

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ

Нижегородский государственный университет им. Н.И. Лобачевского  
Национальный исследовательский университет

**В.И. Перова**  
**Т.А. Сабаева**  
**Д.Т. Чекмарев**

**РАЗРАБОТКА АЛГОРИТМОВ ДЛЯ РЕШЕНИЯ ЗАДАЧ  
НА ЭВМ**

*Учебное пособие*

Рекомендовано ученым советом механико-математического факультета  
для студентов ННГУ, обучающихся по направлениям подготовки  
38.03.05 «Бизнес-информатика», 01.03.01 «Математика», 02.03.01 «Математика  
и компьютерные науки», 01.03.02 «Прикладная математика и информатика»,  
01.03.03 «Механика и математическое моделирование»

Нижний Новгород  
2015

УДК 004.421  
ББК 32.973-018  
П26

П26 Перова В.И., Сабаева Т.А., Чекмарев Д.Т. РАЗРАБОТКА АЛГОРИТМОВ ДЛЯ РЕШЕНИЯ ЗАДАЧ НА ЭВМ: Учебное пособие. – Нижний Новгород: Нижегородский госуниверситет, 2015. – 136 с.

Рецензенты: к.ф.м.н., снс Белов А.А.

В учебном пособии изложены основные понятия алгоритмов решения задач на электронных вычислительных машинах (ЭВМ), порядок подготовки и решения задач. Большое внимание уделено таким темам, как методы разработки алгоритмов, алгоритмы работы с различными типами данных, алгоритмы комбинаторики. Теоретический материал иллюстрируется реально работающими примерами.

Учебное пособие предназначено для студентов механико-математического факультета ННГУ, обучающихся по направлениям подготовки 01.03.01 «Математика», 02.03.01 «Математика и компьютерные науки», 01.03.02 «Прикладная математика и информатика», 01.03.03 «Механика и математическое моделирование», и для студентов Института экономики и предпринимательства ННГУ, обучающихся по направлению подготовки 38.03.05 «Бизнес-информатика». Оно адресовано и студентам других факультетов, изучающим языки программирования. Данное учебное пособие будет полезно всем пользователям, стремящимся к эффективному использованию современных информационных технологий.

Ответственный за выпуск:  
председатель методической комиссии механико-математического  
факультета ННГУ, к.ф.-м.н., доцент **Н.А. Денисова**

УДК 004.421  
ББК  
32.973-018

© В.И. Перова, Т.А. Сабаева, Д.Т. Чекмарев, 2015  
© Нижегородский государственный  
университет им. Н.И. Лобачевского, 2015

## ОГЛАВЛЕНИЕ

<b>ПРЕДИСЛОВИЕ</b> .....	5
<b>ГЛАВА 1. Понятие алгоритма. Формализация понятия «Алгоритм» ..</b>	8
<b>ГЛАВА 2. Подходы и требования к разработке алгоритмов и программ для ЭВМ</b> .....	12
2.1. Подходы к разработке алгоритмов и программ .....	12
2.2. Основные требования к разработке алгоритмов .....	16
<b>ГЛАВА 3. Способы описания алгоритмов</b> .....	17
3.1. Графическое описание алгоритмов .....	18
3.2. Описание алгоритмов с помощью алгоритмического языка ..	23
<b>ГЛАВА 4. Этапы построения алгоритмов</b> .....	27
4.1. Основные свойства алгоритма .....	28
4.2. Типы алгоритмов .....	29
4.3. Построение алгоритма .....	32
4.3.1. Постановка задачи .....	33
4.3.2. Построение математической модели .....	33
4.3.3. Анализ существующих алгоритмов .....	36
4.3.4. Разработка собственного алгоритма и оценка его эффективности .....	36
4.3.5. Доказательство правильности алгоритма .....	39
4.3.6. Описания алгоритмов .....	40
4.3.7. Реализация алгоритма .....	40
4.3.8. Отладка и тестирование программы на ЭВМ .....	41
4.3.9. Составление документации .....	43
4.4. Презентация и сопровождение программного продукта .....	44
<b>ГЛАВА 5. Методы создания алгоритмов</b> .....	48
5.1. Метод частных целей .....	48
5.2. Метод подъёма .....	49
5.3. Метод отработки назад .....	51
5.4. Метод программирования с отходом назад .....	53
5.5. Методы разработки эвристических алгоритмов .....	54
5.6. Рекурсия .....	56
5.7. Моделирование .....	57
<b>ГЛАВА 6. Основные алгоритмы и их реализация</b> .....	64
6.1. Алгоритмы анализа текста .....	64
6.2. Алгоритмы работы с датами .....	71
6.3. Алгоритмы комбинаторики .....	81
6.3.1. Алгоритм полной выборки из n элементов .....	81
6.3.2. Алгоритм получения перестановок .....	87
<b>ГЛАВА 7. Алгоритмы и классы</b> .....	91
7.1. Конструкторы и деструкторы .....	92
7.2. Спецификаторы доступа .....	93

7.3. Наследование классов .....	94
7.3.1. Простое наследование .....	94
7.3.2. Множественное наследование .....	97
7.4. Перегрузка операторов .....	98
7.5. Практические задачи применения алгоритмов с классами .....	101
7.5.1. Алгоритмы и классы для операций с датами .....	101
7.5.2. Алгоритмы и классы для создания баз данных .....	109
7.5.3. Алгоритмы с использованием наследования классов .....	113
7.5.4. Алгоритмы и классы с применением связанных списков ...	117
<b>Список литературы .....</b>	<b>129</b>

## ПРЕДИСЛОВИЕ

Интерес к программированию, резко возросший в последнее время, связан с развитием и использованием информационных и коммуникационных технологий. Все программы, например служебные программы для архивации данных, компьютерные игры, обозреватели для работы в Интернете и др., написаны на одном или нескольких языках программирования. Для разработки интерфейсов стал всё более широко употребляться язык Java [77], но для разработки наиболее сложных и высокоэффективных программ используется язык программирования C++ [4, 17, 29, 34, 46 – 48, 53, 54, 56 – 60, 63 – 65, 67, 77, 78].

Решение задач с применением электронных вычислительных машин (ЭВМ) представляет собой сложный процесс, который состоит из следующих этапов [61].

### 1. Этапы, не связанные с программированием:

- *Постановка задачи.* На данном этапе формулируется исследуемая проблема в терминах предметной области.
- *Построение математической модели.* На этом этапе определяются в окончательном виде объекты и их связи, которые описывают исследуемую задачу.
- *Выбор численной модели.* На основе исходной математической задачи разрабатывается численная модель для аппроксимации поставленной задачи и проводится анализ сходимости построенной численной модели к исходной математической модели.

### 2. Этапы, связанные с программированием:

- *Разработка алгоритма решения задачи.* Этот этап называется ещё *алгоритмизацией задачи*. Он является первым этапом программирования. В процессе выполнения данного этапа разрабатывается алгоритм решения, т.е. устанавливается необходимая последовательность действий для реализации метода. Обычно предварительно анализируются несколько возможных вариантов алгоритмов. Затем выбирается тот вариант, который обеспечивает наиболее эффективное использование ЭВМ. Особенностью этого этапа является то, что алгоритм решения задачи разрабатывается в общем виде.
- *Разработка программы.* Данный этап является вторым этапом программирования, цель которого – непосредственное составление программы. В процессе выполнения этого этапа исходный алгоритм, созданный на предыдущем этапе программирования, детализируется и преобразуется с учетом специфики решения задачи на ЭВМ на применяемом языке программирования. Следует отметить, что составление программы представляет собой трудоёмкий и сложный процесс, который является источником ошибок.
- *Отладка программы.* Задача данного этапа – путём апробации разработанной программы на ЭВМ выявить ошибки, допущенные на всех преды-

дущих этапах. Отладка является завершающим этапом в разработке программы.

- *Решение задачи на ЭВМ.* Содержание этого этапа ясно из его названия. Затем производится обработка результатов вычислений.

Следует отметить, что и уровень начальный подготовки студентов по программированию, поступивших на первый курс Нижегородского государственного университета им. Н.И. Лобачевского (ННГУ), да и других вузов, весьма различен. Увеличение часов самостоятельной работы в учебных программах дисциплин требует от преподавателей действий по активизации самостоятельной познавательной деятельности студентов. Опыт преподавания авторами дисциплин: «Программирование», «Объектно-ориентированный анализ и программирование», «Основы информатики», «Языки и методы программирования» «Информатика и программирование», «Методы программирования» в ННГУ показал, что академическая и самостоятельная деятельность студентов более продуктивна с использованием специальных авторских учебных пособий [54, 56]:

- ✓ Перова В.И. Программирование на C++ в среде Visual Studio.NET. – Нижний Новгород: Нижегородский госуниверситет, 2010. – 261 с.
- ✓ Перова В.И., Сабаева Т.А. Программирование на языке C++. – Нижний Новгород: Изд-во Нижегородского госуниверситета, 2013. – 132 с.

В учебном пособии «Алгоритмы решения задач на ЭВМ» рассматриваются основные алгоритмы решения задач на ЭВМ, методы разработки алгоритмов, этапы их построения и реализации, поскольку многие студенты, и особенно студенты младших курсов, не имеющие навыков самостоятельной работы, испытывают затруднения в построении алгоритмов и оценки их эффективности.

Наряду с указанными выше авторскими учебными пособиями данное учебное пособие даст возможность студентам приобрести навыки, необходимые для создания программ во время академических занятий и самостоятельной работы.

Учебное пособие предназначено для студентов, обучающимся в ННГУ по направлениям подготовки: 38.03.05 «Бизнес-информатика», 01.03.01 «Математика», 02.03.01 «Математика и компьютерные науки», 01.03.02 «Прикладная математика и информатика», 01.03.03 «Механика и математическое моделирование», а также для студентов, обучающимся по другим направлениям подготовки, где изучаются языки программирования.

Авторы выражают глубокую благодарность за поддержку и ценные замечания при подготовке рукописи учебного пособия заместителю по очной форме обучения проректора по учебной и воспитательной работе, руководителю Цен-

тра прикладной статистики Института экономики и предпринимательства ННГУ, доценту Н.Р. Стронгиной .

Авторы выражают искреннюю признательность за поддержку заведующему кафедрой математического моделирования экономических процессов Института экономики и предпринимательства ННГУ, профессору Ю.А. Кузнецову, директору НИИ механики, заведующему кафедрой численного моделирования физико-механических процессов механико-математического факультета ННГУ, профессору Л.А. Игумнову и директору Института радиоэлектроники и информационных технологий Нижегородского государственного технического университета им. Р.Е. Алексеева, действительному члену Международной академии информатизации, профессору В.Г. Баранову.

Авторы выражают искреннюю благодарность заведующему кафедрой «Электроника и сети ЭВМ» Института радиоэлектроники и информационных технологий Нижегородского государственного технического университета им. Р.Е. Алексеева, профессору В.Р. Милову, заведующему кафедрой математики и информатики Нижегородского государственного лингвистического университета им. Н.А. Добролюбова, профессору В.В. Савченко за полезные замечания при рецензировании рукописи учебного пособия.

Авторы выражают искреннюю признательность профессору кафедры численного моделирования физико-механических процессов механико-математического факультета ННГУ В.Л. Котову за обсуждение материала, представленного в учебном пособии.

## ГЛАВА 1. ПОНЯТИЕ АЛГОРИТМА. ФОРМАЛИЗАЦИЯ ПОНЯТИЯ «АЛГОРИТМ»

*Алгоритм* является одним из фундаментальных понятий математики и информатики. Наряду с моделированием, алгоритмизация – это общий метод информатики. Алгоритмы являются объектом систематического исследования научной дисциплины «Теория алгоритмов» – раздела современной математики, где изучаются общие свойства алгоритмов. Эта дисциплина является пограничной между математикой и информатикой и примыкает к математической логике [2, 45].

Термин «алгоритм» появился в математике в XII веке и происходит от латинской формы написания арабского имени среднеазиатского математика IX в. Мухаммеда ибн Мусса аль-Хорезми (Мухаммеда, сына Муссы из Хорезма, ок. 783 – ок. 850 гг.) – *algorithmi* [2, 5, 14, 22]. В 1983 г. отмечалось 1200-летие со дня рождения аль-Хорезми в городе Ургенче – областном центре современной Хорезмской области Узбекистана.

Научные интересы аль-Хорезми связаны с математикой, теоретической и практической астрономией, географией и историей, но наибольшую славу ему принесли математические труды. Он сформулировал правила выполнения арифметических действий, написал два знаменитых трактата: по арифметике и алгебре. Подлинный арабский текст арифметического трактата утерян, но имеется его латинский перевод, который выполнен в XII веке, не имеет названия и начинается со слов: «*Dixit algorizmi*» – «Сказал Алгоризми». Алгебраический трактат аль-Хорезми называется «*Китаб аль-джебр аль-мукабала*» («Книга о восстановлении и противопоставлении») и сохранился в арабской копии до настоящего времени. В нем алгебра впервые рассматривалась как самостоятельный раздел математики, который изучает общие методы решения линейных и квадратных уравнений.

В [42] приведена следующая формулировка понятия алгоритма:

**«АЛГОРИТМ, алгоритм»** – точное предписание, которое задает вычислительный процесс (называемый в этом случае *алгоритмическим*), начинающийся с произвольного *исходного данного* (из некоторой совокупности возможных для данного алгоритма исходных данных) и направленный на получение полностью определяемого этими исходными данными *результата*».

В данной формулировке понятия алгоритма термин «вычислительный процесс» не нужно понимать в узком смысле только цифровых вычислений. Алгоритмы могут оперировать произвольными символами и их комбинациями, например, уже в школьном курсе алгебры говорят о буквенных вычислениях.

Исходными данными и результатами алгоритма могут служить разнообразные *конструктивные объекты*. **Конструктивный объект** – это объект, который может быть полностью описан при помощи конечной последовательности символов. Множество всех конструктивных объектов одинакового вида об-



разует **конструктивное пространство**. Конструктивными объектами являются, например, натуральные и рациональные числа. Алгоритмы также являются конструктивными объектами, если под алгоритмом понимается *конечный* набор инструкций, преобразующих входную информацию в выходную. Это дает возможность широкого использования понятия **алгоритм**.

С понятием алгоритма тесно связано понятие «**исполнитель алгоритма**». В качестве исполнителя может выступать человек или группа людей, робот, компьютер, язык программирования и др. Исполнитель действует *формально*, т.е. только строго исполняет команды алгоритма, отвлекаясь от содержания поставленной задачи [45]. При этом исполнитель получает необходимый результат. Это является одной из важных особенностей алгоритмов.

Следуя [42], данный алгоритм можно охарактеризовать семью параметрами:

- 1) Совокупность возможных исходных данных.
- 2) Совокупность возможных результатов.
- 3) Совокупность возможных промежуточных результатов.
- 4) Правило начала.
- 5) Правило непосредственной переработки.
- 6) Правило окончания.
- 7) Правило извлечения результата.

Понятие алгоритма в его общем виде относится к числу основных первоначальных понятий математики, которые не допускают определения в терминах более простых понятий. Попытке дать определение алгоритма, т.е. строгого его описания или, как ещё говорят, его *формализации*, посвящено огромное количество работ, однако более или менее устоявшаяся терминология связывается с определенной предметной областью, где это понятие используется. При этом каждое уточнение понятия алгоритма состоит в том, что для каждого из 7 указанных выше параметров алгоритма точно описывается некоторый класс, в пределах которого этот параметр может меняться.

Среди подходов к формализации понятия «алгоритм», возникших исторически независимо друг от друга, отметим следующие [1 – 3, 28, 39, 40 – 45]:

- **Теория конечных и бесконечных автоматов.** Теория автоматов является разделом теории *управляющих систем*, изучающим математические модели преобразователей дискретной информации, называемые *автоматами*. Теория автоматов возникла в середине XX века в связи с изучением конечных автоматов как математических моделей нервных систем и вычислительных машин. Наиболее тесно теория автоматов связана с теорией алгоритмов, поскольку конечный автомат можно рассматривать как частный случай формально заданного алгоритма. Примерами формализации понятия алгоритма в теории автоматов являются абстрактные (т.е. существующие в воображении, а не реально) вычислительные машины Э.

Поста и А. Тьюринга. Теория автоматов получила развитие в трудах С. Клини, А.А. Маркова, В.Н. Колмогорова и др. [5, 24, 42, 45, 69].

- **Теория вычислимых функций.** Вычислимая функция является одним из основных понятий теории алгоритмов. Функция  $f$  называется *вычислимой*, если существует алгоритм, перерабатывающий в  $f(x)$  всякий  $x$ , для которого  $f$  определена, и не применимый ни к какому  $x$ , для которого  $f$  не определена. Значениями аргумента и значениями вычислимой функции могут быть лишь конструктивные объекты. Одним из наиболее распространенных вариантов математического уточнения понятия вычислимой арифметической функции, т. е. вычислимой функции, аргументы и значения которой – натуральные числа, является *рекурсивная функция* [38]. Рекурсивные функции являются частичными функциями. *Частичные функции* – это функции, которые не обязательно всюду определены. Поэтому в качестве синонима используется термин «*частично рекурсивные функции*». Рекурсивные функции, которые являются определенными при любых значениях аргументов, называются *общерекурсивными функциями* [41, 42].
- **$\lambda$ -исчисление А. Чёрча.** Идеи  $\lambda$ -исчислений А. Чёрча реализованы в языке программирования **Лисп**, где на  $\lambda$ -исчислении А. Чёрча основано определение функций и их вычисление [75]. В 1936 г. А. Чёрч впервые выдвинул принцип (называемый *тезисом Чёрча*), согласно которому класс функций, вычислимых с помощью алгоритмов в широком интуитивном смысле, совпадает с классом частично рекурсивных функций. Тезис Чёрча является естественнонаучным фактом, который подтверждается опытом, накопленным в математике за всю её историю. Этому тезису удовлетворяют все известные в математике алгоритмы. Если строго доказано, что данная *алгоритмическая проблема* – проблема, в которой нужно найти единый метод (*алгоритм*) для решения бесконечной серии однотипных единичных задач (так называемой *массовой проблемы*) – не может быть решена в рамках того или иного уточнения понятия алгоритма, то тезис Чёрча является основанием для вывода о неразрешимости этой проблемы [42].
- **Нормальные алгоритмы А.А. Маркова.** Марков А.А. предложил способ уточнения понятия алгоритма, который основан на понятии *нормального алгоритма*. Понятие «нормальный алгоритм» Марков А.А. ввёл в 1947 г. в ходе своих исследований по проблеме тождества ассоциативных систем. *Нормальный алгоритм* определяется следующим образом. Пусть имеется алфавит  $A$  и система подстановок  $P$ . Нормальный алгоритм в алфавите  $A$  – это предписание строить, исходя из произвольного слова  $W$  в  $A$ , последовательность слов  $W_i$  согласно следующему правилу. Слово  $W$  берётся в качестве начального члена  $W_0$  этой последовательности, и да-

лее продолжается процесс её построения. Для слова  $W$  подбираются подстановки из  $P$  в том порядке, в котором они следуют в  $P$ . Если подходящая подстановка отсутствует, то процесс останавливается. В противном случае берётся первая из подходящих подстановок и производится замена её правой частью первого вхождения её левой части в  $W$ . Затем все действия повторяются для слова  $W_1$  и т.д. Процесс останавливается, если применяется последняя подстановка из системы подстановок  $P$ .

Следует отметить, что нормальные алгоритмы могут отличаться друг от друга алфавитами и системами подстановок. В случаях, когда объекты рассмотрения допускают представление в виде слов в некоторых алфавитах, нормальные алгоритмы являются эффективным аппаратом в исследованиях, где требуется точное понятие алгоритма. Нормальный алгоритм Маркова А.А. можно рассматривать в качестве универсальной формы задания любого алгоритма [5, 39, 40, 45].

- **Уточнение А.Н. Колмогорова понятия алгоритма.** Подход Колмогорова А.Н. к уточнению понятия алгоритма является наиболее общим. Он предложил трактовать конструктивные объекты как топологические комплексы определенного вида. [5, 40, 70].

Рассмотренные подходы к формализации понятия алгоритма впоследствии оказались эквивалентными [5, 45].

## ГЛАВА 2. ПОДХОДЫ И ТРЕБОВАНИЯ К РАЗРАБОТКЕ АЛГОРИТМОВ И ПРОГРАММ ДЛЯ ЭВМ

Создание алгоритмов и программ для ЭВМ – это творческая задача, которая в общем случае требует выработки или привлечения нового знания. Построение алгоритма является важным конструктивным компонентом программирования. Он не связан с особенностями синтаксиса языков программирования, а также со спецификой функционирования конкретных ЭВМ.

В ходе эволюции компьютеров изменялись подходы и требования к созданию алгоритмов.

### 2.1. Подходы к разработке алгоритмов и программ

Отметим следующие подходы к построению алгоритмов:

➤ **Операциональный подход.** Данный подход использовался в эпоху ЭВМ первого и второго поколения, когда их возможности с точки зрения сегодняшних достижений были скромны [22, 45, 61]. Основными требованиями к алгоритмам были:

- а) *минимальное время исполнения*, т.е. минимальное число операций;
- б) использование возможно *наименьшего числа ячеек оперативной памяти компьютера* при выполнении программы.

Операциональный подход требует от программиста детального описания решения задачи, т.е. формулировки *алгоритма* и его специальной записи, но при этом обычно не указываются ожидаемые свойства результата. *Оператор* и *данные* – это основные понятия языков операционального программирования.

Типичными языками программирования в операциональном подходе являются Ассемблер, Бейсик, Фортран.

➤ **Процедурный (процедурно-ориентированный) подход.** В данном подходе, как и в операциональном, необходимо детальное описание решения задачи, а именно формулировка *алгоритма* и его специальная запись. Ожидаемые свойства результата также обычно не указываются. *Оператор* и *данные* являются основными понятиями языков процедурного программирования, однако операторы объединяются в группы (*процедуры*).

Типичные языки программирования в процедурно-ориентированном подходе [71, 72] – Бейсик, Фортран, Паскаль, Си.

➤ **Структурный подход.** Этот подход к разработке алгоритмов возник в конце 1970-х гг. и связан с появлением ЭВМ третьего поколения. Структурный

подход к разработке алгоритмов в целом не выходит за рамки процедурного подхода. Согласно *теореме Бема – Якопини*, основой технологических принципов структурного программирования является утверждение о том, что логическая структура программы может состоять из комбинации трех базовых структур [8, 22, 45, 61, 71, 74]:

а) *следования*, означающего, что действия могут быть выполнены друг за другом;

б) *ветвления* – структуры, которая обеспечивает выбор между двумя альтернативами. После выполнения проверки выбирается один из путей;

в) *цикла*, который повторно выполняет некоторый набор команд программы.

Одним из компонентов структурного подхода к разработке алгоритмов является *модульность*.

*Модуль представляет собой последовательность операций, которые логически связаны между собой и оформлены как отдельная часть программы.*

Преимущества использования модулей, согласно [45], заключаются в следующем:

- ✓ возможность разработки программы несколькими пользователями;
- ✓ простота проектирования и модификаций программы;
- ✓ упрощение отладки программы, т.е. поиска и устранения ошибок;
- ✓ возможность применения библиотек модулей.

Важным достижением структурного подхода к построению алгоритмов является *нисходящее проектирование программ*. Оно основано на идее уровней абстракции. Эти уровни в разрабатываемой программе становятся уровнями модулей. При нисходящем проектировании подлежащая решению задача разбивается на несколько подзадач, которые по своему содержанию подчиняются главной задаче. В свою очередь, эти подзадачи разбиваются на более мелкие подчиненные подзадачи до тех пор, пока не будет достигнут уровень относительно небольших подзадач, требующих для решения небольших модулей. Данное разбиение называется *декомпозицией* или *детализацией*. Схема иерархии при проектировании программ позволяет преодолеть проблему сложности разработки программы [45].

Типичными языками программирования в структурном подходе являются Паскаль, Модула.

➤ ***Параллельное программирование.*** Данный подход является развитием процедурно-ориентированного подхода в программировании.

*«Параллельное программирование – раздел программирования, связанный с изучением и разработкой методов и средств для:*

а) адекватного выражения в программах естественного параллелизма решаемых на ЭВМ задач;

б) распараллеливания обработки информации в многопроцессорных и мультипрограммных ЭВМ с целью ускорения вычислений и эффективного использования ресурсов машины.» [5].

Основу обычного (последовательного) программирования составляет понятие *алгоритма*, который исполняется строго последовательно во времени по шагам. В параллельном программировании программа генерирует совокупность процессов обработки информации. Эти процессы могут быть связаны между собой статическими либо динамическими отношениями пространственно-временного или причинно-следственного характера [9 – 11, 36, 49 – 52]. Параллельные вычисления могут выступать в разных конкретных формах. Это зависит от этапа программирования, от сложности параллельно выполняемых фрагментов вычислений и от характера связей между ними [5]. Современные технологии параллельного программирования позволяют:

- ✓ создавать и анализировать эффективные *параллельные алгоритмы*;
- ✓ использовать программные инструменты и библиотеки для разработки, отладки, анализа и оптимизации параллельных программ.

К таким технологиям относятся технологии [49 – 52]:

- ✓ ориентированные на кластеры / суперкомпьютеры (технология MPI);
- ✓ состоящие из вычислительных узлов на базе традиционных многоядерных центральных процессоров (технологии OpenMP, Intel Cilk Plus, Intel TBB, Intel ArBB, OpenCL);
- ✓ состоящие из гетерогенных узлов с использованием графических процессоров (технологии NVIDIA CUDA, OpenCL).

Отметим, что исследования в области параллельного программирования способствуют дальнейшему развитию архитектуры высокопроизводительных машин и комплексов.

- **Объектный (Объектно-ориентированный) подход.** Объектно-ориентированное программирование (ООП) – это программирование с использованием объектов. Программа является моделью некоторой другой реально существующей или искусственной системы. При разработке программной модели некоторой предметной области – части реального мира – большая сложность возникает в семантическом разрыве между реальностью

и программой. Программа и предметная область описываются в разных терминах и понятиях. В объектной модели этот разрыв уменьшается, так как реальный мир описывается как набор взаимодействующих объектов [35, 54, 57, 58, 64, 65, 73].

Основными свойствами ООП являются [54]:

- ✓ *Инкасуляция* – объединение данных с функциями их обработки в сочетании со скрытием информации, ненужной для использования этих данных.
- ✓ *Наследование* – возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые.
- ✓ *Полиморфизм* – возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и выбирать требуемое действие во время выполнения программы. Чаще всего понятие полиморфизма связывается с механизмом виртуальных функций.

Типичными языками программирования в объектном подходе являются С++, Делфи [16, 64, 65].

- *Декларативный подход*. Данный подход в разработке компьютерных программ появился в начале 70-х гг. Он направлен на решение задач искусственного интеллекта. При применении декларативного подхода описывается не алгоритм получения результата, а свойства, взаимосвязи исходных данных и свойства, которыми должен обладать результат. Алгоритм получения результата порождается автоматически системой, поддерживающей декларативно-ориентированный язык программирования. Декларативный подход в разработке компьютерных программ развивается в двух вариантах: *логическом* и *функциональном*. В логическом варианте описание задачи происходит с помощью совокупности фактов и правил в некотором формальном логическом языке. В функциональном варианте задача описывается в виде функциональных соотношений между фактами [45, 61].

Типичные языки программирования в декларативном подходе [75]: в логическом варианте – Пролог, в функциональном варианте – Лисп.

## 2.2. Основные требования к разработке алгоритмов

Рассмотренные выше подходы к разработке алгоритмов предполагают выполнение основных требований, предъявляемых к алгоритмам:

1. *Дискретность*, т.е. *прерывистость*, – работа алгоритма осуществляется тактами (шагами).

2. *Детерминированность*, т.е. *определенность*, – предшествующие шаги алгоритма вполне определяют его каждый последующий шаг.

3. *Результативность* – алгоритм обязан выдать результат после конечного числа шагов.

4. *Универсальность* – алгоритм должен быть применим к любой задаче данного класса задач.



### ГЛАВА 3. СПОСОБЫ ОПИСАНИЯ АЛГОРИТМОВ

Для описания алгоритмов используются средства, которые в большой мере определяются тем, кто будет исполнителем. Если исполнителем является человек, то описание алгоритма может быть формализовано не полностью. В данном случае на первое место выдвигаются наглядность и понятность. Поэтому можно применить *словесное* описание алгоритма.

Алгоритмы, предназначенные для исполнителей-автоматов, необходимо формализовать. В таких случаях применяются *формальные специальные языки*. Преимущество формального способа описания алгоритмов состоит в возможности изучения их как математических объектов.

Алгоритмы задач, решаемых на ЭВМ, обычно содержат много этапов, которые по-разному связаны между собой. По своему назначению этапы являются различными. Математические этапы, или математическая часть программы, заключаются в проведении вычислений по определенным формулам. Логические этапы (логическая часть программы) состоят в проверке выполнения заданных условий, которые устанавливают моменты перехода от одного арифметического этапа к другому.

Исходный алгоритм, разрабатываемый на *первом этапе программирования*, отражает сущность методики решения задачи вообще. Он обычно содержит только арифметическую и логическую части.

На *втором этапе программирования* разрабатывается логическая схема программы, представляющая собой расширенный исходный алгоритм. Здесь предусматривается ряд других этапов, которые обусловлены спецификой решения задачи на ЭВМ: ввод программы и исходных данных в машину, вывод результатов и др.

При этом важно записать алгоритм в наглядной и компактной форме, удобной для практического использования. Алгоритм, составленный для некоторого исполнителя, может быть представлен одним из способов [45, 61]:

- ✓ с помощью графического описания;
- ✓ с помощью словесного описания;
- ✓ в виде таблицы;
- ✓ последовательностью формул;
- ✓ с помощью алгоритмического языка (языка программирования).

Рассмотрим графическое описание алгоритмов и описание с помощью алгоритмического языка. Эти способы имеют ряд преимуществ, благодаря явному отображению управления в алгоритме и высокой «читаемости» алгоритма.




### 3.1. Графическое описание алгоритмов

При графическом описании алгоритмов применяются последовательности графических символов, которые связаны между собой и выполняют определенные функции. Конфигурацию, перечень и размеры графических изображений, а также правила построения схем алгоритмов устанавливает ГОСТ 19.701-90 «Схемы алгоритмов, программ, данных и систем».

В табл. 3.1 приведены основные символы, которые используются при описании алгоритмов [14, 25, 45].

Таблица 3.1

Графические символы для описания алгоритмов

Символ	Наименование символа
	<b>Начало - конец алгоритма.</b> Прерывание процесса обработки информации. Символ имеет один вход и один выход.
	<b>Данные. Ввод - вывод данных.</b> Носитель не определен. Символ имеет один вход и один выход.
	<b>Процесс.</b> Обработка данных любого вида. Выполнение определенной операции или группы операций, приводящее к изменению значения, формы или размещения информации. Символ имеет один вход и один выход.
	<b>Решение.</b> Функция переключательного типа. Имеет один вход и ряд альтернативных выходов, только один из которых может быть активизирован после вычисления условий, определенных внутри этого символа. Символ имеет один вход и несколько альтернативных выходов.
	<b>Подготовка.</b> Модификация команды или группы команд с целью воздействия на некоторую последующую функцию (инициализация программы, модификация индекса и др.). Символ имеет один вход и один выход.
	<b>Документ.</b> Отображает итоговые данные в удобочитаемой форме. Вывод информации результата на принтер. Символ имеет один вход и один выход.

Графическое описание алгоритма называется **блок-схемой**. **Блок-схема** представляет собой ориентированный граф, который указывает порядок исполнения команд алгоритма. Вершины графа, представленные на рис. 3.1, могут быть одного из трех типов [45]:

- **Функциональная вершина (F).** Данная вершина имеет один вход и один выход.

- **Предикатная вершина ( $P$ ).** Эта вершина имеет один вход и два выхода. Функция  $P$  может передавать управление по одной из ветвей в зависимости от значения  $P$  ( $t$ , т.е. *true*, ИСТИНА, либо  $f$ , т.е. *false*, ЛОЖЬ). Часто вместо  $t$  пишут «да» или «+», а вместо  $f$  – «нет» или «-».
- **Объединяющая вершина (вершина «слияния») ( $U$ ).** Она обеспечивает передачу управления от одного из двух входов к выходу.

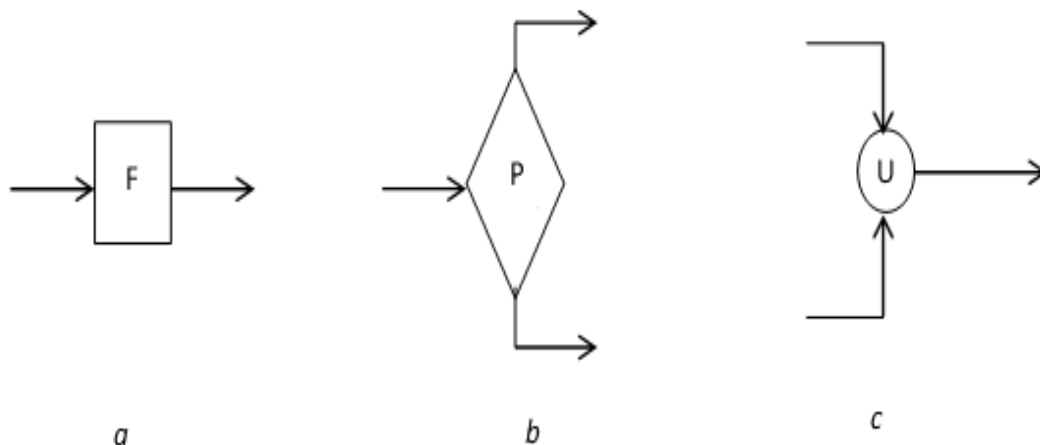


Рис. 3.1. Вершины, используемые в блок-схемах:  $a$  – функциональная вершина,  $b$  – предикатная вершина,  $c$  – объединяющая вершина

Функциональная вершина применяется для представления функции  $F: X \rightarrow Y$ .

Предикатная вершина используется для отображения функции (предиката)  $P: X \rightarrow \{T, F\}$ , то есть логического выражения, которое передает управление одной из двух ветвей.

Объединяющая вершина представляет передачу управления от одной из входящих ветвей к одной выходящей.

Элементарные блок-схемы, изображенные на рис 3.1, используются при построении основных блок-схем, имеющих особое значение для практики алгоритмизации:

- ✓ *следование* или *композиция* (рис. 3.2  $a$ );
- ✓ *ветвление* или *альтернатива* (рис. 3.2  $b$ );
- ✓ *цикл* или *итерация* (рис. 3.2  $c, d$ ).

Использование этих базовых блок-схем, согласно *теореме Бема – Якопини* [14, 45], может составлять **структурную блок-схему** (см. Главу 2).

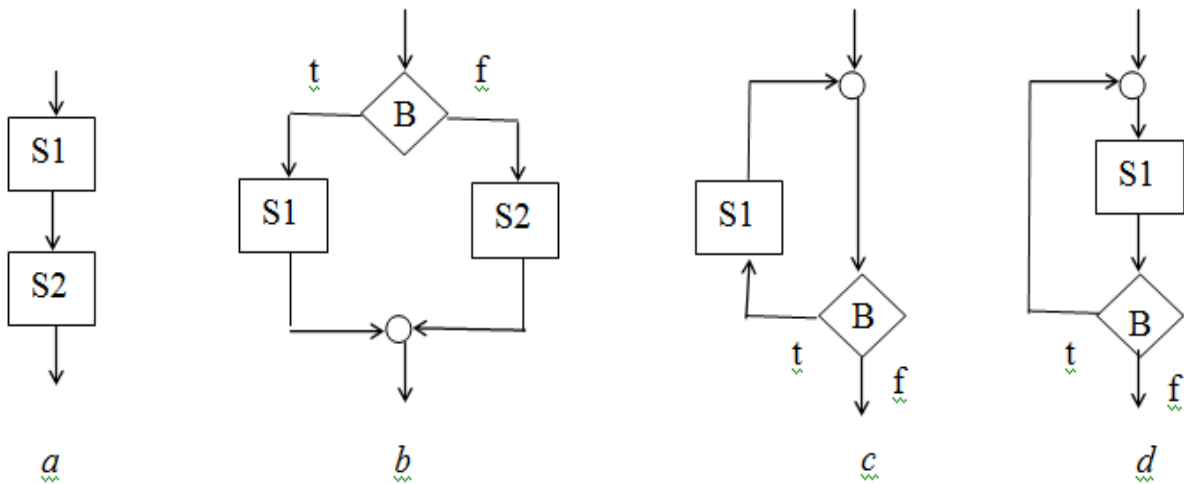


Рис. 3.2. Базовые блок-схемы, используемые в структурных блок-схемах:  
*a* – следование, *b* – ветвление,  
*c* – цикл с предусловием, *d* – цикл с постусловием

Каждая из блок-схем, представленных на рис. 3.2, эквивалентна определённому оператору любого структурированного языка программирования. Например, на языке C++ блок-схема, изображенная

- ✓ на рис. 3.2 *a*, – это последовательность операторов присваивания,
- ✓ на рис. 3.2 *b* – оператор `if (B) S1; else S2; ,`
- ✓ на рис. 3.2 *c* – оператор `while (B) S1; ,`
- ✓ на рис. 3.2 *d* – оператор `do S1; while (B); .`

где *B* интерпретируется как булевское выражение, а *S1* и *S2* интерпретируются как программные операторы.

Часто используется укороченный оператор

`if (B) S1; ,`

блок-схема которого отличается от блок-схемы на рис. 3.2 *b* отсутствием блока *S2*.

Важной особенностью этих блок-схем является то, что они имеют один вход и один выход. Из этого следует, что любая структурная блок-схема, составленная из них, также будет иметь один вход и один выход, например, блок-схема, представленная на рис. 3.3.

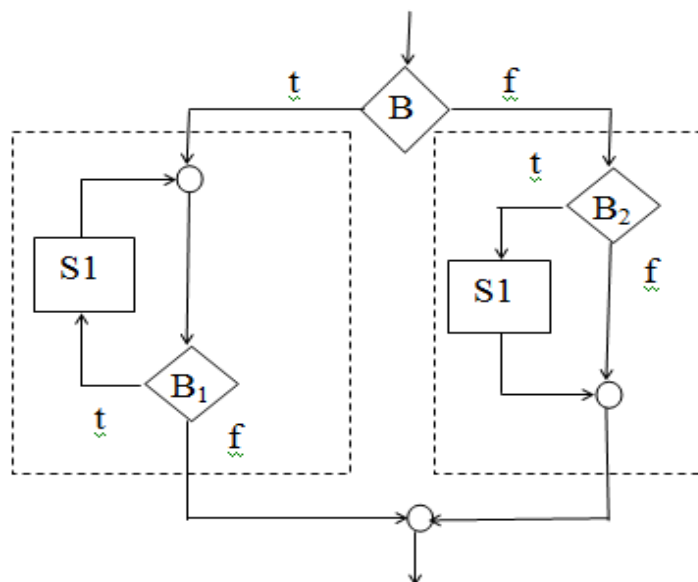


Рис. 3.3. Структурная блок-схема

Ее программный аналог на языке C++ имеет вид:

```

if (B)
    while (B1)
        S1;
else if (B2)
    S2;
  
```

Структурное программирование допускает больше элементарных структур, чем те, которые были предложены *Бемом и Якопини*. Это связано с тем, что практически во всех языках программирования определен *оператор-переключатель*, который имеет множество разветвлений. Алгоритмическая структура оператора-переключателя приведена на рис. 3.4.

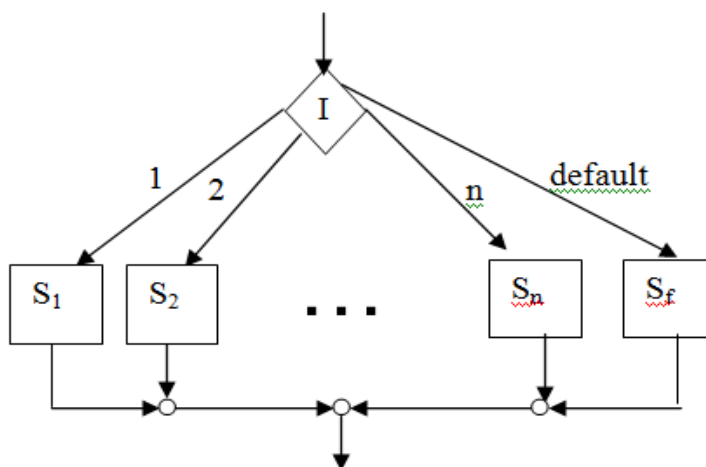


Рис. 3.4. Алгоритмическая структура оператора-переключателя

Его программный аналог на языке C++ имеет вид:

```
switch (I)
{
    case 1: S1;
    case 2: S2;
    . . .
    case n: Sn;
    default:
        Sf;
}
```

В операторе **switch** допускается отсутствие блока *Sf*. В этом случае при несовпадении значения выражения *I* с метками, указанными после служебного слова **case**, оператор **switch** игнорируется.

При разработке блок-схемы для описания алгоритма необходимо придерживаться следующего правила. Сначала реализуется блок-схема, содержащая крупные блоки программы. Затем для каждого из крупных блоков проводится последующая детализация в методике структурного программирования. Процесс пошагового разбиения алгоритма на более мелкие части называется **программированием сверху-вниз** [14].

**Структурное программирование сверху-вниз** – это процесс программирования сверху вниз, который ограничен применением структурных блок-схем.

На рис.3.5 приведена структурная блок-схема, которая имеет глубину вложения три.

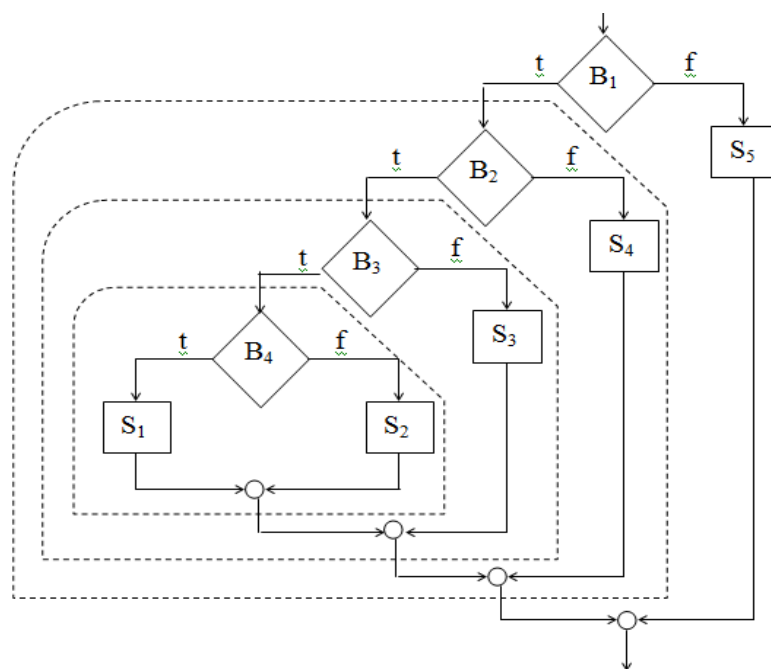


Рис. 3.5. Структурная блок-схема глубины вложения три [14]

Из рис. 3.5. видно, что в структурированной программе структура управления – это *дерево*. Поэтому такие программы называются *программами со структурой дерева*.

Следует отметить, что этот метод разработки алгоритмов не дает гарантии отсутствия ошибок в программе. Например, из опыта следует, что при глубине вложения структур управления, более или равной трем, сильно возрастает вероятность совершения ошибки.

В качестве характеристики структурного программирования часто используется термин «*программирование без goto*». Структурное программирование не запрещает использование оператора **goto**, однако применение оператора **goto** делает текст программы менее понятным.

### 3.2. Описание алгоритма с помощью алгоритмического языка

Распространенным способом описания алгоритма является его представление с помощью алгоритмического языка (языка программирования). Алгоритмический язык в общем случае – это система обозначений и правил для точной и единообразной записи алгоритмов, а также их исполнения [45].

Следует отметить, что понятия «алгоритмический язык» и «язык программирования» являются разными понятиями. В качестве исполнителя в алгоритмическом языке может служить не только компьютер, но и устройство для работы «в обстановке», так как программа, написанная на алгоритмическом языке, не обязательно предназначается компьютеру. Практическая реализация алгоритмического языка в каждом конкретном случае – это отдельный вопрос.

Алгоритмический язык имеет свой *словарь*, основу которого составляют слова для записи команд, которые входят в систему команд исполнителя алгоритма. Такие команды называются *простыми командами*. Кроме этого в алгоритмическом языке используются *служебные слова*, смысл и способ употребления которых заданы раз и навсегда. Применение служебных слов служит для единообразной формы представления различных алгоритмов, а также делает запись алгоритма более наглядной.

Каждый алгоритм, созданный на алгоритмическом языке, должен иметь название, перед которым записывается служебное слово **АЛГ** (АЛГоритм). После названия алгоритма с новой строки записывается служебное слово **НАЧ** (НАЧало), а затем записываются его команды, причем каждая команда тоже записывается с новой строки. Конец алгоритма помечается служебным словом **КОН** (КОНец) Таким образом, последовательность записи алгоритма имеет вид [15, 61]:

**АЛГ** название алгоритма

**НАЧ**

Команды алгоритма

**КОН**

При разработке новых алгоритмов можно использовать алгоритмы, построенные ранее. Ранее созданные алгоритмы, которые целиком используются в составе других алгоритмов, называются *вспомогательными алгоритмами*. В качестве вспомогательного алгоритма может быть и алгоритм, который содержит ссылку на вспомогательные алгоритмы.

При разработке алгоритмов часто применяют *встроенные* (или *стандартные*) алгоритмы. Это такие алгоритмы, которые постоянно имеются в распоряжении исполнителя.

Если алгоритм содержит обращение к самому себе как вспомогательному, то он называется *рекурсивным алгоритмом*. В случае, когда команда обращения алгоритма к самому себе находится в самом алгоритме, рекурсия называется *прямой*. Если рекурсивный вызов данного алгоритма происходит из вспомогательного алгоритма, который вызывается в данном алгоритме, то рекурсия называется *косвенной*.

В случае, когда порядок выполнения команд алгоритма зависит от проверки некоторых условий, алгоритм называется *разветвляющимся*. Для описания таких алгоритмов в алгоритмическом языке применяется составная команда – *команда ветвления*. Эта команда может иметь либо полную форму:

```
ЕСЛИ условие  
    ТО команды 1  
    ИНАЧЕ команды 2  
ВСЕ
```

либо сокращенную форму:

```
ЕСЛИ условие  
    ТО команды  
ВСЕ
```

и соответствует блок-схеме «альтернатива» (рис. 3.2.b).

Развитием команды ветвления является *команда выбора*. На алгоритмическом языке она записывается в виде:

```
ВЫБОР  
    ПРИ условие 1: команды 1  
    ПРИ условие 2: команды 2  
  
    ПРИ условие N: команды N  
    ИНАЧЕ команды N+1  
ВСЕ
```



Алгоритмы, при выполнении которых команда или несколько команд выполняются неоднократно, называются *циклическими алгоритмами*. На алгоритмическом языке организация циклических алгоритмов осуществляется с помощью составной *команды цикла*.

Команда цикла с предусловием

**ПОКА** условие  
**НЦ**  
 команды  
**КЦ**

Здесь НЦ и КЦ означают соответственно начало и конец цикла. Данная команда цикла соответствует блок-схеме «итерация» (рис. 3.2.c).

Команда цикла с постусловием

**НЦ**  
 команды  
**ДО** условие  
**КЦ**

соответствует блок-схеме «итерация» (рис. 3.2.d).

Следуя [45], рассмотрим алгоритм, составленный для исполнителя-робота. По этому алгоритму робот должен перенести объекты со склада в левый нижний угол рабочего поля, имеющего вид:

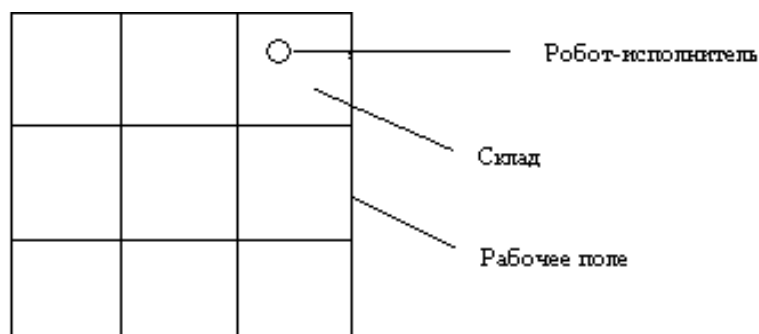


Рис. 3.6. Рабочее поле для робота-исполнителя

Система команд исполнителя содержит следующие команды:

<b>ВПЕРЕД</b>	– переход на одну клетку в направлении ориентации
<b>ВПРАВО</b>	– поворот на месте на $90^0$ вправо
<b>ВЛЕВО</b>	– поворот на месте на $90^0$ влево
<b>ВЗЯТЬ</b>	– захват объекта; должен находиться в той же клетке, что и объект
<b>УСТАНОВИТЬ</b>	– установка объекта в данной клетке поля

Первый вспомогательный алгоритм имеет следующее содержание:

```
АЛГ до_края
НАЧ
  ПОКА не_край
    НЦ
      вперед
    КЦ
  КОН
```

Второй вспомогательный алгоритм, использующий первый вспомогательный алгоритм, имеет вид:

```
АЛГ в_угол
НАЧ
  до_края
  вправо
    до_края

  вправо
  КОН
```

Основной рекурсивный алгоритм:

```
АЛГ перенос
НАЧ
  в_угол
  ЕСЛИ есть
    ТО
      взять
      в_угол
      установить
        перенос
    ИНАЧЕ
      в_угол
  ВСЕ
  КОН
```

## ГЛАВА 4. ЭТАПЫ ПОСТРОЕНИЯ АЛГОРИТМОВ

При решении задач с использованием ЭВМ необходимо выполнить следующие этапы:

1. Постановка задачи.
2. Формализация задачи.
3. Построение алгоритма.
4. Составление программы на языке программирования.
5. Отладка и тестирование программы.
6. Проведение расчетов и анализ полученных результатов.

Эта последовательность этапов называется *технологической цепочкой решения задач на ЭВМ*. Из данного списка непосредственно к программированию относятся пункты 3, 4, 5.

На *этапе постановки задачи* четко формулируется, что дано и что должны получить в результате решения задачи. Важным моментом здесь является определение полного набора исходных данных. Неправильно выделенные исходные или требуемые данные могут привести к результатам, которые не могут нас удовлетворить. Поэтому на этапе постановки задачи, во-первых, необходимо определить и перечислить все исходные и требуемые данные, и, во-вторых, определить условия, при которых возможно получение требуемых результатов, а при которых нет. И, наконец, нужно определить, какие результаты будут считаться правильными. Итак, четкая формулировка задачи означает извлечение из информации об изучаемом явлении или объекте исходных данных и определение того, что будет результатом. Следует отметить, что точность и четкость постановки задачи является половиной успеха ее решения.

На *этапе формализации задачи* обычно задача переводится на язык математических формул, уравнений, отношений. Если при решении требуется математическое описание некоторого реального объекта, либо явления, либо процесса, то формализация равносильна соответствующей *математической модели*.

На *этапе построения алгоритма* опытные программисты обычно сразу пишут программы на языках программирования. При этом они не применяют какие-либо способы описания алгоритмов, например блок-схемы. Однако в учебных целях полезно использовать блок-схемы, а затем переводить полученный алгоритм на язык программирования. Процесс создания алгоритма состоит, во-первых, в подборе и упорядочении действий для осуществления ввода-вывода информации и, во-вторых, в организации вычислений, строго соответствующих выбранным методам решения задач. При конструировании алгоритма методом пошаговой детализации необходимо обязательно следить за тем, чтобы алгоритм был рациональным и удовлетворял всем свойствам.

На *этапе составления кода программы на языке программирования* осуществляется перевод алгоритма в программу. Написание программы на вы-

бранном языке программирования, то есть собственно программирование, при наличии алгоритмов решения задач является кодированием алгоритмов на языке программирования.

На *этапе отладки и тестирования программы* решение задачи состоит в проведении компьютерного эксперимента. При этом необходимо многократно запускать программу на выполнение при различных значениях начальных условий. При исследовании информационной модели в виде программы в какой-либо среде программирования необходимо выполнить следующие действия [37]:

- а) запустить выбранную среду программирования;
- б) набрать код программы;
- в) сохранить этот код на диске;
- г) запустить программу на выполнение.

На *этапе проведения расчетов и анализа полученных результатов* выполняется созданная и отлаженная программа, анализируются получаемые результаты и корректируется, если необходимо, исследуемая математическая модель с повторным выполнением этапов 2 – 5.

#### 4.1. Основные свойства алгоритма

Чтобы алгоритм выполнил свое предназначение, он должен обладать следующими основными свойствами [23, 26 – 28]:

1. **Результативность.** Необходимо задать множество объектов, с которыми алгоритм будет работать. Формализованное (закодированное) представление этих объектов носит название *исходных данных*. Алгоритм приступает к работе с некоторым набором исходных данных, которые называются *входными*, и в результате своей работы выдает данные, которые называются *выходными*. Цель выполнения алгоритма – получение *конкретного результата*, имеющего определенное отношение к исходным данным.
2. **Определенность.** Для решения задачи каждый шаг алгоритма должен быть четко определен и не допускать произвольной трактовки.
3. **Массовость.** Один и тот же алгоритм может применяться для решения множества задач данного класса, которые различаются данными.
4. **Дискретность.** Алгоритм осуществляется конечной последовательностью шагов, т.е. решение задачи алгоритм сводит к решению отдельных более простых задач.
5. **Эффективность.** Алгоритм должен быть выполнен за разумно конечное время, а не просто за конечное время.
6. **Конечность.** Решение задачи обязательно получается за конечное число шагов при действии в соответствии с алгоритмом. При этом строится

бесконечный процесс, который сходится к искомому решению. Процесс обрывается на некотором шаге. Полученное значение принимается за приближенное решение рассматриваемой задачи, точность которого зависит от числа шагов.

7. **Компактность.** Данное свойство предполагает лаконичность изложения алгоритма. При потере компактности алгоритм в большой мере теряет право на существование.

## 4.2. Типы алгоритмов

По способам построения и реализации можно выделить следующие типы алгоритмов.

**Линейный алгоритм** – это набор предписаний, где все шаги выполняются последовательно друг за другом один раз в порядке их следования.

Допустим, дано  $X$ . Вычислить  $Z = Y^2$  и  $Z_1 = 1/Z$ , если  $Y = X^3 + 7$ .

Словесное описание линейного алгоритма имеет следующий вид:

1. Ввести  $X$ .
2. Вычислить  $Y = X^3 + 7$ .
3. Вычислить  $Z = Y^2$ .
4. Вычислить  $Z_1 = 1/Z$ .
5. Вывести  $Z, Z_1$ .
6. Конец.

Следует отметить, что команды словесного алгоритма нумеруются, чтобы можно было на них ссылаться.

**Разветвляющийся (или разветвленный) алгоритм** – алгоритм, содержащий блок проверки некоторого условия (логического выражения). В зависимости от результата проверки условия изменяется последовательность выполнения шагов алгоритма, т.е. выполняется та или иная последовательность команд. При этом осуществляется выбор одного из двух или нескольких возможных вариантов. Такую конструкцию можно записать в виде:

**ЕСЛИ** условие справедливо,  
**ТО** выполнить команды 1,  
**ИНАЧЕ** выполнить команды 2.

Построим алгоритм для вычисления функции:

$$z = \begin{cases} X - 5, Y > 0, \\ X + 2 - Y, Y < 0. \end{cases} \quad (4.1)$$

Словесное описание разветвляющегося алгоритма для вычисления функции (4.1) имеет следующий вид:

1. Ввести  $X$ .
2. Если  $Y > 0$ , то  $Z = X - 5$ .
3. Если  $Y < 0$ ,  
Вывести  $Z = X + 2 - Y$ .
4. Конец.

На рис. 4.1. представлено графическое описание разветвляющегося алгоритма для вычисления функции (4.1).

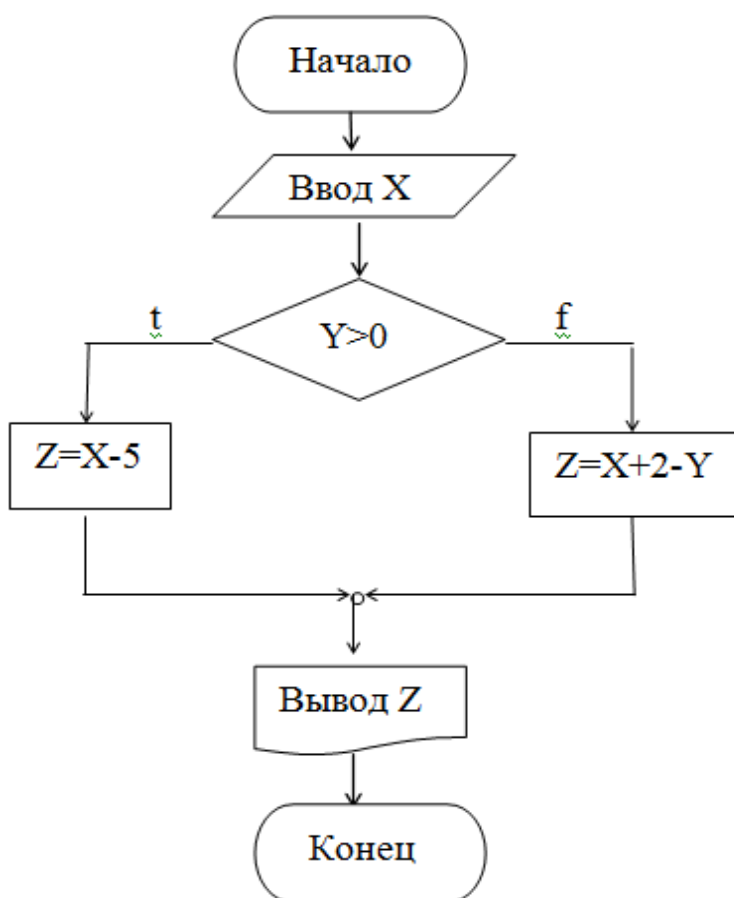


Рис. 4.1. Графическое изображение разветвляющегося алгоритма

**Циклический алгоритм** – это алгоритм, описывающий повторение одних и тех же вычислений над различными вариантами данных для получения необходимого результата. Требуемые вычисления производит основной блок цикла – *тело цикла*. Циклический процесс организуют вспомогательные блоки цикла. Они устанавливают начальное значение, новые значения данных и проверяют условие окончания циклического процесса.

Для организации любого цикла необходимо выполнить следующие действия:

- Задать перед началом выполнения цикла начальные значения параметров цикла.
- Изменять параметры цикла перед каждым следующим повторением цикла.
- Проверять условия повторения цикла либо окончания цикла.
- Переходить к началу цикла, если он не закончен, или выходить из цикла по его окончании.

На рис. 4.2 приведен *цикл с предварительным условием* – *цикл с предусловием*. Тело такого цикла может не выполняться ни одного раза.

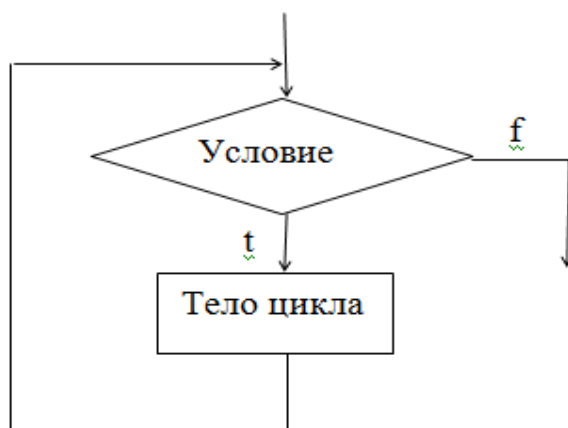


Рис. 4.2. Цикл с предварительным условием

На рис. 4.3 представлен *цикл с последующим условием* – *цикл с постусловием*. Тело цикла постусловием выполнится хотя бы один раз.

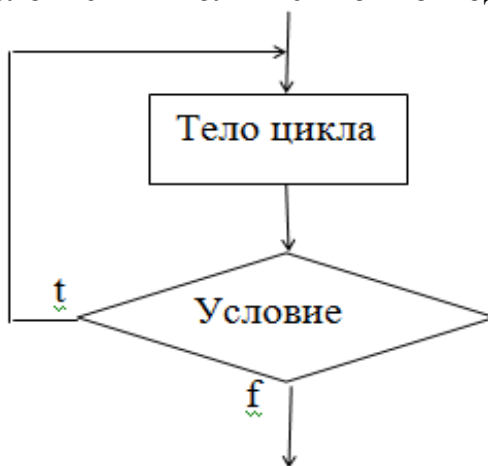


Рис. 4.3. Цикл с последующим условием

По способу определения числа повторений циклы подразделяются на:

- циклы с неизвестным числом повторений;
- циклы с явно заданным числом повторений.

Например, в языке С++ циклами с неизвестным числом повторений являются *while* и *do...while*, а циклом с явно заданным числом повторений – *for*.

### 4.3. Построение алгоритма

Рассмотрим подробнее *этап построения алгоритма*.

*Полное построение алгоритма* можно разбить на следующие этапы, последовательное выполнение которых приведет к построению программы, максимально свободной от логических ошибок [14].

1. **Постановка задачи** с обсуждением терминологии, которая характерна для предметной области, и непротиворечивости описываемых явлений.
2. **Построение математической модели, адекватной поставленной задаче.** Определить понятия решения построенной математической модели и корректности этого решения.
3. **Анализ существующих алгоритмов** для решения данного класса задач. Оценить трудности, которые не позволяют использование этих алгоритмов при решении поставленной задачи.
4. **Разработка собственного алгоритма и оценка его эффективности.** При этом отметить, что построение алгоритма – задача, не обладающая существованием единственного решения. Это приводит к обязательной оценке эффективности различных алгоритмов.
5. **Доказательство правильности алгоритма.** Необходимо показать сходимость решения, полученного с помощью разработанного алгоритма, к решению исходной задачи.
6. **Описания алгоритма** (базовые структуры, метод разработки сверху вниз, иерархические структуры и др.).
7. **Реализация алгоритма.** Необходимо составить программу на языке программирования.
8. **Отладка и тестирование программы на ЭВМ.** Нужно провести поиск и исправление ошибок в программе, а также проверить программу на ЭВМ с помощью тестов.
9. **Составление документации.** Нужно составить документацию к решенной задаче: к математической модели, алгоритму, программе, набору тестов, использованию программы.

Рассмотрим подробнее этапы разработки алгоритма. Эти этапы связаны друг с другом и объединяются в единое целое. Каждый этап характеризуется серией вопросов, ответы на которые позволяют правильно провести разработку алгоритма.



### 4.3.1. Постановка задачи

На первом этапе необходимо точно сформулировать задачу, т.е.

- ✓ определить цель решения задачи;
- ✓ описать ее содержание;
- ✓ проанализировать сущность и характер всех величин, которые используются в задаче;
- ✓ определить условия, при которых задача решается.

Процесс точной формулировки задачи требует постановки правильных вопросов. Следует получить ответы, например, на такие вопросы:

- Понятна ли используемая терминология и однозначна ли она?
- Что является входными и выходными данными?
- Все ли данные являются необходимыми для исследуемой задачи и не опущены ли какие-либо необходимые данные?
- Какие сделаны допущения?

Корректность постановки задачи является важной, поскольку от нее зависят другие этапы построения алгоритма.

В качестве примера рассмотрим следующую задачу. Специалист по наладке торгового оборудования обслуживает сеть магазинов одного из районов города Нижний Новгород. Фирма оплачивает ему 50% стоимости затраченного бензина при обслуживании клиентов. Специалист вычислил стоимость проезда между магазинами и стоимость проезда от своего дома до каждого из магазинов. Задача состоит в минимизации стоимости затрат на бензин.

Итак, в качестве входных данных используются перечень магазинов и матрица стоимости затрат на проезд из магазина  $i$  в магазин  $j$ . Результатом будет маршрут, который минимизирует стоимость затрат на бензин. Однако в такой постановке получить решение задачи весьма затруднительно, поскольку необходима дополнительная информация. Например, есть ли магазины, посещение которых обязательно, или необходимо посетить все магазины, т.е. при составлении графика поездок нужно учитывать предпочтения. Допустим, что должны посещаться все магазины, причем по одному разу, а начинается поездка всегда от его дома и заканчивается там же. В такой постановке формулировка задачи становится точной и ясной.

### 4.3.2. Построение математической модели

Если имеется точно поставленная задача на этапе её постановки, то необходимо сформулировать для нее математическую модель, т.е. формализовать

задачу: нужно записать математические соотношения (формулы, уравнения, неравенства и т.д.), которые связывают результаты с исходными данными.

Формализация постановки задачи позволяет определить путь, который ведет от постановки решения задачи к ее решению. В процессе разработки алгоритма решения задачи выбор модели оказывает существенное влияние на остальные этапы этого процесса. Набора правил, которые автоматизируют стадию моделирования, нет. Поэтому большинство задач рассматриваются индивидуально. Однако имеются следующие руководящие принципы, которые могут быть полезными на стадии формализации алгоритма, а именно:

- ✓ изучение различных моделей способствует приобретению опыта в моделировании;
- ✓ выбор модели происходит интуитивно на основе этого опыта.

На этапе построения математической модели следует, прежде всего, ответить на два вопроса [14, 45]:

- Какие существующие математические модели подходят для решения данной задачи?
- Имеются ли решенные аналогичные задачи?

Второй вопрос является важным, поскольку разработано большое количество алгоритмов решения разных задач, и возможно решение поставленной задачи может быть достигнуто путем комбинации или модернизации существующих алгоритмов. Таким образом, в контексте моделирования второй вопрос дает ответ на первый вопрос.

Первый вопрос определяется тем, что мы хотим описать математически, что задано и что мы хотим получить. На выбор математической структуры оказывают влияние:

- ✓ ограниченность наших знаний небольшим количеством математических моделей из-за малого опыта моделирования;
- ✓ удобство описания входных данных;
- ✓ разработанные действия над данными;
- ✓ простота выполнения определенных над ними операций.

Определившись с математической структурой, необходимо переформулировать исходную задачу в терминах соответствующих математических объектов. Это можно считать одной из возможных математических моделей, если дать утвердительный ответ на следующие вопросы.

- Все ли входные данные и другая важная информация хорошо описаны математическими объектами?
- Есть ли математическая величина, отождествляемая с решением?

- Отражены ли все отношения, существующие в исходной постановке задачи?
- Не появилось ли новые полезные связи между объектами модели, не отраженные в исходной постановке?
- Удобно ли работать с данной моделью?

Таким образом, математическая модель должна быть реалистичной и реализуемой.

Рассмотрим пример, приведенный в разделе 4.3.1. Предположим, что специалист обслуживает шесть магазинов. Магазины и дом специалиста можно представить с помощью *графа* (или *сети*). Его вершинами являются магазины и дом специалиста, который определим нулевым индексом. Вес рёбер графа совпадает с суммой затрат проезда от вершины  $i$  до вершины  $j$ . Стоимость затрат можно описать с помощью *матрицы смежности*. Граф и матрица смежности (A) приведены на рис. 4.1. Кроме того, будем считать, что затраты переезда из пункта  $i$  в пункт  $j$  равны затратам переезда в обратном направлении, то есть матрица смежности симметричная.

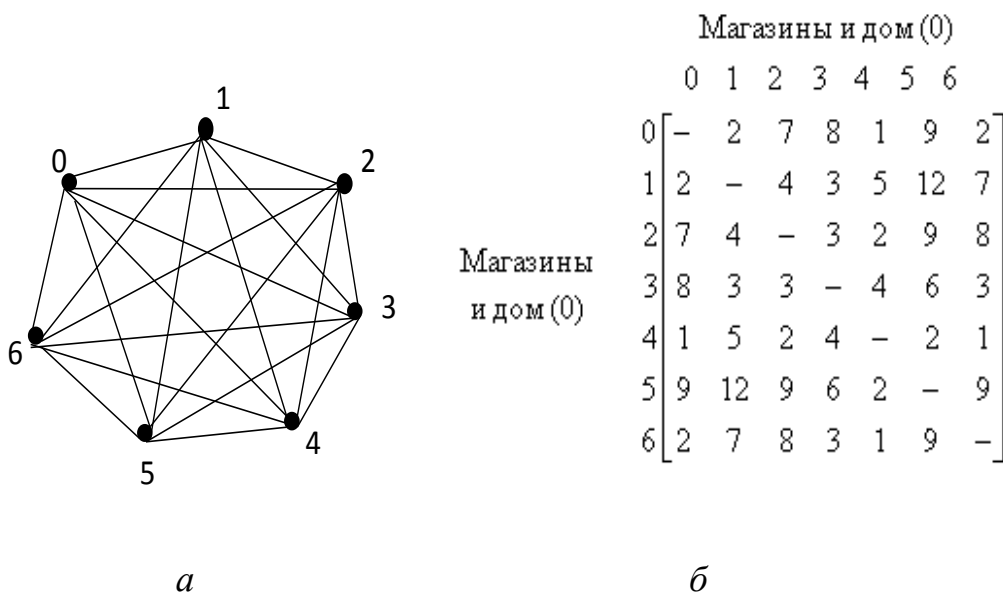


Рис. 4.1. Граф (а) и матрица смежности (б), описывающие обслуживание магазинов: вершина 0 – дом, вершины 1- 6 – магазины

Решением задачи является замкнутый цикл, который начинается в вершине 0, проходит через все вершины и заканчивается в вершине 0. Данный цикл соответствует неотрывному движению карандаша вдоль линий сети, которое начинается и оканчивается в одной и той же точке и проходит через каждую точку только один раз. Обход такого рода называется *туром* [14]. Стоимость тура – это сумма весов всех пройденных рёбер. Стоимость тура обслуживания шести магазинов равна

$$s = A[0,r[0]] + A[r[0], r[1]] + A[r[1], r[2]] + A[r[2], r[3]] + \\ + A[r[3], r[4]] + A[r[4], r[5]] + A[r[5],0], \quad (4.2)$$

где  $r$  – массив, содержащий номера вершин в порядке их обхода;  $A$  – матрица смежности;  $A[r[i], r[j]]$  – стоимость пути из вершины  $r[i]$  в вершину  $r[j]$ .

Эта стоимость должна быть минимальной. Следует отметить, что решение данной задачи может быть не единственным, так как минимальное значение может дать не единственный путь. В этом случае в качестве решения можно выбрать любой из них. В литературе эта задача носит название «Задача о коммивояжёре».

### 4.3.3. Анализ существующих алгоритмов

При решении конкретной задачи необходимо рассмотреть различные существующие алгоритмы, которые могут быть использованы. Например, при решении задачи о коммивояжёре, рассмотренной в разделе 4.3.2, каждый тур однозначно соответствует единственной перестановке целых чисел от 1 до 6. С другой стороны, каждая перестановка этих чисел соответствует единственному туру. Следовательно, для любой данной перестановки на сетевой модели, изображенной на рис. 4.1, можно проследить соответствующий тур и в то же время вычислить его стоимость.

Задачу о коммивояжёре можно решать и по-другому, например, применяя алгоритм получения перестановок целых чисел, образуем *все* перестановки чисел от 1 до 6. Далее для каждой перестановки нужно построить соответствующий тур и вычислить его стоимость. Затем необходимо использовать алгоритм поиска минимума в ограниченном множестве чисел для нахождения тура с минимальной стоимостью.

Если возникает необходимость решения систем алгебраических уравнений, то существует множество алгоритмов для их решения. При этом чаще всего применяется комбинация имеющихся алгоритмов, например, алгоритм вычисления определителя матрицы, алгоритм проверки определителя на равенство его нулю и лишь затем используется один из существующих алгоритмов решения систем алгебраических уравнений.

### 4.3.4. Разработка собственного алгоритма и оценка его эффективности

После того, как построена математическая модель поставленной задачи и проведен анализ существующих алгоритмов, на основе математической модели начинается этап конструирования собственного алгоритма. Для этого нужно:

- ✓ выбрать метод проектирования алгоритма;
- ✓ определить форму описания алгоритма;
- ✓ выбрать тесты и методы тестирования;
- ✓ осуществить проектирование алгоритма.

При этом выбор метода разработки алгоритма существенно зависит от выбранных моделей данных. Наиболее распространенными методами проектирования алгоритмов являются метод частных целей и эвристический алгоритм, которые рассматриваются в Главе 5.

Следует отметить, что на стадии разработки алгоритма необходимо тщательное обдумывание, так же нужно уделять внимание предшествующим и следующим за стадией разработки этапам, поскольку все этапы построения алгоритма взаимосвязаны. На любом этапе при записи действий должны учитываться, что исполнителем алгоритма будет *компьютер*. Он умеет совершать только вполне определенные действия: присваивание, ветвление, циклическое повторение. Поэтому при построении алгоритма любая его детализация должна приводить к данным конструкциям. Таким образом, в процессе построения алгоритма получаются в итоге последовательности алгоритмических конструкций.

Построение алгоритма – это задача, не обладающая существованием единственного решения, что приводит к обязательной оценке эффективности разработанных алгоритмов. С *практической* точки зрения одной из важных причин для анализа алгоритмов является необходимость получения оценок либо границ для относительно дефицитных ресурсов – машинного времени и памяти. На эти ресурсы одновременно могут претендовать многие пользователи. При этом во время анализа алгоритма можно выявить те разделы программы, на которые расходуется большая часть времени.

С точки зрения *теории* желательно:

- ✓ иметь некоторый количественный критерий для сравнения алгоритмов, которые разработаны для решения одной и той же задачи;
- ✓ иметь механизм для выявления наиболее эффективных алгоритмов;
- ✓ установить абсолютный критерий эффективности алгоритма, когда решение задачи можно считать оптимальным.

Отметим, что в некоторых случаях невозможно составить четкое мнение об относительной эффективности алгоритмов решения задачи. Например, один из алгоритмов может лучше работать со специальными входными данными, а другой – со случайными входными данными. Поэтому при рассмотрении алгоритмов необходимо проводить их сравнительный анализ с целью выявления их достоинств.

Следуя [14], рассмотрим алгоритм  $A$  для решения некоторого класса задач. Пусть  $n$  – размерность отдельной задачи из данного класса. Например, для

задачи о коммивояжёре, рассмотренной в разделе 4.3.2,  $n$  равно числу вершин графа. Функцию  $f_A(n)$  определим как *рабочую функцию*, значением которой является верхняя граница максимального числа основных операций при реализации алгоритма для любой задачи размерности  $n$ . Для оценки качества алгоритма часто используется следующий критерий, который основан на времени работы в худшем случае:

Алгоритм  $A$  называется *полиномиальным*, если  $f_A(n)$  растет не быстрее, чем полином от  $n$ , в противном случае алгоритм называется *экспоненциальным*.

Практическая основа этого критерия качества алгоритма заключается в том, что последовательные или параллельные машины способны лучше воспринимать для задач большой размерности полиномиальные алгоритмы, чем экспоненциальные.

Функцию  $f_A(n)$  можно использовать при оценке времени работы алгоритма, поскольку обычно мера эффективности – это скорость и время, в течение которого алгоритм выдает результат [14, 28].

Для сравнения двух алгоритмов  $A$  и  $B$  вычисляется предел отношения рабочих функций:

$$\lim_{n \rightarrow \infty} \frac{f_A(n)}{f_B(n)}. \quad (4.3)$$

Если

$$\lim_{n \rightarrow \infty} \frac{f_A(n)}{f_B(n)} = \text{const} \neq 0, \quad (4.4)$$

то рабочие функции возрастают с одинаковой скоростью при  $n \rightarrow \infty$ . Пишут

$$f_A(n) = O[f_B(n)]. \quad (4.5)$$

В этом случае говорят, что данные алгоритмы одного порядка при  $n \rightarrow \infty$ , то есть при реализации алгоритмов будет затрачено примерно одно и то же время.

Если

$$\lim_{n \rightarrow \infty} \frac{f_A(n)}{f_B(n)} = 0, \quad (4.6)$$

то рабочая функция  $f_B(n)$  возрастает быстрее при  $n \rightarrow \infty$  по сравнению с рабочей функцией  $f_A(n)$ . Пишут

$$f_A(n) = o[f_B(n)]. \quad (4.7)$$

В данном случае при реализации алгоритма  $B$  будет затрачено большее время, чем при выполнении алгоритма  $A$ .

Детальные сведения об алгоритме можно получить только из его конкретной реализации. В таких случаях возможен более точный анализ алгоритмов, поскольку будут определены конкретные формулы для вычисления рабочих функций.

Задача о коммивояжёре, рассмотренная в разделах 4.3.1 и 4.3.2, относится к классу *NP-полных задач* [28], которые практически значимы, но для них в настоящее время неизвестны эффективные алгоритмы их решения, а известны только экспоненциальные алгоритмы. Поэтому при решении таких задач важно пользоваться наиболее эффективным из экспоненциальных алгоритмов. Например, алгоритм, который решает задачу размерности  $n$  за  $O(2^n)$  шагов, является предпочтительным по сравнению с алгоритмами, решающими эту задачу за  $O(n!)$  шагов, так как

$$\lim_{n \rightarrow \infty} \frac{2^n}{n!} = 0, \quad (4.7)$$

то есть  $2^n = o[n!]$ .

Рассмотрим следующий алгоритм решения задачи о коммивояжёре, рассмотренной в разделе 4.3.2:

- Пока есть возможные перестановки,
  - ✓ получить текущую перестановку;
  - ✓ вычислить суммарную стоимость тура;
  - ✓ определить является ли она минимальной.
- Печатать тур, дающий минимальную стоимость.
- Печатать полученную минимальную стоимость.

Проведем оценку рабочей функции  $f_A(n)$  этого алгоритма. Число перестановок  $n$  элементов равно  $n!$ , Если сделать очень грубую оценку, что перестановку можно получить за одну операцию, то  $f_A(n) \sim n!$ . Поскольку эта функция растет быстрее чем многочлен любой конечной степени, то предложенный алгоритм будет экспоненциальным и использовать его при большом числе магазинов  $n$  нельзя, так как требуемое время для его выполнения растет слишком быстро.

#### 4.3.5. Доказательство правильности алгоритма

Данный этап является одним из сложных и продолжительных этапов разработки алгоритма. При доказательстве правильности алгоритма обычно применяется прогон его на различных тестовых задачах, решение которых известно, либо сравнение полученных результатов с существующими экспериментальными данными. Однако такой способ доказательства правильности алгоритма не исключает такие начальные условия, при которых результаты не будут совпадать.

Следуя [14], рассмотрим следующую общую методику доказательства правильности алгоритма. Предположим, что алгоритм состоит из  $m$  последовательных шагов. Необходимо доказать обоснованность каждого из шагов и правомерность перехода от шага  $i$  к шагу  $i+1$ . Затем доказать ограниченность  $m$ , то есть конечность алгоритма. При этом нужно проверить все подходящие входные данные и получить все подходящие выходные данные.

В качестве примера проведем доказательство экспоненциального алгоритма ETS (*Исчерпывающий коммивояжёр*) для задачи из раздела 4.3.2, суть которого состоит в следующем. Алгоритм требует в качестве входных данных число обслуживаемых специалистом магазинов и матрицу стоимостей. Последовательно рассматриваются все перестановки от 1 до  $n$  целых чисел. Таким образом, рассматривается каждый возможный тур и выбирается вариант с наименьшей стоимостью.

Докажем правильность данного алгоритма. Так как проверяется каждый тур, то должен быть проверен и тур с минимальной стоимостью. Этот тур будет запомнен. Если существует тур, стоимость которого меньше стоимости минимального тура, то минимальный тур будет отброшен. Число проверяемых туров конечно, поэтому алгоритм должен закончить работу.

Приведенный метод доказательства правильности алгоритма основан на переборе всех возможных вариантов туров и называется «*доказательство исчерпыванием*». Этот метод является самым грубым среди всех методов доказательства правильности алгоритмов.

Следует отметить, что правильность и эффективность алгоритма являются разными понятиями. Если алгоритм правильный, то это еще ничего не говорит о его эффективности.

#### **4.3.6. Описания алгоритмов**

Способы описания алгоритмов приведены в Главе 3. Здесь отметим, что наиболее распространенным способом описания алгоритмов является структурный метод «сверху-вниз» с использованием блок-схем. В алгоритмах, разработанных данным методом, меньше ошибок. Правильность в них встроена шаг за шагом, следовательно, её легче доказать.

#### **4.3.7. Реализация алгоритма**

Алгоритм может быть реализован либо путем кодирования на конкретном языке программирования, либо с использованием специального программного приложения. Для программирования обычно применяются языки высокого уровня. Составленная программа переводится на машинный язык ЭВМ. После этого выполняется соответствующая машинная программа.

Для реализации алгоритма необходимо выполнить следующие действия:



- ✓ выбрать нужный язык программирования;
- ✓ уточнить способы организации данных;
- ✓ записать алгоритм на выбранном языке программирования.

После выбора языка программирования нужно определить структуры данных, используемые в данном алгоритме, либо построить целую систему структур данных, чтобы отразить все аспекты построенной модели. При решении этой задачи необходимо дать ответы на вопросы:

- Какие основные переменные будут использованы для описания параметров модели?
- Каковы типы переменных?
- Есть ли необходимость использовать массивы?
- Сколько нужно массивов, и какой они размерности?
- Возникает ли необходимость использовать связанные списки?
- Какие стандартные подпрограммы или функции нужно применить?
- Какой будет вид интерфейса при работе с готовой программой?

При реализации алгоритма некоторые его шаги могут представлять собой целые задачи, требующие реализации в виде подпрограмм или функций. Например, в задаче о коммивояжёре, рассмотренной выше, шаг получения перестановок может быть записан в виде подпрограммы.

Важно помнить, что одно дело доказать правильность алгоритма и совсем другое дело – доказать, что его программная реализация соответствует построенному алгоритму. За этим необходимо тщательно следить в процессе написания программы.

#### **4.3.8. Отладка и тестирование программы на ЭВМ**

Эксплуатации программы предшествуют этапы её *отладки* и *тестирования*. Оба этапа сводятся к прогону программы на ЭВМ, но по своему назначению они имеют разные цели. Отладка позволяет выявить ошибки в программе, а тестирование показывает возможности программы при эксплуатации.

Процесс отладки программы состоит из двух видов:

- ✓ синтаксической отладки;
- ✓ отладки семантики и логической структуры.

и заключается в поиске и устранении синтаксических, семантических и логических ошибок. При этом прогон программы можно осуществить на простом примере, который может быть проверен вручную.

В процессе синтаксического контроля программы транслятор выявляет сочетания символов и конструкции, которые недопустимы с точки зрения правил их построения или написания, принятых в используемом языке программирования. Сообщения об ошибках ЭВМ выдает на экран дисплея, причем вид и форма выдачи сообщений зависят от языка программирования и версии используемого транслятора.

После устранения синтаксических семантических ошибок необходимо проверить логику работы программы в процессе ее выполнения на ЭВМ. Для этого в программе можно выбрать контрольные точки и для них вручную рассчитать промежуточные результаты. Эти результаты нужно сверить со значениями, получаемыми ЭВМ в контрольных точках при выполнении отлаживаемой программы.

После того, как в программе исправлены все синтаксические, семантические и логические ошибки, необходимо:

- ✓ выполнить тестовые расчеты;
- ✓ провести анализ результатов тестирования;
- ✓ совершенствовать программу.

*Тестирование программ* – это процесс проверки программ на ЭВМ с помощью *тестов*, то есть наборов входных данных с перечнем правильных результатов. Важным моментом для процесса тестирования программы является выбор входных данных. Встает вопрос: как выбрать входные данные? Общего ответа на этот вопрос нет, так как для любого алгоритма ответ зависит от имеющегося ресурса времени и от сложности алгоритма. Программа должна быть протестирована для широкого диапазона входных данных. Поскольку алгоритм конструируется для определенных допустимых входных данных, то для тестирования обязательно надо выбрать входные данные, не принадлежащие области допустимых значений. Далее, нужно иметь тестовые задания, в которых нарушены ограничения, сформулированные при постановке задачи. Это позволяет проследить, как программа реагирует на экстремальные ситуации.

Если, например, решается задача о коммивояжере с помощью исчерпывающего алгоритма ETS (см. раздел 4.5), то при числе входных данных (магазинов)  $\leq 5$ , алгоритм работает хорошо, а при их увеличении время работы программы возрастает очень быстро, и её эффективная эксплуатация становится невозможной при количестве магазинов  $\geq 10$ .

Следует отметить, что, не дожидаясь окончательного рационального алгоритма, нужно проводить его тестирование, чтобы быть уверенным в правильности алгоритма. В таком случае для регистрации исполнения алгоритма необходимо указать:

- ✓ шаги алгоритма;
- ✓ аргументы;
- ✓ промежуточные величины;

- ✓ результаты на конкретном шаге;
- ✓ проверяемые условия.

Шаги выполнения алгоритма показывают, сколько их исполнено на данный момент. Они также дают возможность определить, за какое число шагов весь алгоритм будет исполнен. Нумерация шагов алгоритма позволяет лучше ориентироваться в описании алгоритма и во время процесса его исполнения. При этом в каждый момент видно, какая команда алгоритма выполняется, что помогает устранению ошибок.

Таким образом, *отладка и тестирование программы* – это необходимый и важный этап построения алгоритма, дающий возможность экспериментально установить границы использования алгоритма (программы).

### 4.3.9. Составление документации

Этап составления документации для программы является обязательным и важным, с одной стороны, для её использования, а с другой – для долгого существования программы. Документация включает в себя всю информацию, поэтому процессы составления документации и разработки алгоритма должны проходить параллельно и взаимодействовать между собой. Главная цель документации – оказать помощь людям понять программы, написанные другими людьми, чтобы они могли эффективно их использовать.

Документация подразделяется на два типа: *внешнюю документацию* и *программную документацию*. Каждый программный продукт должен быть снабжен как внешней, так и программной документацией.

*Внешняя документация* обычно оформляется либо в виде инструкции для пользователей, либо в виде технического отчета. Но в зависимости от размера и сложности программы ее можно, например, оформить как блок-схему, полное описание построения сверху-вниз, словесное описание алгоритма. Во внешней документации должна присутствовать информация, не содержащаяся в самой программе, например, описание интерфейса, ограничений на входные данные.

Внешняя документация должна также содержать информацию об особенностях программы и об авторских правах на программный продукт.

Для создания *программной документации* не существует никаких жестких правил. Самой лучшей программной документацией является ясный и понятный код самой программы.

В начале программы необходимо указать заголовок программы, дату ее создания, фамилию и имя автора программы и другую информацию, которую автор считает нужной для документации. Например, можно кратко описать:

- ✓ какая задача решается с помощью данной программы;
- ✓ какие входные данные и выходная информация;

- ✓ какие функции и подпрограммы используются в программе.

В программную документацию желательно включать комментарии, поясняющие конкретные фрагменты программы. При этом они должны быть информативными и не повторять используемые в программе операторы. Лучше всего писать комментарии одновременно с кодом программы. Это будет способствовать правильному отражению всех особенностей и логики алгоритма в окончательном коде программы.

Для обеспечения наглядности программы необходимо писать код, используя пробелы и делая отступы от начала строки для фрагментов программы в соответствии с логическими уровнями, особенно в операторах циклов и условных операторах.

В программной документации должна быть вся необходимая информация для ввода данных и вывода результата, которые нужно правильно описать. В любой программе выходная информация – это самый важный документ. Выходную информацию следует представлять в удобном для чтения виде, используя соответствующие форматы для вывода информации.

Из всего сказанного в данном разделе следует, что программу необходимо оформлять в таком виде, в каком хотелось бы ее видеть написанной другими.

#### **4.4. Презентация и сопровождение программного продукта**

По характеру использования все программы распределяются на два класса:

- *утилитарные программы*, создаваемые для удовлетворения нужд их разработчиков;
- *программные продукты*, которые предназначены для удовлетворения потребностей пользователей, широкого их распространения и продажи.

*Программный продукт* – это программный комплекс, который соответствующим образом подготовлен к эксплуатации. Кроме программы с исполняемым кодом программный комплекс чаще всего содержит различные вспомогательные программы: утилиты, настройщики, справочники и т.д., а также необходимые для работы программ файлы с графической, текстовой, звуковой информацией. Программный комплекс должен иметь и необходимую техническую документацию, включая инструкцию для пользователя. Качество программного продукта характеризуют надежность, простота и эффективность его использования.

Основной потребитель программ – это *конечный пользователь*, который, как правило, не является специалистом в области программирования. *Презентация* программы является важным этапом, который позволяет продемонстри-

ровать пользователям, например, простоту ее эксплуатации, возможности применения для решения различных задач и т.д.

Каждый программный продукт имеет свой жизненный цикл. В условиях существования рынка программных продуктов важны следующие их характеристики:

- ✓ стоимость программного продукта;
- ✓ количество его продаж;
- ✓ время нахождения программного продукта на рынке;
- ✓ известность программы и фирмы-производителя;
- ✓ наличие на рынке программных продуктов, имеющих аналогичное назначение.

Поэтому со временем программы необходимо обновлять, развивать, модифицировать, дорабатывать для решения конкретных задач, увеличивая время полезной жизни. Такая деятельность относится к категории *обслуживания* программных продуктов.

Эксплуатация программ пользователями и их *сопровождение* со стороны разработчиков идут, как правило, параллельно. В процессе эксплуатации могут выявляться ошибки. Устранение этих ошибок ведётся обычно в режиме сопровождения: оказывается сервисная помощь, организуются «горячие телефонные линии» для консультаций, возможно обеспечение новыми версиями программ.

### **Упражнения**

1. С помощью датчика случайных чисел задается число и вводится его двоичное представление. Необходимо определить правильность его представления в двоичной системе счисления.
2. Число 123456789 обладает следующими интересными свойствами. Например, при умножении его на 2 или на 4 получается число, представленное цифрами от 1 до 9, а именно  $123456789 \cdot 2 = 246913578$  и  $123456789 \cdot 4 = 493827156$ . Построить алгоритм нахождения всех чисел, при умножении на которые заданного числа 123456789 получаются девятизначные числа, представленные цифрами от 1 до 9 без повторений.
3. Создать алгоритм построения числовой последовательности, которая строится следующим образом. Первый элемент последовательности – произвольное число, кратное 3. Второй элемент – число, полученное из суммы кубов цифр первого элемента. Следующий элемент – число, полученное из суммы кубов цифр предыдущего элемента. Определить число шагов, начиная с которого элемент последовательности станет равным 153, и последующие элементы будут совпадать по значению с ним, так как  $1^3 + 5^3 + 3^3 = 153$ .

4. Построить алгоритм получения числовой последовательности, образованной следующим образом. Первым элементом последовательности является любое четырехзначное число, не все цифры которого одинаковы. Второй элемент последовательности получается таким образом. Строим два числа: первое число – это первый элемент последовательности, цифры которого располагаются в порядке убывания слева на право; второе число – это первый элемент последовательности, цифры которого располагаются в порядке возрастания слева на право. Второй элемент последовательности – разность между первым и вторым числами. Следующий элемент последовательности получается аналогичным образом из предыдущего элемента последовательности. Например, первый элемент последовательности равен 7815. Второй элемент последовательности будет:  $8751 - 1578 = 7173$  и так далее. Определить число шагов, когда элемент последовательности станет равным 6174 и далее не будет изменяться, так как  $7641 - 1467 = 6174$ .
5. В пространстве  $\mathbf{R}^1$  задано  $n$  интервалов. Разработать алгоритм определения общей части этой совокупности интервалов, если она существует или указать ее отсутствие.
6. В пространстве  $\mathbf{R}^1$  задано  $n$  интервалов. Разработать алгоритм, который определяет, является ли это совокупностью вложенных друг в друга интервалов.
7. Задана последовательность символов, состоящая из цифр. Построить алгоритм, превращающий эту последовательность символов в число без использования библиотечных функций.
8. Задано целое число. Построить алгоритм, переводящий это число в последовательность символов без использования библиотечных функций.
9. Числовая последовательность образована следующим образом: первый элемент произвольное число, следующий элемент – число, полученное из квадратов цифр предыдущего элемента. Разработать алгоритм построения данной последовательности и определения числа шагов, когда элемент последовательности станет равным 1, либо образуется циклическая последовательность вида 89 145 42 20 4 16 37 58 89.
10. Разработать алгоритм нахождения приближенного значения корня уравнения  $f(x) = 0$  с точностью  $\varepsilon$  методом деления отрезка пополам.
11. Реакция организма на лекарство через  $n$  часов после инъекции задается последовательностью  $r_n = \alpha r_{n-1} + 0.4^n$ , где  $\alpha$  – положительное число, меньшее единицы, которое характеризует конкретный препарат лекарственной группы, и  $r_0 = 1$ . Разработать алгоритм, который определяет количество часов наступления максимальной реакции организма на введенный препарат, а также количество часов, когда реакция будет ниже 50% от первоначального уровня.
12. Численность двух конкурирующих популяций в  $n$ -ом году описываются величинами  $x_n, y_n$ . Их взаимное влияние на численность в  $n+1$  году задается уравнениями:  $x_{n+1} = 2x_n - y_n, y_{n+1} = -x_n + 2y_n$ .

- Начальные значения численности популяций заданы и не равны между собой. Построить алгоритм определения численности обеих популяций за все годы, предшествующие полному вымиранию любой из них.
13. Разработать алгоритм нахождения простых чисел, используя решето Эратосфена. Для построения решета Эратосфена нужно выполнить следующие действия. Необходимо создать список из  $n$  подряд идущих чисел. Оставив число 2, удалить из этого списка все числа, кратные 2. Затем оставляется следующее число 3, удалить все числа, кратные 3. Оставляя в списке следующее число, удалить из списка числа, кратные ему. Оставшиеся числа – это простые числа  $\leq n$ .
  14. Натуральное число назовем палиндромом, если его запись одинаково читается с начала и с конца. Например, 14233241. Разработать алгоритм нахождения всех чисел-палиндромов, меньших заданного числа  $N$ .
  15. Даны две прямые  $y=ax+b$  и  $y=cx+d$ . Построить алгоритм, который определяет, являются ли эти прямые параллельными. Если прямые не параллельны, найти точку их пересечения.
  16. Даны три не параллельные прямые  $y=ax+b$ ,  $y=cx+d$  и  $y=fx+g$ . Создать алгоритм определения координат вершин образованного ими треугольника.
  17. Задана окружность радиуса  $R$  с центром в точке  $M(x_0, y_0)$  и прямая линия  $y=ax+b$ . Разработать алгоритм определения точек пересечения прямой с окружностью. Если точек пересечения нет, то выдать соответствующее сообщение.
  18. Заданы три окружности радиусов  $R_0, R_1, R_2$  с центрами в точках  $O_1(x_0, y_0), O_2(x_1, y_1), O_3(x_2, y_2)$ . Построить алгоритм, который определяет, образуют ли прямые, соединяющие их центры, треугольник и если да, то вычисляет длины его сторон.
  19. Прямоугольники со сторонами, параллельными осям координат, задаются координатами двух вершин: левой верхней и правой нижней. Задано  $n$  прямоугольников. Разработать алгоритм определения прямоугольника, являющегося их пересечением. Если общей части прямоугольники не имеют, то выдать соответствующее сообщение.
  20. Заданы два квадрата со сторонами, параллельными осям координат. Они задаются левой верхней вершиной и длиной стороны. Создать алгоритм, определяющий, пересекаются ли эти квадраты. Если квадраты пересекаются, определить, является ли полученная фигура квадратом или прямоугольником.

## ГЛАВА 5. МЕТОДЫ СОЗДАНИЯ АЛГОРИТМОВ

При создании алгоритмов необходимо владеть некоторыми основными методами. К фундаментальным методам разработки алгоритмов относятся следующие методы [14]:

- ✓ метод частных целей;
- ✓ метод подъема;
- ✓ метод отрабатывания назад;
- ✓ метод программирования с отходом назад;
- ✓ методы разработки эвристических алгоритмов;
- ✓ рекурсия;
- ✓ моделирование.

Первые три метода в изолированном виде встречаются редко, чаще всего при разработке алгоритмов используется их комбинация.

### 5.1. Метод частных целей

*Метод частных целей* состоит в том, что первоначальная задача сводится к последовательности более простых задач. Поскольку более простые задачи легче обрабатывать по сравнению с первоначальной задачей, то ее решение может быть получено из решений более простых задач. При этом не существует общего набора правил как для выбора более простых задач, так и для определения класса задач, которые можно решить с помощью данного подхода.

Для создания алгоритма методом частных целей необходимо ответить на следующие вопросы:

- Можно ли решить задачу, не учитывая некоторые условия?
- Можно ли решить задачу для некоторых частных случаев?
- Можно ли упростить задачу, наложив дополнительные ограничения?
- Можно ли разработать алгоритм, который дает решение при всех условиях задачи, но при ограничениях на входные данные?
- Существуют ли решения аналогичных задач и нельзя ли эти решения модифицировать для решения поставленной задачи?

В качестве иллюстрации метода частных целей решим задачу преобразования фигуры «Греческий крест» в прямоугольник, предложенную в одном из упражнений в работе [14]. Заданный крест состоит из пяти одинаковых квадратов. Двумя прямыми линиями его следует разделить на фигуры, из которых можно составить прямоугольник, одна сторона которого вдвое больше другой стороны (рис. 5.1).



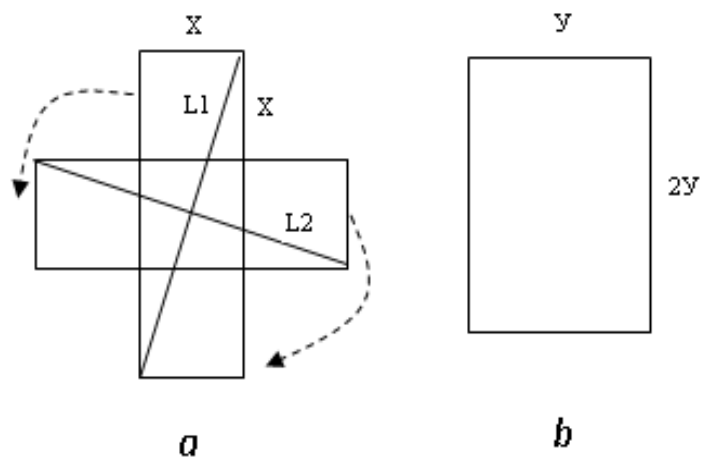


Рис. 5.1. Иллюстрация метода частных целей:  
*a* – греческий крест, *b* – искомый прямоугольник

Прежде всего, отметим, что инварианты в данной задаче – это площади фигур. Данное утверждение приводит к следующей задаче: найти соотношение между стороной квадратов  $X$ , из которых состоит греческий крест, и меньшей стороной прямоугольника  $Y$ . Так как площадь креста равна  $5X^2$ , а площадь прямоугольника –  $2Y^2$ , то меньшая сторона прямоугольника определяется соотношением  $Y = \sqrt{\frac{5}{2}}X$ . Теперь нужно решить следующий вопрос: где на фигуре, изображенной на рис. 5.1.*a*, находятся отрезки длиной  $2Y$ ? Легко видеть, что это наклонные линии  $L1$  и  $L2$ . Далее необходимо показать, что эти линии перпендикулярны и в точке их пересечения делятся пополам (соединяем концы каждой из линий друг с другом, получаем два одинаковых квадрата). Затем, разрезая фигуру вдоль линий  $L1$  и  $L2$  и перемещая ее части так, как показано стрелками, получим прямоугольник, длина одной из сторон которого равна половине отрезка  $L1$ , то есть равна  $Y$ , а другой –  $2Y$  (рис. 5.2.*b*).

## 5.2. Метод подъема

Второй основной метод разработки алгоритмов – это *метод подъема*. Он начинается либо с начального вычисления решения поставленной задачи, либо с принятия предполагаемого начального решения. Начальное решение необходимо улучшить, то есть нужно найти оптимальное решение задачи. По аналогии с алгоритмами нахождения максимумов функций нескольких переменных начинается подъем «вверх» от начального решения по направлению к оптимальному решению. Алгоритм останавливается, если он достигает точку, из которой невозможно двигаться вверх. При этом окончательное решение может быть не оптимальным, что ограничивает применение данного метода.

Метод подъема получил свое название от алгоритмов поиска максимумов функций нескольких переменных. Следуя [14], рассмотрим функцию  $z = f(x, y)$ , график которой представлен на рис.5.2. Задача состоит в применении метода подъема для нахождения максимального значения этой функции.

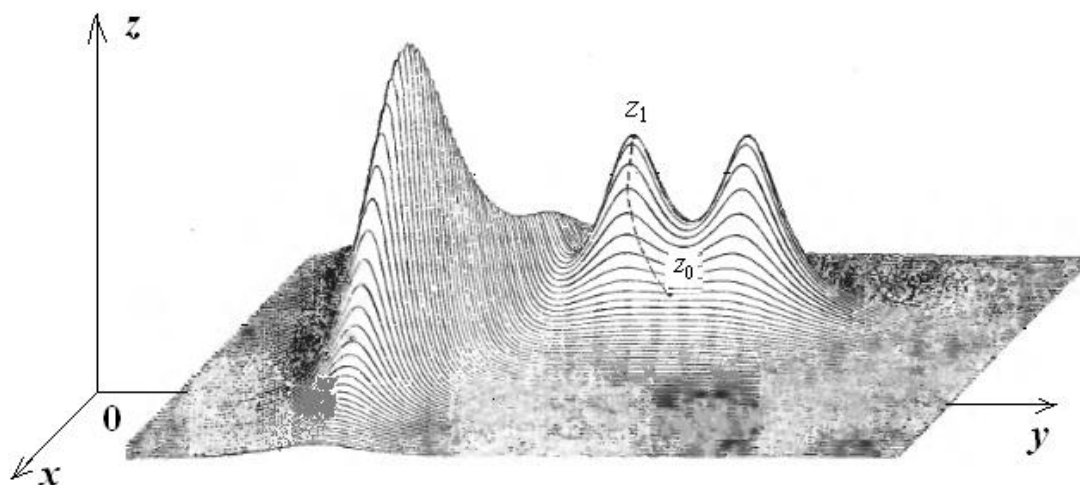


Рис. 5.2. График функции  $z = f(x, y)$ :  $z_0$  – начальное значение функции,  $z_1$  – максимальное значение функции

В качестве начального значения функции возьмем некоторое значение  $z_0 = f(x_0, y_0)$ . Затем выбираем направление поиска максимального значения функции и делаем шаги в этом направлении. У функции может быть несколько максимумов, поэтому какой из них будет найден, зависит от выбора начальной точки. На рис. 5.2 показано, что если в качестве начальной точки выбрана точка  $z_0$ , то найденная максимальная точка – это точка *локального* максимума  $z_1$ . Следовательно, в данном случае метод подъема не дает оптимального решения, поскольку *глобальный* максимум не достигнут.

Подобная ситуация возникает при использовании большинства итерационных алгоритмов. Например, любой алгоритм нахождения корня непрерывной функции начинается с поиска отрезка, на концах которого значения функции имеют разный знак. Затем внутри найденного отрезка ищется точка, анализируется знак функции в этой точке и сравнивается с ее знаком на одном из концов отрезка. В качестве нового отрезка выбирается тот, на концах которого функция имеет различные знаки. Таким образом, получается система вложенных отрезков, которая в пределе имеет общую точку, совпадающую с корнем функции. Однако в случае, когда на начальном отрезке функция имеет несколько корней, то метод даст лишь значение одного из них.

Следует отметить, что методы подъема относятся к «грубым» методам. Они могут быть полезны, когда необходимо быстро получить приближенное решение задачи.

### 5.3. Метод отработывания назад

*Метод отработки назад* хорошо знаком школьникам, решающим трудные задачи. Сначала нужно посмотреть ответ задачи, а затем попытаться от него перейти к условиям задачи. Если обратное восстановление решения, полученного таким способом, возможно, то имеем алгоритм решения этой задачи. Таким образом, метод отработки назад состоит в том, что начинаем с цели либо результата и идем в обратном направлении к начальной постановке задачи. Затем, в случае обратимости действий, двигаемся от постановки задачи к результату.

Для иллюстрации этого метода, следуя [14], рассмотрим следующую задачу. Предположим, что надо проехать 1000 км по пустыне из точки *A* в точку *C* на автомобиле с *одним* топливным баком, объемом 500 литров, причем на 1 км пути расходуется один литр бензина. В начале пути имеется резервуар с топливом неограниченных размеров. В течение всего пути нужно делать временные хранилища топлива, перевозя в них топливо в баке машины. В конце пути топливный бак автомобиля должен быть пустым, и все временные хранилища горючего должны быть пустыми. Необходимо определить количество топлива, изъятого из резервуара.

Решение задачи начинаем с конца. Так как при полной заправке автомобиль может проехать половину пути, то последнее временное хранилище должно располагаться в 500 км от конца пути и иметь 500 литров горючего (рис.5.3). В этом случае в конце пути бак автомобиля будет пустым. Последнее хранилище также будет пустым.

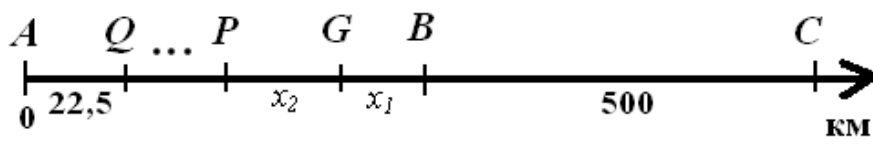


Рис. 5.3. Расположение временных хранилищ топлива для автомобиля

В данной задаче будем также применять метод частных целей, описанный в разделе 5.1. Согласно этому методу, вместо вопроса: «Сколько топлива нужно автомобилю для преодоления заданного расстояния?» рассмотрим более простой вопрос: «Какое расстояние может проехать автомобиль на заданном количестве горючего?». Если ответ на второй вопрос – «не меньше 1000 км», то можно ответить и на первый вопрос.

Усложним задачу: пусть автомобиль имеет *два* бака с топливом, и временное хранилище топлива располагается в точке *G* на расстоянии  $x_1$  км от точки *B* к началу пути. Определим максимальное значение  $x_1$ , такое, что отправляясь из точки *G* и имея 1000 литров горючего, автомобиль перевезет в точку *B* топливо, достаточное для завершения поездки в точке *C*. В данном случае один бак топлива используется для того, чтобы три раза преодолеть расстояние дли-

ной  $x_1$  км, а из второго бака горючее переносится в хранилище, находящееся в точке **B**. Для преодоления расстояния  $x$  км требуется  $3x_1$  литров горючего, а расстояния от **B** до **C** – 500 литров. Поэтому величину  $x_1$  можно найти из уравнения

$$3x_1 + 500 = 1000, \quad (5.1)$$

то есть расстояние  $x_1$  составляет  $500/3 \approx 166,7$  км. Таким образом, имея два бака с топливом, автомобиль проедет расстояние

$$d_2 = 500 + x_1 = 500(1 + 1/3) \text{ км.} \quad (5.2)$$

Если автомобиль имеет три бака с топливом, а еще одно временное хранилище горючего находится в точке **P** (рис. 5.3), то  $x_2 = 500/5 = 100$  км и автомобиль проедет расстояние

$$d_3 = 500 + x_1 + x_2 = 500(1 + 1/3 + 1/5) \text{ км.} \quad (5.3)$$

Применяя метод математической индукции [42] к процессу отработывания назад, получаем, что имея  $n$  баков горючего, автомобиль проедет расстояние

$$d_n = 500 + x_1 + x_2 = 500(1 + 1/3 + 1/5 + \dots + 1/(2n - 1)) \text{ км.} \quad (5.4)$$

Из неравенства

$$d_n \geq 1000 \quad (5.5)$$

определяем значение  $n = 7$ , при котором  $d_7 = 977,5$  км, то есть последнее временное хранилище топлива расположено в точке **Q** на расстоянии  $\approx 22,5$  км от точки **A**.

Рассчитаем, какое количество топлива необходимо иметь в начале пути для создания первого временного хранилища из 7 баков (3500 литров топлива). Для того чтобы 15 раз преодолеть расстояние в 22,5 км, необходимо 337,5 литров горючего. Значит, из исходного резервуара для преодоления пустыни изымается 3837,5 литров топлива. Таким образом, стартуя в начале пустыни и заполнив полностью бак горючим, проезжаем 22,5 км, сливаем 455 литров, оставляя 22,5 литров для обратного пути. Так повторяется 7 раз. В последнее возвращение в первом временном хранилище будет находиться 3185 литров топлива. Для того чтобы в нем находилось 3500 литров, последний раз необходимо заправить 315 литров (недостающий объем до 3500 литров) плюс 22,5 литров (для проезда 22,5 км пути), то есть заправка составляет 337,5 литров топлива. В этом случае автомобиль находится в точке первого временного хранилища на расстоянии 22,5 км от начала пути и во временном хранилище находится (7 баков) 3500 литров горючего. Далее, второе временное хра-

нилище будет объемом 6 баков, третье – объемом 5 баков и так далее. Последнее временное хранилище, объем которого равен одному баку, будет находиться в середине пустыни в точке **B**, автомобиль будет там же. Окончательно имеем, что необходимо 7 временных хранилищ объемом соответственно 7, 6, ..., 1 баков, при этом затрачено топлива – 3837,5 литров.

Таким образом, алгоритм транспортировки горючего можно представить следующим образом. Автомобиль стартует из точки **A** и имеет 3837,5 литров топлива. Этого топлива достаточно, чтобы постепенно перевезти 3500 литров в точку **Q**, в которой, в конце концов, будет запас горючего на 7 полных заправок и автомобиль с пустым баком. Благодаря методу отработывания назад, последующие перевозки топлива во временные хранилища и продвижение автомобиля по заданному маршруту до точки **B** позволят перевезти 500 литров горючего в эту точку. Здесь топливо будет залито в бак автомобиля, и он без остановки доедет до точки **C**.

#### 5.4. Метод программирования с отходом назад

При решении задач, которые требуют проверки большого, но конечного числа решений, удобно применять метод разработки алгоритма, известный как *метод программирования с отходом назад*. Этот метод ориентирован на исчерпывающий поиск, позволяющий не проводить исследование всех возможностей.

В качестве примера рассмотрим получение перестановок из трех элементов: 0, 1, 2. Следует отметить, что алгоритм можно распространить на любое число элементов, а ограничение до трех элементов продиктовано лишь наглядностью графического изображения. Графически поставленную задачу можно отобразить в виде дерева (рис. 5.4) со степенью вершин (кроме корня) либо 4, либо 1 (тупиковая вершина). Под степенью вершины понимается число вершин, соединенных с ней ребрами [14].

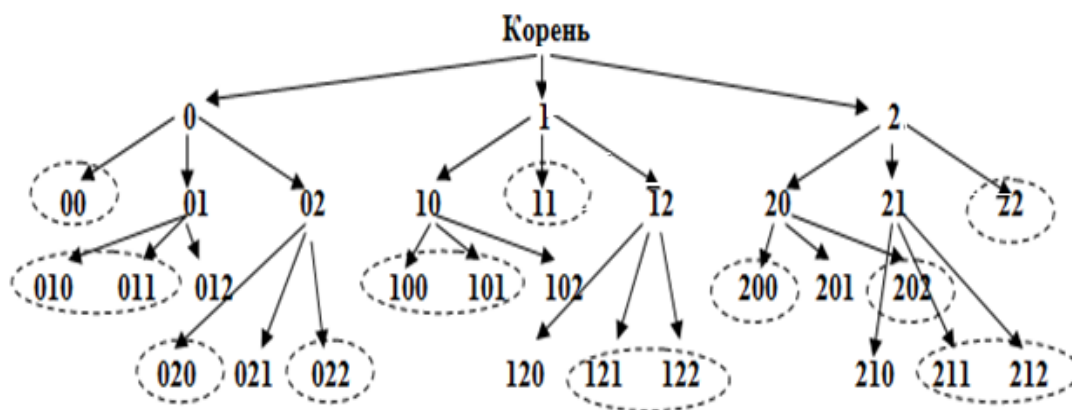


Рис. 5.4. Корневое дерево для получения перестановок

Корень дерева, изображенного на рис. 5.4, имеет степень 3. При построении перестановок из  $N$  элементов вершины будут иметь степень  $N+1$  или 1, а корень будет иметь степень  $N$ . Если цифры в вершине повторяются, то такие вершины должны быть заблокированы. На рис. 5.4 эти вершины обозначены пунктиром. В дальнейшем построении они не участвуют. Номера ребер на каждом уровне и для каждой вершины будем нумеровать как 1; 2; 3.

Для получения перестановок двигаемся от корня по ребру 1 до самого нижнего уровня. Если вершина заблокирована, то переходим на этом уровне к вершине со следующим номером. В данном случае от корня переходим к вершине 0, затем, так как вершина 00 заблокирована, переходим к вершине 01. Далее переходим на следующий уровень и, поскольку вершины 010 и 011 заблокированы, переходим на вершину 012, принадлежащую самому низкому уровню. Делаем отход на предыдущий уровень и просматриваем, нет ли в этой вершине ребер с более высоким номером к незаблокированным вершинам нижнего уровня. Если их нет, то поднимаемся выше, и процедура повторяется вновь, включая и корень дерева. Для рассматриваемого случая после получения первой перестановки возвращаемся к вершине 01 и, так как у нее нет больше разрешенных перемещений вниз, поднимаемся к вершине 0. Из этой вершины есть еще одно допустимое передвижение к вершине 02 и далее к вершине 021. После этого восхождение вверх будет продолжаться до самого корня. Следующий переход будет к вершине 1 и так далее. Полученные перестановки будут: 012; 021; 102; 120; 201; 210. Этот порядок носит название *лексикографического порядка перестановок*. Алгоритм останавливается, когда возвращение к корню не дает возможности двигаться дальше (все незаблокированные ребра просмотрены).

## 5.5. Методы разработки эвристических алгоритмов

Термин «*эвристика*» происходит от греческого «*heuresko*», означающего открываю, отыскиваю. В настоящее время под эвристикой понимают несколько значений этого термина [7]:

- наука о закономерностях организации процессов творческого мышления;
- процесс обучения, основанный на принципах и правилах эвристики как науки – эвристическая педагогика;
- решения задач, которые противопоставляются формальным методам решения при помощи математических алгоритмов и связанные с резким уменьшением перебора вариантов путей решения;
- один из способов создания компьютерных программ [6].

Рассмотрим эвристическое программирование, которое выражается в программах на ЭВМ, использующих *эвристические алгоритмы*, то есть алго-

ритмы, в которых достижение конечного результата программы действий однозначно не предопределено и не обозначена вся последовательность действий.

Эвристические алгоритмы обычно применяются для решения задач NP-класса, то есть задач высокой вычислительной сложности. К числу таких задач относятся, например, частотное планирование для систем мобильной связи, синтез расписаний производственных процессов, маршрутизация транспортных средств, раскрой материалов и др. Представляя все этапы решения некоторой задачи в виде пути на графе потенциально возможных решений, получим большое количество вариантов решений. Вместо полного перебора вариантов, который занимает существенное время, можно применить значительно более быстрый, однако недостаточно теоретически обоснованный эвристический алгоритм. Эвристическое решение задачи можно сформулировать как решение, которое связано с резким уменьшением перебора вариантов путей её решения [7]. Эвристический алгоритм, в отличие от корректного алгоритма решения задачи, дающего оптимальное решение, имеет следующие особенности:

- ✓ он не гарантирует нахождение оптимального решения;
- ✓ в случаях, когда решение заведомо существует, он не гарантирует нахождение решения;
- ✓ в некоторых случаях он может дать неверное решение.

Следует отметить, что нередко очень хорошие и быстрые алгоритмы прекрасно работают на тестовых задачах, однако не доказана их правильность в общем случае. Тогда алгоритм должен считаться эвристическим до тех пор, пока его правильность не будет доказана в общем случае. Эвристические алгоритмы решения задач, хотя не полностью математически обоснованы, тем не менее, часто имея быструю и простую реализацию, дают приемлемое решение в большинстве практически значимых случаев.

Для решения каждой конкретной задачи допустимость использования эвристик зависит от соотношения затрат на решение задачи точным и эвристическим методами, цены ошибки и статистических параметров эвристики. Важной является и критическая оценка результата решения человеком, то есть когда на выходе имеется так называемый «фильтр здравого смысла».

Эвристические алгоритмы широко применяются в таких областях искусственного интеллекта как распознавание образов. Различные эвристические подходы используются также в антивирусных программах, компьютерных играх, программах, играющих в шахматы, и др.

Рассмотрим в качестве алгоритма эвристического поиска алгоритм наискорейшего спуска по дереву решений на примере задачи о коммивояжёре. Следуя [80], предположим, что имеется пять городов **A**, **B**, **C**, **D**, **E**, в которых хотели бы побывать туристы (рис. 5.5). Задача заключается в следующем: начиная с города **A**, найти минимальный путь, который проходит через все остальные города только один раз и обратно приводит в город **A**. Идея метода

состоит в том, что из каждого города туристы едут в ближайший из городов, где они еще не были.

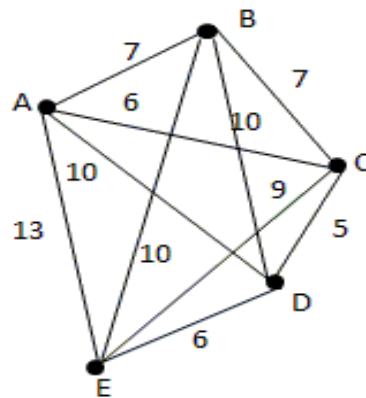


Рис. 5.5. Схема расположения городов

Решение задачи представлено на рис. 5.6. Данный алгоритм поиска решения получил название *алгоритма наискорейшего спуска*.

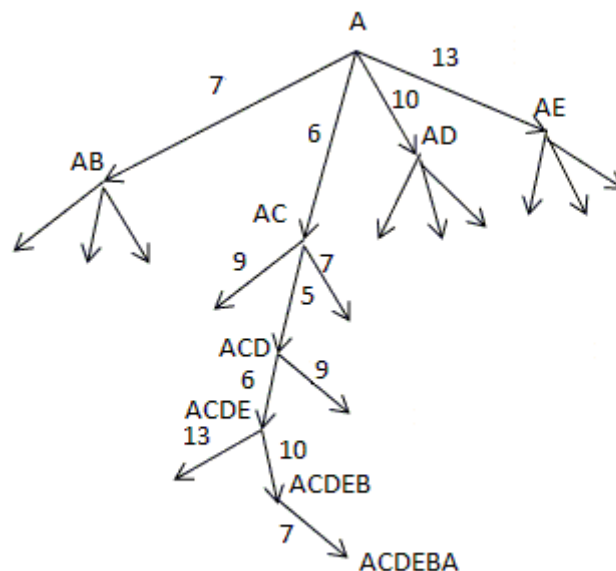


Рис. 5.6. Алгоритм наискорейшего спуска

## 5.6. Рекурсия

В программировании и математике *рекурсия* – это метод определения функции, процедуры, решения задачи с помощью той же функции, процедуры и т.д. В качестве примера рассмотрим функцию вычисления факториала, которая определяется рекурсивно следующим образом:

$$0! = 1, \tag{5.6}$$

$$n! = n \cdot (n - 1)!, \text{ если } n > 0. \tag{5.7}$$



или в виде фрагмента программы на языке программирования C++:

$$\text{fact}(0) = 1; \quad (5.8)$$

$$\text{if } (n > 0) \\ \text{fact}(n) = n * \text{fact}(n - 1); \quad (5.9)$$

Область определения функции  $\text{fact}()$  – это множество неотрицательных целых чисел. Уравнение (5.9) является *рекуррентным соотношением*. Рекуррентные соотношения выражают значение функции через значение той же функции, вычисленные для меньших аргументов. Уравнение (5.8) определяет *начальное значение* функции  $\text{fact}()$ . Начальное значение необходимо для каждой рекурсивной функции, в противном случае ее нельзя вычислить в явном виде.

Для вычисления функции  $\text{fact}(n)$  нужно выполнить  $n$  рекурсивных обращений и вычислить  $\text{fact}(n - 1)$ ,  $\text{fact}(n - 2)$  и т.д. Последнее рекурсивное обращение выполняется для  $\text{fact}(0) = 1$ . Количество обращений  $n$ , необходимое для вычисления функции  $\text{fact}(n)$ , называют *глубиной рекурсии* [14]. Глубина рекурсии в программах на ЭВМ – это мера вычислительной сложности рекурсивно определенной функции.

## 5.7. Моделирование

Задачи, описывающие очень сложные системы, например, космические полеты, сети связи, принятие административных решений, управление производством и др., значительно труднее моделируются, анализируются и решаются [13, 66]. Это связано с тем, что на поведение системы влияет множество переменных, существует трудность в определении внутренних связей между переменными и обычно неизвестны распределения вероятностей случайных переменных системы. Появление ЭВМ позволило изучать такие системы с помощью имитационного моделирования [18 – 20, 30 – 33, 55]. Имитационную модель необходимо создавать. Для этого нужно специальное программное обеспечение – *система моделирования*. Специфика такой системы определяется технологией работы, набором языковых средств, сервисных программ и приемов моделирования. Имитационное моделирование контролируемого процесса или управляемого объекта – это высокоуровневая информационная технология. Она обеспечивает работы по созданию или модификации имитационной модели, а также эксплуатацию имитационной модели и интерпретацию результатов.

Сложность реальных процессов приводит к тому, что для имитационного моделирования этих процессов применяются достаточно технологичные инструментальные средства имитационного моделирования, которые обладают собственными языковыми средствами. В середине 1970-х гг. XX века появились первые инструментальные средства имитационного моделирования, например, система GPSS. Она позволяла создавать модели контролируемых

процессов и объектов в основном технического или технологического назначения [20]. В настоящее время для моделирования процессов, происходящих в реальных системах, применяются следующие системы моделирования: EXTEND, ITHINK, VENSIM, POWERSIM, PILGRIM PROCESS CHAPTER, SIMULA и др. [30].

Имитация – это своеобразная попытка дублировать особенности, внешний вид и характеристики реальной системы. Применение имитационного моделирования оправдано, если вопросы, ответ на которые должна дать модель, относятся не к выяснению фундаментальных законов и причин, определяющих динамику реальной системы, а к практическому анализу поведения системы [18 – 20, 62].

При правильном представлении исходной информации имитационные модели характеризуются большей близостью к реальной системе, чем аналитические и численные модели. С помощью имитационного моделирования и соответствующих современных программных средств можно создавать даже те модели, которые невозможно сформулировать традиционными методами. Кроме того, имитационное моделирование позволяет создавать модели тех систем, с которыми нельзя провести эксперимент. Это позволяет упростить и усовершенствовать управление ими.

Имитационные эксперименты можно проводить в разных масштабах времени: время может быть замедленно, либо ускорено. В имитационном эксперименте время также можно остановить, повернуть вспять и заново провести эксперимент, чтобы более детально изучить поведение системы. Развитые средства анализа чувствительности обеспечивают автоматическое многократное исполнение моделей с различными входными данными.

Но имитационное моделирование обладает и рядом недостатков: оно является итеративной, экспериментальной техникой решения проблем; требует значительных затрат компьютерного времени; чаще всего находит решение, только близкое к оптимальному. Однако очень часто этот метод является единственным способом создания модели сложной системы.

Рассмотрим пример *имитационного алгоритма*. Наиболее распространенными среди имитационных алгоритмов являются алгоритмы, моделирующие *очереди*. В очереди (линии обслуживания) основные объекты – это клиенты, заполняющие очередь через случайные промежутки времени, и обслуживающее устройство, которое в течение случайного интервала времени обслуживает каждого клиента. Очередь функционирует по принципу FIFO (FIRST IN – FIRST OUT): первый пришел – первый ушел. Следовательно, клиент, стоящий в очереди первым, будет первым обслужен и покинет очередь. На рис. 5.7 приведена система, иллюстрирующая одну очередь и одно обслуживающее устройство.

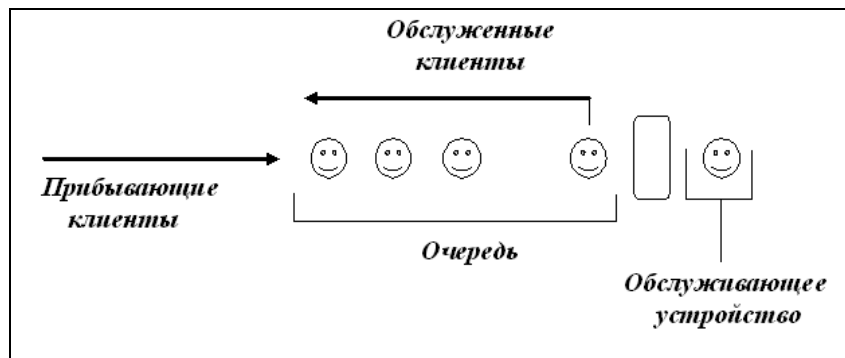


Рис. 5.7. Очередь и обслуживающее устройство

Для функционирования такой очереди, следуя [14], необходимо сделать следующие предположения:

1. Допустим, имеется случайная переменная  $x$ , определяющая время прибытия следующего клиента. Если клиент  $K$  прибывает в момент времени  $t$ , то клиент  $K+1$  прибывает в момент  $t+T$ , где  $T$  является случайной переменной. Значение  $T$  лежит между 1 и  $MAX\_A$  – некоторым постоянным целым числом с заданным распределением вероятностей.
2. Допустим, имеется случайная переменная  $S$ , определяющая длительность времени обслуживания клиента  $K$ . Значение  $S$  лежит между 1 и  $MAX\_C$  – фиксированным целым числом с заданным распределением вероятностей. Предположим, что обслуживание клиентов не прерывается до его полного завершения и не допускается обслуживание с приоритетом.
3. Существует очередь клиентов, которая обслуживается по принципу FIFO. При этом как только клиент встает в очередь, он остается в ней до тех пор, пока не будет обслужен. После обслуживания он немедленно покидает очередь.
4. Все действия в системе обслуживания описываются дискретными независимыми событиями. Здесь независимыми событиями являются: а) прибытие клиента; б) завершение обслуживания клиента.
5. В начальном состоянии система обслуживания пуста, т.е. в момент времени  $t = 0$  в очереди никого нет и обслуживающее устройство не занято.
6. Поток действий управляется часами с дискретным, постоянным приращением времени. Часы отсчитывают постоянные единицы времени: секунды, либо наносекунды, либо месяцы, либо годы и т.д.
7. Для определения интервалов времени между прибытиями клиентов, а также продолжительностей их обслуживания, используется датчик случайных чисел.

При разработке алгоритма моделирования конкретной системы «одна очередь / одно обслуживающее устройство» необходимы данные, которые нужно собрать в процессе имитации [14]:

- число событий в процессе имитации;
- средняя длина очереди;
- среднее время ожидания в очереди;
- максимальная длина очереди;
- эффективность использования обслуживающего устройства – процент времени, в течение которого оно было занято.

Следует отметить, что фактически во всех вычислительных устройствах имеется системный датчик случайных чисел для равномерного распределения случайной переменной на интервале  $[0, 1)$ . Данное распределение можно использовать для моделирования других распределений случайной переменной. Кроме того, большинство систем имитационного моделирования имеют заранее установленный неизменный порядок очереди.

В системах с дискретными событиями предполагается, что состояние системы изменяется только тогда, когда событие происходит. После совершения события необходимо продвинуть имитируемое время ко времени следующего события. Данный подход к моделированию называется *методом планирования событий* [14]. Принципиальная блок-схема алгоритма моделирования системы «одна очередь / одно обслуживающее устройство», которая иллюстрирует метод планирования событий, приведена на рис. 5.8.

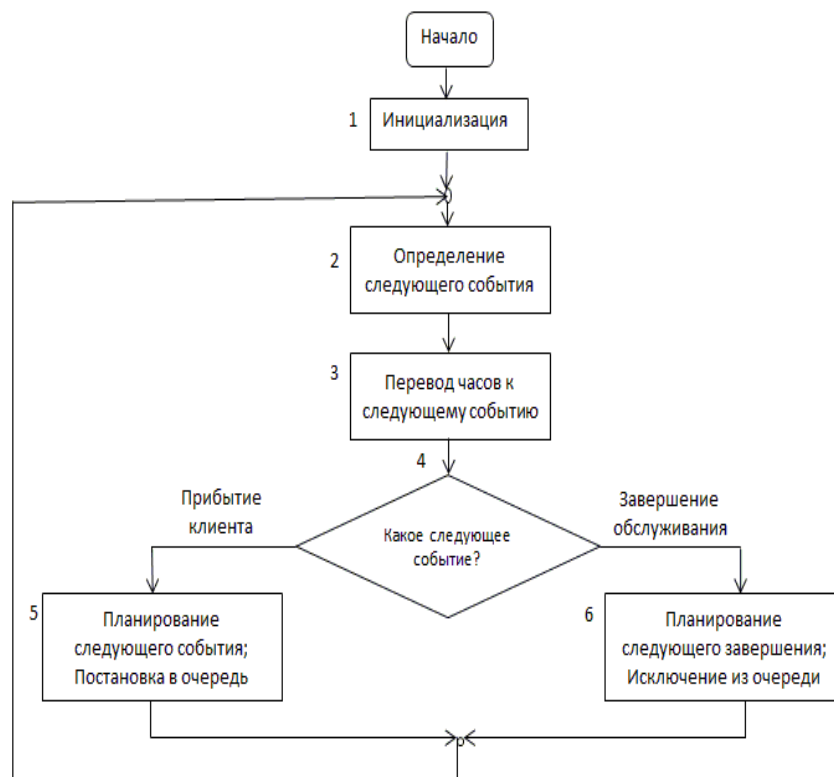


Рис. 5.8. Принципиальная блок-схема моделирования системы «одна очередь / одно обслуживающее устройство» методом планирования событий [14]

Таким образом, методы создания алгоритмов, рассмотренные в данной главе, позволяют разрабатывать эффективные алгоритмы для решения поставленных задач.

### *Упражнения*

1. Разработать алгоритм и написать программу игры, которая заключается в следующем. Имеются три кучи с произвольно заданным количеством камней. Двое игроков по очереди делают ходы. Ход заключается в том, что игрок берет произвольное число камней из какой-либо кучи. Выигрывает тот, кто берет последний камень.
2. Чтобы зашифровать текст, необходимо заменить каждую букву текста буквой, полученной сдвигом вправо на заданное число. Если при сдвиге выходим за пределы алфавита, то отсчет продолжается с первой буквы алфавита. Разработать алгоритм и написать программу зашифровки и расшифровки заданного текста.
3. Разработать алгоритм и написать программу выбора элемента из заданного множества элементов, связанных в кольцо. Выбор элемента осуществляется следующим образом. задается число и указатель на некоторый элемент. Указатель сдвигается по кольцу на это заданное число. Элемент, на который после сдвига переместится указатель, удаляется из кольца, а указатель перемещается на следующий элемент. Затем шаги алгоритма повторяются вновь. Выбранным считается последний оставшийся элемент.
4. Построить алгоритм и написать программу определения максимального числа частей круга, получающихся разбиением его четырьмя прямыми линиями.
5. Три человека играют в игру, где один выигрывает, а два проигрывают. Каждый из проигравших игроков отдает выигравшему игроку столько шаров, сколько было у выигравшего игрока на начало игры. Они сыграли в игру три раза, причем каждый из них выиграл по одному разу. У первого оказалось 4 шара, у второго – 20, у третьего – 6.
6. Разработать алгоритм и написать программу для определения числа шаров у каждого игрока на начало игры.
6. Необходимо упаковать коробки с одинаковым основанием, но разной высотой в коробки (тару), с таким же основанием, но с заданной высотой. Предполагается, что максимальная высота коробок, которые мы упаковываем, имеет высоту, меньшую или равную высоте тары. Разработать алгоритм и написать программу, определяющую минимальное число коробок тары для заданного количества упаковываемых коробок.

7. Задана система множеств  $S_i, i=1, \dots, n$ , для которых выполнено  $\bigcup_{i=1}^n S_i = S$ . Разработать алгоритм и написать программу нахождения минимального числа множеств  $S_i$ , объединение которых совпадает с  $S$ .
8. Построить алгоритм и написать программу разбиения числа 45 на четыре части так, что если к первой части прибавить 2, от второй отнять 2, третью умножить на 2, а четвертую разделить на 2, то все результаты будут равными.
9. Создать алгоритм и написать программу определения трехзначных чисел, обладающих следующим свойством. Если от числа отнять 7, то оно разделится на 7, если от числа отнять 8, то оно разделится на 8, если из числа вычесть 9, то оно разделится на 9.
10. Разработать алгоритм и написать программу, определяющую существование числа, меньшего 1000, которое при делении на 3 дает в остатке 1, при делении на 4 – в остатке 2, при делении на 5 – в остатке 3 и при делении на 6 дает в остатке 4.
11. Разработать алгоритм и написать программу расстановки всех десяти цифр 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 в таком порядке, чтобы получившееся число делилось на все числа от 2 до 18.
12. Прямоугольники задаются координатами левого верхнего и правого нижнего углов. Разработать алгоритм и написать программу, определяющую пересечение двух прямоугольников. Если прямоугольники пересекаются, то определить координаты полученного прямоугольника или выдать сообщение об отсутствии их пересечения.
13. Компьютер имеет три процессора и очередь из заданий, у которых указаны номер, фамилия автора и длительность выполнения. Разработать алгоритм и написать программу распределения заданий между процессорами таким образом, чтобы их загруженность была по возможности равномерная.
14. Задан список из фамилий, имен и отчеств. Создать алгоритм и написать программу преобразования этого списка, который будет состоять из фамилий и инициалов, причем у фамилии первая буква прописная, а остальные строчные. Например, запись ИВАНОВ ИВАН ИВАНОВИЧ преобразовывается в Иванов И.И. Полученный список упорядочить в алфавитном порядке.
15. Даны три стержня, на один из которых нанизаны восемь колец, причем кольца отличаются размером и меньшее кольцо лежит на большем. Разработать алгоритм и написать программу переноса пирамиды из восьми колец на другой стержень. За один раз разрешается переносить только одно кольцо, причём нельзя класть большее кольцо на меньшее кольцо.
16. По словам рыбака, он поймал рыбу, у которой голова была длиной 30 см, хвост длиной с голову и половину туши, а туша – с половину

длины рыбы с головы до хвоста. Разработать алгоритм и написать программу определения длины пойманной рыбы.

17. К бассейну подходят четыре трубы, по которым через краны можно контролировать скорость заполнения бассейна. Открыв первый кран, можно заполнить бассейн за 2 часа, второй – за 3 часа, третий – за 4 часа и четвёртый – за 6 часов. Создать алгоритм и написать программу определения времени, которое требуется для наполнения бассейна, если открыть все четыре крана одновременно.
18. Автомобиль должен пересечь пустыню. Однако полного бензобака хватает только на половину пути. В распоряжении базы имеется несколько таких автомобилей, и бензин можно перекачивать из бака одного автомобиля в бак другого. Никакими канистрами и тросами пользоваться нельзя. Разработать алгоритм пересечения пустыни автомобилем, чтобы в конце пути его бак был пуст, и в пустыне не осталось ни одного автомобиля.

## ГЛАВА 6. ОСНОВНЫЕ АЛГОРИТМЫ И ИХ РЕАЛИЗАЦИЯ

### 6.1. Алгоритмы анализа текста

Для анализа текста в лингвистических исследованиях наиболее часто используются следующие алгоритмы [56]:

- алгоритм анализа частоты встречаемости различных символов и букв в тексте;
- алгоритм выделения слов в строке текста;
- алгоритм определения слов, подходящих под шаблон.

*Алгоритм анализа частоты встречаемости различных символов и букв в тексте* сводится к анализу каждого символа строки текста. Здесь необходимо решить вопрос о представлении данных: использовать массив, размером с алфавит, или составлять связный список. В связном списке выделяется место только для тех символов, которые имеются в анализируемом тексте.

*Алгоритм выделения слов в строке текста* широко применяется при анализе текстов. Под словом будем понимать последовательность символов, которая не содержит пробелов и заключена между пробелами или концами строки. Существует несколько алгоритмов выделения слов в строке. Например, в одном из них строка рассматривается как символьный массив. Запоминается индекс начала слова ( $b$ ) и индекс, следующий за последним символом слова ( $e$ ). Алгоритм сводится к следующим шагам: переменной  $e$  присваивается значение 0 и ее значение увеличивается на 1 до тех пор, пока текущий символ в строке текста является пробелом. Затем определяется переменная  $b = e$ . Переменная  $e$  вновь увеличивается на 1 до тех пор, пока текущий символ в строке отличен от пробела. При выходе из этого цикла переменная  $b$  будет совпадать с индексом массива, где начинается слово, а  $e$  – с индексом первого пробела за последним символом слова. Существование слова между символами  $b$  и  $e$  гарантируется, если  $e - b > 0$ . Эти шаги следует повторять до тех пор, пока значение  $e$  меньше конца строки. Блок-схема алгоритма приведена на рис. 6.1. Данный алгоритм позволяет решать задачи о количестве слов в тексте, определять пустые строки, позволяет провести анализ слов в тексте.



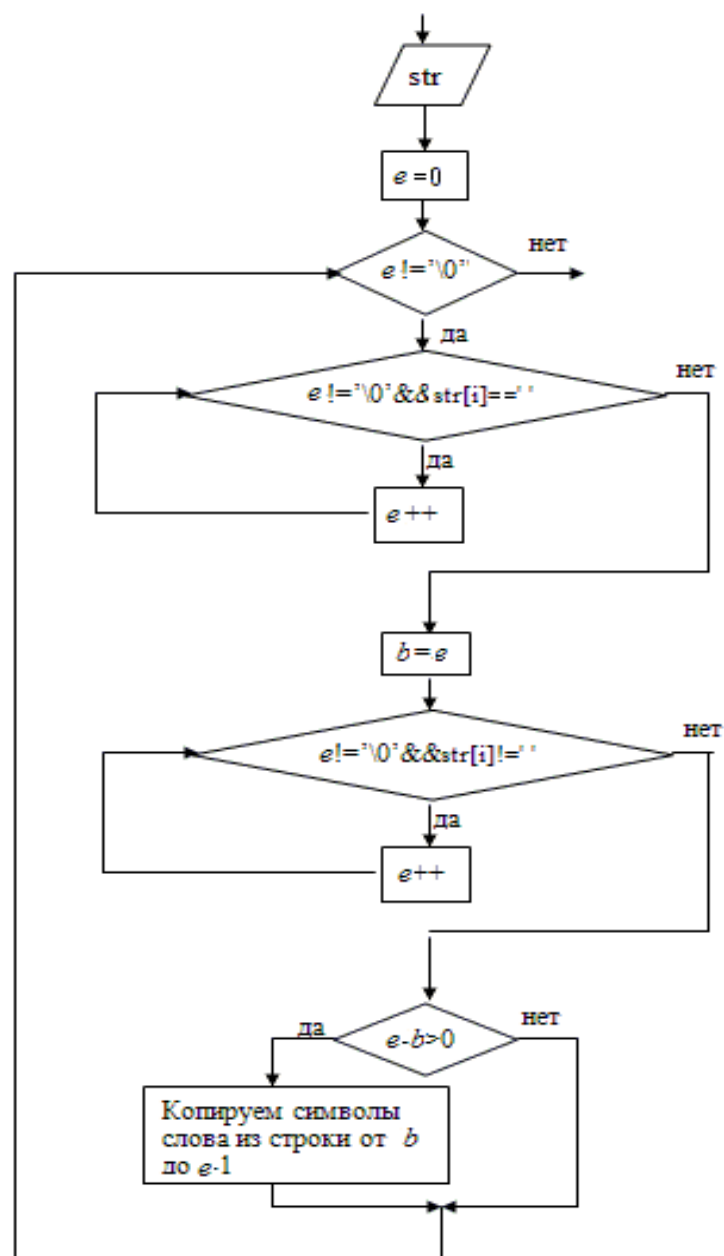


Рис. 6.1. Блок-схема выделения слов в строке текста **str**

**Листинг 6.1.** Определить количество слов в тексте, записанном в файле «input.txt» (рис. 6.2).

В файле с именем «input.txt» содержится информация о количестве чисел и сами числа.  
 Необходимо сформировать массив, не содержащий повторяющихся значений.

Рис. 6.2. Информация, находящаяся в файле «input.txt»

```

// L6_1.cpp
#include "stdafx.h"
#include <fstream>
#include <iostream>
using namespace std;
int sum_word(char*st);           //Прототип функции
int _tmain()
{
    int sum=0;
    char s[81];
    fstream f("input.txt"),f1;
    f.getline(s,80);             //Читаем строку из файла
    while(!f.eof())             //Проверка достижения конца файла
    {
        sum+=sum_word(s);       //Добавляем в сумму число
                                 //слов в текущей строке.
        f.getline(s,80);        //Чистим следующую строку
    }
    f.close();
    f1.open("input.txt",ios::app); //Открываем файл для пополнения
    f1<<"\n\nЧисло слов в тексте = "<<sum<<"\n";
    f1.close();
    return 0;
}
// Функция, определяющая число слов в строке:
int sum_word(char*st)
{
    int s=0,b,e=0;
    while(st[e]!='\0')
    {
        //Пропускаем ведущие пробелы:
        while(st[e]!='\0'&&st[e]==' ')e++;
        b=e;
        //Пропускаем символы слова:
        while(st[e]!='\0'&&st[e]!=' ')e++;
        if(e-b>0)                 //Слово выделено
            s++;
    }
    return s;
}

```

Результат работы программы листинга 6.1 представлен на рис. 6.3.

**В файле с именем «input.txt» содержится информация о количестве чисел и сами числа.**

**Необходимо сформировать массив, не содержащий повторяющихся значений.**

**Число слов в тексте = 20**

Рис. 6.3. Результат работы программы листинга 6.1

При анализе текста *определение слов, подходящих под шаблон*, является одним из наиболее применяемых алгоритмов. Рассмотрим шаблон, содержащий символы ? и \*. Наличие символа ? в шаблоне означает, что его можно заменить любым символом, не являющимся пробелом. Символ \* можно заменить любой последовательностью символов, не являющихся пробелами, при этом последовательность символов может быть пустой. Если в шаблоне присутствует символ \*, то должен быть последним. Задача заключается в определении числа слов текста, отвечающих заданному шаблону.

**Листинг 6.2.** Определить количество слов текста, отвечающих заданному шаблону.

```
// L6_2.cpp
#include "stdafx.h"
#include <string.h>
#include <fstream>
#include <iostream>
using namespace std;
int n_shablon(char st[], char s[]); // Определение числа слов в строке,
// подходящих под шаблон.

int _tmain()
{
    int sum=0;
    char s[81], sh[30];
    fstream f("input.txt"),f1;
    f.getline(sh,80); //Читаем шаблон из файла
    f.getline(s,80); //Чистим следующую строку
    while(!f.eof()) //Проверка достижения конца файла
    {
        sum+=n_shablon(s,sh); //Добавляем в сумму число
        //слов в текущей строке.
        f.getline(s,80); //Чистим следующую строку
    }
}
```

```

    f.close();
    //Открываем файл для пополнения:
    f1.open("input.txt", ios::app);
    f1<<"\n\nЧисло слов в тексте, подходящих под шаблон =
"<<sum<<"\n';
    f1.close();
    return 0;
}
int n_shablon(char st[], char sh[])
{
    int s=0,b,e=0,dl,i,f;
    if(strstr(sh,"*"))
    {
        dl=0;
        for(b=0; sh[b]!='*'; b++)
            dl++;
        while(st[e]!='\0')
        {
            //Пропускаем ведущие пробелы:
            while(st[e]!='\0'&&st[e]==' ') e++;
            b=e;
            //Пропускаем символы слова:
            while(st[e]!='\0'&&st[e]!=' ') e++;
            //Слово выделено
            if(e-b>0&&e-b>dl)
            {
                f=1;
                for(i=0; i<dl; i++)
                    if(st[b+i]!=sh[i])
                    {
                        f=0;
                        break;
                    }
                if(f)
                    s++;
            }
        }
        return s;
    }
    if(strstr(sh,"?"))
    {
        dl=strlen(sh);
        while(st[e]!='\0')
        {

```

```

//Пропускаем ведущие пробелы:
while(st[e]!='\0'&&st[e]==' ') e++;
b=e;
//Пропускаем символы слова
while(st[e]!='\0'&&st[e]!=' ') e++;
//Слово выделено
if(e-b>0&&e-b==dl)
{
    f=1;
    for(i=0; i<dl; i++)
        if(st[b+i]!=sh[i]&&sh[i]!='?')
            {
                f=0;
                break;
            }
    if(f)
        s++;
}
}
return s;
}
dl=strlen(sh);
while(st[e]!='\0')
{
    //Пропускаем ведущие пробелы:
    while(st[e]!='\0'&&st[e]==' ') e++;
    b=e;
    //Пропускаем символы слова:
    while(st[e]!='\0'&&st[e]!=' ') e++;
    //Слово выделено
    if(e-b>0&&e-b==dl)
    {
        f=1;
        for(i=0; i<dl; i++)
            if(st[b+i]!=sh[i])
                {
                    f=0;
                    break;
                }
        if(f)
            s++;
    }
}
return s;

```

}

Результаты работы программы листинга 6.2 при различных шаблонах приведены на рис. 6.4 – рис. 6.6.

**паро\***

**По реке плыл красивый пароход.  
Прежде чем мы оказались на реке,  
была долгая поездка: нас вёз паровоз,  
затем была паромная переправа.**

**Число слов в тексте, подходящих под  
шаблон, равно 3.**

Рис. 6.4. Содержимое файла «input.txt» с шаблоном **паро\***

**паро?о?**

**По реке плыл красивый пароход.  
Прежде чем мы оказались на реке,  
была долгая поездка: нас вез паровоз,  
затем была паромная переправа.**

**Число слов в тексте, подходящих под  
шаблон, равно 2.**

Рис. 6.5. Содержимое файла «input.txt» с шаблоном **паро?о?**

**Паромная**

**По реке плыл красивый пароход.  
Прежде чем мы оказались на реке,  
была долгая поездка: нас вёз паровоз,  
затем была паромная переправа.**

**Число слов в тексте, подходящих под  
шаблон, равно 1.**

Рис. 6.6. Содержимое файла «input.txt» с шаблоном **паромная**

## 6.2. Алгоритмы работы с датами

Для работы с датами основными являются следующие алгоритмы:

- проверка правильности введенной даты;
- определение дня недели введенной даты;
- алгоритм сложения даты с целым числом;
- алгоритм вычисления количества дней между двумя датами.

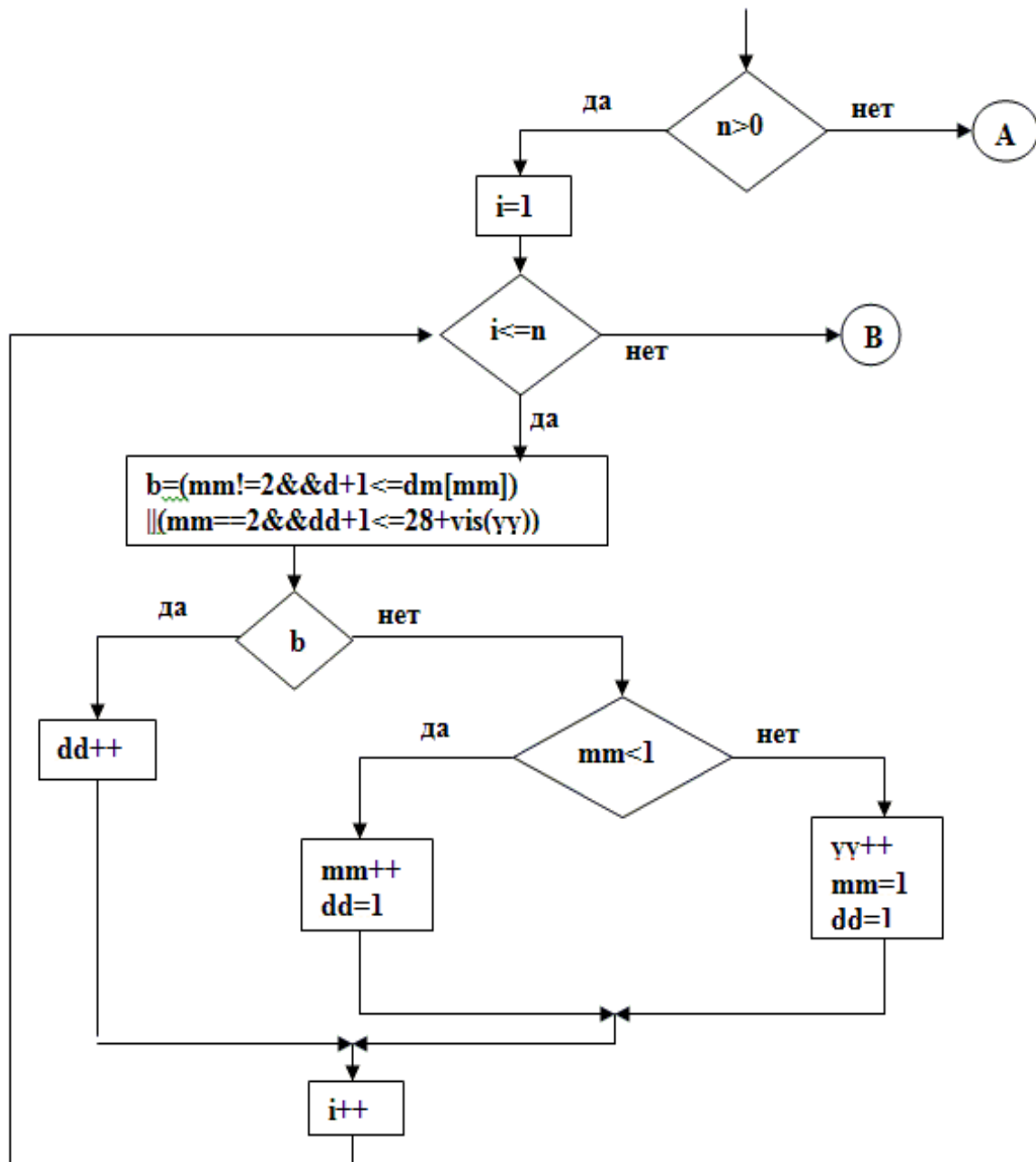
Очень важно определить операции сложения даты и переменной целого типа, сравнения двух дат, а также операцию вычитания между двумя датами, которая дает число дней между ними.

Для работы с датами удобнее всего ввести тип данных – *класс* [54] с именем *date*, и операции с элементами этого типа определить как методы данного класса. Следует определить функцию-член класса для печати даты в каком-либо принятом формате, например, 24.03.15.

*Листинг 6.3.* Описание класса *date*.

```
// L6_3.h
class date
{
    int dd, mm, yy;                //День, месяц, год
public:
    date(int d=1, int m=1, int y=2012) //Конструктор
    {
        dd=d;
        mm=m;
        yy=y;
    }
// Прототипы функций:
    int vis(int);                  //1 – год високосный
                                   //0 – год не високосный
    int date_control();           //Контроль правильности ввода даты
    int& day();                    //Возвращает значение дня,
                                   //месяца,
    int& month();                  //года.
    int& year();
    date operator+(int);          //Сложение даты с числом и
                                   //получение новой даты.
    int operator-(date&d);        //Вычитание двух дат
    int operator>(date&d);        //Сравнение двух дат
    int operator>=(date&d);
    void date_print(void);        //Печать даты
    char* date_day(void);         //Определение дня недели
};
```

Алгоритм сложения даты с целым числом приведен на рис. 6.7, причем значение числа может быть как положительным, так и отрицательным. Если число, с которым складывается дата, будет отрицательным, то значение полученной даты будет меньше текущей. Например,  $d = (12.05.1999)$ . Тогда  $d + 5 = (17.05.1999)$ , а  $d - 5 = (07.05.1999)$ .





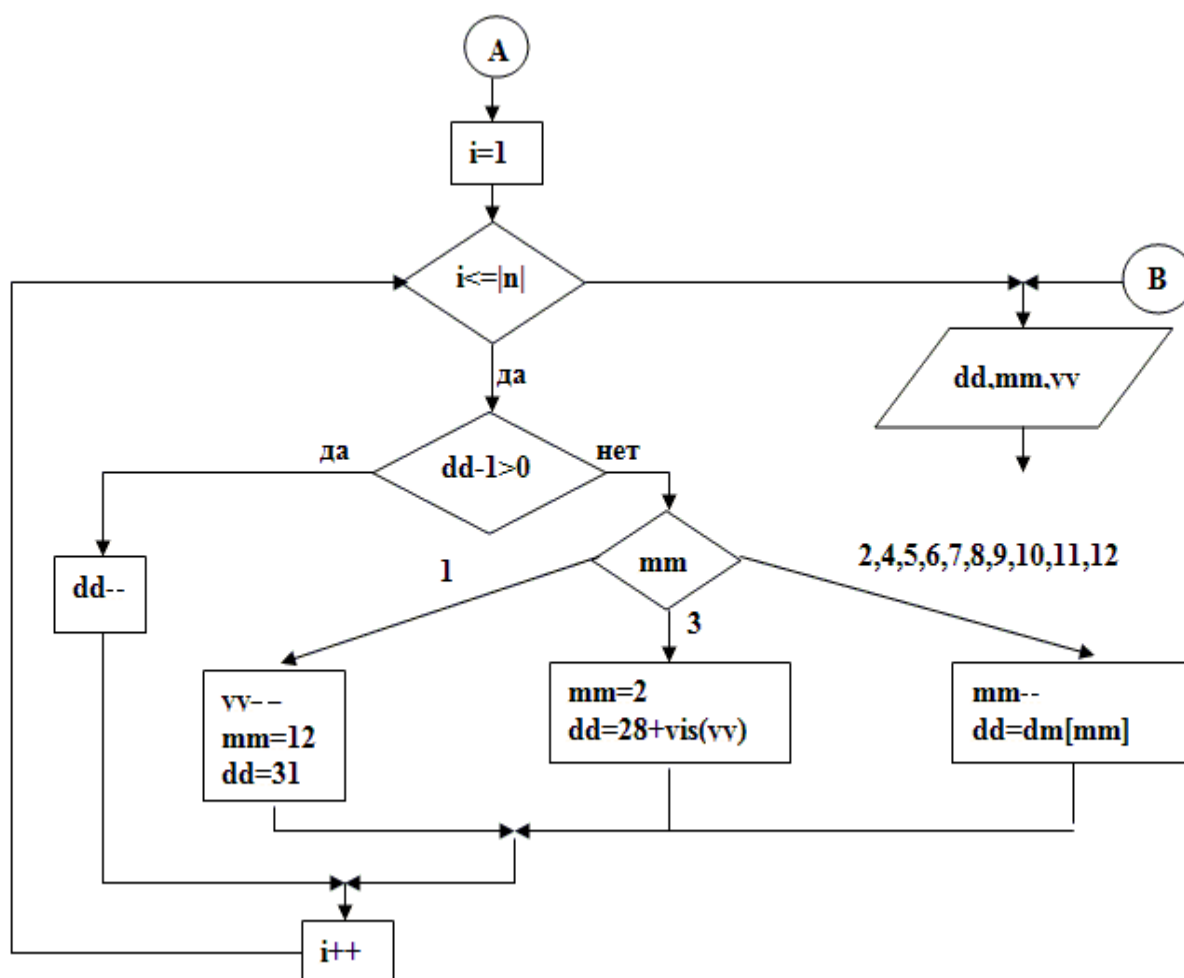


Рис. 6.7. Сложение даты с целым числом: **dd** – день, **mm** – месяц, **yy** – год;  
**dm [i]** – число дней в месяце не високосного года;  
**vis(yy)** – функция, равная 1, если **yy** – високосный год,  
и равная 0, если **yy** – не високосный год

Алгоритм вычитания двух дат состоит в следующем. Выбирается дата с наименьшим значением года и вычисляется, сколько дней от 1 января этого года до первой и второй дат. Разность дат минус один день дает количество дней между двумя датами.

Следуя [21], приведем алгоритм определения дня недели. Обозначим через **dd**, **mm**, **yy** – день, месяц и год даты, лежащей в диапазоне от 1582 до 4902 года. Дни недели имеют следующие номера: 0 – Воскресенье, 1 – Понедельник, 2 – Вторник, 3 – Среда, 4 – Четверг, 5 – Пятница, 6 – Суббота. Будем считать, что нумерация месяцев начинается с марта, то есть Март имеет номер 1, Апрель – номер 2 и т. д. Январь и февраль являются соответственно 11-м и 12-м месяцами предыдущего года. Для определения дня недели вычисляется выражение [21]:

$$k = [2,6 \cdot mt - 0,2] + dd + ym + [ym/4] + [yc/4] - 2 \cdot yc, \quad (6.1)$$

где  $um$  – последние две цифры года,  $us$  – первые две цифры года,  $[ ]$  – операция взятия целой части числа. Далее, нужно определить остаток от деления  $k$  на 7 ( $k \% 7$ ). Если остаток отрицательный, то необходимо сложить его с числом 7. Так как остаток от деления  $k$  на 7 меньше 7 ( $k \% 7 < 7$ ), то полученное число будет положительным. Остаток от деления  $k$  на 7 ( $k \% 7$ ) (в случае если он положительен) или  $(7 + k \% 7)$  (если он отрицателен), совпадают с номером дня недели. Для создания более универсального календаря можно использовать непосредственный отсчет от 1 января 1 года. Это был понедельник.

Приведем листинги методов класса `date` и главной программы.

#### *Листинг 6.4.* Реализация методов класса `date`.

```
// L6_4.cpp
// Описание методов класса date
// include "L6_3.h"
int dm[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};
int date::vis(int y)
{
    if((y%4==0&& y%100!=0)||y%400==0)
        return 1;
    else
        return 0;
}
//Контроль правильности ввода даты:
int date::date_control()
{
    if(yy<=0)return 0;
    if(mm<1||mm>12)return 0;
    if(mm!=0 && (dd<1 || dd>dm[mm])) return 0;
    if(mm==2 && (dd<1 || dd>28+vis(yy))) return 0;
    return 1;
}
int& date::day(void)      //Возвращает значение дня
{
    return dd;
}
int& date::month(void)    //Возвращает значение месяца
{
    return mm;
}
int& date::year(void)     //Возвращает значение года
{
    return yy;
}
```

*//Блок-схема operator+() представлена на рис. 6.7.  
 //Сложение даты с числом и получение новой даты:*  
 date date::operator+(int n)

```

{
    int i;
    if (n>0)
    {
        for(i=1; i<=n; i++)
        {
            if((mm!=2&&dd+1<=dm[mm])||(mm==2&&dd+1<=28+vis(yy)))
                dd++;
            else
            {
                if(mm<12)
                {
                    mm++;
                    dd=1;
                }
                else
                {
                    yy++;
                    mm=1;
                    dd=1;
                }
            }
        }
    }
    else
    {
        for(i=1; i<=abs(n); i++)
        {
            if(dd-1>0)
                dd--;
            else
            {
                switch(mm)
                {
                    case 2: case 4: case 5: case 6: case 7: case 8:
                    case 9: case 10: case 11: case 12:
                    {
                        mm--;
                    }
                }
            }
        }
    }
}

```

```

        dd=dm[mm];
        break;
    }
    case 1:
    {
        yy--;
        mm=12;
        dd=31;
        break;
    }
    case 3:
    {
        mm=2;
        dd=28+vis(yy);
    }
}
}
}
}
return *this;
}
// Вычитание двух дат:
int date::operator-(date&d)
{
    int r=0, ymin,dp1,dp2,i;
    if(yy<=d.yy)
        ymin=yy;
    else
        ymin=d.yy;
    dp1=0;
    for(i=ymin; i<yy; i++)
        dp1+=364+vis(i);
    dp2=0;
    for(i=ymin; i<d.yy; i++)
        dp2+=365+vis(i);
    if(mm==2)
        dp1+=31;
    if(mm>2)
    {
        dp1+=31;
        for(i=2; i<mm; i++)
            dp1+=dm[i];
        dp1+=vis(this->yy);
    }
}

```

```

    dp1+=dd;
    if(d.mm==2)
        dp2+=31;
    if(d.mm>2)
    {
        dp2+=31;
        for(i=2; i<d.mm; i++)
            dp2+=dm[i];
        dp2+=vis(d.yy);
    }
    dp2+=d.dd;
    return abs(dp1-dp2)-1;
}
// Сравнение двух дат:
int date::operator>(date&d)
{
    if(yy>d.yy)return 1;
    if(yy<d.yy)return 0;
    if(mm>d.mm)return 1;
    if(mm<d.mm)return 0;
    if(dd>d.dd)return 1;
    if(dd<d.dd)return 0;
    return 0;
}
// Сравнение двух дат:
int date::operator>=(date&d)
{
    if(yy>d.yy)return 1;
    if(yy<d.yy)return 0;
    if(mm>d.mm)return 1;
    if(mm<d.mm)return 0;
    if(dd>d.dd)return 1;
    if(dd<d.dd)return 0;
    return 1;
}
// Печать даты:
void date::date_print(void)
{
    char s[9];
    int y=yy%100;
    s[0]='0'+dd/10;
    s[1]='0'+dd%10;
    s[2]='.';
    s[3]='0'+mm/10;

```

```

s[4]='0'+mm%10;
s[5]='.';
s[6]='0'+y/10;
s[7]='0'+y%10;
s[8]='\0';
cout<<s;
}
// Определение дня недели:
char* date::date_day(void)
{
    char *s;
    int ym,yc,k,mt,yt;
    if(mm>2)
    {
        mt=mm-2;
        yt=yy;
    }
    else
    {
        if(mm==1)
            mt=11;
        else
            mt=12;
        yt=yy-1;
    }
    ym=yt%100;
    yc=yt/100;
    k=int(2.6*mt-0.2)+dd+ym+ym/4+yc/4-2*yc;
    k%=7;
    if(k<0)
        k=7+k;
    switch (k)
    {
        case 0:
            s=strdup("Воскресенье");
            break;
        case 1:
            s=strdup("Понедельник");
            break;
        case 2:
            s=strdup("Вторник");
            break;
        case 3:
            s=strdup("Среда");

```

```

        break;
    case 4:
        s=strdup("Четверг");
        break;
    case 5:
        s=strdup("Пятница");
        break;
    case 6:
        s=strdup("Суббота");
    }
    return s;
}

```

**Листинг 6.5.** Задаются две даты: dt , dt1 и целое число n. Получить новую дату, определенную выражением dt+n, сравнить ее с dt1 и вычислить количество дней между ними.

```

// L6_5.cpp
#include "stdafx.h"
#include <string.h>
#include <fstream>
#include <iostream>
#include <locale>
#include "L6_4.cpp"
using namespace std;

int _tmain()
{
    setlocale(LC_ALL, "Russian");
    date dt,dt1;
    int d,m,y,n;
    do
    {
        cout<<"Введите день месяц и год ";
        cin>>d>>m>>y;
        dt.day()=d;
        dt.month()=m;
        dt.year()=y;
    }while(!dt.date_control());
    cout<<"Введите n ";
    cin>>n;
    dt=dt+n;
    dt.date_print();
}

```

```

cout<<"\n";
do
{
    cout<<"Введите день месяц и год второй даты ";
    cin>>d>>m>>y;
    dt1.day()=d;
    dt1.month()=m;
    dt1.year()=y;
}while(!dt1.date_control());
n=dt-dt1;
dt.date_print();
cout<<" - ";
dt1.date_print();
cout<<"\n";
cout<<"Число дней между двумя датами "<<n<<"\n";
dt.date_print();
cout<<" "<<dt.date_day()<<"\n";
if(dt>dt1)
{
    dt.date_print();
    cout<<" > ";
    dt1.date_print();
}
else
{
    dt.date_print();
    cout<<" <= ";
    dt1.date_print();
}
cout<<"\n";
return 0;
}

```

На рис. 6.8 представлен результат работы программы (листинги 6.4 и 6.5).

```

Введите день месяц и год 12 2 2000
Введите n 17
29.02.00
Введите день месяц и год второй даты 15 3 2000
29.02.00 - 15.03.00
Число дней между двумя датами 14
29.02.00 Вторник
29.02.00 <= 15.03.00
Для продолжения нажмите любую клавишу . . . _

```

Рис. 6.8. Результат работы программы листингов 6.4 и 6.5



## 6.3. Алгоритмы комбинаторики

### 6.3.1. Алгоритм полной выборки из n элементов

Предположим, что задано  $ns$  элементов. Необходимо перебрать все возможные варианты выборки из этих элементов (выборки по одному элементу, по 2 элемента, ..., выборка по  $n$  элементов). Алгоритм полной выборки из  $n$  элементов может быть реализован с использованием массива из  $ns$  элементов, которые могут принимать лишь два значения – одно соответствует тому, что элемент выбран, а другое – соответствует тому, что элемент не выбран. Это можно осуществить, используя двоичное представление чисел от 1 до числа

$$\sum_{i=1}^n 2^i + 1$$

Получить эти числа можно, если добавлять каждый раз единицу и проводить операции в двоичной системе счисления.

**Листинг 6.6.** Получить полную выборку из  $n$  элементов. Здесь  $ns$  – число элементов, из которых происходит выборка.  $v$  – одномерный вектор из  $ns$  элементов, которые принимают два значения (0 – элемент не выбран, 1 – элемент выбран).

```
// L6_6.cpp
#include "stdafx.h"
#include <iostream>
#include <locale>
using namespace std;
int main()
{
    int *v,ns,i,s;
    setlocale(LC_ALL,"Russian");
    cout<<"Введите число элементов ";
    cin>>ns;
    v=new int[ns];
    for(i=0; i<ns; i++)
        v[i]=0;
    do
    {
        i=ns-1;
        while (i>=0)
        {
            if(v[i]==0)
```

```

        {
            v[i]=1;
            i=-1;
        }
    else
    {
        v[i]=0;
        i=i-1;
    }
}
for(i=0; i<ns; i++)
{
    cout.width(5);
    cout<<v[i];
}
cout<<'\n';
s=0;
for(i=0; i<ns; i++)
    s+=v[i];
}while(s<ns);
return 0;
}

```

Результат работы программы показан на рис. 6.9.

```

Введите число элементов 4
0 0 1 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
Для продолжения нажмите любую клавишу . . . _

```

Рис. 6.9. Результат работы программы листинга 6.6

В качестве примера применения данного алгоритма рассмотрим следующую задачу. Студент Иванов И. имеет некоторую сумму денег для покупки канцелярских принадлежностей. Он составил список необходимых канцелярских принадлежностей с указанием их названия и количества. Предположим, что в файле «input.txt» содержится прейскурант магазина, в котором указаны наименования товаров, имеющих в наличии, с указанием цены (рис. 6.10).

```

200
7
Альбом 1 Справочник 1 Тетрадь 5
Фломастер 5
Ручка 2 Скрепки 3 Степлер 1
11
Альбом 85 Тетрадь 5 Файл 12
Обложка 8 Фломастер 15 Ручка 12.2
Скрепки 25 Папка 32 Степлер 120
Справочник 180 Скоросшиватель 30

```

Рис. 6.10. Содержимое файла «input.txt»

Определить, какие канцелярские принадлежности может купить студент Иванов И. в пределах выделенной на их покупку суммы денег при условии, что количество наименований в списке покупок должно быть максимальным.

Введем одномерный массив размером, равным числу записей в списке покупок, и осуществим полную выборку в этом массиве. Значение элемента массива, равное 0, означает, что этот товар не покупается, а значение 1 означает покупку. Для описания списка покупок и прейскуранта цен создадим две структуры: `spisok` и `price`.

**Листинг 6.7.** Определить максимальное количество наименований в списке покупок, которые возможно купить в пределах определенной суммы денег.

```

// L6_7.cpp
#include "stdafx.h"
#include <string.h>
#include <fstream>
#include <iostream>
#include <locale>
using namespace std;
struct spisok
{
    char *name;           // Наименование товара
    float ves;           // Количество товара
};
// Прейскурант цен:
struct price
{

```

```

        char*name;                // Наименование товара
        float sum;                // Цена за единицу товара
};

int vibor(int*v,int ns);         //Прототип функции построения
                                //текущей выборки.

// Печать покупок:
void print_tov(int*,spisok*,price*,int,int);
int _tmain()
{
// *v – текущая выборка,
// *vmax – искомая выборка,
// ns – число записей в списке покупок,
// nt – число записей в прейскуранте цен,
// ntmp – текущее число купленных товаров,
// nmax – максимальное число купленных товаров.
// stmp – текущая сумма покупок,
// smax – имеющаяся сумма для покупок.
    int *v,*vmax,ns,nt,s,i,j,ntmp,nmax=0;
    float stmp,smax;
    spisok *my;
    price *tovar;
    char st[20];
    setlocale(LC_ALL,"Russian");
    ifstream ff("input.txt");
    ff>>smax;
    ff>>ns;
    v=new int[ns];
    vmax=new int[ns];
    my=new spisok[ns];
    for(i=0; i<ns; i++)
        v[i]=0;
    for(i=0; i<ns; i++)
    {
        ff>>st;
        my[i].name=strdup(st);
        ff>>my[i].ves;
    }
    ff>>nt;
    tovar=new price[nt];
    for(i=0; i<nt; i++)
    {
        ff>>st;
        tovar[i].name=strdup(st);

```

```

        ff>>tovar[i].sum;
    }
do
{
    s=vibor(v,ns);
    ntmp=0;
    stmp=0;
// Формирование текущего списка покупок:
    for(i=0; i<ns; i++)
    {
        if (v[i]==1)
        {
            for(j=0; j<nt; j++)
                if(strcmp(my[i].name,tovar[j].name)==0)
                {
                    stmp+=my[i].ves*tovar[j].sum;
                    ntmp++;
                }
        }
    }
// Если текущая сумма денег меньше имеющейся суммы для покупок,
// а список покупок больше, то запоминаем эту выборку:
    if(stmp<smax&&ntmp>nmax)
        for(i=0;i<ns;i++)
            vmax[i]=v[i];
}while(s);
cout<<"Выделенная сумма "<<smax<<"\n";
cout<<"\nСписок купленных товаров\n";
print_tov(vmax,my,tovar,ns,nt);
return 0;
}
int vibor(int*v,int ns)
{
    int i,s;
    i=ns-1;
    while (i>=0)
    {
        if(v[i]==0)
        {
            v[i]=1;
            i=-1;
        }
        else
        {

```

```

        v[i]=0;
        i=i-1;
    }
}
s=0;
for(i=0; i<ns; i++)
    s+=v[i];
if(s<ns)
    return 1;
else
    return 0;
}
void print_tov(int*v,spisok*my,price*tovar,int ns,int nt)
{
    int i,j;
    cout<<"\nНазвание Сумма\n\n";
    for(i=0; i<ns; i++)
    {
        if (v[i]==1)
        {
            for(j=0; j<nt; j++)
                if(strcmp(my[i].name,tovar[j].name)==0)
                {
                    cout.width(10);
                    cout.setf(ios::left);
                    cout<<my[i].name;
                    cout<<my[i].ves*tovar[j].sum<<"\n";
                }
        }
    }
}
}

```

Результат работы программы представлен на рис. 6.11.

<b>Выделенная сумма 200</b>	
<b>Список купленных товаров</b>	
<b>Название</b>	<b>Сумма</b>
Альбом	85
Тетрадь	25
Фломастер	75

Рис. 6.11. Результат работы программы листинга 6.7

### 6.3.2. Алгоритм получения перестановок

Рассмотрим, следуя [25], алгоритм получения перестановок. Переход от текущей перестановки  $V = (v_0, v_1, \dots, v_{n-1})$  к следующей за ней перестановке в лексикографическом порядке осуществляется при помощи алгоритма, блок-схема которого приведена на рис. 6.12. При обращении к функции в массиве  $v[i]$  находится текущая перестановка, а при выходе из нее в этом массиве находится следующая перестановка.

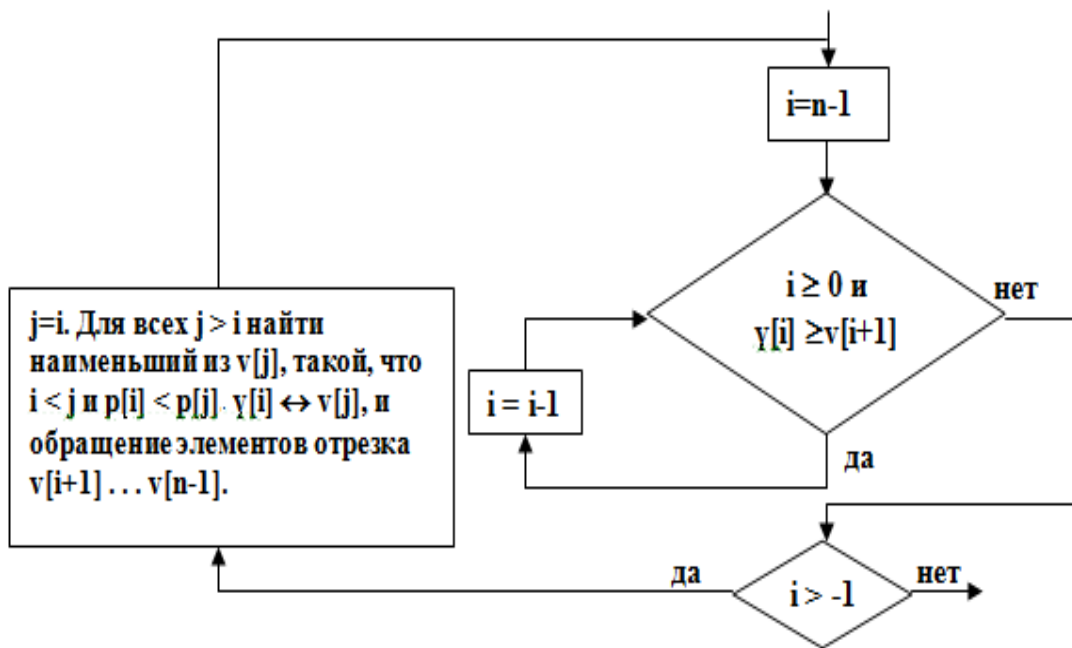


Рис. 6.12. Блок-схема получения следующей после текущей перестановки  $P = (p_0, p_1, \dots, p_{n-1})$  в лексикографическом порядке

**Листинг 6.8.** Функция нахождения перестановки, следующей за текущей перестановкой, в лексикографическом порядке. Функция возвращает 1, если эта перестановка не последняя, и 0, если она последняя.

```

// L6_8.cpp
int perestanovka(int*v,int n)
{
    int i,j,vr,ir,vm;
    for(i=n-2; v[i]>=v[i+1] && i>=0; i--)
        if(i<0) return 0;
    for(j=i+1,ir=j,vm=v[j]; v[i]<=v[j] && j; j++)
        if(v[j]<vm)
            {
  
```

```

        vm=v[j];
        ir=j;
    }
    vr=v[i];
    v[i]=v[ir];
    v[ir]=vr;
    for(i++, j=n-1; i<j; i++, j--)
    {
        vr=v[i];
        v[i]=v[j];
        v[j]=vr;
    }
    return 1;
}

```

В качестве примера использования алгоритма, рассмотрим задачу о коммивояжере, приведенную в пунктах 4.3.1 и 4.3.2. Напомним, что необходимо найти путь во взвешенном графе, проходящий через каждую вершину один раз, начинающийся и заканчивающийся в вершине с номером 0 и имеющий минимальную стоимость. Для решения этой задачи рассмотрим все перестановки от 1 до n-1. Первый путь предполагает посещение магазинов и дома в порядке: 0 (дом), 1, 2, ..., n-1, 0. Подсчитывается стоимость пути, затем получаем следующую перестановку, подсчитываем стоимость пути и т.д.

Содержимое исходного файла «input.txt» приведено на рис. 6.13.

		7				
60	2	7	8	1	9	2
2	60	4	3	5	12	7
7	4	60	3	2	9	8
8	3	3	60	4	6	3
1	5	2	4	60	2	1
9	12	9	6	2	60	9
2	7	8	3	1	9	60

Рис. 6.13. Содержимое файла «input.txt»: на главной диагонали стоят значения (в данном случае 60), большие всех возможных значений

**Листинг 6.9.** Программная реализация задачи о коммивояжере.

```

/L6_9.cpp
#include "stdafx.h"
#include <fstream>
#include <locale>

```



```

#include <iostream>
using namespace std;
int perestанovka(int*v,int n); // Прототип функции получения
                               // перестановок.
void print_v(int*v,int n);     // Прототип функции печати перестановок
int _tmain()
{
    ifstream ff("input.txt");
    int i,j,n,**A,*v,*vmin,s,smin=10000,f=1,k,l;
    setlocale(LC_ALL,"Russian");
    ff>>n;
    A=new int*[n]; // Матрица стоимости пути
    for(i=0;i<n;i++)
        A[i]=new int[n];
    v=new int[n];
    for(i=1;i<n;i++)
        v[i-1]=i; // Текущие перестановки
    vmin=new int[n];
    for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        ff>>A[i][j];
    cout<<"Исходная матрица\n";
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            cout.width(4);
            cout<<A[i][j];
        }
        cout<<"\n";
    }
    while(f)
    {
        s=A[0][v[0]];
        for(i=0;i<n-2;i++)
        {
            k=v[i];
            l=v[i+1];
            s+=A[k][l]; // Стоимость текущего пути
        }
        s+=A[v[n-2]][0];
        if(s<smin) // Поиск минимальной стоимости
        {
            smin=s;
        }
    }
}

```

```

        for(i=0;i<n-1;i++)
            vmin[i]=v[i];
    }
    f=perestanovka(v,n-1);
}
cout<<"Минимальная стоимость = "<<smin<<"\n";
cout<<"Путь:\n";
print_v(vmin,n-1);

return 0;
}
int perestanovka(int*v,int n)
{
    // см. листинг 6_8.
}
void print_v(int*v,int n)
{
    for(int i=0; i<n; i++)
        cout<<v[i]<<" ";
    cout<<"\n";
}

```

Результат работы программы показан на рис. 6.14.

```

Исходная матрица
60  2  7  8  1  9  2
 2 60  4  3  5 12  7
 7  4 60  3  2  9  8
 8  3  3 60  4  6  3
 1  5  2  4 60  2  1
 9 12  9  6  2 60  9
 2  7  8  3  1  9 60
Минимальная стоимость = 20
Путь:
1 2 3 5 4 6
Для продолжения нажмите любую клавишу . . .

```

Рис. 6.14. Результат работы программы листинга 6.9

## ГЛАВА 7. АЛГОРИТМЫ И КЛАССЫ

При разработке и реализации алгоритмов и программ возникает необходимость создания новых типов данных, отражающих особенности предметной области. При этом необходимо определить операции, допустимые для объектов этих типов. В языке C++ для типов, определяемых пользователем, используется понятие *класса*.

**Класс** – это структурированный тип данных, созданный на основе стандартных типов, который представляет собой модель реального объекта в виде данных и функций для работы с ними [54].

Построение классов основано на трех основных принципах: **инкапсуляции**, **наследовании** и **полиморфизме** [4, 47, 53, 54, 64 – 67].

**Инкапсуляция** – это принцип объединения в едином объекте и данных, и функций, обрабатывающих эти данные. Такой подход позволяет максимально изолировать объект от внешнего воздействия, что приводит к высокой надежности программ, использующих объекты.

**Наследование** – принцип построения классов, состоящий в том, что классы, в общем случае, могут составлять иерархию. Наследование предполагает, что все характеристики класса-родителя присваиваются классу-потомку. После этого потомку добавляют новые характеристики. Иногда некоторые методы в классе-потомке *переопределяются*, то есть наполняются новым содержанием.

**Полиморфизм** (дословно: *многоформие*) – принцип, лежащий в основе создания класса, когда родственные объекты, то есть происходящие от общего родителя, могут вести себя по-разному в зависимости от ситуации, которая возникает во время выполнения программы. Чтобы добиться полиморфизма, нужно один и тот же метод в классе-родителе переопределить в классе-потомке. Полиморфизм достигается за счет того, что методам из класса-родителя позволено выполняться в классе-потомке. Такие методы должны объявляться с атрибутом **virtual**. Если метод имеет атрибут **virtual**, то он может быть переопределен в классе-потомке. Переопределенный метод перекрывает метод базового класса.

В терминологии объектно-ориентированного программирования данные класса называются *полями*, а функции класса – *методами*. В литературе встречаются термины: «*данные-члены*» и «*функции-члены*», а также «*компонентные данные*» и «*компонентные функции*». Поля и методы называются *элементами класса*.

Описание класса имеет вид [54]:

```
class имя_класса {  
    [private:]          // Скобки [] указывают, что спецификатор  
                        // доступа действует по умолчанию.  
    закрытые данные и методы  
    спецификатор доступа;
```

```

        данные и методы
    спецификатор доступа:
        данные и методы
    // ...
    спецификатор доступа:
        данные и методы
} список_объектов; // Описание заканчивается точкой с запятой

```

Список объектов указывать не обязательно. Он просто позволяет объявлять **объекты класса** или **экземпляры класса**.

Методы связываются с конкретным классом оператором **::** (оператор разрешения области видимости) и обычно описываются сразу после описания класса, к которому они принадлежат, а в классе указываются их прототипы. В остальном они выглядят как обычные Си-функции:

```

тип имя_класса::имя_метода(список_формальных_параметров)
{ //операторы_тела_метода }

```

## 7.1. Конструкторы и деструкторы

При создании каждого объекта класса выделяется память, достаточная для хранения всех его полей, и автоматически вызывается метод – **конструктор**, выполняющий их инициализацию. Основная цель конструктора – инициализация переменных объекта данного класса или распределение памяти для их хранения. Конструктор имеет такое же имя, как и класс, в котором он определен.

Конструкторы характеризуются следующими свойствами:

- Конструктор не возвращает никакого значения, даже типа **void**.
- Конструктор без аргументов называется **стандартным конструктором** или **конструктором по умолчанию**.
- Возможно определение нескольких конструкторов с различными наборами аргументов. Возможности инициализации объектов в таком случае расширяются.
- С объектом всегда связано либо явное, либо неявное выполнение конструктора. Если отсутствует явно описанный конструктор, то есть нет метода, имя которого совпадает с именем класса, то С++ *генерирует автоматически конструктор по умолчанию* как функцию без параметров и с пустым телом.
- *Параметры конструктора* могут иметь любой тип, кроме этого же класса. С целью уменьшения числа конструкторов можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.

- Нельзя получить указатель на конструктор.
- Конструкторы нельзя описывать с модификаторами **const**, **virtual**, **static**.
- Конструкторы не наследуются.
- **Конструктор копирования** – это специальный вид конструктора, получающий в качестве единственного параметра ссылку на объект этого же класса:

$$T :: T(const T\&) \{ /* Тело конструктора */ \}$$

где  $T$  – имя класса. Этот конструктор вызывается в тех случаях, когда новый объект создается путем копирования существующего объекта.

При выходе объекта из области действия он уничтожается, при этом автоматически вызывается метод – **деструктор**. Функция – деструктор разрушает объект данного класса явно или неявно. Неявный вызов деструктора связан с прекращением существования объекта из-за завершения области его определения. Например, объект является локальным внутри функции, и функция возвращает управление. Явное уничтожение объекта выполняет оператор **delete**. Деструктор имеет такое же имя, как и класс, но предваряется символом  $\sim$  (**тильда**). В отличие от конструкторов деструкторы не могут получать аргументы и быть «перегружены».

## 7.2. Спецификаторы доступа

Для доступа к элементам объекта используется операция « . » (**точка**) при обращении к элементу через имя объекта и операция « -> » при обращении через указатель.

В качестве *спецификатора доступа* используется одно из ключевых слов языка C++: **public**, **private**, **protected**. Спецификаторы доступа **private** и **public** управляют видимостью элементов класса. Элементы, описанные после служебного слова **private**, видимы только внутри класса. Этот вид доступа принят в классе *по умолчанию*. Спецификатор доступа **protected** необходим только при *наследовании классов*. Интерфейс класса описывается после спецификатора **public** (то есть общедоступный). Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. В классе можно задавать несколько секций **private** и **public**, порядок их следования значения не имеет.

Поля класса:

- могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);

- могут быть описаны с модификатором **const**, при этом они инициализируются с помощью конструктора только один раз и не могут изменяться;
- могут быть описаны с модификатором **static**.

Инициализация полей при описании класса *не* допускается.

Каждый объект содержит свой экземпляр полей класса. Методы класса находятся в памяти в единственном экземпляре и используются всеми объектами совместно. Для обеспечения работы методов с полями именно того объекта, для которого они были вызваны, в метод передается скрытый указатель **this** на вызвавший функцию объект. Указатель **this** неявно используется внутри метода для ссылок на элементы объекта. В явном виде этот указатель применяется в основном для возвращения из метода указателя (**return this;**) или ссылки (**return \*this;**) на вызвавший объект.

## 7.3. Наследование классов

### 7.3.1. Простое наследование

Класс может *наследовать* данные-члены и методы одного или нескольких других классов [54]. При этом новый класс называют *производным классом* (или *подклассом*, или *классом-потомком*). Класс, элементы которого наследуются, называется *базовым классом* (или *родительским классом*, или *суперклассом*, или *надклассом*, или *классом-предком*) для своего производного класса. Часто говорят, что производный класс наследует свойства базового класса, поэтому их отношения называются *наследованием*. С помощью наследования некоторые общие черты поведения классов абстрагируются в одном базовом классе. Производные классы, наследуя это общее поведение, могут его несколько видоизменить путем переопределения некоторых методов базового класса, либо дополнять, вводя новые данные-члены и методы. Таким образом, определение производного класса значительно сокращается, так как нужно определить только те черты поведения, которые отличаются в производных классах. Если производный класс имеет всего *один* базовый класс, то говорят о *простом* (или *одионочном*) *наследовании*.

При описании производного класса в его заголовке перечисляются все классы, которые являются для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью *спецификаторов доступа* (или *ключей доступа*) **private**, **protected** и **public**. Если имеется несколько базовых классов, то они перечисляются через запятую. Ключ доступа может стоять перед каждым классом [47, 54]:

```

class имя_производного_класса : [private, protected, public]
имя_базового_класса {
    // тело производного класса
};

```

Здесь квадратные скобки [] означают, что ключи доступа, заключенные в них, указывать необязательно. Например, класс *student* – производный, он наследует три базовых класса: *NNGU*, *NGTU*, *NIMB*:

```

class NNGU { ... };
class NGTU { ... };
class NIMB { ... };
class student : NNGU, protected NGTU, public NIMB { ... };

```

При наследовании ключ доступа определяет уровень доступа к элементам базового класса, который получают элементы производного класса. Если в базовом классе элементы описаны со спецификатором доступа **private**, то эти элементы в производном классе недоступны вне зависимости от ключа доступа. Иначе говоря, закрытые члены базового класса остаются закрытыми, *независимо от того, как этот класс наследуется*. Обращаться к таким элементам можно только через методы базового класса. Элементы, описанные в базовом классе со спецификатором **protected**, при наследовании с ключом **private** становятся в производном классе закрытыми (**private**), в остальных случаях права доступа к ним не изменяются. Доступ к элементам, описанным в базовом классе со спецификатором **public**, при наследовании становится соответствующим ключу доступа [47, 54].

Следует отметить, что не все члены класса наследуются. Не наследуются следующие члены класса:

- Конструкторы.
- Конструкторы копирования.
- Деструкторы.
- Операторы присвоения, определяемые программистом.
- Друзья класса.

Следующий пример иллюстрирует простое наследование. Допустим, создаём программу, обрабатывающую информацию о студентах ННГУ, нам понадобятся классы:

```

class NNGU {
public:
    char * faculty;
    int number_of_course;
    // ...
};

```

```

class student : public NNGU {
    int number_of_group;
    student group[25];      // Список группы студентов
    long number_of_telephone;
    // ...
};

```

Класс *student* является производным от класса *NNGU*, а *NNGU* является базовым классом для класса *student*.

Класс *student* кроме своих собственных членов (*int number\_of\_group*; *student group[25]*; *long number\_of\_telephone*) наследует члены класса *NNGU* (*char \* fakultet*; *int number\_of\_kurs*;). Наследуемые компоненты *не* перемещаются в производный класс, а остаются в базовом классе. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс.

Конструктор базового класса всегда вызывается и выполняется до конструктора производного класса. Поскольку конструкторы не наследуются, в производном классе либо объявляется свой конструктор, либо производный класс должен предоставить возможность компилятору сгенерировать конструктор по умолчанию. Для конструктора производного класса, предоставляемого программистом, используется следующий синтаксис [47]:

```

    констр_производн_класса (список_параметров): констр_баз_класса
(список_arg)
    { // тело конструктора производного класса }

```

В конструкторе производного класса должны объявляться *все* параметры, которые необходимы базовому классу. Часть параметров, которые переданы конструктору производного класса, обычно используется в качестве аргументов конструктора базового класса. Для этого они указываются после двоеточия в списке аргументов конструктора базового класса. Затем в теле конструктора производного класса выполняется инициализация данных-членов, принадлежащих собственно этому классу.

Отметим, что конструктор производного класса может произвольно использовать *все* параметры, указанные в его объявлении, даже если они передаются конструкторам базового класса. Иначе говоря, передача параметров конструкторам базовых классов не исключает их использования внутри производного класса [60].

Деструктор производного класса должен выполняются *раньше* деструктора базового класса, то есть порядок уничтожения объекта противоположен по отношению к порядку его конструирования [35, 47, 53, 54, 60, 64, 65].

Если имена компонентов базового класса повторно определены в производном классе, то для доступа к таким компонентам базового класса из произ-



водного класса используется операция “::” разрешения (указания, уточнения) области видимости.

### 7.3.2. Множественное наследование

Если производный класс имеет несколько базовых классов, то говорят о *множественном наследовании*. *Множественное наследование* позволяет сочетать в одном производном классе свойства и поведение нескольких классов. Чаще всего один из базовых классов является основным, а другие обеспечивают некоторые дополнительные свойства, поэтому называются *классами подмешивания* [47, 54].

Класс называют *непосредственным (прямым) базовым классом (прямой базой)*, если он входит в список базовых классов при определении класса. Любой производный класс, в свою очередь, может быть базовым для других классов. В этом случае для производного класса могут существовать *косвенные* или *непрямые* предшественники, которые служат базовыми классами для классов, входящих в список базовых. Наследование в иерархии классов удобно представлять в виде дерева или в виде направленного ациклического графа (НАГ) [54, 59, 64, 65, 67, 73], где стрелкой изображают отношение «производный от». Производные классы принято изображать ниже базовых классов. Компилятор рассматривает их объявления именно в таком порядке, и их тексты размещаются в листинге программы.

Например,

```
class University { ... };  
class NNGU : public University { ... };  
class NGTU : public University { ... };  
class Institute { ... };  
class NIMB : public Institute { ... };  
class student : public NNGU, public NGTU, public NIMB { ... };
```

В графическом виде будем иметь (рис. 7.7):

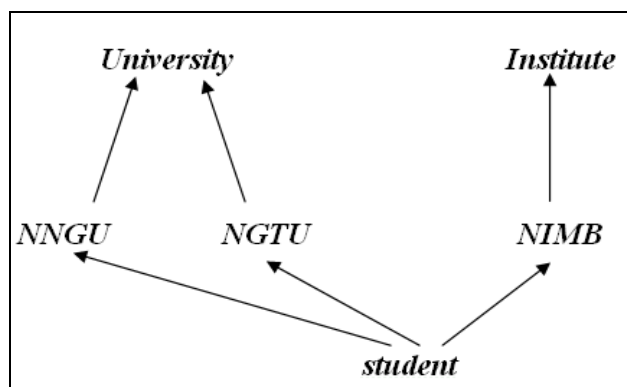


Рис. 7.1. Направленный ациклический граф классов

Здесь *University* – базовый класс, прямая база для классов *NNGU* и *NGTU*, а класс *Institute* – базовый класс, прямая база для класса *NIMB*. Классы *NNGU* и *NGTU* – производные от класса *University*, являются прямыми базами для класса *student*. Класс *NIMB* – производный от класса *Institute*, является прямой базой для класса *student*. Класс *student* – производный, этот класс имеет прямые базы *NNGU*, *NGTU*, *NIMB* и косвенные базы *University* и *Institute*.

Следует отметить, что ни базовый, ни производный классы *не* могут быть объявлены с помощью ключевого слова **union**, то есть при построении иерархии классов объединения использовать нельзя.

#### 7.4. Перегрузка операторов

В объектно-ориентированном программировании моделирование объектов реального мира осуществляется в терминах самих объектов. Классы часто носят названия предметов, например, класс СТРОКА, класс АВТОМОБИЛЬ. Встает вопрос: «Можно ли программистам обозначать операции-функции сложения и вычитания объектов этого класса традиционными символами «+», «-»?» В языке программирования C++ вводить новые операторы нельзя, но можно перегружать большинство встроенных операторов, настраивая их на конкретный класс, кроме указанных в табл. 7.1 [54]:

Таблица 7.1

Неперегружаемые операторы	
Оператор	Название
.	Выбор члена
.*	Выбор члена по указателю
::	Оператор расширения области видимости
? :	Оператор условия
#	Препроцессорный символ (оператор превращения в строку - stringize)
##	Препроцессорный символ (оператор конкатенации - pasting)

Перегруженные операторы реализуются в виде функций, имена которых содержат ключевое слово **operator**, за которым следует перегружаемый оператор. Такие функции называются *операциями-функциями* или *операторными функциями* [12, 47, 48, 54, 67].

Существуют *три* способа определения операций-функций [54, 59, 64, 65]:

а) операция-функция – *метод* класса:

```
тип_возвращаемого_значения имя_класса :: operator X (список параметров)
{ тело операции-функции }
```

где  $X$  – перегружаемый оператор.

При перегрузке унарного оператора список параметров остается пустым. При перегрузке бинарного оператора список параметров содержит один параметр. Причина заключается в том, что операнд, стоящий в левой части оператора, передается операторной функции неявно с помощью указателя **this**. Отсюда следует вывод: при перегрузке бинарного оператора вызов операторной функции генерируется объектом, стоящим в левой части оператора [54].

б) операция-функция – *дружественная функция*, не являющаяся членом класса:

```
friend тип_возвращаемого_значения operator  $X$  (список параметров);
```

где  $X$  – перегружаемый оператор.

Это значит, что дружественная функция не получает неявно указатель **this**. Следовательно, перегруженная операторная функция получает параметры явно. Таким образом, при перегрузке бинарного оператора дружественная функция получает два параметра, а при перегрузке унарного оператора – один. Первым параметром дружественной функции, перегружающей бинарный оператор, является его левый операнд, а вторым – правый операнд.

На применение дружественных операторных функций налагаются несколько ограничений [15, 24]. Во-первых, с помощью дружественных функций нельзя перегружать операторы «=», «()», «[]» и «->». Во-вторых, при перегрузке операторов инкрементации и декрементации параметр дружественной функции следует передавать по ссылке.

в) операция-функция – *обычная глобальная функция*:

```
тип_возвращаемого_значения operator  $X$  (список параметров)  
{ тело операции-функции }
```

где  $X$  – перегружаемый оператор.

Отметим, что операторная функция должна иметь хотя бы один параметр, который имеет тип класса, указателя или ссылки на класс. Как правило, операторные функции возвращают объекты классов, с которыми они работают

Какой же способ определения операции-функции предпочтителен? Во многих случаях способ перегрузки операторов не важен. В ситуациях, когда методы класса и дружественные функции эквивалентны, следует предпочесть методы класса. Однако бывают ситуации, в которых использование дружественных или глобальных операторных функций повышает гибкость перегружаемого оператора. Например,

```
class Overload {  
    int  $x$ ,  $y$ ;  
    public:
```

```

    Overload (int a=0, int b=0)    // Конструктор
    {
        x=a;
        y=b;
    }
    Overload operator+= (Overload ob) // Операторная функция –
                                        // метод класса Overload.

    {
        x+=ob.x;
        y+=ob.y;
        return *this;
    }
}; // Конец описания класса
Overload operator+ (Overload ob1, Overload ob2) // Операторная
                                                // функция – глобальная функция,
                                                // имеет аргументы типа класса.

{
    ob1+=ob2;
    return ob1;
}
void _tmain( )
{
    Overload ob1(15), ob2(17);
    Overload c1, c2, c3;
    c1=ob1+ob2;    // (1)
    c2=ob1+7;     // (2)
    c3=7+ob2;     // (3)
    ob1+=9;
}

```

Следует отметить, если бы операторная функция *operator+* была *методом* класса, то сложение вида (3) было невозможно, поскольку нет операции сложения переменной 7 целого типа **int** с объектом типа *Overload*.

При перегрузке операторов действуют следующие ограничения [47]:

- нельзя изменять число аргументов;
- нельзя изменять правила ассоциации (справа налево или слева направо) и правила приоритетов, используемые в стандартных типах данных;
- для стандартных типов данных переопределять операторы нельзя;
- операторные функции не могут иметь аргументов по умолчанию;
- операторные функции наследуются (за исключением оператора «=»);
- операторные функции не могут определяться со спецификатором **static**.

Операторную функцию можно объявлять и вызывать как любую другую функцию. Использование операторной функции как оператора – это сокращенная форма её вызова. Например,

```
Matrix& operator-( Matrix& p1, Matrix& q2) { ... }  
...  
void Func (Matrix& m1, Matrix& m2)  
{ Matrix a = m1 - m2;           // сокращённая форма, неявный вызов  
                                // операторной функции.  
  Matrix b = m1.operator-(m2); // явный вызов операторной функции  
}
```

## 7.5. Практические задачи применения алгоритмов с классами

### 7.5.1. Алгоритмы и классы для операций с датами

Рассмотрим класс date, который позволяет выполнять операции с датами. Поля класса: день, месяц и год являются закрытыми.

Методы класса позволяют выполнить следующие действия с датами:

- ✓ получение дня месяца и года;
- ✓ сложение даты с целым числом (результатом этой операции является новая дата);
- ✓ вычитание двух дат (результатом является число дней между датами);
- ✓ сравнение двух дат;
- ✓ печать даты в заданном формате;
- ✓ определение дня недели даты.

**Листинг 7.1.** Описание класса date и определение его методов.

```
// L7_1.cpp  
class date  
{  
    int dd, mm, yy;           //День, месяц, год  
public:  
                                //Конструктор по умолчанию, задается  
                                //дата: 1 января 2012 года.  
    date(int d=1,int m=1,int y=2012)  
    {  
        dd=d;  
        mm=m;  
        yy=y;  
    }  
};
```

```

    }
    int vis(int);           //Функция возвращает 1, если год
                           //високосный и 0, если не високосный.
    int date_control(void); //Контроль правильности ввода даты
    int& day(void);        //Возвращает день
    int& month(void);      //Возвращает месяц
    int& year(void);       //Возвращает год
    date operator+(int);
    int operator-(date&d);
    int operator>(date&d);
    int operator>=(date&d);
    void date_print(void); //Печать даты в формате dd.мм.yy
    char* date_day(void);  //Определение дня недели
};
//Массив, содержащий число дней в месяцах не високосного года.
//Индекс массива – это порядковый номер месяца в году.
int dm[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};
int date::vis(int y)
{
    if((y%4==0 && y%100!=0) || y%400==0)
        return 1;
    else
        return 0;
}
int date::date_control(void)
{
    if (yy<=0) return 0;
    if(mm<1 || mm>12) return 0;
    if(mm!=0 && (dd<1 || dd>dm[mm])) return 0;
    if(mm==2 && (dd<1 || dd>28+vis(yy))) return 0;
    return 1;
}
int& date::day(void)
{
    return dd;
}
int& date::month(void)
{
    return mm;
}
int& date::year(void)
{
    return yy;
}

```

```

date date::operator+(int n)
{
    int i;
    if (n>0)
    {
        for(i=1; i<=n; i++)
        {
            if((mm!=2&&dd+1<=dm[mm])||(mm==2&&dd+1<=28+vis(yy)))
                dd++;
            else
            {
                if(mm<12)
                {
                    mm++;
                    dd=1;
                }
                else
                {
                    yy++;
                    mm=1;
                    dd=1;
                }
            }
        }
    }
    else
    {
        for(i=1; i<=abs(n); i++)
        {
            if(dd-1>0)
                dd--;
            else
            {
                switch(mm)
                {
                    case 2: case 4: case 5: case 6: case 7: case 8: case 9:
                    case 10: case 11: case 12:
                    {
                        mm--;
                        dd=dm[mm];
                        break;
                    }
                    case 1:

```

```

        {
            yy--;
            mm=12;
            dd=31;
            break;
        }
        case 3:
        {
            mm=2;
            dd=28+vis(yy);
        }
    }
}
return *this;
}
int date::operator-(date&d)
{
    int r=0, ymin, dp1, dp2, i;
    if(yy<=d.yy)
        ymin=yy;
    else
        ymin=d.yy;
    dp1=0;
    for(i=ymin; i<yy; i++)
        dp1+=364+vis(i);
    dp2=0;
    for(i=ymin; i<d.yy; i++)
        dp2+=365+vis(i);
    if(mm==2)
        dp1+=31;
    if(mm>2)
    {
        dp1+=31;
        for(i=2; i<mm; i++)
            dp1+=dm[i];
        dp1+=vis(this->yy);
    }
    dp1+=dd;
    if(d.mm==2)
        dp2+=31;
    if(d.mm>2)
    {

```



```

        dp2+=31;
        for(i=2; i<d.mm; i++)
            dp2+=dm[i];
        dp2+=vis(d.yy);
    }
    dp2+=d.dd;
    return abs(dp1-dp2)-1;
}
int date::operator>(date&d)
{
    if(yy>d.yy) return 1;
    if(yy<d.yy) return 0;
    if(mm>d.mm) return 1;
    if(mm<d.mm) return 0;
    if(dd>d.dd) return 1;
    if(dd<d.dd) return 0;
    return 0;
}
int date::operator>=(date&d)
{
    if(yy>d.yy) return 1;
    if(yy<d.yy) return 0;
    if(mm>d.mm) return 1;
    if(mm<d.mm) return 0;
    if(dd>d.dd) return 1;
    if(dd<d.dd) return 0;
    return 1;
}
void date::date_print(void)
{
    char s[9];
    int y=yy%100;
    s[0]='0'+dd/10;
    s[1]='0'+dd%10;
    s[2]='.';
    s[3]='0'+mm/10;
    s[4]='0'+mm%10;
    s[5]='.';
    s[6]='0'+y/10;
    s[7]='0'+y%10;
    s[8]='\0';
    cout.width(12);
    cout<<s;
}

```

```

char* date::date_day(void)
{
    char *s;
    int ym,yc,k,mt,yt;
    if(mm>2)
    {
        mt=mm-2;
        yt=yy;
    }
    else
    {
        if(mm==1)
            mt=11;
        else
            mt=12;
        yt=yy-1;
    }
    ym=yt % 100;
    yc=yt / 100;
    k=int(2.6*mt-0.2)+dd+ym+ym/4+yc/4-2*yc;
    k %=7;
    if(k<0)
        k=7+k;
    switch (k)
    {
        case 0:
            s=strdup("Воскресенье");
            break;
        case 1:
            s=strdup("Понедельник");
            break;
        case 2:
            s=strdup("Вторник");
            break;
        case 3:
            s=strdup("Среда");
            break;
        case 4:
            s=strdup("Четверг");
            break;
        case 5:
            s=strdup("Пятница");
            break;
        case 6:

```

```

        s=strdup("Суббота");
    }
    return s;
}

```

Следуя [21], при определении дня недели полагаем, что дата лежит в диапазоне от 1582 г. до 4902 г. Согласно [21], номер дня недели (0 – воскресенье, 1 – понедельник, ..., 6 – суббота) равен остатку от деления на 7 значения выражения

$$D = [2,6 \cdot m - 0,2] + d + y + [y/4] + [c/4] - 2 \cdot c. \quad (7.1)$$

Здесь  $d$  – номер дня в месяце;  $m$  – номер месяца в году (год начинается с марта, то есть март – 1, апрель – 2, ..., декабрь – 10, а январь и февраль считаются 11-м и 12-м месяцами предыдущего года);  $y$  – две младшие цифры года;  $c$  – две старшие цифры года. Квадратные скобки  $[x]$  означают целую часть выражения  $x$ . Если остаток отрицателен ( $D < 0$ ), то номер дня недели определяется как  $D \% 7 + 7$ , где  $D \% 7$  – остаток от деления на 7.

*Листинг 7.2.* Программа, использующая класс date.

```

// L7_2.cpp
#include "stdafx.h"
#include <string.h>
#include <iostream>
#include <locale>
#include "L7_1.cpp"
using namespace std;
int _tmain()
{
    date dt,dt1;
    int d,m,y,n;
    setlocale(LC_ALL,"Russian");
    do
    {
        cout << "Введите день месяц и год ";
        cin >> d >> m >> y;
        dt.day()=d;
        dt.month()=m;
        dt.year()=y;
    }while(!dt.date_control());
    cout << "Введите n ";
    cin >> n;
    dt=dt+n;
}

```

```

dt.date_print();
cout << "\n";
do
{
    cout << "Введите день месяц и год второй даты ";
    cin >> d >> m >> y;
    dt1.day()=d;
    dt1.month()=m;
    dt1.year()=y;
}while(!dt1.date_control());
n=dt-dt1;
if(dt1>dt)
{
    dt.date_print();
    cout<<" - ";
    dt1.date_print();
}
else
{
    dt1.date_print();
    cout<<" - ";
    dt.date_print();
}
cout << "\n";
cout << "Число дней между двумя датами " << n << "\n";
dt.date_print();
cout << " " << dt.date_day() << "\n";
if(dt>dt1)
{
    dt.date_print();
    cout << " > ";
    dt1.date_print();
}
else
{
    dt.date_print();
    cout<<" <=";
    dt1.date_print();
}
cout << "\n";
return 0;
}

```

Результат работы программы (листинги 7.1 и 7.2) представлен на рис. 7.2.

```

Введите день месяц и год 30 2 1999
Введите день месяц и год 27 2 1999
Введите n 5
04.03.99
Введите день месяц и год второй даты 23 2 1998
23.02.98 - 04.03.99
Число дней между двумя датами 372
04.03.99 Четверг
04.03.99 > 23.02.98
Для продолжения нажмите любую клавишу . . . █

```

Рис. 7.2. Результат работы программы (листинги 7.1 и 7.2)

### 7.5.2. Алгоритмы и классы для создания баз данных

**Листинг 7.3.** Допустим, в файле «input.txt» содержатся сведения о студентах ННГУ: номер его студенческого билета, ФИО, дата рождения, дата поступления в университет, номер группы, адрес проживания, номер паспорта (рис. 7.3).

```

2
123 Иванов Иван Иванович 12 3 1998 31 8 2014 1 Н.Новгород, пр.Гагарина
23, к.1
12 3546
124 Петрова Анна Ивановна 22 4 1999 31 8 2014 1 Н.Новгород, Коминтерна
23, к.6
43 2456

```

Рис. 7.3. Содержимое файла «input.txt»

Необходимо осуществить ввод и вывод данных о других студентах. Для решения этой задачи нужно разработать класс *student*, считая, что в файле «input.txt» даты введены правильно.

```

// L7_3.cpp
#include "stdafx.h"
#include <string.h>
#include <fstream>
#include <iostream>
#include <locale>
using namespace std;
class student
{
    char *nsb; //Номер студенческого билета

```

```

char *fname,*iname,*oname;           //Фамилия, имя, отчество
int db,mb,yb;                         //Дата рождения
int di,mi,yi;                         //Дата поступления
int group;                            //Группа
char *adres;                          //Адрес
char *npasport;                       //Номер паспорта

public:
//Конструктор:
student(char *ns='\0',char *f='\0',char*i='\0',char*o='\0',int d=1,int
m=1,int y=1997,int d1=1,int m1=1,int y1=2001,int g=0,char*a='\0',char*np='\0')
{
    nsb=strdup(ns);
    fname=strdup(f);
    iname=strdup(i);
    oname=strdup(o);
    db=d;
    mb=m;
    yb=y;
    di=d1;
    mi=m1;
    yi=y1;
    group=g;
    adres=strdup(a);
    npasport=strdup(np);
}
//Функция ввода данных о новых студентах:
void student_new(char*ns,char *f,char*i,char*o,int d,int m,int y,int
d1,int m1,int y1,int g,char*a,char*np)
{
    nsb=strdup(ns);
    fname=strdup(f);
    iname=strdup(i);
    oname=strdup(o);
    db=d;
    mb=m;
    yb=y;
    di=d1;
    mi=m1;
    yi=y1;
    group=g;
    adres=strdup(a);
    npasport=strdup(np);
}
//Функция печати информации о студентах:

```

```

void student_print(void)
{
    char ss[9];
    char sf[50];
    int i;
    cout.width(4);
    cout<<nsb;
    for(i=0;fname[i]!='\0';i++)
        sf[i]=fname[i];
    sf[i++]=' ';
    sf[i++]=iname[0];
    sf[i++]='.';
    sf[i++]=oname[0];
    sf[i++]='.';
    sf[i]='\0';
    cout.width(20);
    cout<<sf;
    ss[0]='0'+db/10;
    ss[1]='0'+db%10;
    ss[2]='.';
    ss[3]='0'+mb/10;
    ss[4]='0'+mb%10;
    ss[5]='.';
    int y=yb%100;
    ss[6]='0'+y/10;
    ss[7]='0'+y%10;
    ss[8]='\0';
    cout.width(10);
    cout<<ss<<" ";
    ss[0]='0'+di/10;
    ss[1]='0'+di%10;
    ss[2]='.';
    ss[3]='0'+mi/10;
    ss[4]='0'+mi%10;
    ss[5]='.';
    y=yi%100;
    ss[6]='0'+y/10;
    ss[7]='0'+y%10;
    ss[8]='\0';
    cout.width(10);
    cout<<ss<<" ";
    cout<<group<<" ";
    cout<<npasport<<'\n';
    cout<<adres;
}

```

```

    }
};

int _tmain()
{
    setlocale(LC_ALL,"Russian");
    char ns[7],f[20],i[10],o[15],a[50],np[12];
    int ii,n,d,m,y,d1,m1,y1,g;
    ifstream ff("input.txt");
    student *st;
    ff>>n;
    st=new student[n];
    for(ii=0; ii<n; ii++)
    {
        ff>>ns>>f>>i>>o>>d>>m>>y>>d1>>m1>>y1>>g;
        ff.getline(a,50);
        ff.getline(np,12);
        st[ii].student_new(ns,f,i,o,d,m,y,d1,m1,y1,g,a,np);
    }
    cout<<"Инд.Ном.бил.        ФИО        Дата рождения Дата поступления
Группа Паспорт\n";
    for(ii=0; ii<n; ii++)
    {
        cout<<ii<<" ";
        st[ii].student_print();
        cout<<"\n";
    }
    return 0;
}

```

Результат работы программы листинга 7.3 приведен на рис. 7.4.

Инд.Ном.бил.	ФИО	Дата рождения	Дата поступления	Группа	Паспорт
0 123	Иванов И.И. Н.Новгород, пр.Гагарина 23, к.1	12.03.98	31.08.14	1	12 3546
1 124	Петрова А.И. Н.Новгород, Коминтерна 23, к.6	22.04.99	31.08.14	1	43 2456

Для продолжения нажмите любую клавишу . . .

Рис. 7.4. Результат работы программы листинга 7.3

### 7.5.3. Алгоритмы с использованием наследования классов



Следует обратить внимание на то, что в классе *student* содержится информация о конкретной персоне и о студенте, то есть этот класс перегружен. Имеет смысл класс *student* разделить на два класса: класс *persona* содержит информацию о человеке, а класс *student* – дополнительную информацию о студенте. Элементы этих классов будут относиться к одному и тому же человеку, если совпадают их номера (поле *number* в том и другом классе). В обоих классах содержится даты: дата рождения – в классе *persona* и дата поступления в ВУЗ – в классе *student*. Чтобы не повторять в этих классах одни и те же методы для работы с датами, будем использовать наследование. Класс *date* будет являться базовым классом для классов *persona* и *student* (рис. 7.5). Кроме того, для печати информации используем дружественную функцию, для которой доступны закрытые поля производных классов.

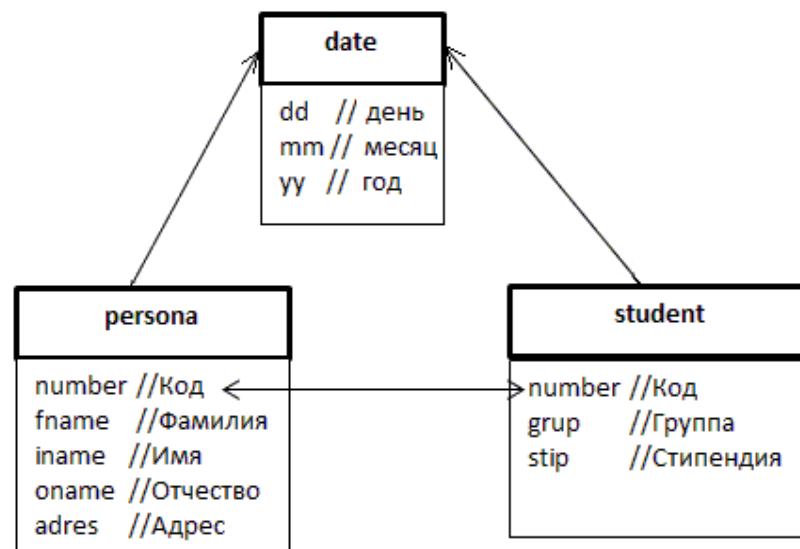


Рис. 7.5. Алгоритм взаимодействия классов: класс *date* – базовый для производных классов *persona* и *student*. Значение поля *number*, относящееся к одному и тому же человеку, должно совпадать для объектов производных классов

**Листинг 7.4.** Описание классов *date*, *persona*, *student* и их использование для обработки информации, содержащейся в файле «input.txt» (рис. 7.6).

```

2
123 Иванов Иван Иванович 12 3 1998 Н.Новгород, пр.Гагарина 23, к.1
621 678 31 8 2014
124 Петрова Анна Ивановна 22 4 1999 Н.Новгород, Коминтерна 23, к.6
611 0 31 8 2014
  
```

Рис. 7.6. Информация, содержащаяся в файле «input.txt»

```

// L7_4.cpp
#include "stdafx.h"
#include <string.h>
#include <iostream>
#include <fstream>
#include <locale>
using namespace std;
//Базовый класс date:
class date
{
    int dd,mm,yu;                //День, месяц, год
public:
    date(int d=1,int m=1,int y=2012) //Конструктор
    {
        dd=d;
        mm=m;
        yu=y;
    }
    void date_print(void);
};
class student;                //Прототип производного
                               //класса student.
class persona:public date    //Производный класс persona
{
    int number;                //Код
    char *fname;               //Фамилия
    char *iname;               //Имя
    char *oname;               //Отчество
    char*adres;                //Адрес
//Прототип дружественной функции печати:
    friend void student_print_all(persona,student);
public:
//Конструктор производного класса persona:
    persona(int nn=0, char* nfname=NULL, char* niname=NULL, char*
noname=NULL, char* ad=NULL, int d=1, int m=1, int y=2012) : date(d,m,y)
    {
        number=nn;
        fname=strdup(nfname);
        iname=strdup(niname);
        oname=strdup(noname);
        adres=strdup(ad);
    }
};
//Производный класс student:

```

```

class student: public date
{
    int number;           //Код
    int grup;            //Группа
    double stip;         //Стипендия
    //Прототип дружественной функции печати:
    friend void student_print_all(persona,student);
public:
    //Конструктор производного класса date:
    student(int num=0, int g=0, double s=0, int d=1, int m=1, int y=2012) :
date(d,m,y)
    {
        number=num;
        grup=g;
        stip=s;
    }
};
void date::date_print(void)
{
    char s[9];
    int y=yy%100;
    s[0]='0'+dd/10;
    s[1]='0'+dd%10;
    s[2]='.';
    s[3]='0'+mm/10;
    s[4]='0'+mm%10;
    s[5]='.';
    s[6]='0'+y/10;
    s[7]='0'+y%10;
    s[8]='\0';
    cout.width(12);
    cout<<s;
}
void student_print_all(persona p, student s)
{
    char st[40];
    int i;
    if(p.number==s.number)
    {
        cout.width(3);
        cout<<p.number<<" ";
        strcpy(st,p.fname);
        i=strlen(st);
        st[i++]=' ';
    }
}

```

```

        st[i++]=p.iname[0];
        st[i++]='.';
        st[i++]=p.oname[0];
        st[i++]='.';
        st[i]='\0';
        cout.width(25);
        cout.setf(ios::left);
        cout<<st<<" ";
        p.date_print();
        cout<<" ";
        s.date_print();
        cout<<" ";
        cout.width(4);
        cout<<s.grup<<"\n";
        cout<<"АДРЕС: ";
        cout.width(50);
        cout<<p.adres<<"\n\n";
    }
}
int _tmain()
{
    int d,m,y,d1,m1,y1,nn,n,i,g;
    double s;
    char sf[20],si[10],so[15],sa[50];
    ifstream ff("input.txt");
    persona *p;
    student *ps;
    setlocale(LC_CTYPE,"russian");
    ff>>n;
    p=new persona[n];
    ps=new student[n];
    for(i=0; i<n; i++)
    {
        ff>>nn>>sf>>si>>so>>d>>m>>y;
        ff.getline(sa,50);
        ff>>g>>s>>d1>>m1>>y1;
        p[i]=persona::persona(nn,sf,si,so,sa,d,m,y);
        ps[i]=student::student(nn,g,s,d1,m1,y1);
    }
    cout<<"Номер    ФИО            Дата рождения  Дата поступления
Группа\n";
    for(i=0; i<n; i++)
        student_print_all(p[i], ps[i]);
    return 0;
}

```

}

Результат работы программы листинга 7.4 представлен на рис. 7.7.

Номер	ФИО	Дата рождения	Дата поступления	Группа
123	Иванов И.И.	12.03.98	31.08.14	621
АДРЕС: Н.Новгород, пр.Гагарина 23, к.1				
124	Петрова А.И.	22.04.99	31.08.14	611
АДРЕС: Н.Новгород, Коминтерна 23, к.6				
Для продолжения нажмите любую клавишу . . .				

Рис. 7.7. Результат работы программы листинга 7.4

#### 7.5.4. Алгоритмы и классы с применением связанных списков

Для работы с объектами, рассмотренными в разделе 7.5.3, более эффективно пользоваться не массивами, а связными списками и очередями. Рассмотрим задачу о работе абонента библиотеки с применением связанных списков.

Предположим, что имеется список читателей, содержащий код, фамилию, имя, отчество, адрес и дату рождения. Содержимое хранилища книг характеризуется следующими параметрами: код книги, название, список авторов и признак наличия книги. Работа с картотекой читателей описывается параметрами: код читателя, код книги и дата выдачи книги. Необходимо заполнить список читателей в алфавитном порядке, описать содержимое хранилища книг в алфавитном порядке по названию книг и на этом основании обеспечить заполнение информации о выдаче книг. Введем обозначение классов: *persona* характеризует читателей, *book* – содержимое хранилища книг, *abonement* – работу абонента библиотеки.

При создании очереди объектов класса *abonement* сначала рассматривается список читателей и предлагается выбрать номер читателя. Далее рассматривается список книг, у которых признак выдачи означает, что книга имеется в наличии (признак выдачи – 0), и предлагается выбрать номер книги. Полученные номера читателя и книги заносятся в поля объекта класса *abonement* с указанием даты выдачи. Печатается список выданных книг. Затем выбирается текущая запись, и по коду читателя и книги находятся соответствующие объекты в списках объектов классов *persona* и *book*.

**Листинг 7.5.** Исходная информация о книгах, имеющихся в библиотеке, и читателях приведена в файле «input.txt» (рис. 7.8). Разработать программу, описывающую работу абонента библиотеки.

```

3
11 Иванов Иван Сергеевич 12 3 1978 Н.Новгород ул. Коминтерна 4/2 кв.58
12 Петрова Инна Васильевна 23 6 1978 Нижегородская обл. Арзамас ул.
Комарова 23 кв.66
13 Абрикосов Иван Иванович 22 6 1979 Тверь ул. Космонавтов 55 кв.12
4
1 Город в тумане
Петров И.В., Сергеева А.А.
2 Накануне
Тургенев И.С.
3 Цифровая крепость
Браун Д.
4 Двенадцать стульев
Ильф И., Петров Е.

```

Рис. 7.8. Содержимое файла «input.txt»

Алгоритм взаимодействия классов представлен на рис. 7.9.

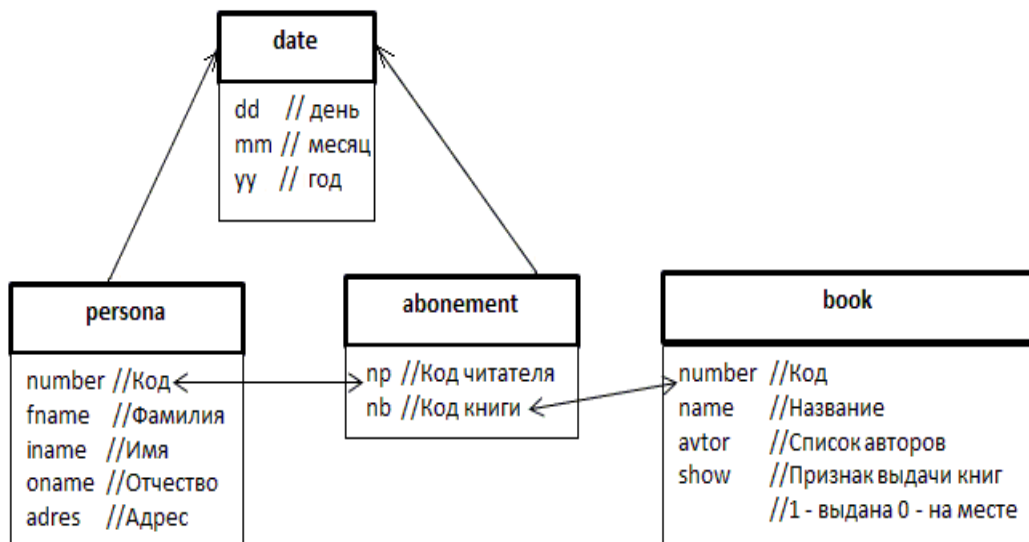


Рис. 7.9. Алгоритм взаимодействия классов: класс **date** – базовый для производных классов **persona** и **abonement**

```

// L7_5.cpp
#include "stdafx.h"
#include <string.h>
#include <fstream>
#include <iostream>
#include <locale>
using namespace std;
class date
{

```

```

        int dd,mm,yy;                //День, месяц, год
public:
    date(int d=1,int m=1,int y=2012) //Конструктор базового класса
    {
        dd=d;
        mm=m;
        yy=y;
    }
    void date_print(void);           //Прототип метода печати даты
};
class book;                         //Прототип класса book
class abonement;                    //Прототип класса abonement
class persona:public date           //Производный класс persona
{
    int number;                      //Код
    char *fname;                     //Фамилия
    char *iname;                     //Имя
    char *oname;                     //Отчество
    char *adres;                     //Адрес
    //Дружественная функция печати выданных книг:
    friend void abonement_print(abonement*,persona*,book*);
public:
    persona*next;
    //d,m,y – дата рождения:
    persona(int nn=0, char*nfname=NULL, char*niname=NULL,
char*noname=NULL ,char*ad=NULL,int d=1,int m=1,int y=2012):date(d,m,y)
    {
        number=nn;
        fname=strdup(nfname);
        iname=strdup(niname);
        oname=strdup(noname);
        adres=strdup(ad);
        next=NULL;
    }
    persona* persona_add(persona*); //Добавить читателя
    void persona_print(void);       //Печать списка читателей
};
class book                           //Класс book
{
    int number;                      //Код
    char *name;                      //Название
    char*avtor;                      //Список авторов
    int show;                        //Признак выдачи книг:
    //1 – выдана, 0 – не выдана.

```

```

        friend void abonement_print(abonement*,persona*,book*);
public:
    book*next;
    book(int nn=0,char*nam=NULL,char*a=NULL)
    {
        number=nn;
        name=strdup(nam);
        avtor=strdup(a);
        show=0;
        next=NULL;
    }
    book*book_add(book*);           //Добавить книгу
    void book_free_print(void);     //Печать книг,
                                    //находящихся в данный
                                    //момент в хранилище.

    int book_out(int);              //Установка признака
                                    //выдачи книги.
};
class abonement:public date        //Производный класс abonement
                                    //выдачи книг.
{
    int np;                          //Код читателя
    int nb;                          //Код книги
    friend void abonement_print(abonement*,persona*,book*);
public:
    abonement* next;
    //d,m,y – дата выдачи книги:
    abonement(int nnp=0,int nnb=0,int d=1,int m=1,int y=2012):date(d,m,y)
    {
        np=nnp;
        nb=nnb;
        next=NULL;
    }
    //Добавить выдаваемую книгу:
    void abonement_add(abonement**,abonement**);
};
void date::date_print(void)
{
    char s[9];
    int y=yy%100;
    s[0]='0'+dd/10;
    s[1]='0'+dd%10;
    s[2]='.';
    s[3]='0'+mm/10;

```



```

        s[4]='0'+mm%10;
        s[5]='.';
        s[6]='0'+y/10;
        s[7]='0'+y%10;
        s[8]='\0';
        cout.width(12);
        cout<<s;
    }
    persona* persona::persona_add(persona*f)
    {
        persona *ptr,*prev;
        int pf;
        if(!f)
            f=this;
        else
        {
            ptr=f;
            prev=NULL;
            pf=strcmp(ptr->fname,fname)<0;
            pf=pf||((strcmp(ptr->fname,fname)==0&&strcmp(ptr->iname,iname)<0);
            pf=pf||((strcmp(ptr->fname,fname)==0&&strcmp(ptr->
iname,iname)==0&&strcmp(ptr->oname,oname)<0);
            while(pf&&ptr)
            {
                prev=ptr;
                ptr=ptr->next;
                if(ptr)
                {
                    pf=strcmp(ptr->fname,fname)<0;
                    pf=pf||((strcmp(ptr->fname,fname)==0&&strcmp(ptr->iname,iname)<0);
                    pf=pf||((strcmp(ptr->fname,fname)==0&&strcmp(ptr->
iname,iname)==0&&strcmp(ptr->oname,oname)<0);
                }
            }
            if(!prev)
            {
                this->next=f;
                f=this;
            }
            else
            {
                prev->next=this;
                this->next=ptr;
            }
        }
    }

```

```

    }
    return f;
}
void persona::persona_print(void)
{
    persona *ptr=this;
    int i;
    char st[25];
    while(ptr)
    {
        cout<<"\n";
        cout.width(3);
        cout<<ptr->number<<" ";
        strcpy(st,ptr->fname);
        i=strlen(st);
        st[i++]=' ';
        st[i++]=ptr->iname[0];
        st[i++]='.';
        st[i++]=ptr->oname[0];
        st[i++]='.';
        st[i]='\0';
        cout.width(25);
        cout.setf(ios::left);
        cout<<st;
        ptr=ptr->next;
    }
}

void book::book_free_print(void)
{
    book*ptr=this;
    while(ptr)
    {
        if(ptr->show==0)
        {
            cout<<ptr->number;
            cout.width(40);
            cout<<ptr->name;
            cout.width(30);
            cout<<ptr->avtor<<"\n";
        }
        ptr=ptr->next;
    }
}

```

```

int book::book_out(int n)
{
    book *ptr=this;
    while(ptr&&ptr->number)
        ptr=ptr->next;
    if(ptr&&ptr->show==0)
    {
        ptr->show=1;
        return ptr->number;
    }
    else
    {
        cout<<"Такой книги нет!";
        return 0;
    }
}
book*book::book_add(book*f)
{
    book*ptr,*prev;
    if(!f)
        f=this;
    else
    {
        ptr=f;
        prev=NULL;
        while(ptr&&strcmp(ptr->name,name)<0)
        {
            prev=ptr;
            ptr=ptr->next;
        }
        if(!prev)
        {
            this->next=f;
            f=this;
        }
        else
        {
            prev->next=this;
            this->next=ptr;
        }
    }
    return f;
}
void abonement::abonement_add(abonement**b,abonement**e)

```

```

{
    if(*b==NULL)
        *b=*e=this;
    else
    {
        (*e)->next=this;
        *e=this;
    }
}
void abonement_print(abonement*b,persona*pf,book*pb)
{
    abonement *ptr=b;
    persona*pptr;
    book *bptr;
    char st[25];
    int i;
    cout<<"\nВзятые книги\n";
    while(ptr)
    {
        pptr=pf;
        while(ptr->np!=pptr->number)
            pptr=pptr->next;
        cout<<"\nЧИТАТЕЛЬ: ";
        strcpy(st,pptr->fname);
        i=strlen(st);
        st[i++]=' ';
        st[i++]=pptr->iname[0];
        st[i++]='.';
        st[i++]=pptr->oname[0];
        st[i++]='.';
        st[i]='\0';
        cout.width(25);
        cout.setf(ios::left);
        cout<<st;
        bptr=pb;
        while(bptr->number!=ptr->nb)
            bptr=bptr->next;
        cout<<"\n КНИГА: ";
        cout.width(30);
        cout<<bptr->name;
        cout<<" АВТОР: ";
        cout.width(30);
        cout<<bptr->avtor;
        ptr=ptr->next;
    }
}

```

```

    }
}

int main()
{
    int d,m,y,nn,n,i,np,nb;
    char sf[20],si[10],so[15],sn[30],sa[50];
    char c='y';
    ifstream ff("input.txt");
    persona *p,*pfirst=NULL;
    book *pb,*bfirst=NULL;
    abonement*pa,*begin=NULL,*end=NULL;
    setlocale(LC_CTYPE,"russian");
    //Заполнение списка читателей в алфавитном порядке
    ff>>n;
    for(i=0;i<n;i++)
    {
        ff>>nn>>sf>>si>>so>>d>>m>>y;
        ff.getline(sa,50);
        p=new persona(nn,sf,si,so,sa,d,m,y);
        pfirst=p->persona_add(pfirst);
    }
    //Заполнение хранилища книг:
    ff>>np;
    for(i=0;i<np;i++)
    {
        ff>>nn;
        ff.getline(sn,30);
        ff.getline(sa,50);
        pb=new book(nn,sn,sa);
        bfirst=pb->book_add(bfirst);
    }
    //Заполнение информации о выданных книгах:
    do
    {
        pfirst->persona_print();
        cout<<"\nВведите номер читателя ";
        cin>>np;
        bfirst->book_free_print();
        cout<<"Введите номер книги ";
        cin>>nb;
        i=bfirst->book_out(nb);
        if(i)
        {
            cout<<"Введите дату выдачи ";

```

```

        cin>>d>>m>>y;
        pa=new abonement(np,i,d,m,y);
        pa->abonement_add(&begin,&end);
    }
    cout<<"\nПродолжить? (y/n)";
    c=getch();
}while(c=='y'||c=='Y');
cout<<"\n";
//Печать списка выданных книг
abonement_print(begin,pfirst,bfirst);
getch();
return 0;
}

```

Результат работы программ листинга 7.5 показан на рис. 7.10.

```

13  Абрикосов И.И.
11  Иванов И.С.
12  Петрова И.В.
Введите номер читателя 12
1  Город в тумане
4  Двенадцать стульев
2  Накануне
3  Цифровая крепость
Введите номер книги 4
Введите дату выдачи 13 5 2014

Продолжить? <y/n>
13  Абрикосов И.И.
11  Иванов И.С.
12  Петрова И.В.
Введите номер читателя 13
1  Город в тумане
2  Накануне
3  Цифровая крепость
Введите номер книги 4
Такой книги нет!
Продолжить? <y/n>
13  Абрикосов И.И.
11  Иванов И.С.
12  Петрова И.В.
Введите номер читателя 13
1  Город в тумане
2  Накануне
3  Цифровая крепость
Введите номер книги 3
Введите дату выдачи 15 6 2014

Продолжить? <y/n>

Взятые книги
ЧИТАТЕЛЬ: Петрова И.В.
КНИГА: Двенадцать стульев
ЧИТАТЕЛЬ: Абрикосов И.И.
КНИГА: Цифровая крепость

Петров И.В., Сергеева А.А.
Ильф И., Петров Е.
Тургенев И.С.
Браун Д.

Петров И.В., Сергеева А.А.
Тургенев И.С.
Браун Д.

Петров И.В., Сергеева А.А.
Тургенев И.С.
Браун Д.

АВТОР: Ильф И., Петров Е.
АВТОР: Браун Д.

```

Рис. 7.10. Результат работы программы листинга 7.5

## **Упражнения**

1. Разработать алгоритм и программу обслуживания и ведения заказов авторемонтной мастерской. Информация должна содержать следующие сведения: клиент (Код, ФИО, адрес), исполнитель (Код, ФИО, должность) и информация о выполнении заказа (Код клиента, Код исполнителя, оплата и дата выполнения).
2. Создать алгоритм и программу обработки результатов сессии. Информация должна содержать сведения о студентах (Код, ФИО, группа, дата рождения), сведения о сдаваемом предмете (Код, название, семестр) и ведомость (Код студента, Код предмета, оценка и дата сдачи экзамена).
3. Построить алгоритм и написать программу, описывающую обращение больных в поликлинику. Информация должна содержать сведения о больных (Код, ФИО, адрес, дату рождения), о врачах (Код, ФИО, специальность) и сведения о приеме у врача (Код пациента, Код врача и дата осмотра).
4. Разработать алгоритм и программу, описывающую работу оптовой базы со своими покупателями. Информация должна содержать сведения о заказчиках (Код, название фирмы, адрес, телефон), сведения о заказываемых товарах (Код, наименование, дата выпуска, стоимость единицы продукции) и бланки заказов (Код заказчика, Код товара, дату заказа и количество заказанного товара).
5. Создать алгоритм и программу описания работы с клиентами фирмы по техническому обслуживанию торгового оборудования. Информация должна содержать сведения о мастерах, выполняющих ремонтные работы (Код, ФИО, квалификация, телефон), о магазинах, подающих заявки на ремонт оборудования (Код, наименование оборудования, магазин, адрес, телефон) и о выполнении заказов (Код мастера, Код магазина, дата выполнения и оплата).
6. Построить алгоритм и написать программу создания календаря чемпионата высшей лиги по футболу. Информация должна содержать сведения о клубах (Код, название, главный тренер, место дислокации), местах проведения матчей (Код, город, площадка) и календаря игр (Код клуба\_1, Код клуба\_2, Код места проведения матча и дата проведения матча).
7. Разработать алгоритм и программу работы страховой компании. Информация должна содержать сведения о компаниях (Код, название, ФИО агента, дату рождения, телефон связи, адрес), о клиентах (Код, ФИО, адрес, телефон) и договорах (Код компании, Код клиента, дату заключения сделки, страховую сумму).

8. Создать алгоритм и программу деятельности ремонтной бригады ЖКХ. Информация должна содержать сведения о работниках бригады (Код, ФИО, специальность), сведения о заказчиках (Код, ФИО, адрес, телефон), заказ-наряд (Код работника, Код заказчика, дата выполнения заказа).
9. Построить алгоритм и написать программу работы фермерского хозяйства. Информация должна содержать сведения о наемных работниках (Код, ФИО, адрес, дата рождения), о проводимых работах (Код, название, оплата) и сведения о выполнении работ (Код работника, Код работ и даты начала и окончания работы).
10. Разработать алгоритм и программу календаря проведения чемпионата высших учебных заведений по баскетболу. Информация должна содержать сведения об учебных заведениях (Код, название, главный тренер, место нахождения), местах проведения матчей (Код, город, площадка) и календаря игр (Код учебного заведения\_1, Код учебного заведения\_2, Код места проведения матча, дата проведения матча).
11. Создать алгоритм и программу работы центра занятости населения. Информация должна содержать сведения о работодателях (Код, название фирмы, адрес, телефон, должность, ставка), о потенциальных претендентах (Код, ФИО, адрес, телефон, дата рождения, стаж работы) и договор найма (Код фирмы, Код претендента и дата заключения договора о найме).
12. Построить алгоритм и написать программу работы бригады ремонта дорожных покрытий. Информация должна содержать сведения о сотрудниках бригады (Код, ФИО, адрес, телефон, специальность), о месте проведения работ (Код, адрес, оплата), о выполнении работ (Код исполнителя, Код места проведения, даты начала и окончания работ).
13. Разработать алгоритм и программу, описывающую ведение журнала успеваемости в школе. Информация должна содержать сведения о школьниках (Код, ФИО, дата рождения, адрес, телефон, класс), о преподаваемых предметах (Код, название, класс) и журнал успеваемости (Код школьника, Код предмета, дата ответа и оценка).
14. Создать алгоритм и программу, описывающую работу фотоателье. Информация должна содержать сведения о сотрудниках фотоателье (Код, ФИО, адрес, телефон, дата рождения), сведения о клиентах (Код, ФИО, адрес) и договорах обслуживания (Код сотрудника, Код клиента, дата проведения съемки, стоимость работы).
15. Построить алгоритм и написать программу графика распределения ролей в театре. Информация должна содержать сведения об актерах (Код, ФИО, дата рождения, адрес и телефон), о ролях (Код, название пьесы, название роли) и список исполнителей (Код актера, Код роли и дата проведения спектакля).



## Список литературы

1. Алгоритмы: построение и анализ. / Т.Х. Кормен, Ч.И. Лейзерсон, Р.Л. Ривест, К. Штайн. – М.: Издательский дом «Вильямс», 2005. – 1296 с.
2. Алябьева В.Г., Пастухова Г.В. Теория алгоритмов: Учебное пособие. // Пермь: Перм. гос. гуманитар.-пед. ун-т., 2013. – 125 с.
3. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979. – 536 с.
4. Ашарина И.В. Объектно-ориентированное программирование в C++: лекции и упражнения: учебное пособие для вузов. – М.: Горячая линия – Телеком, 2008. – 320 с.
5. Большая советская энциклопедия. – М.: Советская энциклопедия, 1969 – 1978.
6. Большая Энциклопедия Нефти Газа. URL: <http://www.ngpedia.ru/id328955p1.html>. Режим доступа – свободный.
7. Большой психологический словарь / Под ред. Б.Г. Мещерякова, В.П. Зинченко. – М.: Прайм – Еврознак, 2003. – 672 с.
8. Вирт Н. Алгоритмы и структура данных. – М.: Мир, 1989. – 358 с.
9. Гергель В.П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем. – М.: Издательство Московского университета, 2010. – 544 с.
10. Гергель В.П. Новые языки и технологии параллельного программирования. – М.: Издательство Московского университета, 2012. – 434 с.
11. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. – Н. Новгород: ННГУ, 2000. – 121 с.
12. Глушаков С.В., Коваль А.В., Смирнов С.В. Язык программирования C++: учебный курс. – Харьков: Фолио; М.: ООО «Издательство АСТ», 2001. – 500 с.
13. Горелик А.Л., Скрипкин В.А. Методы распознавания. М.: Высшая школа, 1989. – 231 с.
14. Гудман С., Хидетниемеи С. Введение в разработку и анализ алгоритмов. – М.: Мир, 1981. – 368 с.
15. Дайтибегов Д.М, Черноусов Е.А. Основы алгоритмизации и алгоритмические языки: Учебник для вузов по специальности «Экономическая информатика и АСУ». – М.: Финансы и статистика, 1992. – 495 с.
16. Дарахвелидзе П., Марков Е. Delphy – среда визуального программирования. СПб.: ВУН-Санкт-Петербург, 1996. – 352 с.
17. Дейтел Х.М., Дейтел П.Дж. Как программировать на C++, 5-е изд.: пер. с англ. – М.: ООО «Бином-Пресс», 2008. – 1456 с.
18. Емельянов А.А. Имитационное моделирование в управлении рисками. – СПб.: Инжекон, 2000. – 376 с.
19. Емельянов А.А., Власова Е.А. Имитационное моделирование в экономических информационных системах. – М.: Изд-во МЭСИ, 1998. – 108с.

20. Емельянов А.А., Власова Е.А., Дума Р.В. Имитационное моделирование экономических процессов. – М.: Финансы и статистика, 2004. – 368 с.
21. Задачи по программированию / С.А. Абрамов, Г.Г. Гнездилова, Е.Н. Капустина, М.И. Селюн – М.: Наука, 1998. – 224 с.
22. Информатика: Учебник. – 3-е перераб. изд. / Под ред. Проф. Н.В. Макаровой. – М.: Финансы и статистика, 2000. – 768 с.
23. Каймин В.А. Методы разработки программ на языках высокого уровня: Учебное пособие. – М.: МИЭМ, 1985. – 120 с.
24. Карпов Ю.Г. Теория автоматов: Учебное пособие. – СПб.: Питер, 2002. – 224 с.
25. Касьянов В, Н., Сабельфельд В,К. Сборник заданий по практикуму на ЭВМ. – М.: Наука, 1986. – 272 с.
26. Кинг Д. Создание эффективного программного обеспечения. – М.: Мир, 1991. – 284 с.
27. Кнут Д.Э. Искусство программирования. Т. 1-3. – М.: Издательский дом «Вильямс», 2000. – 832 с.
28. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ: Пер. с англ. – М.: Издательский дом «Вильямс», 2005. – 1296 с.
29. Кузнецов М.В., Симдянов И.В. С++. Мастер-класс в задачах и примерах. – СПб.: БХВ-Петербург, 2007. – 480 с.
30. Кузнецов Ю.А., Перова В.И. Имитационное моделирование экономических процессов: Учебно-методическое пособие. – Нижний Новгород: Издательство Нижегородского государственного университета, 2007. – 90с.
31. Кузнецов Ю.А., Перова В.И., Мичасова О.В. Имитационное моделирование экономических процессов с применением программного пакета ITHINK // Экономический анализ: теория и практика, №6, 2006. – С. 11 – 15.
32. Кузнецов Ю.А., Перова В.И., Мичасова О.В. Моделирование экономического поведения фирмы с применением программного пакета ITHINK // Государственное регулирование экономики. Региональный аспект: Материалы V Международной научно-практической конференции. – Нижний Новгород: Изд-во ННГУ, 2005. – С.186 – 188.
33. Кузнецов Ю.А., Перова В.И., Мичасова О.В. Работа с программным пакетом ITHINK: Учебно-методическое пособие. – Нижний Новгород: Издательство Нижегородского государственного университета, 2005. – 73с.
34. Лаптев В.В. С++. Объектно-ориентированное программирование: учебное пособие. – СПб.: Питер, 2008. – 464 с.
35. Лафоре Р. Объектно-ориентированное программирование в С++. Классика Computer Science: 4-е изд. – СПб.: Питер, 2008. – 928 с.
36. Линев А.В., Боголепов Д.К., Бастраков С.И. Технологии параллельного программирования для процессоров новых архитектур: Учебник. / Под ред. В.П. Гергеля. – М.: Издательство Московского университета, 2010. – 160 с.

37. Майерс Г. Искусство тестирования программ. – М.: Финансы и статистика, 1982. – 174 с.
38. Мальцев А.И. Алгоритмы и рекурсивные функции. М.: Наука, 1965. – 394 с.
39. Марков А.А. Теория алгорифмов. – М.-Л.: Изд-во Академии наук СССР, 1954. – 377 с.
40. Марков А.А., Нагорный Н.М. Теория алгорифмов. – М.: Наука, 1984. – 432 с.
41. Математика в понятиях, определениях и терминах. Часть 1: Пособие для учителей. / Авторы: О.А. Мантуров, Ю.Л. Солнцев, Ю.И. Соркин, Н.Г. Федин. Под ред. Л.В. Сабина. – М.: Просвещение, 1978. – 320 с.
42. Математический энциклопедический словарь. / Гл. ред. Ю.В. Прохоров; Ред. кол.: С.И. Адян, Н.С. Бахвалов, В.И. Битюцков, А.П. Ершов, Л.Д. Кудрявцев, А.Л. Онищик, А.П. Юшкевич. – М.: Советская энциклопедия, 1988. – 847 с.
43. Матросов ВЛ. Теория алгоритмов. М.: Прометей, 1989. – 188 с.
44. Минский М. Вычисления и автоматы. – М.: Мир, 1971. – 366 с.
45. Могилев А.В., Пак Н.И., Хеннер Е.К. Информатика. – Учебное пособие для студ. пед. вузов. – 4-е изд., стер. – М.: Издательский центр «Академия», 2007. – 848 с.
46. Москвин П.В. Азбука STL. – М.: Горячая линия – Телеком, 2003. – 262 с.
47. Павловская Т.А. С/С++. Программирование на языке высокого уровня. – СПб.: Питер, 2007. – 461 с.
48. Павловская Т.А., Щупак Ю.А. С++. Объектно-ориентированное программирование: Практикум. – СПб.: Питер, 2008. – 265 с.
49. Параллельные вычисления: технологии и численные методы: Учебное пособие в 4 томах. Том 1. / В.П. Гергель, К.А. Баркалов, И.Б. Мееров и др. – Н. Новгород: Изд-во Нижегородского госуниверситета, 2013. – 239 с.
50. Параллельные вычисления: технологии и численные методы: Учебное пособие в 4 томах. Том 2. / В.П. Гергель, К.А. Баркалов, И.Б. Мееров и др. – Н. Новгород: Изд-во Нижегородского госуниверситета, 2013. – 367 с.
51. Параллельные вычисления: технологии и численные методы: Учебное пособие в 4 томах. Том 3. / В.П. Гергель, К.А. Баркалов, И.Б. Мееров и др. – Н. Новгород: Изд-во Нижегородского госуниверситета, 2013. – 415 с.
52. Параллельные вычисления: технологии и численные методы: Учебное пособие в 4 томах. Том 4. / В.П. Гергель, К.А. Баркалов, И.Б. Мееров и др. – Н. Новгород: Изд-во Нижегородского госуниверситета, 2013. – 369 с.
53. Пахомов Б.И. С/С++ и MS Visual C++2008 для начинающих. – СПб.: БХВ-Петербург, 2008.-624 с.
54. Перова В.И. Программирование на С++ в среде Visual Studio .NET: Учебное пособие. – Нижний Новгород: Нижегородский госуниверситет, 2010. – 261 с.
55. Перова В.И., Матюков А.В. Имитационное моделирование в среде ITHINK: модель кредитования предприятий // Материалы международной

- научно-методической конференции “Болонский процесс: сотрудничество российских и европейских университетов”. – Нижний Новгород: Изд-во “Пламя”, 2006. – С. 65 – 69.
56. Перова В.И., Сабаева Т.А. Программирование на языке С++: Учебное пособие. – Нижний Новгород: Изд-во Нижегородского госуниверситета, 2013. – 132 с.
  57. Перова В.И., Чугунова Е.А. Объектно-ориентированное программирование на языке С++ в среде Visual Studio .NET: Часть 1: Учебно-методическое пособие. / Под ред. В.И. Перовой. – Нижний Новгород: Нижегородский госуниверситет, 2010. – 95 с.
  58. Перова В.И., Чугунова Е.А. Объектно-ориентированное программирование на языке С++ в среде Visual Studio .NET: Часть 2: Учебно-методическое пособие. / Под ред. В.И. Перовой. – Нижний Новгород: Нижегородский госуниверситет, 2009. – 95 с.
  59. Подбельский В.В. Язык Си++: Учебное пособие. – М.: Финансы и статистика, 2001. – 560 с.
  60. Прата С. Язык программирования С++. Лекции и упражнения, 5-е изд.: пер. с англ. – М.: ООО «И.Д. Вильямс», 2007. – 1184 с.
  61. Прохоров В.И., Погорелко И.А., Яковлев В.А. Основы программирования для электронных цифровых вычислительных машин. – М.: Высшая школа, 1967. – 340 с.
  62. Сидоренко В.Н. Системная динамика. – М.: ТЕИС, 1998. – 205 с.
  63. Солтер Н.А., Клеппер С.Дж. С++ для профессионалов: пер. с англ. – М.: ООО «И.Д. Вильямс», 2006. – 912 с.
  64. Страуструп Б. Дизайн и эволюция С++: пер. с англ. – М.: ДМК Пресс; СПб.: Питер, 2007. – 445 с.
  65. Страуструп Б. Язык программирования С++. Специальное издание: пер. с англ. – М.: ООО «Бином-Пресс», 2005. – 1104 с.
  66. Стронгин Р.Г. Исследование операций. Модели экономического поведения: Учебник. – Нижний Новгород: Издательство ННГУ, 2002. – 244 с.
  67. Тарасов В.Л. Программирование на С++: учебное пособие. – Н. Новгород: Изд-во ННГУ, 2006. – 310 с.
  68. Уайс М.А. Организация структур данных и решение задач на С++. – М.: ЭКОМ Паблишерз, 2008. – 896 с.
  69. Успенский В.А. Машина Поста. – М.: Наука, 1988. – 96 с.
  70. Успенский В.А., Семенов А.Л. Теория алгоритмов: основные открытия и приложения. – М.: Наука, 1987. – 288 с.
  71. Уэйт М., Прата С., Мартин Д. Язык Си. – М.: Мир, 1988. – 512 с.
  72. Федоров А., Рогаткин Д. Borland Pascal в среде Windows. – Киев: Диалектика, 1993. – 656 с.
  73. Фридман А.Л. Основы объектно-ориентированного программирования на языке Си++. – М.: Горячая линия – Телеком, 2001. – 232 с.
  74. Хьюз Дж., Мичтом Дж. Структурный подход к программированию. – М.: Мир, 1989. – 278 с.

75. Хювёнен Э., Сеппянен Й. Мир Лиспа. В 2-х т. Т.1: Введение в язык лисп и функциональное программирование. / Пер. с финск. – М.: Мир, 1990. – 447 с.
76. Чен М.С., Грифис С.В., Изи Э.Ф. Программирование на JAVA: 1001 совет: Наиболее полное руководство по Java и Visual J++: пер. с англ. – Минск: Попурри, 1997. – 640 с.
77. Шилдт Г. Полный справочник по C++, 4 – е издание. : пер. с англ. – М.: Издательский дом «Вильямс», 2006. – 800 с.
78. Шилдт Г. C++: методики программирования Шилдта: пер. с англ. – М.: ООО «И.Д. Вильямс», 2009. – 480 с.
79. [http://phys.bspu.by/static/lib/inf/prg/vb6\\_1/index.html](http://phys.bspu.by/static/lib/inf/prg/vb6_1/index.html). Режим доступа – свободный. Дата обращения: 04.01.2015.
80. <http://www.intuit.ru/studies/courses/46/46/lecture/1372?page=2>. Режим доступа – свободный. Дата обращения: 04.01.2015.

Валентина Ивановна **Перова**  
Татьяна Анатольевна **Сабаева**  
Дмитрий Тимофеевич **Чекмарев**

## РАЗРАБОТКА АЛГОРИТМОВ ДЛЯ РЕШЕНИЯ ЗАДАЧ НА ЭВМ

*Учебное пособие*

Федеральное государственное автономное образовательное учреждение  
высшего образования  
«Нижегородский государственный университет им. Н.И. Лобачевского».  
603950, Нижний Новгород, пр. Гагарина, 23.