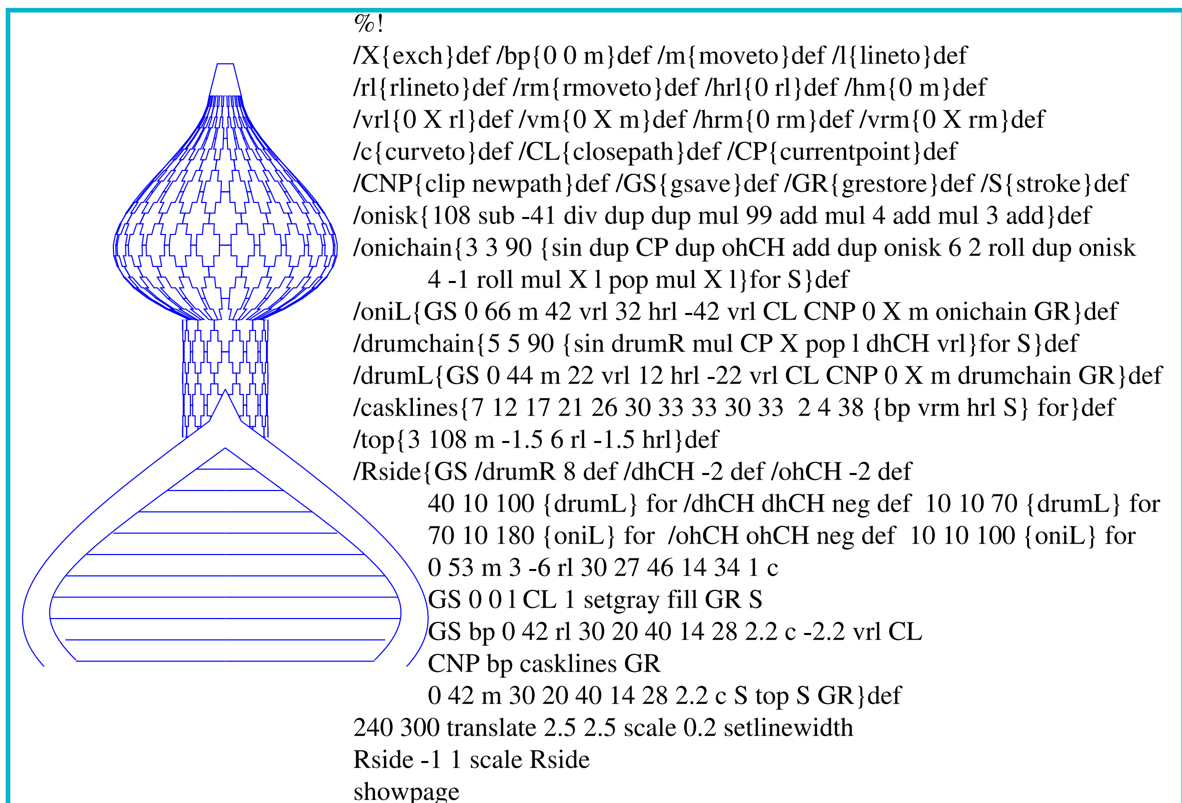
И. В. Романовский

О СТИЛЕ ПРОГРАММИРОВАНИЯ



Санкт-Петербург, 2014

Введение

Знаменитый Талейран, сказал, что **война — слишком серьезное дело, чтобы доверять его генералам**. Этот всем полюбившийся тезис можно с успехом распространять и на другие сферы человеческой деятельности.

В частности, вероятно, придется сказать, что **программирование — слишком серьезное дело, чтобы доверять его преподавателям программирования**. Я убедился в этом недавно, детально знакомясь с программистскими работами студентов разных курсов, от первого до последнего, захотелось высказать свое мнение о программировании, его стиле и технологии, и привести несколько примеров алгоритмов, в основном, по тематике курса дискретного анализа. Я написал статью на эту тему, а потом решил, что следует написать более подробно о том, какие навыки нужны программистам и как этих навыков добиваться. Никто меня не поддержал.

Начнем с того, что обучать программированию можно по-разному. Просто разъяснить, что такое алгоритм и как он записывается на каком-либо языке, — это тоже обучение. Думаю, что в нашем матмеховском преподавании нужно сразу же подчеркивать аспекты, относящиеся к навыкам **профессиональной деятельности**.

Занимаясь программированием в какой-то прикладной области, человек пишет не одну какую-то программу, которую сразу же забывает. Скорее, он занимается этой программой в течение ряда лет, модернизирует ее, приспосабливает к новым возможностям вычислительной среды и увеличивающимся параметрам, пишет аналогичные программы, наконец, передает ее другим разработчикам. Все это требует от них не только написания программ, но и заботы об упрощении их последующего чтения. Посмотрите, например, на мой собственный код на обложке, — я сам сейчас не могу его понять (и даже написал об этом отдельную лекцию [1]).

Мне посоветовали отметить еще, что

> кроме перечисленного, профессиональным программистам почти всегда приходится читать чужой код. С одной стороны, это навык самого программиста, но с другой, этот факт – причина еще строже относиться к своему коду, который будет жить даже когда программист уйдет с проекта и не сможет объяснить коллеге, что же он там понаписал

и посоветовали обратить внимание студентов на аббревиатуру IDE. Она расшифровывается как Integrated Development Environmen — интегрированная система разработки. Многие из вас уже безусловно встречались с такими системами (классика этого жанра — системы Visual Studio и Delphi). В них, конечно, даются свои рекомендации на обсуждаемые здесь темы. Эти рекомендации иногда противоречат нашим, а иногда противоречат друг другу. Поэтому нужно иметь собственное мнение по этому важному вопросу. Очень рекомендую книгу [2].

Еще признаюсь в тяжком грехе: мне стало казаться, что навыки, прививаемые «соревновательным» программированием на различных олимпиадах, начинают приносить вред, соизмеримый с вредом от профессионального спорта (только биостимуляторов нам пока еще недоставало).

Один из наших спортивных программистов сказал, что некоторые из моих рекомендаций неприменимы в их скоростном программировании (а на

полуфинале мирового первенства 2004/2005 года две лучшие команды написали и отладили программы по четырем новым для них задачам меньше чем за час).[1] Я привел ему слова известного летчика-испытателя М. Галлая «Делать быстро – значит делать медленные движения без перерывов между ними». Очень правильные слова, если учесть, что наши скоростники истратили в этом соревновании несколько часов без видимых продвижений. На что же они тратили это время? В значительной степени, на разработку методов, но также и на отладку программ.

Вот я и считаю, что заранее натренированные навыки программистского «чистописания» могли бы им помочь. Могли бы И ИМ помочь.

Но все же главная наша цель — это профессиональная деятельность. Это разработка долгоживущих, модернизируемых программ, приспособленных к работе в огромной среде из сетей, баз данных, готовых компонентов и т. д. Необходимые для такой деятельности привычки нужно воспитывать (и у себя и у других) с самого начала.

Начнем с общих рекомендаций по стилю программирования и со вступительных сведений, и не будем упускать кое-каких интересных алгоритмов.

1. Некоторые подготовительные сведения

При выполнении нужной нам программы, своей или чужой, мы передаем некое задание на ее выполнение операционной системе. Другого у нас сейчас не бывает (во всяком случае, для людей, которые только учатся программировать).

Задание передается в виде одной, причем обычно не очень длинной, *командной строки.* Ее состав и форма существенно зависят от стандартов используемой операционной системы. Обычно в состав строки входит имя (и расположение) выполняемой программы и всевозможные параметры ее выполнения.

1.1. Командная строка и ее состав

1.1.1. Задание параметров командной строки в оболочках

Естественно спросить «А где нам может встретиться эта командная строка, если мы работаем в Windows, к клавиатуре почти не притрагиваемся, обходимся легкими прикосновениями к тактильному экрану?». Рассмотрим несколько случаев, когда система ждет от нас с вами именно командной строки.

а) **Новый ярлык.** Когда вы создаете средствами Windows новый ярлык на рабочем столе, открывается окно диалога, в котором полагается обязательно заполнить поле командной строки. Командной строки требует и поле `Open` в меню `Start>Run`.

б) **Выполнение команд операционной системы.** Некоторые действия операционной системы трудно заменить штатными возможностями Windows (из-за отсутствия нужного средства или от недостаточного опыта пользователя). Например, если нужно составить список файлов какого-либо

[1] Впрочем, на финале первенства 2013/2014 года скорость была значительно ниже.

каталога, удовлетворяющих данному образцу поиска, используют команду dir в формате вроде

```
dir p*.tex > list_tex
```

Здесь p*.tex — это образец поиска, а list_tex — имя файла, куда нужно записать результат. Команда dir выводит результат в «стандартный выходной поток» sysout, по умолчанию это экран дисплея. Знак > «перенаправляет» выходной поток в файл, имя которого указывается за знаком.[2]

Возможность исполнения таких действий появляется при открытии так наз. «окна MS DOS». Окно может быть открыто и непосредственно системой Windows при выборе Start>Programs>MS-DOS Prompt,[3] и многими текстовыми редакторами (например, Notepad++), и практически всеми сервисными программами типа Norton Commander, в которых кроме открытия окна операционной системы предусмотрено и специальное окно для непосредственного набора командной строки. Современный Total Commander, работающий под Windows, имеет даже стековое хранилище использованных командных строк, что очень удобно: можно взять старую строку, исправить ее и исполнить заново.

в) **Вызов программы с большим числом параметров.** Несмотря на распространенность «оболочек», скрывающих от нас работу операционной системы с командной строкой, все еще существуют программы, которые требуют старомодного вызова через командную строку. В качестве примера рассмотрим интерпретатор gawk (или его более современный вариант, например, nawk) интересного языка программирования AWK, встроенного в операционные системы типа UNIX. Когда программа, исполняемая интерпретатором, мала, ее можно вписать прямо в командную строку. Например, для записи в файл c12 конкатенации (сцепки) файлов c1 и c2 нужно написать

```
gawk '{print $0}' c1 c2 > c12
```

Если бы мы хотели попутно перенумеровать строки файла c12, включив номер строки в ее начало, мы написали бы

```
gawk '{print NR ") " $0}' c1 c2 > c12
```

В этих двух командных строках первым стоит имя самого интерпретатора, затем идет в апострофах текст AWK-программы (объяснять который мы сейчас не будем), а затем имена сцепляемых файлов в нужном порядке. Список файлов входного потока прекращается знаком >, который *перенаправляет* выходной поток. Знак >> также перенаправляет входной поток, но не заменяет файл, если он уже существует, а дописывает к нему новые строки.

[2]Когда-то было популярно, а сейчас стало неактуально использование при перенаправлении стандартных имен «файлов», закрепленных за внешними устройствами. Например, prn обозначает системный принтер, а com1 просто порт com1. Таким образом, выполнив строку echo AT DP321-21-21 > com1, мы заставим модем, присоединенный к этому порту, набрать указанный номер. — Боже мой, какая древность!

[3]Аналогичные средства, открывающие похожую версию окна, есть и в совсем современных операционных системах: поищите в Интернете “командная строка Windows 8” и сможете прочесть много интересного.

1.2. Работа с параметрами командной строки внутри программы

Программная обстановка, создаваемая компиляторами у транслируемых программ, обеспечивает им доступ к параметрам командной строки, заданной при вызове этой программы. В Паскале доступ обеспечивается стандартными функциями — параметрами главной процедуры, `ParamCount`, выдающей число параметров в строке, и `ParamStr(i: integer)`, выдающей в форме строки параметр номер `i`. В этот счет не входит нулевой параметр — имя самой вызываемой программы.

Как они работают, легко увидеть по процедуре, печатающей каждый параметр в отдельной строке

```
procedure PrintParamStr;
  var nP,i: integer;
begin
  nP := ParamCount;
  for i := 1 to nP do writeln(ParamStr(i));
end;
```

При вызове этой программы с набором параметров

```
  test '{print $0}' c1 c2 > c12
```

(здесь `test` — имя программы) выводятся строки

```
'{print
$0}'
c1
c2
>
c12
```

Обратите внимание — никакие кавычки, апострофы или скобки не объединят несколько параметров в один.

В языке Си доступ к параметрам командной строки тоже предусмотрен стандартом языка. Именно, главная процедура `main` может иметь два параметра, ее стандартный заголовок выглядит так:

```
void main(int argc,char *argv[])
```

Здесь `argc` — число параметров, которое на единицу больше, чем паскалевское `ParamCount`, а `argv` — массив строк параметров, индексированный от 0 до `argc-1`. В качестве нулевого параметра выступает имя самой программы. Эта конструкция унаследована языками C++ и C#.

Возвратимся к Паскалю и рассмотрим хотя бы самые простые использования параметров командной строки. Важный пример — передача программе параметров счета, среди которых, прежде всего, нужно назвать имена входного и выходного файлов.[4] Вы можете, например, решить, что в вашей программе первым параметром задается имя входного файла, а вторым — имя выходного файла, и вписать в программу хотя бы такие строки:

[4] Недавно обнаружилось, что некоторые наши студенты вплоть до защиты дипломных работ думают, что всегда исходные данные для программы лежат в файле `input.txt`, а результаты выводятся в файл `output.txt`.

```
  if ParamCount < 2 then begin
    writeln('формат вызова: prog <in-filename> <out-filename>');
    exit
  end;
  AssignFile(fInput,ParamStr(1));    Reset(fInput);
  AssignFile(fOoutput,ParamStr(2));  Rewrite(fOutput);
```

При нехватке параметров программа завершится, но сама подскажет пользователю правильный формат вызова. Более «продвинутые» варианты обработки командной строки будут рассмотрены ниже.

1.3. Конвейер

Мало кто (кроме пользователей систем UNIX и LINUX) знает, что команды операционной системы можно сцеплять в командной строке, образуя так называемый *конвейер* (английский термин — pipeline). Если мы записываем несколько команд подряд, разделяя их знаком |, то выходной поток каждой команды становится входным потоком предыдущей.

Но такие штуки можно было делать и в MS DOS. Например, строка

```
  dir *.~* | gawk '{print "del " substr($0,45) }' | command
```

удалит из каталога все файлы, у которых расширение начинается со знака волны ~. А именно:
a) команда `dir` создаст поток строк с информацией о таких файлах,
b) этот поток поступит на вход к программе `gawk`, которая
b1) применит к нему программу, вырезающую из очередной строки имя удаляемого файла,
b2) сформирует команду удаления,
c) команда `command`, вызвав программу `command.com`, выполнит сформированные команды.

1.4. Оформление программы

Зачем беспокоиться об оформлении программ? Почему не писать как придется? Лучше, говорят, экономить место. Я взял одну свою программу и стал ее переписывать в таком экономном стиле, но не выдержал — стало противно. Полюбуйтесь.

```
procedure TForm1.btRunClick(Sender: TObject); var i,j,p,
kStart: longword;  res: string; function IsNewPoint(var
p0: longword): Boolean; var iP: longword; begin  result
:= False; if kTour < nArc then begin iP := 0; repeat if
curD[tour[iP].s] < 2 then begin result := True; p0 := iP;
iP := 0 end else iP := tour[iP].next; until iP = 0; end;
end; procedure FormContour(iP: longword); var i,j: longword;
begin i := iP; kStart := kTour; repeat j := curD[i];
Inc(curD[i]); with tour[kTour] do begin if curD[i] > 2 then
ShowMessageFmt('Impossible value curD[%d]=%d',[i,curD[i]]);
d := j; s := i; i := (i+i+j) mod nVert; next := kTour+1;
end; Inc(kTour); until i = iP; tour[kTour-1].next := kStart;
end; procedure AddContour(p,j: longword); var tmp: TArc;
begin if tour[p].s <> tour[j].s then ....
```

Цель правильного оформления программы — это экономия усилий при чтении программы, которым приходится заниматься для ее модификации, а особенно при поиске ошибок. Забота об оформлении состоит из многих компонентов, не только из расположения текста. Эти компоненты мы сейчас и рассмотрим.

1.4.1. Выбор идентификаторов

Идентификаторы мы выбираем сами — что захотим, то и выберем. Все, что здесь будет предлагаться, только ограничивает нашу свободу.

а) **Не следует писать русские слова латинскими буквами**, а тем более производить латинские формы от русских слов и писать как бы поанглийски русифицированное английское слово. Самые кошмарные примеры последнего рода — это `fail` для обозначения файла и `stek` для стека.

Мне довелось читать один очень большой и очень плохой студенческий текст, который я буду часто цитировать и для удобства обозначу через **F**. Из русских слов в этом тексте на меня самое большое впечатление произвели булева переменная `gotovo`, которую все время хотелось принять за оператор перехода `goto`, массив `vershinas` и серия булевских переменных с названиями типа `daleeq`.

Кроме таких, почти русских, названий, были и помеси, вроде интригующего `ShowOstTree`. Придется пояснить, что в теории экстремальных задач на графах используется термин «остовное дерево», соответствующий английскому «spanning tree». Химера `SdvigChild`, по-видимому, ясна без пояснений.

Про такие сравнительно невинные названия как `obraz`, `hvatit`, `narod`, `krazr` и т. п. я уж и не говорю. Заверю вас только, что сам ничего не придумывал.

б) **Не следует писать одно и то же английское слово в разных вариантах** (конечно же, из всех вариантов нужно стараться выбирать правильный). Когда-то я (грешен) исправлял слово `hesh` на `hash`. Нужно объяснять человеку, что он потом может забыть о том, что он отличается от прочего мира, более грамотного, и, используя введенное где-то далеко слово, может его случайно написать уже правильно. Кстати, это может быть связано и с использованием правильных английских написаний слов вместо «еще более правильных» американских (например, `Colour` вместо `Color`).

в) Нужно выбирать **размер идентификатора** в зависимости от частоты его появления (чем чаще, тем короче, конечно). В иностранных книжках по «хорошему стилю» в программировании я встречал «образцы понятных идентификаторов» примерно такого типа:

```
basicmatrix[rowindex,columnindex] :=
  basicmatrix[rowindex,columnindex]*value[columnindex];
```

Вместе с тем, не хочется вводить, как иногда предлагается, строгих априорных ограничений идентификаторов по длине. Почему? Потому что автор должен думать не о соблюдении внешнего ограничения (идентификаторы `i32`, `i33`, `i97` и т. п. очень лаконичны), а о понятности текста [5].

[5] Профессор В. О. Сафонов рассказал мне о замечательном прецеденте: В одной ор-

г) **Не надо пользоваться идентификаторами для убогих**. Я имею в виду те идентификаторы, которые автоматически создаются при использовании систем визуального программирования (IDE, типа `Visual Studio` или `Delphi`) для компонентов, установленных на форму. Все названия типа `Button1`, `Button2`, `Label18` должны сразу же заменяться на осмысленные. Это же относится и к названиям модулей: `Unit1` и т. д. Ужасные последствия сохранения таких имен ощущаются после того, как человек уже расставил несколько десятков элементов и тратит время на размышления, что дороже: продолжать в том же духе или переименовывать (безусловно, с возможностью новых ошибок).

Переименование не освобождает нас от проблем автоматически. Иногда человек меняет `Button27` на `bt27` и полагает, что все проблемы решены, имея в виду, что к нему с этим делом больше приставать не будут.

д) **Дисциплина использования регистров шрифта**. В Паскале и родственных языках нужно преследовать студента за произвольную смену регистров букв. Очень плохо, если одно и то же слово в соседних строчках написано по-разному: `StrToInt`, `Strtoint`, `strtoint`. Даже такие простые слова как `True` и `False` не должны смешиваться с `true` и `false`. Неприятны провоцируемые использованием `Word` (а, возможно, и написанием, принятым в системе MS Visual Basic: `If nMode = 1 Then ... End If`) написания служебных слов с большой буквы. Почему неприятны? Потому что эти фокусы крадут наше читательское время — мы невольно начинаем размышлять «А зачем он (она) здесь поставил(а) большую букву?».[6]

Нарушение этого естественного правила я встретил даже в книгах опытных педагогов.

Мои привычки здесь таковы: я пишу строчными (маленькими) буквами все служебные слова, кроме `Boolean`, `True` и `False`. Системные процедуры пишутся так, как это рекомендует система: `Inc`, `Dec`, `ShowMessage`, `ShowMessageFmt` и т. п. Даже при полном дефиците времени, нечаянно отклонившись от этого правила, я возвращаюсь и исправляю.

е) Всяческой поддержки заслуживает так называемая **венгерская система обозначений**, в которой идентификатор начинается с написанного строчными буквами префикса, определяющего тип объекта (например, `s` для `string`, `r` для `real`, `f` для `file`, и соответственно для компонентов формы: `bt` для `Button`, `sg` для `StringGrid` и т. п.).

Индивидуальная часть идентификатора обязательно начинается с большой буквы. Отмечу, что, пропагандируя это правило, сам регулярно «ловился» на том, что связывал ввод с идентификатором `sIn`, который компилятор естественно отождествляет со стандартной функцией. Пришлось приучить себя к идентификатору `sInput`.

ж) **Осмысленность идентификаторов**. Оказывается, довольно сложно выбрать систему идентификаторов, в которой они сами подсказывали бы, что они обозначают. Недавно я читал образцовую в этом отношении

ганизации настоятельно рекомендовалось делать глобальные идентификаторы пятисимвольными, а локальные трехсимвольными.

[6] Убежден, что вы сейчас задумались, зачем автор написал эту фразу в гендерно-корректной формк (т. е. не обижающей представителей никакого пола). Это сделано как неожиданный «подарок» читателю: почувствуйте, как такие неожиданности мешают пониманию.

статью, в которой основные массивы назывались `sa` для суффиксного массива (это был главный герой программы) и `bptr`, от Bucket pointer — указатель на корзину (все суффиксы в алгоритме были разложены по корзинам, и данный массив сопоставлял каждому суффиксу корзину, в которой тот лежит).

А потом я встретился с другой программой, в которой булевская переменная называлась `ResultComparison` — результат сравнения. Вот, поди жь, догадайся, какому результату сравнения здесь соответствует `True`!

з) **Обозначения для констант.** Очень четко этот вопрос ставится в [3]: в правильно написанной программе числовые константы в явном виде почти не должны встречаться. Они должны быть заданы посредством именованных констант, которые и должны затем использоваться.

Даже, если вам нужно использовать количество дней в неделе, то нельзя писать просто 7, а нужно сделать описание вроде

```
const nWeekDays = 7;
```

и далее в тексте употреблять этот идентификатор (чтобы отличить от возможных появлений того же числа в другом смысле, например, для выбора номера месяца — июля или размера шрифта).

Совсем уж безобразно выглядит буквальное воспроизведение системных констант, например, цветовых. Плохо, если для установки белого цвета используется запись

```
pen.color:=$00FFFFFF;
```

вместо

```
pen.color := clWhite;
```

В упоминаемом мною тексте **F** автор несколько десятков раз использовал измеренный в пикселях радиус изображаемой на картинке точки. Он выбрал значение 5. Представьте себе, что будет, если он или его научный руководитель решит попробовать другой радиус. На самом деле, вводя процедуру рисования точки, он уменьшил бы число упоминаний радиуса до четырех, но и в этом случае правильно было бы написать

```
const kR = 5;     // радиус рисуемой точки
```

и далее использовать `kR`.

1.4.2. Форматирование текста программы

а) **Пробелы в строке.** Нужно свободнее пользоваться пробелами. Знак присваивания и знаки операций следует, как говорят полиграфисты, «отбивать» от окружающего текста. Есть интересный сайт, где копится статистика программистских предпочтений в форматировании текста программ. Очень вам рекомендую — `http://sideeffect.kr/popularconvention/`. С гордостью отмечаю. что я оказался с большинством.

В качестве примера, несколько забегая вперед, фрагмент студенческой программы

```
Brush.Color:=clWhite;
TextOut(0,15,copy(s,1,i-1));x1:=PenPos.X;
if l=1 then begin  Brush.Color:=$00E6E6E6;
```

```
TextOut(x1,15,' '+s[i]+' ');x1:=PenPos.X; end;
Brush.Color:=$00FEE6C5;
TextOut(x1,15,copy(s,i+1,m-1));
```

Ясно, что и на строки делить нужно не так и отступы нужно делать.

```
Brush.Color := clWhite; TextOut(0,kTop,copy(s,1,i-1));
x1 := PenPos.X;
if l=1 then begin
  Brush.Color := clLightGrey; TextOut(x1,kTop,' '+s[i]+' ');
  x1 := PenPos.X;
end;
Brush.Color := clPinky; TextOut(x1,kTop,copy(s,i+1,m-1));
```

Я позволил себе ввести обозначения для цветов и для числа 15, предполагая, что это вертикальная позиция выводимой строки.

Очень раздражают записи вида

```
if( k=5 )then
```

в которых скобки «приклеиваются» к наружному тексту, а от внутреннего отделяются пробелами. Кто-то из студентов сказал мне, что такая запись оправдывается встроенными редакторами сред программирования, которые учитывают синтаксис и стремятся особо выделить служебные слова языка. Все равно с правильной расстановкой пробелов текст нагляднее.

б) **Отступы.** Я запрещаю себе использовать знак табуляции, так как он в разных средах и на разных машинах интерпретируется по-разному, и это неудобно — внешний вид текста изменяется при переходе на другой компьютер, а в вузе *comp-swapping* почти неизбежен). Для меня почти всегда отступ означает две позиции, как это показано в приведенном выше примере. Причины понятны: одной позиции мало, а больше двух жалко. Почему жалко? Потому что в сложной программе вложенность уровней может быть довольно высокой (больше десятка — это еще не предел).[7]

Потребность в правильном выравнивании у меня так высока, что я не могу читать не выровненную чужую программу — сижу и исправляю. Хотя это полезно, — иногда уже при этом обнаруживаются ошибки. Исправляя текст, нужно добиваться того, чтобы расположение по уровням было надежным и заменяло подсчет уровней вложенности операторных скобок.

в) **Расположение операторных скобок.** Имеются в виду `begin` и `end` в Паскале и фигурные скобки в Си. Обычно рекомендуется каждому программисту выработать свой собственный стиль. Я придерживался этого либерального подхода, но сейчас частично изменил ему и требую, чтобы в программах для меня операторные скобки располагались так, чтобы мне было удобно.

Теперь хотелось бы убедить коллег, что им нужно требовать у студентов того же самого и не устраивать разнобоя.

Керниган и Пайк рекомендуют ставить `begin` в той же строке, что и использующая его конструкция (например, цикл, условие или `with`), а `end` выравнивать по началу этой конструкции (см. выше условный оператор).

[7]Говорят, что большая вложенность свидетельствует о непродуманности алгоритма, и нужно стараться ввести процедуры. Так надо говорить, но это не всегда помогает.

Такое расположение хорошо подчеркивает структуру программы и не провоцирует лишних строчек и лишних сдвигов (как это бывает при расположении `begin` в отдельной строчке). Рассуждения о парности `begin` и `end` не вполне корректны, поскольку `end`, как известно, не всегда стоит в паре с `begin`.

Например,

```
case k of
  0: s := 'Мало данных';
  1: s := 'Неправильный формат';
  2: s := 'Запятая вместо точки';
  3: s := 'Слишком большая задача';
end;
```

г) **Принцип «один оператор в строке».** В использовании этого принципа нужна умеренность. В некоторых случаях, напротив, нужно собирать в одной строчке операторы, составляющие более крупное действие. Например, если мы манипулируем с трехмерными геометрическими объектами и почему-либо храним координаты точек не в единой структуре, а по отдельности, «присваивание точки» состоит из трех покоординатных присваиваний, которые правильнее записывать в одной строке, вроде следующего

```
xCur := x[i]; yCur := y[i]; zCur := z[i];
```

При рисовании отрезков на «холсте» в системе Delphi, идущие парами операторы `MoveTo` и `LineTo` также правильнее располагать в одной строке:

```
MoveTo(x1,y1); LineTo(x2,y2);
```

Существенно улучшается от такой группировки «читабельность» любого текста, который состоит из чередующихся операторов, скажем, `A` и `B`. Имеются в виду тексты вроде:

```
A(P1); B(P1); A(P2); B(P2); A(P3); B(P3); A(P4); B(P4); . . . .
```

Хотя бы несколько этих операторов выпишем в столбик.

```
A(P1);
B(P1);
A(P2);
B(P2);
A(P3);
B(P3);
```

Такая запись не только неэкономна, но и раздражает чередованием операторов. Расположение операторов парами с нарочитым выравниванием упрощает в этом случае восприятие текста очень существенно.

```
A(PFirst); B(PFirst);
A(PSecond); B(PSecond);
A(PThird); B(PThird);
A(PFourth); B(PFourth);
```

Лучше, но некрасиво! Так будет лучше:

```
A(PFirst);  B(PFirst);
A(PSecond); B(PSecond);
A(PThird);  B(PThird);
A(PFourth); B(PFourth);
```

д) **Изменение расположения в ходе работы над программой.** Совершенно не обязательно сохранять одно и то же расположение в течение всей работы над программой. Хорошо проверенные фрагменты можно и нужно писать несколько компактнее (однако, не усердствуя), а фрагмент, в котором разыскивается ошибка, можно временно «распотрошить» на мелкие строчки, снабдив их подробным комментарием. К оным мы сейчас и обратимся.

1.4.3. Комментарии

Заставить писать комментарии очень трудно, я не могу до конца заставить и самого себя. По правде, рекомендуемый и распространенный стиль обширных комментариев с дополнительным выделением «заголовков» из звездочек, знаков равенства, косых палочек и т. п. меня раздражает. Все эти украшения сильно усложняют чтение самой программы. [8]

Но можно выделить несколько типов комментариев, которые особенно важны и на которые следует обратить внимание.

а) **При описаниях типов, констант и переменных.** Здесь достаточно писать всего по несколько слов. Лучше, если комментарий будет помещаться в той же строке, что и описываемый идентификатор, и комментарии будут выровнены. Допустимы отклонения при комментировании буквально «из ряда вон выходящих», т. е. более длинных, чем остальные, идентификаторов, и идентификаторов, по какой-либо причине исключительных. Такое отклонение может быть временным.

б) **При разметке структуры текста.** Это комментарии всего в одну строку. Как заголовки в книгах. Вспомним президента Франции де Голля, в мемуарах которого название каждой главы состояло из одного слова.

К этой же категории я бы отнес комментарии к ветвям условного оператора. Их хорошо размещать (если получается) после открывающей операторной скобки.

в) **После закрывающей операторной скобки.** Достаточно совсем коротких комментариев, помогающих в идентификации многих завершений. Например:

```
      end; {for j}
    end; {while}
  end; {case}
end; {for i}
```

Эти комментарии можно добавлять по мере надобности, в тех местах, которые вызывают трудности.

г) **Перед процедурами (функциями).** Не требуется писать их всегда, и уровень описания может быть различным.

Основная масса. В большинстве случаев можно не писать ничего.

Небольшая неразбериха. При возникновении каких-либо сложностей нужно описывать действие, выполняемое процедурой (смысл параметров, иногда используемый метод).

[8] Как современные рекламы на телевизионном экране мешают смотреть основную передачу.

Действительно сложный случай. При описании очень сложных процедур нужна полная спецификация всех параметров, результата и побочных эффектов.

Процедура, где были серьезные трудности. В особо серьезных случаях требуется подробная детализация используемых данных, возможно, с примером. Если описание получается очень сложным, то полезно разбить его на две части: перед самой процедурой оставить минимальный комментарий, а длинное объяснение перенести в конец файла.

д) **В конце файла.** Здесь можно себе позволить все, что угодно. Целый научный трактат о методе и, если хочется, образец отладочных данных с ожидаемым результатом. Очень полезно в долгоживущих программах иметь в конце историю изменений. Иногда ее пишут в обратном хронологическом порядке, это очень удобно для читателей с ограниченным ресурсом времени и памяти. [9]

е) **В начале файла.** Вот комментарий для нас совсем непривычный и почти обязательный на Западе. В начале каждого файла должна содержаться «учетная информация». Обязательно должно быть написано имя самого файла. Затем имя автора, время изготовления, название проекта и назначение данного файла. Если угодно, запись о копирайте.

Например:[10]

```
//  da_02_11_lexperm.pas
//    Процедура лексикографического перебора перестановок
//    И. В. Романовский, 2004-10-29
//       Пример заголовка файла. Самой такой программы нет.
//       Возможно, что потом будет.
```

ж) **Грамматические ошибки в комментариях.** Я пришел в бешенство, увидев несколько раз в комментарии слово **вершына** (именно так). Особенно из-за понимания того, что обычная общечеловеческая грамотность формально на правильность исполнения программы не влияет. Но это формально. А на самом деле таким комментарием автор демонстрирует свое отношение к работе. Думаю, что здесь следует идти на публичное обсуждение подобных случаев, и с упомянутым комментарием я так и поступил.

1.5. Учет особенностей компьютера и программной системы

Современные языки программирования в значительной мере стандартизовали работу с данными, но все же остаются некоторые различия и в самих языках и в компьютерах. Характерный пример — различие в кодировании

[9] Имеются программные средства, которые поддерживают ведение такой летописи. Боюсь, что для многих предметов, о которых здесь идет речь, уже есть специальные средства. Лично мне не хватает времени и сил на отслеживание этого бесконечного потока, и я концентрирую внимание на том, что можно быстро сделать самостоятельно.

[10] Как времечко-то летит!

команды перехода на следующую строчку в текстовых файлах MS DOS и UNIX.

В файлах MS DOS сохраняется традиция «телетайпной» записи этого перехода в две команды, условно обозначаемые `CR` (carriage return — возврат каретки, код ASCII `0D`) и `LF` (line feed — перевод строки, код `0A`). В файлах UNIX ограничиваются знаком `LF`.

Сейчас имеется достаточно много средств, позволяющих перевести файл из одного представления в другое. Например, в редакторе `Notepad++`, который я теперь использую, можно просто выбрать формат перевода строки в основном меню.

Различий языков, влияющих на запись данных, немного. Для нас существеннее знать о некоторых особенностях и различиях компьютеров.

1.5.1. Структурные данные, доступ к полям

Надеюсь, что все знают, что во всех современных языках программирования можно определить структурный тип данных. Например, сделать комплект из двух целых чисел, одного символа и двух чисел с плавающей точкой

```
type
  aggregate = record
    iA,iB: integer;
    cD:    char;
    rX,rY: real;
  end;
```

Отдельные составляющие этого комплекта называются *полями* структуры.

Структуры активно использует и сам транслятор. Например, когда задаются многочисленные свойства шрифта (размер, насыщенность, начертание, цвет и др.), ему сопоставляется такая структура. Таким образом, свойства компонентов можно воспринимать как поля соответствующей структуры.

Уже давно система машинных команд обеспечивает высокую эффективность доступа к полям структуры. Адрес каждого поля структуры является суммой адреса самой структуры и локального адреса поля внутри структуры — *смещения*. Обычно адрес структуры помещается в машинный регистр, а смещение, которое легко вычисляется уже при компиляции программы, записывается непосредственно в машинной команде. Таким образом, о вычислении адреса поля программисту дополнительно заботиться не нужно.

В связи с этим можно обратить внимание на `with` — замечательную конструкцию языка Паскаль. Если у нас описан, например, массив структур `aggregate` и мы работаем с каким-то конкретным элементом этого массива, мы можем написать так

```
var agA: array[1..40] of aggregate;
. . . . . . . . .
for i := 1 to 40 do with agA[i] do begin
  iA := 0;   iB := i;
  cD := 'n'; rX := 0.0;
end;
```

Использование конструкции `with`, с одной стороны, подсказывает компилятору, что предстоит несколько действий с одним и тем же структурным

набором agA[i], с другой стороны, облегчает запись программисту. Сравните

```
for i := 1 to 40 do begin
  agA[i].iA := 0;    agA[i].iB := i;
  agA[i].cD := 'n';  agA[i].rX := 0.0;
end;
```

И вот ты объясняешь эту студенту, а получаешь от него

```
lbWall2.Canvas.Pen.Width := 3;
lbWall2.Canvas.Pen.Color := clRed;
for i := 1 to 6 do
for j := 1 to 4 do begin
   if (i = 6) and (j = 4) then Break;
   if aMin1[i,j,1] <> 0 then begin
      lbWall2.Canvas.MoveTo((i-1)*50+20,(j-1)*50+20);
      lbWall2.Canvas.LineTo((i-1)*50+20,j*50+20)
   end                  else begin
      lbWall2.Canvas.MoveTo((i-1)*50+20,(j-1)*50+20);
      lbWall2.Canvas.LineTo(i*50+20,(j-1)*50+20)
   end;
end;
```

— это только маленький фрагмент. Всего в одной процедуре 11 повторов lbWall2.Canvas. И таких процедур не одна. Наши замечания по поводу этого фрагмента относятся не только к этим повторам.

1.5.2. Выравнивание данных

При проектировании структур данных и при их использовании нужно иметь в виду, что во многих компьютерах, в частности, в персональных компьютерах, с которыми мы в основном и работаем, имеются технические ограничения на размещение полей: если длина поля кратна 2^k (если это массив, то имеется в виду длина одной ячейки массива), то адрес этого поля также должен быть кратен 2^k. Адрес всей структуры должен быть кратен 2^k с наибольшим k из всех полей структуры.

Для приведенного выше примера это означает, что при 4-битовых целых числах и 8-битовых числах с плавающей точкой адрес каждой структуры должен быть кратен 8, а внутренние адреса полей должны быть такими: 0 для iA, 4 для iB, 8 для cD, 16 для rX, 24 для rY. Адрес следующей структуры должен будет отстоять на 32 байта. Это обстоятельство очень важно при создании массивов из таких структур.

На самом деле, выравниванием можно управлять с помощью *опций транслятора*, {$A1},...,{$A8}. Легко сделать программу, позволяющую поэкспериментировать. Вот результаты экспериментов, собранные в таблицу

Опция	iA	iB	cD	rX	rY	next
A8	0	4	8	16	24	32
A4	0	4	8	12	20	28
A2	0	4	8	10	28	26
A1	0	4	8	9	17	32

1.5.3. Расположение байтов в записи числа

Здесь нам достаточно рассмотреть вопрос о расположении двух байтов, используемых в записи 16-битового целого числа. Возьмем, к примеру, число 789, которое в двоичной системе записывается как `1100010101`, а в 16-ричной как `03 15`. Лишний нуль в последней записи показывает, что используется четыре 16-ричных цифры. Эти четыре цифры записываются в два байта, расположенных в памяти подряд, — все понятно.

Мы обсуждаем, в каком порядке в памяти расположены эти два соседних байта: сначала `03`, а потом `15`, или сначала `15`, а потом `03`.

Настолько привычно начинать со старшего байта, что даже некоторым и непонятно, почему может быть иначе. Между тем, в разных компьютерах приняты разные технические решения. С недавнего времени их принято называть форматами BE и LE:

Формат	Объяснение	Пример	Что сначала
BE	BigEndian	03 15	старший байт
LE	LittleEndian	15 03	младший байт

Сам термин взят из «Путешествий Гулливера»: герои Свифта в Лилипутии делятся на партии, с принципиально разными позициями на то, с какого конца разбивать яйцо. Политически правильно там было разбивать яйцо с острого конца, и сторонник этой позиции назывался Little Endian (а в русском переводе — остроконечник).

Персональные компьютеры распространенного у нас типа тоже остроконечники. Это нужно иметь в виду, когда приходится анализировать состояние памяти.

1.5.4. Типы представления и кодировки строк

Для современного программирования одно из важнейших приложений — это обработка строк. Строка может в первом приближении рассматриваться как массив символов. Обычно языки программирования даже поддерживают характерные для массивов операции над элементами, в данном случае, отдельными символами строки. Однако, строка — это гораздо более продвинутая конструкция, и операций над строками гораздо больше, чем над массивами.

Конструкции, принятые для работы со строками в Паскале и в Си, различаются. В строке Паскаля в «нулевом байте», предшествующем строке, записывается число символов строки. Например, строка `Little Endian` запишется так

```
позиция  0   1   2   3   4   5   6   7   8   9  10  11  12  13
символ       L   i   t   t   l   e       E   n   d   i   a   n
hex         4C  69  74  74  6C  65  20  45  6E  64  69  61  6E
числа   13  76 105 116 116 108 101  32  69 110 100 105  97 110
```

Само побайтовое задание строки приводится в последней строчке.

Строки Си такого счетчика не имеют, в них добавляется стоп-символ в конце — байт, содержащий нулевое значение. Та же строка здесь выглядит так

```
позиция  0   1   2   3   4   5   6   7   8   9  10  11  12  13
символ   L   i   t   t   l   e       E   n   d   i   a   n
числа   76 105 116 116 108 101  32  69 110 100 105  97 110   0
```

В предположении, что для кодировки символов используются коды ASCII (American Standard Code for Information Interchange), это представление строк называется ASCIIZ — Zero-ended ASCII-string.

Сейчас компьютерный мир переходит от байтовой кодировки, ассоциирующейся с ASCII, к двухбайтовой кодировке, представляемой международным стандартом Unicode, и появится новый тип данных `wide string`.

1.5.5. Ввод данных

Детская непосредственность студентов в вопросах ввода данных — постоянная наша проблема. Естественно, что при начальном школьном обучении программированию важно сначала научиться вводить данные хоть как-то, и файл с именем `input.txt` вполне приемлем. Но немыслимо сохранять это простодушие до третьего курса университета!

В какой-то момент студенты должны узнать, что обычно программа рассчитывается на работу с многочисленными, разнообразными файлами, что обычно нужно задать имя файла при запуске программы, что ... много еще чего! Давайте по пунктам.

а) Имя файла для ввода (а тем более путь) не должно жестко задаваться внутри программы. Следует предусмотрен запрос на ввод имени из программы в диалоговом режиме (а тогда лучше всего с выбором одного из существующих файлов), либо, что более просто, предусмотреть задание имени через командную строку. В последнем случае поведение программы может быть более гибким: она может обнаружить, что соответствующий параметр не задан, и, предупредив об этом пользователя, установить имя файла по умолчанию (хотя бы, тот же несчастный `input.txt`).

б) Файл, как правило, должен быть текстовым. Это, конечно, не относится к случаям, когда вы пишете программу, обрабатывающую файлы специального вида, например, если вы изучаете заголовки графических файлов — тут уж, что есть, то есть.

Но в нормальной счетной программе вам не полагается экономить, заменяя текстовый формат чисел потоком чисел в двоичном представлении.

в) Желательно оставлять возможность включения в файл со сложными данными комментариев, которые затем читающая файл программа пропустит (или при желании проверит).

г) Если один и тот же большой набор данных нужно обрабатывать в разных режимах и параметры режимов также нуждаются в обработке, эти параметры лучше собрать в отдельном файле. Иногда может оказаться проще, если этот файл параметров будет содержать имя файла данных.

д) Ввод больших объемов информации в режиме диалога допустим только в специальных программах первичной подготовки данных (набивки). Имеется в виду, что единожды введенные так данные будут потом храниться и многократно использоваться. В студенческих программах диалоговый ввод может использоваться только как способ наказания пользователей (например, самих авторов подобных программ).

1.5.6. Форматный вывод

Про задание имени файла для вывода мы, конечно, говорить не будем, — все, как в случае ввода. Здесь нас больше должно интересовать, что именно выводится.

Когда смотришь американский учебник программирования, то сначала создается впечатление, что главное в программировании — это красивый вывод результата. И это правильное впечатление, так как вывод данных — долгое время был самым главным действием компьютера (пока не появились задачи связи).

Но мы, вроде бы, выше этого.

И вот наши студенты пишут (пример типичный и воспроизводится без изменений, кроме того, что в исходном тексте каждый оператор печати был в одной строке)

```
writeln(f,'Оценка 0 дала лучший результат в '+inttostr(bests[3])
  +' случаях. ('+floattostrF(bests[3]/kmaps*100,ffGeneral,8,5)+'%)');
writeln(f,'Оценка 1 дала лучший результат в '+inttostr(bests[4])
  +' случаях. ('+floattostrF(bests[4]/kmaps*100,ffGeneral,8,5)+'%)');
writeln(f,'Оценка 2 дала лучший результат в '+inttostr(bests[5])
  +' случаях. ('+floattostrF(bests[5]/kmaps*100,ffGeneral,8,5)+'%)');
writeln(f,'Оценка 3 дала лучший результат в '+inttostr(bests[6])
  +' случаях. ('+floattostrF(bests[6]/kmaps*100,ffGeneral,8,5)+'%)');
```

Обратите внимание, это уже далеко продвинувшийся студент, которому велели красиво вывести данные большого эксперимента. Он уже умеет форматировать в Паскале. А если бы он знал функцию `Format`, которая введена в Delphi по примеру Си и предназначена для подстановки в строку числовых и текстовых параметров, он мог бы записать этот фрагмент так

```
writeln(f,Format('Оценка 0 дала лучший результат в %d случаях.(%8.5f%)',
  [bests[3],bests[3]/kmaps*100]));
writeln(f,Format('Оценка 1 дала лучший результат в %d случаях.(%8.5f%)',
  [bests[4],bests[4]/kmaps*100]));
writeln(f,Format('Оценка 2 дала лучший результат в %d случаях.(%8.5f%)',
  [bests[5],bests[5]/kmaps*100]));
writeln(f,Format('Оценка 3 дала лучший результат в %d случаях.(%8.5f%)',
  [bests[6],bests[6]/kmaps*100]));
```

или даже так

```
sFormatStat := 'Оценка %d дала лучший результат в %d случаях.(%8.5f%)';
writeln(f,sFormatStat,[0,bests[3],bests[3]/kmaps*100]));
writeln(f,sFormatStat,[1,bests[4],bests[4]/kmaps*100]));
writeln(f,sFormatStat,[2,bests[5],bests[5]/kmaps*100]));
writeln(f,sFormatStat,[3,bests[6],bests[6]/kmaps*100]));
```

а то и так

```
sFormatStat := 'Оценка %d дала лучший результат в %d случаях.(%8.5f%)';
for kStat := 0 to 3 do
  writeln(f,sFormatStat,[kStat,bests[kStat+3],bests[kStat+3]/kmaps*100]));
```

Нужно, безусловно, учить студентов пользоваться функцией `Format` хотя бы минимально. Отдельное определение форматных строк для вывода следует всячески поддерживать.

Вместе с тем, не хотелось бы, чтобы студенты заучивали ВСЕ возможности форматного вывода и тратили время на упражнения в особо экстравагантных вариантах печати.

1.5.7. Стандартные функции

Нужно уметь пользоваться стандартными функциями. Это умение закладывается еще в школьные годы — тригонометрические вычисления естественно использовать для упражнений в начальном программировании, и школьник сразу получает с десяток стандартных функций, учится распознавать заграничный `tan` и отечественный `tg` и, казалось бы, должен понять из этого опыта, что в программной системе есть богатый выбор стандартных функций и действий.

Это его знание подкрепляется преобразованиями типов, например, такими, как `StrToInt` и `IntToStr`, `Ord` и `Char`, генераторами случайных чисел, стандартными процедурами работы с файлами.

Но набор стандартных функций в современных системам много шире, и, к сожалению, об этих богатствах наши студенты даже не подозревают. Вот некоторые из них, я сам часто глядел в этот список. Очень вам советую завести собственный список "любимых функций": взять этот список за основу, разделить его на две части: свои любимые и остальные, а потом добавлять в каждый список и удалять из него по мере накопления опыта.

Прежде всего, следует назвать процедуры и функции **для работы со строками**, некоторые из них мы перечислим

`Length(s)` Функция, возвращающая длину строки или массива `s` в единицах измерения данного объекта (в байтах для обычной строки, в парах байтов для строки Unicode и т. д.).

`Pos(p,s)` Функция возвращает номер первой позиции, где в строке `s` начинается вхождение образца `p`. Если образец в строке отсутствует, то результат равен 0.

`Copy(s,b,sz)` Функция вырабатывает подстроку строки `s` длины `sz`, копирующую элементы `s`, начиная с позиции `b`.

`Delete(s,b,sz)` Процедура удаляет из `s` подстроку, параметры которой определяются как в предыдущей функции. Очень важны особые случаи. Смотрите их в справочнике.

`Insert(new,s,b)` Процедура вставляет строку `new` в строку `s`, начиная с позиции `b`.

`Trim(s)` Функция вырабатывает подстроку строки `s`, удаляя начальные и конечные пробелы. Можно сказать, левые и правые пробелы, тогда будет понятно назначение функций `TrimLeft` и `TrimRight`.

`CompareStr(s1,s2)` Функция лексикографического сравнения строк по кодам символов. Есть ее аналог `CompareText`, сравнивающий текстовые строки без учета регистра шрифта (т. е. отождествляя большие и малые буквы).

`FillChar(V,sz,Val)` Процедура, заполняющая строку (на самом деле, сплошное место), начинающееся с позиции переменной `V` значениями `Val` на протяжении `sz` байтов.

`WrapText(...)` Очень сложная функция, предназначенная для разбиения длинной строки на кусочки, причем разбиения в удобных местах. Просто знайте, что такая функция существует. Я сам ею ни разу не пользовался.

Вот пример использования некоторых из этих функций

```
// разбор строкового параметра типа ключ=значение
// например, i=input.txt
  k := Pos('=',sPar);
  sKey := Copy(sPar,1 k-1); sVal := Delete(sPar,1,k);
  if sKey = 'i' then sInput := sVal;
```

Математические функции, как уже говорилось, известны лучше, но некоторые стоит добавить. Начнем с преобразований.

`Floor(x)` Функция, возвращающая целочисленное значение $\lceil x \rceil$, — наибольшее целое, не превосходящее x.

`Ceil(x)` Функция, возвращающая целочисленное значение $\lfloor x \rfloor$, — наименьшее целое, не меньшее x.

`Hypot(x,y)` Функция, вырабатывающая значение $\sqrt{x^2+y^2}$ (гипотенузу прямоугольного треугольника с катетами x и y).

`Round(x)` Функция округляет x до ближайшего целого.

`DegToRad(xD)` Функция переводит градусное измерение угла в радианное.

`SinCos(a,vSin,vCos)` Процедура одновременно вычисляет значения $\sin(a)$ и $\cos(a)$ за такое же время, как одно из этих двух значений.

`DivMod(a,b,r,d)` Процедура деления с остатком. В ней при делении a на b получаются одновременно частное r и остаток d.

`InRange(x,lx,r[)` Функция, возвращающая логическое значение «точка x принадлежит диапазону $[lx, rx]$».

Почти математические функции — те, которые работают с временем. Каждый современный компьютер сам считает время. У него есть счетчик, в котором текущее время (с годом, месяцем и т. д.) хранится с точностью до тысячных долей секунды. Измерить временной промежуток между двумя моментами можно, вычитая ранний отсчет из позднего. Состояние счетчика имеет тип -TDateTime-. Для перевода показания счетчика (и разности тоже) в привычный формат есть несколько специальных функций и процедур. Назовем самую простую в употреблении процедуру

`DateTimeToString(sTime,f,timeCounter;` Процедура преобразует значение счетчика $timeCounter$ в строку `sTime` в соответствии с форматом f. Например, `'ss.zzz'` вырабатывает строку из секунд и тысячных долей секунды с точкой в качестве разделителя.

Не имеющая параметров функция `Time` вырабатывает значение «текущее время» в формате `TDateTime`, так что набор операторов

```
  var
t1,t2: TDateTime; sDeltaTime: string;
............
t1 := Time;
............
t2 := Time-t1;
DataTimeToString(sDeltaTime,'ss.zzz',t2);
```

запишет в строку `sDeltaTime` время, прошедшее между двумя вызовами процедуры `Time`.

Полезны, но почему-то мало популярны, процедуры, выделяющие отдельные **компоненты полного имени файла**[11] (опять только некоторые, всего не перечесть). Сюда же включена пара родственных функций

`ChangeFileExt(fn,newEext)` Функция выдает имя файла, полученное заменой расширения в имени файла `fn` на указанное параметром `newext`.

`ExtractFileDir(fn)` Функция выдает полный путь в имени файла `fn`.

`ExtractFileDrive(fn,dir)` Функция выдает имя диска в имени файла `fn`.

`ExtractFileExt(fn,dir)` Функция выдает расширение в имени файла `fn`.

`ExtractFileName(fn,dir)` Функция выдает собственно имя файла в `fn`.

`ExtractFilePath(fn,dir)` Функция выдает путь (без имени диска) в `fn`.

`GetCurrentDir` Функция выдает имя текущего каталога.

`SetCurrentDir(dirname)` Процедура устанавливает текущий каталог. Процедура очень опасна, ею не нужно пользоваться для обеспечения работы с файлом в нужном каталоге. Параметр «текущий каталог» используется во многих местах, и за результатами ее вызова вам следить будет трудно.[12]

Процедур и функций для работы с файлами много. Мы их напоминать не будем, ограничимся только менее популярной разновидностью файла — потоком (`TStream`), каждый поток представляет собой **элемент класса**, все работающие с ним функции и процедуры являются его **свойствами**. Поток нужно сначала создать, при этом определяется его связь с конкретным файлом и способы работы с ним, а затем уже можно работать. Очень удобно создавать поток в памяти, `TMemomryStream`, в который можно просто копировать все содержимое настоящего файла. Потоки очень удобны при сплошном переписывании произвольной информации.[13]

Мы не будем здесь давать формального описания действий с потоками, а ограничимся маленьким, но зато очень свежим, примером использования файлов-потоков при изготовлении многоцветной иконки 32×32 из файла BMP с цветной картинкой такого же размера. Оказывается, чтобы сделать иконку, следует выделить из файла BMP саму картинку и приделать к ней нужное начало и завершение.

```
procedure BMP_2_ICO(sFN_BMP,sFN_Res: string);
  var stBMP: TMemoryStream;
      stRes: TFileStream;
      head: array[0..61] of byte;
      body: array[0..3071] of byte;
      last: array[0..127] of byte;
begin
  stBMP := TMemoryStream.Create;  //  создали поток для исходной картинки
  stBMP.LoadFromFile(sFN_BMP);    //  в него переписали полностью файл
  stBMP.ReadBuffer(head,54);      //  пропустили заголовок BMP
  stBMP.ReadBuffer(body,3072);    //  прочли картинку
  FillChar(head,62,0);            //  начало ICO
  head[ 2] :=  1; head[ 4] :=  1; head[ 6] := 32; head[ 7] := 32;
```

[11] Помня, что такие функции есть, я с трудом находил их. А потом уже стал пользоваться этот текст, как справочником.

[12] Один знакомый пользователь хотел воспользоваться этой функцией для управления выводом. Бедняга не умел првильно задавать полное имя файла, для чего и нужны описанные выше функции.

[13] К сожалению, не знаю аналога этой конструкции в других языках.

```
  head[10] :=  1; head[12] := 24; head[14] :=168; head[15] := 12;
  head[18] := 22; head[22] := 40; head[26] := 32; head[30] := 64;
  head[34] :=  1; head[36] := 24; head[42] :=128; head[43] := 12;
  FillChar(last,128,0);           // конец ICO
 // Запись результата
  stRES:= TFileStream.Create(sFN_Res,fmCreate);  // создали файл-поток
  stRes.Write(headICO,62);       // пишем начало, а потом все остальное
  stRes.Write(body,3072);  stRes.Write(endICO,128);
  stRes.Free;                    // закрыли поток
end;
```

1.5.8. Стандартные компоненты

Сейчас стандартные средства все более укрупняются. Элемент такого укрупнения мы видели только что: файл `stRes` был элементом класса потоков.

Широко вошли в обиход графические классы, описывающие компоненты окон системы Windows с развитой системой свойств и методов.[14]

Теперь, когда вы разрабатываете какую-либо программу, предназначенную для работы в диалоге и полноценно отображающую информацию на экран, вам будет прилично пользоваться стандартными компонентами, для ... всего опять не перечислить, назовем хоть сколько-нибудь.

`TButton` Кнопка. Есть вариант этого компонента, на котором можно кроме надписи разместить картинку.

`TRadioGroup` Группа радиокнопок. Нужно пользоваться этим компонентом, а не аналогичным `TRadioButton`. Радиокнопок обычно несколько, и важна «коллективность» их поведения — включение одной подавляет остальные.

`TMemo` Поле для вывода текстовой информации.

`TEdit` Поле для ввода текстовой информации.

`TOpenDialog` Компонент для выбора имени читаемого файла.

`TSaveDialog` Компонент для выбора файла для записи.

`TTimer` Компонент для создания периодических прерываний (для пауз, анимации и др.).

`TImage` Поле для рисования.

`TDraw` Компонент для построения графиков.

`TProgressBar` Компонент, показывающий состояние процесса вычислений.

`TStringGrid` Компонент для построения таблиц.

Нет! Всего не перечесть. Я даже не упомянул компонентов для работы с базами данных, с Интернетом и многого другого. Все это нужно учить отдельно (часто по мере надобности) и смотреть в специальных книгах.

Ясно только, что без использования таких средств уже серьезно программировать невозможно.

Список литературы

[1] Романовский И. В., Анализ старой программы на Постскрипте. — СПб, 2014, 8 с. См.: сайт кафедры исследования операций СПбГУ

[14] Речь идет о характерных IDE, упоминавшихся во Введении.

[2] Мартин Р., Чистый код: Создание анализ и рефакторинг. — СПб, Питер, 2010, 464 с.

[3] Керниган Б., Пайк Р., Практика программирования. — СПб, Невский диалект, 2001.

[4] Parnas D. L. *On the criteria to be used in docomposing systems into modules.* — Communications of the ACM, 1972, 15:12, 1053-1058.