

控件开发:讲述如何开发一个控件,很有价值

(五)

疯狂代码 <http://CrazyCoder.cn/> <http://CrazyCoder.cn/Delphi/Article11910.html>

To start with I used the Edit1 Control to display the results of all these variables. I then tried manipulating text in the RichEdit to see what values I got. You should do the same. Type slowly in:

```
1234567890<CR>1234567890<CR>1234567890
```

and see how the results are reflected in the Edit control as you do so. Then experiment - try adding stuff to the ends of lines, and in the beginning of the line, and middle of lines. You may have to refer back to the Code to work out which number represents which variable.

Okay, now using the variables we have, lets try selecting the text of the current line, and display it in a Edit Control (Edit2).

Add the following code to see what happens (don' t forget to add the second edit control and make it as wide as possible):

```
MyRe.SelStart := BeginSelStart;  
MyRe.SelLength := EndSelStart - BeginSelStart;  
Edit2.Text := MyRe.SelText;  
end;
```

Run the program and try it out.

OOPS - That doesn't work - the text res selected and the original cursor position is lost. We need to re SelStart and SelLength before we finish in the [OnChange] event. So let' s add at the end:

```
MyRe.SelStart := WasSelStart; //back to were we started  
MyRe.SelLength := 0; // nothing selected  
end;
```

While playing with text in the edit control I discovered something weird.

If you typed [1] then <CR> then [2] the Edit1 displayed [4-1-3-4].

But there were `_disibledevent=>`, but before the `\\"reting\\"` part, lets put some logic in to turn lines RED when they are longer than a certain length:

```
(MyRe.SelLength > 10) then MyRe.SelAttributes.Color := clRed;
```

You'll notice two things you test this out. First - it does work. Second however, is that you type a line > 10 characters, press and type `_disibledevent=>Red`. This is because it inherits the Attributes of the preceding text. Just like you have bold `_disibledevent=>`

That seems to work - except when you press in the middle of a > 10 character line you have already typed (which is already Red) to leave a stump < 10 characters `_disibledevent=>`

8. Basically it all seems to kind-of work.. can't we do some real programming now?

Okay, okay. But first we have a problem. Actually a rather big problem. The problem is PasCon. Why?

First: It s RTF code.

Problem: We can't use RTF code.

Second: its designed to work an entire stream, and then give it back to us again as a whole.

Problem: We actually want greater control over it than this `\\"all or nothing\\"` approach.

OOP to the Rescue

When you have something that works in a situation, and needs to be applied in another situation were it has to do a similar, but subtly dferent job - you have two choices: copy the function, and re-write it for the situation, or

kludge around it (e.g use Pas2Rtf, and then write a RtfCodes2RtfControl procedure).

Modern languages however give you an option: OOP it. \Object\ it. This is more than just deriving something from an existing object. It is in a sense programming in a \state of mind\. Controls should be created so they can be used in a variety of situations - rather than situation specific. In this all PasCon can deal with is tokenising the input stream and ing code RTF text. What we really need to do is divide it o two entitites. We need to separate the [Parsing/Recognise the Token and TokenType] from the [Encode it in RTF codes].

So lets start with ConvertReadStream, editing it so it looks something like this:

```
function TPasConversion.ConvertReadStream: Integer;
begin
  FOutBuffSize := size+3;
  ReAllocMem(FOutBuff, FOutBuffSize);
  FTokenState := tsUnknown;
  FComment := csNo;
  FBuffPos := 0;
  FReadBuff := Memory;

  {Write leading RTF}

  WriteToBuffer(\{ tf1ansideff0deftab720\});
  WriteFontTable;
  WriteColorTable;
  WriteToBuffer(\'deflang1033pardplainf2fs20 \');

  Result:= Read(FReadBuff^, Size);

  Result > 0 then
  begin

  FReadBuff[Result] := #0;
  Run := FReadBuff;

  while Run^ <> #0 do
  begin
```

```

Run := GetToken(Run,FTokenState,TokenStr);
ScanForRTF;
SetRTF;
WriteToBuffer(Prefix + TokenStr + PostFix);

end;

{Write ending RTF}

WriteToBuffer(#13+#10+'\par }{\'+#13+#10);

end;

Clear;

SetPoer(FOutBuff, fBuffPos-1) ;

end; { ConvertReadStream }

```

The code for ConvertReadStream is now much smaller, and also easier to understand. We can then take all the code that used to be in

ConvertReadStream that did the tokenizing and create a subroutine - the GetToken function that just does the recognizing and labelling of the individual tokens. In the process we also lose a huge number of repeated lines of code, as well as a number of sub-routines such as HandleBorCom and HandleString.

```

//
// My Get Token routine
// function TPasConversion.GetToken(Run: PChar; var aTokenState: TTokenState;
var aTokenStr: ):PChar;

```

```
begin
```

```
aTokenState := tsUnknown;  
aTokenStr := \';  
TokenPtr := Run; // Mark were we started
```

```
Case Run^ of
```

```
#13:
```

```
begin
```

```
aTokenState := tsCRLF;  
inc(Run, 2);
```

```
end;
```

```
#1..#9, #11, #12, #14..#32:
```

```
begin
```

```
while Run^ in [#1..#9, #11, #12, #14..#32] do inc(Run);  
aTokenState:= tsSpace;
```

```
end;
```

```
\'A\'..'Z\' , \'a\'..'z\' , \'_\':
```

```
begin
```

```
aTokenState:= tsIdentier;  
inc(Run);  
while Run^ in [\'A\'..'Z\' , \'a\'..'z\' , \'0\'..'9\' , \'_\'] do inc(Run);  
TokenLen:= Run - TokenPtr;  
SetString(aTokenStr, TokenPtr, TokenLen);
```



```
while Run^ in [\'!\',\'\"',\'%\',\'&\',\'(\'.\'/\'',\':\'.\'@\'',\'[\'..\'^\',  
\'\\',\'~\']] do
```

```
begin
```

```
Case Run^ of
```

```
\'\'':  
  (Run + 1)^ = \'\' then  
begin
```

```
  (aTokenState = tsUnknown) then  
begin
```

```
  while (Run^ <> #13) and (Run^ <> #0) do inc(Run);  
  FComment:= csSlashes;  
  aTokenState := tsComment;  
  ;
```

```
end
```

```
begin
```

```
aTokenState := tsSymbol;  
;
```

```
end;
```

```
end;
```

```
\'(\':  
  (Run + 1)^ = \'*\'' then  
begin
```

```
(aTokenState = tsUnknown) then  
begin
```

```
while (Run^ <> #0) and not ( (Run^ = '\') and ((Run - 1)^ = \'*\') ) do inc(Run);
```

```
FComment:= csAnsi;  
aTokenState := tsComment;
```

```
inc(Run);  
;
```

```
end
```

```
begin
```

```
aTokenState := tsSymbol;  
;
```

```
end;
```

```
end;
```

```
end;
```

```
aTokenState := tsSymbol; inc(Run);
```

```
end;
```

```
aTokenState = tsUnknown then aTokenState := tsSymbol;
```

```
end;
```

```
#39:
```

begin

aTokenState:= tsString;

FComment:= csNo;

repeat

Case Run^ of

#0, #10, #13: raise exception.Create(\Invalid \');

end;

inc(Run);

until Run^ = #39;

inc(Run);

end;

\#\':

begin

aTokenState:= tsString;

while Run^ in [\#\', \'0\'.\'9\'] do inc(Run);

end;

\\$\':

begin

FTokenState:= tsNumber;

```
while Run^ in [\'$\,\'0\..\9\', \'A\..\F\', \'a\..\f\'] do inc(Run);
```

```
end;
```

```
Run^ <> #0 then inc(Run);
```

```
end;
```

```
TokenLen := Run - TokenPtr;
```

```
SetString(aTokenStr, TokenPtr, TokenLen);
```

```
Result := Run
```

```
end; { ConvertReadStream }
```

2009-2-12 3:36:44

疯狂代码 <http://CrazyCoder.cn/>