плохой успеваемости студент не может получать повышенную стипендию», «Во всех корпусах университета курение запрещено», «Стоянка запрещена».

К алгоритмам относятся только предписывающие правила. Например, кулинарный рецепт приготовления какого-либо блюда является алгоритмом. Алгоритмом является также инструкция по включению какого-либо прибора, например компьютера, составленная из предписывающих правил.

Но не всякое предписывающее правило можно считать алгоритмом. Так, например, предписывающее правило «Взвейтесь кострами, синие ночи!» не является алгоритмом, так как предполагаемый исполнитель «синие ночи» желаемого действия («взвейтесь кострами») не в состоянии выполнить.

Таким образом, *алгоритм* – последовательность предписывающих правил, понятно и точно указывающих исполнителю, какую последовательность действий он должен исполнить по переработке исходных данных в искомые результаты.

1.2. Свойства алгоритма

Рассмотрим основные свойства, которыми должен обладать создаваемый алгоритм.

Точность определяется как свойство, согласно которому исполнителю точно известно, какая команда должна выполняться следующей.

Для иллюстрации важности этого свойства рассмотрим предписывающее правило: «Уходя, гасите свет». Является ли это правило алгоритмом? С одной стороны, рассматриваемое правило предписывает вполне определенному исполнителю (человеку, уходящему из данного помещения) выполнить в определенный момент (в момент ухода) вполне определенное действие (выключить свет). Исходя из этого, можно считать рассматриваемое правило алгоритмом. Но, с другой стороны, для человека, «страдающего» излишним формализмом, указанное правило не вполне точно определяет последовательность необходимых действий, так как для этого человека остается неясным, должен ли он, покидая помещение, выключить свет, если в помещении остаются люди. Анализируемый пример показывает на существование серьезной проблемы – то, что однозначно (точно) понимается одним исполнителем, может совсем не так восприниматься другим.

Понятность – это свойство состоит в том, что каждая команда алгоритма должна входить в систему команд исполнителя. Система команд исполнителя – это совокупность команд, которые могут быть выполнены исполнителем. Например, система команд микрокалькулятора описывается в его паспорте, в нее входят такие команды, как четыре арифметические операции, запись числа в ячейку памяти, вычисление элементарных функций и т. д.

Дискретность алгоритма заключается в том, что предписание представляет собой последовательность четко выраженных отдельных команд. Выполнив одну команду, исполнитель выполняет следующую команду и т. д. Бессмысленно ставить вопрос – что делает исполнитель между выполнением двух последовательных команд – таких моментов просто нет.

Массовость – для каждого алгоритма существует класс объектов, допустимых в качестве исходных данных. Так, например, алгоритм нахождения наибольшего общего делителя двух целых чисел применим для любой пары целых чисел. Из двух алгоритмов, разработанных для решения одной задачи, более ценным является тот, у

которого шире класс допустимых исходных данных. Можно сказать, что у такого алгоритма массовость больше.

Результативность – это свойство состоит в том, что результат выполнения алгоритма исполнитель должен получить, выполнив конечное число действий. Если для каких-то допустимых исходных данных исполнитель не может получить результат выполнения алгоритма за конечное число шагов, то говорят, что алгоритм не применим для этих исходных данных. Так, например, алгоритм получения точного значения частного при делении уголком не применим для чисел 20 и 3. При этом класс допустимых исходных данных для указанного алгоритма – пара целых чисел, второе из которых не равно 0.

1.3. Средства записи алгоритма

Для записи алгоритмов на различных этапах формализации и постановки задачи используются различные способы:

– словесно-формульная;

– графическая схема алгоритма (блок-схема);

– запись на алгоритмическом языке;

– в виде программы на языке программирования.

Рассмотрим причины указанного разнообразия и области употребления каждого способа записи.

Алгоритм разрабатывается и записывается для какого-то конкретного исполнителя. Можно выделить два класса исполнителей – люди и вычислительные устройства. К последним относятся роботы, микрокалькуляторы, компьютеры и т. д. Так как разные исполнители имеют различные системы команд, то в записях алгоритмов должны использоваться команды, зависящие от конкретного исполнителя (свойство понятности). Кроме наличия различных систем команд, каждый исполнитель отличается своим способом восприятия команд алгоритма – человек может прочитать или выслушать данные ему для исполнения команды, микрокалькулятор воспринимает команды, вводимые в него нажатием клавиши, в компьютеры команды можно вводить нажатием клавиш на клавиатуре или с помощью указателя мыши.

Итак, причина разнообразия средства записи алгоритмов – различия в системах команд исполнителей и в способах восприятия исполнителем команд алгоритма.

Язык программирования – это способ записи алгоритма, ориентированный на исполнение его системой программирования компьютера. Дня записи алгоритмов, предназначенных для безмашинного исполнения, можно использовать естественный язык (например,

русский), дополненный математическими символами – это так называемая словесная запись алгоритма.

Наиболее часто при разработке и документировании программы лежащие в ее основе алгоритмы изображаются в виде графических схем алгоритма или блок-схем.

Алгоритм большой сложности обычно представляется с помощью схем двух видов:

– обобщенной схемы алгоритма – раскрывает общий принцип функционирования алгоритма и основные логические связи между отдельными модулями на уровне обработки информации (ввод и редактирование данных, вычисления, печать результатов и т. п.);

– детальной схемы алгоритма – представляет содержание каждого элемента обобщенной схемы с использованием управляющих структур в блок-схемах алгоритма, псевдокода либо алгоритмических языков высокого уровня.

1.4. Графические схемы алгоритмов

Перед тем как записать алгоритм в виде программы, его, как правило, представляют в виде схемы алгоритма. **А не наоборот, как пытаются сделать многие начинающие!** Схема алгоритма, если она правильно составлена, способствует правильному и более быстрому написанию программы!

Схема алгоритма – это графическое представление метода решения задачи, в котором используются символы для отображения операций, данных, потока, оборудования и т. д. Схема алгоритма состоит из:

1) символов процесса, указывающих фактические операции обработки данных (включая символы, определяющие путь, которого следует придерживаться с учетом логических условий);

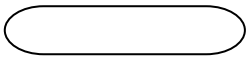
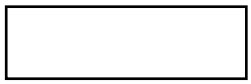
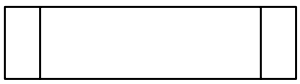
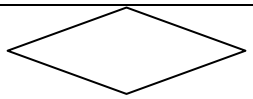
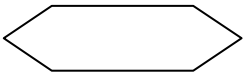
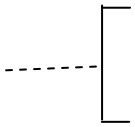
2) линейных символов, указывающих поток управления;

3) специальных символов, используемых для облегчения написания и чтения схемы.

Обозначение наиболее часто употребляемых символов и описание отображаемых ими действий приведено в табл. 1.1.

Таблица 1.1

Символы схемы алгоритма

Название символа	Обозначение	Значение
Терминатор		Начало или конец схемы алгоритма
Данные		Ввод или вывод данных; носитель данных не определен
Процесс		Обработка данных любого вида, приводящая к изменению значения, формы или размещения информации
Предопределенный процесс		Использование подпрограммы (или модуля)
Решение		Проверка условия и выбор одного из нескольких альтернативных выходов
Подготовка		Модификация команды, группы команд, индексного регистра (создание цикла)
Линия		Отображает поток данных и управления. При необходимости могут быть добавлены стрелки – указатели
Соединитель		Обрыв линии и продолжение ее в другом месте. Соответствующие символы должны иметь одно и то же уникальное обозначение
Комментарий		Пояснения к выполняемым действиям. Располагается около ограничивающей фигуры (символа или блока символов, обведенных пунктирной линией)

Символы могут быть вычерчены в любой ориентации, но по возможности предпочтительной является горизонтальная ориентация.

Текст, необходимый для понимания функции данного символа, следует помещать внутри данного символа и записывать слева направо и сверху вниз независимо от направления потока. Если объем текста превышает размеры символа, нужно использовать символ ком-

ментария. В схемах может использоваться идентификатор символов (например, в виде номера), которым можно воспользоваться в справочных целях в других элементах документации (или для ссылки на символ). Идентификатор символа должен располагаться слева над символом.

Правила выполнения соединений. Потоки данных или потоки управления в схемах показываются линиями. Направление потока слева направо и сверху вниз считается стандартным. Если необходимо внести большую ясность в схему (например, при соединениях), на линиях используются стрелки. Если направление потока отличается от стандартного, то стрелки должны указывать это направление.

В схемах следует избегать пересечения линий. Пересекающиеся линии не имеют логической связи между собой, поэтому изменения направления потока в точках пересечения не допускаются. Две или более входящие линии могут объединяться в одну исходящую.

Линии в схемах должны подходить к символу либо слева, либо сверху, а исходить либо справа, либо снизу. Линии должны быть направлены к центру символа. При необходимости линии в схемах нужно разрывать во избежание излишних пересечений или слишком длинных линий, а также если схема состоит из нескольких страниц. Соединитель в начале разрыва называется внешним соединителем, а соединитель в конце разрыва – внутренним соединителем. Совместно с символом комментария можно указать, с какой страницы или на какую страницу схемы совершается переход.

Примеры соединителей приведены на рис. 1.1.

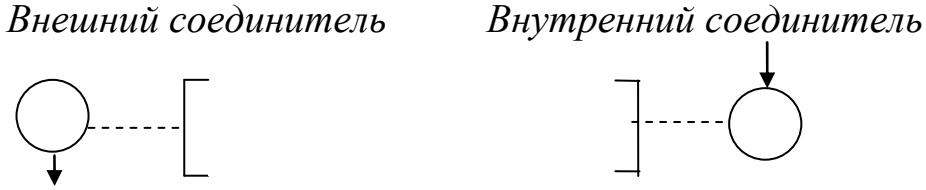


Рис. 1.1. Виды соединителей

Несколько выходов из символа можно показывать:

– несколькими линиями от данного символа к другим символам;

– одной линией от данного символа, которая затем разветвляется в соответствующее число линий.

Примеры изображения выходов из символа приведены на рис. 1.2.

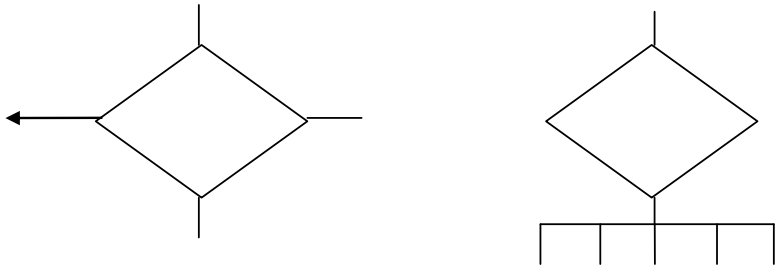


Рис. 1.2. Изображение выходов из символа

Каждый выход из символа нужно пометить значениями соответствующих условий, чтобы показать логический путь, который он представляет, с тем, чтобы эти условия и соответствующие ссылки были идентифицированы.

Примеры идентификации ссылок приведены на рис. 1.3.

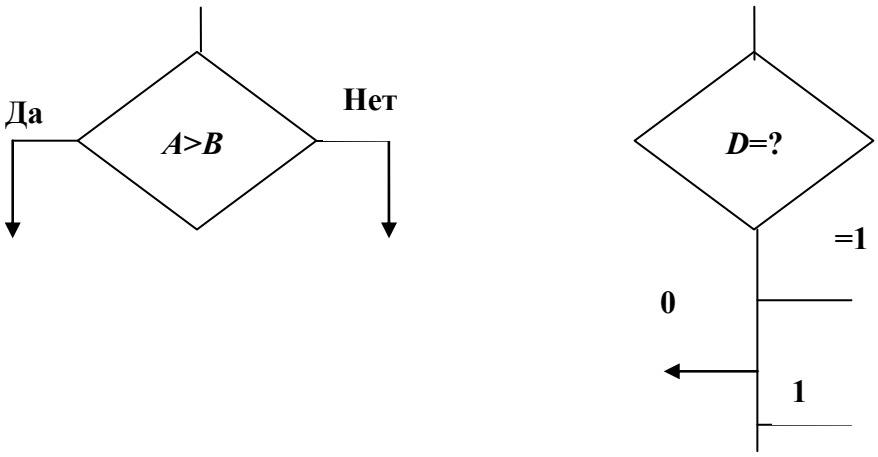
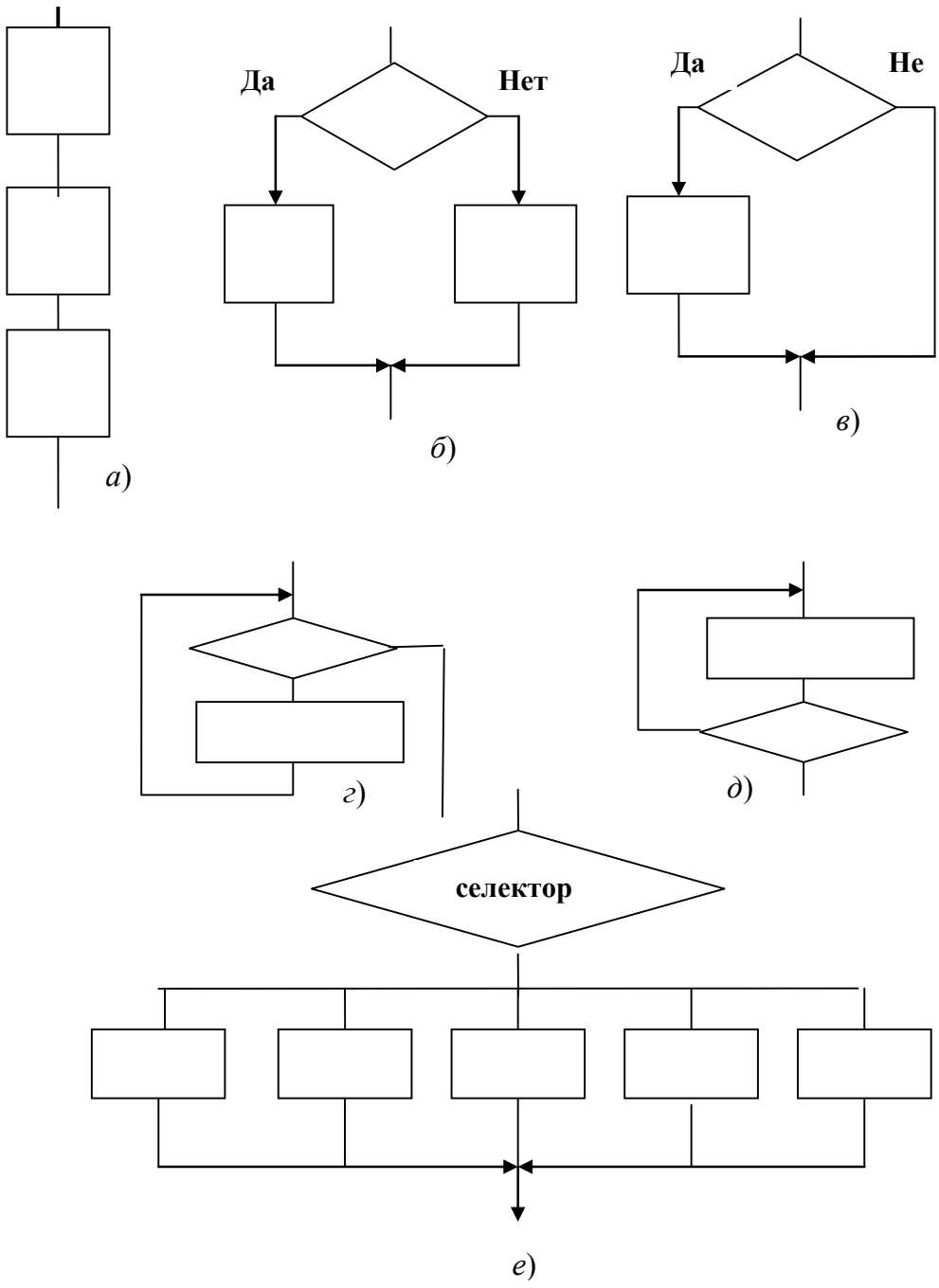


Рис. 1.3. Примеры идентификации ссылок

При всем разнообразии структур алгоритмов можно выделить четыре типовых структуры (рис. 1.4), из которых как из кирпичиков можно построить здание алгоритма любой сложности: *следование*, *ветвление*, *цикл* и *выбор*.

Типовые структуры хороши тем, что все они имеют одну точку входа и одну точку выхода. Это упрощает просмотр и понимание алгоритма и сокращает количество возможных ошибок, сделанных при составлении схемы и время на их поиск. Типовые структуры могут быть вложенными друг в друга, например, символ, входящий в структуру *следование*, может включать в себя структуры *ветвление* или *цикл*, а каждая из ветвей структур *ветвление* или *выбор* может содержать структуры *следование*, *цикл* или еще одно *ветвление* и *выбор*.



а) *б*) *в*) *г*) *д*) *е*)

Рис. 1.4. Типовые структуры алгоритмов:
а – следование; *б*, *в* – ветвление (полное и неполное);
г – цикл с предусловием; *д* – цикл с постусловием; *е* – выбор

1.5. Типы алгоритмов

На основе перечисленных в подразделе 1.4 структур строятся следующие типы алгоритмов:

– *линейный* (на основе структуры следование); характеризуется тем, что все действия, определяемые символами, входящими в схему, выполняются последовательно, в порядке их написания;

– *разветвляющийся* (на основе структур ветвление и выбор): характеризуется тем, что в ходе выполнения решение задачи идет только по одному из имеющихся направлений, выбор которого зависит от выполнения заданного условия;

– *циклический* (на основе структуры цикл): характеризуется многократным повторением определенной группы действий.

В последующих разделах на конкретных примерах рассматриваются проекты в системе программирования *C*, основанные на всех вышеперечисленных типах алгоритмов.

2. ОСНОВНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ

2.1. Структура программы

Программа на языке *C* состоит из директив препроцессора, объявлений глобальных переменных, комментариев, одной главной функции (main) и, возможно, ряда неглавных функций.

Следующий короткий пример иллюстрирует структуру программы на *C*.

```
/*Пример программы на языке C*/
#include <stdio.h>                    // директива препроцессора
main ()                               // заголовок функции main
{                                     // начало тела функции main
char name[20];                        // объявление переменной
printf("Введите Ваше имя: ");         // вывод сообщения
scanf("%s",name);                     // ввод переменной
printf("Здравствуйте, %s!\n",name);
return(0);
}                                     // конец тела функции main ()
```

Комментарии предназначены для пояснения назначения переменных и действий программы. Комментарий является частью кода, который игнорируется компилятором. Могут вставляться в любое место программы, где допускаются пробелы. Существует два способа вставки комментариев в программу:

а) с помощью открывающей скобки комментария «/*» и закрывающей скобки комментария «*/». Текст, который находится между парой таких скобок, является комментарием. Оформленный таким образом комментарий может занимать более одной строки;

б) с помощью пары символов «//». Текст, записанный после данной пары символов до конца строки, является комментарием. С помощью этого способа можно вставлять только однострочные комментарии.

Директива #include сообщает компилятору, что необходимо включить в текст программы содержимое файла, имя которого указано в угловых скобках либо в кавычках. Такой файл называется заголовочным, имеет расширение *h* (от англ. *header* – заголовок). Фактически с помощью директивы #include к программе подключаются необходимые библиотеки функций. Например, в заголовочном файле stdio.h определяются некоторые макросы и переменные, используемые библиотекой ввода-вывода.

Существует большое количество стандартных функций. Они объединены в группы и хранятся в отдельных библиотеках по тематическому принципу (функции для математических расчетов, ввода-вывода, обработки строк и т. д.). Некоторые стандартные библиотеки подключаются автоматически. Подключение других библиотек нужно делать вручную, с помощью директивы #include <имя файла>.

Программа на языке Си состоит из отдельных подпрограмм (функций). В общем виде функция имеет вид:

```
тип_значения имя_функции (параметры)
{
        //тело функции
}
```

В рассматриваемом простом примере программа состоит из одной функции main(), не имеющей параметров. Функция main() присутствует в каждой программе, это основная функция, с нее начинается выполнение программы.

Функция main() по умолчанию должна возвращать целое значение. Значение, возвращаемое функцией, записывается с помощью оператора return. В примере возвращается целое значение 0, сигнализирующее операционной системе о нормальном завершении программы.

Тело функции – это блок, заключенный в фигурные скобки. Он может содержать объявления данных и операторы обработки данных. Каждый оператор может быть записан в нескольких строках. Оператор должен заканчивается символом «;». Например,

```
for(i=0; i<6; i++)
    printf(«i=%d\n», i);
```

2.2. Типы данных

2.2.1. Основные понятия

Основная цель любой программы состоит в обработке данных. Данные различного типа хранятся и обрабатываются по-разному. В любом алгоритмическом языке каждая константа, переменная, результат вычисления выражения или функции должны иметь определенный тип.

Тип данных определяет:

– *множество значений*, которые могут принимать величины этого типа;

– *внутреннее представление* данных в памяти компьютера;

– *операции и функции*, которые можно применять к величинам этого типа.

Исходя из этих характеристик, программист выбирает тип каждой величины, используемой в программе для представления реальных объектов. *Обязательное описание типа* позволяет компилятору производить проверку допустимости различных конструкций программы. От типа величины зависят машинные команды, которые будут использоваться для обработки данных.

Все типы языка *C*/*C++* можно разделить на *основные* и *составные.* В языке определено шесть основных типов данных для представления *целых*, *вещественных*, *символьных* и *логических* величин. На основе этих типов программист может вводить описание составных типов. К ним относятся *массивы*, *перечисления*, *функции*, *структуры*, *ссылки*, *указатели*, *объединения* и *классы*.

Основные (*стандартные*) типы данных часто называют *арифметическими*, поскольку их можно использовать в арифметических операциях. Для описания основных типов определены следующие ключевые слова:

int (целый);

char (символьный);

wchar_t (расширенный символьный);

bool (логический);

float (вещественный);

double (вещественный с двойной точностью).

Первые четыре типа называют целочисленными (*целыми*), последние два – *типами с плавающей точкой.* Код, который формирует компилятор для обработки целых величин, отличается от кода для величин с плавающей точкой.

Существует четыре *спецификатора типа*, уточняющих внутреннее представление и диапазон значений стандартных типов:

short (короткий);

long (длинный);

signed (знаковый);

unsigned (беззнаковый).

Диапазоны значения и объем занимаемой памяти для простых типов приведены в табл. 2.1.

Таблица 2.1

Числовые типы данных языка *C*

Тип данных	**Размер памяти, бит**	**Диапазон значений**
char (символьный)	8	от –128 до 127
signed char (знаковый символьный)	8	от –128 до 127
unsigned char (беззнаковый символьный)	8	от 0 до 255
short int (короткое целое)	16	от –32768 до 32767
unsigned int (беззнаковое целое)	16	от 0 до 65535 (16-битная платформа) от 0 до 4294967295 (32-битная платформа)
int (целое)	16 32	от –32768 до 32767 (16-битная платформа) от –2147483648 до 2147483647 (32-битная платформа)
long (длинное целое)	32	от –2147483648 до 2147483647
unsigned long (длинное целое без знака)	32	от 0 до 4294967295
long long int (C99)	64	от $-(2^{63}-1)$ до $2^{63}-1$
unsigned long long int (C99)	64	от 0 до $2^{64}-1$
float (вещественное)	32	от 3,4E–38 до 3,4E38
double (двойное вещественное)	64	от 1,7E–308 до 1,7E308
long double (длинное вещественное)	80	от 3,4E–4932 до 3,4E4932
_Bool (C99)	8	true(1), false(0)
bool (*C*++)	8	true(1), false(0)

В 32-разрядной ОС Windows тип int занимает в памяти 32 бита, и диапазон допустимых значений для знакового int в этом случае от –2147483648 до 2147483647. Такое различие в размере памяти, выделяемой под переменную типа int, объясняется тем, что тип int – машин-

но-зависимый, и для него выделяется одно машинное слово, длина которого в 16-разрядных процессорах – 16 бит, в 32-разрядных – 32 бита.

К основным типам языка относится тип также *void*. Множество значений этого типа *пусто*. Он используется:

– для определения функций, которые не возвращают значения;

– для указания пустого списка аргументов функции.

2.2.2. Целый тип (*int*)

Размер типа int не определяется стандартом, а зависит от компьютера и компилятора. Для 16-разрядного процессора под величины этого типа отводится 2 байта, для 32-разрядного – 4 байта.

Спецификатор *short* перед именем типа указывает компилятору, что под число требуется отвести 2 байта независимо от разрядности процессора.

Спецификатор *long* означает, что целая величина будет занимать 4 байта. Таким образом, на 16-разрядном компьютере эквиваленты *int* и *short int*, а на 32-разрядном – *int* и *long int*.

Внутреннее представление величины целого типа – целое число в двоичном коде. При использовании спецификатора *signed* старший бит числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное). Спецификатор *unsigned* позволяет представлять только положительные числа, поскольку старший разряд рассматривается как часть кода числа. Таким образом, диапазон значений типа *int* зависит от спецификаторов. Диапазоны значений величин целого типа с различными спецификаторами для IBM PC-совместимых компьютеров приведены в табл. 2.1.

По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор *signed* можно опускать.

Константам, встречающимся в программе, приписывается тот или иной тип в соответствии с их видом. Если этот тип по каким-либо причинам не устраивает программиста, он может явно указать требуемый тип с помощью суффиксов *L*, *l* (*long*) и *U*, *u* (*unsigned*). Например, константа *32L* будет иметь тип *long* и занимать 4 байта. Можно использовать суффиксы *L* и *U* одновременно, например, *OX22UL* или *05Lu*.

Примечание. Типы *short int*, *long int*, *signed int* и *unsigned int* можно сокращать до *short*, *long*, *signed* и *unsigned* соответственно.

2.2.3. Символьный тип (*char*)

Под величину символьного типа отводится количество байт, достаточное для размещения любого символа из набора символов для данного компьютера, что и обусловило название типа. Как правило, это 1 байт. Тип *char*, как и другие целые типы, может быть со знаком или без знака. В величинах со знаком можно хранить значения в диапазоне от –128 до 127. При использовании спецификатора *unsigned* значения могут находиться в пределах от 0 до 255. Этого достаточно для хранения любого символа из 256-символьного набора ASCII. Величины типа *char* применяются также для хранения целых чисел, не превышающих границы указанных диапазонов.

2.2.4. Расширенный символьный тип (*wchar_t*)

Тип *wchar_t* предназначен для работы *с* набором символов, для кодировки которых недостаточно 1 байта, например, Unicode. Размер этого типа зависит от реализации; как правило, он соответствует типу *short*.

Строковые константы типа *wchar_t* записываются с префиксом *L*, например, *L* «Gates».

2.2.5. Логический тип (*bool*)

Величины логического типа могут принимать только значения *true* и *false*, являющиеся зарезервированными словами.

Внутренняя форма представления значения *false* – 0 (нуль).

Любое другое значение интерпретируется как *true*. При преобразовании к целому типу *true* имеет значение 1.

2.2.6. Типы с плавающей точкой (*float*, *double* и *long double*)

Стандарт *C++* определяет три типа данных для хранения вещественных значений: *float*, *double* и *long double*.

Типы данных с плавающей точкой хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей – мантиссы и порядка. В IBM PC-совместимых компьютерах величины типа float занимают 4 байта, из которых один двоичный разряд отводится под знак мантиссы, 8 разрядов под порядок и 23 под мантиссу. Мантисса – это число, большее 1.0, но меньшее 2.0. Поскольку старшая цифра мантиссы всегда равна 1, она не хранится.

Для величин типа *double*, занимающих 8 байт, под порядок и мантиссу отводится 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка – его диапазон. Как можно видеть из табл. 2.1, при одинаковом количестве байт, отводимом под величины типа float и long int, диапазоны их допустимых значений сильно различаются из-за внутренней формы представления.

Спецификатор *long* перед именем типа *double* указывает, что под величину отводится 10 байт.

Константы с плавающей точкой имеют по умолчанию тип *double*. Можно явно указать тип константы с помощью суффиксов *F*, *f* (*float*) и *L*, *l* (*long*). Например, константа *2E+6L* будет иметь тип *long double*, а константа *1.82f* – тип *float*.

Для вещественных типов в таблице приведены абсолютные величины минимальных и максимальных значений.

Для написания переносимых на различные платформы программ нельзя делать предположений о размере типа int. Для его получения необходимо пользоваться операцией *sizeof*, результатом которой является размер типа в байтах. Например, для операционной системы MS-DOS *sizeof* (*int*) даст в результате 2, а для Windows 9X или OS/2 результатом будет 4.

В стандарте ANSI диапазоны значений для основных типов не задаются, определяются только соотношения между их размерами, например:

sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)

sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)

Различные виды целых и вещественных типов, различающиеся диапазоном и точностью представления данных, введены для того, чтобы дать программисту возможность наиболее эффективно использовать возможности конкретной аппаратуры, поскольку от выбора типа зависит скорость вычислений и объем памяти. Но оптимизированная для компьютеров какого-либо одного типа программа может стать не переносимой на другие платформы, поэтому в общем случае следует избегать зависимостей от конкретных характеристик типов данных.

2.2.7. Тип *Void*

К основным типам языка относится тип также *void*. Множество значений этого типа *пусто*. Он используется:

– для определения функций, которые не возвращают значения;

– для указания пустого списка аргументов функции.

2.3. Переменные и константы

Переменная – это величина, которая имеет имя и которая может изменять свои значения в процессе выполнения программы.

Примеры объявления переменных:

```
int q, w, r;
float t1, u;
shot b, j;
long r, u, p, g;
char s34;
unsigned w234;
double b_1, G6;
enum seasons{spring, summer, autumn, winter} m. n. i;
```

Имя переменной (или идентификатор) составляется из латинских букв, цифр и знака подчеркивания. Первым символом с имени должна быть буква или знак подчеркивания. Современные компиляторы допускают длинные имена.

Переменные можно инициализировать, т. е. определять их значения при объявлении с помощью оператора объявления:

имя_типа имя_переменной = значение;

Например:

```
float a=–2.4e4;
int b=456;
```

Константа – это величина, которая не изменяет своего значения в процессе выполнения программы. Константы могут быть неименованными и именованными.

Неименованные константы – это обычные константы, представленные в выражении своими значениями, например, числами (–3,1; 456; 2,123e-2), строкой символов ("Поздравляю с поступлением в университет!").

Имя и значение *именованной константы* определяются с помощью спецификатора *const*. Форма объявления именованной константы:

const тип имя_константы = выражение;

Например:

```
const float k1=2.48567,
            k2= 1/k1;
const char c='*';
const char *f="Маша+Ваня";
const char s[]="г. Гомель";
```

***Типы констант*:**

– *вещественные*;

– *целые*;

– *длинные целые*, в конце записи которых добавляется буква L, например, 4567382L;

– *беззнаковые*, в конце записи которых добавляется буква U;

– *восьмеричные*, в которых перед первой значащей цифрой записывается нуль (0), например, 071;

– *шестнадцатеричные*, в которых перед первой значащей цифрой записывается пара символов нуль-икс (0X), например, 0X2D1;

– *символьные*, в которых единственный символ заключен в одинарные кавычки, например, '2', '&', 'y';

– *строковые*, представляющие собой последовательность символов, заключенные в двойные кавычки, например, "Гомельский государственный технический университет имени П. О. Сухого".

2.4. Разработка и отладка линейных алгоритмов

Линейными называются алгоритмы, в которых выполняются все команды последовательно, одна за другой. При программировании таких алгоритмов используются операторы ввода и вывода данных, операторы присваивания, арифметические выражения. *Арифметические выражения* составляются из констант, переменных, обращений к математическим функциям с помощью знаков операций и скобок. Арифметические выражения могут использоваться в операторах присваивания, операторах вывода. Например,

$r = \text{sqrt}(\text{d}/\ (0.56 * \text{c} - \text{a}));$

$\mu = y - b * \text{pow}(\text{fabs}(\text{r} - \text{d}), 1./3));$

$\alpha = \mu + r;$

$\text{printf}("r = \%.2\text{f}\ \text{c} = \%.3\text{f}\ \text{знач.выражения} : \%.4\text{f}\backslash\text{n}", \text{r}, \text{c}, \text{r} * \sin(\text{c}));$

Сразу же оговоримся, что в рассмотренных ниже примерах исходные данные задаются корректно, т. е. например, если в расчете функции используется ln(x), то значение x будет задано большим, чем 0, так как в противном случае произойдет ошибка вычисления (логарифм для $x \leq 0$ не существует). Поэтому схема алгоритма проверки условия $x > 0$ не предусматривает. Хотя, конечно, в серьезных задачах нужно обязательно проверять все вводимые значения на предмет того, допустимы ли они или нет.

Пример составления и отладки линейной программы.
Составить программу вычисления *y* и *z* по формулам:

$$y = \ln\left|\alpha - \sqrt{|x|}\right|;\ z = x - \frac{b}{\alpha + \frac{x^2}{4}}.$$

Решение: исходными данными являются значения α, *b* и *x*.

Таблица 2.2

Таблица соответствия переменных

Переменные в задаче	**Имя на языке Си**	**Тип**	**Комментарий**
α	a	float	исх. данное
b	b	float	исх. данное
x	x	float	исх. данное
y	y	float	результат
z	z	float	результат

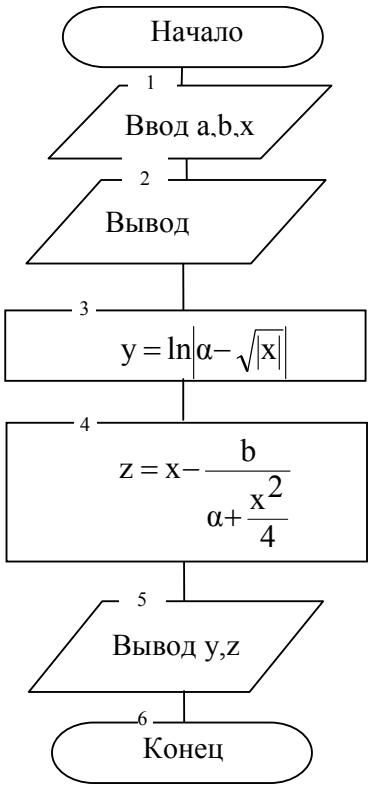


Рис. 2.1. Графическая схема алгоритма

Графическая схема алгоритма приведена на рис. 2.1. В данном случае она слишком тривиальна, поэтому необязательна.

Ниже приведен текст программы.

```
/* Пример линейной программы */
#include <stdio.h>
#include <math.h>
main()
{
  float x,y,z,a,b;                  // Описание переменных
  printf("Введите x,a,b\n");          // Вывод сообщения
  scanf("%f %f %f»,&x,&a,&b);           // Ввод исх. данных
// Вывод исх. данных
  printf("Исходные данные:\nx=%7.3f a=%7.3fb=%7.3f\n",x,a,b);
// Вычисление значения y
  y=log(fabs(a-sqrt(fabs(x))));
// Вычисление значения z
  z=x-b/(a+x*x/4);
// Вывод результатов
  printf("Результаты\n y=%f  z=%f\n",y, z);
  fflush(stdin);                        // Очистка буфера
  getchar();                    // Ожидание ввода символа
  return(0);
}
```

Для проверки правильности составленной программы выполняется ее отладка с помощью ПК следующим образом. Программа запускается на выполнение с вводом исходных данных, для которых ответ известен (указанный набор исходных данных с соответствующими результатами называется ***тестом***). Результаты выполнения программы сравниваются с известными. Ниже приводится тест и результаты выполнения программы. Значения *y* и *z* для теста подсчитаны в MachCad.

Тест:	Результаты выполнения программы:
$\alpha = 1{,}2$, $b = 5$, $x = 0{,}3$	Исходные данные:
$y = -0{,}42729$, $z = -3{,}78998$	$x = 0{,}300$ $a = 1{,}200$ $b = 5{,}000$
	Результаты:
	$y = -0{,}427285$ $z = -3{,}78997$

Видим, что результаты выполнения программы совпали с тестом.

3. ПРОГРАММИРОВАНИЕ РАЗВЕТВЛЯЮЩИХСЯ АЛГОРИТМОВ

3.1. Понятие разветвляющегося алгоритма и программы

Разветвляющимся называется алгоритм, в котором последовательность и количество выполняемых команд зависит от выполнения или не выполнения некоторых условий. В разветвляющихся алгоритмах используются команды *ветвления* и *выбора из большого количества вариантов*. Графическое изображение команд ветвления представлено на рис. 3.1.

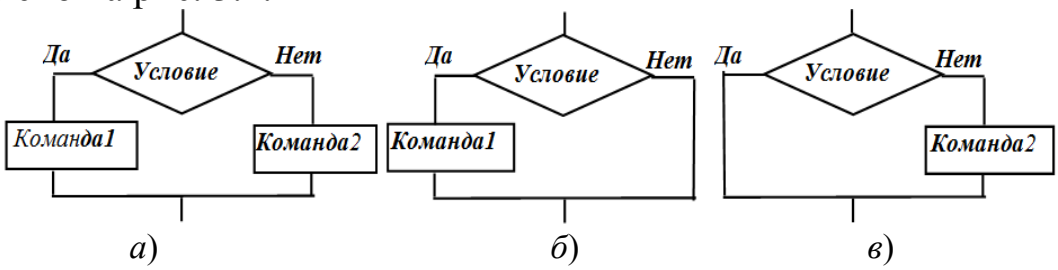


Рис. 3.1. Команды ветвления:
а – в полной форме; *б*, *в* – в сокращенной форме

Для программирования ветвлений используется оператор *if*, а для программирования выбора – операторы *switch* и *break*.

3.2. Операторы управления разветвляющимся вычислительным процессом

3.2.1. Логические выражения

Логическое выражение (условие) – выражение, которое содержит *знаки операций отношения* и/или *знаки логических операций*. Значением логического выражения может быть только 1, если логическое выражение есть ИСТИНА (true), или 0, если логическое выражение ЛОЖЬ (false).

Операции отношения являются бинарными и обозначаются следующим образом (приведены в порядке уменьшения приоритета):

< (меньше);
<= (меньше или равно);
> (больше);
>= (больше или равно);

== (равно);
!= (не равно).

Примеры: a<b, x!=5, y==c, t>=8.1, d<>(w-c).

Логические операции – это операции:

&& – логическое И (бинарная операция);

|| – логическое ИЛИ (бинарная операция);

! – логическое НЕ (унарная операция).

Логические операции имеют более низкий приоритет, чем операции отношения. Результаты выполнения логических операций приведены в табл. 3.1–3.3.

Таблица 3.1

x	**y**	**x && y**
1	1	1
0	1	0
1	0	0
0	0	0

Таблица 3.2

x	**y**	**x || y**
1	1	1
0	1	1
1	0	1
0	0	0

Таблица 3.3

x	**!x**
1	0
0	1

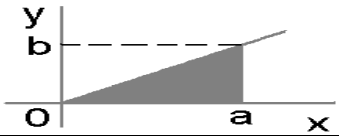
Например, логическое выражение a>3 && c<7 при a=5, c=6 будет иметь значение ИСТИНА, а при a=3, c=8 – ЛОЖЬ; логическое выражение a>3 || c<7 при a=5, c=6 будет иметь значение ИСТИНА, при a=2, c=6 – ЛОЖЬ; при a=7 логическое выражение !a>2 будет иметь значение ЛОЖЬ.

При составлении логических выражений следует учитывать приоритет выполнения операций.

Примеры записи логических выражений приведены в табл. 3.4.

Таблица 3.4

Таблица записи логических выражений

Условие	Логическое выражение
$x \in [a,b]$	x>=a && x<=b
$x \notin [a,b]$	x<a || x>b
$x \in [a,b]$ или $x \in [c,d]$	x>=a && x<=b || x>=c && x<=d
Хотя бы одно из чисел x, y положительное	x>0 || y>0
Только одно из чисел x, y положительное	(x>0 && !y>0) || (y>0 && !x>0)
Ни одно из чисел x, y не является четным	x % 2 !=0 && y % 2 !=0
Точка (x, y) принадлежит заштрихованной области 	x>=0 && x<=a && y>=0 && y<=b/a*x

3.2.2. Оператор *if*

Оператор *if* называется *условным* оператором и используется для программирования ветвлений. Имеет две формы записи – сокращенную и полную.

Сокращенная форма записи оператора *if* имеет вид:

if (выражение) оператор1;

Например, if (a>b) y=2*x;

Оператор *if* в полной форме называется оператором *if-else* и имеет следующую форму записи:

if (выражение) оператор1; else оператор2;

Например, *if (x>0 && x!=10)*

*y=2*x;*

else

*y=x*x;*

Здесь *выражение* – выражение, которое может иметь арифметический тип или тип указателя; *оператор1*, *оператор2* – простые или составные операторы языка. *Простой оператор* – это один оператор. *Составной оператор* (блок операторов) – это последовательность из нескольких любых операторов, в том числе операторов описания, заключенных в фигурные скобки.

Например, *if (x<=5.6)*

```
{ float a=2.1;
  y=a*sin(x);
  printf("%.4f\n",y) ;
}
```

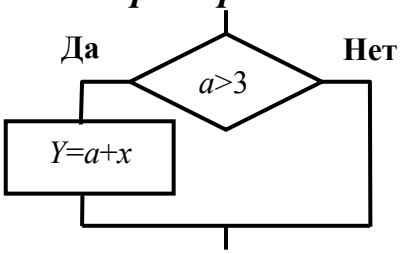
Следует учитывать, что переменная, описанная в блоке, не существует вне блока.

Оператор *if* называется *вложенным*, если хотя бы один из операторов *оператор1*, *оператор2* содержит условные операторы, и используется для программирования вложенных ветвлений.

При выполнении оператора *if* и *if-else* сначала вычисляется значение выражения *выражение*. Если полученное значение не равно нулю (имеет значение *true*), то выполняется *оператор1*, иначе: для оператора *if* – управление передается на оператор, следующий за условным; для оператора *if-else* – выполняется *оператор2*, а затем управление передается на оператор, следующий за условным.

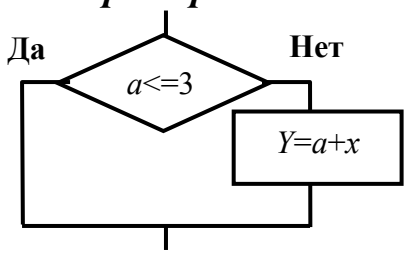
Ниже приведены примеры программирования ветвлений.

Пример 1



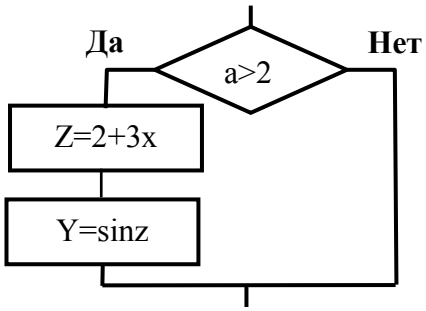

```
if (a>3) y=a+x;
```

Пример 2



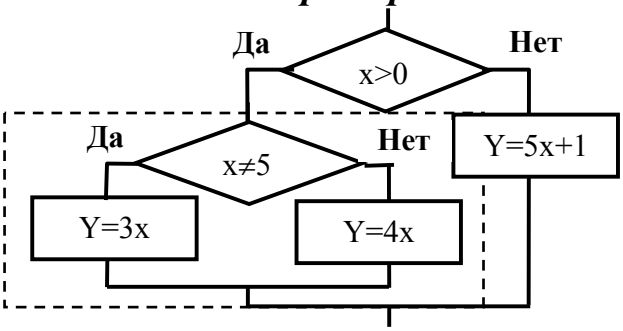

```
if (a<=3) { ; } else y=a+x;
```

Пример 3



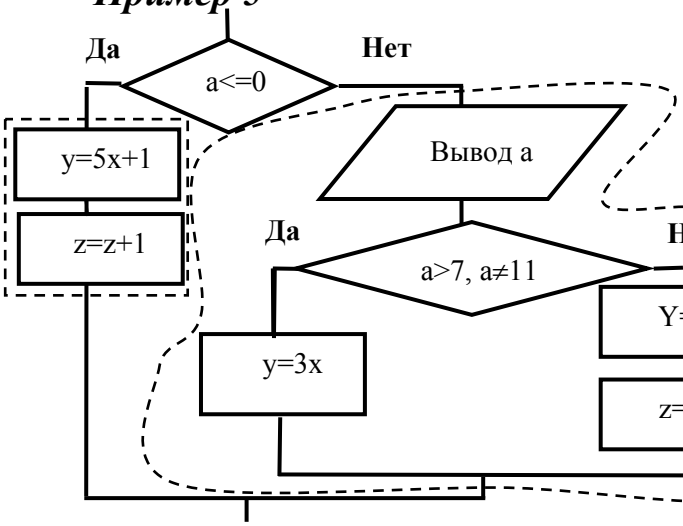

```
if (a>2) { z=2+3*x;
   y=sin(z);
 }
```

Пример 4




```
if (x>0) { if (x!=5) y=3*x;
else y= 4*x; }
else y=5*x+1;
```

Пример 5




```
if (a<=0)
 { y=5*x+1;
   z++;
 }
   else
 { cout << " a=" << a;
   if (a>7 && a!= 11
      y=3*x;
      else{ y=4*x; z--;
       }
 }
```

В примерах 1 и 2 оператор *y=a+x;* является простым оператором. В примере 3 оператор *{ z=2+3*x; y=sin(z);}* является составным. Кроме того, примеры 1 и 2 показывают, что неполное ветвление, в котором отсутствуют команды на ветви «Да», может быть приведено к другой форме неполного ветвления путем изменения условия на противоположное (условие, противоположное условию *a>3*, есть условие *a<=3*, и наоборот). Из примера 2 видно, что отсутствующие команды ветви «Да» программируются с помощью пустого оператора {;}.

Пример 4 иллюстрирует программирование вложенного ветвления. Фигурные скобки в данном случае не обязательны, так как компилятор относит часть *else* к ближайшему *if*.

В 5-м примере на ветви «Да» внешнего ветвления имеется группа команд, которой соответствует составной оператор *{ y=5*x+1; z++;}*. На ветви «Нет» этого ветвления имеется группа команд, содержащая другое ветвление. Этой группе команд соответствует составной оператор *{ cout << "a=" << a; if (a>7 && a! = 11) y=3*x; {y=4*x; z--; } }*. Вложенное ветвление на ветви «Да» содержит одну команду (соответствующий оператор *y=3*x;*), а на ветви «Нет» – две команды, которым соответствует составной оператор *{ y=4*x; z--; }*.

3.2.3. Примеры программирования разветвляющихся алгоритмов

Задача 1. Даны два вещественных числа x и y – координаты точки (x, y) на плоскости и область D, заданная графически (рис. 3.2).

Вычислить значение z, заданное формулой

$$z = \begin{cases} 1 + xy, \text{ если } (x, y) \in D; \\ 5, \text{ если } (x, y) \notin D. \end{cases} \qquad (3.1)$$

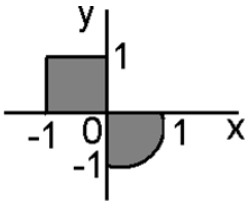


Рис. 3.2. D – заштрихованная область

Решение. Из рис. 3.2 видно, что область *D* является объединением двух областей – квадрата со стороной, равной 1, и части круга с радиусом, равным 1. Следовательно, условие $(x, y) \in D$ состоит в том, что точка с координатами (x, y) принадлежит квадрату или сегменту круга. Условие принадлежности точки квадрату, изображенному на рис. 1.2, можно записать в виде логического выражения

$$x>=-1 \;\&\&\; x<=0 \;\&\&\; y<=1 \;\&\&\; y>=0, \qquad (3.2)$$

а условие принадлежности точки сегменту – в виде логического выражения

$$x>0 \;\&\&\; x<=1 \;\&\&\; y>=-1 \;\&\&\; y<0 \;\&\&\; x*x+y*y<=1. \qquad (3.3)$$

Объединив выражения (3.2) и (3.3) знаком логической операции || (ИЛИ), получим логическое выражение, соответствующее условию $(x, y) \in D$, которое затем используем в операторе *if-else*.

Графическая схема алгоритма решения задачи изображена на рис. 3.3. Алгоритм содержит ветвление, на каждой ветви которого имеется лишь по одной команде. Программируется такое ветвление с помощью оператора *if-else*.

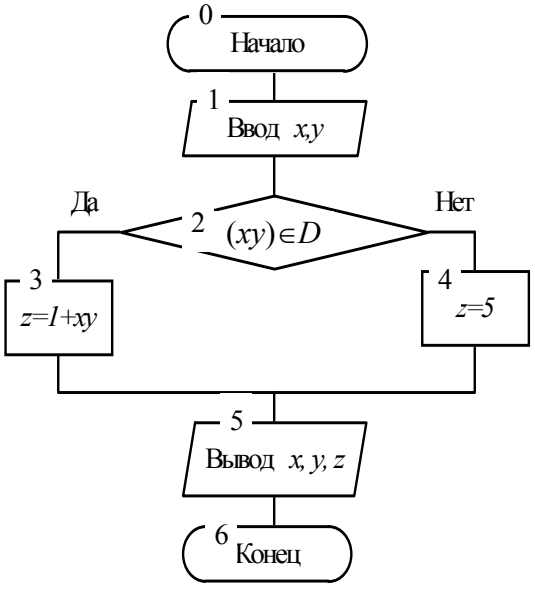


Рис. 3.3. Решение задачи 1

```
/*   Текст программы  */
#include <stdio.h>
#include <math.h>
main()
{    float x,y,z;
   puts("Введите координаты точки (x,y) ");
   scanf("%f %f»,&x,&y);
   if ( x>= -1 && x<=0 && y<=1 && y>=0
     || x>0  && x<=1 && y>=-1 && y<0 && x*x+y*y<=1 )
                    z=1+x*y;
              else z=5;
  printf("Для точки (%.2f,%.2f) z=%.4f",  x,  y,  z);
  fflush(stdin);  getchar();
 return(0);
}
```

Задача 2. Вычислить значение функции

$$y=\begin{cases} x^2, \quad \text{если} \quad x>5 \text{ и } x\neq 10; & //1 \\ \sin x \text{ , если} \quad 0\le x\le 5; & //2 \\ 2x- \text{ в остальных случаях.} & //3 \end{cases}$$

Решение. Необходимо задать значение x и проанализировать, по какой из трех формул необходимо вычислить y. При этом придется задать один или два вопроса. Например, для x=6 зададим вопрос x>5 и $x\neq 10$? Ответ будет «Да», следовательно, вычислять y надо по первой формуле. Пусть x = 3,14159. На вопрос x>5 и $x\neq 10$? ответ будет «Нет», т. е. вычислять y надо по второй или третьей формуле. Для выяснения, по какой конкретно, необходимо задать еще один вопрос, например, $0\le x\le 5$? В этом случае ответ будет «Да». Следовательно, y должен вычисляться по второй формуле. Рассуждая аналогично, придем к выводу, что для x=–5 необходимо задать два вопроса x>5 и $x\neq 10$? и $0\le x\le 5$? На каждый из вопросов будет получен ответ «Нет», следовательно, y надо вычислять по третьей формуле. Таким образом, приходим к разветвляющемуся алгоритму, изображенному на рис. 3.4. В алгоритме предусмотрен вывод исходных данных, формулы и номера формулы, по которой производится вычисление y.

Сведем в таблицу соответствия (табл. 3.5) обозначения переменных в задаче и в программе.

Таблица 3.5

Таблица соответствия

Обозначения в задаче	Имена в алгоритме и программе	Тип данных	Комментарий
x	x	вещественный	исходное данное – аргумент функции
y	y	-//-	результат – значение функции
–	n	целочисленный	номер формулы
–	z	строка	вид формулы

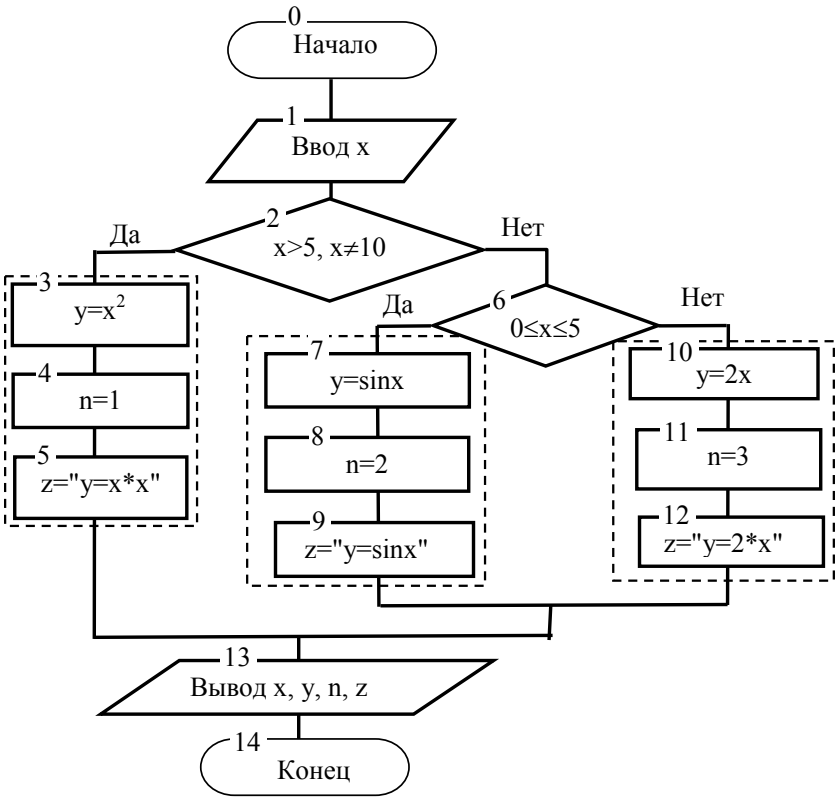


Рис. 3.4. Алгоритм решения задачи 2

В программе блокам 3–5, 7–9, 10–12 графической схемы соответствуют составные операторы *оператор1*, *оператор2*, *оператор3*, соответственно.

```
/*  Программа задачи 1.2  */
#include <stdio.h>
#include <math.h>
main()
{
```

```
float x,y;
int n;
char *z;
puts("Введите x");
scanf("%f",&x);
if(x>5 && x!=10)
        {    y=x*x;                          } Оператор 1
             n=1;
             z="y=x*x";
         }
   else
      if(x>=0 && x<=5)
         {                                    } Оператор 2
           y=sin(x);
           n=2;
           z="y=sinx";
         }
                 else
             {                                } Оператор 3
               y=2*x;
               n=3;
               z="y=2x";
             }
 printf("При x=%.2f y=%.4f (формула %d:%s)\n",x,y,n,z);
 fflush(stdin);
 getchar();
 return(0);
}
```

Для проверки правильности разработанного алгоритма и программы достаточно подготовить семь тестов. Сделаем это с помощью рис. 3.5. На рис. 3.5 на числовой оси отмечены промежутки значений *x* с указанием формул для вычисления *y* для каждого промежутка и каждой граничной точки. В табл. 3.6 приведены тесты.

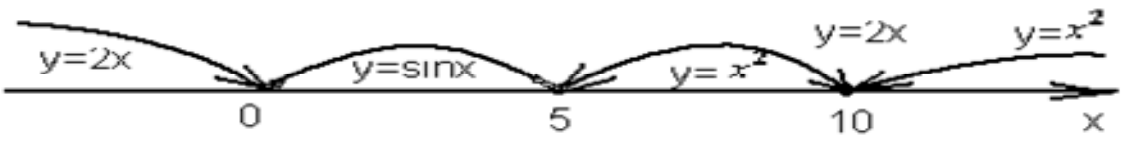


Рис. 3.5. Промежутки значений *x*

Таблица 3.6

Тесты к задаче 1.2

Тест №	Значение *x*	Значение *y*	Формула для вычисления *y*
Тест 1	–7,2	–14,4	$y = 2x$
Тест 2	0	0	$y = \sin x$
Тест 3	1,5707968	«1	$y = \sin x$
Тест 4	5	–0,9589	$y = \sin x$
Тест 5	6	36	$y = x^2$
Тест 6	10	20	$y = 2x$
Тест 7	11	121	$y = x^2$

3.2.4. Выбор из большого числа вариантов

Выбор из многих вариантов можно сделать с помощью вложенных операторов *if-else*. Более удобный способ – использование операторов *switch* и *break*. Общий вид оператора выбора:

```
switch (i)
 {
  case k1 : op1;
  [ break;]
  case k2 : op2;
  [ break;]
  …
  case kn : opn;
  [ break;]
  [ default : opn+1;]
 }
```

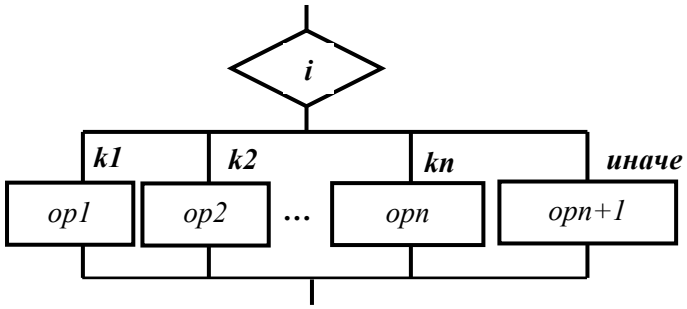


Рис. 3.6. Структура команды выбора, соответствующая оператору *switch* с оператором *break* после каждого *case*

Здесь *i* – любое выражение, дающее целое значение (в том числе – символьное);

k1, *k2*, ..., *kn* – константы или константные выражения – возможные значения *i* (например, *2*g*, *'a'-'b'*, *'1'*) – записываются после слова *case*;

op1, *op2*, ..., *opn*, *opn+1* – простые или составные операторы языка.

Выполнение оператора switch:

1. Вычисляется значение выражения *i.*

2. Значение *i* сравнивается с *k1*, *k2*, ..., *kn.*

3. При совпадении *i* со значением *ki* выполняется оператор *opi*. Затем управление передается оператору, стоящему после следующего

case, если после *opi* нет оператора *break.* Если после оператора *opi* есть оператор *break*, то управление передается на оператор, следующий после оператора *switch*.

4. Если *i* не совпало ни с одним *k1*, *k2*, ..., *kn*, то выполняется оператор *opn+1*, стоящий после *default*, а затем оператор, следующий после оператора *switch*. При отсутствии *default* выполняется следующий после *switch* оператор.

Рассмотрим примеры решения задач с использованием оператора выбора.

Задача 3. Задан номер текущего дня недели. Вывести названия дней, оставшихся до конца недели, включая текущий день.

Решение. Возможные значения номера дней недели (обозначим их через *n*) – это числа 1–7. При вводе числа 1 программа должна выводить названия "Понедельник", "Вторник", …, "Воскресенье", при вводе числа 2 – названия "Вторник", …, "Воскресенье" и т. д. При вводе числа 7 – только название "Воскресенье". При вводе любого другого числа – сообщение «Неправильно введен номер дня недели». Реализация выбора из такого множества вариантов может быть осуществлена с помощью оператора *switch(k)*. Ниже приведен текст программы с оператором *switch(k)*, в котором только после последнего *case* есть оператор *break*;, по которому и происходит выход из *switch* и переход на следующий оператор программы (*fflush(stdin);*) для $k = 1–7$. Если $k \neq 1–7$, то выполнится оператор *puts ("Неправильно введен номер дня недели")*;, стоящий после *default*, затем следующий оператор программы (*fflush(stdin)*;).

```
 /* Программа задачи 3 */
#include <stdio.h>
#include <math.h>
main()
{
  int k;                        // Номер дня недели
  puts("Введите номер дня недели");
  scanf("%d»,&k);
  printf("До конца недели:\n");
  switch (k)
  {
   case 1: puts ("Понедельник");
   case 2: puts ("Вторник");
   case 3: puts ("Среда");
   case 4: puts ("Четверг");
   case 5: puts ("Пятница");
```

```
  case 6: puts ("Суббота");
  case 7: puts ("Воскресенье \n");
  break;
  default: puts ("Неправильно введен номер дня недели");
 }
 fflush(stdin);  getchar();
 return(0);
}
```

Следующий пример показывает, что после *case ki:* может не быть оператора.

Задача 4. Ввести символ и определить, является ли он цифрой.

Решение. Пусть *k* – вводимый символ. В программе использованы операторы ввода-вывода в стиле Си.

В задаче 5 используется оператор выбора с оператором *break* после каждого *case ki:*

Задача 5. Дано целое число в диапазоне 1–5. Вывести строку – словесное описание соответствующей оценки (1 – «плохо», 2 – «неудовлетворительно», 3 – «удовлетворительно», 4 – «хорошо», 5 – «отлично»).

Решение. В программе *N* – число, которое вводится, а затем анализируется. Если ввести, например, 4, то программа выведет слово «Хорошо», а если ввести 7, то программа выведет строку «Нет оценки» и т. д. В программе использованы операторы ввода-вывода в стиле *C++*.

Программа задачи 4

```
#include <stdio.h>
main()
{
 char k;
 puts("Введите символ");
 k=getchar();
 switch (k)
 {
  case '1':
  case '2':
  case '3':
  case '4':
  case '5':
  case '6':
  case '7':
  case '8':
  case '9':
```

Программа задачи 5

```
#include <stdio.h>
#include <iostream.h>
main()
{
 int N;
 cout << "\nВведите число";
 cin >> N;
 switch (N)
 {
  case 1: cout << "\nПлохо";
  break;
  case 2: cout << "\nНеуд. ";
  break;
  case 3: cout << "\nУдовл. ";
  break;
  case 4: cout << "\nХорошо";
  break;
```

```
    case '0':printf("Это число
 %c\n",k);
    break;
    default: printf("%c-не
 число\n",k);
  }
  fflush(stdin);
  getchar();
  return(0);
 }
```

```
  case 5: cout << "\nОтлично";
  break;
 default: cout << "\nНет оценки";
 }
 fflush(stdin);
 getchar();
 return(0);
}
```

4. ПРОГРАММИРОВАНИЕ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ

4.1. Понятие цикла

При составлении алгоритмов решения практических задач нередко приходится некоторые действия выполнять многократно, каждый раз с новыми значениями для величин, входящих в повторяющуюся группу действий.

Многократно повторяемые участки вычислений называют циклами. Один проход цикла называется итерацией или повторением. Величины, определяющие количество повторений цикла, называются параметрами циклов. Параметрами цикла могут быть константы и переменные, причем среди них обязательно должна быть хотя бы одна переменная, значение которой изменяется при выполнении цикла. Целочисленные параметры цикла, изменяющиеся с постоянным шагом на каждой итерации, называются счетчиками цикла. Вычислительные процессы, содержащие циклы, называют циклическими.

Циклы состоят из следующих частей:

– подготовки к выполнению цикла (присваивания параметрам цикла начальных значений);

– определения момента завершения цикла (условия повторения цикла);

– тела цикла, включающего:

а) рабочую часть цикла, содержащую действия, непосредственно связанные с решением задачи, для которой разработан алгоритм;

б) подготовку данных для очередного шага цикла (изменение параметров цикла).

Различают циклы с предусловием и циклы с постусловием. В циклах с предусловием условие повторения цикла проверяется раньше, чем выполняется тело цикла, поэтому возможно, что цикл не выполнится ни разу. В циклах с постусловием условие повторения цикла проверяется после выполнения тела цикла и подготовки данных для очередного шага цикла, поэтому цикл выполняется всегда хотя бы один раз. В общем виде циклы с предусловием и постусловием графически можно изобразить, как показано на рис. 4.1. В языке Си циклы программируются с помощью трех разных операторов *for*, *while*, *do-while*. Возможно принудительное завершение текущей итерации и цикла в целом. Для этого служат операторы *break*, *continue*, *return*, *goto*. Передавать управление извне внутрь цикла не рекомендуется.

Использование оператора *goto* приводит к созданию запутанного, трудно читаемого кода, прозванного программистами «спагетти». Использование оператора *goto* среди программистов считается верхом неприличия. Чтобы избежать оператора *goto*, используют сложные условия повторения цикла.

На рис. 4.1 представлены графические схемы циклических алгоритмов.

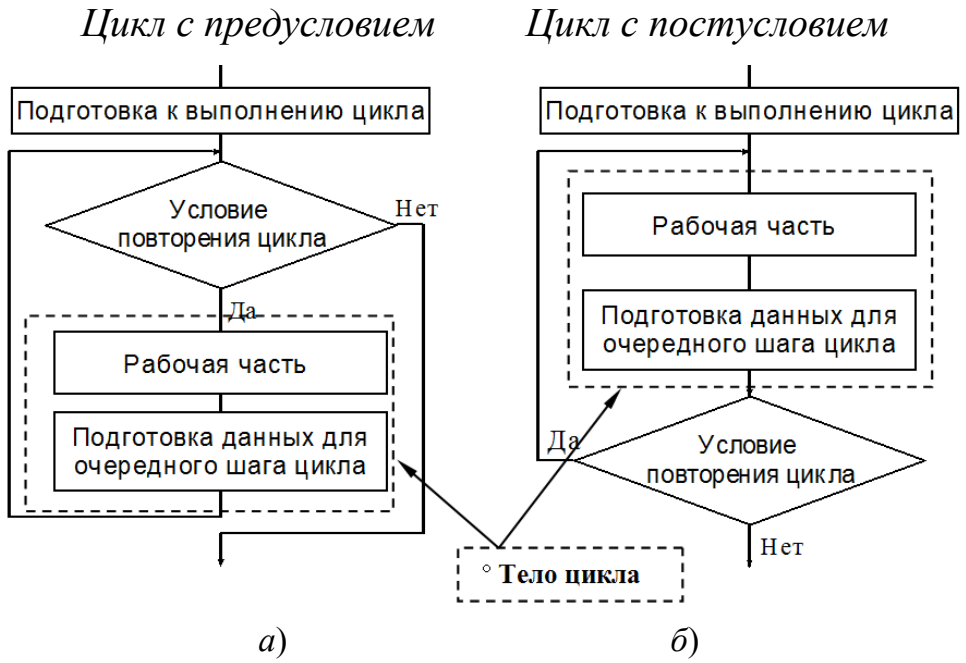


Рис. 4.1. Графические схемы циклов:
а – с предусловием; *б* – с постусловием

4.2. Программирование цикла с заранее известным числом повторений

Оператор цикла *for* – оператор цикла с параметром и с предусловием. Используется, если заранее известно количество повторений тела цикла, начальные значения параметров цикла, условие повторения цикла, как изменяются параметры цикла. Общий вид оператора цикла *for*:

for (p1; p2; p3) S;

Здесь

for(p1; p2; p3) – заголовок цикла.

S – простой или составной оператор языка Си – тело или рабочая часть цикла;

p1 – список операторов, разделенных запятой, инициирующих начальные значения (как правило) параметров цикла. Эти операторы выполняются один раз до начала выполнения рабочей части цикла. Областью действия переменных, объявленных в этой части цикла, является цикл;

p2 – список операторов и выражений, определяющих условие повторения цикла. Выполняется перед каждым выполнением тела цикла. При этом, если значение последнего выражения списка *p2* истинно (!=0), то тело цикла выполняется, а если ложно (==0), происходит выход из цикла к следующему оператору программы;

p3 – список операторов и выражений изменения параметров цикла (подготовка данных для очередного шага цикла). Выполняется после каждого выполнения тела цикла.

Пример (вычисление и вывод квадратов чисел от 1 до 10):

for (i=1; i<=10; *i++) printf*("*i*i*=%d\n",*i*i*);

Здесь *p1* – оператор *i=1* (инициирует начальное значение параметра цикла *i*); *p2* – выражение i<=10 (определяет условие повторения цикла); *p3* – оператор *i++* (оператор изменения параметра цикла).

S – оператор *printf*("*i*i*=%d\n",*i*i*); (рабочая часть цикла).

Пример (вычисление суммы чисел от 1 до 15)

for (int i=1,q=0; i<=15; *i++) q+=i;*

Здесь *p1* – список операторов, разделенных запятой. Первый из этих операторов (*int i=1*) объявляет параметр цикла *i* и инициирует его начальное значение и объявляет и инициирует начальное значение вычисляемой суммы *q*. Как и в первом примере, *p2* – выражение *i*<=15; *p3* – оператор *i++*. *S* – оператор *q+=i.*

Основные правила и порядок выполнения оператора цикла for:

1. До начала выполнения тела цикла выполняется один раз список *p1*.

2. Выполняются операторы и выражения списка *p2*; производится анализ полученного значения последнего выражения из *p2* (в дальнейшем *p2*):

а) если *p2* истинно (!=0), то тело цикла выполняется;

б) если *p2* ложно (==0), то тело цикла завершается;

в) если *p2* ложно (==0) до первого выполнения тела цикла, то тело цикла не выполняется ни разу.

3. После выполнения рабочей части цикла выполняются операторы и выражения *p3* и осуществляется переход к п. 2 данных правил.

4. Появление в любом месте тела цикла оператора *continue* вызывает переход к выполнению списка *p3* п. 3 данных правил.

5. Появление в любом месте тела цикла оператора *break* вызывает переход к оператору, следующему после оператора цикла, т. е. осуществляется выход из цикла. При этом параметр цикла сохраняет то значение, при котором произошло завершение выполнения цикла.

6. После нормального завершения цикла значение параметра цикла равно значению, которое привело к завершению цикла.

Пример

```
for (i=1; i<5; i++ );
printf ("i=%d\n", i );
```

Приведен цикл, не содержащий тела цикла, так как сразу после заголовка цикла стоит символ «;» (тело цикла – пустой оператор), и оператор вывода значения *i*. Выполнение цикла завершится при *i = 5*. Значение *5* и будет выведено.

7. Если в заголовке цикла отсутствует список *p1* и/или *p2* и/или *p3*, то символы «;» должны остаться. Например:

```
for ( ; ; );          //Бесконечный цикл
for ( i=1; ; i++ );   //Бесконечный цикл
for ( i=1; i<5 ; );   //Бесконечный цикл
```

Примечание. Использование в заголовке цикла посторонних вычислений считается плохим стилем. Следует использовать заголовок цикла *for* для операций по управлению циклом.

Пример. Вычислить и вывести степени двойки – значения 2^n для $n = 1, 2, \ldots, k$ ($k > 1$).

Решение. Пусть x – искомые значения. Положим $x = 1$. Тогда первая степень двойки x равна $x*2$, что равно 2^1. Если положить

*x=x*2*, то вторая степень двойки есть *x*2*, что равно 2^2. И так далее. Таким образом, степени двойки от первой до *k*-й можно вычислить, выполнив *k* раз рекуррентное соотношение *x = x*2*, положив предварительно *x*=1. Реализовать такой алгоритм можно с использованием цикла с параметром и с предусловием. Графическая схема алгоритма, программа на языке Си, результаты выполнения программы при *k* = 15 изображены на рис. 2.1. Параметр цикла ***i*** (степень двойки) принимает значения 1, 2, …, *k*, следовательно, является счетчиком.

```
 #include <stdio.h>
main()
 {
  int i, k;
  long x;
  puts("Введите k-степень двойки");
  scanf("%d", &k);
  x=1;
  for(i=1; i<=k; i++)  //Заголовок цикла
    {
     x*=2;                          } Раб. часть
     printf("%5ld",x);              } цикла
     if(i%5==0) printf("\n");        }
    }
  printf("\nНажмите любую клавишу\n");
  fflush(stdin);
  getchar();
  return(0);
 }
```

Результаты выполнения программы при *k* = 15 представлены на рис. 4.2.

```
    2     4     8    16    32
   64   128   256   512  1024
 2048  4096  8192 16384 32768
```

Рис. 4.2. Результаты выполнения программы при *k* = 15

Примечание. Блок-схему алгоритма разработать самостоятельно.

4.3. Программирование цикла с заранее неизвестным числом повторений

4.3.1. Оператор цикла *while*

Оператор цикла ***while*** – оператор цикла итеративного типа с предусловием. Используется, когда количество повторений операторов тела цикла заранее неизвестно и определяется в процессе выполнения цикла. В этом операторе анализ конца цикла производится до выполнения операторов тела цикла.

Общий вид оператора цикла ***while***:

***while (b) S*;**

Здесь ***b*** – выражение любого типа, например, логическое, приводимое к арифметическому типу, определяющее условие повторения цикла; ***S*** – простой или составной оператор – тело цикла. Он должен включать операторы рабочей части цикла и операторы изменения операндов выражения ***b*** (подготовки данных для очередного шага цикла).

Основные правила использования и порядок выполнения оператора цикла ***while***:

1. До оператора ***while*** в программе должны содержаться операторы подготовки к выполнению цикла.

2. Выполнение оператора ***while*** начинается с вычисления значения выражения ***b*** (выражение ***b*** вычисляется перед каждой итерацией цикла). Производится анализ полученного значения ***b***:

а) если ***b*** истинно (!=0), то выполняется ***S***;

б) если ***b*** ложно (= =0), то тело цикла завершается;

в) если ***b*** ложно (= =0) до первого выполнения тела цикла, то тело цикла не выполняется ни разу.

3. После выполнения оператора ***S*** вычисляется значение выражения ***b*** и осуществляется переход к п. 2 данных правил.

4. После нормального завершения цикла значения параметров цикла равно значениям, которые привели к завершению цикла.

Цикл с предусловием ***while***, таким образом, реализуется по схеме, изображенной на рис. 4.1, *а*.

***Примеры*:**

1. while(true); – бесконечный цикл.

2. Какое значение *y* будет напечатано после выполнения операторов?

```
x=2;
while(x<0) y=x*3;
y=x-1;
printf("y=%d",y);
```

Условие $x<0$ в операторе цикла является ложным, поэтому цикл не выполнится ни разу. Значит, $y = 2 – 1 = 1$. Будет выведено значение 1.

3. Чему будет равно j после завершения цикла?

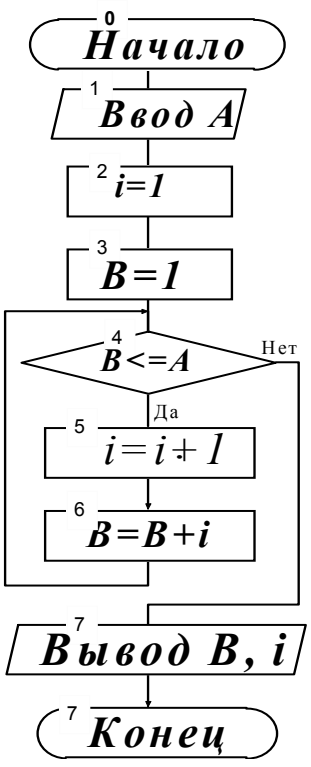
```
j=3; while(j<=7) j=j+2;
```

После завершения цикла j будет иметь значение 9.

4. Дано вещественное число $A > 0$. Числа B_i образуются по закону: $B_i = \sum_{k=1}^{i} k, i = 1, 2, \ldots$. Найти среди B_i первое число, большее A.

Решение. Очевидно, что $B_1 = 1$. Каждое следующее значение B_i может быть получено по рекуррентному соотношению $B_i = B_{i-1}+i$, где $i = 2, 3, \ldots$, что нетрудно проверить. В самом деле $B_2 = 1 + 2 = B_1 + 2 = = 1 + 2 = 3$; $B_3 = 1 + 2 + 3 = B_2 + 3 = 3 + 3 = 6$; $B_4 = 1 + 2 + 3 + 4 = B_3 + 4 = = 6 + 4 = 10$; и т. д. При этом заранее неизвестно, сколько раз соотношение $B_i = B_{i-1} + i$ будет выполнено, нет необходимости сохранять в памяти компьютера все полученные значения B_i. Поэтому тело цикла будет состоять в выполнении операторов $i = i + 1$ и $B = B + i$. Условие повторения – неравенство $B<=A$. Параметром цикла будет являться B. Подготовка данных для очередной итерации цикла будет заключаться в увеличении параметра B на величину i. Подготовка к выполнению цикла будет состоять в присвоении переменным B и i начальных значений: $B = 1$; $i = 1$. Таким образом, получим графическую схему, изображенную на рис. 4.3, *а*. Текст программы представлен на рис. 4.3, *б*. В программе для организации цикла использован оператор *while*, поскольку число итераций цикла заранее неизвестно.

Результаты выполнения программы для разных значений A:
$b = 6$ $i = 3$ для $a = 4.00$; $b = 21$ $i = 6$ для $a = 16.00$; $b = 45$ $i = 9$ для $a = 40.00$.



а)

```
#include <stdio.h>
main()
{
 float a;
 int i, b;
 puts("Введите число a>0");
 scanf("%f", &a);
 b=i=1;
 while(b<=a)
 {
  i++;  // i=i+1;
  b+=i; // b=b+i;
 }
 printf("b=%d i=%d для a=%.2f\n",b,i,a);
 fflush(stdin);
 getchar();
 return(0);
}
```

б)

Рис. 4.3. Решение примера 4:
а – графическая схема; *б* – текст программы

5. Табулирование функции.

Вычислить значение функции $y = x^2$ для x, изменяющегося от $x_н$ до $x_к$ с шагом Δx.

Решение:

Исходные данные:

– левая граница интервала $x_н$;

– правая граница интервала $x_к$;

– шаг табулирования Δx.

Результаты: таблица значений заданной функции на интервале $[x_н, x_к]$.

Таблица 4.1

Таблица соответствия переменных

Имя переменной в условии	Имя переменной в программе	Тип переменной	Комментарий
$x_н$	*x_n*	float	Левая граница интервала
$x_к$	*x_k*	float	Правая граница интервала
Δx	*h*	float	Шаг табулирования
y	*y*	float	Значение функции
x	*x*	float	Аргумент функции

Подготовка тестов

Для проверки правильности работы программы следует задать исходные данные, вычислить значение n – количество строк в таблице результатов по формуле $n = \left[\frac{x_k - x_n}{\Delta_x} \right] + 1$. Для одного из возможных значений *x* вычислить значение *y*. После выполнения программы сверить результаты работы с заранее найденным значением *y*, проверить правильность изменения *x* для заданного Δ_x и убедиться, что количество строк таблицы результатов совпадает со значением *n*.

Например, для $x_н = 2$, $x_к = 6$, $\Delta x = 1$ возможными значениями *x* будут значения 2, 3, 4, 5, 6. При этом $n = \left[\frac{6-2}{1} \right] + 1 = 5$. Найдем *y* для $x = 4$. Получим $y = 16$. После выполнения программы останется убедиться в том, что количество строк в таблице результатов совпадает с *n*, $y = 16$ при $x = 4$, а в столбце *x* значения есть 2, 3, 4, 5, 6.

Альтернативный способ проверки правильности выполнения программы – проведение аналогичных расчетов в MS Excel либо в математическом пакете.

На рис. 4.4 представлена графическая схема алгоритма решения задачи 5.

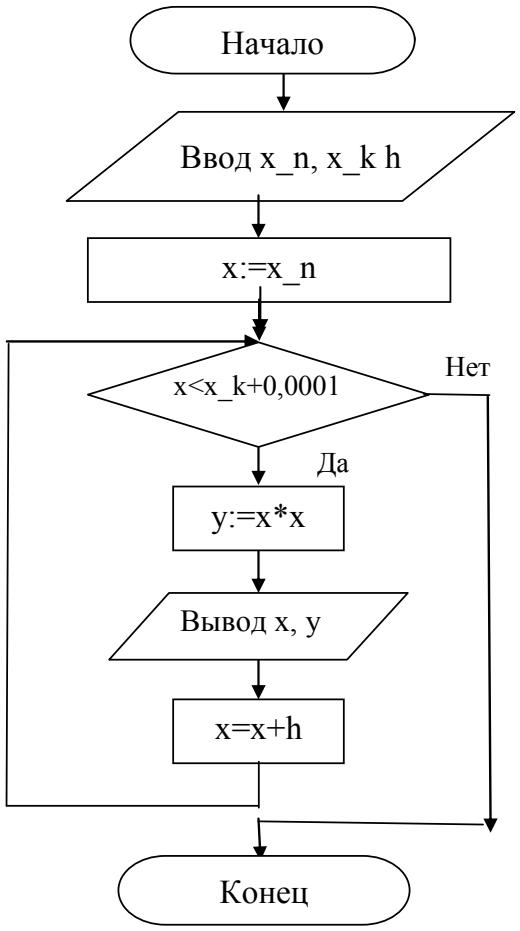


Рис. 4.4. Графическая схема алгоритма решения
задачи табулирования функции

Текст программы:

```
/*      Подключение стандартных библиотек */
#include<stdio.h>
#include<conio.h>
main()
{
    /*Описание используемых переменных*/
    float x_n,x_k,h,x,y;
    int i;      //Порядковый номер значения в таблице результатов
    /*Ввод исходных данных*/
    puts("Введите левую границу интервала");
    scanf("%f",&x_n);
    puts("Введите правую границу интервала");
    scanf("%f",&x_k);
    puts("Введите шаг табулирования");
    scanf("%f",&h);
```

```
    /* Формирование шапки таблицы   */
    puts(" ┌──┬──────────┬──────────────┐ ");
    puts(" │ № │    x     │      Y       │ ");
    puts(" ├──┼──────────┼──────────────┤ ");

    /*Расчет таблицы значений функции y=x*x
        на заданном интервале*/
    x=x_n; i=1;
    while (x<=x_k + 0.0001)
     {
      y=x*x;
      printf(" │%3d │%7.1f│%7.1f│\n", i, x, y);
      x=x+h ; i=i+1;
     }
     puts(" └──┴──────────┴──────────────┘ ");
}
```

6. Определить для заданного натурального числа *N*, составляют ли цифры этого числа неубывающую последовательность.

Решение. Рассмотрим, например, числа 113479 и 64801. Цифры первого числа (1, 1, 3, 4, 7, 9) составляют неубывающую последовательность, а цифры второго числа (6, 4, 8, 0, 1) – не составляют неубывающую последовательность.

Для ответа на поставленный вопрос задачи будем выделять цифры числа, начиная с младшей (правой) до старшей (левой) цифры. Если при этом для каждой пары соседних цифр выполняется неравенство C1>=C2, где C1 – младшая из цифр в паре, C2 – старшая из цифр в паре, то цифры заданного числа составляют неубывающую последовательность. Если для очередной пары цифр это неравенство не выполняется, то цифры заданного числа не составляют неубывающую последовательность. Например, для первого числа C1=9; C2=7. Так как 9>7, то рассматривается следующая пара C1=C2=7; C2=4. Так как 7>4, то рассматривается следующая пара C1=C2=4; C2=3. Так как 4>3, то рассматривается следующая пара C1=C2=3; C2=1. Так как 3>1, то рассматривается следующая пара, последняя пара, C1=C2=1; C2=1, для которой C1=C2. Для второго числа уже на второй итерации выявляется, что C1<C2 (0<8). Процесс следует остановить и сделать вывод о том, что цифры числа не составляют неубывающую последовательность.

Цифры числа будем определять как остаток от деления числа на 10. Младшая цифра – это N%10 (% – операция определения остатка от

деления целых чисел). Следующая цифра – это остаток от деления числа N1 на 10, где N1 – число, получаемое из N отбрасыванием его младшей цифры, и т. д. Процесс выделения цифр заканчивается, когда N1 становится равным нулю.

Если последовательность цифр числа не является неубывающей (не выполняется для очередной пары цифр неравенство C1>=C2), то процесс лучше всего прекратить. Для этой цели используется переменная *t*, принимающая значение 1, если не найдена пара цифр такая, что C1<C2, и 0 в противном случае.

Таким образом, описанный процесс является циклическим. Он будет иметь два параметра цикла ***N1*** и ***t***, в качестве условия повторения будет иметь сложное условие ***N1!=0 и t!=0*** (не закончились цифры и не установлено условие неупорядоченности цифр по не убыванию). Начальные значения параметров ***N1=N/10*** и ***t=1***.

В программе для организации цикла использован оператор ***while***, так как число повторений цикла заранее неизвестно.

Текст программы:

```
        include <stdio.h>
main()
{
        long N, N1;
        int C1, C2, t=1;
        puts("Введите N");
        scanf("%ld",&N);
        C1=(N%10);
        N1=N/10;
        while (N1!=0 && t!=0)
         {
           C2=(N1%10);
           N1=N1/10;
           if(C1>=C2) C1=C2;
            else t=0;
         }

        if(t==0) printf("N");
             else printf("Y");
        printf("N=%ld\n",N);
        fflush(stdin);
        getchar();
        return(0);
}
```

4.3.2. Оператор цикла *do-while*

Оператор цикла *do-while* является оператором цикла с постусловием, так как в нем анализ конца цикла производится после операторов тела цикла. Он используется, как и оператор *while*, когда количество итераций цикла заранее неизвестно и определяется в процессе выполнения цикла.

Особенностью оператора является выполнение тела цикла хотя бы один раз.

Общий вид оператора:

do S while (b);

Здесь *S* – простой или составной оператор – тело цикла. Он должен включать рабочую часть цикла и операторы изменения операндов выражения *b* (подготовки данных для очередного шага цикла); *b* – выражение любого типа, например, логическое, приводимое к арифметическому типу, определяющее условие повторения цикла.

Оператор цикла *do-while* выполняется по схеме цикла с постусловием, изображенной на рис. 4.1, *б*.

Пример. Найти сумму первых *N* членов ряда $\frac{x}{2}+\frac{x^3}{4}+\frac{x^5}{8}+\frac{x^7}{16}+\dots$.

Решение. Введем обозначения: *C* – сумма ряда, *U* – произвольный член ряда. Сумму вычислим в цикле как нарастающую сумму: *C* = *C* + *U*. Для вычисления значения очередного члена ряда достаточно значение предыдущего члена ряда умножить на $\frac{x^2}{2}$, т. е. $U = U\frac{x^2}{2}$.

Полученная формула называется рекуррентной. Она позволяет вычислить любой член ряда, если известен первый член ряда.

К моменту исполнения первой итерации цикла значения *C* и *U* должны быть определены: $C = 0;\ U = \frac{x}{2}$. Параметром цикла пусть будет переменная *k*. Параметр *k* должен изменяться от 1 до *N* с шагом 1. Выражение *C* = *C* + *U* должно быть выполнено хотя бы один раз (для *N* = 1 – один раз; для *N* = 2 – два раза и т. д.), поэтому для вычисления суммы следует использовать цикл с постусловием. Графическая схема алгоритма приведена на рис. 4.5.

Рис. 4.5. Графическая схема алгоритма решения задачи

4.4. Вложенные циклы

В теле цикла может располагаться еще один цикл. В этом случае говорят, что алгоритм или программа содержат цикл в цикле или вложенные циклы. Общий вид графической схемы цикла в цикле изображен на рис. 4.6. Здесь внешний цикл – цикл с предусловием, внутренний цикл – цикл с постусловием. Рабочая часть внешнего цикла выделена пунктирной линией. Число выполнений команд рабочей части внутреннего цикла может быть найдено как произведение числа повторений внутреннего цикла на число повторений внешнего цикла.

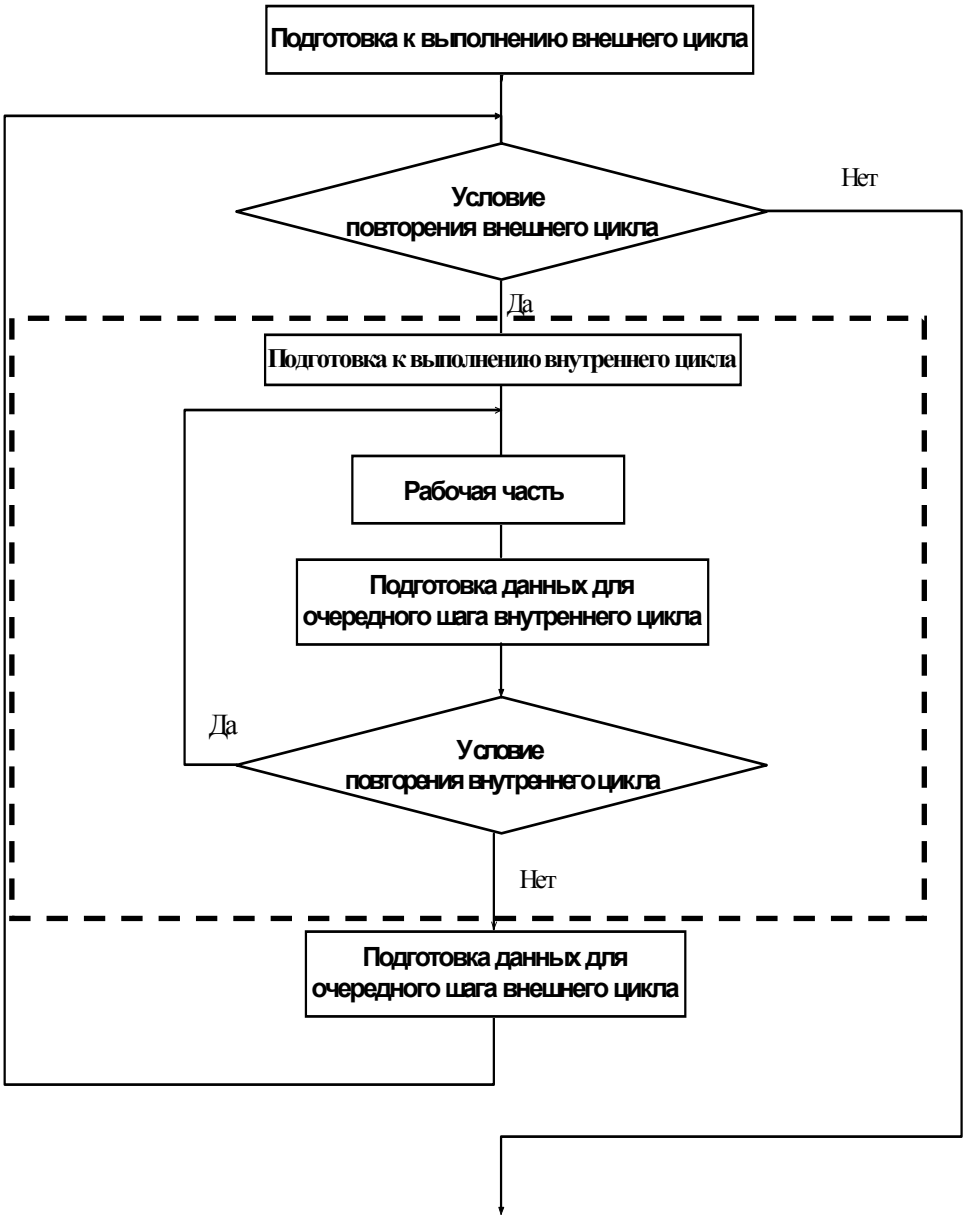


Рис. 4.6. Схема цикла с постусловием в цикле с предусловием

Кратность вложения циклов может равной не только двум, но и трем, четырем и т. д. Следующий пример демонстрирует трехкратное вложение циклов.

Пример. Определить количество трехзначных натуральных чисел, сумма цифр которых равна n.

Решение. Пусть l – искомое количество натуральных трехзначных чисел; i – старшая цифра числа; j – средняя цифра; k – младшая цифра. Тогда сумма цифр трехзначного числа есть $i + j + k$, где $i = 1, 2, \ldots, 9$; $j = 0, 1, 2, \ldots, 9$; $k = 0, 1, 2, \ldots, 9$ (старшая цифра трехзначного числа не может быть нулем). Начальное значение $l = 0$.

Для вычисления всех возможных сумм $i + j + k$ организуем вложенные циклы (кратность вложения равна трем) с предусловием с параметрами i, j, k соответственно. Например, цикл по k в цикле по j, цикл по j в цикле по i (т. е. цикл в цикле и еще раз в цикле). В самом внутреннем цикле значение l будем увеличивать на единицу, если выполнится условие $i + j + k = n$. Графическая схема изложен-ного алгоритма приведена на рис. 3.2. Рабочие части циклов выде-лены пунктирными линиями. Рабочей частью самого внутреннего цикла является ветвление с логическим блоком 9. Это ветвление выполнится 900 раз (10 · 10 · 9), что равно произведению числа повторений внутреннего, среднего и внешнего циклов. При этом для каждого из значений i = 1, 2, …, 9 j принимает значения 0, 1, 2, …, 9. k принимает значения 0, 1, 2, …, 9 для каждого значения j.

Графическая схема алгоритма представлена на рис. 4.7.

Текст программы:

```
#include <stdio.h>
#include <iostream.h>
main()
{
  int L,i,j,k,n;
  L=0;
  cout << "\nВведите целое число";
  cin >> n;
  for(i=1; i<=9; i++)
     for(j=0; j<=9; j++)
          for(k=0; k<=9; k++)
            if (i+j+k==n) {L=L+1; cout<< "\n» <<i<<j<<k;}
  cout<<"\nКоличество трехзначных чисел, сумма цифр которого равна"<<n<<" есть "<<L;
  cout << "\n Нажмите любую клавишу \n";
  fflush(stdin);
getchar();
return(0);
}
```

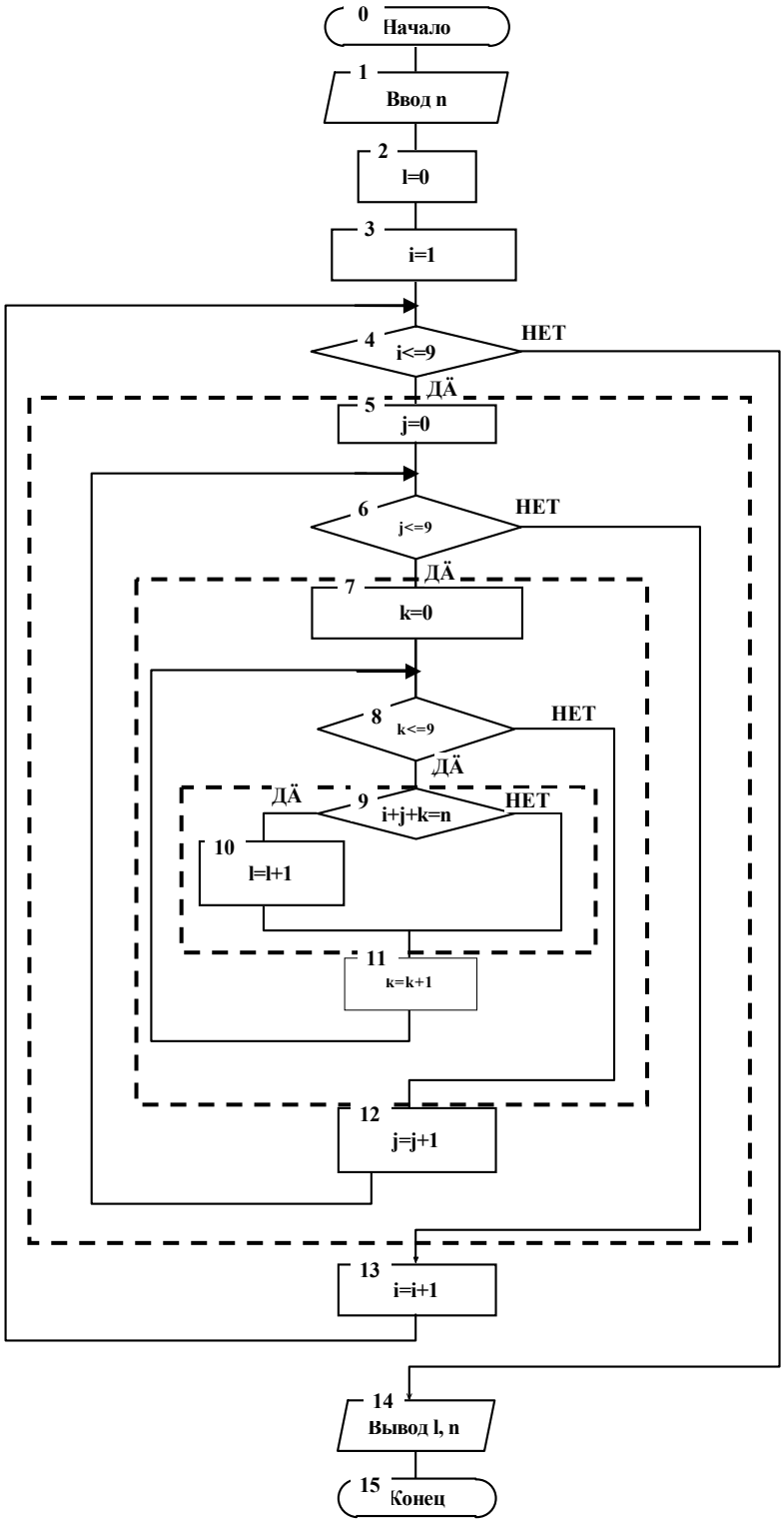


Рис. 4.7. Графическая схема алгоритма задачи о нахождении суммы *N* первых членов ряда

5. СТРОКИ И СИМВОЛЫ

5.1. Работа с символами

Для хранения отдельных символов используются переменные типа *char*. Пример описания:

char t, p; // Описаны 2 переменные символьного типа.

Возможными значениями символьной переменной могут быть различные символы. Такими символами могут быть почти все знаки из кодовой таблицы ПК за исключением специальных управляющих знаков, не имеющих графического изображения. Например, можно записать присваивания: *t*='*Q*'; *p*='*p*'. Здесь *t* и *p* – переменные, которые могут принимать разные значения, а '*Q*' и '*p*' – конкретные значения, присваиваемые этим переменным. После этого в ячейке памяти, которая зарезервирована для переменной *t*, будет записан код символа '*Q*', а для переменной *p* – код символа '*p*'. *Переменная p и значение 'p' – это не одно и то же!*

Код символа – это целое число. Занимает в памяти 1 байт. Коды всех символов сведены в кодовую таблицу, имеющую 16 строк и 16 столбцов, пронумерованных 16-ми цифрами: 0, 1, …, 9, A, B, C, D, E, F. Таким образом, кодовая таблица хранит коды 256 символов. Из кодовой таблицы можно определить код символа в 16-й системе счисления – это последовательность 16-х цифр, составленная из номера столбца и номера строки. Например, символ, стоящий на пересечении столбца 9 и строки F, имеет 16-й код 9*F*16. Можно перевести код в 10-ю систему счисления: 9*F*16 = 9 · 16 + 15 = 15910.

Существуют специальные программы вывода таблицы кодов символов на экран и вставки символов в текст программы. Например, ascii.com.

Хотя в языке Си можно одновременно использовать переменные типов *int* и *char*, все же отметим, что если объявлено *int a; char b;* и назначено *a=5; b='5';*, то *a* и *b* – это не одно и то же: переменная *a* имеет числовое значение *5*, а *b* – значение символа *'5'*. Число *5* в двоичном коде равно *101*, а двоичный код символа *'5'* есть *110101* ($110101_2 = 35_{16} = 53_{10}$).

Пример. Ввод значения символьной переменной, вывод введенного значения в виде символа и десятичного, восьмеричного и шестнадцатеричного кодов.

Текст программы:

```
/* Вывод символьной переменной в различных формах*/
#include<stdio.h>
#include<conio.h>
main()
{
 char b;                    //Объявление символьной переменной
 puts("Введите один символ");
 b=getchar();                //Ввод значения переменной b
 printf("\tb=");
 putchar(b);                 //Вывод значения переменной b
 printf("\nДесятичный код b=%d\n",b); //Вывод десятичного кода b
 printf("\tВосьмеричный код b=%o\n",b); //Вывод восьмерич. кода b
 printf("\t\tШестнадцатеричный код b=%X\n",b);
 printf("b='%c'\n",b);
 getch();
}
```

Результаты выполнения программы:

```
Введите один символ
*
        b=*
Десятичный код b=42
        Восьмеричный код b=52
                Шестнадцатеричный код b=2A
b='*'

Введите один символ
{
        b={
Десятичный код b=123
        Восьмеричный код b=173
                Шестнадцатеричный код b=7B
b='{'
Введите один символ
#
        b=#
Десятичный код b=35
        Восьмеричный код b=43
                Шестнадцатеричный код b=23
```

```
b='#'

Введите один символ
5
      b=5
Десятичный код b=53
      Восьмеричный код b=65
            Шестнадцатеричный код b=35
b='5'
```

В библиотеке *ctype* определен целый ряд функций, проверяющих принадлежность символа какому-либо множеству: множеству букв (*isalpha*), букв или цифр (*isalnum*), разделителей (*isspace*), знаков пунктуации (*ispunct*), цифр (*isdigit*), видимых символов (*isgraph*), букв верхнего регистра (*isuper*), букв нижнего регистра (*islower*), печатаемых символов (*isprint*) и т. д.

Аргументом каждой из этих функций является символ. Функция возвращает значение *true*, если символ принадлежит конкретному множеству символов, или *false* в противном случае.

Пример. Ввести символ с клавиатуры и проверить, является ли он знаком пунктуации (. , ! ? : ;).

Текст программы:

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
void main()
{
 char a;
 puts("Введите один символ");
 a=getchar();
 if(ispunct(a))
  printf("Символ %c явл. знаком пунктуации\n",a);
 else
  printf("Символ %c не явл. знаком пунктуации\n",a);
 getch();
}
```

Результаты выполнений программы:

```
Введите один символ
.
Символ . явл. знаком пунктуации
Введите один символ
^
Символ ^ явл. знаком пунктуации
Введите один символ
e
Символ e не явл. знаком пунктуации
Введите один символ
#
Символ # явл. знаком пунктуации
Введите один символ
[
Символ [ явл. знаком пунктуации
Введите один символ
-
Символ - явл. знаком пунктуации
Введите один символ
n
Символ n не явл. знаком пунктуации
```

Следующий *пример* показывает, что символьная переменная может быть определена символом или кодом символа.

```
#include<stdio.h>
void main()
{
  char c,c1;
  c='A';
  c1=65;
  printf("%c %d\n",c,c);
  printf("%i %c\n",c1,c1);
  if(c1=='A' && c== 65)
     puts("Оба символа A");
   else puts("****");
}
```

Результат выполнения программы:
A 65
65 A
Оба символа A

Пример использования функции *char* из библиотеки *ctype*

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
  int cod;
  char S;
  puts("Введите 1 символ");
  S=getchar();
  cod=char(S);
  putchar(S);
  printf("\t\n");
  printf("%c %d %d\n",S,S,cod);
  getch();
}
```

Результаты выполнения программы:
Введите 1 символ
2
2
2 50 50

Введите 1 символ
%
%
% 37 37

Введите 1 символ
D
D
D 68 68

5.2. Понятие и описание строки

В языке Си отсутствует специальный строковый тип. *Строка в Си – это массив символов.* В памяти ЭВМ строка представляется как массив элементов типа *char*, в конце которого помещается символ '\0' – *завершающий нуль-символ*. Это не цифра 0, а символ, у которого в кодовой таблице все нули (т. е. 00000000).

Строковая константа – это последовательность символов, заключенная в кавычки ("). Например, "Задача имеет множество решений». В памяти представляется как массив элементов типа *char* с завершающим нуль-символом ('\0').

Общий вид описания строки:

1. *char* <имя строки>**[**<длина строки с учетом нуль-символа>**]**;

Например, char S[100];

или const int N=100;

char S[N]; //Удобно задавать длину строки с

//помощью именованной константы

В такой строке можно хранить 99 символов и завершающий нуль-символ.

2. При описании строки можно выполнять ее инициализацию. Например,

char S1[100]= "Исходный массив";

При этом нуль-символ автоматически формируется за последним символом символьной константы.

Если строка при описании инициализируется, то можно опускать длину строки. *Например*,

char S1[]= "Исходный массив"; //16 символов.

Примечание. Имя строки, как и любого массива – это указатель-константа. Поэтому ошибкой будет попытка использовать имя строки в некоторых операциях адресной арифметики. Например, нельзя выполнить такую «пересылку» строки символов в массив:

```
char a[20];                    //Так
a= "Строковая константа";      //нельзя
```

5.3. Описание динамической строки

Для размещения строки в динамической памяти необходимо описать указатель на Char, а затем выделить память с помощью *new* или *malloc* (*new* предпочтительнее).

Например,

*char *S3=new char[m]; //m* должно быть уже определено.

Или

*char *S3;*

S3=new char[m];

Примечание. Динамические строки, как и другие массивы, нельзя инициализировать.

Например, оператор,

char *S4= "На нуль делить нельзя";

Создает не строковую переменную, а указатель на строковую константу, изменить которую невозможно. S4 – адрес первого символа строковой константы.

5.4. Ввод-вывод строк

Рассмотрим ввод-вывод строк с помощью функций, унаследованных из библиотеки Си: *gets* и *puts*, *scanf* и *printf*.

Функции *gets* и *puts* используются, если работа производится только со строками. Функции *scanf* и *printf* удобнее использовать в том случае, если в одном операторе требуется ввести или вывести данные разных типов.

Функция *gets* предназначена для ввода строки. Имеет один параметр, задающий адрес области памяти, в которую помещаются символы вводимой строки. В языке Си имя переменной, имеющей строковый тип, является этим адресом. Обращение имеет вид: *gets*(*name*), где *name* – переменная строкового типа – имя вводимой строки. Выполняя эту функцию, программа приостанавливает свою работу и ждет от пользователя ввода последовательности символов и/или нажатия клавиши Enter. Символ новой строки в строку не включается, вместо него в строку заносится нуль-символ. Функция возвращает указатель на строку *s*, а в случае возникновения ошибки или конца файла – *NULL*.

Функция *puts* предназначена для вывода строки на стандартное устройство вывода. Имеет один параметр, задающий адрес области памяти, из которой на экран выводятся символы. Как уже отмечалось,

имя переменной, имеющей строковый тип, является этим адресом. Обращение имеет вид: *puts*(*name*), где *name* – переменная-строка – имя выводимой строки или строка символов, заключенная в кавычки. После вывода строки курсор перемещается к началу новой строки экрана, т. е. завершающий нуль-символ строки заменяется на символ новой строки. Возвращает неотрицательное значение при успехе или *EOF* при ошибке.

Функция *printf* предназначена для вывода форматированной последовательности данных.

Функция *scanf* предназначена для ввода данных в заданном формате. Обращение имеет вид:

scanf(nf,&a1,&a2,...)

Здесь *nf* – форматная строка; *&a1*,*&a2*,... – список ввода – указатели на значения вводимых переменных *a1*, *a2*, ...

Подробно эти функции рассматривались в разделе «Консольный ввод-вывод».

5.5. Операции над строками

5.5.1. Реализация операции присваивания

Поскольку строка является массивом, а не специальным типом данных, то для строк не определена операция присваивания. Присваивание можно выполнить посимвольно, т. е. «вручную», или с помощью стандартных функций *strcpy* и *strncpy*.

Рассмотрим первый способ. *Пример*:

s[0]='В'; s[1]='в'; s[2]='о'; s[3]='д';

Рассмотрим второй способ. Для использования функций *strcpy* и *strncpy* к программе следует подключить заголовочный файл <*string.h*> предложением #include <*string.h*>

Обращение к функции *strcpy* имеет вид:

strcpy(s1,s2);

Функция копирует все символы строки *s2*, включая завершающий нуль-символ, в строку *s1* и возвращает *s1*.

Пример 1

```
/* Копирование строки s2 в строку s1 с помощью strcpy  */

#include<stdio.h>
#include<string.h>
```

```
#include<conio.h>
void main()
{
 int n;                    // Длина строки
 char s2[20]= "Скоро сессия!";  //Исходная строка
 n=strlen(s2)+1;           //Длина s1
 char *s1=new char[n];     //Строка - копия s1
 // char s1[20];
 strcpy(s1,s2);            //Копирование s2 в s1
 puts(s2);                 // Вывод s2
 puts(s1);                 //Вывод s1
 puts(strcpy(s1,s2));      //strcpy возвращает s1
 getch();
}
```

Результат выполнения программы – три строки:

Скоро сессия!
Скоро сессия!
Скоро сессия!

Обращение к функции *strncpy* имеет вид:

strncpy(s1,s2,n);

Функция копирует не более *n* символов из строки *s2* в строку *s1* и возвращает *s1*. При этом, если длина исходной строки (*s2*) превышает или равна *n*, нуль-символ в конец строки *s1* не добавляется. В противном случае строка *s1* дополняется нуль-символами до *n*-го символа. Если строки перекрываются, поведение не определено.

Пример 2

```
/* Копирование строки s2 в строку s1 с помощью strncpy  */
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
 int n;                    // Длина строки
 char s2[20]=«Скоро сессия!»;    //Исходная строка
 n=strlen(s2)+1;           //Длина s1
 //char *s1=new char[n];     //Строка – копия s1
 char s1[20];
 strncpy(s1,s2,n);         //Копирование s2 в s1
```

```
  puts(s2);                  // Вывод s2
  puts(s1);                  //Вывод s1
  puts(strncpy(s1,s2,n));        //strncpy возвращает s1
  getch();
}
```

Результат выполнения программы такой же, как и в примере 1.

Примечание 1. В рассмотренных примерах использована функция *strlen(s)*, возвращающая фактическую длину строки *s*, не включая нуль-символ.

Примечание 2. Программист должен сам заботиться о том, чтобы в строке-приемнике (s1) хватило места для строки-источника (s2), и о том, чтобы строка всегда имела нуль-символ.

Выход за пределы строки и отсутствие нуль-символа являются распространенными причинами ошибок в программах обработки строк.

5.5.2. Преобразование строки в число

Преобразование строк в числа можно выполнить с помощью функций *atoi*, *atol*, *atof*. Обратные преобразования – с помощью функции *sprintf*.

Функция *atoi(s)* преобразует строку, содержащую символьное представление целого числа в соответствующее целое число. Признаком конца числа служит первый символ строки, который не может быть интерпретирован как принадлежащий целому числу. Если преобразование не удалось, возвращает 0.

Функция *atol(s)* преобразует строку, содержащую символьное представление длинного целого числа, в соответствующее целое число.

Функция *atof(s)* преобразует строку, содержащую символьное представление вещественного числа, в соответствующее вещественное число двойной точности.

Для использования функций *atoi(s)*, *atol(s)* и *atof(s)* к программе следует подключить заголовочный файл <*stdlib.h*> предложением #include <*stlib.h*>

Пример. Данные об участнике соревнований (номер участника, рост и вес) содержатся в строке символов. Вывести номер, рост и вес.

```
/* Преобразование строки в число */
#include<stdio.h>
```

```
#include<string.h>
#include<stdlib.h>
#include<conio.h>
void main()
{
 char s[]="10) Рост – 162 см, вес – 63.4 кг";     //Строка
 int n;                    // Целое число
 long h;                   //Длинное целое
 double w;                 //Вещественное число
 n=atoi(s);                //Преобразование в целое
 h=atol(&s[11]);           //Преобразование в длинное целое
 w=atof(&s[26]);                  //Преобразование в вещественное
число
 printf("Участник # %d Его рост %ld и вес %lf\n",n,h,w);
 puts("Исходная строка");
 puts(s);
 puts("Первая подстрока");
 puts(&s[11]);
 puts("Вторая подстрока");
 puts(&s[26]);
 getch();
}
```

Результат выполнения программы:

```
Участник # 10 Его рост 162 и вес 63.400000
Исходная строка
10) Рост – 162 см, вес – 63.4 кг
Первая подстрока
162 см, вес – 63.4 кг
Вторая подстрока
63.4 кг
```

Заметим, что строка определяется адресом ее нулевого символа. Для всей строки *s* таким адресом является имя строки *s*. Число 10 в символьном представлении находится в строке *s* в самом ее начале. Поэтому аргументом функции *atoi* является строка *s*. Число 162 в символьном представлении находится в строке *s*, начиная с позиции 11. Поэтому аргументом функции *atol* является подстрока строки *s*, определяемая адресом &s [11]. Число 63.4 в символьном представлении на-

ходится в строке *s*, начиная с позиции 26. Поэтому аргументом функции *atof* является подстрока строки *s*, определяемая адресом &s [26].

5.5.3. Поиск подстроки в строке

Функция *strstr(s1,s2)* выполняет поиск подстроки *s2* в строке *s1* (первого вхождения подстроки *s2* в строку *s1*). Обе строки должны завершаться нуль-символами. В случае успешного поиска функция возвращает указатель на найденную подстроку. В случае неудачи – *NULL*.

Пример. Определить, содержится ли строка ***s2*** в строке ***s1*** в качестве подстроки.

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
void main()
{
 const int n=81;
 char s1[n],s2[n];
 char *p;
 clrscr();
 puts("Введите строку s1?");
 gets(s1);
 puts("Введите строку s2?");
 gets(s2);
 p=strstr(s1,s2);
 if(p)
  { printf("Подстрока '%s'\n начинается в строке: \n %s\n",s2,s1);
      printf("символом '%c'\n этим символом начинается
               подстрока: %s\n",*p,p);
  }
   else puts("NO");
 getch();
}
```

Заметим, что ****p*** – символ в строке ***s1***, с которого начинается ***s2***;
p – подстрока, начиная с ****p*** до конца строки ***s1***.

Результат выполнения программы:

```
Введите строку s1?
Скоро ли сессия? Так хочется сдать экзамен по ОАиП!
Введите строку s2?
чет
Подстрока 'чет'
```

начинается в строке:
Скоро ли сессия? Так хочется сдать экзамен по ОАиП!
символом 'ч'
этим символом начинается подстрока:
чется сдать экзамен по ОАиП!

5.5.4. Сцепление двух строк (конкатенация)

Функция *strcat(s1,s2)* присоединяет строку *s2* в конец строки *s1* и возвращает указатель на строку, совпадающий с первым аргументом. При этом сначала из строки *s1* удаляется завершающий нуль-символ. В конце новой строки *S1* помещается *'\0'*.

Программист должен сам обеспечить достаточную длину строки s1.

***Примеры*:**

1) char s1[40]= "Скоро Новый Год!";
   char s2[]="Скоро первая сессия!";
   printf("%s\n",strcat(s1,s2));

Будет получена строка:
Скоро Новый Год! Скоро первая сессия!

2) Вторым параметром является текстовая константа:
   char s3[50]=«Успешной сдачи»;
   printf("%s\n\t%s\n",strcat (s3, "1-ой сессии!!!"),s3);

   Будет получено две строки:
   Успешной сдачи 1-ой сессии!!!
   Успешной сдачи 1-ой сессии!!!

3) char s4[35]="Успехов и здоровья";
   char s5[]="в Новом Году!";
   char *p;
   p=strcat(s4, s5);
   pintf("%s\n%s\n",s4,p);

   Будет получено две строки:
   Успехов и здоровья в Новом Году!
   Успехов и здоровья в Новом Году!

Функция *strncat(s1,s2,n)* присоединяет *n* символов строки, на начало которой указывает *s2*, в конец строки, на начало которой

указывает *s1* и возвращает указатель на строку, совпадающий с первым аргументом. Сформированная строка *s1* ограничивается *'\0'*.

Если *n* больше длины строки *s2*, то выполняется простая конкатенация.

Если длина строки *s2* меньше *n*, то все символы *s2* присоединяются к строке *s1*.

5.5.5. Определение позиции первого вхождения символа из заданного набора символов

Функция *strcspn(s1,s2)* сопоставляет каждый символ строки *s1* со всеми символами строки, на начало которой указывает *s2*, и возвращает позицию первого вхождения символа строки *s2* в строке *s1*. Символ '/0' в сравнении не участвует.

Если строка *s1* начинается с символа, встречающегося в строке *s2*, то функция возвращает значение нуль. Если строка *s1* не содержит ни одного символа строки *s2*, то возвращаемое функцией значение совпадает с длиной строки *s1*.

***Примеры*:**

```
printf("%d %d %d\n",
strcspn("asdf",  « "hjyars"),    //s1 начинается с 'a'
strcspn("asdf", "ghtdu"),    //в s1 'd' в позиции № 2
strcspn("asrt", "hj"));        // в s1 не символов s2
```

Получим:

```
0 2 4
```

Здесь 0 и 2 – позиции в строке s1, 4 – длина строки s1.

5.5.6. Сравнение двух строк

Сравнение строк производится посимвольно слева направо. Большей считается та строка, в которой первый несовпадающий символ имеет больший код в кодовой таблице.

Функция *strcmp(s1,s2)* сравнивает строки *s1* и *s2*. Возвращает отрицательное значение, если *s1<s2*, нулевое, если *s1=s2* или положительное значение, если *s1>s2*.

Функция *strncmp(s1,s2,n)* сравнивает строку *s1* и первые *n* символов строки *s2*. Возвращает отрицательное значение, если *s1< чем первые n символов s2*, нулевое, если *s1=первым n символам s2* или положительное значение, если *s1> чем первые n символов s2*.

Пример. Ввести две строки. Вывести их в лексикографическом порядке.

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
  char s1[20];
  char s2[20];
  puts("Введите 1-ую строку");
  gets(s1);
  puts("Введите 2-ую строку");
  gets(s2);
  if(strcmp(s1,s2)<0)
      {
      puts(s1);
      puts(s2);
      }
   else
    if(strcmp(s1,s2)>0)
      { puts(s2);
        puts(s1);
      }
     else puts("Строки совпадают");
  getch();
}
```

Результаты выполнения программы:

Введите 1-ую строку
abcdefg
Введите 2-ую строку
abcdefg
Строки совпадают

Введите 1-ую строку
bcdert
Введите 2-ую строку
bccertyu
bc**c**ertyu
bc**d**ert

```
Введите 1-ую строку
rtyu
Введите 2-ую строку
rtdh
rtdh
rtyu
```

Пример. Сколько раз подстрока s1 содержится в строке s?

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
 char s[20];
 char s1[20];
 puts("Введите строку");
 gets(s);
 puts("Введите подстроку");
 gets(s1);
 int k=0,i=0;
 char *p=s;
 int n=strlen(s1);
 printf("n=%d %c\n",n,*p);
 while(strstr(p,s1) && i<n)
  {
   k=k+1;
   p=p+n+1; printf("%c\n",*p);
  }
 puts("В строке");
 puts(s);
 printf("%d раз встречается %s\n",k,s1);
 getch();

}
```

Результат:

мама мыла, мама стирала, мама варила, будила студента – строка *s*

мама – подстрока *s*1

Эта же задача с использованием *srtcmp:*

```
/*Сколько раз подстрока s1 встречается в строке s? */
```

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
 char s[20];
 char s1[20];
 puts("Введите строку");  gets(s);
 puts("Введите подстроку");  gets(s1);
 int k=0,i=0;
 int m=strlen(s);
 int n=strlen(s1);
 printf("m=%d n=%d\n",m,n);
 while(i<m)
  {
   if(strncmp(s1,&s[i],n)==0)
        {
         k=k+1; i=i+n;
        }
       else i=i+1;
  }
 puts("В строке: ");
 puts(s);
 printf("%d раз(а) встречается %s\n",k,s1);
 getch();
}
```

5.5.7. Примеры решения задач

```
/*Сколько раз слово s1 встречается в строке s? */
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<ctype.h>
void main()
{
 char s[20];
 char s1[20];
 puts("Введите строку");  gets(s);
 puts("Введите слово");  gets(s1);
```

```
  int k=0,i=0;
  int m=strlen(s);
  int n=strlen(s1);
  printf("m=%d n=%d\n",m,n);
  while(i<m)
   {
    if(strncmp(s1,&s[i],n)==0)
        { if(i==0)
           {
            if(isspace(s[i+n]) || ispunct(s[i+n]) || (s[i+n]=='\0'))
              { k++; i=i+n; }
              else i++;
           }
          else
           if((isspace(s[i-1])||ispunct(s[i-1])) &&
            (isspace(s[i+n])|| ispunct(s[i+n]))||(s[i+n]=='\0'))
            { k++; i=i+n; }
           else i++;
          }
    else i++;
   }
  puts("В строке: ");
  puts(s);
  printf("%d раз(а) встречается %s\n",k,s1);
  getch();
}
/*Сколько раз слово s1 встречается в строке s? */
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<ctype.h>
void main()
{
  char s[20];
  char s1[20];
  puts("Введите строку");  gets(s);
  puts("Введите слово");  gets(s1);
  int k=0,i=0;
  int m=strlen(s);
```

```
  int n=strlen(s1);
  printf("m=%d n=%d\n",m,n);
  //Добавление пробела в начало слова
  for(i=m;i>=0;i--)
    s[i+1]=s[i];
  s[0]=' ';
  puts(s);
  //Поиск в строке
  while(i<m+1)
     if(strncmp(s1,&s[i],n)==0)
          {
              if((isspace(s[i+n]) || ispunct(s[i+n]) || (s[i+n]=='\0')) &&
              (isspace(s[i-1])||ispunct(s[i-1])))
                { k++; i=i+n; }
                else i++;
          }
     else i++;
  //Удаление из строки символа s[0]
  for(i=1;i<=m+1;i++)
     s[i-1]=s[i];
  puts("В строке: ");
  puts(s);
  printf("%d раз(а) встречается %s\n",k,s1);
  getch();
}
/*Сколько букв в третьем слове строки s? */
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<ctype.h>
void main()
{
  char s[80];
  int k,k3,i,n;
  puts("Введите строк");  gets(s);
  n=strlen(s);
  k=0;//Кол. слов
  //Поиск в строке
  i=0; //Номер символа
```

```
  while(i<n && k!=3)
    {
     if(isalpha(s[i]))   //Слово началось
         {
          k++;
          k3=0; //Кол. букв в слове
          while(isalpha(s[i]))
            {
              k3++; i++;
            }
         }
     i++;
    }
  if(k==3)
    {
     printf("В третьем слове строки: ");
     puts(s);
     printf(" %d символ(ов,а) \n",k3);
    }
   else {
          puts("В строке: ");
          puts(s);
          puts("меньше трех слов");
         }
  getch();
}
/*Вставить пробел в начало строки (с использ. указателей) */
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<ctype.h>
void main()
{
  char *s,*p;
  puts("Введите строку");  gets(s);
  int m=strlen(s);
  p=s-1;
  *p=' ';
  puts(s);
  puts(p);
```

```
  s=p;
  puts(s);
  getch();
}
/*Вставить пробел перед'.' */
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<ctype.h>
void main()
{
  char *s,*p,*c;
  puts("Введите строку");  gets(s);
  puts(s);
  int m=strlen(s);
  p=strchr(s,'.');//Позиция .
  puts(p);
  strncpy(c,s,p-s); //Подстрока до .
  puts(c);
  strcat(c,» «);
  strcat(c,p);
  s=c;
  puts(s);
  puts(p);
  puts(c);
  getch();
}
```

6. АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ ЗАДАЧ, СВОДЯЩИХСЯ К ОБРАБОТКЕ МАССИВОВ

6.1. Понятие массива

Массив – это совокупность элементов одного типа, имеющих одно имя и расположенных в памяти ПК вплотную друг к другу. Массивы могут состоять из арифметических данных, символов, строк, структур, указателей. Доступ к отдельным элементам массива осуществляется по имени массива и индексу (порядковому номеру) элемента.

При объявлении массива в программе определяется *имя массива*, *тип его элементов*, *размерность* и *размер*. *Размерность* или количество измерений массива определяется количеством индексов при обращении к элементам массива. Массивы бывают одномерные, двухмерные, трехмерные и т. д. *Размер массива* – это количество его элементов по соответствующим размерностям. Общий вид объявления массива:

<имя_типа> <имя_массива> [k1] [k2] ... [kn];

где *k1*, *k2*, ..., *kn* – количество элементов массива – константы или константные выражения по *1*, *2*, …, *n* измерениям. Причем значения индексов могут изменяться от *0* до *ki – 1*.

Такое объявление массива называют *статическим*, поскольку предельное количество его элементов известно заранее и оно уже не может быть изменено в ходе выполнения программы. При работе с массивами необходимо следовать следующим правилам:

– современные трансляторы языка Си не контролируют допустимость значений индексов, это должен делать программист;

– количество измерений массива не ограничено;

– в памяти элементы массива располагаются так, что при переходе от элемента к элементу наиболее быстро меняется самый правый индекс массива, т. е. матрица, например, располагается в памяти по строкам;

– имя массива является указателем – константой на первый элемент массива;

– операций над массивами в Си нет, поэтому пересылка элементов одного массива в другой может быть реализована только поэлементно с помощью цикла;

– над элементами массива допускаются те же операции, что и над простыми переменными того же типа;

– ввод/вывод значений элементов массива можно производить только поэлементно;

– начальные значения элементам массива можно присвоить при объявлении массива.

Примеры объявления массивов:

```
int   A [10];     //одномерный массив из 10 целочисленных величин
float   X [20];   //одномерный массив из 20 вещественных величин
int  a[5]={1, 2, 3, 4, 5};  //массив с инициализацией его элементов
int  c[]={–1, 2, 0, –4, 5, –3, –5, –6, 1}; // массив, размерность которого определяется числом инициализирующих элементов
```

Обращения к элементам одномерного массива могут иметь вид: *A[0]*, *A[1]*, *A[2]*, *…A[9]*, *A[2*3]*.

В Си нет массивов с переменными границами. Но, если количество элементов массива известно до выполнения программы, можно определить его как константу с помощью директивы *#define*, а затем использовать ее в качестве границы массива, например,

```
#define  n  10;
main  ( )
{ int a[n], b[n];     // Объявление 2-х одномерных массивов
```

Если количество элементов массива определяется в процессе выполнения программы, используют *динамическое* выделение оперативной памяти компьютера.

6.2. Динамические массивы

Если до начала работы программы неизвестно, сколько в массиве элементов, в программе используют динамические массивы. Память под них выделяется с помощью оператора *new* во время выполнения программы. Адрес начала массива хранится в переменной, называемой *указателем*. Например:

```
int  n=20;
int  *a = new int[n];
```

Здесь описан указатель *a* на целую величину, которому присваивается адрес начала непрерывной области динамической памяти, выделенной с помощью оператора *new*. Выделяется столько памяти,

сколько необходимо для хранения *n* величин типа *int*. Величина *n* может быть переменной.

Примечание: Обнуление памяти при ее выделении не происходит. Инициализировать динамический массив нельзя.

Обращение к элементу динамического массива осуществляется так же, как и к элементам обычного массива. Например:

a*[0], a[1], ..., a[9]*.

Можно обратиться к элементу массива другим способом:

*(*a*+1), …, *(*a*+9), *(*a*+*i*),

так как в переменной – указателе *a* хранится адрес начала массива. Для получения адреса, например, 9-го его элемента к этому адресу прибавляется 9·*sizeof(int)* (9 умножить на длину элемента типа *int*), т. е. к начальному адресу *a* прибавляется смещение 9. Затем с помощью операции *(разадресации) выполняется выборка значения из указанной области памяти.

После использования массива выделенная динамическая память освобождается с помощью оператора:

delete [ ] имя массива.

Например, для одномерного массива *a*:

delete [ ] a.

Время «жизни» динамического массива определяется с момента выделения динамической памяти до момента ее освобождения.

6.3. Основные алгоритмы обработки одномерных массивов

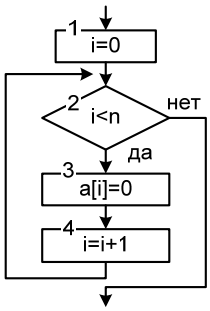
6.3.1. Инициализация массива

Инициализация массива – это присваивание элементам массива начальных значений. Инициализацию массива можно выполнить на этапе описания массива. Но в том случае, когда начальные значения получают лишь некоторые элементы массива, а остальные вычисляются в процессе выполнения программы, в программе записывают операторы присваивания.

Например: a[0]= –1; a[1]=1.1;

Присваивание всем элементам массива одного и того же значения осуществляется в цикле. Например, чтобы всем элементам масси-

ва ***a*** присвоить значение 0, можно воспользоваться алгоритмом, изображенном на рис. 6.1.



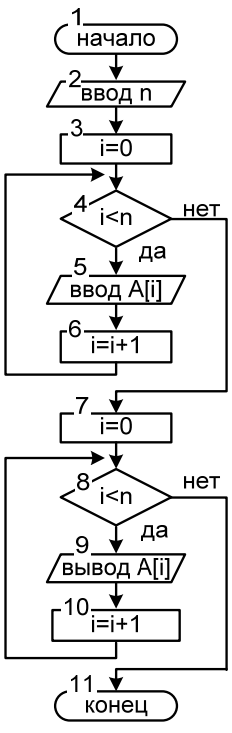

```
    for(i=0;i<n;i++)
       a[i]=0;
// или с помощью цикла while
    i=0;
    while (i<n)
    {
        a[i]=0;
         i=i+1;
    }
```

Рис. 6.1. Алгоритм и фрагмент программы инициализации массива

В представленном алгоритме все элементы массива в цикле последовательно инициализируются значением 0.

6.3.2. Ввод-вывод одномерного массива

Для ввода ***n*** элементов одномерного массива, назовем его ***A***, требуется организовать цикл, для ввода каждого ***i***-го элемента, где $i = 0, 1, 2, \ldots, n–1$. Аналогичный цикл требуется организовать и для вывода элементов массива. На рис. 6.2 изображена графическая схема ввода и вывода элементов массива.




```
/* Ввод–вывод статического массива*/
#include <stdio.h>
#define n 50;
 void main()
{
  int n,i;
  float A[n];
  puts("Введите число элементов массива ");
  scanf("%d",&n);
// Ввод массива
  for (i=0; i<n; i++)
    {  printf("Введите число A[%2d]= ",i);
       scanf("%f",&A[i]);
    }
  // Вывод массива
  puts("Массив A");
  for(i=0;i<n;i++)
    printf("%6.3f ",A[i]);
  printf("\n");
}
```

Рис. 6.2. Алгоритм и программа ввода-вывода статического массива

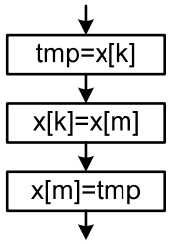
Ввод-вывод динамического массива осуществляется по тому же алгоритму. Из приведенного ниже примера программы ввода и вывода динамического массива видно, что отличие заключается лишь в описании массива.

```
/* Ввод – вывод динамического  массива*/
#include <stdio.h>
void main()
{
  int n,i;
  puts("Введите число элементов массива a");
  scanf("%d",&n);
  float *a=new float[n]; // Описание динамического массива
// Ввод массива
  for (i=0;i<n;i++)
    { printf("Введите число a[%2d]= ",i);
      scanf("%f",a+i);      // или  scanf("%f",&a[i]);
    }
// Вывод массива
  puts("Массив a");
  for(i=0;i<n;i++)
    printf("%.3f ",*(a+i)); // или   printf("%.3f ",a[i]);
  printf("\n");
  delete[] a;      // Освобождение памяти выделенной под массив
}
```

6.3.3. Перестановка двух элементов массива

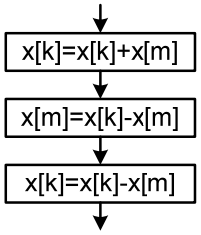
Для перестановки двух элементов массива *x[]* с индексами *k* и *m*, необходимо использование дополнительной переменной (*tmp*) для хранения копии одного из элементов (рис. 6.3, *а*), но можно обойтись и без использования дополнительной переменной *tmp*. В этом случае алгоритм перестановки имеет следующий вид (рис. 6.3, *б*).

В большинстве случаев предпочтительнее использовать первый способ, поскольку он не содержит дополнительных вычислений, что особенно важно при перестановке вещественных чисел.




```
tmp=x[k];
x[k]=x[m];
x[m]=tmp;
```

а)




```
x[k]=x[k]+x[m];
x[m]=x[k]-x[m];
x[k]=x[k]-x[m];
```

б)

Рис. 6.3. Алгоритм и фрагмент программы перестановки двух элементов массива с использованием дополнительной переменной (*а*) и без нее (*б*)

Пример 6.1. Переставить первый и последний элемент массива *x[]* местами. Количество элементов массива *n*.

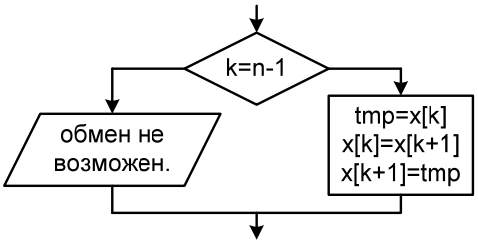
Решение. Нумерация элементов массива начинается с нуля, поэтому номер последнего элемента массива (*n*–1).

1 способ:
```
tmp=x[0];
x[0]=x
[n-1]; x[n-1]=tmp;
```

2 способ:
```
x[0]=x[0]+x[n-1];
x[n-1]=x[0]-x[n-1];
x[0]=x[0]-x[n-1];
```

Пример 6.2. Поменять местами заданный элемент массива *x[k]* с последующим.

Решение. При решении этой задачи необходимо учитывать, что если заданный элемент массива *x[k]* является последним, то обмен выполнить невозможно, поскольку последующий элемент отсутствует (рис. 6.4).




```
if(k == n-1)
  puts("Обмен не возможен. ");
else
{
  tmp=x[k];
  x[k]=x[k+1];
  x[k+1]=tmp;
}
```

Рис. 6.4. Алгоритм и фрагмент программы перестановки заданного элемента массива ***x[k]*** с последующим

При перестановке с предыдущим элементом обмен невозможен, если заданный элемент является первым (***k=0***).

6.3.4. Вычисление суммы элементов массива

Часто возникают задачи, требующие вычислить сумму всех или некоторых элементов массива, например, сумму элементов, стоящих в массиве на заданных местах, или сумму элементов, удовлетворяющих некоторому условию (сумму только положительных элементов, сумму ненулевых элементов второй половины массива и т. д.).

Пусть ***a[]*** – заданный массив из ***n*** элементов. Сумма всех его элементов в математической форме выглядит следующим образом:

$$s = a_0 + a_1 + \ldots + a_{n-1} = \sum_{i=0}^{n-1} a_i . \qquad (6.1)$$

Для вычисления суммы элементов части массива, например, с ***in-***го до ***ik***-го. Следует использовать формулу

$$s = a_{in} + a_{in+1} + \ldots + a_{ik} = \sum_{i=in}^{ik} a_i . \qquad (6.2)$$

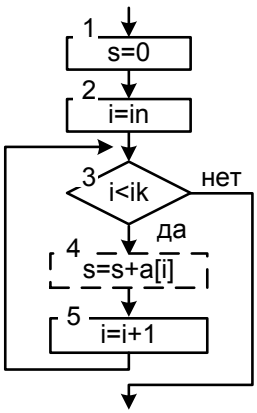
Очевидно, что формула (6.2) получается из формулы (6.1) при ***in = 0*** и ***ik*** = ***n*** – 1.

Алгоритм вычисления суммы состоит в следующем:

1) установить значение переменой для накопления суммы (***s***) в нулевое значение (***s* = 0**);

2) в цикле изменяя ***i*** от ***in*** до ***ik***, вычислить сумму элементов массива по выражению $s = s + a_i$.

При первой итерации цикла (***i* = *in***) получим $s = s + a_{in} = 0 + a_{in}$. На второй (***i* = *in* + 1**) – $s = s + a_{in+1} = a_{in} + a_{in+1}$ и т. д. На последней итерации цикла будем иметь $s = s + a_{ik} = a_{in} + a_{in+1} + \ldots + a_{ik}$. То есть в цикле по параметру ***i*** «старое» значение ***s***, содержащее накопленную сумму на предыдущей итерации, изменяется на значение a_i. На рис. 6.5 представлен алгоритм и фрагменты программ вычисления суммы элементов массива.



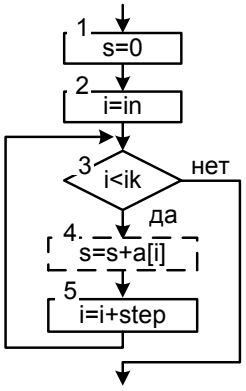

```
*/вычисление суммы элементов массива
с in до ik
s=0;
for(i=in;i<ik;i++)
  s=s+a[i];     // или  s+=a[i]; или s+=*(a+i);

*/вычисление суммы всех элементов массива
s=0;
for(i=0;i<n;i++)
    s+=a[i];
```

Рис. 6.5. Графическая схема и фрагмент программы вычисления суммы элементов массива

Если в алгоритме (см. рис. 6.5) в блоке 2 записать $i = 0$, а в блоке 3 – (***i < n***), то получим алгоритм вычисления суммы всех элементов массива.

Рассмотренный алгоритм вычисления суммы можно применить для *вычисления суммы элементов, стоящих в массиве на заданных местах* (рис. 6.6). В этом случае шаг изменения параметра цикла определяется переменной ***step***.




```
/* с помощью цикла  for */
s=0;
for(i=in;i<ik;i=i+step)
    s+=a[i];  // или s=s+a[i];
/* с помощью цикла  while */
s=0;   i=in;
while (i<ik)
{
    s+=a[i];
     i=i+step;
}
```

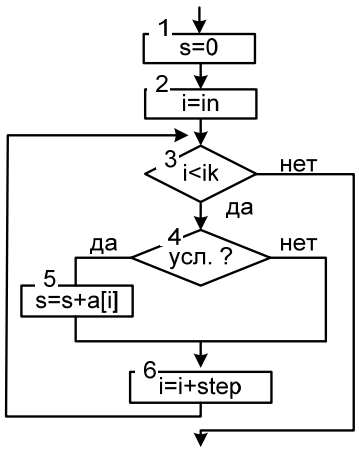
Рис. 6.6. Графическая схема и фрагмент программы вычисления суммы элементов массива, стоящих на заданных местах

Например, чтобы вычислить сумму элементов, стоящих в массиве на ***четных*** местах, необходимо «заставить» ***i*** принимать значения *1*, *3*, *5*, … (поскольку нумерация элементов массива в *C* начинается с нуля, т. е. элемент массива с индексом ***a[0]*** – первый элемент

массива). Для этого достаточно в блоке 2 записать ***i=1***, в блоке 3 – (***i<n***), а в блоке 5 записать ***i=i+2*** (***step=2***). В программе на языке *C* соответствующий фрагмент будет выглядеть следующим образом:

```
s=0;
for(i=1;i<n;i=i+2)        // или for(i=1;i<n;i+=2)
s+=a[i];                  // или s=s+a[i];
```

Для вычисления суммы только тех элементов, которые *удовлетворяют некоторому условию*, необходимо в алгоритме вычисления суммы (см. рис. 6.6) блок 4 заменить на ветвление, которое обеспечивает выполнение команды $s=s+a_i$ только тогда, когда условие выполнено для рассматриваемого элемента массива a_i. В этом случае алгоритм вычисления суммы примет следующий вид (рис. 6.7).




```
/* с помощью цикла  for */
s=0;
for(i=in;i<ik;i+=step)
   if( условие )  s+=a[i];  // или s=s+a[i];
/* с помощью цикла  while */
s=0;   i=in;
while (i<ik)
{
    if( условие )  s+=a[i];
     i=i+step;
}
```

Рис. 6.7. Графическая схема и фрагмент программы вычисления суммы элементов массива, стоящих на заданных местах

Применим полученный алгоритм для вычисления суммы положительных элементов массива, стоящих на нечетных местах. Для этого в блоке 2 запишем ***i=0***, в блоке 3 (***i<n***), в 4-м условие – (***a[i]>0***), а в блоке 6 изменение параметра цикла (***step=2***) ***i=i+2***. Тогда соответствующий фрагмент программы можно записать в виде:

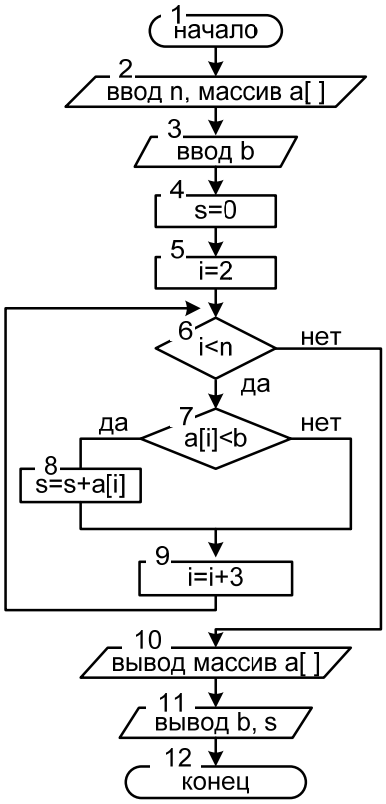
```
s=0;
for(i=0;i<n;i+=2)
    if( a[i]>0 )  s+=a[i];  // или s=s+a[i];
```

Рассмотрим примеры использования рассмотренных алгоритмов.

Пример 6.3. В одномерном массиве *a* размерностью *n* вычислить сумму элементов массива, меньших заданного значения *B* и стоящих на местах, кратных трем.

Решение. Объединим алгоритмы ввода-вывода массива (см. рис. 6.2) и вычисления суммы (см. рис. 6.7). Для сокращения записи графической схемы алгоритма ввода и вывода массива здесь и в дальнейшем используем простые блоки вида:

В алгоритме для вычисления искомой суммы рассматриваются только те элементы, которые в массиве стоят на местах, кратных трем, при этом необходимо учитывать, что нумерация элементов массива в *C* начинается с нуля, т. е. элемент массива с индексом *a[0]* – это первый элемент массива. Таким образом, элементы, стоящие на местах, кратных трем, – a_2, a_5, a_8, …, индекс элемента массива (он же – параметр цикла) должен последовательно принимать значения 2, 5, 8, …, т. е. изменяться от 2 с шагом 3, что и достигается изменениями в блоках 2 и 6 алгоритма вычисления суммы (см. рис. 6.7). Так, в блоке 2 запишем *i=2*, в блоке 3 – *i<n*, а в блоке 6 – (*step=3*) *i=i+3*. Для суммирования из рассмотренных элементов только тех, которые меньше заданного *B*, используется ветвление с условием $a_i<B$ (блок 4). Окончательный алгоритм вычисления суммы заданных элементов примет следующий вид (рис. 6.8). В задаче будем использовать динамический способ задания массива. В данном примере для обращения к элементам массива используются указатели. Как уже отмечалось в п. 6.1, имя массива является указателем на его первый элемент.



Используемые переменные:

n – число элементов массива;
a[] – динамический массив;
s – сумма элементов массива;
B – заданное число;
i – параметр цикла

```
#include <stdio.h>
main()
{
 int n,i;
 float s, B;

 puts("Введите число элементов массива a");
 scanf("%d",&n);

 float *a=new float[n];

 for (i=0;i<n;i++)
  {printf("Введите число a[%2d]= ",i);
    scanf("%f",a+i);
  }
 puts("Введите B");
scanf("%f",&B);
 s=0;
 for(i=2;i<n;i+=3)
    if(*(a+i)<B) s+=*(a+i);
 puts("Массив a");
 for(i=0;i<n;i++)
  printf("%.1f ",*(a+i));
 printf("\n");
 printf("Сумма чисел, меньших %.1f, стоящих на местах, кратных 3, равна %.2f\n",B,s);
 delete[] a; // освобождение памяти
 return(0);
}
```

Рис. 6.8. Графическая схема и программа примера 6.3

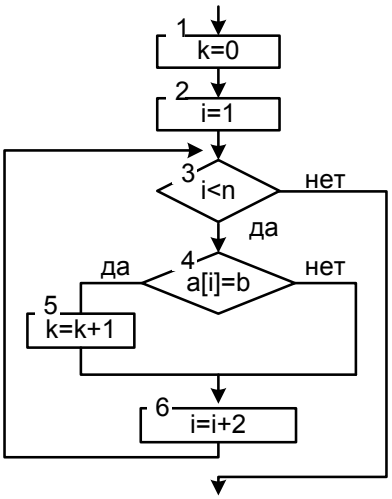
6.3.5. Подсчет количества элементов массива, удовлетворяющих заданному условию

Подсчет количества элементов массива, удовлетворяющих заданному условию, производится по алгоритмам, аналогичным вычислению суммы. Отличие заключается в том, что вместо добавления элемента массива к сумме, переменная – счетчик (*k*) увеличивается на единицу ($k = k + 1$). Таким образом, если в графических схемах алго-

ритмов (см. рис. 6.5–6.7) вместо $s = 0$ и $s = s + a_i$ записать $k = 0$ и $k = k + 1$, то получим алгоритмы подсчета количества элементов массива.

Пример 6.4. В одномерном массиве *a* размерностью *n* вычислить количество элементов, равных заданному числу *B* и стоящих на четных местах.

Решение. Графическая схема алгоритма решения задачи и фрагмент программы изображены на рис. 6.9.




```
/* с помощью цикла  for */
k=0;
for(i=1;i<n;i+=2)
   if(a[i]==b ) k++;  // или k=k+1;

/* с помощью цикла  while */
k=0;   i=1;
while (i<n)
{
   if(a[i]==b)  k++;
    i=i+2;
}
```

Рис. 6.9. Графическая схема и программа для примера 6.4

Следующий пример показывает, как в одном алгоритме находить сумму и количество элементов, удовлетворяющих заданному условию.

Пример 6.5. В одномерном массиве *a* размерностью *n* вычислить среднее арифметическое положительных элементов второй половины массива, стоящих на нечетных местах.

Решение. Среднее арифметическое чисел (*sr*) – частное от деления их суммы (*s*) на их количество (*k*): $sr = s/k$, где $k \neq 0$. Таким образом, задача сводится к нахождению суммы и количества положительных элементов второй половины массива, стоящих на нечетных местах. Для решения данной задачи применим алгоритм, приведенный на рис. 5.7, в соответствии с которым можно найти сумму и количество части элементов массива (второй половины), удовлетворяющих заданному условию (положительных элементов).

Определим номер *in* того элемента, с которого будем просматривать элементы второй половины массива. Поскольку по условию

задачи обрабатываются элементы, стоящие на нечетных местах, то начальное значение *in* тоже должно быть четным (поскольку нумерация элементов массива начинается с нуля). Значение переменной *in* можно определить по формуле (6.3), где пара квадратных скобок *[ ]* определяет операцию вычисления целой части числа:

$$in = \begin{cases} \left[\frac{n}{2}\right] + 1, & \text{если } \left[\frac{n}{2}\right] - \textit{не} \text{ четное число;} \\ \left[\frac{n}{2}\right], & \text{если } \left[\frac{n}{2}\right] - \text{четное число.} \end{cases} \qquad (6.3)$$

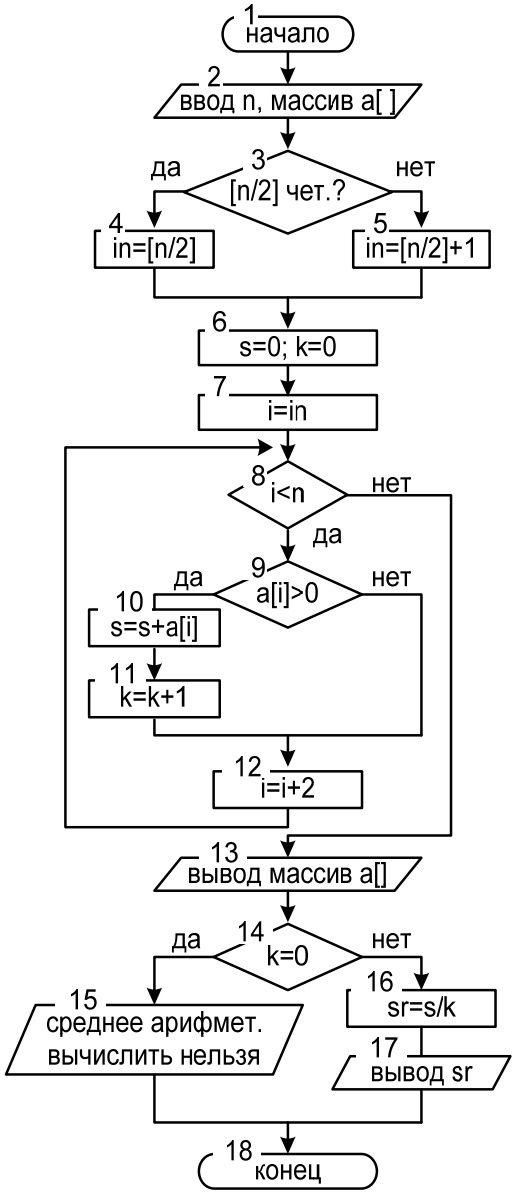
Значение *ik* совпадает с *n*, поскольку массив надо просматривать до конца. Параметр цикла *i* (номер элемента массива) необходимо изменять с шагом 2, чтобы обеспечить четные значения индекса (значения номеров элементов массива), т. е. *i=i+2*. Вычисление суммы и количества будем вычислять в одном цикле по параметру *i*, что значительно сократит алгоритм.

При вычислении среднего арифметического (*sr = s/k)* необходимо избежать возможного деления на ноль, поскольку, если во второй половине массива на нечетных местах не окажется положительных элементов, то *k* будет равным нулю. Например, для массива *X={2, 0, –6, 7, 9, 0, –14, –5, 0, –4, –32}* *n=11*, *in=[n/2]+1=6*, в цикле по *i* будут просматриваться только выделенные элементы, а среди них нет положительных чисел, поэтому переменная *k* останется равной нулю. В графической схеме (рис. 6.10) для этой цели используется ветвление, в котором проверяется условие *k = 0*. Если это условие выполнено, то выводится сообщение о том, что среднее значение не может быть вычислено, в противном случае вычисляется и выводится значение *sr*.

В приведенном ниже фрагменте программы четность [*n/2*] определяется по остатку от деления *n* на 2 с помощью операции *%* – определения остатка целочисленного деления (*примечание*: результат операции целочисленный, так как *n* и 2 – целочисленные операнды).

Используемые переменные:	`#include <stdio.h>`
n – число элементов массива;	`main()`
a[] – статический массив;	`{`
in – первый четный номер второй половины массива;	`float a[20];`
s – сумма элементов массива,	`int n, i , in, k;`
	`float s, sr;`

удовлетворяющих условию;
k – количество элементов массива, удовлетворяющих условию;
sr – среднее значение элементов массива, удовлетворяющих условию;
i – параметр цикла




```
 puts("Введите число элементов массива a");
 scanf("%d",&n);

 for (i=0;i<n;i++)
  { printf("Введите число a[%2d]= ",i);
    scanf("%f",&a[i]);
  }

 if ((n/2)%2==0) in=n/2;
 else in=n/2+1;

 s=0; k=0;
 for(i=in;i<n;i+=2)
    if(a[i]>0) { s+=a[i]; k++;}

 puts("Массив a");
 for(i=0;i<n;i++)
  printf("a[%2d]=%6.2f \n", i, a[i]);

 if (k==0)
  puts("Среднее арифметическое вычислить нельзя!");
  else
  {
    sr=s/k;
   printf("Среднее ариф. положж. элементов на нечетных. местах второй полов. массива =%6.3f\n", sr);
  }
 return(0);
}
```

Рис. 6.10. Графическая схема и программа для примера 6.5

6.3.6. Вычисление произведения элементов массива

Формулы, по которым вычисляется произведение элементов массива, аналогичны формулам вычисления сумм:

$$p = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1} = \prod_{i=0}^{n-1} a_i; \tag{6.4}$$

$$p = a_{in} \cdot a_{in+1} \cdot \ldots \cdot a_{ik} = \prod_{i=in}^{ik} a_i. \tag{6.5}$$

Поэтому вычисление произведения элементов массива выполняется по алгоритмам, аналогичным вычислению суммы. Отличие заключается в том, что начальное значение произведения p должно быть равным 1, а в цикле по параметру i надо вычислять $p = p \cdot a_i$. Таким образом, если в графических схемах алгоритмов (см. рис. 6.5–6.7) вместо $s = 0$ и $s = s + a_i$ записать $p = 1$ и $p = p \cdot a_i$, то получим алгоритмы вычисления произведения элементов массива.

Пример 6.6. В одномерном массиве a размерностью n, вычислить среднее геометрическое ненулевых элементов массива.

Решение. Среднее геометрическое k элементов массива – это корень степени k из произведения этих элементов. Таким образом, сначала необходимо вычислить произведение P ненулевых элементов массива и их количество k, а затем среднее геометрическое Sg по формуле

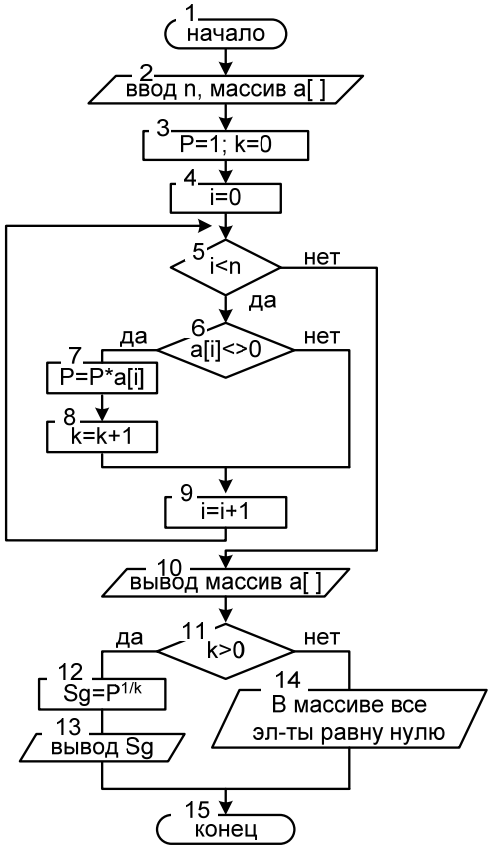
$$Sg = \sqrt[k]{P} = P^{\frac{1}{k}}. \tag{6.6}$$

Например, если элементы массива равны *A= {1, 0, 2, 4, 0}*, то – $P = 8, \quad k = 3, \quad SG = \sqrt[3]{P} = 2.$

Графическая схема алгоритма решения задачи изображена на рис. 6.11. В приведенном алгоритме в цикле по i (блоки 5–9) помимо вычисления произведения вычисляется и количество ненулевых элементов массива. После цикла с помощью ветвления проверяется, есть ли в массиве ненулевые элементы ($k>0$ – условие наличия в массиве ненулевых элементов), в этом случае вычисляется и выводится среднее геометрическое. В противном случае выводится сообщение «*В массиве все элементы равны нулю*». В программе переменные P и Sg имеют вещественный тип двойной точности (*double*), так как произведение вещественных чисел может быть очень большим числом.

Используемые переменные:

n – число элементов массива;
a[] – статический массив;
P – произведение не нулевых элементов массива;
k – количество не нулевых элементов массива;
Sg – среднее геометрическое элементов массива;
i – параметр цикла




```
#include <stdio.h>
#include <math.h>

main()
{
 float a[20];
 int n, i , k;
 double P, Sg;
 puts("Введите число элементов массива a");
 scanf("%d",&n);
 for (i=0;i<n;i++)
   { printf("Введите число a[%2d]= ",i);
     scanf("%f",&a[i]);
   }
 P=1; k=0;
 for(i=0;i<n;i++)
     if(a[i]!=0) {P*=a[i]; k++;}
 puts("Массив a");
 for(i=0;i<n;i++)
   printf("a[%2d]=%6.2f \n", i, a[i]);
  if(k>0)
  {
   if(P>0) Sg=powl(P,1.0/k);
   else Sg= –powl(fabs(P),1.0/k);
   printf("Среднее геометрическое ненулевых элементов массива =%.4lf \n", Sg);
      printf("P=%.4lf  k=%d \n", P, k);
  }
  else puts("В массиве все элементы равны нулю! ");
 return(0);
}
```

Рис. 6.11. Графическая схема и программа для примера 6.6

В программе для возведения P в степень $1/k$ используется функция *powl*(*основание, степень*), первый аргумент которой может быть только положительным числом. Поэтому для отрицательного P использовано выражение $-|P|^{1/k}$, запись которого на языке *C* имеет вид: *–powl(fabs(P), 1.0/k)*.

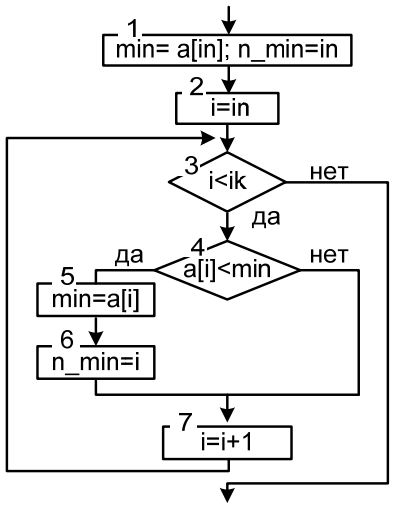
6.3.7. Поиск минимального и максимального элемента массива и его положения в массиве

Пусть требуется найти минимальный элемент (*min*) и его индекс (*n_min*) во всем массиве (*in* = 0 и *ik* = *n*) или какой-то его части (с *in*-го по *ik*-й). В этом случае алгоритм решения задачи можно записать так:

1) в качестве начального значения переменной *min* выберем любой из рассматриваемых элементов (обычно выбирают первый). Тогда $min = a_{in}$, *n_min* = *in*;

2) затем в цикле по параметру *i*, начиная со следующего элемента (*i* = *in* + *1*, …, *ik*), будем сравнивать элементы массива a_i с текущим минимальным *min*. Если окажется, что текущий (*i-й*) элемент массива меньше минимального ($a_i < min$), то переменная *min* принимает значение a_i, а *n_min* – на *i*, т. е. $min = a_i$, *n_min* = *i*.

Графическая схема алгоритма и фрагмент программы поиска минимального элемента в массиве приведены на рис. 6.12.




```
min=a[in];
n_min=in;
for(i=in+1; i<ik; i++)
   if(a[i]<min)
   {
     min=a[i];
    n_min=i;
   }
```

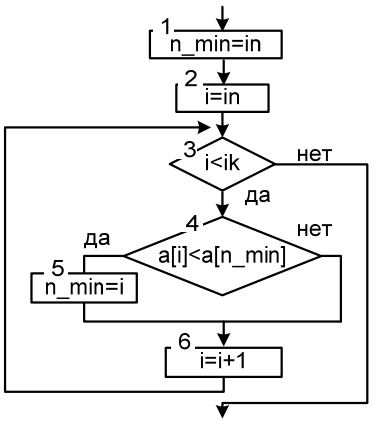
Рис. 6.12. Графический алгоритм и фрагмент программы поиска минимального элемента в массиве

Заметим, что при наличии в массиве нескольких минимальных элементов найден будет первый из них (самый левый минимальный элемент) при просмотре массива слева направо. Если в неравенстве $a_i < min$ знак > поменять на знак ≥, то будет найден последний из них (самый правый минимальный элемент).

Для поиска максимального элемента *max* и его индекса *n_max* используется аналогичный алгоритм, в котором сначала надо принять

$max = a_{in}$, $n\_max = in$, вместо неравенства $a_i < min$ используется неравенство $a_i > max$. В случае выполнения условия $a_i > max$ записать в $max = a_i$ и в $n\_max = i$.

Для поиска в массиве экстремума можно не использовать вспомогательную переменную *min* (*max*). В этом случае минимальный элемент массива определяется только по его индексу *n_min* (*n_max*) (рис. 6.13).



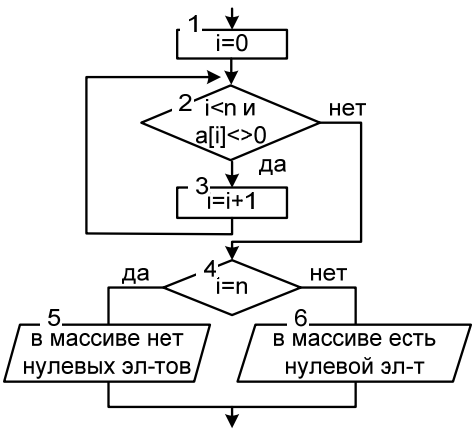

```
/*поиск минимального элемента*/
   n_min=in;
   for(i=in+1; i<ik; i++)
      if(a[i]<a[n_min])
         n_min=i;
/*поиск максимального элемента*/
   n_max=in;
   for(i=in+1; i<ik; i++)
      if(a[i]>a[n_max])
         n_max=i;
```

Рис. 6.13. Графический алгоритм и фрагмент программы поиска минимального элемента в массиве по его индексу

6.3.8. Поиск элементов, обладающих заданным свойством

При поиске элементов, обладающих заданным свойством, необязательно просматривать все элементы массива. Например, требуется определить, есть ли в массиве хотя бы один нулевой элемент. Для ответа на этот вопрос достаточно в цикле просматривать элементы массива до тех пор, пока не закончится массив или не встретится равный нулю элемент. Если, например, уже третий элемент равен нулю, то остальные элементы просматривать нет необходимости.

В таких случаях для просмотра массива обычно используется оператор цикла ***while*** со сложным условием. Графическая схема для рассматриваемого примера изображена на рис. 6.14. После цикла достаточно проверить, чему равно ***i***. Если окажется, что ***i=n***, т. е. были просмотрены все элементы, то в массиве нет нулевых элементов.

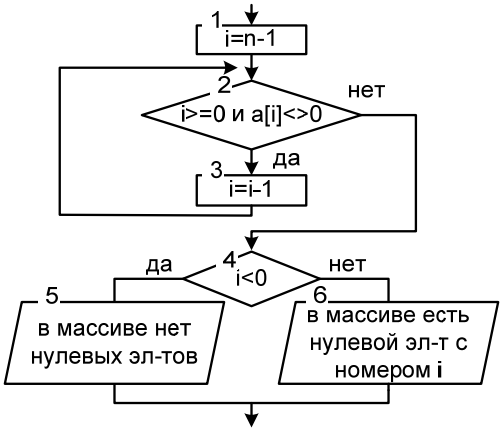



```
i=0;
while(i<n && a[i]!=0) i=i+1;

if(i == n)
   puts("В массиве нет нулевых элементов");
else
   puts("В массиве есть нулевой элемент")
```

Рис. 6.14. Графическая схема и фрагмент программы поиска нулевого элемента в массиве

Встречаются задачи, в которых требуется не только определить, есть ли элемент, обладающий заданным свойством в массиве, но и номер (индекс) такого элемента. Например, найти максимальный элемент в части массива, находящейся после последнего нуля. Решение задачи следует начать с вычисления индекса последнего нулевого элемента. Для определения индекса самого правого элемента, обладающего заданным свойством, массив следует просматривать с конца до тех пор, пока не закончатся элементы и текущий элемент не равен нулю (рис. 6.15).




```
i=n–1;
while(i>=0 && a[i]!=0) i=i–1;

if(i<0)
   puts("В массиве нет нулевых элементов");
 else
   printf("Индекс последнего нуля – %d \n", i);
```

Рис. 6.15. Графическая схема и фрагмент программы поиска номера последнего нулевого элемента в массиве

Номер (индекс) первого встретившегося нулевого элемента можно узнать по значению параметра цикла ***i***. Этот номер можно использовать в дальнейших вычислениях, например, как номер начального элемента для поиска максимума.

6.3.9. Поиск в упорядоченном массиве

Упорядоченность элементов массива позволяет значительно увеличить скорость его обработки за счет снижения числа проверяемых элементов массива. В таких алгоритмах массив проверяется, пока выполняется (или не выполняется) дополнительное условие, определяющее досрочный выход из цикла. Также при составлении алгоритма необходимо учитывать, возрастающим или убывающим является проверяемый массив, что оказывает влияние на то, как удобнее обрабатывать массив с начала или с конца. В общем случае алгоритм обработки упорядоченного массива имеет следующий вид (рис. 6.16).

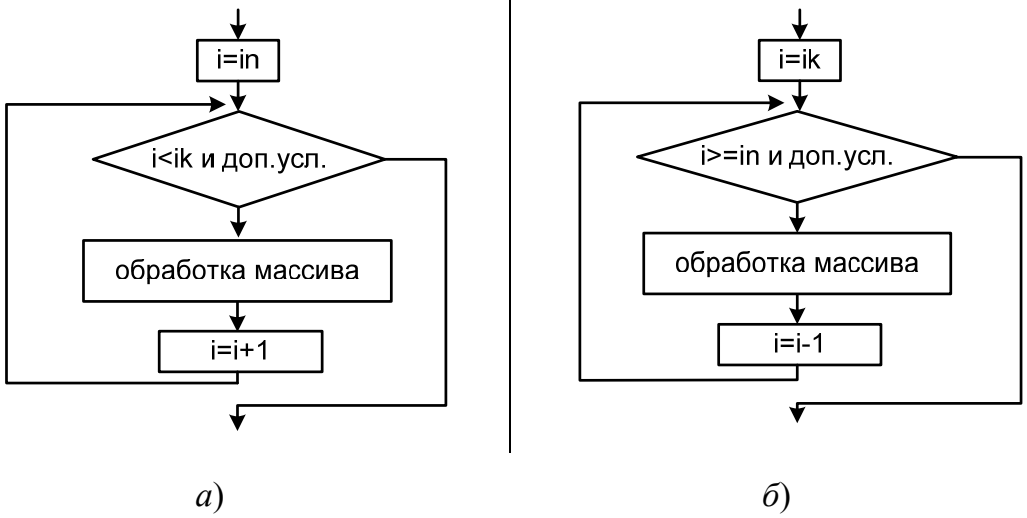


Рис. 6.16. Графический алгоритм обработки упорядоченного массива с перебором с начала (*а*), с конца (*б*)

Как видно из графической схемы, дополнительное условие управляет досрочным выходом из цикла. Пока дополнительное условие истина и не конец массива ($i < ik$), цикл выполняется, как только одно из условий будет не выполнено, происходит выход из цикла.

Пример 6.7. В возрастающем одномерном массиве X с количеством элементов n определить, есть ли число, равное A, и на какой позиции оно находится, если числа A нет, определить место, на котором оно должно находиться, чтобы не нарушить упорядоченность массива.

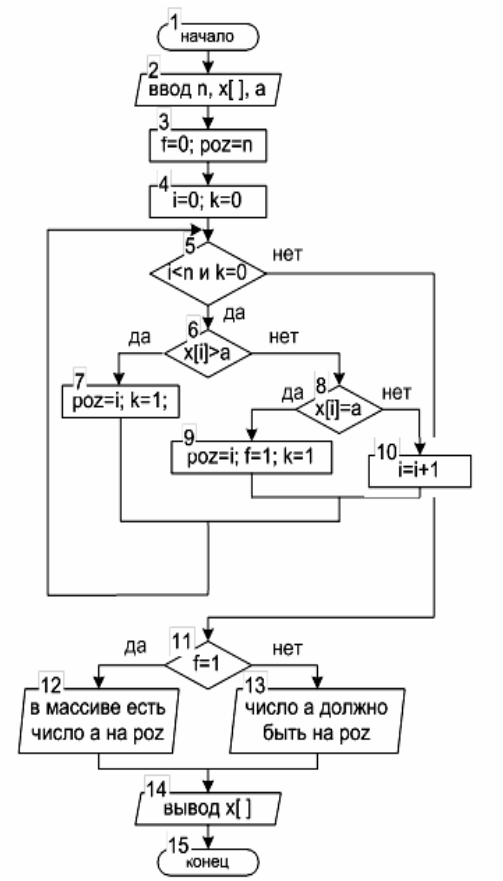
Решение. В данной задаче обработку массива будем проводить сначала. Выход из цикла по дополнительному условию будет выполнен, если в массиве найден элемент, больший либо равный A ($k = 1$). Для индикации наличия в массиве элемента, равного A, введем вспомогательную переменную f с начальным значением $f = 0$. При обна-

ружении элемента *A* переменная *f* = 1. Для определения номера позиции числа *A* в массиве введем дополнительную переменную *poz* с начальным значением *n*, т. е. предполагая, что все элементы массива меньше *A*. При обнаружении в массиве числа, большего или равного *A*, в переменной *poz* сохраняется его индекс – *i*. После выхода из цикла по значению переменной *f* определяется наличие и место переменной *A* в массиве. Описанный алгоритм поиска и программа представлены на рис. 6.17.

Используемые переменные:
n – число элементов массива;
a[] – статический массив;
k – переменная для досрочного выхода из цикла при нахождении элемента, большего или равного a;
f – вспомогательная переменная для индикации наличия в массиве числа, равного a;
poz – номер элемента массива, на котором должно находиться число a;
i – параметр цикла

```
#include <stdio.h>

main()
{
  int f, k, n, poz, i, x[100], a;
  puts("Введите число элементов массива: ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
    printf("x[%2d]=",i);
    scanf("%d",&x[i]);
  }
  puts("Введите число a: ");
  scanf("%d",&a);
  f=0; poz=n; k=0;
  i=0;while (i<n && k==0)
  {
    if(x[i]>a) { poz=i;k=1;}
    else
    {
      if(x[i]==a)
        {poz=i; f=1; k=1;} else i=i+1
    }
  }
  if(f==1)
    printf("В массиве есть число =%d, на позиции-%d\n", a, poz);
  else
    printf("Число %d должно
```



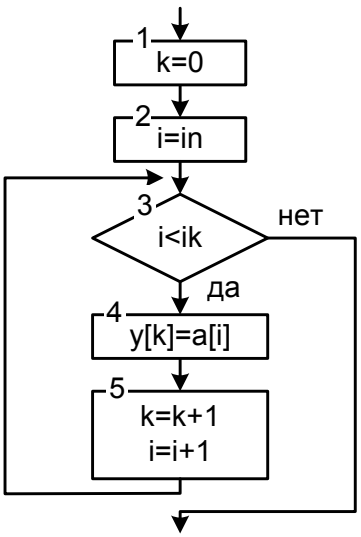

```
находиться на позиции-
%d\n",a, poz);
 for(i=0;i<n;i++)
 printf("x[%d]=%d\n",i,x[i]);
  return 0;
}
```

Рис. 6.17. Графический алгоритм и программа для примера 6.7

Описанный алгоритм можно дополнить предварительным сравнением последнего элемента массива *X[n-1]* с числом *A*. Если *X[n-1] = A*, то заданное число находится на последнем месте, а в случае выполнения *X[n-1]>A*, число *A* должно находиться в массиве на позиции *n*. Если ни одно из этих условий не выполнено, то это означает, что необходимо выполнить поиск числа *A* в массиве.

6.3.10. Копирование массивов

В ряде задач для организации дополнительных или промежуточных вычислений требуется создание копии всего массива или части его элементов. Для этого можно воспользоваться алгоритмом, представленным на рис. 6.18.




```
k=0;
for(i=in;i<ik;i++)
{
  y[k]=a[i];
  k++;
}
```

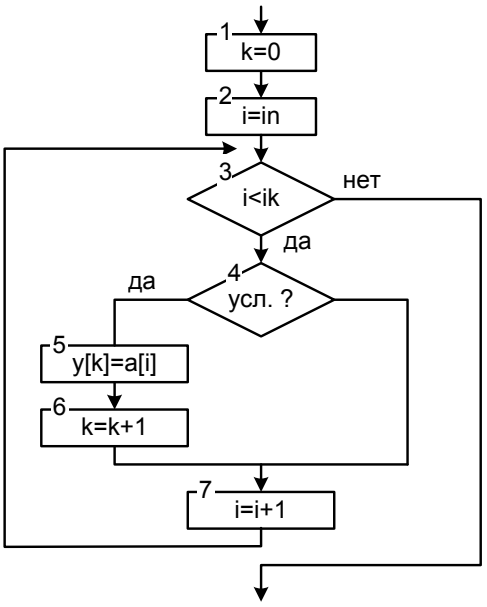
Рис. 6.18. Алгоритм и фрагмент программы создания копии массива

В зависимости от параметров *in* и *ik* в массив *y[ ]* копируются элементы из исходного массива *a[ ]*. Так, для копирования всех элементов массива *a[ ]* необходимо задать *in* = 0, *ik* = *n* (*n* – количество элементов массива *a[ ]*). При копировании части массива, например с 3 по 9, принимаем *in* = 2 (поскольку нумерация элементов массива в *C++* начинается с нуля) и *ik* = 9.

6.3.11. Формирование нового массива

В задачах формирования нового массива требуется создать массив из элементов существующего массива (массивов), удовлетворяющих заданному условию. В новый массив элементы заносятся последовательно, начиная с нулевого индекса. Максимальное число элементов в формируемом массиве может достигать количества элементов в исходном массиве (массивах), минимальное значение равняется нулю. В этом случае считается, что новый массив не сформирован.

При формировании новых массивов удобно использовать динамические массивы, поскольку число его элементов заранее не известно. Алгоритм создания нового массива схож с алгоритмом копирования (рис. 6.19).




```
k=0;
for(i=in;i<ik;i++)
{
  if (условие)
  {
    y[k]=a[i];
    k++;
  }
}
```

Рис. 6.19. Алгоритм и фрагмент программы формирования нового массива

Для последовательной записи элементов в новый массив используется дополнительная переменная *k – счетчик элементов в новом массиве*. Начальное значение этой переменной принимается равной *нулю*, т. е. считается, что в новом массиве нет элементов. При обнаружении в исходном массиве элемента, удовлетворяющего заданному условию, его значение заносится в новый массив на позицию *k*, а после счетчик элементов увеличивается на единицу (*k=k+1*). Таким образом, после обработки всего исходного массива по значению счетчика *k* можно определить, сформирован новый массив (*k>0*) и сколько в нем элементов (*k*).

Пример 6.8. Даны два одномерных массива *X* и *Y*. Необходимо сформировать массив *Z* из положительных элементов массива *X*, стоящих на четных местах, и элементов массива *Y*, больших первого элемента массива *X*.

Решение. Если число элементов массива *X – n*, а массива *Y – m*, то с учетом того, что из первого массива выбираются элементы, стоящие только на четных местах, максимальное число элементов в новом массиве *Z* может достигать *m+n/2* элементов. Поэтому для массива *Z* с помощью оператора динамического выделения памяти (*new*) выделим *m+[n/2]* ячейки памяти (*[n/2]* – целая часть от деления). Начальное значение счетчика элементов нового массива *k* принимается равным *нулю*.

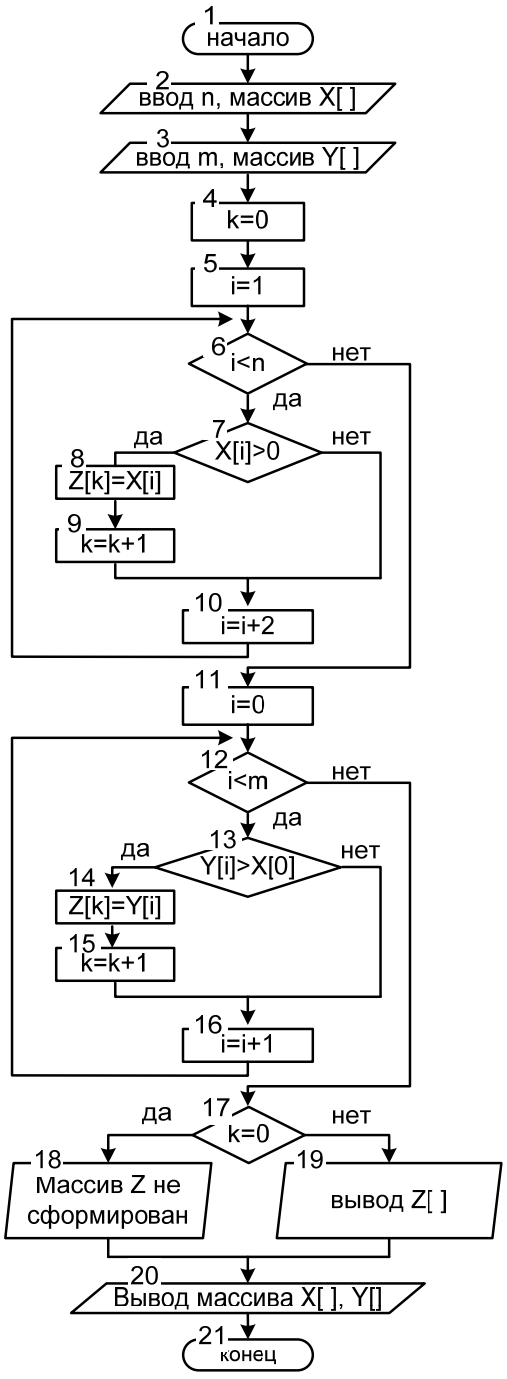
При обработке массива *X* необходимо проверять только элементы, стоящие на четных местах, т. е. параметр цикла *i* изменяется от *in = 1* до *ik = n* с шагом *2*. Условие отбора элементов из первого массива *X[i] > 0*. При обработке массива *Y* учитываются все его элементы, т. е. параметр цикла *i* изменяется от *in = 0* до *ik = m* с шагом 1. Условие отбора элементов из второго массива – *Y*[*i*] > *X*[*0*].

Описанный алгоритм формирования нового массива и программа представлены на рис. 6.20.

Используемые переменные:

x[] – статический (исходный) массив;
n – число элементов массива X;
y[] – статический (исходный) массив;
m – число элементов массива;
z[] – динамический (формируемый) массив;
k – счетчик элементов нового массива Z;
i – параметр цикла

```
#include <stdio.h>
main()
{
  int  k, n, m, i, x[10], y[10];
  puts("Введите число элементов массива X: ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
   printf("x[%2d]=",i);
   scanf("%d",&x[i]);
  }
  puts("Введите число элементов массива Y: ");
  scanf("%d",&m);
  for(i=0;i<m;i++)
  {
   printf("y[%2d]=",i);
   scanf("%d",&y[i]);
  }
  int *z=new int[15]; // выделение памяти под массив Z
  k=0;
  for(i=1;i<n;i+=2)
  {
   if(x[i]>0)
   {
    z[k]=x[i];
    k++;
   }
```




```
 }
 for(i=0;i<m;i++)
 {
  if(y[i]>x[0])
  {
   z[k]=y[i];
   k++;
  }
 }
 puts("Массив X:");
 for(i=0;i<n;i++)
  printf("x[%d]=%d\n",i,x[i]);
 puts("Массив Y: ");
 for(i=0;i<m;i++)
  printf("y[%d]=%d\n",i,y[i]);
 if(k==0)
   puts("Массив Z не сформиро-
ван. ");
 else
 {
    puts("Массив Z:");
    for(i=0;i<k;i++)
    printf("z[%d]=%d\n",i,z[i]);
 }
 delete[] z; // освобождение
памяти
}
```

Рис. 6.20. Графический алгоритм и программа для примера 6.8

6.3.12. Примеры решения задач по обработке одномерных массивов

Задача 1. В одномерном массиве *A* размерностью *n* найти количество чисел, меньших заданного *X*, и произведение отрицательных чисел, стоящих на четных местах.

Решение. При решении использовать табл. 6.1.

Таблица 6.1

Таблица соответствия переменных

Переменные в задаче	Имя на языке Си	Тип	Комментарий
A[]	A[]	float	Одномерный статический массив
n	n	int	Количество элементов массива
P	P	float	Произведение отрицательных чисел
X	x	float	Заданное число
k	k	int	Количество чисел, меньших *X*
–	k1	int	Количество отрицательных чисел
–	i	int	Номер элемента массива; параметр цикла

Графическая схема алгоритма представлена на рис. 6.21.

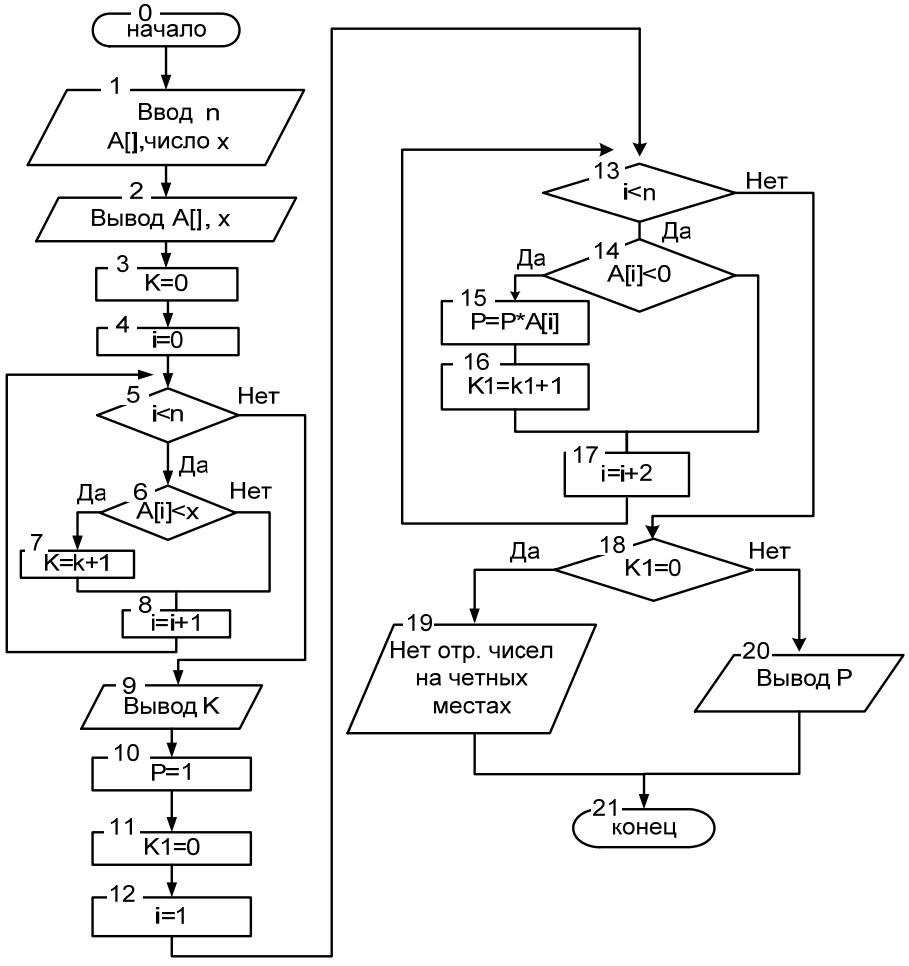


Рис. 6.21. Графическая схема алгоритма решения задачи 1

Описание алгоритма:

1) Блоки 1–2 – вводятся исходные данные.

2) Блоки 3–9 – вычисление и вывод количества ***K***, меньших заданного ***X***.

3) Блоки 10–18 – вычисление количества ***K1*** и произведения ***P*** отрицательных элементов массива, стоящих на четных местах.

4) Блоки 18–20 – проверка на наличие в массиве отрицательных чисел на четных местах и в случае выполнения проверки вывод их произведения ***P***.

Листинг программы:

```
#include <stdio.h>
#include <math.h>
main ()
{
 float P, A[10], x;                  //Описание переменных
 int i, k1, k, n;
 puts (" Введите число x");
 scanf ("%f", &x);                   // Ввод x
 puts (" Введите число элементов массива A");
 scanf ("%d", &n);
 for (i=0; i<n; i++)                 // Ввод массива
 {
  printf("Введите число A[%d]=", i);
  scanf("%f", &A[i]);
 }
 puts("Массив A");                   // Вывод массива
 for (i=0; i<n; i++)
  printf("%.2f ", A[i]);
 printf("\n");
 printf(" x=%.3f \n", x);            // Вывод x
 k=0;
 for (i=0; i<n; i++)
   if (A[i]<x) k=k+1;                // Определение кол-ва чисел <x
 printf(" k=%d\n" ,k);
 P=1;
 k1=0;
 for (i=1; i<n; i=i+2)
    if (A[i]<0)
    {                                // Вычисление произведения
```

```
    P=P*A[i];                    // отрицательных чисел
    k1=k1+1;                     // стоящих на четных местах
     }
  if (k1==0)
    printf("В массиве на четных местах нет отрицательных чисел \n");
  else
  printf(" P=%6.2f \n" ,P);
  }
```

***Тесты*:**

1)

Массив A
3.00 6.00 10.00 -7.00 -3.00 -7.00 -5.00 17.00 6.00 10.00
x=10.000
k=7
P= 49.00

2)

Массив A
2.00 9.00 4.00 6.00 7.00 8.00
x=0.000
k=0
В массиве на четных местах нет отрицательных чисел

Задача 2. В одномерном массиве *A* размерностью *n* найти максимальный элемент, расположенный между первым и последним нулевыми элементами массива.

Решение. Для решения использовать табл. 6.2.

Таблица 6.2

Таблица соответствия переменных

Переменные в задаче	**Имя на языке Си**	**Тип**	**Комментарий**
max	max	int	Максимальное число
A[]	A[]	int	Одномерный динамический массив
n	n	int	Количество элементов
–	t1	int	Индекс первого нулевого элемента
–	t2	int	Индекс последнего нулевого элемента
–	i	int	Номер элемента массива; параметр цикла

Графическая схема алгоритма представлена на рис. 6.22.

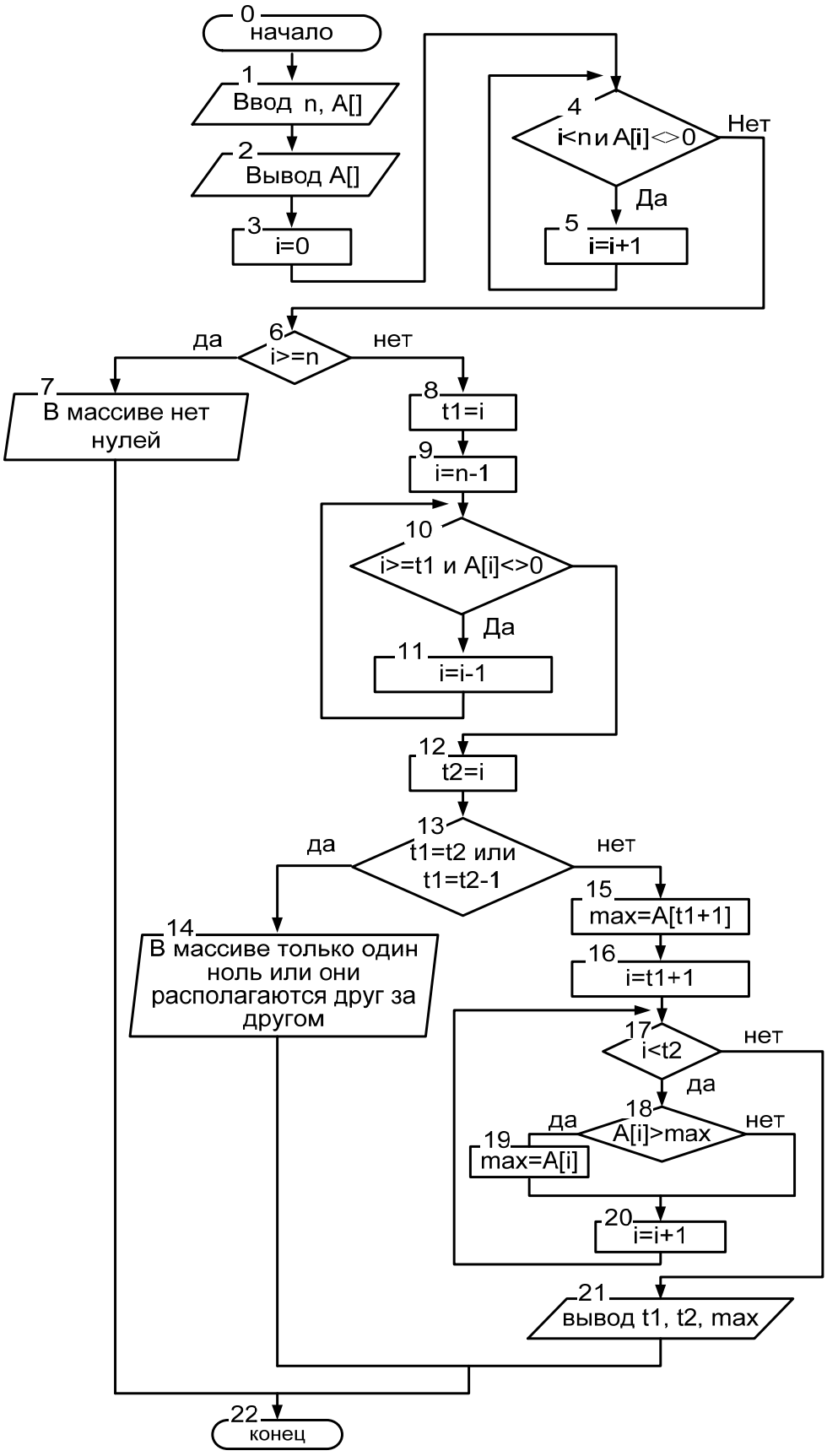


Рис. 6.22. Графическая схема алгоритма решения задачи 2

Описание алгоритма:

1) Блоки 1–2 – ввод/вывод исходного массива ***A[]***.

2) Блоки 3–5 – поиск первого нулевого элемента в массиве.

3) Блоки 6–7 – проверка на наличие в массиве хотя бы одного нулевого элемента и в случае их отсутствия вывод сообщения "*В массиве нет нулей*" и переход на конец алгоритма.

4) Блоки 8–12 – поиск последнего нулевого элемента в массиве и сохранение позиции первого нуля в ***t1***, а последнего в ***t2***.

5) Блоки 13–14 – проверка расположения нулевых элементов в массиве и в случаи ошибки вывод сообщения "*В массиве только один ноль или они располагаются друг за другом*" и переход на конец алгоритма.

6) Блоки 15–21 – в случае выполнения проверки (блок 13) поиск максимального элемента между крайними нулевыми элементами и вывод полученного результата.

Тесты:

1)
Массив A:
a[0]=1
a[1]=3
a[2]=0
a[3]=x6
a[4]=-3
a[5]=7
a[6]=4
a[7]=0
a[8]=1
t1=2 t2=7 max=7

2)
Массив A:
a[0]=0
a[1]=1
a[2]=-2
a[3]=4
a[4]=0
t1=0 t2=4 max=4

3)
Массив A:
a[0]=1
a[1]=2
a[2]=0
a[3]=5
a[4]=0
a[5]=3
a[6]=8
a[7]=0
a[8]=3
a[9]=1
t1=2 t2=7 max=8

4)
Массив A:
a[0]=1
a[1]=2
a[2]=3
a[3]=4
В массиве нет нулей

5)
Массив A:
a[0]=1
a[1]=0
a[2]=2
a[3]=3
a[4]=4
В массиве только один ноль или они располагаются друг за другом
t1=1 t2=1 max=0

6)
Массив A:
a[0]=1
a[1]=2
a[2]=0
a[3]=0
a[4]=5
a[5]=2
В массиве только один ноль или они располагаются друг за другом

Листинг программы:

```
#include <stdio.h>
main()
{
  int i,t1,t2,n,max;
  puts("Введите число элементов массива A");
  scanf ("%d", &n);
  int*A=new int[n];   //выделение памяти под массив
  for(i=0;i<n;i++)    //ввод массива
  {
    printf("a[%2d]= ",i);
    scanf("%d",&A[i]);
  }
  puts("Массив A: ");
  for(i=0;i<n;i++)    //вывод массива
    printf("a[%d]=%d\n",i,A[i]);
  i=0;
  while(i<n && A[i]!=0) i=i+1;  //поиск позиции первого нуля
  if(i>=n)
    printf("В массиве нет нулей \n");
  else
  {
    t1=i;
    i=n-1;
    while(i>=t1 && A[i]!=0) i=i-1; //поиск позиции последнего нуля
    t2=i;
    if(t1==t2 || t1==t2-1)
       printf("В массиве только один ноль или они располагаются друг
за другом\n");
    else
    {
       max=A[t1+1];
       for(i=t1+1;i<t2;i++)
        if(A[i]>max) max=A[i];   //поиск максимального элемента
       printf("t1=%d t2=%d max=%d \n",t1,t2,max);
    }
  }
 delete[] A;          // освобождение динамической памяти
       }
```

6.4. Основные алгоритмы обработки двумерных массивов

6.4.1. Понятие многомерных массивов

Элементом массива может быть в свою очередь тоже массив. *Размерность* или количество измерений массива определяется количеством индексов при обращении к элементам массива. Массивы бывают одномерные, двумерные, трехмерные и т. д.

Общий вид объявления массива:

<имя_типа> <имя_массива> [k1] [k2] ... [kn]; ,

где *k*1, *k*2, …, *kn* – количество элементов массива – константы или константные выражения по 1, 2, …, *n* измерениям. Причем значения индексов могут изменяться от 0 до *ki*–1.

Описание двумерного массива (или матрицы) строится из описания одномерного путем добавления второй размерности:

int a[4][3].

Анализ подобного описания необходимо проводить в направлении выполнения операций [], т. е. слева направо. Таким образом, переменная *a* является массивом из четырех элементов, что следует из первой части описания *a*[4]. Каждый элемент *a*[*i*] этого массива в свою очередь является массивом из трех элементов типа *int*, что следует из второй части описания.

Имя двумерного массива с двумя индексными выражениями в квадратных скобках за ним обозначает соответствующий элемент двумерного массива и имеет тот же тип. Например, *a*[2][1] является величиной типа *int*, а именно ячейкой, в которой находится число 8, и может использоваться везде, где допускается использование величины типа *int*.

Примеры объявления массивов:

```
int C[10] [20] [15];            //трехмерный целочисленный массив
```

Примеры обращение к элементам:

```
float   B[50][20];        //  двумерный вещественный массив
                          //  из 50 строк и 20 столбцов.
```

Примеры обращения к их элементам:

B[0][0], B[0][1] ,.., B[i][j], …, B[4][19], …

C[i] [j] [k], ..., C[2*i] [0] [3*k+1], …

6.4.2. Динамические двумерные массивы

Обращение к элементу динамического массива осуществляется также, как и к элементам обычного массива.

Например, *a*[0], *a*[1], …, *a*[19], *c*[2][5] и т. д.

Другой способ обращения к элементу *b*[*i*][*j*] двумерного массива имеет вид:

$$*(*(b+i)+j).$$

Поясним это. Двумерный массив представляет собой массив массивов, т. е. это массив, каждый элемент которого является массивом. Имя двумерного массива также является константным указателем на начало массива. Например, *int b*[5][10] является массивом, состоящим из 5-ти массивов. Для обращения к *b*[*i*][*j*] сначала требуется обратиться к *i*-й строке массива, т. е. к одномерному массиву *b*[*i*]. Для этого надо к адресу начала массива *b* прибавить смещение, равное номеру строки *i*: *b*+*i* (при сложении указателя *b* с *i* учитывается длина адресуемого элемента, т. е. ***i*·(*n*·*sizeof*(*int*))**, так как элементом массива *b*[*i*] является строка, состоящая из *n* элементов типа int). Затем требуется выполнить разадресацию: *(*b*+*i*). Получим массив из 10 элементов. Далее необходимо обратиться к *j*-му элементу полученного массива. Для получения его адреса опять применяется сложение указателя с *j*: ***(*b*+*i*)+*j*** (на самом деле прибавляется ***j*·*sizeof*(*int*)**). Затем применяется операция разадресации: ***(*(*b*+*i*)+*j*)**. Таким образом, получаем формулу для вычисления адреса элемента *b*[*i*][*j*]:

$$b+k\cdot i\cdot n+k\cdot j=b+k\cdot(i\cdot n+j),$$

где *k* – длина в байтах одного элемента массива; *b* – адрес начала массива. Эта формула может быть использована в дальнейшем, например, для организации передачи в подпрограмму двумерного массива переменной размерности.

Приведем универсальный способ выделения памяти под двумерный массив, когда обе его размерности задаются на этапе выполнения программы:

```
int i, m, n;   //i – номер строки, m, n – количество строк и столбцов
puts("Введите m и n");
scanf("%d %d",&m, &n);        // Ввод m, n
int **b=new int *[m];          //1
for (i=0; i<m; i++)            //2
    b[i]=new int[n];           //3
```

Здесь в операторе //1 объявляется переменная типа «указатель на указатель на *int*» и выделяется память под массив указателей на

строки массива (*m* строк). В операторе //2 организуется цикл для выделения памяти под каждую строку массива. В операторе //3 выделяется память под каждую строку массива и *i*-му элементу массива указателей на строки присваивается адрес начала участка памяти, выделенного под строку двумерного массива. Каждая строка массива состоит из *n* элементов типа *int*.

Примечание. Для выделения динамической памяти для вещественного двумерного массива достаточно в приведенном фрагменте программы в строках //1, //3 имя типа *int* поменять на имя типа *float*.

Освобождение выделенной динамической памяти. Динамическая память освобождается с помощью операции *delete[] имя массива*, например, освобождение памяти для двумерного массива *b* выглядит следующим образом:

```
for (i=0; i<n; i++)
    delete [ ] b[i];
 delete [ ] b;
```

Время «жизни» динамического массива определяется с момента выделения динамической памяти до момента ее освобождения.

6.4.3. Алгоритмы обработки двумерных массивов

На рис. 6.23 представлен алгоритм ввода и вывода двумерного массива.

Примечание. Ввод-вывод динамической матрицы отличается от ввода-вывода статической матрицы лишь описанием матрицы.

```
//Ввод матрицы
float  x[10][10];
puts(«Введите m,n»);
scanf("%d %d",&m,&n);
for(i=0;i<m;i++)
  for(j=0;j<n;j++)
   {
    printf("Введите x[%d][%d]= ",i,j);
    scanf("%f",&x[i][j]);
   }

// Вывод матрицы
  puts("Матрица x: ");
  for(i=0;i<m;i++)
    { for(j=0;j<n;j++)
          printf("%8.2f",x[i][j]);
     printf("\n");
    }
```

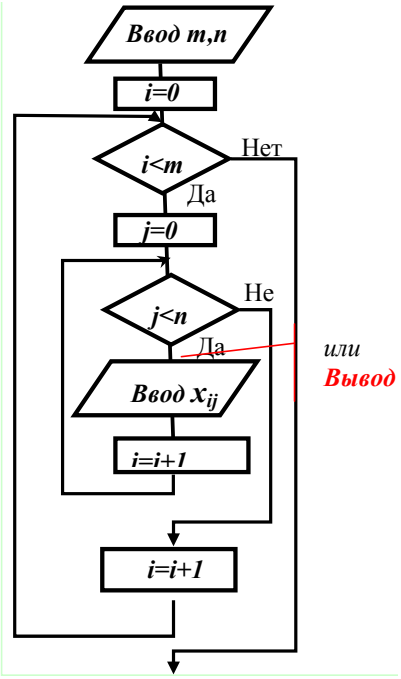


Рис. 6.23. Графическая схема алгоритма ввода и вывода двумерного массива

Пример 6.1. Определить количество элементов, больших заданного *A* и расположенных в строках с нечетными номерами.

Таблица 6.3

Таблица соответствия переменных

Переменные в задаче	Имя переменной на языке Си	Тип	Комментарий
K	K	int	Искомое количество элементов
B	B	float	Двумерный статический массив
A	A	float	Заданное число
–	i	int	Номер строки
–	j	int	Номер столбца

Графическая схема алгоритма представлена на рис. 6.24.

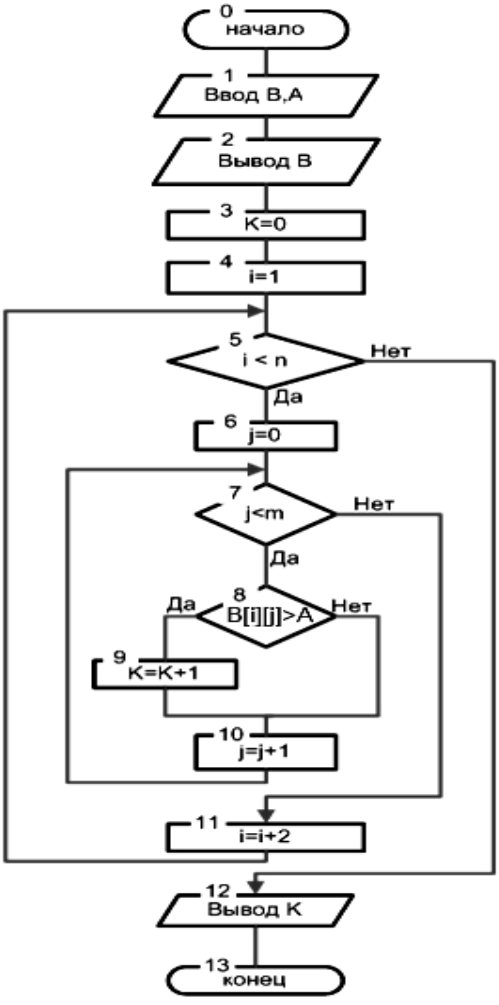


Рис. 6.24. Графическая схема алгоритма примера 6.1

Листинг программы:

/* Определение количества элементов, больших заданного *A* и расположенных в строках с нечетными номерами */

```
 # include <stdio.h>
 # include <math.h>
 void main ( )
 {
  int i, j, m, n, K;
  float B [10][10];
  float A; //Описание переменных
  printf ("Введите число строк и столбцов");
  scanf ("%d %d", &m, &n);
  for(i=0; i<m; i++)
    for(j=0; j<n; j++)
    {
      printf("Введите B[%d, %d]= ", i, j);
      scanf("%f", &B[i] [j]);
    }

  puts("Матрица B: ");
  for(i=0;i<m;i++)
    { for(j=0;j<n;j++)
          printf("%8.2f",B[i][j]);
      printf("\n");
    }
    puts (" введите число A");
    scanf ("%f", &A);
    K=0;
    for ( i=1; i<m; i=i+2)
      for ( j=0; j<n; j++)
        if( B[i][j]>A) K=K+1;
    printf("%d \n", K);
  }              //конец main
```

Тесты:

1) B = $\begin{pmatrix} -3; 8; -2; 10; 7; 82; -4 \\ -9; 0; -3; 85; 3; 40; -9 \\ -1; 8; 7; -95; 4; -5; 67 \end{pmatrix}$ A= 4; K=7.

2) B = $\begin{pmatrix} -4; -7; 8; 9; 2; -8; -5 \\ -2; 10; 0; 9; -8; 7; -5 \\ -7; 6; -9; 7; 0; -3; 7 \end{pmatrix}$ A=10; K=0.

Пример 6.2. Найти в матрице строку с максимальной суммой элементов.

Графическая схема алгоритма представлена на рис. 6.25.

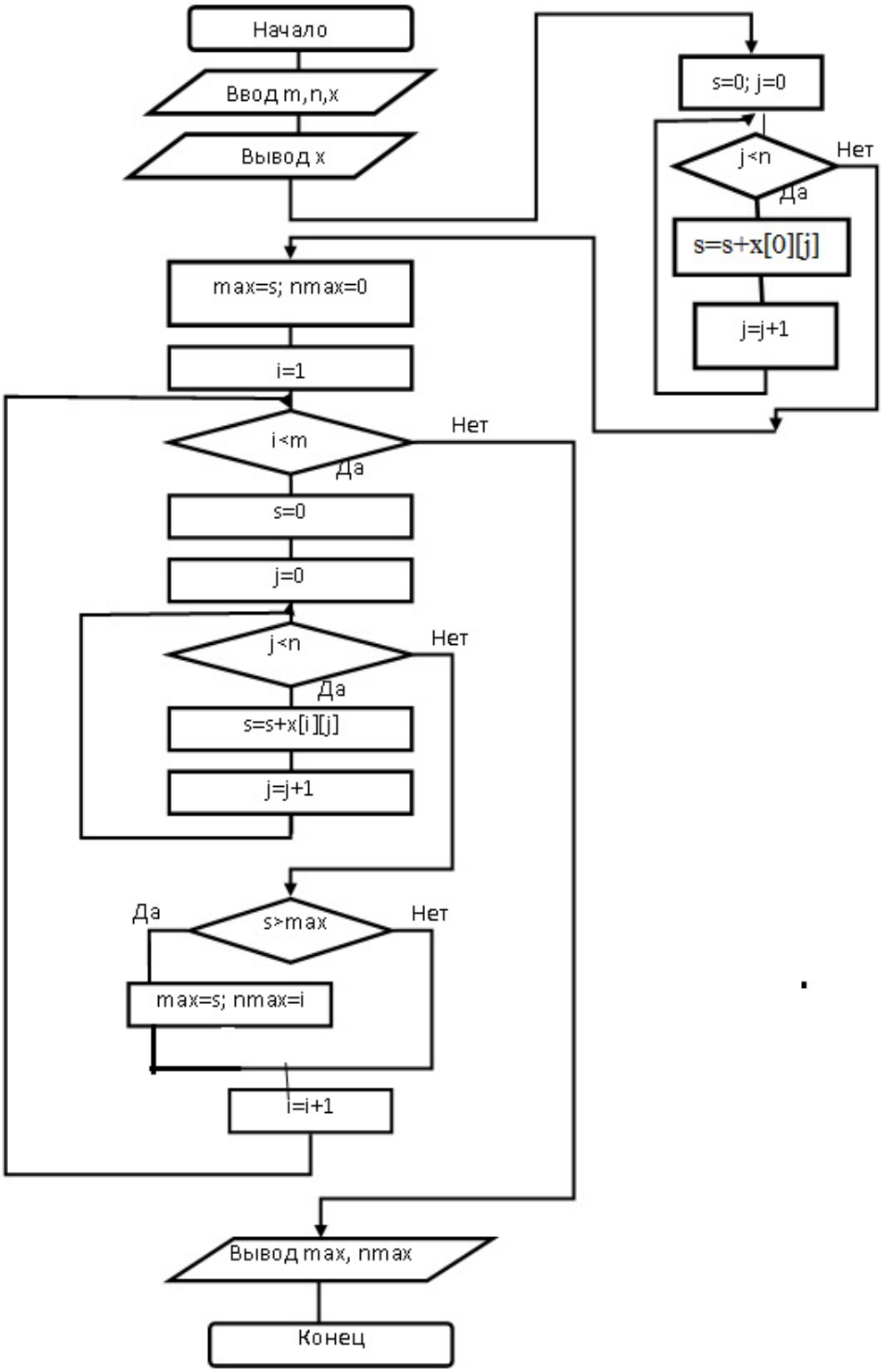


Рис. 6.25. Графическая схема алгоритма примера 6.2

Листинг программы:

```
/*  Строка с максимальной суммой*/
#include <stdio.h>
main()
{
 int m, n, nmax, i, j;
 float max, s, x[10][10];
 puts("Введите m, n");
 scanf("%d %d", &m, &n);

 for(i=0;i<m;i++)
   for(j=0;j<n; j++)
       { printf ("Введите x[%d][%d]= ", i, j);
         scanf ("%f", &x[i][j]);
       }

 puts("Матрица x: ");
 for(i=0;i<m; i++)
  { for(j=0;j<n; j++)
        printf("%8.2f",x[i][j]);
    printf("\n");
  }

 s=0;
 for(j=0; j<n; j++)
   s =s+x[0][j]; //Нач. знач. max – сумма элем. 0-ой стр.

 max=s;  nmax=0;
 for(i=0;i<m;i++)
   { s=0;
     for(j=0;j<n;j++)  s+=x[i][j];    // Сумма элем. строк
     if(max<s)
        { max=s;   nmax=i;}
   }

 printf("Максимальная сумма %.2f в строке %d\n",max,nmax);
 fflush(stdin);  getchar();
 return(0);
}
```

Пример 6.3. Определить количество строк матрицы, в которых суммы всех элементов отрицательные. Массив объявить как динамический.

Графическая схема алгоритма представлена на рис. 6.26.

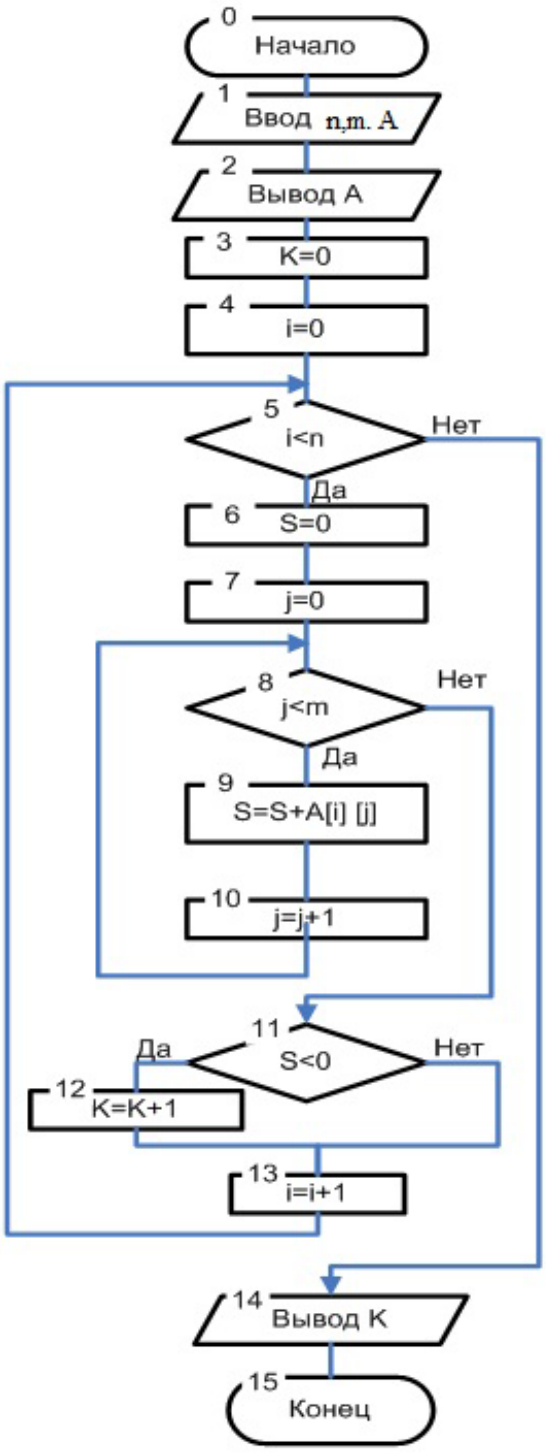


Рис. 6.26. Графическая схема алгоритма примера 6.3

Таблица 6.4

Таблица соответствия переменных

Переменные в задаче	Имя на языке Си	Тип	Комментарий
S	S	float	Сумма элементов *i*-й строки
A	A	float	Двумерный динамический массив
K	K	float	Количество искомых строк
–	n	int	Количество всех строк в матрице
–	m	int	Количество столбцов
–	i	int	Номер строки
–	j	int	Номер столбца

Листинг программы:

```
/*Пример обработки двумерного динамического массива*/
 # include <stdio.h>
 # include <math.h>
 void main ( )
 {
  int i, j, m, n;
  float K, S; //Описание переменных
  puts ("введите n, m"); // Вывод сообщения
  scanf ("%d %d", &n, &m); // Ввод исх.числа строк и столбцов
  float**A=new float*[n];
  for(i=0; i<n; i++)
    A[i]=new float[m];
  for(i=0; i<n; i++)
     for(j=0; j<m; j++)
       {
         printf("Введите A[%d; %d]= ", i, j);
         scanf("%f", &A[i] [j]);
       }
   K=0;
   for(i=0; i<n; i++)
      {
          S=0;
          for(j=0; j<m; j++)
             S=S+A[i] [j];
          if(S<0) K=K+1;
```

```
        }
     printf("%f \n", K);
     for(i=0; i<n; i++) //Освобождение динамической памяти
       delete[]A[i];
    delete[]A;
  }
```

***Тесты*:**

1) $A = \begin{pmatrix} -3 & -2 & 2 & 6 & -3 \\ 6 & 7 & -1 & 20 & -4 \\ 4 & -2 & -3 & 6 & -1 \end{pmatrix}$ K=2.

2) $A = \begin{pmatrix} -4 & -2 & 4 & 6 & 4 \\ 3 & 5 & 7 & 2 & 0 \\ 5 & 0 & -2 & 9 & 0 \end{pmatrix}$ K=0.

Пример 6.4. Определить, есть ли в матрице столбец, содержащий хотя бы один нулевой элемент.

Решение. Введем две вспомогательные переменные *t* и *w*:

$$t = \begin{cases} 1, & \text{если столбец не найден;} \\ 0, & \text{если столбец найден.} \end{cases}$$

$$w = \begin{cases} 1, & \text{если в столбце не найден элемент, равный нулю;} \\ 0, & \text{если в столбце найден элемент, равный нулю.} \end{cases}$$

Графическая схема алгоритма представлена на рис. 6.27.

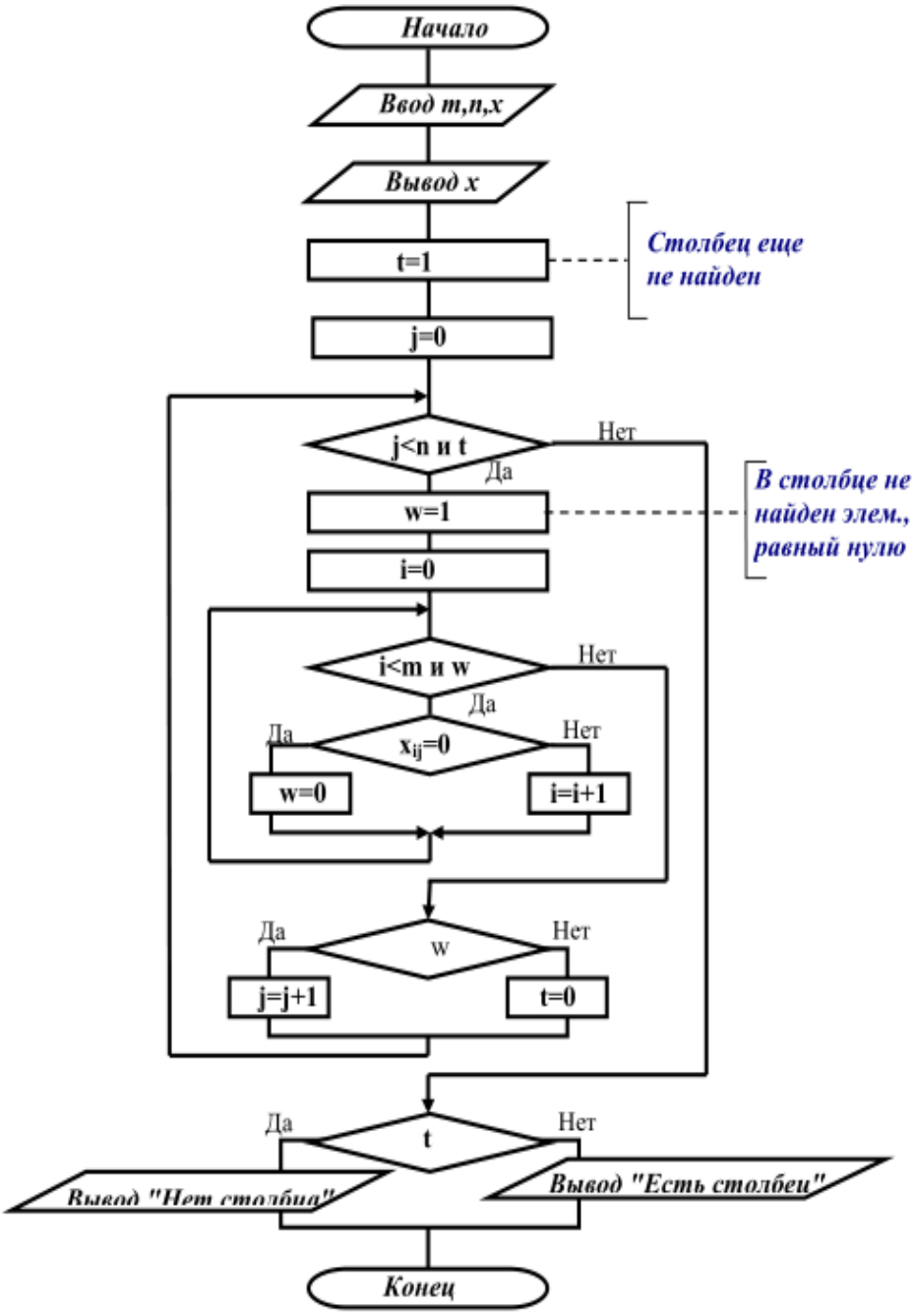


Рис. 6.27. Графическая схема алгоритма примера 6.4

Листинг программы:

```
#include <stdio.h>
#include <conio.h>
main()
{
  int m,n,i,j,t,w;
  float  x[10][10];
  puts("Введите m,n");
```

```
 scanf("%d %d",&m,&n);
 for(i=0;i<m;i++)
   for(j=0;j<n;j++)
       {
        printf("Введите x[%d][%d]= ",i, j);
        scanf("%f", &x[i][j]);
       }
 puts("Матрица x: ");
 for(i=0;i<m;i++)
  {
   for(j=0;j<n;j++)
        printf("%8.2f",x[i][j]);
   printf("\n");
  }
 t=1; j=0;   //Столбец еще не найден
 while(j<n && t)
  {
   w=1; i=0;  //В столбце не найден элем., равный нулю
   while(i<m && w)
       if(x[i][j]==0) w=0;
            else   i++;
   if(w)   j++;  // В столбце нет равного 0 элем.
     else  t=0;  // В столбце есть равный 0 элем.
  }
 if(t)  printf("Нет\n");
  else printf("Есть %d %d\n",i,j);
 getch();
 return(0);
}
```

7. ЗАПИСИ

7.1. Определение записи (структуры)

Предположим, что нужно написать программу для деканата факультета. Эта программа должна обрабатывать информацию о студентах, такую как фамилия студента, домашний адрес, домашний телефон, дата рождения, экзаменационные оценки, средний балл и др. Написать такую программу с помощью простых типов данных и массивов не так просто, хотя и возможно. Описание переменных для такой программы может выглядеть так:

```
char  fam[30];                  //Фамилия
```

```
  char  address[150];          // Домашний адрес
  char  phone[10];             // Телефон
  int     day;                 // День рождения
  int     month;               // Месяц рождения
  int     year;                // Год рождения
  int     oc[4];               // Оценки
  float  sr;                   // Средний балл
```

Если нужно обрабатывать данные сразу всех студентов, то нужно объявить не просто переменные, а массивы переменных. Конечно, этот способ работает. Более того, именно так (только на языке Fortran) были написаны многочисленные программы «Кадры» для ЕС ЭВМ. Но этот подход выглядит не эстетично, да и чреват большим количеством ошибок. Целостные понятия, такие как дата или человек, представлены набором не связанных между собой переменных. Отсутствует логическая и семантическая целостность данных.

В современных языках программирования логическое объединение целостных понятий называется *записью* (Pascal) или *структурой* (*C*/*C*++).

Структуры в *C*/*C*++ используются для логического и физического объединения данных произвольных типов, так же как массивы служат для группирования данных одного типа.

Определение структуры вводит новый пользовательский тип в программу, который может быть использован так же, как любой встроенный тип.

Например, в языке *C* можно так переписать приведенный выше фрагмент:

```
struct date
 {
   int     day;              // День
   int     month;            // Месяц
   int     year;             // Год
 };

 struct student
 {
   char  fam[30];             //Фамилия
   char  address[150];        // Домашний адрес
   struct date birth;         // Дата рождения
   char  phone[10];           // Телефон
   int     oc[4];             // Оценки
   float  sr;                 // Средний балл
```

```
};
struct student S1;
struct student S [100];
```

Последняя строка в примере показывает, как легко можно объявить массив структур.

Из табл. 7.1 видна разница в описании переменных.

Таблица 7.1

Таблица объявления переменных для обработки данных

Объявление переменных для обработки данных одного студента	**Объявление переменных для обработки данных одного студента (S1) и всех студентов (S[100])**

char fam[30]; //Фамилия
char address[150]; // Домашний адрес
char phone[10]; // Телефон
int day; // День рождения
int month; // Месяц рождения
int year; // Год рождения
int oc[4] // Оценки
float sr; // Средний балл
	struct date
	{
	int day; // День
	int month; // Месяц
	int year; // Год
	};
	
	struct student
	{
	char fam[30]; //Фамилия
	char address[150]; // Домашний адрес
	struct date birth; // Дата рождения
	char phone[10]; // Телефон
	int oc[4]; // Оценки
	float sr; // Средний балл
	};
	
	struct student S1;
	struct student S [100];

Объявление массивов для обработки данных всех студентов	

char Sfam[100] [30]; //Фамилии
char Saddress[100] [150]; //Домашние адреса
char Sphone[100] [10]; // Телефоны
int Sday[100]; // Дни рождения
int Smonth[100]; // Месяцы рождения
int Syear[100]; // Годы рождения
int Soc[100] [4] // Оценки
float Ssr[100]; // Средние баллы	

Работа со структурными переменными так же проста, например:

```
S1. birth. day=21;
S1.sr=(S1.oc[0]+ S1.oc[1]+ S1.oc[2]+ S1.oc[3])/4;
S[15].sr=S1.sr;
puts(S1.fam);
```

То, что поля структур сами могут быть структурами, позволяет целостным образом описывать достаточно сложные понятия реального мира. Так как структуры, в отличии от массивов, хранят данные различных типов, они и относятся к неоднородным типам данных.

Итак, *структура* (struct) или *запись* (record) – это составной тип данных, состоящий из фиксированного числа компонентов разного типа, называемых полями (Helds). Структуры целесообразно использовать тогда, когда необходимо объединить разные данные, относящиеся к одному объекту.

Описание структуры состоит из двух этапов:

1. Описание шаблона структуры.
2. Описание структурной переменной.

7.2. Описание шаблона структуры

Синтаксис описания шаблона структуры:

```
struct <имя_ шаблона>
 {
  <тип1>  <имя_поля1>;
  <тип2>  <имя_поля2>;
    …
  <типN>  <имя_поляN>;
 };
```

где <тип1>, <тип2>, … ,<типN> – любые основные типы (int, char, float, и т. д.), массив, указатель, структура, объединение;
<имя_поля1>, <имя_поля2>, … ,<имя_поляN> – допустимые идентификаторы языка.

В табл. 7.2 приведены примеры описания шаблона структуры.

Таблица 7.2

Примеры описания шаблона структуры

Пример 1	Пример 2

struct date //Имя шаблона – date
{
int day; // Имя поля1 – day
int month; // Имя поля2 – month
int year; // Имя поля3 – year
};
	struct student //Имя шаблона – student
	{
	char fam[30]; // Имя поля1 – fam
	char address[150]; // Имя поля2 – address
	struct date birth; // Имя поля3 – birth
	char phone[10]; // Имя поля4 – phone
	int oc[4]; // Имя поля5 – oc
	float sr; // Имя поля6 – sr
	};

7.3. Синтаксис описания структурной переменной

struct <имя_шаблона> <имя_переменной>;

или

struct <имя_шаблона> <список имен переменных>;

Примеры

struct student S1;
struct student S1, S2, S3;
struct student S [100];

7.4. Вложенные структуры

Определение структуры вводит новый пользовательский тип в программу, который может быть использован так же, как любой встроенный тип.

Структура называется вложенной, если хотя бы одно из ее полей является структурой.

Описание вложенной структуры состоит из двух этапов:

1. Описание вкладываемой структуры, т. е. создание нового типа данных.

2. Описание вложенной структуры.

Пример

```
struct date   //Вкладываемая структура
              //используется для описания поля birth структуры   student
{
  int    day;            // День
  int    month;          // Месяц
  int    year;           // Год
};

struct student          //Вложенная структура
                        //Одно из ее полей (birth) является структурой
{
  char  fam[30];          //Фамилия
  char  address[150];     // Домашний адрес
  struct date birth;      // Дата рождения
  char  phone[10];        // Телефон
  int    oc[4];           // Оценки
  float  sr;              // Средний балл
};
```

7.5. Доступ к отдельным полям структурной переменной

Обработка переменной, являющейся структурой, т. е. структурной переменной, сводится к обработке ее полей. При этом поле и сама переменная связываются с помощью ***составного имени***, имеющего следующий синтаксис:

<имя_структурной_переменной>.<имя_поля>

Здесь . – операция ***выбора*** или ***ссылки*** на поле (обычная точка).

В табл. 7.3 рассматриваются примеры обращения к полям структуры.

Таблица 7.3

Примеры обращения к полям структуры

Обращение к полю структуры	Комментарий
S1.fam	Фамилия студента S1
S1.oc[0]	Первая оценка студента S1
S1.adress	Адрес студента S1
S1.oc[i]	*i*-я оценка студента S1
S1. birth. day	День рождения студента S1
S[3].oc[2]	3-я оценка 4-го студента
S[57]. birth. year	Год рождения 58-го студента

7.6. Совмещение описания шаблона структуры и структурной переменной

Синтаксис описания:

<шаблон структуры><имя переменной или список имен переменных>;

Пример

```
struct date              //Описание шаблона структуры
  {
     int    day;          // День
     int    month;        // Месяц
     int    year;         // Год
  } d1,d2;                // Список имен переменных
```

7.7. Совмещение описания шаблона, описания структурных переменных и инициализации полей в одном предложении

Пример 1

```
struct date
  {
     int    day;              // День
     int    month;            // Месяц
     int    year;             // Год
  } d1, d2={2,3,1987};
```

Здесь описаны две переменные-структуры (d1 и d2). При этом сразу определены значения полей переменной d2:

```
d2. day=2;
d2. month=3;
d2. year=1987;
```

Т.о., дата d2 – это 2 марта 1987 года

Пример 2

```
struct student
  {
    char  fam[30];              //Фамилия
    char  address[150];    // Домашний адрес
    struct date birth;         // Дата рождения
    char  phone[14];         // Телефон
    int    oc[4];               // Оценки
    float  sr;                    / Средний балл
  }
     S1={ "Петров",
            "г. Жлобин, ул. Строителей, д.2, кв.4",
             {8,12,2009},
             "+3750232483580",
             {9,8,5,7},
             0
            };
```

7.8. Определение объема памяти, выделяемой под структурную переменную

Число байтов, выделенное под структурную переменную, не всегда равно сумме байтов для отдельных ее полей из-за некоторых особенностей работы процессора с данными с фиксированной и плавающей точкой, что приводит к так называемому «выравниванию», размещению элементов с четного адреса. Выделение памяти осуществляется по шаблону в последовательности объявления ее полей.

Для определения объема памяти, выделяемой под структурную переменную, используется стандартная функция **sizeof**, имеющая синтаксис описания:

sizeof (struct <имя_шаблона>);

Примеры

```
sizeof ( struct date );
sizeof ( struct student );
```

7.9. Копирование структур-переменных

С помощью оператора присваивания можно копировать значения одной структуры-переменной в другую при условии, что обе структуры-переменные имеют один и тот же тип. Например, пусть переменные a1, a2 описаны следующим образом:

```
struct complex
{
 float  re;  // Действительная составляющая комплексного числа
 float im;    //Мнимая составляющая комплексного числа
} a1, a2;
```

Пусть a1.re=3.5; a1.im=0.34; Тогда после выполнения оператора a2=a1 окажется, что a2.re равно 3.5, a2.im равно 0.34.

Однако следует учитывать, что при копировании структур-переменных оператор присваивания выполняет *поверхностное копирование*, суть которого состоит в следующем. Происходит копирование бит за битом значений полей переменной-источника (a1) и перенесение их в соответствующие поля переменной-цели (a2). При этом может возникнуть проблемы с такими полями данных, как указатели. Поэтому использовать поверхностное копирование структур надо осторожно.

7.10. Структурные переменные и указатели

Для любого типа T тип T* означает «указатель на объект типа T». Указатель является объектом (переменной), хранящим адрес другого объекта. В случае, если указатель используется для хранения адреса объекта типа структуры, то доступ к полям структуры может быть осуществлен двумя способами.

Предположим, что в программе сделано описание:

```
struct vec
   {
       double x, y;
    };
vec v1, *v2;
```

Здесь v1 структура типа vec, v2 – указатель на структуру типа vec. Для выбора полей *x*, *y* структуры необходимо использовать конструкции: (*v2).x, (*v2).y. Действительно, v2 – адрес структуры, *v2 – сама запись. Круглые скобки здесь необходимы, так как точка (.) имеет более высокий приоритет, чем звездочка (*).

Для доступа к полям структурной переменной через указатель структуры можно использовать специальную операцию -> (минус больше). Например, v2->x, v2->y.

Таким образом, конструкции (*v2).x, (*v2).y и v2->x, v2->y представляют собой разные способы обращения к полям структуры.

7.11. Массивы структурных переменных

Описание массива структур не отличается от описания массива обычных переменных.

Пример

```
Struct Man
  {
    char   fio[31];          // ФИО
     int   year;             // Год рождения
     float pay;              // Оклад
  };
Man d[100], *p=d;     // массив структур d из 100 элементов и
                      // указатель p, инициализированный адресом
                      //первого элемента массива d
```

Примеры обращения к полям:
d[i].year, (*(d+i)).year, (d+i)->year, (p+i)-> year

7.12. Пример программы работы со структурами

```
#include <stdio.h>
main()
{
/*        Описание шаблона структуры          */
 struct Student
 { char *fio;              // Фамилия       - указатель на char
  char  Adress[40];    // Адрес           - строка
  int     Age;             // Возраст        - целое
  int     oc[4];           // Оценки         - целочис. массив
        float   sr;              // Средний балл – вещественное
 } ;
 struct Student S;          // Описание структурной переменной S
 int i;
 float sr;
 S.fio="Петушков";       //Присваивание полю значения
 printf("Введите адрес студента %sa",S.fio);
 gets(S.Adress);             //Ввод значения поля
 S.Age=1987;                 //Присваивание полю значения
 S.oc[0]=3;
 S.oc[1]=5;
 S.oc[2]=4;
 S.oc[3]= S.oc[2];
 sr=0;                                          //Вычисление
 for(i=0;i<=3;i++)                              //среднего
       sr=sr+S.oc[i];
 sr=sr/4;                                       //балла
 S.sr=sr;                              //Присваивание полю значения
 printf("Средний балл студента %sa",S.fio);
 printf("%d года рождения,\n проживающего по адресу: %s,",
                        S.Age,S.Adress);
 printf(" равен %5.2f\n",S.sr);
 fflush(stdin);  getchar();  return(0);
}
```

Вид экрана после выполнения программы:
Введите адрес студента Петушкова ул. Солнечная, д.1, кв. 4
Средний балл студента Петушкова 1987 года рождения,
проживающего по адресу: ул. Солнечная, д.1, кв. 4, равен 4.00

7.13. Поиск в массиве структур, вводимых с клавиатуры

Задача 7.1. Ввести данные об *n* сотрудниках некоторой фирмы: *фамилия*, *год рождения*, *оклад*. Вывести список сотрудников в форме таблицы и фамилии сотрудников, имеющих оклад выше среднего.

Используемые переменные:
n – число сотрудников;
db – массив записей с данными о сотрудниках;
s – сумма окладов сотрудников;
sr – средний оклад сотрудников;
fio – имя поля "фамилия";
pay – имя поля "оклад";
year – имя поля "год рождения";
lfio – длина поля "фамилия":
lpay – длина поля "окладм;
lyear – длина поля "фамилиям;
ldb – размер массива db.

Графическая схема алгоритма решения задачи 7.1 представлена на рис. 7.1.

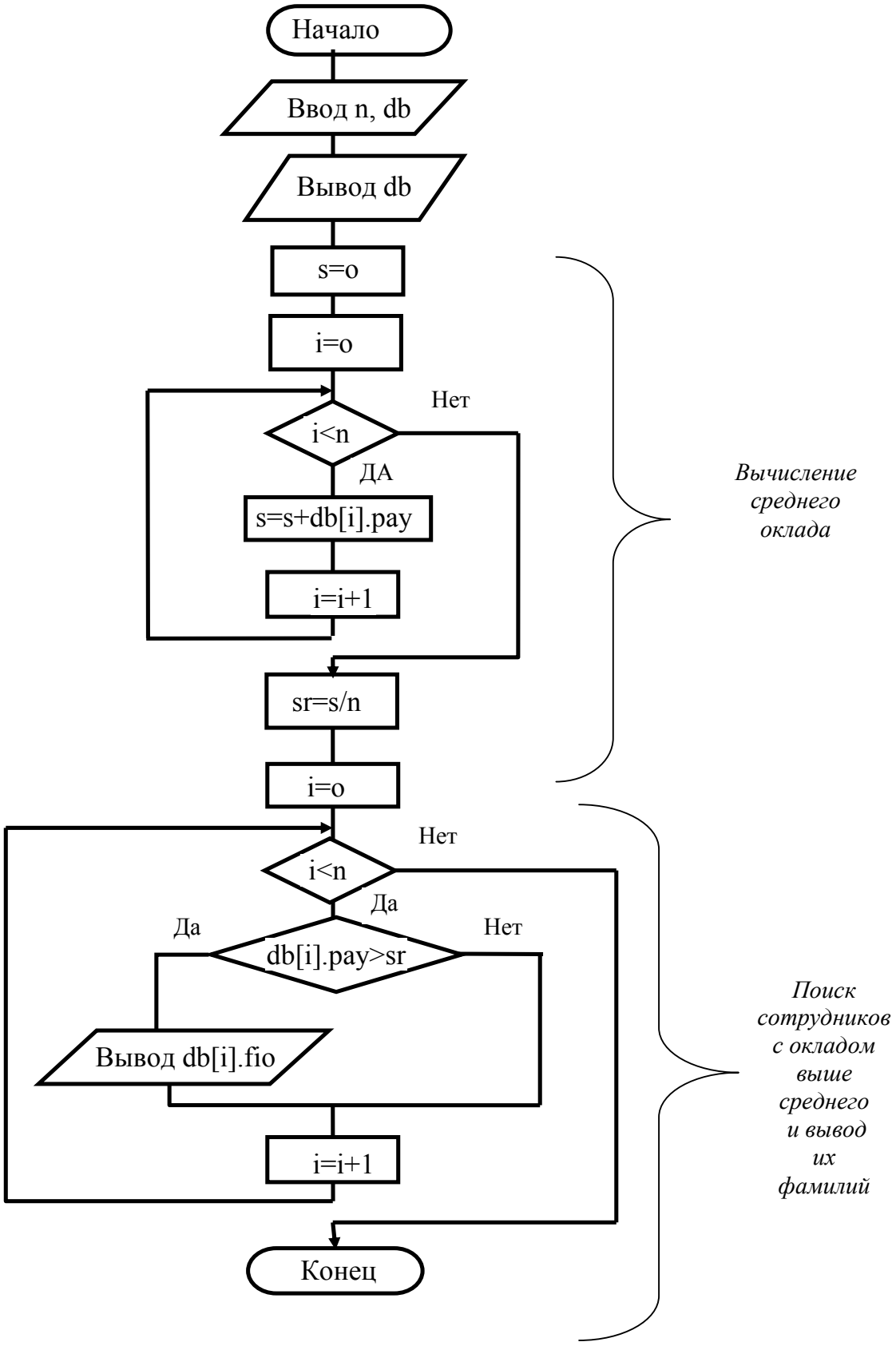


Рис. 7.1. Графическая схема решения задачи 7.1

Листинг программы:

```
/*   Фамилии сотрудников, имеющих оклад выше среднего  */
#include <stdio.h>
#include <string.h>
main()
{
 const int lfio=20, lyear =5, lpay =7,     //длины полей фио, г.рожд., оклада
           ldb=100;
 struct Man
     {  char   fio[lfio+1];     // фио
        int    year;           // год рожд.
        float  pay;            // оклад
     };
 Man db[ldb];                  // массив структур
 int i, n;
 float s;
 puts("Число записей?(1<n<=10)");
 scanf("%d",&n);
 for(i=0; i<n; i++)
   {
    puts("Фамилия?");
    gets(db[i].fio);
    puts("Год рождения?");
    scanf("&d",&db[i].year);
    puts("Оклад?");
    scanf("&f",&db[i].pay);
   }
 puts("          Список сотрудников");
 puts(" ┌──────────────────────┬─────┬─────────┐ ");
 puts(" │                      │     │         │ ");
 puts(" ├──────────────────────┼─────┼─────────┤ ");
 for(i=0; i<n; i++)
   printf("|%-20s|% 5d | % 7.2f |\n", db[i].fio, db[i].year, db[i].pay);
 puts(" └──────────────────────┴─────┴─────────┘ ");
 puts("Фамилии сотрудников, имеющих оклад выше среднего");
//Вычисление среднего оклада
 s=0;
 for(i=0; i<n; i++)
```

```
        s+=db[i].pay;    //s=s+ db[i].pay;
      s/=n;              //s=s/n;
     // Поиск сотрудников с окладом выше среднего и вывод их фамилий
      for(i=0;i<n;i++)
        if(db[i].pay>s)  printf("%-s\n",db[i].fio);
      fflush(stdin);  getchar();
      return 0;
     }
```

8. ФАЙЛЫ В ЯЗЫКЕ *C*

8.1. Общие понятия

Под файлом понимается поименованная область внешней памяти ПК (жесткого диска, электронного «виртуального диска», …), хранящая данные. Под файлом понимается также логическое устройство – потенциальный источник или приемник информации.

Любой файл имеет следующие характеристики (атрибуты):

• *Имя файла*. Составляется по правилам составления идентификаторов в рассматриваемой ОС, например, C:\MCDOC\d.txt.

• *Тип компонентов*. Например, файл может представлять собой последовательность строк или последовательность байтов.

• *Длина файла*. Это число компонент файла.

• *Указатель файла*. Это переменная специального типа, предназначенная для указания на компонент (позицию) файла. Значение указателя файла изменяется после каждого выполнения операции чтения или записи данных.

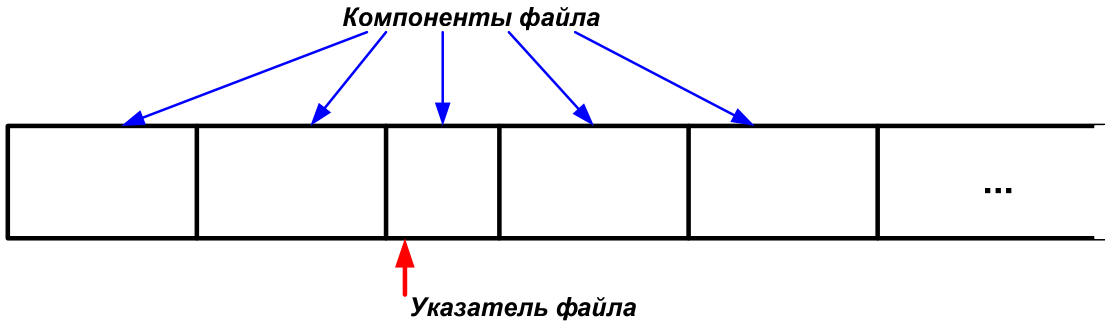


Рис. 8.1. Указание на компонент (позицию) файла

Стандартные аппаратные устройства ПК, такие как клавиатура, экран дисплея, печатающее устройство (принтер) и коммуникационные каналы ввода-вывода, определяются специальными именами, которые называются *логическими устройствами*. К ним относятся:

– con – логическое имя, которое определяет консоль (клавиатуру или экран дисплея);

– prn – логическое имя принтера. Если к ПК подключено несколько принтеров, доступ к ним осуществляется по логическим именам *LPT1*, *LPT2*, *LPT3*. Первоначально *prn* и *LPT3* – синонимы;

– aux – логическое имя коммуникационного канала, который используется для связи ПК с двумя машинами. Коммуникационный канал может осуществлять передачу и прием данных. Как правило, имеется 2 коммуникационных канала: *com1* и *com2*. Первоначально *aux* и *com1* – синонимы;

– *NULL* – логическое имя «пустого» устройства. Чаще всего используется в отладочном режиме как устройство-приемник информации неограниченной емкости. При обращении к *NULL* как к источнику информации выдается признак конца файла (EOF).

При вводе:выводе данные рассматриваются как *поток байтов*.

Поток – это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику, таким образом, физически ***поток*** в *C* – это файл или устройство. Чтение данных из потока называется *извлечением*, вывод в поток – *помещением* или *включением*.

Обмен с потоками для увеличения скорости передачи данных производится, как правило, через специальную область ОП – *буфер*. При выводе фактическая передача данных происходит после заполнения буфера. При вводе фактическая передача данных происходит, если буфер исчерпан.

По направлению обмена потоки можно разделить на *входные* (данные вводятся в память), *выходные* (данные выводятся из памяти) и *двунаправленные* (происходит как извлечение, так и включение).

По виду устройств, с которыми работает поток, можно разделить потоки на *стандартные*, *файловые* и *строковые*.

Стандартные потоки предназначены для передачи данных от клавиатуры в память ПК, или из памяти ПК на экран дисплея и принтер.

Файловые потоки предназначены для обмена информацией с файлами на внешних носителях данных.

Строковые потоки – для работы с массивами символов в ОП.

По структуре данных поток может быть *текстовым* или *бинарным* (двоичным).

Текстовый поток – это последовательность строк, каждая из которых имеет нуль или более ASCII-символов и заканчивается символом «\n» – конца строки и перехода к следующей строке. Текстовые файлы могут быть просмотрены и отредактированы с клавиатуры любым текстовым редактором.

Бинарный (*двоичный*) поток – это последовательность байтов без учета разделения на строки. Каждая программа для своих бинарных файлов определяет собственную структуру.

Для работы с потоками существуют стандартные библиотеки *stdio*, *string*, *stdlib*, *io*, которые становятся доступными из функций пользователя после использования директивы препроцессора *#include*, например, *#include <stdio.h>*.

8.2. Работа с файлами (потоками)

8.2.1. Этапы работы с файлами (потоками)

Файл в программе на языке *C* – это переменная-указатель на тип *FILE*, называемая *файловой переменной*.

Работа с файлами состоит их трех этапов:

1. Открытие файла.
2. Обработка файла.
3. Закрытие файла.

8.2.2. Открытие файла (потока)

Поток можно открыть для чтения, записи и для *чтения и записи* с помощью стандартной функции *fopen*, прототип которой имеет вид:

*FILE *fopen (const char *filename, const char*mode);*

Здесь первый параметр функции (*filename*) – имя открываемого файла в виде строки символов, второй (*mode*) – режим открытия файла (тип доступа к файлу) – строка из одного и более символов:

"*r*" – файл открывается для чтения;

"*w*" – открывается пустой файл для записи (если файл существует, то его содержимое теряется);

"*a*" – файл открывается для добавления информации в конец файла; если файла нет, он создается;

"*r+*" – файл открывается для чтения и записи (файл должен существовать);

"*w+*" – открывается пустой файл для записи и чтения (если файл существует, то его содержимое теряется);

"*a+*" – файл открывается для чтения и добавления данных в конец файла;

"*t*" – открытие файла в текстовом режиме;

"*b*" – открытие файла в двоичном режиме.

Примечание. По умолчанию (когда в режиме открытия файла отсутствует *t* или *b*) файл открывается в *текстовом режиме*.

Возможны следующие режимы доступа: "*w+b*", "*wb*", "*rw+*", "*w+t*", "*rt+*" и др.

При успешном открытии потока функция *fopen* возвращает указатель на структуру типа *FILE*. Эта структура связана с физическим файлом и содержит всю необходимую информацию для работы с ним (указатель на текущую позицию в файле, тип доступа и др.).

Возвращаемое функцией значение нужно сохранить и использовать для ссылки на открытый файл.

Если произошла ошибка при открытии файла, то возвращается *NULL*.

Таким образом, для работы с функцией *fopen* в программе на языке *C* необходимо:

1. Объявить переменную-указатель на структуру типа *FILE*, например:

FILE f;*

Эта переменная (*f*) называется файловой переменной или файлом.

2. Файловой переменной присвоить значение, возвращаемое функцией *fopen*, например:

f=fopen("c:\mdoc\d.txt", "r+");

Примечание. Описание и инициализация файловой переменной (например, *f*) могут быть осуществлены в одном операторе, например:

*FILE*f = fopen ("c:\mdoc\d.txt", "r+");*

После открытия файла все действия над файловой переменной будут отождествляться с действиями над внешним файлом (физическим файлом), т. е. с файловой переменной связывается определенный физический поток. При этом структуре типа *FILE* выделяется область ОП, адрес которой возвращает функция *fopen*. Поток связывается со структурой типа *FILE*.

Структура типа *FILE* используется функциями ввода/вывода для хранения информации, связанной с устройством или файлом. Указатель на тип *FILE* используется для операций с файлом с помощью библиотечных функций. Его передают библиотечным функциям в качестве параметра. Такие библиотечные функции называются функциями ввода/вывода.

Примечания:

1. При открытии потока с ним связывается область памяти – буфер.

2. При аварийном завершении программы выходной буфер может быть не выгружен, поэтому возможна потеря данных.

3. С помощью функций *setbuf* и *setvbuf* можно управлять размерами и наличием буферов.

4. Перед началом работы программы операционная система автоматически открывает пять стандартных потоков:

1) *stdin* – стандартный ввод;

2) *stdout* – стандартный вывод;

3) *stdprn* – стандартная печать;

4) *stderr* – стандартный вывод сообщений об ошибках;

5) *stdaux* – стандартный дополнительный поток (стандартные порты).

Потоки 1), 2), 4) относятся к консоли (*con*), 3) – к *prn*, 5) – к *aux* (коммуникационный канал), т. е. в начале работы программы автоматически открываются потоки следующими операторами:

1) *FILE *stdin=fopen("con", "r");*

2) *FILE *stdout=fopen("con", "w");*

3) *FILE *stdprn=fopen("prn", "w");*

4) *FILE *stderr=fopen("con", "w");*

5) *FILE *stdaux=fopen("aux", "wb");*

Файлы *stdin* и *stdout* можно переназначать при запуске *exe*-программы или в программе с помощью функции *freopen*. Ее прототип:

FILE **freopen*(*const char *filename*, *const char *mode*, *FILE *stream*);

где *freopen* – имя функции; *filename* – имя файла; *mode* – режим работы файла; *stream* – поток.

Функция *freopen* работает аналогично функции *fopen*, но предварительно закрывает поток *stream*, если тот был ранее открыт.

8.2.3. Закрытие файла (потока)

Для закрытия потоков, используемых в программе, применяются стандартные функции *fclose* и *fcloseall*.

Один заданный поток закрывается функцией *fclose*. Ее прототип:

int fclose (*FILE *stream*);

где *stream* – поток.

Пример обращения к функции *fclose*:

fclose (*f*);

Если поток успешно закрыт, функция *fclose* возвращает значение 0. Если при закрытии потока произошла ошибка – значение *EOF* (–1).

Функция *fcloseall* закрывает все потоки, открытые с помощью *fopen*, кроме *stdin*, *stdout*, *stderr*. Прототип функции *fcloseall*:

int fcloseall();

Функция *fcloseall* возвращает количество закрытых потоков или значение *EOF* (–1), если при закрытии произошла ошибка.

Примечание. Функция *perror* позволяет вывести сообщение об ошибке при неуспешном закрытии файла. Ее прототип:

void perror(*const char *s*);

Пример

```
FILE * fp;
fp=fopen("d.dat", "r");
if(!fp) perror("Нельзя открыть файл для чтения");
   else fclose ("d.dat");
```

8.2.4. Удаление файла

Удалить файл можно с помощью функции *remove*. Ее прототип:

int remove (*const char *filename*);

где *filename* – указатель на строку с именем физического файла (имя файла).

Функция *remove* возвращает значение 0 при успешном удалении файла и не нуль в противном случае. Открытый файл необходимо предварительно закрыть.

8.3. Пример открытия и закрытия файла

В текстовый файл с именем ***rez.txt*** записываются результаты выполнения программы в начало нового файла, если файл не существует, или в конец существующего файла.

```
#include <stdio.h>
void main ()
{
  int n;  //номер теста
  float y,x;
  FILE *f;          //Описание файловой переменной f (файла)
  puts(“Введите x”);
  scanf(“%f”,&x);
  y=2*x;
  f=fopen("rez.txt", "a");  //Открытие файла
```

```
  puts("Номер теста ? ");
  scanf("%d",&n);
  fprintf(f, "Тест N %d\n",n);  //Запись в файл f значения
  fprintf(f, "%f %f\n",x,y);
  fclose(f);                    //Закрытие файла
 fflush(stdin);  getchar();
}
```

В результате первого выполнения программы в текущем каталоге будет создан файл с именем ***rez.txt***. В его начало запишутся значения ***n***, ***x***, ***y***. При последующих запусках программы на выполнение значения ***n***, ***x***, ***y*** будут записываться в конец файла ***rez.txt***, на что указывает режим "***a***" открытия файла в функции *fopen*. Отсутствие символа "***t***" или "***b***" в режиме открытия файла «говорит» о работе с текстовым файлом (но не расширение ***txt*** в имени файла).

8.4. Ввод-вывод в поток

8.4.1. Основные понятия

Ввод-вывод в поток можно осуществить разными способами:

– в виде последовательности байтов;

– в виде символов и строк;

– с использованием форматных преобразований.

Операции ввода-вывода выполняются, начиная с текущей позиции потока. Текущая позиция определяется положением ***указателя потока***. Указатель потока устанавливается на начало или конец файла при открытии в соответствии с режимом открытия. После каждой операции ввода-вывода указатель потока автоматически изменяется.

8.4.2. Позиционирование в файле

8.4.2.1. Функции получения текущего положения указателя потока ftell и fgetpos

Прототип функции ***ftell*** (из <stdio.h>):

***long int ftell(File *f)*;**

Функция ***ftell*** возвращает текущую позицию в файле, связанном с потоком ***f***, как длинное целое. В случае ошибки возвращает число (–1).

Прототип функции ***fgetpos*** (из <stdio.h>):

***long int fgetpos (File *f, fpos_t *pos)*;**

Функция ***fgetpos*** возвращает текущую позицию в файле, связанном с потоком ***f***, и копирует значение текущей позиции по адресу ***pos***. Это значение позднее может использоваться функцией ***fsetpos***. Возвращаемое значение имеет тип ***fpos_t***, который используется функциями ***fgetpos*** и ***fsetpos*** для хранения текущей позиции файла:

typedef long ***fpos_t***.

8.4.2.2. Функции задания положения указателя fseek и fsetpos, rewind

Прототип функции ***fseek***:

int fseek (File *f, long off, int org);

Функция перемещает текущую позицию в файле, связанном с потоком ***f***, на позицию ***off***, отсчитываемую от значения ***org***, которое должно быть равно одной из констант, определенных в <***stdio.h***>:

SEEK_CUR – от текущей позиции указателя (1);

SEEK_END – от конца файла (2);

SEEK_SET – от начала файла (0).

Параметр ***off*** задает количество байтов, на которое необходимо сместить указатель соответственно параметру ***org***. Величина смещения может быть как положительной, так и отрицательной, но нельзя сместиться за пределы начала файла.

Такой доступ к данным в файле называется произвольным.

Прототип функции ***fsetpos***:

int fsetpos (File *f, const fpos_t *pos);

Функция перемещает текущую позицию в файле, связанном с потоком ***f***, на позицию *pos, предварительно полученную с помощью функции ***fgetpos***.

Примечание. Функции ***fseek*** и ***fsetpos*** нельзя использовать для стандартных потоков.

Функция ***rewind*** очищает флаги ошибок при работе с потоками и устанавливает указатель на начало файла. Ее прототип:

void rewind(File *f);

8.4.3. Функции чтения и записи потока байтов *fread* и *fwrite*

Прототип функции ***fread***:

size_t fread (***void *buffer***, ***size_t size***, ***size_t count***, ***FILE *stream***);

Функция ***fread*** считывает ***count*** элементов длиной ***size*** байтов в область, заданную указателем ***buffer***, из потока ***stream***. Возвращает количество прочитанных элементов, которое может быть меньше ***count***, если при чтении произошла ошибка или встретился конец файла.

Прототип функции ***fwrite***:

size_t fwrite (const ***void *p***, ***size_t size***, ***size_t n***, ***FILE *f*)*;***

Функция записывает ***n*** элементов длиной ***size*** байтов из буфера, заданного указателем ***p***, в поток ***f***. Возвращает число записанных элементов.

***Примеры*:**

1. Записать в бинарный файл целочисленный массив из *n* элементов.

Пусть описан массив *A* следующим образом:

```
int A[100];
```

Запишем в бинарный файл *f* первые *n* элементов этого массива:

fwrite (A, ***sizeoff(int)***, ***n***, ***f*)*;***

Файл *f* может иметь описание:

```
FILE*f=fopen("b1.dat", "wb");
```

2. Записать в бинарный файл массив записей.

```
struct train
{ int ntrain;
  char s[50];
  char v1[5];
  char v2[5];
}t[20];
FILE*f=fopen("b2.dat", "wb");
```

fwrite (*t*, ***sizeoff(***struct train***)***, ***n***, ***f*)*;***

3. Прочитать из бинарного файла третий элемент массива *t*.

```
fseek (f, 2* sizeoff(struct train), 0);
struct train g;
```

fread (***g***, ***sizeoff(***struct train***)***, ***1***, ***f*)*;***

```
printf("Номер третьего поезда:%d\n", g.ntrain);
```

8.4.4. Функции чтения символа из потока (*getc*, *fgetc*, *getchar*)

Функция ***getc***, имеющая прототип:

int getc (FILE *f*)*;

возвращает очередной символ в формате ***int*** из стандартного потока ***f***. Если символ не может быть прочитан, то возвращает значение ***EOF***.

Функция ***fgetc***, имеющая прототип:

int fgetc (FILE *f);

возвращает очередной символ в формате ***int*** из потока ***f***. Если символ не может быть прочитан, то возвращает значение ***EOF***.

Функция ***getchar***, имеющая прототип:

int getchar (void);

возвращает очередной символ в формате ***int*** из стандартного потока (*stdin*). Если символ не может быть прочитан, то возвращает значение ***EOF***.

8.4.5. Функции записи символа в поток (*putc*, *fputc*, *putchar*)

Функция ***putc*** записывает символ ***ch*** в поток ***f***. При ошибке возвращает значение ***EOF***. Ее прототип:

int putc (***int ch***, ***FILE *f***);

Выполняется ***fputc*** аналогично функции ***putc***.

Ее прототип:

int fputc (***int ch***, ***FILE *f***);

Функция ***putchar()*** выводит символ ***ch*** на стандартное устройство вывода, добавляя в конце символ новой строки. Возвращает неотрицательное значение при успехе или *EOF* – при ошибке.

8.4.6. Функции чтения строки из потока (*fgets*, *gets*)

Прототип функции ***fgets***:

char *fgets (***char *s***, ***int n***, ***FILE *f***);

Функция читает не более ***n-1*** байт из потока ***f*** и помещает прочитанные байты в строку ***s***, прекращая чтение при обнаружении символа новой строки или конца файла. Символ новой (**'\n'**) строки при чтении не отбрасывается, помещается в конец строки ***s***. Прочитанная строка дополняется ограничителем строки (**'\0'**). При обнаружении ошибки или конца файла возвращает ***NULL***, в противном случае – указатель на строку ***s***.

Функция ***gets*** читает символы с клавиатуры до появления символа новой строки и помещает их в строку ***s***. Сам символ новой строки в строку не включается. Возвращает указатель на ***char***.

Прототип функции ***gets***:

char *gets (***char *s***);

8.4.7. Функции записи строки в поток (*fputs*, *puts*)

Прототип функции ***fputs***:

int fputs (***const char * s***, ***FILE *f***)**;**

Функция ***fputs*** записывает строку символов ***s*** в поток ***f***. Символ конца строки в файл не записывается. При ошибке возвращает значение ***EOF***, иначе – неотрицательное число.

Функция ***puts*** выводит строку ***s*** на стандартное устройство вывода, добавляя в конце символ новой строки. Возвращает неотрицательное значение при успехе или ***EOF*** – при ошибке.

Прототип функции ***puts***:

int puts (***char *s***)***;***

8.4.8. Функции форматированного ввода (чтения) из потока (*fscanf* , *scanf*, *sscanf*)

Прототип функции ***fscanf***:

int fscanf (***FILE *f***, ***const char *fmt*** [, ***par1***, ***par2***, …])***;***

Эта функция вводит (читает) строку параметров ***par1***, ***par2***, … в формате, определенном строкой ***fmt***, из потока (файла) ***f***. Возвращает число переменных, которым присвоено значение.

Прототип функции ***scanf***:

int scanf (***const char *fmt*** [, ***par1***, ***par2***, …])***;***

Функция ***scanf*** вводит (читает) строку параметров ***par1***, ***par2***, … в формате, определенном строкой ***fmt***, со стандартного устройства ввода (обычно с клавиатуры (***stdin***)). Возвращает число переменных, которым присвоено значение.

Прототип функции ***sscanf***:

int sscanf (***const char *buf***, ***const char *fmt*** [, ***par1***, ***par2***, …])***;***

Функция ***sscanf*** вводит данные аналогично функции ***scanf***, но не с клавиатуры, а из строки символов, переданной ей первым параметром. Аргумент ***buf*** – строка символов, из которой вводятся значения, ***fmt*** – строка формата, в соответствии с которой происходит преобразование данных. Многоточие указывает на наличие необязательных аргументов, соответствующих адресам вводимых значений.

8.4.9. Функции форматированного вывода в поток (*fprintf*, *printf*, *sprintf*)

Прототип функции ***fprintf***:

int fprintf (***FILE *f***, ***const char *fmt***, **…**)**;**

Функция записывает в поток ***f*** значения переменных, список которых обозначен многоточием (**…**), в формате, указанном строкой ***fmt***. Возвращает число записанных значений.

Прототип функции ***printf***:

int printf (***const char *fmt***, **…**);

Функция выводит на стандартное устройство вывода (***stdout***) значения переменных, перечисленных в списке, обозначенном многоточием (**…**), в формате, определенном строкой ***fmt***. Возвращает число выведенных значений.

Прототип функции ***sprintf***:

int sprintf (***char *buf***, ***const char *fmt***, **…**);

Функция выводит в строку ***buf*** значения переменных, перечисленных в списке, обозначенном многоточием (**…**), в формате, определенном строкой ***fmt***.

Примечание. Спецификации формата ***fmt*** были рассмотрены в разделе «Консольный ввод-вывод».

8.5. Обработка ошибок

Функции работы с потоками возвращают значения, которые рекомендуется анализировать в программе и обрабатывать ошибочные ситуации, возникающие, например, при открытии файлов или чтении из потока. При работе с файлами часто используют функции ***feof*** и ***ferror***.

Прототип функции ***feof***:

int feof (***FILE *f***);

Функция возвращает ***EOF*** или значение, отличное от 0, если достигается конец файла; в противном случае 0.

Прототип функции ***ferror***:

int ferror (***FILE *f***);

Возвращает не равное нулю целое значение, означающее код ошибки, если обнаружена ошибка ввода/вывода; 0 – в противном случае.

8.6. Пример обработки текстового файла

В текстовом файле «dbase.txt» хранятся данные о сотрудниках фирмы. В каждой строке файла указана фамилия, год рождения, оклад сотрудника. Для простоты обработки файла данные записаны единообразно: первые 15 символов занимает фамилия, следующие 5 – год рождения, последние 10 – оклад.

Требуется, интерпретируя структурой строки файла, вывести на экран или принтер содержимое файла в виде таблицы, создать из строк файла с данными о сотрудниках фирмы моложе 20 лет массив структур. Вывести на экран или в текстовый файл полученный массив или вывести сообщение о том, что такие молодые сотрудники не работают в фирме.

Структура записи файла изображена на рис. 8.2.

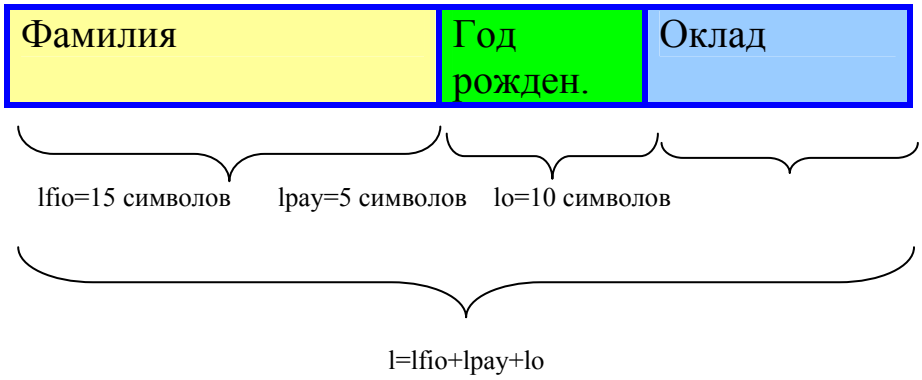


Рис. 8.2. Структура записи файла

Записи файла могут иметь вид:

Иванов И.П. **1982 453120**
Авчинникова **1999 578320**
Васильков **1955 456780**
Чернов **1967 1345600**

Требуется на экран, на принтер или в текстовый файл вывести список в форме таблицы. Например:

Список сотрудников фирмы

Фамилия	**Год рожд.**	**Оклад**
Иванов И.П.	1982	453120
Авчинникова	1999	578320
Васильков	1955	456780
Чернов	1967	1345600

```
/*   Поиск в массиве структур, читаемых из текстового файла */
#include <stdio.h>    // с записью рез-тов в новый текстовый файл
#include <string.h>
#include <stdlib.h>
// #include <windows.h>
main()
{
  const int lfio=15,            //длина поля фио,
      lpay=5,                   // длина поля г.рожд
      lo=10,                    //длина поля оклада
```

```
        l=lfio+lpay+lo;          //длина записи в файле
      struct Man
        {  char   fio[lfio];       // фио
           int    year;            // год рожд.
           float  pay;             // оклад
        };
      Man db;
      char  s[l];                  //строка для записи файла
      FILE *fin,                  // Исх. файл
        *fo,*f1 ;                 // Вых. Файлы
      fin =fopen("dbase.txt", "r");
      if  (fin==NULL)
          puts("Ошибка открытия файла\n");
     else
     {
      puts("Файл открыт");
      fo=fopen("dbout.txt","w");
      f1=fopen("dbout1.txt","w");
      while (!feof(fin))
     {          fgets(s,l,fin);  ┐
puts(s);                         │                  fscanf(fin,"%s%d%f",&db.fio,
                                 ├ или     &db.year, &db.pay);
strncpy(db.fio,s,lfio-1);        │
          db.fio[lfio-1]='\0';   ┘

db.year=atoi(&s[15]);

db.pay=atof(&s[20]);
          if(db.year>2000)
            fprintf(fo,"%-15s %10.1f\n", db.fio, db.pay);
          fputs(s,f1);
        }
      fclose(fin); fclose(fo); fclose(f1);   //   или  fcloseall();
     }
      fflush(stdin); getchar();
      return 0;
     }
```

8.7. Пример обработки текстового файла и бинарного файла

/* Построчное считывание данных из текстового файла ***dbase.txt*** в буферную переменную ***s***, формирование из них структуры ***db*** и запись ее в двоичном режиме в выходной файл "***dbout.dat***".

Считывание из двоичного файла записи с номером *i* и вывод ее на экран.

Считывание из двоичного файла записей и вывод на экран только тех записей, для которых фамилия есть "***ivanoff***". */

Таблица 8.1

Таблица соответствия переменных

№ п/п	Идентификатор	Тип	Комментарий
1	lfio	const int	Длина поля фио
2	lyear	const int	Длина поля года рождения
3	lo	const int	Длина поля оклада
4	l	const int	Длина записи
5	db	Запись	Запись
6	s	Строка	Строка с содержанием записи
7	fin	Текстовый файл	Исходный текстовый файл
8	fo	Двоичный файл	Двоичный файл, полученный из fin
9	kol	int	Кол-во записей файла fin
10	i	int	Номер записи файла fin

Листинг программы:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>
main()
{
 const int lfio=15, lpay=5, lo=10,//длины полей: фио, г.рожд., оклада в т. файле
 l=lfio+lpay+lo;    //длина записи в т. файле
 struct Man
    {   char   fio[lfio];    // фио
        int    year;         // год рожд.
        float  pay;          // оклад
```

```
    };
  Man db;
  char  s[l];        //строка для записи в файл
  FILE *fin,         // Исх. файл
       *fo ;      // Вых. файлы
  if ((fin=fopen("dbase.txt", "r"))==NULL)
       {
        puts("Ошибка открытия файла\n");
       fflush(stdin); getchar();  return (1);
       }
  puts("Файл открыт");
  fo=fopen("dbout.dat", "w+b");
  int kol=0;  //кол. зап. в текст. файле
  while (!feof(fin))    // пока не конец файла
   {
    fgets(s,l,fin);      // читается строка
    puts(s);
    strncpy(db.fio,s,lfio-1);    //из строки формир. структура db
    db.fio[lfio-1]='\0';
    db.year=atoi(&s[lfio]);
    db.pay=atof(&s[lfio+lpay]);
    fwrite(&db, sizeof (db) ,1, fo); // структура зап. в бинарный файл
    kol++;
   }
  fclose(fin);
  // чтение зап. с ном. i из бин. файла и вывод ее на экран
  int i;
  printf(«Введите номер записи (0-%d)», kol-1);
  cin>>i;
  if(i>=kol || i<0) { cout<<"Запись не существует"; fclose(fo); return
(1);}
  fseek(fo, sizeof (db)*i, SEEK_SET); //установ. указ. на зап. с
ном. i
  fread(&db, sizeof(db), 1, fo);       //чтение зап. из файла
  cout << db.fio << " "<< db.year<<" " << db.pay; //вывод зап. на
экран
 // Вывод записей с фам. ivanoff
  fseek(fo,0,SEEK_SET);  //указ. на нач. файла
  for(i=0; i<kol;i++)
```

```
    {  printf("\n");
       fread(&db,sizeof (db),1,fo);
       if(!strcmp (db.fio, "ivanoff"))
              printf("% -15s % 5d %/.0f \n",  db.fio, db.year, db.pay);
     }
  fclose(fo);
  fflush(stdin); getchar();
  return (0);
}
```

Вопросы для самопроверки

1. Алгоритм и его свойства.
2. Средства записи алгоритма – графическая схема.
3. Типы алгоритмов и их графическое изображение.
4. Общая характеристика языка программирования Си.
5. Структура программы на языке Си.
6. Основные типы данных языка Си.
7. Операции и выражения языка Си.
8. Операции присваивания и операторы присваивания.
9. Математические функции. Правила их использования в выражениях.
10. Операции сдвига и дополнения.
11. Поразрядные логические операции.
12. Логические операции и операции отношения.
13. Смешанные операнды и преобразование типов.
14. Указатели. Понятие указателя. Операции над указателями.
15. Консольный ввод-вывод. Функции ввода-вывода символов.
16. Консольный ввод-вывод. Функции ввода-вывода строк.
17. Консольный форматированный ввод данных. Оператор ввода. Форматная строка. Спецификации преобразования.
18. Консольный форматированный вывод данных. Оператор вывода. Форматная строка. Спецификации преобразования.
19. *Escape*-последовательности.
20. Программирование линейных алгоритмов.
21. Логические выражения и правила их составления.
22. Оператор *if*. Две его формы.
23. Команда выбора. Операторы *switch* и *break*.
24. Понятие цикла. Типы циклов.
25. Операторы циклов. Оператор *for*.
26. Оператор цикла *while*.
27. Задача о табулировании функции в равноотстоящих точках.
28. Оператор цикла *do-while*.
29. Вложенные циклы.
30. Понятие рекуррентного соотношения. Пример рекуррентных вычислений.
31. Вывод рекуррентного соотношения для вычисления члена ряда через предыдущий член ряда (на примере).
32. Понятие массива. Понятия статического и динамического массивов. Размерность, размер массива.

33. Синтаксис описания массивов.

34. Понятие и описание одномерного динамического массива.

35. Ввод-вывод одномерного массива.

36. Инициализация массивов.

37. Перестановка двух элементов одномерного массива.

38. Вычисление суммы элементов одномерного массива.

39. Подсчет количества элементов одномерного массива, удовлетворяющих заданным условиям.

40. Вычисление произведения элементов одномерного массива.

41. Поиск элементов одномерного массива, обладающих заданным свойством.

42. Поиск в упорядоченном одномерном массиве.

43. Поиск максимального (минимального) элемента одномерного массива и его местоположения.

44. Копирование массивов.

45. Формирование массива из элементов заданных массивов.

46. Понятие и описание двумерных массивов.

47. Ввод-вывод двумерного массива.

48. Динамические двумерные массивы.

49. Типовые алгоритмы обработки двумерных массивов.

50. Организация циклов для просмотра элементов двумерного массива со сложными условиями повторения.

51. Понятие массива. Описание двумерного массива.

52. Понятие и описание динамического двумерного массива.

53. Ввод-вывод двумерного массива.

54. Ввод-вывод двумерного динамического массива.

55. Основные алгоритмы обработки двумерных массивов.

56. Строки в языке Си. Описание строки.

57. Ввод-вывод строк.

58. Операции со строками. Реализация операции присваивания.

59. Операции со строками. Преобразование строки в число.

60. Операции со строками. Поиск подстроки в строке.

61. Операции со строками. Определение позиции первого вхождения символа из заданного набора символов.

62. Операции со строками. Сравнивание двух строк.

63. Указатели. Понятие указателя.

64. Операции над указателями.

65. Функции в языке Си. Общие сведения.

66. Функции в языке Си. Синтаксис описания.

67. Объявление функций в языке Си.

68. Вызов функции. Область действия переменных.

69. Механизм передачи параметров в функцию.

70. Структуры – параметры функции.

71. Массивы – параметры функции. Передача одномерных массивов в функцию.

72. Массивы – параметры функции. Передача двумерных массивов в функцию.

73. Указатели на функцию. Определение указателя на функцию. Функции – параметры функции.

74. Файлы в языке Си. Общие сведения.

75. Понятие потока. Стандартные библиотеки для работы с потоками.

76. Этапы работы с файлами (потоками). Открытие файла (потока).

77. Этапы работы с файлами (потоками). Закрытие файла (потока).

78. Удаление и переименование файла.

79. Ввод-вывод в поток. Основные понятия. Функции чтения символа из потока (*getc*, *fgetc*, *getchar*).

80. Функции чтения и записи потока байтов *fread* и *fwrite*.

81. Функции записи символа в поток (*putc*, *fputc*, *putchar*).

82. Функции чтения строки из потока (*fgets*, *gets*).

83. Функции форматированного ввода (чтения) из потока (*fscanf*, *scanf*, *sscanf*).

84. Функции форматированного вывода в поток (*fprintf*, *printf*, *sprintf*).

85. Обработка ошибок работы с файлами.

86. Позиционирование в файле.

87. Общие сведения о сортировках. Классификация методов сортировки.

88. Сортировки методом извлечения.

89. Сортировки методом обменов рядом стоящих элементов с фиксированным числом просмотров.

90. Сортировки методом обменов рядом стоящих элементов с минимально необходимым (переменным) числом просмотров.

91. Сортировки методом обменов рядом стоящих элементов за один просмотр (с возвратами).

92. Сортировки методом включения.

93. Сортировки методом слияния.

94. Сортировки методом распределения по массиву ключей.

95. Сортировка строк в лексикографическом порядке.

96. Размещение данных в памяти ПЭВМ.

97. Принцип программного управления.

98. Системы программирования. Состав системы. Понятие входного языка системы.

99. Понятие и виды транслятора.

100. Понятие редактора связей. Определения исходного, объектного и загрузочного модулей.

101. Средства отладки систем программирования. Типы ошибок, допускаемых при написании программ.

102. Функциональная и модульная декомпозиции.

103. Размещение программ в памяти ЭВМ.

104. Побитовые операции. Их назначение.

105. Таблица истинности побитовых операций.

106. Установка заданного бита или группы битов.

107. Сброс (очистка) заданного бита или группы битов.

108. Инвертирование заданного бита.

109. Выделение заданных битов.

110. Этапы постановки и решения задач на компьютере.

111. Назначение и классификация языков программирования.

112. Общие сведения о языках Си/*C*++.

ЛИТЕРАТУРА

1. Информатика. Базовый курс / С. В. Симонович [и др.]. – СПб. : Питер, 2001. – 640 с.

2. Касаткин, А. И. Профессиональное программирование на языке СИ: от Turbo C к Borland C++ : справ. пособие / А. И. Касаткин, А. Н. Вальвачев. – Минск : Выш. шк., 1992. – 240 с.

3. Бруно, Б. Просто и ясно. Borland C++ : пер. с англ. / Б. Бруно. – М. : БИНОМ, 1996. – 400 с.

4. Крячков, А. В. Программирование на С и С++. Практикум : учеб. пособие для вузов / А. В. Крячков, И. В. Сухина, В. К. Томшин. – М. : Горячая линия – Телеком, 2000. – 344 с.

5. Павловская, Т. А. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2002. – 464 с.

6. Демидович, Е. М. Основы алгоритмизации и программирования. Язык СИ : учеб. пособие / Е. М. Демидович. – СПб. : БХВ-Петербург, 2006. – 440 с.

7. Бронштейн, И. Н. Справочник по математике для инженеров и учащихся втузов / И. Н. Бронштейн, К. А. Семендяев. – 13-е изд., испр. – М. : Наука, 1986. – 544 с.

8. Информатика : учебник / под ред. проф. Н. В. Макаровой. – М. : Финансы и статистика, 1998.

9. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – СПб. : Невский диалект, 2001. – 352 с.

10. Страуструп, Б. Язык программирования Си++ : пер. с англ. / Б. Страуструп. – М. : Радио и связь, 1991. – 352 с.

11. Морис, С. Объектно-ориентированное программирование / С. Морис. – Ростов н/Д : Феникс, 1997. – 952 с.

12. Топп, У. Структуры данных в С++ : пер. с англ. / У. Топп, У. Форд. – М. : БИНОМ, 1994. – 816 с.

13. Хэнпок, Л. Введение в программирование на языке Си : пер. с англ. / Л. Хэнпок, М. Кригер. – М. : Радио и связь, 1986. – 192 с.

14. Кнут, Д. Э. Искусство программирования : учеб. пособие / Д. Э. Кнут. – М. : Вильямс, 2000. – Т. 1. Основные алгоритмы.

15. Кнут, Д. Э. Искусство программирования : учеб. пособие / Д. Э. Кнут. – М. : Вильямс, 2000. – Т. 2. Получисленные алгоритмы.

16. Кнут, Д. Э. Искусство программирования : учеб. пособие / Д. Э. Кнут. – М. : Вильямс, 2000. – Т. 1. Сортировка и поиск.

17. Уоррен, Г. С. Алгоритмические трюки для программистов / Г. С. Уоррен. – М. : Вильямс, 2004.

18. Кравченко, О. А. Электронный учебно-методический комплекс дисциплины «Основы алгоритмизации и программирования» для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)», 1-40 01 02-01 «Информационные системы и технологии (в проектировании и производстве)» / О. А. Кравченко.

19. Кравченко, О. А. Электронный учебно-методический комплекс дисциплины «Модели и структуры данных» для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» / О. А. Кравченко.

20. Основы алгоритмизации : практ. пособие к лаборатор. и контрол. работам по курсу «Информатика» и «Основы информатики и вычислительной техники» для студентов всех специальностей днев. и заоч. отд-ний / авт.-сост.: Н. В. Водополова, В. И. Мисюткин, С. А. Чабуркина. – Гомель : ГГТУ им. П. О. Сухого, 2005. – 32 с. (м/ук № 3097)

21. Программирование ввода-вывода данных и линейных вычислительных алгоритмов на языке *C* : практ. пособие к выполнению лаборатор. и контрол. работ по дисциплине «Вычислительная техника и программирование» для студентов техн. специальностей днев. и заоч. форм обучения / авт.-сост.: О. А. Кравченко, А. М. Мартыненко. – Гомель : ГГТУ им. П. О. Сухого, 2005. – 33 с. (м/ук № 308).

22. Мовшович, С. М. М/ук № 1909 к л/з по теме «Методы сортировок» / С. Мовшович. – Гомель : ГПИ, 1995. – 29 с.

23. Программирование разветвляющихся и циклических алгоритмов на языке *C* : пособие по выполнению лаборатор. и контрол. работ по дисциплине «Вычислительная техника и программирование» для студентов техн. специальностей днев. и заоч. форм обучения / авт.-сост.: О. А. Кравченко, Е. В. Коробейникова. – Гомель : ГГТУ им. П. О. Сухого, 2005. – 34 с. (м/ук № 3106).

24. Программирование на языке *C*. Массивы : пособие по выполнению контрол. и лаборатор. работ по дисциплине «Вычислительная техника и программирование» для студентов техн. специальностей днев. и заоч. форм обучения / авт.-сост.: О. А. Кравченко, Д. А. Литвинов. – Гомель : ГГТУ им. П. О. Сухого, 2005. – 51 с. (м/ук № 3474).

25. Программы и программирование с их использованием на языке *C* : пособие по курсу «Основы алгоритмизации и программирования» для студентов специальностей 1-36 04 02 «Промышленная электроника» и 1-40 01 02 «Информационные системы и технологии (по направлениям)» днев. и заоч. форм обучения / авт.-сост.:

О. А. Кравченко, Д. А. Литвинов. – Гомель : ГГТУ им. П. О. Сухого, 2009. – 46 с. (м/ук № 3721).

26. Структуры данных в языке СИ : пособие по курсам «Модели и структуры данных» и «Основы алгоритмизации и программирования» для студентов специальностей 1-40 01 02 «Информационные системы и технологии (по направлениям)» и 1-36 04 02 «Промышленная электроника» днев. и заоч. форм обучения / авт.-сост.: О. А. Кравченко. – Гомель : ГГТУ им. П. О. Сухого, 2010.

27. Основы алгоритмизации и программирования : курс лекций по дисциплине для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» днев. формы обучения / авт.-сост.: О. А. Кравченко, С. М. Мовшович, Е. В. Коробейникова. – Гомель : ГГТУ им. П. О. Сухого, 2010. – 112 с. (м/ук № 3963).

СОДЕРЖАНИЕ

УДК 004.42(075.8)
ББК 32.973я73
К78

Рецензенты: зав. каф. информационно-вычислительных систем БТЭУ ПК
д-р техн. наук, проф. *А. Н. Семенюта*;
доц. каф. «Автоматизированные системы обработки информации»
ГГУ им. Ф. Скорины *А. В. Воруев*

Кравченко, О. А.

К78 Основы алгоритмизации и программирования : учеб.-метод. пособие для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» / О. А. Кравченко, Л. К. Титова ; М-во образования Респ. Беларусь, Гомел. гос. техн. ун-т им. П. О. Сухого, Гомель : ГГТУ им. П. О. Сухого, 2015. – 159 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

ISBN 978-985-535-279-3.

Изложены основные темы, изучаемые в курсе «Основы алгоритмизации и программирования». Рассмотрены вопросы основ алгоритмизации и программирования, а также получения практических навыков подготовки, отладки и решения задач на современных ЭВМ.

Для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» дневной и заочной форм обучения.

УДК 004.42(075.8)
ББК 32.973я73

ISBN 978-985-535-279-3

Учебное электронное издание комбинированного распространения

Учебное издание

Кравченко Ольга Алексеевна
Титова Людмила Константиновна

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие

Электронный аналог печатного издания

Редактор *Н. В. Гладкова*
Компьютерная верстка *Е. Б. Ящук*

Подписано в печать 31.12.15.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Ризография. Усл. печ. л. 9,30. Уч.-изд. л. 9,22.
Изд. № 133.
http://www.gstu.by

Издатель и полиграфическое исполнение:
Издательский центр
Учреждения образования «Гомельский государственный технический университет имени П. О. Сухого».
Свидетельство о гос. регистрации в качестве издателя печатных изданий за №1/273 от 04.04.2014 г.
246746, г. Гомель, пр. Октября, 48