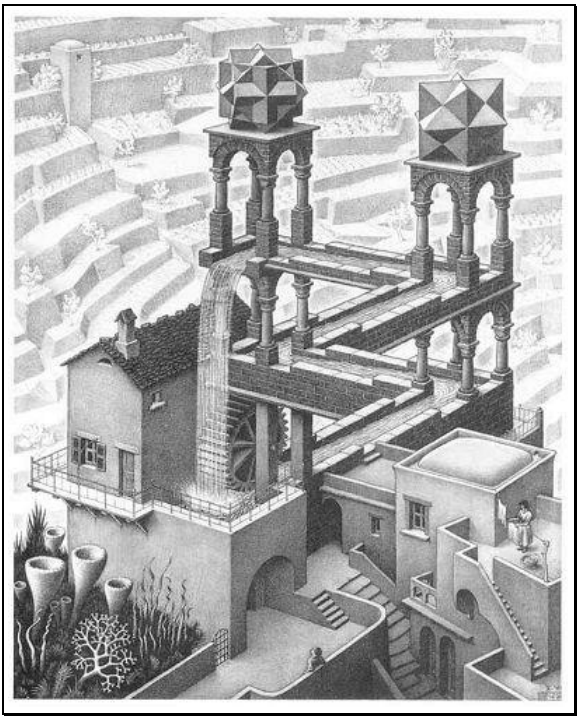
Санкт-Петербургский государственный политехнический университет



Е. В. Пышкин

Структуры данных и алгоритмы: реализация на C/C++

Редакция 2006 года с исправлениями

Санкт-Петербург

ФТК СПбГПУ

2009

УДК 004.415.2.043:004.42

Пышкин Е.В. **Структуры данных и алгоритмы: реализация на C/C++.** - СПб.: ФТК СПБГПУ, 2009.- 200 с., ил.

Учебное пособие дополняет курс лекций, читаемых студентам Санкт-Петербургского государственного политехнического университета. Рассматриваются основные структуры данных, в том числе связанные типы (списки, деревья, графы) и сложные контейнерные типы (массивы, ассоциативные массивы, стеки и очереди), а также особенности управления подобными типами в компьютерной программе. Применительно к анализу фундаментальных абстракций данных анализируются наиболее важные для проектной практики алгоритмы сортировки, поиска, обработки древовидных структур, алгоритмы лексического и синтаксического анализа и др.

Особое внимание уделяется построению алгоритмов, инвариантных к типам обрабатываемых данных – обобщенных алгоритмов, – и методов реализации таких алгоритмов средствами языка C++.

Предполагается, что студенты прослушали курс по программированию на языке C/C++ и имеют основные представления о процедурной и объектно-ориентированной парадигме программирования.

Курс рассчитан на 32 часа лекционных занятий (8 занятий по 4 часа) и самостоятельный практикум (соответствует примерно двум практическим занятиям по 4 часа).

Учебное пособие может быть также полезно студентам, изучающим курсы алгоритмизации, теории и технологии программирования, программирования на языке высокого уровня.

Ил. 63. Библиогр.: 29 назв.

Иллюстрация на обложке: М.К. Эшер. Водопад. Литография (1961)

Содержание

Введение **8**

Организация книги 8

Задания и упражнения по материалам лекций 9

Глава 1. Алгоритмы и типы данных **10**

1.1. Парадигмы программирования 10

Понятие об императивном программировании 10

Процедурная парадигма 12

1.2. Основные виды абстракций процедурного программирования 13

Иерархии процедур и функций 14

Модульность в процедурном программировании 14

1.3. Типы данных 16

Структуры и классы 19

Массивы 23

Множества 28

1.4. Алгоритмы и способы их записи 33

Текстуальная форма записи 34

Схема алгоритма 34

Псевдокод 35

Запись в форме программы на языке программирования 36

Запись алгоритмов функционирования реактивных систем 36

Построение программной модели конечного автомата 39

Глава 2. Оценка алгоритмов, рекурсия, сортировка 46

2.1. Постановка задачи внутренней сортировки и подходы к оценке эффективности 46

2.2. Сортировка простыми обменами 47

Реализация на примере сортировки массива целых чисел 48

Предварительная оценка эффективности 49

Улучшенная сортировка простыми обменами 51

Обобщение решения с использованием функций обратного вызова 52

Реализация в виде шаблонной функции 57

2.3. Рекурсивные алгоритмы 59

Задача поиска выхода из лабиринта 60

Быстрая сортировка Хоара (рекурсивный вариант) 62

Оценка эффективности быстрой сортировки 65

Реализация быстрой сортировки в итерационной форме 66

2.4. Другие классические алгоритмы внутренней сортировки и их анализ 70

Сортировка простыми вставками 70

Сортировка бинарными вставками 71

Сортировка Шелла 72

Сортировка простым выбором 74

Глава 3. Управление связанными структурами данных 77

3.1. Обработка линейного однонаправленного списка: основные операции 78

Постановка задачи 78

Реализация абстракции списка и базового комплекта операций 78

3.2. Определение других операций над списком. Первоначальное представление об итераторах 84

Недостатки просмотра списка с использованием внутреннего итератора 87

Внешнее управление работой внутреннего итератора 89

Построение внешнего итератора списка....91

3.3. Другие виды списков 95

Глава 4. Древовидные структуры и их применение....96

4.1. Организация древовидных структур....96

Бинарные деревья и их применение 97

Бинарное дерево как вариант организации данных в одномерном массиве....99

4.2. Алгоритм поиска с использованием бинарного дерева 99

Реализация бинарного поиска для ключей-строк символов (демонстрационный пример)....102

4.3. Алгоритм сортировки с использованием бинарного дерева 108

4.4. Двоичные деревья общего вида: удаление и добавление узлов 113

Красно-черные деревья как инструмент восстановления сбалансированности двоичных деревьев 118

Глава 5. Алгоритмы на графах....129

5.1. Некоторые напоминания из теории графов 129

Определение графа 129

Смежность и инцидентность....129

Подграф 130
Путь и расстояние 130
Связность 130
Древовидный граф 130
Способы задания 131
5.2. Поиск кратчайшего пути на графе 133
Структуры данных 133
Реализация алгоритма 133
Анализ работы алгоритма 137
5.3. Поиск минимального остовного дерева 138
Глава 6. Перемешанные таблицы и ассоциативные массивы 141
6.1. Алгоритм поиска по ключу с использованием hash-функций .. 141
Понятие hash-функции 141
Заполнение hash-таблицы 141
Поиск в подготовленной таблице 143
Удаление элементов из hash-таблицы 145
6.2. Распознавание служебных слов из фиксированного набора .. 146
6.3. Ассоциативные массивы 149
Выводы 152
Глава 7. Элементы лексического и синтаксического анализа 153
7.1. Постановка задачи 154
7.2. Лексический анализ 155

Понятие синтерма: непересекающиеся и пересекающиеся синтермы 157

Формирование и распознавание синтерма 157

7.3. Использование визуального формализма на базе L-сети в методических целях преподавания 161

Среда последовательного управления 162

Среда разветвленного управления 164

7.4. Решение задачи синтаксического анализа логических выражений в форме L-сети....165

Лексический анализатор в форме L-сети 166

Синтаксический анализатор в форме L-сети 167

Элементы реализации на языке C++....168

Глава 8. Алгоритмы обработки контейнеров....169

8.1. Специализированные контейнеры....169

Итерируемые специализированные контейнеры 169

Разработка итерируемого специализированного контейнера....171

8.2. Стандартные контейнеры 173

Понятие об итерации стандартного контейнера....174

Разработка контейнеров и алгоритмов, совместимых с STL 175

8.3. Реализация альтернативных вариантов размещения элементов контейнера в памяти 187

Список использованных источников....196

Введение

Независимо от объема и сложности разрабатываемого программного обеспечения, важным фактором, обеспечивающим надежность и гибкость конструируемых программ, является умение определить основные абстракции данных, используемых в проекте, и разработать или выбрать подходящие алгоритмы для эффективной обработки таких данных.

Программа курса согласуется и дополняет курсы процедурного и объектно-ориентированного программирования, при этом, не ограничиваясь одной парадигмой программирования, мы используем свойственный языку C++ гибридный характер и конструируем программный код, совмещающий преимущества структурного и объектно-ориентированного подходов.

При этом основная задача состоит не в изучении всего многообразия алгоритмов, а в обсуждении общих идей, создающих понятийную основу для дальнейшего самообразования, квалифицированной разработки и использования алгоритмов в соответствии с запросами практики.

В учебном пособии выделяется несколько важных для проектной практики областей, в том числе: представление типовых структур данных (массивов, множеств, списков, деревьев, графов), реализация связанных с ними алгоритмов (внутренней сортировки, поиска по ключу, поиска на графе). С разной степенью детализации в рамках курса рассматриваем развитие базовых структур (стеки, очереди, хеш-таблицы) и примеры организации кода, ориентированного на обработку таких структур. В курсе рассматриваются также основные алгоритмы, используемые в ходе синтаксического и лексического анализа.

Организация книги

В первой главе обсуждается понятие алгоритма в интуитивном смысле, вводится классификация вычислительных и поведенческих алгоритмов, рассматриваются различные способы описания алгоритмов. Обсуждается понятие типа данных, определение множества допустимых операций над данными, анализируется понятие абстрактного типа как основного механизма построения пользовательских типов. Рассматриваются простейшие варианты контейнерных типов (массивы и множества) и механизмы их реализации на C++.

Глава 2 содержит постановку задачи внутренней сортировки. Вводится понятие о сортировке на месте. Рассматривается устойчивость алгоритмов сортировки, а также основные факторы, влияющие на оценку эффективности. Анализируется классификация скоростей роста математических функций применительно к анализу алгоритмов.

На примере алгоритма сортировки простыми обменами рассматривается проблема проектирования обобщенных алгоритмов.

Далее изучаются рекурсивные алгоритмы, в частности, рекурсивный вариант быстрой сортировки Хоара. Анализируются проблемы рекурсивного

решения, и разрабатывается реализация алгоритма сортировки в итерационной форме. Вводится общая задача преобразования рекурсивных алгоритмов в итерационные. Рассматривается ряд других алгоритмов сортировки.

Глава 3 посвящена изучению связанных структур данных на примере линейного однонаправленного списка. Рассматриваются задачи представления списочной структуры, обеспечения обработки элементов списка во внешнем коде, построения итерируемого списка. Анализируются реализации на основе использования внутренних и внешних итераторов, рассматриваются их отличительных особенности.

В **главе 4** основное внимание уделяется построению и использованию древовидных структур. Наиболее подробно рассматриваются бинарные деревья, использование их в алгоритмах сортировки и поиска, обеспечение сбалансированности деревьев (включая метод на основе использования красно-черных деревьев).

В **главе 5** изучается программная реализация представления и обработки графов. Рассматривается алгоритм поиска кратчайшего пути между вершинами графа (постановка задачи и решение в форме законченной программы), а также алгоритм поиска минимального остова графа (постановка задачи и решение в псевдокоде).

Глава 6 посвящена решению задачи поиска и смежных задач на основе hash-таблиц. Подробно рассматривается метод поиска с линейным апробированием. Также в этой главе обсуждается реализация и использование ассоциативных массивов.

В **главе 7** рассматриваются основные задачи, возникающие в ходе лексического и синтаксического анализа. Рассматривается пример построения синтаксического анализатора логических выражений в скобочной форме.

В **главе 8** обсуждается понятие о контейнере данных. Рассматриваются особенности представления контейнеров и их обработка средствами стандартной библиотеки C++. Также уделяется внимание проектированию вмещающих типов с различными способами хранения элементов данных в памяти.

Задания и упражнения по материалам лекций

На лекциях студентам предлагаются проверочные работы продолжительностью 40-45 мин., а также задания самостоятельного практикума. Результаты проверочных работ и практикума учитываются при подведении итогов и выставлении экзаменационной оценки.

Глава 1. Алгоритмы и типы данных

Традиционно на начальном этапе основное внимание уделяется изучению двух парадигм программирования, а именно: процедурному и объектно-ориентированному программированию (ООП). Несмотря на то, что основным методом разработки коммерческого ПО на сегодняшний день является объектно-ориентированное проектирование (а также во многом произошедшее из него компонентное проектирование), изучение фундаментальных основ структурного программирования и принципов разработки функционально-ориентированного (процедурного) программного обеспечения остается актуальным.

Кроме того, на начальном этапе к задачам изучения основ организации данных и вычислительного процесса в основе своей легче адаптируется именно структурное программирование.

Некоторым недостатком традиционного обучения структурному программированию является изучение его обособленно, в отрыве от других методологий проектирования и даже в противопоставлении одних другим. В рамках курса мы постараемся хотя бы частично преодолеть данный учебно-методический разрыв. В этом смысле важнейшим результатом обучения должно быть понимание студентами того факта, что основные концепции проектирования, такие как абстрагирование, модульность, иерархическое проектирование и полиморфизм, не «принадлежат» какой-либо одной методологии программирования.

1.1. Парадигмы программирования

В данном разделе мы обсудим парадигмы программирования, на которые мы ориентируемся, разрабатывая структуры данных и алгоритмы.

Понятие об императивном программировании

Важнейшими факторами, влияющими на разработку языков программирования, являются архитектура компьютера и методологии программирования [Sebesta, 2002]. В конечном счете, большинство языков программирования являются выразителями концепций программирования и одновременно с этим своего рода моделями вычислительной системы [Пышкин, 2005]. Поэтому, определяя модель программирования, наряду с такими ключевыми факторами как организация управляющих конструкций языка, система типов данных, семантика основных конструкций, необходимо учитывать архитектуру вычислителя, на которую ориентирована эта модель.

Методология **императивного программирования** основана на принципе конструирования программ как описаний последовательного изменения состояния вычислителя пошаговым образом. Для императивного программирования характерна полная определенность и контролируемость переходов из одного состояния в другое [Одинцов, 2002].

Императивное программирование хорошо подходит для решения задач, характеризующихся не очень сложной логической структурой, для которых определение последовательности исполнения команд является наиболее естественным решением. На базе императивного программирования разработано множество управляющих программ (драйверов, компонентов встраиваемого ПО), обширная библиотека численных методов (например, на языке FORTRAN), программы для вычислительных машин с параллельной архитектурой. Императивная методология лежит в основе таких языков как FORTRAN, Pascal, Algol, C.

Императивное программирование ориентировано в первую очередь на получение эффективного и компактного исполняемого кода. Поэтому при увеличении сложности системы, необходимости реализации многовариантности исполнения программы, роста числа проверяемых условий и функциональных блоков императивное программирование затрудняет проектирование, сопровождение и модификацию системы.

Большинство императивных языков программирования разрабатывались на основе модели архитектуры компьютера, названной фон Неймановской, по имени одного из ее авторов – американского математика родом из Будапешта Джона (Яноша) фон Неймана.

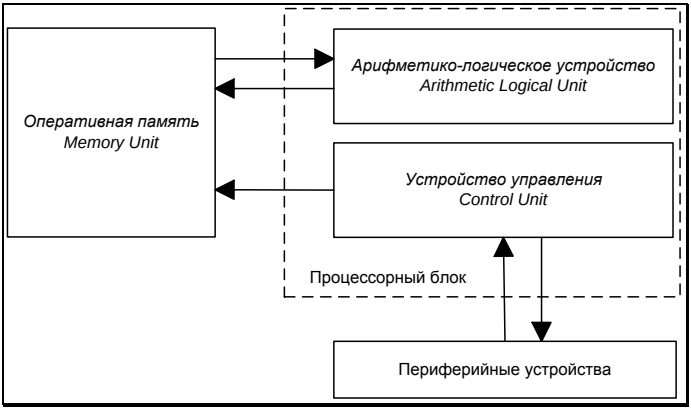


Рис. 1.1. Элементы архитектуры фон Неймана

В основе архитектуры фон Неймана лежит представление ЭВМ как композиции процессорного блока (выполняющего команды) и блока оперативной памяти (рис. 1.1) При этом основными элементами блока процессора является арифметико-логическое устройство (выполняющее операции) и устройство управления (обеспечивающее итерационную процедуру выполнения команд).

Одним из важнейших принципов, заложенных в фон Неймановскую модель, является хранение в одной и той же оперативной памяти и данных, и команд для обработки этих данных. Таким образом, главными элементами императивных языков программирования являются следующие элементы:

- переменные, моделирующие ячейки оперативной памяти;
- операции над переменными, включая важнейшую операцию присваивания, основанную на пересылке данных из одной ячейки в другую;
- комплект управляющих конструкций, основанный на итеративной форме повторений, обеспечивающих организацию пошаговых вычислений.

Императивный язык, по сути, определяет виртуальную вычислительную машину с архитектурой фон Неймана. Именно поэтому понимание организации вычислительного процесса на низком уровне является важной составляющей изучения императивного основания процедурных языков.

Процедурная парадигма

Развитием императивной методологии является методология структурно-императивного программирования, или просто – структурное программирование. И. Одинцов характеризует структурное программирование как подход, заключающийся в задании хорошей топологии императивных программ (см. [Одинцов, 2002]), что подразумевает следование ряду проектных принципов:

- Отказ от безусловной передачи управления (или, по крайней мере, ужесточение правил использования).
- Функционально-иерархическая декомпозиция.
- Разработка модулей, исходя из логической связи определяемых в модуле данных и процедур обработки этих данных.
- Использование для связи между модулями механизма связывания через параметры и возвращаемые значения процедур и функций.
- Независимая компиляция модулей.
- Ограничение использования переменных с глобальной областью видимости.
- Иерархическое проектирование по принципу «сверху – вниз».

В литературе встречаются два эквивалентных термина, обозначающих обсуждаемый здесь предмет: процедурное программирование и функционально-ориентированное программирование (не путать с функциональным программированием). Появление второго наименования, вероятно, обусловлено развитием технологий программирования, ориентированных на языки C и C++, где понятие «процедура», как правило, не используется.

Существенная часть основания процедурной парадигмы – структурно-императивное программирование – было рассмотрено выше. Основное отличие процедурных языков от простых императивных языков заключается в том, что процедурные языки предоставляют высокоуровневые средства для реализации таких важных концепций проектирования как абстрагирование,

иерархия и модульность. Процедурные языки ориентированы на развитие таких систематических методов проектирования как пошаговое уточнение структур данных и функционально-иерархическая декомпозиция. Изучение указанных концепций и методов и составляет основу изучения процедурного программирования.

1.2. Основные виды абстракций процедурного программирования

Процедурное программирование имеет дело с тремя основными видами абстракций: процессов, управляющих конструкций и данных. Абстракции процессов представлены процедурами и функциями, абстракции управляющих конструкций представлены комплектом основных управляющих конструкций структурного программирования, абстракции данных представлены структурами данных.

Использование в наименовании парадигмы определения «процедурная» показывает приоритетность абстракций процессов в процедурных языках. Не случайно в первых языках программирования основными высокоуровневыми понятиями были именно подпрограмма (subroutine) и ее категории – процедура и функция. Обособление элементов реализации в виде отдельных подпрограмм позволяет обеспечить следующие преимущества исходного текста:

- Бóльшую наглядность и понятность за счет абстрагирования от мелких деталей, требуемых для понимания внутреннего устройства процесса, но не нужных для понимания целей процесса.
- Исключение дублирования кода (в этом случае функциональное или процедурное обособление является обязательным).
- Упрощение построения модульной структуры на основе группирования процедур или функций по принципу общности выполняемых действий или обрабатываемых данных.
- Возможность распределения участков кода между различными проектировщиками.
- Возможность реализации функциональных блоков, выполняющих распространенные вычисления, с целью их повторного использования.
- Возможность раздельной отладки функциональных блоков.

Итак, проектирование на основе процедурной программы, предполагает организацию взаимодействия процедур и функций. Даже при реализации простых учебных задач студентам следует учитывать этот факт.

Важность конструирования абстракций данных стала осознаваться проектировщиками примерно в середине 70-х гг. прошлого века с началом интенсивного роста распространенности компьютеров. При проектировании программы очень важную часть разработки составляет построение модели данных, адекватно отображающей понятия предметной области в той части,

которая существенна для программной реализации. Таким образом, программист должен найти средства выражения не только для вычислительных алгоритмов, но и для обрабатываемых данных, используя существующие в современных процедурных и гибридных языках изобразительные средства.

Начало исследований в связи с совершенствованием абстракций управляющих конструкций приходится на 90-е гг., когда несовершенство управляющих конструкций большинства языков высокого уровня (не претерпевших существенных изменений за десятилетия развития вычислительной техники и областей ее применения) стало отмечаться многими специалистами. В литературе эта ситуация обычно называется методологическим разрывом между состоянием теории программирования и запросами практики проектирования, имеющей дело с возрастающей сложностью разрабатываемого программного обеспечения [Лекарев, 1997].

Иерархии процедур и функций

Процедурное программирование воплощает концепцию иерархии через иерархию процессов (в отличие от ООП, где на первый план выходит иерархия данных, и, скажем, от функционального программирования, где иерархичность проектирования вообще выражена существенно в меньшей степени).

Процедурное программирование иерархично по своей природе. Решение задачи средствами процедурного программирования всегда связано с построением иерархии процедур и функций (рис. 1.2).

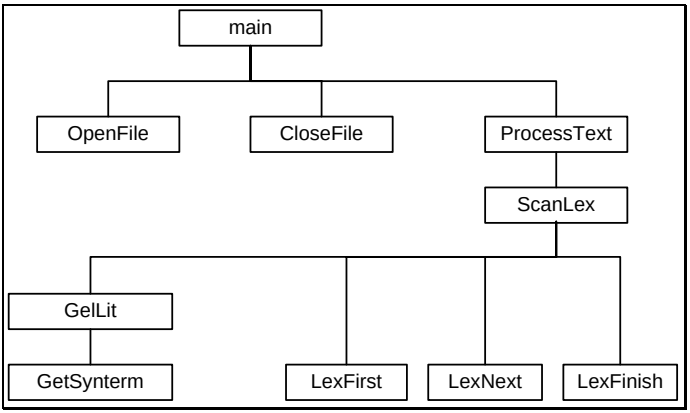


Рис. 1.2. Иерархия функций

Основным систематическим методом проектирования, нацеленным на реализацию иерархического проектирования в процедурном программировании, является функционально-иерархическая декомпозиция.

Модульность в процедурном программировании

Сложный проект обычно содержит большое число функциональных блоков. Однако рассмотрение программы только как единого набора процедур

и функций, не позволяет обеспечить адекватного сложности задачи уровня организации программы и управления вычислительным процессом. Поэтому происходит группирование функций в соответствии с общностью решаемых задач (рис. 1.3).

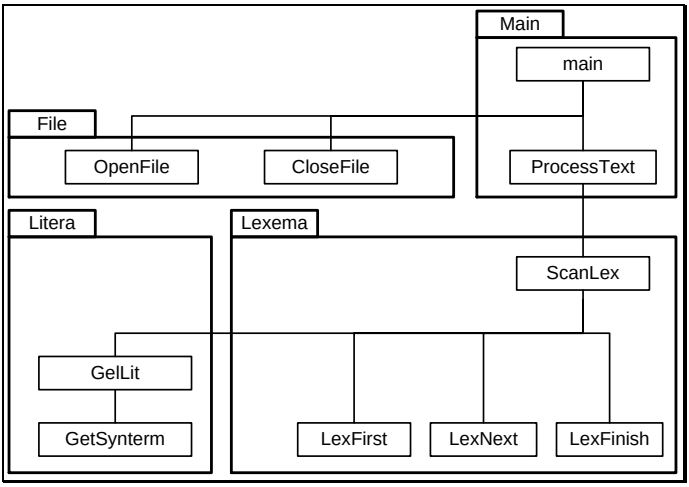


Рис. 1.3. Функции и модули

Наиболее распространенной единицей модульности чаще всего является отдельный файл (module, unit). При этом говорят о **модульности** на физическом уровне. Однако при построении сложных проектов физического уровня часто оказывается недостаточно, поскольку относительно обособленные наборы функций могут относиться к одной подобласти предметной области проекта.

Например, комплект средств по считыванию символов, определению синтерма считанной литеры и конструированию лексем входного текста удобно реализовать в виде отдельных физических модулей проекта, хотя все они и относятся к построению лексического анализатора (рис. 1.4).

Для выражения понятия модульности на уровне, учитывающем логические связи между функциями и обрабатываемыми им данными, во многих языках программирования реализованы механизмы **пространств имен**.

Пространство имен (namespace) является абстракцией более высокого уровня, поскольку, в общем случае, одно пространство имен может объединять объекты, определенные в разных файлах, с другой стороны, в одном файле могут быть определены объекты из нескольких пространств имен. Таким образом, концепции физической и логической модульности являются ортогональными (рис. 1.5).

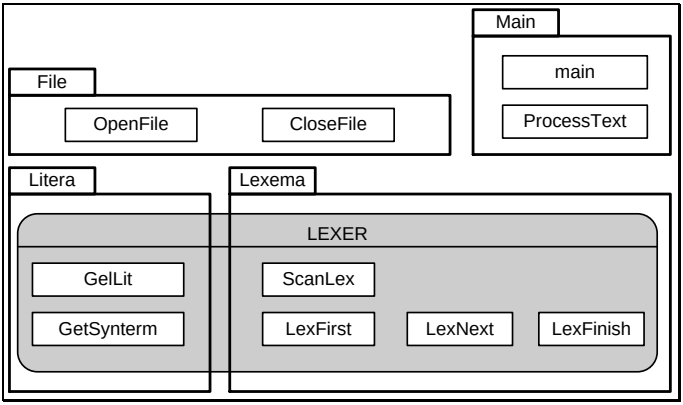


Рис. 1.4. Логический уровень модульности

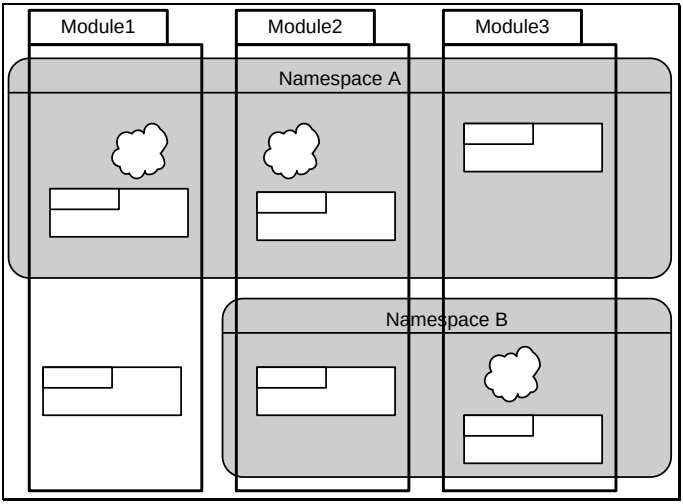


Рис. 1.5. Пространства имен

Подробно реализация пространств имен в ряде современных языков программирования (в том числе, в C++), рассматривается в учебном пособии [Пышкин, 2005].

1.3. Типы данных

В математике принято классифицировать математические объекты в соответствии с их основными особенностями. Поэтому вещественные переменные отличаются от комплексных, действительные от рациональных и трансцендентных, целочисленные вычисления от алгебры логики и т. п. Постановка и решение задач существенно облегчается, если отличать отдельные значения от множеств значений, а функции от множеств функций [Wirth, 1976]. Несмотря на то, что концепция типа данных в наибольшей

степени связана в настоящее время с языками и технологиями программирования, формирование основных элементов этой концепции началось еще до появления первых вычислительных машин.

Рассмотрим исторический пример из теории множеств. Несмотря на то, что в целом логики предпочитают не иметь дело с переменными различных типов, иногда без использования типизации трудно решить некоторые задачи.

Примером таких задач являются так называемые логические парадоксы (антиномии). Вот известный парадокс, сформулированный Расселом в 1902 году: «Пусть имеется множество *A* всех множеств *X*, не содержащих себя в качестве одного из элементов. Содержит ли множество *A* себя в качестве элемента?» Нетрудно убедиться, что любое предположение ведет к противоречию. Общеизвестна почти эквивалентная формулировка этого парадокса без использования понятий теории множеств: «Бреет ли брадобрей сам себя, если он бреет только тех, кто не бреет себя сам?» Рассел и Уайтхед еще в 1910-13 гг. предложили идею разрешения подобных парадоксов введением концепции типа переменных [Mendelson, 1963]. Действительно, если элементы множества приписать некоторому типу *Type1*, сами множества – типу *Type2*, а множества множеств – типу *Type3*, то парадокс просто перестает существовать: объект типа *Type3* может содержать внутри себя только объекты типа *Type2*, следовательно, формулировка парадокса является невозможным построением, похожим на рисунки Эшера (см. рис. на обложке).

Вероятно, наиболее полное перечисление отличительных особенностей понятия типа дал Ч. Хоар [Dahl, Dijkstra, Hoare, 1972]:

- Тип определяет класс значений, которые могут принимать переменная или выражение.
- Каждое значение принадлежит одному и только одному типу.
- Тип значения константы, переменной или выражения можно вывести либо из контекста, либо из самого операнда, не обращаясь к значениям, вычисляемым во время работы программы.
- Каждой операции соответствует некоторый фиксированный тип ее операндов и некоторый фиксированный (обычно такой же) тип результата. Разрешение систематической неопределенности в случае, когда один и тот же символ (например, символ '+') применяется к операндам разного типа, производится на стадии компиляции.
- Для каждого типа свойства значений и элементарных операций над значениями задаются с помощью аксиом.
- При работе с языком программирования знание типа позволяет обнаруживать бессмысленные конструкции и решать вопрос о методе представления данных и преобразования их в ЭВМ.

Тип данных позволяет абстрагировать то, как представляются данные в памяти компьютера и какие операции разрешены над этими данными. Таким

образом, обеспечивается однозначность интерпретации транслятором самих определений переменных и констант и однозначность семантики операций над этими объектами. Информация о типе позволяет транслятору удостовериться в корректности программных конструкций без выполнения программы. Справедливо, однако, и то, что в некоторых случаях транслятор не имеет возможности установить действительный тип обрабатываемого объекта во время компиляции (см., например, [Пышкин, 2005]), в этом случае, информация о типе объекта используется в процессе выполнения программы.

Несмотря на то, что концепция типа данных является одной из фундаментальных концепций программирования, некоторые языки не предоставляют синтаксических средств идентификации типа объекта (например, многие языки сценариев). Для некоторых объектов принадлежность к типу выводится из контекста. Однако в целом такие языки не обеспечивают полноценный контроль типов, некорректные операции могут быть выявлены (или, что еще опаснее, не выявлены) на этапе выполнения.

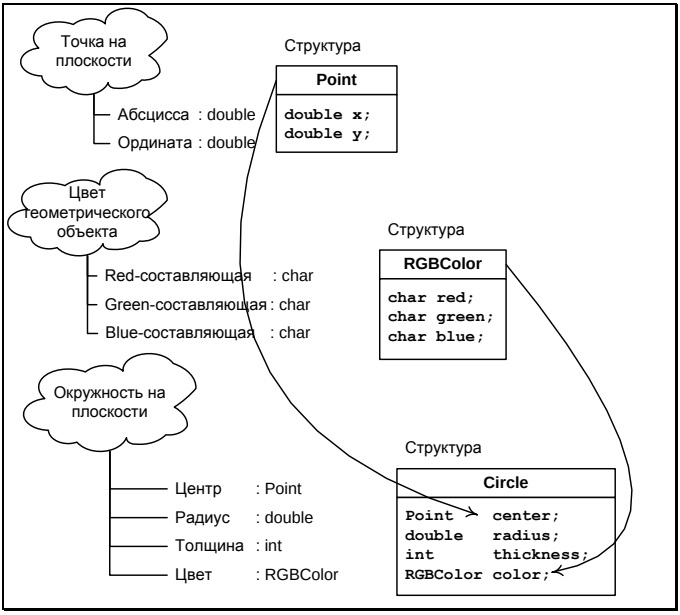


Рис. 1.6. Конструирование составных типов

Тип всегда является выразителем некоторой абстракции. Некоторые разновидности данных встречаются столь часто, что в большинстве языков для них предназначены **встроенные**, или **стандартные**, типы. К стандартным типам обычно относят числовые типы данных, логические (булевские) типы и символьные типы. Некоторые языки могут поддерживать стандартные типы, специфические для различных классов задач (например, тип `decimal` в языке

C#). Многие языки поддерживает типы для хранения адресной информации (например, указатели в языке C).

Тип, определяющий множество элементов, равномощное множеству натуральных чисел, (это означает, что элементы этого типа можно перенумеровать), называется **интегральным**, или **порядковым**, типом. В математике такие множества называются счетными. Например, типы `int`, `short`, `long`, `char`, `bool`, а также типы-перечисления (`enum`) в языке C++ являются интегральными.

Что касается абстракций, которые напрямую не выражаются встроенными типами, то большинство языков структурного программирования предоставляют инструментарий для конструирования новых типов данных, основой которого являются структуры (рис. 1.6).

Структуры и классы

Для многих практических случаев изобразительных средств, обеспечиваемых встроенными типами, недостаточно, поскольку требуется конструировать более сложные составные типы, включающие в себя элементы разных типов.

Так, при разработке шахматной программы, уместными могут оказаться типы «шахматная позиция» и «ход» [Linger, Mills, Witt, 1979]. При этом примером разрешенной операции может служить операция хода, применяемого к позиции (рис. 1.7). Как видно из рисунка, эта операция может быть определена как операция с побочным эффектом (шах, мат или пат).

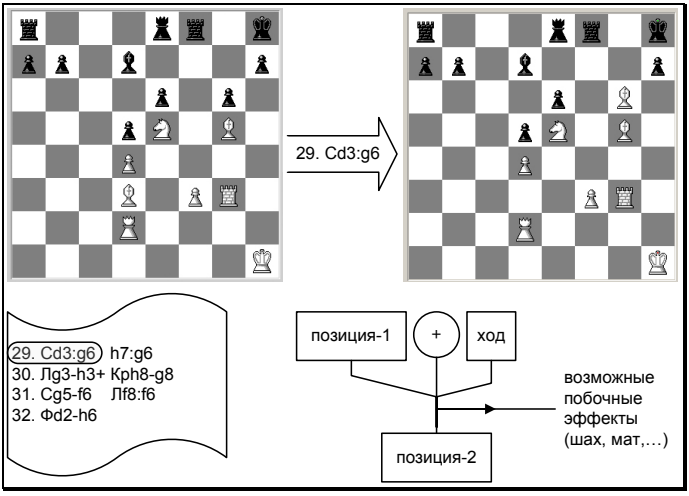


Рис. 1.7. Белые начинают и выигрывают

Такие составные типы обычно называют структурными типами, или просто – структурами. Примеры структурных типов были приведены ранее на рис. 1.6. Соответствующие определения типов на C++ выглядят следующим образом:

```
struct Point
{
    double x;
    double y;
};

struct RGBColor
{
    char red;
    char green;
    char blue;
};

struct Circle
{
    Point    center;
    double   radius;
    int      thickness;
    RGBCOlor color;
};
```

Обращение к полям структуры осуществляется посредством операции выбора поля структура («точка»):

```
Circle crcl;

crcl.radius = 10.0;
crcl.center.x = -1.5;
crcl.center.y = 0.7;
```

Во многих случаях структурные объекты управляются указателями:

```
Point *p;
```

В этом случае для обращения к полям структуры можно использовать явное разыменование с последующим обращением к полю структуры через «точку» или используя специальную операцию `->` (во втором случае разыменование осуществляется неявно):

```
// В предположении, что указатель p связан с какой-либо
// структурой типа Point...
(*p).x = 1.0; // Обращение с явным разыменованием
p->x = 1.0;   // Обращение с неявным разыменованием
```

Умение правильно определить основные типы данных разрабатываемой программы во многом определяет квалификацию программиста. Н. Вирт отмечает по этому поводу: «Сутью искусства программирования обычно считается умение составлять операции. Однако мы увидим, что не менее важно умение составлять данные» [Wirth, 1976].

Определение типа обычно предполагает не только моделирование данных, но и моделирование поведения, то есть определения правильных вариантов работы с объектами данных этого типа. Совмещение модели данных и поведения в рамках одной программной абстракции приводит к появлению понятия *класса* как основного механизма абстрагирования в объектно-ориентированном программировании. Листинг 1.1 иллюстрирует определение фрагмента класса, воплощающего абстракцию рациональной дроби.

Листинг 1.1. Определение класса «рациональная дробь»

```
/*
```

```
         * INCLUDE-файл    : Rational.h
         *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Определение класса "рациональное число"
 *
 * Дата создания     : 19.11.2006
 * Дата корректировки:
 */
#ifndef _Rational_h_
#define _Rational_h_

#include <iostream>
using namespace std;

// Определение класса "рациональная дробь"
class Rational
{
     int num;   // Числитель (может иметь знак)
     int denom;// Знаменатель (положителен)

public:
     // Конструктор
     Rational( int _num = 0, int _denom = 1 ) :
          num( _num ), denom( _denom )
     {
          if( denom <= 0 )
               throw out_of_range( "Incorrect rational value "
                                   "denominator "
                                   "(should be positive)" );
          simplify();
     }

     // Изменение значения рациональной дроби
     Rational& assign( int _num, int _denom = 1 )
     {
          num = _num;
          denom = _denom;

          if( denom <= 0 )
               throw out_of_range( "Incorrect rational value "
                                   "denominator "
                                   "(should be positive)" );
          simplify();
          return *this;
     }

     Rational& assign( const Rational& Val )
     {
          return assign( val.num, val.denom );
     }

     // Перегрузка арифметических операций
     Rational& operator+=( const Rational& arg2 )
     {
          int cmnDivisor = gcd( denom, arg2.denom );

          num = num * (arg2.denom / cmnDivisor) +
                arg2.num * (denom / cmnDivisor);
          denom = denom / cmnDivisor * arg2.denom;

          return simplify();
     }
```

```
     Rational operator+( const Rational& arg2 ) const
     {
          Rational result( num, denom );
          return result += arg2;
     }

     // Другие арифметические операции
     // ...

     // Перегрузка операций отношения
     int operator<( const Rational& arg2 ) const
     {
          int cmnDivisor = gcd( denom, arg2.denom );
          return num * (arg2.denom / cmnDivisor) <
                 arg2.num * (denom / cmnDivisor);
     }

     // Другие операции отношения
     // ...

     // Перегрузка операции вывода в выходной поток
     friend ostream& operator << ( ostream& out,
                                   const Rational& value )
     {
          int whole = value.num/value.denom;
          if( whole!=0 )
          {
               out << value.num/value.denom << " ";
               if( value.num%value.denom == 0 ) return out;
          }

          if( value.num != 0 )
               out << value.num%value.denom << "\\";
               out << value.denom;
          else out << "0";

          return out;
     }
     // ...

private:
     // "Сокращение" дроби
     Rational& simplify()
     {
          int cmnDivisor = gcd( abs( num ), denom );

          num /= cmnDivisor;
          denom /= cmnDivisor;

          return *this;
     }

     // Служебная функция:
     // вычисление наибольшего общего делителя
     static int gcd( int a, int b )
     {
          if( a==0 ) return 1;
          while( a!= b ) a>b ? a-=b : b-=a;
          return a;
     }
};
```

Приводимая ниже тестовая программа иллюстрирует использование объектов типа `Rational` аналогично объектам встроенных числовых типов:

```
#include "Rational.h"

void main()
{
     Rational value1( 7, 24 );
     Rational value2( 10, 36 );

     cout << "value1=" << value1 << endl;
     cout << "value2=" << value2 << endl;

     value2.assign( 1, 3 );
     cout << "value2=" << value2 << endl;

     value1 += value2;
     cout << "value1=" << value1 << endl;

     Rational value3 = value1 + Rational( 2 );
     cout << "value3=" << value3 << endl;

     if( value1 < value3 )     cout << "Ok" << endl;
     else                      cout << "???" << endl;
}
```

Массивы

Решение многих задач упрощается, если набор однотипных объектов хранить в виде последовательности соседних ячеек памяти. Доступ к этим ячейкам обычно осуществляется посредством индекса.

В языке C++ под индексом элемента массива понимают смещение относительно адреса начала массива в оперативной памяти, выраженное в элементах (рис. 1.8). В связи с этим в C++ принята модель, в соответствии с которой описание массива, по существу, сводится к описанию адресной переменной (указателя), хранящей адрес первого элемента массива (с индексом 0).

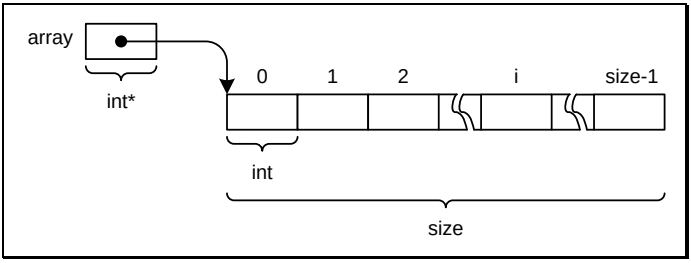


Рис. 1.8. Массивы в C/C++

Таким образом, в C++ индекс может изменяться от 0 до `size`-1, где `size` – число элементов в массиве. Отметим, что компилятор не имеет возможности контролировать ошибки индексирования при обращении к элементу массива. Ошибки индексирования (равно как и ошибки при работе с неинициализированными или некорректно инициализированными указателями) являются наиболее неприятными в связи с трудностью их обнаружения:

```
const int size = 100;
```

```
void main()
{
    int array[ size ];
    int ixFirst = 0;
    int ixLast = size - 1;

    // Корректные обращения
    array[ ixFirst ] = 10; // Эквивалентный код:
                           // *(array + ixFirst) = 10;

    array[ ixLast ] = 10;  // Эквивалентный код:
                           // *(array + ixLast) = 10;

    // Некорректные обращения
    //(выход за границу массива НЕ контролируется транслятором!)
    array[ -1 ] = 10;   // Эквивалентный код:
                        // *(array - 1) = 10;

    array[ size ] = 10; // Эквивалентный код:
                        // *(array + size) = 10;
}
```

Отметим, что другие языки могут реализовывать модель индексации, отличную от модели C++. Например, в языке Pascal индекс элемента массива по умолчанию изменяется в интервале 1..size. Программист может определить и другой диапазон изменения индекса (включая отрицательные значения индексов и использование в качестве индекса значений других порядковых типов):

```
const
    size = 100;
type
    IntArray = array[0..size-1] of integer;
    HexDigit = array[0..15] of char;
    IntGrades = -5..5;
    StringGrades = array[ IntGrades ] of string;
    WeekDay = ( Sunday, Monday, Tuesday, Wednesday, Thursday,
                Friday, Saturday );
var
    array1 : IntArray;
    hex : HexDigit = ( '0', '1', '2', '3', '4', '5', '6', '7',
                       '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' );
    grades : StringGrades = ( 'чудовищно',
                              'отвратительно',
                              'очень плохо',
                              'плохо',
                              'неважно',
                              'так себе',
                              'удовлетворительно',
                              'хорошо',
                              'очень хорошо',
                              'отлично',
                              'великолепно' );
    workHours : array[ WeekDay ] of integer;
```

Фактически, индекс массива в языке Pascal, является объектом данных типа «диапазон», жестко фиксирующего ограниченное множество значений, которые могут принимать элементы этого типа. Модель массива, реализованная в языке Pascal, не предусматривает явной работы программиста с адресной информацией. За счет промежуточного уровня абстрагирования

компилятор Паскаля обеспечивает строгий контроль корректности индексирования.

Программа на Паскале, распечатывающая значения элементов массива `grades` в файл, может выглядеть следующим образом:

```
const
    minGrade = -5;
    maxGrade = 5;
type
    IntGrades = minGrade..maxGrade;
    StringGrades = array[ IntGrades ] of string;
var
    grades : StringGrades = ( 'чудовищно',
                              'отвратительно',
                              'очень плохо',
                              'плохо',
                              'неважно',
                              'так себе',
                              'удовлетворительно',
                              'хорошо',
                              'очень хорошо',
                              'отлично',
                              'великолепно' );
    grd : Grades;
    fout : text;
begin
    assign( fout, 'output.txt' );
    rewrite( fout );
    for grd:=minGrade to maxGrade do
        writeln( fout, grades[ grd ] );
    close( fout )
end.
```

Отметим, что, если для типов `IntArray` и `HexDigit` можно сформулировать почти идентичные эквиваленты на C/C++, то для типа `Grades` и объявляемого на его основе типа `StringGrades` строгого эквивалента на C/C++ нет, хотя, конечно, подобную семантику реализовать можно (листинг 1.2).

Листинг 1.2. Массивы в C/C++

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

const int minGrade = -5;
const int maxGrade = 5;
const int totalGrades = maxGrade-minGrade+1;

inline int gradeToIndex( int grd )
{
    // Ответственность за корректность диапазона - на программисте
    assert( grd >= minGrade && grd <= maxGrade );

    return grd + 5;
}

typedef char *   GradeStr;
typedef GradeStr StringGrades[ totalGrades ];

void main()
{
```

```
    StringGrades grades = { "чудовищно",
                            "отвратительно",
                            "очень плохо",
                            "плохо",
                            "неважно",
                            "так себе",
                            "удовлетворительно",
                            "хорошо",
                            "очень хорошо",
                            "отлично",
                            "великолепно" };

    // Распечатать все элементы массива grades
    // в файл "output.txt"

    // Открыть поток
    FILE *fout = fopen( "output.txt", "wt" );
    if( fout == NULL )
    {
        printf( "Error while opening output file" );
        exit( 1 );
    }

    // Напечатать содержимое массива grades "в столбик"
    for( int grd = minGrade; grd <= maxGrade; grd++ )
    {
        fputs( grades[ gradeToIndex( grd ) ], fout );
        fputs( "\n", fout );
    }

    // Закрыть поток
    if( fclose( fout ) == EOF )
    {
        printf( "Error while closing output file" );
        exit( 2 );
    }
    exit( 0 );
}
```

Перечень операций над массивами в существенной степени зависит от того, как реализована абстракция массива в языке программирования. Рассмотрим правила работы с массивами, принятые в C/C++.

В языке C любая операция над массивом в целом (например, копирование одного массива в другой) сводится к последовательности операций над отдельными элементами. При этом ответственность за корректное использование памяти, отведенной под массивы, полностью лежит на программисте:

```
void CopyIntArray( int *array1, int size1,
                   int *array2, int size2 )
{
    if( size1 < size2 )
    {
        // Ошибка. Результирующий массив слишком мал
        // ...
        return;
    }
    for( int ix = 0; ix < size2; ix++ )
    {
        array1[ ix ] = array2[ ix ];
    }
```

```
}
```

В C++ вместо массивов C программист может использовать контейнерные классы из библиотеки STL (например, `vector`). Класс `vector` определяет абстракцию динамического массива объектов произвольного типа. Листинг 1.3 содержит пример использования класса `vector` для обработки рациональных дробей (определение класса `Rational` - см. листинг 1.1).

Листинг 1.3. Использование класса vector

```
#include <vector>
#include <algorithm>
#include <stdexcept>
#include <iostream>
#include <iomanip>

using namespace std;

#include "Rational.h"

// Функция печати содержимого произвольного контейнера,
// поддерживающего прямую итерацию
template <class C>
void print( const C& c, ostream& out = cout )
{
     typename C::const_iterator it = c.begin();
     int counter = 0;

     while( it != c.end() )
     {
          out << out.width( 2 ) << counter << ": " << *it << endl;
          ++counter;
          ++it;
     }
}

// Функция вычисления суммы элементов произвольного контейнера,
// поддерживающего прямую итерацию
template <class C>
typename C::value_type sum( const C& c )
{
     typename C::value_type s( 0 );
     typename C::const_iterator it = c.begin();

     while( it != c.end() )
     {
          s += *it; // NB! Для пользовтельского типа должна
                    //     быть определена семантика операции +=
          ++it;
     }
     return s;
}

void main()
{
     vector<Rational> vec;

     vec.push_back( Rational( 7, 24 ) );
     vec.push_back( Rational( 10, 36 ) );
     cout << "Size(vec) = " << vec.size() << endl;
     print( vec );

     Rational result = sum( vec );
```

```
    cout << "sum = " << result << endl;

    vec.insert( vec.begin()+1, Rational( 2, 3 ) );
    cout << "Size(vec) = " << vec.size() << endl;
    print( vec );
    result = sum( vec );
    cout << "sum = " << result << endl;

    sort( vec.begin(), vec.end() );
    print( vec );
}
```

Подробнее об использовании типов и алгоритмов STL см., например, книги [Austern, 1999], [Sytroustrup, 2000] или учебное пособие [Давыдов, 2005]. Основные принципы конструирования контейнерных типов и алгоритмов, совместимых с STL, обсуждаются в гл. 8 данного учебного пособия.

Массивы, рассмотренные в приведенных выше примерах, являются одномерными: для доступа к отдельным элементам требуется один индекс. В большинстве практических случаев средств, обеспечиваемых одномерными массивами, вполне достаточно, однако иногда программисту приходится сталкиваться с необходимостью обработки массивов, имеющих два (матричные контейнеры, см. также гл. 8) или более измерений.

Множества

Аналогично массивам, множества представляют собой набор однородных объектов, однако, в отличие от массивов, порядок следования элементов множества является несущественным. Если для массива основной операцией доступа к элементу является операция обращения по индексу, то для множества – это операция проверки принадлежности элемента множеству. Кроме того, над множеством определяют ряд специфических операций: добавление элемента в множество, объединение, пересечение и разность множеств, проверка является ли одно множество подмножеством или надмножеством другого, дополнение множества и др.

Очевидно, что обычный одномерный массив можно использовать в качестве внутренней реализации множества, однако такая реализация может оказаться неэффективной, поскольку либо многие типичные для множества операции будут выполняться слишком долго. В программистской практике, однако, чаще всего встречаются не множества общего вида, а такие множества, которые могут быть сведены к множеству целых числе из некоторого диапазона (обычно не слишком большого). В этом случае наиболее эффективная реализация получается, если хранить не сами элементы множества, а только информацию о принадлежности данного элемента множеству. В основе такой реализации обычно лежит использование так называемой шкалы битов. Каждый бит и определяет, принадлежит элемент множеству или нет. Общее число битов в шкале равно числу всех возможных элементов множества (то есть мощности универсального множества для данной предметной области).

Средствами языка C++ можно представить битовую шкалу посредством массива слов одинаковой разрядности (в рассматриваемом ниже примере используются 16-разрядные слова, соответствующие типу `unsigned short`). Тогда для доступа к биту номер *n* следует сначала определить индекс слова (в нашем случае – это *n/16*), а затем определить конкретный бит в слове, вычислив остаток от деления на *16*.(рис. 1.9).

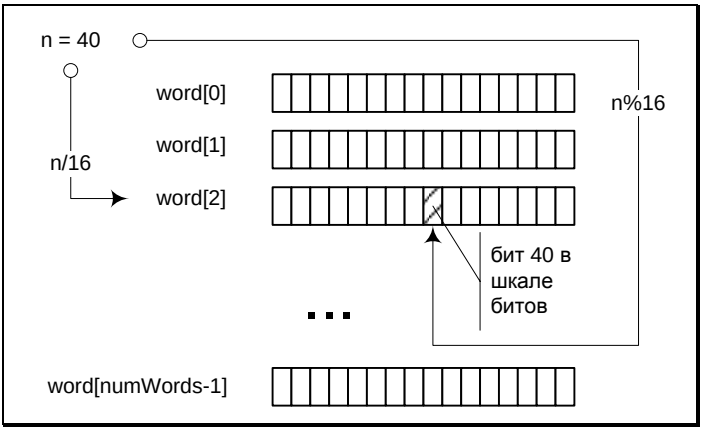


Рис. 1.9. Шкала битов

Соответствующие действия эффективно реализуются посредством операций поразрядного сдвига. Определив местоположение искомого бита, можно реализовать две базовые операции – добавление элемента (используется операция поразрядного ИЛИ) и проверка принадлежности элемента множеству (используется операция поразрядного И). Возможную реализацию шкалы битов заданного размера представляет листинг 1.4.

Листинг 1.4. Определение шкалы битов

```
/*
 * INCLUDE-файл   : BitScale.h
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Определение класса "шкала битов"
 *
 * Дата создания     : 22.09.2006
 * Дата корректировки:
 */
#ifndef _BitScale_h_
#define _BitScale_h_

class BitScale
{
protected:
     // Базовый тип "слово шкалы битов"
     typedef unsigned short Word;

     // Размер слова шкалы битов (в битах)
     static const int WordBitSize;

     Word *words;    // Массив слов
```

```
     int numWords;   // Число слов
     int numBits;    // Число битов
public:
     // Конструкторы
     BitScale( int numBits = 256 );
     BitScale( const BitScale& );

     ~BitScale();

     bool hasBit( int ixBit ); // Проверка принадлежности
     BitScale& setBit( int ixBit ); // Установка бита
     BitScale& clearBit( int ixBit ); // Очистка бита
     BitScale& inverse(); // Операция инвертирования

     // Операция занесения элемента
     // (то же , что установка бита)
     BitScale& operator |=( int ixBit )
     {
          return setBit( ixBit );
     }

     // Операция удаления элемента
     // (то же , что очистка бита)
     BitScale& operator -=( int ixBit )
     {
          return clearBit( ixBit );
     }

     // Операции над множествами
     BitScale& operator |=( const BitScale& arg ); // Объединение
     BitScale& operator &=( const BitScale& arg ); // Пересечение
     BitScale& operator -=( const BitScale& arg ); // Разность

     void Print(); // Печать содержимого шкалы битов
};

#endif

/*
 * Файл   : BitScale.cpp
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Реализация методов класса BitScale
 *
 * Дата создания     : 22.09.2006
 * Дата корректировки:
 */
#include <iostream>
#include <iomanip>
#include <stdexcept>
using namespace std;

#include <memory.h>

#include "BitScale.h"

const int BitScale::WordBitSize = 8*sizeof( BitScale::Word );

BitScale::BitScale( int numBits )
{
     this->numBits = numBits;
     numWords = (numBits+WordBitSize-1) / WordBitSize;
     words = new Word[ numWords ];
```

```
                    memset( words, 0, numWords*sizeof(Word) );
}

BitScale::BitScale( const BitScale& arg )
{
     numBits = arg.numBits;
     words = new Word[ numWords = arg.numWords ];

     memcpy( words, arg.words, numWords*sizeof(Word) );
}

BitScale::~BitScale()
{
     delete [] words;
}

void BitScale::Print()
{
     char oldFill = cout.fill( '0' );
     for( int i=numWords-1; i>=0; i-- )
     {
          cout << hex << setw( sizeof(Word)*2 );
          cout << words[i] << " ";
     }
    cout << dec << setfill( oldFill ) << endl;
}

bool BitScale::hasBit( int ixBit )
{
     if( ixBit < 0 || ixBit >= numBits ) return false;

     return     ( words[ ixBit / WordBitSize ] &
                ( 1 << ixBit % WordBitSize ) ) != 0;
}

BitScale& BitScale::setBit( int ixBit )
{
     if( ixBit < 0 || ixBit >= numBits )
          throw out_of_range( "Cannot set the bit that "
                              "is out of bit scale range" );

     words[ ixBit / WordBitSize ] |= ( 1 << ixBit % WordBitSize );
     return *this;
}

BitScale& BitScale::clearBit( int ixBit )
{
     if( ixBit < 0 || ixBit >= numBits )
          throw out_of_range( "Cannot set the bit that "
                              "is out of bit scale range" );

     words[ ixBit / WordBitSize ] &= ~( 1 << ixBit % WordBitSize
);
     return *this;
}

BitScale& BitScale::inverse()
{
     for( int i=numWords-1; i>=0; i-- )
     {
          words[ i ] = ~words[ i ];
     }
     return *this;
```

```
}

BitScale& BitScale::operator |=( const BitScale& arg )
{
     if( numBits < arg.numBits )
          throw out_of_range( "Incomparable bit scales" );
     for( int i=arg.numWords-1; i>=0; i-- )
     {
          words[ i ] |= arg.words[ i ];
     }
     return *this;
}

BitScale& BitScale::operator &=( const BitScale& arg )
{
     for( int i=numWords-1; i>=0; i-- )
     {
          words[ i ] &= arg.words[ i ];
     }
     return *this;
}

BitScale& BitScale::operator -=( const BitScale& arg )
{
     for( int i=numWords-1; i>=0; i-- )
     {
          words[ i ] &= ~arg.words[ i ];
     }
     return *this;
}
```

С использованием шкалы битов нетрудно определить класс, реализующий множество целых значений в заданном диапазоне:

```
/*
 * INCLUDE-файл   : intSet.h
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Определение класса "множество целых чисел"
 *
 * Дата создания     : 22.09.2006
 * Дата корректировки:
 */
#ifndef _IntSet_h_
#define _IntSet_h_

#include "BitScale.h"

class IntSet
{
     BitScale bitScale;
     int ixBegin;
     int ixLast;
public:
     IntSet( int ixBegin = 0, int ixLast = 255 ) :
          bitScale( ixLast-ixBegin+1 )
     {
          if( ixBegin < ixLast )
          {
               this->ixBegin = ixBegin;
               this->ixLast = ixLast;
          }
          else
```

```
                    {
                        this->ixLast = ixBegin;
                this->ixBegin = ixLast;
            }
        }

        IntSet( const IntSet& arg ) : bitScale( arg.bitScale )
        {
            ixBegin = arg.ixBegin;
            ixLast = arg.ixLast;
        }

        // Проверка принадлежности элемента множеству
        bool in( int item )
        {
            return bitScale.hasBit( item - ixBegin );
        }

        IntSet& operator |=( int item )
        {
            bitScale.setBit( item - ixBegin );
            return *this;
        }

        IntSet& operator -=( int item )
        {
            bitScale.clearBit( item - ixBegin );
            return *this;
        }

        IntSet& operator |=( const IntSet& arg )
        {
            bitScale |= arg.bitScale;
            return *this;

        }

        // Другие операции
        // ...
};

#endif
```

1.4. Алгоритмы и способы их записи

Понятие алгоритма является одним из основополагающих понятий информатики. В начале изучения информатики и основ программирования обычно достаточно руководствоваться определением алгоритма «в интуитивном смысле», например, таким: алгоритм – это однозначно определенная на некотором языке конечная последовательность предписаний (инструкций, команд), задающая порядок исполнения элементарных операций для систематического решения задачи.

Неформальный характер этого определения связан с тем, что некоторые его элементы, в свою очередь, формально не определены. С формальными моделями алгоритмов и основными положениями теории алгоритмов читатель может ознакомиться в соответствующей литературе, например, в учебном пособии [Карпов, 2003]. Здесь выделим одно важное положение, заключающееся в том, что любое разумное определение

алгоритма, которое может быть предложено в будущем, окажется эквивалентным уже известным определениям, что означает, по сути, предположение об адекватности понятий алгоритма в интуитивном смысле и алгоритма в точном смысле в одном из существующих эквивалентных формализмов.

Каждая операция предполагает наличие объекта (или объектов) данных, к которому (или к которым) эта операция применяется. Выполнение каждой операции приводит к изменению состояния исполняющего устройства, что и позволяет судить о том, что операция выполнена. Систематизируя это описание, В.Ф. Мелехин выделяет 7 параметров, характеризующих алгоритм:

- Совокупность возможных исходных данных.
- Совокупность возможных результатов.
- Совокупность возможных промежуточных результатов.
- Правило начала процесса обработки данных.
- Правило непосредственной обработки.
- Правило окончания обработки.
- Правило извлечения результата.

Алгоритм должен обладать свойствами массовости, то есть возможности многократного применения для любых наборов исходных данных из совокупности возможных, и результативности, то есть получения результата из совокупности возможных за конечное число шагов.

Существует множество способов записи алгоритма. Рассмотрим наиболее распространенные способы записи алгоритмов, используемые в проектной практике, на примере алгоритма нахождения наибольшего общего делителя двух чисел X и Y.

Текстуальная форма записи

1. Скопировать значение X во вспомогательную переменную x.
2. Скопировать значение Y во вспомогательную переменную y.
3. Если $x \neq y$, перейти к п. 4, иначе – к п. 7.
4. Если $x > y$, перейти к п. 5, иначе – к п. 6.
5. Записать в x результат вычисления выражения $x-y$ и перейти к п. 3.
6. Записать в y результат вычисления выражения $y-x$ и перейти к п. 3.
7. Конец. x – результат работы.

На начальном этапе проектирования текстуальная запись алгоритма, иногда бывает полезной как наименее формализованная, однако даже для рассмотренного несложного алгоритма уже очевидно, что она не обеспечивает достаточную наглядность решения.

Схема алгоритма

Схемы алгоритмов (flowcharts) являются визуальным формализмом, используемым уже в течение десятилетий. Правила изображения схем

алгоритмов регулируются различными международными и национальными стандартами, в том числе международным стандартом ISO 5807-85 и российским [ГОСТ 19.701-90]. Схема алгоритма решения задачи поиска наибольшего общего делителя представлена на рис. 1.10.

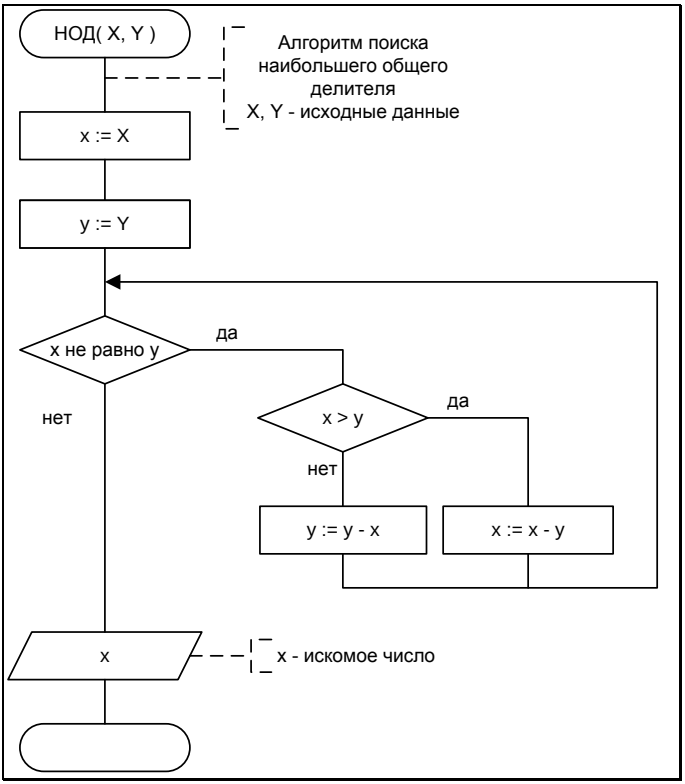


Рис. 1.10. Схема алгоритма

В монографии [Лекарев, 1997] отмечается, что если для простой задачи схемы алгоритмов обеспечивают безусловную наглядность, то по мере роста сложности программы «его логическая структура начинает «тонуть» в «клубке спагетти», в который постепенно превращается схема алгоритма». Еще в первом издании книги [Brooks, 1995], вышедшей в свет в 1975 году, Ф. Брукс замечает, что «пошаговая блок-схема является досадным анахронизмом, пригодным только для новичков в алгоритмическом мышлении», и большинство программистов, если это необходимо, рисуют схему алгоритма на основе уже законченной программы.

Псевдокод

Во многих случаях эффективной формой текстуальной записи алгоритма, абстрагированной от конкретного языка, является псевдокод:

```
НОД( X, Y )
```

```
    x:=X;
    y:=Y;
    пока( x ≠ y ) повторять
        если( x > y ) то x:=x-y;
        иначе           y:=y-x;
    конец цикла
    вывести x
конец
```

Довольно часто при написании программ на конкретном языке программирования, разработчики используют элементы псевдокода, когда хотят отложить окончательную запись решения на более позднее время.

Запись в форме программы на языке программирования

Наконец, алгоритм может быть записан в форме законченной программы на некотором языке программирования. Листинг 1.5 иллюстрирует возможное решение задачи в форме функции на языке C++.

Листинг 1.5. Функция вычисления наибольшего общего делителя

```
int grtCmnDivsor( int x, int y )
{
    while( x != y )
    {
        if( x > y ) x = x - y;
        else        y = y - x;
    }
    return x;
}
```

Запись алгоритмов функционирования реактивных систем

Под реактивными системами понимается особый класс систем, а именно: системы управления, взаимодействующие с внешним окружением и реагирующие на внешние события. Основным средством описания алгоритмов функционирования систем, основанных на событиях, является формализм конечных автоматов и построенные на его основе другие формализмы. Являясь простейшей моделью вычислительного устройства, конечные автоматы используются также при решении задач анализа и синтеза дискретных систем.

В основе определения конечного автомата лежит методически важное понятие **преобразователя информации**. Большинство используемых в настоящее время вычислительных машин являются цифровыми. Это означает, что для представления данных используются дискретные величины (числа). Для наглядного представления вычислительные машины обычно изображаются в виде схем, состоящих из блоков и связей между ними. С каждым блоком связаны входные данные, выходные данные и выполняемая блоком функция. Таким образом, блоки являются преобразователями информации.

Пусть преобразователь информации имеет n входов и p выходов. Иными словами, определены вектор входных сигналов $X = (x_1, x_2, ..., x_n)$ и вектор выходных сигналов $Y = (y_1, y_2, ..., y_p)$. Зависимость, реализуемая преобразователем информации, может быть функциональной или алгоритмической.

Функциональная зависимость имеет место, если значения выходных переменных полностью определяются значениями входных переменных и никак не зависят от предыстории. Реализацией функционального преобразователя информации (его иногда называют **автоматом без памяти**) является **комбинационная схема** – сеть логических элементов.

Однако результат преобразования входных сигналов в выходные может зависеть и от предыстории процесса обработки. Такие преобразователи называются **автоматами с памятью**. В этом случае выходные переменные находятся в **алгоритмической зависимости** от входных переменных, а для представления предыстории требуются дополнительные переменные $Q = (q_1, q_2, ..., q_s)$, представляющие собой внутренние состояния автомата. Если множество внутренних состояний конечно (на математическом языке это означает, что количество классов эквивалентности входных историй конечно), то такой алгоритмический преобразователь информации и называют **конечным автоматом**.

Абстрактный конечный автомат с выделенным начальным состоянием определяют как шестерка объектов <**In, Out, State, S0, FTrans, FOut**>, где:

In = { In_i }, i=1..n – входной алфавит;

Out = { Out_j }, j=1..p – выходной алфавит;

State = { $State_k$ }, k=1..s – множество состояний;

S0 – начальное состояние (элемент множества **State**);

FTrans : State×In→State – функции перехода, устанавливающие зависимость перехода автомата из одного состояния в другое от входного сигнала, например $State_{k+1}$=**FTrans**($State_k$, In_i) – функция перехода из состояния $State_k$ в состояние $State_{k+1}$ при входном сигнале In_i;

FOut : State×In→Out – функции выхода, устанавливающие зависимость выходного сигнала от текущего состояния и сигнала на входе, например, Out_j=**FOut**($State_k$, In_i) – функция выхода Out_j, формируемого при активации перехода из $State_k$ при входном сигнале In_i.

Наиболее наглядным способом представления конечного автомата является **граф переходов**. Отметим, что графы как форма представления потоков управления хорошо изучены, и для них существует большое количество алгоритмически формализованных решений. Кроме того, представление алгоритмов функционирования реактивных систем в форме графов имеет явно выраженный визуальный характер, то есть такое представление лучше воспринимается человеком.

На рис. 1.11 приведен граф переходов конечного автомата, распознающего идентификаторы, разделенные пробельными символами. Автомат имеет два конечных состояния: «успех», соответствующее символу «конец файла» во входном потоке, и «ошибка», соответствующее ситуации, если во входном потоке обнаружен некорректный символ.

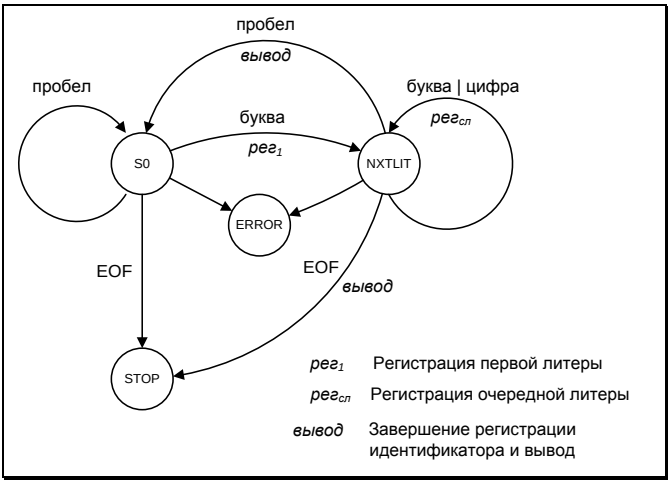


Рис. 1.11. Граф переходов конечного автомата

Автомат, определяемый таким образом, называют **автоматом Мили**. Существует также модель автомата, называемая **автоматом Мура**, отличающаяся от автомата Мили тем, что выходная функция зависит только от множества состояний, то есть **FOut : State→Out**. По вычислительной мощности автомат Мура полностью эквивалентен автомату Мили.

Главным недостатком автоматной модели является отсутствие параметров, в том числе – отсутствие понятия времени. Другими недостатками являются: отсутствие иерархии состояний, обобщения переходов, средств выражения прерываний и продолжения нормальной работы после их обработки. Кроме того, в классической модели не определена семантика взаимодействия конечных автоматов [Карпов, 2003].

Указанных недостатков лишены диаграммы состояний, предложенные Д. Харелом [Harel, 1988], а также SWITCH-технология – подход, основывающийся на программировании реактивных систем в форме сети взаимодействующих автоматов [Шалыто, 1998].

Определение конечного автомата, дополненное условием необязательного продвижения по входной цепочке, положено в основу среды разветвленного управления визуального формализма проектирования ПО на базе L-сети, предложенного, теоретически обоснованного и реализованного М.Ф. Лекаревым [Lekarev, 1993], [Лекарев, 1997].

Построение программной модели конечного автомата

Мультиветвления позволяют достаточно наглядно и просто реализовать программную модель конечного автомата. Для конечного автомата, распознающего идентификаторы, построение соответствующей программной модели иллюстрирует листинг 1.6:

Листинг 1.6. Конечный автомат, распознающий идентификаторы

```
/*
 * INCLUDE-файл   : FSM_Model.h
 *
 * Проект : FSM_Idents
 *          Console application
 *
 * Основной модуль проекта: FSM_Idents_Main.cpp
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Определение класса "Конечный автомат,
 *                                 распознающий идентификаторы"
 *
 * Дата создания     : 06.02.2006
 * Дата корректировки:
 */
#ifndef _FSM_Model_h_
#define _FSM_Model_h_

#include <stdio.h>

class FSM_Model
{
public:
     /*
      * Состояния автомата
      * (этапы анализа текста)
      */
     enum StateType
     {
          S0,       // Ожидание первой буквы (начальное состояние)
          NXTLIT,   // Ожидание очередной литеры идентификатора
          STOP,     // Конец текста (завершающее состояние)
          ERROR     // Ошибка (завершающее состояние)
     };
private:
     FILE *input;    // Входной поток конечного автомата
     FILE *output;   // Выходной поток конечного автомата

     StateType  state;     // Состояние конечного автомата
public:
     FSM_Model( FILE *input, FILE *output) : state( S0 )
     {
          this->input = input;
          this->output = output;
     }
     void Start();
private:
     /*
      * Функции, вызываемые при переходе в состояния конечного
автомата
      */
     void Process_S0( const Litera&, Lexema& );
```

```
    void Process_NXTLIT( const Litera&, Lexema& );
    void Process_STOP();
    void Process_ERROR();
};

#endif

/*
 * Файл   : FSM_Model.cpp
 *
 * Проект : FSM_Idents
 *          Console application
 *
 * Основной модуль проекта: FSM_Idents_Main.cpp
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Реализация методов класса FSM_Model
 *             (программной модели конечного автомата)
 *
 * Дата создания     : 06.02.2006
 * Дата корректировки:
 */
#include <stdio.h>
#include <stdlib.h>

#include "Litera.h"
#include "Lexema.h"
#include "FSM_Model.h"

/*
 * Реализация модели конечного автомата
 */
void FSM_Model::Start()
{
    Litera lit;   // Литера сканируемого текста
    Lexema ident; // Идентификатор в сканируемом тексте

    // Цикл работы конечного автомата
    while( 1 )
    {
        // Считать очередной символ из входного потока
        lit.GetLit( input );

        // Состояния и переходы в зависимости от
        // входного символа
        switch( state )
        {
        case S0     : Process_S0( lit, ident );
                      break;
        case NXTLIT : Process_NXTLIT( lit, ident );
                      break;
        case STOP   : Process_STOP();
                      return;
        case ERROR  : Process_ERROR();
                      return;
        }
    }
}

/*
 * Определения функций обработки состояний конечного автомата
 */
void FSM_Model::Process_S0( const Litera& lit, Lexema& ident )
```

```
        {
                    switch( lit.GetSynterm() )
    {
          case Litera::SPACE   :    state = S0;
                                    break;
          case Litera::ENDFILE :    state = STOP;
                                    break;
          case Litera::LETTER  :    ident.LexFirst( lit );
                                    state = NXTLIT;
                                    break;
        default                :    state = ERROR;
                                    break;
    }
}

void FSM_Model::Process_NXTLIT( const Litera& lit, Lexema& ident )
{
    switch( lit.GetSynterm() )
    {
        case Litera::SPACE   :ident.LexStop();
                              ident.Print( output );
                              fputc( '\n', output );
                              state = S0;
                              break;
        case Litera::ENDFILE :ident.LexStop();
                              ident.Print( output );
                              fputc( '\n', output );
                              state = STOP;
                              break;
        case Litera::LETTER  :
        case Litera::DIGIT   :ident.LexNext( lit );
                              state = NXTLIT;
                              break;
        default              :state = ERROR;
                              break;
    }
}

void FSM_Model::Process_STOP()
{
    fputs( "Stopped successfully\n", output );
}

void FSM_Model::Process_ERROR()
{
    fputs( "Stopped in error state\n", output );
}

/*
 * INCLUDE-файл   : Litera.h
 *
 * Проект : FSM_Idents
 *          Console application
 *
 * Основной модуль проекта: FSM_Idents_Main.cpp
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Определения класса "литера из входного потока"
 *
 * Дата создания     : 06.02.2006
 * Дата корректировки:
 */
#ifndef _Litera_h_
```

```
#define _Litera_h_

#include <stdio.h>
#include <ctype.h>

/*
 * Типы входных сигналов автомата
 * (синтермы литер сканируемого текста)
 */
enum InputType
{
    ENDFILE, // Конец файла
    SPACE,   // Символ пробельной группы
    LETTER,  // Латинская буква
    DIGIT,   // Десятичная цифра
    NOALP    // Запрещенный символ
};

/*
 * Тип: литера сканируемого текста
 */
class Litera
{
public:
     /*
      * Типы входных сигналов автомата
      * (синтермы литер сканируемого текста)
      */
     enum InputType
     {
          ENDFILE, // Конец файла
          SPACE,   // Символ пробельной группы
          LETTER,  // Латинская буква
          DIGIT,   // Десятичная цифра
          NOALP    // Запрещенный символ
     };
private:
    char         value;   // Значение литеры
    InputType    synterm; // Значение синтерма
public:
     /*
      * Функция считывания литеры и определения синтерма
      * (продвижение по входной цепочке конечного автомата)
      */
     InputType GetLit( FILE* input )
     {
          value = fgetc( input );
          if( value == EOF )
          {
               return synterm = ENDFILE;
          }
          else if( isspace( value ) )
          {
               return synterm = SPACE;
          }
          else if( isdigit( value ) )
          {
               return synterm = DIGIT;
          }
          else if( isalpha( value ) )
          {
               return synterm = LETTER;
          }
```

```
                        return synterm = NOALP;
                    }

     /*
      * Функция получения синтерма
      */
     InputType GetSynterm() const { return synterm; }

     friend class Lexema;
};

#endif

/*
 * INCLUDE-файл   : Lexema.h
 *
 * Проект : FSM_Idents
 *          Console application
 *
 * Основной модуль проекта: FSM_Idents_Main.cpp
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Определение класса "лексема из входного потока"
 *
 * Дата создания     : 06.02.2006
 * Дата корректировки:
 */
#ifndef _Lexema_h_
#define _Lexema_h_

#include <stdio.h>

#include "Litera.h"

/*
 * Лексема сканируемого текста
 */
class Lexema
{
    char value[ 32 ];
    char ix;
    char ixLast;
public:
     /*
      * Функции формирования лексемы исходного текста
      */

     // LEXema: register FIRST litera
     // Функция регистрации первой литеры в составе лексемы
     void LexFirst( const Litera& lit )
     {
          ix = 0;
          ixLast = 30;

          value[ 0 ] = lit.value;
     }

     // LEXema: register NEXT litera
     // Функция регистрации очередной литеры в составе лексемы
     void LexNext( const Litera& lit )
     {
          if( ix != ixLast )
          {
```

```
                value[ ++ix ] = lit.value;
            }
        }

        // LEXema: STOP register literas
        // Функция завершения регистрации литер в составе лексемы
        void LexStop()
        {
            value[ ++ix ] = '\0';
        }

        // lexema: PRINT value
        // Вывод лексемы в выходной поток
        void Print( FILE * output ) const
        {
            fputs( value, output );
        }
};

#endif

/*
 * Файл   : FSM_Idents_Main.cpp
 *
 * Проект : FSM_Idents
 *          Console application
 *
 * Проект также содержит файлы: FSM_Model.cpp, FSM_Model.h,
 *                               Litera.cpp, Litera.h,
 *                               Lexema.cpp, Lexema.h
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Демонстрационный пример.
 *             Программная модель конечного автомата,
 *             распознающего идентификаторы во входном потоке
 *
 * Темы: Функционально-иерархическая декомпозиция
 *       Формализм конечного автомата в задачах лексического и
 *       синтаксического анализа
 *
 * Дата создания     : 10.07.2005
 * Дата корректировки: 06.02.2006
 *
 * Исходные тексты примеров программ к курсу
 * "Структуры и алгоритмы компьютерной обработки данных"
 * Лекция 1. Раздел "Конечные автоматы"
 *
 * Copyright (C) Пышкин Евгений Валерьевич, 2005-2006
 *               Санкт-Петербургский государственный
политехнический
 *               университет,
 *               Факультет технической кибернетики,
 *               Кафедра автоматики и вычислительной техники
 */
#include <stdio.h>
#include <stdlib.h>



#include "Litera.h"
#include "Lexema.h"
#include "FSM_Model.h"
```

```
void main()
{
// Подготовка входного потока
FILE *fin = fopen( "input.txt", "rt" );
if( fin == NULL )
{
     printf( "Error while opening input file\n" );
     exit( 1 );
}

FSM_Model FiniteStateMachine( fin, stdout );
FiniteStateMachine.Start();

fclose( fin );
exit( 0 );

}
```

Отметим, что, как показано в [Лекарев, 2000-1], продвижение по входной цепочке не является в общем случае обязательным при каждом переходе конечного автомата. Если конечный автомат не осуществляет автоматического чтения при каждом переходе, то такая модель автомата может распознавать язык, не распознаваемый обычным конечным автоматом. Поэтому программная модель позволит обеспечить бóльшую общность решения, если операцию `GetLit` перенести внутрь обработчиков состояний. Соответствующую модификацию кода оставляем для самостоятельного упражнения.

Глава 2. Оценка алгоритмов, рекурсия, сортировка

В ходе данной лекции мы рассмотрим постановку и принципы решения задачи внутренней сортировки.

2.1. Постановка задачи внутренней сортировки и подходы к оценке эффективности

Во многих случаях упорядоченность данных в соответствии с некоторым критерием упрощает дальнейшую обработку данных. Например, значительная экономия времени при реализации двоичного поиска по сравнению с последовательным поиском является достаточным основанием для того, чтобы, потратив некоторое время на предварительную сортировку набора данных, в дальнейшем обеспечить существенный выигрыш при реализации алгоритмов двоичного или других видов поиска.

Наиболее важными являются алгоритмы сортировки in situ (на месте), которые обеспечивают перестановку элементов в пределах участка памяти, занимаемой сортируемой последовательностью. Под последовательностью данных при этом обычно понимается массив, располагающийся в оперативной памяти – сортировка таких массивов и называется внутренней сортировки в отличие от внешней сортировки – упорядочения данных из некоторого внешнего источника (например, файла на диске).

Как правило, сортировка данных осуществляется в порядке возрастания или убывания некоторого ключевого значения. В качестве ключей могут выступать целые или символьные данные, строки символов, наконец, отношение порядка может определяться на основе более сложных критериев – все это не мешает рассматривать особенности реализации алгоритмов сортировки, абстрагируясь до поры до времени от разнообразных вариантов ключевых данных.

Анализ алгоритма предполагает получение представление о том, сколько времени будет затрачено на решение задачи с использованием этого алгоритма. При оценке алгоритма исходят из количества наиболее значимых для данного алгоритма операций, выполняемых в ходе его работы. Для алгоритмов сортировки такими операциями являются операции сравнения ключей и операции пересылки элементов сортируемой последовательности.

При оценке эффективности алгоритма обычно стараются оценить три варианта: наилучший случай (когда упорядочение достигается за наименьшее время), наихудший случай (когда упорядочение достигается за максимальное время) и средний случай, анализ которого и является обычно самым сложным. Основными показателями алгоритмов сортировки является среднее число сравнений и пересылок, осуществляемых в ходе работы алгоритма.

При этом точное знание количества операций, выполняемых алгоритмом, не очень существенно с точки зрения анализа эффективности. Более важным является скорость роста числа операции при возрастании n – числа элементов массива. В [McConnell, 2001] приведена наглядная таблица

основных классов скоростей роста: $\log_2 n$, n, $n \cdot \log_2 n$, n^2, n^3, 2^n. Быстрорастущие функции доминируют в оценке суммарной эффективности алгоритма, поэтому, если выясняется, скажем, что сложность алгоритма представляет собой сумму линейной и квадратичной функций, то линейную функцию нужно отбросить, а скорость роста будет оцениваться как функция, сопоставимая с n^2.

С точки зрения оценки эффективности алгоритма наиболее важным является класс функций, обозначаемый как $O(f)$ (читается «о большое»), который состоит из функций, растущих не быстрее f. В этом случае f является верхней границей для класса $O(f)$. Важность этого класса сложности для оценки алгоритмов объясняется следующим обстоятельством. Если удается показать, что сложность одного алгоритма принадлежит классу «о большое» от сложности второго, то это означает, что второй алгоритм решает задачу не лучше первого.

Символ O был введен П. Бахманом в книге «Analytische Zahlentheorie» в 1892 г. (см. [Knuth, 1968]). По сути, введение класса сложности позволяет заменить в оценочных функциях знак приближенного равенства знаком равенства. Для функции $f(n)$, где $n \in \mathrm{N}$ запись $O(f(n))$ применяется для обозначения величины, точное значение которой неизвестно, а известно лишь, что величина это «не очень большая», а именно: $\exists M = const, M > 0$ (причем не обязательно целое) такое, что x_n, представляемая посредством $O(f(n))$, удовлетворяет условию $|x_n| \le M \cdot |f(n)|$. Например, для суммы ряда $1+2+...+n$ имеем: $1+2+...+n = (n+1)\frac{n}{2} = O(n^2)$, то есть $\exists M : n^2 \le M \cdot (n+1)\frac{n}{2}$ (в данном случае $M = 2$).

2.2. Сортировка простыми обменами

Алгоритм сортировки простыми обменами (или так называемая «пузырьковая сортировка») важен не столько сам по себе, сколько основа для изучения более сложных алгоритмов и инструмент для изучения способов записи алгоритмов на конкретном языке программирования.

В разных источниках приводятся разные варианты сортировки простыми обменами, полностью или почти полностью эквивалентные с точки зрения их сложности. Здесь мы опишем один из них.

Реализация на примере сортировки массива целых чисел

Реализацию функции сортировки простыми обменами для массива целых чисел иллюстрирует листинг 2.1.

Листинг 2.1. Сортировка простыми обменами

```
void BubbleSortInt( int *array, int n )
{
    for( int i = 1; i < n; i++ )
        for( int j = n-1; j>=i; j-- )
            if( array[ j ] < array[ j-1 ] ) {

                int copy = array[ j ];
                array[ j ] = array[ j-1 ];
                array[ j-1 ] = copy;
            }
}
```

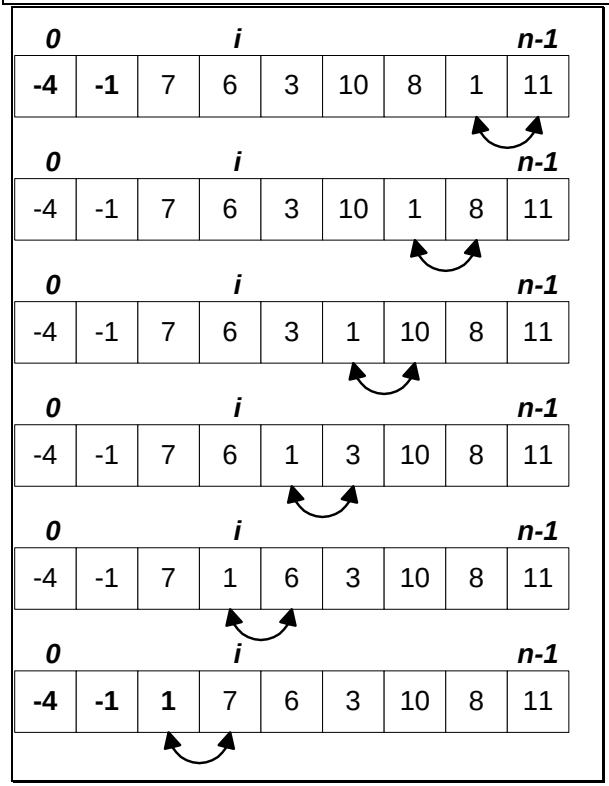


Рис. 2.1. Один просмотр в ходе пузырьковой сортировки

В ходе сортировки простыми обменами на каждом `i`-м проходе последовательно сравниваются пары соседних элементов массива, начиная с пары, составленной из элементов с индексами `n-1` и `n-2`, и заканчивая парой

элементов с индексами `i` и `i-1` (один просмотр исходной последовательности изображен на рис. 2.1). При этом `i` пробегает значения от `1` до `n-1` включительно.

Если порядок следования элементов рассматриваемой пары нарушен (в данной версии массив `array` сортируется по неубыванию), то элементы меняются местами. Не позднее чем, за `n-1` просмотров массива, он оказывается упорядоченным.

Отметим, что если при очередном просмотре не произошло ни одного обмена, процесс сортировки можно завершать. Соответствующую модификацию алгоритма иллюстрирует листинг 2.2.

Листинг 2.2. Сортировка простыми обменами (вариант с контролем наличия обменов)

```
void BubbleSortInt2( int *array, int n ) {

    bool changed = true;

    int i = 1;
    while( changed ) {

        changed = false;

        for( int j = n-1; j>=i; j-- )
            if( array[ j ] < array[ j-1 ] ) {

                int copy = array[ j ];
                array[ j ] = array[ j-1 ];
                array[ j-1 ] = copy;

                changed = true;
            }

        i++;
    }
}
```

Предварительная оценка эффективности

Количество сравнений ключей в варианте из листинга 2.1 оценить довольно просто. На первом проходе совершается $n-1$ сравнений, на втором – $n-2$ сравнений, и т. д. На последнем проходе совершается 1 сравнение. В среднем на каждом проходе совершается $\frac{(n-1)-1}{2}=\frac{n}{2}$ сравнений. Таким образом, среднее число сравнений равно $C_{cp}=(n-1)\frac{n}{2}=O(n^2)$.

Для упрощения оценки среднего числа пересылок, иногда полагают, что проверяемое условие выполняется примерно в половине случаев (см., например, [Давыдов, 2003]). В этом случае то среднее число пересылок

$M_{ср} = \frac{1}{2}(n-1)\frac{n}{2}\cdot 3 = 3\frac{n(n-1)}{4} = O(n^2)$. Если массив уже упорядочен, не совершается ни одной пересылки, то есть $M_{\min} = 0$. В наихудшем случае (когда каждое сравнение приводит к обмену) $M_{\max} = 3\frac{n(n-1)}{2}$.

Приведем анализ алгоритма в версии из листинга 2.2.

При первом проходе цикл ***for*** исполняется полностью, то есть совершается $n-1$ сравнений. Поэтому даже в наилучшем случае (массив и так упорядочен) совершается $n-1$ сравнений.

Теперь рассмотрим наихудший случай. Анализируя рис. 2.1, нетрудно убедиться, что он возникает, если элементы массива следуют в порядке, обратном требуемому.

В этом случае цикл for будет повторен $n-1$ раз. На первой итерации выполняется $n-1$ сравнений на второй итерации – $n-2$ сравнений, и т. д. На последней итерации будет выполнено 1 сравнение. Общее число сравнений в наихудшем случае равно $C_{\max} = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$. Перестановка возникает, если порядок сравниваемых элементов противоположен требуемому (а в наихудшем случае всегда). Поскольку каждая перестановка содержит 3 пересылки, общее число пересылок $M_{\max} = 3 \cdot C_{\max} = O(n^2)$.

Средний случай рассмотрим в предположении, что появление прохода без перестановки равновероятно в любом из $n-1$. проходов. Пусть C_i – число сравнений, выполненных на первых i проходах. Тогда среднее число сравнений $C_{ср} = \frac{1}{n-1}\sum_{i=1}^{n-1} C_i$. Из анализа структуры цикла ***for*** получаем:

$$C_i = \sum_{j=i}^{n-1} j = \sum_{j=1}^{n-1} j - \sum_{j=1}^{i} j = \frac{n(n-1)}{2} - \frac{i(i-1)}{2}.$$

Выполнив несложные преобразования, получим $C_{ср} = \frac{n^2}{3} - \frac{n}{6} = O(n^2)$.

Точный анализ $C_{ср}$ несколько отличается от приведенного выше, однако оценка класса сложности алгоритма как по числу сравнений, так и по числу пересылок описывается квадратичной функцией.

Отметим, что вариант из листинга 2.2 превосходит вариант из листинга 2.1 только по количеству сравнений – легко понять, что количество пересылок от введения change не меняется, а ведь пересылка обычно существенно более дорогостоящая операция чем сравнение ключей.

Улучшенная сортировка простыми обменами

Реализацию из листинга 2.2 можно еще немного улучшить, фиксируя не только факт отсутствия обменов, но и последнее значение индекса `j`, при котором был произведен обмен. В этом случае получим решение, приведенное в листинге 2.3.

Листинг 2.3. Сортировка простыми обменами с запоминанием индекса последнего участвовавшего в обмене элемента

```
template <class T>
void BetterBubbleSortPlus( T *a, int n )
{
    int ixLastChange = 0;

    int i = 1;
    while( ixLastChange < n )
    {
        lastChange = n;
        for( int j = n-1; j>=i; j-- )
        {
            if( a[ j ] < a[ j-1 ] )
            {
                T copy = a[ j ];
                a[ j ] = a[ j-1 ];
                a[ j-1 ] = copy;
                ixLastChange = j;
            }
        }
        i = lastChange + 1;
    }
}
```

Теперь приведем более строгий анализ эффективности улучшенного варианта пузырьковой сортировки. Напомним, что поскольку все улучшения касаются уменьшения числа сравнений, оценка для числа пересылок остается прежней.

В основе подхода к оценке эффективности алгоритма пузырьковой сортировки в улучшенном варианте лежит анализ так называемой таблицы инверсий. Напомним, что инверсия определяется следующим образом. Пусть $a_1 a_2 ... a_n$ – перестановка множества $\{1,2,...,n\}$. Если для $i < j$ имеет место нарушение порядка соответствующих элементов перестановки (в случае сортировки по возрастанию нарушением является $a_i > a_j$), то пара (a_i, a_j) называется инверсией. Очевидно, что перестановка, не содержащая инверсий, является упорядоченной.

Таблицей инверсий перестановки $a_1 a_2 ... a_n$ называется последовательность чисел $b_1 b_2 ... b_n$, где b_i – число элементов, бóльших i и расположенных левее i, то есть число элементов, нарушающих порядок относительно i-го элемента. Очевидно, что на каждом проходе все ненулевые элементы таблицы инверсий уменьшаются на 1, поэтому общее число проходов определяется величиной $1+\max(b_1, b_2, ..., b_n)$.

Оценим среднее число проходов $S_{\text{ср}}$ сортировки. Обозначим за $P_{S \le k}$ вероятность того, что будет совершено $S \le k$ проходов. Очевидно, что $P_{S \le k}$ является произведением $\frac{1}{n!}$ на число таблиц инверсий, не содержащих компонент, больших или равных k, то есть:

$$P_{S \le k} = \frac{1}{n!}(k^{n-k} k!)$$

Тогда вероятность того, что будет совершено ровно k просмотров, может быть вычислена следующим образом:

$$P_{S=k} = P_{S \le k} - P_{S \le (k-1)} = \frac{1}{n!}(k^{n-k} k! - (k-1)^{n-(k-1)} (k-1)!)$$

Тогда $S_{cp} = \sum_{k=0}^{n} k P_{S=k} = n - \sqrt{\pi \frac{n}{2}} + O(1)$ [Knuth, 1973]

Для среднего числа сравнений Кнут приводит следующую оценку: $C_{cp} = \frac{1}{2}(\mathrm{n}^2 - n \ln n - (\gamma + \ln 2 - 1)n) + O(\sqrt{n})$, где γ – константа Эйлера ($\gamma \approx 0{,}5772...$).

Пузырьковая сортировка является, таким образом, не слишком эффективным алгоритмом. Несмотря на это, мы воспользуемся простейшим вариантом из листинга 2.1 для обсуждения некоторых важных для практики аспектов реализации алгоритмов сортировки.

Обобщение решения с использованием функций обратного вызова

Функция обратного вызова (callback function) – это функция, вызываемая не по имени, а с использованием указателя на функцию, хранящего адрес местоположения функции (рис. 2.2).

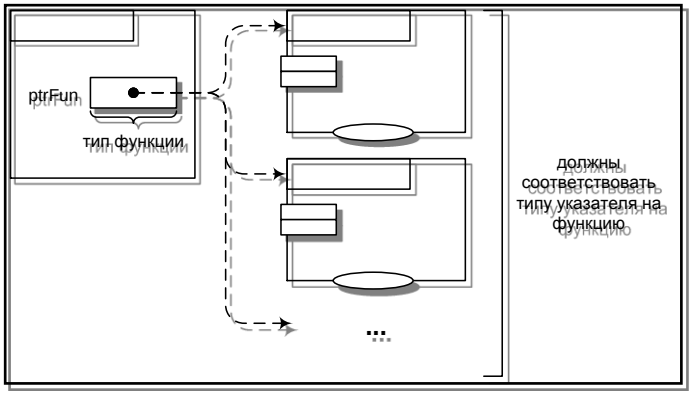


Рис. 2.2. Функции обратного вызова

Чтобы иметь возможность воспользоваться алгоритмом сортировки для обработки массивов, состоящих из данных других типов, нужно определить две вещи:

- тип функции сравнения элементов пользовательского типа;
- функцию сортировки, использующую эту функцию сравнения.

В [Пышкин, 2005] приведен пример реализации простейшего алгоритма сортировки простыми обменами таким образом, чтобы посредством использования этой функции можно было сортировать данные произвольных типов (листинг 2.4).

Листинг 2.4. Функции обратного вызова

```
// Определение типа функции, сравнивающей два значения
// пользовательского типа
typedef int (*CompareFunctionType)( const void* pArg1,
                                    const void* pArg2 );
    // Функция должна возвращать:
    //    отрицательное значение,
    //           если значение *pArg1 меньше значения *pArg2;
    //    положительное значение,
    //           если значение *pArg1 больше значения *pArg2;
    //    0,     если значения *pArg1 и *pArg2 равны.

// Обобщенная реализация функции сортировки простыми обменами
void BubbleSortGeneric(
    void *begin, // Адрес начала сортируемого массива
                 // (любой указатель может быть присвоен
                 //  указателю на void)
    int n,       // Число элементов сортируемого массива
    int elemSize,// Размер одного элемента в байтах
    CompareFunctionType compare )
                 // Указатель на функцию сравнения
{

    // Преобразование указателя на начало массива
    // к типу char*
    // NB! Необходимо, так как для указателя на void
    //     не разрешены арифметические операции
    char *base = static_cast <char *> ( begin );
```

```
    for( int i = 1; i < n; i++ )
    {
        for( int j = n-1; j>=i; j-- )
        {
            char *basej = base+j*elemSize;          // base[ j ]
            char *basejmin1 = base+(j-1)*elemSize; // base[ j-1 ]

            if( compare( basej, basejmin1 ) < 0 )
            {
                // Меняем местами base[ j ] и base[ j-1 ]
                // (переставляем байты)
                for( int curByte = 0;
                         curByte < elemSize;
                         curByte++ )
                {
                    char byteCopy        = basej[ curByte ];
                    basej[ curByte ]     = basejmin1[ curByte ];
                    basejmin1[ curByte ] = byteCopy;
                }
            }
        }
    }
}
```

Как явствует из приведенного текста программы, параметр `compare` функции `BubbleSortGeneric` является указателем на функцию, то есть принимает в качестве аргумента адрес функции, определяющей семантику сравнения двух элементов пользовательского типа (рис. 2.3).

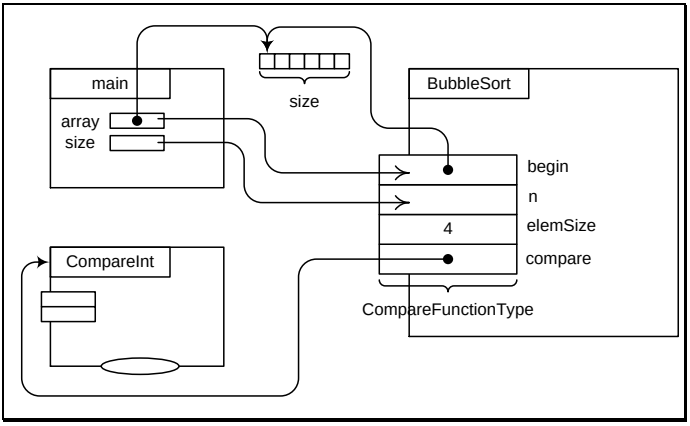


Рис. 2.3. Использование функции обратного вызова в реализации обобщенного алгоритма сортировки

Например, для сортировки целых чисел нужно определить функцию сравнения, список параметров и возвращаемое значение которой в точности соответствует типу `CompareFunctionType`:

```
int CompareInt( const void* px, const void* py )
{
    int difference = *((const int*) px) - *((const int*) py);
    return difference;
}
```

Вызов функции `BubbleSortGeneric` для сортировки некоторого массива `array`, содержащего `size` элементов целого типа, будет выглядеть следующим образом:

```
BubbleSortGeneric( array, size, sizeof( int ), CompareInt );
```

Совместную работу функций `BubbleSort` и `CompareInt` иллюстрирует рис. 2.4. Для сортировки массива, состоящего из элементов другого типа, следует определить соответствующий этому типу вариант функции сравнения.

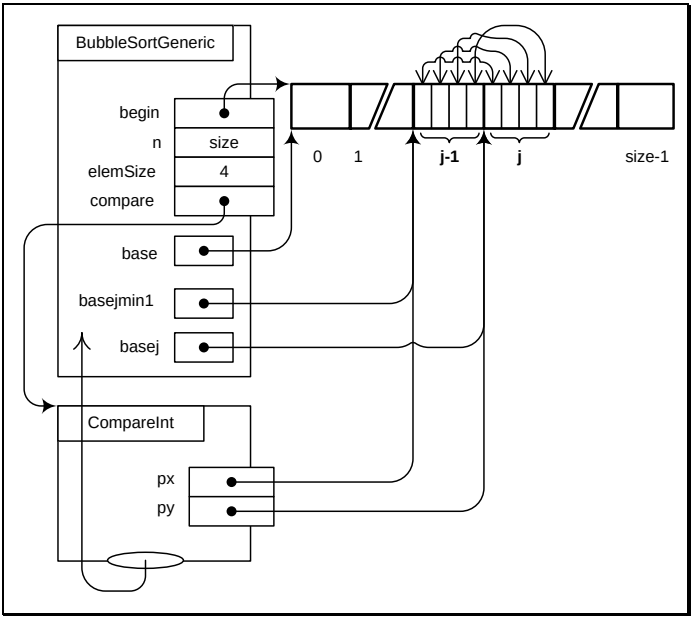


Рис. 2.4. Обобщенная сортировка простыми обменами

Приведенное здесь решение имеет существенное ограничение, состоящее в том, что для объектов сложных типов, содержащих ссылки на внешние ресурсы (например, указатели на участки выделенной динамической памяти), семантика побайтного копирования объектов, используемая функцией `BubbleSortGeneric` при обмене элементов массива, может оказаться неподходящей. Поэтому решение в общем случае должно содержать и вызов функции, отвечающей за реализацию операции обмена объектов (см. листинг 2.5).

Листинг 2.5. Исправленная семантика обмена объектов данных

```
/* Файл BubbleSort.h
 * Определение необходимых типов функций обратного вызова и
 * объявление обобщенной функции сортировки
 */
#ifndef _BubbleSort_h_
#define _BubbleSort_h_
```

```
// Определение типа функции, сравнивающей два значения
// пользовательского типа
typedef int (*CompareFunctionType)( const void* pArg1,
                                    const void* pArg2 );

// Определение типа функции, осуществляющей обмен двух объектов
// пользовательского типа
typedef void (*SwapFunctionType)( void* pArg1,
                                  void* pArg2 );

// Объявление обобщенной функции сортировки простыми обменами
void BubbleSortGeneric(
    void *begin, // Адрес начала сортируемого массива
                 // (любой указатель может быть присвоен
                 //  указателю на void)
    int n,       // Число элементов сортируемого массива
    int elemSize,// Размер одного элемента в байтах
    CompareFunctionType compare,  // Указатель на функцию
сравнения
    SwapFunctionType swap = 0 );  // Указатель на функцию обмена
                                  //    Если не задан,
обеспечивается
                                  //    побайтное копирование
#endif

/* Файл BubbleSort.cpp
 * Реализация
 */
void BubbleSortGeneric( void *begin, int n, int elemSize,
                        CompareFunctionType compare,
                        SwapFunctionType swap )
{
    // Преобразование указателя на начало массива
    // к типу char*
    // NB! Необходимо, так как для указателя на void
    //     не разрешены арифметические операции
    char *base = static_cast <char *> ( begin );

    for( int i = 1; i < n; i++ )
    {
        for( int j = n-1; j>=i; j-- )
        {
            char *basej = base+j*elemSize;         // base[ j ]
            char *basejmin1 = base+(j-1)*elemSize; // base[ j-1 ]

            if( compare( basej, basejmin1 ) < 0 )
            {
                if( swap != 0 ) swap( basej, basejmin1 );
                else
                {
                    // Меняем местами base[ j ] и base[ j-1 ]
                    // (переставляем байты)
                    for( int curByte = 0;
                             curByte < elemSize;
                             curByte++ )
                    {
                        char byteCopy        = basej[ curByte ];
                        basej[ curByte ]     = basejmin1[ curByte
];
                        basejmin1[ curByte ] = byteCopy;
                    }
                } // для else
            }
```

```
            }
        }
}
```

Отметим, что подобным образом взаимодействие с внешним кодом реализовано в функции сортировки `qsort` стандартной библиотеки C/C++. При этом функция `qsort` использует алгоритм быстрой сортировки, анализируемый далее.

Реализация в виде шаблонной функции

Недостаток приведенного выше обобщенного решения (единственно возможного при программировании на языке C) заключается в отсутствии какой-либо информации о типе сортируемых элементов. Преобразование любого указателя к указателю на `void` по определению успешно, поэтому нет никакой гарантии, что на всех этапах используются одни и те же типы. Функция сравнения, требуемая для работы `BubbleSortGeneric`, не такая уж наглядная, поскольку обязана оперировать указателями на `void`. Таким образом, из объявления функции не очевидно, что она обеспечивает реализацию семантики сравнения для данных конкретного пользовательского типа.

Другим недостатком является то, что реализация функции `BubbleSortGeneric` в стиле языка C требует аккуратной работы с отдельными байтами сортируемых данных, и поэтому менее наглядна и, следовательно, существенно сложнее для восприятия, чем специализированная версия `BubbleSortInt`.

Для того чтобы определить шаблонную функцию, нужно воспользоваться специальным предложением языка C++, предваряющим это определение. В простейшем случае такое предложение выглядит следующим образом:

```
template <class T>
```

Идентификатор `T` исполняет в этой конструкции роль параметра шаблона. Слово `class` указывает на основную область применения шаблонов – определение и использование параметризуемых типом классов, однако шаблонными могут быть объявлены и отдельные функции. Шаблон может иметь несколько параметров.

Вариант шаблонной реализации функции `BubbleSort` представлен на листинге 2.6.

Листинг 2.6. Шаблонная функция сортировки простыми обменами

```
template <class T>
void BubbleSort( T *array, int n ) {

    for( int i = 1; i < n; i++ )
        for( int j = n-1; j>=i; j-- )
            if( array[ j ] < array[ j-1 ] ) {

                T copy = array[ j ];
                array[ j ] = array[ j-1 ];
                array[ j-1 ] = copy;
            }
```

```
}
```

Процесс инстанцирования шаблона иллюстрирует рис. 2.5.

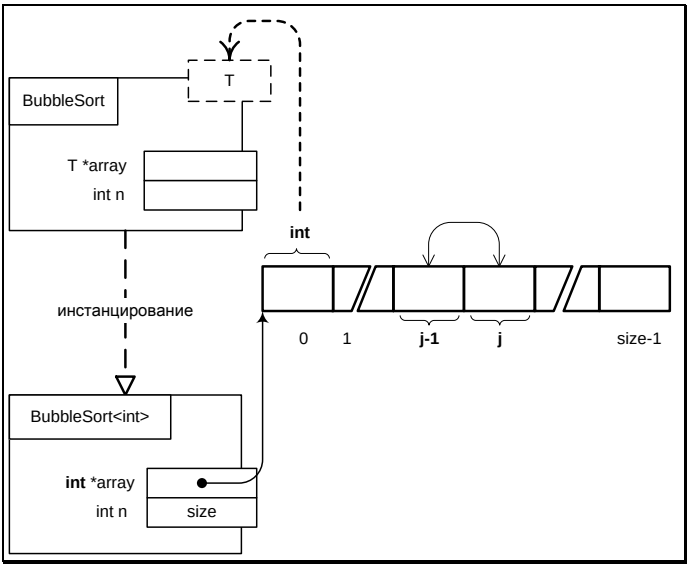


Рис. 2.5. Использование шаблонной функции сортировки

Эта версия незначительно отличается от специализированной версии для сортировки массива целых чисел (листинг 2.1). Кроме того, в ней не используется указатель на функцию, осуществляющую сравнение. Вместо этого, пользователю предлагается перегрузить операцию отношения < для работы с данными нужного типа. Разумеется, для встроенных типов этого делать не нужно, поскольку семантика сравнения данных встроенных типов известна. Таким образом, для сортировки данных целого типа эта функция может быть использована точно так же, как и функция `BubbleSortInt`:

```
void SortInt( int *a, int length ) {

    BubbleSort( a, length );

}
```

Для того чтобы воспользоваться шаблонной функцией для произвольного пользовательского типа, следует определить подходящую семантику операции сравнения, например:

```
class SportResult { // Представление результата
                    // спортивного соревнования
   int points; // Набранные очки
   int wins;   // Одержанные победы

public:
   // ...

   int operator < ( const SportResult& arg2 ) {
```

```
            // Если набранные очки совпадают, то сравниваем
          победы
      if ( points == arg2.points ) return wins < arg2.wins;

      // Основная ветвь: сравниваем набранные очки
      return points < arg2.points;
   }
};

// Пример использования функции сортировки
void SortExample( SportResult *standings, int length ) {

   BubbleSort( standings, length );

}
```

Очевидно, что не составляет никаких проблем определение семантики копирования объектов. Для этого при определении пользовательского типа следует предусмотреть реализацию перегруженной операции присваивания.

Процесс генерирования определения функции на основе информации об аргументе шаблона называется инстанцированием шаблона. Компилятор выводит типы параметров шаблона на основе типов аргументов функции при вызове. После того, как тип параметра шаблона становится известным, на основании шаблонного определения генерируется специализированная функция. Версия шаблона для конкретного аргумента называется специализацией шаблона. В последнем примере такой специализацией является функция с интерфейсом `BubbleSort( SportResult*, int )`, код которой автоматически генерируется транслятором.

Таким образом, шаблоны обеспечивают возможность генерации кода на основе параметризованных определений (поэтому иногда шаблоны называют реализацией параметрического полиморфизма, то есть полиморфизма времени компиляции). В этом кроется и недостаток шаблонных определений. Обращение к шаблонным функциям с разными типами аргументов приведет к появлению множества почти идентичных функций, код которых, разумеется, будет занимать место в памяти (чего не происходит при использовании обобщенных функций в стиле C – там работает единственный экземпляр). Риск расточительного расходования памяти является платой за те удобства, которые предоставляют шаблоны.

2.3. Рекурсивные алгоритмы

Многие понятия, используемые в математике, лингвистике, философии являются рекурсивными по своей природе. Это означает, что для определения понятия используется само это понятие. Классическим примером является определение факториала: 0! :: =1; n! ::= n*(n-1)!

Рекурсивная функция – эта такая функция, которая прямо или опосредованно вызывает сама себя. Примеры функций, непосредственно вызывающих сами себя, мы рассмотрим в данной лекции. Примеры взаимно рекурсивных функциональных модулей будут представлены позже

Задача поиска выхода из лабиринта

Одним из классических примеров рекурсивного алгоритма является решение задачи обхода лабиринта, представленное на листинге 2.7 в форме псевдокода.

Листинг 2.7. Простой алгоритм обхода лабиринта (псевдокод)

```
// Необходимые типы:
//
// MAZE_TYPE  - тип-лабиринт
// MAZE_POINT - тип-координата позиции в лабиринте
// MAZE_PATH  - путь через лабиринт

MAZE_TYPE  maze; // Определение лабиринта
MAZE_PATH  path; // Путь через лабиринт
                 // (занесение-извлечение координат по
                 //  правилам стека)

// Рекурсивный алгоритм поиска пути
bool FindMazePath( MAZE_TYPE&  maze,
                   MAZE_POINT& position )
{
    MAZE_POINT attempt; // Попытки движения в лабиринте
                        // (вправо, влево, вверх, вниз)

    // Проверяем, не посещали ли раньше
    if( AlreadyTried( maze, position ) ) return false;

    // Проверяем, не выход ли это
    if( PositionIsExit( maze, position ) )
    {
        PushToPath( path, position );
        return true;
    }

    MarkAsTried( maze, position ); // Отметить позицию как
                                   // посещенную
    PushToPath( path, position );  // Втолкнуть в стек

    // Совершаем попытки движения и рекурсивно
    // вызываем поиск пути из новой позиции
    if( MoveLeft( maze, position, attempt ) )
    {
        if( FindMazePath( maze, attempt ) ) return true;
    }

    if( MoveRight ( maze, position, attempt ) )
    {
        if( FindMazePath( maze, attempt ) ) return true;
    }

    if( MoveUp( maze, position, attempt ) )
    {
        if( FindMazePath( maze, attempt ) ) return true;
    }

    if( MoveDown ( maze, position, attempt ) )
    {
        if( FindMazePath( maze, attempt ) ) return true;
    }
    // Здесь: ни одна попытка не увенчалась успехом,
    // следовательно, из этой позиции пути нет
    PopFromPath( path ); // Исключить из стека
```

```
    return false;
}
```

В процессе поиска пути функция `FindMazePath` рекурсивно вызывает сама себя до тех пор, пока не будет выполнено одно из условий – выход из лабиринта обнаружен или рассмотрены все возможные пути. Процесс поиска пути из лабиринта в соответствии с приведенным алгоритмом иллюстрирует рис. 2.6.

При написании рекурсивного алгоритма следует учитывать следующее:

- Убедитесь, что рекурсия останавливается (особую внимательность следует проявлять при написании взаимно-рекурсивных функций, если уж их никак не избежать).
- Оцените максимальную глубину рекурсии и вытекающие из этого требования к системному стеку. Сопоставьте полученные результаты с затратами на возможную реализацию алгоритма в итерационной форме.
- Не используйте рекурсию для тривиальных алгоритмов.

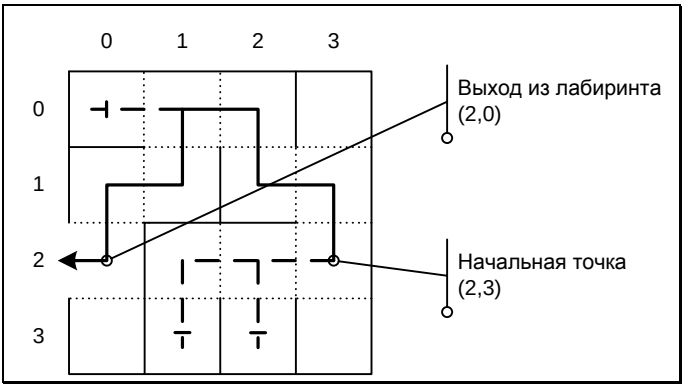


Рис. 2.6. Поиск пути по алгоритму FindMazePath

Не все то, что определяется рекурсивно, следует рекурсивно реализовывать. Например, задача вычисления факториала прекрасно решается чисто итерационной процедурой. В принципе, любой рекурсивный алгоритм может быть преобразован к итерационной форме. В простейших случаях это преобразование «бесплатно» (как, например, в случае факториала или вычисления чисел Фибоначчи). Однако в большинстве практических случаев такое преобразование достигается за счет определения вспомогательных структур данных, заменяющих стек рекурсивных вызовов, поэтому при выборе варианта решения следует сопоставить расходы на организацию рекурсивных вычислений (главным образом, связанные с затратами на хранение информации в стеке приложения) с расходами на организацию вспомогательных структур данных.

Исследуем преобразование рекурсивного алгоритма к итерационному на примере функции быстрой сортировки массива, хранящегося в оперативной

памяти, или так называемой сортировки Хоара [Dahl, Dijkstra, Hoare, 1972]. Для простоты изложения рассмотрим сортировку массива целых чисел.

Быстрая сортировка Хоара (рекурсивный вариант)

Быстрая сортировка является представителем класса обменных сортировок (простейшим примером обменной сортировки является представленный выше алгоритм сортировки простыми обменами). Основная проблема алгоритма сортировки простыми обменами заключается в том, что осуществляются обмены соседних элементов. Поэтому наиболее «неудачные» элементы достигают своего окончательного места очень медленно, что и определяет низкую эффективность сортировки простыми обменами, оцениваемую величиной $O(n^2)$ (по числу сравнений и пересылок). Поэтому возникает идея увеличить расстояния, на которых происходят обмены.

Эта идея была реализована Ч. Хоаром в виде следующего алгоритма. В основе алгоритма лежат разделение последовательности элементов на две подпоследовательности относительно значения одного из элементов (который называется разделяющим элементом). Очевидно, что такое разделение может быть достигнуто однократным просмотром последовательности (см. рис. 2.7).

Если обозначить за *x* значение разделяющего элемента массива *a*, состоящего из *n* элементов целого типа, фрагмент программы, осуществляющий соответствующее разделение, выглядит следующим образом:

Листинг 2.8. Разделение сегмента

```
int ixLeft;  // Индекс просмотра элементов массива слева направо
int ixRight; // Индекс просмотра элементов массива справа налево
int copy;    // Вспомогательная переменная для перестановки

ixLeft = 0;
ixRight = n-1;

while( ixLeft <= ixRight )
{
    while( a[ ixLeft ] < x            ) ixLeft++;
    while( x           < a[ ixRight ] ) ixRight--;

    if( ixLeft <= ixRight )
    {
        copy = a[ ixLeft ];
        a[ ixLeft ] = a[ ixRight ];
        a[ ixRight ] = copy;

        ixLeft++;
        ixRight--;
    }
}
```

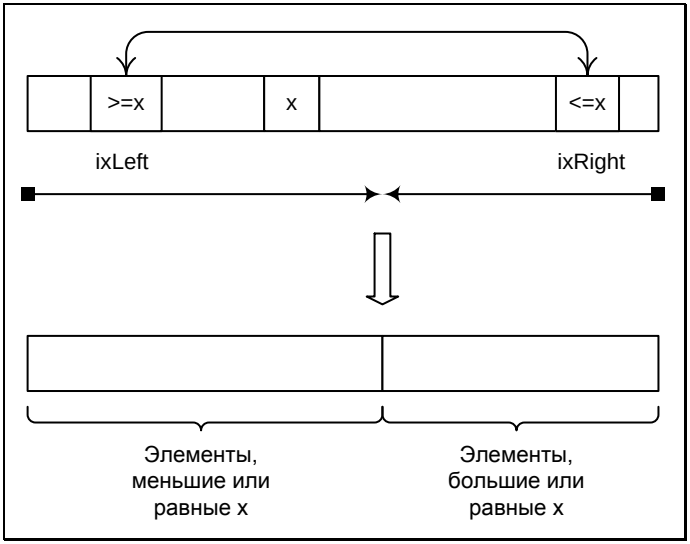


Рис. 2.7. Разделение сегмента в ходе сортировки

В результате цикла разделения в левой части массива оказываются все элементы, меньшие или равные x, а в правой части массива – все элементы, большие или равные x. Таким образом, исходная последовательность разделена на 2 подпоследовательности, которые далее можно рассматривать независимо. Иными словами, если каждую из получившихся подпоследовательностей упорядочить, упорядоченной окажется и исходная последовательность. Нетрудно заметить, что, продолжая далее разделение последовательностей, мы и получим требуемое упорядочение элементов. Соответствующий вычислительный процесс реализуется посредством рекурсивной процедуры (см. листинг 2.9). Для сортировки всего массива следует вызвать функцию `Split`, указав в качестве границ левой и правой частей сегмента индексы первого и последнего элементов исходного массива.

Листинг 2.9. Рекурсивная функция сортировки Хоара

```
// Функция разделения сегмента, ограниченного индексами
// left и right
void Split( int *a, int n, int left, int right )
{
    int ixLeft;   // Индекс при просмотре слева направо
    int ixRight;  // Индекс при просмотре справа налево

    int copy;     // Вспомогательная переменная для перестановки

    int x;        // Разделяющий элемент

    if( right <= left ) return;

    // Здесь должен быть выбран разделяющий элемент x
    // среди элементов a[ left ] ... a[ right ]
    // ...
```

```
    ixLeft  = left;
    ixRight = right;

    while( ixLeft <= ixRight )
    {
        while( a[ ixLeft ] < x            ) ixLeft++;
        while( x           < a[ ixRight ] ) ixRight--;

        if( ixLeft <= ixRight )
        {
            copy = a[ ixLeft ];
            a[ ixLeft ] = a[ ixRight ];
            a[ ixRight ] = copy;

            ixLeft++;
            ixRight--;
        }
    }
    // Здесь: получены две подпоследовательности
    //    Первая: a[ left ] ... a[ ixRight ]
    //    Вторая  a[ ixLeft ] ... a[ right ]

    // Теперь нужно разделить получившиеся подпоследовательности,
    // причем для уменьшения максимальной глубины рекурсии
    // лучше сначала разделять более короткую
    if( ixRight - left < right - ixLeft )
    {
        Split( a, n, left, ixRight );
        Split( a, n, ixLeft, right );
    }
    else
    {
        Split( a, n, ixLeft, right );
        Split( a, n, left, ixRight );
    }
}

// Функция сортировки Хоара (рекурсивный вариант)
void QuickSort( int *a, int n )
{
    Split( a, n, 0, n-1 );
}
```

Понятно, что наибольшая скорость сортировки может быть достигнута в том случае, когда при каждом очередном разделении подпоследовательность делится на 2 равные части (в этом случае исходная последовательность сортируется за $\log_2 n$ разбиений). Наихудший случай возникает тогда, когда в качестве разделяющего элемента выбирается наибольший или наименьший элемент разделяемой подпоследовательности. В этом случае результатом разделения являются две подпоследовательности, одна из которых содержит лишь 1 элемент, а вторая – все остальные элементы. Очевидно, что при таких разбиениях эффективность сортировки будет сопоставима с эффективностью сортировки простыми обменами.

К сожалению, выбор элемента *x*, приводящего к желаемому разбиению (такой желаемый элемент называется медианой подпоследовательности), в свою очередь, является трудоемкой задачей. Поэтому разделяющий элемент выбирают каким-либо простым регулярным образом, рассчитывая на то, что в

среднем мы не будем получать неравномерные разделения слишком часто. Математики показали, что наибольшая эффективность сортировки достигается при выборе разделяющего элемента случайным образом. Соответствующая модификация спроектированных функций из листинга 2.6 с использованием генератора случайных чисел стандартной библиотеки C выглядит следующим образом:

```
#include <stdlib.h>
#include <time.h>

void Split( int *a, int n, int left, int right )
{
    // ...

    // Выбор разделяющего элемента
    x = a[ left + rand() % (right-left) ];

    // ...
}

void QuickSort( int *a, int n )
{
    // "Рассеивание" генератора случайных чисел
    // на основе некоторого псевдослучайного фактора
    // (чаще всего используют текущее время)
    srand( (unsigned) time( NULL ) );

    // Вызов функции разделения сегментов
    Split( a, n, 0, n-1 );
}
```

Заметим, что согласно документации к стандартной библиотеке, функция `rand` генерирует псевдослучайное значение в диапазоне `0..0x7FFF`. При работе в рамках 32-разрядной архитектуры размер объекта данных типа `int` составляет 4 байта, а это означает, что неотрицательные значения индексов `left` и `right` находятся в диапазоне `0..0x7FFFFFFF`. Поэтому, строго говоря, выражение `rand%(right-left)` позволяет получить псевдослучайное значение в диапазоне от `0` до `min(right-left,0x7FFF)`.

Оценка эффективности быстрой сортировки

Для оценки эффективности быстрой сортировки предположим, что множество ключей представляет собор набор неповторяющихся натуральный чисел от 1 до n, то есть мы имеем ключи 1, 2, … , n. Пусть некоторый ключ $x \in \{1,2,...,n\}$ выбран в качестве разделяющего элемента. Тогда после разделения элемент с ключом x будет занимать позицию x в сортируемой последовательности. Число обменов, совершаемых в ходе разделения, равно числу элементов в левой части последовательности (их $x-1$), умноженному на вероятность того, что этот элемент следует обменять (то есть на вероятность того, что ключ не меньше, чем x), а эта вероятность определяется величиной $\frac{n-(x-1)}{n}$. Таким образом, общее число обменов

в ходе разделения равно $(x-1)\frac{n-x+1}{n}$. Усредняя по всем возможным вариантам выбора граничного ключа и учитывая, что каждый обмен содержит три пересылки, получаем:

$$M_{разд} = 3 \cdot \frac{1}{n} \sum_{x=1}^{n} (x-1) \frac{n-x+1}{n} = \frac{n}{2} - \frac{1}{2n} \approx \frac{n}{2}$$

Если нам очень везет, и мы всегда выбираем в качестве граничного элемента медиану, то каждое разделение приводит к разбиению последовательности на 2 равные части или части, отличающиеся не более чем на 1 элемент. Тогда число разделений, необходимых для завершения сортировки равно $\log_2 n$, следовательно, в наилучшем случае: $C_{\min} = n \log_2 n$, $M_{\min} = \frac{n}{2} \log_2 n$.

Разумеется, на самом деле мы не всегда будем попадать на медиану (вероятность этого всего лишь $\frac{1}{n}$). Существуют оценки, в соответствии с которыми для случая, когда разделяющий элемент выбирается случайным образом, эффективность хуже оптимальной лишь в $2 \ln 2$ раз.

Теперь оценим наихудший случай, который возникает, когда в качестве разделяющего элемента выбирается наименьший или наибольший элемент разделяемой последовательности (вероятность этого $\frac{2}{n}$). Разделение последовательности в этом случае приводит к получению двух частей: одна содержит один элемент, а другая – все остальные. Поэтому вместо $\log_2 n$ разбиений требуется n разбиений, а суммарная скорость равна $O(n^2)$, так что при неблагоприятном выборе разделяющих элементов скорость работы быстрой сортировки может оказаться сопоставимой со скоростью пузырьковой сортировки.

Реализация быстрой сортировки в итерационной форме

Каждый рекурсивный вызов приводит к созданию очередного кадра стека, в который, в частности, помещаются параметры и локальные переменные функции `QuickSort`. Поэтому даже при относительно небольших значениях *n* стек приложения может переполниться. Таким образом, постановка задачи избавления от рекурсии является в данном случае обоснованной.

Анализируя структуру данных, используемых в функции `QuickSort`, можно заметить, что основная информация в связи с организацией рекурсии относится к переменным `left` и `right`, обозначающим границы разделяемого сегмента. Если вместо использования стека приложения создать вспомогательный стек запросов на разделение (стек отложенных сегментов), то рекурсивного определения можно избежать.

Для этого определим вспомогательный класс `QuickSortStack`, содержащий структуру `StackItem`, представляющую собой описание отдельного запроса на разделение (пару индексов `left` и `right`), а также методы для занесения очередного запроса в стек и извлечения из него. Эти функции очень просты, поэтому их разумно определить как встроенные (inline) во избежание потерь эффективности, связанных с организацией вызова функций:

```
class QuickSortStack // Стек отложенных сегментов
{
    struct StackItem // Границы сегмента
    {
        int left;
        int right;
    };
public:
    QuickSortStack( int _size )
    {
        size = _size;
        body = new StackItem[ size ];

        top = -1;
    }

    ~QuickSortStack()
    {
        delete [] body;
    }

    // Занесение очередного запроса на разделение в стек
    void push( int left, int right )
    {
        if( (right-left)>=1 )
        {
            top++;
            body[ top ].left = left;
            body[ top ].right = right;
        }
    }

    // Извлечение запроса на разделение из стека
    void pop( int& left, int& right )
    {
        left = body[ top ].left;
        right = body[ top ].right;
        top--;
    }

    // Проверка: стека пуст
    int isEmpty()
    {
        return top == -1; // Больше нет запросов на разделение
    }
```

```
private:
    StackItem *body; // Адрес начала области в памяти,
                     // где хранится массив с элементами стека
    int size;        // Размер массива body
    int top;         // Вершина стека
};
```

Тогда схематичная реализация функции сортировки в нерекурсивной форме в первом приближении выглядит следующим образом:

```
#include <stdlib.h>
#include <time.h>

void QuickSort2( int *a, int n )
{
    int left;  // Левая граница разделяемого сегмента
    int right; // Правая граница разделяемого сегмента

    int ixLeft;    // Индекс при просмотре слева направо
    int ixRight;   // Индекс при просмотре справа налево

    int copy;      // Вспомогательная переменная для перестановки

    int x;         // Разделяющий элемент

    QuickSortStack stack( ... ); // Стек запросов
                                 // (о размере стека запросов
                                 //  см. далее в этом разделе)

    // Инициализация генератора случайных чисел
    srand( (unsigned) time( NULL ) );

    // Первым разделяемым сегментом является массив в целом
    stack.push( 0, n-1 );

    // Разделяем, пока в стеке есть запросы
    while( !stack.isEmpty())
    {
        stack.pop( left, right );

        // Выбор разделяющего элемента
        x = a[ left + rand() % (right-left) ];

        // Разделение очередного сегмента (см. листинг 2.8)
        // ...

        // Получившиеся подпоследовательности заносим в стек
        // причем для уменьшения требуемого размера стека
        // сначала лучше помещать более длинные
        // (то есть короткие будут извлечены раньше)
        if( ixRight - left < right - ixLeft )
        {
            stack.push( ixLeft, right );
            stack.push( left, ixRight );
        }
        else
        {
            stack.push( left, ixRight );
            stack.push( ixLeft, right );
        }
    }
}
```

Критическим моментом в связи с данным алгоритмом является выделение памяти для хранения стека запросов. Следует по возможности уменьшить требования к этой памяти. Оказывается, данный алгоритм можно существенно улучшить, если помещать в стек запросов только длинные сегменты, а с короткими «разбираться на месте». Цикл обработки запросов на разделение в этом случае выглядит следующим образом:

Листинг 2.10. Обработка запросов на разделение в нерекурсивном варианте

```
    // Разделяем, пока в стеке есть запросы
    while( !stack.isEmpty() )
    {
        stack.pop( left, right );

        // Разделяем короткие сегменты, пока они есть
        while( left < right )
        {
            // Выбор разделяющего элемента
            x = a[ left + rand() % (right-left) ];

            // Разделение очередного сегмента
            ixLeft = left;
            ixRight = right;

            while( ixLeft <= ixRight )
            {
                while( a[ ixLeft ] < x            ) ixLeft++;
                while( x           < a[ ixRight ] ) ixRight--;

                if( ixLeft <= ixRight )
                {
                    copy = a[ ixLeft ];
                    a[ ixLeft ] = a[ ixRight ];
                    a[ ixRight ] = copy;

                    ixLeft++;
                    ixRight--;
                }
            }

            // В стек заносим только длинную
            // подпоследовательность,
            // с короткой разбираемся сразу, изменяя значение
            // индекса left или right
            if( ixRight - left < right - ixLeft )
            {
                stack.push( ixLeft, right );
                right = ixRight;
            }
            else
            {
                stack.push( left, ixRight );
                left = ixLeft;
            }
        } // while( left < right )
    } // while( !stack.isEmpty() )
```

В этом случае упорядоченность исходного массива достигается не более чем за $\log_2 n$ разделений. С учетом первого запроса стек запросов

должен содержать $\log_2 n + 1$ элементов. Следовательно, в текст программы должны быть внесены следующие изменения:

```
#include <math.h>

// ...

int stackSize = log10( (double) n ) / log10( 2.0 ) + 1;
QuickSortStack stack( stackSize );
```

Произвести сравнительный анализ затрат стека приложения при реализации рекурсивного алгоритма с затратами на вспомогательный стек запросов в итерационном варианте читателю предлагается самостоятельно.

2.4. Другие классические алгоритмы внутренней сортировки и их анализ

В данном разделе приводится описание некоторых известных алгоритмов внутренней сортировки в их обобщенной реализации и анализ их эффективности.

Сортировка простыми вставками

В основе сортировки простыми вставками, или включениями, лежит следующая идея, часто используемая игроками в карты. Элементы сортируемой последовательности условно разделяются на две части. В первой находятся уже упорядоченные элементы, во второй – все остальные элементы. Очевидно, что в начале первая часть состоит из одного элемента (с индексом 0). В ходе сортировки очередной элемент (в начале – элемент с индексом 1) вставляется на подходящее место уже упорядоченной части, в результате чего эта часть удлиняется на 1 элемент (см. рис. 2.8).

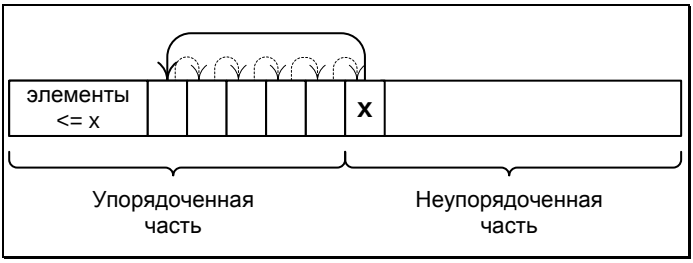


Рис. 2.8. Сортировка вставками

Очевидно, что те элементы, последовательности, которые в результате должны располагаться после вставляемого элемента, сдвигаются на один элемент.

Листинг 2.11. иллюстрирует реализацию алгоритма сортировки простыми вставками в обобщенной форме.

Листинг 2.11. Сортировка простыми вставками

```
template <class T>
void StraightInsertionSort( T *a, int n )
{
```

```
            for( int i = 1; i < n; i++ )
            {
        T x = a[ i ]; // Вставляемый элемент

        int j = i - 1;

        // Поиск подходящего места вставки
        while( j != -1  && (x < a[ j ]) )
        {
            a[ j + 1 ] = a[ j ]; // Перемещение пропускаемого
элемента
                                 // вправо
            j--;
        }

        a[ j + 1 ] = x; // Элемент заполняет освободившееся место
    }
}
```

В ходе сортировки простыми вставками осуществляется $n-1$ просмотров. Каждый i-й сравнивается минимум с одним, максимум с $i-1$ предшественником. Таким образом, среднее значение $C_{i,cp}$ лежит в интервале от 1 до $i-1$. В предположении, что перестановки исходных ключей равновероятны, получаем $C_{i,cp}=\frac{i}{2}$. Тогда для среднего числа сравнений в целом получаем следующую зависимость:

$$C_{cp}=\sum_{i=1}^{n-1}C_{i,cp}=\frac{1}{2}\sum_{i=1}^{n-1}i=\frac{(n-1)(n-2)}{4}=O(n^2).$$

Поскольку на каждом просмотре число пересылок на 2 больше, чем число сравнений, по числу пересылок также алгоритм также имеет квадратичный класс сложности:

$$M_{cp}=\sum_{i=1}^{n-1}(C_{i,cp}+2)=\frac{(n-1)(n-2)}{4}+2(n-1)=O(n^2)$$

Сортировка бинарными вставками

Алгоритм сортировки простыми вставками можно легко улучшить, если учесть возможность убыстрения поиска места вставки в упорядоченном фрагменте массива. Воспользовавшись бинарным поиском, можно уменьшить максимальное число сравнений на i-м просмотре до величины $\log_2 i$ (см. листинг 2.12).

Листинг 2.12. Сортировка бинарными вставками.

```
template <class T>
void BinaryInsertionSort( T *a, int n )
{
    for( int i = 1; i < n; i++ )
    {
        T x = a[ i ]; // Вставляемый элемент
```

```
        int left = 0;     // Левая граница участка вставки
        int right = i-1; // Правая граница участка вставки

        while( left <= right )
        {
            int j = (left+right)/2; // Индекс элемента
                                    // посередине участка вставки

            if( x < a[ j ] )   right = j-1;
            else               left  = j+1;
        }
        // Здесь: нашли место вставки (left>right)

        // Сдвигаем элементы, расположенные правее места вставки
        for( int k = i-1; k >=left; k-- )
            a[ k+1 ] = a[ k ];

        a[ left ] = x;
    }
}
```

С точки зрения количества сравнений получаем существенное улучшение:

$$C_{cp} = \mathrm{O}(n \cdot \log_2 n)$$

Однако использование бинарного поиска места вставки не позволяют существенно улучшить эффективность алгоритма в целом, поскольку на каждом просмотре все равно нужно сдвинуть примерно $\frac{i}{2}$ элементов данных, то есть M_{cp} по-прежнему остается величиной порядка n^2.

Сортировка Шелла

Д. Шелл предложил улучшение метода сортировки вставками, основывающееся на механизме, позволяющем увеличить расстояния, на которые перемещаются элементы.

Рассмотрим простейший вариант сортировки Шелла, известной также под названиями «сортировка сложными вставками» и «сортировка вставками с убывающим приращением». Пусть задана последовательность из 9 элементов (рис. 2.9, а). Исходная последовательность делится на 4 группы элементов, отстоящих друг от друга на 4 элемента (это расстояние называется дистанцией). В результате сортировки каждой группы по отдельности получаем преобразованный массив (рис. 2.9, б).

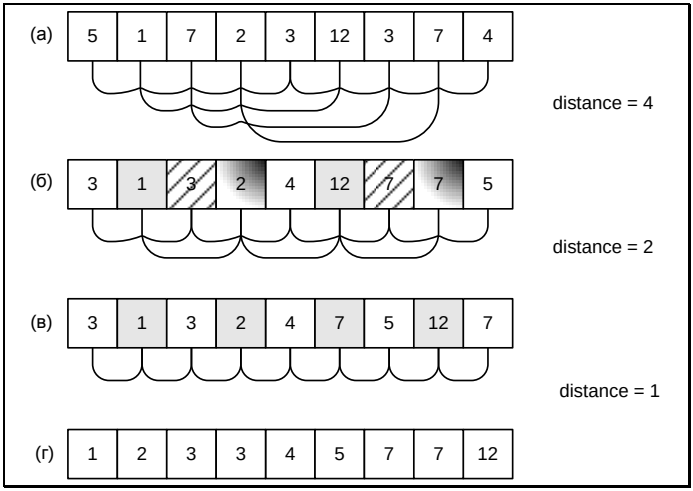


Рис. 2.9. Сортировка Шелла

Затем расстояние между элементами групп уменьшается (например, вдвое). Получаем две группы элементов, отстоящих друг от друга на 2 позиции. В результате независимой сортировки получаем результат, представленный на рис. 2.9, в.

Далее дистанция принимает значение 1, и вся последовательность в целом сортируется вставками (рис. 2.9, г). Заметим, что к этому моменту исходная последовательность близка к упорядоченной, а в этом случае эффективность алгоритма сортировки вставками является довольно высокой.

Листинг 2.13 представляет реализацию алгоритма с только что рассмотренной стратегией выбора дистанции.

Отметим, что в представляемой функции независимая сортировка групп элементов осуществляется как бы «параллельно». На выходе цикла по `i` получается «комплексный» результат: все группы отсортированы.

Листинг 2.13. Сортировка Шелла (простейший вариант)

```
template <class T>
void ShellSort( T *a, int n )
{
    int distance = n/2;

    while( distance > 0 )
    {
        // Подпоследовательности, состоящие из элементов,
        // удаленных друг от друга на distance,
        // сортируются вставками
        for( int i = distance; i < n; i++ )
        {
            T x = a[ i ];

            int j = i - distance;
```

```
            while( j >= 0  && (x < a[ j ]) )
            {
                a[ j + distance ] = a[ j ];
                j-=distance;
            }

            a[ j + distance ] = x;
        }
        distance /= 2;
    }
}
```

Стратегия выбора дистанции, использованная в программе из листинга 2.13, не является обязательной. Более того, она не является и наилучшей. Основной недостаток подобной стратегии состоит в том, что группы, образующиеся на каждом новом шаге сортировки, являются результатом «склеивания» прежних групп. Это означает, что многие элементы в действительности уже сравнивались друг с другом.

Принципиально в качестве значений дистанции можно использовать любую последовательность $d_t, d_{t-1}, ..., d_1$, удовлетворяющую условиям: $d_1 = 1$ и $d_k > d_{k-1}$. Исследования показали, что наилучшие результаты достигаются в том случае, когда последовательные значения дистанции не кратны друг другу. Например, Кнут рекомендует последовательные значения $d_{k-1} = 3 \cdot d_k + 1$, при этом $t = \lfloor \log_3 n \rfloor - 1$.

Перспективными являются также значения $d_{k-1} = 2 \cdot d_k + 1$, где $t = \lfloor \log_2 n \rfloor - 1$.

Для использования иной стратегии выбора дистанции начальный фрагмент функции сортировки следует изменить. Например, для варианта $d_{k-1} = 2 \cdot d_k + 1$ исправление может выглядеть следующим образом:

```
    int distance = 1;

    while( distance < n ) distance *=2;
    distance = distance / 2 - 1;
    if( distance <= 0 ) distance = 1;
```

Математический анализ сортировки Шелла весьма не тривиален. Кнут приводит верхнюю оценку эффективности $O(n^{1,5})$, одновременно отмечая, что анализ алгоритма Шелла и, в частности, поиск наилучшей стратегии выбора дистанции приводит к многочисленным математическим задачам, многие из которых до конца не решены.

Сортировка простым выбором

Сортировка простым выбором основана на поиске объекта данных с минимальным значением ключа с последующим обменом его с первым элементом последовательности. После этого первый элемент исключается из рассмотрения и аналогичным образом обрабатывается фрагмент из

оставшихся $n-1$ элементов. Аналогичным образом процесс продолжается и далее (см. листинг 2.14).

Листинг 2.14. Сортировка простым выбором

```
template <class T>
void StraightSelectionSort( T *a, int n )
{
    for( int i = 0; i < n-1; i++ )
    {
        T min = a[ i ];
        int ixMin = i;

        for( int k = i+1; k < n; k++ )
        {
            if( a[ k ] < min )
            {
                min = a[ k ];
                ixMin = k;
            }
        }

        a[ ixMin ] = a[ i ];
        a[ i ]     = min;
    }
}
```

Оценка среднего числа сравнений не представляет труда, поскольку оно не зависит от начального положения ключей. Внешний цикл функции `StraightSelectionSort` выполняется $n-1$ раз. На каждой итерации в среднем выполняется $\frac{n}{2}$ сравнений, поэтому общее число сравнений $C = (n-1)\frac{n}{2} = O(n^2)$.

Нетрудно оценить максимальное и минимальное значения числа пересылок.

Поскольку пересылки за пределами цикла, управляемого переменной `k`, выполняются всегда, то в случае изначальной упорядоченности данных $M_{\min} = 3(n-1)$.

В наихудшем случае (когда имеет место обратный требуемому порядок) на первой итерации, управляемого переменной `k`, выполняется $n-1$ пересылок, на второй – $n-3$ пересылок, на последней итерации происходит 1 пересылка. Суммируя эти значения и добавляя остальные «обязательные» пересылки, получаем:

$$M_{\max} = \left\lfloor \frac{n^2}{4} \right\rfloor + 3(n-1) = O(n^2)$$

Вроде бы выглядит не очень перспективно. Однако это был наихудший случай.

Оценка среднего числа пересылок представляет здесь наибольшую трудность, поскольку оно находится в довольно сложной зависимости от первоначального порядка ключей.

Среднее число пересылок на i-м проходе зависит от того, сколько раз выполнилось условие `a[k]<a[0], a[k]<a[1], … , a[k]<a[k-1]`.

Иногда для упрощения анализа алгоритма полагают, что условия выполняются примерно в половине случаев, однако это не совсем верно, поскольку из равновероятности **исходных** перестановок не следует, что на каждом просмотре элемент `x` сравнивается в среднем с половиной возможных кандидатов.

В [Knuth, 1968] показано, что число пересылок, осуществляемых в цикле по `k`, на очередной итерации цикла по `i` равно $C_{i,\text{ср}} = H_i - 1$, где величину $H_i = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i} = \sum_{k=1}^{i} \frac{1}{k}$ Кнут называет i-м гармоническим числом (часто используется приближенное значение: $H_i \approx \ln i + \gamma + \frac{1}{2n} - \frac{1}{12n^2}$).

Аппроксимируя H_i вышеприведенным значением и суммируя по i, получаем [Wirth, 1976]:

$$M_{ср} = \sum_{i=1}^{n} \ln i + \gamma + 1 = n(\gamma + 1) + \sum_{i=1}^{n} \ln i = O(n \ln n)$$

Таким образом, принимая во внимание тот факт, что присваивания объектов обычно сложнее сравнения ключей, можно сделать вывод о наибольшей эффективности данного алгоритма среди простых алгоритмов внутренней сортировки.

Глава 3. Управление связанными структурами данных

Многие используемые на практике структуры данных представляют собой не просто объект, содержащий объекты других типов, но объект, связанный с внешними объектами или ресурсами посредством ссылок или указателей. Правильная организация управления подобными структурами данных является важной составляющей искусства программирования вообще и структурного программирования в частности.

Массивы являются удобной формой хранения данных в том случае, когда объем данных в процессе обработки заранее известен и не подвержен изменениям, а также требуется высокая эффективность доступа к данным.

Иногда структура данных в связи с моделируемой предметной областью такова, что заранее нельзя предсказать их объем, а также часто возникает необходимость в операциях удаления и добавления элементов.

Действительно, удаление элемента из массива является довольно трудоемкой задачей, причем время, требующееся на завершение этой операции, сильно зависит от положения удаляемого элемента (рис. 3.1). Еще больше проблем возникает при необходимости добавить элементы в массив. Если изначально не предусмотрено резервирование «лишней» памяти, эта операция либо требует перераспределения памяти, либо в определенных условиях (например, при работе со статическим массивом) вообще может оказаться неосуществимой.

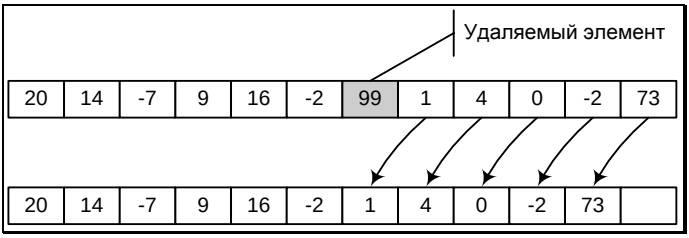


Рис. 3.1. Удаление элемента из массива

В таких случаях оправданным оказывается использование списочных структур данных, где определенная расточительность расходования памяти может окупаться простотой реализации операций добавления и извлечения элементов.

Отметим, что на практике часто встречаются своего рода гибридные решения, когда список хранит не отдельные элементы данных, а ссылки на «куски» памяти (chunks), управляемые иным образом, например, на динамические массивы (рис. 3.2). Такая организация данных позволяет совместить эффективность доступа к данным на уровне отдельного «куска» с гибкостью списочной организации.

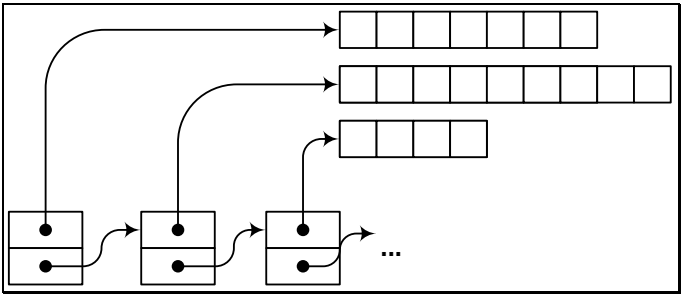


Рис. 3.2 Гибридные структуры данных

3.1. Обработка линейного однонаправленного списка: основные операции

Мы рассмотрим простейший вариант списка – линейный однонаправленный список, оставив разработку более сложных структур управления данными для самостоятельного практикума.

Постановка задачи

Линейный однонаправленный список – это такой набор элементов, где каждый отдельный элемент в явном виде хранит информацию о размещении «соседнего» элемента (следующего элемента). В C++ эта информация обычно хранится в виде указателя на следующий элемент. Последний элемент списка хранит нулевой указатель (рис. 3.3).

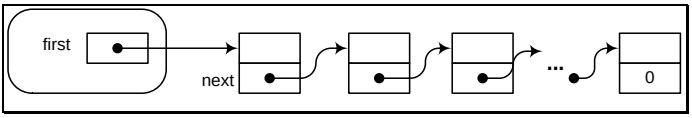


Рис. 3.3. Линейный однонаправленный список

Задача состоит в том, чтобы разработать структуры данных для представления элементов списка и самого списка, а также реализовать комплект наиболее распространенных операций над списком.

Реализация абстракции списка и базового комплекта операций

В обобщенном виде определение класса для представления линейного однонаправленного списка выглядит следующим образом (листинг 3.1):

Листинг 3.1. Элемент списка

```
template <class T>
class List
{
public:
     // Исключения в связи с выполнением
     // операций над списком
     class RemoveItemException{};   // генерируется при
                                    // невозможности
                                    // удаления элемента
     class InsertItemException{};   // генерируется при
```

```
                                                  // невозможности
                                                  // вставки
элемента

private:
     template <class ITEM>
     struct ListItem // Элемента списка
     {
          ITEM data; // Объект данных, хранящийся в списке
          ListItem<ITEM> *next; // Указатель на следующий
                                // элемент списка

          // Конструктор элемента списка
          ListItem( const ITEM& data, ListItem<ITEM>* next = 0 )
          {
               this->data = data;
               this->next = next;
          }
     };

public:
     // Конструктор списка:
     // обеспечивает корректную инициализацию first
     List()
     {
          first = 0;
     }

     // Деструктор:
     // освобождает память, занятую элементами списка
     ~List()
     {
          if( !isEmpty() ) removeAll();
     }

     // Проверка: список пуст?
     bool isEmpty()
     {
          return first == 0;
     }

     /*
      * Определение основных действий над списком
      */

     // Добавление элемента в начало списка
     void pushBegin( const T& data );
     // Добавление элемента в конец списка
     void pushEnd( const T& data );

     // Удаление элемента из начала списка
     void removeBegin();
     // Удаление всех элементов списка
     void removeAll();
     // Удаление элемента из конца списка
     void removeEnd();

     // Проверка: есть ли в списке заданный объект?
     bool hasItem( const T& dataToFind );

     // Служебный метод: печать значений объектов,
     // хранящихся в списке
     void print( ostream& out = cout );
```

```
private:
     ListItem<T> *first; // Указатель на первый элемент списка
};
```

Заполнение списка элементами осуществляется посредством операции добавления элемента в список. Наиболее просто реализуется добавление элемента в начало списка (см. рис. 3.4 и листинг 3.2).

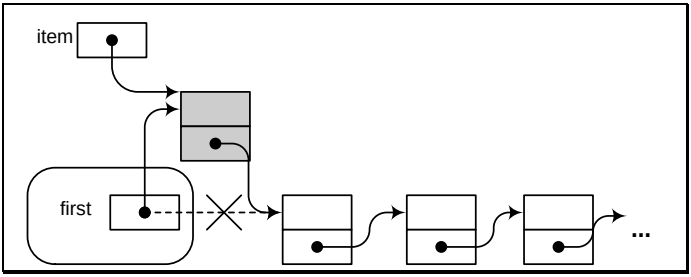


Рис. 3.4. Добавление элемента в начало списка

Листинг 3.2. Добавление элемента в начало списка

```
template <class T>
void List<T>::pushBegin( const T& data )
{
     ListItem<T> *item = new ListItem<T>( data, first );

     first = item;
}
```

Следует заметить, что согласно требованиям Standard C++, операция `new` генерирует исключение `bad_alloc`. Поэтому если необходимо воспользоваться «старым» механизмом проверки значения указателя `item`, следует применять спецификатор `nothrow`. При этом обязательно подключение заголовочного файла `new` и стандартного пространства имен:

```
#include <new>
using namespace std;
```

В некоторых случаях, когда важно, чтобы порядок следования данных в списке соответствовал порядку занесения элементов, элементы должны добавляться в конец списка (рис. 3.5). Добавление элемента в конец списка состоит из трех подпроцессов: размещение нового элемента в памяти, поиск последнего элемента списка, собственно добавление элемента (листинг 3.3).

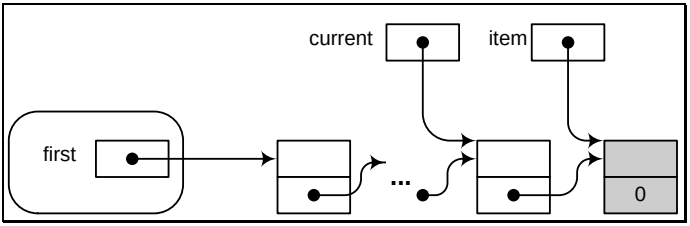


Рис. 3.5. Добавление элемента в конец списка

Листинг 3.3. Добавление элемента в конец списка

```
template <class T>
```

```
void List<T>::pushEnd( const T& data )
{
     ListItem<T> *item = new ListItem<T>( data );

     // Обработка случая, когда список пуст
     if( first == 0 )
     {
          first = item;
          return;
     }

     // Поиск последнего элемента
     ListItem<T> *current = first;
     while( current->next != 0 )
     {
          current = current->next;
     }

     // Помещение созданного элемента в конец
     current->next = item;

     return;
}
```

Уместно сразу же рассмотреть несложные в реализации операции удаления элемента из начала списка и из конца списка.

Удаление элемента из начала списка приводит к обязательной модификации указателя `list.first` (см. рис. 3.6 и листинг 3.4).

Листинг 3.4. Удаление элемента из начала списка

```
template <class T>
void List<T>::removeBegin()
{
     ListItem<T> *delItem; // Указатель на удаляемый элемент

     if( first == 0 ) // нечего удалять
     {
          throw RemoveItemException();
     }

     delItem = first; // Удаляемым элементом является первый

     first = delItem->next; // Изменяем указатель на начало
                            // списка

     // Собственно удаление
     delete delItem;
     return;
}
```

С использованием функции `RemoveBegin` легко определяется функция удаления всего списка (листинг 3.5).

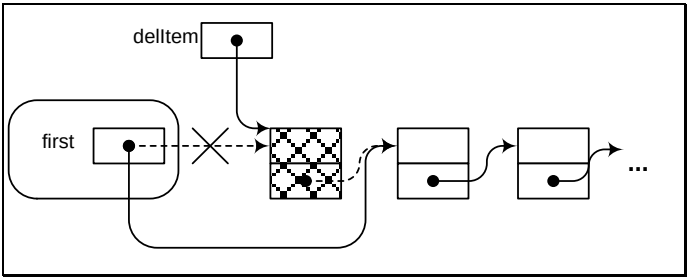


Рис. 3.6. Удаление элемента из начала списка

Листинг 3.5. Удаление списка из памяти

```
template <class T>
void List<T>::removeAll()
{
     if( first == 0 )
     {
          throw RemoveItemException();
     }

     while( first != 0 )
     {
          removeBegin();
     }
     return;
}
```

Удаление элемента из конца списка «удлиняется» на фазу просмотра списка (см. рис. 3.7 и листинг 3.6).

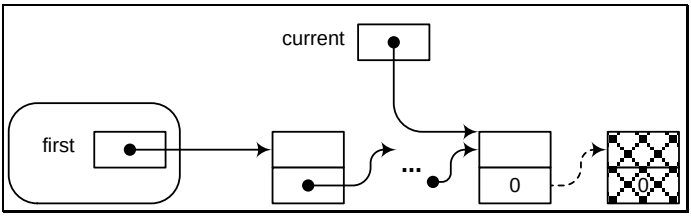


Рис. 3.7. Удаление элемента из конца списка

В процессе просмотра списка нужно найти элемент списка, непосредственно предшествующий последнему (см. листинг 3.6).

Листинг 3.6. Удаление элемента из конца списка

```
template <class T>
void List<T>::removeEnd()
{
     ListItem<T> *current; // Указатель на обозреваемый элемент

     if( first == 0 ) // нечего удалять
     {
          throw RemoveItemException();
     }

     current = first;
     if( current->next == 0 ) // Список состоит из одного элемента
```

```
                    {
                         return removeBegin();
     }

     // Цикл поиска элемента, предшествующего последнему
     while( current->next->next != 0 )
     {
          current = current->next;
     }
     // Здесь: current указывает на элемент,
     //         предшествующий последнему

     // Собственно удаление посленего элемента
     delete current->next;
     current->next = 0;

     return;
}
```

Полезной во многих случаях операцией является проверка, содержится ли заданный объект данных в списке или нет (листинг 3.7).

Листинг 3.7. Поиск элемента списка

```
template <class T>
bool List<T>::hasItem( const T& dataToFind )
{
     ListItem<T> *current;

     current = first;
     while( current != 0 )
     {
          if( current->data == dataToFind )
          {
               return true;
          }
          current = current->next;
     }

     return false;
}
```

Функция работает при условии, что для пользовательского типа определена семантика операции отношения ==. В C++ для этого следует воспользоваться синтаксисом перегрузки операций. Предположим, что элементы данных списка имеют следующую структуру:

```
struct DataBlock
{
    int value;   // Целое число
    int counter; // Частота встречаемости этого числа
                 // в наборе данных
};
```

Будем считать элементы совпадающими, если значения полей value у двух объектов данных равны. Тогда для того чтобы воспользоваться функцией `FindItem`, операцию == нужно перегрузить следующим образом:

```
int operator==( const DataBlock& arg1, const DataBlock& arg2 )
{
    return arg1.value == arg2.value;
}
```

3.2. Определение других операций над списком. Первоначальное представление об итераторах

Довольно часто возникает ситуация, когда требуется выполнить некоторые действия над объектами данных, хранящихся в списке. Самый простой вариант решения заключается в реализации соответствующих действий в виде отдельного метода класса `List`. Однако такое решение годится только для наиболее распространенных действий, имеющих смысл для разных списков.

В общем случае необходима такая организация вычислительного процесса, при которой часть действий по обработке объекта данных, хранящегося в списке, происходит за пределами класса `List`. Одно из возможных решений состоит в определении абстрактного класса, один из методов которого является чисто виртуальной функцией:

```
template <class T>
class SimpleVisitor
{
public:
     virtual void process( T& data ) = 0;
};
```

Далее определяется метод класса `List`, реализующий просмотр элементов списка и для каждого объекта данных вызывающий метод `process`, вызываемый с использованием ссылки на объект производного от `SimpleVisitor` класса:

```
template <class T>
class ListIt
{
     // ...
public:
     void iterate( SimpleVisitor<T>& visit )
     {
          ListItem<T> *current = first;

          while( current != 0 )
          {
               visit.process( current->data );
               current = current->next;
          }
     }
private:
     ListItem<T> *first;
};
```

Рассмотрим использование данного механизма на примере вычисления суммы рациональных дробей, хранящихся в списке.

Сначала определим класс, производный от `SimpleVisitor`:

```
// Используем тип Rational, рассмотренный ранее

class SumRational : public SimpleVisitor<Rational>
{
     Rational sum;
public:
     SumRational() : sum( 0 ) {}
```

```
                    void process( Rational& data )
     {
          sum += data;
     }

     Rational getSum()
     {
          return sum;
     }
};
```

Основное действие, выполняемое в ходе итерации списка, состоит в добавлении к значению переменной `sum` значения очередного рациональной дроби. Метод `getSum` позволяет извлечь результат вычислений:

```
ListIt<Rational> list2;
list2.pushBegin( Rational( 1, 3 ) );
list2.pushEnd( Rational( 2, 3 ) );

SumRational rsum;
list2.iterate( rsum );
cout << "Sum of rationals = " << rsum.getSum() << endl;
```

Если перед итерацией элементов списка требуются какие-либо подготовительные действия, специфичные для решаемой задачи, их также можно определить в контексте класса, разрабатываемого на основе интерфейса `SimpleVisitor`. Например, вывод перенумерованных элементов списка, содержащего целочисленные значения, может быть организован следующим образом:

```
class ListPrinter : public SimpleVisitor<int>
{
     ostream& out;
     int counter;
public:
     ListPrinter( ostream& _out = cout ) :
          out( _out ), counter (0)
     {}
     void start()
     {
          out << "List:" << endl;
          counter = 0;
     }
     void process( int& data )
     {
          out << ++counter << " : " << data << endl;
     }
     void stop()
     {
          if( counter  == 0 ) out << "Empty" << endl;
     }
};
```

Ниже приводится фрагмент создания списка из пяти элементов и печати хранящихся в нем значений:

```
ListIt<int> list1;

list1.pushBegin( 1 );
list1.pushEnd( 2 );
list1.pushEnd( 3 );
list1.pushBegin( 0 );
```

```
list1.pushEnd( 4 );

ListPrinter prn;
prn.start();
list1.iterate( prn );
prn.stop();
```

Данную реализацию можно еще упростить, если исходить из того, что единственное назначение абстрактного класса `SimpleVisitor` состоит в задании интерфейса обработчика отдельного объекта данных (то есть, метода `process`). Учитывая это обстоятельство, можно, вместо использования абстрактного класса, определить метод `iterate` с параметром-ссылкой на объект, который можно интерпретировать как функцию с параметром типа `T`.

```
template <class T>
class ListIt
{
public:
     template <class Op>
     void iterate( Op& callback )
     {
          ListItem<T> *current = first;

          while( current != 0 )
          {
               callback( current->data );
               current = current->next;
          }
     }
private:
     ListItem<T> *first;
};
```

В качестве функции, инстанцирующей шаблонный метод `iterate`, может выступать как обычная функция, так и функциональный объект, то есть объект класса, в котором перегружена операция «круглые скобки», например:

```
// Функциональный класс, используемый для вычисления
// суммы целочисленных объектов, хранящихся в списке
class Summator
{
     int sum;
public:
     Summator() : sum( 0 ) {}

     void operator() ( int& value )
     {
          sum += value;
     }

     int getSum()
     {
          return sum;
     }
};

// Функциональный класс, используемый для увеличения
// каждого объекта на заданное число
class Adder
{
     int valueToAdd;
public:
```

```
                    Adder( int value ) : valueToAdd( value ) {}

     void operator() ( int& value ) const
     {
          value += valueToAdd;
     }
};

// Функция, которая может быть использована
// для увеличения каждого объекта на 10
void adderFunction( int& value )
{
     value += 10;
}

void main()
{
     ListIt<int> list1;

     list1.pushBegin( 1 );
     list1.pushEnd( 2 );
     list1.pushEnd( 3 );
     list1.pushBegin( 0 );
     list1.pushEnd( 4 );

     list1.print();

     // Суммирование элементов списка
     Summator summator;
     list1.iterate( summator );
     cout << "sum=" << summator.getSum() << endl;

     // Увеличение всех элементов на 5
     list1.iterate( Adder( 5 ) );
     list1.print();

     // Увеличение всех элементов на 10
     list1.iterate( adderFunction );
     list1.print();
}
```

С точки зрения процесса обработки списка оба приведенных варианта (и с абстрактным классом, и с шаблонным методом `iterate`) являются реализациями внутреннего итератора, то есть такого способа обхода элементов списка, который жестко определен внутри класса `List`. Отметим, что подобная организация реализует общий подход к сквозной обработке списка и, безусловно, не лишена изящества, однако следует указать и на ряд очевидных недостатков.

Недостатки просмотра списка с использованием внутреннего итератора

Первый недостаток представленного подхода состоит в том, что не всякий процесс требует просмотра всех элементов. Рассмотрим пример решения задачи поиска заданного элемента в списке, представив на минутку, что соответствующего метода нет в классе `List`.

Решение задачи может выглядеть так (приводится вариант для списка, содержащего целочисленные значения):

```
class ListFind : public SimpleVisitor<int>
{
     int valueToFind;
     bool found;
public:
     ListFind( int userValue ) : valueToFind( userValue ), found(
false ) {}

     void process( int& data )
     {
          if( data == valueToFind ) found = true;
     }

     bool isFound() { return found; }
};
```

Для поиска элемента данных в некотором списке следует создать объект класса `ListFind`, вызвать метод `iterate` на объекте списка, а затем – проанализировать результат работы, используя метод `isFound`, определенный в классе `ListFind`:

```
ListFind find( 5 ); // Ищем значение 5
list1.iterate( find );
if( find.isFound() ) cout << "Found" << endl;
else                 cout << "Not found" << endl;
```

Очевидно, что если даже первый элемент списка содержит искомое значение, просмотрены будут все элементы списка. В данном случае эту проблему можно устранить, используя механизм обработки исключений:

```
class ListFindModified : public SimpleVisitor<int>
{
     int valueToFind;
     bool found;
public:
     class ObjectFoundException {};

     ListFindModified( int userValue ) :
          valueToFind( userValue ), found( false )
     {}

     void process( int& data )
     {
          if( data == valueToFind ) throw ObjectFoundException();
     }
};
```

Тогда вместо проверки успешности поиска можно перехватить исключение `ObjectFoundException`:

```
ListFindModified find2( 4 );
try
{
     list1.iterate( find2 );
     cout << "Not found" << endl;
}
catch( ListFindModified::ObjectFoundException )
{
     cout << "Found" << endl;
}
```

Однако, во-первых, передача, в общем-то, позитивного результата поиска в виде исключения выглядит несколько неестественно. Во-вторых,

подобный трюк вряд ли претендует на общее решение. Наконец, можно привести мнение Страуструпа, осуждающего использование обработки исключений не для выявления ошибок, а в качестве альтернативного управляющего механизма [Stroustrup, 2000].

Следующий недостаток состоит в том, что для реализации новой операции приходится проделать довольно большую работу по проектированию класса, производного от `SimpleVisitor` или функционального класса.

Далее, нужно отметить, что не любая операция над списком может быть сведена к независимой последовательной обработке отдельных элементов списка (например, задача поиска повторяющихся элементов). Кубенский замечает, что трудности возникнут даже при определении такой простой операции как проверка упорядоченности списка [Кубенский, 2004] (хотя эта конкретная задача в рамках данной структуры кода все же решается – оставляем ее читателю в качестве упражнения).

Наконец, использование данного метода не позволяет одновременно обрабатывать несколько списков (например, нельзя реализовать такой естественный процесс как сравнение содержимого двух списков).

Внешнее управление работой внутреннего итератора

Частично указанные проблемы можно решить, реализовав внешнее управление работой итератора. В основе решения – определение абстрактного итератора и построение на его основе специализированной реализации итерируемого списка (см. листинг 3.8).

Листинг 3.8. Внешнее управление работой итератора

```
#ifndef _ListIt_h_
#define _ListIt_h_

#include <iostream>
using namespace std;

// Абстрактный итератор (простейший вариант)
template <class T>
class Iterator
{
public:
     virtual void start() = 0;    // Инициализирует итерацию
     virtual bool hasMore() = 0; // Проверяет, есть ли еще
                                 // элементы
     virtual void next() = 0;     // Переходит к очередному
     virtual T* get() = 0;        // Возвращает адрес элемента
};

// Специализированная версия списка
template <class T>
class ListIt : public Iterator<T>
{
public:
     class RemoveItemException{};
     class InsertItemException{};
```

```
private:
     template <class ITEM>
     struct ListItem
     {
          ITEM data;
          ListItem<ITEM> *next;

          ListItem( const ITEM& data, ListItem<ITEM>* next = 0 )
          {
               this->data = data;
               this->next = next;
          }
     };

public:

     // Методы управления списком рассмотрены ранее
     // ...

     // Методы, обеспечивающие итерацию списка
     void start()
     {
          currentIt=first;
     }

     bool hasMore()
     {
          return currentIt!=0;
     }

     void next()
     {
          if( currentIt == 0 ) throw BadIteratorException();
          currentIt = currentIt->next;
     }

     T* get()
     {
          if( currentIt == 0 ) throw BadIteratorException();
          return &(currentIt->data);
     }

private:
     ListItem<T> *first;       // Адрес первого элемента списка
     ListItem<T> *currentIt;   // Адрес текущего обозреваемого
                               // элемента списка
};

#endif
```

Процедуру использования итерируемого списка иллюстрирует следующий текст:

```
// Проверка наличия задаваемого объекта данных в списке
template <class T>
bool HasItem( ListIt<T>& list, const T& checkData )
{
     for( list.start(); list.hasMore(); list.next() )
          // Необходима точная согласованность вызовов –
          // ПЕРВЫЙ НЕДОСТАТОК
     {
          T* ptrItem = list.get();
          if( *ptrItem == checkData ) return true;
     }
```

```
        return false;

    // В каждый момент времени над списком может быть запущен
    // только ОДИН итератор - ВТОРОЙ НЕДОСТАТОК =>
    // ограничения на структуру алгоритмов
}

void main()
{
    ListIt<int> list1;

    list1.pushBegin( 1 );
    list1.pushEnd( 2 );
    list1.pushEnd( 3 );
    list1.pushBegin( 0 );
    list1.pushEnd( 4 );

    list1.print();

    if( HasItem( list1, 4 ) )
    {
        cout << "Found" << endl;
    }
}
```

В представленном фрагменте указаны два основных недостатка рассматриваемого метода, главным из которых является невозможно запуска над одним и тем же списком нескольких итераторов, поскольку информация о текущем положении итератора хранится непосредственно в списочном объекте.

Построение внешнего итератора списка

Решить указанные в предыдущем разделе проблемы можно, если информацию о текущем положении итератора хранить в самостоятельном объекте, который и создавать для осуществления итерации списка. Иными словами, нужно построить внешний итератор списка.

Поскольку итератор использует элементы внутреннего представления списка, его естественно поместить в класс списка. Итератор может допускать или не допускать возможность модификации списка. Мы рассмотрим реализацию итератора, допускающего изменение списка (см. листинг 3.9).

Листинг 3.9. Построение внешнего итератора списка

```
template <class T>
class ListIt
{
public:
    class RemoveItemException{};
    class InsertItemException{};

private:
    template <class ITEM>
    struct ListItem
    {
        ITEM data;
        ListItem<ITEM> *next;

        ListItem( const ITEM& data, ListItem<ITEM>* next = 0 )
```

```
          {
               this->data = data;
               this->next = next;
          }
     };

public:
     // Методы управления списком рассмотрены ранее
     // ...

     friend class Iterator;    // Дружественный доступ необходим,
                               // поскольку класс Iterator
                               // работает в тесной кооперации
                               // с классом ListIt

     // Определение класса итератора с возможностью
     // модификации списка
     class Iterator
     {
          ListItem<T> *currentIt;
          ListItem<T> **ptrToCurrentIt;

     public:
          Iterator( ListIt<T>* pList )
          {
               ptrToCurrentIt = &( pList->first );
               currentIt = *ptrToCurrentIt;
          }

          bool hasMore()
          {
               return currentIt!=0;
          }

          void  next()
          {
               if( currentIt==0 ) throw BadIteratorException();

               ptrToCurrentIt = &( currentIt->next );
               currentIt = *ptrToCurrentIt;
          }

          T* get()
          {
               if( currentIt==0 ) throw BadIteratorException();
               return &(currentIt->data);
          }

          // Далее определены методы, модифицирующие список
          // с использованием итератора

          // Удаление элемента списка, на который
          // указывает итератор
          void remove()
          {
               if( currentIt == 0 ) throw RemoveItemException();

               *ptrToCurrentIt = currentIt->next;
               delete currentIt;

               currentIt = *ptrToCurrentIt;
          }

   // Вставка нового элемента после элемента списка,
```

```
          // на который указывает итератор
                         void insertAfter( const T& newData )
          {
               if( currentIt == 0 ) throw InsertItemException();

               ListItem<T> *newItem =
                    new ListItem<T>( newData, currentIt->next );
               currentIt->next = newItem;

               ptrToCurrentIt = &( newItem->next );

               currentIt = *ptrToCurrentIt;
          }

          // Можно определить и другие методы
          // ...
     };

     Iterator *iterator()
     {
          return new Iterator( this );
     }
private:
     ListItem<T> *first;
};
```

Ниже следует фрагмент, иллюстрирующий варианты использования внешнего итератора для обработки и модификации списка.

```
template <class T>
bool HasItem( ListIt<T>& list, const T& checkData )
{
     ListIt<T>::Iterator *iterator = list.iterator();

     bool found = false;
     while( iterator->hasMore() )
     {
          T* ptrItem = iterator->get();
          if( *ptrItem == checkData )
          {
               found = true;
               break;
          }

          iterator->next();
     }

     delete iterator;
     return found;
}

template <class T>
bool FindAndDelete( ListIt<T>& list, const T& checkData )
{
     ListIt<T>::Iterator *iterator = list.iterator();

     bool found = false;
     while( iterator->hasMore() )
     {
          T* ptrItem = iterator->get();
          if( *ptrItem == checkData )
          {
               iterator->remove();
               found = true;
```

```
                break;
            }
            iterator->next();
        }

        delete iterator;
        return found;
}

template <class T>
bool FindAndInsert( ListIt<T>& list,
                    const T& checkData, const T& insertData )
{
        ListIt<T>::Iterator *iterator = list.iterator();

        bool found = false;
        while( iterator->hasMore() )
        {
            T* ptrItem = iterator->get();
            if( *ptrItem == checkData )
            {
                iterator->insertAfter( insertData );
                found = true;
                break;
            }
            iterator->next();
        }

        delete iterator;
        return found;
}

void main()
{
        ListIt<int> list1;

        list1.pushBegin( 1 );
        list1.pushEnd( 2 );
        list1.pushEnd( 3 );
        list1.pushBegin( 0 );
        list1.pushEnd( 4 );
        list1.print();

        if( HasItem( list1, 4 ) )
        {
            cout << "Found" << endl;
        }

        if( FindAndDelete( list1, 3 ) )
        {
            cout << "Found and deleted" << endl;
        }
        list1.print();

        if( FindAndInsert( list1, 2, 3 ) )
        {
            cout << "Found and inserted" << endl;
        }
        list1.print();
}
```

Заметим, что в одном и том же классе можно определить несколько внешних итераторов, обеспечивающих различные варианты обработки списка

(например, отдельные итераторы для просмотра списка в разных направлениях, итераторы с возможностью вставки, итераторы, допускающие изменения содержимого элементов списка и др.)

3.3. Другие виды списков

Изучение линейного однонаправленного списка позволяет получить основные представления об управлении связанными объектами данных. При этом необходимо учитывать, что на практике возможностей, предоставляемых линейным однонаправленным списком, может оказаться недостаточно.

Дополнительную гибкость списочным структурам приводит добавление новых полей, обеспечивающих связь не только с последующим элементом, но и с другими элементами списка. Например, двунаправленный список содержит указатели не только на последующий, но и на предшествующий элемент. Во многих случаях увеличение числа полей для связи используется для ускорения процесса просмотра списка, при этом одно поле может использоваться для связи со следующим элементом списка, а другое – для связи со следующим «разделом» списка.

Глава 4. Древовидные структуры и их применение

В теории графов дерево определяется как граф без циклов. Элементы теории графов обычно изучаются студентами в рамках курсов математики и информатики. Мы сосредоточимся на прикладных аспектах применения древовидных структур в программировании.

4.1. Организация древовидных структур

Как форма организации данных, дерево является связанной структурой, где с каждым элементом данных (узлом, или вершиной), кроме одного, связан один **узел-родитель** (parent node) и может быть связано произвольное число **узлов-потомков** (child nodes). Если узел не имеет потомков, он называется концевым узлом, или **листом** (leaf node). Если узел не имеет родителя, он называется **корневым узлом** (root node).

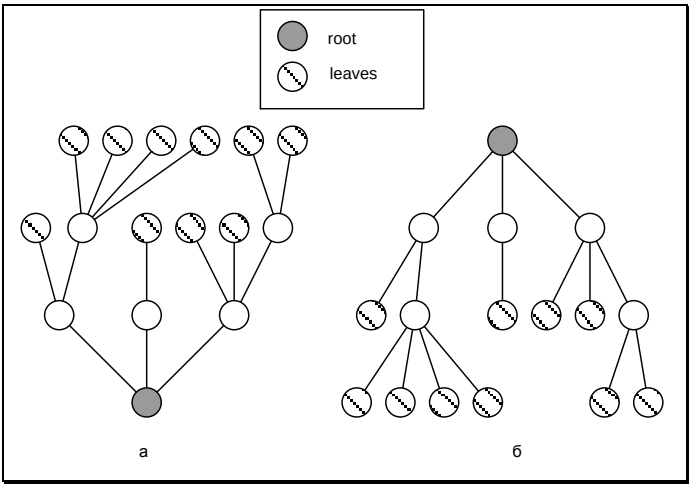


Рис. 4.1. Деревья общего вида

В соответствии с терминологией, логично изображать дерево, помещая корневой узел внизу картинки (рис. 4.1, а). Как это ни парадоксально, сложившаяся традиция изображения деревьев в задачах информатики и программирования предполагает размещение корневого узла вверху (рис. 4.1, б).

При выборе структур данных для представления древовидной организации исходят из требований и особенностей конкретной задачи. Если требуется быстрое перемещение между узлами дерева как вверх, так и вниз по дереву, в соответствующую структуру включают информацию как о родительском, так и о дочерних узлах.

Частным случаем дерева общего вида является **бинарное**, или двоичное, дерево, в котором у каждого узла не может быть более двух

потомков. Бинарные деревья являются, вероятно, наиболее широко используемым подмножеством древовидных структур. Определение типа для представления узла бинарного дерева может иметь следующий вид:

```
// Binary TREE NODE
template <class DATA>
struct BTreeNode
{
     BTreeNode<DATA> *parent;// Указатель на родительский узел

     BTreeNode<DATA> *left;    // Указатель на узел-корень левого
                               // поддерева
     BTreeNode<DATA> *right;   // Указатель на узел-корень правого
                               // поддерева

     DATA data; // Объект данных, хранящихся в узле дерева
};
```

Вообще говоря, дерево общего вида чаще всего хранят в памяти именно в виде бинарного (см., например, [Кубенский, 2004]). Для этого вместо указателей на всех потомков определяют два указателя. Первый указывает на одного из потомков (son node), второй – на соседний узел, имеющий того же родителя (brother node):

```
// General TREE NODE

template <class T>
struct GTreeNode
{
    // Вспомогательное определение типа "указатель на узел"
    typedef TreeNode<T>* PtrTreeNode;

    PtrTreeNode parent; // Указатель на родительский узел
                        // (для корневого узла: 0)

    PtrTreeNode brother; // Указатель на очередной узел
                         // на этом уровне
    PtrTreeNode son;     // Указатель на "старшего" из потомков
                         // (узлов следующего уровня)

    T data; // Объект данных, хранящихся в узле дерева
};
```

В данном учебном пособии мы сосредоточим внимание на важных для практики частных случаях использования древовидных структур, в основе которых лежат бинарные деревья.

Бинарные деревья и их применение

Узлы, имеющие одного родителя, часто называют узлами, находящимися на одном **уровне** дерева. Число уровней дерева определяет **высоту** дерева (рис. 4.2).

Дерево называют выровненным, если оно имеет минимальную высоту для заданного числа узлов. Бинарное дерево, изображенное на рис. 4.2, является невыровненным, в то время как дерево на рис. 4.3 – выровненное.

Если узлы дерева перенумеровать, то в выровненном дереве число уровней может быть вычислено как $[\log_2 n]+1$, где n – число узлов, а

квадратные скобки обозначают ближайшее целое, меньшее или равное значения в скобках.

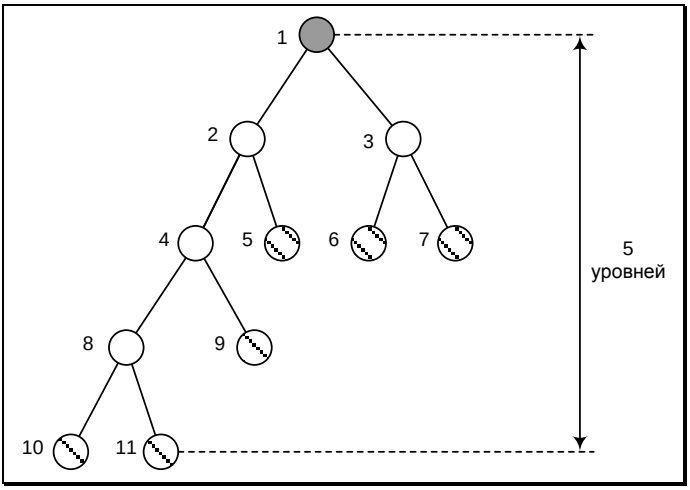


Рис. 4.2. Бинарное дерево

Сбалансированным бинарным деревом называют дерево, у которого для любого узла высота левого и правого поддеревьев отличается не более чем на 1. Разумеется, любое выровненное дерево является сбалансированным.

Большое практическое значение имеют выровненные бинарные деревья, организованные таким образом, что уровни заполнены последовательно слева направо (см. рис. 4.3).

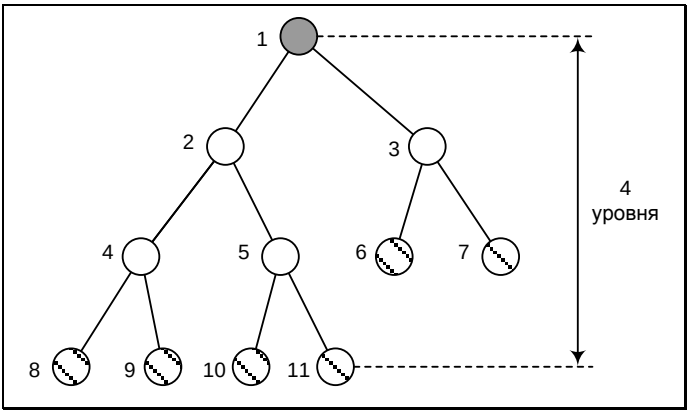


Рис. 4.3. Выровненное бинарное дерево

Выровненные деревья обеспечивают наибольшую эффективность при поиске. Однако выполнение операций добавления и удаления узлов может потребовать перестройки всего дерева. Поэтому в тех случаях, когда состав узлов дерева подвержен постоянным изменениям, на практике используют

сбалансированные, или АВЛ-деревья (Адельсон-Вельский и Ландис, 1962). Сбалансированные деревья уступают выровненным по гарантированной скорости поиска, однако выполнение операций вставки и удаления узлов во многих случаях затрагивает только конечное число узлов и не требует перестройки всего дерева.

Бинарное дерево как вариант организации данных в одномерном массиве

Выровненное бинарное дерево, подобное тому, которое изображено на рис. 4.3, можно «хранить» в упакованном виде в массиве, где индекс элемента массива определяется как «номер узла минус 1», поэтому переход от узла дерева к родительскому узлу или к узлам-потомкам осуществляется за счет элементарных арифметических действий над номерами узлов.

Пусть r – номер узла дерева, причем $1 < r \le n$. Тогда номер его родительского узла легко вычислить как $[r/2]$. Номер корневого узла левого поддерева для r есть $2 \cdot r$; номер корневого узла правого поддерева есть $2 \cdot r + 1$.

Оказывается, что во многих случаях обработка дерева с корнем в узле r сводится к последовательности обработки его левого и правого поддеревьев, которые, в свою очередь, являются сбалансированными бинарными деревьями. Следовательно, такая обработка хорошо описывается рекурсивным алгоритмом. В псевдокоде соответствующий процесс выглядит следующим образом:

```
// Обработать бинарное дерево из n узлов с корнем в узле noRoot
void ProcessTree( int noRoot )
{
    if( noRoot > n ) return; // выход из рекурсии

    // Обработать левое поддерево
    ProcessTree( noRoot*2 );

    // Обработать корень дерева
    // ...

    // Обработать правое поддерево
    ProcessTree( noRoot*2 + 1 );
}
```

Организация данных в виде упакованного бинарного дерева лежит в основе наиболее распространенного алгоритма поиска по ключу (в частности, именно этот алгоритм реализует функция `bsearch` стандартной библиотеки C++).

4.2. Алгоритм поиска с использованием бинарного дерева

Задача поиска ставится следующим образом. Пусть имеется множество K уникальных ключей $k_1, k_2, ..., k_n$, причем на множестве K определено

отношение строгого порядка. С каждым из ключей $k_i, i = \overline{1,n}$ связаны некоторые объекты данных из множества $\Delta = \{\delta_j \mid j = \overline{1,m}\}$. Требуется по заданному ключу κ найти $k_s \in \mathrm{K}$ и, следовательно, связанные с искомым ключом данные.

Простейший алгоритм поиска предполагает последовательное сравнение ключа κ с элементами множества K, однако эффективность такого алгоритма при равновероятной встречаемости ключей невысока и составляет в среднем $\frac{n}{2}$ сравнений ключей. Использование организации ключей в форме бинарного дерева позволяет повысить эффективность поиска до оценки $O(\log_2 n)$.

Пусть $k(r)$ – значение ключа, соответствующее узлу с номером r.

Разместим n уникальных ключей в узлах бинарного дерева таким образом, что для каждого узла r справедливы следующие отношения:

- В поддереве с корнем $2 \cdot r$ все ключи меньше ключа $k(r)$.
- В поддереве с корнем $2 \cdot r + 1$ все ключи больше ключа $k(r)$.

В этом случае, сравнивая исходный ключ с ключом, хранящимся в корне бинарного дерева, мы либо находим искомый ключ, либо можем однозначно определить поддерево, в котором находится искомый ключ. На каждом сравнении число нерассмотренных ключей уменьшается примерно вдвое, что и дает приведенную выше оценку эффективности.

Основная проблема заключается в подготовке массива ключей. Нетрудно заметить, что **при условии первоначальной упорядоченности ключей** приведенный выше псевдокод практически решает задачу первоначальной подготовки, обеспечивая обход узлов дерева в нужном порядке. Листинг 4.1 иллюстрирует адаптацию алгоритма «абстрактного» обхода дерева применительно к задаче первоначальной подготовки массива ключей.

Листинг 4.1. Подготовка дерева бинарного поиска (псевдокод)

```
struct SearchTreeNode
{
    KeyType  key;      // Ключ
    DataType *ptrData; // Указатель на объект данных,
                       // связанных с данным ключом
};

SearchTreeNode btree[ n ]; // Дерево поиска, упакованное в массив

// Первоначальная подготовка массива ключей для бинарного поиска
void PrepareTree( int noRoot )
{
    if( noRoot > n ) return; // выход из рекурсии
```

```
        // Подготовить левое поддерево (все меньшие ключи)
    PrepareTree( noRoot*2 );

    // Занести ключ и информацию о данных в корневой узел
    btree[ noRoot-1 ].key = GetNextKey(); // Извлечение очередного
                                          // ключа
                                          // из упорядоченного
                                          // набора
    btree[ noRoot-1 ].ptrData = GetNextData(); // Извлечение
                                               // данных

    // Подготовить правое поддерево (все большие ключи)
    PrepareTree( noRoot*2 + 1 );
}
```

Реализация поиска в подготовленном дереве не представляет труда. Листинг 4.2 содержит псевдокод решения задачи поиска.

Листинг 4.2. Бинарный поиск в подготовленном дереве (псевдокод)

```
// Поиск ключа key в подготовленном массива
// Результат, передаваемый через список параметров:
//    индекс элемента в массиве
// Возвращает true, если ключ найден,
//             false – в противном случае
bool BSearch(
              KeyType key,           // Исходный ключ
              int& ixFound           // Ссылка на искомый индекс
             )
{
    int noRoot = 1;

    while( noRoot <= n )
    {
        if( key == btree[ noRoot-1 ].key )
        {
            ixFound = noRoot-1;
            return true;
        }

        if( key < btree[ noRoot-1 ].key ) noRoot = noRoot*2;
        else                              noRoot = noRoot*2 + 1;
    }

    ixFound = -1;
    return false;
}
```

Результат предварительной подготовки дерева и поиска в нем некоторого символьного ключа иллюстрирует рис. 4.4.

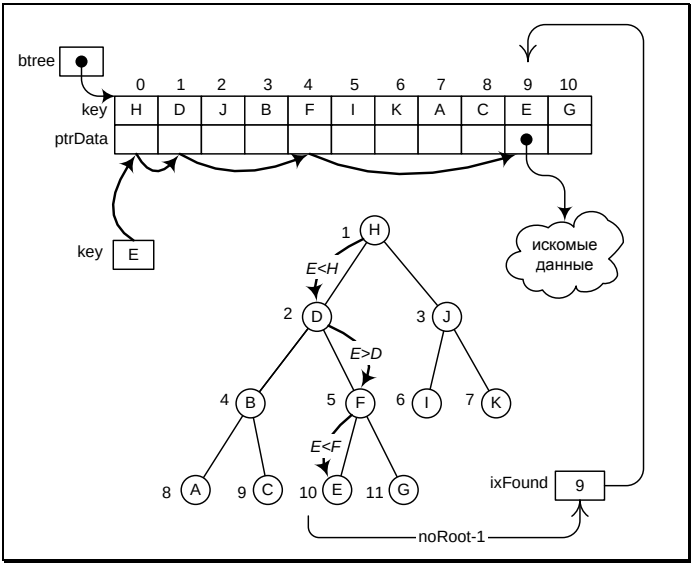


Рис. 4.4. Бинарный поиск

Реализация алгоритма поиска для конкретного типа ключей может быть предложена студентам в качестве самостоятельного практикума. Впрочем, пример для ключей, представляющий собой строки символов, мы рассмотрим на лекции.

Реализация бинарного поиска для ключей-строк символов (демонстрационный пример)

Анализируемый проект состоит из следующих файлов:

- Файл TreeNode.h содержит определение структуры для представления узла дерева поиска.
- Файл BSearch.cpp содержит комплект функций, необходимых для реализации бинарного поиска.
- Файл BSearch.h является заголовочным файлом, выступающим в роли интерфейса к модулю BSearch.cpp.
- Файл main.cpp содержит основную программу демонстрационного примера, обеспечивающую загрузку необходимых данных и запуск процедуры поиска.

Листинги 4.3—4.6 содержат тексты всех модулей проекта.

Листинг 4.3. Определение узла дерева (строки в таблице поиска)

```
/*
 * INCLUDE-файл   : TreeNode.h
 *
 * Проект : BSearch
 *          Console application
 *
```

```
         * Основной модуль проекта: main.cpp
         *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Определение структурного типа
 *             "узел дерева поиска"
 *
 * Тема: Использование бинарных деревьев в проектной практике
 *
 * Дата создания     : 06.08.2005
 * Дата корректировки:
 *
 * Исходные тексты примеров программ к курсу лекций
 * "Структуры и алгоритмы компьютерной обработки данных"
 *  Лекция 4
 *  Листинги 4.3-4.6
 *
 * Copyright (C) Пышкин Евгений Валерьевич, 2005
 *               Санкт-Петербургский государственный
 *               политехнический университет
 */
#ifndef _TreeNode_h_
#define _TreeNode_h_

#define KEYLEN 8
#define DATALEN 64

struct SearchTreeNode
{
    char  key[ KEYLEN ];      // Ключ (строка символов)
    char  ptrData[ DATALEN ]; // Указатель на строку данных,
                              // связанных с данным ключом
};

#endif

/*
 * Конец файла TreeNode.h
 */
```

Листинг 4.4. Функции подготовки и реализации бинарного поиска

```
/*
 * Файл   : BSearch.cpp
 *
 * Проект : BSearch
 *          Console application
 *
 * Основной модуль проекта: main.cpp
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Демонстрационный пример.
 *             Реализация функций для бинарного поиска.
 *             Ключи - строки символов, данные - строки символов
 *
 * Тема: Использование бинарных деревьев в проектной практике
 *
 * Дата создания     : 06.08.2005
 * Дата корректировки:
 *
 * Исходные тексты примеров программ к курсу лекций
 * "Структуры и алгоритмы компьютерной обработки данных"
 *  Лекция 4
```

```
 *  Листинги 4.3-4.6
 *
 * Copyright (C) Пышкин Евгений Валерьевич, 2005
 *               Санкт-Петербургский государственный
 *               политехнический университет
 */
#include <string.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include <new>
using namespace std;

#include "TreeNode.h"
#include "BSearch.h"

static SearchTreeNode *btree; // Таблица поиска (бинарное дерево,
                              //   "упакованное" в массив)
static int size;              // Его размер

static FILE *fInput; // Файл, из которого считывается
                     // упорядоченная последовательность ключей
                     // и связанные с ключами данных

// Подготовка таблицы поиска на основе информации,
// загружаемой из файла
void PrepareTreeFromFile( int size, const char* filename )
{
    fInput = fopen( filename, "rt" );
    if( fInput == NULL )
    {
       printf( "Error while opening key file\n" );
       exit( 1 );
    }

    InitSearchTree( size );

    PrepareTree( 1 );

    if( fclose( fInput ) == EOF )
    {
        printf( "Error while closing key file \n" );
        exit( 2 );
    }
}

// Создание дерева поиска заданного размера
void InitSearchTree( int size )
{
    if( size < 1 )
    {
        printf( "Incorrect key table size for binary search\n" );
        exit( 3 );
    }

    btree = new (nothrow) SearchTreeNode[ size ];
    if( btree ==  0 )
    {
        printf( "Error while allocating key table \n" );
        exit( 4 );
    }

    // Инициализация таблицы
```

```
            for( int ix = 0; ix < size; ix++ )
            {
        strcpy( btree[ ix ].key, "" );
        strcpy( btree[ ix ].ptrData, "" );
    }

    ::size = size;
}

// Удаление дерева поиска из памяти
void DestroyTree()
{
    delete [] btree;
}

// Первоначальная подготовка массива ключей для бинарного поиска
//    Работает в предположении, что набор ключей и связанных с
//    ними данных
//    загружается из файла (ключи упорядочены по возрастанию)
//    В каждой строке файла первые 8 позиций занимает ключ,
//    последующие 64 позиции занимают данные
void PrepareTree( int noRoot )
{
    if( noRoot > size ) return; // выход из рекурсии

    // Подготовить левое поддерево (все меньшие ключи)
    PrepareTree( noRoot*2 );

    // Занести ключ и информацию о данных в корневой узел
    int retScan = fscanf( fInput, "%8s%64s",
                                  btree[ noRoot-1 ].key,
                                  btree[ noRoot-1 ].ptrData );
    if( retScan != 2 )
    {
        printf( "Not enough data in the key file\n" );
        exit( 5 );
    }

    // Подготовить правое поддерево (все большие ключи)
    PrepareTree( noRoot*2 + 1 );
}

// Поиск ключа key в подготовленном массива
// Результат, передаваемый через список параметров:
//    индекс элемента в массиве
// Возвращает true, если ключ найден,
//             false - в противном случае
bool BSearch(
              char key[ KEYLEN ],    // Исходный ключ
              int& ixFound           // Ссылка на искомый индекс
            )
{
    int noRoot = 1;

    while( noRoot <= size )
    {
        if( strcmp( key, btree[ noRoot-1 ].key ) == 0 )
        {
            ixFound = noRoot-1;
            return true;
        }

        if( strcmp( key, btree[ noRoot-1 ].key ) < 0 )
                noRoot = noRoot*2;
```

```
        else    noRoot = noRoot*2 + 1;
    }

    ixFound = -1;
    return false;
}
// Извлечение информации из таблицы поиска по
// индексу строки таблицы
bool GetFromTable( int ixGet, char data[ DATALEN ] )
{
    if( ixGet < 0 || ixGet >= size )     return false;

    strcpy( data, btree[ ixGet ].ptrData );
    return true;
}

/*
 * Конец файла BSearch.cpp
 */
```

Листинг 4.5. Заголовочный файл к модулю BSearch.cpp

```
/*
 * INCLUDE-файл   : BSearch.h
 *
 * Проект : BSearch
 *          Console application
 *
 * Основной модуль проекта: main.cpp
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Заголовочный файл к модулю BSearch.cpp
 *
 * Тема: Использование бинарных деревьев в проектной практике
 *
 * Дата создания     : 06.08.2005
 * Дата корректировки:
 *
 * Исходные тексты примеров программ к курсу лекций
 * "Структуры и алгоритмы компьютерной обработки данных"
 *  Лекция 4
 *  Листинги 4.3-4.6
 *
 * Copyright (C) Пышкин Евгений Валерьевич, 2005
 *               Санкт-Петербургский государственный
 *               политехнический университет
 */
#ifndef _BSearch_h_
#define _BSearch_h_

#include "TreeNode.h"

void PrepareTreeFromFile( int size, const char* filename );
void InitSearchTree( int size );
void DestroyTree();
void PrepareTree( int noRoot );
bool BSearch(
              char key[ KEYLEN ],    // Исходный ключ
              int& ixFound           // Ссылка на искомый индекс
             );
bool GetFromTable( int ixGet, char data[ DATALEN ] );

#endif
```

```
        /*
 * Конец файла BSearch.h
 */
```

Листинг 4.6. Основной модуль проекта

```
/*
 * Файл   : main.cpp
 *
 * Проект : BSearch
 *          Console application
 *
 * Проект также содержит файлы: BSearch.cpp, BSearch.h,
 *                               TreeNode.cpp
 *
 * Язык программирования: Microsoft Visual C++ .NET
 *
 * Назначение: Демонстрационный пример.
 *             Иллюстрация бинарного поиска
 *
 * Тема: Использование бинарных деревьев в проектной практике
 *
 * Дата создания     : 06.08.2005
 * Дата корректировки:
 *
 * Исходные тексты примеров программ к курсу лекций
 * "Структуры и алгоритмы компьютерной обработки данных"
 *  Лекция 4
 *  Листинги 4.3-4.6
 *
 * Copyright (C) Пышкин Евгений Валерьевич, 2005
 *               Санкт-Петербургский государственный
 *               политехнический университет
 */
#include <stdio.h>

#include "BSearch.h"

void main()
{
    int ixFound;
    char data[ DATALEN ];

    PrepareTreeFromFile( 11, "btable.txt" );

    if( BSearch( "E", ixFound ) )
    {
        printf( "Key \"E\" is found in line %d\n", ixFound );
        GetFromTable( ixFound, data );
        printf( "Associated data: %s\n", data );
    }
    else
    {
        printf( "Key \"E\" is not found\n" );
    }

    if( BSearch( "L", ixFound ) )
    {
        printf( "Key \"L\" is found in line %d\n", ixFound );
        GetFromTable( ixFound, data );
        printf( "Associated data: %s\n", data );
    }
    else
```

```
    {
        printf( "Key \"L\" is not found\n" );
    }

    DestroyTree();
}

/*
 * Конец файла main.cpp
 */
```

Предположим, что в файле btable.txt содержится следующая таблица:

```
A       Adam
B       Bartok
C       Chopin
D       Dvorak
E       Elgar
F       Francais
G       Grieg
H       Hindemith
I       Ives
J       Jensen
K       Klemperer
```

В этом случае программа порождает следующий вывод:

```
Key "E" is found in line 9
Associated data: Elgar
Key "L" is not found
```

4.3. Алгоритм сортировки с использованием бинарного дерева

На использовании идеи обработки элементов массива как узлов дерева основан довольно эффективный алгоритм сортировки массива, известный под названием пирамидальной сортировки, или сортировки сложным выбором (предложен Дж. Уильямсом). Сортировка основана на первоначальной подготовке массива таким образом, чтобы элементы соответствующего бинарного дерева образовывали так называемую пирамиду, то есть выполнялись следующие условия:

- Для любого узла r, имеющего одного потомка (то есть связанного с левым поддеревом) $k(r) \geq k(2 \cdot r)$.
- Для любого узла r, имеющего двух потомков, $k(r) \geq k(2 \cdot r)$ и $k(r) \geq k(2 \cdot r + 1)$.

Таким образом, в пирамиде корневая вершина любого поддерева соответствует наибольшему ключу в этом поддереве (естественно, любой лист дерева автоматически является пирамидой). Предположим, что пирамида сконструирована. Тогда процесс сортировки массива из n элементов описывается следующим итерационным алгоритмом:

```
for( last = n; // Номер последнего узла в начале равен n
     last > 1; // Продолжаем, пока не рассмотрели все узлы
     last-- )
{
    // Поменять местами узлы с номерами 1 (корневой) и last
```

```
            // ...

    // Восстановить пирамиду из дерева с узлами от 1 до last-1
    // ...
}
```

Поскольку на вершине пирамиды всегда находится элемент с наибольшим значением ключа, то после обмена этого элемента с последним элементом анализируемой части исходного массива, он оказывается на своем месте и его больше рассматривать не нужно.

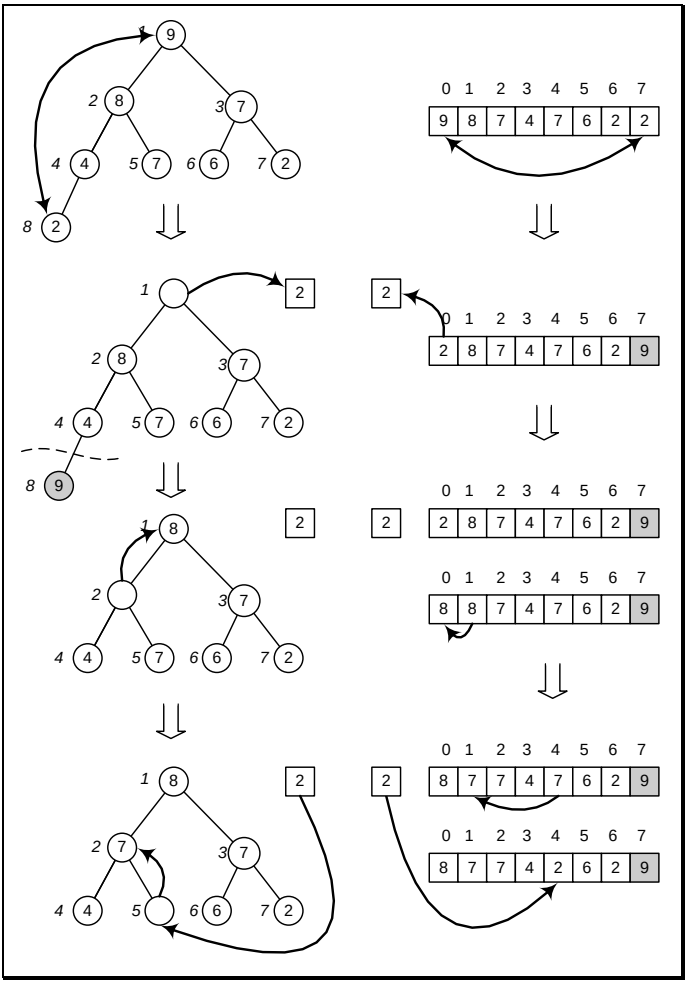


Рис. 4.5. Пирамидальная сортировка

Наиболее сложной частью реализации является восстановление пирамиды. Процесс однократного восстановления пирамиды иллюстрирует рис. 4.5.

После совершенного обмена элемент, оказавшийся в корне дерева, копируется в некоторую вспомогательную переменную (обозначим ее x). Претендентом на заполнение образовавшейся вакансии («дыры» в дереве) является наибольший из корневых элементов левого и правого поддеревьев. Далее возможны два варианта:

- Претендент меньше или равен x. В этом случае вакансия заполняется значением из x. Пирамида восстановлена.
- Претендент больше значения ключа, хранящегося в x, он занимает вакансию, оставляя за собой новую «дыру». Претендентом на новую вакансию являются наибольший из корневых элементов левого и правого поддеревьев относительно «дыры». Претендент сравнивается с ключом из x, и процесс продолжается до тех пор, пока либо пирамида не будет восставлена по первому варианту, либо не обнаружен лист дерева.

Процесс восстановления пирамиды последовательным заполнением образующихся на месте продвигаемых «вверх» элементов вакансий, обычно называют просеиванием. Функцию просеивания представляет листинг 4.7.

Листинг 4.7. Просеивание дыр

```
// Просеивание дерева с корнем в root
template <class T>
void Sift( T *a,     // Обрабатываемый массив
           int root, // Номер корневого узла
           int last  // Номер последнего узла
         )
{
    int hole = root;   // Дыра образуется на месте корня

    T x = a[ hole-1 ];

    for( ; ; )
    {
        int left = 2*hole;    // Корень левого поддерева
        int right = left + 1; // Корень правого поддерева

        int pretender;        // Номер претендента

        if( left > last )  break; // Если это лист - просеивание
                                  // закончено

        // Выбираем претендента
        if( left == last ) pretender = left;
        else
        {
            if( a[ left-1 ] < a[ right-1 ] )  pretender = right;
            else                              pretender = left;
        }

        // Если претендент не лучше x - просеивание закончено
        if( a[ pretender-1 ] < x  ) break;
        if( a[ pretender-1 ] == x ) break;

        a[ hole-1 ] = a[ pretender-1 ]; // Образование новой дыры
        hole = pretender;
    }
    // Заполняем дыру
```

```
    a[ hole-1 ] = x;
}
```

Последовательное просеивание ($n-1$ циклов) обеспечивает упорядоченность всего массива. Остается обеспечить выполнение заявленного условия: в начале сортировки дерево должно быть пирамидой.

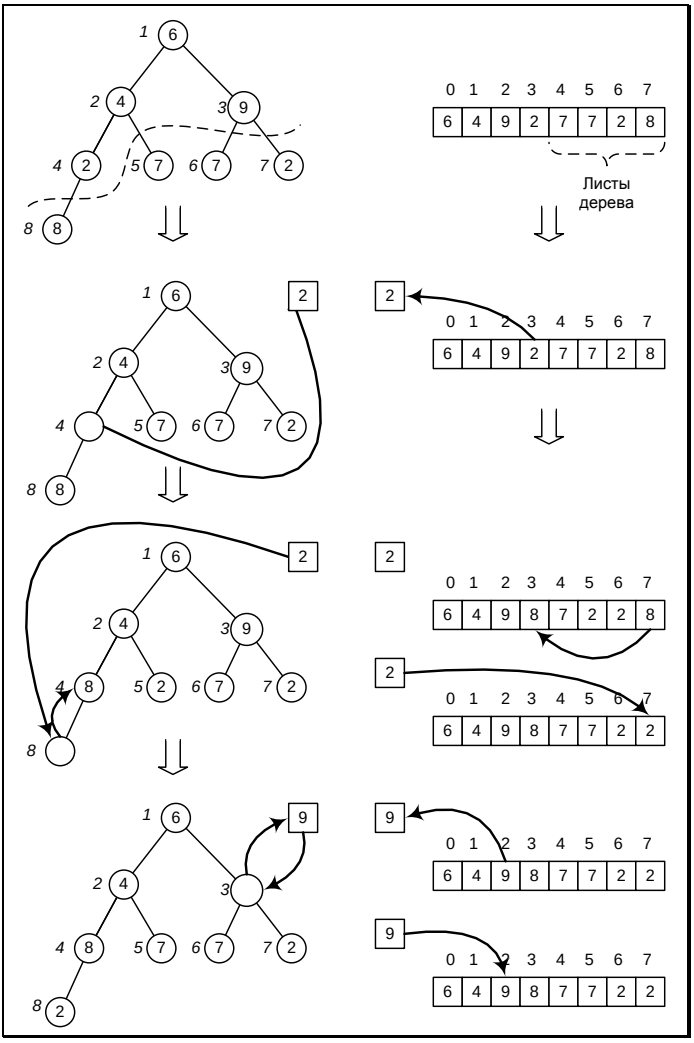


Рис. 4.6. Подготовка пирамиды

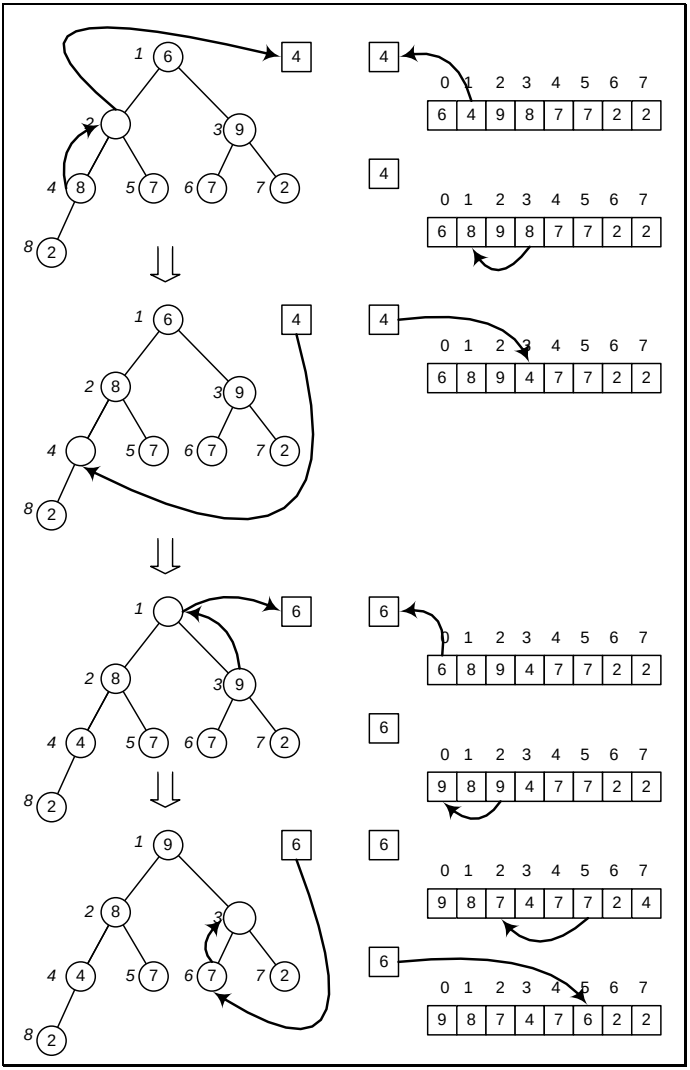


Рис. 4.6 (продолжение)

Поскольку каждый лист является пирамидой по определению, для построения пирамиды нужно осуществить последовательное просеивание «снизу вверх»: от поддерева с корнем номер $\frac{n}{2}$ до поддерева с корнем номер 1. (рис. 4.6).

Окончательный вариант пирамидальной сортировки иллюстрирует листинг 4.8.

Листинг 4.8. Пирамидальная сортировка

```
        template <class T>
void HeapSort( T *a, int n )
{
    int root;
    int last;

    // Первоначальная подготовка пирамиды
    for( root = n/2; root > 0; root-- ) Sift( a, root, n );

    // Цикл сортировки
    for( last = n; last > 1; last-- )
    {
        // Обмен корневого и последнего узлов
        T copy = a[ 0 ];
        a[ 0 ] = a[ last-1 ];
        a[ last-1 ] = copy;

        // Просеивание
        Sift( a, 1, last-1 );
    }
}
```

В [McConnell, 2001] показано, что эффективность пирамидальной сортировки практически не зависит от первоначального расположения элементов и оценивается величиной $O(n \log_2 n)$.

В заключение отметим, что в стандартной библиотеке C++ построение пирамидальной структуры используется при реализации некоторых алгоритмов обработки векторов STL.

4.4. Двоичные деревья общего вида: удаление и добавление узлов

В некоторых случаях важно обеспечить работу с динамическими структурами данных, для которых следует обеспечить возможность добавления и удаления элементов. В этом случае формат хранения в виде массива не подходит, поскольку добавление или удаление элемента в большинстве случаев потребует полной перестройки дерева.

Основываясь на приведенной в начале главы структуре, представляющий узел бинарного дерева, рассмотрим определение класса, реализующего наиболее распространенные операции над двоичным деревом(листинг 4.9).

Листинг 4.9. Определение класса бинарного дерева

```
template <class T>
class BTree
{
public:
     // Узел бинарного дерева
     template <class DATA>
     struct BTreeNode
     {
          BTreeNode<DATA> *parent;  // Указатель на родительский
                                    // узел

          BTreeNode<DATA> *left;    // Указатель на узел-корень
                                    // левого поддерева
```

```
          BTreeNode<DATA> *right;   // Указатель на узел-корень
                                    // правого поддерева

          DATA data; // Объект данных, хранящихся в узле дерева

          BTreeNode(const DATA& _data,
                         BTreeNode<DATA>* _parent = NULL,
                         BTreeNode<DATA>* _left = NULL,
                         BTreeNode<DATA>* _right = NULL
                    )
          {
               data = _data;
               parent = _parent;
               left = _left;
               right = _right;
          }
     };

private:
     BTreeNode<T> *root; // Корень дерева

public:
     BTree()
     {
          root = NULL;
     }

     // Вставка узла в бинарное дерево
     BTreeNode<T> *InsertNode( const T& data );

     // Поиск узла в бинарном дереве
     BTreeNode<T> *FindNode( const T& data );

     // Удаление узла, содержащего заданный объект
     void DeleteNode( const T& data );

     // Удаление заданного узла дерева
     void DeleteNode( BTreeNode<T> *toDelete );

     // Контрольная печать узлов дерева
     void Print();

private:
     // Печать с информацией об уровнях
     void PrintTree( BTreeNode<T> *root );
};
```

Операции добавления узла дерева сводится к поиску подходящего места для вставки листа дерева (в соответствии с алгоритмом бинарного поиска) и «присоединению» нового элемента к древовидной структуре (см. листинг 4.10). Отметим, что функция работает корректно для данных произвольного пользовательского типа при условии, что для этого типа определены операции отношения == и <.

Листинг 4.10. Вставка узла в бинарное дерево

```
// Вставка узла в бинарное дерево
template <class T>
typename BTree<T>::BTreeNode<T> *BTree<T>::InsertNode( const T&
data )
{
     // Вспомогательные указатели:
     BTreeNode<T> *newItem; // на новый элемент
```

```
                    BTreeNode<T> *current; // на обозреваемый
      элемент
      BTreeNode<T> *parent;  // на родительский элемент для
                             // обозреваемого элемента

      // Поиск места вставки
      current = root;
      parent = NULL;

      while( current != NULL )
      {
            if( data == current->data ) return current;
                                        // Т.е. такой узел
                                        // уже есть
            parent = current;
            if( data < current->data ) current = current->left;
            else                       current = current->right;
      }

      // Создание нового узла
      newItem = new BTreeNode<T>( data, parent );

      // Привязка нового узла к дереву
      if( parent == NULL )
      {
            root = newItem;
      }
      else
      {
            // Здесь: новый элемент не является корнем дерева
            if( newItem->data < parent->data )
                  parent->left = newItem;
            else  parent->right = newItem;
      }

      return newItem;
}
```

Заметим, что в общем случае в результате работы функции значение `root` может измениться.

Как явствует из рис. 4.7, добавление узла может привести к нарушению сбалансированности дерева. Если сбалансированность дерева на каждом шаге обработки не поддерживается, нельзя гарантировать наилучшую эффективность алгоритма двоичного поиска $O(\log_2 n)$, поэтому во многих практических случаях после вставки элемента требуется произвести «балансировку» двоичного дерева (этому посвящен следующий раздел данного главы).

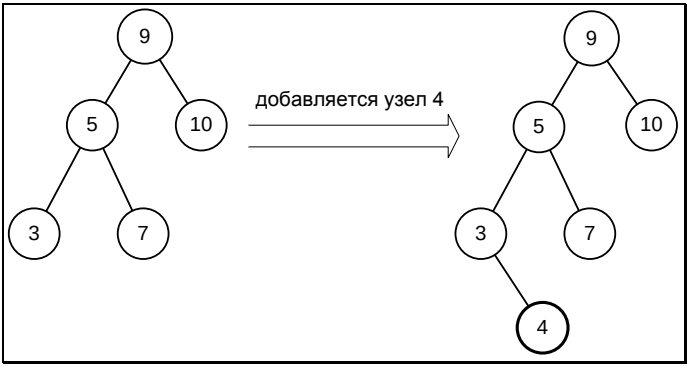


Рис. 4.7. Нарушение сбалансированности при добавлении нового узла бинарного дерева

В процессе обработки и использования бинарного дерева часто возникает необходимость решения задачи поиска объекта данных. Листинг 4.11 иллюстрирует вариант реализации алгоритма поиска в бинарном дереве общего вида.

Листинг 4.11. Поиск в бинарном дереве общего вида

```
// Поиск узла в бинарном дереве
template <class T>
typename BTree<T>::BTreeNode<T> *BTree<T>::FindNode( const T& data
)
{
     BTreeNode<T> *current = root;

     for( ; ; )
     {
          if( current == NULL ) return NULL;

          if( current->data == data ) return current;

          if( data < current->data )     current = current->left;
          else                           current = current->right;
     }
}
```

Функция возвращает указатель на узел дерева, что позволяет использовать ее в качестве подпроцесса в других алгоритмах, например, при удалении узла из двоичного дерева.

Процедура удаления узла является несколько более сложной в связи с тем, что после удаления в дереве остается «дыра» и требуется реорганизовать одно из поддеревьев (см. рис. 4.8). Для этого нужно найти узел с наименьшим значением в правом поддереве относительно удаляемого узла. Поиск такого узла состоит из спусков «шаг вправо - шаг влево» до достижения концевого узла.

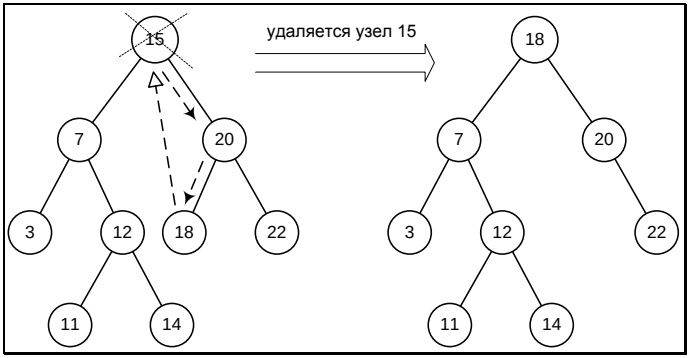


Рис. 4.8. Удаление узла и восстановление структуры бинарного дерева

Реализацию алгоритма удаления узла иллюстрирует листинг 4.12.

Листинг 4.12. Удаление узла из бинарного дерева и восстановление структуры дерева

```
// Удаление узла, содержащего заданный объект
template <class T>
void typename BTree<T>::DeleteNode( const T& data )
{
     BTreeNode<T> *toDelete = FindNode( data );
     DeleteNode( toDelete );
}

// Удаление заданного узла дерева
template <class T>
void typename BTree<T>::DeleteNode( BTreeNode<T> *toDelete )
{
     BTreeNode<T> *x, *y; // Указатели на узлы в ходе
                          // спусков влево-вправо

     if( toDelete == NULL ) return;

     if( toDelete->left == NULL || toDelete->right == NULL )
     {
          // Хотя бы одно из поддеревьев отсутствует
          y = toDelete;
     }
     else
     {
          // Есть оба поддерева
          y = toDelete->right;

          while( y->left != NULL ) y = y->left;
     }

     // Обрабатываем случай, когда у узла "y" только один потомок
     if( y->left != NULL ) x = y->left;
     else                  x = y->right;

     // Исключаем узел "y" из "родительской" цепочки
     if( x!= NULL ) x->parent = y->parent;
     if( y->parent != NULL )
```

```
    {
        if( y == y->parent->left ) y->parent->left = x;
        else                       y->parent->right = x;
    }
    else root = x;

    if( y != toDelete )
    {
        toDelete->data =y->data;
    }

    delete y;
}
```

Нетрудно заметить, что и при удалении узла процесс восстановления структуры бинарного дерева также может привести к его несбалансированности.

Реализацию функций контрольной печати содержимого дерева оставляем читателю в качестве самостоятельного упражнения.

Красно-черные деревья как инструмент восстановления сбалансированности двоичных деревьев

Как было указано в предыдущем разделе, поддержание сбалансированности двоичного дерева во многих случаях является необходимым. Одним из распространенных приемов, используемых для частичной балансировки двоичных деревьев, являются красно-черные деревья. Название происходит от условной раскраски узлов дерева в черный и красный цвета. Во время операций вставки и удаления узлов происходят так называемые повороты дерева для достижения его сбалансированности. Эффективность алгоритма как в наихудшем случае, так и в среднем оценивается классом сложности $O(\lg n)$ [Niemann, 1995].

Красно-черное дерево – это бинарное дерево, обладающее следующими свойствами:

- Каждый узел покрашен либо в красный, либо в черный цвет.
- Листьями объявляются **NIL-узлы** – «виртуальные» узлы, являющиеся как бы потомками концевых узлов (которые обычно и называют листьями). Можно считать, что указатель NULL «указывает» на лист красно-черного дерева. NIL-узлы окрашены в черный цвет.
- У красного узла оба потомка обязательно черные (однако у черного узла не обязательно красные потомки).
- На всех ветвях дерева, ведущих от корня к листьям (NIL-узлам), число черных узлов одинаково (это значение называется **черной высотой** дерева).
- Корень дерева окрашен в черный цвет.

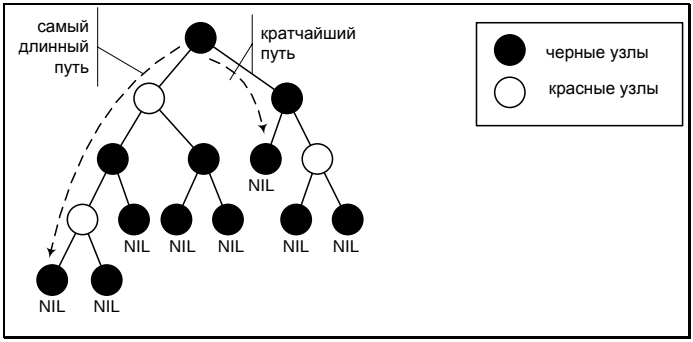


Рис. 4.9. Красно-черное дерево с черной высотой 3

Перечисленные свойства гарантируют, что самая длинная ветвь от корня к листу не более чем вдвое длиннее любой другой ветви от корня к листу (см. рис. 4.9). Иными словами, красно-черные деревья гарантируют лишь частичную сбалансированность, однако на практике в большинстве случаев и этого достаточно.

Структура данных для представления узла красно-черного дерева дополняется одним полем для представления цвета узла. Инициализация красно-черного дерева включает в себя следующие действия (вместо TYPE нужно вставить имя типа данных, хранящихся в дереве):

- Определение сторожевого NIL-узла.
- Инициализация корня дерева.

Заметим, что введение сторожевых элементов в качестве листьев, вообще говоря, не обязательно, но существенно упрощает разработку алгоритма. В связи с этим в приводимой ниже реализации следует отличать значение указателя NULL от значения NIL (адрес сторожевого узла).

Листинг 4.13 содержит определение класса для представления красно-черного бинарного дерева. Пояснения к этому тексту приводятся далее в этом разделе.

Листинг 4.13. Класс для представления красно-черного дерева

```
template <class T>
class RBTree
{
public:
     typedef enum { BLACK, RED } NodeColor;
     // Определение сторожевого узла
     // (фиктивного черного листа красно-черного дерева
     #define NIL &sentinel

     // Red Black TREE NODE
     template <class DATA>
     struct RBTreeNode
     {
          RBTreeNode<DATA> *parent; // Указатель на родительский
                                    // узел
```

```
          RBTreeNode<DATA> *left;  // Указатель на узел-корень
                                   // левого поддерева
          RBTreeNode<DATA> *right; // Указатель на узел-корень
                                   // правого поддерева

          NodeColor      color;  // Цвет узла

          DATA data; // Объект данных, хранящихся в узле дерева

          RBTreeNode(    const DATA& _data,
                         RBTreeNode<DATA>* _parent = NULL,
                         RBTreeNode<DATA>* _left = NIL,
                         RBTreeNode<DATA>* _right = NIL,
                         NodeColor _color = RED
                    )
          {
               data = _data;
               parent = _parent;
               left = _left;
               right = _right;
               color = _color;
          }
     };

private:
     RBTreeNode<T> sentinel;  // Сторожевой узел
     RBTreeNode<T> *root;     // Корень дерева

public:
     RBTree() : sentinel( 0, NULL, NIL, NIL, BLACK )
     {
          root = NIL;
     }

     // Вставка узла
     RBTreeNode<T> *InsertNode( const T& data )
     {
          // Вспомогательные указатели:
          RBTreeNode<T> *newItem; // на новый элемент
          RBTreeNode<T> *current; // на обозреваемый элемент
          RBTreeNode<T> *parent;  // на родительский элемент для
                                  // обозреваемого элемента

          // Поиск места вставки
          current = root;
          parent = NULL;

          while( current != NIL )
          {
               if( data == current->data ) return current;
                                           // Такой узел
                                           // уже есть

               parent = current;
               if( data < current->data )
                    current = current->left;
               else current = current->right;
          }

          // Создание нового узла
          newItem = new RBTreeNode<T>( data,parent,NIL,NIL,RED );

          // Привязка нового узла к дереву
```

```
                    if( parent == NULL )
                    {
                root = newItem;
            }
            else
            {
                // Здесь: новый элемент не является корнем дерева
                if( newItem->data < parent->data )
                        parent->left = newItem;
                else    parent->right = newItem;
            }

            // Восстановление баланса после вставки
            InsertFixup( newItem );
            return newItem;
        }

        // Поиск узла
        RBTreeNode<T> *FindNode( const T& data )
        {
            RBTreeNode<T> *current = root;

            for( ; ; )
            {
                if( current == NIL ) return NULL;

                if( current->data == data ) return current;

                if( data < current->data )
                        current = current->left;
                else    current = current->right;
            }
        }

        // Удаление узла по значению данных
        void DeleteNode( const T& data )
        {
            RBTreeNode<T> *toDelete = FindNode( data );
            DeleteNode( toDelete );
        }


        // Удаление узла по адресу узла
        void DeleteNode( RBTreeNode<T> *toDelete )
        {
            RBTreeNode<T> *x, *y;     // Указатели на узлы в ходе
                                      // спусков влево-вправо

            if( toDelete == NULL || toDelete == NIL ) return;

            if( toDelete->left == NIL || toDelete->right == NIL )
            {
                // Хотя бы одно из поддеревьев отсутствует
                y = toDelete;
            }
            else
            {
                // Есть оба поддерева
                y = toDelete->right;

                while( y->left != NIL ) y = y->left;
            }

            // Обрабатываем случай, когда у узла "y"
```

```
          // только один потомок
          if( y->left != NIL ) x = y->left;
          else                 x = y->right;

          // Исключаем узел "y" из "родительской" цепочки
          x->parent = y->parent;
          if( y->parent != NULL )
          {
               if( y == y->parent->left ) y->parent->left = x;
               else                       y->parent->right = x;
          }
          else root = x;

          if( y != toDelete )
          {
               toDelete->data =y->data;
          }

          if( y->color == BLACK )
          {
               // Восстановление баланса после удаления
               DeleteFixup( x );
          }

          delete y;
     }

private:
     // Поворот дерева с корнем root вправо относительно узла x
     void RotateRight( RBTreeNode<T> *x )
     {
          RBTreeNode<T> *y = x->left;

          // Формируем левое поддерево для x
          x->left = y->right;
          if( y->right != NIL ) y->right->parent = x;

          if( y != NIL ) y->parent = x->parent;
          if( x->parent != NULL )
          {
               if( x == x->parent->right ) x->parent->right = y;
               else                        x->parent->left = y;
          }
          else root = y;

          // Связываем x и y
          y->right = x;
          if( x != NIL ) x->parent = y;
     }

     // Поворот дерева с корнем root влево относительно узла x
     void RotateLeft( RBTreeNode<T> *x )
     {
          RBTreeNode<T> *y = x->right;

          // Формируем правое поддерево для x
          x->right = y->left;
          if( y->left != NIL ) y->left->parent = x;

          if( y != NIL ) y->parent = x->parent;
          if( x->parent != NULL )
          {
               if( x == x->parent->left ) x->parent->left = y;
               else                       x->parent->right = y;
```

```
                    }
                    else root = y;

     // Связываем x и y
     y->left = x;
     if( x != NIL ) x->parent = y;
}

// Восстановление баланса после вставки узла x
void InsertFixup ( RBTreeNode<T> *x )
{
     // Двигаемся в направлении root пока не будет
     // восстановлено свойство 3
     while( x != root && x->parent->color == RED )
     {
          if( x->parent == x->parent->parent->left )
          {
               // Родитель в левом поддереве
               // относительно деда
               RBTreeNode<T> *uncle =
                    x->parent->parent->right;

               if( uncle->color == RED )
               {
                    // Перекрашиваем родителя и дядю в
                    // черный...
                    x->parent->color = uncle->color = BLACK;
                    // ... а деда - в красный
                    x->parent->parent->color = RED;

                    x = x->parent->parent;    // Теперь дед -
                                              // это x
               }
               else // Дядя - черный
               {
                    if( x == x->parent->right )
                    {
                         x = x->parent;
                         RotateLeft( x );
                    }
                    x->parent->color = BLACK;
                    x->parent->parent->color = RED;
                    RotateRight( x->parent->parent );
               }
          }
          else
          {
               // Родитель в правом поддереве
               // относительно деда
               // (зеркальное отражение предыдущего блока)
               RBTreeNode<T> *uncle =
                    x->parent->parent->left;

               if( uncle->color == RED )
               {
                    x->parent->color = uncle->color = BLACK;
                    x->parent->parent->color = RED;
                    x = x->parent->parent;
               }
               else
               {
                    if( x == x->parent->left )
                    {
                         x = x->parent;
```

```
                              RotateRight( x );
                         }
                         x->parent->color = BLACK;
                         x->parent->parent->color = RED;
                         RotateLeft( x->parent->parent );
                    }
               }
          }

          root->color = BLACK;
     }

     // Восстановление баланса после удаления узла x
     void DeleteFixup ( RBTreeNode<T> *x )
     {
          while( x != root && x->color == BLACK )
          {
               if( x == x->parent->left )
               {
                    // Удаляемый элемент – корень левого
                    // поддерева
                    RBTreeNode<T> *brother = x->parent->right;

                    if( brother->color == RED )
                    {
                         brother->color = BLACK;
                         x->parent->color = RED;
                         RotateLeft( x->parent );
                         brother = x->parent->right;
                    }

                    if( brother->left->color == BLACK &&
                         brother->right->color == BLACK )
                    {
                         brother->color = RED;
                         x =  x->parent;
                    }
                    else
                    {
                         if( brother->right->color == BLACK )
                         {
                              brother->left->color = BLACK;
                              brother->color = RED;
                              RotateRight( brother );
                              brother = x->parent->right;
                         }
                         brother->color = x->parent->color;
                         x->parent->color = BLACK;
                         brother->right->color = BLACK;
                         RotateLeft( x->parent );
                         x = root;
                    }
               }
               else
               {
                    // Удаляемый элемент – корень правого
                    // поддерева
                    RBTreeNode<T> *brother = x->parent->left;

                    if( brother->color == RED )
                    {
                         brother->color = BLACK;
                         x->parent->color = RED;
                         RotateRight( x->parent );
```

```
                                        brother = x->parent->left;
                                   }

                    if( brother->right->color == BLACK &&
                        brother->left->color == BLACK )
                    {
                        brother->color = RED;
                        x = x->parent;
                    }
                    else
                    {
                        if( brother->left->color == BLACK )
                        {
                            brother->right->color = BLACK;
                            brother->color = RED;
                            RotateLeft( brother );
                            brother = x->parent->left;
                        }
                        brother->color = x->parent->color;
                        x->parent->color = BLACK;
                        brother->left->color = BLACK;
                        RotateRight( x->parent );
                        x = root;
                    }
               }
          }

          x->color = BLACK;
     }

     void DeleteNode( RBTreeNode<T> *&root,
                      RBTreeNode<T> *toDelete )
     {
          RBTreeNode<T> *x, *y;     // Указатели на узлы в ходе
                                    // спусков влево-вправо

          if( toDelete == NULL || toDelete == NIL ) return;

          if( toDelete->left == NIL || toDelete->right == NIL )
          {
               // Хотя бы одно из поддеревьев отсутствует
               y = toDelete;
          }
          else
          {
               // Есть оба поддерева
               y = toDelete->right;

               while( y->left != NIL ) y = y->left;
          }

          // Обрабатываем случай, когда у узла "y" только
          // один потомок
          if( y->left != NIL ) x = y->left;
          else                 x = y->right;

          // Исключаем узел "y" из "родительской" цепочки
          x->parent = y->parent;
          if( y->parent != NULL )
          {
               if( y == y->parent->left ) y->parent->left = x;
               else                       y->parent->right = x;
          }
          else root = x;
```

```
        if( y != toDelete )
        {
           toDelete->data =y->data;
        }

        if( y->color == BLACK )
        {
             // Восстановление баланса после удаления
             DeleteFixup( root, x );
        }

        delete y;
    }
```

Вставка узла состоит из двух последовательно выполняющихся этапов. Первый этап – это собственно вставка нового узла в дерево (этот процесс отличается от вставки узла в обычное бинарное дерево только необходимостью окраски узла в красный цвет). Второй этап – восстановление красно-черных свойств дерева и, следовательно, его сбалансированности. Эту задачу решает специальный метод класса `InsertFixup`.

Теперь рассмотрим реализацию алгоритма, восстанавливающего сбалансированность красно-черного дерева. В процессе вставки мы покрасили вставленный узел в красный цвет. Теперь следует посмотреть на родительский узел с целью проверки, не нарушается ли при вставке свойство 3. Рассмотрим ситуацию, когда родительский узел красный. В этой ситуации достаточно рассмотреть следующие два случая:

Красный «родитель», красный «дядя» (рис. 4.10). В этом случая от нарушения свойства 3 можно избавиться простым перекрашиванием узлов с последующим движением к корню дерева, то есть цвет родителя и дяди меняется на черный, а цвет деда – на красный. Далее процесс повторяется для деда. В самом конце корень окрашивается в черный цвет. Если до этого он был красным, увеличивается черная высота дерева.

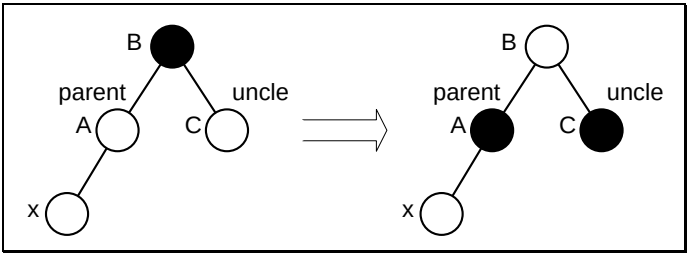


Рис. 4.10. Вставка «красный родитель – красный дядя»

Красный «родитель», черный «дядя» (рис. 4.11). В этом случае корректировка поддеревьев происходит так называемым вращением узлов. Например, если вставляемый узел является левым потомком своего родителя, то цвет родителя меняется на черный, цвет деда – на красный, а дерево «поворачивается» вправо относительно родителя вставляемого узла (см. также рис. 4.12).

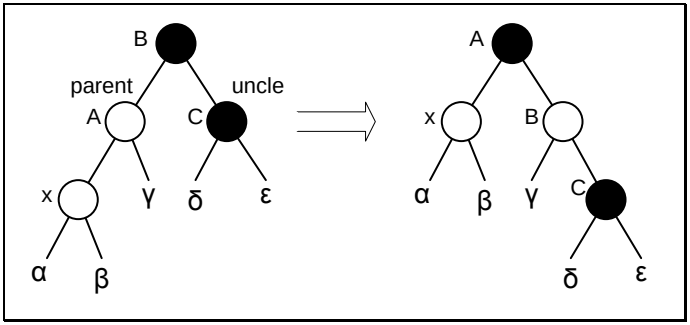


Рис. 4.11. Вставка «красный родитель – черный дядя»

На рис. 4.11 показан поворот дерева вправо. В процессе восстановления сбалансированности может также потребоваться поворот дерева влево. Соответствующие действия реализую закрытые методы `RotateRight` и `RotateLeft`.

Рис. 4.12 и 4.13 иллюстрируют начальное и конечное состояние фрагмента дерева после осуществления вращений вправо и влево относительно узла ***x***. При этом утолщенными линиями выделены связи, ориентация которых не изменяется в результате вращения.

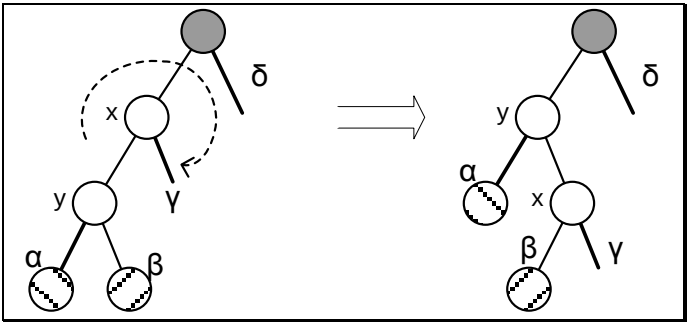


Рис. 4.12. Вращение фрагмента дерева вправо

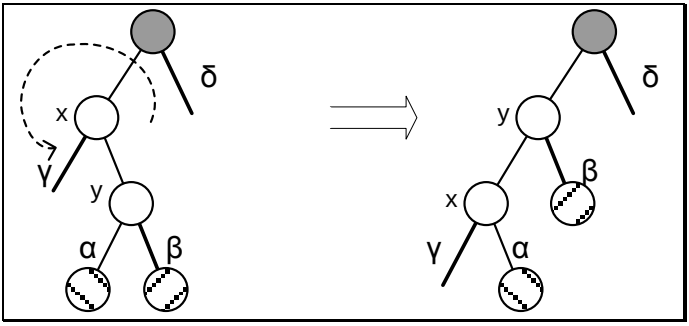


Рис. 4.13. Вращение фрагмента дерева влево

На основе функций поворота и определяется функция восстановления сбалансированности после вставки `InsertFixup`.

Аналогично процесс удаления узла удлиняется на фазу восстановления сбалансированности `DeleteFixup`, если это требуется.

В заключение необходимо заметить, что применение красно-черных деревьев оправдано тогда, когда данные часто меняются. Для деревьев с неизменной или редко меняющейся структурой, скорей всего, более подходящим решением будет заранее оптимизированное бинарное дерево или использование хеширования.

Глава 5. Алгоритмы на графах

Графы предоставляют механизм формализации многих задач, в которых в основе постановки лежит набор отношений объектов, например, задач по оптимизации систем дорог (коммуникаций), сетей взаимодействующих объектов, задач, имеющих дело с программными или лингвистическими моделями (наглядные примеры таких задач приведены, например, в книге [Новиков, 2000]).

В наиболее распространенных случаях формулирование задачи в терминах теории графов предполагает дальнейшее решение одной из следующих задач (или их комбинации):

- нахождение кратчайшего пути из одной вершины графа в другую;
- нахождение минимального остовного дерева графа, или кратчайшего остова;
- нахождение пути минимальной стоимости, включающего все вершины графа или его подграфа (задача коммивояжера);
- нахождение компонент связности графа.

В этом курсе мы рассмотрим алгоритмы решения задач поиска кратчайшего пути и минимального остова графа, оставив изучение остальных задач для самостоятельной работы студентов.

5.1. Некоторые напоминания из теории графов

В данном разделе мы напомним основные определения теории графов, которые потребуются для дальнейшего изложения.

Определение графа

Напомним, что с формальной точки зрения граф G определяется как пара множеств (V, E), где V – непустое множество вершин, а E – множество пар различных элементов множества V, причем если речь идет о неупорядоченных парах, то граф G называется неориентированным (а множество E называется множеством ребер). Если же E состоит из упорядоченных пар, это означает, что мы имеем дело с ориентированным графом (орграфом), вершины которого обычно называют узлами, а множество упорядоченных пар – множеством дуг.

Смежность и инцидентность

Пусть имеются две вершины v_1 и v_2, а $e = (v_1, v_2)$ – соединяющее эти вершины ребро. Тогда ребро e называется инцидентным вершинам v_1 и v_2, а сами вершины – смежными.

Множество вершин, смежных с некоторой вершиной, называется множеством смежности этой вершины.

Подграф

Подграф (V_s, E_s) графа или орграфа (V, E) состоит из некоторого подмножества вершин $V_s \subseteq V$ и некоторого подмножества ребер (дуг) $E_s \subseteq E$, причем если $V_s = V$, подграф (V_s, E_s) называется остовным подграфом.

Путь и расстояние

Чередующаяся последовательность вершин и ребер $v_0, e_1, v_1, e_2, v_2, ..., e_k, v_k$, в которой любые два соседних элемента инцидентны, называется маршрутом длины k. При этом, если все ребра различны, то маршрут называется цепью, а если все вершины различны – простой цепью. В орграфе цепь называется путем (соответственно, элементами пути являются узлы и дуги). Замкнутая цепь называется циклом.

Расстоянием между двумя вершинами является цепь кратчайшей длины.

Если граф или орграф является взвешенным (то есть каждому ребру или дуге приписано число, называемое весом ребра), можно ввести понятие стоимости маршрута, как суммы весов всех ребер маршрута. Таким образом, кратчайшим путем в орграфе является путь минимальной стоимости.

Связность

Две вершины графа называются связанными, если существует соединяющая их цепь. Граф, в котором все вершины связаны, называется связным. Поиск компонент связности графа является важной задачей, решаемой при построении отказоустойчивых сетевых и вычислительных структур. Связность графов, введение мер связности, решение задач о потоках в сетях является довольно обширным материалом и обычно изучается в курсах дискретной математики и информатики, поэтому в данном курсе мы этот раздел опускаем.

Древовидный граф

Деревом называется связный граф без циклов. Пусть p – число вершин в графе G, а q – число ребер, Тогда для древовидного графа справедливо $q = p - 1$. Особое значение для решения многих задач имеет минимальное остовное дерево. Минимальным остовным деревом связного взвешенного графа называется дерево, состоящее из всех вершин графа, и некоторых его ребер, причем сумма весов ребер является наименьшей среди всех возможных остовных деревьев.

Отметим, что один и тот же граф может иметь несколько минимальных остовных деревьев.

Способы задания

Существуют различные способы задания графа для компьютерной обработки. Выбор конкретного способа зависит от специфики решаемой задачи, структуры алгоритма, от объема памяти, доступной для хранения информации о графе и от скорости выполнения конкретных операций по доступу к элементам определения графа. Рассмотрим некоторые из этих способов.

Матрица смежности $AdjMat$ размером $p \times p$ состоит из элементов $AdjMat[i][j]$ таких, что $AdjMat[i][j] = 1$, если вершина v_i является смежной с вершиной v_j. В противном случае, $AdjMat[i][j] = 0$. Объем памяти, необходимый для хранения данных в матрице смежности оценивается величиной $O(p^2)$. В общем случае, применение матрицы смежности можно считать разумным для сильно связных графов.

Для взвешенного орграфа матрицу смежности обычно определяют несколько иначе. Во взвешенном графе $AdjMat[i][j] = \infty$, если v_i не является смежной с вершиной v_j. Если же из узла v_i в узел v_j ведет дуга с весом w_{ij}, то $AdjMat[i][j] = w_{ij}$. В частности, $AdjMat[i][j] = 0$. Очевидно, что для неориентированного графа $w_{ij} = w_{ji}$, то есть $AdjMat[i][j] = AdjMat[j][i]$.

Матрица инцидентности представляет собой матрицу размером $p \times q$, в которой $IncMat[i][j] = 1$, если вершина v_i инцидентна ребру e_j. В противном случае, $IncMat[i][j] = 0$. Объем памяти, необходимый для хранения данных в матрице смежности оценивается величиной $O(p \cdot q)$. Очевидно, что если граф имеет много вершин, но относительно мало ребер, хранение его в виде матрицы инцидентности более экономично, чем использование матрицы смежности.

С точки зрения удобства использования при записи алгоритмов во многих случаях наиболее привлекательным оказывается способ хранения информации о графе в виде списков смежности, или списков примыканий. С точки зрения структур данных, используемых в программировании, списочная структура представляет собой массив из p указателей на списки смежных вершин. Для неориентированного графа затраты на хранение оцениваются величиной $O(p + 2 \cdot q)$. Для орграфа получим $O(p + q)$. Использование списков смежности позволяет хранить только информацию о реально

присутствующих ребрах или дугах, однако за это приходится платить накладными расходами, связанными с организацией списка. Требуется дополнительная память для указателей на элементы списка, кроме того, доступ к информации требует больше времени. Если требуется обеспечить возможность частых изменений структуры графа в ходе работы программы (например, добавление или удаление вершин и ребер), то использование списочных структур, безусловно, является, наиболее эффективным вариантом.

Простым (и во многих случаях – вполне приемлемым с точки зрения затрат) способом представления графа является массив ребер (или массив дуг для орграфа). Пример использования массива дуг иллюстрирует приводимая ниже реализация алгоритма поиска кратчайшего пути.

На рис. 5.1 приведена диаграмма некоторого неориентированного графа и способ его задания в форме списка смежности. Для удобства сопоставления рисунков и массивов C/C++ здесь и далее вершины графа именуются, начиная со значения 0.

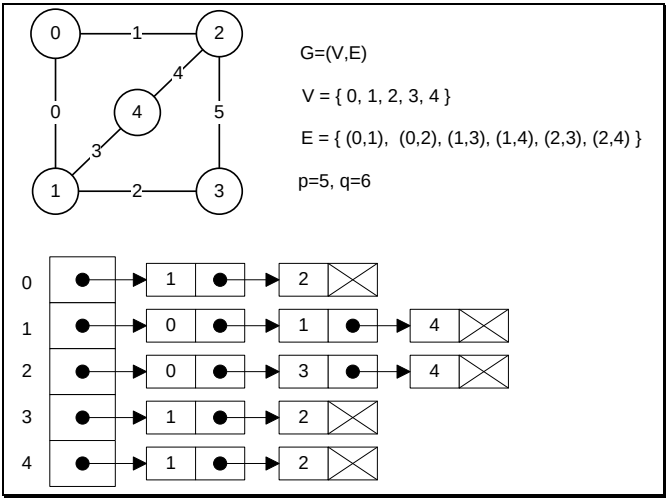


Рис. 5.1. Способы задания графа

Матрицы смежности и инцидентности данного графа имеют следующий вид:

$$AdjMat = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}, IncMat = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

5.2. Поиск кратчайшего пути на графе

Задачу поиска кратчайшего пути рассмотрим для случая взвешенного орграфа, заданного в виде массива дуг. Предположим, что каждая вершина характеризуется уникальным индексом (для конкретной задачи этот индекс одновременно может выступать в роли индекса массива, содержащего какие-либо сведения о вершине).

Структуры данных

Дуга графа представлена в программе следующей структурой:

```
struct Arc
{
    int    from;   // Индекс исходящей вершины
    int    to;     // Индекс входящей вершины
    double weight; // Вес дуги
};
```

Для представления информации о пути между вершинами, используем массив структур следующего вида:

```
struct Way;
{
    bool   exist;     // Признак существования пути (true, если
есть)
    double sumWeight; // Суммарный вес (стоимость пути)
    int    refPrev;   // Индекс предыдущего узла, через который
                      // проходит путь
};
```

Для представления графа и сведений о процесс поиска пути на графе определим следующую структуру.

```
struct Graph
{
    int numNodes; // Число вершин
    int numArcs;  // Число дуг
    Arc *arcs;    // Указатель на начало массива дуг

    Way *pMinWay; // Адрес первого элемента массива с информацией
                  // о минимальном пути от start до finish
                  //     pMinWay[ i ] хранит сведения о пути
                  //     минимальной стоимости из i в start
};
```

Реализация алгоритма

На основе введенных типов данных рассмотрим реализацию алгоритма, предложенного Э. Дейкстрой, в рекурсивной форме. В основе алгоритма Э. Дейкстры лежит следующая идея. Предположим, что путь от вершины start до некоторой вершины transit найден (см. рис. 5.2.).

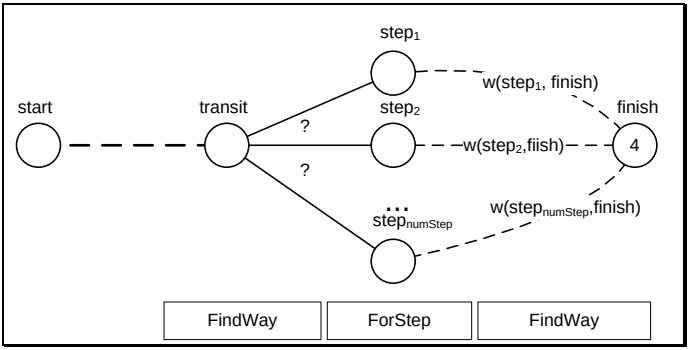


Рис. 5.2. Раскрытие вершин в ходе поиска кратчайшего пути

Рассмотрим множество вершин $Step = \{step_i\}, i \in \{1, numStep\}$, смежных с вершиной transit, причем $numStep$ – число смежных вершин. Очевидно, что кратчайший путь от вершины transit до вершины finish проходит через вершину $step_{\min}$, для которой достигается

$$\min_{i \in \{1, numStep\}} (w(transit, step_i) + sumW(step_i, finish)),$$

где $w(transit, step_i)$ – вес дуги между transit и $step_i$, а $sumW(step_i, finish)$ – вес минимального пути между $step_i$ и `finish`.

Таким образом, если путь до вершины `transit` известен, наилучший путь до `finish` выбирается путем рекурсивных вызовов функции нахождения кратчайшего пути и выбора соответствующей вершины $step_{\min}$. В начале `transit` совпадает с вершиной `start`. Алгоритм останавливается, когда `transit` совпадает с `finish`.

Конструктивно алгоритм можно реализовать в виде пары взаимно рекурсивных функций. Функция `FindWay` ищет кратчайший путь из заданной (промежуточной) вершины `transit` до вершины `finish`, осуществляет перебор вершин, связанных с `transit`, и вызывает функцию `ForStep`. Функция `ForStep` проверяет, не был ли раньше обнаружен путь, проходящий через выбранную смежную вершину, и вычисляет новую стоимость пути, вызывая `FindWay` для оценки пути от данной смежной вершины до `finish`. Реализацию этих функций иллюстрирует листинг 5.1.

В данном разделе мы рассчитываем на то, что заполнение структуры графа конкретными значениями и резервирование памяти для массивов `arcs` и `pMinWay` не составит труда для студента, имеющего элементарные навыки процедурного программирования, поэтому дальнейшее решение обсуждается в предположении, что вся подготовительная работа уже проведена.

Листинг 5.1. Поиск кратчайшего пути в графе (рекурсивный алгоритм Дейкстры)

```
void FindWay( Graph&, int, int ); // Предварительное объявление

// Шаг по дуге ixArc к вершине, смежной с вершиной transit
void ForStep( Graph& g, int ixArc )
{
    int transit = g.arcs[ ixArc ].from; // Транзитная вершина
    int step    = g.arcs[ ixArc ].to;   // Смежная вершина

    // Вычисляем новый суммарный вес пути через transit до step
    double newSumWeight = g.pMinWay[ transit ].sumWeight +
                          g.arcs[ ixArc ].weight;

    if( !g.pMinWay[ step ].exist )
    {
        // Эта вершина раскрыта впервые
        g.pMinWay[ step ].exist = true;

        g.pMinWay[ step ].sumWeight = newSumWeight;

        g.pMinWay[ step ].refPrev = transit;
        FindWay( g, step, finish );
    }
    else
    {
        // Эта вершина раньше встречалась
        if( g.pMinWay[ step ].sumWeight > newSumWeight )
        {
            // Новый путь короче
            g.pMinWay[ step ].sumWeight = newSumWeight;
            g.pMinWay[ step].refPrev = transit;
            FindWay( g, step, finish );
        }
        // Здесь: прежний путь лучше, то есть дальше искать
        // нечего, поэтому игнорируем этот шаг
    }
}

// Поиск кратчайшего пути от transit до finish
void FindWay( Graph& g,
              int transit,
              int finish
            )
{
    if( transit == finish ) return;

    // Находим дуги, исходящие из transit и
    // выполняем шаг по каждой найденной дуге
    for( int ixArc=0; ixArc < g.numArcs; ixArc++ )
    {
        // Рассматриваем случай орграфа, поэтому анализируем
        // только исходящую вершину дуги
        if( g.arcs[ ixArc ].from == transit )
            ForStep( g, ixArc );
    }
    return; // Больше путей нет
}
```

Отметим, что в листинге 5.1 приведен вариант решения для орграфа. В этом случае в функции `FindWay` в массиве дуг просматриваются только исходящие вершины. При этом функции `ForStep` достаточно передать только индекс дуги (индексы вершин извлекаются из соответствующего

элемента массива дуг). Если требуется обеспечить обработку неориентированных графов, то в реализацию следует внести описываемые ниже изменения.

Функции `ForStep` нужно передать не только индекс ребра, но и индексы вершин в явном виде, например:

```
// Для неориентированного графа
void ForStep( Graph& g,
              int ixArc,
              int transit, // транзитная вершина
              int step     // смежная вершина
             )
{ /* ... */ }
```

В цикле `for` функции `FindWay` следует проверять не только вершину `from`, но и вершину `to`:

```
// Поиск кратчайшего пути от transit до finish в версии
// для неориентированного графа
void FindWay( Graph& g,
              int transit,
              int finish
             )
{
    if( transit == finish ) return;

    // Находим ребра, инцидентные вершине transit и
    // выполняем шаг по каждому найденному ребру
    for( int ixArc=0; ixArc < g.numArcs; ixArc++ )
    {
        if( g.arcs[ ixArc ].from == transit )
            ForStep( g, ixArc, from, to );
        else if( g.arcs[ ixArc ].to == transit )
            ForStep( g, ixArc, to , from );
    }

    return; // Больше путей нет
}
```

Для запуска функции `FindWay` следует подготовить `pMinWay`:

```
// Поиск кратчайшего пути от g.start до g.finish
void FindBestWay( Graph& g, int start, int finish )
{
    for( ix ixNode=0; ixNode <g.numNodes; ixNode++ )
    {
        g.pMinWay[ ixNode ].exist = false;
    }

    g.pMinWay[ start ].exist = true;
    g.pMinWay[ start ].sumWeight = 0.0;
    g.pMinWay[ start ].refPrev = -1;

    FindWay( g, start, finish );
}
```

Для вывода искомого пути достаточно просмотреть содержимое массива `pMinWay` в порядке ссылок `refPrev` от вершины `finish` до вершины, у которой ссылка `refPrev` равна -1, например:

```
void PrintBestWay( FILE *output,
                   const Graph& g,
                   int start,
```

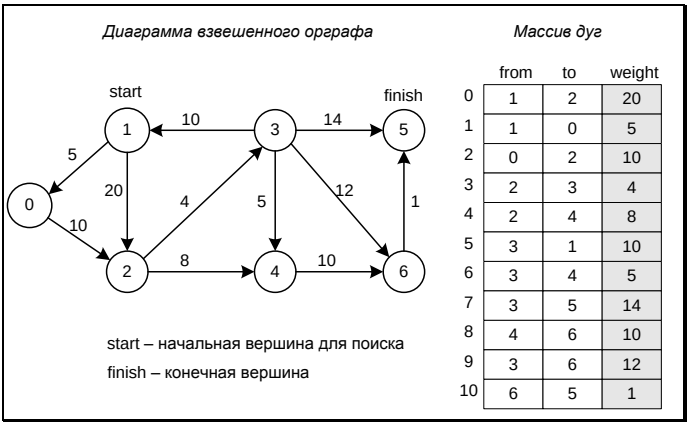
```
                          int finish
                         )
{
    fprintf( output, "Поиск пути от вершины %d до вершины %d\n",
                     start,
                     finish );

    if( !g.pMinWay[ finish ].exist )
    {
        fprintf( output, "Искомого пути не существует\n" );
        return;
    }
    fprintf( output, "Кратчайший путь имеет вес %le\n",
                     g.pMinWay[ finish ].sumWeight );
    fprintf( output, "Путь проходит через следующие вершины: " );

    int wayNode = finish;
    while( wayNode != -1 )
    {
        fprintf( output, "%d ", wayNode );
        wayNode = g.pMinWay[ wayNode ].refPrev;
    }

    fprintf( output, "\n";
}
```

Отметим, что для экономии стека при рекурсивных вызовах число параметров рекурсивных функций должно быть по возможности минимальным.



Массив дуг

	from	to	weight
0	1	2	20
1	1	0	5
2	0	2	10
3	2	3	4
4	2	4	8
5	3	1	10
6	3	4	5
7	3	5	14
8	4	6	10
9	3	6	12
10	6	5	1

Рис. 5.3. Орграф для иллюстрации работы алгоритма поиска кратчайшего пути

На ссылочной переменной `g` можно сэкономить, если определить переменную для представления графа на уровне модуля (см. [Давыдов, 2003]). Однако при этом комплект функций будет пригоден только для одновременной обработки одного графа. Окончательное решение зависит от специфики конкретной задачи.

Анализ работы алгоритма

Предположим, что задан орграф, изображенный на рис. 5.3.

Начальное состояние массива `pMinWay` и процесс его изменения в ходе обработки графа иллюстрирует рис. 5.4.

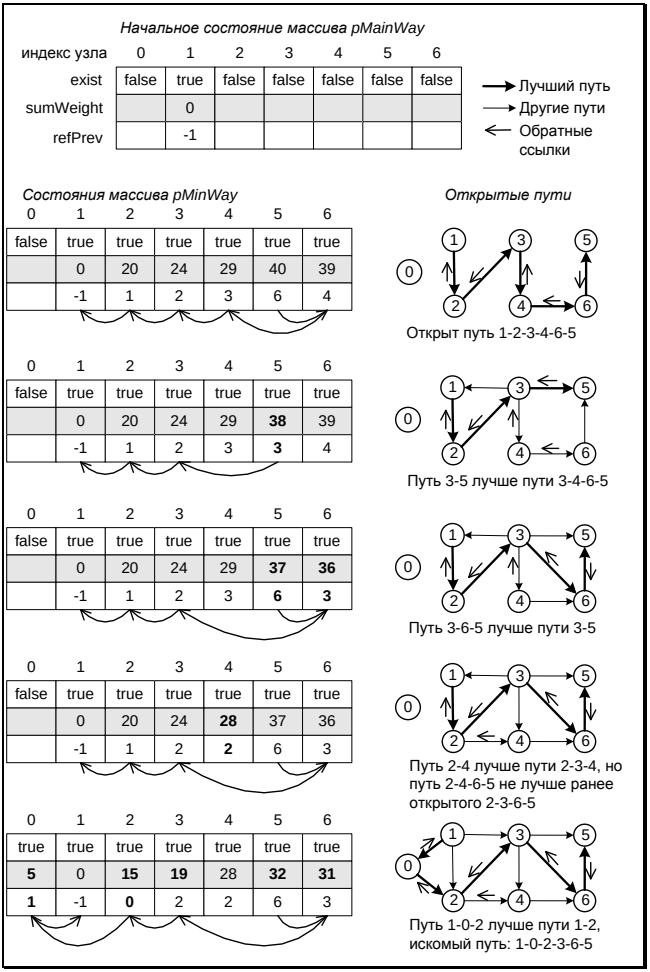


Рис. 5.4. Иллюстрация работы алгоритмаFindWay

Заметим, что в общем случае гарантируется только обнаружение кратчайшего пути от `start` до `finish`, хотя в ходе работы алгоритма могут быть раскрыты и некоторые другие кратчайшие пути.

5.3. Поиск минимального остовного дерева

Наиболее простым для изучения алгоритмом построения минимального остова графа, является, пожалуй, алгоритм Краскала. Данный алгоритм является представителем так называемых жадных алгоритмов (то

есть алгоритмов, которые в каждый отдельный момент используют лишь часть исходных данных и принимают решение на основе анализа этой части).

Пусть исходный граф G задан в виде списка ребер E. Требуется найти множество ребер T, определяющее искомый минимальный остов.

Перед началом построения минимального остова массив ребер упорядочивается по возрастанию. В процессе работы алгоритма ребра добавляются к множеству T в порядке возрастания. Если ребра закончатся раньше, чем все вершины графа G будут соединены, это означает, что исходный граф несвязный, а полученное множество является объединением минимальных остовных деревьев компонентов связности графа G.

Алгоритм, реализующий описанную процедуру, в псевдокоде выглядит следующим образом (листинг5.2).

Листинг 5.2. Алгоритм поиска минимального остовного дерева (псевдокод)

```
// Поиск минимального остова
//    Функция возвращает true, если минимальный остов построен,
//    иначе функция возвращает false
bool FindSST( const Graph& g,  // Исходный граф
              Arc *sst,        // Множество ребер минимального
остова
              int& numArcsSST  // Число ребер минимального остова
                               // (результат работы функции)
             )
{
    Arc *sst = new Arc[ g.numNodes-1 ]; // В минимальном остове
                                        // число ребер на единицу
                                        // меньше числа вершин

    numArcsSST = 0;                   // Текущее число ребер
                                      // в начале равно нулю

    // Сначала сортируем массив ребер исходного графа
    // (используйте любой известный вам алгоритм сортировки)
    SortArcs( g.arcs, g.numArcs );

    int ixArc = 0; // Индекс ребра в исходном графе

    // Цикл построения минимального остова
    for( int i = 1; i < g.numNodes; i++ )
    {
        while( ixArc < g.numArcs )
        {
            // Здесь осуществляется попытка добавить очередное
ребро
            // к минимальному остову.
            // Функция AddEdge возвращает false, если при
добавлении
            // ребра в sst образуется цикл,
            // иначе sst дополняется новым ребром с индексом
            // numArcsSST, а функция AddEdge возвращает true
            if( AddEdge( sst, numArcsSST, g.arcs[ ixArc ] )
            {
                numArcsSST++;
                break;
```

```
            }
            ixArc++; // Пропускаем ребро
        }
    }
    // Остается проверить, все ли ребра в sst были построены
    if( numArcsSST != g.numNodes - 1 ) return false;

    return true;
}
```

Функции `SortArcs` и `AddEdge` читателям предлагается написать самостоятельно.

Процесс поиска минимального остова иллюстрирует рис. 5.5. В качестве исходного графа используется вариант на рис. 5.3 с отменой ориентации дуг.

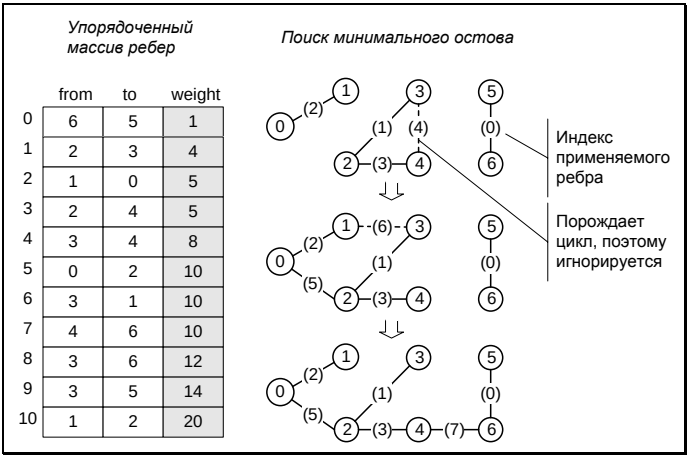


Рис. 5.5. Иллюстрация работы алгоритма FindSST

Отметим, что рассмотренные алгоритмы, разумеется, являются не единственными вариантами решения указанных задач. Сведения о других алгоритмах можно почерпнуть из разнообразной литературы по теории алгоритмов (см., например, [McConnell, 2001], [Давыдов, 2003], [Новиков, 2000] и др.).

Глава 6. Перемешанные таблицы и ассоциативные массивы

Одним из вариантов решения задачи ускорения поиска информации по ключу является использование так называемых перемешанных таблиц, или hash-таблиц (от англ. hash – путаница, беспорядок). Hash-таблицы являются также стандартным вариантом реализации ассоциативных массивов.

6.1. Алгоритм поиска по ключу с использованием hash-функций

В основе алгоритма лежит использование специальной функции, называемой hash-функцией.

Понятие hash-функции

Пусть задано некоторое функциональное преобразование $h : \mathrm{K} \to \mathrm{N}$, где K – множество ключей, а N – множество натуральных чисел, причем значения функции h для разных ключей могут совпадать. Рассмотрим сравнение двух ключей k_1 и k_2. При этом возможны два случая:

$h(k_1) \neq h(k_2)$. В этом случае очевидно, что $k_1 \neq k_2$.

$h(k_1) = h(k_2)$. В этом случае требуется дополнительное сравнение ключей.

Наиболее «продуктивен» именно первый случай, так как он создает основу для быстрого обнаружения факта отсутствия ключа в таблице (как показано в лекции 4, в алгоритме бинарного поиска именно выяснение отсутствия ключа требует наибольшей работы).

Заполнение hash-таблицы

Пусть hash-функция спроектирована таким образом, что порождает значения из некоторого конечного подмножества множества N, например, в диапазоне чисел от 0 до $m-1$, где $m > n$, а $n = |\mathrm{K}|$ – число ключей. На практике m обычно выбирается не меньше, чем $\approx 3 \cdot n$.

Определим таблицу поиска как массив записей, где каждая запись содержит информацию о ключе и указатель на данные. Тогда заполнение таблицы поиска (которая содержит m записей «ключ + ссылка на данные») можно организовать следующим образом. Значение hash-функции (hash-адрес) для каждого ключа из множества K будем трактовать как индекс строки таблицы, в которую следует поместить очередной ключ. Если данная строка таблицы уже заполнена (то есть ключ с таким hash-адресом уже встречался), возникает так называемый конфликт hash-адресов, или **коллизия**. В этом случае очередной ключ помещается в ближайшую в порядке роста индекса

свободную строку таблицы. Учитывая, что $m > n$, такая свободная строка всегда найдется (рис. 5.6).

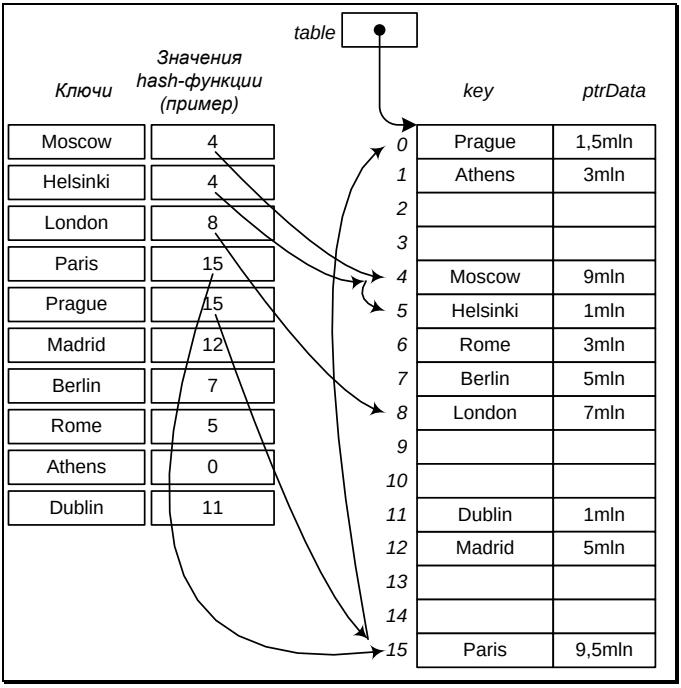


Рис. 6.1. Hash-поиск: заполнение таблицы

Метод заполнения таблицы указанным образом называется разрешением коллизий с открытой адресацией линейным апробированием [Knuth, 1973]. На рис. 6.1 процесс разрешения конфликтов показан стрелками для первых пяти ключей. Как видно из рисунка, в процессе заполнения таблицы некоторые образующиеся группы соседних записей могут «склеиваться». В дальнейшем нельзя будет, основываясь только на информации о таблице, определить, соответствует ли группа соседних записей ключам с одинаковыми hash-адресами, или она является результатом такой «склейки» (впрочем, это и не потребуется). Алгоритм заполнения таблицы в псевдокоде иллюстрируется листингом 6.1.

Листинг 6.1. Заполнение hash-таблицы (псевдокод)

```
struct HashTableLine // Строка hash-таблицы
{
    KeyType  key;      // Ключ
    DataType *ptrData; // Указатель на объект данных,
                       // связанных с данным ключом
};

int hash( KeyType );   // Прототип hash-функции

void PrepareHashTable( HashTableLine *table, // Hash-таблица
```

```
                                int m                   // Размер
        таблицы
                        )
{
    int numKeyLines = 0; // Число заполненных строк таблицы
                         // (не должно быть >= m)

    while( NextKeyIsExist() )
    {
        if( numKeyLines >= m )
        {
            // Обработка ошибки переполнения таблицы
            // ...
        }

        KeyType key = GetNextKey();

        int hashAddress = hash( key );

        while( !LineIsEmpty( table[ hashAddress ] )
        {
            // Увеличение на 1 по модулю m:
            //    гарантирует переход от строки с индексом m-1
            //    к строке с индексом 0
            hashAddress = (hashAddress + 1) % m;
        }

        table[ hashAddress ].key = key;
        table[ hashAddress ].ptrData = GetNextData()

        numKeyLines++;
    }
}
```

В результате заполнения получаем таблицу, в которой группы записей с ключами, соответствующими одинаковым hash-адресам, разделены пустыми строками таблицы. Чем более равномерно «размещены» группы в таблице и чем меньше длина каждой группы, тем лучше hash-функция (она порождает меньше конфликтов и выдает hash-адреса, равномерно «размазанные» по таблице). Косвенным признаком неравномерности hash-функции является большое число склеек групп, происходящее в ходе заполнения таблицы.

Поиск в подготовленной таблице

Процесс поиска в таблице, подготовленной таким образом, иллюстрирует рис. 6.2. Отметим, что на рисунках приводятся гипотетические значение hash-функции.

Листинг 6.2 содержит набросок функции, реализующей алгоритм hash-поиска.

Листинг 6.2. Поиск в hash-таблице (псевдокод)

```
// Поиск ключа key в подготовленной hash-таблице
// Результат, передаваемый через список параметров:
//    индекс элемента в массиве
// Возвращает true, если ключ найден,
//             false - в противном случае
bool HashSearch( HashTableLine *table, // Hash-таблица
                 int m,                // Размер hash-таблицы
                 KeyType key,          // Исходный ключ
                 int& ixFound          // Ссылка на искомый индекс
```

```
                )
{
    int hashAddress = hash( key );

    while( !LineIsEmpty( table[ hashAddress ] ) )
    {
        if( table[ hashAddress ].key == key )
        {
            ixFound = hashAddress;
            return true;
        }

        // Увеличение на 1 по модулю m:
        //    гарантирует переход от строки с индексом m-1
        //    к строке с индексом 0
        hashAddress = (hashAddress + 1) % m;
    }

    ixFound = -1;
    return false;
}
```

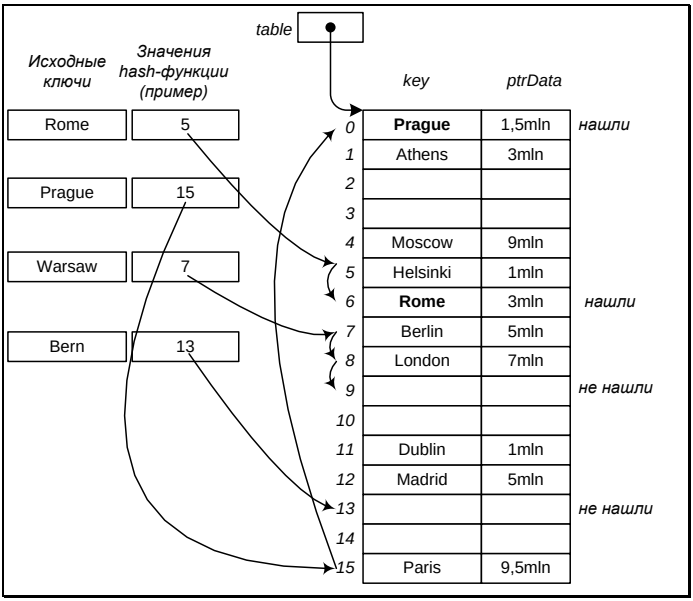


Рис. 6.2. Поиск в hash-таблице

Очевидно, что эффективность поиска напрямую зависит от качества hash-функции: чем меньше размер групп соседних непустых строк таблицы, тем быстрее заканчивается процесс поиска. Разумеется, на эффективность поиска влияет и размер таблицы. Однако даже для сильно избыточной таблицы качество поиска может оказаться очень низким, если hash-функция будет выдавать близкие или совпадающие значения для различных ключей. Наиболее исчерпывающий анализ эффективности hash-поиска для различных

стратегий разрешения конфликтов, а также анализ качества hash-функций, читатель найдет в [Knuth, 1973]. Так, для метода разрешения коллизий с линейным апробированием Кнут приводит следующие оценки эффективности:

Среднее число сравнений при неудачном поиске: $C_{cp}^{(нет)} = \frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$, где $\alpha = \frac{n}{m}$ – коэффициент заполнения таблицы.

Среднее число сравнений при удачном поиске: $C_{cp}^{(да)} = \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$.

С реализацией алгоритмов hash-поиска для конкретных типов ключей и данных можно познакомиться, например, в [Давыдов, 2003], [Niemann, 1995] и др.

Удаление элементов из hash-таблицы

Отметим, что необходимость помещения в hash-таблицу новых ключей не влечет обязательной перестройки всей таблицы. Наоборот, удаление ключей из таблицы представляет известную сложность, поскольку просто пометить элемент таблицы как свободный нельзя из-за возможной потери последующих ключей. Пример алгоритма удаления ключей из hash-таблицы при линейном апробировании рассматривается в [Knuth, 1973].

В терминах, использованных в листингах 6.1 – 6.2, процедура удаления заданной строки таблицы выглядит следующим образом (листинг 6.3).

Листинг 6.3. Удаление заданной строки из hash-таблицы (псевдокод)

```
// Удаление записи из hash-таблицы с восстановлением целостности
void DeleteLine( HashTableLine *table, // Hash-таблица
                 int m,                // Размер hash-таблицы
                 int ixDelete          // Индекс удаляемой записи
               )
{
    // Индексы, используемые при восстановлении целостности
таблицы
    int  ixLine = ixDelete; // Индекс текущей обозреваемой строки
    int  ixEmpty;           // Индекс последней строки, помеченной
                            //    как свободная

    for( ; ; )
    {
        // Отметить строку таблицы как свободную
        MarkLineEmpty( table[ ixLine ] );
        ixEmpty = ixLine;

        bool noConflicts = true;
        while( noConflicts )
        {
            ixLine = (ixLine+1) % m;
```

```
            // Работа завершается, если нашли свободную запись
            if( LineIsEmpty( table[ ixLine ] ) return;

            int hashAddress = hash( table[ ixLine ].key );

            // Проверка выполнения следующего условия:
            //    hashAdress циклически располагается
            //    между ixDelete и ixLine, то есть
            //    записи после ixDelete образовались не
            //    в результате склейки.
            if( (ixDelete< hashAddress && hashAddress <= ixLine)
||
                (ixLine < ixDelete && ixDelete < hashAddress) ||
                (hashAddress <= ixLine && ixLine < ixDelete) )
            {
                continue; // Перейти к следующей записи
            }
            noConflicts = false;
        }
        // Здесь: обнаружена запись, которую нужно присоединить
        //        к группе, прервавшейся на месте удаления,
        //        то есть переместить запись с индексом ixLine
        //        на место ixEmpty
        MoveLine( table, m, ixEmpty, ixLine );
    }
}
```

Заметим, что данный алгоритм корректен только для таблиц, при заполнении которых для разрешения конфликтов используется линейное апробирование. В [Knuth, 1973] показано, что используемый в данном алгоритме процесс восстановления целостности таблицы не вызывает ухудшения приведенных выше показателей эффективности поиска.

6.2. Распознавание служебных слов из фиксированного набора

В задачах синтаксическим анализа и компиляции часто оказывается, что распознаваемыми ключами являются служебные слова некоторого языка. Их число обычно не очень велико и не изменяется.

Задачей, постоянно решаемой в ходе трансляции, является задача распознавания служебных слов: требуется определить, является прочитанный идентификатор служебным словом или нет. Поскольку число идентификаторов в компилируемой программе может быть довольно большим, и только часть из них является зарезервированными, эффективное и компактное решение проблемы распознавания служебных слов является важнейшей задачей.

Если число служебных слов невелико (например, не превышает три-четыре десятка), при разработке соответствующей части синтаксического анализатора можно попытаться реализовать упрощенный и более компактный вариант hash-поиска. Единственное условие применимости этого варианта состоит в возможности написания hash-функции, выдающей различные значения для всех служебных слов в приемлемом диапазоне (например, в диапазоне 0..255 для трех-четырех десятков служебных слов).

Если подобную hash-функцию удается сконструировать, то для распознавания служебных слов можно использовать упрощенную инфраструктуру данных, представленную на рис. 6.3.

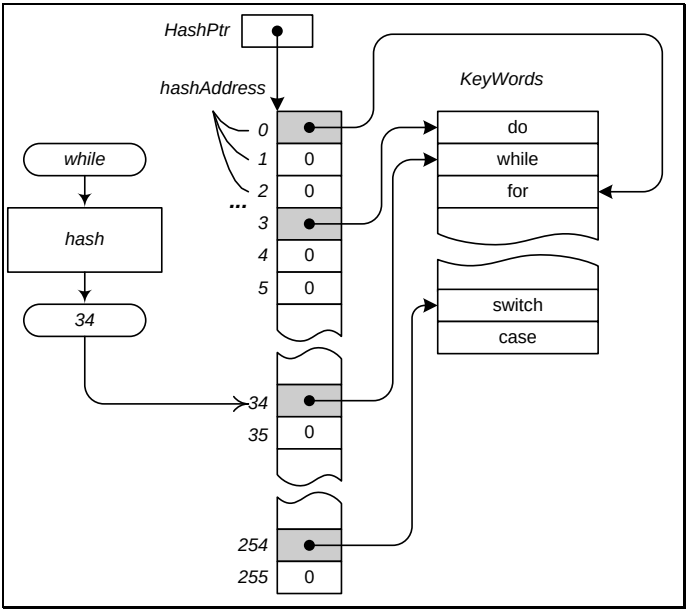


Рис. 6.3. Использование hash-функций для распознавания служебных слов

Значение hash-адреса трактуется как индекс массива указателей на элементы набора зарезервированных слов, размер которого определяется диапазоном значений hash-функции.

В ходе предварительной подготовки в ячейку массива указателей, соответствующую hash-адресу зарезервированного идентификатора, записывается адрес места в памяти, где хранится данный идентификатор. Остальным указателям присваивается значение 0. Набросок реализации соответствующих структур данных и функций иллюстрируется листингом 6.4.

Листинг 6.4. Использование hash-функции для распознавания зарезервированных идентификаторов (подготовка)

```
const int HashLimit = 255;  // Предельное значение hash-функции
const int NumKeyWords = 40; // Число зарезервированных
                            // идентификаторов (служебных слов)

// Массив служебных слов
static const char *KeyWords[ NumKeyWords ] =
{
    "do",
    "while",
    "for",
    ...
};
```

```
// Массив указателей на служебные слова
static const char *HashPtr[ HashLimit + 1 ];

int hash( const char* ); // Прототип hash-функции

// Подготовка массива указателей на служебные слова
// в соответствии со значениями, генерируемыми hash-функцией
void PrepareHashPtrs()
{
    for( int ixHash = 0; ixHash <= HashLimit; ixHash++ )
    {
        HashPtr[ ixHash ] = 0;
    }

    for( int ixWord = 0; ixWord < NumKeyWords; ixWord++ )
    {
        int hashAddress = hash( KeyWords[ ixWord ] );

        HashPtr[ hashAddress ] = &KeyWord[ ixWord ];
    }
}
```

Распознавание служебных слов в этом случае осуществляется не более чем за одно сравнение. Для прочитанного идентификатора определяется hash-адрес. Если соответствующий элемент массива указателей содержит 0, следовательно, идентификатор не является служебным словом. В противном случае (поскольку возможен конфликт hash-адреса прочитанного идентификатора с hash-адресом служебного слова) следует осуществить одно сравнение (см. листинг 6.5).

Листинг 6.5. Использование hash-функции для распознавания зарезервированных идентификаторов (сравнение)

```
#include <string.h>

// Проверка: идентификатор является служебным словом?
bool IdentIsKeyWord( const char* ident )
{
    int hashAddress = hash( ident );

    if( HashPtr[ hashAddress ] == 0 )
        return false;

    if( strcmp( ident, HashPtr[ hashAddress ] ) != 0 )
        return false;

    return true;
}
```

Если не удается разработать hash-функцию, генерирующую уникальные значения hash-адресов, или требуется обеспечить возможность настройки решения на изменения в составе служебных слов, то с элементами массива указателей можно связывать цепочки указателей на служебные слова (рис. 6.4).

Данный метод Кнут называет разрешением коллизий методом цепочек [Knuth, 1973]. В этом случае добавление и удаление ключей сводится к стандартным операциям над линейным списком (см. материал лекции 3).

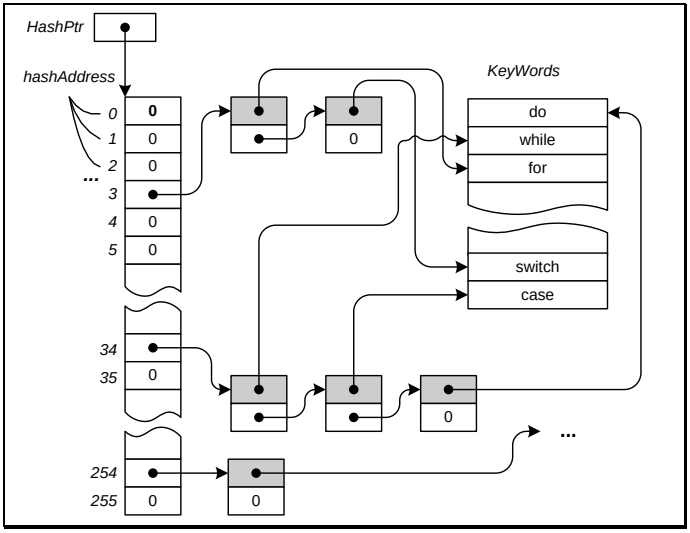


Рис. 6.4. Разрешение коллизий методом цепочек

6.3. Ассоциативные массивы

Hash-таблицы являются основным инструментом построения ассоциативных массивов. В [Sebesta, 2002] ассоциативный массив определяется как множество элементов данных, индексированных таким же количеством величин, называемых ключами. Если в обычных массивах индексы идут по порядку и не требуют специального представления в структуре массива, то в ассоциативном массиве ключи определяются пользователем и должны быть явно определены.

Многие современные языки содержат реализацию ассоциативных массивов либо в виде встроенных элементов языка (Perl, Visual Basic), либо в виде стандартных библиотечных средств (C#, C++).

Например, в стандартной библиотеке C++ концепцию ассоциативного массива с уникальными ключами реализует класс `map`. Листинг 6.6 представляет пример использования класса `map` для организации информации о товаре, хранящемся на некотором складе. Данный набросок реализации написан, исходя из предположения, что каждый товар характеризуется уникальным именем и количеством единиц хранения на складе.

Листинг 6.6. Использование ассоциативного массива map

```
#include <map>
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

void ReadItems( map<string,int>& );
```

```
void PrintSummary( map<string,int>& );

void main()
{
     map<string,int> goods;    // Информация о товарах на складе

     ReadItems( goods ); // Прочитать сведения о товарах
     PrintSummary( goods ); // Распечатать сводную таблицу
}

// Считывание сведений о товарах
void ReadItems( map<string,int>& goods )
{
     string name;    // Наименование товара
     int amount;     // Количество единиц товара

     cout << "Enter next item's name and amount:";
     for( ; ; )
     {
          cin >> name;
          if( name == "stop" ) break;

          cin >> amount;
          if( amount >= 0 )
          {
               cout << "Accepted" << endl;
               goods[ name ] += amount;
                    // NB! В начале элемент массива map
                    // инициализируется нулем,
                    // поэтому использование операции
                    // += корректно
          }
          else cout << "Skippied. "
                    "Amount value should be positive" << endl;

          cout << "Enter next item's name and amount:";
     }
}

// Печать сводной таблицы
void PrintSummary( map<string,int>& goods )
{
     typedef map<string, int>:: const_iterator MapConstIter;

     int totalAmount = 0;

     for( MapConstIter it = goods.begin(); it!=goods.end(); ++it )
     {
          totalAmount += it->second;

          // Печать наименования товара
          cout << left << setw( 12 ) << it->first;

          // Печать суммарного количества этого товара на складе
          cout << '\t' << setw( 10 ) << right << it->second;
          cout << endl;
     }

     cout << "-------------------------------------------------"
<< endl;
     cout << left << setw( 12 ) << "Total" << '\t' << setw( 10 )
<< right << totalAmount << endl;
}
```

В процессе формирования массива goods для каждого ключа обеспечивается вычисление суммарного количества единиц товара на складе.

Иногда одному и тому же ключу может соответствовать несколько записей. В этих случаях можно создать ассоциативный массив map, используя в качестве типов объектов хранения контейнерные типы (например, вектора или списки). Однако стандартная библиотека предоставляет альтернативное (и, вероятно, более простое в использовании решение), основанное на применении класса multimap (ассоциативного массива с дубликатами). Листинг 6.7 представляет пример работы с ассоциативным массивом с дубликатами на примере создания записей телефонного справочника (очевидно, что каждому абоненту может соответствовать несколько телефонных номеров).

Листинг 6.7. Использование ассоциативного массива с дубликатами

```
#include <map>
#include <iostream>
#include <string>

using namespace std;

typedef string PhoneNumber;    // Пока предположим, что
                               // телефонный номер
                               // представлен в виде
                               // строки символов

// Длина предельного размера текстуального
// представления телефонного номера
#define PHONE_NUM_TXT_LEN 30

void ReadPhones( multimap<string,PhoneNumber>& phoneBook );
void PrintNumbers( multimap<string,PhoneNumber>& phoneBook, const
string& name );

void main()
{
     multimap<string,PhoneNumber> phoneBook;  // Телефонный
                                              // справочник

     // Загрузить информацию о новых номерах
     ReadPhones( phoneBook );

     string checkName; // Имя для поиска в справочнике
     cout << "Enter name to review numbers:";
     cin >> checkName;

     // Распечатать все телефоны для заданного имени
     PrintNumbers( phoneBook, checkName );
}

// Составление телефонного справочника
void ReadPhones( multimap<string,PhoneNumber>& phoneBook )
{
     string name; // Имя абонента
     PhoneNumber number; // Номер

     cout << "Enter name and phone number:";
     for( ; ; )
```

```
    {
        cin >> name;
        if( name == "#" ) break;

        cin.ignore(1);

        char buf[ PHONE_NUM_TXT_LEN ];
        cin.getline( buf, PHONE_NUM_TXT_LEN, '\n' );
        if( !cin )
        {
            cin.clear();

            char temp;
            do { cin.get( temp ); } while( temp != '\n' );
        }

        number.assign( buf );

        // Для multimap операция [] не определена,
        // поэтому используется метод insert
        phoneBook.insert( make_pair( name, number ) );
        cout << "Enter name and phone number:";
    }
}

// Печать списка найденных номеров для заданного имени
void PrintNumbers( multimap<string,PhoneNumber>& phoneBook, const
string& name )
{
    typedef multimap<string, PhoneNumber>::const_iterator
        MultimapConstIter;

    pair<MultimapConstIter,MultimapConstIter> rangePair =
        phoneBook.equal_range( name );
    if( rangePair.first == phoneBook.end() )
    {
        cout << "No such name" << endl;
        return;
    }

    MultimapConstIter it = rangePair.first;
    for( MultimapConstIter it = rangePair.first;
        it!=rangePair.second;
        ++it )
    {
        cout << it->second << endl;
    }

}
```

Выводы

Нетрудно заметить, что рассмотренный нами алгоритм hash-поиска и предоставляет механизм обращения к данным по значению некоторого ключа. Неслучайно внутренняя реализация ассоциативного массива очень часто основана на использовании алгоритмов хеширования.

Глава 7. Элементы лексического и синтаксического анализа

При решении задач, связанных с обработкой текста (а таких задач не так уж и мало), выделяют два относительно независимых этапа. Первый этап – **лексический анализ** – состоит в распознавании в исходном тексте отдельных лексем, получаемых объединением соседних символов в соответствии с лексическими правилами языка. Второй этап – **синтаксический и семантический анализ** – заключается в распознавании приложений языка, сконструированных из лексем по синтаксическим правилам языка. Лексический и синтаксический анализатор являются основными компонентами любого компилятора (рис. 7.1).

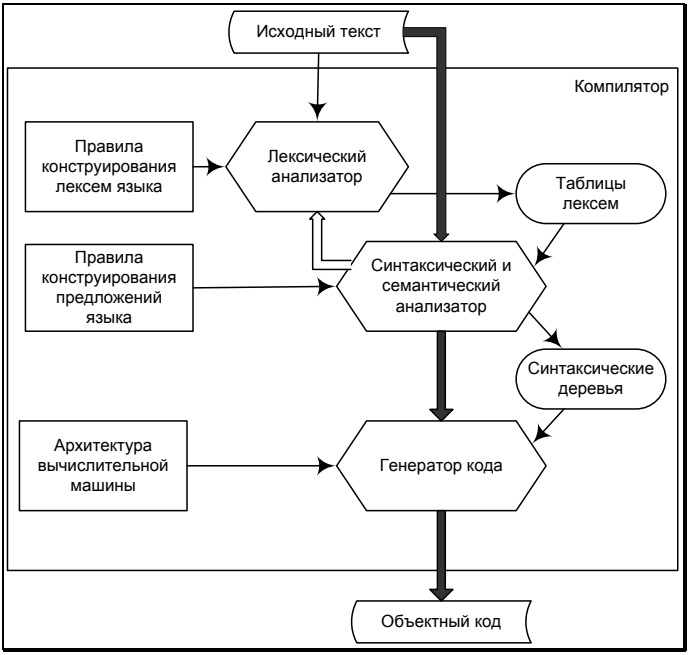


Рис. 7.1. Компоненты компилятора

Из курса программирования на языке высокого уровня студентам известно о синтаксически-ориентированной трансляции компьютерных программ. Синтаксически-ориентированная трансляция предполагает, что смысл программных конструкций однозначно выводится транслятором на основе правил синтаксиса. Синтаксический анализ текстов на формальных языках является непростой задачей, характеризующейся существенной структурной сложностью. За счет выделения этапов лексического и синтаксического анализа эта сложность может быть сильно уменьшена [Aho, Ullman, 1972].

Элементарный пример лексического анализатора был представлен на первой лекции в форме конечного автомата, распознающего идентификаторы. На этой лекции мы исправим некоторые недостатки, присущие ранее рассмотренным решениям.

7.1. Постановка задачи

В качестве примера рассмотрим задачу распознавания логических выражений [Лекарев, Пышкин, 2005]. Задача синтаксического разбора логических и арифметических выражений относится к числу типовых задач синтаксического анализа и компиляции, поэтому исследование решения этой задачи в методических целях преподавания интересно и само по себе, и как средство оценки преимуществ, предоставляемых используемыми моделями организации данных и вычислительного процесса. В рамках учебного пособия мы отдаем преимущество именно логическим выражениям в связи с меньшим числом операций и, как следствие, большей компактностью примера. Впрочем, информация, представленная в этой главе, нетрудно обобщается и на случай арифметических выражений.

Рассмотрим скобочную (факторизованную) форму записи логических выражений, синтаксис которых определяется правилами контекстно-свободной грамматики $E = (\mathbf{VN}, \mathbf{VT}, \mathbf{P}, \mathbf{S})$, где **VN** – множество нетерминальных символов грамматики, **VT** – множество терминальных символов, **S** – начальный символ грамматики, **P** – множество правил грамматики. Пусть правила грамматики E выглядят следующим образом (элементы множества терминальных символов выделены полужирным начертанием и заключены в кавычки или апострофы):

```
S ::= факторизованная-ЛФ
факторизованная-ЛФ ::= имя-ЛФ "=" выражение ";"
имя-ЛФ ::= идентификатор
выражение ::= терм [ "+" терм ]…
терм ::= фактор [ "*" фактор ]…
фактор ::= { [ "~" ] имя-переменной | двоичная-константа | "("
             выражение ")" }
имя-переменной := идентификатор
идентификатор ::= латинская-буква [ { латинская-буква |
десятичная-цифра } ] ...
латинская-буква ::= {'_'|'A'|'B'|...|'Z'|'a'|'b'|...|'z'}
десятичная-цифра ::= {'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'}
двоичная-константа ::= {'0'|'1'}
```

Знак + соответствует операции логического сложения (дизъюнкции), знак * соответствует операции логического умножения (конъюнкции), знак ~ соответствует операции логического отрицания.

Синтаксические диаграммы, определяющие нетерминал `выражение`, приведены на рис. 7.2.

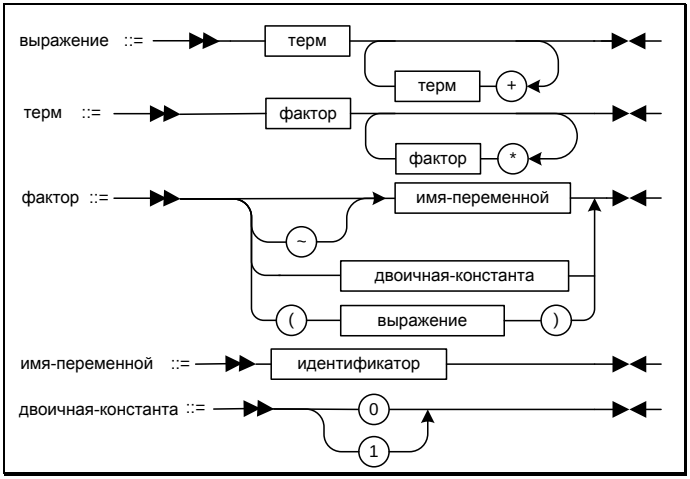


Рис. 7.2. Синтаксис логических выражений

Привычная для человека инфиксная запись логических выражений не вполне удобна для анализа формулы на ЭВМ. Результатом синтаксического анализа выражения, записанного по сформулированным правилам, будем считать преобразованное выражение, не содержащее скобок, в котором операции следуют в том порядке, в каком должны выполняться.

Примером такой записи является постфиксная польская запись, в которой знак операции следует за операндами. Например, для выражения

```
result=start*((x1 + x2)*ready + user1*user2)
```

соответствующее текстуальное представление в польской записи будет иметь следующий вид:

```
result start x1 x2 + ready * user1 user2 * + * =
```

Поскольку в рамках решаемой задачи конкретная форма представления результата синтаксического анализа является непринципиальной, будем считать польскую запись сформированной, если ее элементы сгенерированы в нужном порядке (например, напечатаны в некотором выходном потоке).

Выражение, получаемое в постфиксной форме записи, можно представить в виде синтаксического дерева, содержащего два типа вершин: элементы-переменные (или константы), элементы-операции (рис. 7.3).

7.2. Лексический анализ

На этапе лексического анализа из текста выделяются последовательности литер, имеющие самостоятельный синтаксический смысл – лексемы (tokens). Основная сложность реализации лексического анализатора (сканера) заключается в определении принадлежности считываемой литеры множеству литер, используемых для записи лексем определенной природы (например, множество букв, множество десятичных

цифр, множество разделителей и т.д.). С использованием принятой в теории синтаксического анализа и компиляции терминологии такая задача обычно называется задачей определения синтерма литеры.

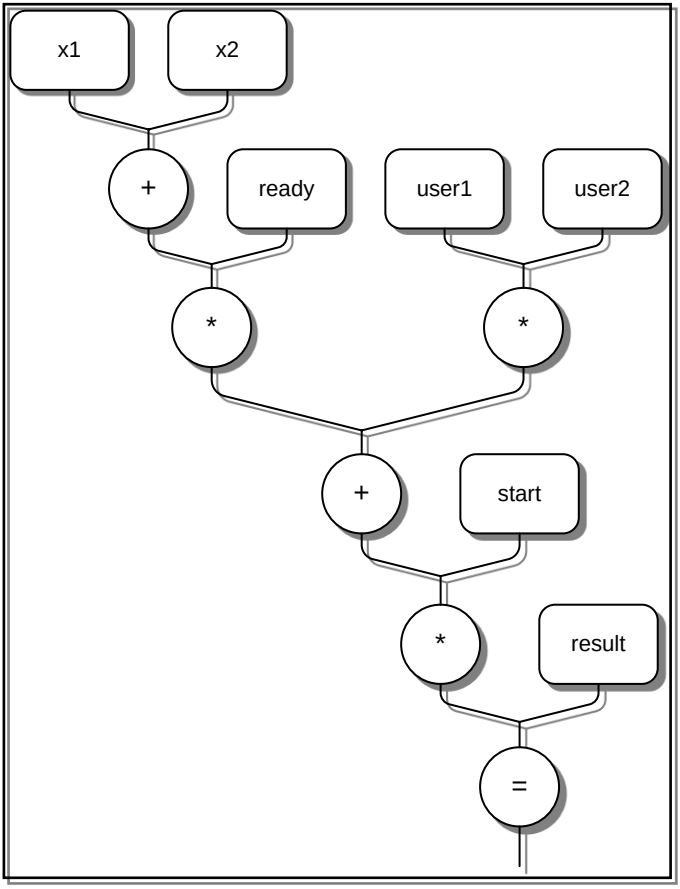


Рис. 7.3. Синтаксическое дерево логического выражения

Решение, представленное в гл. 1 (см. реализацию функции `GetLit`, листинг 1.5), не слишком конструктивно, поскольку для определения синтерма в общем случае приходится осуществить несколько сравнений и вызвать несколько стандартных функций. Очевидно, требуется более «скоростное» решение.

Сразу же отметим и более общую проблему: некоторые литеры могут принадлежать сразу нескольким синтермам (например, литера `'A'` является одновременно латинской буквой и шестнадцатеричной литерой).

Понятие синтерма: непересекающиеся и пересекающиеся синтермы

В соответствии с синтаксисом логических выражений нужно формально определить разновидности синтермов (мы будем называть их классами синтермов), которые должен распознавать лексический анализатор. Это позволит уменьшить количество условий, которые необходимо проверить в ходе лексического анализа. Например, для распознавания первого символа в составе идентификатора, который должен являться латинской буквой, можно использовать следующий синтерм:

```
<латинская-буква> ::= {'A','B',...,'Z','a','b',...,'z'}
```

Таким образом, число рассматриваемых альтернатив может быть сокращено с 52 до 1. При необходимости несложно обеспечить возможность работы с синтермами, включающими прочие буквы (греческие, русские и т.д.). Так, представляемая далее таблица синтермов обеспечивает, в частности, определение синтерма `<русская-буква>`.

Для распознавания остальных литер в составе идентификатора наибольшее сокращение количества сравнений достигается при определении синтерма `<буква-или-цифра>`, который пересекается с синтермом `<латинская-буква>` по составу литер (и, фактически, объединяет множество десятичных цифр и множество латинских букв). Заметим, что использование классов лексем с пересечениями позволяет интерпретировать, например, литеру `'0'` и как десятичную цифру (при распознавании идентификатора), и как двоичную константу.

С теоретических позиций, множество синтермов $\mathbf{S_T}$ - это множество всех подмножеств **VT**. Однако на практике обычно требуется значительно меньшее количество синтермов $\mathbf{S_P}$, так что $\mathbf{S_P} \subseteq \mathbf{S_T}$, $|\mathbf{S_P}| << |\mathbf{S_T}|$ [Лекарев, 1997]. Теперь, исходя из правил грамматики, выделим следующие две разновидности нетерминалов:

Нетерминальные символы грамматики, определяемые непосредственно через синтермы, т.е. такие $V_i \in \mathbf{VN}$, что $V_i ::= P_t(V_j)$, $V_j \in \mathbf{S_T}$, где $P_t(V_s)$ означает, что символ V_s содержится в правой части правила P_t, для которого справедливо: $P_t \in \mathbf{P}$.

Одиночные литеры, которые образуют такие подмножества $\mathbf{S_{Ti}}$ множества синтермов $\mathbf{S_T}$, что $|\mathbf{S_{Ti}}| = 1$.

Эти две группы элементов образуют множество **классов лексем**, формируемых сканером и распознаваемых синтаксическим анализатором. Так, в грамматике *E* в это множество входят нетерминалы `<идентификатор>` и `<двоичная-константа>`, а также множество однолитерных лексем: `';'`, `'+'`, `'*'`, `'~'`, `'('`, `')'`, `'='`.

Формирование и распознавание синтерма

Для использования в таблице синтермов определим следующие универсальные классы синтермов, значения которых представлены двоичными кодами, являющимися степенями двойки, то есть имеющие ровно один

единичный разряд (в листинге 7.1 приводятся определения констант, сгруппированные в виде перечисления).

Листинг 7.1. Классы синтермов

```
/*
 * Классы синтермов
 */
enum LitClass
{
    NOALP = 0x0000,     // NOn ALPhabetic :
                        //не входящая в алфавит

    // В связи с идентификаторами:
    LAT   = 0x0001,     // LATin : латинская буква
    LD10  = 0x0002,     // Latin or Digit_10 : латинская буква
                        //     или десятичная цифра
    RUS   = 0x0004,     // RUSsian : русская буква
    LRD10 = 0x0008,     // Latin or Russian or Digit_10:
                        //     латинская или русская буква
                        //     или десятичная цифра

    // В связи с константами в разных системах счисления:
    D10   = 0x0010,     // Digit_10 : десятичная цифра
    D16   = 0x0020,     // Digit_16 : шестнадцатеричная цифра
    D0    = 0x0040,     // Digit_0  : "нуль" (также логический)
    D1    = 0x0080,     // Digit_1  : "единица"
                        //            (также логическая)
    X     = 0x0800      // латинская литера 'X' или 'x'
};
```

Для представления подмножества непересекающихся синтермов (образуемого однолитерными лексемами), определим следующие значения:

```
// В связи с классами без пересечений:
//    из одной или нескольких допустимых
//    литер алфавита, которые входят в единственный класс:
enum LexIsLitClass
{
    ONLY     = 0x8000,      // ONLY :
                            // признак класса без пересечений
    A        = ONLY | '\'', // Apostrophe : апостроф
    BLANK    = ONLY | ' ',  // BLANK :
                            // "пробельная" литера
                            // (разделитель)
    LF       = ONLY | '\n', // Line Finish : конец строки
    LPARENT  = ONLY | '(',  // Left PARENThese  : левая скобка
    RPARENT  = ONLY | ')',  // Right PARENThese : правая скобка
    NEGATE   = ONLY | '~'   // NEGATion opEration : отрицание
};

// Можно использовать и другие непересекающиеся синтермы:
// (для них не вводится самостоятельных имен)
// (логическое) умножение  ONLY | '*'
// (логическое) сложение   ONLY | '+'
// точка с запятой         ONLY | ';'
// присваивание            ONLY | '='
```

При использовании языка C вместо перечисления можно использовать константы препроцессора.

Признак `ONLY` обеспечивает установку в единицу старшего разряда значения синтерма, что в предложенных терминах определяет непересекающийся синтерм, значение которого однозначно соответствует

значению кода литеры, образующей однолитерную лексему. Ниже приводятся фрагменты кода, позволяющие получить представление о том, как выглядит таблица синтермов в нашем случае (листинг 7.2).

Листинг 7.2. Таблица синтермов

```
typedef               // Тип: синтерм литер анализируемого текста
    unsigned int
    TSynterm [256];   // Type of SYNtax TERM

// Таблица синтермов
TSynterm synterm =
{
    // Управляющие и служебные символы (не имеют графических
очертаний)
    /* ..0  0x00        ^@  NUL */   LF,
    /* ..1  0x01        ^A  SOH */   BLANK,
    /* ..2  0x02        ^B  STX */   BLANK,
    /* ..3  0x03        ^C  ETX */   BLANK,
    //...
    // Символы, имеющие графические очертания
    /* .32  0x20  ' '  SP SPace */   BLANK,
    /* .33  0x21  '!'  */            NOALP,
    /* .34  0x22  '\"' */            NOALP,
    /* .35  0x23  '#'  */            NOALP,
    /* .36  0x24  '$'  */            LAT  | LD10 | LRD10,
    /* .37  0x25  '%'  */            NOALP,
    /* .38  0x26  '&'  */            NOALP,
    /* .39  0x27  '\'' */            NOALP,
    /* .40  0x28  '('  */            LPARENT,
    /* .41  0x29  ')'  */            RPARENT,
    /* .42  0x2A  '*'  */            ONLY | '*',
    /* .43  0x2B  '+'  */            ONLY | '+',
    // ...
    /* .48  0x30  '0'  */   D10  | LD10 | LRD10 | D0 | D16,
    /* .49  0x31  '1'  */   D10  | LD10 | LRD10 | D1 | D16,
    /* .50  0x32  '2'  */   D10  | LD10 | LRD10 | D16,
    // ...
    /* .57  0x39  '9'  */   D10  | LD10 | LRD10 | D16,
    /* .58  0x3A  ':'  */   NOALP,
    /* .59  0x3B  ';'  */   ONLY | ';',
    /* .60  0x3C  '<'  */   NOALP,
    /* .61  0x3D  '='  */   ONLY | '=',
    /* .62  0x3E  '>'  */   NOALP,
    /* .63  0x3F  '?'  */   NOALP,
    /* .64  0x40  '@'  */   LAT  | LD10 | LRD10,
    /* .65  0x41  'A'  */   LAT  | LD10 | LRD10 | D16,
    /* .66  0x42  'B'  */   LAT  | LD10 | LRD10 | D16,
    // ...
    /* .90  0x5A  'Z'  */   LAT  | LD10 | LRD10,
    /* .91  0x5B  '['  */   NOALP,
    /* .92  0x5C  '\\' */   NOALP,
    /* .93  0x5D  ']'  */   NOALP,
    /* .94  0x5E  '^'  */   NOALP,
    /* .95  0x5F  '_'  */   LAT  | LD10 | LRD10,
    /* .96  0x60  '`'  */   NOALP,
    /* .97  0x61  'a'  */   LAT  | LD10 | LRD10 | D16,
    /* .98  0x62  'b'  */   LAT  | LD10 | LRD10 | D16,
    // ...
    /* 122  0x7A  'z'  */   LAT  | LD10 | LRD10,
    // ...
    /* 126  0x7E  '~'  */   ONLY | '~',
    // ...
    /* 128  0x80  'A'  */   RUS  | LRD10,
```

```
    /* 129  0x81  'Б'  */   RUS  | LRD10,
    // ...
    /* 159  0x9F  'Я'  */   RUS  | LRD10,
    /* 160  0xA0  'а'  */   RUS  | LRD10,
    /* 161  0xA1  'б'  */   RUS  | LRD10,
    // ...
    /* 174  0xAE  'о'  */   RUS  | LRD10,
    /* 175  0xAF  'п'  */   RUS  | LRD10,
    /* 176  0xB0 '-'   */   NOALP,
    /* 177  0xB1   '-' */   NOALP,
    // ...
    /* 224  0xE0  'р'  */   RUS  | LRD10,
    /* 225  0xE1  'с'  */   RUS  | LRD10,
    // ...
    /* 239  0xEF  'я'  */   RUS  | LRD10,
    /* 240  0xF0  'Ё'  */   NOALP,
    /* 241  0xF1  'ё'  */   NOALP,
    // ...
    /* 255  0xFF  ' '  */   NOALP
};
// Конец инициализации таблицы синтермов
```

Список инициализации таблицы синтермов с пересечениями имеет довольно большой объем, поэтому его уместно оформлять в виде самостоятельного файла.

Отметим, что единожды определенная таблица синтермов просто перестраивается для самых разнообразных задач синтаксического анализа. При этом для определения принадлежности литер синтерму удобно работать с масками синтермов (листинг 7.3).

Листинг 7.3. Маски синтермов

```
enum LitClassMask
{
    MASK_LAT    ONLY | LAT,
    MASK_LD10   ONLY | LD10,
    MASK_RUS    ONLY | RUS,
    MASK_LRD10  ONLY | LRD10,
    MASK_D10    ONLY | D10,
    MASK_D16    ONLY | D16,
    MASK_D0     ONLY | D0,
    MASK_D1     ONLY | D1,
    MASK_X      ONLY | X
};
```

При наличии таблицы синтермов определение синтерма прочитанной литеры происходит очень просто:

```
litera.synterm = synterm[ litera.value ];
```

Формирование значения синтерма (на примере латинской литеры `'A'`) и последующее его распознавание иллюстрирует рис. 7.4. Отметим, что в рассматриваемой постановке задачи синтермы LRD10 и D16 нами не используются.

Ограничение данного подхода связано с размером целочисленной константы, биты которой используются для определения синтерма. Так, при использовании двухбайтовых целых можно обеспечить представление не более 15 классов синтермов с пересечениями и не более чем 2^{15} синтермов для представления однолитерных лексем с установленным битом `ONLY` (в

действительно, такое огромное число однолитерных лексем на практике не требуется). Для большинства практических приложений 15 вариантов синтермов с пересечениями оказывается достаточно. Если лексическая структура языка столь сложна, что двухбайтового представления недостаточно, то можно использовать в качестве базового типа константы типа `unsigned long` – это увеличивает множество представимых синтермов с пересечениями до 31.

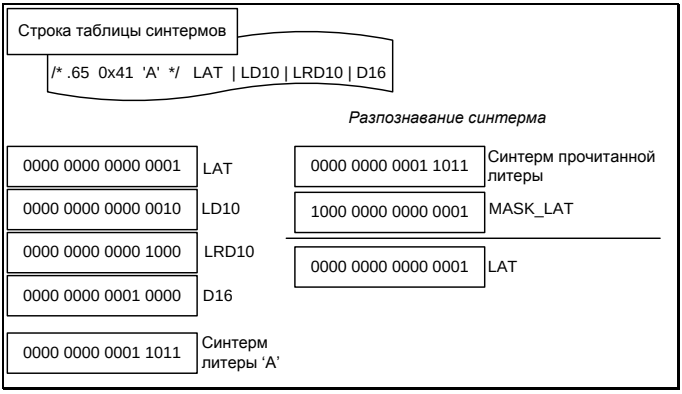


Рис. 7.4. Формирование и распознавание синтерма

Результатом однократного вызова сканера является очередная лексема входного текста, для которой выявлен класс лексемы. После распознавания синтермов конструирование лексем не представляется сложной задачей – обычно она может быть решена посредством комплектов специальных функций (см. материал гл. 1).

7.3. Использование визуального формализма на базе L-сети в методических целях преподавания

М.Ф. Лекарев предложил формализм, названный им L-сеть, который ориентирован на решение широкого класса задач [Лекарев, 1997]. Помимо чисто практических аспектов, данный формализм предоставляет довольно удобный инструментарий, пригодный для целей преподавания. Визуальный язык L-сетей зарекомендовал себя как методически удобная форма представления многих алгоритмов, в том числе при описании решения задач синтаксического анализа.

В основе визуальной модели программирования на базе L-сети лежит два комплекта примитивов, один из которых ориентирован на представление программы в форме сети автоматов (сеть разветвленного управления), а другой – на представление программы в форме иерархической сети функциональных модулей (сеть последовательного управления). В качестве функционального модуля может выступать отдельный фрагмент сети, имеющий явно определенную точку входа (такой функциональный модуль

называется сетевой программой) или операционный модуль. Операционные модули представляют собой процедуры, разрабатываемые на конечном языке программирования (например, на C++).

Среда последовательного управления

В рамках L-сети используется модель функционального модуля с двумя выходами, которая, как показано в работе [Лекарев, 2000-2], хорошо отвечает запросам практики.

Действительно, даже в стандартной библиотеке C многие функции (Лекарев приводит оценку в 60%) определены таким образом, что могут завершиться успехом или неуспехом (например, открытие файла, резервирование динамической памяти, считывание данных из файла и т. д.).

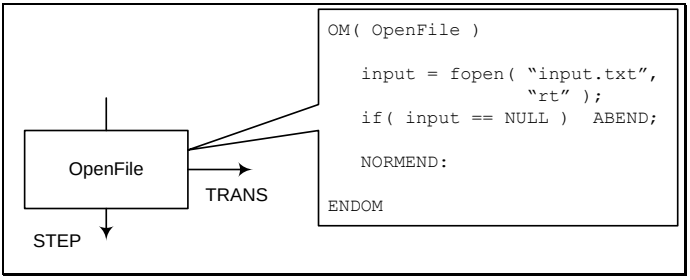


Рис. 7.5. Варианты завершения операционного модуля

Для подобных действий абстракция модуля с двумя выходами оказывается как нельзя более пригодной. В модели L-сети определены следующие способы завершения операционных модулей (рис. 7.5):

- Завершение с шагом (Step). В среде последовательного управления этот вид завершения операционного модуля (изображаемый внизу графического примитива) также называется успешным (NORMEND, от англ. Normal End).
- Завершение с переходом по дуге сети (Trans, от англ. transition). В среде последовательного управления этот вид завершения (изображаемый справа или слева) также называется неуспешным завершением (ABEND, от англ. abnormal end).

На рис. 7.6 представлена реализация алгоритма нерекурсивной сортировки Хоара в форме L-сети, идентичная по организации вычислительного процесса алгоритму, рассмотренному на лекции 2. Отметим наглядный характер обращения к тем функциональным модулям, для которых используются два способа завершения. Схема управления вычислительным процессом столь наглядна, что, вероятно, не требует пояснений.

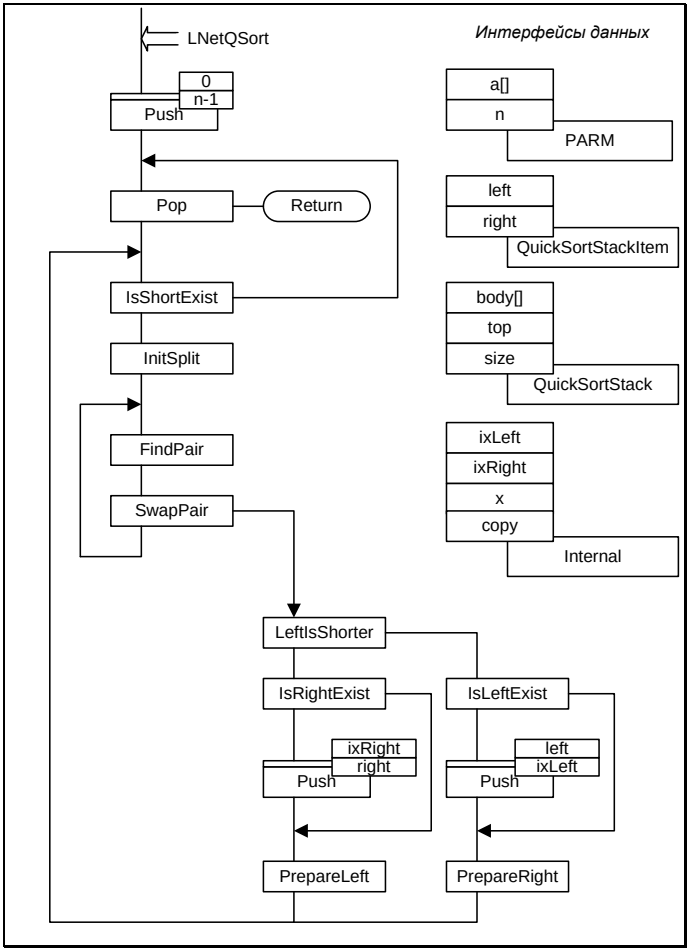


Рис. 7.6. Сортировка Хоара в форме L-сети

Приведенный на рис. 7.6 фрагмент L-сети называется сетевой программой. Сетевая программа представляет собой функционально обособленный сегмент L-сети, имеющий хотя бы одну поименованную точку входа (может быть и больше) и завершающуюся одним из двух способов (рис. 7.7):

- Возврат управления с шагом по сети (SReturn), который в среде последовательного управления можно также называть возвратом с успехом (Return).

- Возврат управления с переходом (TReturn), который в среде последовательного управления можно также называть возвратом с неуспехом (AbReturn).

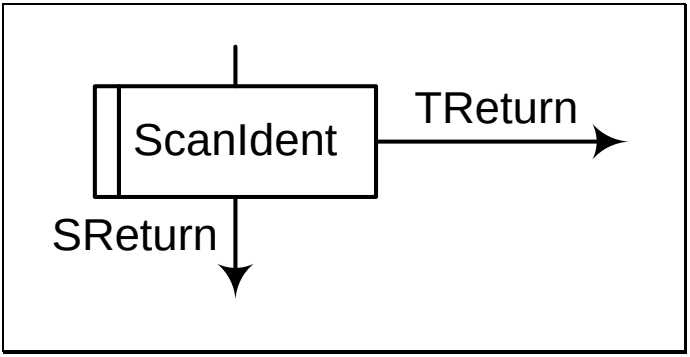


Рис. 7.7. Варианты завершения сетевой программы

Среда разветвленного управления

Основным элементом модели разветвленного управления L-сети является дуга L-сети (рис. 7.8). Дуга L-сети является обобщением (generalization) дуги графа переходов конечного автомата и состоит из трех основных элементов:

- обособленное условие (возможно, маскируемое) – CondS (CONDition Special);
- функциональный модуль, вызываемый при истинности обособленного условия;
- переход по дуге (Transition).

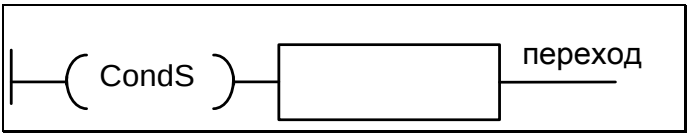


Рис. 7.8. Дуга L-сети

Совокупность нескольких расположенных друг под другом дуг сети (см. рис. 7.9) позволяет реализовать разветвления, свойственные многим задачам синтаксического анализа. При этом вертикальная линия сложного ветвления является геометрическим образом состояния конечного автомата [Лекарев, 1997].

Представленный на рис. 7.9. алгоритм считывания идентификатора является комбинацией двух ветвлений по результатам считывания очередной литеры. В начале идентификатора должна быть латинская буква. Здесь используется маскируемое условие, эквивалентное следующей проверке:

```
litera.value & MASK_LAT == LAT
```

Очередная литера идентификатора должна быть латинской буквой или десятичной цифрой. Здесь используется маскируемое условие, эквивалентное следующей проверке:

```
litera.value & MASK_LD10 == LD10
```

При этом используется те же константы, что и при формировании таблицы синтермов, рассмотренной в первой части лекции.

Результатом однократного вызова сканера является очередная лексема входного текста, формируемая функциями `LexFirst`, `LexNext` и `LexStop` (реализация которых идентична функциям из проекта, рассмотренного в гл. 1).

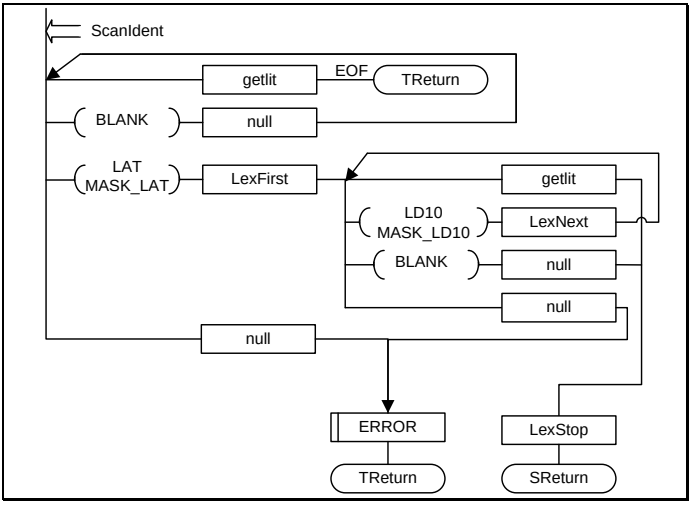


Рис. 7.9. Сетевая программа считывания идентификатора

7.4. Решение задачи синтаксического анализа логических выражений в форме L-сети

Синтаксический анализатор руководствуется правилами синтаксиса (см. рис. 7.2) и пытается «сконструировать» правильное предложение языка из лексем, формируемых лексическим анализатором. Таким образом, несмотря на то, что этапы лексического и синтаксического анализа вполне самостоятельны, синтаксический и лексический анализатор работают в тесной кооперации.

Задача синтаксического анализа характеризуется значительной логической сложностью, поскольку связана с проверкой большого числа условий. В большой степени логическую сложность синтаксического анализа позволяет преодолеть использование визуального формализма на основе L-сети [Лекарев, 1997].

Лексический анализатор в форме L-сети

По сравнению с рис. 7.9 возможности лексического анализатора применительно к задаче распознавания лексем, допустимых в логических выражениях, должны быть расширены:

- Необходимо обеспечить занесение считанных имен переменных (идентификаторов) в таблицу имен.
- Необходимо обеспечить возможность распознавания однолитерных лексем.

Полученная сетевая программа приведена на рис. 7.10. Результатом однократного вызова сканера является очередная лексема входного текста, формируемая, как и раньше, функциями `LexFirst`, `LexNext` и `LexStop`, однако процесс завершения регистрации очередной многолитерной лексемы включает также этап занесения новых имен в таблицу идентификаторов (сетевая программа `LexFinish`).

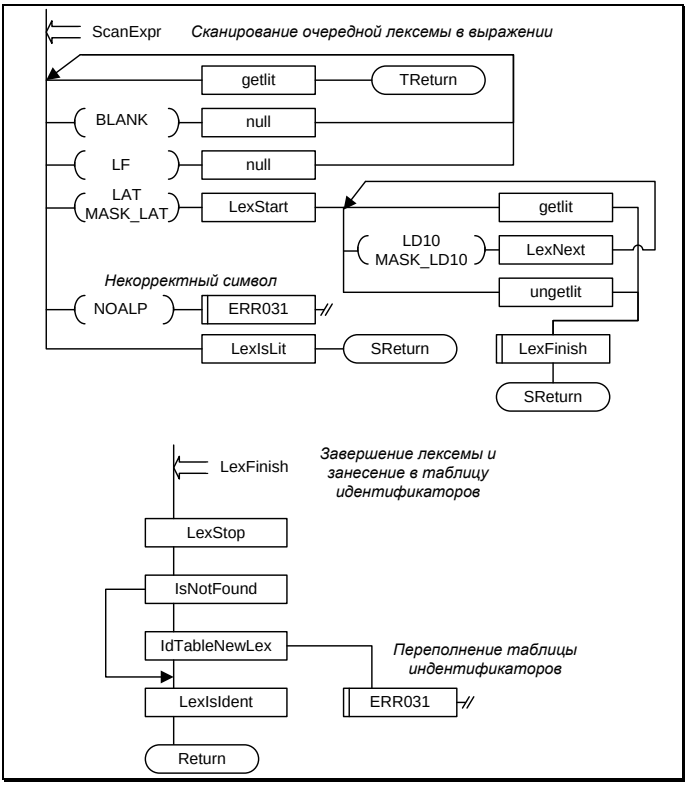


Рис. 7.10. Сетевые программы ScanExpr и LexFinish

В сетевой программе `ScanExpr` успешному завершению работы сканера соответствует возврат с шагом по L-сети (SReturn), что позволяет по результатам работы сканера строить сложные ветвления при анализе

соответствия текста выражения объявленному синтаксису. Возврат с переходом (TReturn) соответствует ситуации конца файла.

Опыт использования формализма на базе L-сети в учебном процессе, позволяет заключить, что даже те люди, которые не знакомы с точной спецификацией исследуемого формализма, могут легко разобраться в алгоритмах решений логически сложных задач проектирования, представленных в форме фрагментов L-сетей [Лекарев, Пышкин, 2005]. В конечном счете, сетевые программы можно рассматривать как способ записи алгоритма, обеспечивающий большую наглядность и понятность по сравнению со многими другими формализмами (в частности, по сравнению со схемами, определяемыми стандартом [ГОСТ 19.701-90]).

На основе представленных здесь рисунков несложно разработать эквивалентную реализацию на языке C++, не использующую специфических элементов среды проектирования на базе L-сетей. В этом случае и сетевые программы, и операционные модули могут быть реализованы в виде обычных функций C++.

Синтаксический анализатор в форме L-сети

Схема решения задачи синтаксического анализа логического выражения, представлена на рис. 7.12. Видно, что структура анализатора графически довольно точно соответствует синтаксическим диаграммам, представленным на рис. 7.2.

Вывод элементов польской записи в выходной поток осуществляется следующими сетевыми программами (которые могут быть реализованы в виде обычных функций C++ или даже в виде методов класса, воплощающего абстракцию польской записи):

- `OutVar` – помещение в польскую запись идентификатора, соответствующего переменной в составе выражения;
- `OutConst` – помещение в польскую запись двоичной константы в составе выражения;
- `OutConj` – помещение в польскую запись операции логического умножения;
- `OutDisj` – помещение в польскую запись операции логического сложения;
- `ExprFinish` – помещение в польскую запись служебной операции завершения польской записи.

Реализация этих функциональных модулей не представляет проблем. На рис. 7.12 соответствующие фрагменты L-сети изображены с затемнением.

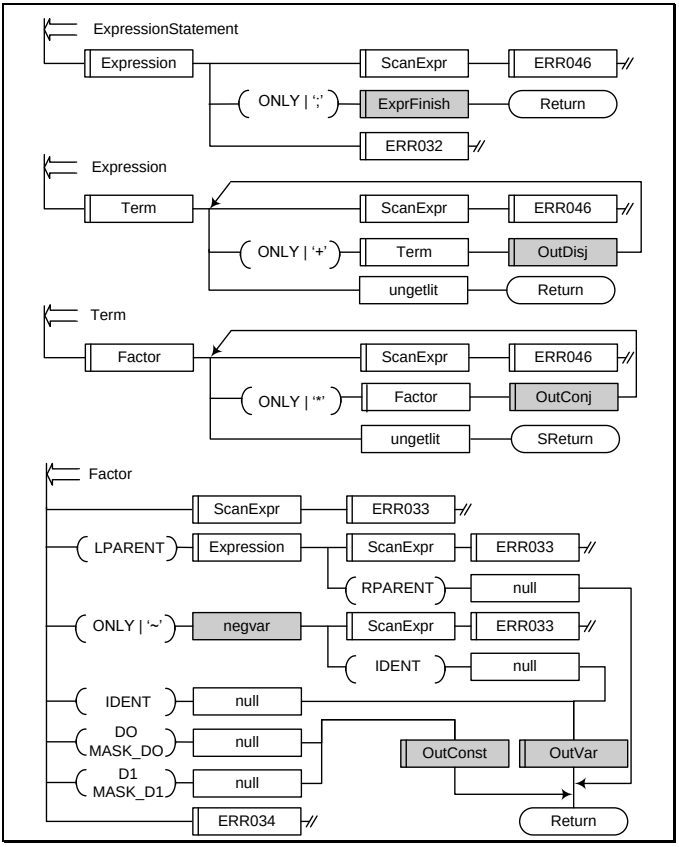


Рис. 7.12. Синтаксический анализ выражений (L-сеть)

Элементы реализации на языке C++

В заключение отметим, что вопреки сложившейся практике использования классических схем программ (когда соответствующая схема, если это необходимо, неизменно рисуется после разработки программы, см. [Brooks, 1995] и материал гл. 1), мы, действительно, сначала разработали детальные алгоритмы решения задачи, и только затем реализовали их средствами конечного языка проектирования.

Глава 8. Алгоритмы обработки контейнеров

Одним из очевидных применений шаблонных классов является проектирование так называемых контейнерных (вмещающих) классов, обеспечивающих различные варианты хранения наборов объектов (стеки, очереди, hash-таблицы, векторы и т. д.). Основные операции над контейнером – это добавление и извлечение элемента из него. Некоторые контейнеры поддерживают последовательный просмотр хранящихся в них элементов. Страуструп выделяет два традиционных типа: контейнеров специализированные и стандартные [Stroustrup, 2000]. Рассмотрим последовательно оба этих типа.

8.1. Специализированные контейнеры

В основе проектирования специализированных контейнеров лежит проектный принцип «одна хорошо спроектированная функция решает одну задачу». При этом за скобками остается вопрос о совместимости алгоритмов, обрабатывающих контейнеры, между собой. Таким образом, унификация использования различных контейнеров приносится в жертву удобству и простоте работы с конкретными контейнерами.

В качестве примера рассмотрим два вида специализированных контейнерных классов: вектор и стек. Для контейнера-вектора добавление и извлечение элементов может быть реализовано через операцию индексирования `[]`, а для контейнера-стека традиционное управление данными, хранящимися в стеке, обеспечивается парой операций push/pop:

```
template <class T>
class Vector {
public:
    explicit Vector( int size );
    T& operator[]( int index );
    // ...
};

template <class T>
class Stack {
public:
    explicit Stack( int size );
    void push( const T& item );
    T& pop();
    // ...
};
```

Таким образом, интерфейсы специализированных контейнеров различны.

Итерируемые специализированные контейнеры

Обычное использование многих видов контейнеров – это просмотр содержимого контейнера, или итерация. Набор действий, соответствующих итерации, может отличаться в зависимости от типа контейнера: для массива итерация предполагает увеличение (или уменьшение) на 1 индекса массива, а для списка – переход по значению указателя, связывающего элементы списка.

Несмотря на это различие, пользователю контейнеров было бы удобно иметь возможность написать унифицированный код перебора элементов контейнеров разных типов. Одним из классических объектно-ориентированных механизмов, позволяющих решить эту задачу, является механизм наследования. В этом случае задача сводится к определению общего интерфейса для осуществления итераций по контейнеру (в C++ для этого нужно определить абстрактный класс, содержащий только чисто виртуальные методы):

```
template <class T>
class Iterator {
public:
    virtual T* first() = 0;
            // Получить адрес первого элемента
    virtual T* next() = 0;
            // Получить адрес следующего элемента
};
```

При разработке контейнерного типа, унаследованного от `Iterator`, следует реализовать виртуальные методы `first` и `next`. Если это сделано, реализация обобщенной функции, обрабатывающей любой контейнер, поддерживающий интерфейс `Iterator`, не представляет труда, например:

```
// Поиск элемента в итерируемом контейнере
template <class T>
const T* FindItem( Iterator<T> *itCont, // Адрес контейнера,
                                        // поддерживающего
интерфейс
                                        // Iterator<T>
                   const T& item        // Элемент для поиска
                 ) {

    const T *ptr = itCont->first(); // Получить адрес первого
                                    // элемента контейнера

    while( ptr!= 0 ) { // Пока есть элементы…

        if( *ptr == item ) return ptr; // Если это искомый элемент
-
                                       // завершаем работу

        ptr = itCont->next(); // Переходим к следующему элементу
    }

    return 0; // Возвращает 0, если не нашли
}
```

Если контейнер содержит данные некоторого пользовательского типа, не поддерживаемого языком напрямую, для правильной работы обобщенного алгоритма `FindItem` должна быть определена семантика сравнения двух элементов пользовательского типа, то есть должна быть определена перегруженная операция отношения ==.

Разработка итерируемого специализированного контейнера

По существу, принципы построения специализированных контейнеров были рассмотрены в гл. 3 при построении внутреннего итератора списка с внешним управлением.

Приведем еще один пример. Для этого рассмотрим шаблонное определение класса, реализующего абстракцию очереди (листинг 8.1).

Листинг 8.1. Определение шаблонного класса Queue

```
// Типы исключений, генерируемых методами класса Queue
class BadSizeException {};
class PutFullException {};
class GetEmptyException {};

// Определение шаблонного класса для представления очереди
template <class T>
class Queue          // Стандартная очередь:
                     // "первым пришел – первым обслужен"
{
protected:
    enum
    {
        EMPTY = -1,   // Признак пустоты очереди
        MAXSIZE = 100 // Предельный размер очереди
    };

    T *body;                // Массив для хранения данных
    int size;               // Предельный размер
                            // инициализированной очереди
    int length;             // Текущая длина очереди

    int last;               // Индекс последнего занесенного
                            // элемента очереди
    int next;               // Индекс извлекаемого
                            // элемента очереди

public:
    // Конструктор обеспечивает создание очереди заданного размера
    // NB! Конструктор может генерировать исключения
    //     BadSizeException и bad_alloc
    Queue( int actualSize = MAXSIZE )
        throw( BadSizeException, bad_alloc );

    // Занесение элемента в очередь
    // NB! Метод может генерировать исключение PutFullException
    void Enqueue( const T& ) throw ( PutFullException );
    // Извлечение элемента из очереди
    // NB! Метод может генерировать исключение GetEmptyException
    T Dequeue() throw ( GetEmptyException );

    // Печать текущего содержимого очереди
    void Print( ostream& );
};

// Конструктор
template <class T>
Queue <T>::Queue( int customSize )
    throw( BadSizeException, bad_alloc )
{
    if( customSize < 1 || customSize > MAXSIZE )
```

```
        throw BadSizeException();

    body = new T[ customSize ]; // может генерировать исключение
                                // bad_alloc

    size = customSize;

    last = EMPTY;
    next = 0;

    length = 0;
}

// Добавление элемента в очередь
template <class T>
void Queue<T>::Enqueue( const T& item )
{
    if( length == size ) throw PutFullException();

    last = ( last+1 ) % size;
    length++;

    body[ last ] = item;
}

// Извлечение элемента из очереди
template <class T>
T Queue<T>::Dequeue()
{
    if( length == 0 ) throw GetEmptyException();

    T copy = body[ next ];

    next = ( next+1 ) % size;
    length--;

    return copy;
}

// Печать содержимого очереди
template <class T>
void Queue<T>::Print( ostream& out )
{
    if( length == 0 )
    {
        out << "Queue is empty." << endl;
        return;
    }

    int count;
    int ix;
    for( ix = next % size, count = 1;
         count<=length;
         ix = ( ix+1 ) % size, count++ )
    {
        out << body[ ix ] << " ";
    }
    out << endl;
}
```

Для определения итерируемого контейнера нужно унаследовать новый класс от двух классов: класса, воплощающего концепцию очереди, и класса-итератора, предоставив реализацию методов `first` и `next` (листинг 8.2):

Листинг 8.2. Итерируемая версия класса Queue

```
        template <class T>
class IteratedQueue : public Queue<T>, public  Iterator<T>
{
    int counter; // Счетчик итерируемых элементов
    int index;   // Индекс итерируемого элемента

public:
    IteratedQueue( int actualSize = MAXSIZE ) :
        Queue<T>( actualSize )
    {
        counter = 0;
    }

    // Реализация получения адреса первого элемента
    T* first()
    {
        if( length > 0 )
        {
            counter = 1;
            index = Queue<T>::next; // Индекс первого извлекаемого
                                    // элемента очереди
            return &body[ index ];  // Адрес этого элемента
        }

        return 0; // Возвращает 0, если нет элементов в очереди
    }

    // Реализация получения адреса очередного элемента
    T* next()
    {
        counter++;
        if( counter <= length )
        {
            index = ( index+1 ) % size; // Индекс очередного
элемента
            return &body[ index ];      // Адрес этого элемента
        }
        return 0; // Возвращает 0, если больше нет элементов
    }
};
```

К числу достоинств специализированных контейнеров относится возможность разработки контейнеров независимо друг от друга. «Вписывание их в общие рамки» осуществляется посредством конструирования классов, поддерживающих общий интерфейс итератора. При необходимости создания универсальных служб это достоинство оборачивается недостатком: контейнеры не имеют ничего общего, и объекты в контейнерах также не имеют ничего общего. Другой недостаток связан с затратами на вызов виртуальных функций типа `first` и `next`. Не забудем также о перечисленных в гл. 3 общих недостатках модели внутреннего итератора.

8.2. Стандартные контейнеры

Стандартная библиотека шаблонов (Standard Template Library – STL) использует другой подход к проектированию контейнеров. В основе этого подхода лежит идея о том, что обобщенное программирование предполагает не только возможность определения и использования контейнерных типов. Обобщенное программирование должно также предоставлять инструмент для

обработки данных, хранящихся в контейнерах, с использованием одних и тех же алгоритмов (сортировки содержимого контейнера, копирования содержимого одного контейнера в другой контейнер, поиска среди элементов контейнера, применения к элементам контейнера некоторой общей функции и т. д.). Для этого как определение самого контейнерного класса, так и определения поддерживаемых им итераторов должны соответствовать некоторым общим требованиям.

Понятие об итерации стандартного контейнера

Обобщенное представление контейнера иллюстрирует рис. 8.1.

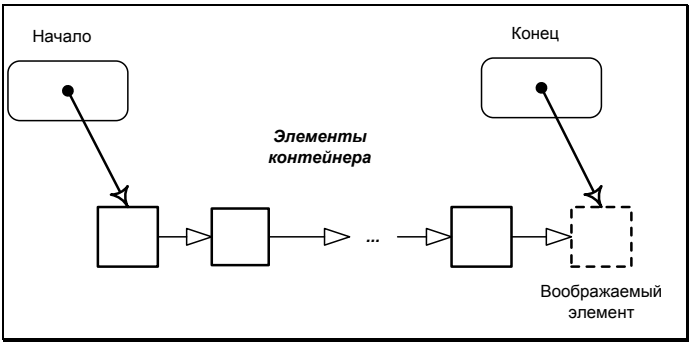


Рис. 8.1. Обобщенное представление контейнера

Началом контейнера в этом представлении считается первый элемент контейнера. Концом контейнера считается несуществующий элемент, следующий за последним элементом (one-past-the-last-element). Шаг итерации состоит в переходе от одного элемента контейнера к другому. Таким образом, для работы с элементами контейнера необходимо иметь возможность ссылаться на элементы контейнера. Для этого, как было показано в предыдущем разделе, необходимо определить итератор контейнера.

Итератор указывает на элемент и предоставляет операцию, заставляющую его (итератор) ссылаться на следующий элемент. Например, для массива C++, итератор образуют две операции: получение адреса элемента массива (`&array[i]` или `array+i`) и переход к следующему элементу (инкремент).

Поскольку итератор абстрагирует понятие указателя на элемент последовательности, требуется реализация следующих операций над итератором:

- получение доступа к элементу контейнера через итератор (используются операции `*` и `->`);
- изменение итератора таким образом, чтобы он ссылался на следующий элемент (используется операция `++`);
- сравнение итераторов (используются операции `==` и `!=`).

Если нужно обеспечить совместимость класса с контейнерами стандартной библиотеки, разработчик контейнера должен определить такие операции для итератора, которые поддерживаются контейнерным классом. Например, для того чтобы обеспечить возможность прямой итерации по контейнеру (см. рис. 8.1) разработчик должен определить два метода:

- метод для получения итератора, указывающего на первый элемент последовательности элементов контейнера (в STL используется имя `begin`);
- метод для получения итератора, указывающего на элемент, следующий за последним (в STL используется имя `end`).

Для итерации в обратном порядке нужно определить соответственно методы `rbegin` (итератор, указывающий на первый элемент в обратной последовательности) и `rend` (итератор, указывающий на элемент, следующий за последним, в обратной последовательности).

Разработка контейнеров и алгоритмов, совместимых с STL

Теперь рассмотрим возможность переработки реализации класса `Queue` из листинга 8.1 в стиле STL-контейнеров. В STL-версии класса `Queue` обеспечим поддержку двух прямых итераторов – итератора для чтения (`const_iterator`) и итератора для записи (`iterator`).

Листинг 7.3 содержит определение класса `STLQueue` и пример программы тестирования возможностей обобщенной реализации очереди в стиле определений библиотеки STL. В частности, следует обратить внимание на использование стандартных алгоритмов, успешно работающих как с классами библиотеки STL, так и с только что спроектированным классом `STLQueue`. Отметим, что комментарии к данному тексту в основном предназначены для прояснения тех элементов решения, которые связаны с проектированием стандартных контейнеров. При этом наиболее существенные изменения по сравнению с прежней версией выделены полужирным начертанием.

Обратите внимание на некоторые изменения реализации членов класса `STLQueue` по сравнению с классом `Queue` из листинга 7.1. Во-первых, появилась возможность заменить спецификатор доступа `protected` на более строгий `private`. Во-вторых, из-за особенностей представления итераторов, при размещении массива `body` в памяти приходится зарезервировать один «лишний» элемент. Если этого не сделать, то в том случае, когда очередь будет заполнена, итератор, возвращаемый функцией `begin`, будет совпадать с итератором, возвращаемым функцией `end`. Впрочем, эти изменения никак не затрагивают внешний интерфейс класса в той его части, которая совпадает с прежними реализациями.

Одновременно следует заметить, что, несмотря на довольно длинное определение класса, здесь иллюстрируется только реализация **основных**

элементов определения STL-контейнера. Класс `STLQueue` не может считаться законченной версией контейнерного класса. Для обеспечения полной совместимости класса с библиотекой STL определение класса `STLQueue` следует доработать, что предлагается студентам в качестве самостоятельного упражнения.

Листинг 8.3 иллюстрирует также основные принципы написания обобщенных функций, способных обрабатывать итерируемые контейнеры (см. содержимое файла STLAlg.h). В качестве параметра шаблона при написании таких функций обычно выступают или типы контейнера (как в случае функции `PrintContents`), или типы итераторов контейнера (как в случае функции `MyCopy`). Условием возможности обработки контейнера такими функциями является поддержка контейнером необходимых итераторов и обеспечение реализации методов `begin` и `end`, возвращающих значение итератора. В файле QueueAlgTest.cpp содержатся примеры обращений к спроектированным методам. В целях обучения функции `PrintContents` и `MyCopy` снабжены контрольной печатью, которая, конечно, не нужна в библиотечных реализациях.

Листинг 8.3. Реализация очереди как класса STL

```
/*
 * STLQueue.h
 *
 * Определение класса для работы с очередью
 * Шаблонная версия, проектируемая в стиле STL
 */
#ifndef _STLQueue_H
#define _STLQueue_H

#include <iostream>
using namespace std;

/*
 * Определения классов исключений в связи с организацией работы
очереди
 */
class QueueException
{
public:
    virtual const char* what()
    {
        return "Undefined queue exception.";
    }
};

class InitializationException : public QueueException
{
public:
    const char* what()
    {
        return "Queue initialization exception.";
    }
};

class ProcessingException : public QueueException
{
public:
    const char* what()
```

```
        {
            return "Queue processing exception.";
    }
};

class BadSizeException : public InitializationException
{
public:
    const char* what()
    {
        return "Incorrect size error.";
    }
};

class PutFullException : public ProcessingException
{
public:
    const char* what()
    {
        return "Put item error. Queue is full.";
    }
};

class GetEmptyException : public ProcessingException
{
public:
    const char* what()
    {
        return "Get item error. Queue is empty.";
    }
};

/*
 * Определение контейнерного класса
 */
template <class T>
class STLQueue { // Контейнер-очередь:
                 // класс, проектируемый в стиле STL-контейнера
public:
    typedef T value_type; // Тип элементов контейнера

private:
    enum { EMPTY = -1, MAXSIZE = 100 };

    T   *body;
    int size;  // Увеличенный на единицу размер очереди

    int last;
    int next;

    int length;

public:
    STLQueue( int actualSize = MAXSIZE )
        throw( BadSizeException, bad_alloc );

    void Enqueue( const T& ) throw( PutFullException );
    T Dequeue() throw ( GetEmptyException );

    void Print();

    /*
     * Объявления и определения в связи с итератором для чтения
     */
```

```
class const_iterator; // Объявление итератора для чтения
friend class const_iterator; // Предоставление итератору
                             // дружественного доступа к
                             // представлению класса

// Определение класса-итератора для чтения
class const_iterator {

    const value_type* ptr; // Указатель на элемент контейнера
    const STLQueue<T>* q;  // Указатель на контейнер
public:
      // Определения типов в соответствии с
      // соглашениями STL
      typedef output_iterator_tag iterator_category;
      typedef T value_type;
      typedef size_t difference_type;
      typedef T* pointer;
      typedef T& reference;

    // Конструктор по умолчанию
    const_iterator() {}

    // Конструктор: инициализация итератора
    // значением адреса элемента
    const_iterator(
        const STLQueue<T>* _q, // Адрес контейнера
        const value_type* _ptr // Адрес элемента
                    ) : ptr( _ptr),
                        q ( _q ) {}

    // Конструктор: инициализация итератора
    // значением другого итератора
    const_iterator(
        const const_iterator& _it
                    ) : ptr( _it.ptr ),
                        q( _it.q ) {}

    // Операция разыменования итератора
    const value_type& operator*() const {
         return *ptr;
    }

    // Операция доступа к элементу контейнера
    // через итератор
    const value_type* operator->() const {
         return ptr;
    }

    // Продвижение по контейнеру
    // (префиксная форма операции инкремента)
    const_iterator& operator++() {
        ++ptr; // Увеличить адрес очередного элемента

        if( ptr >= q->body + q->size ) {
             // Если вышли за пределы body, установить
             // prt на начало массива body
             // (см. схему организации хранения элементов
             //  на рис. 3.1)
             ptr = q->body;
        }
        return *this;
    }
```

```
            // Операции сравнения итераторов
            bool operator==( const_iterator& it2 ) const {
        return ptr == it2.ptr;
    }

    bool operator!=( const_iterator& it2 ) const {
        return (!(*this == it2));
    }
};

// Получение итератора для чтения, ссылающегося на
// начало контейнера
// NB! Объявление функции константной позволяет отличить
//     ее от перегруженной версии iterator begin() (см. ниже)
const_iterator begin() const {

    if( length == 0 ) return const_iterator();

    return const_iterator( this, &body[next] );
}

// Получение итератора для чтения, ссылающегося на
// конец контейнера
// NB! Объявление функции константной позволяет отличить
//     ее от перегруженной версии iterator end() (см. ниже)
const_iterator end() const {

     if( length == 0 ) return const_iterator();

     return const_iterator( this, &body[(last+1)%size] );
}

/*
 * Объявления и определения в связи с итератором для записи
 */

class iterator; // Объявление итератора для записи
friend class iterator; // Предоставление итератору
                       // дружественного доступа к
                       // представлению класса

// Определение класса-итератора для записи
class iterator : public const_iterator {

    value_type* ptr; // Указатель на элемент контейнера
    STLQueue<T>* q;  // Указатель на контейнер
public:
      // Определения типов в соответствии с
      // соглашениями STL
      typedef forward_iterator_tag iterator_category;
      typedef T value_type;
      typedef size_t difference_type;
      typedef T* pointer;
      typedef T& reference;

    // Конструктор по умолчанию
    iterator() {}

    // Конструктор: инициализация итератора
    // значением адреса элемента
    iterator(
        STLQueue<T>* _q, // Адрес контейнера
        value_type* _ptr // Адрес элемента
                  ) : ptr( _ptr),
```

```
                            q ( _q ),
                            const_iterator( _q, _ptr ) {}

        // Конструктор: инициализация итератора
        // значением другого итератора
        iterator(
            const const_iterator& _it
                        ) : ptr( _it.ptr ),
                            q( _it.q ),
                            const_iterator( _it.q, _it.ptr ) {}

        // Операция разыменования итератора
        value_type& operator*() const {
             return *ptr;
        }

        // Операция доступа к элементу контейнера
        // через итератор
        value_type* operator->() const {
             return ptr;
        }

        // Продвижение по контейнеру
        // (префиксная форма операции инкремента)
        iterator& operator++() {
            ++ptr; // Увеличить адрес очередного элемента

            if( ptr >= q->body + q->size ) {
                 // Если вышли за пределы body, установить
                 // prt на начало массива body
                 // (см. схему организации хранения элементов
очереди
                 //  на рис. 3.1)
                 ptr = q->body;
            }
            return *this;
        }

        // Операции сравнения итераторов
        bool operator==( iterator& it2 ) const {
            return ptr == it2.ptr;
        }

        bool operator!=( iterator& it2 ) const {
            return (!(*this == it2));
        }
    };

    // Получение итератора для записи, ссылающегося на
    // начало контейнера
    // NB! Эта функция не константная!
    //     Ср. с перегруженной версией const_iterator begin()
    iterator begin() {

        if( length == 0 ) return iterator();

        return iterator( this, &body[next] );
    }

    // Получение итератора для записи, ссылающегося на
    // конец контейнера
    // NB! Эта функция не константная!
    //     Ср. с перегруженной версией const_iterator end()
    iterator end() {
```

```
                if( length == 0 ) return iterator();

        return iterator( this, &body[(last+1)%size] );
    }
};

/*
 * Реализация методов класса STLQueue
 */

template <class T>
STLQueue <T>::STLQueue( int customSize )
throw( BadSizeException, bad_alloc ) {

    if( customSize < 1 || customSize > MAXSIZE )
        throw BadSizeException();

    size = customSize+1; // Лишний (фиктивный) элемент необходим
                         // для организации итераций

    body = new T[ size ];

    last = EMPTY;
    next = 0;

    length = 0;
}

template <class T>
void STLQueue<T>::Enqueue( const T& item )
throw( PutFullException ) {

    if( length == size-1 ) throw PutFullException();

    last = (last+1) % size;
    length++;

    body[ last ] = item;
}

template <class T>
T STLQueue <T>::Dequeue() throw ( GetEmptyException ) {

    if( length == 0 ) throw GetEmptyException();

    T copy = body[ next ];

    next = (next+1) % size;
    length--;

    return copy;
}

template <class T>
void STLQueue <T>::Print() {

    if( length == 0 ) {
        cout << "Queue is empty" << endl;
        return;
    }

    int count;
    int ix;
```

```
    for( ix=next, count=1;
         count<=length;
         ix = (ix+1) % size, count++ ) {

        cout << body[ ix ] << " ";
    }

    cout << endl;
}
#endif
/*
 * STLAlg.h
 *
 * Примеры реализации обобщенных алгоритмов в стиле STL:
 *     1. Копирование контейнеров;
 *     2. Печать содержимого контейнера на консоль
 */
#ifndef _STLAlg_h
#define _STLAlg_h

// Копирование контейнеров:
// Реализация, идентичная STL-алгоритму копирования
// (функция снабжена тестовой печатью для наблюдения за
итерированием)
template <class In, class Out> void MyCopy(
        In from,    // Итератор, указывающий на начало контейнера-
                    // источника данных
        In too_far, // Итератор, указывающий на конец контейнера-
                    // источника данных
        Out to      // Итератор, указывающий на начало контейнера-
                    // приемника данных
        )
{
    cout << "MyCopy:" << endl;
    while( from != too_far )
    {
        *to = *from;
        cout << "\t" << *from << " copied" << endl;
        ++to;
        ++from;
    }
    cout << "End of MyCopy" << endl;
}

// Печать содержимого контейнера
// (функция снабжена тестовой печатью)
template <class Cont> void PrintContents( Cont& c )
{
    cout << "PrintContents:" << endl;

    Cont::const_iterator from = c.begin();
    Cont::const_iterator too_far = c.end();
    int counter = 0;

    while( from != too_far )
    {
        ++counter;
        cout << "\t" << counter << " : " << *from << endl;
        ++from;
    }
    if( counter == 0 ) cout << "\tEmpty" << endl;

    cout << "End of PrintContents" << endl;
}
```

```
#endif
```

```
/*
 * QueueAlgTest.cpp
 *
 * Программа, иллюстрирующая использование класса STLQueue,
 * алгоритмов из файла STLAlg.h
 * и стандартных средств STL
 */
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

#include "STLQueue.h"
#include "STLAlg.h"

// Вспомогательная функция печати целочисленного массива,
// используемого в ходе тестирования
void PrintArray( int *a, int size )
{
    cout << "Copied array: ";
    for( int i=0; i<size; i++ )
        cout << a[ i ] << " ";
    cout << endl;
}

// Программа тестирования
int main()
{
    int array[ 3 ]; // Создаем целочисленный массив из трех чисел
                    // (не инициализирован)
    vector<int> vect( 4 ); // Создаем стандартный вектор из
четырех
                           // элементов целого типа

    // Создаем очередь queue класса STLQueue, имеющую размер
    // по умолчанию, способную хранить целочисленные данные
    cout << "Creating queue: size = 101... ";
    STLQueue<int> *queue;
    try
    {
        queue = new STLQueue<int>( 101 );
    }
    catch( BadSizeException& e )
    {
        cout << endl << e.what() << endl;
        cout << "OK." << endl;
    }
    catch( bad_alloc )
    {
        cout << endl << "Memory allocation error" << endl;
        cout << "Fatal error. Application stopped." << endl;
        return -1;
    }

    // Создаем очередь queue1 класса STLQueue, имеющую размер 3,
    // способную хранить целочисленные данные
    cout << "Creating queue: size = 3... ";
    STLQueue<int> *queue1;
    try
    {
        queue1 = new STLQueue<int>( 3 );
```

```
        cout << "OK." << endl;
    }
    catch( bad_alloc )
    {
        cout << endl << "Memory allocation error" << endl;
        cout << "Fatal error. Application stopped" << endl;
        return -1;
    }

    // Работаем с очередью queue1 (добавляем и удаляем элементы)
    // NB! Сопровождающими комментариями показывается
    //     текущее состояние очереди
    cout << "queue1.Enqueue( 100 )... ";
    queue1->Enqueue( 100 ); // 100
    cout << "OK." << endl;

    cout << "queue1.Print(): ";
    queue1->Print( cout );
    cout << "OK." << endl;

    cout << "queue1.Dequeue(): ";
    cout << queue1->Dequeue() << " "; // Empty
    cout << "OK." << endl;

    cout << "queue1.Print(): ";
    queue1->Print( cout );
    cout << "OK." << endl;

    cout << "queue1.Enqueue( 200 )... ";
    queue1->Enqueue( 200 ); // 200
    cout << "OK." << endl;

    cout << "queue1.Enqueue( 300 )... ";
    queue1->Enqueue( 300 ); // 200 300
    cout << "OK." << endl;

    // Копируем содержимое очереди в массив array
    // (используем функцию из STLAlg.h)
    MyCopy( queue1->begin(), queue1->end(), &array[ 0 ] );
    PrintArray( array, 2 ); // Контрольная печать array:
                            // должен содержать значения 200 300

    // Заносим в queue1 еще один элемент
    cout << "queue1.Enqueue( 400 )... ";
    queue1->Enqueue( 400 ); // 200 300 400
    cout << "OK." << endl;

    // Печатаем содержимое контейнера queue1
    // (используем функцию из STLAlg.h)
    PrintContents( *queue1 ); // Должны быть напечатаны значения
                              // 200 300 400

    // Контрольная печать методом класса STLQueue
    // (содержательно результаты этой и предыдущей печати должны
    //  совпадать)
    cout << "queue1.Print(): ";
    queue1->Print( cout ); // 200 300 400
    cout << "OK." << endl;

    // Выполняем попытку занесения в переполненную очередь
    try
    {
        queue1->Enqueue( 500 ); // Выбрасывается PutFullException
    }
```

```
        catch( PutFullException& e )
        {
    cout << e.what() << endl;
    cout << "OK." << endl;
}

// Извлекаем элементы из очереди queue1
cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 300 400
cout << "OK." << endl;

cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 400
cout << "OK." << endl;

cout << "queue1.Print(): ";
queue1->Print( cout ); // 400
cout << "OK." << endl;

// Заносим элемент в очередь queue1
cout << "queue1.Enqueue( 600 )... ";
queue1->Enqueue( 600 ); // 400 600
cout << "OK." << endl;

cout << "queue1.Print(): ";
queue1->Print( cout ); // 400 600
cout << "OK." << endl;

cout << "queue1.Enqueue( 600 )... ";
queue1->Enqueue( 600 ); // 400 600 600
cout << "OK." << endl;

cout << "queue1.Print(): ";
queue1->Print( cout ); // 400 600 600
cout << "OK." << endl;

// Копируем содержимое очереди в массив array
// (используем функцию copy из библиотеки STL)
copy( queue1->begin(), queue1->end(), &array[ 0 ] );
PrintArray( array, 3 ); // Контрольная печать array:
                        // должен содержать 400 600 600


// Сравниванием содержимое очереди queue1 и массива array
// (используем функцию equal из библиотеки STL)
if ( equal( queue1->begin(), queue1->end(), &array[ 0 ] ) )
    cout << "Queue and array are now equal" << endl;

// Подсчитываем число вхождений элемента 600 в очередь queue1
// (используем функцию count из библиотеки STL)
int counted = count( queue1->begin(), queue1->end(), 600 );
cout << "Value 600 counted: " << counted << endl;

// Копируем содержимое очереди в вектор vect
// и сравниваем содержимое двух контейнеров
// (используем функции copy и equal из библиотеки STL)
copy( queue1->begin(), queue1->end(), vect.begin() );
if ( equal( queue1->begin(), queue1->end(), vect.begin() ) )
    cout << "Queue and vector are now equal" << endl;

// Изменяем содержимое вектора
vect[ 3 ] = 1000;

// Печатаем содержимое контейнера queue1
```

```
// (используем функцию из STLAlg.h)
PrintContents( vect ); // Должны быть напечатаны значения
                       // 400 600 600 1000

// Извлекаем элементы из очереди queue1
cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 600 600
cout << "OK." << endl;

cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 600
cout << "OK." << endl;

// Заносим элементы в очередь queue1
cout << "queue1.Enqueue( 800 )... ";
queue1->Enqueue( 800 ); // 600 800
cout << "OK." << endl;

cout << "queue1.Enqueue( 900 )... ";
queue1->Enqueue( 900 ); // 600 800 900
cout << "OK." << endl;

cout << "queue1.Print(): ";
queue1->Print( cout ); // 600 800 900
cout << "OK." << endl;

// Осуществляем поиск элемента со значением 600 в очереди
// (используем функцию find библиотеки STL)
// NB! Элемент должен быть обнаружен
STLQueue<int>::iterator found; // Итератор для представления
                               // результата поиска
found = find( queue1->begin(), queue1->end(), 600 );
cout << "Value requested: 600 => Value found: " << *found;
cout << endl;

// Осуществляем поиск элемента со значением 800 в очереди
// (используем функцию find библиотеки STL)
// NB! Элемент должен быть обнаружен
found = find( queue1->begin(), queue1->end(), 800 );
cout << "Value requested: 800 => Value found: " << *found;
cout << endl;

// Осуществляем поиск элемента со значением 900 в очереди
// (используем функцию find библиотеки STL)
// NB! Элемент должен быть обнаружен
found = find( queue1->begin(), queue1->end(), 900 );
cout << "Value requested: 900 => Value found: " << *found;
cout << endl;

// Осуществляем поиск элемента со значением 1000 в очереди
// (используем функцию find библиотеки STL)
// NB! Ожидаемый результат: элемент не обнаружен
found = find( queue1->begin(), queue1->end(), 1000 );
if( found == queue1->end() )
    cout << "Value requested: 1000 => Not found: " << endl;

// Извлекаем элементы из очереди
cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 800 900
cout << "OK." << endl;

cout << "queue1.Dequeue(): ";
cout << queue1->Dequeue() << " "; // 900
cout << "OK." << endl;
```

```
            cout << "queue1.Dequeue(): ";
    cout << queue1->Dequeue() << " "; // Empty
    cout << "OK." << endl;

    // Выполняем попытку извлечения элемента из пустой очереди
    try
    {
        queue1->Dequeue(); // Генерируется GetEmptyException
    }
    catch( GetEmptyException& e )
    {
        cout << e.what() << endl;
        cout << "OK." << endl;
    }

    // Печатаем содержимое очереди
    // (убеждаемся, что очередь действительно пуста)
    cout << "queue1.Print(): ";
    queue1->Print( cout ); // Empty
    cout << "OK." << endl;

    // Разрушаем объект queue1
    cout << "Destroying queue... ";
    delete queue1;
    cout << "OK." << endl;

    // Успешное завершение тестирующей программы
    return 0 ;
}
```

Студентам рекомендуется внимательно изучить содержание этого примера и результаты работы программы тестирования.

В данном разделе рассмотрены только основы разработки и использования типов и алгоритмов STL. Более подробные сведения можно получить из книг [Stroustrup, 2000], [Austern, 1999], [Давыдов, 2005] и др.

8.3. Реализация альтернативных вариантов размещения элементов контейнера в памяти

Во многих случаях алгоритмы обработки данных, хранящихся в контейнере, могут быть сформулированы независимо от организации хранения элементов в памяти в то время как при определении содержания отдельных операций невозможно абстрагироваться от модели хранения.

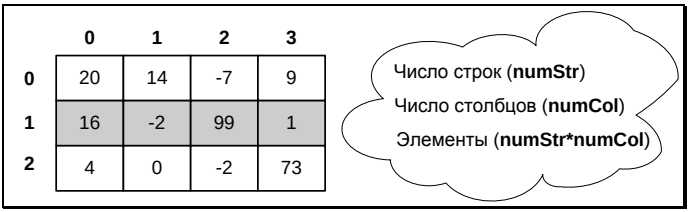


Рис. 8.2. Двумерный массив как математический объект и как объект данных

Например, при определении двумерного массива можно использовать различные формы размещения элементов массива в памяти

[Рытенков, Пышкин, 2006]. Двумерный массив как математический объект и как объект данных представлен на рис. 8.2 (на примере хранения целочисленных данных).

Наиболее распространенными вариантами организации памяти, занимаемой элементами массива, являются две альтернативы. Вариантом размещения, нацеленным на обеспечение наиболее эффективной обработки данных, является такое использование памяти, при котором элементы массива располагаются в непрерывном участке памяти (рис. 8.3, а). При использовании подобной модели организации хранения массивов (rectangular arrays) требуется наличие в динамической памяти относительно больших сегментов свободной динамической памяти.

Альтернативой является размещение отдельных строк матрицы в виде одномерных массивов, размещаемых в разных участках динамической памяти, и управление этими массивами с использованием указателей или ссылок (если язык поддерживает ссылочные типы), как представлено на рис. 8.3, б. При этом не требуется обязательного наличия в динамической памяти непрерывных свободных сегментов большого размера, однако время обращения к отдельному элементу массива в общем случае больше, чем в первом варианте. Иногда такие массивы называют «ступенчатыми». Обычно язык программирования явно поддерживает ту или иную модель размещения двумерных массивов. Например, FORTRAN работает с массивами, размещаемыми в непрерывных «прямоугольных» участках памяти, а Java использует ссылочную модель (jagged array).

В общем случае, при реализации «ступенчатых» массивов размеры отдельных массивов-строк не обязательно совпадают, что позволяет экономить память в том случае, когда не требуется заполнение всех ячеек массива.

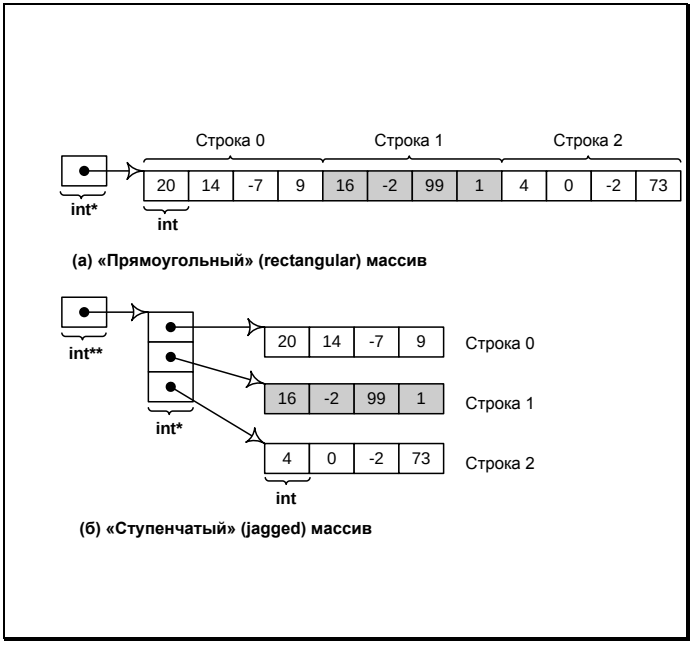


Рис. 8.3. Размещение элементов двумерного массива в памяти

Задача состоит в том чтобы разработать на языке C++ механизм, позволяющего обеспечить возможность создания двумерных массивов, использующих различные модели размещения элементов в памяти и обеспечить выполнение следующих требований:

- возможность абстрагирования от способа хранения в клиентском коде;
- реализация общего интерфейса классов, предоставляющих различные варианты реализации 2D-контейнеров;
- возможность перехода от одного типа контейнера к другому без существенной переделки клиентского кода;
- возможность встраивания в разработанную инфрастуктуру контейнеров, построенных на основе других вариантов управления динамической памятью.

Одно из возможных решений использует шаблон проектирования «стратегия» [Gamma, Helm, Johnson, Vlissides, 1995] и представляет собой иерархию классов:

```
template <class CStorage>
class CMatrix : public CStorage
{
```

```
  //реализация класса CMatrix
};

class CRectStorage
{
  //реализация первого способа хранения 2D-контейнера
};

class CJaggedStorage
{
  //реализация второго способа хранения 2D-контейнера
};

//матрица с прямоугольным способом хранения данных
typedef CMatrix<CRectStorage> CRectMatrix;

//матрица со ступенчатым способом хранения данных
typedef CMatrix<CJaggedStorage> CJaggedMatrix;
```

В приведенной иерархии шаблонный класс `CMatrix` называется «главным классом» (host class). Он абстрагируется от конкретного алгоритма хранения данных за счет шаблонного параметра `CStorage`, на место которого подставляется класс, реализующий конкретный способ хранения данных. Классы `CRectStorage` и `CJaggedStorage` называются «классами стратегий» и они реализуют конкретные способы хранения данных. Конкретизируя класс `CMatrix` с помощью одного из классов стратегий, мы получаем 2D-контейнер с интересующим нас типом хранения данных. Конструктивной особенностью решения является тот факт, что интерфейс, задаваемый `CStorage`, определяется неявно: в действительности, класса `CStorage` нет, однако код для инстанцируемых классов `CRectMatrix` и `CJaggedMatrix` не будет компилироваться, если реализуемый ими интерфейс не будет соответствовать следующему «виртуальному» определению `CStorage`:

```
template <class T>
class CStorage
{
  public:
    CStorage<T>();
    CStorage<T>(const int, const int);
    CStorage<T>& operator=(const CStorage<T>&);
    T &getEl(const int, const int) const;
    void setEl(int, int, T );
    int getNRows()const;
    int getNCols()const;
  protected:
    ~CStorage<T>();
};
```

Для использования шаблона `CMatrix` с объектами данных пользовательского типа следует специализировать `CMatrix` соответствующим образом. При этом для пользовательского типа должны быть определены: конструктор по умолчанию, деструктор, базовые арифметические и логические операции, а также оператор присваивания целого числа (нуля, т.е. у типа должен быть определен “нуль”).

Листинг 8.4 содержит определение класса `CRectStorage`, реализующего «прямоугольную» модель хранения.

Листинг 8.4. Шаблон класса, реализующего «прямоугольный» способ хранения матрицы

```
template <class T>
class CRectStorage
//Шаблонный класс CRectStorage является классом стратегии
//хранения матрицы и предназначен для использования вместе с
//главным шаблонным классом CMatrix.
{
  public:
    //конструктор, берущий размеры матрицы, как аргументы
    CRectStorage<T>(const int newNRows = 2, const int newNCols =
2);
    //оператор присваивания
    CRectStorage<T>& operator=(const CRectStorage<T> &cont);
    //метод, для получения элемента по его положению
    T &getEl(const int i, const int j) const;
    //метод, для изменения элемента по его положению
    void setEl(int i, int j, T val);
    //метод, возвращающий количество строк
    int getNRows()const;
    //метод, возвращающий количество столбцов
    int getNCols()const;
  protected:
    //деструктор, защищенный для безопасного использования
производный классов
    ~CRectStorage<T>();
  private:
    //вспомогательный метод, копирующий все данные из newSource в
source
    void setSource(T* newSource);
    T* source; //указатель на идущие подряд строки матрицы
    int nRows; //количество строк
    int nCols; //количество столбцов
};

template <class T>
CRectStorage<T>::CRectStorage(const int newNRows, const int
newNCols)
{
  nRows = newNRows;
  nCols = newNCols;
  source = new T[nRows*nCols];
}

template <class T>
CRectStorage<T>::~CRectStorage()
{
  if(source)
    delete[] source;
  source = 0;
}

//Это метод предназначен только для внутреннего использования,
//размер newSource должен быть равен nRows*nCols.
template <class T>
void CRectStorage<T>::setSource(T* newSource)
{
  for(    T* pSource = source, *pNewSource = newSource;
          pSource-source < nRows*nCols;
          pSource++, pNewSource++ )
    *pSource = *pNewSource;
  return;
}
```

```
template <class T>
CRectStorage<T>& CRectStorage<T>::operator=(const CRectStorage<T>
&stor)
{
  //При присваивании самому себе ничего делать не надо
  if(this != &stor)
  {
    //Если кол-во ячеек одинаково, то память повторно выделять не
надо
    if(nRows*nCols == stor.nRows*stor.nCols)
    {
      nRows = stor.nRows;
      nCols = stor.nCols;
    }
    //иначе повторно выделяем память
    else
    {
      delete[] source;
      nRows = stor.nRows;
      nCols = stor.nCols;
      source = new T[nRows*nCols];
    }
    //копируем данные
    setSource(stor.source);
  }

  return *this;
}

template <class T>
T &CRectStorage<T>::getEl(const int i, const int j)const
{
  //проверка корректности обращения к элементу
  assert((i < nRows)&&(i >= 0)&&(j < nCols)&&(j >= 0));

  //Элементы матрицы хранятся по строкам, так что положение
  //требуемого элемента - i + nRows * j
  return source[i + nRows * j];
}

template <class T>
void CRectStorage<T>::setEl(int i, int j, T val)
{
  //проверка корректности обращения к элементу
  assert((i < nRows)&&(i >= 0)&&(j < nCols)&&(j >= 0));

  //Элементы матрицы хранятся по строкам, так что положение
  //требуемого элемента - i + nRows * j
  source[i + nRows*j] = val;
}

template <class T>
int CRectStorage<T>::getNRows()const
{
  return nRows;
}

template <class T>
int CRectStorage<T>::getNCols()const
{
  return nCols;
}
```

Разработка шаблонного класса `CJaggedStorage` не представляет труда, поэтому эту часть работы оставляем читателю в качестве упражнения.

Шаблонный класс `CMatrix` наследует классу, подставляемому в качестве параметра `CStorage`, в результате чего сам класс `CMatrix` может рассматриваться как 2D-контейнер, который в процессе инстанцирования шаблона становится контейнером со специфическим способом хранения данных. Кроме этого, в классе `CMatrix` можно реализовать различные специфические матричные операции, а также – необходимые итераторы (листинг 8.5 содержит фрагмент определения класса `CMatrix`).

Листинг 8.5. Определение класса CMatrix (фрагмент)

```
//Шаблонный класс CMatrix обладает
//стратегией хранения массива (CStorage),
//реализует матричные операции,
//а также может иметь методы для получения
//необходимых итераторов (например, для совместимости с STL)
template <class T, class CStorage>
class CMatrix : public CStorage
{
  public:
    //конструктор, берущий размеры матрицы, как аргументы
    CMatrix<T, CStorage>(const int newNRow = 2, const int newNCol
= 2);
    //конструктор копирования
    CMatrix<T, CStorage>(const CMatrix<T, CStorage> &matr);
    //операция присваивания с прибавлением (+=)
    //размеры матриц должны совпадать
    CMatrix<T, CStorage>& operator+=(const CMatrix<T, CStorage>&
matr);
    //операция присваивания с умножением (*=)
    CMatrix<T, CStorage>& operator*=(const T val);
    //операция сложения
    //размеры матриц должны совпадать
    CMatrix<T, CStorage> operator+(const CMatrix<T, CStorage>&
matr)const;
    //операция умножения на число
    CMatrix<T, CStorage> operator*(const T val)const;
    //операция умножения на матрицу
    //кол-во столбцов у первой матрицы и кол-во строк у второй
матрицы в этой
    //операции должны совпадать.
    CMatrix<T, CStorage> operator*(const CMatrix<T, CStorage>&
matr)const;

    //Другие операции и методы
    //...

};

//Этот конструктор вызывает соответствующий конструктор класса
стратегии
template <class T, class CStorage>
CMatrix<T, CStorage>::CMatrix(const int newNRow, const int
newNCol):
    CStorage(newNRow, newNCol) {}

//Конструктор копирования вызывает конструктор, создающий
//матрицу заданных размеров,
```

```
//у класса стратегии и затем копирует данные
template <class T, class CStorage>
CMatrix<T, CStorage>::CMatrix(const CMatrix<T, CStorage>&matr):
    CStorage(matr.getNRows(), matr.getNCols())
{
  for(int i = 0; i < matr.getNRows(); i++)
  {
    for(int j = 0; j < matr.getNCols(); j++)
    {
      setEl(i, j, matr.getEl(i, j));
    }
  }
}

template <class T, class CStorage>
CMatrix<T, CStorage>& CMatrix<T, CStorage>::operator+=(const
CMatrix<T, CStorage>& matr)
{
  //Проверка того, что размеры матриц совпадают
  assert((getNRows() == matr.getNRows())&&(getNCols() ==
matr.getNCols()));

  for(int i = 0; i < getNRows(); i++)
  {
    for(int j = 0; j < getNCols(); j++)
    {
      setEl(i, j, getEl(i, j) + matr.getEl(i, j));
    }
  }

  return *this;
}

template <class T, class CStorage>
CMatrix<T, CStorage> CMatrix<T, CStorage>::operator+(const
CMatrix<T, CStorage>& matr)const
{
  CMatrix<T, CStorage> res = *this;
  res += matr;

  return res;
}

template <class T, class CStorage>
CMatrix<T, CStorage>& CMatrix<T, CStorage>::operator*=(const T
val)
{
  for(int i = 0; i < getNRows(); i++)
  {
    for(int j = 0; j < getNCols(); j++)
    {
      setEl(i, j, getEl(i, j) * val);
    }
  }

  return *this;
}

template <class T, class CStorage>
CMatrix<T, CStorage> CMatrix<T, CStorage>::operator*(const T
val)const
{
  CMatrix<T, CStorage> res = *this;
  res *= val;
```

```
        return res;
}

template <class T, class CStorage>
CMatrix<T, CStorage> CMatrix<T, CStorage>::operator*(const
CMatrix<T, CStorage>& matr)const
{
  CMatrix<T, CStorage> res(getNRows(), matr.getNCols());
  T tmp;
  //Проверка того, что матрицы позволяют произвести умножение
  assert(getNCols() == matr.getNRows());

  for(int i = 0; i < getNRows(); i++)
  {
    for(int k = 0; k < matr.getNCols(); k++)
    {
      tmp = 0;
      //Вычисление элемента результирующей матрицы
      for(int j = 0; j < getNCols(); j++)
        tmp += getEl(i, j) * matr.getEl(j, k);
      res.setEl(i, k, tmp);
    }
  }

  return res;
}

//...
```

Ниже приведен пример специализации шаблонного класса CMatrix для хранения прямоугольной матрицы целых чисел:

```
CMatrix<int,CRectStorage> matrix( 3, 4 );
```

Подведем итоги.

- Абстрагирование от способа хранения в клиентском коде обеспечено созданием классов `CJaggedStorage` и `CRectStorage`, воплощающих различные модели организации данных в памяти.
- Наличие класса `CMatrix` позволяет констатировать, что реализован общий интерфейс для создания 2D-контейнеров с различными способами распределения памяти. Это обеспечивает возможность проектирования функций-обработчиков контейнеров независимо от деталей их внутреннего представления, а также возможность перехода от одного типа контейнера к другому без существенной переделки клиентского кода.
- Наличие общего интерфейса и построение иерархии классов на базе универсального шаблона проектирования «стратегия» позволяет при необходимости встроить в разработанную инфрастуктуру контейнеры, построенные на основе других вариантов управления динамической памятью, например, использующие блочную организацию памяти, двусвязные списки и т. п.

Список использованных источников

Ссылки на зарубежные источники даются по возможности в их оригинальном наименовании. Одновременно приводятся библиографические сведения о переводном издании (при этом автор и название напечатаны полужирным шрифтом). Если полужирным начертанием выделено издание на языке оригинала, это означает, что автору неизвестен или оказался недоступен перевод этого издания на русский язык, и он пользовался оригинальной версией.

[Austern, 1999] Austern, Matthew H. Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley. 1999. Имеется перевод: **Остерн М. Обобщенное программирование и STL: Использование и наращивание стандартной библиотеки шаблонов** / Пер. с англ.- СПб.: Невский диалект, 2004.- 544 с., ил.

[Brooks, 1995] Brooks, Frederick P., Jr. *The Mythical Man-Month. Essays on Software Engineering.* Anniversary Edition. Addison-Wesley, 1995. Имеется перевод: **Брукс Ф.** ***Мифический человеко-месяц, или как создаются программные системы*** / Пер. с англ.- СПб.: Символ, 2000.- 304 с., ил.

[Dahl, Dijkstra, Hoare, 1972] Dahl O.-J., Dijkstra E.W., Hoare C.A.R. *Structured Programming.* Academic Press. London and New York. 1972. Имеется перевод: **Дал У., Дейкстра Э., Хоор К.** ***Структурное программирование*** / Пер. с англ.- М.:Мир,1975.-248 с.

[Gamma, Helm, Johnson, Vlissides, 1995] Gamma E., Helm R., Johnson R., Vlissides, J. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley. 1995. Имеется перевод: **Джонсон Р., Хелм Р., Влиссидес Дж., Гамма Э.** ***Приемы объектно-ориентированного проектирования.***- СПб.: Питер, 2001.- 368 с.

[Harel, 1988] **Harel D.** ***On Visual Formalisms*** // Communications of the ACM, Vol.31, No.5, May 1988.- pp.514-530.

[Knuth, 1968] Knuth, Donald E. The art of computer programming. Volume 1. Fundamental Algorithms. Addison-Wesley, 1968. Имеется перевод: **Кнут Д. Искусство программирования для ЭВМ. Т.1. Основные алгоритмы.** М.: Мир, 1976.- 736 с.

[Knuth, 1973] Knuth, Donald E. The art of computer programming. Volume 3. Sorting and Searching. Addison-Wesley, 1973. Имеется перевод: **Кнут Д. Искусство программирования для ЭВМ. Т.3. Сортировка и поиск.** М.: Мир, 1978.- 844 с.

[Lekarev, 1993] Lekarev M.F. *Das graphische Verfahren der Software-Entwicklung für logisch komplizierte Anwendungen* // Technische Berichte der Fachhochschule Hamburg, №25 (Aug.1993), s.36-38.

[Linger, Mills, Witt, 1979] Linger, Richard S. Mills, Harland D. Witt, Bernard I. *Structured Programming: Theory and Practice.* Addison-Wesley, 1979.

Имеется перевод: **Лингер Р., Миллс Х., Уитт Б.** ***Теория и практика структурного программирования*** / Пер. с англ.- М.: Мир, 1982- 406 с., ил.

[McConnell, 2001] McConnell, Jeffrey J. *Analysis of Algorithms: An Active Learning Approach.* Jones & Barlett Publishers, 2001. Имеется перевод: **Макконелл Дж.** ***Анализ алгоритмов. Вводный курс*** / Пер. с англ.- М.: Техносфера, 2002- 304 с.

[Mendelson, 1963] Mendelson, Elliott. *Introduction to Mathematical Logic.* D. van Nostrand Company, Inc. 1963. Имеется перевод: **Мендельсон Э.** ***Введение в математическую логику*** / Пер. с англ.- М.: Наука, 1971- 320 с., ил.

[Niemann, 1995] Ниман Т. Сортировка и поиск: рецептурный справочник / Пер. с англ. П. Дубнера, 1995.

[Sebesta, 2002] Sebesta, Robert W. *Concepts of Programming Languages.* Addison-Wesley Inc. 2002. Имеется перевод: **Себеста Р.У.** ***Основные концепции языков программирования*** / Пер. с англ.- М.: Издательский дом «Вильямс», 2001.-672 с., ил.

[Stroustrup, 2000] Stroustrup, Bjarne. *The C++ Programming Language.* Special Edition. Addison-Wesley. 2000. Имеется перевод: **Страуструп Б.** ***Язык программирования C++.*** Специальное изд. / Пер. с англ.- М.: «Издательство Бином», СПб.: «Невский диалект», 2001.- 1099 с., ил.

[Wirth, 1976] Wirth, Niclaus. *Algorithms + Data Structures = Programs.* Prentice-Hall, Inc. Englewood Cliffs, N.J. 1976. Имеется перевод: **Вирт Н.** ***Алгоритмы + структуры данных = программы*** / Пер. с англ.- М.: Мир, 1985.- 406 с., ил.

[ГОСТ 19.701-90] ГОСТ 19.701-90 (ИСО 5807-85). Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения.. – Действует с 01.01.1992.

[Давыдов, 2003] Давыдов В.Г. *Программирование и основы алгоритмизации: Учебное пособие.* - М.: Высшая школа, 2003.- 447 с., ил.

[Давыдов, 2005] Давыдов В.Г. *Технологии программирования. C++. Учебное пособие.* – СПб.: БХВ-Петербург, 2005.- 672 с., ил.

[Карпов, 2003] Карпов Ю.Г. *Теория автоматов. Учебник для вузов* – СПб.: Питер, 2003.- 208 с.

[Кубенский, 2004] Кубенский А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++.в – СПб.: БХВ-Петербург, 2004.- 464 с.: ил.

[Лекарев, 1997] Лекарев М.Ф. *Визуальный формализм для разработки программного обеспечения.* – Санкт-Петербургский гос. техн. ун-т. СПб,1997.- 95 с.

[Лекарев, 2000-1] Лекарев М.Ф. Программная реализация конечных автоматов с использованием L-сети. –СПб.: СПбГТУ, 2000.-32 с., ил.

[Лекарев, 2000-2] Лекарев М.Ф. *Модули с двумя выходами в программных проектах.* –СПб.: СПбГТУ, 2000.-72 с., ил.

[Лекарев, Пышкин, 2005] **Лекарев М.Ф., Пышкин Е.В.** ***Синтаксический анализ выражений в скобочной форме на основе визуальный формализма L-сети*** // Вычислительные, измерительные и управляющие системы: Сборник научных / Под. ред. Ю.Б. Сениченкова.- СПб.: Изд-во Политехн. ун-та, 2005.- с.161-169.

[Новиков, 2000] Новиков Ф.А. *Дискретная математика для программистов* - СПб.: Питер, 2000.- 304 с., ил.

[Одинцов, 2002] Одинцов И.С. *Профессиональное программирование. Системный подход*.- СПб.: БХВ-Петербург, 2002.- 512 с., ил.

[Пышкин, 2005] Пышкин Е.В. Основные концепции и механизмы объектно-ориентированного программирования. Учебное пособие - СПб.: БХВ-Петербург, 2005.- 640 с., ил.

[Рытенков, Пышкин, 2006] **Рытенков А.С., Пышкин Е.В.** ***Шаблон проектирования 2D-контейнера с альтернативными вариантами размещения элементов в памяти*** // XXXIV Неделя науки СПбГПУ. Ч.VII: Материалы межвузовской научной конференции. СПб.: Изд-во СПбГПУ, 2006.- с.57-59.

[Шалыто, 1998] **Шалыто А.А.** ***SWITCH-технология. Алгоритмизация и программирование задач логического управления.*** - СПб.: Наука, 1998.- 628 с.

ПЫШКИН Евгений Валерьевич

СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ:
РЕАЛИЗАЦИЯ НА C/C++
Редакция 2006 года с исправлениями

Подписано в печать 25.08.2009 Формат 60x84 1/16 Усл.печ.л. 12,5
Уч.-изд.л 12,5 Тираж 100 экз.

Отпечатано с оригинал-макета автора в центре
оперативной полиграфии
факультета технической кибернетики СПбГПУ.
195251, Санкт-Петербург, Политехническая ул., 21