

МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

А. В. ШЕВЧЕНКО

Программирование и основы алгоритмизации

Учебное пособие

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2018

УДК 519.683+519.682+004.021

ББК 3 973.2-018я7

ШЗ7

Шевченко А. В.

ШЗ7 Программирование и основы алгоритмизации: учеб. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2011. 144 с.

ISBN 978-5-7629-2309-5

Рассматриваются вопросы разработки программного обеспечения, в том числе архитектурные особенности современной компьютерной техники; языки программирования высокого уровня С и С++; технологии структурного и объектно-ориентированного программирования; основные структуры данных; типовые алгоритмы сортировки, поиска, оптимизации.

Предназначено для студентов, обучающихся по направлению подготовки бакалавров 13.03.02 «Электроэнергетика и электротехника».

УДК 519.683+519.682+004.021

ББК 3 973.2-018я7

Рецензенты: кафедра информационных управляющих систем СПбГУТ им. проф. М. А. Бонч-Бруевича; д-р техн. наук, проф. Д. А. Первухин (СПбГУ).

Утверждено

редакционно-издательским советом университета
в качестве учебного пособия

ISBN 978-5-7629-2309-5

© СПбГЭТУ «ЛЭТИ», 2018

Введение

Современный уровень развития науки и техники, а также социальных процессов достигнут во многом благодаря массовому применению компьютерной техники во всех сферах человеческой деятельности. Всем этим устройствам, начиная с микропроцессоров и заканчивая суперкомпьютерами, требуется программное обеспечение для эффективного выполнения возложенных на них задач. Без преувеличения можно сказать, что создание программного обеспечения стало одной из самых масштабных отраслей в сфере производства интеллектуальной продукции. Об этой отрасли часто говорят как об индустрии программного обеспечения.

Вместе с тем, программирование и алгоритмизация – достаточно молодые области науки. С момента появления первых компьютеров прошло немногим более полувека. За это время компьютерная наука, заложенная в трудах Норберта Винера, Джона фон Неймана, Алана Тьюринга, Клода Шеннона и многих других ученых, сделала огромный скачок. Это касается как технических средств компьютерной техники, мощность которых растет невероятными темпами, так и технологий программирования и обработки данных.

В настоящем пособии в гл. 1 кратко рассматриваются особенности архитектуры компьютеров, которые необходимо учитывать при создании программного обеспечения.

Приемы программирования даются в гл. 2 на примере широко распространенного языка высокого уровня С. Много внимания уделяется приемам создания качественного программного обеспечения – структурному подходу к программированию (гл. 3) и объектно-ориентированному программированию на языке С++ (гл. 4).

Глава 5 посвящена вопросам алгоритмизации. В ней рассматриваются и сравниваются различные алгоритмы сортировки массивов, поиска в массивах, строках и файлах данных. Показываются приемы работы с динамическими структурами данных – списками, очередями, стеками, деревьями и графами. Затрагиваются задачи комбинаторной оптимизации и методы их решения.

Текст пособия насыщен большим числом примеров на языках С и С++, что позволяет применять рассматриваемые структуры данных и алгоритмы для решения практических задач.

Глава 1. АРХИТЕКТУРА КОМПЬЮТЕРНЫХ СИСТЕМ

Создание правильно и эффективно работающего программного обеспечения требует хорошего понимания принципов работы аппаратной части компьютерной техники, ее составных элементов и их взаимодействия. В этой главе кратко рассматриваются такие элементы, как процессор, оперативная память, периферийные устройства, с позиций разработки программного обеспечения.

1.1. История создания архитектуры современных компьютеров

История современных компьютерных систем берет начало в сороковых годах двадцатого века, когда в 1946 г. трое ученых – Артур Бёркс, Герман Голдстейн и Джон фон Нейман – опубликовали статью «Предварительное рассмотрение логического конструирования электронного вычислительного устройства». В статье обосновывалось применение двоичной системы для представления данных, выдвигалась идея использования общей памяти для программы и данных. Имя фон Неймана было достаточно широко известно в науке того времени, что отодвинуло на второй план его соавторов, и данные идеи получили название «принципы фон Неймана». Они перечислены ниже:

- использование двоичной системы счисления для данных и команд;
- программное управление (процессор исполняет команды из памяти);
- однородность памяти (команды и данные хранятся в одной памяти);
- адресуемость памяти (все ячейки одинаково доступны процессору);
- последовательное выполнение команд (одна за другой);
- условный переход (возможность изменения порядка команд).

Указанные принципы позволили при достаточно ограниченных технических возможностях того времени реализовать первые работоспособные вычислительные системы (ЭНИАК, США – 1946 г.; Mark I, Великобритания – 1948 г. и др.). За прошедшие с тех пор более полувека сменилось не одно поколение элементной базы компьютерных систем – электромеханические устройства, электронные лампы, транзисторы, микросхемы, сверхбольшие интегральные схемы. Современные компьютерные системы обладают невероятной вычислительной мощностью, возможностями хранения огромных объемов данных, развитыми средствами взаимодействия с человеком-пользователем.

1.2. Архитектура компьютера

Современные компьютеры в основном унаследовали архитектуру, разработанную фон Нейманом. В составе любого компьютера присутствует центральная часть, включающая процессор (один или несколько) и оперативную память, а также периферийные устройства. Центральная часть обеспечивает выполнение программ (процессор выполняет операции над хранящимися в оперативной памяти данными). Сами программы (команды процессора) также хранятся в оперативной памяти и могут рассматриваться как данные. Периферийные устройства обеспечивают операции ввода-вывода, долговременное хранение данных, а также взаимодействие с пользователем.

1.3. Представление данных в двоичной системе счисления

В архитектуре фон Неймана данные и команды представляются в двоичной системе счисления. При этом разделяют два вида данных: целые числа и числа с плавающей точкой.

Для представления натуральных целых чисел их раскладывают в ряд по степеням числа два и представляют отдельными разрядами – битами. Единица, установленная в некотором разряде, добавляет к числу два в степени, соответствующей номеру разряда. Полученное значение называют весом разряда. Для представления некоторого диапазона значений целых чисел требуется число разрядов, которое определяется по формуле $\log_2(N)$, где N – максимальное значение числа.

Для представления целых чисел со знаком старший разряд выделяют под знак, при этом 0 в знаковом разряде указывает на то, что число положительное, а 1 – отрицательное. Следует отметить, что выделение одного разряда под знак не уменьшает диапазон представляемых целых чисел, а только смещает его в область отрицательных значений так, что ноль оказывается посередине диапазона (табл. 1.1).

Таблица 1.1

Двоичное целое число без знака	Двоичное целое число со знаком
00000000 = 0	10000000 = -128
00000001 = 1	11111111 = -1
00000010 = 2	00000000 = 0
11111111 = 255	01111111 = 127

Для представления вещественных чисел в компьютерных системах используют формат числа с плавающей точкой. При этом число представляется двумя компонентами – мантиссой и порядком.

Таблица 1.2

Число	Мантисса	Порядок
123	0.123	3
0.0123	0.123	–1
0.123	0.123	0

Так как мантисса представляет собой дробь, она раскладывается по отрицательным степеням числа 2 ($1/2$, $1/4$, ...). Поскольку число разрядов, выделяемых под мантиссу, ограничено, то при представлении чисел с плавающей точкой возникает проблема точности (см. 2.3).

1.4. Процессор и оперативная память

Процессор представляет собой электронное устройство, в составе которого имеются регистры общего назначения, предназначенные для кратковременного хранения данных, специальные регистры (программный счетчик и указатель стека), а также арифметико-логическое устройство, которое выполняет операции над содержащимися в регистрах данными.

Поскольку процессор обрабатывает данные, представленные в двоичной системе счисления, то его важнейшей характеристикой является разрядность (размер в битах) основных регистров процессора. Разрядность процессора влияет на скорость выполнения программ. Например, сложение двух 64-разрядных чисел на 64-разрядном процессоре выполняется одной командой, тогда как на 32-разрядном процессоре для этого требуется как минимум три команды.

Возможности процессора в части обработки данных определяются его системой команд, которые включают действия над данными, проверки условий, команды переходов. В зависимости от типов обрабатываемых данных команды относят к целочисленной или плавающей арифметике.

Быстродействие процессора также определяют отдельно по двум группам команд. Для этого применяются характеристики MIPS (миллион команд в секунду) и MFLOPS (миллион команд с плавающей точкой в секунду).

Оперативная память компьютера состоит из набора адресуемых ячеек, каждая размером в один байт. Адреса начинаются с нулевого и следуют непрерывно. Память, которую можно адресовать при данной архитектуре компьютера, называют адресным пространством. Возможности компьютера по адресации определяются разрядностью процессора. Если 16-разрядные процессоры могли адресовать 64 Кбайт памяти, 32-разрядные – 4 Гбайт, то появившиеся несколько лет назад 64-разрядные процессоры снимают на долгие годы вперед вопрос об ограничениях на возможную память компьютера благодаря возможности адресации 2^{64} байт памяти.

Современные компьютеры могут иметь не один, а несколько процессоров. Это позволяет выполнять параллельно команды разных программ или различные фрагменты одной и той же программы, которые находятся в единой оперативной памяти. Существуют различные архитектуры многопроцессорных компьютеров – скалярные, векторные, матричные, которые ориентируются на решение определенных классов вычислительных задач.

1.5. Создание и выполнение программ

В соответствии с принципами фон Неймана программирование заключается в размещении в памяти команд и данных для решения поставленной задачи. Если на заре компьютерной техники программирование велось непосредственно в машинных кодах (командах процессора), то затем для ускорения этого процесса стали применять языки программирования и специальные программы-трансляторы, которые переводят программы с языка программирования в машинный код.

Существуют языки программирования низкого и высокого уровня. Языки низкого уровня (их еще называют ассемблерами) позволяют разрабатывать программы на уровне команд процессора. Их применяют там, где требуется высокая эффективность программного кода. Программирование на языке низкого уровня – задача сложная и требует высокой квалификации программиста.

Программирование на языке высокого уровня не зависит от архитектуры компьютера и позволяет создавать программы, которые могут переноситься с одной компьютерной системы на другую. Языки высокого уровня бывают процедурные, в которых задается последовательность действий,

а также непроцедурные, в которых определяется, каким должен быть результат работы программы.

Трансляторы с языков высокого уровня делятся на компилирующие и интерпретирующие. Первые создают программы в машинном коде, вторые непосредственно исполняют программы на языке высокого уровня. Достоинство компиляторов – эффективность при выполнении программ, достоинство интерпретаторов – гибкость при создании программного обеспечения.

Процесс создания программного обеспечения включает несколько этапов (рис. 1.1). На этапе концептуального дизайна определяется архитектура будущей программы или комплекса программ. На этапе кодирования пишется программный код отдельных модулей. Следующие два этапа – компиляция и построение – приводят к созданию готовых к исполнению программ. Их работоспособность проверяется на последнем этапе, который называется отладкой.



Рис. 1.1

При компиляции программы, написанной на языке высокого уровня, компилятором создается образ будущей исполняемой программы. Обычно это делается в два этапа. На первом компилятор обрабатывает исходные фай-

лы программы и создает так называемый объектный код. На втором этапе осуществляется построение программы – сборка всех модулей, включая библиотечные, в единый образ задачи (exe-файл).

Подготовленные к выполнению программы могут храниться, переноситься на другие компьютеры, имеющие аналогичную архитектуру, и запускаться в тот момент, когда требуется выполнение соответствующей задачи. При запуске программы ее образ загружается из файла в оперативную память. После загрузки в память управление передается на определенный адрес – *точку входа* – и процессор начинает выполнять программу.

Загрузка и выполнение программы происходит под управлением *операционной системы* – комплекса специальных программ, управляющих работой компьютера. Основное назначение операционной системы – управление ресурсами компьютера (процессором, памятью и периферийными устройствами) и их предоставление различным прикладным программам.

По способу управления ресурсами операционные системы делятся на *системы пакетной обработки, системы разделения времени и системы реального времени*. Системы пакетной обработки обеспечивают последовательное выполнение входного пакета программ-заданий. Системы разделения времени распределяют время процессора между одновременно выполняемыми программами в зависимости от их готовности к выполнению и приоритетов. Системы реального времени перераспределяют ресурсы при наступлении внешних событий в пользу программ, которые должны оперативно реагировать на эти события.

Одновременное выполнение нескольких программ на одном процессоре возможно с применением техники *мультипрограммирования*. При этом процессор периодически переключается с выполнения одной задачи на другие. Для того чтобы возобновить выполнение программы с того места, на котором оно было прервано, операционная система при переключении сохраняет *контекст* выполнения программы – состояние регистров процессора, в том числе программного счетчика и указателя стека. Для возобновления выполнения программы достаточно загрузить в регистры процессора сохраненные значения – и процессор продолжит работу над программой.

Управление памятью заключается в ее распределении между программами, выгрузке программ при нехватке памяти и в их восстановлении.

Глава 2. ЯЗЫК ПРОГРАММИРОВАНИЯ C

Язык C (си) – стандартизированный язык процедурного программирования – был создан в начале 1970-х гг. Кеном Томпсоном и Денисом Ричи. От других языков его отличали небольшое число элементов языка, высокая скорость выполнения программ, поддержка модульного программирования, хорошая мобильность (переносимость создаваемых программ), а также возможность работы на «нижнем уровне» [1].

В 1989 г. проект языка C был принят комитетом ANSI, а затем и Международной организацией по стандартизации (ISO). При этом комитетом ANSI был принят ряд директив, направленных на сохранение характерных особенностей языка:

1. Существующий код важен, существующий инструментарий – нет. Следует избегать внесения изменений в существующие программные коды. В крайнем случае следует менять компилятор, но не программный код.
2. C-программы должны быть мобильными. Стандарт ANSI предоставляет программисту возможность переносить программы без изменений в среды других операционных систем.
3. C-программы могут быть и немобильными. Программист не обязан ограничивать свою свободу стандартом, он может писать и немобильные программы, привязанные к определенной аппаратной среде.
4. Стандарт – это договор между разработчиком языка и программистом. При согласовании изменений должны учитываться интересы как разработчиков компиляторов, так и пользователей-программистов.

2.1. Структура C-программы

C-программа состоит из определения данных и неограниченного числа программных блоков – функций, одна из которых должна именоваться `main`. Функция `main` представляет собой точку входа в программу, т. е. ей передается управление после запуска программы. Далее приведен пример простейшей программы на C, выводящей строку на терминал (он приводится практически во всех учебниках по языку C).

```
void main()  
{  
    printf("Hello, world.\n");  
}
```

Следует отметить, что в программах, написанных для работы в среде операционной системы Windows, функция `main` заменяется на специальную функцию `WinMain`.

2.2. Синтаксис языка C

Текст программы на языке C представляет набор операторов. Каждый оператор должен заканчиваться точкой с запятой. Отдельно встречающаяся точка с запятой называется *пустым оператором* и используется как разделитель. Для улучшения читаемости программы между операторами могут включаться пробелы, знаки табуляции и пустые строки.

Группа операторов, заключенная в фигурные скобки, называется *блоком операторов* или *составным оператором*.

```
void main()  
{  
    a = 1; b = 2; c = 3;  
  
    d = 5;  
  
    if(a < b)  
    {  
        ;  
    }  
}
```

Для улучшения понимания логики программы в нее могут включаться комментарии. Комментарии в C заключаются в `/* ... */`. В языке C++ комментарии также могут начинаться с `//`. Ниже приведен пример использования комментариев для пояснения программы в целом, а также отдельных строк.

```
/*  
Демонстрационная программа  
Версия 1.0  
*/  
  
void main()  
{  
    a = 1; b = 2; c = 3;          /* присвоение значений */  
  
    if(a < b) c = d;              // проверка условия  
}
```

Важным элементом синтаксиса языка являются идентификаторы, которые предназначены для присвоения имен элементам программы. В иденти-

фикаторах могут использоваться буквы латинского алфавита, цифры и знак подчеркивания «_». Идентификаторы не могут начинаться с цифры. Прописные и строчные буквы различаются. Максимальная длина идентификатора не ограничивается, но значение имеет только 31 символ от начала, остальные игнорируются. Примеры корректных идентификаторов:

```
Name
_code
nValue
TEXT
text
icon16_16
AddPersonToBase
screen_width
__mode__
```

2.3. Простые типы данных

Языки C и C++ предусматривают строгую типизацию данных. При определении многих элементов языка (констант, переменных, функций, параметров функций и т. п.) для них задается тип данных. Типы данных делятся на простые и составные, к которым относятся массивы, объединения и структуры. В свою очередь простые типы данных делятся на две группы – целочисленные и числа с плавающей точкой.

Целочисленные типы данных используются для представления целых чисел, символов, логических и перечислимых значений.

Логический тип *bool* (применяется только в языке C++) имеет только два значения – *false* или *true*, которые являются зарезервированными словами. Если переменной логического типа присвоено целое значение, то 0 интерпретируется как *false*, а значение, не равное нулю, как *true*. В памяти *bool* занимает 1 байт.

Для представления целых чисел имеются два базовых типа *char* и *int*. Тип *char* занимает в памяти 1 байт и используется, главным образом, для представления символьных данных. Размер типа *int* зависит от архитектуры компьютера и, как правило, равен длине машинного слова. Таким образом, на 32-битовых платформах тип *int* занимает 4 байт. Для задания в явном виде размера типа *int* к нему могут применяться модификаторы *short int* и *long int*, которые означают, что на любой архитектуре размер типа *int* будет, соответственно, 2 и 4 байт. Часто для упрощения программ для типа *int* пишут толь-

ко модификаторы `short` или `long`, и это нормально воспринимается всеми компиляторами.

Все целые типы данных могут восприниматься как со знаком, так и без знака. Для задания этого в явном виде используются модификаторы *signed* и *unsigned* соответственно. По умолчанию числа воспринимаются со знаком, поэтому модификатор `signed` практически не используется при написании программ. В табл. 2.1 приведена сводная информация о целочисленных типах данных.

Таблица 2.1

Тип	Размер, байт	Диапазон
<code>bool</code>	1	false, true
<code>signed char (char)</code>	1	-128...127
<code>unsigned char</code>	1	0...255
<code>signed short int (short)</code>	2	-32 768...32 767
<code>unsigned short int (unsigned short)</code>	2	0...65 535
<code>signed long int (long)</code>	4	$-2^{31} \dots (2^{31}-1)$
<code>unsigned long int (unsigned long)</code>	4	$0 \dots (2^{32}-1)$

На базе целого типа `int` могут создаваться специализированные типы данных – *перечисления*. Перечисление является типом данных, все возможные значения которого задаются списком целочисленных констант:

```
enum [имя_типа] { имя1[ = значение1], имя2[ = значение2], ... };
```

Элементы перечисления могут инициализироваться явно или автоматически. Те элементы, которые не инициализированы явно, получают значения предшествующих им элементов, увеличенные на единицу. Первому элементу, если он не инициализирован, присваивается значение 0. Размер перечислимых типов данных зависит от установленных опций компилятора – либо он равен типу `int`, либо вычисляется компилятором по наибольшему из значений констант. Примеры перечислимых типов данных:

```
enum Animal {Cat, Dog, Tiger, Elephant};
enum Error {ERR_READ = 101, ERR_WRITE = 105};
enum Status {Free = 1, Working, Ok};
enum Figure {Rectangle = 10, Square, Ellipse = 20, Circle} ;
```

Для представления вещественных чисел в языке C имеется три типа данных с плавающей точкой – *float*, *double* и *long double*, которые различают-

Таблица 2.2

Тип	Размер, байт	Диапазон	Точность
float	4	$3.4 \cdot 10^{-38} \dots 3.4 \cdot 10^{38}$	7
double	8	$1.7 \cdot 10^{-308} \dots 1.7 \cdot 10^{308}$	15
long double	10	$3.4 \cdot 10^{-4932} \dots 3.4 \cdot 10^{4932}$	19

ся размером занимаемой памяти (4, 8 и 10 байт соответственно), диапазоном и точностью представления (определяемой числом значащих цифр). В табл. 2.2 приведена сводная информация о типах данных с плавающей точкой.

2.4. Сложные типы данных

Сложные типы данных строятся на основе объединения простых или других сложных типов. В языке С имеется два механизма создания сложных типов данных – структуры и объединения.

Структуры объединяют под одним именем различные данные. Синтаксис объявления структуры имеет следующий вид:

```
struct [имя_структуры] { тип_1 элемент_1; тип_2 элемент_2; ... };
```

В памяти элементы структуры размещаются последовательно, при этом компилятор выравнивает данные таким образом, что смещение любого элемента от начала структуры всегда кратно размеру этого элемента, а общий размер структуры всегда кратен размеру ее наибольшего элемента. В результате в структуре могут появляться неиспользуемые байты. Выравнивание данных может приводить к неожиданным результатам – так, в примере на рис. 2.1 размеры структур A1 и A2 различаются на 4 байта.

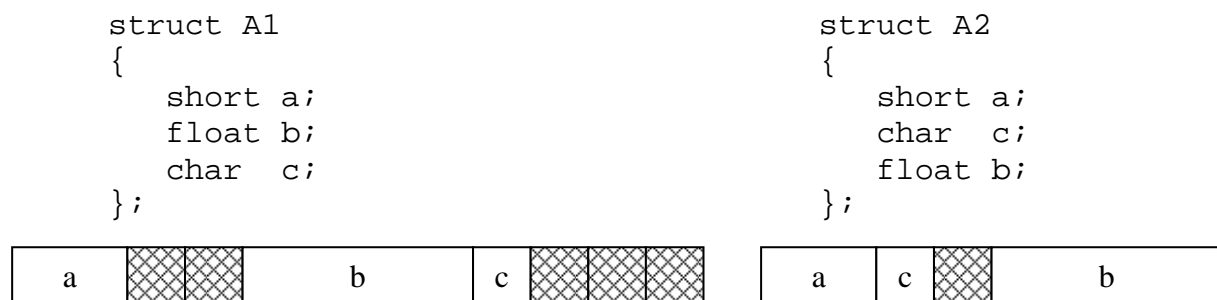


Рис. 2.1

Объединения отличаются от структур тем, что в памяти элементы объединения «накладываются» друг на друга. Элементы объединения никогда не используют совместно, поскольку в этом случае они будут затирать друг

друга. Размер, занимаемый в памяти объединением, всегда равен размеру его наибольшего элемента. Синтаксис объявления объединения имеет следующий вид:

```
union [имя_объединения] { тип_1 элемент_1; тип_2 элемент_2; ... };
```

Для конвертации одних типов данных в другие в языке имеются операторы приведения типа. Синтаксически такие операторы представляются именем типа данных, заключенным в круглые скобки, например (int).

2.5. Константы и переменные

В языке C для представления неизменяемых данных используются *константы*. Синтаксис объявления константы имеет следующий вид:

```
const тип имя_константы = значение;
```

Для задания значений констант применяются следующие правила. Целые числовые константы, которые начинаются с цифры, отличной от нуля, интерпретируются как десятичные. Начинаящиеся с нуля константы интерпретируются как восьмеричные. Константы, начинающиеся с «0x» или с «0X», интерпретируются как шестнадцатеричные. Если в числовой константе встречается точка, она воспринимается как число с плавающей точкой. Для чисел с плавающей точкой также может применяться экспоненциальный формат (с символом «e» или «E»). Примеры значений числовых констант:

```
100          // десятичное число 100
0100         // восьмеричное число 100 (десятичное 64)
0x100        // шестнадцатеричное число 100 (десятичное 256)
0xA          // шестнадцатеричное число A (десятичное 10)
3.14         // число с плавающей точкой
5.67e4       // число с плавающей точкой в экспоненциальном формате
-5           // отрицательное целое число
-.12         // отрицательное число с плавающей точкой
```

Для хранения в памяти изменяемых данных используются *переменные*, которые объявляются следующим образом:

```
тип имя_переменной1 [= значение] [, имя_переменной2 [= значение]];
```

Переменные и константы всегда должны объявляться до их использования. В памяти переменные и константы размещаются по адресу, кратному своему размеру. При объявлении переменным могут присваиваться начальные значения. Пример объявления констант и переменных:

```
short a;
int day, month, year;
const double PI = 3.14;
unsigned long x, y = 255, z;
```

Переменные сложных типов (структуры, объединения) объявляются аналогично простым. При инициализации значений элементов структур (для объединений инициализация нескольких элементов не имеет смысла) они заключаются в фигурные скобки. Для доступа к отдельным элементам переменных сложной структуры используется оператор «.».

```
struct Point { int x; int y };
Point p1 = {25, 38}, p2;
p2.x = 25;
p2.y = 38;
```

2.6. Массивы

Массив – это размещение в памяти блока переменных одного типа. Синтаксис объявления массива показан ниже, при этом прямые скобки являются обязательным элементом:

Тип_данных имя_массива [число_элементов];

Тип данных и число элементов определяют объем памяти, выделяемой под массив. Число элементов может быть задано константами или константными выражениями. Значение числа элементов должно выражаться целым положительным числом. Пример объявления массивов:

```
int months[12];
Animal animals[200];
enum {MIN = 20, MAX = 200};
double values[MIN*10];
```

При определении можно одновременно инициализировать массив. Для этого после имени массива ставят знак равенства и в фигурных скобках по порядку перечисляют значения элементов массива. Нет необходимости инициализировать все элементы, в этом случае неинициализированным элементам присваивается значение 0. Если значений больше, чем позволяет размер массива, то выдается сообщение об ошибке. Если размер массива не указан, то он вычисляется компилятором по числу инициализирующих элементов. Пример инициализации массивов:

```
float values1[3] = {1.23, 4.56};
float values2[] = {1.23, 4.56};
```


В данном примере третий элемент массива `values1` будет инициализирован нулем. В массиве `values2` будет два элемента.

Доступ к элементам массива осуществляется при помощи индекса, указанного в квадратных скобках. Индекс первого элемента массива всегда равен 0. Индекс последнего элемента равен числу элементов массива, уменьшенному на единицу. Пример доступа к элементам массива:

```
float values[3] = {1.23, 4.56};  
float x = values[2];  
values[2] = 7.89;  
int index = 1;  
float y = values[index+1];
```

Язык C поддерживает многомерные массивы. Размерность определяется числом индексов, используемых для ссылки на конкретный элемент массива. Элементы многомерного массива последовательно размещаются в памяти, при этом быстрее всего меняется последний индекс. Пример инициализации и работы с многомерными массивами:

```
int matrix1[3][2] = {{11, 12}, {21, 22}, {31, 32}};  
int matrix2[3][2] = {11, 12, 21, 22, 31, 32};  
int matrix3[3][2];  
matrix3[0][0] = 11;  
matrix3[0][1] = 12;  
matrix3[1][0] = 21;  
matrix3[1][1] = 22;  
matrix3[2][0] = 31;  
matrix3[2][1] = 32;
```

В данном примере все три массива `matrix1`, `matrix2` и `matrix3` будут заполнены одинаковыми данными. Способ инициализации массива `matrix2` показывает, что многомерный массив размещается в памяти как одномерный, а наличие нескольких индексов просто облегчает вычисление конечного индекса элемента в памяти.

Особым видом массивов являются *строки*, которые представляют собой массивы типа `char`. Элементами таких массивов являются символы, кодируемые однобайтовыми целыми числами в соответствии с принятыми правилами. Такие правила называются *кодowymi таблицами*. Для первых 127 символов принята общая кодировка, которая называется код ASCII. Для символов с кодами 128–255 существуют различные таблицы, например Windows-1251 для кириллицы, Windows-1252 для западных языков, Windows-1253 для греческого языка, и т. д.

В языке С принято соглашение, по которому строки завершаются нулевым байтом (нуль-терминатор). Для удобства написания программ в языке С имеются символьные константы, которые заключаются в апострофы, а также строковые константы, заключаемые в кавычки. Для включения в строки непечатаемых символов имеются специальные последовательности, начинаемые с обратной наклонной черты, например `\n` означает перевод строки, `\r` – возврат каретки, `\t` – табуляцию, `\"` – кавычки. Пример работы с символами и строками приведен ниже:

```
char country1[] = "Russia";
char country2[7] = {'R', 'u', 's', 's', 'i', 'a', 0};
char country3[7];
country3[0] = 'R';
country3[1] = 'u';
country3[2] = 's';
country3[3] = 's';
country3[4] = 'i';
country3[5] = 'a';
country3[6] = 0;
```

Зачастую на практике создаются массивы сложных типов данных. В следующем примере показана декларация структуры, объявление и инициализация массива данных. При инициализации структуры ее элементы заключаются в фигурные скобки, что позволяет инициализировать не все элементы, а только их часть.

```
struct Product
{
    int    code;
    char   name[20];
    float  price;
    char   comment[256];
};

Product products[20] =
{
    {1, "Яблоки",      55.50, "Сорт \"Гольден\""},
    {2, "Апельсины",  45.00},
    {5, "Бананы",      22.00},
    {8, "Груши",       64.45, "Очень спелые!"},
    {4, "Сливы",       82.50},
    {6, "Грейпфруты"},
};
```

2.7. Размещение данных в памяти. Адреса. Указатели

Размещение переменных в памяти программы зависит от места и способа их объявления. Переменные, объявляемые внутри блоков, называются локальными. Их область видимости ограничивается пределами блока. Переменные, объявляемые вне блоков, называются глобальными. Их область видимости – весь компилируемый файл. В приведенном далее примере переменная *A* – глобальная, а *B* – локальная, и ее время жизни заканчивается при выходе программы из блока.

```
int A = 1;

{
    int B = 5;
}
```

Для управления размещением переменных в памяти применяются спецификаторы классов памяти. Ключевое слово *auto* явно определяет переменную как автоматическую. Эта характеристика класса памяти применяется только внутри области блока, так как время жизни автоматических переменных ограничивается временем выполнения блока, в котором они описаны. Поскольку по умолчанию локальные переменные – автоматические, то этот спецификатор на практике как правило не используется.

Переменные, описанные с ключевым словом *register*, представляют частный случай автоматических переменных. Использование таких переменных позволяет повысить скорость выполнения программы. Если такая опция задана при компиляции, переменные по возможности будут размещены в регистрах процессора.

Для глобальных переменных ключевое слово *static* означает локальную в пределах файла видимость и однократную инициализацию. Для локальных переменных *static* означает сохранение значений между вызовами функций.

С помощью спецификатора *extern* переменная явно устанавливается как внешняя, т. е. находящаяся в другом модуле (отдельно компилируемом файле). Пример использования классов памяти приведен ниже.

```
extern double Value;
{
    register int i = 0;
    static short a;
    auto int x, y;
}
```

Для определения размера памяти, требуемой для размещения переменной, применяется оператор *sizeof*, который возвращает число байтов, занимаемых в памяти любым элементом данных – типом, переменной, массивом, структурой. В следующем примере программы оператор *sizeof* используется для определения числа элементов инициализируемого массива:

```
long y[] = {43, 56, 34};
int n = sizeof(y)/sizeof(long);
```

Оператор *typedef* позволяет определять новые типы данных на основе уже существующих, в том числе сложных. Ниже приведен пример использования данного оператора, для создания типов данных *number*, *Vector* и *LONG_ARRAY*:

```
typedef short number;
typedef struct {int x; int y} Vector;
typedef long[64] LONG_ARRAY;
```

Поскольку все переменные находятся в оперативной памяти и имеют определенные адреса, то работа с ними возможна не только по имени, но и непосредственно по адресу. Для этого в языке C имеются специальные виды переменных – *указатели*. Указатель представляет собой переменную, которая хранит адрес другой переменной. Размер указателя зависит от платформы и всегда равен размеру типа *int*.

При объявлении указателя для него обязательно указывается тип данных, на который указывает данный указатель. Исключение составляет так называемый неопределенный указатель (тип *void**). Декларация указателя аналогична любой другой переменной, при этом имя указателя предваряется символом «*». Можно создавать массивы из указателей, указатели могут быть элементами структур и объединений. На базе указателей также можно создавать новые типы данных, используя оператор *typedef*. Пример объявления указателей:

```
int *p1, *p2;
typedef char* TextPointer;
TextPointer a, b, c;
double * val[32];
```

В языке C возможно создание указателей на указатели. Такой указатель содержит адрес памяти, по которому находится другой указатель. В объявлении такого указателя используются два символа «*»:

```
char** pointer;      // pointer – указатель на указатель на char
```

Следует отметить, что имя массива фактически является указателем на первый элемент массива, поэтому при работе с указателями также можно применять индексацию.

Инициализация указателя заключается в присвоении ему адреса и выделении по этому адресу соответствующей типу указателя области памяти. Инициализацию можно выполнять различными способами:

- присваивание адреса существующего объекта (с помощью оператора `&`, указателя, массива);
- присваивание `NULL`;
- динамическое выделение памяти (`malloc` или `new`).

Для получения адреса переменной в языке C имеется оператор «`&`», который указывается перед именем переменной. Этот адрес может быть присвоен указателю на соответствующий тип данных. Для того чтобы отличить указатель, который не содержит никакого адреса, используется специальное имя `NULL` («пустой»). Для получения значения переменной по указателю используется так называемая разадресация указателя через оператор «`*`». Ниже приведены примеры инициализации указателей и работы с ними.

```
int x;                // целая переменная x
int *px = &x;         // указатель px содержит адрес переменной x
int *py = NULL;       // указатель py не содержит адреса
py = px;              // указатель py содержит адрес переменной x
*py = 5;              // переменной x присваивается значение 5
*py = *px;            // переменной y присваивается переменная x
```

Указатели часто используются для работы с *динамической памятью*. В динамической памяти, как правило, размещаются переменные, которые используются эпизодически и требуют большого объема памяти. В стандартной библиотеке языка C имеются функции для управления динамической памятью. К ним относятся *malloc* (выделение памяти), *realloc* (изменение размера блока памяти) и *free* (освобождение памяти). В функцию *malloc* передается размер требуемой памяти, и она возвращает адрес выделенного блока (или `NULL`, если выделение невозможно). Функции *realloc* передается адрес блока и новый размер, она возвращает новый адрес – при этом старые данные копируются по новому адресу. В функцию *free* передается адрес блока, который может быть высвобожден. Функции *malloc* и *realloc* возвращают указатель на неопределенный тип данных (`void*`), который должен быть приведен к нужному типу указателя. Пример работы с динамической памятью:

```

int *x = (int*)malloc(10000*sizeof(int)); // выделение памяти
x[0] = 10;                               // ячейка 0
x[1] = 20;                               // ячейка 1
x = (int*)realloc(x, 20000*sizeof(int)); // изменение размера
x[19999] = x[0]+x[1];                    // ячейка 19999
free(x);                                 // освобождение памяти

```

В языке C++ для динамического выделения памяти также может использоваться оператор *new*. Для освобождения памяти в этом случае используется оператор *delete*.

```

double *a = new double; // выделение памяти под тип double
int *x = new int[100];   // выделение памяти под массив int
Person *p = new Person;  // выделение памяти под структуру Person
delete a;                // освобождение памяти
delete[] x;              // освобождение памяти массива (с [])
delete p;                // освобождение памяти

```

Нельзя смешивать концепции *malloc/free* и *new/delete*, поскольку при этом задействуются различные механизмы управления памятью. Таким образом память, выделенную с помощью *malloc*, можно высвободить только с помощью *free*, а выделенную с помощью *new* – только с помощью *delete*.

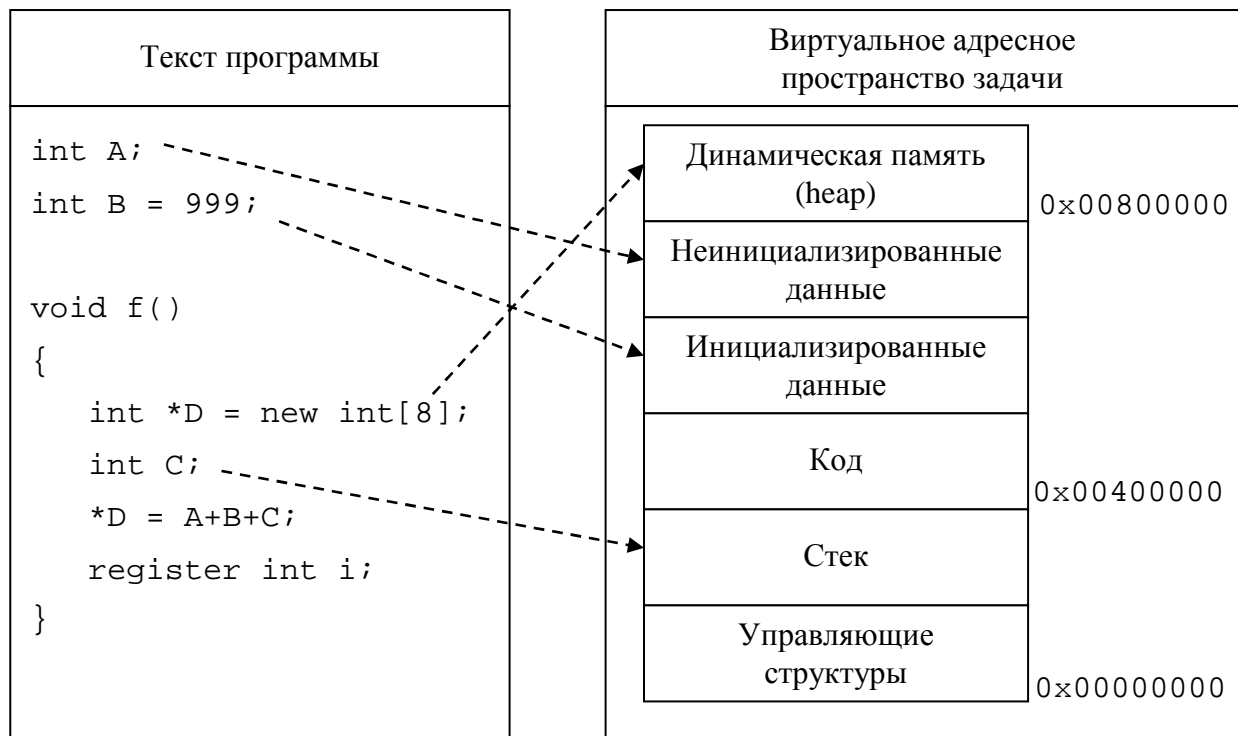


Рис. 2.2

Часто указатели используют для доступа к элементам структур или объединений. Доступ к элементам структуры, адрес которой содержится в указателе, осуществляется с использованием оператора «->»:

```
typedef struct
{
    int    code;
    char   name[20];
    float  salary;
} Person;

Person persons[100];
Person* p = &persons[20];
p->salary = 200.0;
```

Подводя итоги размещению переменных в памяти программы, следует отметить, что переменные разных классов памяти компилятор размещает в различных областях адресного пространства задачи (рис. 2.2). Глобальные переменные размещаются в области инициализированных или неинициализированных переменных (переменные В и А соответственно). Локальные переменные размещаются в стеке (С и D). Динамическая память, выделяемая оператором `new`, находится в отдельной области адресного пространства (ее называют `heap` – «куча»). Для регистровых переменных память не требуется.

2.8. Операторы

Для выполнения различных операций над переменными и константами (выражений) в языке С имеется набор операторов, которые по их назначению можно отнести к следующим классам:

- знаковые;
- арифметические;
- присваивания;
- отношения;
- логические;
- доступа;
- побитовые;
- прочие.

Все операторы можно также систематизировать по группам в зависимости от того, со сколькими операндами они работают:

- унарные (один операнд);
- бинарные (два операнда);
- тернарные (три операнда).

К знаковым операторам относятся *унарный плюс* и *унарный минус*. Эти операторы позволяют присваивать знак арифметическим выражениям. Поскольку по умолчанию все числа воспринимаются как положительные, унарный плюс практически не используется. Пример знаковых операторов:

```
int a = -5;
float b = +45.67;
double c = -b;
```

Все арифметические операторы (табл. 2.3) – бинарные. Операции умножения, деления и остатка имеют более высокий приоритет, чем операции сложения и вычитания. При равном приоритете операторы обрабатываются в последовательности слева направо. Можно изменить порядок выполнения операторов с помощью скобок.

Таблица 2.3

Оператор	Назначение
+	Суммирует два операнда
-	Вычитает из первого операнда второй
*	Умножает два операнда
/	Делит первый операнд на второй
%	Остаток от целочисленного деления первого операнда на второй

В следующем примере показано использование арифметических операторов:

```
int a = 5+3;           // a будет равно 8
int b = a-2;           // b будет равно 6
int c = a*b;           // c будет равно 48
int d = c/10;           // d будет равно 4 (целое деление!)
int e = c%10;           // e будет равно 8 (остаток)
int f = 2+3*4-5;        // f будет равно 9
int g = (2+3)*(4-5);    // g будет равно -5
double h1 = 48/10;      // h1 будет равно 4 (целое деление!)
double h2 = 48./10.;    // h2 будет равно 4.8
```

Операторы присваивания делятся на простые и составные. При выполнении *простого оператора присваивания*, который обозначается знаком «=», левому операнду присваивается значение правого операнда. Слева от любого оператора присваивания должно находиться так называемое *L-значение* (выражение, ссылающееся на некоторую именованную область памяти). Примером L-значения может быть имя переменной или разадресованный указатель.

Составные операторы присваивания (табл. 2.4) выполняют арифметические или побитовые операции над двумя операндами и присваивают получившийся результат первому операнду.

Таблица 2.4

Оператор	Назначение
<code>+=</code>	Суммирует два операнда и присваивает результат первому
<code>-=</code>	Вычитает из первого операнда второй и присваивает результат первому
<code>*=</code>	Умножает два операнда и присваивает результат первому
<code>/=</code>	Делит первый операнд на второй и присваивает результат первому
<code>%=</code>	Присваивает остаток от деления первого операнда на второй
<code><<=</code>	Присваивает результат побитового сдвига влево
<code>>>=</code>	Присваивает результат побитового сдвига вправо
<code>&=</code>	Присваивает результат логического И
<code> =</code>	Присваивает результат логического ИЛИ
<code>^=</code>	Присваивает результат логического исключающего ИЛИ

Пример использования в программе операторов присваивания:

```
int a = 5+3;           // простое присваивание, a будет равно 8
a += 2;                // a будет равно 10
a *= 3;                // a будет равно 30
a /= 20;               // a будет равно 1
a %= 2;                // a будет равно 1
a -= 5;                // a будет равно -4
a <=<= 2;               // a будет равно -16
a &= 1;                // a будет равно 0
```

Операторы отношения (табл. 2.5) сравнивают два операнда, которые также могут быть представлены выражениями. Результатом этих операций всегда является значение `true` или `false`.

Таблица 2.5

Оператор	Назначение
<code>==</code>	Сравнение двух операндов на равенство
<code>!=</code>	Сравнение двух операндов на неравенство
<code><</code>	Сравнение двух операндов на «меньше»
<code>></code>	Сравнение двух операндов на «больше»
<code><=</code>	Сравнение двух операндов на «меньше или равно»
<code>>=</code>	Сравнение двух операндов на «больше или равно»

Пример использования в программе операторов отношения:

```
int a = 2, b = 3, c = 4, d = 5;
bool e = (b+d == a*c);           // e будет равно true
bool f = (2*a != c);             // f будет равно false
bool g = (d > b);                 // g будет равно true
bool h = (d < b);                 // h будет равно false
bool i = (b-a >= -2);            // i будет равно true
bool j = (-2 <= a-c);            // i будет равно true
```

Логические операторы (табл. 2.6) выполняют операции булевой алгебры над операндами логического типа. Для задания порядка операций могут использоваться круглые скобки.

Таблица 2.6

Оператор	Назначение
&&	Логическое И между двумя операндами (бинарный оператор)
	Логическое ИЛИ между двумя операндами (бинарный оператор)
!	Логическое НЕ (унарный оператор)

Пример использования в программе логических операторов:

```
bool a = true, b = false;
bool c = a && b;                 // c будет равно false
bool d = a || b;                 // d будет равно true
bool e = !a || b;                // e будет равно false
bool f = !(a && b);              // f будет равно true
```

Побитовые операторы (табл. 2.7) выполняют операции над отдельными разрядами целых чисел. Побитовое И возвращает единицы в тех разрядах, в которых у обоих операндов установлены единицы. Побитовое ИЛИ возвращает единицы в тех разрядах, в которых хотя бы у одного из операндов установлены единицы. Исключающее ИЛИ возвращает единицы в тех разрядах, в которых значения битов не совпадают. Побитовое НЕ инвертирует биты. Сдвиг влево на разряд эквивалентен умножению на 2, вправо – делению на 2.

Таблица 2.7

Оператор	Назначение
&	Побитовое И (бинарный)
	Побитовое ИЛИ (бинарный)
^	Побитовое исключающее ИЛИ (бинарный)
~	Побитовое НЕ (унарный)
>>	Сдвиг первого операнда вправо на заданное вторым число разрядов
<<	Сдвиг первого операнда влево на заданное вторым число разрядов

Пример использования в программе побитовых операторов:

```
unsigned char a = 1, b = 2;  
unsigned char c = a&b;           // c будет равно 0  
unsigned char d = a|b;           // d будет равно 3  
unsigned char e = a^b;           // e будет равно 3  
unsigned char f = ~a;            // f будет равно 254  
unsigned char g = b<<3;          // g будет равно 16  
unsigned char h = 7>>1;          // h будет равно 3
```

Операторы *инкремента* и *декремента* предназначены для увеличения или уменьшения значения переменных соответственно (табл. 2.8). Позиция операторов инкремента и декремента по отношению к операнду определяет, какое действие будет выполнено сначала: возвращение значения операнда или его изменение. В префиксной форме операнд изменяется до возвращения. В постфиксной форме в качестве возвращаемого значения выражения используется значение операнда до его изменения. Для арифметических типов данных величина увеличения и уменьшения равна 1. Операции также могут применяться к указателям, в этом случае значение указателя меняется на размер соответствующего ему типа данных.

Таблица 2.8

Оператор	Назначение
L-значение++	Инкрементация операнда, постфиксная форма
++L-значение	Инкрементация операнда, префиксная форма
L-значение--	Декрементация операнда, постфиксная форма
--L-значение	Декрементация операнда, префиксная форма

Пример использования в программе различных форм операторов инкремента и декремента:

```
short a = 20;  
short b = a++;           // b будет равно 20  
short c = ++a;           // c будет равно 22  
short d = --a;           // d будет равно 21  
short e = a--;           // e будет равно 21  
int f[] = {5, 7, 9, 11}; // массив из четырех элементов  
int *p = &f[1];          // указатель на элемент с индексом 1  
int g = *p++;             // g будет равно 7, p увеличится на 4
```

Для управления последовательностью вычисления выражений применяется оператор «запятая». Два выражения, разделенные запятой, вычисляются слева направо, и значение левого выражения отбрасывается. Данный оператор в основном применяется для обработки нескольких выражений там,

где разрешено использование только одного выражения. Пример использования оператора «запятая»:

```
int i = 0;
float a = 5.25;
float b = (i++, a*2);    // i будет равно 1, b будет равно 10.5
```

Таблица 2.7

Приоритет	Оператор	Ассоциативность
↑	[] . -> ~ ! *(разадресация) &(адрес) ++ --	Слева направо
	sizeof()	Справа налево
	*(умножение) / %	Слева направо
	+ -	Слева направо
	<< >>	Слева направо
	&(И)	Слева направо
	^	Слева направо
		Слева направо
	&&	Слева направо
		Слева направо
	= += -= *= /= %= <<= >>= &= = ^=	Справа налево
	,	Слева направо

При использовании нескольких операторов в выражении порядок их обработки будет определяться двумя характеристиками – *приоритетом* и *ассоциативностью* (табл. 2.7). Сначала выполняются операторы с более высоким приоритетом, затем с более низким. Для операторов с одинаковым приоритетом порядок определяется ассоциативностью. В любом случае порядок можно изменить с помощью круглых скобок.

2.9. Управление программным потоком

Для управления ходом выполнения программы в языке С имеются операторы безусловного и условного переходов, а также операторы цикла.

Оператор безусловного перехода *goto* позволяет реализовать передачу программного управления из одной точки программы в другую, отмеченную меткой. Метка состоит из идентификатора и завершающего двоеточия.

```
a = b+c;
goto M5;
...
```

М5:

```
d = e-a;
```

Оператор условного ветвления *if* позволяет выполнять следующий за ним оператор или блок кода в зависимости от некоторого условия. Оператор имеет следующий синтаксис:

```
if(выражение) { блок кода }
```

Сначала вычисляется выражение в круглых скобках. Если выражение истинно, то выполняется следующий оператор (блок кода). Если выражение ложно, то следующий оператор (блок кода) не выполняется и управление передается за него. Пример использования оператора *if*:

```
int a;
int b;
...

if(a < b)
{
    int c = a;
    a = b;
    b = c;
}

int d = a;
```

Оператор условного ветвления *if-else*, позволяющий выполнять один из следующих за ним операторов или блоков кода, имеет следующий синтаксис:

```
if(выражение) { блок кода «true» } else { блок кода «false» }
```

Сначала вычисляется выражение в круглых скобках. Если выражение истинно, то выполняется следующий оператор или блок кода. Если выражение ложно, то выполняется оператор или блок кода, следующий за *else*.

```
int a;
int b;
...
int c;

if(a >= b)
    c = a;
else
    c = b;

int d = c;
```

С помощью ключевых слов `if` и `else` можно составлять так называемые вложенные *else-if*-конструкции, которые могут проверить сразу несколько выражений. Пример использования таких конструкций:

```
int a;
char* p;
...
if(a >= 1000)
    p = "Очень много!";
else
if(a >= 100 and a < 1000)
    p = "Много!";
else
if(a >= 10 and a < 100)
    p = "Мало!";
else
    p = "Очень мало!";
```

Существует специальная компактная форма условного оператора, которая часто применяется в выражениях. Эта форма имеет следующий вид:

выражение1 ? выражение2 : выражение3

Смысл указанной конструкции заключается в следующем. При истинности выражения 1 возвращается значение выражения 2, в противном случае – значение выражения 3. Пример использования данного оператора:

```
int a = (b >= 0 ? b : -b);
```

Оператор условного перехода *switch* иногда обеспечивает более наглядную технику программирования. Вычисляется целочисленное выражение и управление передается в одну из точек программы, указанных метками (*case-константами*). Если ни с одной из *case-констант* совпадения нет, то управление передается на конструкцию с *default-меткой* при условии ее наличия, в противном случае ни один из операторов не выполняется. Использование оператора *switch* иллюстрирует следующий пример:

```
int note; char* text;

switch(note)
{
case 5: text = "Отлично";          break;
case 4: text = "Хорошо";           break;
case 3: text = "Удовлетворительно"; break;
case 2: text = "Неудовлетворительно"; break;
default:
    text = "Ошибка";
}
```

В языке C для организации циклов имеются три оператора. Цикл *while* является циклом с предпроверкой условия и имеет синтаксис

while(выражение) { блок кода }

Следующий за выражением оператор или блок кода будет многократно выполняться, пока выражение будет иметь отличное от нуля значение:

```
double x = 5, y = 1; int n = 10, i = 0;
```

```
while(i < n)
{
    y *= x;
    i++;
}
```

Цикл *do-while* является циклом с проверкой условия выхода после выполнения тела цикла:

do { блок кода } while(выражение);

Следующий за выражением блок кода будет выполняться хотя бы один раз. Цикл завершится, когда выражение будет иметь значение, равное нулю.

Цикл *for* является циклом с предпроверкой условия выхода:

for(выражение 1; выражение 2; выражение 3) { блок кода }

В цикле могут инициироваться переменные (выражение 1), проверяться условия (выражение 2) и выполняться действия после каждого выполнения тела цикла (выражение 3):

```
double x = 5, y = 1; int n = 10;
```

```
for(int i = 0; i < n; i++)
    y *= x;
```

Для управления программным потоком в циклах часто в сочетании с условными операторами применяются операторы *break* и *continue*. Оператор *continue* возвращает управление к началу цикла. Оператор *break* завершает выполнение цикла:

```
int N = 10; long S = 0;
```

```
for(int i = 1; i <= N; i++)
{
    if(i%2) continue;

    if(S > 10) break;

    S += i;
}
```

2.10. Функции

Программа на С состоит из одной или нескольких функций, имена которых должны быть правильными идентификаторами. Функция есть логически самостоятельная именованная часть программы, которой могут передаваться параметры и которая может возвращать некоторое значение. Разделение программы на функции позволяет избежать избыточности кода, объединить часто используемые функции в библиотеки и создать модульные, более простые в сопровождении программы.

При объявлении функции (также называемом декларацией) задаются ее тип, имя и список параметров. Тип функции – это тип возвращаемого функцией значения. Функция может не возвращать никакого значения, в этом случае в качестве типа используется ключевое слово *void*. В списке параметров задаются через запятую типы и имена параметров. Имена параметров не играют никакой роли при декларации функции и задаются, как правило, для улучшения воспринимаемости программы. Пример декларации функций:

```
double square(double length, double width);
short RandomValue();
void func(int a, int b, char* p);
```

Определение функции (или реализация) дополнительно содержит тело функции, представляющее собой заключенную в фигурные скобки совокупность операторов. Пример реализации функции:

```
double square(double length, double width)
{
    return(length*width);
}
```

Оператор *return* может встречаться в любом месте функции. Этот оператор завершает выполнение функции и возвращает указанное значение (или результат выражения) вызывающей программе.

При вызове функции ей в качестве параметров передаются константы, переменные или выражения. Вызывать функцию можно из любого места программы, реализовывать – также в любом месте кода. Единственным условием является необходимость декларировать функцию до ее вызова. Пример декларации, реализации и использования функции приведен далее:

```
int abs(int val);                // Декларация функции
...
int abs(int val)                 // Реализация функции
```



```

{
    if(val < 0) val = -val;

    return(val);
}
...
int a = abs(b);           // Вызов функции
int c = a+abs(b+d);       // Вызов функции

```

Аналогично переменным для функций при декларации могут быть заданы классы памяти *extern* (по умолчанию) или *static*. Если класс не задан (или задан *extern*), то функция может вызываться любым модулем программы. Если задан класс *static*, то функция может вызываться только в том модуле, в котором она определена. Примеры декларации с заданием классов памяти:

```

extern int f1();           // Класс памяти - extern
int f2();                  // Класс памяти - extern
static int f3();           // Класс памяти - static

```

Параметры предназначены для передачи некоторых значений из вызывающей функции в вызываемую. Параметры – это всегда локальные переменные, область видимости которых – тело функции. Существует два способа передачи параметров функции: *по значению* (call by value) и *по ссылке* (call by reference).

При первом способе передаваемые в функцию переменные копируются в локальные переменные функции (находящиеся в стеке). Вызываемая функция не может воздействовать на передаваемые в качестве параметров переменные.

При втором способе вызова в функцию передаются не копии переменных, а их адреса. Вызываемая функция может изменять значения передаваемых ей переменных. Для передачи адреса при декларации функции используется оператор *&*. Пример двух способов передачи параметров:

<pre> int add(int x) // по значению { x += 1; } ... int a = 5; add(a); int b = a; // b = 5 </pre>	<pre> int add(int &x) // по ссылке { x += 1; } ... int a = 5; add(a); int b = a; // b = 6 </pre>
--	---

В языке C++ имеется возможность задания при декларации функции значений параметров по умолчанию. Если значения по умолчанию задаются не для всех параметров функции, то параметры со значениями по умолчанию должны быть последними в списке параметров. Если при вызове функции заданы не все параметры, то подставляются значения по умолчанию. Пример функции со значениями параметров по умолчанию:

```
int func(int a, int b = 3, int c = 5);           // Декларация
...
int func(int a, int b, int c) {return(a+b+c);} // Реализация
...
int x1 = func(7, 5, 2);                         // x1 = 14
int x2 = func(7, 5);                             // x2 = 17
int x3 = func(7);                               // x3 = 15
int x4 = func();                                 // Ошибка!
```

При вызове функции в стеке резервируется некоторый объем памяти, в котором размещаются копии фактических аргументов, запоминаются точки возврата и значения регистров процессора. Размещение этих данных в памяти требует затрат времени, которые называют затратами на вызов функции (function overhead). Это необходимо учитывать при использовании рекурсивных вызовов – когда функция вызывает саму себя. Пример рекурсивной функции вычисления факториала приведен далее:

```
double factorial(double N)
{
    return(N == 1 ? 1 : N*factorial(N-1));
}
...
double x = factorial(170);    // x = 7.25741561530799e306
```

2.11. Препроцессор

ANSI-стандарт языка C описывает фазу, предшествующую переводу исходного кода программы в машинный код. Такая фаза выполняется препроцессором и включает:

- «склеивание строк» – удаление пары \+перевод строки, получение лексем;
- обработку лексем – замену текста и макрорасширения;
- включение текста из других файлов в исходный файл;
- исключение определенных частей кода (условная трансляция).

«Склеивание строк» заключается в том, что препроцессор выбрасывает пару символов, состоящую из обратной наклонной черты (\) и перевода строки (\n). Следующий программный код:

```
Show\  
Message("Длинная ст\  
рока текста");
```

после обработки препроцессором приобретает вид:

```
ShowMessage("Длинная строка текста");
```

Разделительные символы (пробелы и знаки табуляции) роли при компиляции не играют, поэтому фрагмент кода:

```
if(      a < b      )  
      c = 5;  
a += c;
```

полностью эквивалентен следующему:

```
if(a < b) c = 5; a += c;
```

Директивы препроцессора начинаются с символа # (этот символ должен стоять в начале строки, но перед ним также могут быть и пробелы) и заканчиваются концом строки. Основные директивы приведены в табл. 2.10.

Таблица 2.10

Директива	Назначение
#	Оператор расширения строк
##	Оператор конкатенации лексем
#define	Определение идентификатора или макроса
#undef	Отмена определения
#if	Оператор условной трансляции
#ifdef	Оператор проверки определения
#ifndef	Оператор проверки неопределенного имени
#else	Блок else директивы if
#endif	Завершение директивы if
#include	Включить файл при компиляции
#error	Выдача сообщения об ошибке
#line	Задаёт номер следующей строки

Препроцессор имеет ряд predefined идентификаторов, которые заменяет специальной информацией (табл. 2.11). Эти идентификаторы нельзя повторно переопределять, к ним нельзя применять директиву `#undef`.

Таблица 2.11

Имя	Назначение
<code>__cplusplus</code>	Определено, если компилируется код C++
<code>__DATE__</code>	Дата начала компиляции текущего файла
<code>__FILE__</code>	Имя текущего файла
<code>__FUNC__</code>	Имя текущей функции
<code>__LINE__</code>	Номер текущей строки
<code>__STDC__</code>	Определено, если применяется стандарт ANSI
<code>__TIME__</code>	Время начала компиляции текущего файла

Директива `#include` позволяет включать при компиляции код из других файлов, как правило h-файлов. Имя файла указывается после директивы в кавычках, если включаемый файл находится в текущем каталоге, или в угловых скобках, если файл находится в другом каталоге. Во втором случае каталог должен быть указан в соответствующих настройках компилятора.

Файл `file1.h`:

```
const float PI = 3.14;
```

Файл `file2.h`:

```
typedef struct
{
    int      x;
    int      y;
} Point;
```

Основная программа:

```
#include "file1.h"
#include <file2.h>
...
Point points[100];
points[0].x = 200;
points[0].y = 120;
...
double area = PI*R*R;
```

Мощным средством препроцессора являются макросы, которые позволяют автоматизировать генерацию программного кода и избежать ошибок

при внесении изменений в программы. Макрос можно определить с помощью директивы препроцессора *#define*:

#define имя_макроса последовательность_лексем

Имя макроса должно отвечать требованиям к другим именам программы. Последовательность лексем заканчивается концом строки. Если требуется продолжение макроса на несколько строк, то в конце строки ставится обратная наклонная черта (**).

Отменить определение макроса можно с помощью директивы *#undef*:

#undef имя_макроса

При компиляции имя макроса заменяется на последовательность лексем. Пример использования макросов:

```
#define MAX 200
...
int data[MAX];

for(int i = 0; i < MAX; i++)
    data[i] = 0;
...
#define Red    0x0000FF
#define Green  0x00FF00
#define Blue   0xFF0000
...
int Color = Red;
```

Макросы могут быть определены с аргументами, вследствие чего замещающий текст будет варьироваться в зависимости от задаваемых параметров, как это показано в следующем примере:

```
#define GETINT(var, edit) var = StrToInt(edit->Text)

int rows;
int cols;
GETINT(rows, RowsEdit);
GETINT(cols, ColsEdit);
```

При использовании макросов с аргументами следует быть осторожными, поскольку препроцессор выполняет лишь текстовую подстановку передаваемых параметров. Следующий пример показывает, как применение скобок позволяет получить более безопасный макрос.

Вариант 1:

```
#define square(a, b) (a*b)
...
int s = square(3+1, 5+1);           // s = 9 или 3+1*5+1
```

Вариант 2:

```
#define square(a, b) ((a)*(b))
...
int s = square(3+1, 5+1);           // s = 24 или (3+1)*(5+1)
```

Как видно из приведенных ранее примеров, макросы представляют собой некоторую альтернативу функциям. Выбор того или иного механизма написания программного кода должен основываться на преимуществах и недостатках обоих вариантов. Код макросов компилируется каждый раз, когда встречается вызов макроса. Это приводит к увеличению размера скомпилированного кода программы. Код функции генерируется компилятором один раз, что сокращает размер требуемой для программы памяти. Вместе с тем, на вызов функции и возврат требуется дополнительное время. Поэтому функции предпочтительны с точки зрения размера памяти, а макросы – с точки зрения времени выполнения программы. Сравнительная характеристика функций и макросов приведена в табл. 2.12.

Таблица 2.12

Макросы	Функции
☺ Быстрота выполнения	☹ Дополнительные затраты времени
☹ Большие затраты памяти	☺ Экономия памяти
☹ Нет контроля типов параметров	☺ Контроль типов параметров

Директивы условной трансляции позволяют выборочно включать в текст программы некоторые фрагменты в зависимости от значения заданных условий. Директива *#if* начинает блок условной трансляции, который компилируется при выполнении заданного в директиве условия (константное целое выражение). Директива *#ifdef* начинает блок условной трансляции, который компилируется, если заданное в директиве имя определено. Директива *#ifndef* начинает блок условной трансляции, который компилируется, если заданное в директиве имя не определено. Директива *#else* начинает блок условной трансляции, компилируемый при невыполнении заданного в директиве *#if* условия. Директива *#endif* завершает блок условной трансляции. Пример использования условной трансляции:

```
#define DEBUG
#define TRACE
...
long password;
```

```

#ifdef DEBUG
#ifdef TRACE
    ShowMessage("Точка 1");
#endif
    password = 1;
#else
    GetPassword(password);
#endif

```

Применение условной трансляции позволяет избежать ошибок, связанных с многократным включением одних и тех же определений из файлов заголовков. Часто возникает ситуация, при которой одни заголовки включаются в другие и при компиляции программы один и тот же файл заголовка может быть включен несколько раз. Для того чтобы заголовок компилировался только один раз, в него включаются директивы условной трансляции. При первом включении компилируется код и определяется некоторое имя, которое исключает последующие трансляции при повторных включениях файлов.

Файл lib.h:

```

#ifndef LIB // Проверка определения имени LIB
#define LIB // Определение имени LIB
...
const float PI = 3.14;
...
#endif // Конец блока условной трансляции

```

Файл form1.h:

```

#include <lib.h> // Включение заголовка lib.h
...

```

Файл form2.h:

```

#include <lib.h> // Включение заголовка lib.h
...

```

Файл prog.cpp:

```

#include <form1.h> // Включение заголовка form1.h
#include <form2.h> // Включение заголовка form2.h
...

```

Оператор расширения символьных строк в макросах **#** позволяет преобразовать передаваемый макросу аргумент в символьную строку:

```

#define message(text)\
    ShowMessage(#text);
...
message(Информация); // Результат: Информация
message("Информация"); // Результат: "Информация"

```

С помощью оператора конкатенации лексем `##` отдельные лексемы «склеиваются» в одну. Оператор `##` и все находящиеся между лексемами пробелы удаляются препроцессором. Пример конкатенации:

```
#define message(var, num) ShowMessage(var##num);  
...  
int code1 = 200;  
int code2 = 210;  
int code3 = 244;  
...  
message(code, 1);  
message(code, 2);  
message(code, 3);
```

С помощью директивы `#line` можно назначить номер строки внутри компилируемого файла:

`#line номер_строки [имя_файла]`

Директива `#error` указывает на необходимость прекращения компиляции и вывода сообщения об ошибке:

`#error текст_сообщения`

Пример использования директив `#error` и `#line`:

```
#line 100  
#ifndef PARAMETER_X  
#error Ошибка компиляции, не задан параметр X!  
#endif;
```

2.12. Стандартная библиотека C

Стандартная библиотека C предоставляет программисту широкий набор функций, предназначенных для выполнения в программах типовых действий. Среди этих функций можно выделить следующие группы:

- управление вводом-выводом (заголовок `stdio.h`);
- работа со строками (заголовок `string.h`);
- математические функции (заголовок `math.h`);
- функции времени (заголовок `time.h`);
- управление выполнением программы (заголовок `stdlib.h`).

Для использования библиотечных функций в программу нужно включить соответствующие заголовки. Детальное описание функций и их параметров приведено в справочниках по языку C [2].

Пример использования в программе математических функций:

```
#include <math.h>

void main()
{
    double a = -23.75;
    double b = fabs(a);           // абсолютное значение (b = 23.75)
    double c = ceil(b);           // округление вверх (c = 24.0)
    double d = fmod(c, 2.5);       // остаток от деления (d = 1.5)
    double e = floor(d);          // округление вниз (e = 1.0)
    double f = exp(e);             // экспонента (f = 2.71...)
    double g = log(f*f);           // натуральный логарифм (g = 2.0)
    double h = pow(g, 10);         // степень (h = 1024.0)
    double i = sqrt(h);           // квадратный корень (i = 32.0)
}
```

Пример использования функций работы со строками:

```
#include <string.h>

void main()
{
    char buf[256];
    strcpy(buf, "Язык C");         // копирование строки
    strcat(buf, "++");             // добавление строки
    char* p1 = strchr(buf, '+');   // поиск символа
    int length1 = strlen(p1);      // длина строки
    char* p2 = strstr(buf, " C");  // поиск подстроки
    int x = strcmp(buf, "Язык PASCAL"); // сравнение строк
    int y = strncmp(buf, "Язык PASCAL", 5); // сравнение символов
}
```

Пример использования функций времени:

```
#include <time.h>

void main()
{
    int t1 = clock();              // время в «тиках» процессора
    ...                            // фрагмент для измерения времени
    int t2 = clock();              // время в «тиках» процессора
    float diff = (float)(t1-t2)/CLK_TCK; // разница в секундах
    long t = time(NULL);           // текущее время
    ShowMessage(ctime(t));         // вывод времени в виде строки
}
```

Часто в программах приходится выполнять преобразование данных из двоичного формата в символьный и наоборот. В библиотеке C имеются функции для выполнения таких действий:

```
sprintf(char* buf, char *format, список преобразуемых данных);
sscanf(char* buf, char *format, список преобразуемых данных);
```

Рассмотрим форматное преобразование для различных типов данных.

Преобразование целых чисел (черный квадрат – ноль-терминатор):

```
char buf[8];
int x = 123;
sprintf(buf, "%d", x);    // в формате целого
sprintf(buf, "%6d", x);   // ширина поля - 6
sprintf(buf, "%06d", x);  // с нулями
sprintf(buf, "%+06d", x); // со знаком
```

1	2	3					
			1	2	3		
0	0	0	1	2	3		
+	0	0	1	2	3		

Форматное преобразование чисел с плавающей точкой:

```
char buf[10];
float x = 123.45;
sprintf(buf, "%g", x);
sprintf(buf, "%.3f", x);
sprintf(buf, "%9.3f", x);
sprintf(buf, "%09.3f", x);
sprintf(buf, "%+09.3f", x);
```

1	2	3	.	4	5				
1	2	3	.	4	5	0			
		1	2	3	.	4	5	0	
0	0	1	2	3	.	4	5	0	
+	0	1	2	3	.	4	5	0	

Форматное преобразование строк:

```
char buf[10];
char* p = "Строка";
sprintf(buf, "%s", p);
sprintf(buf, "%9s", p);
sprintf(buf, "%-8s", p);
sprintf(buf, "%.5s", p);
sprintf(buf, "%5.3s", p);
```

С	т	р	о	к	а				
			С	т	р	о	к	а	
С	т	р	о	к	а				
С	т	р	о	к					
		С	т	р					

Важную роль в написании большинства программ играют функции работы с файлами. Файл – именованный набор данных, хранимый во внешней памяти. Размещением файлов, а также доступом программ к ним управляет операционная система. Перед выполнением операций чтения или записи файл должен быть открыт (связан с файловым дескриптором). После завершения ввода или вывода файл закрывается:

```
FILE* inp = fopen("file.txt", "r"); // открыть файл для чтения

if(inp == NULL)                      // проверка на ошибку
{
    ShowMessage("Ошибка!");
    return;
}

...                                  // работа с данными

fclose(inp);                         // закрыть файл
```

При работе с файлом все операции чтения и записи осуществляются по указателю, который «продвигается» на размер блока читаемых или записываемых данных. Имеется ряд функций, связанных с позиционированием указателя файла:

```
fseek(inp, 10, SEEK_SET);    // указатель на начало файла + 10
fseek(inp, -100, SEEK_END);  // указатель на конец файла - 100
fseek(inp, 20, SEEK_CUR);    // указатель + 20
long offset = ftell(inp);    // текущее положение указателя
rewind(inp);                 // указатель на начало файла
```

Функции `fread` и `fwrite` применяются для чтения и записи данных во внутреннем формате (так, как они представлены в программе):

```
double x[] = {100.25, 102.34, 101.73, 110.18};
FILE* f1 = fopen("Данные.dat", "wb");    // открыть файл
fwrite(x, sizeof(x), 1, f1);             // запись массива x
fclose(f1);                              // закрыть файл
```

Функции `gets` и `puts` применяются для чтения и записи текстовых данных в файлах текстового формата:

```
FILE* f2 = fopen("Текст.txt", "w");      // открыть файл
fputs("Строка 1", f2);                  // запись строки
fputs("Строка 2", f2);                  // запись строки
fputs("Строка 3", f2);                  // запись строки
fclose(f2);                             // закрыть файл
```

Функции `fscanf` и `fprintf` применяются для чтения и записи неоднородных данных (числовых и символьных) в файлах текстового формата:

```
FILE* f3 = fopen("Текст.txt", "w");      // открыть файл
int a = 10;
float b = 15.3;
char* c = "ABCDE";
fprintf(f3, "%3d %6.2f %.3s", a, b, c);  // запись строки
fclose(f3);                             // закрыть файл
```

Функция `fflush` применяется для синхронизации буферов операционной системы и файлов (гарантирует что данные записаны в файл):

```
FILE* f4 = fopen("Текст.txt", "w");      // открыть файл
fwrite(...);                            // запись в файл
fflush(f4);                             // синхронизация
fclose(f4);                             // закрыть файл
```

Более подробная информация о файлах и работе с ними может быть получена из справочного руководства по языку C [2].

2.13. Примеры решения типовых задач программирования на С

При написании программ на языке С часто встречаются задачи работы с массивами, текстовыми строками, файлами данных. Знание приемов решения характерных задач программирования позволяет сэкономить время на написание программ и лучше подойти к пониманию более сложных алгоритмов. Здесь будет рассмотрен ряд типовых примеров.

Пример 1. Имеется целочисленный массив из 1000 элементов. Требуется очистить массив (заполнить нулями).

```
int DATA[1000];                // исходный массив
int N = sizeof(DATA)/sizeof(int); // число элементов

for(int i = 0; i < N; i++)       // цикл по массиву
    DATA[i] = 0;                // запись 0 в массив
```

Пример 2. Имеется целочисленный массив из 1000 элементов. Требуется заполнить массив натуральным рядом чисел (1, 2, 3...).

```
short DATA[1000];              // исходный массив
int N = sizeof(DATA)/sizeof(short); // число элементов

for(int i = 0; i < N; i++)       // цикл по массиву
    DATA[i] = i+1;              // запись в массив
```

Пример 3. Имеется целочисленный массив. Требуется «развернуть» массив (поменять местами элементы – первый с последним, второй с предпоследним, и т. д.). Следует обратить внимание, что здесь число циклов в два раза меньше, чем число элементов массива.

```
long DATA[] = {4, 7, -5, 2, 3, ...}; // исходный массив
int N = sizeof(DATA)/sizeof(long);    // число элементов

for(int i = 0; i < N/2; i++)           // цикл по массиву
{
    long tmp = DATA[i];
    DATA[i] = DATA[N-i-1];
    DATA[N-i-1] = tmp;
}
```

Пример 4. Имеется массив чисел с плавающей точкой. Требуется найти сумму значений элементов массива.

```
float M[] = {5.1, -3.4, -7.0, ...}; // исходный массив
int N = sizeof(M)/sizeof(float);    // число элементов

float S = 0;                         // переменная для суммы
```

```
for(int i = 0; i < N; i++)           // цикл по массиву
    S += M[i];                       // добавление к S
```

Пример 5. Имеется массив чисел с плавающей точкой, содержащий как минимум один элемент. Требуется найти минимальное и максимальное значения элементов массива.

```
double M[] = {8.5, -4.1, 9.3, ...}; // исходный массив
int N = sizeof(M)/sizeof(double);   // число элементов

double Min = M[0];                  // минимальное значение
double Max = M[0];                  // максимальное значение

for(int i = 1; i < N; i++)           // цикл по массиву
{
    if(M[i] < Min) Min = M[i];

    if(M[i] > Max) Max = M[i];
}
```

Пример 6. Имеется массив чисел с плавающей точкой. Требуется найти число положительных, отрицательных и нулевых элементов массива.

```
double M[] = {...};                 // исходный массив
int N = sizeof(M)/sizeof(double);   // число элементов

int N_pos = 0;                      // счетчик положительных
int N_neg = 0;                      // счетчик отрицательных
int N_zer = 0;                      // счетчик нулевых

for(int i = 0; i < N; i++)           // цикл по массиву
{
    if(M[i] < 0)                     // увеличение счетчика
        N_neg++;
    else
    {
        if(M[i] > 0)                 // увеличение счетчика
            N_pos++;
        else
            N_zer++;                 // увеличение счетчика
    }
}
```

Пример 7. Имеется массив чисел с плавающей точкой. Требуется найти число элементов массива, значения которых находятся в диапазоне от -1 до 1 включительно.

```
double M[] = {...};                 // исходный массив
int N = sizeof(M)/sizeof(double);   // число элементов

int COUNT = 0;                      // счетчик

for(int i = 0; i < N; i++)           // цикл по массиву
{
    if(M[i] >= -1 && M[i] <= 1)      // проверка
        COUNT++;                    // увеличение счетчика
}
```

Пример 8. Имеется массив чисел с плавающей точкой. Требуется найти индекс первого элемента, имеющего отрицательное значение. Следует обратить внимание, что если в массиве нет отрицательных значений, то переменная `ind` сохранит значение `-1`.

```
double M[] = {...};           // исходный массив
int N = sizeof(M)/sizeof(double); // число элементов

int ind = -1;                 // искомый индекс

for(int i = 0; i < N; i++)    // цикл по массиву
    if(M[i] < 0)              // проверка
    {
        ind = i;             // присвоение индекса
        break;               // разрыв цикла
    }
```

Пример 9. Имеется массив чисел с плавающей точкой. Требуется заменить элементы, имеющие отрицательное значение, на равные по модулю положительные значения.

```
double M[] = {...};           // исходный массив
int N = sizeof(M)/sizeof(double); // число элементов

for(int i = 0; i < N; i++)    // цикл по массиву
    if(M[i] < 0)              // проверка
        M[i] = -M[i];        // замена
```

Пример 10. Имеется массив чисел с плавающей точкой. Требуется ограничить значения элементов диапазоном от `-10` и до `10`.

```
double M[] = {...};           // исходный массив
int N = sizeof(M)/sizeof(double); // число элементов

for(int i = 0; i < N; i++)    // цикл по массиву
    if(M[i] < -10)            // проверка
        M[i] = -10;          // замена
    else
        if(M[i] > 10)         // проверка
            M[i] = 10;        // замена
```

Пример 11. Имеется целочисленный массив `A`, размером `1000`, содержащий `N` элементов (`N < 1000`). Переписать в массив `B` элементы, имеющие четные значения. Подсчитать число таких элементов в переменной `K`.

```
long A[1000] = {...};        // исходный массив
int N = ...;                  // число элементов
long B[1000];                 // создаваемый массив
int K = 0;                    // счетчик элементов
```

```

for(int i = 0; i < N; i++)           // цикл по массиву
    if(A[i]%2 == 0)                  // проверка на четность
        B[K++] = A[i];              // добавление в массив

```

Пример 12. Написать функцию `fmin`, которая получает в качестве параметров два числа двойной точности и возвращает наименьшее из них.

```

double fmin(double val1, double val2) // функция
{
    if(val1 < val2)                    // сравнение
        return(val1);                 // вернуть val1
    else
        return(val2);                 // вернуть val2
}

```

Второй вариант функции `fmin` использует оператор "?".

```

double fmin(double val1, double val2) // функция
{
    return(val1 < val2 ? val1 : val2); // вернуть выражение
}

```

Пример 13. Написать функцию `fcomp`, которая получает в качестве параметра число двойной точности и возвращает `-1`, если число меньше `0`; `1`, если число больше `0`, и `0`, если число равно `0`.

```

int fcomp(double val)
{
    if(val < 0)                        // сравнение
        return(-1);                  // вернуть -1
    else
        if(val > 0)                   // сравнение
            return(1);                // вернуть 1
        else
            return(0);                // вернуть 0
}

```

Пример 14. Написать функцию `flim`, которая получает в качестве параметров число двойной точности и две границы `lmin` и `lmax`. Функция возвращает число, если оно находится в заданных границах, в противном случае число заменяется соответствующей границей.

```

double flim(double val, double lmin, double lmax)
{
    if(val < lmin)                     // сравнение
        return(lmin);                 // вернуть lmin
    else
        if(val > lmax)                 // сравнение
            return(lmax);              // вернуть lmax
        else

```

```

        return(val);                // вернуть val
    }

```

Пример 15. Написать функцию `forder`, которая получает по ссылке две целочисленные переменные и записывает в первую переменную наименьшее из двух значений, а во вторую – наибольшее.

```

void forder(short &v1, short &v2)
{
    if(v2 < v1)                    // сравнение
    {
        short t = v1;             // обмен (v1 -> t)
        v1 = v2;                  // обмен (v2 -> v1)
        v2 = t;                   // обмен (t -> v2)
    }
}

```

Пример 16. Написать функцию `favg`, которая получает указатель на массив двойной точности и число элементов. Функция возвращает среднее значение элементов массива.

```

double favg(double *M, int N)
{
    int sum = 0;                  // переменная для суммы

    for(int i = 0; i < N; i++)    // цикл по массиву
        sum += M[i];             // добавление

    return(sum/N);               // вернуть среднее
}

```

Пример 17. Написать функцию `findzero`, которая получает указатель на массив двойной точности и число элементов. Функция возвращает индекс последнего нулевого значения или `-1`, если в массиве нет нулевых значений.

```

int findzero(double M[], int N)
{
    int ind = -1;                 // переменная индекса

    for(int i = N-1; i >= 0; i--) // цикл по массиву
        if(!M[i])                // проверка на 0
        {
            ind = i;              // сохранить индекс
            break;                // разорвать цикл
        }

    return(ind);                 // вернуть индекс
}

```


Пример 18. Написать функцию `fstrlen`, которая получает указатель на строку символов. Функция возвращает число символов в строке. Следует обратить внимание, что для продвижения по массиву используется передаваемый через стек указатель на строку.

```
int fstrlen(char* text)
{
    int len = 0;                                // счетчик

    while(*text++)                               // цикл по строке
        len++;                                  // увеличить счетчик

    return(len);                                // вернуть счетчик
}
```

Пример 19. Написать функцию `fspace`, которая получает указатель на строку символов. Функция возвращает число пробелов в строке.

```
int fspace(char* text)
{
    int len = 0;                                // счетчик

    while(*text)                                // цикл по строке
        if(*text++ == ' ')                     // сравнение с пробелом
            len++;                              // увеличить счетчик

    return(len);                                // вернуть счетчик
}
```

Пример 20. Задан массив структур, описывающий точки на плоскости. Написать функцию `fpoints`, которая получает указатель на массив, число элементов массива и радиус некоторой окружности. Функция возвращает число точек, попадающих в пределы заданной окружности.

```
typedef struct { float x; float y; } POINT;
POINT points[] = {{25.1, -13.2}, {134.7, 29.1}, ...};
int N = sizeof(points)/sizeof(POINT);
...
int fpoints(POINT p[], int N, float R)
{
    int count = 0;                                // счетчик

    for(int i = 0; i < N; i++)                    // цикл по массиву
        if(sqrt(p[i].x*p[i].x+p[i].y*p[i].y) <= R)
            count++;                              // увеличить счетчик

    return(count);                                // вернуть счетчик
}
```

Глава 3. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

3.1. Предпосылки структурного программирования

Раньше хорошими программистами считали тех, кто писал весьма хитроумные программы, которые занимали минимум оперативной памяти и выполнялись за кратчайшее время. Это было естественно, потому что в «старое доброе время» размер оперативной памяти был сильно ограничен, а машины были намного медленнее, чем сейчас. Результатом хитроумного кодирования оказывались программы, которые было трудно понять другим лицам. Программисты зачастую сами признавали, что свою собственную программу они с трудом понимают уже через полгода, а то и через месяц [3]. Многократные изменения приводили к тому, что в программах появлялись новые фрагменты, логически не связанные друг с другом; многочисленные переходы между фрагментами кода привели к появлению термина «блюдо спагетти». Это означает, что попытка изменить или удалить фрагмент программы требует пересмотра программы в целом из-за наличия большого числа запутанных связей (рис. 3.1).

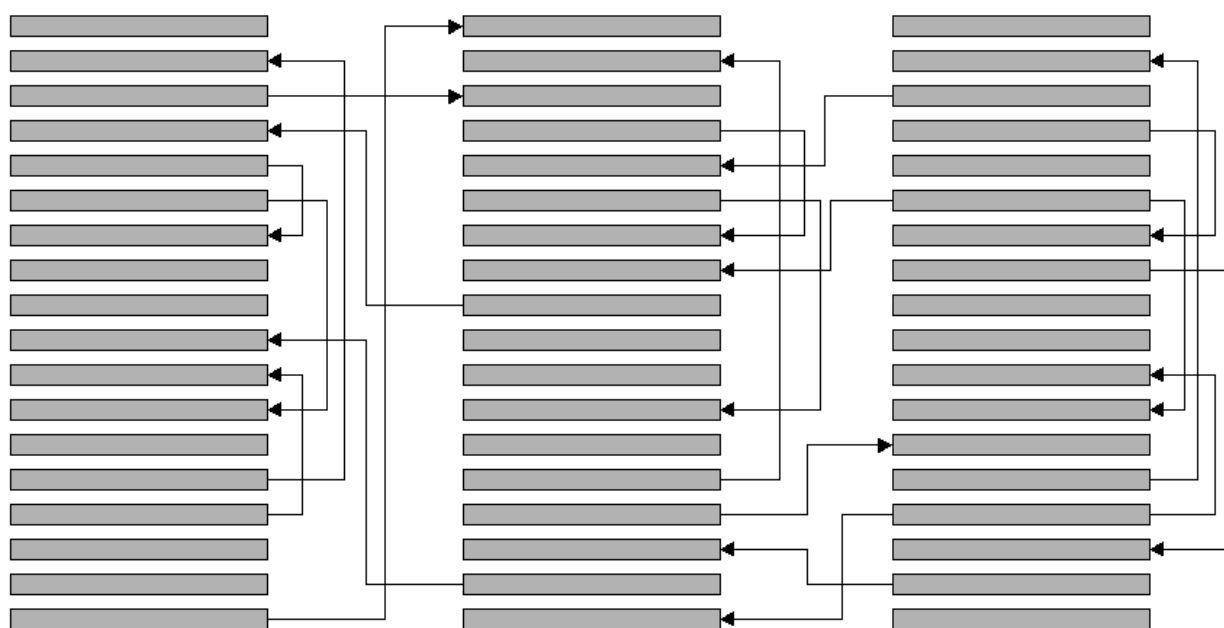


Рис. 3.1

Возникающие проблемы толкали к поиску новых подходов и технологий программирования. В 1968 г. Эдсгер Дейкстра писал: «На протяжении многих лет я очень хорошо знал, что квалификация программистов – убывающая функция от плотности операторов GOTO в создаваемых ими програм-

мах. Но лишь совсем недавно я обнаружил, почему использование оператора GOTO имеет такие губительные последствия. Я пришел к убеждению, что этот оператор должен быть исключен из всех языков программирования высокого уровня».

3.2. Структурный подход к программированию

Цель структурного подхода к программированию – разработка понятных, правильных, легко сопровождаемых программ. Основу структурного подхода составляют три элемента:

- нисходящая разработка;
- пошаговая детализация;
- сквозной структурный контроль.

Нисходящая разработка организует проектирование программ сверху вниз с использованием принципа модульности. Этот принцип заключается в разбиении функциональности сложного программного проекта на более простые части – *модули*. Реализация модулей начинается с верхнего уровня, при этом на ранних стадиях проекта часть модулей заменяется программными имитаторами – *заглушками* (рис. 3.2). Это позволяет с самого начала видеть функциональность проекта в целом, отлаживать и демонстрировать готовые модули, не дожидаясь завершения остальных.

При разбиении программы на модули следует учитывать ряд правил:

1. Модуль может быть отдельной программой или подпрограммой (функцией).
2. На модуль можно ссылаться с помощью имени, называемого именем модуля.
3. Модуль должен возвращать управление тому модулю, который его вызвал.
4. Модуль может обращаться к другим модулям.
5. Модуль должен иметь один вход и один выход.
6. Модуль должен быть сравнительно небольшим (до 200 строк кода).
7. Модуль не должен зависеть от истории своих вызовов.
8. В идеале модуль должен реализовывать одну функцию, причем целиком.

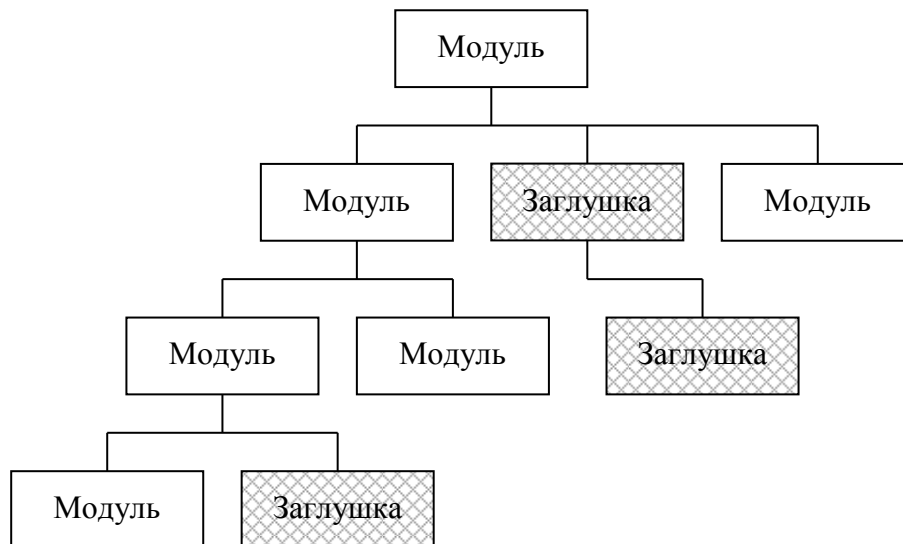


Рис. 3.2

Пошаговая детализация применяется для реализации функциональности выделенных модулей. При этом последовательность действий формируется сначала укрупнено, затем более детально, потом еще более детально, пока получившиеся действия не станут легко и понятно реализуемы на выбранном языке программирования. Пример пошаговой детализации модуля «Загрузка данных» приведен на рис. 3.3.

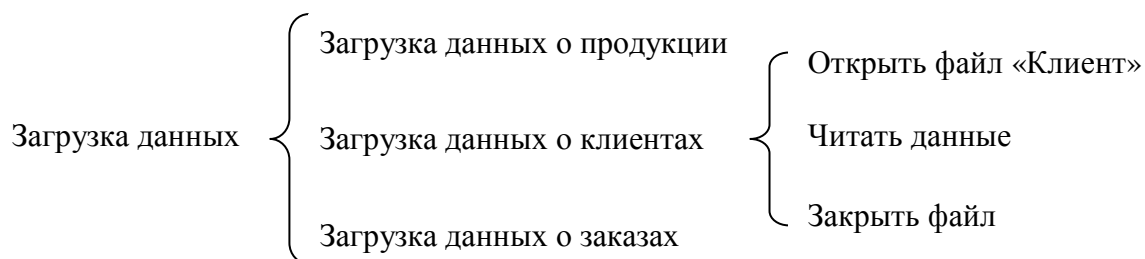


Рис. 3.3

При программной реализации модулей применяются правила структурного программирования, заключающиеся в том, что логическая структура любой программы может быть выражена комбинацией трех базовых структур – следования, развилки и цикла (рис. 3.4). *Следование* означает последовательное выполнение модулей или частей кода. *Развилка* в зависимости от некоторых условий направляет программный поток по разным ветвям. *Цикл* означает повторение определенного фрагмента программы до выполнения некоторого условия.

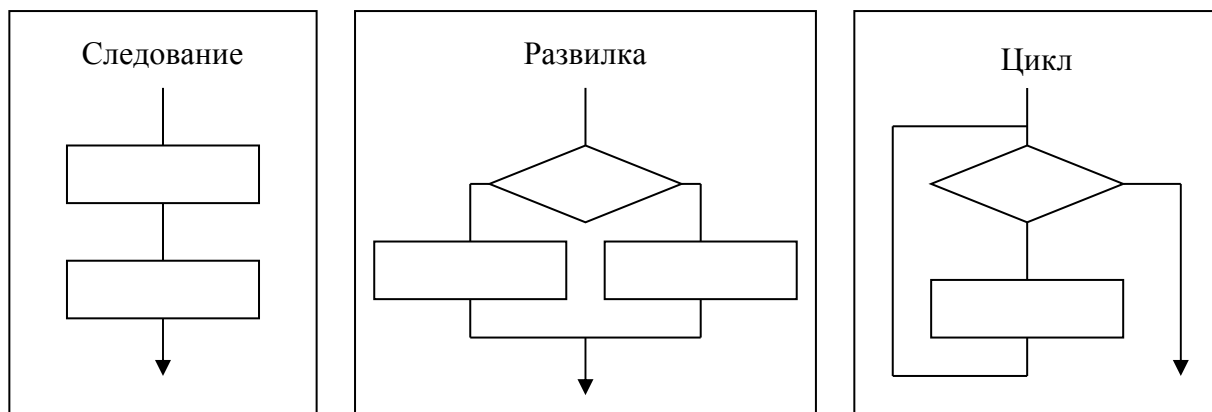


Рис. 3.4

Структуры могут быть вложены одна в другую в произвольном порядке. При этом сохраняется главный принцип модульности – любой фрагмент программы имеет только один вход и один выход (рис. 3.5). Таким образом структурное программирование исключает использование оператора безусловного перехода GOTO, который вызывал столько нареканий.

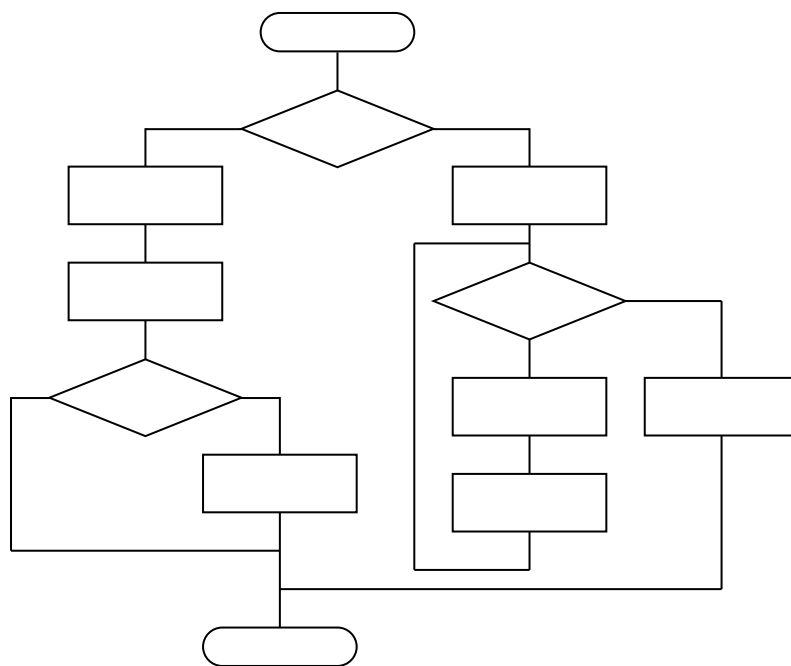


Рис. 3.5

Структурные программы могут быть наглядно представлены в виде *псевдокода* – произвольного текста, включающего ключевые слова для выделения конструкций развилки от ЕСЛИ ТО ИНАЧЕ до ВСЁ ЕСЛИ и цикла от ЦИКЛ ПОКА до ВСЁ ЦИКЛ. Вложенные конструкции записывают со сдвигом текста вправо. Пример псевдокода для функции «Загрузка данных о клиентах» приведен далее:

```
Открыть файл "Клиенты"
ЕСЛИ успешно
ТО
    Сбросить счетчик элементов массива "Клиенты"
    ЦИКЛ ПОКА не встречен конец файла
        Читать очередную запись в массив
        Увеличить счетчик на 1
    ВСЁ ЦИКЛ
    Заккрыть файл "Клиенты"
ИНАЧЕ
    Сообщение об ошибке "Файл не найден"
ВСЁ ЕСЛИ
```

Сквозной структурный контроль заключается в обнаружении и исправлении ошибок на ранних стадиях проекта, пока стоимость исправления минимальна, а последствия наименее значительны. Смысл структурного контроля состоит в том, что сложная программа, состоящая из правильно работающих модулей, будет работать правильно. Поэтому *верификация* (проверка) осуществляется на всех этапах проекта. При этом применяются две техники: тестирование и анализ кода.

Тестирование заключается в проведении запланированных действий по проверке работоспособности программы при заданном наборе данных. Для этого подготавливается набор тестовых случаев (test cases), которые должны покрывать все выполняемые программой функции. Результаты работы программы сравниваются с ожидаемыми и делается вывод о наличии или отсутствии ошибок. Сложность тестирования состоит в том, что как правило невозможно проверить все комбинации входных данных, поэтому тестирование не гарантирует отсутствия ошибок в программе.

Анализ кода (peer review) проводится программистами по принципу «проверь коллегу». Эта техника оказывается достаточно эффективной, поскольку сторонний взгляд часто находит ошибки, не замечаемые автором кода. При проверке кода много внимания уделяется правильному оформлению программы, применению наработанных приемов программирования, исключению «рискованных» фрагментов кода. Постепенно в каждой организации, специализирующейся в области разработки программного обеспечения, накапливается своя «база знаний», где отражаются наилучшие методы создания программного кода.

3.3. Разработка программ в среде C++Builder

Как было отмечено в гл. 1, разработка программ на языке высокого уровня включает ряд этапов (рис. 1.1). Для повышения эффективности работы программистов создаются специальные среды, в которых присутствуют инструменты для поддержки всех стадий разработки программного обеспечения, начиная с концептуального дизайна и заканчивая отладкой. Особое значение в настоящее время также имеет поддержка структурного программирования и коллективной работы над проектами.

Одной из сред, обладающих всей необходимой функциональностью, является Embarcadero C++Builder (ранее она называлась Borland C++Builder). В C++Builder интегрировано множество инструментов разработчика, из которых наиболее важны следующие:

- менеджер проектов;
- дизайнер форм;
- библиотека компонентов;
- редактор кода;
- компилятор и построитель;
- отладчик.

Менеджер проектов позволяет организовывать работу с файлами проектов, объединяя их в проектные группы, что позволяет структурировать разработку сложных программных комплексов и иметь под рукой логически связанные проекты. Результатом проекта всегда является одна программная единица – это может быть интерактивное или консольное приложение, объектная или разделяемая библиотека, сервис. Имеется также ряд других целевых назначений проектов (рис. 3.6).

Дизайнер форм позволяет наглядно создавать пользовательский интерфейс для будущих приложений, используя компоненты из библиотеки VCL (Visual Component Library). Компоненты интерфейса (поля, метки, кнопки, панели, закладки и т. д.) помещаются на разрабатываемые формы. Их местоположением, видом и функциональностью можно управлять с помощью редактора объектов (рис. 3.7).

Разработанные формы сохраняются в виде текстовых файлов с расширением .dfm, в которых описывается иерархия объектов формы и их свойства. Эти файлы можно просматривать и по ним осуществлять поиск.

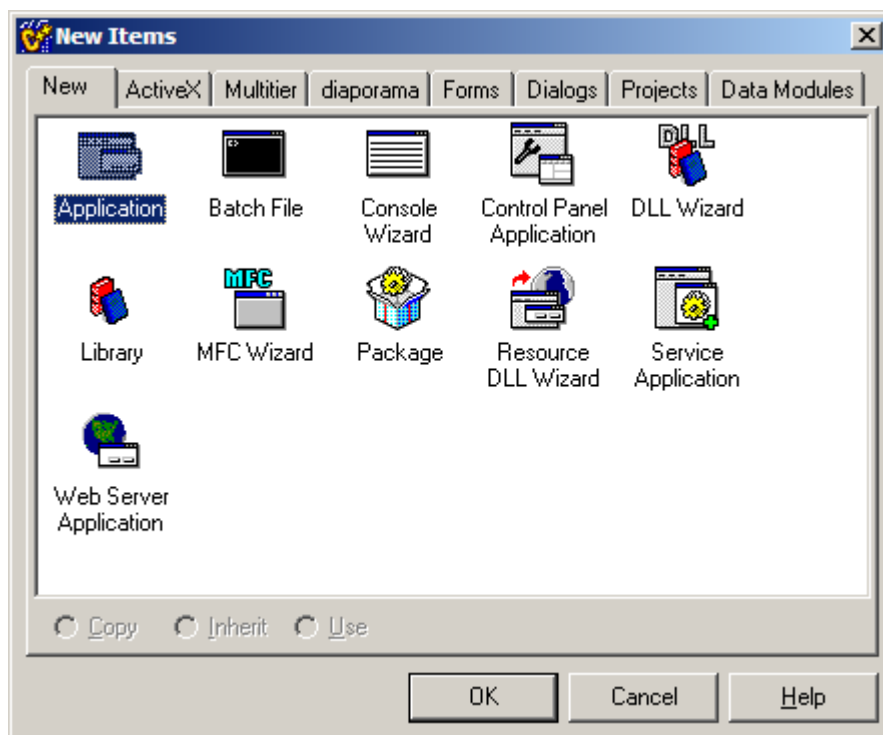


Рис. 3.6

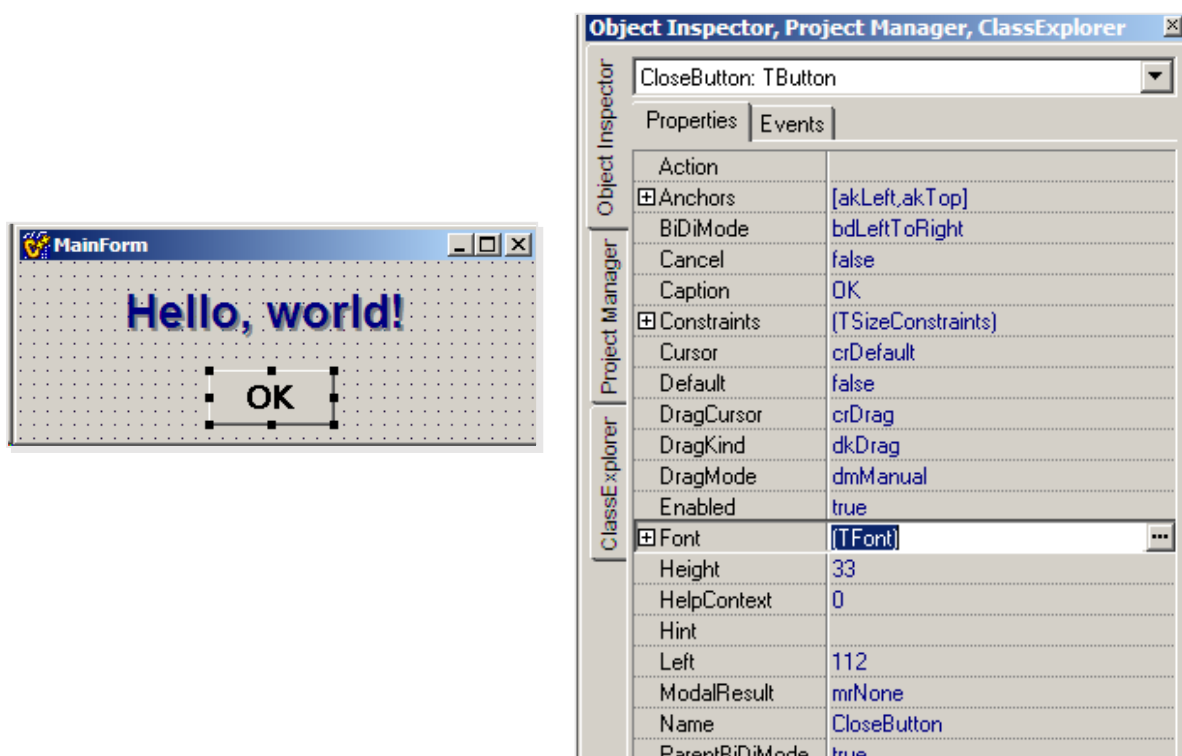


Рис. 3.7

Библиотека компонентов предоставляет разработчикам широкий набор модулей, структурированный по разделам: пользовательский интерфейс, работа с базой данных, клиент-серверное взаимодействие, веб и т. д.

Редактор кода представляет собой мощный синтаксически ориентированный редактор с возможностью цветового выделения элементов языка, автоматизированного форматирования текста программы, построения основных конструкций (условия, циклы), отслеживания парности скобок и т. п.

Компилятор и построитель в автоматическом режиме обеспечивают трансляцию и сборку модулей программного проекта, а также пользовательских и системных библиотек. В составе среды имеются компиляторы для различных платформ (Windows, OS X, Android). Это позволяет из одного исходного кода создавать кроссплатформенные приложения. Также имеется возможность отдельно видеть результат работы препроцессора и создавать на выходе компилятора файл на языке ассемблера.

Отладчик позволяет трассировать выполнение программы, задавая точки останова. В этих точках возможен контроль памяти и регистров процессора, просмотр переменных и объектов программы.

Для интерактивных программ часть кода генерируется средой автоматически. Например, далее показан файл заголовка, сгенерированный для формы на рис. 3.7:

```
#ifndef WindowsAppFormH
#define WindowsAppFormH

#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>

class TMainForm : public TForm
{
__published:      // IDE-managed Components
    TLabel         *Label1;
    TLabel         *Label2;
    TButton        *CloseButton;
    void __fastcall ButtonOKClick(TObject *Sender);
private:          // User declarations
public:           // User declarations
    __fastcall TMainForm(TComponent* Owner);
};

extern PACKAGE TMainForm *MainForm;

#endif
```

В этом файле для формы создается класс (см. гл. 4), содержащий указатели на объекты, соответствующие размещенным на форме компонентам.

Для реакции приложения на внешние события (например, на нажатие кнопки) автоматически декларируются функции-обработчики. Это происходит при задании событий в редакторе объектов. Основной файл модуля также создается автоматически и имеет следующий вид:

```
#include <vcl.h>
#pragma hdrstop

#include "WindowsAppForm.h"

#pragma resource "*.dfm"

TMainForm *MainForm;

__fastcall TMainForm::TMainForm(TComponent* Owner): TForm(Owner)
{
}

void __fastcall TMainForm::ButtonOKClick(TObject *Sender)
{
}
```

В данном файле разработчик должен вписать код, выполняющий желаемые действия, в тела функций. Также генерируется головной модуль программы, пример которого приведен ниже. В этом файле инициализируется взаимодействие приложения с библиотекой компонентов, создается форма и запускается цикл обработки сообщений:

```
#include <vcl.h>
#pragma hdrstop

USEFORM("WindowsAppForm.cpp", MainForm);

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TMainForm), &MainForm);
        Application->Run();
    }
    catch(Exception &exception)
    {
        Application->ShowException(&exception);
    }

    return(0);
}
```

Глава 4. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

4.1. Объектно-ориентированное программирование и язык C++

Хотя структурный подход вывел разработку программного обеспечения на новый качественный уровень, все возрастающие потребности в производстве программ, а также сокращение длительности цикла разработки потребовали поиска новых идей в области программирования. В 1980-е гг. был предложен объектно-ориентированный подход к программированию, который до настоящего времени активно используется при создании программного обеспечения.

Цель объектно-ориентированного программирования (ООП) заключается в повышении скорости разработки и качества программ за счет лучшей структуризации и повторного использования кода (code reusing). В основе концепции ООП лежат три основных положения:

- *инкапсуляция* – объединение данных и методов их обработки в виде классов с ограничением доступа к данным и методам для различных категорий пользователей;
- *наследование* – создание новых классов на основе существующих с дополнением или изменением их функциональности;
- *полиморфизм* – выполнение разных действий одноименными методами различных классов.

С момента появления ООП разработано значительное число языков программирования, поддерживающих этот подход. Вот далеко не полный список объектно-ориентированных языков: C#, C++, Java, Eiffel, Simula, D, Io, Objective-C, Object Pascal, VB.NET, Visual DataFlex, Perl, Php, PowerBuilder, Python, Scala, ActionScript, JavaScript, JScript.NET, Ruby, Smalltalk, Ada, Xbase++, X++, Vala.

Один из первых языков ООП C++ (си плюс плюс) был разработан в 1985 г. Бьерном Страуструпом как расширение языка C [4]. Одним из основных принципов, лежащих в основе C++, является его практически полная совместимость с C, которая существенно облегчила переход программистов, работавших на языке C, к объектно-ориентированному программированию и гарантировала применимость разработанных на C программ. За более чем 25-летнюю историю C++ завоевал большую популярность и в настоящее время

является одним из основных языков программирования. Поэтому рассмотрение основных аспектов ООП будет сопровождаться примерами на языке C++.

4.2. Инкапсуляция. Классы

Основным понятием ООП является понятие *класса*. Классы служат для инкапсуляции – объединения данных и методов работы с ними в новые типы данных. Классы могут также предоставлять различные права доступа к своим данным и методам. Возможно создание иерархии классов посредством наследования.

В объявлении класса в C++ могут присутствовать как переменные, так и функции. Переменные называют *атрибутами* (другое название – переменные – члены класса), функции – *методами* (другое название – функции – члены класса):

```
class Rectangle      // класс «Прямоугольник»
{
    int width;        // атрибут «Ширина»
    int height;       // атрибут «Высота»
    int Area();       // метод «Площадь»
};
```

Понятие класса в C++ аналогично понятию структуры в C, поэтому в программах на C++ можно включать методы в такие элементы языка, как структуры и объединения.

Для атрибутов и методов может применяться ограничение видимости. Все определения в классе относятся к одной из трех областей – закрытой, защищенной или открытой, задаваемых ключевыми словами *private*, *protected* или *public* соответственно.

Атрибуты и методы, определенные в закрытой области, могут использоваться только методами данного класса или дружественными функциями. Атрибуты и методы, определенные в защищенной области, могут использоваться методами данного и наследуемых от него классов. Атрибуты и методы, определенные в открытой области, могут использоваться любыми функциями программы. Если область видимости не задана, то для классов по умолчанию устанавливается *private*, а для структур и объединений – *public*. Пример класса с различными областями видимости:

```
class Example
{
    int a;              // закрытые данные
```

```

    int b;           // закрытые данные
protected:
    int c;           // защищенные данные
public:
    int d;           // открытые данные
private:
    int e;           // закрытые данные
public:
    int f;           // открытые данные
};

```

Открытую область класса часто называют его *интерфейсом*, поскольку именно этими данными и методами пользуются прикладные программисты, применяя разработанные другими классы.

Реализация методов класса может осуществляться двумя способами: в рамках класса или вне класса. При первом способе реализация метода непосредственно следует его декларации:

```

class Rectangle
{
private:
    int    width;
    int    height;
public:
    int    Area()
    {
        return(width*height);
    }
};

```

Методы, реализованные в рамках класса, компилятор воспринимает как *инлайновые* (inline). Это означает, что код метода будет генерироваться каждый раз в точке вызова метода (как в случае макроса). Обычно такую реализацию делают для коротких методов либо если нужно сократить затраты времени на вызов метода.

При реализации вне класса декларация и реализация могут осуществляться в разных местах программы, даже в разных модулях. Для компилятора важно только, чтобы декларация предшествовала реализации. Для указания, к какому классу относится метод, применяется оператор расширения области видимости, который представляет собой имя класса с последующими двумя двоеточиями:

```

class Rectangle
{
private:

```

```

        int    width;
        int    height;
public:
        int    Area();           // декларация метода
};

int Rectangle::Area()           // реализация метода
{
    return(width*height);
}

```

Если класс в С++ является аналогом сложного типа данных в С, то аналогом переменной в С++ служит *объект*. Компилятор С++ не разделяет переменные и объекты, поэтому выделение памяти под объекты классов производится аналогично работе с переменными.

Для доступа к элементам класса могут использоваться оператор «.» или указатели. Пример двух способов работы с атрибутами и методами:

```

class Rectangle
{
public:
    int    width;
    int    height;
    int    Area();           // декларация метода
};

Rectangle rect;              // создание объекта

rect.width  = 200;
rect.height = 100;

int S1 = rect.Area();

Rectangle *r = &rect;        // указатель на объект

r->width  = 400;
r->height = 150;

int S2 = r->Area();

```

Следует отметить, что в рассмотренном примере данные класса находились в открытой области (public). Если же их переместить в закрытую часть, то при компиляции программы будет выдана ошибка:

```

class Rectangle
{
private:
    int    width;

```

```

    int    height;
public:
    int    Area();
};

void main()
{
    Rectangle rect;
    rect.width  = 100;
    rect.height = 50;
}

```

```

[C++ Error] test.cpp: 'Rectangle::width' is not accessible
[C++ Error] test.cpp: 'Rectangle::height' is not accessible

```

Поскольку по соображениям инкапсуляции данные классов размещают в закрытой области, то для их инициализации используют специальный метод – *конструктор*. Конструктор вызывается автоматически при создании объекта. Для конструкторов установлен ряд правил:

1. Имя конструктора совпадает с именем класса.
2. Конструктор не возвращает никакого значения.
3. Для класса без конструктора генерируется конструктор по умолчанию.
4. Конструкторы могут быть перегружены.
5. Конструкторы не наследуются.
6. Конструкторы не могут вызываться явно из программы.

Далее приведен пример декларации, реализации и использования конструкторов при создании объектов:

```

class Rectangle
{
private:
    int    width;
    int    height;
public:
    Rectangle(int Width, int Height); // декларация конструктора
};

Rectangle::Rectangle(int Width, int Height) // реализация
{
    width  = Width;
    height = Height;
}

Rectangle rect(200, 100); // использование конструктора

```

Перегрузка конструктора позволяет одному классу иметь несколько конструкторов, отличающихся числом или типами параметров. Более подробно перегрузка методов будет рассмотрена в 4.5. Для инициализации объекта через другой объект того же класса применяется конструктор копирования, единственным параметром которого должен быть передаваемый по ссылке объект класса. Далее приводится пример перегрузки конструкторов для класса `Rectangle`:

```
class Rectangle
{
private:
    int    width;
    int    height;
public:
    Rectangle(int Width, int Height);    // конструктор 1
    Rectangle(int Side);                // конструктор 2
    Rectangle(Rectangle& R);            // конструктор 3
};

Rectangle::Rectangle(int Side)          // конструктор 2
{
    width  = Side;
    height = Side;
}

Rectangle::Rectangle(Rectangle& R)      // конструктор 3
{
    width  = R.width;
    height = R.height;
}

Rectangle rect1(200, 100); // использование конструктора 1
Rectangle rect2(180);      // использование конструктора 2
Rectangle rect3 = rect1;   // использование конструктора 3
```

Для динамического создания объектов применяется оператор `new`:

```
Rectangle *r1 = new Rectangle(200, 100); // конструктор 1
Rectangle *r2 = new Rectangle(150);      // конструктор 2
Rectangle *r3 = new Rectangle(*r1);      // конструктор 3
```

Для разрушения динамически созданных объектов применяется оператор `delete`:

```
delete r1;
delete r2;
delete r3;
```


Для выполнения каких-либо действий при разрушении объектов применяется специальный метод – *деструктор*. Вызов деструктора осуществляется автоматически. Для деструкторов установлен ряд правил:

1. Имя деструктора состоит из тильды (~) и имени класса.
2. Деструктор не возвращает никакого значения.
3. Для класса без деструктора генерируется деструктор по умолчанию.
4. Деструкторы не наследуются.
5. Деструкторам не могут передаваться аргументы.
6. Деструкторы могут описываться как виртуальные (virtual).
7. Деструкторы могут вызываться явно.

Пример декларации и реализации деструктора:

```
class Rectangle
{
    ...
public:
    Rectangle(int Width, int Height);
    ~Rectangle();
};

Rectangle::~~Rectangle()
{
    ShowMessage("Прямоугольник удален.");
}
```

Атрибуты и методы класса могут быть объявлены как *статические*. Статические атрибуты – общие (имеют одинаковые значения) для всех объектов данного класса. Память под статические атрибуты выделяется не в объектах класса, а отдельно. Статические методы могут вызываться без объектов. Соответственно, они не могут работать с данными объектов, за исключением статических. Пример статических данных и методов:

```
class Rectangle
{
private:
    static int count; // статический атрибут
public:
    Rectangle(int Width, int Height);
    ~Rectangle();

    static int Count() { return(count); } // статический метод
};

int Rectangle::count = 0; // выделение памяти и инициализация
```

```

Rectangle::Rectangle(int Width, int Height)
{
    int w = Width;
    int h = Height;
    count++;          // увеличение счетчика в конструкторе
}

Rectangle::~~Rectangle()
{
    count--;          // уменьшение счетчика в деструкторе
}

void main()
{
    int c0 = Rectangle::Count();    // c0 равно 0
    Rectangle *r1 = new Rectangle(200, 100);
    Rectangle *r2 = new Rectangle (150, 120);
    int c1 = Rectangle::Count();    // c1 равно 2
    delete r1;
    int c2 = Rectangle::Count();    // c2 равно 1
    delete r2;
    int c3 = Rectangle::Count();    // c3 равно 0
}

```

Компилятор предоставляет каждому объекту указатель на него самого – *this*. Этот указатель может быть использован только внутри методов класса для получения адреса текущего объекта. Указатель *this* часто используется, когда нужно вернуть адрес текущего объекта:

```

class Rectangle
{
private:
    int    width;
    int    height;
public:
    Rectangle& Zoom(float Rate);    // метод возвращает объект
    int    Area();
};

Rectangle& Rectangle::Zoom(float Rate)
{
    width  = (float)width*Rate;
    height = (float)height*Rate;
    return(*this);                // возврат адреса объекта
}

Rectangle r(300, 200);
int S = r.Zoom(0.25).Area();      // использование метода Zoom

```

4.3. Наследование

Понятие наследования в ООП тесно связано с понятием абстракции. Абстракция в ООП – это выделение существенных характеристик объектов, которые отличают их от всех других объектов, четко определяя концептуальные границы каждого класса. Противоположностью абстракции является конкретизация, которая уточняет наборы свойств различных классов и позволяет создавать их иерархии. Пример фрагмента такой иерархии показан на рис. 4.3.

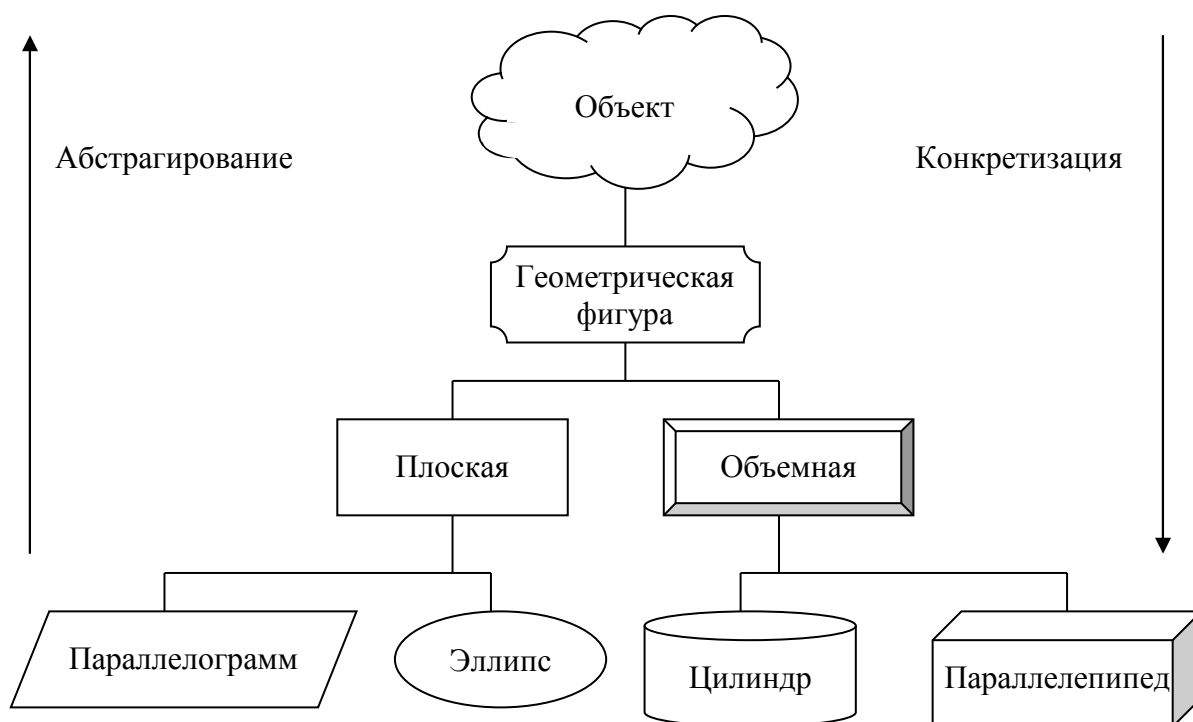


Рис. 4.3

Наследование – важная составляющая концепции объектно-ориентированного программирования, благодаря которой возможно создание новых классов на базе уже существующих. При этом новые классы (производные) получают все элементы (атрибуты и методы) наследуемых классов (базовых).

В ООП разделяют простое и множественное наследования. При *простом наследовании* производный класс имеет только один базовый класс. Базовый класс может иметь несколько производных классов. Иерархия классов имеет вид дерева. При *множественном наследовании* производный класс имеет несколько родительских классов. Иерархия классов имеет вид ориентированного графа. В обоих случаях имеются рекомендации создания «хорошей» иерархии классов:

1. *Критерий включения.* Объекты производных классов должны обладать всеми свойствами базовых классов.

2. *Критерий независимости.* Наследуемые свойства не должны ограничивать использование собственных элементов производных классов.

3. *Критерий специализации.* Если производный класс приводит к специализации (ограничению) свойств базового класса, то это является нарушением принципа независимости. Классы, которые отличаются только специализацией, обычно составляют один уровень иерархии.

4. *Критерий единства.* Наследование и полиморфизм обеспечивают единообразное выполнение аналогичных функций классов. Базовые классы используются для создания интерфейса между классами и «внешним миром».

В языке C++ поддерживается как простое, так и множественное наследование. Для этого при декларации класса после его имени через двоеточие указывается список наследуемых классов:

```
class A
{
private:
    ...
};

class B
{
private:
    ...
};

class C : public A, public B
{
private:
    ...
};
```

В списке наследуемых классов кроме имен задаются ограничения доступа при наследовании. С помощью ключевых слов `public`, `protected` и `private` возможно изменение области видимости атрибутов наследуемых классов. При этом область видимости может только оставаться неизменной или сокращаться, но никак не расширяться. Правила изменения области видимости указаны в табл. 4.1.

Таблица 4.1

Атрибут доступа, указанный при наследовании	Атрибут доступа базового класса	Атрибут доступа производного класса
public	public	public
	protected	protected
	private	private
protected	public	protected
	protected	protected
	private	private
private	public	private
	protected	private
	private	private

В отличие от других методов конструкторы и деструкторы не наследуются. При создании объекта производного класса наследуемые им данные должны инициализироваться конструктором базового класса. Если наследуются несколько базовых классов, то их конструкторы вызываются явно конструктором производного класса или компилятор иницииирует вызов конструкторов по умолчанию в порядке описания базовых классов. Конструкторы производных классов могут передавать аргументы конструкторам базовых классов. Пример наследования и использования конструкторов показан далее на для базового класса Figure и производных классов Rectangle и Circle:

```
class Figure                                     // класс «Фигура»
{
protected:
    TColor color;                               // атрибут «Цвет»
public:
    Figure(TColor Color);                       // конструктор класса
    TColor Color() { return(color); }
};

Figure::Figure(TColor Color)
{
    color = Color;
}

class Rectangle : public Figure                 // класс «Прямоугольник»
{
private:
    float width;
    float height;
```

```

public:
    Rectangle(float Width, float Height, TColor Color);
};

Rectangle::Rectangle(float Width, float Height, TColor Color) :
    Figure(Color) // вызов конструктора базового класса
{
    width  = Width;
    height = Height;
}

class Circle : public Figure          // класс «Круг»
{
private:
    float radius;
public:
    Circle(float Radius, TColor Color);
};

Circle::Circle(float Radius, TColor Color) :
    Figure(Color) // вызов конструктора базового класса
{
    radius = Radius;
}

```

4.4. Полиморфизм

Понятие полиморфизма тесно связано с концепцией наследования. Оно означает феномен различного выполнения одноименных функций в наследуемых классах. В языке C++ полиморфизм в первую очередь проявляется в *переопределении* методов. Если в производном классе появляется метод с декларацией, идентичной используемой для метода базового класса (имя, число и типы параметров), то для объектов производного класса происходит «замещение» функциональности метода базового класса вновь определенным методом. Рассмотрим пример переопределения метода Area класса Figure в производных классах Rectangle и Circle:

```

class Figure                                // класс «Фигура»
{
    ...
public:
    float Area() { return(-1); }           // метод «Площадь»
};

class Rectangle : public Figure             // класс «Прямоугольник»
{

```

```

public:
    float Area() { return(width*height); }
};

class Circle : public Figure          // класс «Круг»
{
    ...
public:
    float Area() { return(M_PI*radius*radius); }
};

```

При переопределении методов применяются два вида связывания – раннее и позднее.

Раннее связывание осуществляется на этапе компиляции. Оно заключается в том, что компилятор выбирает метод по типу объекта или указателя на объект, с помощью которых вызывается метод. В рассмотренном примере применялось раннее связывание. В следующем примере показывается особенность раннего связывания – если метод вызывается через указатель, то вызываемый метод выбирается по типу указателя, а не по типу объекта. В этой ситуации для точной спецификации метода можно использовать оператор расширения видимости:

```

Figure    f;                // объект класса Figure
Rectangle r(200, 100);      // объект класса Rectangle
Circle    c(150);          // объект класса Circle
Rectangle *rp0 = &r;        // указатель на класс Rectangle
Circle    *cp0 = &c;        // указатель на класс Circle
Figure    *fp = &f;         // указатель на класс Figure
Figure    *rp = &r;         // указатель на класс Figure
Figure    *cp = &c;         // указатель на класс Figure
float      S;

S = f.Area();               // метод класса Figure
S = r.Area();               // метод класса Rectangle
S = c.Area();               // метод класса Circle
S = r.Figure::Area();       // метод класса Figure
S = c.Figure::Area();       // метод класса Figure
S = rp0->Area();             // метод класса Rectangle
S = cp0->Area();             // метод класса Circle
S = fp->Area();              // метод класса Figure
S = rp->Area();              // метод класса Figure!!!
S = cp->Area();              // метод класса Figure!!!

```

Позднее связывание в отличие от раннего осуществляется в процессе выполнения программы. В декларации методов для позднего связывания указывается ключевое слово *virtual* и такие методы называются *виртуальными*.

При позднем связывании компилятор размещает в объекте указатель на таблицу виртуальных функций (методов) для данного класса. Вызов метода идет через указатель и таблицу, что создает дополнительные затраты времени и памяти, но предоставляет программисту возможность не задумываться о типе объекта, для которого будет вызываться метод. В следующем примере показан механизм позднего связывания:

```
class Figure                                     // класс «Фигура»
{
...
public:
    virtual float Area() { return(-1); } // метод «Площадь»
};

class Rectangle : public Figure                 // класс «Прямоугольник»
{
...
public:
    virtual float Area() { return(width*height); }
};

class Circle : public Figure                   // класс «Круг»
{
...
public:
    virtual float Area() { return(M_PI*radius*radius); }
};

Figure    f;                                // объект класса Figure
Rectangle r(200, 100);                       // объект класса Rectangle
Circle    c(150);                           // объект класса Circle
Rectangle *rp0 = &r;                         // указатель на класс Rectangle
Circle    *cp0 = &c;                         // указатель на класс Circle
Figure    *fp = &f;                          // указатель на класс Figure
Figure    *rp = &r;                          // указатель на класс Figure
Figure    *cp = &c;                          // указатель на класс Figure
float      S;

S = f.Area();                                // метод класса Figure
S = r.Area();                                // метод класса Rectangle
S = c.Area();                                // метод класса Circle
S = r.Figure::Area();                        // метод класса Figure
S = c.Figure::Area();                        // метод класса Figure
S = rp0->Area();                             // метод класса Rectangle
S = cp0->Area();                             // метод класса Circle
S = fp->Area();                              // метод класса Figure
S = rp->Area();                              // метод класса Rectangle!!!
S = cp->Area();                              // метод класса Circle!!!
```


Метод может быть объявлен как *чисто виртуальный*. Это означает, что он не имеет реализации (программного кода). Чисто виртуальные методы предназначены для задания интерфейса переопределяемых методов в иерархии классов.

Классы, которые включают хотя бы один чисто виртуальный метод, называются *абстрактными*. Для абстрактных классов не могут быть созданы объекты. Абстрактные классы всегда находятся на вершине иерархии классов и предназначены для придания иерархии единообразия. В следующем примере метод Area объявлен как чисто виртуальный, что делает класс Figure абстрактным:

```
class Figure                                // абстрактный класс «Фигура»
{
    ...
public:
    virtual float Area() = 0;  // чисто виртуальный метод
};
```

При множественном наследовании могут возникать конфликты, связанные с тем, что в наследуемых классах могут присутствовать атрибуты и методы с одинаковыми именами. При использовании таких атрибутов или методов в производных классах возникает двусмысленность и компилятор выдает ошибку. Для устранения конфликтов имен при множественном наследовании должен использоваться оператор расширения области видимости, как это показано в следующем примере:

```
class A1
{
public:
    int    a;
    void   Func();
};

class A2
{
public:
    int    a;
    void   Func();
};

class B : public A1, public A2
{
    ...
};
```

```

void main()
{
    B b;
    b.a = 5;           // Ошибка!
    b.Func();          // Ошибка!
    b.A1::a = 6;       // Правильно
    b.A1::Func();      // Правильно
    b.A2::a = 7;       // Правильно
    b.A2::Func();      // Правильно
}

```

Еще одна неприятная ситуация, которая может возникать при множественном наследовании, заключается в том, что объект базового класса может многократно создаваться в объекте производного класса, если он наследуется по различным ветвям иерархии классов. Так, в следующем примере объект класса D будет два раза содержать объект класса A, поскольку он наследуется как от класса B, так и от класса C:

```

class A
{
    ...
};

class B : public A
{
    ...
};

class C : public A
{
    ...
};

class D : public B, public C
{
    ...
};

```

Для решения этой проблемы класс A может быть объявлен *виртуальным базовым классом*. В этом случае в производных классах объект такого класса будет создаваться только один раз. В следующем примере объект класса D будет включать один объект класса A:

```

class A
{
    ...
};

```

```

class B : virtual public A
{
    ...
};

class C : virtual public A
{
    ...
};

class D : public B, public C
{
    ...
};

```

4.5. Перегрузка

Перегрузка функций (методов) – одна из форм полиморфизма, которая заключается в возможности использования в одном пространстве имен нескольких функций с одинаковым именем, но с разными параметрами. Если в языке C нельзя было иметь одноименные варианты функции, различающиеся типами параметров, то язык C++ это позволяет. Например, в стандартной библиотеке C есть две реализации функции получения абсолютного значения: `abs` и `fabs`:

```

int abs(int x);
double fabs(double x);

```

Язык C++ разрешает определение нескольких функций с одним и тем же именем, если функции различаются числом или типом параметров. Поэтому в языке C++ возможны следующие реализации:

```

int abs(int x);
short abs(short x);
char abs(char x);
float abs(float x);
double abs(double x);

```

Перегрузка методов в C++ осуществляется аналогично перегрузке функций – в рамках одного класса определяются методы с одинаковыми именами, но с разными списками параметров. Например, класс `Circle` может иметь множество реализаций метода `Fit` (вписать) для изменения радиуса круга по заданной геометрической фигуре:

```

class Circle
{
private:
    double radius;
public:
    void    Fit(Rectangle &R);
    void    Fit(Square &S);
    void    Fit(Triangle &T);
    void    Fit(Ellipse &E);
    ...
};

void Circle::Fit(Rectangle &R)
{
    radius = min(R.Height(), R.Width())/2;
}

void Circle::Fit(Square &S)
{
    radius = S.Side()/2;
}

```

При переопределении и перегрузке функций (методов) следует учитывать следующие правила:

1. Перегружаемые методы должны находиться в одной области определения.
2. Перегружаемые функции различаются компилятором по их параметрам.
3. Посредством перегрузки вызывается версия функции, соответствующая конкретным типам параметров.
4. Переопределение функции осуществляется новым описанием функции в другой области определения в иерархиях классов.
5. Переопределяемые функции имеют идентичные имена и типы параметров.

Для методов наследуемых классов, имеющих одинаковые имена, возможна еще одна ситуация, называемая *сокрытием*. Сокрытие, или «затенение», осуществляется для методов, находящихся в различных областях определения, причем метод, описанный во внутренней области, скрывает описание, данное во внешней области. Доступ к скрытым функциям осуществляется через оператор расширения области видимости. Следующий пример показывает различные ситуации с одноименными методами:

```

class Figure
{
public:
    double Area();
    void Draw();
};

class Circle : public Figure
{
public:
    double Area();           // переопределение
    void Draw(int x, int y); // сокрытие
    void Fit(Rectangle &R);  // перегрузка
    void Fit(Triangle &T);   // перегрузка
    void Fit(Square &S);     // перегрузка
};

Circle c(100);
c.Draw(120, 50);           // правильно
c.Draw();                  // ошибка
c.Figure::Draw();          // правильно

```

Если в языке С операторы встроенные и однозначно определены для стандартных типов данных, то в языке С++ операторы аналогичны функциям, что дает возможность их перегрузки. Для операторов в языке С++ существует альтернативная форма записи, применяемая при перегрузке. В следующем примере показаны две конструкции, полностью аналогичные с точки зрения компилятора:

```

a = b+c;           // оператор сложения
a = b.operator+(c); // оператор сложения, записанный как функция

```

При перегрузке операторов программисты должны соблюдать следующие правила:

1. Нельзя определять новые операторы.
2. Нельзя перегружать операторы `::`, `?:`, `..`, `.*`, `#`, `##`.
3. По крайней мере один из операндов перегруженного оператора должен быть объектом класса или ссылкой на объект класса.
4. Нельзя изменять общий синтаксис оператора (число операндов, приоритет, задаваемые аргументы).

Операторы могут перегружаться как в области класса, так и вне классов. В первом случае оператор определяется как метод класса, во втором оператор представляется независимой функцией, первый параметр которой есть объект некоторого класса.

Для новых типов данных (классов) перегрузка операторов позволяет, сохраняя семантику, получить новую функциональность. Так, например, для класса Circle (круг) оператор сложения может означать вычисление радиуса таким образом, чтобы площадь была равна сумме площадей двух слагаемых кругов. В следующем примере показана перегрузка оператора «+» в рамках класса:

```
class Circle
{
private:
    double radius;
public:
    Circle(double R) { radius = R; }

    Circle& operator+(Circle &C);      // перегруженный оператор
};

Circle& Circle::operator+(Circle& C) // реализация оператора
{
    static Circle x(0);                // промежуточный объект
    x.radius = sqrt(radius*radius+C.radius*C.radius);
    return(x);
}
```

В следующем примере показана перегрузка оператора вне класса:

```
Circle& operator+(Circle& C1, Circle& C2)
{
    static Circle x();
    x.r = sqrt(C1.r*C1.r+C2.r*C2.r);
    return(x);
}
```

В следующем примере показана многократная перегрузка оператора присваивания:

```
class Circle
{
private:
    double radius;
public:
    Circle(double R) { radius = R; }

    Circle& operator=(Circle& C);      // перегруженный оператор 1
    Circle& operator=(Square& S);     // перегруженный оператор 2
};

Circle& Circle::operator=(Circle& C) // реализация оператора 1
{
}
```

```

        radius = C.radius;
        return(*this);
    }

Circle& Circle::operator=(Square& S) // реализация оператора 2
{
    radius = sqrt(S.Area())/M_PI;
    return(*this);
}

```

Перегрузка операторов ++ и -- имеет свои особенности. Для того чтобы различить префиксную и постфиксную формы операторов, при декларации постфиксной формы нужно задать один параметр типа int:

```

class Circle
{
private:
    double radius;
public:
    Circle(double R) { radius = R; }
    Circle& operator++();           // префиксная форма
    Circle& operator++(int);        // постфиксная форма
    Circle& operator--();           // префиксная форма
    Circle& operator--(int);        // постфиксная форма
};

Circle& operator++()                // префиксная форма
{
    radius *= 2;
    return(*this);
}

Circle& operator++(int)              // постфиксная форма
{
    static Circle x(0);
    x.radius = radius;
    radius *= 2;
    return(x);
}

```

```

Circle A(10);           // радиус A равен 10
Circle B = A++;         // радиус B равен 10, радиус A равен 20
Circle C = ++A;         // радиус C равен 40, радиус A равен 40

```

Перегруженные операторы, как правило, возвращают по ссылке объект того класса, для которого они определены. Это позволяет применять такие операторы в цепочных выражениях:

```

Circle a, b, c, d, e1, e2, e3;

```

```

e1 = a+b+c+d;           // выражение 1 и его интерпретация
e1.operator=(a.operator+(b).operator+(c).operator+(d));

e2 = (a+b)+(c+d);       // выражение 2 и его интерпретация
e2.operator=(a.operator+(b).operator+(c.operator+(d)));

e3 = a+(b+c)+d;         // выражение 3 и его интерпретация
e3.operator=(a.operator+(b.operator+(c)).operator+(d));

```

4.6. Шаблоны

Перегрузка в C++ позволяет создавать под одним именем целые семейства функций или операторов, настраиваемых на различные типы данных, передаваемые в качестве параметров. Зачастую эти реализации имеют одинаковый код и различаются только типами данных. В этом случае в языке C++ имеется мощное средство автоматической генерации кода – *шаблоны*.

Шаблоны могут создаваться для функций, классов и методов классов. Как правило, шаблоны описываются в файлах заголовков. При использовании в программе функции или класса компилятор автоматически генерирует по шаблону программный код для типов, заданных набором параметров. Таким образом, шаблоны сочетают в себе преимущества макросов и функций – автоматическую генерацию кода, которая выполняется однократно.

Рассмотрим пример создания шаблона для функции `min`. Предположим, что требуется программная реализация этой функции для следующих типов данных:

```

char      min(char x, char y);
short     min(short x, short y);
long      min(long x, long y);
float     min(float x, float y);
double    min(double x, double y);

```

Программный код этих функций будет одинаковым:

```

char min(char x, char y)
{
    return(x < y ? x : y);
}

short min(short x, short y)
{
    return(x < y ? x : y);
}

...

```


Все эти функции могут быть заменены следующим шаблоном, в котором параметр T означает некоторый тип данных (простой или класс):

```
template<class T>
T min(T x, T y)
{
    return(x < y ? x : y);
}
```

Объявление шаблона не вызывает никакой генерации кода. Если нужно форсировать создание версии функции для определенного типа данных, то можно выполнить так называемую *инсталляцию* шаблона, задав декларацию функции со специфицированным набором параметров:

```
long double min(long double x, long double y); // инсталляция
```

Если для каких-либо типов данных реализация функции по заданному шаблону невозможна, то можно создать отдельный код для таких реализаций. Это называется замещением шаблона. В следующем примере создается специфическая реализация функции min для строк символов:

```
char* min(char* x, char* y)
{
    return(strcmp(x, y) < 0 ? x : y);
}
```

Шаблоны классов создаются аналогично шаблонам функций. При декларации шаблона в угловых скобках задается перечень параметров типов данных, по которым будет генерироваться класс. В следующем примере создается шаблон класса Stack (стек) для произвольных типов данных:

```
template<class T>
class Stack
{
private:
    T*      data;           // адрес массива данных
    int     count;         // число элементов
public:
    Stack(int Number);      // конструктор
    ~Stack();               // деструктор

    void    Input(T Value); // помещение в стек
    T       Output();       // изъятие из стека
};
```

При реализации методов шаблона вне класса в операторе расширения области видимости указывается полная спецификация шаблона:

```
template<class T>
```

```

Stack<T>::Stack(int Number)
{
    data = new T[Number];
}

template<class T>
Stack<T>::~~Stack()
{
    delete [] data;
}

template<class T>
void Stack<T>::Input(T Value)

{
    data[count++] = Value;
}

template<class T>
T Stack<T>::Output()
{
    return(data[--count]);
}

```

Следующий пример показывает использование шаблона класса Stack в программе:

```

void main()
{
    Stack<int> istack(20);           // стек для типа int
    istack.Input(14);
    istack.Input(11);
    istack.Input(25);
    int i = istack.Output();

    Stack<float> fstack(10);        // стек для типа float

    fstack.Input(230.21);
    fstack.Input(18.513);
    fstack.Input(83.234);
    float f = fstack.Output();

    Stack<char*> cstack(24);        // стек для типа char*
    cstack.Input("First");
    cstack.Input("Second");
    cstack.Input("Third");
    char* c = cstack.Output();
}

```

Как и для шаблонов функций, для шаблонов классов применяется замещение, которое может быть полным или частичным. При полном замещении все параметры шаблона заменяются конкретными классами. При частичном замещении часть параметров шаблона сохраняется. Пример полного и частичного замещений шаблона:

```
template<class T1, class T2>           // шаблон класса
class Converter
{
    T1 obj1;
    T2 obj2;
    ...
};

class Converter<Rectangle, Circle>    // полное замещение
{
    ...
};

template<class T>
class Converter<T, Circle>            // частичное замещение
{
    ...
};
```

На основе шаблонов возможно создание новых типов данных с использованием оператора typedef:

```
typedef Stack<Circle> StackOfCircles;
typedef Stack<Square> StackOfSquares;

StackOfCircles circles(100);
StackOfSquares squares(120);
```

Созданные на основе шаблонов новые типы данных обладают всеми свойствами классов. В частности, они могут быть базовыми классами, а также наследовать атрибуты и методы других классов.

Для разработки программного обеспечения на C++ вместе с компилятором поставляется стандартная библиотека шаблонов STL. Эта библиотека включает большое число шаблонов для реализации различных структур данных, а также методов работы с ними [5]. В частности, поддерживаются контейнерные классы для хранения данных (множество, стек и др.), итераторы для унифицированного доступа к элементам контейнерных классов, а также функции, реализующие алгоритмы манипулирования данными контейнеров с помощью итераторов.

4.7. Примеры решения типовых задач программирования на C++

В этом разделе будет продолжено начатое в 2.13 рассмотрение типовых примеров программирования с использованием возможностей языка C++.

Пример 1. Реализовать пример 20 в гл. 2, используя возможности объектно-ориентированного программирования. Создать для точек класс, включающий координаты, конструктор и метод, вычисляющий расстояние от начала координат. Написать функцию, позволяющую определить, сколько точек находится в пределах окружности заданного радиуса.

```
class Point                                // декларация класса
{
private:
    float x;                               // координата x
    float y;                               // координата y
public:
    Point(float X, float Y);               // конструктор
    float Dist();                          // метод для расстояния
};

Point::Point(float X, float Y)             // конструктор
{
    x = X;
    y = Y;
}

float Point::Dist()                       // метод для расстояния
{
    return(sqrt(x*x+y*y));
}

Point* points[1000];                      // массив указателей
int n = 0;                                 // число точек

points[n++] = new Point(25.1, -13.2);      // заполнение массива
points[n++] = new Point(134.7, 29.1);      // заполнение массива
...

int fpoints(POINT* p[], int N, float R)
{
    int count = 0;                         // счетчик

    for(int i = 0; i < N; i++)              // цикл по массиву
        if(p[i]->Dist() <= R)             // проверка
            count++;                       // увеличить счетчик

    return(count);                         // вернуть счетчик
}
```

Пример 2. Рассчитать площадь изделия, состоящего из геометрических примитивов (прямоугольников, кругов и т. п.). Создать абстрактный базовый класс, включающий чисто виртуальный метод вычисления площади. Создать производные классы (прямоугольник, круг и т. п.), в которых определить специфические данные, конструкторы и переопределить функцию вычисления площади. Создать объекты и вычислить площадь.

```
class Figure                                     // базовый класс
{
public:
    virtual double Area() = 0;                  // метод для площади
};

class Rectangle : public Figure                 // класс прямоугольника
{
private:
    double a;                                  // сторона 1
    double b;                                  // сторона 2
public:
    Rectangle(double A, double B);             // конструктор
    virtual double Area();                     // метод для площади
};

Rectangle::Rectangle(double A, double B)
{
    a = A;
    b = B;
}

double Rectangle::Area()
{
    return(a*b);
}

class Circle : public Figure                   // класс круга
{
private:
    double r;                                  // радиус
public:
    Circle(double R);                          // конструктор
    virtual double Area();                     // метод для площади
};

Circle::Circle(double R)
{
    r = R;
}
```

```

double Circle::Area()
{
    return(M_PI*r*r);
}

Figure* figs[1000];           // массив указателей
int n = 0;                     // число примитивов

figs[n++] = new Rectangle(44.7, 15.2); // заполнение массива
figs[n++] = new Circle(63.7);         // заполнение массива
figs[n++] = new Rectangle(23.8, 43.6); // заполнение массива
...

double area = 0;               // общая площадь

for(int i = 0; i < n; i++)     // цикл по массиву
    area += figs[i]->Area();   // добавить площадь

printf("Площадь: %g", area);   // вывести на терминал

```

Пример 3. Имеются классы `Rectangle` и `Circle`, описывающие прямоугольники и круги соответственно. Перегрузить операторы сравнения, для того чтобы иметь возможность сравнивать в выражениях объекты указанных классов. Сравнение осуществлять по площади фигур.

```

class Rectangle;               // класс прямоугольника
class Circle;                  // класс круга

class Rectangle                 // класс прямоугольника
{
private:
    double a;                  // сторона 1
    double b;                  // сторона 2
public:
    Rectangle(double A, double B); // конструктор
    virtual double Area();          // метод для площади
    bool operator>(Rectangle &R);  // оператор >
    bool operator>(Circle &C);     // оператор >
    bool operator<(Rectangle &R);  // оператор <
    bool operator<(Circle &C);     // оператор <
};

class Circle                    // класс круга
{
private:
    double r;                  // радиус
public:
    Circle(double R);           // конструктор
    virtual double Area();      // метод для площади
    bool operator>(Circle &C);  // оператор >

```

```

    bool operator>(Rectangle &R);           // оператор >
    bool operator<(Circle &C);             // оператор <
    bool operator<(Rectangle &R);         // оператор <
}

Rectangle::Rectangle(double A, double B)
{
    a = A;
    b = B;
}

double Rectangle::Area()
{
    return(a*b);
}

bool Rectangle::operator>(Rectangle &R)
{
    return(Area() > R.Area());
}

bool Rectangle::operator>(Circle &C)
{
    return(Area() > C.Area());
}

bool Rectangle::operator<(Rectangle &R)
{
    return(Area() < R.Area());
}

bool Rectangle::operator<(Circle &C)
{
    return(Area() < C.Area());
}

double Circle::Area()
{
    return(M_PI*r*r);
}

bool Circle::operator>(Circle &C)
{
    return(Area() > C.Area());
}

bool Circle::operator>(Rectangle &R)
{
    return(Area() > R.Area());
}

```

```

bool Circle::operator<(Circle &C)
{
    return(Area() < C.Area());
}

bool Circle::operator<(Rectangle &R)
{
    return(Area() < R.Area());
}

```

Пример 4. Решить задачу из примера 3, используя шаблоны для перегруженных операторов.

```

class Rectangle;           // класс прямоугольника
class Circle;              // класс круга

class Rectangle             // класс прямоугольника
{
private:
    double a;              // сторона 1
    double b;              // сторона 2
public:
    Rectangle(double A, double B); // конструктор
    virtual double Area();         // метод для площади
    template<class T>
    bool operator>(T &O);          // шаблон оператора >
    template<class T>
    bool operator<(T &O);         // шаблон оператора <
};

class Circle                // класс круга
{
private:
    double r;              // радиус
public:
    Circle(double R);       // конструктор
    virtual double Area();  // метод для площади
    template<class T>
    bool operator>(T &O);   // шаблон оператора >
    template<class T>
    bool operator<(T &O);   // шаблон оператора <
}

template<class T>
bool Rectangle::operator>(T &O)
{
    return(Area() > O.Area());
}

```



```

template<class T>
bool Rectangle::operator<(T &O)
{
    return(Area() < O.Area());
}

bool Rectangle::operator<(Circle &C)
{
    return(Area() < C.Area());
}

template<class T>
bool Circle::operator>(T &O)
{
    return(Area() > O.Area());
}

template<class T>
bool Circle::operator<(T &O)
{
    return(Area() < O.Area());
}

```

Пример 5. Написать шаблон функции для вычисления суммы значений элементов массива.

```

template<class T>
T fsum(T M[], int N)
{
    T sum = 0; // переменная для суммы

    for(int i = 0; i < N; i++) // цикл по массиву
        sum += M[i]; // добавление

    return(sum); // вернуть сумму
}

```

Пример 6. Вызвать функцию из примера 5 для массивов различных типов данных.

```

short M1[] = {5, 7, 4, -3, ...};
int N1 = sizeof(M1)/sizeof(short);
float M2[] = {2.5, 4.9, -1.3, ...};
int N2 = sizeof(M2)/sizeof(float);

...

short S1 = fsum(M1, N1);
float S2 = fsum(M2, N2);

```

Глава 5. ОСНОВЫ АЛГОРИТМИЗАЦИИ

5.1. Понятие алгоритма

Как было показано в гл. 3 и 4 для разработки правильных и хорошо сопровождаемых программ важно четко выделить функции программы и реализовать их в виде хорошо структурированного набора модулей. Для каждой функции требуется определить логику ее выполнения. Эта логика называется *алгоритмом*. Дональд Эрвин Кнут, признанный авторитет в области программирования, определяет алгоритм как конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: конечностью, определенностью, вводом, выводом, эффективностью. Рассмотрим эти и другие важные свойства алгоритмов.

Дискретность – алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени. Знание числа шагов алгоритма и времени, затрачиваемого на один шаг, можно оценить время работы алгоритма для решения поставленной задачи.

Детерминированность – определенность. В каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Таким образом, алгоритм должен выдавать один и тот же результат для одних и тех же исходных данных.

Конечность – при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов.

Результативность – завершение работы алгоритма определенными результатами.

Эффективность – завершение работы алгоритма определенными результатами за определенное число шагов (время).

Масштабируемость – алгоритм должен быть применим к разным наборам исходных данных. Размер данных может требовать различных по масштабам ресурсов и влиять на время работы алгоритма, но не должен сказываться на его результативности.

В качестве иллюстрации свойств алгоритмов можно привести хорошо известную в математике задачу о Ханойских башнях. Старинная легенда гласит, что в одном из буддистских монастырей монахи уже тысячу лет занима-

ются перекладыванием колец. Они располагают тремя штырями, на которых надеты кольца разных размеров, образующие пирамиду (рис. 5.1). В начальном состоянии 64 кольца были надеты на первый штырь и упорядочены по размеру. Монахи должны переложить все кольца из первой пирамиды во вторую (на любой из свободных штырей), выполняя единственное условие – кольцо нельзя класть на кольцо меньшего размера. При перекладывании можно использовать все три штыря. В выработанном годами ритме монахи перекладывают одно кольцо за одну секунду. Как только они закончат свою работу, наступит конец света...



Рис. 5.1

Чтобы оценить степень риска для человечества, можно подсчитать число шагов алгоритма и тогда определить необходимое для решения задачи время. Рассмотрим последовательность действий для трех колец (рис. 5.2). Для перекладывания потребовалось 7 шагов.

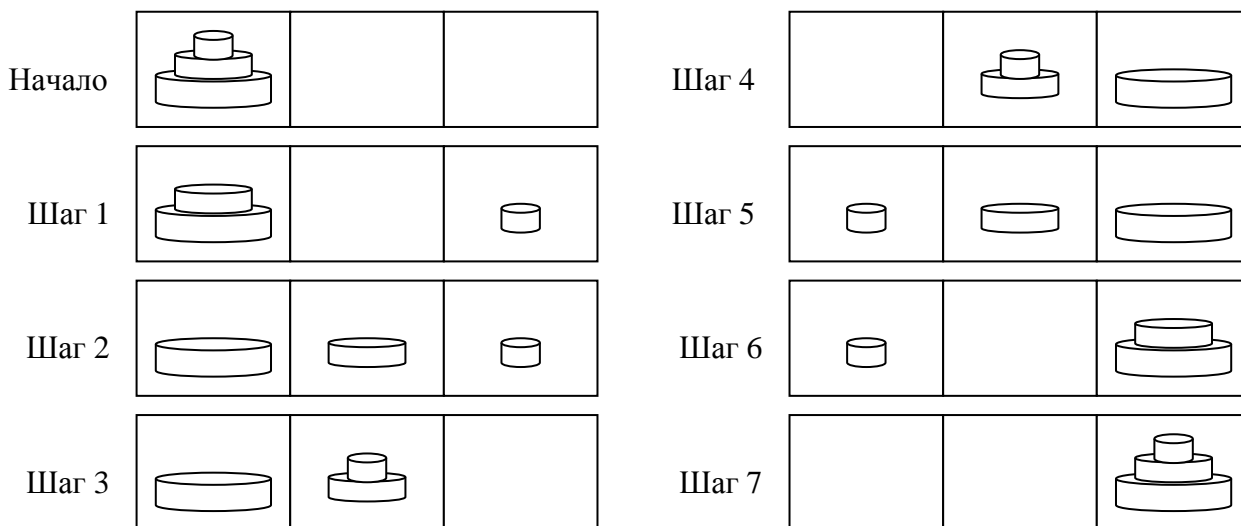


Рис. 5.2

В общем случае число шагов алгоритма вычисляется по формуле $2^N - 1$, где N – число колец. Таким образом, для перекладывания 64 колец потребуется 18 446 744 073 709 551 615 шагов. При скорости в одно перекладывание в секунду для решения задачи потребуется приблизительно 584 542 046 091 год, поэтому до конца света еще можно многое успеть...

Рассмотренный пример показывает важность такого свойства алгоритмов, как эффективность. Часто, говоря об эффективности алгоритмов, используют понятие временной функции сложности, которая показывает зависимость времени решения задачи от размерности исходных данных. Для простых задач эта зависимость может быть линейной, для более сложных – полиномиальной или экспоненциальной. Соответственно, задачи, для которых временная функция сложности определяется степенной зависимостью, называются полиномиальными (*P-задачи*), а задачи с экспоненциальной зависимостью – экспоненциальными (*NP-задачи*). Графики характерных временных функций сложности приведены на рис. 5.3.

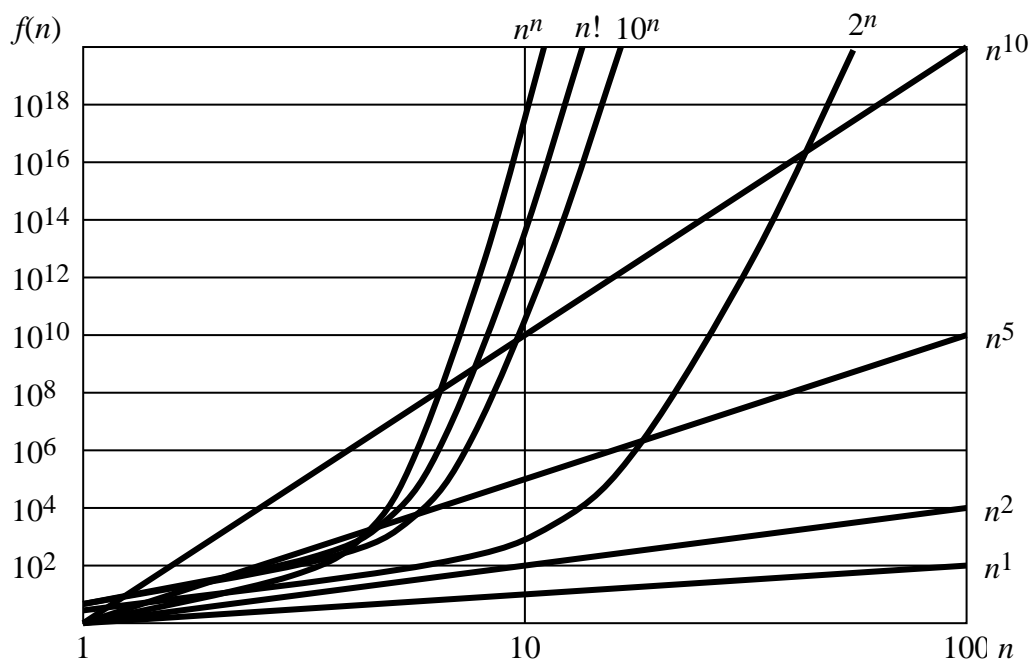


Рис. 5.3

Как видно из рис. 5.3, некоторые функции, например факториал, растут очень быстро. В связи с этим возникает вопрос о некотором пределе, за которым решение задачи некоторым методом не имеет смысла, поскольку требует невозможных ресурсов. В 1964 г. Ханс Бреммерман опубликовал статью, в которой доказал, что не существует системы обработки данных, искусственной или естественной, которая могла бы обрабатывать более $2 \cdot 10^{47}$ бит/с на грамм своей массы. Исходя из этого положения, гипотетический компьютер, который имел бы массу, равную массе Земли, за время, равное времени существования Земли, сумел бы обработать 10^{93} бит информации. За этим пределом, называемым пределом Бреммермана, находится область *трансвычис-*

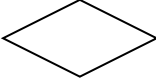
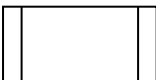
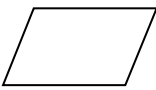


лительных задач, решение которых вычислительными методами заведомо невозможно. Достижение предела Бреммермана реально в NP-задачах. Так, функция $n!$ достигает указанного предела уже при $n = 67$, что еще раз подчеркивает важность разработки и применения эффективных алгоритмов.

5.2. Способы представления алгоритмов

Для хорошего понимания логики работы алгоритма требуется представить его в наглядном виде. Существуют разные способы представления алгоритмов. Далее будут рассмотрены два из них – блок-схемы и псевдокод.

Блок-схемы дают наглядное представление алгоритма в графическом виде. Имеется стандарт ЕСПД «Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения», который устанавливает требования к графическому представлению алгоритмов в виде блок-схем [6]. Основные символы, применяемые в блок-схемах алгоритмов, приведены в табл. 5.1.

Таблица 5.1

Наименование	Обозначение	Функция
Терминатор		Элемент отображает вход из внешней среды или выход из нее (основное применение – начало и конец программы)
Процесс		Выполнение одной или нескольких операций, обработка данных любого вида
Решение		Отображает решение с одним входом и двумя или более альтернативными выходами, из которых только один может быть выбран
Предопределенный процесс		Символ отображает выполнение процесса, который определен в другом месте программы
Данные		Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод)
Соединитель		Символ отображает выход в часть схемы и вход из другой части этой схемы
Комментарий		Используется для более подробного описания шага, процесса или группы процессов

Блок-схемы обладают хорошей наглядностью для сложных, но небольших по объему алгоритмов. Если схема не умещается на один лист, то приходится ее разбивать на части и пользоваться межстраничными соединителями, а это сразу сильно ухудшает наглядность.

С появлением структурного подхода к программированию графическое представление алгоритмов потеряло свои преимущества, поскольку программы стали иметь четкую структуру, образованную вложенными друг в друга типовыми конструкциями – следования, развилки, цикла. Для записи таких алгоритмов может более эффективно применяться псевдокод. Для сравнения блок-схем и псевдокода на рис. 5.4 приведены оба эти способа описания одного алгоритма вычисления факториала.

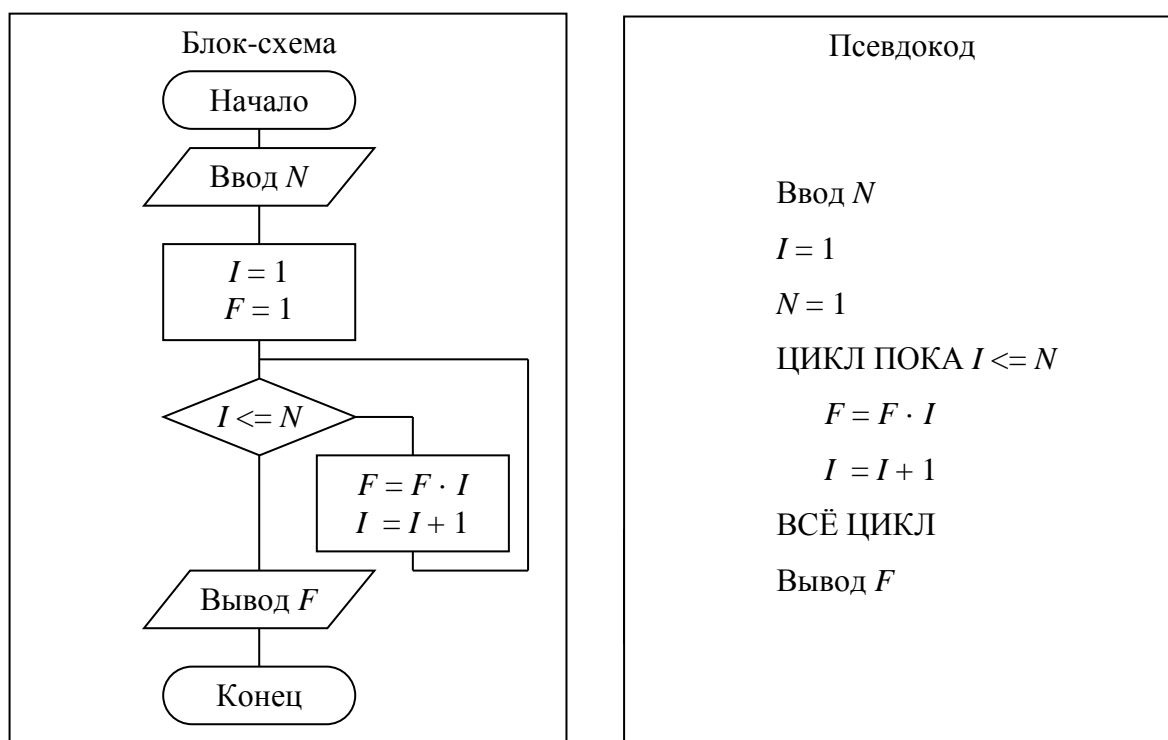


Рис. 5.4

Следует отметить, что программа, написанная на языке высокого уровня, также считается одним из способов представления алгоритмов. Во всех языках высокого уровня имеются конструкции, аналогичные псевдокоду, поэтому хорошо написанная программа легко читается, понимается и сопровождается другими программами. Основными правилами написания хорошо читаемых программ (иногда их называют самодокументированными) являются соблюдение стандартов на оформление кода (отступы для показа вложенности, пустые строки для отделения модулей и др.), а также подбор

«говорящих» имен для переменных, функций, классов и других именуемых элементов языка. В этом случае даже для больших проектов не требуется комментирования строк кода, поскольку разбиение на модули и использование понятных имен дают хорошо читаемый текст программы. В дальнейшем при рассмотрении примеров алгоритмов будет использоваться язык С.

5.3. Задача сортировки массивов

К наиболее часто встречающимся в программировании задачам относится сортировка массива данных. При этом считается, что массив целиком находится в оперативной памяти и сортировка должна осуществляться «на месте», т. е. без перезаписи массива в другую область памяти. Для примера рассмотрим массив, состоящий из восьми записей, имеющих определенную структуру, включающую три элемента данных (табл. 5.2).

Данная структура может быть представлена в программе следующим кодом:

```
typedef struct
{
    short   code;
    char    name[10];
    float   price;
} PROD;

PROD prod[8];
```

Таблица 5.2

Код	Наименование	Цена
44	Яблоки	35.50
55	Апельсины	29.90
12	Бананы	22.00
42	Лимоны	32.50
94	Мандарины	44.00
18	Груши	65.80
06	Сливы	39.95
67	Манго	58.00

Таблица 5.3

Методы	Сортировка включением		Сортировка выделением	Сортировка обменом	
	Прямое включение	Двоичное включение	Прямой выбор	Пузырьковая	Шейкерная
Улучшенные	С уменьшающимися расстояниями (Шелла)		С помощью дерева	Разделением (быстрая)	

Как правило, сортировка осуществляется в соответствии со значением одного из полей структуры, который называется *ключом сортировки*. В данном примере в качестве ключа будет использоваться поле «Код». В ряде слу-

чаев в качестве ключа могут использоваться несколько полей. Тогда упорядочение будет производиться по комбинации значений ключевых полей, которую часто называют *составным ключом*.

Известны различные методы сортировки, которые одинаково результативны (дают правильный результат при любом наборе входных данных), но обладают различной эффективностью. Классификация методов приведена в табл. 5.3.

5.4. Прямые методы сортировки массивов

Прямые методы сортировки отличаются простотой реализации, но низкой эффективностью. У всех прямых методов временная функция сложности полиномиальная и определяется выражением $N(N - 1)/2$, где N – число элементов массива. Для упрощения на иллюстрациях будет показана сортировка только ключевого поля.

Метод прямого включения (Straight insertion) заключается в последовательном выборе элементов массива начиная со второго (рис. 5.5). Выбранный элемент сравнивается с предыдущими, пока не будет найден элемент меньше выбранного или не будет достигнуто начало массива. Выбранный элемент включается сразу после найденного или в начало массива. Элементы от места включения и до места, где находился выбранный элемент, сдвигаются вправо на одну позицию.

В примере на рис. 5.5 на первом шаге выбранный элемент 55 остается на месте, поскольку слева от него находится меньший элемент 44. На втором шаге 12 включается в начало массива, а 44 и 55 сдвигаются на одну позицию вправо. На третьем шаге 42 включается после 12, а 44 и 45 сдвигаются. На четвертом шаге 94 остается на месте. На пятом шаге 18 включается после 12, а 42, 44, 55 и 94 сдвигаются. На шестом шаге 06 включается в начало массива, а 12, 18, 44, 55 и 94 сдвигаются. На седьмом шаге 67 включается после 55, а 94 сдвигается. Сортировка закончена. Программная реализация сортировки прямым включением приведена далее.

Входными параметрами функции `StraightInsertion` являются указатель на массив `prod` и число элементов массива `n`. Внешний цикл `for` начинается с индекса 1. Переменная `tmp` предназначена для временного хранения выбранного элемента, а переменная `j` – для запоминания места вставки.

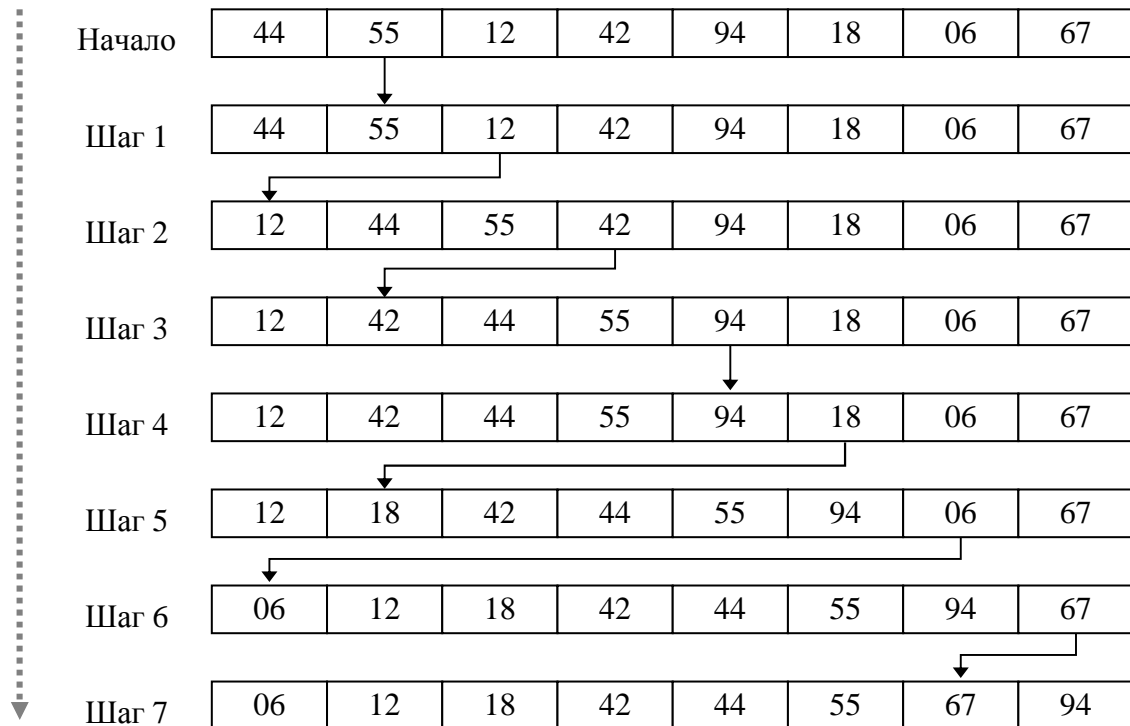


Рис. 5.5

```

void StraightInsertion(PROD* prod, int n)
{
    for(int i = 1; i < n; i++)
    {
        PROD tmp = prod[i];
        int j;

        for(j = i; j > 0 && tmp.code < prod[j-1].code; j--)
            prod[j] = prod[j-1];

        prod[j] = tmp;
    }
}

```

Метод прямого выбора (Straight selection) заключается в последовательном выборе элементов массива начиная с первого (рис. 5.6). Выбранный элемент меняется местами с наименьшим из последующих элементов.

В примере на рис. 5.6 на первом шаге выбранный элемент 44 меняется местами с наименьшим элементом справа – 06. На втором шаге меняются 55 и 12. На третьем шаге меняются 55 и 18. На четвертом шаге 44 остается на месте, поскольку справа нет меньших элементов. На пятом шаге меняются 94 и 44. На шестом шаге перестановок нет. На седьмом шаге 94 и 67 меняются

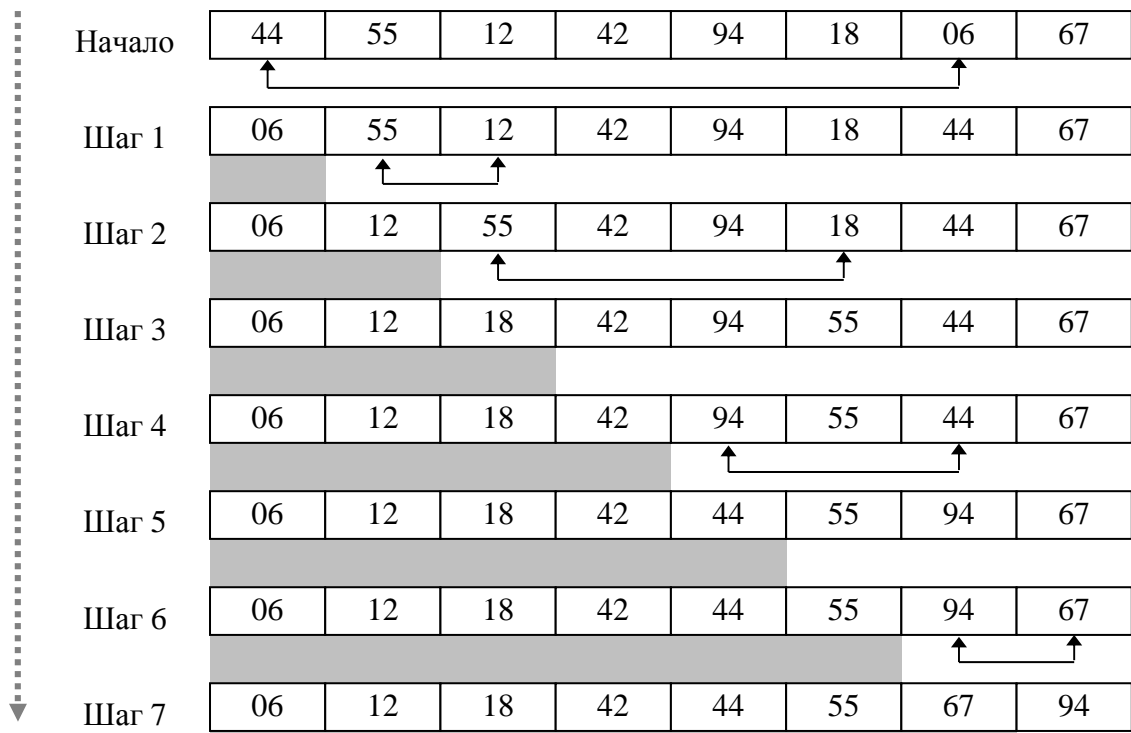


Рис. 5.6

местами. Сортировка закончена. Программная реализация сортировки прямым выбором приведена ниже.

```
void StraightSelection(PROD* prod, int n)
{
    for(int i = 0; i < n-1; i++)
    {
        PROD tmp = prod[i];
        int k = i;

        for(int j = i+1; j < n; j++)
            if(prod[j].code < tmp.code)
            {
                tmp = prod[j];
                k = j;
            }

        prod[k] = prod[i];
        prod[i] = tmp;
    }
}
```

Входные параметры функции StraightSelection аналогичны предыдущему примеру. Внешний цикл начинается с индекса 0. Переменная tmp предназначена для хранения выбранного элемента, а переменная k – для запоминания места наименьшего элемента, определяемого во вложенном цикле.

Метод прямого обмена, более известный как метод «пузырька» (Bubble sort), заключается в попарном сравнении соседних элементов и обмене их местами, если правый элемент меньше левого (рис. 5.7). На каждом шаге осуществляется проход массива справа налево, за каждый проход «всплывает» один самый легкий элемент. На каждом же последующем шаге делается на одно сравнение меньше.

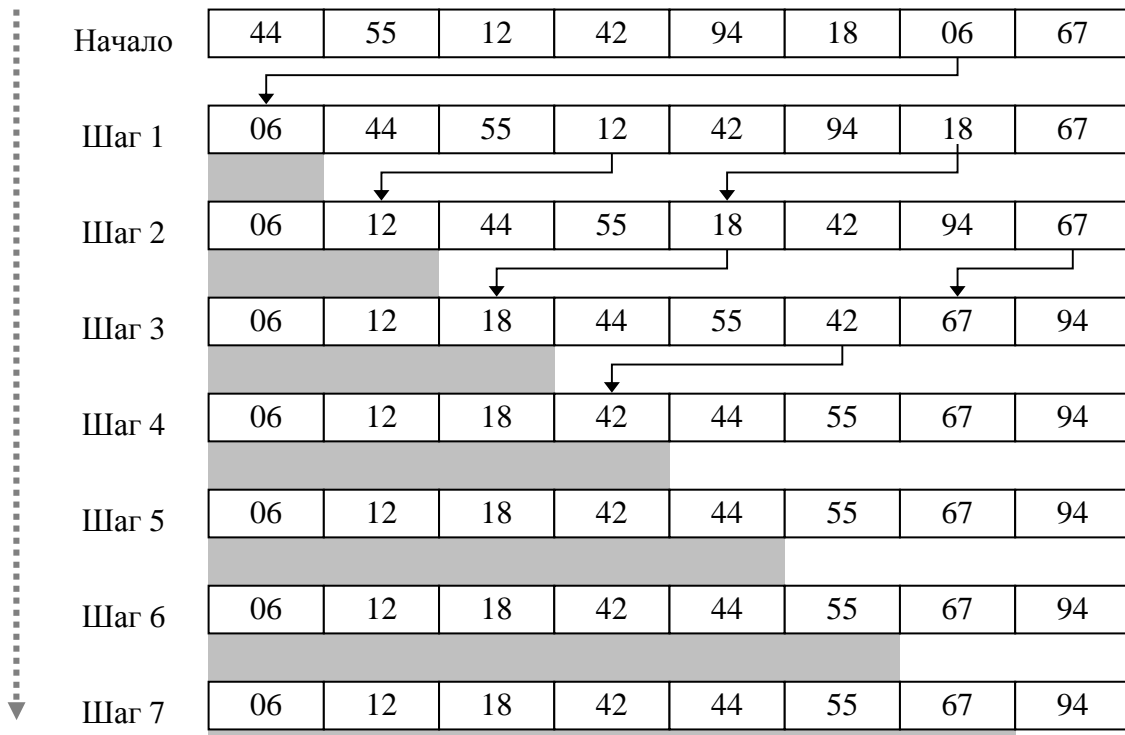


Рис. 5.7

В примере на рис. 5.7 на первом шаге всплывает элемент 06, на втором – 12, на третьем – 18, на четвертом – 42. На дальнейших шагах никаких перестановок не выполняется, поскольку массив уже отсортирован. Программная реализация пузырьковой сортировки (приводится ниже) очень проста.

```
void BubbleSort(PROD* prod, int n)
{
    for(int i = 1; i < n; i++)
        for(int j = n-1; j > i; j--)
            if(prod[j-1].code > prod[j].code)
            {
                PROD tmp = prod[j-1];
                prod[j-1] = prod[j];
                prod[j] = tmp;
            }
}
```

В функции BubbleSort два вложенных цикла for. Во втором цикле сравнивают соседние элементы, и если левый элемент больше правого, то выполняется перестановка этих элементов с использованием временной переменной tmp.

Если проанализировать пример на рис. 5.7, то можно увидеть, что при пузырьковой сортировке легкие элементы всплывают быстро – за один шаг (например, 06 на шаге 1, 12 на шаге 2, и т. д.), а тяжелые тонут медленно, смещаясь за один шаг только на одну позицию (94 достигает конца массива только за три шага). Кроме того, последние три шага оказались лишними, поскольку сортировка фактически завершилась уже на шаге 4.

Метод шейкерной сортировки (Shaker sort) был создан в попытке улучшить метод пузырька. В нем направление прохода массива при сравнении соседних элементов меняется на каждом шаге – происходит как бы встряхивание массива, откуда и взято название данного метода (рис 5.8).

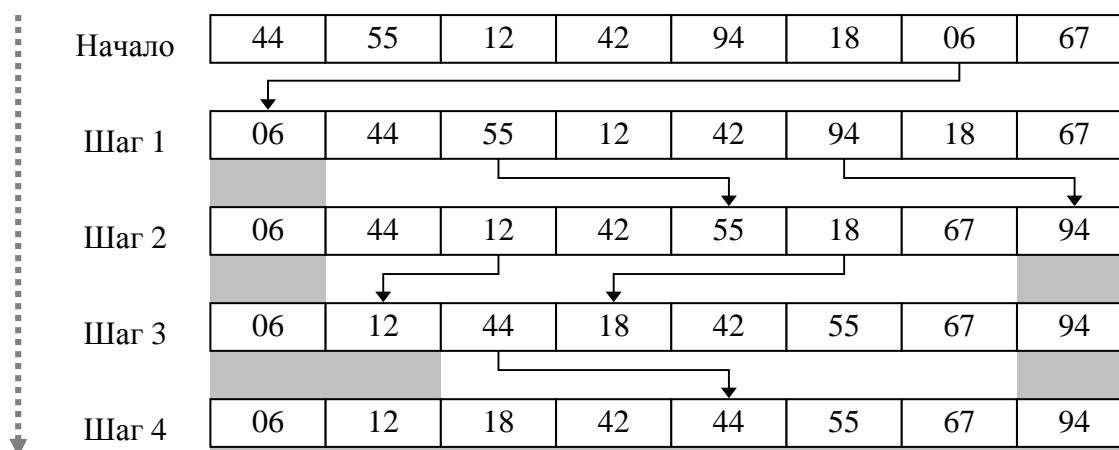


Рис. 5.8

В рассматриваемом примере на первом шаге сравнение идет справа налево, и элемент 06 всплывает в начало массива. На втором шаге сравнение идет слева направо и элементы 55 и 94 тонут, последний – в конец массива. На третьем шаге всплывают 12 и 18. На четвертом шаге тонет 44. Сортировка фактически закончена. Для того чтобы отказаться от последующих ненужных шагов сортировки, в алгоритме запоминается место последнего обмена на каждом шаге. Этим самым «суживаются» границы «встряхивания». Когда границы сходятся, можно с уверенностью сказать, что массив отсортирован. Программная реализация шейкерной сортировки приведена далее.

```

void ShakerSort(PROD* prod, int n)
{
    int L = 1;
    int R = n-1;
    int k = n-1;

    do
    {
        for(int j = R; j >= L; j--)
            if(prod[j-1].code > prod[j].code)
            {
                PROD tmp = prod[j-1];
                prod[j-1] = prod[j];
                prod[j] = tmp;
                k = j;
            }

        L = k+1;

        for(int j = L; j <= R; j++)
            if(prod[j-1].code > prod[j].code)
            {
                PROD tmp = prod[j-1];
                prod[j-1] = prod[j];
                prod[j] = tmp;
                k = j;
            }

        R = k-1;
    } while(L < R)
}

```

В функции ShakerSort используются переменные L и R для установления, соответственно, левой и правой границ для сортированной части массива. Переменная k применяется для запоминания места последней перестановки. Шаги сортировки реализуются в цикле do-while, пока границы L и R не сойдутся. На каждом шаге в двух циклах for происходит проход массива справа налево и слева направо в рамках границ L и R соответственно. После каждого цикла for изменяются границы по запомненному с помощью переменной k месту последней перестановки.

Метод шейкерной сортировки считается более быстрым, чем метод пузырька. В литературе [7] его эффективность оценивают как

$$\{N^2 - N[k + \ln(N)]\}/2,$$

где k – некоторый коэффициент.

5.5. Улучшенные методы сортировки массивов

Поскольку прямые методы сортировки имеют квадратичную временную функцию сложности, то их применение для больших массивов (более 50 тыс. элементов) существенно замедляет выполнение программ. В поисках более эффективных алгоритмов были созданы несколько улучшенных методов [7]. Рассмотрим подробно два из них.

Улучшенный метод Шелла (его еще называют сортировкой включением с уменьшающимися расстояниями). В этом методе сначала сортируется не весь массив, а только группы элементов, которые отстоят друг от друга на определенные расстояния (рис. 5.9).

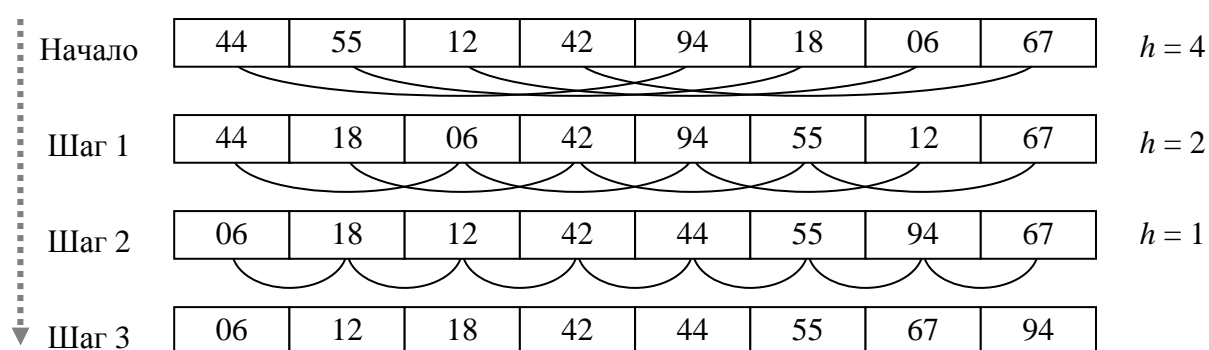


Рис. 5.9

В примере на рис. 5.9 расстояния уменьшаются в последовательности 4, 2, 1. На первом шаге идет сортировка в четырех группах – 44 и 94, 55 и 18, 12 и 06, 42 и 67 (расстояние между элементами $h = 4$). На втором шаге сортируются две группы – 44, 06, 94, 12 и 18, 42, 55, 67 ($h = 2$). На третьем шаге сортируется весь массив ($h = 1$). Сортировка в группах осуществляется прямым включением.

Может показаться, что эффективность метода Шелла должна быть низкой, поскольку выполняется много сортировок и в последний раз массив сортируется целиком. На самом деле это не так. Сначала сортировка выполняется в небольших группах, и это существенно улучшает порядок в массиве. На последних шагах производится очень мало перестановок, и здесь метод прямого включения работает быстро (для уже отсортированного массива эффективность метода прямого включения пропорциональна N).

Вопрос выбора последовательности расстояний в методе Шелла очень важен, поскольку от него зависит эффективность сортировки. В [7] рекомендуется следующий ряд расстояний (начиная с конца) – 1, 3, 7, 15, 31, ... Число

шагов t и расстояние h_k определяется по формуле $t = \log_2 N - 1$, $h_t = 1$, $h_{k-1} = 2h_k + 1$, где k – номер шага. В этом случае эффективность метода Шелла оценивают как $N^{1.2}$, что значительно превышает показатели прямых методов сортировки.

Метод быстрой сортировки (Quick sort), разработанный в 1962 г. Хоаром, оказался самым эффективным из известных методов сортировки. В основе этого метода лежит принцип разделения (рис. 5.9).

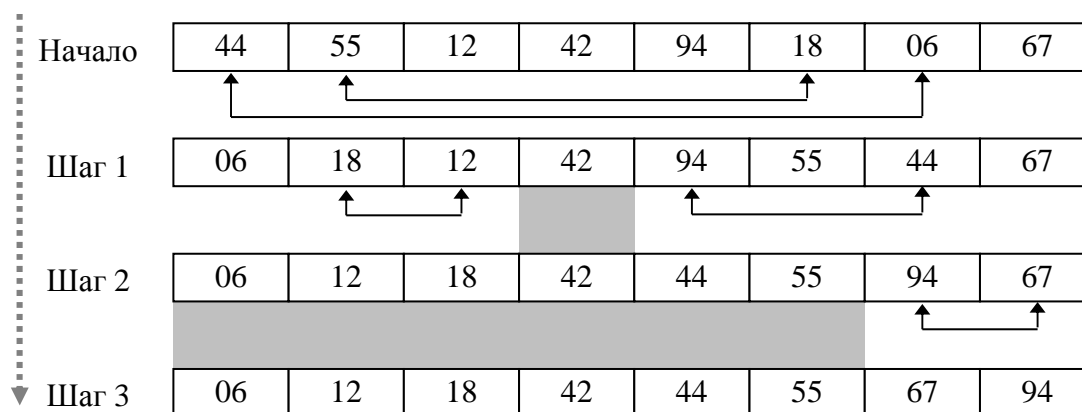


Рис. 5.9

На первом шаге берется некоторый элемент массива, который называется *опорным элементом*. Какой из элементов будет опорным, существенной роли не играет. Обычно берется средний элемент, но можно выбрать и любой другой, например первый. В примере на рис. 5.9 выбран находящийся в середине массива элемент 42. От концов массива к его середине справа и слева начинается просмотр элементов и их сравнение с опорным. Если слева оказывается элемент больше опорного, а справа – меньше, то эти элементы меняются местами (на первом шаге 44 и 06, а также 55 и 18). Когда просмотр заканчивается, то слева от места, на котором сошелся просмотр, находятся меньшие элементы, а справа – большие. На втором шаге процедура просмотра повторяется для полученных таким образом двух частей массива. В рассматриваемом примере на втором шаге в левой части (разделитель 18) меняются 18 и 12, в правой (разделитель 55) – 94 и 44. На третьем шаге разделение повторяется для тех частей массива, где осталось более одного элемента. Местами меняются 94 и 67. Сортировка закончена. Программная реализация быстрой сортировки приведена далее:

```

void QuickSort(PROD* prod, int n)
{
    sort(prod, 0, n-1);
}

void sort(PROD* prod, int L, int R)
{
    int i = L
    int j = R;
    PROD x = prod[(L+R)/2];

    do
    {
        while(prod[i].code < x.code) i++;

        while(x.code < prod[j].code) j--;

        if(i < j)
        {
            PROD tmp = prod[i];
            prod[i] = prod[j];
            prod[j] = tmp;
        }

        i++;
        j--;
    } while(i <= j);

    if(L < j) sort(prod, L, j);

    if(i < R) sort(prod, i, R);
}

```

Функция QuickSort вызывает рекурсивную функцию sort и передает ей массив, а также его границы L и R, в которых будет осуществляться просмотр части массива. В функции sort для просмотра элементов слева и справа используются индексы i и j соответственно. Переменная x считается опорным элементом, который берется посередине части массива между L и R. Цикл do-while продолжается, пока индексы i и j не сойдутся. В цикле осуществляется сравнение индексируемых переменными i и j элементов массива с опорным и обмен их местами. После цикла do-while функция sort рекурсивно вызывает саму себя для разделенных частей массива (если они имеют более одного элемента).

Эффективность быстрой сортировки определяется в [7] как $N \log(N)$.

5.6. Задача поиска в массивах

Задача поиска в массиве заключается в нахождении записи по значению одного из полей, которое называется *ключом поиска*. Если ключ поиска не имеет повторяющихся значений, то он называется *уникальным ключом*, в противном случае – *неуникальным*. На практике в большинстве случаев поиск осуществляется по уникальному ключу. Взяв в качестве примера массив из табл. 5.2, можно сформулировать задачу поиска следующим образом: «Какую цену имеет товар с кодом 55?».

Возможность применения различных методов поиска зависит от того, упорядочен ли массив по значениям ключа или нет. Для неупорядоченного массива единственно возможным алгоритмом является линейный поиск.

Линейный поиск (Linear search) заключается в последовательном переборе элементов массива, пока не встретится искомый элемент или не будет достигнут конец массива. Программная реализация линейного поиска очень проста:

```
int LinearSearch(PROD* p, int n, short Val)
{
    for(int i = 0; i < n; i++)
        if(p[i].code == Val)
            return(i);

    return(-1);
}
```

В качестве входных параметров в функцию LinearSearch передаются указатель на массив p, число элементов массива n, значение ключа поиска Val. В цикле for последовательно сравниваются элементы массива с искомым значением. При совпадении значений возвращается индекс найденного элемента. Если значение не найдено, цикл доходит до конца и функция возвращает –1, что означает отсутствие в массиве искомого элемента. Эффективность линейного поиска составляет в среднем $N/2$.

Если массив, в котором производится поиск, упорядочен по значению ключа, то это обстоятельство позволяет применить более эффективные методы поиска.

Двоичный поиск (Binary search) основан на делении массива на две части путем взятия некоторого элемента (как правило находящегося посередине) и сравнения с ним. Если искомый элемент меньше взятого, то поиск продолжается в левой части, если больше – то в правой. Процедура повторя-

ется, пока выбранный элемент не совпадет с искомым. В примере на рис. 5.10 ищется элемент со значением ключа 18. Деление массива пополам дает элемент 44. Деление левой части пополам дает 12. Деление оставшейся справа части – 42. Следующее деление оставшейся слева части дает искомый элемент 18. Поиск закончен.

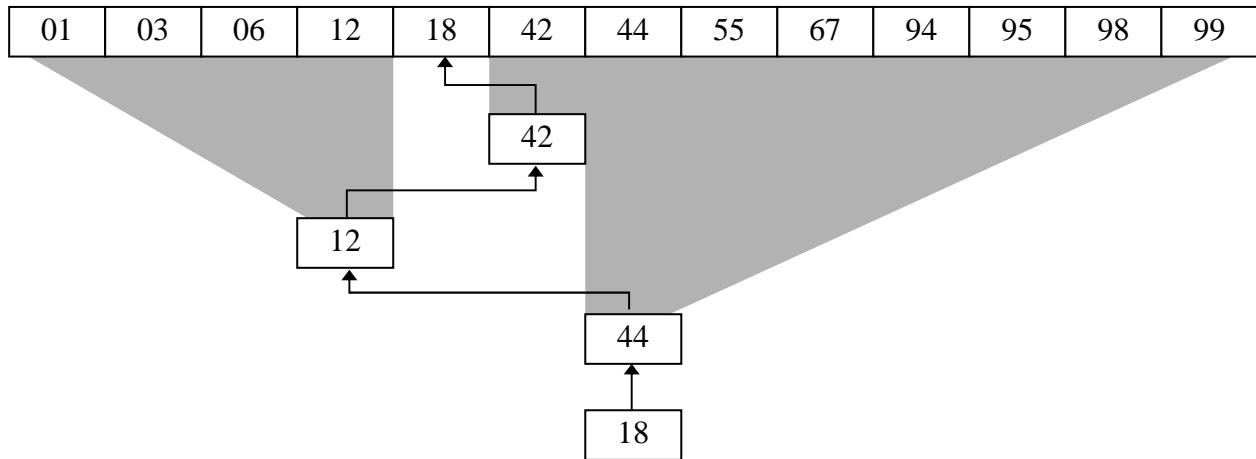


Рис. 5.10

Программная реализация двоичного поиска приведена ниже в виде нерекурсивной функции BinarySearch.

```
int BinarySearch(PROD* p, int n, short Val)
{
    int L = 0;
    int R = n-1;

    while(L <= R)
    {
        int m = (L+R)/2;

        if(p[m].code < Val)
            L = m+1;
        else
            if(p[m].code > Val)
                R = m-1;
            else
                return(m);
    }

    return(-1);
}
```

В качестве входных параметров в функцию BinarySearch передаются указатель на массив p, число элементов массива n, значение ключа поиска

Val. Переменные L и R устанавливаются на границы массива (0 и $n - 1$). Цикл while продолжается, пока границы L и R не сойдутся. Для деления массива пополам используется переменная m, которая устанавливается посередине L и R. Если искомый элемент больше $p[m]$, то левая граница смещается правее m. Если искомый элемент меньше $p[m]$, то правая граница смещается левее m. Если элемент найден, то функция возвращает индекс m. Если границы сошлись, а элемент не найден, то функция возвращает -1. Эффективность двоичного поиска $\log_2 N$.

Интерполяционный поиск (Interpolation search) похож на двоичный, только место деления массива выбирается исходя из гипотезы о линейном распределении значений элементов массива. Формула для определения индекса m имеет вид $m = L + ((Val - p[L]) (R - L)) / (p[R] - p[L])$, где L и R – границы поиска; Val – значение ключа поиска; $p[i]$ – значение элемента массива.

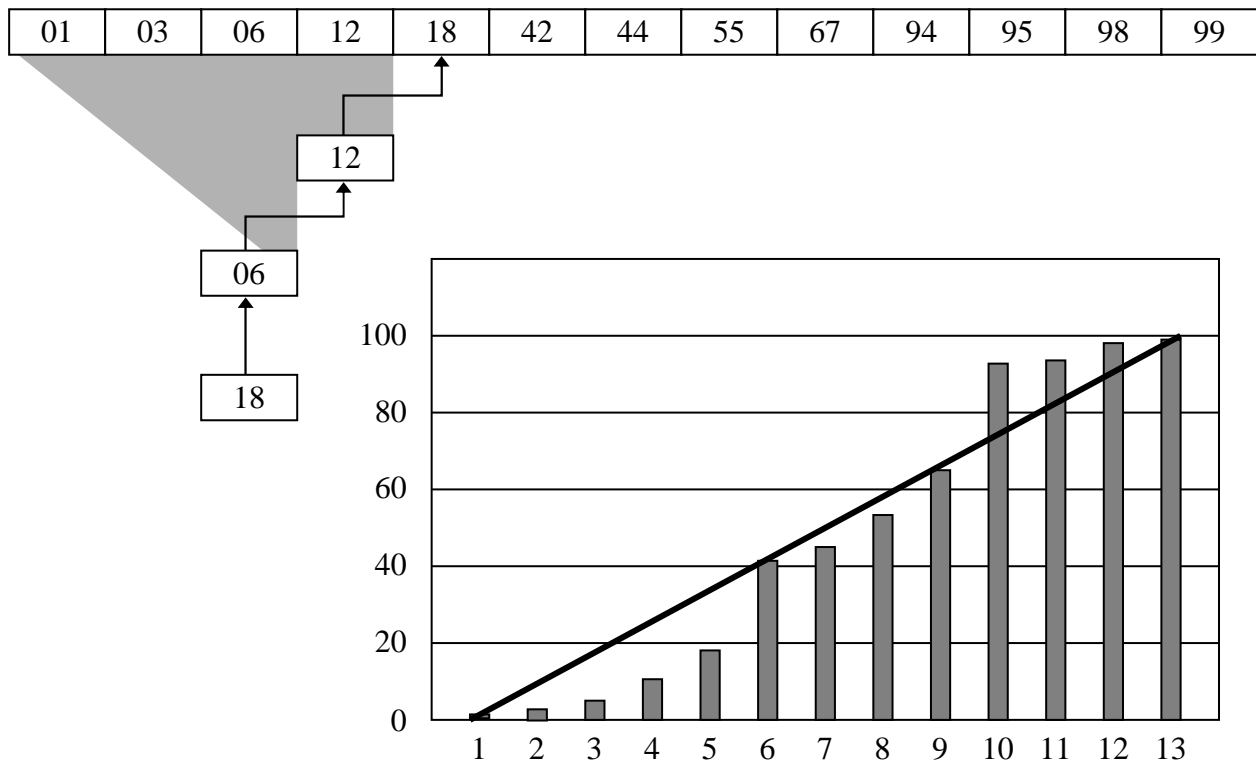


Рис. 5.11

В примере на рис. 5.11 первый индекс будет $0 + [(18 - 1)(12 - 0)] / (99 - 1)$, что дает 2, элемент 06. На втором шаге индекс будет равен 3, на третьем – 4, что даст искомый элемент 18. Приведенная далее программная реализация функции InterpolationSearch отличается от двоичного поиска только строчкой, где вычисляется переменная m.

```

int InterpolationSearch(PROD* p, int n, short Val)
{
    int L = 0;
    int R = n-1;

    while(L <= R)
    {
        int m = L+((Val-p[L].code)*(R-L))/(p[R].code-p[L].code);

        if(p[m].code < Val)
            L = m+1;
        else
            if(p[m].code > Val)
                R = m-1;
            else
                return(m);
    }

    return(-1);
}

```

Эффективность интерполяционного поиска не хуже двоичного, а при достаточно равномерном распределении данных в массиве он работает очень быстро.

5.7. Задача поиска в строках

Задача поиска в строках встречается довольно часто при обработке текстовых данных, например при работе с текстовыми документами. Поиск отдельных слов или фраз в тексте большого размера (например, в совокупности документов) может оказаться сложной задачей, требующей применения эффективных методов решения.

Формально задача поиска в строках формулируется следующим образом. Текст представляет собой неупорядоченный массив символов, принадлежащих некоторому алфавиту. Образец поиска – также массив символов меньшего размера, принадлежащих к тому же алфавиту. Требуется найти первое совпадение части текста и образца.

Прямой поиск (Direct search) заключается в посимвольном сравнении текста и образца начиная от начала текста. При несовпадении очередного символа образец «сдвигается» на одну позицию вправо и процедура сравнения повторяется. В примере на рис. 5.12 в тексте ищется строка «УЧЕНИК», а несовпадающие символы перечеркнуты.

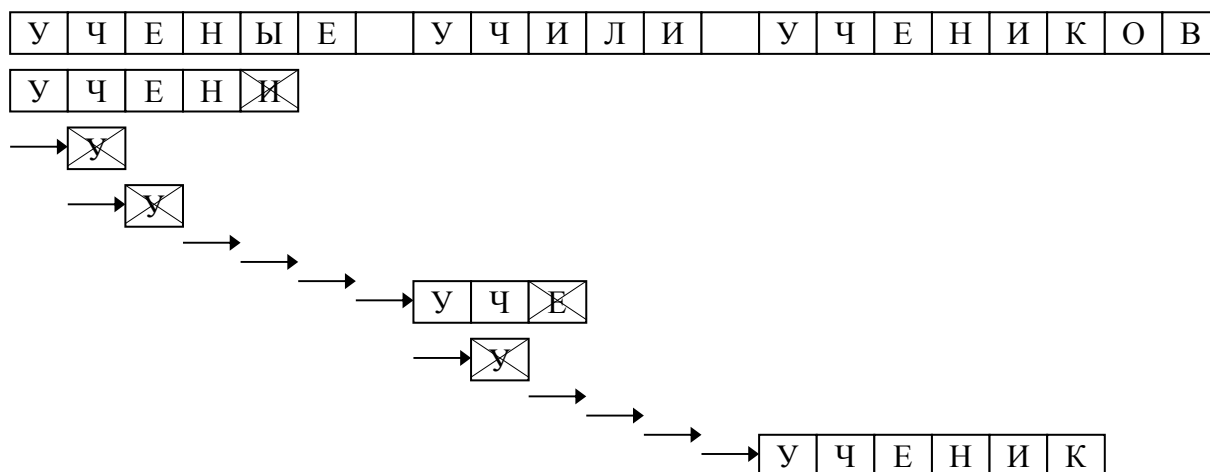


Рис. 5.12

Программная реализация прямого поиска строки приведена ниже:

```
char* StringSearch(char* Text, char* Val)
{
    for(char* p = Text; *p; p++)
    {
        bool found = true;

        for(char* v = Val, *s = p; *v; v++, s++)
            if(!*s || *v != *s)
            {
                found = false;
                break;
            }

        if(found) return(p);
    }

    return(NULL);
}
```

Входными параметрами функции StringSearch являются указатель Text на строку, в которой осуществляется поиск, а также указатель Val на строку-образец. Во внешнем цикле for указатель p «скользит» по тексту, смещаясь на одну позицию после тела цикла. Логическая переменная found нужна для хранения результата посимвольного сравнения во вложенном цикле for. Для сравнения используются два указателя: s – на строку от текущего положения p и v – на образец. Внутренний цикл продолжается, пока не будет встречено несовпадение или конец одной из строк. Если встречен конец образца, то это означает успешный поиск и функция StringSearch возвращает указатель на найденное место в строке. В противном случае функция возвращает NULL.

Алгоритм Кнута–Морриса–Пратта в отличие от прямого поиска, недостатком которого является смещение каждый раз всего на один шаг, использует полученную в результате сравнения информацию и смещается на большее число позиций после частичного совпадения строк.

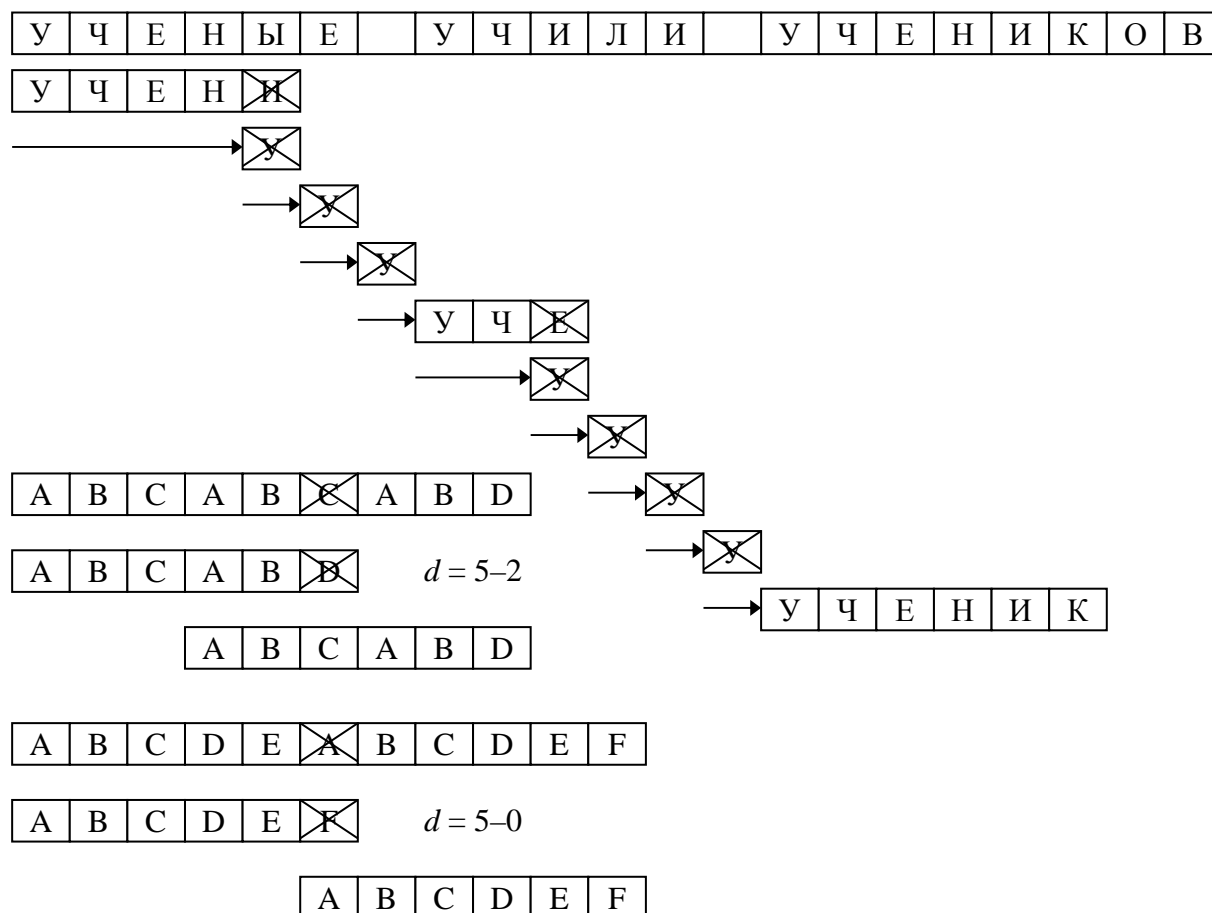


Рис. 5.13

Интересная особенность данного алгоритма – то, что смещение, на которое можно сдвигать образец, зависит не от текста, а только от образца. Соответственно, в начале работы алгоритма можно сформировать таблицу, в которой для каждой позиции образца можно сопоставить соответствующее смещение. На рис. 5.13 показан принцип вычисления смещения. Для i -го элемента образца смещение d будет равно $i - k$, где k – длина наибольшей строки слева от i -го элемента, совпадающей с началом образца.

Рассмотренный алгоритм Кнута–Морриса–Пратта экономит время при большом числе частичных совпадений, а это случается довольно редко на практике. Интересная идея, которая заключается в максимальном смещении при несовпадении с образцом, реализована в другом алгоритме.

Алгоритм Бойера–Мура. В этом алгоритме сравнение с образцом начинается не сначала, а с конца образца (рис. 5.14). При несовпадении образец смещается на некоторое число символов в зависимости от символа строки, на котором было несовпадение. Таблица смещений формируется в начале поиска, но механизм ее формирования несколько иной. Для каждого символа алфавита берется расстояние от места его последнего вхождения в образец до конца образца. Если символ в образце отсутствует, то в качестве смещения берется длина образца.

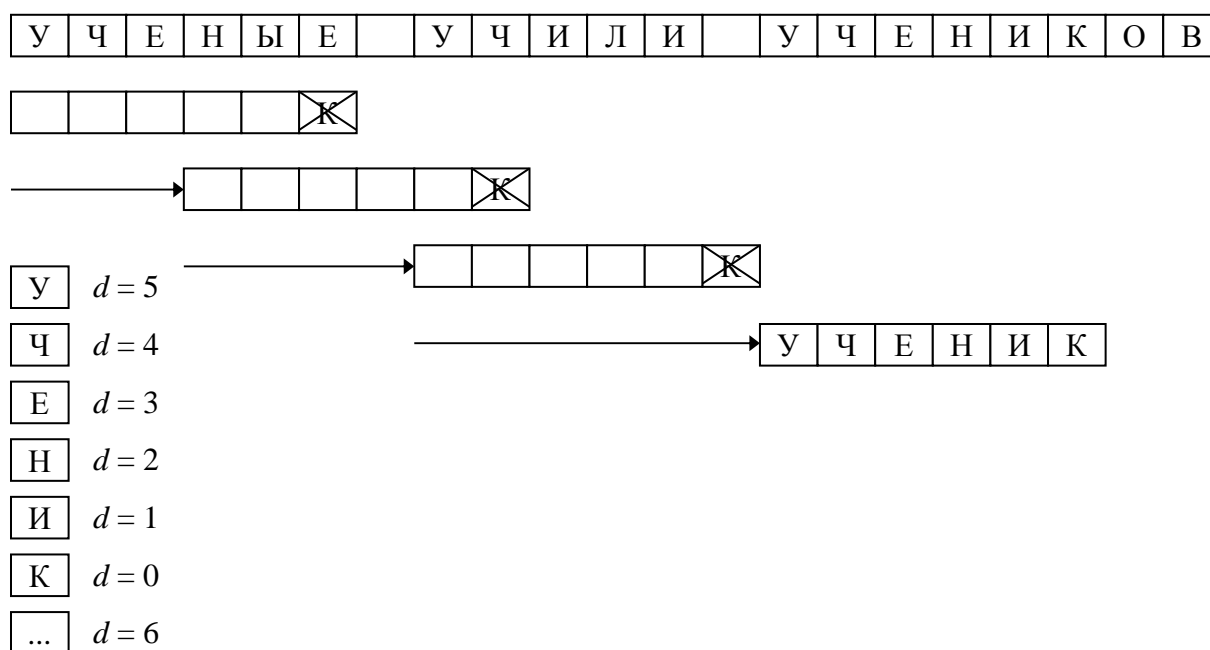


Рис. 5.14

В примере на рис. 5.14 при несовпадении на 1-м шаге на символе «Е» образец смещается на 3 позиции, на 2-м шаге на «Ч» – на 4 позиции, на 3-м шаге на пробеле – сразу на 6 позиций.

5.8. Задача поиска в файлах

В отличие от поиска в массивах, когда все данные находятся в оперативной памяти и время доступа к любому элементу массива одинаково, при поиске в файлах наибольшая часть времени тратится на чтение данных с периферийных устройств (дисков). По аналогии с линейным поиском последовательное чтение всех блоков данных из файла было бы крайне непроизводительно. Поэтому для повышения быстродействия стремятся использовать такие методы поиска, которые позволяли бы сразу прочитать в оперативную память именно тот блок, который содержит требуемые данные.

В практических задачах данные, как правило, ищутся по значению одного из полей, которое, как и в случае поиска в массивах, называется ключом поиска. Таким образом, поиск в файлах может быть сформулирован как получение номера блока файла по значению ключа. Рассмотрим два способа организации файлов, которые позволяют эффективно решить эту задачу.

Индексно-последовательный метод доступа (Index sequence access method – ISAM) требует сортировки файла по значению ключа. Для значений ключа строится и запоминается отдельно индекс, который представляет собой двоичное дерево принятия решений (рис. 5.15).

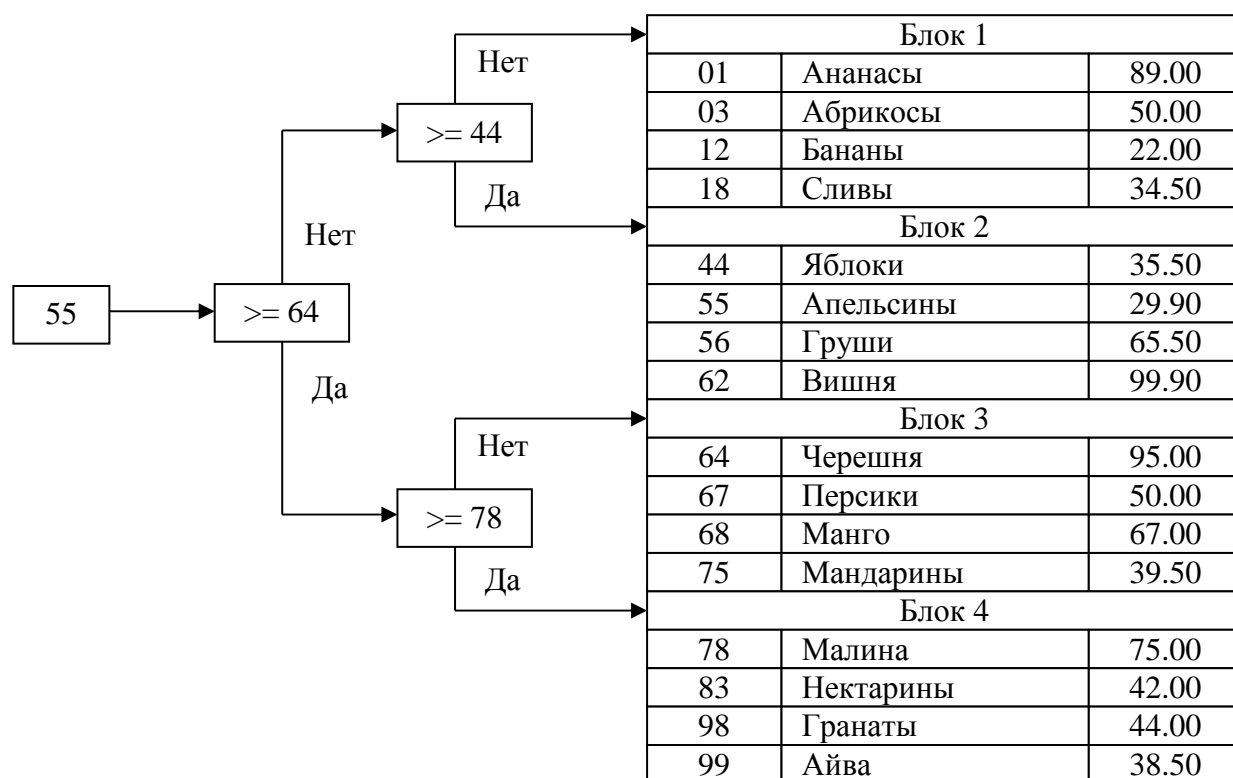


Рис. 5.15

В узлах индекса находятся значения ключа, с которых начинаются блоки файла. Поиск заключается в проходе по дереву индекса и выходе на конкретный блок файла. Число уровней индекса может быть вычислено по формуле $K = \log_2 N$, где N – число блоков файла. Для получения номера искомого блока требуется K сравнений, поэтому ISAM – достаточно эффективный метод поиска.

Хэширование – еще более эффективный метод поиска. В этом методе значение ключа непосредственно преобразуется в номер блока с помощью специальной функции, называемой хэш-функцией. Для примера рассмотрим

файл, содержащий 10 блоков (рис. 5.16). В качестве хэш-функции будем использовать остаток от деления значения ключа на 10. При размещении данных в файле номер блока для каждой записи определяется с помощью хэш-функции. Таким образом, запись с ключом 01 оказалась в первом блоке, записи 12 и 62 – во втором, 03 и 83 – в третьем, и т. д.

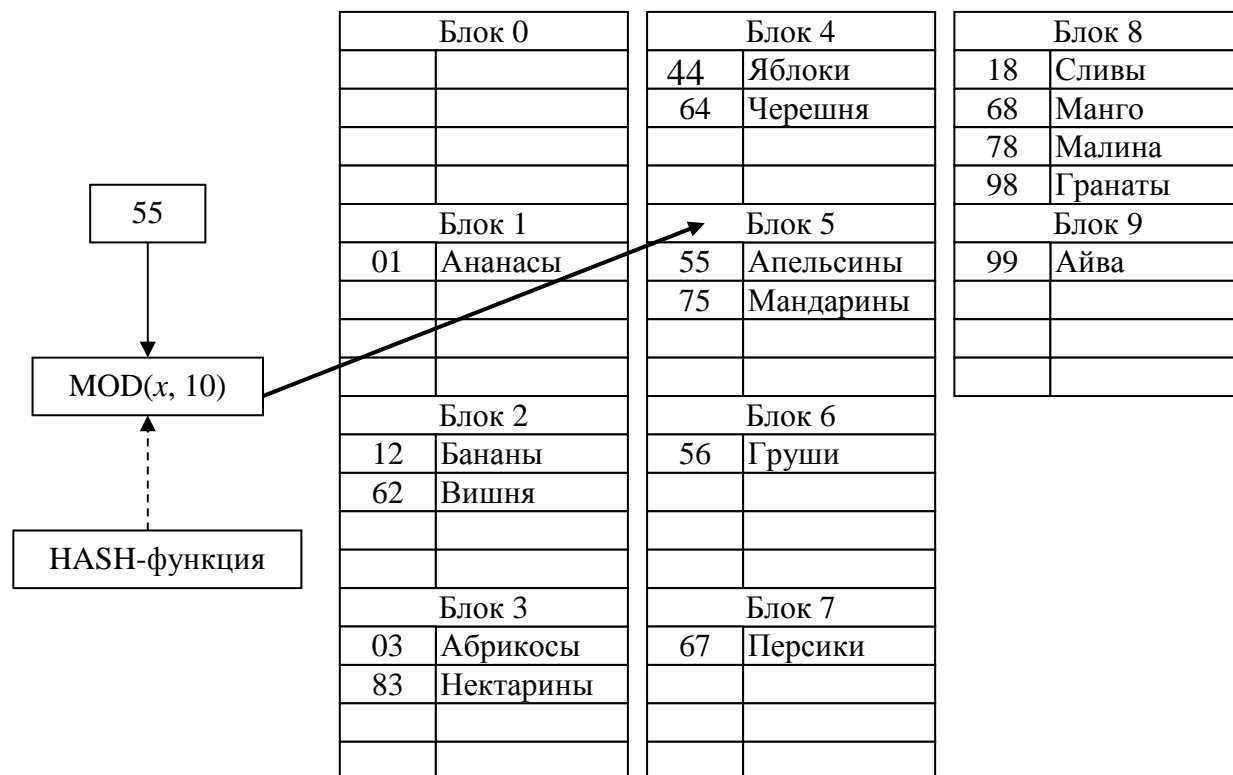


Рис. 5.16

При поиске по значению ключа, например 55, вычисляется хэш-функция (которая в данном случае дает 5) и сразу читается соответствующий блок файла, в котором данные уже ищутся как в массиве. Поскольку в блоке файла размещается немного записей, то даже применение линейного поиска будет эффективным.

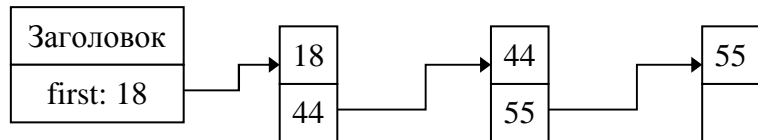
5.9. Динамические структуры данных

Как было показано ранее, хранение данных в массивах, отсортированных по значению ключа, позволяет применять эффективные методы поиска (двоичный, интерполяционный). Вместе с тем, на практике часто возникают задачи, когда требуется связать определенным образом некоторые элементы массива. В таком случае используют дополнительные поля, которые содержат значения ключа связываемых элементов. Такие поля называют *указателями*, а образуемые с их помощью структуры данных – динамическими.

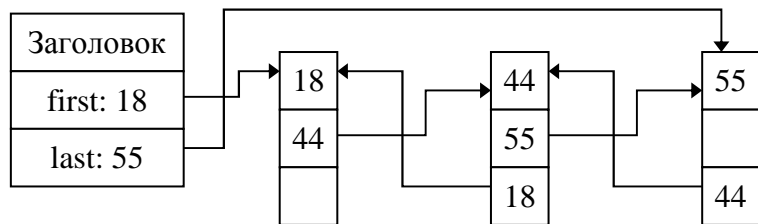
Линейный список (List) – это одна из наиболее простых динамических структур. Список образуется с помощью включаемого в структуру записи (рис. 5.17, а) указателя на следующий элемент. Для входа в список используется отдельная структура, которая называется *заголовком* и содержит указатель на первый элемент списка (рис. 5.17, б).

Код	Фамилия
01	Александров
02	Алексеев
03	Антонов
...	...
18	Иванов
19	Михайлов
...	...
44	Петров
...	...
55	Сидоров
56	Тимофеев

а



б



в

Рис. 5.17

Линейный список бывает однонаправленным, если он поддерживается только указателями на последующие элементы, или двунаправленным, если поддерживаются также указатели на предыдущие элементы. В случае двунаправленного списка (рис. 5.17, в) заголовок включает указатели как на первый элемент списка, так и на последний его элемент. Преимущество двунаправленного списка заключается в возможности обхода списка как вперед, так и назад, а также в лучшем контроле целостности связей в списке. Пример обхода однонаправленного списка иллюстрирует следующая программа:

```
typedef struct {short code; char name[24]; short next;} PERSON;
typedef struct {short first;} HEADER;
PERSON pers[] = {
    { 1, "Александров", 0},
    { 2, "Алексеев", 0},
    { 3, "Антонов", 0},
    {18, "Иванов", 44},
    {19, "Михайлов", 0},
    {44, "Петров", 55},
    {55, "Сидоров", 0},
    {56, "Тимофеев", 0},
};
```

```

HEADER head = {18};

void main()
{
    printf("*** Список персонала ***\n\n ");
    short next = head.first;

    while(next)
    {
        int index = BinarySearch(pers, next);
        printf("%s\n", pers[index].name);
        next = pers[index].next;
    }
}

```

В рассматриваемой программе данные находятся в упорядоченном массиве `pers` типа `PERSON`. Список поддерживается указателем `next`, входящим в структуру `PERSON`. Заголовок списка представлен переменной `head`, имеющей тип структуры `HEADER`. Конец списка определяется по нулевому значению указателя. Для обхода списка в программе используется переменная `next`, которой изначально присваивается взятое из заголовка значение ключа первого элемента. В цикле `while`, пока значение ключа следующего элемента не равно нулю, определяется его индекс (с помощью функции двоичного поиска `BinarySearch`). Поле `name` выводится на печать. В конце цикла переменной `next` присваивается значение указателя на следующий элемент списка. Результат работы программы приведен ниже:

```

*** Список персонала ***

```

```

Иванов
Петров
Сидоров

```

К характерным задачам при работе со списками относятся включение элемента в список и исключение элемента из списка. При этом данные в массиве остаются на своих местах, а манипулируют только указателями. На рис. 5.18 показан пример включения (*а*) и исключения (*б*) элементов двунаправленного списка.

Для иллюстрации операций включения и исключения изменим в рассмотренной ранее программе структуры данных и добавим функции включения и исключения элементов:

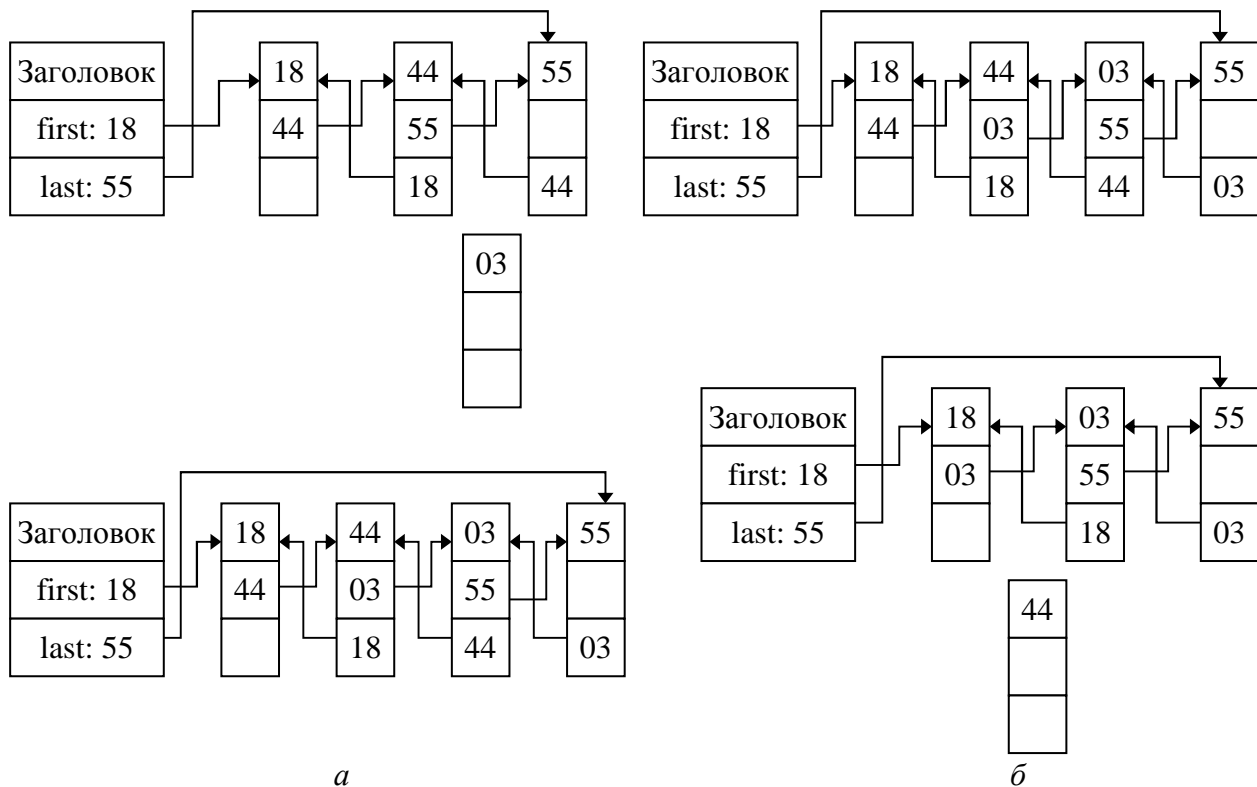


Рис. 5.18

```
typedef struct {short code; char name[24]; short next; short
prev} PERSON;

typedef struct {short first; short last;} HEADER;

PERSON pers[] = {
    { 1, "Александров", 0, 0},
    { 2, "Алексеев", 0, 0},
    { 3, "Антонов", 0, 0},
    {18, "Иванов", 44, 0},
    {19, "Михайлов", 0, 0},
    {44, "Петров", 55, 18},
    {55, "Сидоров", 0, 44},
    {56, "Тимофеев", 0, 0},
};

HEADER head = {18, 55};

void InsertIntoList(short Key, short Where)
{
    int index1 = BinarySearch(pers, Where);
    int index2 = BinarySearch(pers, Key);
    int index3 = BinarySearch(pers, pers[index1].next);
    pers[index3].prev = Key;
    pers[index2].next = pers[index1].next;
    pers[index1].next = Key;
    pers[index2].prev = Where;
}
```

```

void RemoveFromList(short Key)
{
    int index2 = BinarySearch(pers, Key);
    int index1 = BinarySearch(pers, pers[index2].prev);
    int index3 = BinarySearch(pers, pers[index2].next);
    pers[index1].next = pers[index2].next;
    pers[index3].prev = pers[index2].prev;
    pers[index2].next = 0;
    pers[index2].prev = 0;
}

void main()
{
    InsertIntoList(44, 3);
    RemoveFromList(44);
}

```

В функцию `InsertIntoList` первым параметром передается ключ добавляемого к списку элемента, а вторым параметром (Where) – ключ элемента, после которого будет вставлен новый элемент. В функцию `RemoveFromList` передается ключ удаляемого элемента.

В рассматриваемых примерах работы с линейными списками элементы включались и исключались в произвольных местах списка. Если элементы добавляются и удаляются только на концах списка, то могут применяться более простые структуры – *очередь* или *стек*. Очередь представляет собой список, организованный по принципу «первым вошел – первым вышел» (в английском варианте – FIFO, аббревиатура от «First Input First Output»). Стек имеет противоположную очереди дисциплину «последним вошел – первым вышел» (в английском варианте – LIFO, аббревиатура от «Last Input First Output»). В простейшем варианте очередь или стек могут быть представлены массивом указателей. В приведенном далее примере программы показана работа с очередью и стеком (простейший вариант, без контроля переполнения массива и обращения к пустому массиву):

```

short queue[100];
int qsize = 0;

short stack[100];
int ssize = 0;

void QueueIn(short Value)
{
    queue[qsize++] = Value;
}

```

```

short QueueOut()
{
    short value = queue[0];
    qsize--;

    for(int i = 0; i < qsize; i++)
        queue[i] = queue[i+1];

    return(value);
}

void StackIn(short Value)
{
    stack[ssize++] = Value;
}

short StackOut()
{
    return(stack[--ssize]);
}

void main()
{
    printf("*** Очередь ***\n");

    QueueIn(18);
    QueueIn(44);
    QueueIn(3);
    QueueIn(55);

    while(qsize)
        printf("%s\n", pers[BinarySearch(QueueOut())].name);

    printf("*** Стек ***\n");

    StackIn(18);
    StackIn(44);
    StackIn(3);
    StackIn(55);

    while(ssize)
        printf("%s\n", pers[BinarySearch(StackOut())].name);
}

```

В программе очередь представлена массивом queue, стек – массивом stack. Переменные qsize и ssize показывают число элементов в очереди и стеке соответственно. В функции QueueIn новый элемент помещается в конец очереди. В функции QueueOut элемент берется из начала очереди, при этом

очередь сдвигается к началу на одну позицию. В функции StackIn элемент помещается на «вершину» стека, а в функции StackOut – забирается с «вершины». Результаты работы программы приведены ниже:

```

*** Очередь ***
Иванов
Петров
Александров
Сидоров
*** Стек ***
Сидоров
Александров
Петров
Иванов

```

На практике часто приходится сталкиваться с динамическими структурами данных, имеющими иерархический характер. Рассмотрим пример задачи расчета комплектации продукции (рис. 5.19).

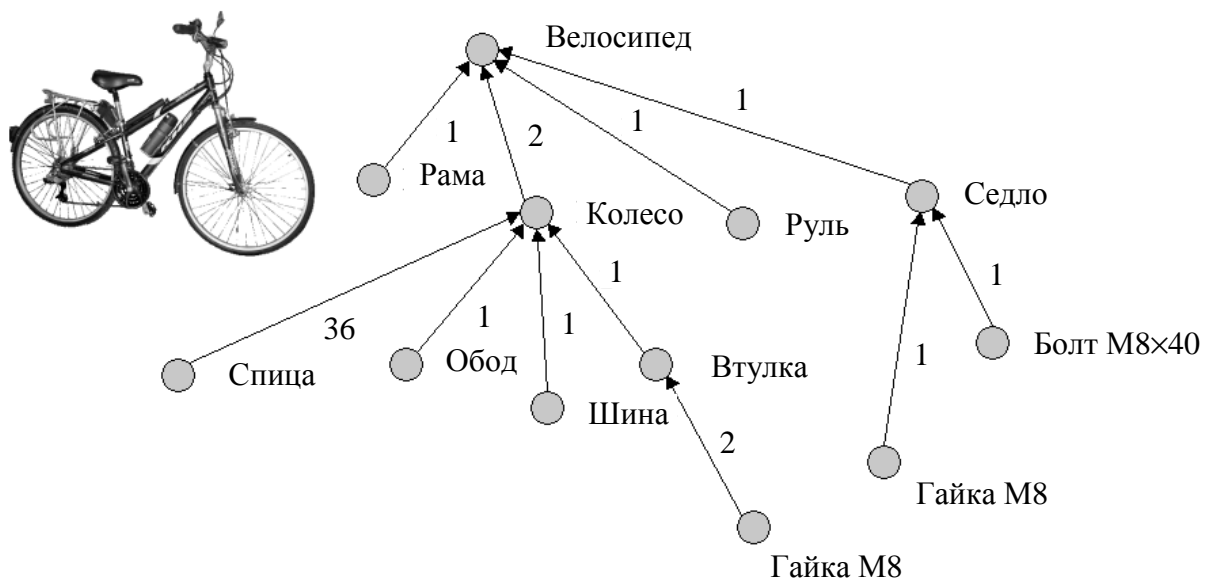


Рис. 5.19

В данном примере комплектация велосипеда представлена *деревом*, узлы которого представляют собой различные детали или комплектующие, а дуги – вхождение одних элементов комплектации в другие. Числа рядом с дугами (коэффициент комплектации) определяют количество деталей, требуемое для комплектации узла. Требуется рассчитать количество комплектующих для партии из 100 велосипедов.

Для решения данной задачи можно применить рекурсивный алгоритм *обхода дерева*. Начиная с корня дерева определяется количество входящих в

него узлов как произведение размера партии на коэффициент комплектации для данного узла. Процедура повторяется для каждого из узлов, пока не будут рассчитаны все. Поскольку в дереве каждый узел связан только с одним вышестоящим узлом, то алгоритм «обходит» дерево, посещая каждый узел один раз. Далее приведен пример программы, реализующей рекурсивный алгоритм обхода дерева.

```
typedef struct
{
    short code;
    float quant;
} CHILD;

typedef struct
{
    short code;
    char name[32];
    CHILD childs[8];
    int nchild;
} PROD;

PROD prod[] =
{
    { 1, "Велосипед", {{2, 1}, {3, 2}, {4, 1}, {5, 1}}, 4},
    { 2, "Рама", {}, 0},
    { 3, "Колесо", {{6, 36}, {7, 1}, {8, 1}, {9, 1}}, 4},
    { 4, "Руль", {}, 0},
    { 5, "Седло", {{10, 1}, {11, 1}}, 2},
    { 6, "Спица", {}, 0},
    { 7, "Обод", {}, 0},
    { 8, "Шина", {}, 0},
    { 9, "Втулка", {{10, 2}}, 1},
    {10, "Гайка М8", {}, 0},
    {11, "Болт М8х40", {}, 0}
};

void treenode(short Code, float Quant)
{
    int index = BinarySearch(prod, Code);
    PROD* p = &prod[index];
    printf("%s : %d\n", p->name, Quant);

    for(int i = 0; i < p->nchild; i++)
        treenode(p->childs[i].code, p->childs[i].quant*Quant);
}
```



```
void main()
{
    treenode(1, 100);
}
```

В программе определены структуры PROD – для описания узлов дерева (конечного изделия, деталей и комплектующих), и CHILD – для организации встроеного в структуру PROD массива входящих узлов childs. Число входящих узлов задается полем nchild. Массив prod содержит данные о комплектации велосипеда. Расчет комплектации для узла с заданным кодом выполняется в рекурсивной функции treenode, параметрами которой служат код узла Code и требуемое количество Quant.

В функции treenode для заданного кода с помощью двоичного поиска определяется индекс узла в массиве prod. Для удобства дальнейших действий вводится указатель p на элемент массива prod. С помощью функции printf выводится информация об узле и количестве деталей в нем. Затем в цикле for по числу входящих узлов функция treenode рекурсивно вызывает саму себя, передавая код входящего узла и его количество.

В головной программе main вызывается функция treenode для корня дерева (код 1) и размера партии 100. Результаты работы программы приведены ниже:

```
Велосипед   : 100
Рама         : 100
Колесо       : 200
Спица        : 7200
Обод         : 200
Шина         : 200
Втулка       : 200
Гайка М8     : 400
Руль         : 100
Седло        : 100
Гайка М8     : 100
Болт М8х40   : 100
```

Представление комплектации продукции в виде дерева позволяет достаточно просто решать задачу расчета ресурсов. Вместе с тем, такое представление имеет и свои недостатки. Как видно из результатов работы программы, потребность в гайках М8 представлена двумя строчками, поскольку в дереве имеется два отдельных узла для гаек втулки и гайки седла. С точки зрения решаемой задачи, было бы правильнее иметь для гаек М8 один узел, но с двумя входящими дугами – от втулки и от седла. Такая структура, в которой узлы могут иметь много входящих дуг, называется *графом* (рис. 5.20).

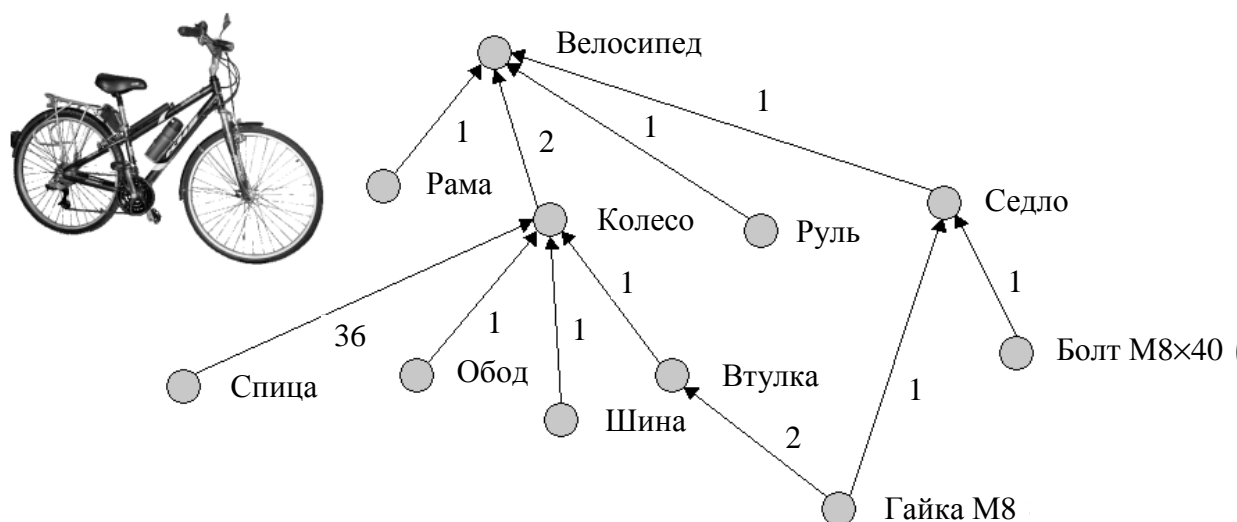


Рис. 5.20

В принципе, для обхода графа можно воспользоваться рассмотренным ранее алгоритмом обхода дерева, при этом результат работы программы будет аналогичным, поскольку в таком случае узел «Гайка М8» будет посещаться дважды – по дугам от втулки и от седла. Если представить расчет комплектации более сложной технической продукции, включающей десятки тысяч узлов и большое число пересекающихся связей (например, современный автомобиль), то более интересным будет алгоритм, который рассчитывает каждый узел только один раз после определения суммарной потребности в данном ресурсе.

Рассматриваемый далее алгоритм *обхода графа* делает два прохода. На первом выполняется «разметка» графа, в ходе которой для каждого узла подсчитывается число входящих дуг. Это нужно для того, чтобы на втором проходе знать, которое из посещений данного узла последнее, и тогда можно двигаться «вглубь», уже зная суммарную потребность, накапливаемую на всех предыдущих посещениях узла.

```
typedef struct {
    short code;
    char name[32];
    CHILD childs[8];
    int nchild;
    int count;
    float quant;
} PROD;
```

```
void graphmark(short Code)
```

```

{
    int index = BinarySearch(prod, Code);
    PROD* p = &prod[index];

    if(++p->count > 1) return;

    for(int i = 0; i < p->nchild; i++)
        graphmark(p->childs[i].code);
}

void graphcalc(short Code, float Quant)
{
    int index = BinarySearch(prod, Code);
    PROD* p = &prod[index];
    p->quant += Quant;

    if(--p->count > 0) return;

    printf("%s %d\n", p->name, Quant);

    for(int i = 0; i < p->nchild; i++)
        graphcalc(p->childs[i].code, p->childs[i].quant*p->quant);
}

void main()
{
    for(int i = 0; i < NPROD; i++)
    {
        prod[i].count = 0;
        prod[i].quant = 0;
    }

    graphmark(1);
    graphcalc(1, 100);
}

```

В программной реализации алгоритма обхода графа в структуру PROD добавлены поля count и quant для подсчета числа вхождений в узел и суммарной потребности в данном ресурсе соответственно. Для разметки используется рекурсивная функция graphmark. В ней при каждом вхождении в узел счетчик count увеличивается на 1. Дальнейший рекурсивный вызов graphmark выполняется только при первом посещении узла.

Собственно расчет производится в рекурсивной функции graphcalc, в которую передается код узла и требуемое количество. Это количество суммируется с полем quant, и счетчик count уменьшается на 1. Дальнейший рекурсивный вызов graphcalc выполняется при последнем посещении узла.

В головной программе main в цикле for инициализируются поля count и quant, вызывается функция разметки для корня графа, а также функция расчета. Результат работы программы приведен ниже:

Велосипед	:	100
Рама	:	100
Колесо	:	200
Спица	:	7200
Обод	:	200
Шина	:	200
Втулка	:	200
Руль	:	100
Седло	:	100
Гайка M8	:	500
Болт M8x40	:	100

5.10. Задачи на связность

Задачи на связность встречаются во многих областях человеческой деятельности. В качестве примера можно привести производство широко применяемых в электронике печатных плат. Печатная плата (рис. 5.21) представляет собой стеклотекстолитовую пластину с металлизированными отверстиями, в которые потом устанавливаются электронные компоненты. В соответствии со схемой отверстия соединяются проводниками.

При проведении контроля качества продукции проверяется наличие или отсутствие соединения между различными точками. Формально плату

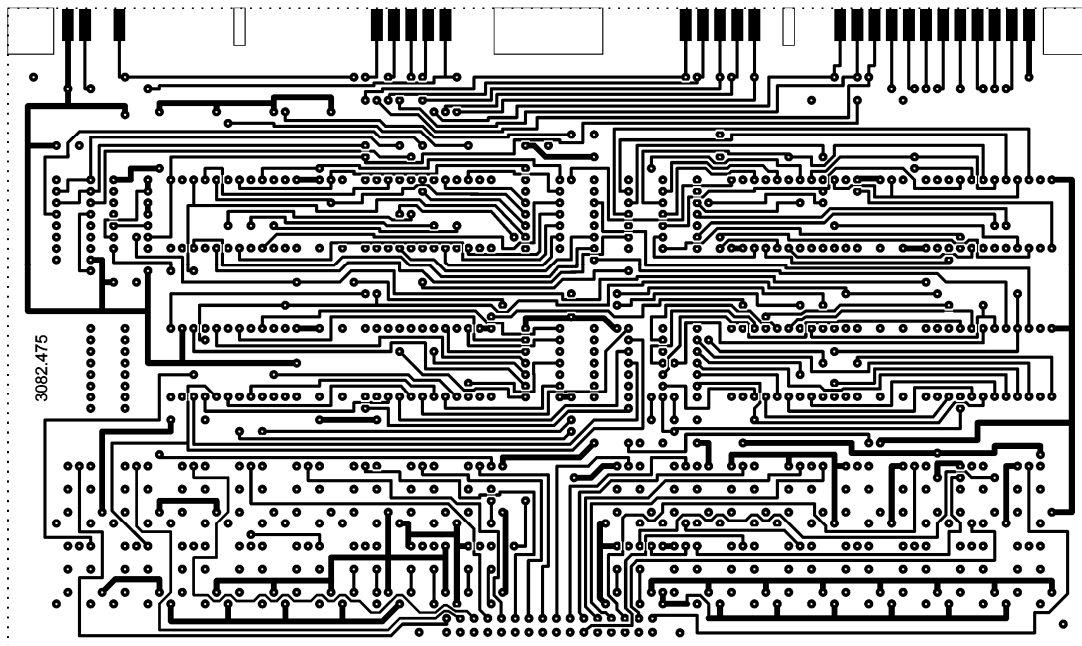










Рис. 5.21

i	j							
	1	2	3	4	5	6	7	8
1		1				1		
2	1							
3								1
4							1	
5							1	
6	1							
7				1	1			
8			1					

Алгоритм Уоршалла позволяет для исходной матрицы C построить матрицу транзитивных замыканий T , в которой единицы будут означать наличие замыкания между соответствующими узлами. Сначала алгоритм копирует матрицу C в T . Затем для каждой пары i и j перебираются оставшиеся узлы, и если находится узел k , для которого $T_{ik} = 1$ и $T_{kj} = 1$, то T_{ij} устанавливается в 1. Пример программной реализации алгоритма приведен далее:

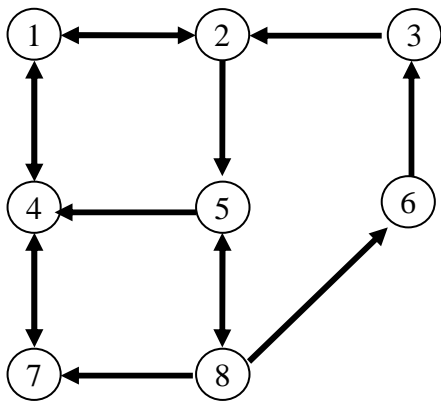
```
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        T[i][j] = C[i][j];

for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        if(j != i)
            for(int k = 0; k < n; k++)
                if(k != j)
                    T[j][k] = T[j][k] || T[j][i] && T[i][k];
```

125

<i>i</i>	<i>j</i>							
	1	2	3	4	5	6	7	8
1		1				1		
2	1					1		
3								1
4					1		1	
5				1			1	
6	1	1						
7				1	1			
8			1					

Рис. 5.23



<i>i</i>	<i>j</i>							
	1	2	3	4	5	6	7	8
1		1		1				
2	1				1			
3		1						
4	1						1	
5				1				1
6			1					
7				1				
8					1	1	1	

Рис. 5.24

Для тестирования соединения между двумя узлами можно написать следующую функцию, которая получает номера узлов и возвращает значение true, если между узлами существует соединение:

```
bool TestConnection(int Node1, int Node2)
{
    return(T[Node1-1][Node2-1] == 1);
}
```

Другой пример задачи на связность взят из области транспортных сообщений. Транспортная сеть представлена ориентированным графом, в котором узлы соответствуют населенным пунктам, точкам пересечения улиц или дорог, а дуги – дорогам, связывающим узлы. При этом движение между уз-

<i>i</i>	<i>j</i>							
	1	2	3	4	5	6	7	8
1		1	1	1	1	1	1	1
2	1		1	1	1	1	1	1
3	1	1		1	1	1	1	1
4	1	1	1		1	1	1	1
5	1	1	1	1		1	1	1
6	1	1	1	1	1		1	1
7	1	1	1	1	1	1		1
8	1	1	1	1	1	1	1	

Рис. 5.25

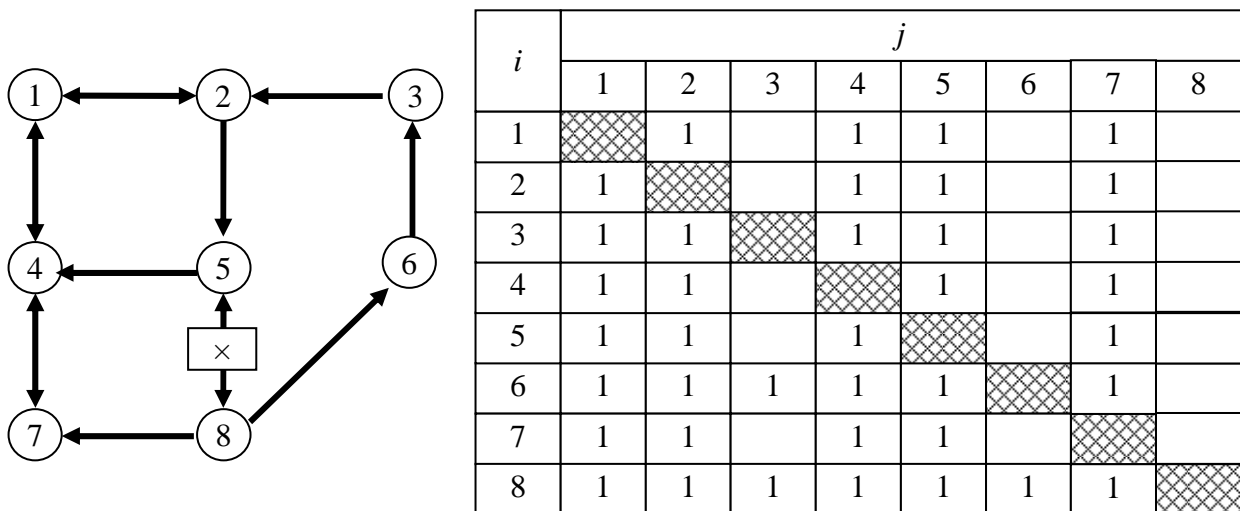


Рис. 5.26

лами может быть односторонним. Требуется проверить возможность путешествия из пункта А в пункт В.

Для решения данной задачи также может быть применен алгоритм Уоршалла. Пример графа транспортной сети и соответствующей матрицы показан на рис. 5.24. Следует отметить, что матрица C – несимметричная.

В результате работы алгоритма формируется матрица транзитивных замыканий, показанная на рис. 5.25. Поскольку все элементы этой матрицы являются единицами, то это означает, что между всеми узлами транспортной сети сообщение возможно.

Внесем изменение в транспортную сеть. Разорвем связь между узлами 5 и 8 (рис. 5.26). В результате этого в матрице транзитивных замыканий исчезает много единиц в столбцах 3, 6 и 8, что говорит о недостижимости этих узлов из узлов 1, 2, 4, 5 и 7.

5.11. Задачи на поиск минимального пути

Этот класс задач широко встречается как в транспортной, так и в производственной логистике. По постановке он похож на задачи на связность. Здесь также имеется транспортная сеть, представленная ориентированным графом. Дополнительная информация здесь – расстояние между узлами, заданное для дуг графа. Требуется найти минимальный путь из пункта А в пункт В (рис. 5.27).

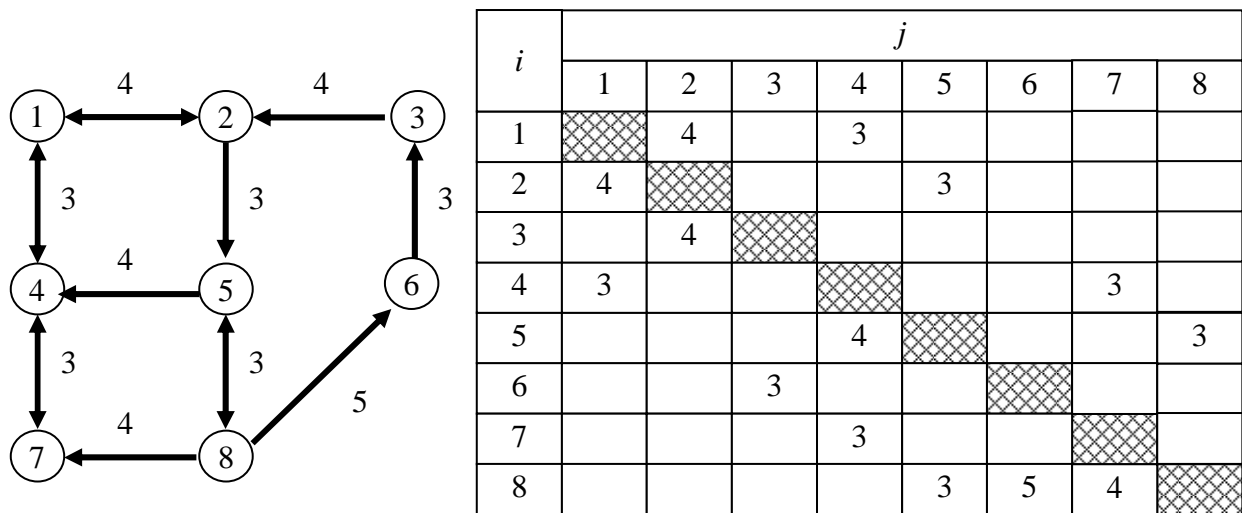


Рис. 5.27

Алгоритм Флойда представляет собой развитие алгоритма Уоршалла. Исходная матрица в этом случае содержит расстояния между соответствующими узлами (рис. 5.27). Пустые ячейки этой матрицы, означающие отсутствие пути, можно рассматривать как бесконечные расстояния.

В приведенной далее программе из исходной матрицы C формируются две матрицы – расстояний T и путей H . Элементы матрицы T_{ij} показывают минимальное расстояние из i в j , а элементы матрицы H_{ij} – из какого узла в j приходит минимальный путь. В программе вместо бесконечности используется заведомо большое значение INF.

```
const int INF = 1000000;

int C[8][8] = { {INF, 4, INF, 3, INF, INF, INF, INF},
                { 4, INF, INF, INF, 3, INF, INF, INF},
                {INF, 4, INF, INF, INF, INF, INF, INF}, ... };

int T[8][8];
int H[8][8];
int n = 8;
```



```

for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
    {
        T[i][j] = C[i][j];
        if(C[i][j] == INF)
            H[i][j] = -1;
        else
            H[i][j] = j;
    }

for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        if(j != i)
            for(int k = 0; k < n; k++)
                if(k != j)
                    if(T[j][i]+T[i][k] < T[j][k])
                    {
                        H[j][k] = H[j][i];
                        T[j][k] = T[j][i]+T[i][k];
                    }

```

Сначала в программе формируются матрицы T и H . Затем для каждой пары i и j перебираются оставшиеся узлы, и если находится узел k , для которого $T_{ji} + T_{ik} < T_{jk}$, то в T_{jk} сохраняется это минимальное значение. При этом в матрице H сохраняется номер узла i . Получившиеся в результате работы алгоритма матрицы T и H показаны на рис. 5.28 и 5.29 соответственно.









i	j							
	1	2	3	4	5	6	7	8
1		4	18	3	7	15	6	10
2	4		14	7	3	11	10	6
3	8	4		11	7	15	14	0
4	3	7	21		10	18	3	13
5	7	11	11	4		8	7	3
6	11	7	3	14	10		17	13
7	6	10	24	3	13	21		16
8	10	12	8	7	3	5	4	

Рис. 5.28

Для получения расстояния из A в B применяется функция Distance:

```

int Distance(int A, int B)
{
    return(T[A-1][B-1]);
}

```

<i>i</i>	<i>j</i>							
	1	2	3	4	5	6	7	8
1		1	6	1	2	8	4	5
2	2		6	5	2	8	8	5
3	2	3		5	2	8	8	5
4	4	1	6		2	8	4	5
5	4	1	6	5		8	8	5
6	2	3	6	5	2		8	5
7	4	1	6	7	2	8		5
8	4	3	6	7	8	8	8	

Рис. 5.29

Для получения пути можно воспользоваться функцией Path, которой передаются номера начального и конечного узлов (А и В), а также указатель Р на массив номеров узлов пути и переменная N по ссылке. В этой переменной будет содержаться число узлов минимального пути.

```
void Path(int A, int B, int P[], int &N)
{
    N = 0;
    P[N++] = H[A-1][B-1];

    while(P[N-1] != A)
        P[N++] = H[A-1][P[N-1]];
}
```

Поскольку алгоритмы Уоршалла и Флойда содержат три вложенных цикла, то их временная функция сложности будет n^3 . Достоинством указанных алгоритмов считается то, что однократно рассчитанные матрицы содержат все множество минимальных путей.

Алгоритм Дейкстры позволяет найти минимальные расстояния от заданного узла сети до всех остальных узлов, при этом граф сети не должен иметь отрицательных циклов. Рассмотрим работу алгоритма на примере сети, показанной на рис. 5.29, а.

Каждому узлу ставится в соответствие некоторое значение расстояния. Для стартового узла (в примере выбран узел 8) это 0, для остальных – бесконечность (5.29, б). Начиная со стартового узла определяются расстояния для соседних узлов путем суммирования значений на дугах. После первого шага (5.29, в) такими узлами являются узлы 7, 5 и 6.

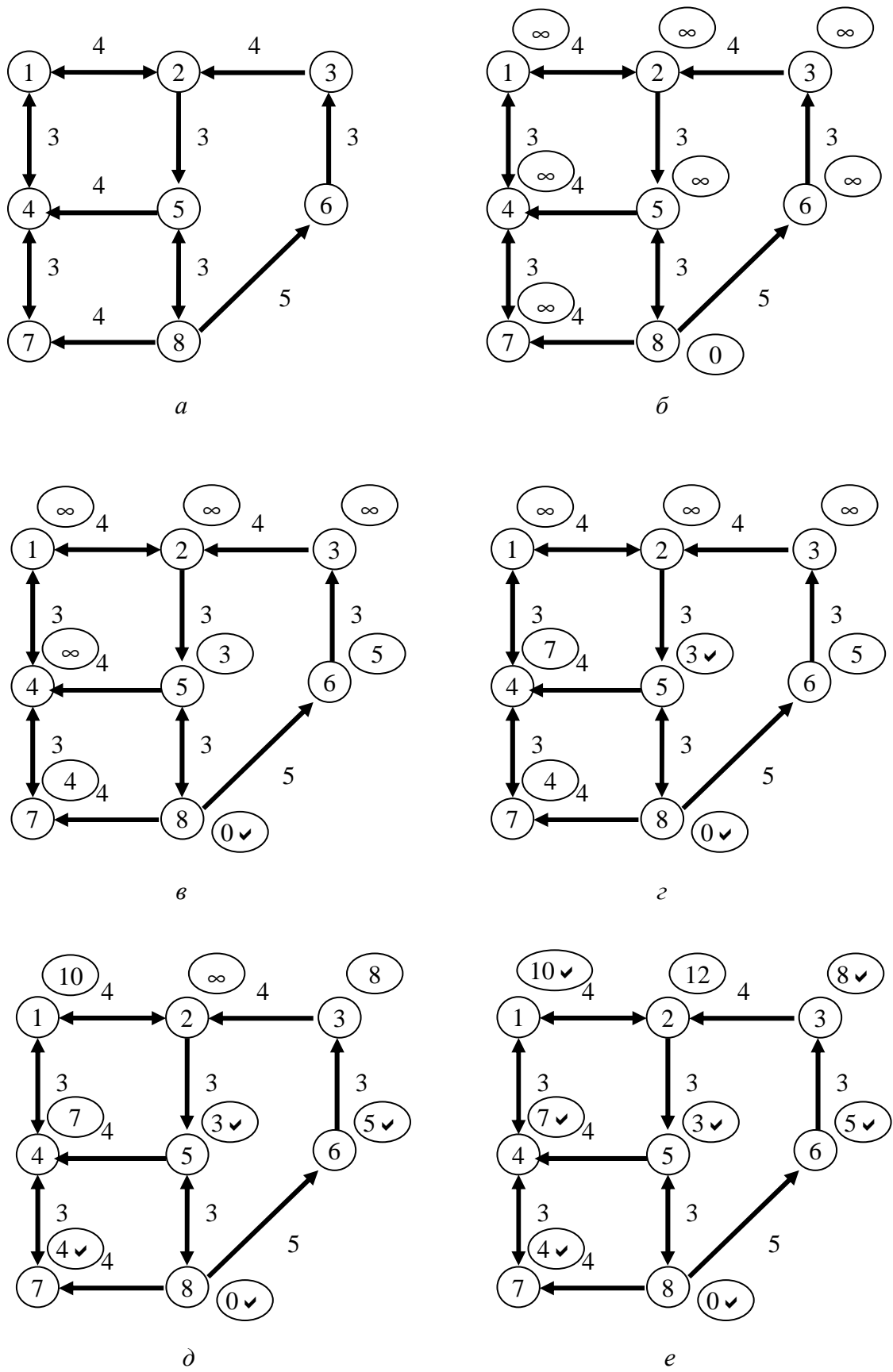


Рис. 5.29

На следующем шаге значение расстояния для исходного узла фиксируется, а из оставшихся узлов выбирается узел с наименьшим значением расстояния. В данном случае это узел 5 с расстоянием 3. Повторение действий от узла 5 дает узел 4 с расстоянием 7 (5.29, з). Расстояние 3 для узла 5 фиксируется. Продолжение выполнения алгоритма приводит к узлу 1 через узел 4 и к узлу 3 через узел 6 (5.29, д), расстояния в узлах 4 и 6 фиксируются. В последнюю очередь алгоритм доходит до узла 2 с расстоянием 12 (5.29, е). На этом работа алгоритма завершается и во всех узлах известны минимальные расстояния от стартовой точки.

Следует отметить, что для нахождения минимальных расстояний от одного узла алгоритм Дейкстры работает быстрее алгоритма Флойда.

5.12. Задачи комбинаторной оптимизации

На практике довольно часто встречаются задачи, когда требуется выбрать один из множества вариантов решения, который наилучшим образом отвечает поставленным условиям. Такие задачи называются задачами комбинаторной оптимизации. В качестве примера можно привести две характерные для производственных систем задачи.

Пример 1. В производстве порошковой краски на одной технологической линии выпускаются краски разных цветов. При смене цвета краски требуется очистка оборудования от оставшегося внутри него порошка. Время очистки зависит от того, с какой краски на какую осуществляется переход. Для заданной производственной программы (набора цветов) требуется выбрать такую последовательность смены цвета, которая давала бы минимальную длительность производственного цикла (рис. 5.30).

Пример 2. Координатно-пробивной пресс позволяет пробивать в листе металла отверстия разной формы. Лист металла лежит на столе и перемещается вместе с ним. Форма отверстия определяется формой инструмента – пуансона. Одновременно в машину могут быть установлены несколько разных пуансонов, один из которых находится на рабочей позиции, остальные – в специальном поворотном магазине. Для перемещения инструмента на рабочую позицию требуется время. Таким образом, время производственного цикла включает время перемещения листа и время смены инструмента. Требуется выбрать такую последовательность пробивки отверстий, которая давала бы минимальную длительность производственного цикла (рис. 5.31).



План производства на 15.07.2010		
Код	Цвет	Вес, кг
712345	RAL1000	4.800
716214	RAL7005	2.600
785023	RAL3010	6.200
725713	RAL4020	1.600
746515	RAL9000	8.800
746525	RAL9001	5.250
715324	RAL1003	3.400

Рис. 5.30

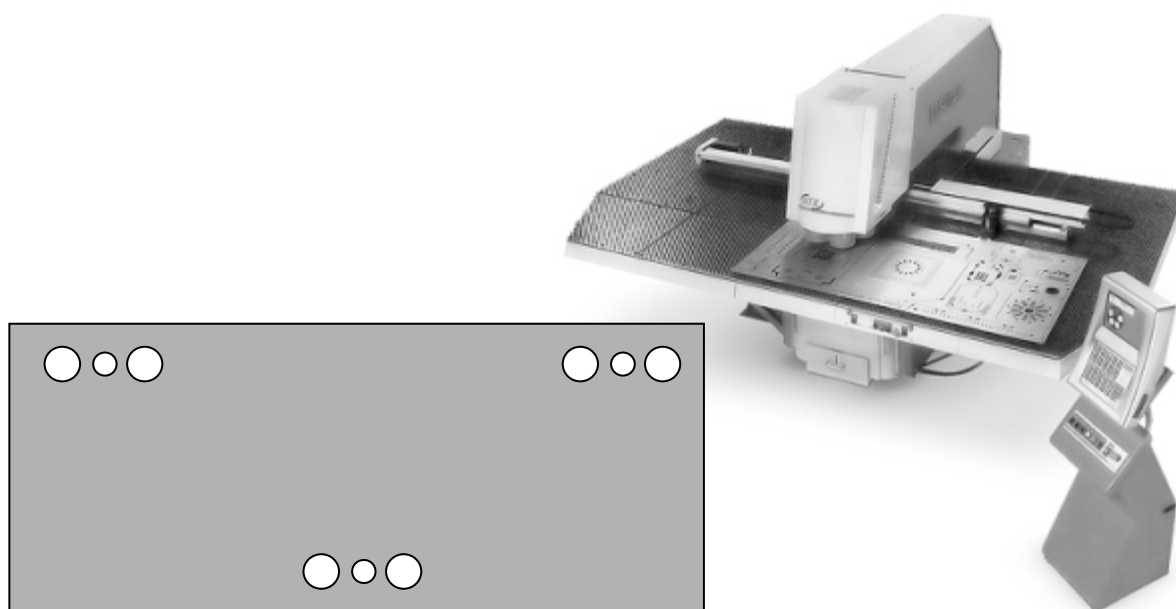
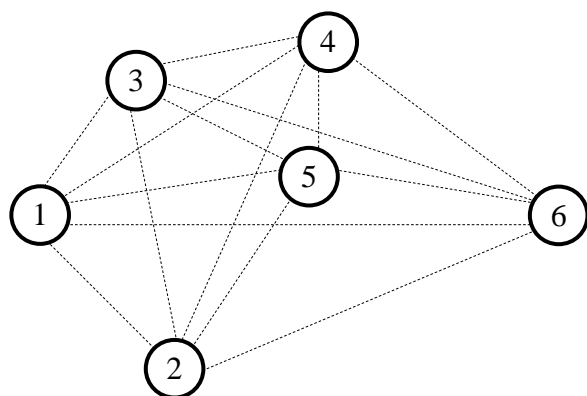


Рис. 5.31

Несмотря на разную постановку обе рассмотренные задачи представляют собой хорошо известную в математике *задачу коммивояжера*. Эта задача формулируется следующим образом. Имеется n городов, расстояния между которыми известны. Требуется посетить все города, побывав в каждом не более одного раза, в такой последовательности, чтобы длина пути была минимальна.

Для формализации данных задачи представим их в виде графа, узлы которого соответствуют городам, а дуги – путям, соединяющим города (рис. 5.32). Расстояния можно описать матрицей C размером $n \times n$, элементы



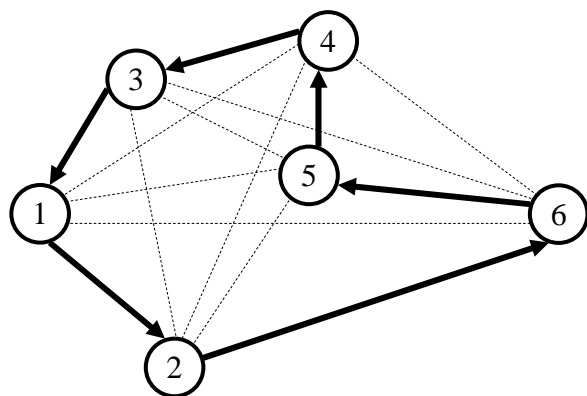
i	j					
	1	2	3	4	5	6
1		6	4	8	7	14
2	6		7	11	7	10
3	4	7		4	3	10
4	8	11	4		5	11
5	7	7	3	5		7
6	14	10	10	11	7	

Рис. 5.32

которой C_{ij} представляют длину пути из i в j . В случае $C_{ij} = C_{ji}$ задача называется симметричной, в противном случае – несимметричной.

Для решения задачи коммивояжера придумано много алгоритмов, отличающихся скоростью работы и качеством решения. Рассмотрим три из них.

Алгоритм перебора «грубой силой» (Brute force enumeration) основан на переборе всех возможных вариантов. Понятно, что такой алгоритм гарантированно дает наилучшее решение, но время работы уже при $n > 15$ быстро становится неприемлемым, поскольку временная функция сложности алгоритма определяется как $(n - 1)!$. Наилучший путь для примера на рис. 5.32, а также число вариантов в несимметричной задаче для различных значений n приведены на рис. 5.33 (решение в таблице выделено рамками).



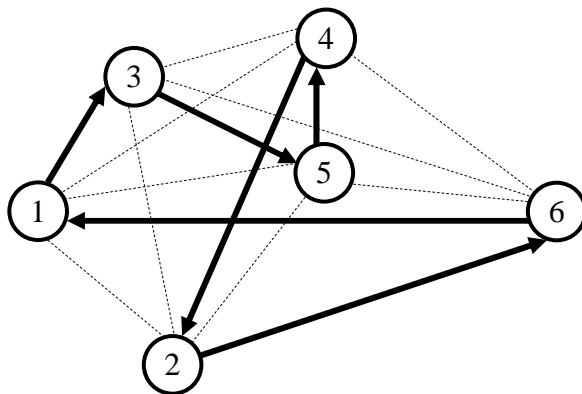
i	j					
	1	2	3	4	5	6
1		6	4	8	7	14
2	6		7	11	7	10
3	4	7		4	3	10
4	8	11	4		5	11
5	7	7	3	5		7
6	14	10	10	11	7	

Решение: 1-2-6-5-4-3-1, длина пути: 36

Число вариантов в несимметричной задаче $(n - 1)!$									
5!	10!	15!	20!	25!	30!	35!	40!	45!	50!
$\sim 10^2$	$\sim 10^6$	$\sim 10^{12}$	$\sim 10^{18}$	$\sim 10^{25}$	$\sim 10^{32}$	$\sim 10^{40}$	$\sim 10^{47}$	$\sim 10^{56}$	$\sim 10^{64}$

Рис. 5.33

Алгоритм ближайшего соседа представляет другую крайность, поскольку работает быстро, но дает неоптимальный результат, а при некотором наборе входных данных может даже выбрать наихудший вариант. Алгоритм заключается в выборе на каждом шаге наиболее короткого расстояния из всех возможных. На первых шагах кажется, что решение будет оптимальным, но на последних шагах приходится расплачиваться за «жадность» (рис. 5.34).



i	j					
	1	2	3	4	5	6
1		6	4	8	7	14
2	6		7	11	7	10
3	4	7		4	3	10
4	8	11	4		5	11
5	7	7	3	5		7
6	14	10	10	11	7	

Решение: 1-3-5-4-2-6-1, длина пути: 47

Рис. 5.34

Метод ветвей и границ (Branch and bound) обладает сочетанием высокого качества решения и хорошего быстродействия. Он основан на разделении всего множества решений на подмножества (ветвлении) и оценке верхних и нижних границ оптимальных решений для каждого подмножества.

Чтобы лучше понять процедуру оценки, выделим из расстояний каждой строки матрицы постоянную составляющую (столбец справа на рис. 5.35). Это называется приведением матрицы по строкам. Физический смысл полученного столбца заключается в том, что каким путем ни выезжать из города i , в любом случае требуется проехать расстояние не менее выделенной постоянной составляющей. После приведения по строкам можно аналогично привести матрицу по столбцам. Полученная строка дает минимальные расстояния, которые требуется проехать, въезжая в город j . Поскольку въезд и выезд для каждого города выполняется только один раз, то просуммировав полученные строку и столбец, можно получить минимальную оценку пути для данной матрицы.

Следующим шагом алгоритма является оценка нулей. Поскольку после приведения в каждой строке и столбце матрицы должен быть хотя бы один ноль, то эти пути представляются как наиболее перспективные. Для них делается оценка отказа от данного пути.

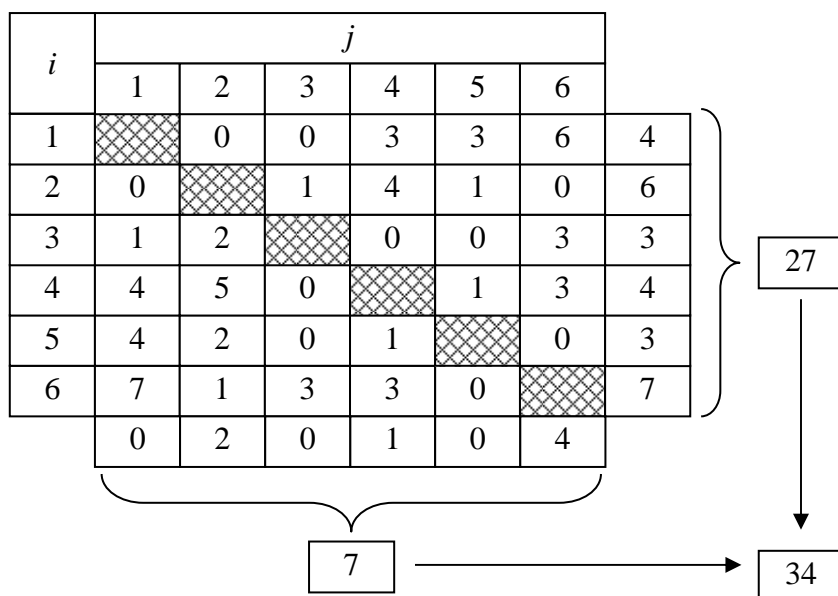


Рис. 5.35

Процедура оценки нулей показана на рис. 5.36. Для каждого из нулей считается сумма минимальных значений в оставшихся частях строки и столбца. Так, для пути 1–2 (рис. 5.36, *а*) минимальное значение в строке равно 0 (путь 1–3), а в столбце – 1 (путь 6–2), соответственно сумма равна 1. Это означает, что отказ от пути 1–2 увеличит общий путь как минимум на 1. На рис. 5.36, *б* верхними индексами показаны оценки всех нулей.

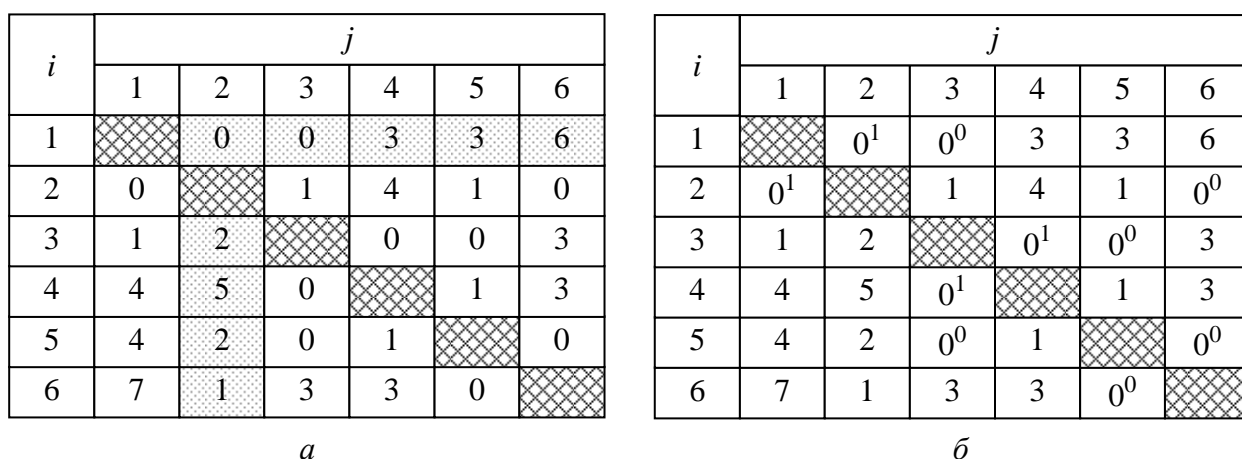


Рис. 5.27

После оценки нулей производится ветвление. Для этого выбирается первый из путей с максимальной оценкой (в данном примере путь 1–2). Все множество решений делится на два класса путей – включающие путь 1–2 и остальные (рис. 5.37). Для класса 1, 2 в матрице убираются строка 1 и столбец 2, а также запрещается путь 2–1, который преждевременно приводил бы в город 1. После этого делается приведение матрицы по строкам и столбцам,

которое добавляет сумму полученных минимальных значений к оценке пути (в данном примере – 1). Для класса $\overline{1,2}$ оценка пути увеличивается на оценку нуля. Таким образом, все множество вариантов с оценкой 34 делится на классы 1, 2 и $\overline{1,2}$ с оценками 35 и 35.

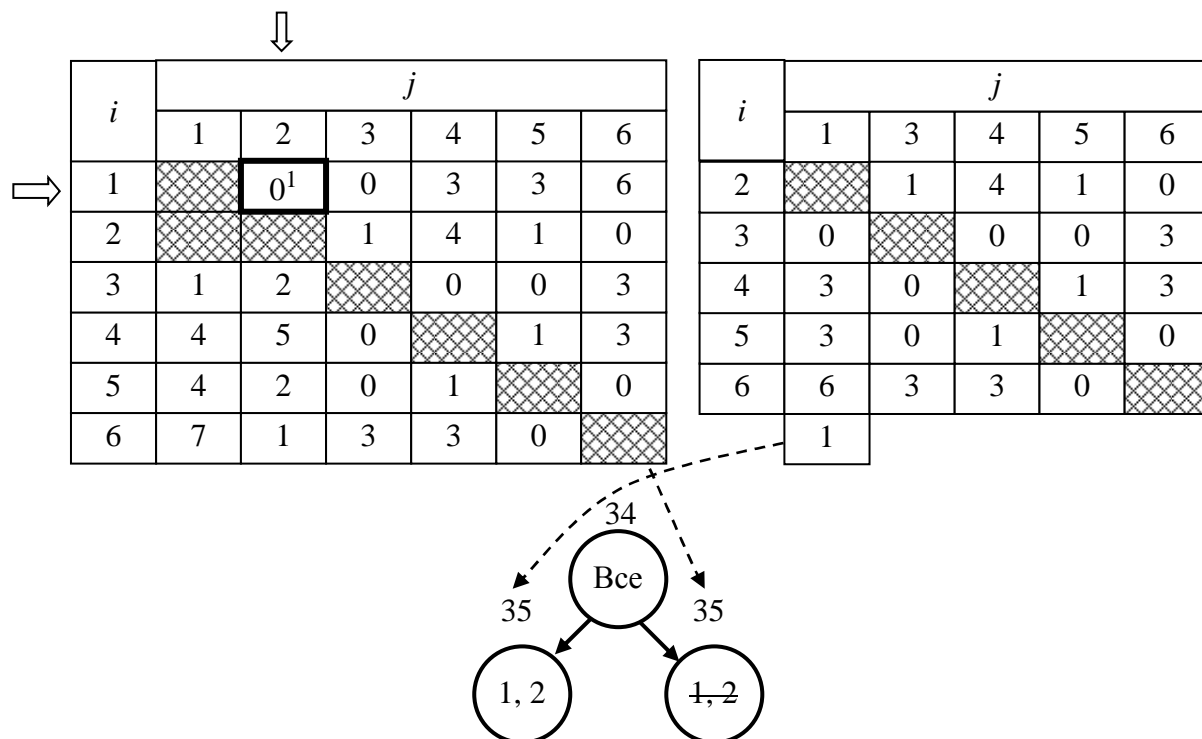


Рис. 5.37

Продолжая ветвь 1, 2, выполняют оценку нулей и ветвление на множества вариантов 3, 1 и $\overline{3,1}$ (рис. 5.38). Путь 3–1 по результатам приведения матрицы добавит к минимальному пути 1, а отказ от данного пути добавит по оценке нуля 3. Таким образом, на дереве решений образуются ветви 3, 1 и $\overline{3,1}$ с оценками 36 и 38 соответственно.

Продолжение ветви 3, 1 показано на рис. 5.39, в ней есть ветвление на классы 6, 5 и $\overline{6,5}$, а затем на классы 2, 6 и $\overline{2,6}$. Дальнейший путь дает безальтернативные варианты 5, 4 и 4, 3, поскольку в матрице не остается выбора. Ветвление закончено и получен путь длиной 36, который на данный момент считается наилучшим.

Теперь остается проверить альтернативные варианты. Единственным перспективным вариантом оказывается класс $\overline{1,2}$, который пока имеет оценку 35 – ниже минимальной. Если исключить путь 1–2 и сделать оценку нулей, то самая высокая оценка 3 окажется у пути 1–3, по которому будет осуществляться ветвление (рис. 5.40). После приведения матрицы оценка пути 1–3 станет 36, что не лучше полученного пути. На этом поиск можно закончить.

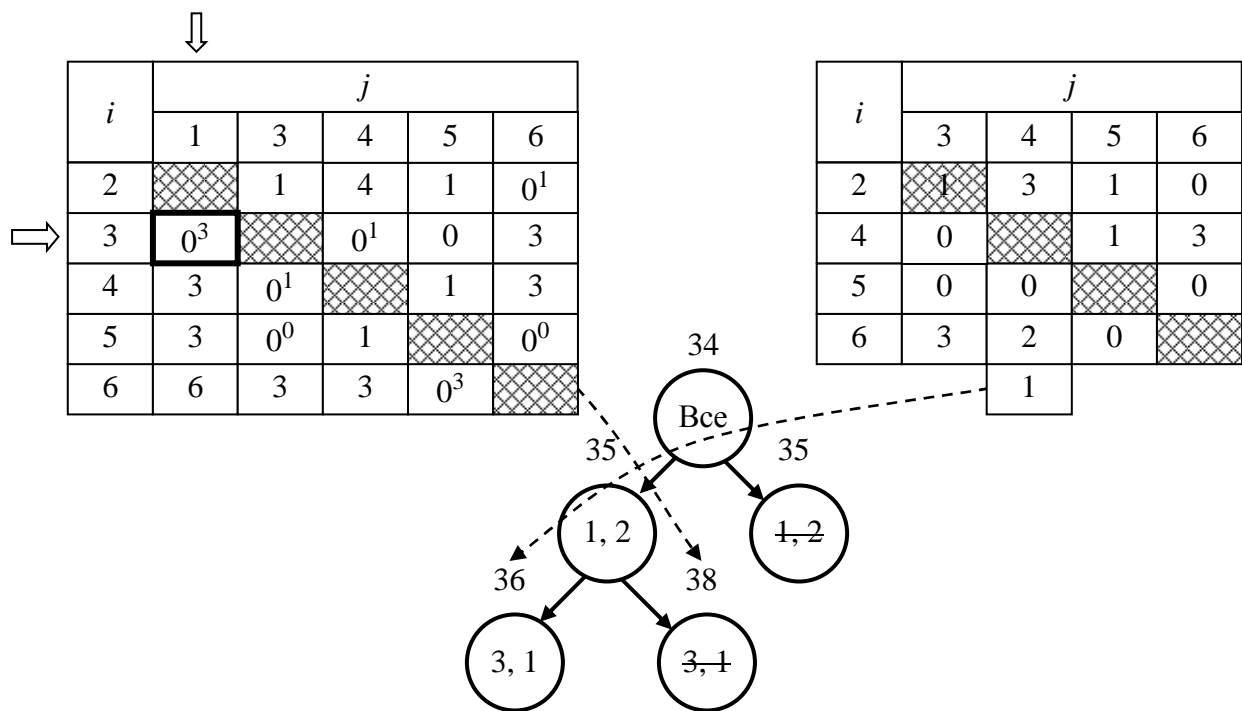


Рис. 5.38

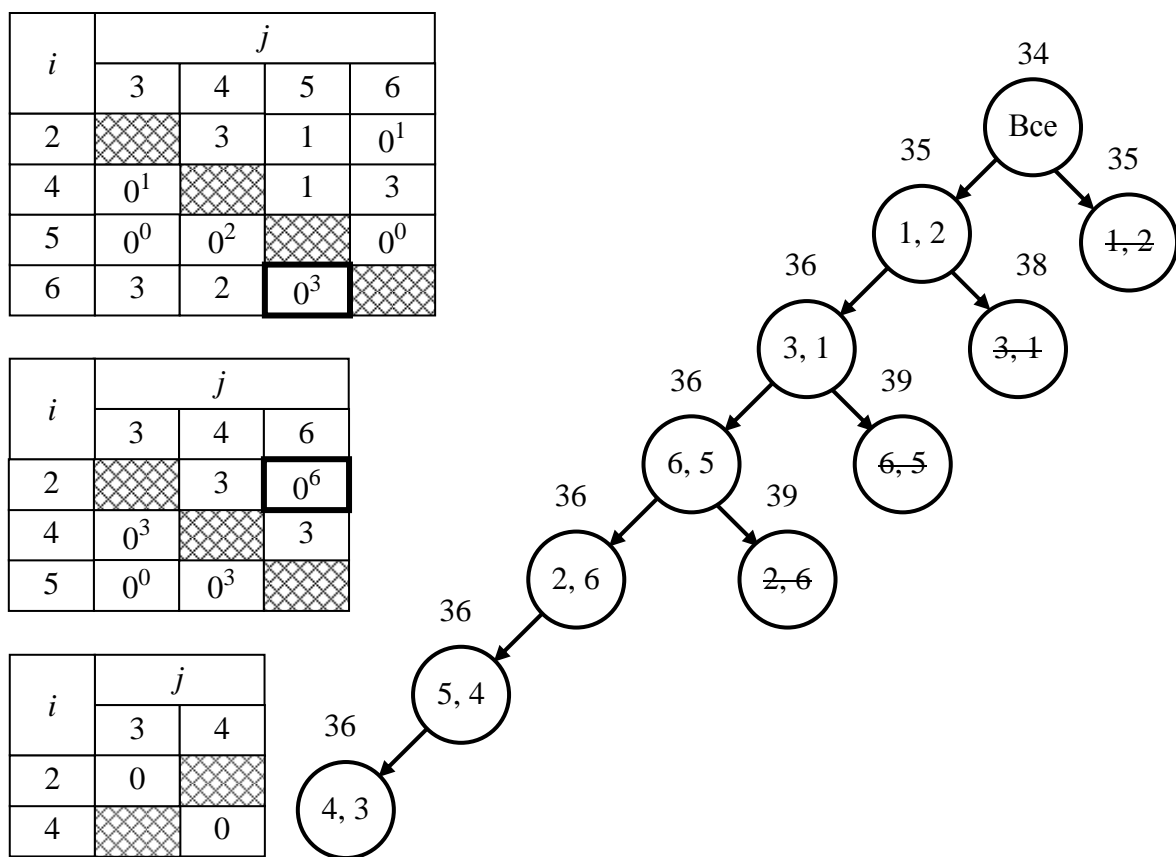


Рис. 5.39

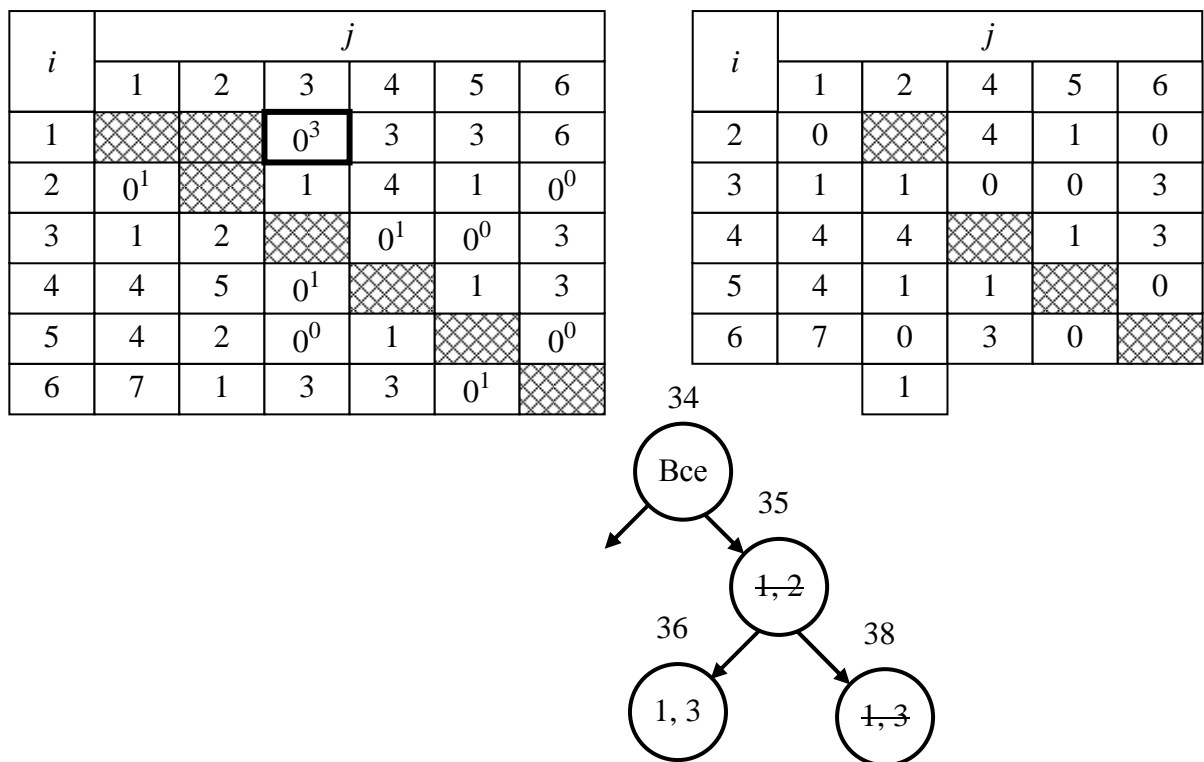


Рис. 5.40

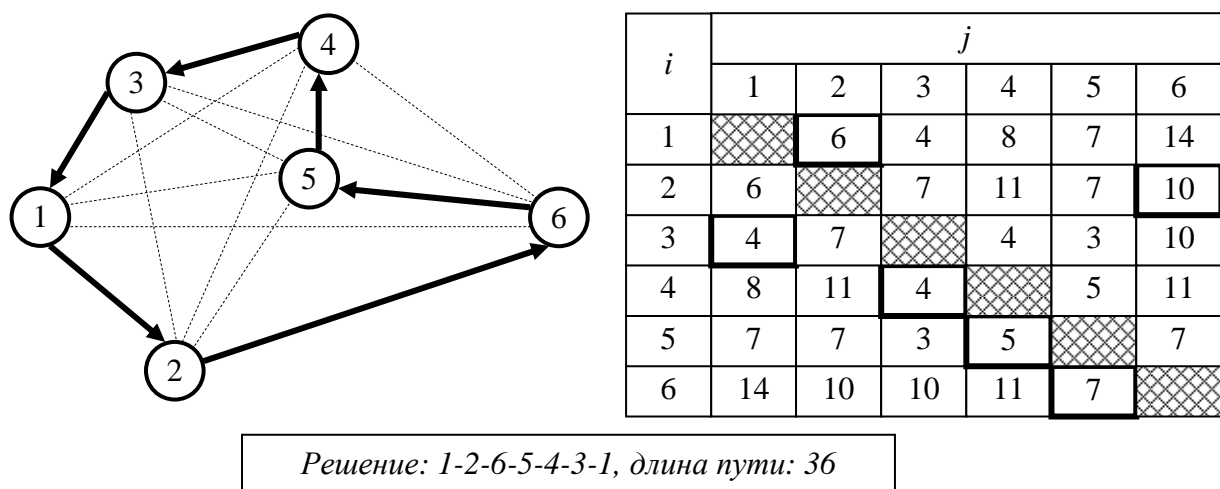


Рис. 5.41

В рассматриваемом примере метод ветвей и границ позволил получить результат (рис. 5.41), аналогичный методу перебора грубой силой. Программная реализация метода ветвей и границ более сложна, чем реализация двух рассмотренных ранее алгоритмов комбинаторной оптимизации. Тем не менее, получение оптимального решения при приемлемой временной функции сложности делает этот алгоритм предпочтительным для задачи коммивояжера с числом городов более 10.

Заключение

В настоящем пособии рассматривались основы создания программ на языках С и С++, структурный и объектно-ориентированный подходы к программированию, а также типовые алгоритмы решения ряда классических задач. Это может быть хорошим фундаментом для продолжения изучения средств компьютерной техники и создания современного программного обеспечения. В заключение хочется отметить ряд не вошедших в пособие вопросов, которые могут быть предметом дальнейшего изучения.

Создание современных программных комплексов требует не только качественного написания отдельных программ, но и организации их взаимодействия с использованием возможностей, предоставляемых операционными системами и сетевыми технологиями. Операционные системы имеют в своем составе различные механизмы, предназначенные для поддержания межпроцессного взаимодействия. К таким механизмам относятся сигналы, трубки, очереди сообщений, разделяемая память, семафоры, удаленный вызов процедур. Сетевые протоколы позволяют организовать взаимодействие между программами, выполняемыми на разных компьютерах, так, как если бы они находились на одной машине.

Эффективное использование современных многопроцессорных архитектур требует новых подходов в распараллеливании выполнения программ. Технологии многопоточного программирования позволяют получить выигрыш в быстродействии, но требуют решения ряда вопросов синхронизации и управления потоками, распределения ресурсов внутри программы.

Важную задачу при разработке интерактивных программ представляет проектирование пользовательских интерфейсов. Современные технические и программные средства позволяют создавать среду для эффективного человеко-машинного общения. Оконный графический интерфейс пользователя с элементами мультимедиа стал уже обыденным явлением, и в ближайшее время следует ожидать массированного появления устройств речевого ввода-вывода. Устройства тактильного ввода и вывода объемных изображений также могут повлиять на организацию диалога пользователя с компьютером.

Большое значение при разработке программного обеспечения информационных систем имеет взаимодействие программ с базами данных. Базы данных служат ключевым звеном современных информационных и управляющих систем, позволяя организовать надежное хранилище взаимосвязанных

структурированных данных. За прошедшие с момента появления первых баз данных четыре десятка лет созданы различные модели данных, большое число систем управления базами данных, различные языки запросов и способы организации интерфейса программ с базами данных. Более подробную информацию об организации баз данных в производственных системах можно получить в [8].

Вопросы алгоритмизации рассматривались в пособии на примере наиболее часто встречающихся задач при разработке программного обеспечения. В силу ограниченного объема пособия в нем не затрагивался ряд классов алгоритмов, знание которых также полезно разработчикам современного программного обеспечения [9]. Отметим кратко некоторые из этих классов.

Алгоритмы компрессии и декомпрессии данных применяются при передаче сообщений по линиям связи, при цифровой обработке изображений и видеосигнала, а также при сжатии данных в целях уменьшения объема хранимой информации. Одним из первых классических алгоритмов сжатия сообщений стал алгоритм Хаффмана [10].

Алгоритмы шифрования и дешифрования относятся к области криптографии и широко применяются в целях защиты информации [11]. Алгоритмы одностороннего кодирования часто используются для проверки паролей и получения контрольных сумм. Симметричные алгоритмы шифрования с открытым или закрытым ключом позволяют кодировать сообщение, а затем точно восстанавливать его.

Вопросы комбинаторной оптимизации рассматривались на примере задачи коммивояжера. В настоящее время для решения этой задачи, особенно актуальной в области транспортной логистики, применяются новые методы, например генетические алгоритмы [12]. Для более сложных задач оптимизации, например в ситуациях, когда условия задачи меняются по ходу ее решения, применяются методы динамического программирования, основу которых положили труды Ричарда Беллмана [13].

Программирование и алгоритмизация – бурно развивающаяся область человеческих знаний, постоянно требующая притока специалистов высокой квалификации. Совершенствуя свои знания в этой области профессиональной деятельности можно стать реально востребованным специалистом.

Список литературы

1. Керниган Б., Ритчи Д. Язык программирования С. 2-е изд. М.: Вильямс, 2011.
2. Шилдт Г. Полный справочник по С++. 4-е изд. М.: Вильямс, 2010.
3. Хьюз Дж., Мичтом Дж. Структурный подход к программированию. М.: Мир, 1980.
4. Страуструп Б. Язык программирования С++: спец. изд. М.: Бином, 2008.
5. Мюссер Д., Дердж Ж., Сейни А. С++ и STL: справ. руководство. 2-е изд. (сер. С++ in Depth). М.: Вильямс, 2010.
6. ГОСТ 19.701–90 Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. М.: Стандартинформ, 2005.
7. Вирт Н. Алгоритмы и структуры данных. М.: Мир, 1989.
8. Калинин А. В., Шевченко А. В. Организация баз данных в интегрированных системах управления процессами производства: учеб. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2010.
9. Кнут Д. Искусство программирования: в 3 т. М.: Вильямс, 2007–2008.
10. Левитин А. В. Гл. 9. Жадные методы: Алгоритм Хаффмана // Алгоритмы: введение в разработку и анализ. М.: Вильямс, 2006.
11. Панасенко С. П. Алгоритмы шифрования: спец. справочник. СПб.: БХВ-Петербург, 2009.
12. Гладков Л. А., Курейчик В. В., Курейчик В. М. Генетические алгоритмы: учеб. пособие. 2-е изд. М.: Физматлит, 2006.
13. Беллман Р. Динамическое программирование. М.: Изд-во иностранной литературы. 1960.

Оглавление

Введение.....	3
Глава 1. АРХИТЕКТУРА КОМПЬЮТЕРНЫХ СИСТЕМ	4
1.1. История создания архитектуры современных компьютеров	4
1.2. Архитектура компьютера.....	5
1.3. Представление данных в двоичной системе счисления.....	5
1.4. Процессор и оперативная память	6
1.5. Создание и выполнение программ	7
Глава 2. ЯЗЫК ПРОГРАММИРОВАНИЯ С.....	10
2.1. Структура С-программы.....	10
2.2. Синтаксис языка С	11
2.3. Простые типы данных	12
2.4. Сложные типы данных	14
2.5. Константы и переменные	15
2.6. Массивы	16
2.7. Размещение данных в памяти. Адреса. Указатели	19
2.8. Операторы.....	23
2.9. Управление программным потоком.....	28
2.10. Функции	32
2.11. Препроцессор.....	34
2.12. Стандартная библиотека С.....	40
2.13. Примеры решения типовых задач программирования на С.....	44
Глава 3. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ.....	50
3.1. Предпосылки структурного программирования.....	50
3.2. Структурный подход к программированию.....	51
Глава 4. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ.....	59
4.1. Объектно-ориентированное программирование и язык С++	59
4.2. Инкапсуляция. Классы	60
4.3. Наследование	67
4.4. Полиморфизм	70
4.5. Перегрузка	75
4.6. Шаблоны	80
4.7. Примеры решения типовых задач программирования на С++	84
Глава 5. ОСНОВЫ АЛГОРИТМИЗАЦИИ.....	90
5.1. Понятие алгоритма.....	90
5.2. Способы представления алгоритмов.....	93

5.3. Задача сортировки массивов	95
5.4. Прямые методы сортировки массивов.....	96
5.5. Улучшенные методы сортировки массивов.....	102
5.6. Задача поиска в массивах	105
5.7. Задача поиска в строках	108
5.8. Задача поиска в файлах	111
5.9. Динамические структуры данных	113
5.10. Задачи на связность.....	124
5.11. Задачи на поиск минимального пути	128
5.12. Задачи комбинаторной оптимизации.....	132
Заключение.....	140
Список литературы	142

Шевченко Алексей Владимирович
Программирование и основы алгоритмизации
Учебное пособие

Редактор Н. В. Лукина

Подписано в печать 24.09.18. Формат 60×84 1/16. Бумага офсетная.
Печать цифровая. Гарнитура «Times New Roman».
Печ. л. 9,0. Тираж 141 экз. Заказ .

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5