Я думаю, что информатика (computer science), достигшая зрелого возраста, становится кровосмесительной: людей обучают люди, думающие односторонне. В результате так называемые "ортогонально мыслящие" встречаются все реже и реже...

*Кен Томпсон*

# 1 Цели и задачи курса. Притча Дейкстры

Данный курс является введением в информатику. *Информатика* – фундаментальная естественная наука, изучающая структуру и свойства информации, а также методы ее обработки, хранения, поиска, передачи и т.д.

Информатика как наука сформировалась в середине прошлого века. Теперь это большая наука, и мы коснемся в этом курсе лишь некоторой ее части. Суть предмета, который мы будем изучать, поясним на примере известной притчи Дейкстры, но сначала несколько слов об ее авторе.

Эдсгер Дейкстра (Edsger W. Dijkstra, 11.05.1930–06.08.2002) — один из тех людей, с именем которых связано превращение программирования из шаманства в науку. Работы Дейкстры уже сегодня можно назвать классическими. Одной из форм научной деятельности Дейкстры являлись письма, которые он время от времени посылал своим корреспондентам, призывая распространять их дальше. Сборник, содержащий некоторые из этих писем, был опубликован в 1982 г. Когда взгляды Дейкстры стали известны широкому кругу программистов, они вызвали сильную (и далеко не всегда положительную) реакцию. Теперь приведем полностью перевод письма Дейкстры, содержащего притчу.

*Недавно среди старых моих бумаг я нашел следующий рукописный текст. Должно быть я написал его в середине 1973, но не думаю, что по прошествии трех лет его смысл хоть в чем-то изменился. Следовательно, я включаю его в EWD-серию.*

**Притча о первом программисте**

В незапамятные времена была организована железнодорожная компания. Один из ее руководителей, вероятно малый не промах, обнаружил, что начальные инвестиции могут быть значительно снижены, если снабжать туалетом не каждый железнодорожный вагон, а лишь половину из них. Так и порешили.

Однако вскоре после начала пассажирских перевозок посыпались жалобы. Провели расследование и обнаружили, что причина крайне проста: хотя компания была только что создана, неразберихи уже хватало, и о распоряжении дирекции о туалетах ничего не знали на сортировочных станциях, где все вагоны считались одинаковыми, и в результате в некоторых поездах туалетов почти совсем не было.

Чтобы исправить положение, каждый вагон снабдили надписью, говорящей, есть ли в нем туалет, и сцепщикам было велено составлять поезда так, чтобы около половины вагонов имели туалеты. Хотя это и осложнило работу сцепщиков, но проблему решили и вскоре ответственные за сцепку с гордостью сообщили, что тщательно выполняют новую инструкцию.

Хотя новый порядок сцепки выполнялся неукоснительно, тем не менее неприятности с туалетами продолжались. Новое расследование их причин показало, что хотя действительно половина вагонов в поезде снабжена туалетами, иногда выходит так, что все они оказываются в одной половине поезда. Чтобы спасти дело, были выпущены инструкции, предписывающие чередовать вагоны с туалетами и без них. Это добавило работы сцепщикам, однако, поворчав(!), они и с этим справились.

Жалобы, однако, продолжались. Как оказалось, причина в том, что поскольку туалеты располагаются в одном из концов вагона, расстояние между двумя соседними туалетами в поезде могло достигать трех длин вагонов и для пассажиров с детьми — особенно если коридоры были заставлены багажом — это могло привести к неприятностям. Тогда вагоны с туалетами были снабжены стрелкой, и были изданы новые инструкции, предписывающие, чтобы в каждом поезде(!) все стрелки были направлены в одну сторону. Нельзя сказать, чтобы эти инструкции были встречены на сортировочных станциях с энтузиазмом — количество поворотных кругов(!) было недостаточным, и, по правде говоря, мы должны восхищаться тем, что несмотря на существующие стандарты и нехватку поворотных кругов, напрягшись, сцепщики сделали и это.

Теперь, когда все туалеты находились на равных расстояниях, компания была уверена в успехе, однако пассажиры продолжали беспокоиться: хотя до ближайшего туалета было не больше одного вагона, но не было ясно, с какой стороны он находится. Чтобы решить и эту проблему, внутри вагонов были нарисованы стрелки с надписью ТУАЛЕТ, сделавшие необходимым правильно ориентировать и вагоны без туалетов.

На сортировочных станциях новая инструкция вызвала шок: сделать требуемое вовремя было невозможно(!). В этот критический момент кто-то, чье имя сейчас невозможно установить, заметил следующее. Если мы сцепим вагон с туалетом и без оного так, чтобы туалет был посередине, и никогда их не будем расцеплять, то сортировочная станция будет иметь дело не с $N$ ориентированными объектами, а с $N/2$ объектами, которые можно во всех отношениях и со всех точек зрения считать симметричными. Это наблюдение решило проблему ценой двух

уступок. Во-первых, поезда могли теперь состоять лишь из четного числа вагонов — недостающие вагоны могли быть оплачены за счет экономии от сокращения числа туалетов, и, во-вторых, туалеты были расположены на чуть-чуть неравных расстояниях. Но кого беспокоит лишний метр?

Хотя во времена, к которым относится наша история, человечество еще не было осчастливлено ЭВМ, неизвестный, нашедший это решение, заслуживает звания первого компетентного программиста.

*Я рассказывал эту историю в разных аудиториях. Как правило, программисты восхищались ею, а менеджеры неизменно становились тем более раздраженными, чем ближе подходил рассказ к концу; настоящим математикам, однако, не удавалось ухватить соль.*

*EWD, 1976*

Мораль притчи — нужно уметь не только кодировать (верно записывать операторы программы), но и правильно подходить к решению задачи в целом, т.е. адекватно выбирать структуры данных и алгоритмы работы с ними.

### Литература для дополнительного чтения


1. Д.Кнут. *Искусство программирования, том 1. Основные алгоритмы.* 3-е изд. М.: Вильямс, 2000.
2. Н.Вирт. *Алгоритмы + структуры данных = программы.* М.: Мир, 1985.
3. Н.Вирт. *Алгоритмы и структуры данных.* СПб: Невский диалект, 2001.
4. Т.Кормен, Ч.Лейзерсон, Р.Ривест. *Алгоритмы: построение и анализ.* М.: МЦНМО, 2001.


# 2 Информация

## 2.1 Введение. Сообщение и информация

Информатика была определена нами с использованием понятия информация. Термин *информация* (от лат. informatio — разъяснение, изложение) был введен в середине прошлого века Клодом Шенноном применительно к теории связи (передаче кодов) и в настоящее время получил очень широкое распространение.

Строгого определения понятия информация нет, и обычно считается, что информация — это передаваемые сведения, при этом информация (нечто абстрактное) передается от одного объекта другому в виде конкретных сообщений.

Отсюда следует, что:

1. При передаче информации всегда существует объект, передающий ее (например, авторы этого текста) и принимающий (в данном случае читатель).
2. Информация преобразуется передающим объектом в сообщение (в данном случае текст).
3. Сообщение, поступившее принимающему объекту (читателю), преобразуется им обратно в информацию.

Сообщение без информации, т.е. само по себе, не существует. Любое сообщение несет информацию (отрицание чего-то в сообщении — тоже информация).

Информация может преобразовываться передающим объектом в сообщения разными способами. В нашем случае текст мог быть также произнесен и записан на магнитную ленту. Аналогично и принимающий объект может получать сообщения по-разному: прочитать написанный текст, воспринять его на слух и т.д.

Как в роли передающего объекта, так и в роли принимающего может выступать и живое существо, и прибор. Если передающим и принимающим объектами являются приборы, рассмотрение сообщений и информации не представляет сложностей, т.к. все происходящие при этом физические процессы хорошо известны. Сложнее, когда передающим и/или принимающим объектом является орган чувств живого существа. Но и здесь при передаче информации всегда присутствует физический носитель.

### 2.1.1 Связь сообщения и информации, их интерпретация и обработка

Связь между информацией и сообщением не является однозначной. Одна и та же информация может быть передана различными сообщениями (например, на разных языках). Сообщение может содержать ненужную информацию. Одно и тоже сообщение может нести разную информацию. Пример — известная фраза "Над всей Испанией безоблачное небо", переданная в сообщении о погоде, но означавшая начало военных действий.

Следовательно, для однозначного восприятия информации объект, передающий сообщение, и объект, его получающий, должны действовать в соответствии со строго определенными *правилами интерпретации сообщений*, т.е. необходимо определить некоторое взаимно-однозначное отображение $\alpha$ сообщений $N$ в информацию $I$

$$I = \alpha(N)$$

Например, в фильме "Бриллиантовая рука" для сообщения $N$, равного фразе "Черт побери!", было определено правило $\alpha$, согласно которому $\alpha(N)$ означало, что нужно прятать бриллианты в гипс. Непосвященный человек интерпретировал бы эту фразу по-другому.

Обработка сообщений, в случае если один из объектов — человек, происходит в коре головного мозга, и происходящие при этом процессы далеко не все изучены. Поэтому упрощенно будем считать, что при передаче информации происходит изменение некоторой физической величины, называемой *сигналом*. Характеристику сигнала, используемую для представления сообщения, будем называть *параметром сигнала*. Если, например, рассмотреть светофор, то сигнал — световые волны, а параметр сигнала — длина волны (соответствующая цвету — красному, желтому, зеленому).

### 2.1.2 Дискретные сообщения. Знаки. Алфавит

Сигнал называется *дискретным*, если параметр сигнала может принимать лишь конечное число значений. Сообщения, переданные с помощью такого сигнала, называются *дискретными сообщениями*.

При обмене сообщениями обычно используются некоторые соглашения относительно их формы. Будем рассматривать *языковую форму*, то есть случай, когда сообщения составлены на некотором языке.

Языковые сообщения состоят из последовательности знаков. При этом под знаками понимаются не только буквы и цифры.

*Знак* — элемент некоторого конечного множества, которое называется *набором знаков*.

Набор знаков, в котором определен некоторый порядок, называется *алфавитом*.

Приведем несколько примеров наборов знаков:

1. Десятичные цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
2. Шестнадцатиричные цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
3. Знаки мужского и женского начала: ♀, ♂
4. Знаки планет (и Солнца): ☉, ☿, ♀, ♁, ♂, ♃, ♄, ♅, ♆, ♇
5. Знаки зодиака: ♈, ♉, ♊, ♋, ♌, ♍, ♎, ♏, ♐, ♑, ♒, ♓
6. Фазы Луны: ○, ☾, ●, ☽
7. Греческие буквы: $\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, o, \pi, \rho, \sigma, \tau, \upsilon, \phi, \chi, \psi, \omega$.
8. Наконец, наборы двоичных знаков ("0"–"1", "да"–"нет" и т.п.), имеющие в информатике большое значение.

Заметим, что алфавитами не являются наборы 3 и 6 (в них не определен порядок), а также то, что некоторые знаки входят в разные наборы (алфавиты).

Как правило, дискретные сообщения из технических соображений либо из соображений чувственного восприятия разбиваются на конечные последовательности знаков — *слова*. В свою очередь каждое слово можно рассматривать тоже как знак, при этом набор знаков, элементы которого слова, будет шире первоначального набора знаков, из которого составлены слова.

### 2.1.3 Наборы двоичных знаков. Слова. Коды. Символы

Слова над набором двоичных знаков называются *двоичными словами*. Вообще говоря, они не обязаны иметь постоянную длину (азбука Морзе тому пример, хотя и с некоторой оговоркой), но если длина слов постоянна, то говорят о $n$-разрядных двоичных словах.

*Кодом* или *кодировкой* называется правило, описывающее отображение одного набора знаков в другой набор знаков (или слов). Так же называют и множество образов этого отображения. В частности, к таким отображениям относятся *шифры*, и в этом случае образы — это шифровки.

Поясним на примере кодирования десятичных цифр двоичными словами. Здесь мы имеем два набора знаков ($\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ и $\{0, 1\}$) и следующее правило отображения первого набора в слова над вторым набором:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 10 | 11 | 100 | 101 | 110 | 111 | 1000 | 1001 |

Очевидно, при кодировании должны приниматься во внимание свойства кодируемых объектов, при кодировании чисел должны учитываться возможные действия с ними, при кодировании букв — их порядок и т.д. Например, если действие, которые предполагается производить с числами, состоит только в их сложении, римские цифры (с некоторой оговоркой) — самое простое представление. Сложение во многих случаях сводится к механическому приписыванию слагаемых: I + II = III.

Знак вместе с его смыслом называется *символом.* Например, знак * часто используется в литературе как символ звездочка, в математике — как символ умножения, в программировании – как символ разыменования (Си) и т.д.

### 2.1.4 Шенноновские сообщения. Количество информации. Теорема кодирования Шеннона

Как ни странно это покажется на первый взгляд, но количество информации, содержащейся в сообщении, можно измерить.

Пусть сообщения состоят только из знаков какого-то алфавита, и эти знаки появляются в сообщениях с определенной вероятностью. Сообщения, в которых вероятности появления знаков не меняются со временем, называются *шенноновскими сообщениями.* Именно такие сообщения и будем рассматривать в дальнейшем.

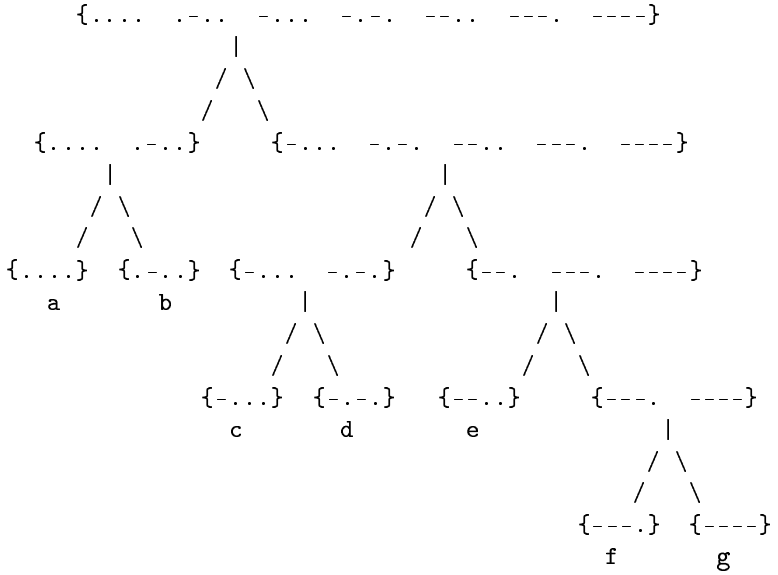
По существу, количество информации — это мера затрат, необходимых для того, чтобы раскодировать все знаки сообщения. Это несколько отличается от интуитивных представлений об информации, но дает удобную для анализа модель.

Наиболее простой алфавит — двоичный, т.е. состоящий из двух знаков (например, “0” и “1” или точка и тире). Количество информации, содержащееся в сообщении, состоящем из одного двоичного знака, примем за единицу и будем называть *битом.* Очевидно, что при раскодировании такого сообщения требуются минимальные затраты — следует сделать только один выбор между двумя альтернативными вариантами.

Рассмотрим, как может производиться анализ более сложных сообщений и как при этом измерить затраты (количество информации). Для простоты будем считать, что информация – это текст, состоящий только из букв {a, b, c, d, e, f, g}, которые кодируется в сообщениях двоичными словами следующим образом:

| a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| .... | .-... | -... | -.-. | --.. | ---. | ---- |

Разделим данное множество кодов на два {.... .-..} и {-... -.-. --.. ---. ----}. Полученные подмножества снова разделим надвое и для подмножеств, содержащих более одного двоичного слова, будем повторять процедуру деления, пока не получим одноэлементные подмножества. Результаты можно представить в виде дерева, состоящего из подмножеств,




Предположим, что поступило сообщение

```
.--- ---- ----
```

и его необходимо раскодировать.

Поскольку первый символ сообщения — точка, то первое двоичное слово попадает в подмножество {.... .-..}, а не альтернативное ему. Таким образом, один шаг по дереву вниз, в направлении отождествления полученного первого слова, сделан.

Рассмотрение следующего знака (тире) позволяет выбрать одноэлементное подмножество {.-..}, соответствующее букве b, что и оканчивает анализ. Дальнейшее исследование знаков сообщения приводит к выявлению букв e и g.

Таким образом, чтобы раскодировать первое слово (.-..), нам потребовалось сделать два шага вниз по дереву, т.е. произвести два альтернативных выбора между его подмножествами, и при раскодировке всего сообщения было сделано 9 альтернативных выборов (2 для отождествления буквы b, 3 — e, и 4 — g),

Из определения единицы информации (бита), видно, что один альтернативный выбор соответствует 1 биту информации. Поэтому можно сказать, что первое полученное двоичное слово дало 2 бита информации, а все сообщение содержало 9 битов информации.

Очевидно, что отождествление — поиск определенного слова в множестве из $n$ слов ($n \geq 2$) всегда можно представить посредством конечного числа следующих друг за другом альтернативных выборов. Если символ встречается часто, то разумно количество выборов, требующихся для его распознавания, сделать по возможности меньшим. Этого можно достичь, разбивая множество знаков на равновероятные подмножества. В нашем примере это имело бы место при следующих вероятностях появления букв в исходном тексте или, что тоже самое, соответствующих двоичных слов в сообщениях:

| a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| 1/4 | 1/4 | 1/8 | 1/8 | 1/8 | 1/16 | 1/16 |

Средняя вероятность обращения при отождествлениях ко всем подмножествам второго горизонтального уровня (т.е. {.... .-..} и {-... -.-. --.. ---. ----}) равна 1/2, третьего — 1/4 и т.д.

Наблюдающиеся в реальных случаях вероятности не всегда позволяют разбивать множества точно на равновероятные подмножества. Тем не менее, рассмотрим более детально множества знаков, для которых это возможно. Если $i$-й знак в них выделяется после $k_i$ альтернативных выборов, то вероятность его появления в (длинных) сообщениях $p_i = 1/2^{k_i}$ (например, для знака g — $i = 7, k_i = 4, p_i = 1/16 = (1/2)^4$). И соответственно наоборот для отождествления знака, вероятность появления которого $p_i$, требуется $k_i = \log_2(1/p_i)$ альтернативных выборов.

По определению количество информации (в битах), даваемой знаком $Z_i$ в сообщении, равно числу выборов $k_i$, т.е.

$$I_i = \log_2\left(\frac{1}{p_i}\right).$$

*Среднее количество информации*, приходящееся на один знак, равно произведению вероятности появления знака $p_i$ на указанное выше количество информации

$$H = \sum_i p_i I_i = -\sum_i p_i \log_2(p_i).$$

Эту величину также называют *энтропией* источника сообщений. Термин энтропия был введен Шенноном по совету фон Неймана, который заметил, что формулы, полученные для этой величины, совпали с соответствующими формулами для энтропии в физике.

Если каждый выбор представить в виде двоичного знака (например, при выборе левого подмножества в дереве брать “0”, а при выборе правого — “1”), то отождествлению каждого знака в сообщении будет соответствовать последовательность двоичных знаков, т.е. двоичное слово, которое можно рассматривать как кодировку знака. В нашем примере это выглядит так

| a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|
| 00 | 01 | 100 | 101 | 110 | 1110 | 1111 |

Как видно, подобные двоичные слова имеют разную длину, и знак, вероятность появления которого $p_i$, кодируется словом длиной $N_i$, совпадающей с количеством сделанных альтернативных выборов, т.е. $N_i = k_i = -\log_2(p_i)$. Поскольку отсюда следует, что

$$H = \sum_i p_i N_i,$$

то становится ясным “физический смысл” $H$ как средней длины слов при двоичном кодировании в рассматриваемом (идеальном) случае.

В нашем примере первая кодировка (4-разрядными двоичными словам над набором знаков — точка и тире) имела среднюю длину слова тождественно равную 4, вторая кодировка (двоичными числами разной длины) — 2.625 ($H$ = 1/4*2 + 1/4*2 + 1/8*3 + 1/8*3 + 1/8*3 + 1/16*4 + 1/16*4), что в 1.5 раза меньше.

Обратимся теперь к более реальной ситуации, когда вероятности появления знаков не позволяют разбить множества на равновероятные подмножества. При кодировании в этом случае код $i$-го знака имеет некоторую длину $\tilde{N}_i$, и средняя длина двоичного слова равна

$$L = \sum_i p_i \tilde{N}_i.$$

При известных вероятностях появления знаков $p_i$ ничто не мешает нам вычислить здесь и энтропию источника $H$ по приведенной ранее формуле. Соотношение между энтропией $H$ и средней длиной слова $L$ для любых наборов знаков определяется теоремой Шеннона.

**Теорема.**

1. Имеет место неравенство $H \leq L$.

2. Для всякого источника сообщений можно найти такую кодировку, что разность $L - H$ будет меньше любого заданного наперед числа.

Разность $L - H$ называется *избыточностью кода*. Если набор знаков можно точно разбить на равновероятные подмножества, то, как мы видели, можно выбрать код с $\tilde{N}_i = N_i$, и следовательно с $L = H$.

Заметим, что на практике отдельные знаки редко встречаются с одинаковой вероятностью, и поэтому кодирование с постоянной длиной кодовых слов, часто применяемое из технических соображений, в большинстве случаев заведомо избыточно.

### 2.1.5 Обработка сообщений

Очевидно, что всякое правило обработки сообщений можно трактовать как отображение $\nu$, которое сообщениям $N$ из некоторого множества сообщений $S$ ставит в соответствие новые сообщения $N'$ из множества $S'$. Сообщения $N$ и $N'$ — это последовательности знаков, которые могут рассматриваться как последовательности букв, слов или предложений. Таким образом, обработку сообщений можно представить как некоторое кодирование.

Чтобы правило служило основой для обработки сообщений, оно должно задавать определенный способ построения сообщения $N' = \nu(N)$. Если множество $S$ велико и задавать отображение $\nu$ посредством перечисления всех соответствий неэффективно, то необходимо задать конечное множество операций (элементарных шагов) так, чтобы каждый переход можно было осуществить с помощью конечного числа таких шагов. Таким элементарным шагом может быть, например, такой:

$$\text{заменить подслово } a_1 a_2 \ldots a_{k-1} a_k a_{k+1} \ldots a_n \text{ на } a_1 a_2 \ldots a_{k-1} b a_{k+1} \ldots a_n$$

Кроме того, нужно задать порядок выполнения элементарных шагов и условия завершения преобразования.

## 2.2 Простейшие данные

Сведения, информацию, передаваемую компьютеру или получаемую от него, будем называть *данными*. Сначала рассмотрим *простейшие типы данных*. Как правило, это группы данных, работа с которыми в той или иной степени поддерживается на уровне инструкций, непосредственно выполняемых процессором. Последнее накладывает определенные ограничения на представление этих данных или, иными словами, на используемые правила интерпретации.

По техническим причинам для представления данных при их хранении и обработке в компьютере используются двоичные знаки “0” и “1”. Напомним, что любую информацию можно закодировать с помощью двоичного алфавита (набора двоичных знаков). При этом, как мы знаем, необходимо установить правила интерпретации. Эти правила будут разные для данных разного типа. Как следствие, одна и та же последовательность двоичных знаков будет интерпретироваться по-разному. Например, слово 0100 0001 может означать и десятичное число 65, и символ латинского алфавита “A”.

### 2.2.1 Биты. Булевы алгебры. Операции

"Да будет слово ваше: да, да; нет, нет; а что сверх этого, то от лукавого"

*Евангелие от Матфея 5,37*

Первое, что мы рассмотрим, — это *биты*. Слово "бит" уже встречалось нам ранее. Оно означало единицу количества информации, содержащейся в сообщении. Теперь же мы сузим это понятие и будем подразумевать под битом объект, который может принимать только два значения "0" или "1". В англоязычной терминологии эти понятия различают — Bit и bit соответственно. Еще раз подчеркнем, что алфавит, используемый для представления информации один и тот же (набор двоичных знаков), но правила интерпретации для данных разного типа различны. Разными будут также и операции, разрешенные для данных того или иного типа. Применительно к битам это логические операции. Свяжем значение "1" с истиной ("TRUE"), а "0" — с ложью ("FALSE"). Для данных этого (логического) типа определены следующие операции:

| | |
|---|---|
| "И" (умножение) | $1\ and\ 1 = 1,\ 1\ and\ 0 = 0,\ 0\ and\ 1 = 0,\ 0\ and\ 0 = 0;$ |
| "ИЛИ" (сложение) | $1\ or\ 1 = 1,\ 1\ or\ 0 = 1,\ 0\ or\ 1 = 1,\ 0\ or\ 0 = 0;$ |
| "НЕ" (дополнение) | $not\ 1 = 0,\ not\ 0 = 1$ |
| "исключающее ИЛИ" (eXcluding OR) | $1\ xor\ 1 = 0,\ 1\ xor\ 0 = 1,\ 0\ xor\ 1 = 1,\ 0\ xor\ 0 = 1.$ |

Множество элементов с определенными таким образом тремя первыми операциями называют *булевой алгеброй* в честь Джорджа Буля, издавшего в 1854 году сочинение "Исследование законов мысли, на которых основаны математические теории логики и теории вероятностей".

*Булевой функцией* является выражение, полученное из ее аргументов (элементов некоторой булевой алгебры) путем сложения, умножения и взятия дополнения. Известно, что для каждой булевой алгебры существует ровно $2^{2^n}$ различных булевых функций $n$ аргументов. В нашем случае возможны 4 булевы функции одного аргумента — две постоянные, равные 0 и 1, тождественная $f(x) = x$ и единственная нетривиальная функция, представляющая собой операцию "НЕ". Это легко видеть из следующей таблицы, показывающей аргумент и значения этих функций:

| $x$ | $f_1(x)$ | $f_2(x)$ | $f_3(x)$ | $f_4(x)$ |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |

Если рассмотреть булевы функции двух аргументов, то мы найдем только 10 нетривиальных функций. Интересно, что все булевы функции одного и двух аргументов можно выразить через одну булеву функцию. Такой функцией может быть и так называемый штрих Шеффера $x_1'x_2 = not(x_1\, and\, x_2)$ (неверно, что $x_1$ и $x_2$), и функция Пирса, равная $(not\, x_1)\, and\, (not\, x_2)$ (ни $x_1$ и ни $x_2$). Выражения для основных функций через штрих Шеффера таковы:

$$\begin{aligned} x_1\ and\ x_2 &= (x_1'x_2)'(x_1'x_2), \\ x_1\ or\ x_2 &= (x_1'x_1)'(x_2'x_2), \\ not\ x_1 &= x_1'x_1. \end{aligned}$$

Это означает, что для представления всех логических операций в компьютере достаточно реализовать только одну функцию (или штрих Шеффера, или функцию Пирса), а использование четырех приведенных выше функций — явная избыточность. Она обусловлена простотой и близостью этих четырех функций человеческому мышлению.

### 2.2.2 Байты. Символы. Кодирование (отображение одного набора знаков в другой)

Любые данные можно представить в виде последовательности битов. Однако на каждом этапе решения реальной задачи мы должны использовать типы данных, наиболее подходящие для этого. Для элементов булевых алгебр (например, логических переменных) это, конечно, биты, но для описания, скажем, какой-нибудь внесолнечной планеты использование только битов сильно затруднило бы дело. Уровень абстрагирования здесь должен быть много выше.

Предположим, мы обрабатываем тексты, написанные на каком-то языке. Если это английский язык, то соответствующий алфавит должен содержать 62 различных кода (26 для латинских строчных букв, 26 для заглавных и 10 арабских цифр), а также коды для знаков препинания и некоторых других символов.

Не все символы равновероятны, а некоторые и вовсе могут отсутствовать в текстах, и чтобы уменьшить избыточность кодирования, было бы правильно кодировать символы, встречающиеся чаще, меньшим количеством битов (см. подробнее п. 2.1.4).

Однако практически все современные компьютеры "работают" только с данными фиксированного размера, и наименьший размер — 8 битов или 1 байт. *Байт* является также наименьшей адресуемой частью памяти компьютера. Если бы адресовался каждый бит, адреса были бы чрезмерно длинными.

Итак, технические требования диктуют нам фиксированный размер кода "символа" (8 битов), и поэтому мы затрачиваем для представления текстовой информации больше битов, чем это необходимо. В этом заключается резерв для "сжатия" данных (точнее файлов), которое мы рассмотрим ниже.

Остается только смириться с навязанной "равновероятностью" всех символов. Для представления латинских символов достаточно 7 битов (так называемый ASCII код), остальные 128 значений (возникающие благодаря использованию 8-го бита) применяются для кодирования дополнительных символов латинского алфавита (умляуты и т.п.), символов псевдографики или символов кириллицы.

Для того, чтобы закодировать стандартные и дополнительные символы латинского алфавита и символы кириллицы одновременно 256 символов (8 битов) недостаточно. А ведь есть еще и большое количество иероглифов, букв древних алфавитов и т.д.

Заметим, что отсутствие единого стандарта кодировки кириллицы приводит к определенным проблемам при работе с русскоязычными текстами. В частности, в системе MS DOS используется кодировка cp866, в MS Windows — cp1251, в Linux — koi8-r, на компьютерах Macintrosh — MAC, в международной организации стандартов (ISO) — ISO-8859-5 и т.д. В результате имеем, например, следующие представления символа кириллицы *а* (двоичные коды записаны в виде шестнадцатиричных чисел):

| | |
|---|---|
| cp866 | A0 |
| cp1251 | E0 |
| MAC | E0 |
| koi8-r | C1 |
| ISO | D0 |
| Unicode | 0430 |

Развитие компьютеров привело к возможности выделять под символы 2 байта (16 битов), что позволяет кодировать как минимум 65536 знаков. Первая кодировка, использующая 2 байта, получила название Unicode. При этом для совместимости со старым программным обеспечением были разработаны различные методы представления номеров символов в Unicode: UTF8, UTF16, UTF16LE, UTF16BE. Сейчас разрабатывается четырехбайтовая кодировка UCS-4 (Universal Character Set).

### 2.2.3 Целые числа. Позиционные системы счисления. Дополнительный и обратный коды. Операции. Переполнение

Вспомним, что такое позиционная система счисления. Обычные (десятичные) числа, как известно, можно записать в виде (для простоты рассмотрим трехзначное число):

$$abc = a \cdot 10^2 + b \cdot 10^1 + c,$$

где $0 \le a, b, c < 10$, а 10 — это основание системы счисления.

Вместо 10 в качестве основания можно взять любое натуральное число (кроме 1), например 8

$$127_8 = 1 \cdot 8^2 + 2 \cdot 8^1 + 7 = 87_{10}$$

или 16

$$57_{16} = 5 \cdot 16^1 + 7 = 87_{10}$$

Разумеется, в позиционной системе с основанием $N$ должно быть $N$ цифр, включая 0. Например, в шестнадцатеричной системе имеем следующие цифры: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$. Пример не позиционной системы — римские цифры.

Для перевода чисел из одной системы счисления в другую существуют простые правила. Проще всего переводить числа в десятичную систему. Действуем прямо по определению (см. предыдущие три формулы), выполняя все действия по правилам десятичной арифметики

$$1357_8 = 1 \cdot 8^3 + 3 \cdot 8^2 + 5 \cdot 8^1 + 7 = 512 + 3 \cdot 64 + 5 \cdot 8 + 7 = 512 + 192 + 40 + 7 = 751$$

Для перевода десятичного числа $M$ в систему счисления с основанием $N$ следует разделить $M$ нацело на $N$, записанное также в десятичной системе, затем полученное частное снова разделить на $N$ и делить

так, пока последнее частное не станет равным нулю. Представлением числа $M$ в системе счисления $N$ будет обратная последовательность остатков деления. Можно было бы поступить точно также как и при переводе в десятичную систему, если бы мы знали таблицу умножения, скажем, восьмеричной системы также хорошо, как десятичной. Но нам легче делить в десятичной системе, чем умножать в восьмеричной.

Компьютеры используют двоичную систему счисления. Для этого есть ряд причин.

- Технические устройства с двумя устойчивыми состояниями наиболее надежны и просты.
- Двоичная арифметика намного проще десятичной в реализации на аппаратном уровне.
- При обработки информации возможно применение аппарата булевых алгебр.

Заметим, что в обычной жизни мы используем десятичную систему счисления, потому что привыкли считать по пальцам на руках. Однако десятичная арифметика используется далеко не всегда. В Китае, например, долгое время пользовались пятеричной системой. Да и сейчас в минуте 60 секунд, а в часе — 60 минут.

Насколько проще было бы нам жить, если бы у нас было восемь пальцев (шестнадцать на руках и ногах). Перевод из двоичной системы, используемой компьютером, в шестнадцатеричную (или восьмеричную) выполняется элементарно (обратный перевод не сложнее). Для этого надо разбить двоичное число на тетрады (или триады) и записать каждую тетраду (триаду), соответствующей шестнадцатеричной (или восьмеричной) цифрой, например

```
0100 1000  0101_2 = 408A_16
010 010 000 101_2 = 40205_8
```

Как правило, архитектура компьютеров поддерживает представление целых чисел в виде двоичных символов только нескольких фиксированных размеров. В частности, короткие целые занимают (для процессоров Intel) 2 байта, а длинные целые — 4 байта. Впрочем, можно ограничиться и одним байтом, что дает возможность представить только 256 различных чисел. В случае же двух байтов можно представить 65536 различных целых чисел, а в случае четырех — 4294967296. Использование слов одной длины для целых чисел приемлемо, поскольку предположение об их равновероятности в среднем оказывается справедливым.

Рассмотрим однобайтовые целые. Они могут принимать значения от $00000000_2$ до $11111111_2$, т.е. от 0 до 255. Если же мы хотим представить и отрицательные числа, то необходимо выделить один бит для информации о знаке числа (отрицательное или неотрицательное). Для этого используется самый левый (старший) бит.

Таким образом, при представлении целого числа со знаком можно работать с числами в диапазоне $-128 \dots 127 (-2^7 \dots 2^7 - 1)$, если используется один байт; $-32768 \dots 32767 \, (-2^{15} \dots 2^{15} - 1)$, если два байта; и $-2147483648 \dots 2147483647 \, (-2^{31} \dots 2^{31} - 1)$, если четыре байта. Положительных чисел на одно меньше, чем отрицательных, поскольку ноль здесь считается положительным числом.

Архитектура большинства компьютеров поддерживает две из трех форм кодирования целых чисел со знаком — так называемый прямой код и обратный или дополнительный коды.

Положительные числа представляются во всех трех кодах одинаково, а отрицательные имеют разное представление.

**Прямой код**. В знаковом разряде 1, а в разряды цифровой части помещен двоичный код абсолютной величины.

**Обратный код**. Двоичный код абсолютной величины числа поразрядно инвертируется (при этом в знаковом разряде получается 1).

**Дополнительный код**. К обратному коду прибавляется единица.

Как правило, отрицательное число автоматически преобразуется в нужный код при вводе в компьютер и далее именно в таком виде хранится и участвует в операциях. При выводе происходит обратное преобразование.

Рассмотрим операцию сложения двух целых чисел, представленных обратными кодами, на простом примере

$$1 + (-3) = 0000\,0001_2 + 1111\,1100_2 = 1111\,1101_2 = -2$$

Вычитание заменяется сложением с заменой знака второго слагаемого на противоположенный.

Поскольку умножение на 2 эквивалентно смещению цифр двоичного числа на одну позицию влево, умножение реализуется как последовательность сложений и сдвигов. При этом используется аппарат булевых алгебр. Деление обычно происходит путем многократного прибавления к делимому кода делителя.

Очевидно, что представление объектов (т.е. правила интерпретации) выбирается так, чтобы облегчить реализацию в компьютере операций, которые предполагается над ними выполнять. Для целых чисел, кроме операций сравнения (равенство/неравенство, больше/меньше), это рассмотренные выше арифметические операции.

Ограниченное число байтов, выделяемых для представления целых чисел, может привести к тому, что при операциях с ними случится *переполнение* – ситуация, когда полученное в результате целое число нельзя представить, поскольку оно выходит за пределы указанных диапазонов. Например, при вычислении факториалов двухзначных чисел ($9! = 362880$ невозможно представить, используя два байта, а $13! = 6227020800$ — четыре байта). В этих случаях для представления целых чисел применяется вещественный тип данных или моделируется целочисленная арифметика с числами произвольной длины, представленными, например, в виде списка из стандартных целых чисел.

### 2.2.4 Вещественные числа. Способы представления, стандарт IEEE 754

Бог создал целые числа; все остальные — дело рук человеческих

*Леопольд Кронекер*

Если множество целых чисел — счетное, и мы можем представить в том или ином виде **все** целые числа из некоторого фиксированного диапазона, то с вещественными числами все значительно сложнее, поскольку множество вещественных чисел имеет мощность континуум, и сколько бы мы не выделяли бит под вещественные числа, нам все равно не удастся представить **все** вещественные числа, даже ограничившись малым диапазоном.

Именно поэтому представление целых чисел в архитектурах компьютеров различных производителей с самого начала было практически стандартным (либо дополнительный, либо обратный код), а представление вещественных чисел, или чисел с плавающей точкой, еще пару десятков лет назад было своим практически у каждого производителя. Различались и количество битов, и точность, и алгоритмы округления, и поведение при переполнении. Можно было насчитать более дюжины различных подходов, что значительно затрудняло переносимость программного обеспечения и сравнение результатов, полученных на компьютерах различных архитектур. Особенно это касалось научных приложений.

Лишь в 1985 году был принят стандарт IEEE 754 (IEEE — Institute of Electrical and Electronics Engineers), который теперь реализуется всеми производителями компьютеров (единственное исключение — компьютеры CRAY).

В стандарте определены три типа форматов вещественных чисел:

- Вещественные числа одинарной точности (4 байта)
- Вещественные числа двойной точности (8 байтов)
- Вещественные числа расширенной точности (10 байтов)

De facto стандартным стал и формат вещественных чисел четверной точности (16 байтов), которые иногда используются в научных расчетах.

Каждый формат включает также представление для некоторых специальных (нечисловых) значений: $\pm\infty$ (бесконечность), NaN (Not a Number, т.е. нечисло). Операции с этими значениями определяются в стандарте следующим образом:

| операция | результат |
|---|---|
| $x/\pm\infty$ | 0 |
| $\pm\infty \cdot \pm\infty$ | $\pm\infty$ |
| $\pm x \ / \ 0 \ (x \neq 0)$ | $\pm\infty$ |
| $\infty + \infty$ | $\infty$ |
| $\pm 0/\pm 0$ | NaN |
| $\infty - \infty$ | NaN |
| $\pm\infty/\pm\infty$ | NaN |
| $\pm\infty \cdot 0$ | NaN |

Кроме этого, любая операция с NaN дает NaN.

Вещественные числа представляются в такой форме

$$2^{k+1-N}n$$

где $k, n$ — целые ($1 - 2^K < k < 2^K$ и $-2^N < n < 2^N$), а $K$ и $N$ — некоторые фиксированные числа. Например, при $K = 7$, $N = 24$ число $2.625 = 2\frac{5}{8} = \frac{21}{8}$, может быть представлено с $k, n = 20{,}21;\ 19{,}42;\ 18{,}84$

и т.д. Из этого примера хорошо видно, насколько мала доля вещественных чисел, которые могут быть точно представлены в компьютере.

Неоднозначность представления разрешается так называемой нормализацией, которая выражает вещественное число в виде

$$\pm 2^{e-b}(1+f)$$

где $e$ — целое число, $b = 2^K - 1$ — так называемое *смещение*, $f$ — неотрицательная дробь ($0 < f < 1$). В нашем примере $b = 127$, $e = 128 = 01000000_2$ и $f = \frac{5}{16} = 0.0101_2$.

Стандарт позволяет работать и с ненормализованными числами

$$\pm 2^{1-b}(0+f),$$

если $k = 2 - 2^K$ и $0 < |n| < 2^{N-1}$, что несколько увеличивает возможности представления чисел, близких к нулю.

Итак, согласно стандарту IEEE 754 вещественные числа кодируются следующим образом:

$$s\ \underbrace{kkkkkkkk}_{\text{e}}\ \underbrace{nnnnnnnnnnnnnnnnnnnnnnn}_{f}$$

Здесь $s$ — знак числа, $kkkkkkkk$ — $K+1$ битов *порядка* $e$, $nnnnnnnnnn$ $nnnnnnnnnnnnn$ — $N-1$ битов *мантиссы* $f$. Для чисел одинарной точности $K+1 = 8, N = 24$, для двойной точности $K+1 = 11, N = 53$ и для четверной точности $K+1 = 15, N = 113$.

В формате чисел одинарной точности приняты следующие правила:

| $e$ | $f$ | значение |
|---|---|---|
| 0 | 0 | $\pm 0$ |
| 0 | $\neq 0$ | $\pm 2^{-126} f$ |
| от 1 до 254 | - | $\pm 2^{e-127}(1+f)$ |
| 255 | 0 | $\pm\infty$ |
| 255 | $\neq 0$ | NaN |

При $e = 0$ $f \neq 0$ мы имеем дело с ненормализованными числами. Их точность (число значащих цифр мантиссы) меньше, чем нормализованных ($e$ =1–254). Заметим также, что $-0$ и $+0$ различаются.

При работе с целыми числами была проблема переполнения. Для вещественных чисел эта проблема сохраняется, но отодвигается существенно дальше благодаря возможности явно указывать порядок числа. Если величина представимых целых чисел не может превышать примерно $2^{31} \approx 2 \cdot 10^9$, то величина представимых вещественных чисел может достигать $2^{128} \approx 3{\cdot}10^{38}$ при одинарной точности и $2^{1024} \approx 2{\cdot}10^{308}$ при двойной.

С другой стороны, ограниченное число знаков мантиссы часто приводит к новой проблеме — так называемой *потере точности* представления вещественных чисел при вычислениях. Например, при вычитании близких чисел (таких, скажем, как $10^6 + \frac{2}{3} = 10000000.6(6)$ и $10^6 + \frac{1}{3} = 1000000.3(3)$) результат будет содержать много "придуманных" компьютером цифр (в указанном примере при одинарной точности результат будет примерно следующим 0.33xxxxxx, т.е. истинный результат, равный $\frac{1}{3}$, будет воспроизведен только с точность двух знаков).

### 2.2.5 Строки

При представлении таких данных как символ или число мы использовали жестко ограниченное сверху количество байтов. Однако для хранения текстовых строк, очевидно, каждый раз требуется разный размер памяти. В таких случаях применяется *строковый тип данных*, интерпретируемых как последовательности символов.

На аппаратном уровне поддерживаются операции копирования и сравнение на равенство данных этого типа. При реализации программным путем других операций (определение длины строки, выделение подстроки, включение и удаление подстрок и т.п.) следует учитывать ряд особенностей кодирования символов. В добавление к проблеме кодировки кириллицы, упомянутой нами ранее, трудности может вызвать также то, что алфавитный порядок букв может не совпадать с порядком их кодов, а также существование особых букв (букв с ударением или с диакритическими знаками), причем в некоторых скандинавских языках "алфавитный" порядок может зависеть от сочетания букв.

Следует отметит, что в большинстве случаев реальная длина строки оказывается меньше размера памяти, который выделен для ее хранения. Поэтому в конце строки записывается специальный символ "конца строки". В шестнадцатеричном виде он записывается как 0D 0A в системе Windows или как 0D в системе Unix/Linux.

### 2.2.6 Указатели

Обмен информацией между памятью и арифметическим устройством компьютера происходит не по 1 биту (это очень медленно), а группами битов — словами (и даже группами слов). Каждое слово имеет свой номер — адрес. Таким образом, с каждой адресуемой областью памяти связано два двоичных числа — ее *адрес* и хранимое в ней *значение*. Многие языки программирования позволяют работать с адресами, как с данными, и данные, значения которых интерпретируются как адреса, называются *указателями*.

Операции с указателями, реализованные аппаратно:
а) проверка равенства и неравенства;
б) разыменование — унарная операция (как отрицание для булевского типа), состоящая в получении значения по адресу;
в) получение адреса — унарная операция, обратная разыменованию;
г) сложение/вычитание.

Поясним суть последних трех операций, используя простой пример. Пусть у нас имеется переменная символьного типа $S$, размещенная по адресу 534. Кроме этого, в памяти машине, начиная с адреса 1056, хранится строка: “Студент имярек любит информатику."Адрес (начала) строки присвоен переменной типа указатель $P$ (т.е. $P = 1056$).

Рассмотрим следующую операцию разыменования (для примера используем синтаксис Си, где символ этой операции “*”):

```
S = *P
```

В результате выполнения этой операции переменная $S$ будет содержать символ “С”.

Если же будет выполнен оператор

```
S = *(P+3)
```

то $S$ станет равной символу “д”, размещенному в данной строке на три позиции правее символа “С”. Отметим, что результат не зависит, используется ли однобайтовая или двухбайтовая (Unicode) кодировка символов. Компьютер сам “выясняет”, в каких единицах (в байтах или двухбайтовых словах) следует производить сложение адресов.

Рассмотрим теперь операцию получения адреса считая, что две предыдущие операции были выполнены (также используем синтаксис Си, где операция получения адреса обозначается символом “&”),

```
P = &S
```

В результате содержимое указателя $P$ изменится с 1056 на 534 (в переменной $P$ будет теперь находиться адрес ячейки $S$).

Указатели играют важную роль при представлении динамических структур данных, т.е. структурированных данных, которые могут изменяться во время работы программы.

## 2.3 Структуры данных

В предыдущем разделе мы рассмотрели представление данных, названных простейшими, то есть данных, операции с которыми поддерживались на аппаратном уровне[1]. Познакомимся теперь с возможной организацией таких данных.

### 2.3.1 Данные и их взаимосвязь

При решении физических, экономических и других задач обрабатываемая информация представляет собой абстракцию интересующей нас части реального мира, и данные задачи — это не просто безликие независимые байты, целые и вещественные числа. Всегда есть определенные *структурные связи* между элементами данных, и, чтобы эффективно решать даже относительно сложные задачи, важно ясно представлять себе как структурные отношения, существующие между данными, так и способы представления этих структур в компьютере, и методы работы с различными структурами.

Простейший пример структурированных данных — комплексные числа. Как известно, они состоят из двух взаимосвязанных частей: вещественной части, представимой естественно вещественным числом, и комплексной части, являющейся также вещественным числом.

Другой пример — характеристики объектов какого-то типа. Ясно, что разные параметры одного объекта образуют единую группу. Например, при решении небесно-механических задач для каждой планеты Солнечной системы потребуются задать:

[1] Следует заметить, что существуют компьютеры, в которых поддерживается работа и с более сложными данными.

1. Название планеты — строка.
2. Масса планеты — вещественное число.
3. Элементы орбиты — группа вещественных чисел.

В большинстве случае структурированные данные состоят из элементарных единиц (атомов), являющихся простейшими данными, и способы, посредством которых эти элементы логически связываются друг с другом, характеризуют различные структуры данных.

### 2.3.2 Простейшие структуры данных

По аналогии с данными, *простейшими структурами* будем называть структуры, операции с которыми в той или иной степени поддерживаются на аппаратном уровне (в данном случае, скорее разрешены в языках программирования). Такие структуры обычно используются не только как самостоятельные объекты, но и служат для представления (моделирования) более сложных структур.

#### Простейшие структуры с элементами разных типов

В терминах Паскаля это *запись*, в Си — *структура* (в узком смысле). Обычно все элементы (или *поля*) такой структуры (записи) связаны с одним объектом решаемой задачи, и каждое поле является какой-то осмысленной единицей информации о нем. Количество полей произвольно. Как правило, элементы такой структуры имеют разный тип: целые или вещественные числа, байты, строки и т.д., включая другие структуры (записи).

В памяти компьютера элементы структуры располагаются последовательно. Важно, что количество и состав полей записи (структуры) не меняются в процессе выполнения программы. Подобная статичность структуры облегчает многие операции, как то: копирование структуры (известен ее размер), доступ к полям структуры (известно положение каждого поля относительно всей структуры) и т.д.

Рассмотрим возможную структуру данных для объектов из каталога скоплений галактик Абеля (всего около 4000 скоплений; электронная база данных, основанная на этом каталоге, была разработана в Астрономическом институте и доступна по адресу — *http://www.astro.spbu.ru/CLUSTERS/*). Очевидно, что записи здесь обязательно должны включать следующие поля:

1. $N$: номер объекта в каталоге (строка: символ A и порядковый номер);
2. $Coords$: координаты скопления (структура):
   - $RA$: прямое восхождение (вещественное число двойной точности);
   - $DE$: склонение (вещественное число двойной точности);
3. $Red$: красное смещение (вещественное число одинарной точности);
4. $Numb$: "богатство" — количество галактик в некотором радиусе (целое число);
5. $S1$: тип галактики по классификации 1 (строка);
6. $S2$: тип галактики по классификации 2 (строка).

Ясно, что подобная структура будет содержать основную информацию об одном объекте — скоплении галактик и состоять из элементов разных типов: строк ($N$, $S1$, $S2$), целого ($Numb$) и вещественного ($Red$) чисел и структуры, состоящей из двух вещественных чисел ($RA$, $DE$). Заметим, что информация о скоплении галактики не ограничивается только приведенными выше характеристиками. С каждой такой записью могут еще быть связаны записи с информацией об объектах, обнаруженных в этом скоплении: галактиках, звездах, неизвестных объектах (может быть брак на фотографии), и для полного описания этих данных потребуется более сложная структура данных.

Как правило, языки программирования позволяют определять такие записи и получать доступ к их отдельным полям. Если, например, $Gal$ — объект описанной выше структуры, то $Gal.Red$ — красное смещение этого объекта, а $Gal.Coords.DE$ — его склонение. Синтаксис, определяющий доступ к отдельным полям, может несколько меняться в зависимости от используемого средства (языка программирования), но это не более, чем детали, суть же одна, мы перечисляем какое-то подмножество характеристик, относящихся к тому или иному объекту.

Для обсуждаемых структур данных с элементами разных типов в целом обычно определена только одна операция — сравнение их на равенство. Операции, которые разрешены для элементов структуры, определяются типом данных, к которому они отнесены, иными словами, если элемент структуры —

строка, то с ним возможны все операции, разрешенные для данных строкового типа. Кроме этого, важной операцией является и выбор элемента структуры по его имени.

**Массивы, многомерные массивы**

С чем-то похожим на *массив* — упорядоченный набор однотипных элементов, число которых фиксировано, — мы уже имели дело при рассмотрении строк (последовательностей символов). Правда, строки имели символ конца, а в массивах этого нет.

Самый простой массив — одномерный (его иногда называют *вектором*). Элементы массива располагаются в памяти компьютера последовательно, и поскольку они одного типа (или являются простейшими структурами одного вида), то адрес $i$-го элемента ($A_i$) может быть получен по формуле

$$A_i = A_{i_1} + sizeof(element) * (i - i_1),$$

где $A_{i_1}$ — адрес первого элемента массива, имеющего в общем случае индекс $i_1$, $sizeof(element)$ — количество байтов, которое необходимо для размещения одного элемента массива, $sizeof(element) * (i - i_1)$ — "смещение" элемента с порядковым номером $i$ относительно первого элемента.

Рассмотрим, как при помощи одномерного массива можно описать каталог (множество) галактик Абеля и получить $i$-й элемент. Элементами такого массива будут структуры данных с элементами разных типов. Воспользуемся структурой для скопления, описанной в предыдущем пункте. Порядковый номер скопления (A+порядковый номер) можно исключить из рассмотрения, т.к. он будет соответствовать номеру элемента в массиве. При описании массива (каталога) необходимо задать имя ($Catalog$) и размерность ($N$) массива. Тогда $Catalog(i)$ — структура, содержащая информацию об $i$-м скоплении, а $Catalog(i).Coords.DE$ даст склонение $i$-го скопления. Заметим, что синтаксис в разных языках может различаться.

Следует обратить внимание на то, что обращение к элементу с номером $i$, большим размерности массива (в нашем случае $N$), приведет к ошибке — выходу за пределы отведенной памяти.

Для массивов в целом так же как и для структур (записей) обычно определена только операция сравнения на равенство. Операции, которые разрешены для элементов массива, определяются типом данных, к которому они относятся (как и для полей записей). Массив — структура по сути статическая, поэтому, как и в предыдущем случае, операция доступа к элементам массива проста и быстра.

Часто встречаются и массивы большей размерности. Простой пример — любая таблица. В математических расчетах многомерные массивы используются для представления тензоров и матриц, особенно часто появляющихся при решении разнообразных уравнений численными методами.

Вообще говоря, многомерные массивы мало чем отличаются от одномерных. Причем многомерный массив можно легко представить в виде эквивалентного одномерного массива.

Положение элемента многомерного массива задается набором индексов (двумя в случае матриц). Адрес элемента, например, с индексами $(i, j)$ можно получить, зная адрес $A_{i_1,j_1}$ первого элемента, имеющего индексы $(i_1, j_1)$,

$$A_{i,j} = A_{i_1,j_1} + (N_1 * sizeof(element)) * (i - i_1) + sizeof(element) * (j - j_1),$$

где $N_1$ — размерность массива по первому индексу.

Как правило, языки программирования предоставляют хорошую поддержку для массивов — сами выделяют требуемую память, обеспечивают доступ к запрашиваемому элементу массива. Необходимо только иметь в виду, что располагаться в памяти многомерные массивы могут по-разному: изменяя сначала первый индекс, а затем следующие (т.е. по столбцам как в Фортране), либо наоборот (т.е. по строкам как в Си и Паскале).

**Хэши и их реализация, хэш-функции, коллизии**

Для рассмотрения следующей структуры немного расширим понятие массива. Допустим, мы имеем дело с каталогом малых планет. У этих планет есть имена, представимые строками (например, планета, названная в честь заведующего кафедрой небесной механики "Холшевников"), и хотелось бы иметь возможность по имени планеты получать всю информацию о ней. Существующие имена планет уникальны, и следовательно их представление в компьютере в виде последовательностей "0" и "1" также уникальны. Трактуя последовательность битов в имени планеты как двоичное целое число, можно рассматривать это число как "индекс" элемента массива и использовать данный элемент для хранения информации о планете. Тогда, например, запись вида $Catalog$("Агекян") позволила бы получить очень

быстрый доступ к информации о малой планете "Агекян" (названной в честь профессора СПбГУ Т.А.Агекяна). Такие индексы называются *ключами*, а подобное обобщение массива — *хэшем*.

Безусловно, использование таких "ассоциативных массивов" (Д.Кнут называет их "памятью с содержательной адресацией") весьма удобно, но как эффективно реализовать работу с такой структурой? Мы можем использовать одномерный массив, в котором, однако, будет заполнена полезной информации только малая (точнее даже бесконечно малая) часть, для которой индексы совпадают с ключами. Однако даже на компьютере с очень большой оперативной памятью мы вряд ли сможем разместить небольшой хэш, если будем использовать для его представления одномерный массив. Например для массива с ключами, являющимися именами малых планет, (не самое длинное!) имя "Холшевников" потребует 11 байтов для кодирования, и следовательно размерность массива должна будет превосходить $(2^8)^{11} \sim 2{\cdot}10^{26}$, а сам занимать более $10^{18}$ Гигабайтов!

Очевидно, что память следует использовать более эффективно. Предположим, что мы нашли функцию $i = f(k)$, отражающую множество всех значений ключей $k$ в отрезок натурального ряда $[1, N]$, такой что $N \ll M$, где $M$ — полное количество ключей. Тогда по произвольному ключу $k$ мы найдем малое целое значение $i$, которое может быть использовано в качестве индекса, и, следовательно, все данные можно таким образом разместить в массиве небольшой размерности $N$ и далее работать с его элементами. Такой метод называется *хэшированием*, а функция $f$ — *хэш-функцией*.

Примером хэш-функции может служить функция взятия остатка от деления $f(k) = k \pmod p$, где $k$ — ключ, трактуемый как целое число, $p$ — простое число. Этот пример, кстати, хорошо поясняет, почему хэширование получило название от английского глагола hash — перемалывать. Хэш-функция должна бы быть взаимооднозначной, но вряд ли можно придумать функцию, тем более простую, которая бы (для любых ключей) обладала таким свойством. На практике будут встречаться случаи, когда разным значениям ключей будут соответствовать одинаковые значения функции. Такая ситуация называется *коллизией*, и существуют стандартные способы ее разрешения (например, элемент хэша может быть сделан головой цепного списка записей, содержащих такие данные).

Есть языки программирования, в которых определены данные типа хэш. Для них обычно разрешена только операция сравнения на равенство и естественно выбор элемента. Операции для элементов хэша определяются типом данных, к которому они относятся.

### 2.3.3 Списки как абстрактные структуры

Подведем некоторые итоги. Мы рассмотрели три простейшие структуры: структуры в узком смысле или записи, массивы и хэши. Очевидно, что данные в задачах могут образовывать и более сложные структуры. При этом существенным может стать не только количество и тип элементов структуры, но и связи между ними, причем эти связи вполне могут возникать и уничтожаться в процессе работы программы. Подобную *структуру данных* можно определить как множество элементов и связей между ними, делая таким образом определенный акцент на связи.

Данное абстрактное определение структуры достаточно общее, т.е. включает широкий класс структур. Это, конечно, плюс, но есть и два минуса. Первый — то, что моделировать эти структуры в компьютере нам придется самим, используя простейшие данные и структуры, рассмотренные выше. При этом представление структур будет определяться теми операциями, которые предполагается над ними производить[2]. Заметим здесь же, что все операции придется реализовывать программно, и это второй минус.

#### Линейные списки; стек, очередь, дек

Что означает список, понятно из опыта обыденного использования этого слова. Можно положить, что *список* — это один элемент и подсписок. Не будем пояснять данное определение, его смысл станет понятен из приводимых далее примеров. Сначала рассмотрим списки, в которых определено отношение упорядоченности. Дадим формальное определение таких линейных структур.

*Линейным списком* будем называть множество, состоящее из $n \geq 0$ элементов или *узлов*, структурные свойства которого ограничиваются лишь линейным относительным положением узлов: существует первый узел $X_1$, и каждому узлу $X_k$ $(1 < k \leq n)$ всегда предшествует узел $X_{k-1}$.

Часто выполняются следующие операции над линейными списками:

- получение доступа к $k$-му узлу (получить или изменить содержимое этого узла);
- включение нового узла перед узлом $k$;

[2] Очевидно, что следует выбирать такое представление, при котором наиболее часто выполняемые операция будет производиться наиболее эффективно.

- исключение $k$-го узла;
- объединение двух линейных списков в один;
- разбиение линейного списка на несколько подсписков;
- копирование линейного списка;
- определение количества узлов в списке;
- сортировка узлов списка по некоторым полям узлов;
- поиск в списке узла с заданным значением.

В редких случаях требуются все перечисленные операции, и в зависимости от требуемых операций мы можем выбрать тот или иной способ представления линейных списков. Собственно, нам уже встречалась удачная реализация линейных списков, в которых определяющей операцией была первая из перечисленных выше, это — одномерный массив.

Предположим, однако, что часто выполняемыми операциями являются операции включения/исключения узла. В этом случае использование одномерного массива едва ли может нас удовлетворить. Допустим для примера, что в какой-то линейный список, представленный массивом $List(N)$ мы должны вставить новый элемент после элемента, имеющего индекс $i$. Для этого нам необходимо изменить (точнее предвинуть вправо, освобождая место) все элементы с индексами больше $i$: $List(i+1) = List(i)$. Конечно, чтобы не потерять информацию, придется начинать этот сдвиг с последнего элемента списка. Если такую операцию нам нужно проделать один раз, то мы можем смириться с тем, что из-за включения в список нового элемента мы должны переместить в среднем $N/2$ элементов. Но если в нашей задаче требуется выполнить такую операцию много раз, а $N$ велико, то потраченное время будет недопустимо большим.

В программирование нередко используются линейные списки, в которых операция включение и исключения производится лишь при граничных значениях индекса $k$. В жизни можно также часто встретиться с такими списками: любая "живая" очередь, стопка больших предметов и т.п. Списки, где в зависимости от того в начале или в конце производятся операции включения/исключения, получили специальные названия: *стек*, если все операции включения/исключения (а часто и операции доступа) делаются на одном конце списка; *очередь*, если все включения делаются в одном конце списка, а все исключения — в другом; *дек*, если включения и исключения делаются на обоих концах списка.

Несмотря на такую исключительность, стек (другие структуры тоже, но в меньшей степени) — очень широко используемая структура. Она даже имеет несколько названий: стек, список push-down, список LIFO (последним пришел, первым ушел) и др.

Стеки явно или неявно используются всегда, когда одна задача сводится к другой, а та — к третьей и т.д. Мы накапливаем в стеке эти задачи и удаляем их по мере решения. Собственно так мы думаем, и также организован процесс входов в подпрограммы и выходов из них в компьютере.

### Представление линейных списков, связные списки

В программировании используется в основном два представления линейных списков: *последовательное* и *связное*.

При последовательном представление для хранения структуры отводится непрерывное адресное пространство, в котором элементы располагаются строго один за другим. Одномерный массив вполне подходит для такого представления линейных списков. Операцию включения в любое место линейного списка в этом случае мы рассмотрели выше. Очевидно, что она требует излишне большого машинного времени.

При связном представлении память для хранения линейной структуры может отводится не единым адресным пространством, и элементы могут располагаться в целом в произвольном порядке. При реализации такого представления можно использовать структуры в узком смысле (записи), где помимо информационной части есть еще одно специальное поле, в котором храниться информация (адрес) о следующем элементе данного линейного списка. Построенный таким образом список называется *связным*.

Для примера добавим в нашу структуру $Catalog$ еще одно поле $Next$, являющееся указателем, и поместим в этот указатель адрес (может быть относительный) следующего элемента типа $Catalog$. Увеличение структуры на одно поле в подавляющем большинстве случаев несущественно, но зато как облегчаются операции включения и исключения элементов! Теперь операция включения будет состоять в присваивании полю $Next$ нового элемента адреса $(i+1)$-го элемента и в присваивании полю $Next$ элемента

$i$ адреса нового элемента. Операция удаления элемента из связного списка еще проще — указатель $(i-1)$-го элемента следует сделать равным адресу $(i+1)$-го элемента.

Остается, правда, неясным вопрос, куда денутся исключенные из списков элементы и откуда берутся вновь включаемые. Большинство языков высокого уровня предоставляют механизмы работы с памятью. Отметим, что просто удалить исключаемый элемент нельзя, поскольку не всегда известно, не ссылается ли на данный узел какой-нибудь элемент. Для удаления элементов существует специальный механизм "сборки мусора", который запускается, когда доступная задаче память исчерпывается.

Можно сформулировать общий критерий использования связного и последовательного представлений. Если при работе с линейным списком связи между элементами не меняются, то использование массива предпочтительнее, поскольку он занимает меньше памяти и с ним удобнее работать. В случаях, когда основными операциями являются включения/исключения элементов, связное представление предпочтительнее.

Заметим, что если это необходимо (например, если используется язык, в котором нет средств работы с указателями) легко смоделировать работу со связными списками при помощи массивов.

### Деревья, бинарные деревья, сбалансированные (AVL) деревья

Наряду с линейными структурами в вычислительных алгоритмах часто встречаются и нелинейные. К важнейшим из таких структур относятся деревья.

Формально можно определить *дерево*, как конечное множество $T$, состоящее из одного или более *узлов* таких, что имеется один узел, который называется *корнем* дерева, а все остальные узлы содержатся в $m \geq 0$ попарно не пересекающихся множествах $T_1, \ldots, T_m$, каждое из которых является деревом. Деревья $T_1, \ldots, T_m$ называются *поддеревьями* корня.

Обратите внимание, что это определение является *рекурсивным*: мы определили дерево в терминах самих же деревьев: деревья с $n$ узлами $(n > 1)$ определяются через деревья с количеством узлов меньшим $n$, и, следовательно, в конце концов мы придем к деревьям, состоящим из одного узла, и на этом рекурсия закончится.

Существуют и не рекурсивные определения деревьев, но они представляются менее подходящими, поскольку рекурсивность является естественным свойством таких структур.

Дадим еще несколько определений. Число поддеревьев данного узла является *степенью* узла. Узел с нулевой степенью называется *концевым узлом* или *листом*. Говорят, что корень имеет *уровень* 1, а остальные узлы имеют уровень на единицу больше уровня содержащего их поддерева. Если имеет значение порядок поддеревьев $T_1, \ldots, T_m$, то такое дерево является *упорядоченным*.

Часто используется и терминология *генеалогического дерева*. Говорят, что корень является *отцом* корней своих поддеревьев. Каждый корень, кроме корня всего дерева, является *ребенком* своего отца. Если не ограничиваться смежными уровнями, то можно употреблять термины *предок* и *потомок*.

Одним из важных типов деревьев является *бинарное дерево* — дерево, в котором каждый узел имеет либо два поддерева, называемых *левым* и *правым*, либо ни одного.

При работе с деревьями часто выполняется операция *обхода* дерева, при которой каждый узел проходиться ровно один раз. Полное прохождение дерева даст нам, кстати, линейную расстановку узлов.

Для прохождения бинарного дерева можно использовать один из трех способов обхода дерева: *прямой*, *обратный* или *концевой*. Определяются эти способы рекурсивно, если бинарное дерево не пусто, то обход состоит в том, чтобы:

| *прямой порядок* | *обратный порядок* | *концевой порядок* |
|---|---|---|
| попасть в корень | пройти левое поддерево | пройти левое поддерево |
| пройти левое поддерево | попасть в корень | пройти правое поддерево |
| пройти правое поддерево | пройти правое поддерево | попасть в корень |

Деревья как структуры широко используются в самых разнообразных задачах. Например, в задачах поиска (см., например, выше раздел о теореме Шеннона). Очевидно, что операция поиска очень эффективна в данных, организованных в бинарные деревья. Если использовать линейные структуры, например массив, то время поиска пропорционально количеству элементов массива $O(N)$. Если же для организации данных использовать бинарное дерево, то, как это подмечено еще в древней Греции (дихотомия), время пропорционально $O(\log_2(N))$. При больших $N$ разница огромна.

Правда, это время зависит не только и не столько от $N$, сколько от глубины данного дерева. В самом плохом случае это время будет пропорционально $N$, в самом хорошем — $\log_2(N)$. Поэтому при поиске следует иметь дело с так называемыми *сбалансированными* деревьями, в которых количество узлов в левом и правом поддеревьях каждого узла различаются не более, чем на единицу. В таких деревьях операции вставки в дерево нового узла или удаления узла из дерева, как правило, приводят к тому,

что дерево перестает быть сбалансированным, и требуются дополнительные усилия, чтобы сохранить свойство сбалансированности. Эта задача немного упрощается, если сбалансированными считать деревья, в которых высота левого и правого поддерева каждого узла, отличаются не более, чем на единицу. Деревья, обладающие этим свойством, называются *AVL-деревьями.* При операции вставки или удаления узла свойство сбалансированности меняться не должно. В этом случае при операции поиска среднее количество действий будет пропорционально $O(\log_2(N))$.

Естественно использовать деревья и в задачах с данными, обладающими иерархической структурой. Рассмотрим, например, задачу эволюции больших звездных систем под действием гравитационных сил — задачу $N$ тел, где $N$ очень велико (например, $\sim 10^7$).

В простейшем случае, когда действуют только гравитационные силы, для каждого тела нужно вычислять силы притяжения Ньютона от всех остальных тел, всего $O(N^2)$ таких вычислений. Если $N \approx 10^7$, то задача уменьшения этих вычислений становится жизненно важной. Был предложен ряд методов (Greengard, Barnes-Hut и др.) для расчета взаимодействий $N$ тел, включающих на каждом шаге интегрирования (временном шаге) два действия:

1. построение дерева с листьями — частицами (корень дерева — вся система);

2. обход этого дерева с тем, чтобы вычислить силы, действующие на частицу, с нужной точностью.

Дерево строится следующим образом. Начнем с такого квадрата (для иллюстрации будем считать пространство двумерным, обобщение на третье измерение элементарно), что все $N$ частиц находятся внутри него. Это — корень дерева. Делим этот квадрат на 4 части (в пространственном случае на 8 частей) и получим таким образом узлы следующего уровня. Если в каком-то из вновь построенных квадратов нет ни одной частицы, то он просто не рассматриваются. Если в квадрате ровно одна частица, то получаем концевой узел, а если в квадрате оказалось более, чем одна частица, продолжаем процесс построения дерева рекурсивно. В результате будет построено дерево Барнса-Хата. Заметим, что в данном случае мы имеем дело не с бинарными деревьями. В пространственном случае каждый узел будет, как правило, иметь восемь поддеревьев, так что можно назвать наши деревья "восьмеричными" (или "октарными").

Выигрыш при вычислении силы, действующий на частицу, получится, если мы действие частиц, некоторого поддерева, находящегося довольно далеко, заменим действием одной частицы с массой, равной сумме масс частиц поддерева, помещенной в центр масс этих частиц (возможно с некоторыми поправками), и честно вычисляя силы для близких частиц. При продуманном выборе параметров, определяющих тот или иной способ расчета взаимодействий частиц, оценка трудоемкости вычислений снизится с $O(N^2)$ до $O(N \ln_2 N)$. Разумеется, плата за такой выигрыш — некоторая потеря точности, но при астрономических исследованиях это обычно допустимо.

Так же, как и для линейных списков, для представления деревьев применяются два способа, один использует массивы, а другой — указатели. Здесь лучше называть их не последовательный и связный как ранее, а статический и динамический соответственно.

Одним из возможных способов представления двоичного дерева при помощи массива является следующий. В первый элемент массива поместим узел дерева, в следующие два элемента — два узла второго уровня, в последующие четыре элемента — четыре узла третьего уровня и т.д.

При динамическом представлении узел должен содержать помимо информационных полей поля, являющиеся указателями на корни соответствующих поддеревьев. Подобное представление деревьев более естественно и просто и поэтому более часто используется.

**Графы и их представление**

Графы, как и деревья, является нелинейными структурами. Они используются для моделирования таких сложных объектов как сети (транспортная сеть, информационная сеть, сеть интернет и т.д.). Сетевой объект состоит из множества элементов одного типа, связанных между собой. Например, в случае (сети) метрополитена элементами могут служить станции метро, а в качестве связей выступать туннели, их соединяющие. Элементы, в данном случае станции, могут характеризоваться: названием, информацией о наличии пересадки на другую линию и т.п., а туннели — длиной и т.д.

Строгие определения таковы. *Граф* — это множество точек, называемых *вершинами*, и линий, соединяющих пары вершин и называемых *ребрами.* Каждая пара вершин может быть соединена не более чем одним ребром. Две вершины, соединенные ребром, называются *смежными.* Если $V_1, V_2, ...V_k$ — подмножество вершин графа таких, что $V_i$ и $V_{i+1}$ — смежные вершины, то последовательность ребер $l_{12}, l_{23}, ...l_{k-1,k}$ называется *путем* длины $k$ от вершины $V_1$ к вершине $V_k$. Путь называется *простым*, если

различны вершины $V_1, V_2, ...V_{k-1}$ и различны вершины $V_2, V_3, ...V_k$. Граф называется *связным*, если любую пару вершин связывает какой-то путь. *Циклом* называется простой путь, имеющий длину больше двух и идущий от вершины к ней самой.

Если для каждого ребра определено направление, то получаем *ориентированный* граф. В отличие от обычного, в ориентированном графе две вершины могут быть соединены множеством дуг (ребер), а также начало дуги может совпадать с концом, т.е. вершина может быть соединена сама с собой.

Наиболее часто используемые операции над графами:

1. Определение принадлежности заданной дуги или ребра графу.

2. Удаление и добавление дуги или ребра.

3. Перебор всех дуг или ребер, выходящих из заданной вершины.

и т.д.

При программировании задач, предполагающих работу с сетевыми объектами, необходимо решить вопрос о представлении графа. Выбор представления определяется методом (алгоритмом) решения задачи, а также соображениями экономии памяти при обработке больших графов.

Существует два стандартных способа представления графа:

1. Используя набор списков смежных вершин.

2. При помощи матрицы смежности.

При представлении графа в виде списка смежных вершин применяется одномерный массив ($A$) размерности $N$, где $N$ — количество вершин. Для $i$-й вершины графа соответствующий элемент массива $A(i)$ содержит указатель на связный список, элементы которого вершины, смежные с $i$-й вершиной.

Второе представление используется, когда количество ребер $M$ сравнимо с $N * N$. Здесь создается матрица (двумерный массив) $B$ размерности $N * N$, элементы которой задаются следующим образом: $B(i, j) = 1$, если $i$-я вершина соединена ребром (дугой для ориентированного графа) с $j$-й вершиной, и $B(i, j) = 0$ в противном случае.

### 2.3.4 Специфические компьютерные списки

#### Файлы, их форматы

Любая информация, любые данные, требующиеся для решения той или иной задачи, или получающиеся в процессе или в результате ее решения, как правило, сохраняются на внешних носителях (винчестерах, CDROM'ах и т.д.) в виде *файлов*.

Файл можно рассматривать как последовательность байтов, вообще говоря, не имеющую каких-либо структурных отношений (т.е. просто последовательность), игнорируя таким образом определенную иерархию, вносимую требованиями к организации информации на физическом носителе (обычно файл — это группа взаимосвязанных *записей*, в свою очередь запись — группа взаимосвязанных *полей*, а поле — группа байтов).

Структурные отношения, присущие данным, т.е. их логическая структура, в файле обычно отсутствует; она определяются и используются лишь в программах, которые работают с данным файлом.

С файлом связан ряд *атрибутов*. В большинстве случаев файл имеет *имя* — строку символов, которая идентифицирует файл. В число атрибутов входит также информация о правах доступа, собственнике файла, времени создания и т.п. Все это записывается в *заголовке* файла.

Операции, производимые над файлами, включают: создание, копирование, удаление и получение информации о файле. Кроме этого, с содержимым файла выполняются следующие действия: открытие, чтение, запись, позиционирование и закрытие файла.

В заключение сделаем два замечания:

1. Оставляя определение структуры данных в файле и их интерпретацию за программами, мы не можем рассчитывать на то, что та или иная программа правильно обработает любой файл. Последовательность байтов файла обрабатывается конкретной программой и поэтому подчиняется правилам, определенным лишь в этой программе. Однако, для часто используемых (графических, мультимедийных, астрономических и т.д.) программ эти правила хорошо известны[3], и файлы, им удовлетворяющие, как говорят, имеют определенный *формат*.

Как правило, указание на формат файла содержится в его расширении, например, `.fits`, `.gif`, `.pdf`, `.ps`, `.txt`, `.rtf` и т.д. Разумеется, соблюдение такого порядка в именовании файлов удобно, но вовсе не обязательно: удовлетворяет ли тем или иным правилам файл или нет, проверяет вызванная программа. Кстати, в `Unix` есть специальная программа `file`, которая определяет формат файла по его заголовку.

[3] И часто представляются как некий стандарт.

2. Мы коснулись только “последовательных” файлов. Программы, работающие, например, со сложными базами данных, обычно обрабатывают большие объемы информации и для повышения эффективности (быстродействия) могут использовать более сложные способы хранения данных на внешних носителях (например, индексные указатели, уточняющие положение нужного фрагмента данных и т.п.).

### Тексты, их разметка, язык SGML

Если строки — это последовательности символов или байтов, то когда говорим о *тексте*, мы подразумеваем нечто большее, чем просто последовательность символов или даже слов. Нам важнее, что текст имеет свою, присущую только этому тексту, логическую структуру. И эта структура, вообще говоря, помогает интерпретировать информацию, содержащуюся в данном тексте.

Логические единицы, присущие тексту, зависят от его вида. В прозе — это главы, абзацы, предложения. Поэтический текст можно разделить на стихотворения, строфы, строки. Драматургический текст содержит действия, сцены, реплики, ремарки. Документация может включать разделы, определения, правила, команды, иллюстрации и т.д. (вплоть до текстов программ). Конечно, между перечисленными типами текстов нет четкой границы, а приведенными выше единицами не исчерпывается все многообразие существующих текстовых структур.

Из примеров видно, что сравнительно большой текст, как правило, имеет иерархическую (нелинейную) структуру. Однако внешние носители заставляют нас хранить тексты по сути в виде линейных списков строк — текстовых файлов. Если переход от иерархической структуры к линейной относительно прост (например, обход дерева, переводящий его в линейный список), то обратный переход невозможен, если мы не дополним текст информацией о его логической структуре. Эта вносимая в текст информация называется его *разметкой*.

Важнейших требования, которым должны удовлетворять любые стандарты разметки, очевидны:

- Использование принципов форматирования (набора синтаксических конструкций для разметки документов), независящих от операционной системы и программных приложений.
- Возможность построения специализированного языка форматирования на базе стандартного набора правил, что необходимо ввиду большого разнообразия (видов) текстов.
- Описание формата (логической структуры) документа в отдельном блоке.

Именно эти свойства характерны для стандартного языка обобщенной разметки SGML (Standard Generalized Markup Language). Собственно, SGML — это метаязык, с помощью которого можно определить нужный язык разметки, а затем использовать этот язык для представления и обработки текстов. Широко известный пример — язык разметки HTML (Hypertext Markup Language), определенный с помощью SGML. По определению и как известно из опыта HTML применим к гипертекстам, поэтому очевидно, что SGML может быть использован для разметки не только текстов, но и структуризации практических любых данных.

В SGML для текстовых структурных единиц используется термин *элемент*. Для элементов SGML определены только отношения к другим элементам, никакими другими свойствами они не обладают. Эти отношения определяются в специальном наборе описательных утверждений с простым синтаксисом — DTD (Document Type Definition, т.е. *определение типа документа*). Подобное определение может быть задано как для конкретного документа, так и для документов какого-то широкого класса.

В виде иллюстрации приведем SGML-разметку (DTD построим позже) текста, включающего два стихотворения, состоящие из одной строфы (четырех строк) каждое, и фотографию одного из авторов

```
<anthology>
<poem><title>***</title>
<author>Саша Белоснежка<img src="abs.jpg"></author>

<stanza>
<line>Кто стучится на Руси</line>
<line>Длинным файлом по PC</line>
<line>Это он, это он,</line>
<line>Электронный почтальон</line>
</stanza>
</poem>

<poem><title>Впечатление</title>
```

```
<author>ВИК</author>

<stanza>
<line>Чуть заметен бледный щит Стожар,</line>
<line>Ярко блещет пояс Ориона.</line>
<line>И морозной искрою зажжгнный</line>
<line>Буйствует сияния пожар!..</line>
</stanza>
</poem>
</anthology>
```

Здесь заложена следующая иерархическая структура: корень — весь текст (anthology); узлы второго уровня — названия стихотворений (poem); узлы третьего — имена авторов (authors); четвертого — строфы (stanza); пятого — строчки (line). При помощи элемента img фотография автора “abs.jpg” включается в документ.

Набор описательных утверждений (DTD) может выглядеть, например, так:

```
<!ELEMENT anthology   - - (poem+)>
<!ELEMENT img         - O  EMPTY>
<!ELEMENT poem        - - (title?, author, stanza+)>
<!ELEMENT title       - O (#PCDATA) >
<!ELEMENT author      - O (img?, #PCDATA) >
<!ELEMENT stanza      - O (line+)   >
<!ELEMENT line        O O (#PCDATA) >
<!ATTLIST img src CDATA>
```

Здесь !ELEMENT — ключевое поле, указывающее на то, что описывается элемент, за ним следует символическое имя элемента (например, anthology), два следующих поля — информация о том, может ли быть опущена начальная и конечная метки в документе, последнее поле — относится к содержимому элемента и указывает, из каких элементов оно состоит.

Далее, #PCDATA означает конечные данные (символы). Знак "+"показывает, что элемент может быть повторен более одного раза (если бы стоял знак "*", то элемента могло не быть или же он мог быть повторен сколько угодно раз, "?-— элемент может встречаться не более одного раза. EMPTY — ключевое слово, означающее, что у элемента нет содержимого.

!ATTLIST — список атрибутов элемента img (фотографии автора), src — имя атрибута, CDATA означает, что значение атрибута может включать любые разрешенные символы. Подчеркнем, что элементы в SGML могут быть и других видов: формулы, таблицы и т.д.

Отметим, что SGML описывает лишь разметку текста, а интерпретация и визуализация (вывод на экран или на принтер) текста в соответствии с заданной разметкой является отдельной задачей. Для ее решения создается DSSSL (Document Style Semantics and Specification Language), в котором задаются правила визуализации.

Например, в языке XML, часто используемым теперь вместо языка HTML, правила визуального оформления отделены от текста путем введения стилевого файла XSL (Extensible Stylesheet Language), являющегося аналогом DSSSL. Таким образом, для того чтобы страничку можно было просматривать не только на большом компьютере, но и на компьютере типа Palm, необходимо произвести соответсвующие изменение лишь в стилевом файле XSL.

На основе языка XML создаются языки разметки (аналогично как основе SGML был создан HTML) для различных Интернет-приложений. В частности, в астрономии на основе XML был разработан формат VOTable (Virtual Observatory Table), предназначенный для хранения и обмена табличными данными через Интернет в рамках проекта виртуальной обсерватории.

Таким образом, размеченные тексты можно назвать *структурированными потоками данных*. Основные операции, которые производятся с такими данными, — обработка (интерпретация) в соответствии с правилами, заданными в DTD (или его аналогах), и визуализация согласно правилам, указанным в DSSSL (или XSL).