

Глава 10. Основные приёмы кодинга	165
10.1 Работа с массивами (динамические массивы).....	166
10.2 Многомерные массивы.	170
10.3 Работа с файлами.....	171
10.4 Работа с текстовыми файлами	174
10.5 Приведение типов	178
10.6 Преобразование совместимых типов	181
10.7 Указатели	182
10.8 Структуры, записи	184
10.9 Храним структуры в динамической памяти	186
10.10 Поиск файлов.....	187
10.11 Работа с системным реестром.....	189
10.12 Потоки	194



Глава 10. Основные приёмы кодинга.



Мы уже изучили достаточно теории, и готовы приступить к реальным примерам кодинга. В этой главе мы будем знакомиться с некоторыми приёмами кодинга и писать простые утилиты и программы.

Здесь мы познакомимся с работой с реестром и потоками. Обе эти технологии просто необходимы любому программисту. Мы напишем кучу примеров максимально приближенных к боевым.

Я постараюсь дать побольше примеров различных приёмов кодинга как это делалось в предыдущих главах. Надеюсь, что это тебе в будущем пригодится.

Эта глава будет более практическая и теории здесь будет уже намного меньше. Да и вообще, чем дальше мы движемся, тем меньше теории и больше практики.

Большинство материала будет посвящено сохранению и чтению данных с разных источников информации (например из файлов или реестра). Остальной материал будет описывать то, что может понадобится при работе с источниками инфы.



10.1 Работа с массивами (динамические массивы).

Это одна из необходимых тем в кодировании. Мы будем достаточно часто использовать массивы при программировании, поэтому тебе необходимо прочитать эту часть полностью и понять всё, что я буду говорить.

Я, наверно очень часто говорю, что эта глава очень важна. Это действительно так, ведь я пытаюсь рассказать в своей книге самое необходимое и самое важное. Поэтому каждая её часть является необходимой. Если ты хочешь создавать удобный в понимании код, то придётся изучать различные приёмы и технологии. Я же даю необходимые основы.

Что такое массив? Это просто набор каких-то данных следующих друг за другом. Массив в Delphi обозначается как **array**. Чтобы объявить переменную типа массива нужно описать её в разделе **var** следующим образом:

```
var  
  r: array [длина массива] of тип данных;
```

Как определяется, длина массива? Очень просто, это даже похоже на геометрическое определение. Например, тебе нужен массив из 12 значений. Длина такого массива может быть [0..11] или [1..12]. В квадратных скобках ты должен поставить начальное значение массива и конечное, а между ними две точки.

Тип данных может быть любой из уже пройденных нами. Например, тебе надо объявить массив из 12 строк, это можно сделать следующим образом:

```
var  
  r: array [0..11] of String;
```

В этом примере я объявил переменную r типа массив из 12 строк.

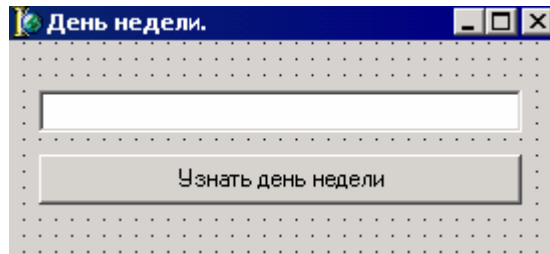
Чтобы получить доступ к какому-то элементу, нужно написать имя переменной массива и после этого, в квадратных скобках написать номер элемента, к которому нужно получить доступ. Например, давай прочитаем 5-й элемент и запишем 7-й элемент нашего массива:

```
var  
  r: array [0..11] of String;  
  Str:String;  
begin  
  Str:=r[5];  
  
  r[7]:='Привет';  
end;
```

В этом примере я в первой строке кода присваиваю переменной *Str* значение пятого элемента массива (*Str:=r[5];*). В следующей строке я седьмому элементу присваиваю строку «Привет» (*r[7]:= 'Привет';*).

Давай напишем какой-нибудь пример для работы с массивами. Допустим, нам надо узнать какой сегодня день недели. Я думаю, это будет полезный примерчик.

Создай новое приложение. Брось на форму один компонент *TEdit* (дадим ему имя *DayOfWeekEdit*) и одну кнопку (дадим ей имя *GetDayButton* и напишем в заголовке «Узнать день недели»). У меня получилась вот такая форма:



По нажатию кнопки мы будем узнавать, какой сегодня день недели и записывать результат в строку *TEdit*.

```
procedure TForm1.GetDayButtonClick(Sender: TObject);
var
  day: Integer;
  week: array[1..7] of string;
begin
  week[1] := 'Воскресенье';
  week[2] := 'Понедельник';
  week[3] := 'Вторник';
  week[4] := 'Среда';
  week[5] := 'Четверг';
  week[6] := 'Пятница';
  week[7] := 'Суббота';

  day:=DayOfWeek(Date);
  DayOfWeekEdit.Text:=week[day];
end;
```


Здесь я объявил массив *week* из семи элементов. После этого, я последовательно всем элементам массива присваиваю названия дней недель.

После этого, я узнаю, какой сегодня день недели. Для этого существует функция *DayOfWeek*. Ей нужно передать только один параметр – дату день недели которой нужно узнать. Я передаю результат работы функции *Date*, которая возвращает текущую дату. Получается, что *DayOfWeek* вернёт мне день недели текущей даты.

Вроде всё нормально, *DayOfWeek* возвращает не строку, в которой написано словами какой сегодня день, а число. Если функция возвращает 0, то это воскресенье, если 1 – понедельник, 2- вторник, 3 – среда и так далее. Как видишь, отсчёт идёт с воскресенья (по европейски). Точно так же я заполнял и массив: 1 – это воскресенье, 2 – понедельник и так далее.

После этого, нам надо превратить число в строку. Это делается очень просто. Нам надо получить только соответствующий элемент массива и всё. Если функция вернула нам 2, то это должен быть понедельник. В массиве под вторым номером тоже идёт

«Вторник», поэтому нам просто нужно получить строку находящуюся под вторым номером в массиве. Вот именно это и происходит в последней строке *week[day]*.

 На компакт диске, в директории \Примеры\Глава 10\Arrays ты можешь увидеть пример этой программы.

Но это мы только закрепили на практике уже пройденный материал. Давай пойдём дальше и познакомимся с динамическими массивами.

Когда ты хочешь создать динамический массив, то не надо указывать его длину. Ты просто указываешь переменную, и её тип:

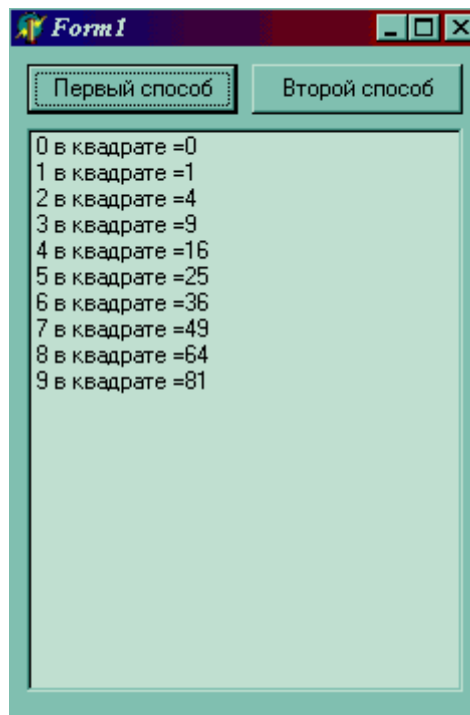
r: array of integer;

В этом примере я объявил переменную *r* типа массив целых чисел без указания размера (как мы указывали это в квадратных скобках [0..10]).

Чтобы указать размер массива можно воспользоваться функцией *SetLength*. У неё два параметра:

1. Переменная типа динамического массива.
2. Длина массива.

Давай посмотрим всё это на практике. Создай новый проект в Delphi, брось на форму две кнопки и один компонент *TListBox*:



Для первой кнопки мы напишем следующий текст:

```
var
  r:array of integer;
  i:Integer;
begin
  ListBox1.Items.Clear;
```

```
SetLength(r,10);

for i:=0 to High(r)-1 do
begin
  r[i]:=i*i;
  ListBox1.Items.Add(IntToStr(i)+' в квадрате =' + IntToStr(r[i]));
end;
```

В области объявлений *VAR* я объявил две переменные. Первая это *r* которая является массивом чисел типа *Integer*. Вторая *i* это переменная, которую я буду использовать в качестве счётчика. Переходим к самой процедуре:

Первая строка очищает все строки у *ListBox1*. Для этого вызывается процедура *ListBox1.Items.Clear*. Мы это уже проходили, но я напомним. У *ListBox1* есть свойство *Items*, где хранятся все строки. У *Items* есть метод *Clear*, который удаляет все находящиеся в нём строки.

Во второй строке вызывается процедура *SetLength*, которая выделила память для массива *r* (первый параметр), размером в 10 элементов (второй параметр). Обращение к элементом будет происходить как *r[номер_элемента]*. Элементы будут нумероваться от 0 до 9. Вообще, в программировании всё нумеруется с нуля.

Далее идёт цикл. Функция *High(r)* возвращает количество элементов в массиве *r*. В итоге получается, что цикл будет выполняться от *i:=0* (от нуля), до количества элементов в массиве *r* минус 1 (до 9). Внутри массива выполняется две строки:

```
r[i]:=i*i //Здесь i-му элементу массива присваивается i*i.
```

ListBox1.Items.Add(IntToStr(i)+' в квадрате =' + IntToStr(r[i])); Эта строка добавляет новый элемент в *ListBox1*. Функция *IntToStr* переводит число в строку.

С первой процедурой мы разобрались, теперь перейдём ко второй:

```
type
  TDynArr=array of integer;
var
  r:TDynArr;
  i:Integer;
begin
  ListBox1.Items.Clear;
  SetLength(r,10);

  for i:=0 to High(r)-1 do
    r[i]:=i*i;

  SetLength(r,20);
  for i:=10 to High(r)-1 do
    r[i]:=i*i;

  for i:=0 to High(r) do
    ListBox1.Items.Add(IntToStr(i)+' в квадрате =' + IntToStr(r[i]));
```

Эта процедура выполняет похожие действия, но с небольшими особенностями. В начале я объявляю новый тип: *TDynArr=array of integer*. После этого конструкция *r:TDynArr;* будет означать, что *r* относится к типу *TDynArr*, а тот относится к *array of integer*; . Это тоже самое, что мы писали в первой процедуре *r:array of integer*; , только такая конструкция удобней, если ты захочешь объявить несколько динамических массивов. Тебе не приходится сто раз писать громоздкую строку *r:array of integer*; , ты объявляешь новый массив как *TDynArr*.


Далее идёт всё та же очистка строк и выделение памяти под массив.

```
for i:=0 to High(r)-1 do  
  r[i]:=i*i;
```

Эта конструкция заполняет десять элементов квадратами числа *i*. После этого я снова вызываю функцию *SetLength(r,20);*, в которой говорю, что массив теперь будет состоять из 20-и элементов. Таким способом можно как увеличивать количество элементов, так и уменьшать.

```
for i:=10 to High(r)-1 do  
  r[i]:=i*i;
```

Здесь я заполняю квадратами числа *i* элементы начиная с 10 по последний. И в конце я снова заполняю *ListBox1* значениями элементов массива.

 На компакт диске, в директории \Примеры\Глава 10\DynArrays ты можешь увидеть пример этой программы.

10.2 Многомерные массивы.

Мы уже разобрались с массивами, но пока это только одномерные массивы, в которых данные располагаются в виде строки. Для Delphi это не предел и он может работать и с несколькими измерениями массива.

Например, допустим, что тебе надо держать таблицу из данных. Таблица будет состоять из пяти колонок и четырёх строк. В этом случае ты можешь завести четыре массива, в каждом из которых будут храниться по 5 элементов. Но это же не солидно!!! Вот тут на встречу приходят многомерные массивы.

Объявляются такие массивы практически так же как и одномерные, разница только в том, что когда ты в квадратных скобках указываешь длину массива, нужно указывать размеры строк и столбцов данных.

Рассмотрим пример объявления двухмерного массива из четырёх строк и пяти столбцов:

```
var  
  t:array[0..3, 0..4] of integer;
```

Как видишь, в квадратных скобках перечислены через запятую размеры строк и столбцов. Заметь, что я объявил массив от 0 до 3 – это будет четыре элемента и от 0 до 4, что будет 5 элементов.

Работа с таким массивом тоже достаточно простая:

```
var
  t:array[0..3, 0..4] of integer;
begin
  t[0][0]:=1;
  t[1][0]:=2;
  t[2][0]:=3;
  t[3][0]:=4;
  t[1][1]:=5;
end;
```

После выполнения этого примера наша таблица будет иметь вид:

```
1 5 0 0 0
2 0 0 0 0
3 0 0 0 0
4 0 0 0 0
```

Двухмерность не предел и ты можешь создавать и 3-х мерные массивы. Давай молча посмотрим на следующий пример:

```
var
  t:array[0..3, 0..4, 0..2] of integer;
begin
  t[0][0][0]:=1;
  t[1][0][0]:=2;
  t[2][0][0]:=3;
  t[3][0][0]:=4;
  t[1][1][0]:=5;
end;
```

10.3 Работа с файлами.

В этой части я познакомлю тебя, как можно работать с файлами. Мы научимся открывать их, записывать и читать из них данные, ну и, конечно же, закрывать.

Для работы с файлами многие предпочитают использовать WinAPI. Не пугайся этого слова, потому что работа с WinAPI в Delphi очень даже прозрачна и ты не ощутишь никаких проблем. Я тоже любил так работать, пока не нарвался на одну неприятность. В самых первых окнах для чтения из файла использовалась функция `_lread`. В Windows 95 появилась `ReadFile`. А сейчас рекомендуют использовать `ReadFileEx`, которая может работать с файлами большего размера. После каждого изменения функций WinAPI приходится переделывать весь код проги, потому что нет гарантии, что старые функции будут работать в новых версиях Windows.

Вот поэтому я стал использовать специализированный в Delphi объект *TFileStream*. Я и тебе советую делать это, потому что если Microsoft снова введёт какие-то нововведения, то Borland учтёт их в объекте и тебе нужно будет только перекомпилировать свои проги. Никаких изменений в код вносить не надо. Ты один раз изменяешь объект (или это делает Borland) и компилируешь все свои проги с новыми возможностями. И в любом случае, использование объекта намного проще.

Итак, давай взглянём на объект *TFileStream*. Для работы с ним, ты должен объявить какую-нибудь переменную типа *TFileStream*.

```
var  
f: TFileStream;
```

Вот так я объявил переменную *f* типа объекта *TFileStream*. Теперь можно проинициализировать переменную.

Инициализация – выделение памяти и установка значений по умолчанию.

За инициализацию в любом объекте отвечает метод *Create*. Нужно просто вызвать его и результат выполнения присвоить переменной. Например, в нашем случае нужно вызвать *TFileStream.Create*(какие-то параметры) и результат записать в переменную *f*.

```
f := TFileStream.Create(параметры);
```

Давай разберёмся, какие параметры могут быть при инициализации объекта *TFileStream*. У этого метода *Create* может быть три параметра, причём последний можно не указывать:

1. Имя файла (или полный путь к файлу) который надо открыть. Этот параметр – простая строка.
2. Режим открытия. Здесь ты можешь указать один из следующих флагов:
 - a. *fmCreate* – создать файл с указанным в первом параметре именем. Если файл уже существует, то он откроется в режиме для записи.
 - b. *fmOpenRead* – открыть файл только для чтения. Если файл не существует, то произойдёт ошибка. Запись в файл в этом случае не возможна.
 - c. *fmOpenWrite* – открыть файл для записи. При этом, во время записи текущее содержимое уничтожается.
 - d. *fmOpenReadWrite* – открыть файл для редактирования (чтения и записи).
3. Права, с которыми будет открыт файл. Тут опять ты можешь указать одно из следующих значений (а можешь вообще ничего не указывать):
 - a. *fmShareCompat* – при этих правах, другие приложения тоже имеют права работать с открытым файлом.
 - b. *fmShareExclusive* – при этом режиме другие приложения вообще не смогут открыть файл.
 - c. *fmShareDenyWrite* – при данном режиме другие приложения не смогут открывать этот файл для записи. Файл может быть открыт только для чтения.
 - d. *fmShareDenyRead* – при данном режиме другие приложения не смогут открывать этот файл для чтения. Файл может быть открыт только для записи.

е. *fmShareDenyNone* - вообще не мешать другим приложениям работать с файлом.

С первыми двумя параметрами всё ясно. Но зачем же нужны права доступа к файлам. Допустим, что ты открыл файл для записи. Потом его открыло другое приложение и тоже для записи. Вы оба записали какие-то данные. После этого твоё приложение закрыло файл и сохранило все изменения. Тут же другое приложение перезаписывает твои изменения, даже не подозревая о том, что они прошли. Вот так твоя инфа пропадает из файла.

Если ты пишешь однопользовательскую прогу и к файлу будет иметь доступ только одно приложение, то про права можно забыть и даже не указывать.

После того, как ты поработал с файлом, достаточно вызвать метод *Free*, чтобы закрыть файл.

f.Free;

Теперь давай познакомимся с методами чтения, записи и путешествия внутри файла. Начнём с путешествия. Когда ты открыл файл, позиция курсора устанавливается в самое начало и любая попытка чтения или записи будет происходить в эту позицию курсора. Если тебе надо прочитать или записать в любую другую позицию, то надо передвинуть курсор. Для этого есть метод *Seek*. У него есть два параметра:

1. Число, указывающее на позицию, в которую надо перейти. Если тебе нужно передвинуться на пять байт, то просто поставь цифру 5.

2. Откуда надо двигаться. Тут возможны три варианта:

- *soFromBeginning* – двигаться на указанные количество байт от начала файла.
- *soFromCurrent* - двигаться на указанные количество байт от текущей позиции в файле к концу файла.
- *soFromEnd* – двигаться от конца файла к началу на указанное количество байт.

Не забывай, что один байт – это один символ. Единственное исключение – файлы в формате Unicode. В них один символ занимает 2 байта. Так что надо учитывать в каком виде хранится информация в файле.

Итак, если тебе надо передвинуться на 10 символов от начала файла, то ты можешь написать следующий код:

f.Seek(10, soFromBeginning);

Метод *Seek* всегда возвращает смещение курсора от начала файла. Этим можно воспользоваться чтобы узнать где мы сейчас находимся, а можно и узнать общий размер файла. Если переместится в конец файла, то функция вернёт нам количество байт от начала до конца, т.е. полный размер файла.

В следующем примере я устанавливаю позицию в файле на 0 байт от конца файла, т.е. в самый конец. Тем самым я получаю полный размер файла:

Размер файла := f.Seek(0, soFromEnd);

Для чтения из файла нужно использовать метод `Read`. . И снова у этого метода два параметра:

1. Переменная, в которую будет записан результат чтения.
2. Количество байт, которые надо прочитать.

Давай взглянем на код чтения из файла, начиная с 15 позиции в файле.

```
var
f:TFileStream; //Переменная типа объект TFileStream.
buf: array[0..10] of char; // Буфер, для хранения прочитанных данных
begin
// В следующей строке я открываю файл filename.txt для чтения и записи.
f:= TFileStream.Create('c:\filename.txt', fmOpenReadWrite);

f.Seek(15, soFromCurrent); // Перемещаюсь на 15 символов вперед.
f.Read(buf, 10); // Читаю 10 символов из установленной позиции.
f.Free; // Уничтожаю объект и соответственно закрываю файл.
end;
```

Заметь, что в методе *Seek* я двигаюсь на 15 символов не от начала, а от текущей позиции, хотя мне нужно от начала. Это потому что после открытия файла текущая позиция и есть начало.

Метод *Read* возвращает количество реально прочитанных байт (символов). Если не произошло никаких проблем, то это число должно быть равно количеству запрошенных для чтения байт. Есть только два случая, когда эти числа отличаются:

1. При чтении был достигнут конец файла и дальнейшее чтение стало не возможным.
2. Ошибка на диске или любая другая физическая проблема.

Осталось только разобраться с методом для записи. Для записи мы будем использовать *Write*. У него так же два параметра.

1. Переменная, содержимое которой нужно записать.
2. Число байт для записи.

Пользоваться этим методом можно точно также как и методом для чтения.

Напоследок одно замечание: после чтения или записи текущая позиция в файле смещается на количество прочитанных байт. То есть текущая позиция становится в конец прочитанного блока данных.

Примера пока не будет, потому что в принципе и так всё ясно. А если есть вопросы, то на практике мы закрепим этот материал буквально через один или два раздела, когда будем работать с структурами.

10.4 Работа с текстовыми файлами

В предыдущей части я показал общий случай работы с файлами. В этой части мы познакомимся с частным случаем – текстовые файлы. В них информация расположена не сплошным одинарным блоком, а в виде строк текста. Было бы удобно воспринимать такие файлы в виде наборов строк.

Если мы попытаемся прочитать текстовый файл методами, описанными в прошлой части, то работать с текстом будет неудобно. Допустим, что у нас есть файл из двух строчек:

Привет!!!
Как жизнь?

Если прочитать его с помощью объекта TFileStream, то мы увидим весь текст в одну строку:

Привет!!!<CR><LF>Как жизнь?

Здесь <CR> - конец строки и <LF> - перевод каретки на новую строку. Так что, чтобы найти конец первой строки мы должны просканировать весь текст на наличие признака конца строки и перевода каретки (CR и LF). Это очень неудобно, каждый раз сканировать конец строки. А допустим, что у тебя файл из 100 строк и тебе нужно получить доступ к 75-й строке. Как ты думаешь, долго мы будем искать нужную строку? Нет, точнее сказать, что не долго, а неудобно.

Тут на помощь нам приходит объект TStrings, который является простым контейнером (хранилищем) для строк. Можно ещё пользоваться более продвинутым вариантом этого объекта TStringList. Если посмотреть на иерархию объекта TStringList (рис 1), то мы увидим, что TStringList происходит от TStrings. Это значит, что TStringList наследует себе все возможности объекта TStrings и добавляет в него новые.

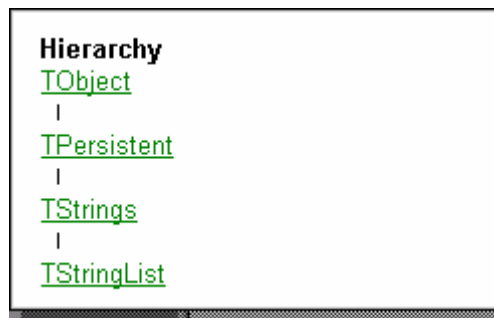


Рис 1 Иерархия объекта TStringList

Как всегда для работы с объектом надо объявлять переменную типа объект:

```
var
f:TStringList; //Переменная типа объект TStringList.
```

Инициализируется эта переменная как всегда методом *Create*. Никаких параметров не надо. Чтобы освободить память объекта и уничтожить его, как всегда применяется метод *Free*. Вот пример простейшего использования объекта:

```
var
f:TStringList; //Переменная типа объект TStringList.
begin
f:= TStringList.Create();
f.Free;
end;
```

В этом примере я только создал новый объект и сразу уничтожил, не используя его.

Давай снова вспомним, что *TStringList* происходит от *TStrings*. Использовать *TStrings* напрямую нельзя, потому что это абстрактный объект. *Абстрактный объект* – объект, который представляет из себя пустой шаблон. Он может даже ничего не уметь делать, а только описывать какой-то вид или шаблон, на основе которого можно выводить полноценные объекты. Вот так *TStringList* добавляет в *TStrings* свои функции так, что он становится полноценным объектом.

Итак, получается, что мы не можешь объявлять переменные типа *TStrings* и использовать этот объект, потому что это всего лишь шаблон. Это и так и не так. Переменную мы можем объявлять, но использовать сам объект не можем. Зато мы можем объявить переменную типа *TStrings*, но использовать эту переменную как объект *TStringList*, потому что он происходит от первого. Это значит, что следующий пример идентичен предыдущему:

```
var
f:TStrings; //Переменная типа объект TStringList.
begin
f:= TStringList.Create();
f.Free;
end;
```

В этом примере я объявил переменную типа *TStrings*, но при создании проинициализировал её объектом *TStringList*. Это вполне законная запись, потому что объект *TStringList* происходит от *TStrings*. Новая переменная *f* будет работать, как объект *TStringList*, хотя и объявлена как *TStrings*. Главное – каким объектом переменная проинициализирована.

Такие трюки можно проводить только с родственными объектами, но я постараюсь не использовать их в своей книге, хотя знать ты всё это обязан. А вдруг тебе попадётся исходник трюкача и тогда такая запись может тебя запутать.

Итак, давай познакомимся, как можно работать с помощью *TStringList* с текстовыми файлами. Всё очень просто. У него есть метод *LoadFromFile* для которого нужно указать имя текстового файла. После этого, через свойство *Strings* можно получить доступ к любой строчке, а в свойстве *Count* находится число указывающее на количество строк в файле.

```
var
f:TStrings; //Переменная типа объект TStringList.
begin
f:= TStringList.Create();
f.LoadFromFile('c:\filename.txt');// Загружаю текстовый файл
f.Strings[0]; // Здесь находится первая строчка файла
f.Strings[1]; // Здесь находится вторая строчка файла

f.Free;
end;
```

Давай напишем пример, который будет искать в файле строку «Привет Вася»:

```
var
f:TStrings; //Переменная типа объект TStringList.
i: Integer; // Счётчик
begin
f:= TStringList.Create();
f.LoadFromFile('c:\filename.txt'); // Загружаю текстовый файл

for i:=0 to f.Count-1 do // Запускаю цикл
begin // Начало для цикла

if f.Strings[i] = 'Привет Вася' then //Если i-я строка равна нужной то

Application.MessageBox('Строка найдена',
'Поиск закончен', MB_OKCANCEL)

end; // Конец для цикла

f.Free;
end;
```

Точно так же очень просто изменять данные в файле. Например, тебе надо изменить 5-ю строку на «Прощай станция Мир». Это можно сделать следующим образом:

```
var
f:TStrings; //Переменная типа объект TStringList.
i: Integer; // Счётчик
begin
f:= TStringList.Create();
f.LoadFromFile('c:\filename.txt'); // Загружаю текстовый файл

if f.Count>=5 then // Если в файле есть 5 строк то изменить
f.Strings[5] = 'Прощай станция Мир';

f.Add('Прощай');// Добавляю новую строку

f.SaveToFile('c:\filename.txt'); // Сохраняю результат
f.Free;
end;
```

На всякий случай, прежде чем изменить пятую строку я проверяю, есть ли в файле эти пять строк. Если окажется меньше пяти, то при попытке изменения данных произойдёт ошибка.

В этом же примере я добавляю в конец файла новую строку с помощью вызова метода Add. После этого я сохраняю результат в том же файле с помощью вызова метода SaveToFile. Если не вызывать метод сохранения, то все изменения пропадут, потому что данные изменяются в объекте, а не в файле, поэтому объект надо сохранять обратно в файл.

На этом можно закончить рассмотрение объекта, но я хочу ещё показать тебе несколько методов:

1. *Clear* – очистка содержимого объекта.
2. *Insert* – вставить строку. У этого метода два параметра – индекс строки куда нужно вставить и сама строка.

3. *Delete* – удалить строку. Здесь только один параметр – индекс удаляемой строки.

Вот теперь можно считать, что с текстовыми файлами и объектом *TStringList* покончено. Можно двигаться дальше.

10.5 Приведение типов

Сейчас я постараюсь, как можно подробнее остановиться на теме приведения типов. В любой программе может понадобиться преобразование данных из одного типа в другой.

Преобразование типов делится на два вида: преобразование несовместимых типов и преобразование совместимых типов. В качестве несовместимых типов можно привести пример превращения строки в число. Допустим, что у тебя есть строка «12345». Это строка, содержащее число. Но ты не можешь производить с такой строкой математических действий, потому что это строка, хотя и содержащее число. Для начала нужно преобразовать эту строку в число.

Преобразование целых чисел в строку и обратно

Начну я с рассмотрения специальных функций для преобразования несовместимых типов. Самое частое, что может тебе понадобиться – преобразование строк в число и обратно. Допустим, что тебе нужно написать программу, в которой пользователь будет вводить число в компонент *TEdit*. Чтобы получить доступ к содержимому *Edit1* надо написать *Edit1.Text*. Так мы получим текстовое представление числа. Чтобы его преобразовать, необходимо воспользоваться спец функцией. Вот давай и будем знакомиться с подобным примером.

Для преобразования строки в число используется функция *StrToInt*. У неё только один параметр – строка, а на выходе она возвращает число.

```
var
  ch:Integer;
begin
  ch:=StrToInt(Edit1.Text); // Преобразовываю Edit1.Text в число
end;
```

В этом примере я присвоил в переменную *ch* значение, содержащееся в *Edit1.Text* преобразованное в число. Теперь мы можешь производить математические действия с введённым числом.

Обратное преобразование (превращение числа в строку) можно произвести с помощью функции *IntToStr*.

```
var
  ch:Integer;
begin
  ch:=StrToInt(Edit1.Text); // Преобразовываю Edit1.Text в число
  ch:=ch+1;
  Edit1.Text:=IntToStr(ch); // Преобразовываю ch в строку
end;
```

Когда ты преобразовываешь строку в число, ты должен быть уверен в том, что строка содержит число. Если в строке будет хоть один символ не относящийся к цифре, то во время преобразования произойдёт ошибка. Чтобы избежать от ошибок, можно использовать исключительные ситуации, заключая преобразование между *try* и *except*. Но есть ещё один способ – использовать функцию *StrToIntDef* у которой уже два параметра:

1. Строка, которую надо преобразовать
2. Значение по умолчанию, которое будет возвращено, если произошла ошибка.

Итак, наш пример можно подкорректировать следующим образом:

```
var
  ch:Integer;
begin
  ch:=StrToIntDef(Edit1.Text, 0); // Преобразовываю Edit1.Text в число
end;
```

В этом примере, если произойдёт ошибка во время преобразование, то функция не будет ругаться, а вернёт значение 0.

Преобразование даты в строку и обратно

Теперь познакомимся с преобразованием даты. Для этого есть несколько функций:

1. *DateToStr* – преобразовывает дату в строку. Единственный параметр, который надо указать – переменную типа *TDateTime* и на выходе получим строку.
2. *StrToDate* – преобразование строки в дату. Указываешь строку (например «11/05/2001»)и получаешь дату.
3. *FormatDateTime* – форматирование даты и времени. Это очень интересная функция, поэтому на ней я остановлюсь подробнее.

У функции *FormatDateTime* два параметра:

1. Формат строки в которую надо переписать дату
2. Переменная типа *TDateTime*, которую надо преобразовать.

Самое интересное здесь – это формат строки. Он может содержать следующие символы:

d – показать дату не подставляя нули в начале (1, 2, 3 ...30, 31).

dd – показать дату подставляя если нужно в начале ноль. В этом случае, если дата меньше 10, то она будет отражаться как 01, 02 ... 09.

ddd – показать день недели используя короткий формат (Пн, Вт, Ср...).

dddd – показать день недели с полным названием (Понедельник, Вторник ...)

dddddd – показать дату используя короткий формат.

dddddd – показать дату используя полный формат (Например 10 дата /02/2002 будет переведена в «10 февраля 2002».

m – показать месяц без добавления нулей (1, 2, ..., 11, 12).

mm – показать месяц с добавлением нулей (01, 02, ...11, 12).

mmm – показать короткое название месяца.

mmmm – показать полное название месяца (январь, февраль....).

yy – показать короткий года (98, 99, 00, 01).

уууу – показать полный год.

h – показать часы не добавляя в начале нулей.

hh – показать часы с добавлением в начале нулей.

n – показать минуты не добавляя в начале нулей.

nn – показать минуты с добавлением в начале нулей.

s – показать секунды не добавляя в начале нулей.

ss – показать секунды с добавлением в начале нулей.

z – показать миллисекунды не добавляя в начале нулей.

zz – показать миллисекунды с добавлением в начале нулей.

am/pm – использовать 12-и часовое представление (до полудня/после полудня).

Это практически полный обзор возможностей, а теперь посмотрим пару примеров:

```
FormatDateTime('dd/mm/yyyy', Date()); // Дата будет в виде "24/02/2002"  
FormatDateTime('dddddd', Date()); // Дата будет в виде "24 февраля 2002"  
FormatDateTime('hh:nn', Time()); // Время будет в виде "10:48"  
FormatDateTime('hh:nn - ss', Time()); // Время будет в виде "10:48 - 24"
```

Преобразование вещественных чисел

Теперь перейдём к числам с плавающей точкой. Когда ты строишь математику в своей программе, то можешь столкнуться с вещественными числами. Например, если у тебя есть какая-то формула, в которой используется деление, то результат её выполнения будет всегда дробным, даже если ты уверен в целостности ответа. Например, ты делишь 10 на 2, и должен получить результат 5. Хотя результат целое число, компилятор будет представлять его как дробное.

```
var  
i:Integer;  
begin  
i:=10/2;  
end;
```

Если ты попытаешься откомпилировать такой код, то увидишь следующую ошибку:
«Incompatible types: 'Integer' and 'Extended'»

Тут появляется два выхода:

1. Записывать результат в переменную вещественного типа. Но он подходит не всегда, поэтому лучше перейти сразу ко второму методу.
2. Округлять результат

Для округления существует очень удобная функция *round*:

```
var  
i:Integer;  
begin  
i:=round(10/2);  
end;
```

Если ты решил хранить результат в переменной вещественного типа, то могут возникнуть проблемы с выводом результата. Для этого может понадобиться

преобразование вещественного числа в строку. Для этого есть функция *FloatToStr*, которой надо передать дробное число и получить строку. Точно так же есть и обратное преобразование *StrToFloat*, где ты передаешь строку, а получаешь вещественное число.

Отдельного разговора требует функция *FormatFloat*, которая форматирует вещественное число по твоим нуждам. Тут есть два параметра: строка формата и само число.

Следующая табличка показывает разные варианты формата. В первой колонке показаны возможные форматы указываемые в первом параметре функции *FormatFloat*. В остальных колонках показано, что произойдет с разными числами при данном формате (табличка взята из файла помощи по Delphi):

Строка указываемая в формате	Форматируемые числа			
	1234	-1234	0.5	0
0	1234	-1234	1	0
0.00	1234.00	-1234.00	0.50	0.00
###	1234	-1234	.5	
###0.00	1,234.00	-1,234.00	0.50	0.00
###0.00;(###0.00)	1,234.00	(1,234.00)	0.50	0.00
###0.00;;Zero	1,234.00	-1,234.00	0.50	Zero
0.000E+00	1.234E+03	-1.234E+03	5.000E-01	0.000E+00
#####E-0	1.234E3	-1.234E3	5E-1	0E0

Вот пока что и всё, что я хотел сказать тебе про преобразование несовместимых типов.

10.6 Преобразование совместимых типов

Теперь можно познакомиться и с несовместимыми типами. Под совместимыми типами я понимаю типы данных, которые схожи по своим признакам, но хранят данные в разном виде. Например, есть несколько видов строк: строка, оканчивающаяся нулём и строка, первый байт которой указывает на её длину. Вроде и то и другое строки, но если где-то нужен определённый тип строки, то придётся преобразовывать свою строку именно в тот формат.

Преобразование строк

Допустим, что у тебя есть строка типа *String* и ты хочешь её преобразовать в *PChar*. Для такого преобразования нужно написать требуемый тебе тип и в скобках указать свою строковую переменную: *NewStr:=PChar(MyStr);*. В этом примере переменная *MuStr* имеет тип *String*, а мы приводим её в вид *PChar*. Вот так происходит преобразования совместимых типов.

Вот такое преобразование строк мы будем делать очень часто при вызове WinAPI функций, потому что там большинство строк имеет именно *PChar* тип.

Сейчас я не могу придумать какой-нибудь ещё пример преобразования, но уже в этой главе ты воспользуешься такой возможностью в одном из моих примеров.

10.7 Указатели

Пора познакомиться с указателями. Это очень удобная и сильная вещь, которой мы так же будем часто пользоваться. Уже в этой главе мы будем работать с указателями на структуры, которые расположим в динамической памяти.

Но прежде чем что-то объяснять, я хочу показать тебе, зачем нужны указатели. Давай вспомним про процедуры, и я расскажу тебе, как происходит их вызов. Допустим, у тебя есть процедура с именем *MyProc* у которой есть два параметра: число и строка. Как происходит вызов такой процедуры, и как ей передаются эти параметры? Очень просто. Сначала параметры поднимаются в стек (напомню, что стек – это область памяти для хранения твоих временных/локальных переменных). Сначала заносится первый параметр, затем второй и после этого вызывается процедура. Прежде чем процедура начнёт своё выполнение, она достаёт эти параметры из стека в обратном порядке.

Теперь вспомним о наших параметрах. Первый – это число, которое будет занимать два байта. Когда мы подыдем его в стек, то оно займёт там свои положенные два байта. Второй параметр – строка. Каждый символ строки – это отдельный байт. Допустим, что наша строка состоит из 10 символов, значит для передачи такой строки в процедуру, в стеке понадобится 10 байт плюс один байт для указания конца строки или размерности (это зависит от типа строки). Всего для передачи в процедуру нам понадобится в стеке как минимум 12 байт. Это не так уж и много, поэтому такое можно себе позволить.

А теперь представь, что строка, которую надо передать в процедуру, состоит из 1000 символов. Вот тут нам понадобится в стеке уже около килобайта. При нынешних размерах памяти на это уже никто не обращает внимания, но народ забывает про то, что такая строка сначала копируется в память стека, а потом достаётся оттуда. Такое копирование большого размера памяти отнимает достаточно много времени и твоя программа тратит лишнее время на бессмысленное копирование большой строки.

Выход из сложившейся ситуации достаточно прост – можно не передавать строку, а только передать указатель на область памяти, где находится эта строка. Любой указатель занимает всего четыре байта, а это уже существенная экономия. Мы просто передаём два байта указывающих на длинную строку, а процедура достаёт эти два байта и уже знает, где можно взять строку, когда она понадобится.

Указатель в Delphi объявляется как *Pointer*. Например, давай объявим переменную *p* типа указатель:

```
var  
p:Pointer
```

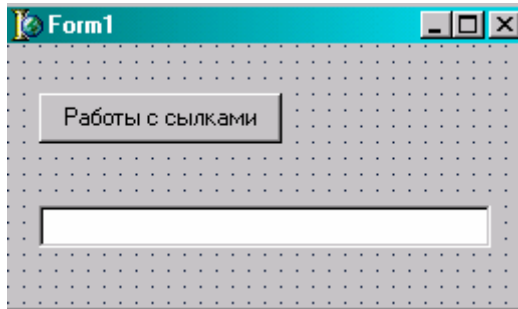
Для того, чтобы получить адрес переменной или объекта, необходимо перед его именем поставить знак *@*. Например, у тебя есть строка *Str* и чтобы присвоить её адрес в указатель *p*, надо выполнить следующее: *p:=@Str;* Теперь в указателе находится адрес строки. Если ты будешь напрямую читать указатель, то увидишь адрес, а для того чтобы увидеть содержащиеся по этому адресу данные, надо разыменовывать указатель. Для этого надо написать *p^*. Итак, мы пришли к следующему:

p:=@Str – получить адрес строки.

p – указатель на строку.

p^ - данные, содержащиеся по адресу указанному в *p*.

Давай создадим маленькое приложение, которое будет работать с указателями. Для этого создай форму приблизительно следующего содержания:




По нажатию кнопки «Работа со ссылками» напиши следующее:

```
var
  p:Pointer
  Str:String;
begin
  p:=@Str; // Присваиваю указателю ссылку на строку
  Str:='Привет мой друг'; // Изменяю значение строки
  Edit1.Text:=String(p^); // Вывожу текст
end;
```

В этом примере, в первой строке я присваиваю указателю *p* ссылку на строку *Str*. После этого я меняю содержимое строки. И в последней строчке я вывожу содержащийся по адресу *p* текст. Для этого приходится явно указывать, что по адресу *p* находится именно строка *String(p^)*. Это необходимо, потому что данные, расположенные по определённому адресу могут иметь совершенно любой тип. Как видишь, жёсткое указание типа похоже на преобразование типов, поэтому никаких проблем с этим не должно возникнуть.

Заметь, что я изменяю строку после присваивания адреса строки в переменную указатель, и изменённые данные всё равно будут отражены в указателе. Это потому что указатель всегда показывает на начало строки, и если мы её изменим, указателю будет всё равно, потому что новые данные будут расположены по тому же адресу, и *p* будет указывать на новую строку.

 На компакт диске, в директории \Примеры\Глава 10\Pointers ты можешь увидеть пример этой программы.

Мы уже не раз использовали указатели, просто не углублялись в их изучение. Каждая переменная типа объекта — это тоже указатель на объект. Просто его использование выполнено так, чтобы не смущать тебя адресацией и разыменовыванием.

Любой переменной указателю можно присвоить нулевое значение, только это не 0, а **nil**, например *p:=nil*;. Когда ты присваиваешь нулевое значение, то ты как бы уничтожаешь ссылку. Точно так же, если ты переменной объекту присвоишь нулевое значение, ты его уничтожишь, и объект больше не будет существовать.

А что если у тебя две переменные указывают на один и тот же адрес данных? Неужели при уничтожении одного из них данные уничтожатся и вторая переменная будет указывать на несуществующие данные? Нет, объект будет уничтожен только после того, как все указатели на него будут уничтожены.

10.8 Структуры, записи

Сейчас я хочу тебя познакомить поближе со структурами и записями. Точнее сказать, оба понятия означают одно и то же. Просто в C++ принято говорить «структура», а в Delphi говорят «запись». Так как я знаю оба языка, я больше привык к понятию «структура», потому что оно больше отражает смысл этого понятия. Тебе нужно только привыкнуть, что структура – это та же запись.

Я уже упоминал о структурах немного раньше. Они похожи на объекты, только не имеют методом и событий, а только свойства.

Для объявления структуры используется следующий вид:

```
Имя структуры = record
  Свойство1: Тип свойства1;
  Свойство2: Тип свойства2;
  ....
end;
```

Давай опишем структуру, в которой будут храниться параметры окна:

```
WindowSize = record
  Left:Integer;
  Top:Integer;
  Width:Integer;
  Height:Integer;
end;
```

Я назвал новую структуру как *WindowSize* и объявил внутри четыре свойства: Left, Top, Width, Height.

Свойства я объявлял каждое в отдельности, хотя в данном случае все они имеют один тип и их можно просто перечислить через запятую и указать, что все они имеют целый тип:

```
WindowSize = record
  Left, Top, Width, Height:Integer;
end;
```

Точно так же можно объявлять и переменные в разделе **var**, когда несколько переменных имеют один и тот же тип.

Теперь давай разберёмся, как можно использовать нашу структуру. Для этого надо определить переменную типа структура:

```
var
  ws: WindowSize;
```

Структуры – это простые сгруппированные наборы свойств, которые по умолчанию не требуют выделения памяти. Они создаются локально в стеке. Напоминаю, что стек – область памяти для хранения локальных/временных переменных. Так же следует напомнить чем отличаются локальные переменные от глобальных:

Локальные переменные – объявляются и создаются при входе в процедуру и уничтожаются при выходе. *Глобальные переменные* – создаются при запуске программы и уничтожаются при выходе из неё. Это значит, что глобальные переменные существуют на протяжении всего времени выполнения программы. Локальные переменные существуют, только когда выполняется код процедуры. После первого выполнения процедуры локальные переменные уничтожаются и при следующем вызове создаются снова с нулевым значением. Поэтому, если надо сохранить значение переменной после выхода из процедуры, её следует объявить как глобальную.

Итак, переменная объявлена. Инициализация и уничтожение не требуется, поэтому можно сразу же её использовать. Для доступа к переменным структуры нужно написать имя структуры и через точку указать тот параметр, который тебя интересует. Например, для доступа к параметру *Left* необходимо написать: *ws.Left*.

Давай напишем пример, который будет после закрытия программы сохранять текущие значения позиции окна в структуре, а потом эту структуру будем записывать в файл. Для записи будет использоваться простой бинарный файл, значит, воспользуемся объектом *TFileStream*.

Итак, создай новое приложение. В разделе **Type** опиши нашу структуру:

```
type
  WindowsSize = record
    Left, Top, Width, Height: Integer;
  end;

  TForm1 = class(TForm)
  private
  public
  end;
```

Теперь создай обработчик события *OnClose* для формы. Здесь мы заполним структуру значениями позиции окна и сохраним в бинарный файл:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
  ws: WindowsSize;
  f: TFileStream;
  Str: String;
begin
  ws.Left:=Left; // Заполняем левую позицию
  ws.Top:=Top; // Заполняем правую позицию
  ws.Width:=Width; // Заполняем ширину окна
  ws.Height:=Height; // Заполняем высоту окна

  f:=TFileStream.Create('size.dat', fmCreate); // Создаю файл Size.dat
  f.Write(ws, sizeof(ws)); // Записываю структуру
  f.Free; // Закрываю файл
end;
```

Обрати внимание, что при записи в файл, я должен указывать в качестве параметра буфер памяти для записи. Я указываю свою структуру. В качестве второго параметра нужно указывать размер записываемых данных. Для определения размера структуры я использую функцию *SizeOf*, которая вернёт мне необходимый размер.


Ну а теперь разберёмся с чтением из файла, которое происходит подобным образом:

```
procedure TForm1.FormShow(Sender: TObject);
var
  ws:WindowsSize;
  fs:TFileStream;
begin
  if FileExists('size.dat') then // Если файл Size.dat существует, то
  begin
    fs:=TFileStream.Create('size.dat', fmOpenRead); // Открываю файл Size.dat
    fs.Read(ws, sizeof(ws)); // Читаю содержимое структуры
    fs.Free; // Закрываю файл

    // Далее я устанавливаю сохранённые размеры и позицию окна на родину
    Left:=ws.Left;
    Top:=ws.Top;
    Width:=ws.Width;
    Height:=ws.Height;
  end;
end;
```

Первым делом я вызвал функцию *FileExists* которой указал имя интересующего меня файла. Эта функция проверила на существование файл. Если он существует, то я могу попытаться прочитать иначе это делать бесполезно. При чтении я так же читаю сразу всю структуру.

Как видишь, структуры очень удобны для хранения каких-либо структурированных данных. Конечно же, пример не очень удачный, потому что такие параметры лучше сохранять в реестре, а не в файле, но главное – это сам процесс. Я буду часто использовать структуры там, где это необходимо, потому что они действительно упрощают процесс кодирования.

 На компакт диске, в директории \Примеры\Глава 10\Records ты можешь увидеть пример этой программы.

10.9 Храним структуры в динамической памяти

Структуры могут быть не только локальными (храниться в стеке) но и динамическими (располагаться в динамической памяти). Почему память называется динамической? Да потому что стек создаётся автоматически при старте проги, а вот дополнительную память нужно выделять самому. Её можно добавлять и удалять в процессе работы проги, наверно поэтому её называют динамической.

Когда объявляешь структуру, то можешь указать и её динамический тип. Для этого нужно объявить ещё одну переменную и присвоить ей «*ИмяСтруктуры*». Чаще всего в качестве нового имени используют то же самое имя, только в начале добавляют букву «Р» и объявление это делают прямо перед объявлением структуры:

```
type
  PWindowSize = ^ WindowSize;
  WindowSize = record
    Left, Top, Width, Height: Integer;
  end;
```

В этом примере *PWindowSize* - ссылка на структуру. Теперь, чтобы разместить нашу структуру не в стеке, а в динамической памяти мы должны использовать именно *PwindowsSize*:

```
var
  ws: PWindowSize;
begin
  ws:=New(PWindowSize); // Выделяем память
  ws.Left:=10; // Изменяем одно свойство
  Dispose(ws); // Уничтожаем память
end;
```

В этом примере я объявил переменную *ws* типа *PWindowSize*. Это значит, что *ws* – это всего лишь указатель и в самом начале он нулевой. Теперь нам надо этому указателю выделить память размером со структуру *PWindowSize*. Для этого ей надо присвоить результат работы функции *New*. Эта функция выделяет динамическую память под указанный в качестве параметра объект и возвращает указатель на эту память. После этого в указателе *ws* находится выделенная память, подготовленная для использования в качестве структуры *PWindowSize*.

Доступ к свойствам остаётся такой же, поэтому нет смысла задерживаться на этом. Но вот в глаза сразу же бросается вызов функции *Dispose*. Так как мы выделили динамическую память, её нужно освободить и для этого служит именно эта функция. Просто передай ей в качестве параметра указатель, и функция корректно обнулит его.

Помни, что если ты объявил переменную типа указатель на структуру (в нашем примере это *PWindowSize*), то для такого указателя обязательно нужно сначала выделить память и потом освободить его. Если ты объявляешь переменную типа структура (в нашем примере это *WindowSize*), а не указатель, то такая структура автоматически расположится в стеке и ничего не надо выделять и освобождать.

10.10 Поиск файлов

В этой главе мы уже узнали о работе с файлами и узнали, что такое структуры и как с ними работать. Сейчас я хочу тебе показать, как можно организовать поиск файлов. В этом примере мы закрепим большинство навыков описанных в этой главе.

Для начала разберёмся с алгоритмом поиска файлов, а потом подробно рассмотрим каждую из необходимых функций:

```
//Запускаю поиск
hFindFile := FindFirst(Маска поиска, Атрибуты , Информация);
//Проверяю корректность найденного файла
if hFindFile <> INVALID_HANDLE_VALUE then
  //Если корректно, то запускается цикл repeat - until.
  repeat
```



```
//Здесь вписаны операторы, которые нужно выполнять.  
until (FindNext(Информация) <> 0);  
FindClose(Информация);
```

FindFirst - открывает поиск. В качестве первого параметра выступает маска поиска. Если ты укажешь конкретный файл, то система найдёт его. Но это не серьёзно, лучше искать более серьёзные вещи. Например, ты можешь запустить поиск всех файлов в корне диска C. Для этого первый параметр должен быть 'C:*.*'. Для поиска только файлов EXE, в папке Fold ты должен указать 'C:\Fold*.exe'.

Второй параметр - атрибуты включаемых в поиск файлов. Я использую *faAnyFile*, чтобы искать любые файлы. Тебе доступны

faReadOnly - искать файлы с атрибутом *ReadOnly* (только для чтения).
faHidden - искать скрытые файлы.
faSysFile - искать системные файлы.
faArchive - искать архивные файлы.
faDirectory - искать директории.

Последний параметр - это структура, в которой нам вернётся информация о поиске, а именно имя найденного файла, размер, время создания и т.д. После вызова этой процедуры, я проверяю на корректность найденного файла. Если всё в норме, то запускается цикл **Repeat - Until**.

Мы уже рассматривали с тобой циклы, но я всё же решил повториться и напомнить тебе работу используемого мной цикла. Он выполняет операторы, расположенные между **repeat** и **until**, пока условие расположенное после слова **until** является верным. Как только условие нарушается, цикл прерывается.

Хочу предупредить, что функция поиска, может возвращать в качестве найденного имени в структуре *SearchRec* (параметр *Name*) точку или две точки. Если ты согласишься на директорию, то таких файлов не будет. Откуда берутся эти имена? Имя файла в виде точки указывает на текущую директорию, а имя файла из двух точек указывает на директорию верхнего уровня. Если я встречаю такие имена, то я их просто отбрасываю.

Структура

```
type  
TSearchRec = record  
  Time: Integer; // Время создания найденного файла  
  Size: Integer; // Размер найденного файла  
  Attr: Integer; // Атрибуты найденного файла  
  Name: TFileName; // Имя найденного файла  
  ExcludeAttr: Integer; // Исключаемые атрибуты найденного файла  
  FindHandle: THandle; // Указатель необходимый для поиска  
  FindData: TWin32FindData; // Структура поиска файла Windows  
end;
```

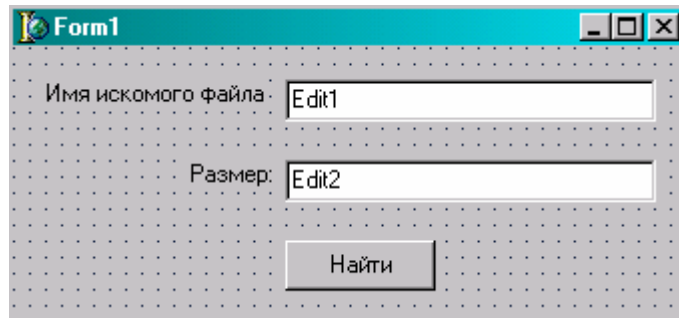
Функция *FindNext* заставляет найти следующий файл, удовлетворяющий параметрам, указанным в функции *FindFirst*. Этой функции нужно передать структуру *SearchRec*, по которой будет определено, на каком месте сейчас остановлен поиск и с этого момента он будет продолжен. Как только будет найден новый файл, функция вернёт в структуре *SearchRec* информацию о новом найденном файле.

Функция *FindClose* закрывает поиск. В качестве единственного параметра нужно указать всё ту же структуру *SearchRec*.

Давай теперь напишем какой-нибудь реальный пример, который наглядно покажет работу с функциями поиска файлов. Какой бы пример тебе написать?

Давай посмотрим на структуру *TSearchRec*. Как видишь, она умеет возвращать размер найденного файла. Вот и тема для примера – мы напишем код, который будет определять размер указанного файла.

Создай новый проект и брось на форму два компонента TEdit и одну кнопку. Можешь ещё украсить всё это текстом. У тебя должно получиться нечто похожее на этот рисунок:




По нажатию кнопки напиши следующий текст:

```
var
  SearchRec:TSearchRec;
begin
  // Ищем файл
  if FindFirst(Edit1.Text,faAnyFile,SearchRec)=0 then

    // Забираем размер
    Edit2.Text:=IntToStr(SearchRec.Size)+ 'байт';

    //Закрываем поиск
    FindClose(SearchRec);
end;
```

 На компакт диске, в директории \Примеры\Глава 10\FindFile ты можешь увидеть пример этой программы.

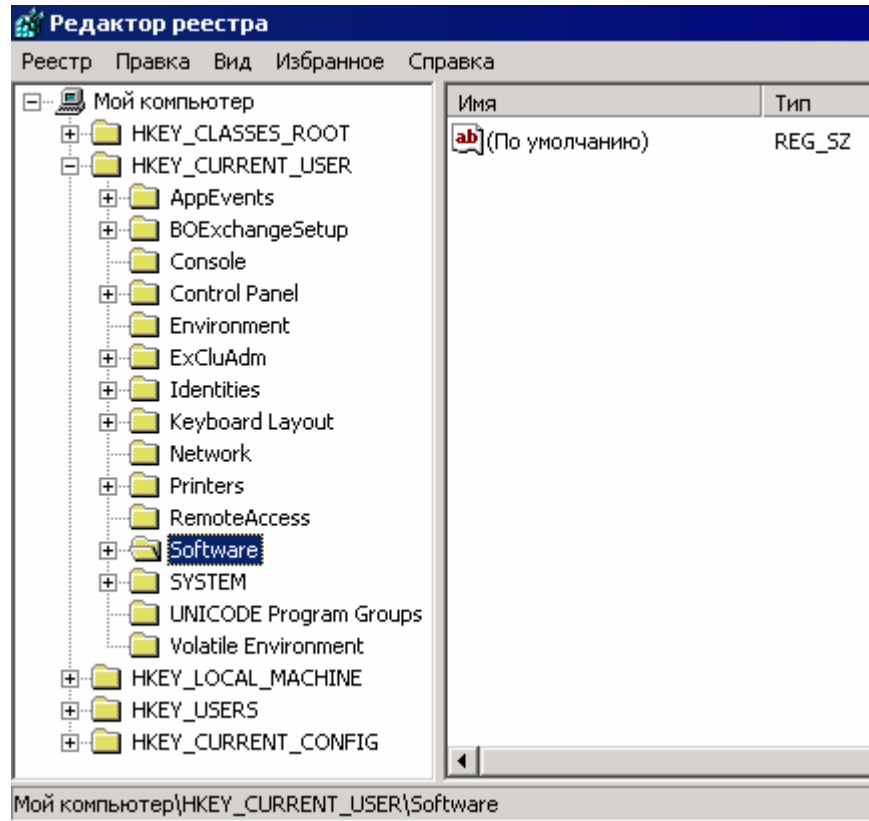
10.11 Работа с системным реестром

В этой главе я хочу тебе рассказать, как можно работать с системным реестром. Лично я люблю для этого использовать объект *TRegIniFile*. Он достаточно прост и удобен, поэтому я и тебе советую работать с ним. Для простого сохранения каких-то параметров программы этого объекта будет достаточно.

Давай разберёмся с этим объектом. Допустим, что у нас есть переменная *RegIni* типа *TRegIniFile*. Чтобы её инициализировать, нужно присвоить переменной результат вызова метода *Create* объекта *TRegIniFile*:

```
RegIni:=TRegIniFile.Create('Software');
```

По умолчанию, при инициализации ты получаешь доступ к разделу HKEY_CURRENT_USER. Методу *Create* нужно передать только один параметр – имя подраздела, который будет сразу открыт в разделе HKEY_CURRENT_USER.



Итак, после выполнения этого кода мы получили доступ к разделу HKEY_CURRENT_USER\Software. А что если ты хочешь открыть ещё подраздел и получить доступ к HKEY_CURRENT_USER\Software\Microsoft. Для открытия подразделов у объекта *TRegIniFile* есть метод *OpenKey*. Вот так можно открыть подраздел «Microsoft»:

```
RegIni.OpenKey('Microsoft', true);
```

У метода *OpenKey* уже два параметра:

1. Имя подраздела, который надо открыть.
2. Надо ли создавать подраздел, если он не существует.

Если в качестве второго параметра передать **false**, и подраздел не будет существовать, то произойдёт ошибка и ничего не откроется, т.е. ты останешься там же, где и был. Ну а если мы передадим **true** и раздел не будет существовать, то программа автоматически создаст его.




Теперь разберёмся с чтением и записью. Для чтения есть несколько методов:

- *ReadBool* – прочитать булево значение (true или false).
- *ReadInteger* – прочитать целое число.
- *ReadString* – прочитать строку.

Все они очень похожи и имеют одинаковое количество параметров. Единственная разница – в типе третьего параметра. Давай подробно рассмотрим *ReadString*, а остальное уже будем использовать по аналогии с этим методом.

У методов чтения есть три параметра:

1. Имя подраздела, в из которого мы хотим прочитать. Допустим, что мы открыли раздел Microsoft и находимся сейчас в реестре по адресу HKEY_CURRENT_USER\Software\ Microsoft. Если мы захотим прочитать строку из подраздела HKEY_CURRENT_USER\Software\Microsoft\MySoftware, то в качестве первого параметра ты должен написать *MySoftware*.
2. Имя параметра. Если ты хочешь, чтобы параметр звался как *Path*, то укажи *Path* :).

Имя	Тип	Значение
 (По умолчанию)	REG_SZ	(значение не присвоено)
 GridLineCount	REG_SZ	1
 Path	REG_SZ	F:\Projects\Организатор

3. Значение, которое будет использоваться по умолчанию, если такой параметр не существует. Для метода *ReadString* это должна быть строка или переменная типа строка.



Сразу хочу предупредить, что даже если ты будешь читать или записывать в реестр число с помощью методов *WriteInteger* или *ReadInteger*, объект *TRegIniFile* всё равно будет сохранять и читать эти числа как строки. А только после прочтения преобразовывать в число. Так что *TRegIniFile* реально хранит все данные в реестре только как строки. Если ты хочешь сохранять числа в реестре как самые простые числа, то нужно воспользоваться объектом *TRegistry*.

Итак, давай взглянем на команду чтения в действии:

```
Str:=RegIni.ReadString('MySoftware', 'Path', 'c:\');
```

В этом примере я читаю из подраздела *MySoftware* параметр *Path*. Если такой параметр не существует и не может вернуть значение, то будет возвращено значение по умолчанию «C:\». Результат чтения я записываю в переменную *Str*.

Точно так же происходит и запись, только в качестве третьего параметра нам надо указать не значение по умолчанию, а значение, которое надо записать. Для записи используются так же три метода:

- *WriteBool* – записать булево значение (true или false).
- *WriteInteger* – записать целое число.
- *WriteString* – записать строку.

Простейший пример записи выглядит так:

```
RegIni.WriteString('MySoftware', 'Path', 'c:\Windows');
```

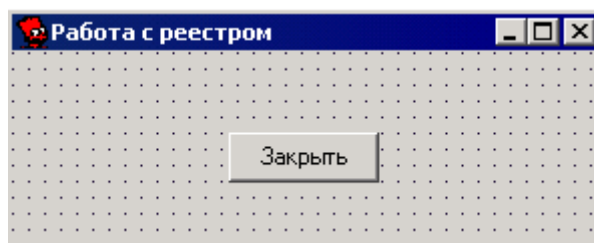
В этом примере я записываю в подраздел *MySoftware* параметр *Path*. Значение, которое будет записано равно третьему параметру - «C:\Windows».

После всех операций с реестром, его нужно закрыть с помощью метода *Free*:

```
RegIni.Free;
```

Для примера давай напишем программку, которая будет сохранять свои параметры при выходе и восстанавливать позицию и размер при запуске.

Я создал простейшую форму с одной только кнопкой «Закреть».



Теперь я создал обработчик события *OnShow*, в котором я должен восстановить параметры программы, которые были после последнего закрытия проги. Чтобы не загромождать этот обработчик я просто написал вызов метода *LoadProgParam*. Этого метода пока не существует, но мы его скоро напишем.

```
procedure TForm1.FormShow(Sender: TObject);  
begin  
  LoadProgParam;  
end;
```

Теперь я создал обработчик события *OnClose*. Здесь будут сохраняться параметры окна. Я опять не буду ничего загромождать и просто вызову метод *SaveProgParam*.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
  SaveProgParam;  
end;
```

Если ты сейчас попытаешься скомпилировать прогу, то получишь три ошибки. Компилятор Delphi проругается на то, что не может найти процедуры *LoadProgParam* и *SaveProgParam*. Давай напишем их. Для этого подымись в начало модуля и найди раздел описания закрытых процедур **private**. Опиши там эти две процедуры без всяких параметров:

```
private
{ Private declarations }
procedure LoadProgParam;
procedure SaveProgParam;
```

Теперь нажми сочетание клавиш Ctrl+Shift+C и Delphi создаст заготовки под эти процедуры:

```
procedure TForm1.LoadProgParam;
begin
end;

procedure TForm1.SaveProgParam;
begin
end;
```

Теперь напишем сами эти процедуры. Начнём с *SaveProgParam*:

```
procedure TForm1.SaveProgParam;
var
  FIniFile: TRegIniFile;

begin
  FIniFile := TRegIniFile.Create('Software'); // Инициализирую реестр

  FIniFile.OpenKey('VR',true); // Открываю раздел
  FIniFile.OpenKey('VR-Online',true); // Открываю ещё один раздел

  if WindowState=wsNormal then
  begin
    FIniFile.WriteInteger('Option', 'Width', Width);
    FIniFile.WriteInteger('Option', 'Heigth', Height);
    FIniFile.WriteInteger('Option', 'Left', Left);
    FIniFile.WriteInteger('Option', 'Top', Top);
  end;

  FIniFile.WriteInteger('Option', 'WinState', Integer(WindowState));

  FIniFile.Free; //Освобождаю реестр
end;
```

После инициализации реестра и подготовки разделов я делаю проверку, в каком состоянии находится окно. Если *WindowState* равно *wsNormal*, то я сохраняю параметры окна. Если нет, то этого делать не надо. Если у тебя стоит разрешение экрана 800x600, то при максимизированном окне значение ширины окна будет 802, а высоты 602. Эти значения больше реального разрешения и если ты установишь их при загрузке, то изменить размеры окна мышкой будет очень трудно.

После этого я сохраняю состояние окна - *WindowState*. Так как оно имеет тип *TWindowState*, то мне приходится приводить этот тип к *Integer* с помощью *Integer(WindowState)*.

Процедура *LoadProgParam* работает таким же образом:


```
procedure TForm1.LoadProgParam;
var
  FIniFile: TRegIniFile;
begin
  FIniFile := TRegIniFile.Create('Software');

  FIniFile.OpenKey('VR',true);
  FIniFile.OpenKey('VR-Online',true);

  Width:=FIniFile.ReadInteger('Option', 'Width', 600);
  Height:=FIniFile.ReadInteger('Option', 'Height', 400);
  Left:=FIniFile.ReadInteger('Option', 'Left', 10);
  Top:=FIniFile.ReadInteger('Option', 'Top', 10);

  WindowState:=TWindowState(FIniFile.ReadInteger('Option', 'WinState', 2));

  FIniFile.Free;
end;
```

 На компакт диске, в директории \Примеры\Глава 10\Register ты можешь увидеть пример этой программы.

10.12 Потоки

Под потоком я понимаю объект *TStream*, который является базовым объектом для потоков разных типов. В этом объекте реализованы все необходимые свойства и методы, необходимые для чтения и записи данных на различные типы носителей (память, диск, медиа носители). Благодаря этому объекту, доступ к разным типам носителей становится одинаковым.

В этой главе, когда я описывал работу с файлами, мы уже использовали потоки. Объект *TFileStream* является потомком главного объекта *TStream* и позволяет получить доступ к диску. Точно так же можно получить доступ:

- к памяти через объект *TMemoryStream*.
- к сети через объект *TWinSocketStream*.
- к СОМ интерфейсу через *TOLEStream*.
- к строкам, находящимся в динамической памяти *TStringStream*.

Это не полный список объектов потоков, но даже все эти объекты мы рассматривать не будем. Я покажу тебе только базовый объект *TStream*, а ты потом посмотри на то, как мы работали с *TFileStream* и увидишь, что всё просто. Точно так же можно будет работать с любым другим потоком, без каких либо изменений.

Итак, давай разберёмся со свойствами и методами потока:

Свойства

Position – указывает на текущую позицию курсора в потоке. Начиная с этой позиции будет происходить чтение данных.

Size – размер данных в потоке.

Методы

CopyFrom – метод предназначен для копирования из другого потока. У него два параметра – указатель на поток, из которого надо копировать и число показывающее размер данных подлежащих копированию.

Read – прочитать данные из потока, начиная с текущей позиции курсора. У этого метода два параметра – буфер, в который будет происходить чтение и число показывающее размер данных для копирования.

Seek – переместиться в новую позицию в потоке. У этого метода два параметра:

1. Число, указывающее на позицию, в которую надо перейти. Если тебе нужно передвинутся на пять байт, то просто поставь цифру 5.

2. Откуда надо двигаться. Тут возможны три варианта:

- *soFromBeginning* – двигаться на указанные количество байт от начала файла.
- *soFromCurrent* - двигаться на указанные количество байт от текущей позиции в файле к концу файла.
- *soFromEnd* – двигаться от конца файла к началу на указанное количество байт.

SetSize – установить размер потока. Здесь только один параметр – число, указывающее новый размер потока. Допустим, что тебе надо уменьшить размер файла. В этом случае, с помощью метода *SetSize* потока *TFileStream* ты можешь уменьшить или даже увеличить размер файла.

Write – записать данные в текущую позицию потока. У этого метода два параметра:

1. Переменная, содержимое которой нужно записать.
2. Число байт для записи.

Это основные методы, которые тебе могут понадобиться при работе с потоками. На практике мы ещё встретимся с подобными объектами, и ты ещё раз увидишь, как с ними работать.