зывается замкнутым «в кольцо». На рисунке 6.10, *в* показана целиком заполненная очередь, а на рис. 6.10, *г* — пустая очередь. Один элемент массива всегда остаётся незанятым[1], иначе невозможно будет различить состояния «очередь пуста» и «очередь заполнена».

Отметим, что приведённая здесь модель описывает работу кольцевого буфера клавиатуры. В современных моделях клавиатур этот буфер может хранить до 15 двухбайтных слов, при его переполнении выдаётся звуковой сигнал.

Кроме того, очередь можно построить на основе связного списка (см. § 41). Детали такой реализации можно найти в литературе или в Интернете.

Существует еще одна линейная динамическая структура данных, которая называется дек.

Дек (от англ. *deque — double ended queue*, двухсторонняя очередь) — это линейная структура данных, в которой можно добавлять и удалять элементы как с одного, так и с другого конца.

Из этого определения следует, что дек может работать и как стек, и как очередь. С помощью дека можно, например, моделировать колоду игральных карт.

Вопросы и задания

1. Что такое стек? Какие операции со стеком разрешены?
2. Как используется системный стек при выполнении программ?
3. Какие ошибки могут возникнуть при использовании стека?
4. В каких случаях можно использовать обычный массив для моделирования стека?
5. Как построить стек на основе динамического массива?
6. Почему при передаче стека в подпрограммы, приведённые в параграфе, соответствующий параметр должен быть изменяемым?
7. Что такое очередь? Какие операции она допускает?
8. Приведите примеры задач, в которых можно использовать очередь.

Подготовьте сообщение

а) «Моделирование стека и очереди в языке Си»

[1] Существует и другой вариант — хранить в отдельной переменной количество элементов в очереди, тогда массив будет использован полностью.

б) «Моделирование стека и очереди в языке Python»

в) «Моделирование очереди с помощью стеков»

г) «Очередь с приоритетом»

Задачи

1. Напишите программу, которая «переворачивает» массив, записанный в файл, с помощью стека. Размер массива неизвестен. Все операции со стеком вынесите в отдельный модуль.
2. Напишите программу, которая вычисляет значение арифметического выражения, записанного в постфиксной форме. Выражение вводится с клавиатуры в виде символьной строки.

*3. Найдите в литературе или в Интернете алгоритм перевода арифметического выражения из инфиксной формы в постфиксную и напишите программу, которая решает эту задачу.

4. Напишите программу, которая проверяет правильность скобочного выражения с четырьмя видами скобок: (), [], {} и <>. Все операции со стеком вынесите в отдельный модуль. Программа должна определять номер ошибочного символа в строке.
5. Напишите вариант предыдущей программы, в котором в качестве стека используется символьная строка.
6. Напишите программу, которая выполняет заливку одноцветной области заданным цветом. Матрица, содержащая цвета пикселей, вводится из файла. Затем с клавиатуры вводятся координаты точки заливки и цвет заливки. На экран нужно вывести матрицу, которая получилась после заливки. Все операции с очередью вынесите в отдельный модуль.

*7. Перепишите программу из задачи 5 — используйте статический массив для организации очереди. Считайте, что в очереди может быть не более 100 элементов. Предусмотрите обработку ошибки «очередь переполнена».

*8. Напишите программу решения задачи о заливке области, помечая при этом точки, добавленные в очередь, чтобы не добавлять их повторно. В чём преимущества и недостатки такого алгоритма?

§ 44 Деревья

Что такое дерево?

Как вы знаете из главы 1 учебника для 10 класса, **дерево** — это структура данных, отражающая иерархию (отношения подчинённости, многоуровневые связи). Напомним некоторые основные понятия, связанные с деревьями.

Дерево состоит из **узлов** и связей между ними (они называются **дугами**). Самый первый узел, расположенный на верхнем уровне (в него не входит ни одна стрелка-дуга), — это **корень дерева**. Конечные узлы, из которых не выходит ни одна дуга, называются **листьями**. Все остальные узлы, кроме корня и листьев, — это промежуточные узлы.

Из двух связанных узлов тот, который находится на более высоком уровне, называется **родителем**, а другой — **сыном**. Корень — это единственный узел, у которого нет родителя; у листьев нет сыновей.

Используются также понятия «предок» и «потомок». **Потомок** какого-то узла — это узел, в который можно перейти по стрелкам от узла-предка. Соответственно, **предок** какого-то узла — это узел, из которого можно перейти по стрелкам в данный узел.

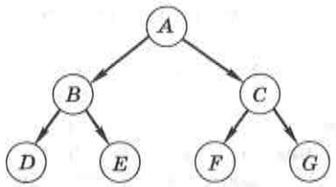


Рис. 6.11

В дереве на рис. 6.11 родитель узла E — это узел B, а предки узла E — это узлы A и B, для которых узел E — потомок. Потомками узла A (корня дерева) являются все остальные узлы.

Высота дерева — это наибольшее расстояние (количество дуг) от корня до листа. Высота дерева, приведённого на рис. 6.11, равна 2.

Формально **дерево** можно определить следующим образом:
1) пустая структура — это дерево;
2) дерево — это корень и несколько связанных с ним отдельных (не связанных между собой) деревьев.

Здесь множество объектов (деревьев) определяется через само это множество на основе простого базового случая (пустого дерева). Такой приём называется **рекурсией** (см. главу 8 учебника для 10 класса). Поэтому можно ожидать, что при работе с деревьями будут полезны рекурсивные алгоритмы.

Чаще всего в информатике используются *двоичные* (или *бинарные*) деревья, т. е. такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

Двоичное дерево:
1) пустая структура — это двоичное дерево;
2) двоичное дерево — это корень и два связанных с ним отдельных двоичных дерева (левое и правое поддеревья).

Деревья широко применяются в следующих задачах:
- поиск в большом массиве данных;
- сортировка данных;
- вычисление арифметических выражений;
- оптимальное кодирование данных (метод сжатия Хаффмана).

Деревья поиска

Известно, что для того, чтобы найти заданный элемент в неупорядоченном массиве из N элементов, может понадобиться N сравнений. Теперь предположим, что элементы массива организованы в виде специальным образом построенного дерева, например как показано на рис. 6.12.

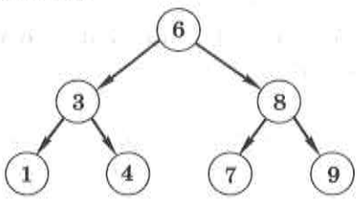


Рис. 6.12

Значения, связанные с узлами дерева, по которым выполняется поиск, называются **ключами** этих узлов (кроме ключа узел может содержать множество других данных). Перечислим важные свойства дерева, показанного на рис. 6.12:
- слева от каждого узла находятся узлы, ключи которых меньше или равны ключу данного узла;
- справа от каждого узла находятся узлы, ключи которых больше или равны ключу данного узла.

Дерево, обладающее такими свойствами, называется **двоичным деревом поиска**.

Например, пусть нужно найти узел, ключ которого равен 4. Начинаем поиск по дереву с корня. Ключ корня — 6 (больше заданного), поэтому дальше нужно искать только в левом поддереве и т. д.

Скорость поиска наибольшая в том случае, если дерево *сбалансировано*, т. е. для каждой его вершины высота левого и правого подде-

ревьев различается не более чем на единицу. Если при линейном поиске в массиве за одно сравнение отсекается 1 элемент, здесь — сразу примерно половина оставшихся. Количество операций сравнения в этом случае пропорционально $\log_2 N$, т. е. алгоритм имеет асимптотическую сложность $O(\log_2 N)$. Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм выгодно применять в тех случаях, когда данные меняются редко, а поиск выполняется часто (например, в базах данных).

Обход двоичного дерева

Обойти дерево — это значит «посетить» все узлы по одному разу. Если перечислить узлы в порядке их посещения, мы представим данные в виде списка.

Существуют несколько способов обхода двоичного дерева:

- **КЛП** — «корень – левый – правый» (обход в прямом порядке):

```
посетить корень
обойти левое поддерево
обойти правое поддерево
```

- **ЛКП** — «левый – корень – правый» (симметричный обход):

```
обойти левое поддерево
посетить корень
обойти правое поддерево
```

- **ЛПК** — «левый – правый – корень» (обход в обратном порядке):

```
обойти левое поддерево
обойти правое поддерево
посетить корень
```

Как видим, это рекурсивные алгоритмы. Они должны заканчиваться без повторного вызова, когда текущий корень — пустое дерево.

Рассмотрим дерево, которое может быть составлено для вычисления арифметического выражения `(1 + 4) * (9 - 5)` (рис. 6.13).

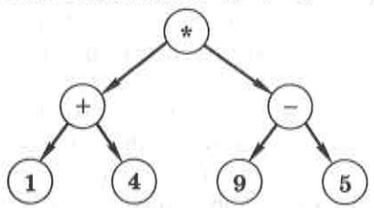


Рис. 6.13

Выражение вычисляется по такому дереву снизу вверх, т. е. посещение корня дерева — это последняя выполняемая операция.

Различные типы обхода дают последовательность узлов:

КЛП: `* + 1 4 - 9 5`
ЛКП: `1 + 4 * 9 - 5`
ЛПК: `1 4 + 9 5 - *`

В первом случае мы получили префиксную форму записи арифметического выражения, во втором — привычную нам инфиксную форму (только без скобок), а в третьем — постфиксную форму. Напомним, что в префиксной и в постфиксной формах скобки не нужны.

Обход КЛП называется «обходом в глубину», потому что сначала мы идём вглубь дерева по левым поддеревьям, пока не дойдём до листа. Такой обход можно выполнить с помощью стека следующим образом:

```
записать в стек корень дерева
нц пока стек не пуст
  выбрать узел V с вершины стека
  посетить узел V
  если у узла V есть правый сын то
    добавить в стек правого сына V
  все
  если у узла V есть левый сын то
    добавить в стек левого сына V
  все
кц
```

На рисунке 6.14 показано изменение состояния стека при таком обходе дерева, изображенного на рис. 6.13. Под стеком записана метка узла, который посещается (например, данные из этого узла выводятся на экран).

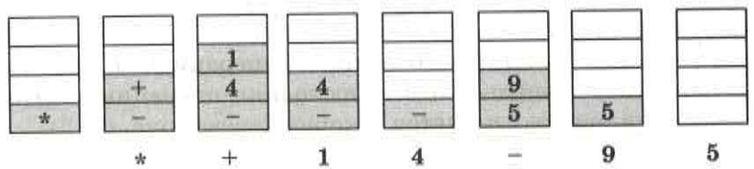


Рис. 6.14

Существует ещё один способ обхода, который называют **обходом в ширину**. Сначала посещают корень дерева, затем — всех

его сыновей, затем — сыновей сыновей («внуков») и т. д., постепенно спускаясь на один уровень вниз. Обход в ширину для приведённого выше дерева даст такую последовательность посещения узлов:

```
* + - 1 4 9 5
```

Для того чтобы выполнить такой обход, применяют очередь. В очередь записывают узлы, которые необходимо посетить. На псевдокоде обход в ширину можно записать так:

```
записать в очередь корень дерева
нц пока очередь не пуста
  выбрать первый узел V из очереди
  посетить узел V
  если у узла V есть левый сын то
    добавить в очередь левого сына V
  все
  если у узла V есть правый сын то
    добавить в очередь правого сына V
  все
кц
```

Вычисление арифметических выражений

Один из способов вычисления арифметических выражений основан на использовании дерева. Сначала выражение, записанное в линейном виде (в одну строку), нужно «разобрать» и построить соответствующее ему дерево. Затем в результате прохода по этому дереву от листьев к корню вычисляется результат.

Для простоты будем рассматривать только арифметические выражения, содержащие числа и знаки четырёх арифметических операций: + - * /. Построим дерево для выражения

```
40 - 2 * 3 - 4 * 5
```

Нужно сначала найти последнюю операцию, просматривая выражение слева направо. Здесь последняя операция — это второе вычитание, оно оказывается в корне дерева (рис. 6.15).

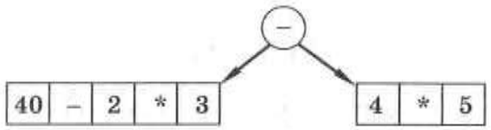


Рис. 6.15

Как выполнить этот поиск в программе? Известно, что операции выполняются в порядке **приоритета** (старшинства): сначала операции с более высоким приоритетом (слева направо), потом — с более низким (также слева направо). Отсюда следует важный вывод.

> В корень дерева нужно поместить последнюю из операций с наименьшим приоритетом.

Теперь нужно построить таким же способом левое и правое поддеревья (рис. 6.16).

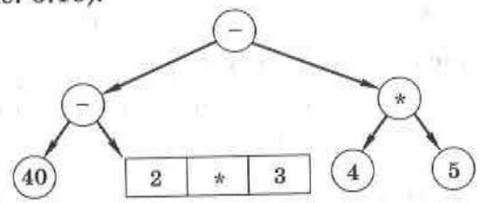


Рис. 6.16

Левое поддерево требует ещё одного шага (рис. 6.17).

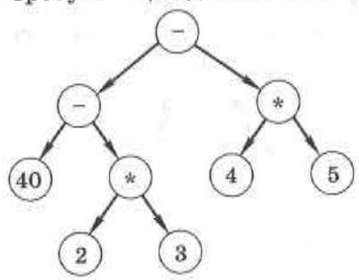


Рис. 6.17

Эта процедура рекурсивная, её можно записать в виде псевдокода:

```
найти последнюю выполняемую операцию
если операций нет то
  создать узел-лист
  выход
все
поместить найденную операцию в корень дерева
построить левое поддерево
построить правое поддерево
```

Рекурсия заканчивается, когда в оставшейся части строки нет ни одной операции, значит, там находится число (это лист дерева).

Теперь вычислим выражение по дереву. Если в корне находится знак операции, её нужно применить к результатам вычисления поддеревьев:

```
n1:=значение левого поддерева
n2:=значение правого поддерева
результат:=операция(n1,n2)
```

Снова получился рекурсивный алгоритм.

Возможен особый случай (на нём заканчивается рекурсия), когда корень дерева содержит число (т. е. это лист). Это число и будет результатом вычисления выражения.

Использование связанных структур

Поскольку двоичное дерево — это нелинейная структура данных, применять динамический массив для размещения элементов не очень удобно (хотя возможно). Вместо этого будем использовать связанные узлы. Каждый такой узел — это структура, содержащая три области: область данных, ссылка на левое поддерево (указатель) и ссылка на правое поддерево (второй указатель). У листьев нет сыновей, в этом случае в указатели будем записывать значение `nil` (нулевой указатель). Дерево, состоящее из трёх таких узлов, показано на рис. 6.18.

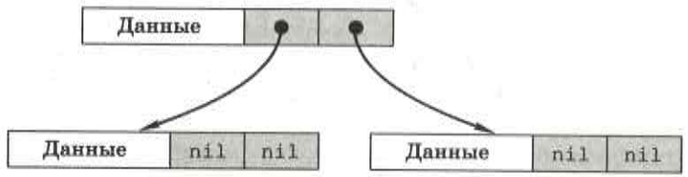


Рис. 6.18

В данном случае область данных узла будет содержать одно поле — символьную строку, в которую записывается знак операции или число в символьном виде.

Введём два новых типа: `TNode` — узел дерева, и `PNode` — указатель (ссылку) на такой узел[1]:

```
type
  PNode = ^TNode;
  TNode = record
    data: string[20];
    left, right: PNode
  end;
```

[1] Заметим, что при объявлении типа указателей `PNode` используется «ссылка вперёд», на тип `TNode`, который объявляется ниже.

Самый важный момент — выделение памяти под новую структуру. Предположим, что *p* — это переменная-указатель типа `PNode`. Для того чтобы выделить память под новую структуру и записать адрес выделенного блока в *p*, используется процедура *New* (англ. *new* — новый):

```
New(p);
```

Как программа определяет, сколько памяти нужно выделить? Чтобы ответить на этот вопрос, вспомним, что указатель `p` указывает на структуру типа `TNode`, размер которой и определяет размер выделяемого блока памяти.

Для освобождения памяти служит процедура `Dispose` (англ. *dispose* — ликвидировать):

```
Dispose(p);
```

В основной программе объявим одну переменную типа `PNode` — это будет ссылка на корень дерева:

```
var T: PNode;
```

Вычисление выражения сводится к двум вызовам функций:

```
T:=Tree(s);
writeln('Результат: ', Calc(T));
```

Здесь предполагается, что арифметическое выражение записано в символьной строке *s*, функция `Tree` строит в памяти дерево по этой строке, а функция `Calc` — вычисляет значение выражения по готовому дереву.

При построении дерева нужно выделять в памяти новый узел и искать последнюю выполняемую операцию — это будет делать функция `LastOp`. Она возвращает 0, если ни одной операции не обнаружено, в этом случае создаётся лист — узел без потомков. Если операция найдена, её обозначение записывается в поле *data*, а в указатели записываются адреса поддеревьев, которые строятся рекурсивно для левой и правой частей выражения:

```
function Tree(s: string): PNode;
var k: integer;
begin
  New(Tree);                {выделить память}
  k:=LastOp(s);
  if k=0 then begin         {создать лист}
```

```
      Tree^.data:=s;
      Tree^.left:=nil;
      Tree^.right:=nil;
    end
    else begin              {создать узел-операцию}
      Tree^.data:=s[k];
      Tree^.left:=Tree(Copy(s,1,k-1));
      Tree^.right:=Tree(Copy(s,k+1,Length(s)-k))
    end
  end;
```

Функция Calc тоже будет рекурсивной:

```
function Calc(Tree: PNode): integer;
var n1, n2, res: integer;
begin
  if Tree^.left = nil then
    Val(Tree^.data, Calc, res)
  else begin
    n1:=Calc(Tree^.left);
    n2:=Calc(Tree^.right);
    case Tree^.data[1] of
      '+': Calc:=n1+n2;
      '-': Calc:=n1-n2;
      '*': Calc:=n1*n2;
      '/': Calc:=n1 div n2;
    end
  end
end;
```

Если ссылка, переданная функции, указывает на лист (нет левого поддерева), то значение выражения — это результат преобразования числа из символьной формы в числовую (с помощью процедуры Val). В противном случае вычисляются значения для левого и правого поддеревьев и к ним применяется операция, указанная в корне дерева.

Осталось написать функцию LastOp. Нужно найти в символьной строке последнюю операцию с минимальным приоритетом. Для этого составим функцию, возвращающую приоритет операции (переданного ей символа):


```
function Priority(op: char): integer;
begin
  case op of
    '+','-': Priority:=1;
    '*','/': Priority:=2
    else     Priority:=100
  end
end;
```

Сложение и вычитание имеют приоритет 1, умножение и деление — приоритет 2, а все остальные символы (не операции) — приоритет 100 (условное значение).

Функция LastOp может выглядеть так:

```
function LastOp(s: string): integer;
var i, minPrt: integer;
begin
  minPrt:=50;
  LastOp:=0;
  for i:=1 to Length(s) do
    if Priority(s[i])<=minPrt then begin
      minPrt:=Priority(s[i]);
      LastOp:=i
    end
end;
```

Обратите внимание, что в условном операторе указано нестрогое неравенство, чтобы найти именно *последнюю* операцию с наименьшим приоритетом. Начальное значение переменной *minPrt* можно выбрать любое между наибольшим приоритетом операций (2) и условным кодом не-операции (100). Тогда если найдена любая операция, условный оператор срабатывает, а если в строке нет операций, условие всегда ложно и в переменной *LastOp* остается начальное значение 0.

Хранение двоичного дерева в массиве

Двоичные деревья можно хранить в (динамическом) массиве почти так же, как и списки. Вопрос о том, как сохранить структуру (взаимосвязь узлов), решается следующим образом. Если нумерация элементов массива A начинается с 1, то сыновья элемента $A[i]$ — это $A[2*i]$ и $A[2*i+1]$. На рисунке 6.19 показан порядок расположения элементов в массиве для дерева, соответствующего выражению

40 - 2 * 3 - 4 * 5

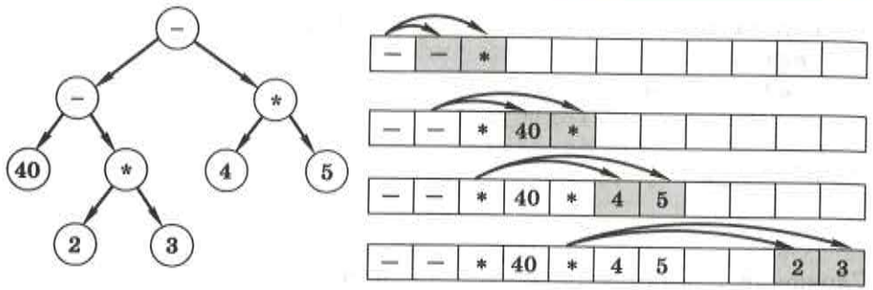


Рис. 6.19

Алгоритм вычисления выражения остаётся прежним, изменяется только метод хранения данных. Обратите внимание, что некоторые элементы остались пустыми, это значит, что их родитель — лист дерева.

Этот способ хорош для хранения сбалансированных (или почти сбалансированных) деревьев, иначе в массиве будет много пустых элементов, которые зря расходуют память.

Вопросы и задания

1. Дайте определение понятий «дерево», «корень», «лист», «родитель», «сын», потомок», «предок», «высота дерева».
2. Где используются структуры типа «дерево» в информатике и в других областях?
3. Объясните рекурсивное определение дерева.
4. Можно ли считать, что линейный список — это частный случай дерева?
5. Какими свойствами обладает дерево поиска?
6. Подумайте, как можно построить дерево поиска из массива данных.
7. Что такое сбалансированное дерево?
8. Какие преимущества имеет поиск с помощью дерева?
9. Что такое обход дерева?
10. Какие способы обхода дерева вы знаете? Придумайте другие способы обхода.
11. Как строится дерево для вычисления арифметического выражения?
12. Как можно представить дерево в программе на Паскале?
13. Как указать, что узел дерева не имеет левого (правого) сына?
14. Как выделяется память под новый узел?
15. Как вы думаете, почему рекурсивные алгоритмы работы с деревьями записываются проще, чем нерекурсивные?
16. Как хранить двоичное дерево в массиве? Можно ли использовать такой приём для хранения деревьев, в которых узлы могут иметь больше двух сыновей? Приведите пример.

Подготовьте сообщение

а) «Деревья в языке Си»
б) «Деревья в языке Python»
в) «Диграммы связей (mind maps)»

Задачи

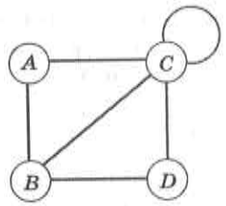
1. Напишите программу, которая вводит и вычисляет арифметическое выражение без скобок. Все операции с деревом вынесите в отдельный модуль.
2. Добавьте в программу из задачи 1 процедуры обхода построенного дерева так, чтобы получить префиксную и постфиксную записи введённого выражения.

*3. Добавьте в программу задачи 2 процедуру обхода дерева в ширину.
*4. Усовершенствуйте программу из задачи 1, чтобы она могла вычислять выражения со скобками.
*5. Включите в вашу программу вычисления арифметического выражения без скобок обработку некоторых ошибок (например, два знака операций подряд). Поработайте в парах: обменяйтесь программами с соседом и попробуйте найти выражение, при котором его программа завершится аварийно (но не выдаст сообщение об ошибке).
*6. Напишите программу вычисления арифметического выражения, которая хранит дерево в виде массива. Все операции с деревом вынесите в отдельный модуль.

§ 45 Графы

Что такое граф?

Как вы знаете из курса 10 класса, **граф** — это набор **вершин (узлов)** и связей между ними **(рёбер)**. Информацию о вершинах и рёбрах графа обычно хранят в виде таблицы специального вида — **матрицы смежности** (рис. 6.20).



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	1	1
D	0	1	1	0

Рис. 6.20

Единица на пересечении строки A и столбца B означает, что между вершинами A и B есть связь. Ноль указывает на то, что связи нет. Матрица смежности симметрична относительно главной диагонали (выделенные фоном ячейки в таблице). Единица на главной диагонали обозначает **петлю** — ребро, которое начинается и заканчивается в одной и той же вершине (в данном примере — в вершине C). Строго говоря, граф — это математический объект, а не рисунок. Конечно, его можно нарисовать на плоскости, но матрица смежности не даёт никакой информации о том, как именно следует располагать вершины друг относительно друга. Для таблицы, приведённой на рис. 6.20, возможны, например, такие варианты (рис. 6.21).

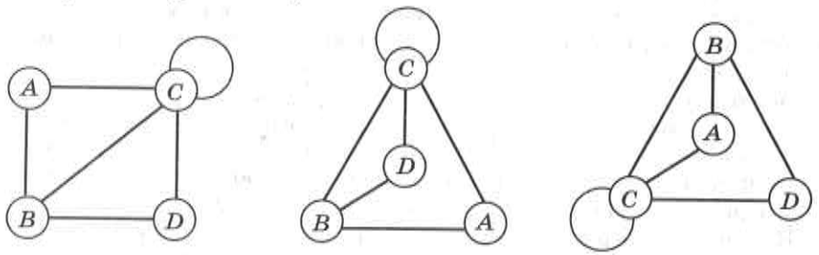


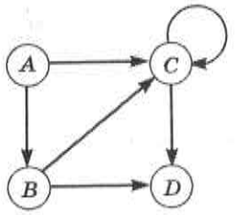
Рис. 6.21

В рассмотренном примере все вершины связаны, т. е. между любой парой вершин существует **путь** — последовательность рёбер, по которым можно перейти из одной вершины в другую. Такой граф называется **связным**.

Вспоминая материал предыдущего параграфа, можно сделать вывод, что дерево — это частный случай связного графа, в котором нет замкнутых путей — **циклов**.

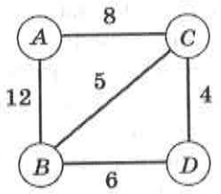
Если для каждого ребра указано направление, граф называют **ориентированным (орграфом)**. Рёбра орграфа называют **дугами**. Его матрица смежности не всегда симметричная. Единица, стоящая на пересечении строки A и столбца B, говорит о том, что существует дуга из вершины A в вершину B (рис. 6.22).

Часто с каждым ребром связывают некоторое число — **вес ребра**. Это может быть, например, расстояние между городами или стоимость проезда. Такой граф называется **взвешенным**. Информация о взвешенном графе хранится в виде **весовой матрицы**, содержащей веса рёбер (рис. 6.23).



	A	B	C	D
A	0	1	1	0
B	0	0	1	1
C	0	0	1	1
D	0	0	0	0

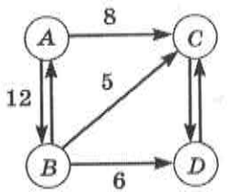
Рис. 6.22



	A	B	C	D
A		12	8	
B	12		5	6
C	8	5		4
D		6	4	

Рис. 6.23

У взвешенного орграфа весовая матрица может быть несимметрична относительно главной диагонали (рис. 6.24).



	A	B	C	D
A		12	8	
B	12		5	6
C				4
D			4	

Рис. 6.24

Если связи между двумя вершинами нет, на бумаге можно оставить ячейку таблицы пустой, а при хранении в памяти компьютера записывать в неё условный код, например 0, –1 или очень большое число (∞), в зависимости от задачи.

«Жадные» алгоритмы

Задача 1. Известна схема дорог между несколькими городами. Числа на схеме (рис. 6.25) обозначают расстояния (дороги не пря-

мые, поэтому неравенство треугольника может нарушаться). Нужно найти кратчайший маршрут из города A в город F.

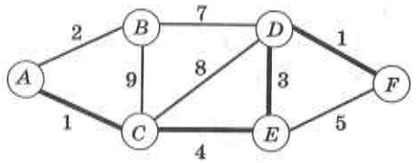


Рис. 6.25

Первая мысль, которая приходит в голову: на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы ещё не были. Для заданной схемы на первом этапе едем в город C (длина 1), далее — в E (длина 4), затем в D (длина 3) и, наконец, в F (длина 1). Общая длина маршрута равна 9.

Алгоритм, который мы применили, называется **«жадным»**. Он состоит в том, чтобы на каждом шаге многоходового процесса выбирать наилучший в данный момент вариант, не думая о том, что впоследствии этот выбор может привести к худшему решению.

Для данной схемы «жадный» алгоритм даёт оптимальное решение, но так будет далеко не всегда. Например, для той же задачи с другой схемой (рис. 6.26) «жадный» алгоритм даст маршрут A–B–D–F длиной 10, хотя существует более короткий маршрут A–C–E–F длиной 7.

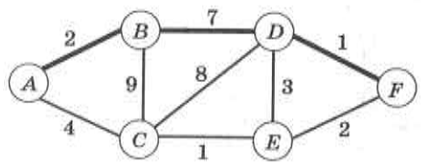


Рис. 6.26

«Жадный» алгоритм не всегда позволяет получить оптимальное решение.

Однако есть задачи, в которых «жадный» алгоритм всегда приводит к правильному решению. Одна из таких задач (её называют **задачей Прима–Крускала** в честь Р. Прима и Д. Крускала, которые независимо предложили её в середине XX века) формулируется так.

Задача 2. В стране Лимонии есть N городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?

В теории графов эта задача называется задачей построения **минимального остовного дерева** (т. е. дерева, связывающего все вершины). Остовное дерево для связного графа с N вершинами имеет $N - 1$ ребро.

Рассмотрим «жадный» алгоритм решения этой задачи, предложенный Крускалом:

1) начальное дерево — пустое;
2) на каждом шаге к будущему дереву добавляется ребро минимального веса, которое ещё не было выбрано и не приводит к появлению цикла.

На рисунке 6.27 показано минимальное остовное дерево для одного из рассмотренных выше графов (сплошные жирные линии).

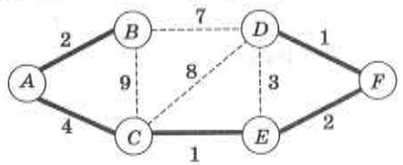


Рис. 6.27

Здесь возможна такая последовательность добавления рёбер: CE, DF, AB, EF, AC. Обратите внимание, что после добавления ребра EF следующее «свободное» ребро минимального веса — это DE (длина 3), но оно образует цикл с рёбрами DF и EF, поэтому не было включено в дерево.

При программировании этого алгоритма сразу возникает вопрос: как определить, что ребро ещё не включено в дерево и не образует цикла в нём? Существует очень красивое решение этой проблемы, основанное на «раскраске» вершин.

Сначала все вершины раскрашиваются в разные цвета, т. е. все рёбра из графа удаляются. Таким образом, мы получаем множество элементарных деревьев (так называемый *лес*), каждое из которых состоит из одной вершины. Затем последовательно соединяем отдельные деревья, каждый раз выбирая ребро минимальной длины, соединяющее разные деревья (выкрашенные в разные цвета). Объединённое дерево перекрашивается в один цвет, совпадающий с цветом одного из вошедших в него поддеревьев. В конце концов все вершины оказываются выкрашены в один цвет, то есть все они принадлежат одному остовному дереву. Можно дока-

зать, что это дерево будет иметь минимальный вес, если на каждом шаге выбирать подходящее ребро минимальной длины.

В программе сначала присвоим всем вершинам разные числовые коды («цвета»):

```
for i:=1 to N do col[i]:=i;
```

Здесь N — количество вершин, а *col* — целочисленный массив с индексами от 1 до N.

Затем в цикле $N-1$ раз (именно столько рёбер нужно включить в дерево) выполняем следующие операции:

1) ищем ребро минимальной длины среди всех рёбер, *концы которых окрашены в разные цвета*;
2) найденное ребро (*iMin*,*jMin*) добавляется в список выбранных, и все вершины, имеющие цвет *col*[*jMin*], перекрашиваются в цвет *col*[*iMin*].

Приведём полностью основной цикл программы:

```
for k:=1 to N-1 do begin
    {поиск ребра с минимальным весом}
  min:=MaxInt;
  for i:=1 to N do
    for j:=1 to N do
      if (col[i]<>col[j]) and (W[i,j] < min)
        then begin
          iMin:=i; jMin:=j; min:=W[i,j]
        end;
    {добавление ребра в список выбранных}
  ostov[k,1]:=iMin;
  ostov[k,2]:=jMin;
    {перекрашивание вершин}
  c:=col[jMin];
  for i:=1 to N do
    if col[i]=c then
      col[i]:=col[iMin]
end;
```

Здесь W — целочисленная матрица размера $N \times N$ (индексы строк и столбцов начинаются с 1); *ostov* — целочисленный массив из $N-1$ строк и двух столбцов для хранения выбранных рёбер (для каждого ребра хранятся номера двух вершин, которые оно соединяет). Если связи между вершинами i и j нет, в элементе $W[i][j]$ матрицы будем хранить «бесконечность» — число, намного большее, чем длина любого ребра. При этом начальное значение переменной *min* должно быть ещё больше.

После окончания цикла остаётся вывести результат — рёбра из массива *ostov*:

```
for i:=1 to N-1 do
  writeln('(', ostov[i,1], ',', ostov[i,2], ')');
```

Кратчайшие маршруты

На примере задачи задачи выбора кратчайшего маршрута (см. задачу 1 выше) мы увидели, что в ней «жадный» алгоритм не всегда даёт правильное решение. В 1960 г. Э. Дейкстра предложил алгоритм, позволяющий найти все кратчайшие расстояния от одной вершины графа до всех остальных и соответствующие им маршруты. Предполагается, что длины всех рёбер (расстояния между вершинами) положительные.

Рассмотрим уже знакомую схему, в которой не сработал «жадный» алгоритм (рис. 6.28).

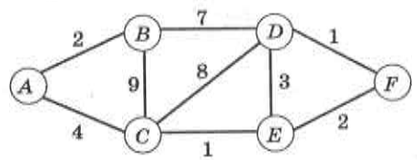


Рис. 6.28

Алгоритм Дейкстры использует дополнительные массивы: в одном (назовём его R) хранятся кратчайшие (на данный момент) расстояния от исходной вершины до каждой из вершин графа, а во втором (массив P) — вершина, из которой нужно «приехать» в данную вершину.

Сначала записываем в массив R длины рёбер от исходной вершины A до всех вершин, а в соответствующие элементы массива P — вершину A (рис. 6.29).

	A	*B*	*C*	*D*	*E*	*F*
R	0	2	4	∞	∞	∞
P	×	*A*	*A*			

Рис. 6.29

Знак ∞ обозначает, что прямого пути из вершины A в данную вершину нет (в программе вместо ∞ можно использовать очень большое число). Таким образом, вершина A уже рассмотрена и соот-

ветствующий элемент массива R выделен фоном. В первый элемент массива P записан символ ×, обозначающий начальную точку маршрута[1].

Из оставшихся вершин находим вершину с минимальным значением в массиве R: это вершина B. Теперь проверяем пути, проходящие через эту вершину: не позволят ли они сократить маршрут к другим вершинам, которые мы ещё не посещали. Идея состоит в следующем: если сумма весов $W[x,z] + W[z,y]$ меньше, чем вес $W[x,y]$, то из вершины X лучше ехать в вершину Y не напрямую, а через вершину Z (рис. 6.30).

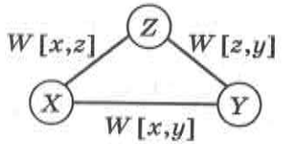


Рис. 6.30

Проверяем наш граф: ехать из A в C через B невыгодно (получается путь длиной 11 вместо 4), а вот в вершину D можно проехать (путь длиной 9), поэтому запоминаем это значение вместо ∞ в массиве R и записываем вершину B на соответствующее место в массив P («в D приезжаем из B») (рис. 6.31).

	A	B	C	D	E	F
R	0	2	4	(9)	∞	∞
P	×	A	A	(B)		

Рис. 6.31

Вершины E и F по-прежнему недоступны.

Следующей рассматриваем вершину C (для неё значение в массиве R минимально). Оказывается, что через неё можно добраться до E (длина пути 5) (рис. 6.32).

	A	B	C	D	E	F
R	0	2	4	9	(5)	∞
P	×	A	A	B	(C)	

Рис. 6.32

[1] В программе можно обозначать вершины номерами и для начальной точки использовать несуществующий номер вершины, например 0 или –1.

Затем посещаем вершину E, которая позволяет достигнуть вершины F и улучшить минимальную длину пути до вершины D (рис. 6.33).

	A	B	C	D	E	F
R	0	2	4	(8)	5	(7)
P	×	A	A	(E)	C	(E)

Рис. 6.33

После рассмотрения вершин F и D таблица не меняется. Итак, мы получили, что кратчайший маршрут из A в F имеет длину 7, причём он приходит в вершину F из E. Как же получить весь маршрут? Нужно просто посмотреть в массиве P, откуда лучше всего ехать в E — выясняется, что из вершины C, а в вершину C — напрямую из начальной точки A (рис. 6.34).

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	×	A	(A)	E	(C)	(E)

Рис. 6.34

Поэтому кратчайший маршрут A–C–E–F. Обратите внимание, что этот маршрут «раскручивается» в обратную сторону, от конечной вершины к начальной. Заметим, что полученная таблица содержит *все* кратчайшие маршруты из вершины A во все остальные вершины, а не только из A в F.

Алгоритм Дейкстры можно рассматривать как своеобразный «жадный» алгоритм: действительно, на каждом шаге из всех невыбранных вершин выбирается такая вершина X, что длина пути от A до X минимальна, если ехать только через уже выбранные вершины. Однако можно доказать, что это расстояние — действительно минимальная длина пути от A до X. Предположим, что для всех предыдущих выбранных вершин это свойство справедливо. При этом X — это ближайшая невыбранная вершина, которую можно достичь из начальной точки, проезжая только через выбранные вершины. Все остальные пути в X, проходящие через ещё не выбранные вершины, будут длиннее, поскольку все рёбра имеют положительную длину. Таким образом, найденная длина пути из A в X — минимальная.

После завершения алгоритма, когда все вершины выбраны, в массиве R находятся длины кратчайших маршрутов.

В программе объявим константу и переменные:

```
const N = 6;
var W: array[1..N,1..N] of integer;
    active: array [1..N] of boolean;
    R, P: array [1..N] of integer;
    i, j, min, kMin: integer;
```

Массив W — это весовая матрица, её удобно вводить из файла. Логический массив *active* хранит состояние вершин (просмотрена или не просмотрена): если значение *active*[*i*] истинно, то вершина активна (ещё не просматривалась).

В начале программы присваиваем начальные значения (объяснение см. выше), сразу помечаем, что вершина 1 просмотрена (не активна), с неё начинается маршрут.

```
active[1]:=False;
R[1]:=0;
P[1]:=0;
for i:=2 to N do begin
  active[i]:=True;
  R[i]:= W[1,i];
  P[i]:=1;
end;
```

В основном цикле, который выполняется $N - 1$ раз (так, чтобы все вершины были просмотрены), среди активных вершин ищем вершину с минимальным соответствующим значением в массиве R и проверяем, не лучше ли ехать через неё:

```
for i:=1 to N-1 do begin
   {поиск новой рабочей вершины R[j] -> min}
  min:=MaxInt; {максимальное целое число}
  for j:=1 to N do
    if active[j] and (R[j]<min) then begin
      min:=R[j];
      kMin:=j;
    end;
  active[kMin]:=False;
   {проверка маршрутов через вершину kMin}
  for j:=1 to N do
```



```
    if R[kMin]+W[kMin,j]<R[j] then begin
      R[j]:=R[kMin]+W[kMin,j];
      P[j]:=kMin
    end
end;
```

В конце программы выводим оптимальный маршрут (здесь — до вершины с номером N) в обратном порядке следования вершин:

```
i:=N;
while i<>0 do begin {для начальной вершины P[i]=0}
  write(i:5);
  i:=P[i]    {переход к следующей вершине}
end;
```

Алгоритм Дейкстры, как мы видели, находит кратчайшие пути *из одной заданной вершины* во все остальные. Найти все кратчайшие пути (*из любой вершины в любую другую*) можно с помощью **алгоритма Флойда–Уоршелла**, основанного на той же самой идее сокращения маршрута (иногда бывает короче ехать через промежуточные вершины, чем напрямую):

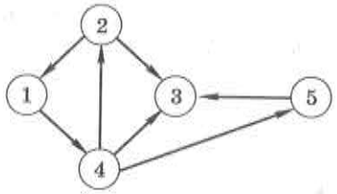
```
for k:=1 to N do
  for i:=1 to N do
    for j:=1 to N do
      if W[i,k]+W[k,j]<W[i,j] then
        W[i,j]:= W[i,k] + W[k,j];
```

В результате исходная весовая матрица графа W размером N × N превращается в матрицу, хранящую длины оптимальных маршрутов. Для того чтобы найти сами маршруты, нужно использовать еще одну дополнительную матрицу, которая выполняет ту же роль, что и массив P в алгоритме Дейкстры.

Использование множеств

Покажем, как описать граф с помощью множеств (см. § 40). Как вы знаете, связи между вершинами графа можно задать списками смежности (см. § 4 учебника для 10 класса). Список смежности для какой-то вершины — это *множество* вершин, с которыми связана данная вершина. Рассмотрим орграф, состоящий из 5 вершин:

Для него списки смежности (с учетом направлений рёбер) выглядят так:



вершина 1: (4)
вершина 2: (1,3)
вершина 3: ()
вершина 4: (2, 3, 5)
вершина 5: (3)

Для того, чтобы описать такой граф, введём два новых типа данных:

```
const N = 5;
type TVertex = 1..N;
     TAdjList = set of TVertex;
```

Тип *TVertex* (английское слово *vertex* означает «вершина») — это тип-диапазон: номер вершины может принимать любые целые значения в диапазоне от 1 до *N*. Второй тип, *TAdjList* — это список смежности, который представляет собой множество номеров вершин (**set of** TVertex).

Граф можно задать списками смежности всех вершин. Для этого объявим массив *Graph*:

```
var Graph: array[1..N] of TAdjList;
```

Граф, показанный на рисунке выше, описывается так:

```
Graph[1]:=[4];
Graph[2]:=[1,3];
Graph[3]:=[];
Graph[4]:=[2,3,5];
Graph[5]:=[3];
```

Используя такое представление, построим функцию *PathCount*, которая находит количество путей из одной вершины в другую. Для упрощения записи будем считать, что *Graph* — глобальный массив. Тогда функции нужно передать только имена начальной и конечной вершин, и её заголовок будет выглядеть так:

```
function PathCount(vStart, vEnd: TVertex): integer;
```


Алгоритм работы функции *PathCount* основан на следующей идее: общее количество путей из вершины *X* в вершину *Y* равно сумме количеств путей из *X* в *Y* через все остальные вершины. Чтобы избежать циклов (замкнутых путей), при этом нужно учитывать только те вершины, которые ещё не посещались. Эту информацию необходимо где-то запоминать, для этой цели мы введём ещё один глобальный массив логических значений:

```
var Visited: array[1..N] of boolean;
```

Если элемент этого массива *Visited*[*i*] принимает значение *True* («истина»), это значит, что вершина с номером *i* уже посещалась. Тогда основной цикл функции *PathCount* приобретает вид

```
count:=0;
for v:=1 to N do
  if (v in Graph[vStart]) and
     (not Visited[v]) then
    count:=count+PathCount(v, vEnd);
```

Здесь переменная *v* имеет тип *TVertex*, т. е. представляет собой номер вершины. Условный оператор содержит сложное условие, которое означает «вершина *v* доступна прямо из вершины *vStart* и ещё не посещалась».

Вы, конечно, заметили, что функция *PathCount* получилась *рекурсивной*, т. е. она вызывает сама себя (в цикле). Поэтому нужно определить условие окончания рекурсии: если начальная и конечная вершины совпадают, то существует только один путь, и можно сразу выйти из функции с помощью оператора **Exit**:

```
if vStart=vEnd then begin
  PathCount:=1;
  Exit
end;
```

Приведём полную программу, которая находит количество путей из вершины 1 в вершину 3:

```
const N = 5;
type TVertex = 1..N;
     TAdjList = set of TVertex;
var Graph: array[1..N] of TAdjList;
    Visited: array[1..N] of boolean;
```

```
function PathCount(vStart, vEnd: TVertex): integer;
var v: TVertex;
    i, count: integer;
begin
  if vStart=vEnd then begin
    PathCount:=1;
    Exit;
  end;
  Visited[vStart]:=True;
  count:=0;
  for v:=1 to N do
    if (v in Graph[vStart]) and
       (not Visited[v]) then
      count:=count+PathCount(v,vEnd);
  Visited[vStart]:=False;
  PathCount:=count
end;
begin
  Graph[1]:=[4];
  Graph[2]:=[1,3];
  Graph[3]:=[];
  Graph[4]:=[2,3,5];
  Graph[5]:=[3];
  writeln(PathCount(1,3))
end.
```

У этой программы есть два существенных недостатка. Во-первых, она не выводит сами маршруты, а только определяет их количество. Во-вторых, некоторые данные могут вычисляться повторно: если мы уже нашли количество путей из какой-то вершины X в конечную вершину, то когда это значение потребуется снова, желательно сразу использовать полученный ранее результат, а не вызывать функцию рекурсивно ещё раз. Попытайтесь улучшить программу самостоятельно так, чтобы исправить эти недостатки.

Некоторые задачи

С графами связаны решения и некоторых других классических задач. Самая известная из них — **задача коммивояжёра** (бродячего торговца).

Задача 3. Бродячий торговец должен посетить N городов по одному разу и вернуться в город, откуда он начал путешествие. Известны расстояния между городами (или стоимость переезда из одного города в другой). В каком порядке нужно посещать города, чтобы суммарная длина пути (или стоимость) оказалась наименьшей?

Эта задача оказалась одной из самых сложных задач оптимизации. По сей день известно только одно надёжное решение — полный перебор вариантов, число которых равно факториалу значения $N - 1$. Это число с увеличением N растёт очень быстро, быстрее, чем любая степень N. Уже для $N = 20$ такое решение требует огромного времени вычислений: компьютер, проверяющий 1 млн вариантов в секунду, будет решать задачу «в лоб» около четырёх тысяч лет. Поэтому математики прилагали большие усилия для того, чтобы сократить перебор — не рассматривать те варианты, которые заведомо не дают лучших результатов, чем уже полученные. В реальных ситуациях нередко оказываются полезными приближённые решения, которые не гарантируют точного оптимального решения, но позволяют получить приемлемый вариант.

Приведём формулировки ещё некоторых задач, которые решаются с помощью теории графов. Алгоритмы их решения вы можете найти в литературе или в Интернете.

Задача 4 (о максимальном потоке). Есть система труб, которые имеют соединения в N узлах. Один узел S является источником, ещё один T — стоком. Известны пропускные способности каждой трубы. Надо найти максимальный поток (количество жидкости, перетекающее за единицу времени) от источника к стоку.

Задача 5. Имеются N населённых пунктов, в каждом из которых живут p_i школьников ($i = 1, \ldots, N$). В каком пункте нужно разместить школу, чтобы общее расстояние, проходимое всеми учениками по дороге в школу, было минимальным?

Задача 6 (о наибольшем паросочетании). Есть M мужчин и N женщин. Каждый мужчина указывает несколько женщин (от 0 до N), на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до M), за которых она согласна выйти замуж. Требуется заключить наибольшее количество многамных браков.

Вопросы и задания

1. Что такое граф?
2. Как обычно задаются связи между вершинами в графах?
3. Что такое матрица смежности?
4. Что такое петля? Как «увидеть» её в матрице смежности?
5. Что такое путь?
6. Какой граф называется связным?
7. Что такое орграф?
8. Как по матрице смежности отличить орграф от неориентированного графа?
9. Что такое взвешенный граф? Как может храниться в памяти информация о нём?
10. Что такое «жадный» алгоритм? Всегда ли он позволяет найти лучшее решение?
11. Подумайте, как можно было бы ускорить работу алгоритма Крускала с помощью предварительной сортировки рёбер.
12. Объясните, как задать граф в языке Паскаль с помощью списков смежности. Подумайте, как можно использовать аналогичный подход в языках программирования, где нет типа данных «множество».
13. Какие достоинства и недостатки, на ваш взгляд, имеют разные способы хранения данных о графе в программе?

Подготовьте сообщение

а) «Работа с графами в языке Си»
б) «Работа с графами в языке Python»
в) «Жадный алгоритм в задаче коммивояжера»
г) «Метод ветвей и границ»
д) «Алгоритм Литтла»
е) «Задача о максимальном потоке»
ж) «Задача о кенигсбергских мостах»
з) «Использование графов для анализа данных в Интернете»
и) «Теория графов в практических задачах»

Задачи

1. Напишите программу, которая вводит из файла весовую матрицу графа и строит для него минимальное остовное дерево.
2. Оцените асимптотическую сложность алгоритма Прима–Крускала.
3. Напишите программу, которая вводит из файла весовую матрицу графа, затем вводит с клавиатуры номера начальной и конечной вершин и определяет кратчайший маршрут.

4. Напишите программу, которая вводит из файла весовую матрицу графа и определяет длины всех кратчайших маршрутов с помощью алгоритма Флойда–Уоршелла.
5. Оцените асимптотическую сложность алгоритмов Дейкстры и Флойда–Уоршелла.
6. Напишите программу, которая решает задачу коммивояжёра для 5 городов методом полного перебора. Можно ли использовать её для 50 городов?

*7. Напишите программу, которая решает задачу 5 (о размещении школы). Для определения кратчайших путей используйте алгоритм Флойда–Уоршелла. Весовую матрицу графа вводите из файла.

8. Перепишите программу для поиска количества путей в графе так, чтобы она не вычисляла данные повторно. Например, если мы подсчитали количество путей из вершины X в конечную вершину, то когда это значение потребуется снова, нужно сразу взять полученный ранее результат, а не вызывать функцию рекурсивно.
9. Перепишите программу для поиска количества путей в графе так, чтобы она выводила не только количество путей, но и сами маршруты как последовательность номеров вершин.

§ 46 Динамическое программирование

Что такое динамическое программирование?

Мы уже встречались с последовательностью чисел Фибоначчи (см. главу 8 учебника для 10 класса):

$$F_1 = F_2 = 1;\ F_n = F_{n-1} + F_{n-2} \text{ при } n > 2.$$

Для их вычисления можно использовать рекурсивную функцию:

```
function Fib(N: integer): integer;
begin
  if N<3 then
    Fib:=1
  else Fib:=Fib(N-1)+Fib(N-2);
end;
```

Каждое из этих чисел связано с предыдущими, вычисление F_5 приводит к рекурсивным вызовам, которые показаны на рис. 6.35. Таким образом, мы два раза вычислили F_3, три раза — F_2 и два раза — F_1. Рекурсивное решение очень простое, но оно неоптимально по быстродействию: компьютер выполняет лишнюю работу, повторно вычисляя уже найденные ранее значения.

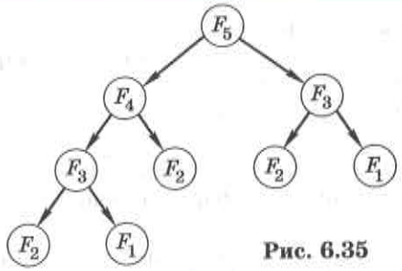


Рис. 6.35

Где же выход? Например, можно хранить все предыдущие числа Фибоначчи в массиве. Пусть этот массив называется F:

```
const N = 10;
var F: array[1..N] of integer;
```

Тогда для вычисления всех чисел Фибоначчи от F_1 до F_N можно использовать цикл:

```
F[1]:=1; F[2]:=1;
for i:=3 to N do
  F[i]:=F[i-1]+F[i-2];
```

Динамическое программирование — это способ решения сложных задач путем сведения их к последовательности более простых подзадач того же типа.

Каждая задача этой послеловательности может быть решена с использованием решений подзадач с меньшими номерами.

Такой подход впервые систематически применил американский математик Р. Беллман при решении сложных многошаговых задач оптимизации. Его идея состояла в том, что оптимальная последовательность шагов оптимальна на любом участке. Например, пусть нужно перейти из пункта A в пункт E через один из пунктов B, C или D (на рис. 6.36 числами обозначена «стоимость» маршрутов).

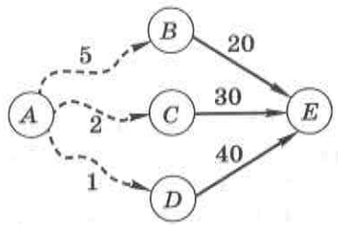


Рис. 6.36

Пусть уже известны оптимальные маршруты из пунктов B, C и D в пункт E (они обозначены сплошными линиями) и их «стоимость». Тогда для нахождения оптимального маршрута из A в E нужно выбрать вариант, который даст минимальную стоимость по сумме двух шагов. В данном случае это маршрут A–B–E, стоимость которого равна 25. Как видим, такие задачи решаются «с конца», т. е. решение начинается от конечного пункта.

В информатике динамическое программирование часто сводится к тому, что мы храним в памяти решения всех задач меньшей размерности. За счёт этого удаётся ускорить выполнение программы. Например, на одном и том же компьютере вычисление F_{45} с помощью рекурсивной функции требует около 8 секунд, а с использованием массива — менее 0,01 с.

Заметим, что в данной простейшей задаче можно обойтись вообще без массива:

```
f2:=1; f1:=1; fN:=1;
for i:=3 to N do begin
  fN:=f1+f2;
  f2:=f1;
  f1:=fN;
end;
```

После окончания работы цикла число Фибоначчи с номером N будет записано в переменную fN.

Задача 1. Найти количество K_N цепочек, состоящих из N нулей и единиц, в которых нет двух стоящих подряд единиц.

При больших N решение задачи методом перебора потребует огромного времени вычисления. Для того чтобы использовать метод динамического программирования, нужно:

1) выразить K_N через предыдущие значения последовательности $K_1, K_2, \ldots, K_{N-1}$;
2) выделить массив для хранения всех предыдущих значений K_i ($i = 1, \ldots, N - 1$).

Самое главное — вывести **рекуррентную формулу**, выражающую K_N через решения аналогичных задач меньшей размерности. Рассмотрим цепочку из N битов, последний элемент которой — 0 (рис. 6.37).

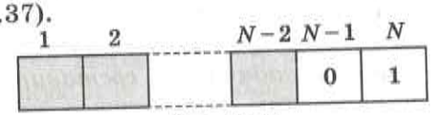


Рис. 6.37

Поскольку дополнительный 0 не может привести к появлению двух соседних единиц, подходящих последовательностей длины N с нулём в конце существует столько, сколько подходящих последовательностей длины $N - 1$, т. е. K_{N-1}. Если же последний символ — это 1, то слева от него обязательно должен быть 0, а остальная цепочка из $N - 2$ битов должна быть правильной, без двух соседних единиц (рис. 6.38). Поэтому подходящих последовательностей длиной N с единицей в начале существует столько, сколько подходящих последовательностей длины $N - 2$, т. е. K_{N-2}.

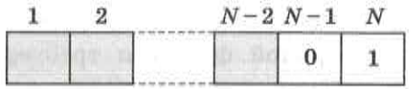


Рис. 6.38

В результате получаем: $K_N = K_{N-1} + K_{N-2}$. Значит, для вычисления очередного числа нам нужно знать два предыдущих.

Теперь рассмотрим простые случаи. Очевидно, что есть две последовательности длины 1 (0 и 1), т. е. $K_1 = 2$. Далее, есть 3 подходящих последовательности длины 2 (00, 01 и 10), поэтому $K_2 = 3$. Легко понять, что решение нашей задачи — число Фибоначчи: $K_N = F_{N+2}$.

Поиск оптимального решения

Задача 2. В цистерне N литров молока. Есть бидоны объёмом 1, 5 и 6 литров. Нужно разлить молоко в бидоны так, чтобы все использованные бидоны были заполнены и их количество было минимальным.

Человек, скорее всего, будет решать задачу перебором вариантов. Наша задача осложняется тем, что требуется написать программу, которая решает задачу для любого введённого числа N.

Самый простой подход — заполнять сначала бидоны самого большого размера (6 л), затем — меньшие и т. д. Это так называемый «жадный» алгоритм. Как вы знаете, он не всегда приводит к оптимальному решению. Например, для $N = 10$ «жадный» алгоритм даёт решение 6 + 1 + 1 + 1 + 1 — всего 5 бидонов, в то время как можно обойтись двумя (5 + 5).

Как и в любом решении, использующем динамическое программирование, *главная задача — составить рекуррентную формулу*. Сначала определим оптимальное число бидонов K_N,

а потом подумаем, как определить, какие именно бидоны нужно использовать.

Представим себе, что мы выбираем бидоны постепенно. Тогда последний выбранный бидон может иметь, например, объём 1 л, в этом случае $K_N = 1 + K_{N-1}$. Если последний бидон имеет объём 5 л, то $K_N = 1 + K_{N-5}$, а если 6 л, то $K_N = 1 + K_{N-6}$. Так как нам нужно выбрать минимальное значение, то

$$K_N = 1 + \min(K_{N-1}, K_{N-5}, K_{N-6}).$$

Вариант, выбранный при поиске минимума, определяет последний добавленный бидон, его объём нужно сохранить в отдельном массиве P. Этот массив будет использован для определения количества выбранных бидонов каждого типа. В качестве начальных значений берём $K_0 = 0$ и $P_0 = 0$.

Полученная формула применима при $N \geq 6$. Для меньших N используются только те данные, которые есть в таблице (рис. 6.39). Например:

$$K_3 = 1 + K_2 = 3,\ K_5 = 1 + \min(K_4, K_0) = 1.$$

На рисунке 6.39 показаны массивы для $N = 10$.

N	0	1	2	3	4	5	6	7	8	9	10
K	0	1	2	3	4	1	1	2	3	4	2
P	0	1	1	1	1	5	6	1	1	1	5

Рис. 6.39

Как по массиву P определить оптимальный состав бидонов? Пусть, например, $N = 10$. Из массива P находим, что последний добавленный бидон имеет объём 5 л (см. значение $P[10]$). Остаётся 10 – 5 = 5 л, в элементе $P[5]$ тоже записано значение 5, поэтому второй бидон тоже имеет объём 5 л. Остаток 0 л означает, что мы полностью определили набор бидонов.

Можно заметить, что такая процедура очень похожа на алгоритм Дейкстры, и это не случайно. В алгоритмах Дейкстры и Флойда–Уоршелла, по сути, используется метод динамического программирования.

Задача 3 (задача о куче). Из камней весом p_i ($i = 1, \ldots, N$) требуется набрать кучу весом ровно W или, если это невозможно, максимально близкую к W (но меньшую, чем W). Все веса камней и значение W — целые числа.

Эта задача относится к трудным задачам целочисленной оптимизации, которые решаются только полным перебором вариантов. Каждый камень может входить в кучу (обозначим это состояние как 1) или не входить (0). Поэтому нужно выбрать цепочку, состоящую из N битов. При этом количество вариантов равно 2^N, и при больших N полный перебор практически невыполним.

Динамическое программирование позволяет найти решение задачи значительно быстрее. Идея состоит в том, чтобы сохранять в массиве решения всех более простых задач этого типа (при меньшем количестве камней и меньшем весе W).

Построим матрицу T, где элемент $T[i,w]$ — это оптимальный вес, полученный при попытке собрать кучу весом w из i первых по счёту камней (w изменяется от 0 до W). Очевидно, что первый столбец заполнен нулями (при заданном нулевом весе никаких камней не берём).

Рассмотрим первую строку (есть только один камень). В начале этой строки будут стоять нули, а дальше, начиная со столбца p_1, — значения p_1 (взяли единственный камень). Это простые варианты задачи, решения для которых легко подсчитать вручную. Рассмотрим пример, когда требуется набрать вес 8 из камней весом 2, 4, 5 и 7 единиц (рис. 6.40).

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0								
5	0								
7	0								

Рис. 6.40

Теперь предположим, что строки с 1-й по $(i-1)$-ю уже заполнены. Перейдем к i-й строке, т. е. добавим в набор i-й камень. Он может быть взят или не взят в кучу. Если мы не добавляем его в кучу, то $T[i,w] = T[i-1,w]$, т. е. решение не меняется от добавления в набор нового камня. Если камень с весом добавлен в кучу, то остаётся «добрать» остаток оптимальным образом (используя только предыдущие камни), т. е. $T[i,w] = T[i-1,w-p_i] + p_i$.

Как же решить, брать или не брать камень? Надо проверить, в каком случае полученный вес будет больше (ближе к w). Таким образом, получается рекуррентная формула для заполнения таблицы:

$$T[i,w] = \begin{cases} T[i-1,w], \text{ при } w < p_i, \\ \max(T[i-1,w], T[i-1,w-p_i] + p_i), \text{ при } w \geq p_i. \end{cases}$$

Используя эту формулу, заполняем таблицу по строкам, сверху вниз; в каждой строке — слева направо (рис. 6.41).

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0	0	2	2	4	4	6	6	6
5	0	0	2	2	4	5	6	7	7
7	0	0	2	2	4	5	6	7	7

Рис. 6.41

Видим, что сумму 8 набрать невозможно, ближайшее значение — 7 (правый нижний угол таблицы).

Эта таблица содержит все необходимые данные для определения выбранной группы камней. Действительно, если камень с весом p_i не включён в набор, то $T[i,w] = T[i-1,w]$, т. е. число в таблице не меняется при переходе на строку вверх. Начинаем с правого нижнего угла таблицы, идём вверх, пока значения в столбце равны 7. Последнее такое значение — для камня с весом 5, поэтому он и выбран. Вычитая его вес из суммы, получаем $7 - 5 = 2$, переходим во второй столбец на одну строку вверх, и снова идём вверх по столбцу, пока значение не меняется (равно 2). Так как мы успешно дошли до самого верха таблицы, взят первый камень с весом 2.

Заметим, что в рассмотренном случае есть и ещё одно решение — взять один камень с весом 7 (подумайте, как в подобных случаях находить все решения задачи).

Как мы уже отмечали, количество вариантов в задаче для N камней равно 2^N, т. е. алгоритм полного перебора имеет асимптотическую сложность $O(2^N)$. В данном алгоритме количество операций равно числу элементов таблицы, т. е. сложность нашего алгоритма — $O(N \cdot w)$. Однако нельзя сказать, что он имеет линейную сложность, так как есть ещё сильная зависимость от заданного веса w. Такие алгоритмы называют *псевдополиномиальными*. В них ускорение вычислений достигается за счёт использования дополнительной памяти для хранения промежуточных результатов.

Количество решений

Задача 4. У исполнителя Утроитель две команды, которым присвоены номера:

1) прибавь 1

2) умножь на 3

Первая из них увеличивает число на экране на 1, вторая — утраивает его. Программа для Утроителя — это последовательность команд. Сколько есть программ, которые число 1 преобразуют в число $N = 20$?

Заметим, что при выполнении любой из команд число увеличивается (не может уменьшаться). Начнем с простых случаев, с которых будем начинать вычисления. Понятно, что для начального значения $N = 1$ существует только одна программа — пустая, не содержащая ни одной команды. Для $N = 2$ есть тоже только одна программа, состоящая из команды сложения. Если через K_N обозначить количество разных программ для получения числа N из 1, то $K_1 = K_2 = 1$.

Теперь рассмотрим общий случай, чтобы построить рекуррентную формулу, связывающую K_N с предыдущими элементами последовательности K_1, K_2, ..., K_{N-1}, т. е. с решениями таких же задач для меньших N.

Если число N не делится на 3, то последней командой для его получения может быть только операция сложения, поэтому $K_N = K_{N-1}$. Если N делится на 3, то последней командой может быть как сложение, так и умножение. Поэтому нужно сложить K_{N-1} (количество программ с последней командой сложения) и $K_{N/3}$ (количество программ с последней командой умножения). В итоге получаем:

$$K_N = \begin{cases} K_{N-1}, \text{если } N \text{ не делится на 3,} \\ K_{N-1} + K_{N/3}, \text{если } N \text{ делится на 3.} \end{cases}$$

Остаётся заполнить таблицу для всех значений от 1 до заданного $N = 20$. Для небольших значений N эту задачу легко решить вручную:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
K_N	1	1	2	2	2	3	3	3	5	5	5	7	7	7	9	9	9	12	12	12

Заметим, что количество вариантов меняется только в тех столбцах, где N делится на 3, поэтому из всей таблицы можно оставить только эти столбцы (и первый), добавляя следующие значения, кратные трём:

N	1	3	6	9	12	15	18	21
K_N	1	2	3	5	7	9	12	15

Заданное число 20 попадает в последний интервал (от 18 до 20), поэтому ответ в данной задаче — 12.

При составлении программы с полной таблицей нужно выделить в памяти целочисленный массив K, индексы которого изменяются от 1 до N, и заполнить его по приведённым выше формулам:

```
K[1]:=1;
for i:=2 to N do begin
  K[i]:=K[i-1];
  if i mod 3 = 0 then
    K[i]:=K[i]+K[i div 3];
end;
```

Ответом будет значение $K[N]$.

Задача 5 (размен монет). Сколькими различными способами можно выдать сдачу размером W рублей, если есть монеты достоинством p_i ($i = 1, \ldots, N$)? Для того чтобы сдачу всегда можно было выдать, будем предполагать, что в наборе есть монета достоинством 1 рубль ($p_1 = 1$).

Это задача, так же, как и задача о куче, решается полным перебором вариантов, число которых при больших N очень велико. Будем использовать динамическое программирование, сохраняя в массиве решения всех задач меньшей размерности (для меньших значений N и W).

В матрице T значение $T[i,w]$ будет обозначать количество вариантов сдачи размером w рублей (w изменяется от 0 до W) при использовании первых i монет из набора. Очевидно, что при нулевой сдаче есть только один вариант (не дать ни одной монеты), так же и при наличии только одного типа монет (напомним, что $p_1 = 1$) есть тоже только один вариант. Поэтому нулевой столбец и первую строку таблицы можно заполнить сразу единицами. Для примера мы будем рассматривать задачу для $W = 10$ и набора монет достоинством 1, 2, 5 и 10 рублей (рис. 6.42).

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2	1										
5	1										
10	1										

Рис. 6.42

Таким образом, мы определили простые базовые случаи, от которых «отталкивается» рекуррентная формула.

Теперь рассмотрим общий случай. Заполнять таблицу будем по строкам, слева направо. Для вычисления $T[i,w]$ предположим, что мы добавляем в набор монету достоинством p_i. Если сумма w меньше, чем p_i, то количество вариантов не увеличивается, и $T[i,w] = T[i-1,w]$. Если сумма больше p_i, то к этому значению нужно добавить количество вариантов с «участием» новой монеты. Если монета достоинством p_i использована, то нужно учесть все варианты «разложения» остатка $w - p_i$ на все доступные монеты, т. е. $T[i,w] = T[i-1,w] + T[i,w-p_i]$. В итоге получается рекуррентная формула

$$T[i,w] = \begin{cases} T[i-1,w], \text{ при } w < p_i, \\ T[i-1,w] + T[i,w-p_i], \text{ при } w \geq p_i, \end{cases}$$

которая используется для заполнения таблицы (рис. 6.43).

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	2	2	3	3	4	4	5	5	6
5	1	1	2	2	3	4	5	6	7	8	10
10	1	1	2	2	3	4	5	6	7	8	(11)

Рис. 6.43

Ответ к задаче находится в правом нижнем углу таблицы.

Вы могли заметить, что решение этой задачи очень похоже на решение задачи о куче камней. Это не случайно, две эти задачи относятся к классу сложных задач, для решения которых известны только переборные алгоритмы. Использование методов динамического программирования позволяет ускорить решение за счёт хранения промежуточных результатов, однако требует дополнительного расхода памяти.

Вопросы и задания

1. Что такое динамическое программирование?
2. Какой смысл имеет выражение «динамическое программирование» в теории многошаговой оптимизации?
3. Какие шаги нужно выполнить, чтобы применить динамическое программирование к решению какой-либо задачи?
4. За счёт чего удаётся ускорить решение сложных задач методом динамического программирования?
5. Какие ограничения есть у метода динамического программирования?

Подготовьте сообщение

а) «Задача о рюкзаке»
б) «Задачи на подпоследовательности»
в) «Задачи на поиск оптимального маршрута»

Задачи

1. Напишите программу, которая определяет оптимальный набор бидонов в задаче 2 из параграфа. С клавиатуры или из файла вводится объём цистерны, количество типов бидонов и их размеры.
2. Напишите программу, которая решает задачу 3 о куче камней заданного веса, рассмотренную в тексте параграфа.
3. **Задача о ранце.** Есть N предметов, для каждого из которых известен вес p_i ($i = 1,\ldots, N$) и стоимость c_i ($i = 1,\ldots, N$). В ранец можно взять предметы общим весом не более W. Напишите программу, которая определяет самый дорогой набор предметов, который можно унести в ранце.
4. У исполнителя Калькулятор две команды, которым присвоены номера:

   1) `прибавь 1`
   3) `умножь на 4`

   Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число M в число N, оба числа вводятся с клавиатуры.
5. У исполнителя Калькулятор три команды, которым присвоены номера:

   1) `прибавь 1`
   2) `умножь на 3`
   3) `умножь на 4`

Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число M в число N, оба числа вводятся с клавиатуры.

6. У исполнителя Калькулятор две команды, которым присвоены номера:

   1) `прибавь 1`

   2) `увеличь каждый разряд числа на 1`

   Сколько есть программ, которые число 24 преобразуют в число 46?

7. У исполнителя Калькулятор две команды, которым присвоены номера:

   1) `прибавь 1`

   2) `увеличь каждый разряд числа на 1`

   Сколько существует программ, которые число 26 преобразуют в число 49?

*8. Прямоугольный остров разделён на квадраты так, что его размеры — $N \times M$ квадратов. В каждом квадрате зарыто некоторое число золотых монет, эти данные хранятся в матрице (двумерном массиве) Z, где $Z[i, j]$ — число монет в квадрате с координатами (i, j). Пират хочет пройти из юго-западного угла острова в северо-восточный, причём он может двигаться только на север или на восток. Как пирату собрать наибольшее количество монет? Напишите программу, которая находит оптимальный путь пирата и число монет, которое ему удастся собрать.

Практические работы к главе 6

Работа № 41 «Решето Эратосфена»
Работа № 42 «Длинные числа»
Работа № 43 «Ввод и вывод структур»
Работа № 44 «Чтение структур из файла»
Работа № 45 «Сортировка структур с помощью указателей»
Работа № 46 «Динамические массивы»
Работа № 47 «Расширяющиеся динамические массивы»
Работа № 48 «Алфавитно-частотный словарь»
Работа № 49 «Модули»
Работа № 50 «Вычисление арифметических выражений»
Работа № 51 «Проверка скобочных выражений»
Работа № 52 «Заливка области»
Работа № 53 «Вычисление арифметических выражений»
Работа № 54 «Хранение двоичного дерева в массиве»
Работа № 55 «Алгоритм Прима–Крускала»
Работа № 56 «Алгоритм Дейкстры»

Работа № 57 «Алгоритм Флойда–Уоршелла»
Работа № 58 «Числа Фибоначчи»
Работа № 59 «Задача о куче»
Работа № 60 «Количество программ»
Работа № 61 «Размен монет»

ЭОР к главе 6 на сайте ФЦИОР (http://fcior.edu.ru)

- Решето Эратосфена
- Основные структуры данных
- Сущность модульного программирования. Программный модуль
- Работа с указателями и структурами (на примере языка Pascal)
- Линейные структуры данных. Список, стек, очередь
- Организация и работа со стеком
- Организация и работа с очередью
- Основы теории графов. Способы представления графов. Обход графа
- Задача о кратчайших путях. Алгоритм Флойда, Дейкстры
- Задачи оптимизации. Динамическое программирование

Самое важное в главе 6

- Структура — это сложный тип данных, который позволяет объединить данные разных типов. Элементы структуры называют полями. При обращении к полям структуры используется точечная запись:

  `<имя структуры>.<имя поля>`.

- Указатель — это переменная, в которой можно хранить адрес другой переменной заданного типа. В указателе можно запомнить адрес новой переменной, место для которой выделено в памяти во время работы программы.
- Динамические массивы — это массивы, память для которых выделяется во время работы программы. Динамический массив в программе на языке Паскаль — это указатель, в который записывается адрес выделенного блока памяти. При записи такой переменной в файл сохранится только значение указателя, а значения элементов массива будут потеряны.

- Список — это упорядоченный набор элементов одного типа, для которых введены операции вставки (включения) и удаления (исключения).
- Стек — это линейный список, в котором добавление и удаление элементов разрешаются только с одного конца. Системный стек применяется для хранения адресов возврата из подпрограмм и размещения локальных переменных.
- Дерево — это структура данных, которая моделирует иерархию — многоуровневую структуру. Как правило, дерево определяется с помощью рекурсии, поэтому для его обработки удобно использовать рекурсивные алгоритмы. Деревья используются в задачах поиска, сортировки, вычисления арифметических выражений.
- Граф — это набор вершин и связывающих их рёбер. Информация о графе чаще всего хранится в виде матрицы смежности или весовой матрицы. Наиболее известные задачи, которые решаются с помощью теории графов, — поиск оптимальных маршрутов.
- Динамическое программирование — это метод, позволяющий ускорить решение задачи за счёт хранения решений аналогичных задач меньшей размерности. Для его использования нужно вывести рекуррентную формулу, связывающее решение исходной задачи с решением задач меньшей размерности, и определить простые базовые случаи (условие окончания рекурсии).

Глава 7
Объектно-ориентированное программирование

§ 47
Что такое ООП?

Как вы знаете, работа первых компьютеров сводилась к вычислениям по заданным формулам различной сложности. Число переменных и массивов в программе было невелико, так что программист мог легко удерживать в памяти все взаимосвязи между ними и детали алгоритма.

С каждым годом производительность компьютеров росла, и человек «поручал» им всё более и более трудоёмкие задачи. Компьютеры следующих поколений стали использоваться для создания сложных информационных систем (например, банковских) и моделирования процессов, происходящих в реальном мире. Новые задачи требовали более сложных алгоритмов, объём программ вырос до сотен тысяч и даже миллионов строк, число переменных и массивов измерялось в тысячах.

Программисты столкнулись с проблемой сложности, которая превысила возможности человеческого разума. Один человек уже не способен написать надёжно работающую серьезную программу, так как не может «охватить взглядом» все её детали. Поэтому в разработке большинства современных программ принимает участие множество специалистов. При этом возникает новая проблема: нужно разделить работу между ними так, чтобы каждый мог работать независимо от других, а потом готовую программу можно было бы собрать вместе из готовых блоков, как из кубиков.

Как отмечал известный нидерландский программист Эдсгер Дейкстра, человечество ещё в древности придумало способ управления сложными системами: «разделяй и властвуй». Это означает, что исходную систему нужно разбить на подсистемы (выполнить **декомпозицию**) так, чтобы работу каждой из них можно было рассматривать и совершенствовать независимо от других.

Для этого в классическом (процедурном) программировании используют метод проектирования «сверху вниз»: сложная задача разбивается на части (подзадачи и соответствующие им *алгоритмы*), которые затем снова разбиваются на более мелкие подзадачи и т. д. (рис. 7.1). Однако при этом задачу «реального мира» приходится переформулировывать, представляя все данные в виде переменных, массивов, списков и других структур данных. При моделировании больших систем объём этих данных увеличивается, они становятся плохо управляемыми, и это приводит к большому числу ошибок. Так как любой алгоритм может обратиться к любым глобальным (общедоступным) данным, повышается риск случайного недопустимого изменения каких-то значений.

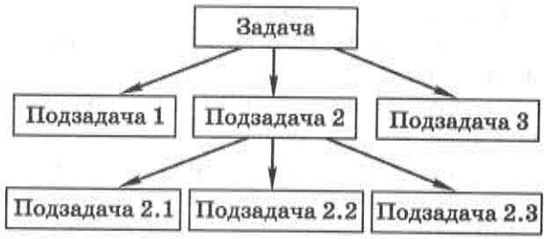


Рис. 7.1

В конце 60-х годов XX века появилась новая идея — применить в разработке программ тот подход, который использует человек в повседневной жизни. Люди воспринимают мир как множество **объектов** — предметов, животных, людей — это отмечал ещё в XVII веке французский математик и философ Рене Декарт. Все объекты имеют внутреннее устройство и состояние, свойства (внешние характеристики) и поведение. Чтобы справиться со сложностью окружающего мира, люди часто не вникают в детали внутреннего устройства и игнорируют многие свойства объектов, ограничиваясь лишь теми, которые необходимы для решения их практических задач. Такой приём называется абстракцией.

Абстракция — это выделение существенных характеристик объекта, отличающих его от других объектов.

Для разных задач существенные свойства одного и того же объекта могут быть совершенно разными. Например, услышав слово «кошка», многие подумают о пушистом усатом животном, которое мурлыкает, когда его гладят. В то же время ветеринарный врач представляет скелет, ткани и внутренние органы кошки, которую ему нужно лечить. В каждом из этих случаев применение абстракции даёт свою модель одного и того же объекта, поскольку различны цели моделирования.

Как применить принцип абстракции в программировании? Поскольку формулировка задач, решаемых на компьютерах, всё более приближается к формулировкам реальных жизненных задач, возникла такая идея: представить программу в виде множества объектов (моделей), каждый из которых обладает своими свойствами и поведением, но его внутреннее устройство скрыто от других объектов. Тогда решение задачи сводится к моделированию взаимодействия этих объектов. Построенная таким образом модель задачи называется *объектной*. Здесь тоже идёт проектирование «сверху вниз», только не по алгоритмам (как в процедурном программировании), а по объектам. Если нарисовать схему такой декомпозиции, она будет представлять собой граф, так как каждый объект может обмениваться данными со всеми другими (рис. 7.2).

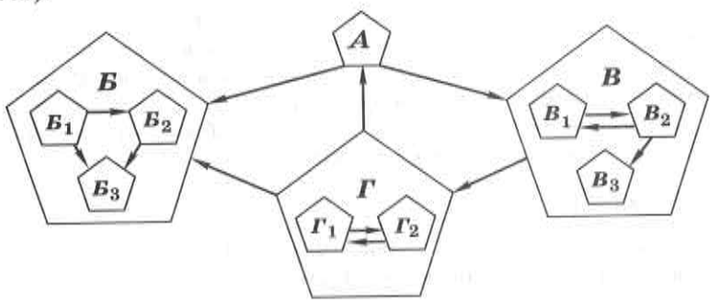


Рис. 7.2

Здесь $А$, $Б$, $В$ и $Г$ — объекты «верхнего уровня»; $Б_1$, $Б_2$ и $Б_3$ — подобъекты объекта $Б$ и т. д.

Для решения задачи «на верхнем уровне» достаточно определить, *что* делает тот или иной объект, не заботясь о том, *как* именно он это делает. Таким образом, для преодоления сложности мы используем *абстракцию*, т. е. сознательно отбрасываем второстепенные детали.

Если построена **объектная модель задачи** (выделены объекты и определены правила обмена данными между ними), можно поручить разработку каждого из объектов отдельному программисту (или группе), которые должны написать соответствующую часть программы, т. е. определить, *как именно* объект выполняет свои функции. При этом конкретному разработчику не обязательно держать в голове полную информацию обо всех объектах, нужно лишь строго соблюдать соглашения о способе обмена данными (*интерфейсе*) «своего» объекта с другими.

Программирование, основанное на моделировании задачи реального мира как множества взаимодействующих объектов, принято называть **объектно-ориентированным программированием (ООП)**. Более строгое определение мы дадим немного позже.

Вопросы и задания

1. Почему со временем неизбежно изменяются методы программирования?
2. Что такое декомпозиция, зачем она применяется?
3. Что такое процедурное программирование? Какой вид декомпозиции в нём используется?
4. Какие проблемы в программировании привели к появлению ООП?
5. Как выполняется декомпозиция алгоритмов в процедурных языках программирования?
6. Что такое абстракция? Зачем она используется в обычной жизни?
7. Объясните, как связана абстракция с моделированием.
8. Какие преимущества даёт объектный подход в программировании?
9. Какой вид декомпозиции используется в ООП?
10. Что такое интерфейс? Приведите примеры объектов, у которых одинаковый интерфейс и разное устройство.

Подготовьте сообщение

а) «Проблемы процедурного программирования»
б) «Глобальные переменные: за и против»
в) «ООП: достоинства и недостатки»

§ 48
Объекты и классы

Как мы увидели в предыдущем параграфе, для того чтобы построить объектную модель, нужно:

- выделить взаимодействующие **объекты**, с помощью которых можно достаточно полно описать поведение моделируемой системы;
- определить **свойства** объектов, существенные в данной задаче;
- описать **поведение** (возможные действия) объектов, т. е. команды, которые объекты могут выполнить.

Этап разработки модели, на котором решаются перечисленные выше задачи, называется **объектно-ориентированным анализом (ООА)**. Он выполняется до того, как программисты напишут самую первую строчку кода, и во многом определяет качество и надёжность будущей программы.

Рассмотрим объектно-ориентированный анализ на примере простой задачи. Пусть нам необходимо изучить движение автомобилей на шоссе, например, для того, чтобы определить, достаточна ли его пропускная способность. Как построить объектную модель этой задачи? Прежде всего, нужно разобраться, что такое объект.

Объектом можно назвать то, что имеет чёткие границы и обладает **состоянием** и **поведением**.

Состояние объекта определяет его возможное поведение. Например, лежачий человек не может прыгнуть, а незаряженное ружьё не выстрелит.

В нашей задаче объекты — это дорога и двигающиеся по ней машины. Машин может быть несколько, причём все они, с точки зрения нашей задачи, имеют общие свойства. Поэтому нет смысла подробно описывать каждую машину по отдельности: достаточно один раз определить их общие черты, а потом просто сказать, что все машины ими обладают. В ООП для этой цели вводится специальный термин — «класс».

Класс — это описание множества объектов, имеющих общую структуру и общее поведение.

Например, в рассматриваемой задаче можно ввести два класса — *Дорога* и *Машина*. По условию, дорога одна, а машин может быть много.

Будем рассматривать прямой отрезок дороги, в этом случае объект «дорога» имеет два свойства, важных для нашей задачи: длину и ширину — число полос движения (рис. 7.3). Эти свойства определяют *состояние* дороги. *«Поведение»* дороги может заключаться в том, что число полос уменьшается, например, из-за ремонта покрытия, но в нашей простейшей модели объект «дорога» не будет изменяться.

Рис. 7.3

Дорога
длина ширина

Рис. 7.4

Схематично класс *Дорога* можно изобразить в виде прямоугольника с тремя секциями: в верхней записывают название класса, во второй части — свойства, а в третьей — возможные действия, которые называют **методами**. В нашей модели дороги два свойства и ни одного метода (рис. 7.4).

Теперь рассмотрим объекты класса *Машина*. Их важнейшие свойства — координаты и скорость движения. Для упрощения будем считать, что:

- все машины одинаковы;
- каждая машина движется по дороге слева направо с постоянной скоростью (скорости разных машин могут быть различными);
- по каждой полосе движения едет только одна машина, так что можно не учитывать обгон и переход на другую полосу;
- если машина выходит за правую границу дороги, вместо неё слева на той же полосе появляется новая машина.

Не все эти допущения выглядят естественно, но такая простая модель позволит понять основные принципы метода.

За координаты машины можно принять расстояние X от левого края рассматриваемого участка шоссе и номер полосы P (натуральное число — рис. 7.5). Скорость автомобиля V в нашей модели — неотрицательная величина.

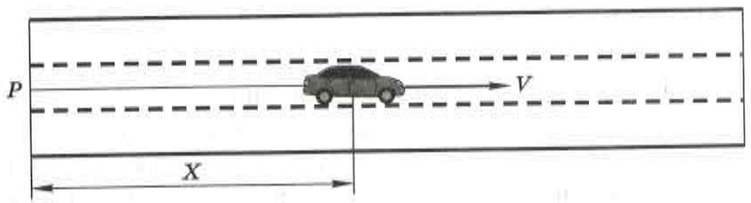


Рис. 7.5

Теперь рассмотрим поведение машины. В данной модели она может выполнять всего одну команду — ехать в заданном направлении (назовём её «двигаться»). Говорят, что объекты класса *Машина* имеют метод «двигаться» (рис. 7.6).

Машина
X(координата) P(полоса) V(скорость)
двигаться

Рис. 7.6

Метод — это процедура или функция, принадлежащая классу объектов.

Другими словами, метод — это некоторое действие, которое могут выполнять все объекты класса.

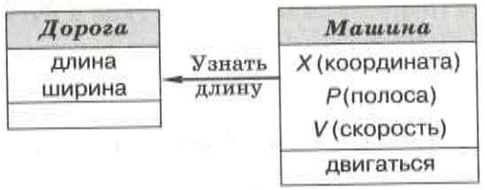


Рис. 7.7

Пока мы построили только модели отдельных объектов (точнее, классов). Чтобы моделировать всю систему, нужно разобраться, как эти объекты взаимодействуют. Объект-машина должен уметь «определить», в каком месте дороги он находится. Для этого машина должна обращаться к объекту «дорога», запрашивая длину дороги (см. стрелку на рис. 7.7).

Схема на рис. 7.7 определяет:

- свойства объектов;
- операции, которые они могут выполнять;
- связи (обмен данными) между объектами.

В то же время мы пока ничего не говорили о том, как устроены объекты и как именно они будут выполнять эти операции, и это не случайно. Согласно принципам ООП, ни один объект не должен зависеть от внутреннего устройства и алгоритмов работы других объектов. Поэтому, построив такую схему, можно поручить разработку двух классов объектов двум программистам, каждый из которых может решать свою задачу независимо от другого. Важно только, чтобы все они чётко соблюдали **интерфейс** — правила, описывающие взаимодействие «своих» объектов с остальными.

Вопросы и задания

1. Какие этапы входят в объектно-ориентированный анализ?
2. Что такое объект?
3. Что такое класс? Чем различаются понятия «класс» и «объект»?
4. Что такое метод?
5. Как изображаются классы на схеме?
6. Почему при объектно-ориентированном анализе не уточняют, как именно объекты будут устроены и как они будут решать свои задачи?

Задачи

1. Подумайте, какими свойствами и методами могли бы обладать объекты следующих классов: *Ученик*, *Учитель*, *Школа*, *Экзамен*, *Турнир*, *Урок*, *Страна*, *Браузер*. Придумайте свои классы объектов и выполните их анализ.
2. Добавьте в рассмотренную в параграфе модель светофора (на дороге их может быть много). Подумайте, какие свойства и методы должны быть у объектов класса *Светофор*. Как могут быть связаны классы *Дорога*, *Светофор* и *Машина* (сравните разные варианты)?
3. Придумайте свою задачу и выполните её объектно-ориентированный анализ. Примеры: моделирование работы магазина, банка, библиотеки и т. п.

§ 49
Создание объектов в программе

Класс Дорога

Объектно-ориентированная программа начинается с описания классов объектов. Класс в программе — это новый тип данных. Как и структура (см. § 39), класс — это сложный тип данных, который может объединять переменные различного типа в единый блок. Однако, в отличие от структуры, класс содержит не только данные, но и методы работы с ними (процедуры и функции).

В нашей программе самый простой класс — это *Дорога* (англ. *road*). Объекты этого класса имеют два свойства: длину (англ. *length*), которая может быть вещественным числом, и ширину (англ. *width*) — количество полос, целое число. Для хранения значений свойств используются переменные, принадлежащие объекту, которые называются полями.

Поле — это переменная, принадлежащая объекту.

Значения полей описывают *состояние* объекта, а методы — его *поведение*.

Описание класса *Дорога* в программе на объектной версии Паскаля (здесь имеется в виду *FreePascal* или *Delphi*) выглядит так:

```
type TRoad = class
       Length: real;
       Width: integer
     end;
```

Эти строки вводят новый тип данных — класс TRoad[1], т. е. сообщают компилятору, что в программе, возможно, будут использоваться объекты этого типа. При этом в памяти не создаётся ни одного объекта. Это описание похоже на чертёж, по которому в нужный момент можно построить сколько угодно таких объектов.

Если мы хотим работать с объектом класса TRoad, в программе нужно объявить соответствующую переменную:

```
var road: TRoad;
```

[1] Буква «T» в начале названия класса — это сокращение от слова *type*.

Однако и это ещё не объект, а ссылка (указатель), т. е. переменная, в которой можно сохранить адрес любого объекта класса TRoad. Чтобы создать сам объект в памяти, нужно вызвать специальный *метод* Create, который называется **конструктором**. Адрес нового объекта записываем в переменную road:

```
road:=TRoad.Create;
```

При вызове этого метода его название записывается через точку после названия класса.

Созданный объект относится к классу TRoad, поэтому его называют **экземпляром класса** TRoad.

При описании класса мы ничего не говорили о методе Create. Он добавляется ко всем классам по умолчанию, при его вызове все переменные объекта заполняются нулями[1].

Конструктор — это метод класса, который вызывается для создания объекта этого класса.

Свойства дороги можно изменять с помощью точечной записи, с которой вы познакомились, работая со структурами:

```
road.Length:=60;
road.Width:=3;
```

Полная программа, которая создаёт объект «дорога» (и больше ничего не делает), выглядит так:

```
{$mode objfpc}
type TRoad = class
       Length: real;
       Width: integer
     end;
var road: TRoad;
begin
  road:=TRoad.Create;
  road.Length:=60;
  road.Width:=3
end.
```

[1] Хотя стандарта на этот счёт нет, так сделано во всех объектных реализациях Паскаля.

Строка {$mode objfpc} по форме похожа на комментарий, потому что заключена в фигурные скобки. Однако для компилятора это команда перейти в режим работы с объектами (англ. *mode* — режим; *object* — объект; FPC — Free Pascal Compiler — свободно распространяемый компилятор Паскаля).

Начальные значения полей можно задавать прямо при создании объекта. Для этого нужно добавить в описание класса новый конструктор. Конструктору будет передаваться два параметра — начальные значения длины и ширины дороги:

```
type TRoad = class
       Length: real;
       Width: integer;
       constructor Create(length0: real;
                          width0: integer)
     end;
```

Реализация конструктора (т. е. программа, которая выполняется при его вызове) может выглядеть так:

```
constructor TRoad.Create(length0: real;
                         width0: integer);
begin
  if length0>0 then
       Length:=length0
  else Length:=1;
  if width0>0 then
       Width:=width0
  else Width:=1;
end;
```

Здесь проверяется правильность переданных параметров, чтобы по ошибке длина и ширина дороги не оказались нулевыми или отрицательными[1]. Теперь создавать объект будет проще:

```
road:=TRoad.Create(60, 3);
```

Длина этой дороги — 60 единиц, она содержит 3 полосы.

Таким образом, класс выполняет роль «фабрики», которая при вызове конструктора «выпускает» (создаёт) объекты «по чертежу» (описанию класса).

[1] Конечно, в реальной программе при передаче неправильных данных нужно выдавать сообщение об ошибке.

Класс Машина

Теперь можно описать класс *Машина* (в программе назовём его TCar). Объекты класса TCar имеют три свойства и один метод — процедуру move. Координата X и скорость V — это вещественные значения, а номер полосы P — целое.

```
type TCar = class
        X, V: real;
        P: integer;
        road: TRoad;
        procedure move;
        constructor Create(road0: TRoad;
                           p0: integer; v0: real);
    end;
```

Так как объекты-машины должны обращаться к объекту «дорога», в область данных включено дополнительное поле *road*. Конечно, это не значит, что в состав машины входит дорога. Напомним, что это только ссылка, и сразу после создания объекта-машины нужно записать в неё адрес заранее созданного объекта «дорога». Эту привязку удобно сделать прямо в конструкторе, при создании объекта. Заодно мы определяем полосу движения и скорость, а начальная координата X автоматически устанавливается равной нулю:

```
constructor TCar.Create(road0: TRoad;
                        p0: integer; v0: real);
begin
  road:=road0; P:=p0; V:=v0
end;
```

Теперь займёмся реализацией (программированием) метода move (англ. *move* — двигаться). В этом методе нужно вычислить новую координату X машины и, если она находится за пределами дороги, установить её в ноль (машина появляется слева на той же полосе). Изменение координаты при равномерном движении описывается формулой

$$X = X_0 + V \cdot \Delta t,$$

где X_0 и X — начальная и конечная координаты, V — скорость, а Δt — время движения. Вспомним, что любое моделирование физических процессов на компьютере происходит в дискретном времени, с некоторым интервалом дискретизации. Для простоты можно измерять время в этих интервалах, а за скорость V принять расстояние, проходимое машиной за один интервал. Тогда метод **move**, описывающий изменение положения машины за один интервал ($\Delta t = 1$), может выглядеть так:

```
procedure TCar.move;
begin
  X:=X+V;
  if X > road.Length then X:=0
end;
```


Основная программа

В основной программе объявим массив объектов-машин:

```
const N=3;
var cars: array [1..N] of TCar;
```

Как вы помните, это ещё не объекты, а ссылки — переменные, в которые можно записать адреса объектов класса TCar. Теперь нужно создать сами объекты:

```
var i: integer;
...
for i:=1 to N do
  cars[i]:=TCar.Create(road, i, 2.0*i);
```

При вызове конструктора задаются три параметра: адрес объекта «дорога» (его нужно создать до выполнения этого цикла), номер полосы и скорость. В приведённом варианте машина на полосе с номером i движется со скоростью $2i$ единиц за один интервал моделирования.

Сам цикл моделирования получается очень простой: на каждом шаге вызывается метод move для каждой машины:

```
repeat
  for i:=1 to N do cars[i].move;
until keypressed;
```

Этот цикл закончится тогда, когда пользователь нажмёт любую клавишу и функция keypressed (расположенная в стандартном модуле Crt) вернёт значение True.

Полностью основная программа выглядит так:

```
uses Crt;
const N = 3;
var road: TRoad;
    cars: array [1..N] of TCar;
    i: integer;
```

```
begin
  road:=TRoad.Create(60, N);
  for i:=1 to N do
    cars[i]:=TCar.Create(road, i, 2.0*i);
  repeat
    for i:=1 to N do cars[i].move;
  until keypressed
end.
```

Можно ли было написать такую же программу, не используя объекты? Конечно, да. И она получилась бы короче, чем наш объектный вариант (с учётом описания классов). В чём же преимущества ООП?

Мы уже отмечали, что ООП — это средство разработки больших программ, моделирующих работу сложных систем. В этом случае очень важно, что при использовании объектного подхода:

- основная программа, описывающая решение задачи в целом, получается простой и понятной; все команды напоминают действия в реальном мире («машина № 2, вперёд!»);
- разработку отдельных классов объектов можно поручить разным программистам, при этом каждый может работать независимо от других;
- если объекты классов *Дорога* и *Машина* понадобятся в других разработках, можно будет легко использовать уже готовые классы.

Вопросы и задания

1. Что такое поле в описании класса объекта?
2. Как объявляется класс объектов в программе?
3. Как объявляется переменная для работы с объектом некоторого класса? Что в ней хранится?
4. Как в памяти создаётся экземпляр класса (объект)?
5. Что такое конструктор?
6. Что такое точечная запись? Как она используется при работе с объектами?
7. Как можно задать начальные значения для полей объекта?
8. Почему в методе `TCar.move` (пример, разобранный в параграфе) не объявлены переменные `X` и `V`?
9. Сравните преимущества и недостатки решения рассмотренной задачи «классическим» способом и с помощью ООП. Сделайте выводы.

Подготовьте сообщение

а) «Классы в языке Си»
б) «Классы в языке Javascript»
в) «Классы в языке Python»

Задачи

1. Добавьте в рассмотренную в параграфе программу операторы, позволяющие изобразить на экране перемещение машин (в текстовом или графическом режиме). Подумайте, какие методы можно добавить для этого в класс `TCar`.

*2. Добавьте в модель из параграфа светофор, который переключается автоматически по программе (например, 5 с горит красный свет, затем 1 с — жёлтый, потом 5 с — зелёный и т. д.). Измените классы так, чтобы машина запрашивала у объекта *Дорога* местоположение ближайшего светофора, а затем обращалась к светофору для того, чтобы узнать, какой сигнал горит. Машины должны останавливаться у светофора с запрещающим сигналом.

§ 50 Скрытие внутреннего устройства

Во время построения объектной модели задачи мы выделили отдельные объекты, которые для обмена данными друг с другом используют *интерфейс* — внешние свойства и методы. При этом все внутренние данные и детали внутреннего устройства объекта должны быть скрыты от «внешнего мира». Такой подход позволяет:

- обезопасить внутренние данные (поля) объекта от изменений (возможно, разрушительных) со стороны других объектов;
- проверять данные, поступающие от других объектов, на корректность, тем самым повышая надёжность программы;
- переделывать внутреннюю структуру и код объекта любым способом, не меняя его внешние характеристики (интерфейс); при этом никакой переделки других объектов не требуется.

Скрытие внутреннего устройства объектов называют **инкапсуляцией** («помещение в капсулу»).

Заметим, что в объектно-ориентированном программировании инкапсуляцией также называют объединение данных и методов работы с ними в одном объекте.

Разберём простой пример. Во многих системах программирования есть класс, описывающий свойства «пера», которое используется при рисовании линий в графическом режиме. Назовём этот класс TPen, в простейшем варианте он будет содержать только одно поле *Color*, которое определяет цвет. Будем хранить код цвета в виде символьной строки, в которой записан шестнадцатеричный код составляющих модели RGB. Например, 'FF00FF' — это фиолетовый цвет, потому что красная (R) и синяя (B) составляющие равны $FF_{16} = 255$, а зелёной составляющей нет вообще. Класс можно объявить так:

```
type TPen = class
      Color: string;
    end;
```

По умолчанию все члены класса (поля и методы) открытые, общедоступные (англ. *public*). Те элементы, которые нужно скрыть, в описании класса помещают в «частный» раздел (англ. *private*), например, так:

```
type TPen = class
    private
      FColor: string;
    end;
```

В этом примере поле FColor **закрытое**. Имена всех закрытых полей далее будем начинать с буквы «F» (от англ. *field* — поле). К закрытым полям нельзя обратиться извне (это могут делать только методы самого объекта), поэтому теперь невозможно не только изменить внутренние данные объекта, но и просто узнать их значения. Чтобы решить эту проблему, нужно добавить к классу ещё два метода: один из них будет возвращать текущее значение поля *FColor*, а второй — присваивать полю новое значение. Эти методы доступа назовем getColor (в переводе с англ. — получить Color) и setColor (в переводе с англ. — установить Color):

```
type TPen=class
    private
      FColor: string;
    public
      function getColor: string;
      procedure setColor(newColor: string);
    end;
```


Обратите внимание, что оба метода находятся в секции **public (общедоступные)**.

Что же улучшилось по сравнению с первым вариантом (когда поле было открытым)? Согласно принципам ООП, внутренние поля объекта должны быть доступны только с помощью методов. В этом случае внутреннее представление данных может как угодно отличаться от того, как другие объекты «видят» эти данные.

В простейшем случае метод getColor можно написать так:

```
function TPen.getColor: string;
begin
  Result:=FColor;
end;
```

В методе setColor мы можем обрабатывать ошибки, не разрешая присваивать полю недопустимые значения. Например, установим, что символьная строка с кодом цвета, передаваемая нашему объекту, должна состоять из шести символов. Если эти условия не выполняются, будем записывать в поле FColor код чёрного цвета '000000':

```
procedure TPen.setColor(newColor: string);
begin
  if Length(newColor)<>6 then
    FColor:='000000' { если ошибка, то чёрный цвет}
  else FColor:=newColor
end;
```

Теперь если pen — это объект класса TPen, то для установки и чтения его цвета нужно использовать показанные выше методы:

```
pen.setColor ('FFFF00'); {изменение цвета}
writeln( 'цвет пера: ', pen.getColor );
                        {получение цвета}
```

Итак, мы скрыли внутренние данные, но одновременно обращение к свойствам стало выглядеть довольно неуклюже: вместо pen.Color:='FFFF00' теперь нужно писать pen.setColor('FFFF00'). Чтобы упростить запись, во многие объектно-ориентированные языки программирования ввели понятие **свойства** (англ. *property*), которое внешне выглядит как переменная объекта, но на самом деле при записи и чтении свойства вызываются методы объекта.

Свойство — это способ доступа к внутреннему состоянию объекта, имитирующий обращение к его внутренней переменной.

Свойство color в нашем случае можно определить так:

```
type TPen = class
     private
       FColor: string;
       function getColor: string;
       procedure setColor(newColor: string);
     public
       property color: string read getColor
                              write setColor;
     end;
```

Здесь методы getColor и setColor перенесены в раздел **private**, т. е. закрыты от других объектов. Однако есть общедоступное свойство color строкового типа:

```
property color: string read getColor
                       write setColor;
```

При чтении этого свойства (англ. *read*) вызывается метод getColor, а при записи нового значения (англ. *write*) — метод setColor. В программе можно использовать это свойство так:

```
pen.color:='FFFF00';    {изменение цвета}
writeln( 'цвет пера: ', pen.Color );
                        {получение цвета}
```

Поскольку приведённая выше функция getColor просто возвращает значение поля *FColor* и не выполняет никаких дополнительных действий, можно было вообще удалить метод getColor и объявить свойство так:

```
property color: string read FColor write setColor;
```

В этом случае при чтении выполняется прямой доступ к полю.

Таким образом, с помощью свойства color другие объекты могут изменять и читать цвет объектов класса TPen. Для обмена данными с «внешним миром» важно лишь то, что свойство color — символьного типа, и оно содержит 6-символьный код цвета. При этом внутреннее устройство объектов TPen может быть любым, и его можно менять как угодно. Покажем это на примере.

Хранение цвета в виде символьной строки неэкономно и неудобно, так как большинство стандартных функций используют числовые коды цвета. Поэтому лучше хранить код цвета как целое число, и поле *FColor* сделать целого типа:

```
FColor: integer;
```

При этом необходимо поменять методы getColor и setColor, которые непосредственно работают с этим полем:

```
function TPen.getColor: string;
begin
  Result:=IntToHex(FColor,6);
end;
procedure TPen.setColor(newColor: string);
begin
  if Length(newColor)<>6 then
    FColor:=0     {если ошибка, то чёрный цвет}
  else
    FColor:=StrToInt('$'+newColor)
end;
```

Для перевода числового кода в символьную запись используется функция IntToHex, входящая в библиотеку FreePascal (модуль SysUtils). Её второй параметр — количество цифр в шестнадцатеричном коде числа. Обратный перевод выполняет функция StrToInt. Для того чтобы указать, что число записано в шестнадцатеричной системе, перед ним добавляют символ $.

В этом примере мы принципиально изменили внутреннее устройство объекта: заменили строковое поле на целочисленное. Однако другие объекты даже не «догадаются» о такой замене, потому что сохранился интерфейс — свойство color по-прежнему имеет строковый тип. Таким образом, инкапсуляция позволяет как угодно изменять внутреннее устройство объектов, не затрагивая интерфейс. При этом все остальные объекты изменять не требуется.

Иногда не нужно разрешать другим объектам менять свойство, т. е. требуется сделать свойство «только для чтения» (англ. *read-only*). Пусть, например, мы строим программную модель автомобиля. Как правило, другие объекты не могут непосредственно менять его скорость, однако могут получить информацию о ней — «прочитать» значение скорости. При описании такого свойства слово write и название метода записи не указывают вообще:

```
type TCar = class
    private
      Fv: real;
      ...
    public
      property v: real read Fv;
      ...
    end;
```

Таким образом, доступ к внутренним данным объекта возможен, как правило, только с помощью методов. Применение свойств (**property**) очень удобно, потому что позволяет использовать ту же форму записи, что и при работе с общедоступной переменной объекта.

При использовании скрытия данных длина программы чаще всего увеличивается, однако мы получаем и важные преимущества. Код, связанный с объектом, разделён на две части: общедоступную часть (секция **public**) и закрытую (**private**). Объект взаимодействует с другими объектами только с помощью своих общедоступных свойств и методов (интерфейса) — рис. 7.8. Поэтому при сохранении интерфейса можно как угодно менять внутреннюю структуру данных и код методов, и это никак не будет влиять на другие объекты. Подчеркнём, что всё это становится действительно важно, когда разрабатывается большая программа и необходимо обеспечить её надёжность.

Рис. 7.8

Вопросы и задания

1. Что такое интерфейс объекта?
2. Что такое инкапсуляция? Каковы её цели?
3. Чем различаются секции **public** и **private** в описании классов? Как определить, в какую из них поместить свойство или метод?
4. Почему рекомендуют делать доступ к полям объекта только с помощью методов?
5. Что такое свойство? Зачем во многие языки программирования введено это понятие?
6. Можно ли с помощью свойства обращаться напрямую к полю объекта, не используя метод?

7. Почему методы доступа, которые использует свойство, делают закрытыми?
8. Зачем нужны свойства «только для чтения»? Приведите примеры.
9. Подумайте, в каких ситуациях может быть нужно свойство «только для записи» (которое нельзя прочитать). Как ввести такое свойство в описание класса? Приведите примеры.

Подготовьте сообщение

а) «Инкапсуляция в языке Си»
б) «Инкапсуляция в языке Javascript»
в) «Инкапсуляция в языке Python»

Задача

Измените построенную ранее программу моделирования движения так, чтобы все поля у объектов были закрытыми. Используйте свойства для доступа к данным.

§ 51 Иерархия классов

Классификации

Как в науке, так и в быту, важную роль играет **классификация** — разделение изучаемых объектов на группы (классы), объединённые общими признаками. Прежде всего это нужно для того, чтобы не запутаться в большом количестве данных и не описывать каждый объект заново.

Например, есть много видов фруктов[1] (яблоки, груши, сливы, апельсины и т. д.), но все они обладают некоторыми общими свойствами. Если перевести этот пример на язык ООП, класс *Яблоко* — *это* **подкласс** (производный класс, **класс-наследник**, потомок) класса *Фрукт*, а класс *Фрукт* — это **базовый класс** (суперкласс, класс-предок) для класса *Яблоко* (а также для классов *Груша*, *Слива*, *Апельсин* и др.).

Стрелка на схеме (рис. 7.9) обозначает наследование. Например, класс *Яблоко* — это наследник класса *Фрукт*.

[1] Фруктами называют сочные съедобные плоды деревьев и кустарников.

Рис. 7.9

Классический пример научной классификации — классификация животных или растений. Как вы знаете, она представляет собой *иерархию* (многоуровневую структуру). Например, горный клевер относится к роду *Клевер* семейства *Бобовые* класса *Двудольные* и т. д. Говоря на языке ООП, класс *Горный клевер* — это наследник класса *Клевер*, а тот, в свою очередь, — наследник класса *Бобовые*, который является наследником класса *Двудольные* и т. д.

Класс *Б* является наследником класса *А*, если можно сказать, что *Б* — это разновидность *А*.

Например, можно сказать, что яблоко — это фрукт, а горный клевер — одно из растений семейства *Двудольные*.

В то же время мы не можем сказать, что «двигатель — это разновидность машины», поэтому класс *Двигатель* не является наследником класса *Машина*. Двигатель — это составная часть машины, поэтому объект класса *Машина* содержит в себе объект класса *Двигатель*. Отношения между двигателем и машиной — это отношение «часть — целое».

Иерархия логических элементов

Рассмотрим такую задачу: составить программу для моделирования управляющих схем, построенных на логических элементах (см. главу 3 в учебнике для 10 класса). Нам нужно «собрать» заданную схему и построить её таблицу истинности.

Как вы уже знаете, перед тем, как программировать, нужно выполнить объектно-ориентированный анализ. Все объекты, из которых состоит схема, — это логические элементы, однако они могут быть разными (НЕ, И, ИЛИ и другие). Попробуем выделить общие свойства и методы всех логических элементов.

Ограничимся только элементами, у которых один или два входа. Тогда иерархия классов может выглядеть, как показано на рис. 7.10.

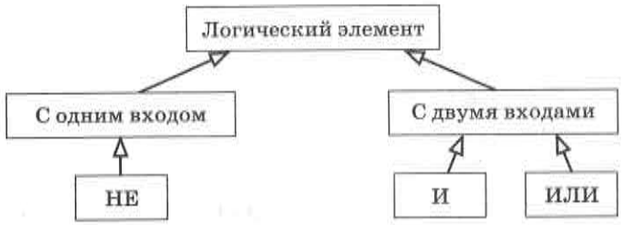


Рис. 7.10

Среди всех элементов с двумя входами мы показали только элементы «И» и «ИЛИ», остальные вы можете добавить самостоятельно.

Итак, для того чтобы не описывать несколько раз одно и то же, классы в программе должны быть построены в виде иерархии. Теперь можно дать определение объектно-ориентированного программирования.

Объектно-ориентированное программирование — это такой подход к программированию, при котором программа представляет собой множество взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию **наследования**.

Базовый класс

Построим первый вариант описания класса *Логический элемент* (`TLogElement`). Обозначим его входы как *In1* и *In2*, а выход назовём *Res* (от англ. *result* — результат) (рис. 7.11). Здесь состояние логического элемента определяется тремя величинами (*In1*, *In2* и *Res*). С помощью такого базового класса можно моделировать не только статические элементы (как НЕ, И, ИЛИ и т. п.), но и элементы с памятью (например, триггеры).

ЛогЭлемент
In1 (вход 1)
In2 (вход 2)
Res (результат)
calc

Рис. 7.11

Любой логический элемент должен уметь вычислять значение выхода по известным входам, для этого введём в класс метод calc:

```
type
  TLogElement = class
    In1, In2: boolean;
    Res: boolean;
    procedure calc
  end;
```

В таком варианте все данные открытые (общедоступные). Чтобы защитить внутреннее устройство объекта, скроем внутренние поля (добавим при этом в их названия первую букву «F») и введём свойства:

```
type
  TLogElement = class
  private
    FIn1, FIn2: boolean;
    FRes: boolean;
    procedure setIn1(newIn1: boolean);
    procedure setIn2(newIn2: boolean);
    procedure calc;
  public
    property In1: boolean read FIn1 write setIn1;
    property In2: boolean read FIn2 write setIn2;
    property Res: boolean read FRes
  end;
```

Обратите внимание, что свойство Res — это свойство только для чтения, и другие объекты не могут его менять. Кроме того, мы поместили процедуру calc в скрытый раздел (**private**), потому что пересчёт результата должен выполняться автоматически при изменении любого входного сигнала (другие объекты не должны об этом беспокоиться).

Несложно написать процедуру setIn1 (и аналогичную ей процедуру setIn2), в ней новое входное значение присваивается полю и сразу пересчитывается результат:

```
procedure TLogElement.setIn1(newIn1: boolean);
begin
  FIn1:=newIn1;
  calc
end;
```


Если внимательно проанализировать построенное описание класса, можно выявить несколько проблем. Во-первых, элемент НЕ имеет только один вход, поэтому не хотелось бы для него открывать доступ к свойству *In2* (это не нужно и может привести к ошибкам).

Во-вторых, процедуру calc невозможно написать, пока мы не знаем, какой именно логический элемент моделируется. Вместе с тем мы знаем, что такую процедуру имеет любой логический элемент, т. е. она должна принадлежать именно классу TLogElement. Здесь можно написать процедуру-«заглушку» (которая ничего не делает):

```
procedure TLogElement.calc;
begin
end;
```

Но нужно как-то дать возможность классам-наследникам изменить этот метод так, чтобы он выполнял нужную операцию. Такой метод называется *виртуальным*. Более точное определение этого понятия мы дадим несколько позже.

Классы-наследники могут по-разному реализовывать один и тот же метод. Такая возможность называется полиморфизмом.

Полиморфизм (от греч. πολυ — много, и μορφη — форма) — это возможность классов-наследников по-разному реализовывать метод, описанный для класса-предка.

Мы уже говорили о том, что метод calc не нужно делать общедоступным (**public**). В то же время его нельзя делать закрытым (**private**), потому что в этом случае он не будет доступен классам-наследникам. В таких случаях в описании класса используется третий блок (кроме **private** и **public**), который называется **protected** (защищённый). Данные и методы в этом блоке доступны для классов-наследников, но недоступны для других классов. В этот же блок **protected** мы переместим объявление поля **FRes** (его будут менять наследники в процедуре calc) и свойства *In2* — оно будет скрыто для элемента «НЕ», а элементы с двумя входами его «откроют» (чуть позже).

```
type
TLogElement = class
  private
    FIn1, FIn2: boolean;
```

```
    procedure setIn1(newIn1: boolean);
    procedure setIn2(newIn2: boolean);
  protected
    FRes: boolean;
    property In2: boolean read FIn2 write setIn2;
    procedure calc; virtual; abstract;
  public
    property In1: boolean read FIn1 write setIn1;
    property Res: boolean read FRes;
end;
```

Обратите внимание на объявление метода `calc`: после него стоят слова **virtual** (виртуальный) и **abstract** (в переводе с англ. — абстрактный). Описатель **virtual** говорит о том, что метод `calc` — виртуальный, и классы-наследники могут его переопределять. Как уже отмечалось, мы должны объявить этот метод (ввести его в описание класса), поскольку он должен быть у любого логического элемента. В то же время невозможно написать процедуру `calc`, пока неизвестен тип логического элемента. Такой метод называется абстрактным и обозначается описателем **abstract**. Для абстрактного метода не нужно ставить «заглушку».

Абстрактный метод — это метод класса, который объявляется, но не реализуется в классе.

Более того, не существует логического элемента «вообще», как не существует «просто фрукта», не относящегося к какому-то виду. Такой класс в ООП называется абстрактным. Его отличительная черта — хотя бы один абстрактный (нереализованный) метод.

Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод.

Итак, полученный класс `TLogElement` — это абстрактный класс (компилятор определит это автоматически). *Его можно использовать только для разработки классов-наследников*, создать в программе объект этого класса нельзя.

Чтобы класс-наследник не был абстрактным, он должен переопределить все абстрактные методы предка, в данном случае метод `calc`. Как это сделать, вы увидите в следующем пункте.

Классы-наследники

Теперь займёмся классами-наследниками от `TLogElement`. Поскольку у нас будет единственный элемент с одним входом (НЕ), сделаем его наследником прямо от `TLogElement` (не будем вводить специальный класс «элемент с одним входом»).

```
type
  TNot = class(TLogElement)
  protected
    procedure calc; override
  end;
```

После слова **class** в скобках указано название базового класса. Все объекты класса `TNot` обладают всеми свойствами и методами класса `TLogElement`.

Новый класс **переопределяет** метод `calc`, на это указывает слово **override** (в переводе с англ. — перекрыть). Заметим, что у базового класса `TLogElement` этот метод не реализован — он абстрактный, поэтому в данном случае мы фактически программируем метод, объявленный в базовом классе. Для элемента «НЕ» он выглядит очень просто:

```
procedure TNot.calc;
begin
  FRes:=not In1
end;
```

Обратите внимание, что этот метод не может обратиться к закрытому полю *FIn1* базового класса, и вместо этого использует свойство *In1*.

Класс `TNot` уже не абстрактный, потому что абстрактный метод предка переопределён и теперь известно, что делать при вызове метода `calc`. Поэтому можно создавать объект этого класса и использовать его:

```
var n: TNot;
...
n:=TNot.Create;
n.In1:=False;
writeln(n.Res);
```

Остальные элементы имеют два входа и будут наследниками класса `TLog2In`:

```
type
  TLog2In = class(TLogElement)
```

```
  public
    property In2
  end;
```

Единственное, что делает этот класс, — переводит свойство In2 в раздел **public**, т. е. делает его общедоступным. Отметим, что видимость можно только повышать, т. е. нельзя, например, в наследнике сделать общедоступное свойство класса-предка закрытым или защищённым.

Класс TLog2In — это тоже абстрактный класс, потому что он не переопределил метод calc. Это сделают его наследники TAnd (элемент И) и TOr (элемент ИЛИ), которые определяют конкретные логические элементы:

```
type
  TAnd = class(TLog2In)
  protected
    procedure calc; override
  end;
  TOr = class(TLog2In)
  protected
    procedure calc; override
  end;
```

Реализация переопределённого метода calc для элемента «И» выглядит так:

```
procedure TAnd.calc;
begin
  FRes:=In1 and In2
end;
```

Для элемента «ИЛИ» этот метод определяется аналогично.

Обратим внимание на метод setIn1, введённый в базовом классе:

```
procedure TLogElement.setIn1(newIn1: boolean);
begin
  FIn1:=newIn1;
  calc
end;
```

В нём вызывается метод calc, который пересчитывает значение на выходе логического элемента при изменении входа. Какой же метод будет вызван, если в базовом классе TLogElement он только объявлен, но не реализован?

Проблема в том, что для вызова любой процедуры нужно знать её адрес в памяти. Для обычных методов транслятор сразу записывает в машинный код нужный адрес, потому что он заранее известен. Это так называемое **статическое связывание** (связывание на этапе трансляции), при выполнении программы этот адрес не меняется.

В нашем случае адрес метода неизвестен: в классе TLogElement его нет вообще, а у каждого класса-наследника адрес метода calc свой собственный. Чтобы выйти из положения, используется **динамическое связывание**, т. е. адрес вызываемой процедуры определяется при выполнении программы, когда уже определён тип объекта, с которым мы работаем. Такой метод нужно объявлять виртуальным, что мы и сделали ранее. Это означает не только то, что его могут переопределять наследники, но и то, что будет использоваться динамическое связывание. Теперь можно дать полное определение виртуального метода.

Виртуальный метод — это метод базового класса, который могут переопределить классы-наследники, при этом конкретный адрес вызываемого метода определяется только при выполнении программы.

Теперь мы готовы к тому, чтобы создавать и использовать построенные логические элементы. Например, таблицу истинности для последовательного соединения элементов «И» и «НЕ» можно построить так:

```
var elNot: TNot;
    elAnd: TAnd;
    A, B: boolean;
begin
  elNot:=TNot.Create;
  elAnd:=TAnd.Create;
  writeln('| A | B | not(A&B) ');
  writeln('------------------');
  for A:=False to True do begin
    elAnd.In1:=A;
    for B:=False to True do begin
      elAnd.In2:=B;
```

```
      elNot.In1:=elAnd.res;
      writeln('| ', integer(A), ' | ', integer(B),
              ' | ', integer(elNot.Res))
    end
  end
end.
```

Сначала создаются два объекта — логические элементы НЕ (класс TNot) и И (класс TAnd). Далее в двойном цикле перебираются все возможные комбинации значений логических переменных A и B, они подаются на входы элемента И, а его выход — на вход элемента НЕ. Чтобы при выводе логических значений вместо False и True выводились более компактные обозначения 0 и 1, значения входов и выхода преобразуются к целому типу (integer).

Модульность

При разработке больших программ нужно разделить работу между программистами так, чтобы каждый делал свой независимый блок (модуль). Все подпрограммы, входящие в модуль, должны быть связаны друг с другом, но слабо связаны с другими процедурами и функциями. Такой подход используется как в классическом программировании, так и в ООП.

В нашей программе с логическими элементами в отдельный модуль можно вынести всё, что относится к логическим элементам. Модуль, содержащий классы логических элементов, на объектной версии языка Паскаль можно записать так:

```
unit log_elem;
{$mode objfpc}
interface
type
  TLogElement = class
  private
    FIn1, FIn2: boolean;
    procedure setIn1(newIn1: boolean);
    procedure setIn2(newIn2: boolean);
  protected
    FRes: boolean;
    property In2: boolean read FIn2 write setIn2;
    procedure calc; virtual; abstract;
  public
```



```
    property In1: boolean read FIn1 write setIn1;
    property Res: boolean read FRes
  end;
  TNot = class(TLogElement)
  protected
    procedure calc; override
  end;
  TLog2In = class(TLogElement)
  public
    property In2
  end;
  TAnd = class(TLog2In)
  protected
    procedure calc; override
  end;
  TOr = class(TLog2In)
  protected
    procedure calc; override
  end;
implementation
  procedure TLogElement.setIn1(newIn1: boolean);
  begin
    FIn1:=newIn1; calc
  end;
  procedure TLogElement.setIn2(newIn2: boolean);
  begin
    FIn2:=newIn2; calc
  end;
  procedure TNot.calc;
  begin
    FRes:=not In1
  end;
  procedure TAnd.calc;
  begin
    FRes:=In1 and In2
  end;
  procedure TOr.calc;
  begin
    FRes:=In1 or In2
  end;
end.
```

Чтобы использовать такой модуль, нужно подключить его в основной программе с помощью ключевого слова **uses**, после которого через запятую перечисляются все используемые модули:

```
program logic;
{$mode objfpc}
uses log_elem;
var elNot: TNot;
    elAnd: TAnd;
    ...
begin
  elNot:=TNot.Create;
  elAnd:=TAnd.Create;
  ...
end.
```

Сообщения между объектами

Когда логические элементы объединяются в сложную схему, желательно, чтобы передача сигналов между ними при изменении входных данных происходила автоматически. Для этого можно немного расширить базовый класс `TLogElement`, чтобы элементы могли передавать друг другу сообщения об изменении своего выхода.

Для простоты будем считать, что выход любого логического элемента может быть подключён к любому (но только одному!) входу другого логического элемента. Добавим к описанию класса два поля и один метод:

```
type
  TLogElement = class
  private
    FNextEl: TLogElement;
    FNextIn: integer;
    ...
  public
    procedure Link(nextElement: TLogElement;
                   nextIn: integer);
    ...
  end;
```

Поле *FNextEl* хранит ссылку на следующий логический элемент, а поле *FNextIn* — номер входа этого следующего элемента, к которому подключён выход данного элемента. С помощью общедоступного метода `Link` можно связать данный элемент со следующим:


```
procedure TLogElement.Link(nextElement: TLogElement;
                           nextIn: integer);
begin
  FNextEl:=nextElement;
  FNextIn:=nextIn
end;
```

Нужно немного изменить методы `setIn1` и `setIn2`: при изменении входа они должны не только пересчитывать выход данного элемента, но и отправлять сигнал на вход следующего

```
procedure TLogElement.setIn1(newIn1: boolean);
begin
  FIn1:=newIn1; calc;
  if FNextEl<>nil then
    case FNextIn of
      1: FNextEl.In1:=res;
      2: FNextEl.In2:=res
    end
end;
```

Условие `FNextEl<>nil` означает «если следующий элемент задан». Если он не был установлен, значение поля *FNextEl* будет равно `nil`, и никакие дополнительные действия не выполняются.

С учётом этих изменений вывод таблицы истинности функции И-НЕ можно записать так (операторы вывода заменены многоточиями):

```
elNot:=TNot.Create;
elAnd:=TAnd.Create;
elAnd.Link(elNot, 1);
...
for A:=False to True do begin
  elAnd.In1:=A;
  for B:=False to True do begin
    elAnd.In2:=B;
    ...
  end
end;
```

Обратите внимание, что в самом начале мы установили связь элементов И и НЕ с помощью метода `Link` (связали выход элемента И с первым входом элемента НЕ). Далее в теле цикла обращения к элементу НЕ нет, потому что элемент И автоматически сообщит ему об изменении своего выхода.

Вопросы и задания

1. Что такое классификация? Зачем она нужна? Приведите примеры.
2. В каком случае можно сказать: «Класс *Б* — наследник класса *А*», а когда: «Объект класса *А* содержит объект класса *Б*»? Приведите примеры.
3. Что такое иерархия классов?
4. Объясните приведённую иерархию логических элементов. Обсудите её достоинства и недостатки.
5. Дайте полное определение ООП и объясните его.
6. Что такое базовый класс и класс-наследник? Какие синонимы используются для этих терминов?
7. На примере класса `TLogElement` (пример из параграфа) покажите, как выполнена инкапсуляция.
8. Что такое виртуальный метод?
9. Что такое полиморфизм?
10. Что такое абстрактный класс? Почему нельзя создать объект этого класса?
11. Как транслятор определяет, что тот или иной класс — абстрактный?
12. Что нужно сделать, чтобы класс-наследник абстрактного класса не был абстрактным?
13. Зачем нужен описатель **protected**? Чем он отличается от **private** и **public**?
14. Что означает описатель **override**?
15. Какие преимущества даёт применение модулей в программе?
16. Из каких частей состоит каждый модуль? Что включают в каждую из них?
17. Можно ли всё содержимое модуля включить в секцию **interface**? Чем это плохо?
18. Можно ли всё содержимое модуля включить в секцию **implementation**? Чем это плохо?
19. Объясните, как объекты могут передавать сообщения друг другу.

Подготовьте сообщение

а) «Иерархия классов в языке Си»

б) «Иерархия классов в языке Javascript»

в) «Иерархия классов в языке Python»

Задачи

1. Добавьте в описанную в параграфе иерархию классов элементы «исключающее ИЛИ», «И-НЕ» и «ИЛИ-НЕ».
2. «Соберите» в программе RS-триггер из двух логических элементов «ИЛИ-НЕ», постройте его таблицу истинности (обратите внимание на вариант, когда оба входа нулевые).

§ 52
Программы с графическим интерфейсом

Особенности современных прикладных программ

Большинство современных программ, предназначенных для пользователей, управляется с помощью графического интерфейса. Вы знакомы с понятиями «окно программы», «кнопка», «флажок», «поле ввода», «полоса прокрутки» и т. п. Такие оконные системы чаще всего построены на принципах объектно-ориентированного программирования, т. е. все элементы окон — это объекты, которые обмениваются данными, посылая друг другу сообщения.

Сообщение — это блок данных определённой структуры, который используется для обмена информацией между объектами.

В сообщении указываются:

- адресат (объект, которому посылается сообщение);
- числовой код (тип) сообщения;
- параметры (дополнительные данные), например координаты щелчка мышью или код нажатой клавиши.

Сообщение может быть **широковещательным**, в этом случае вместо адресата указывается особый код, и сообщение поступает всем объектам определённого типа (например, всем главным окнам программ).

В программах, которые мы писали раньше, последовательность действия заранее определена — основная программа выполняется строчка за строчкой, вызывая процедуры и функции, все ветвления выполняются с помощью условных операторов (рис. 7.12).

В современных программах порядок действий определяется пользователем, другими программами или поступлением новых данных из внешнего источника (например, из сети), поэтому классическая схема не подходит. Пользователь текстового редактора может щёлкать на любых кнопках и выбирать любые пункты меню в произвольном порядке. Программа-сервер, передающая данные с веб-сайта на компьютер пользователя, начинает действовать только при поступлении очередного запроса. При

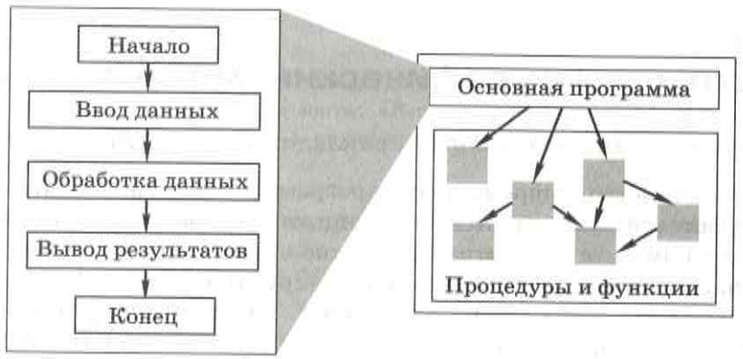


Рис. 7.12

программировании сетевых игр нужно учитывать взаимодействие многих объектов, информация о которых передаётся по сети в случайные моменты времени.

Во всех этих примерах программа должна «включаться в работу» только тогда, когда получит условный сигнал, т. е. произойдёт некоторое *событие* (изменение состояния).

Событие — это переход какого-либо объекта из одного состояния в другое.

События могут быть вызваны действиями пользователя (управление клавиатурой и мышью), сигналами от внешних устройств (переход принтера в состояние готовности, получение данных из сети), получением сообщения от другой программы. При наступлении событий объекты посылают друг другу сообщения.

Таким образом, весь ход выполнения современной программы определяется происходящими событиями, а не жёстко заданным алгоритмом. Поэтому говорят, что программы управляются событиями, а соответствующий стиль программирования называют **событийно-ориентированным**, т. е. основанным на обработке событий.

Нормальный режим работы событийно-ориентированной программы — цикл обработки сообщений. Все сообщения (от мыши, клавиатуры, драйверов устройств ввода и вывода и т. п.) сначала поступают в единую очередь сообщений операционной системы. Кроме того, для каждой программы операционная система создаёт отдельную очередь сообщений и помещает в неё все сообщения, предназначенные именно этой программе (рис. 7.13).

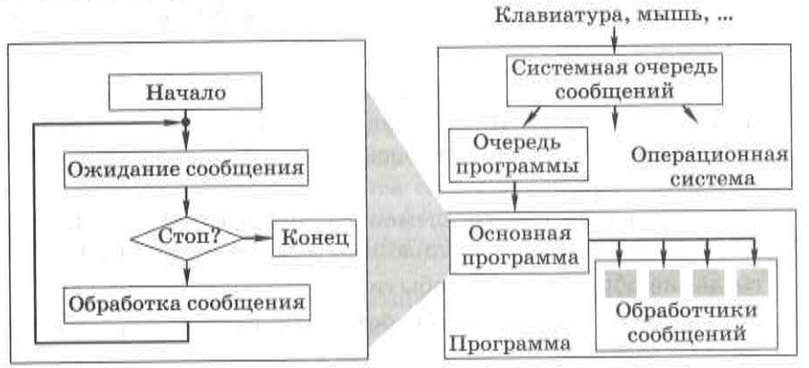


Рис. 7.13

Программа выбирает очередное сообщение из очереди и вызывает специальную процедуру — обработчик этого сообщения (если он есть). Когда пользователь закрывает окно программы, ей посылается специальное сообщение, при получении которого цикл (и работа всей программы) завершается.

Таким образом, главная задача программиста — написать содержание обработчиков всех нужных сообщений. Ещё раз подчеркнём, что последовательность их вызовов точно не определена, она может быть любой в зависимости от действий пользователя и сигналов, поступающих с внешних устройств.

RAD-среды для разработки программ

Разработка программ для оконных операционных систем до середины 1990-х гг. была довольно сложным и утомительным делом. Очень много усилий уходило на то, чтобы написать команды для создания интерфейса с пользователем: разместить элементы в окне программы, написать и правильно оформить обработчики сообщений. Значительную часть своего времени программист занимался трудоёмкой работой, которая почти никак не связана с решением главной задачи. Поэтому возникла естественная мысль — автоматизировать описание окон и их элементов так, чтобы весь интерфейс программы можно было построить без «ручного» программирования (чаще всего с помощью мыши), а человек думал бы о сути задачи, т. е. об алгоритмах обработки данных.

Такие системы программирования получили название **сред быстрой разработки приложений** — **RAD-сред** (от англ. *Rapid Application Development*). Разработка программы в RAD-системе состоит из следующих этапов:

- создание формы (так называют шаблон, по которому строится окно программы или диалога); при этом минимальный код добавляется автоматически и сразу получается работоспособная программа;
- расстановка на форме элементов интерфейса (полей ввода, кнопок, списков) с помощью мыши и настройка их свойств;
- создание обработчиков событий;
- написание алгоритмов обработки данных, которые выполняются при вызове обработчиков событий.

Обратите внимание, что при программировании в RAD-средах обычно говорят не об обработчиках сообщений, а об обработчиках событий. Событием в программе может быть не только нажатие клавиши или щелчок мышью, но и перемещение окна, изменение его размеров, начало и окончание выполнения расчётов и т. д. Некоторые сообщения, полученные от операционной системы, библиотека RAD-среды «транслирует» (переводит) в соответствующие события, а некоторые — нет. Более того, программист может вводить свои события и определять процедуры для их обработки.

Одной из первых сред быстрой разработки стала среда Delphi, разработанная фирмой Borland в 1994 г. Самая известная современная профессиональная RAD-система — *Microsoft Visual Studio* — поддерживает несколько языков программирования. Далее для выполнения практических работ мы будем использовать свободную RAD-среду *Lazarus* (lazarus.freepascal.org), которая во многом аналогична Delphi, но позволяет создавать кроссплатформенные программы (для операционных систем Windows, Linux, macOS и др.).

Среды RAD позволили существенно сократить время разработки программ. Однако нужно помнить, что любой инструмент — это только инструмент, который можно использовать грамотно или безграмотно. Использование среды RAD само по себе не гарантирует, что у вас автоматически получится хорошая программа с хорошим пользовательским интерфейсом.

Вопросы и задания

1. Что такое графический интерфейс?
2. Как связан графический интерфейс с объектно-ориентированным подходом к программированию?
3. Что такое сообщение? Какие данные в него входят?
4. Что такое широковещательное сообщение?
5. Что такое обработчик сообщения?
6. Чем принципиально отличаются современные программы от классических?
7. Что такое событие? Какое программирование называют событийно-ориентированным?
8. Как работает событийно-ориентированная программа?
9. Какие причины сделали необходимым создание сред быстрой разработки программ? В чем их преимущество? Приведите примеры.
10. Расскажите про этапы разработки программы в RAD-среде.
11. Объясните разницу между понятиями «событие» и «сообщение».

Подготовьте сообщение

а) «Обработка сообщений в операционных системах»
б) «Современные среды быстрой разработки программ»
в) «Программы с графическим интерфейсом на Python»

§ 53 Основы программирования в RAD-средах

Общий подход

В этом разделе мы продемонстрируем основные принципы программирования в RAD-средах на примере среды Lazarus, которая распространяется свободно и может работать в различных операционных системах — Windows, macOS, Linux. Тем не менее все изучаемые здесь принципы справедливы также и для других аналогичных программ, например Delphi и Visual Studio.

Разработка программы начинается с создания **проекта**. Так называется набор файлов, из которых компилятор строит исполняемый файл программы. В состав проекта обычно входят:

- проект (файл с расширением lpr[1], от Lazarus Project — проект Lazarus), в котором содержится основная программа;

[1] Здесь и далее расширения имён файлов указаны для среды Lazarus.

- настройки проекта (lpi, от Lazarus Project Information — информация о проекте Lazarus);
- модули, из которых состоит программа (pas);
- формы (lfm, от Lazarus Form — форма Lazarus) — описания внешнего вида и свойств окон и их элементов.

В программе с графическим интерфейсом может быть несколько окон, которые называют **формами**. С каждой формой связана пара файлов: в одном (с расширением lfm) хранятся данные о расположении и свойствах элементов интерфейса, а во втором (с расширением pas) — программный код обработчиков сообщений, связанных с этой формой.

Одна форма — *главная*, она появляется на экране при запуске программы. Когда пользователь закрывает главную форму, работа программы завершается.

Простейшая программа

Для создания проекта в Lazarus нужно выбрать пункт меню **Файл – Создать** и в появившемся окне отметить вариант **Проект – Приложение**. При этом создаётся вполне рабочая программа, которую сразу же можно запустить на выполнение клавишей **F9**.

При работе в среде Lazarus используются четыре окна (рис. 7.14):

- главное окно;
- окно Инспектора объектов;
- окно исходного кода;
- окно формы.

Главное окно среды (расположенное сверху) содержит меню, кнопки для быстрого вызова команд и **палитру (библиотеку) компонентов**. Компонентами называются готовые объекты (кнопки, поля ввода, списки и т. п.), которые можно использовать в программах.

Вся программа, согласно принципам ООП, строится на основе объектов. Для настройки свойств объектов используется окно **Инспектора объектов**, которое состоит из двух частей. В верхней части показано дерево объектов. В простейшей программе мы увидим всего один объект — форму с именем `Form1`. В нижней части окна несколько вкладок, самые важные из них — **Свойства**, где можно изменить общедоступные свойства объекта, и **События**, на которой из списка подходящих подпрограмм выбираются его обработчики событий.

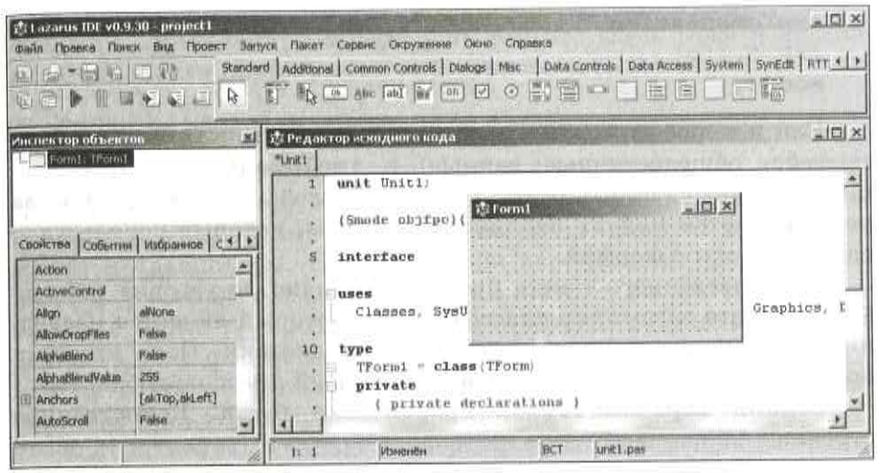


Рис. 7.14

В окно формы можно мышью перетаскивать компоненты с палитры и изменять их размеры и расположение. Таким образом, интерфейс программы полностью строится с помощью мыши.

С каждой формой связан программный модуль, в котором описываются классы и обработчики событий, используемые этой формой. Файлы формы и соответствующего ей модуля имеют одинаковое имя, но разные расширения. По умолчанию созданная главная форма программы называется `Form1`, а модуль — `Unit1`. Содержание модуля примерно такое:

```
unit Unit1;
interface
uses
  Classes, SysUtils, FileUtil, Forms, Controls,
  Graphics, Dialogs;
type
  TForm1=class(TForm)
  private
    {private declarations}
  public
    {public declarations}
  end;
var
  Form1: TForm1;
```

```
implementation
{$R *.lfm}
end.
```

Как и в любом модуле, здесь есть две секции: **interface** (интерфейс, общедоступные данные) и **implementation** (реализация — данные, скрытые от других модулей). После слова **uses** перечисляются модули библиотеки Lazarus, которые используются для работы с формой.

Из приведённого текста программы видно, что форма (объект Form1) — это экземпляр класса TForm1, который является наследником стандартного класса TForm из библиотеки. Пока никаких новых полей, свойств и методов в созданный класс не добавлено.

Секция **implementation** практически пуста. Единственная строчка

```
{$R *.lfm}
```

похожа на комментарий (в фигурных скобках), однако транслятор обращает на неё внимание. Эта команда подключает файл с тем же именем, что и у модуля, но с расширением lfm (описание формы и размещённых на ней компонентов).

Как вы знаете, модуль не может выполняться самостоятельно. Необходима ещё основная программа, которая находится в файле проекта. Для того чтобы увидеть его, нужно нажать клавиши Ctrl+F12 и выбрать файл с расширением lpr. Этот файл будет загружен в отдельную вкладку редактора:

```
program project1;
uses
  Interfaces, Forms, Unit1;
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run
end.
```

В начале файла подключаются используемые модули Interfaces и Forms из библиотеки Lazarus, а также модуль Unit1, связанный с нашей формой.

Вся программа — это объект с именем Application, который относится к классу TApplication. Этот объект создаётся автоматически при загрузке модуля Forms. В основной программе вызываются три его метода: Initialize (начальные установки),

CreateForm (создание формы) и Run (запуск). Цикл обработки сообщений скрыт внутри метода Run, так что здесь тоже использована инкапсуляция (скрытия внутреннего устройства).

Свойства объектов

Инспектор объектов позволяет просматривать и изменять свойства выделенного объекта (например, формы), а также устанавливать обработчики событий для этого объекта с помощью мыши и клавиатуры. Например, при перемещении формы изменяются её свойства Left (в переводе с англ. — левый) и Top (в переводе с англ. — верхний), которые задают координаты левого верхнего угла формы на экране. В то же время можно вручную изменить эти координаты, вводя новые значения в Инспекторе объектов (при этом форма передвигается).

Если изменять мышью размеры формы (перемещая её границы), то будут изменяться свойства Width (в переводе с англ. — ширина) и Height (в переводе с англ. — высота).

Свойство Name (в переводе с англ. — имя) — это название объекта-формы в программе (имя переменной). В имени можно использовать только латинские буквы, цифры и знак подчёркивания. Если изменить название формы в Инспекторе объектов, скажем, на MainForm, то это название автоматически изменится и в тексте модуля этой формы. Более того, название класса тоже изменится на TMainForm. Это означает, что многие изменения вносятся в код программы автоматически.

Перечислим ещё некоторые важные свойства формы:

- Caption — текст в заголовке окна;
- Color — цвет рабочей области;
- Font — шрифт надписей;
- Visible — видимость (да/нет).

Обработчики событий

На вкладке **События** перечислены все события, которые может обрабатывать форма. Названия их обработчиков в Инспекторе объектов начинаются с букв On (в переводе с англ. — в ответ на...). Чтобы создать обработчик, нужно дважды щёлкнуть мышью на поле справа от его названия. При этом открывается окно редактора, и в текст модуля автоматически добавляется пустой обработчик события (шаблон), в который остаётся только добавить нужные команды.

Рассмотрим простой пример. Вы знаете, что многие программы запрашивают подтверждение, когда пользователь завершает их работу. Для этого можно использовать обработчик OnCloseQuery (в переводе с англ. — запрос на закрытие). Обработчик, созданный двойным щелчком мышью, имеет вид:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
                                var CanClose: boolean);
begin

end;
```

Одновременно этот метод добавляется в описание класса TForm1 (выше секции **private**).

Как видно из заголовка процедуры, в обработчик передаются два параметра:

- Sender — ссылка на объект, от которого пришло сообщение о событии (в данном случае это будет сама форма);
- CanClose — изменяемый логический параметр, в который нужно записать результат запроса: истинное значение означает, что можно закрывать окно, ложное — что нельзя.

В Lazarus есть стандартная функция MessageDlg, которая выводит на экран запрос с несколькими кнопками (рис. 7.15) и позволяет получить ответ пользователя (код нажатой кнопки).

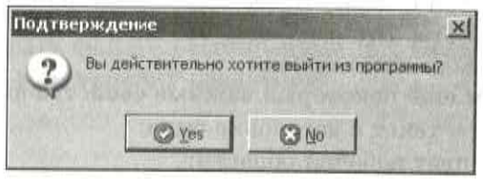


Рис. 7.15

В тело обработчика можно добавить условный оператор, который запишет в переменную CanClose значение True, если пользователь подтвердил выход из программы:

```
procedure TMainForm.FormCloseQuery(Sender: TObject;
                                   var CanClose: boolean);
var res: TModalResult;
begin
  res:=MessageDlg('Подтверждение',
      'Вы действительно хотите выйти из программы?',
       mtConfirmation, [mbYes,mbNo], 0);
```



```
  CanClose:=(res=mrYes)
end;
```

Здесь вызывается функция MessageDlg, и её результат записывается в переменную *res* типа TModalResult. Если это значение совпадает со встроенной константой *mrYes* (т. е. пользователь нажал на кнопку **Yes**), в переменную *CanClose* записывается значение True. Получив от обработчика такое значение (разрешение закрыть окно), программа завершается; иначе команда отменяется.

Функции MessageDlg передаются пять параметров:

- заголовок окна;
- текст вопроса;
- тип запроса, определяющий рисунок слева от текста:

mtError	ошибка;
mtWarning	предупреждение;
mtInformation	информация;
mtConfirmation	подтверждение;

- набор (множество) кнопок, которые появляются под текстом; в нашем случае это кнопки **Yes** и **No**, обозначенные константами *mbYes* и *mbNo*;
- номер раздела справочной системы, в котором есть объяснение этой ситуации (у нас нет справочной системы, поэтому ставим 0).

Итак, мы построили простейшую работоспособную программу и познакомились со средой Lazarus. В следующем параграфе вы узнаете, как работать с компонентами.

Вопросы и задания

1. В каком смысле используется термин «проект» в программировании?
2. Из каких файлов состоит типичный проект в RAD-среде?
3. Что такое форма? Почему для описания формы в Lazarus используются два файла?
4. Какие основные окна используются в среде Lazarus? Зачем они нужны?
5. Покажите, что программа, написанная с помощью Lazarus, состоит из объектов.
6. С помощью какого механизма транслятор «подключает» форму?
7. Где расположена основная программа в проекте Lazarus? Объясните все команды основной программы.
8. Почему в основной программе не виден цикл обработки сообщений?

9. Назовите некоторые важнейшие свойства формы. Какими способами можно их изменять?
10. Приведите примеры автоматического построения и изменения кода в RAD-среде.
11. Как создать новый обработчик события? Подумайте, можно ли сделать это вручную.
12. Как передаются параметры сообщения в обработчик?
13. Как можно вывести сообщение об ошибке на экран?

Подготовьте сообщение

«Простая программа на языке C# в Visual Studio»

Задача

Попробуйте изменять какие-нибудь свойства формы, построив обработчик ещё одного события (например, `OnShow` — вывод формы на экран; `OnClick` — щелчок мыши; `OnResize` — изменение размеров).

§ 54 Использование компонентов

Программа с компонентами

В главном окне Lazarus расположена так называемая **палитра компонентов** (рис. 7.16) — библиотека готовых объектов, которые можно добавить в свою программу, просто перетащив их мышью на форму.

Рис. 7.16

Компоненты разбиты на группы. Мы будем использовать компоненты из групп **Standard** (Стандартные), **Additional** (Дополнительные) и **Dialogs** (Диалоги). Каждый значок обозначает определённый компонент. Если задержать указатель мыши над значком компонента, в тексте всплывающей подсказки можно прочитать его название.

Построим простую программу для просмотра рисунков, используя готовые компоненты. В верхней части (формы) на панели разместим кнопку для загрузки файла и флажок-выключатель, который изменяет масштаб рисунка так, чтобы он вписывался в отведённое ему место (рис. 7.17).

Рис. 7.17

Создадим новый проект (как в предыдущем параграфе), изменим имя формы (свойство `Name`) на `MainForm`, а её заголовок (свойство `Caption`) — на «Просмотр рисунков».

Добавим на форму панель — компонент `TPanel` из группы **Standard** (Стандартные). Для этого можно перетащить эту кнопку на форму или щёлкнуть на кнопке и нарисовать прямоугольник, ограничивающий панель. Теперь размеры панели можно изменять, перетаскивая маркеры на границах или изменяя значения свойств `Width` (ширина) и `Height` (высота) в Инспекторе объектов. Панель можно перетаскивать мышью по форме. Хотелось бы, чтобы панель была всё время прижата к верхней границе окна и её размеры изменялись вместе с размерами окна. Для этого нужно установить свойство `Align` (выравнивание) равным `alTop` (англ. *align top* — выровнять по верху).

На созданной панели можно заметить надпись «Panel1», которая нам не нужна. Чтобы убрать её, нужно стереть значение свойства `Caption` (в переводе с англ. — заголовок) панели.

Теперь панель готова, на ней нужно разместить кнопку (компонент `TButton`) и флажок (компонент `TCheckBox`). На кнопке должна быть надпись «Открыть файл» (свойство `Caption`), а справа от флажка — текст «По размерам окна» (тоже свойство `Caption`) — рис. 7.18. Размеры и расположение компонентов нужно поменять с помощью мыши. Имена компонентов (свойства `Name`) тоже можно изменить, например, на `OpenBtn`

Рис. 7.18

и SizeCb (имена объектов строятся по тем же правилам, что и имена переменных в Паскале).

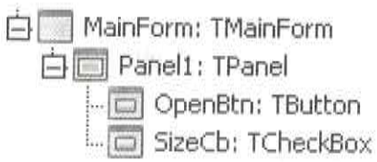


Рис. 7.19

В верхней части Инспектора объектов можно увидеть структуру объектов формы в виде дерева (рис. 7.19). Главный объект — это сама форма MainForm, она является *родительским объектом* для панели Panel1. Это означает, что при перемещении формы панель перемещается вместе с ней. Кроме того, форма отвечает за прорисовку всех дочерних элементов на экране.

В свою очередь, панель — это родительский объект для кнопки и флажка. Для того чтобы редактировать свойства и методы какого-то объекта в Инспекторе объектов, его можно выделить прямо на форме или в дереве объектов.

Теперь добавим на форму специальный объект TImage (группа **Additional**), который «умеет» отображать рисунки различных форматов. Для того чтобы он заполнял все свободное пространство (кроме панели), нужно установить для него выравнивание alClient (свойство Align). Изменим название объекта на Image.

Остается решить два вопроса:

1) как сделать выбор файла и загрузку его в компонент Image;
2) как подгонять размер рисунка по размеру формы.

К счастью, для этого достаточно использовать возможности готовых компонентов. На панели **Dialogs** (Диалоги) есть готовый компонент для выбора рисунка на диске, он называется TOpenPictureDialog. У него есть метод Execute — функция, которая вызывает *стандартный диалог* выбора файла и возвращает

логическое значение: True, если файл успешно выбран, и False, если пользователь отказался от выбора файла. Имя выбранного файла можно получить, прочитав свойство FileName этого компонента.

Добавим компонент TOpenPictureDialog на форму (в любое место). Это невизуальный компонент, его не будет видно во время выполнения программы. В Инспекторе объектов можно проверить, что для него родительским объектом будет сама форма. Для краткости изменим его название на OpenDlg.

Теперь в случае щелчка на кнопке нужно вызвать метод OpenDlg.Execute и, если он вернёт значение True, загрузить выбранный файл. Имя этого файла получаем как значение свойства OpenDlg.FileName.

Щелчок на кнопке — это событие, обработчик которого называется OnClick. Создадим шаблон этого обработчика в Инспекторе объектов и запишем в него команды:

```
if OpenDlg.Execute then
  Image.Picture.LoadFromFile( OpenDlg.FileName );
```

Поясним загрузку файла. Компонент Image имеет свойство Picture, в котором хранится изображение. Это тоже объект, у которого, в свою очередь, есть метод LoadFromFile (загрузить из файла), этому методу передаётся имя файла, выбранного пользователем. Теперь можно запустить программу и проверить, как она работает.

Обратите внимание, что изображение выводится в масштабе 1:1 независимо от размера окна. Флажок **По размерам окна** можно включать и выключать, но он никак не влияет на результат. Чтобы исправить ситуацию, будем использовать событие изменения состояния флажка, его обработчик называется OnChange (при изменении). У объекта Image есть логическое свойство Proportional (пропорциональный), если ему присвоить значение True, компонент Image сам выполнит подгонку размеров рисунка под размер свободной области. Таким образом, обработчик события OnChange компонента SizeCb содержит такой оператор:

```
Image.Proportional:=SizeCb.Checked;
```

Свойство Checked (в переводе с англ. — отмечен) — это логическое значение, определяющее состояние выключателя: если он включен, это свойство равно True, если выключен, то False.

Теперь можно запустить готовую программу и проверить её работу. Отметим следующие важные особенности:

- программа целиком состоит из объектов и основана на идеях ООП;
- мы построили программу практически без программирования;
- использование готовых компонентов скрывает от нас сложность выполняемых операций, поэтому скорость разработки программ значительно повышается.

Ввод и вывод данных

Во многих программах нужно, чтобы пользователь вводил текстовую или числовую информацию. Чаще всего для ввода данных применяют поле ввода — компонент TEdit (вкладка **Standard**). Для доступа к введённой строке используют его свойство Text (в переводе с англ. — текст).

Шрифт текста в поле ввода задаётся сложным свойством Font (шрифт). Это объект, у которого есть свои свойства, их список можно увидеть в Инспекторе объектов, если щёлкнуть на значке ⊞. Например, свойство Size — размер шрифта в пунктах, а Style — свойство-множество, в которое могут входить стили оформления fsBold (жирный), fsItalic (курсив), fsUnderline (подчёркнутый). Если установить шрифт для какого-то объекта, например для формы, все дочерние компоненты по умолчанию будут иметь такой же шрифт.

Программа, которую мы сейчас построим, будет переводить RGB-составляющие цвета в соответствующий шестнадцатеричный код, который используется для задания цвета в языке HTML (см. § 26).

На форме будут расположены (рис. 7.20):

- три поля ввода (в них пользователь может задать значения красной, зелёной и синей составляющих цвета в модели RGB);
- прямоугольник (компонент TShape из группы **Additional**), цвет которого изменяется согласно введённым значениям;
- несколько меток (компонентов TLabel).

Метки — это надписи, которые пользователь не может редактировать, однако их содержание можно изменять из программы через свойство Caption.

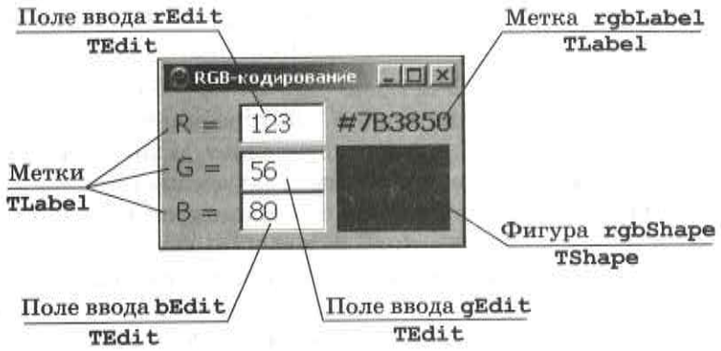


Рис. 7.20

Во время работы программы будут использоваться **поля ввода** rEdit, gEdit и bEdit, метка rgbLabel, с помощью которой будет выводиться результат — код цвета, и фигура rgbShape. В качестве начальных значений полей можно ввести любые целые числа от 0 до 255 (свойство Text).

При изменении содержимого любого из трёх полей ввода нужно обработать введённые данные и вывести результат в заголовок (свойство Caption) метки rgbLabel, а также изменить цвет заливки для фигуры rgbShape. Обработчик события, которое происходит при изменении текста в поле ввода, называется OnChange. Так как при изменении любого из трёх полей нужно выполнить одинаковые действия, для этих компонентов можно установить один и тот же обработчик. Для этого нужно выделить их, удерживая клавишу Shift, и после этого создать новый обработчик двойным щелчком в Инспекторе объектов.

Для того чтобы преобразовать текст из поля ввода в соответствующее целое число, используется стандартная функция StrToInt (для обратного перевода применяется функция IntToStr). Обработчик события OnChange для поля ввода может выглядеть так:

```
procedure TForm1.rEditChange(Sender: TObject);
var r, g, b: integer;
begin
  r:=StrToInt(rEdit.Text);
  g:=StrToInt(gEdit.Text);
  b:=StrToInt(bEdit.Text);
  rgbShape.Brush.Color:=RGBToColor(r,g,b);
```

```
    rgbLabel.Caption:='#'+IntToHex(r,2)+IntToHex(g,2)
                         +IntToHex(b,2)
end;
```

Поясним последние две строки. Фигура класса TShape имеет свойство-объект Brush, которое определяет заливку внутренней области. Свойство Color этого объекта задаёт цвет заливки, который мы строим из составляющих с помощью стандартной функции RGBToColor.

Далее формируется строка, содержащая шестнадцатеричный код цвета. Для перевода значений в шестнадцатеричную систему используется функция IntToHex, второй её параметр 2 указывает на то, что число записывается с двумя знаками.

Вы можете заметить, что при запуске программы код цвета и цвет прямоугольника не изменяются, какие бы значения мы ни установили в полях ввода в Инспекторе объектов. Чтобы исправить ситуацию, нужно вызвать уже готовый обработчик из обработчика OnCreate формы (он вызывается при создании формы):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  rEditChange(rEdit)
end;
```

При вызове в скобках указан объект, который посылает сообщение о событии. Здесь в качестве источника указан компонент rEdit, но в данном случае можно было использовать любой объект, потому что параметр Sender в обработчике OnChange не используется.

Обработка ошибок

Если в предыдущей программе пользователь введёт не числа, а что-то другое (или пустую строку), программа выдаст сообщение о необработанной ошибке на английском языке и предложит завершить работу. Хорошая программа никогда не должна завершаться аварийно, для этого все ошибки, которые можно предусмотреть, надо обрабатывать.

В современных языках программирования есть так называемый **механизм исключений**, который позволяет обрабатывать практически все возможные ошибки. Для этого все «опасные» участки кода (при выполнении которых может возникнуть ошибка) нужно поместить в блок **try — except**:


```
try
  {"опасные" команды}
except
  {обработка ошибки}
end;
```

Слово *try* по-английски означает «попытаться», *except* — «исключение» (исключительная или ошибочная, непредвиденная ситуация). Программа попадает в блок **except — end** только тогда, когда между **try** и **except** произошла ошибка.

В нашей программе «опасные» команды — это операторы преобразования данных из текста в числа (вызовы функции StrToInt). В случае ошибки мы выведем вместо кода цвета знак вопроса. Улучшенный обработчик с защитой от неправильного ввода принимает вид:

```
try
  r:=StrToInt(rEdit.Text);
  g:=StrToInt(gEdit.Text);
  b:=StrToInt(bEdit.Text);
  rgbShape.Brush.Color:=RGBToColor(r,g,b);
  rgbLabel.Caption:='#'+IntToHex(r,2)+IntToHex(g,2)
                      + IntToHex(b,2);
except
  rgbLabel.Caption:='?';
end;
```

Чтобы увидеть результат такой обработки ошибок, нужно запускать программу отдельно, не из среды Lazarus, иначе система перехватывает ошибки в отладочном режиме.

Существует и другой способ защиты — блокировать при вводе символы, которых быть не должно (буквы, скобки и т. п.). В нашей программе для всех полей ввода можно установить такой обработчик события OnKeyPress (в переводе с англ. — при нажатии клавиши[1]):

```
procedure TForm1.rEditKeyPress(Sender: TObject;
                               var Key: char);
begin
  if not (Key in ['0'..'9',#8]) then Key:=#0;
end;
```

[1] Если в программе нужно обрабатывать русские буквы, используется обработчик OnUTF8Key.

Этому обработчику передаётся изменяемый параметр Key — символ, соответствующий нажатой клавише. Если этот символ не входит в допустимый набор (цифры и клавиша BackSpace, имеющая код 8), введённый символ заменяется на символ с кодом 0, который при выводе просто игнорируется.

Вопросы и задания

1. Что такое компоненты? Зачем они нужны?
2. Объясните, как связаны компоненты и идея инкапсуляции.
3. Что такое родительский объект? Что это значит?
4. Объясните роль свойства Align в размещении элементов на форме.
5. Что такое стандартный диалог? Как его использовать?
6. Назовите основное свойство флажка-выключателя. Как его использовать?
7. Расскажите о сложных свойствах на примере свойства Font.
8. Какой шрифт устанавливается для компонента по умолчанию?
9. Что такое метка?
10. Зачем используются функции StrToInt и IntToStr?
11. Как обрабатываются ошибки в современных программах? В чём, на ваш взгляд, преимущества и недостатки такого подхода?
12. Как при вводе можно блокировать некорректные символы?

Подготовьте сообщение

«Использование компонентов в программе на языке C#»

Задачи

1. Добавьте в программу для построения RGB-кода цвета защиту от ввода слишком больших чисел (больших, чем 255).
2. Разработайте программу для перевода морских милей в километры (1 миля = 1852 м).
3. Разработайте программу для решения системы двух линейных уравнений. Обратите внимание на обработку ошибок при вычислениях.
4. Разработайте программу для перевода суммы в рублях в другие валюты.
5. Разработайте программу для перевода чисел из десятичной системы в двоичную, восьмеричную и шестнадцатеричную.
6. Разработайте программу для вычисления информационного объёма рисунка по его размерам и количеству цветов в палитре.
7. Разработайте программу для вычисления информационного объема звукового файла (без учёта служебной информации) при известных длительности звука, частоте дискретизации и глубине кодирования (числу битов на отсчёт).

§ 55
Совершенствование компонентов

Как вы видели в предыдущем параграфе, на практике нередко нужны поля ввода особого типа, с помощью которых можно вводить целые числа. Стандартный компонент TEdit разрешает вводить любые символы и представляет результат ввода как текстовое свойство Text. Поэтому для того, чтобы получить нужное нам поведение (ввод целых чисел), мы:

- добавили обработчик OnKeyPress, заблокировав ошибочные символы;
- для перевода текстовой строки в число каждый раз использовали функцию StrToInt.

Если такие поля ввода нужны часто и в разных программах, можно избавиться от этих рутинных операций. Для этого создаётся новый компонент, который обладает всеми необходимыми свойствами.

Конечно, можно создавать компонент «с нуля», но так почти никто не делает. Обычно задача сводится к тому, чтобы как-то улучшить существующий стандартный компонент, который уже есть в библиотеке Lazarus. Мы будем совершенствовать компонент TEdit (поле ввода), поэтому, согласно принципам ООП, наш компонент (назовём его TIntEdit) будет наследником класса TEdit, а класс TEdit будет соответственно базовым классом для нового класса TIntEdit:

```
type TIntEdit=class(TEdit)
...
end;
```

Изменения стандартного класса TEdit сводятся к двум пунктам:

- все некорректные символы (кроме цифр и кода клавиши BackSpace) должны блокироваться автоматически, без установки дополнительных обработчиков событий;
- компонент должен уметь сообщать числовое значение; для этого мы добавим к нему свойство Value (в переводе с англ. — значение) целого типа.

Таким образом, описание нового класса выглядит так:

```
type
  TIntEdit = class(TEdit)
  private
    function GetValue: integer;
    procedure SetValue(Val: integer);
  protected
    procedure KeyPress(var Key: Char); override;
  public
    property Value: integer read GetValue
                            write SetValue
  end;
```

Из предыдущего материала (см. § 49) вам должно быть понятно, что для чтения введённого числового значения используется закрытый метод GetValue, а для записи — закрытый метод SetValue. Эти методы могут выглядеть так:

```
function TIntEdit.GetValue: integer;
begin
  try    Result:=StrToInt(Text);
  except Result:=0 end
end;
procedure TIntEdit.SetValue(Val: integer);
begin
  Text:=IntToStr(Val)
end;
```

Для преобразования целых чисел из текстового формата в числовой используется функция StrToInt, а для обратного преобразования — функция IntToStr. В метод GetValue введена защита — в случае ошибки функция возвращает 0.

Для обработки вводимых символов будем использовать метод KeyPress. Этот метод не новый, он есть и у базового класса TEdit. Слово **override** говорит о том, что класс-наследник переопределяет этот метод базового класса TEdit. Если посмотреть в исходные тексты библиотеки Lazarus, метод KeyPress состоит из одной строчки — он вызывает обработчик события OnKeyPress, установленный пользователем. Мы переопределим этот метод так:

```
procedure TIntEdit.KeyPress(var Key: Char);
begin
  if not (Key in ['0'..'9', #8]) then Key:=#0;
  inherited
end;
```


В первой строке происходит блокировка неверных символов, а во второй с помощью команды **inherited** вызывается перекрытый метод базового класса. Поэтому если пользователь компонента TIntEdit установит обработчик OnKeyPress, он будет успешно вызван, но уже после того, как введённый символ обработает наша процедура.

Готовый компонент лучше всего поместить в отдельный модуль, назовем его int_edit. В среде Lazarus для создания нового модуля нужно выбрать команду меню **Файл – Создать модуль.** Описание класса размещается в секции **interface**, а сами методы — в секции **implementation**. Кроме того, в список используемых модулей (после слова **uses**) нужно добавить модуль StdCtrls, в котором описан класс TEdit.

Создадим новый проект и сразу добавим в список используемых модулей новый модуль int_edit. Эта программа будет переводить целые числа из десятичной системы в шестнадцатеричную (рис. 7.21).

Рис. 7.21

Поместим на форму метку с именем hexLabel для вывода шестнадцатеричного значения.

Теперь нужно добавить на форму новый компонент класса TIntEdit, но проблема состоит в том, что его нет в палитре компонентов! В этом случае компонент можно добавить при выполнении программы, и все его свойства придётся настраивать вручную, в программе.

Сначала добавим в описание формы переменную типа TIntEdit:

```
TForm1=class(TForm)
  ...
  decEdit: TIntEdit
end;
```