Writing A Cross Reference Tool

by Marco Cantù

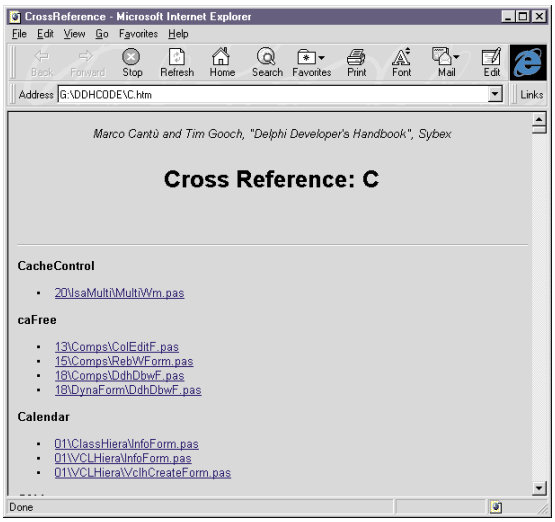
In an article published in Issue 23 of *The Delphi Magazine* (and part of my book *Delphi Developer's Handbook*) I described how to write a parser to translate the source code files of a Delphi program into HTML, maintaining the syntax-highlighting features of the Delphi editor. In that article I mentioned an additional tool I had to build for the CD-ROMs accompany ing my recent books: a source code cross-referencer. The key idea of this cross-referencer is to create an index with all the keywords, class names, method names, property names, VCL functions and every custom identifier used in the source code of a series of programs (actually hundreds of programs for my books). Of course the index should be an HTML file and the entries in the index should have links back to the HTML files with the source code. You can see an example of the output of this program in Figure 1.

In this article I want to share with you a few of the design issues involved in writing this program, discuss portions of the source code and show how the same technique can be applied to a totally different case: publishing a database on the Web.

The Main Form

The cross reference tool I've built works in three steps: it first extracts a list of files to examine to a ListBox, then extracts all the identifiers of those files, building a complex structure in memory, and finally it builds the cross-reference HTML files. Pressing a button activates each of these three steps. You can see the program's main form at design time in Figure 2. Keep in mind that this tool was built for personal use only, so the user interface is quite bare!

On the left of the form there are several components. In the edit box at the top the user can enter the directory to examine. The `FileListBox` component is used by the program but is invisible at run-time. The three buttons below are used to drive the program. Below them there is a progress bar and the list box, which is filled with the names of the source files to examine. Finally, the edit box on the bottom is used to enter the book or project description that will appear on the top of each generated HTML file and the `Save...` button is used to save the list of words to skip to a file.



➤ *Figure 1: A portion of the cross-reference from my last book, displayed in Internet Explorer*

The central portion of the form is covered by a page control, which is empty at start-up and will host 26 list boxes with the list of identifiers found for each letter. The right side of the form has two list boxes, one with the list of identifiers to exclude from the reference (the "skip list") and the other showing the list of the files where each identifier appears.

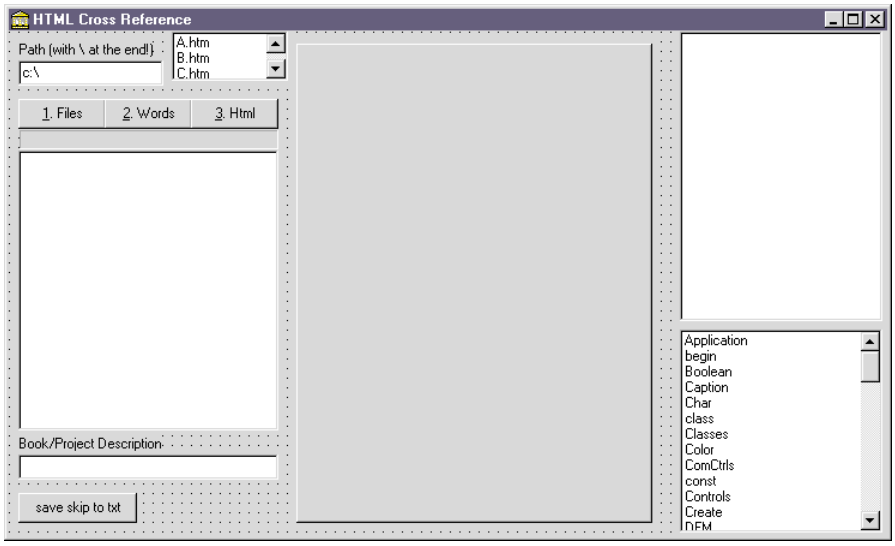
Extracting The files

The first step, extracting the files, is quite simple and can be accomplished in many different ways. The code of the first button is the following:

```
FileListbox1.Directory :=
  EditPath.Text;
ExamineDir ('*.pas');
ExamineDir ('*.dpr');
ExamineDir ('*.dpk');
Beep;
```

The `ExamineDir` method is a simple recursive method. It uses a `FileListBox` component to examine the files in the current directory which

➤ *Figure 2: The CrossRef example main form at design time*



```
procedure TFormCrossRef.ExamineDir (Mask: string);
var
  FileList: TStrings;
  I: Integer;
  CurrDir: string;
begin
  FileListBox1.Mask := Mask;
  FileListBox1.FileType := [ftNormal];
  FileList := TStringList.Create;
  try
    FileList.Assign(FileListBox1.Items);
    // for each file, add its path to the list
    for I := 0 to FileList.Count - 1 do begin
      ListBoxFiles.Items.Add(FileListbox1.Directory + '\' + FileList[I]);
    end;
    // examine sub directories
    FileListBox1.Mask := '*.*';
    FileListBox1.FileType := [ftDirectory];
    FileList.Assign(FileListBox1.Items);
    CurrDir := FileListbox1.Directory;
    for I := 2 to FileList.Count - 1 do begin
      // for each directory, re-examine
      FileListbox1.Directory := CurrDir + '\' + Copy(FileList[I], 2,
        Length(FileList [I]) - 2);
      ExamineDir(Mask);
      Application.ProcessMessages;
    end;
    FileListbox1.Directory := CurrDir;
  finally
    FileList.Free;
  end;
end;
```

➤ *Listing 1: The ExamineDir method used to locate Delphi source code files in a tree of directories*

correspond to the mask passed as a parameter, then it uses the `ftDirectory` value for the `FileType` property to get a list of the subdirectories and recursively calls itself for each of them. The effect of these calls is to add the complete path of all the found files to the `ListBox1` component. You can find the complete source code of this method in Listing 1.

Extracting The Identifiers To Memory

Once the `ListBox1` component contains a list of the Pascal files to examine, the user can press the second button and extract the identifiers from the files. To accomplish this I've used the `TNewParser` class I built for the source code parsing (and described in the references mentioned at the beginning of this article). The core of the `ButtonWordsClick` method is a `for` loop opening each of the files we've found with a `TFileStream` object, creating a parser object for this stream, and then parsing each token in the stream. The parsing code actually considers only tokens of the `toSymbol` type, that is the symbols, the identifiers of the source code. In other words, the parser automatically excludes numeric constants, comments and strings.

In the code for this loop, shown in Listing 2, you can see that the program has a little user feedback. It updates a progress bar based on the number of files (not the actual percentof the parsing complete), and selects in the `ListBox` the file it is currently processing. The reason for this extra code (even in a program with little attention paid to the user interface) is that the execution of this method can take a while. When I used this program for the *Delphi Developer's Handbook* it took about 20 minutes to complete this method: it would have taken much longer if I hadn't spent some time optimising it!

Most of my optimisation effort involved the inner code, so I'll describe this in a while, after we've seen what the program does when the type of the parser token is `toSymbol`. The first statement in this internal code is a test used to check:

- If the token is more than one character long, to exclude simple identifiers as loop counters or co-ordinates.
- If it doesn't end with a number, to exclude default names as Form1 or Button1, which are not terribly useful.
- If it isn't part of a special list called exclude list and stored in the `lbSkip` ListBox. This list includes common identifiers and keywords, such as `begin`, `if`, `Application`, `TForm`, etc.

If a token passes all these tests it is added to the memory structure holding all the found tokens. The basic idea is that, for each token, the program creates a `TStringList` object which stores the names of all the files containing the token. The name of the token and the related string list are then added to the main list, as the string and the object of the string item (as shown in Figure 3). When a new token is found, if it is part of the main list the program adds the new file name to the secondary list of the corresponding object, otherwise the program creates the new secondary list and adds the new entry to the main list.

Optimising The Code

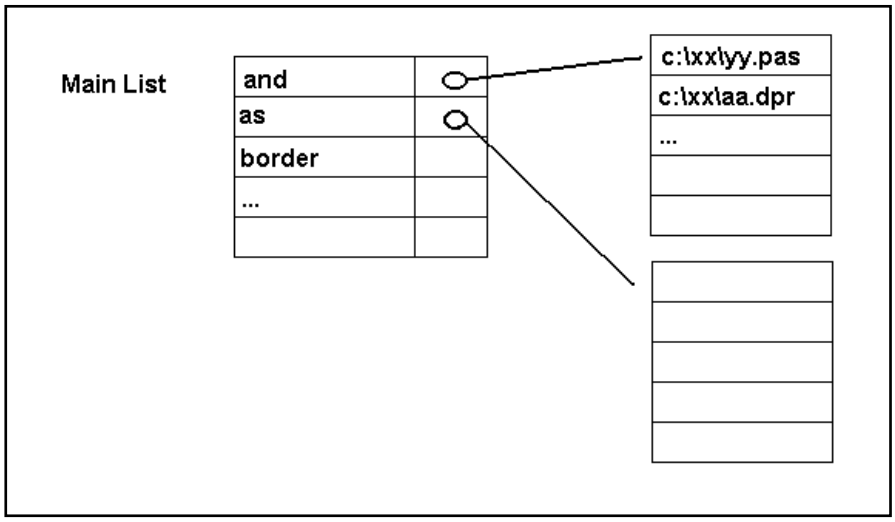
Actually the situation I've depicted in Figure 2 corresponds to the original version of the program. The current situation is a little more complex. First of all, instead of creating a single main list the program creates 26 of them, one for each letter of the alphabet. There are two reasons for doing this. Firstly, the program is going to generate one HTML file for each letter, to keep the HTML files smaller. Secondly, the program is going to be faster, since accessing the proper list is more efficient than searching a huge one. This is the code used to select the proper `ListBox` depending on the first letter of the token string:

```
LettList := LetterLists[
  Upcase(TokenStr[1])];
```

In this statement `LetterList` is an array of 26 `ListBox` components. These components are not present at design-time but they are created when the program starts. In the `OnCreate` event handler, shown in Listing 3, you can see that for each letter the program creates a new `TabSheet` inside a page control, and then places a new `ListBox` in it (as you can see in Figure 4). Finally, the program saves the list box in the `LetterList` array, to make its selection as fast as possible.

```
procedure TFormCrossRef.ButtonWordsClick(Sender: TObject);
var
  CurrFile, TokenStr: string;
  I, Item, Idx: Integer;
  FileText: TStream;
  Parse: TNewParser;
  sList: TStringList;
  LettList: TListBox;
begin
  // for each file listed
  ProgressBar1.Max := ListBoxFiles.Items.Count - 1;
  for I := 0 to ListBoxFiles.Items.Count - 1 do begin
    // select the current file, to show the progress
    ListBoxFiles.ItemIndex := I;
    // get the current file
    CurrFile := ListBoxFiles.Items [I];
    // open it as a text file
    FileText := TFileStream.Create (CurrFile, fmOpenRead);
    // pass the file to the custom parser
    Parse := TNewParser.Create (FileText);
    try
      while Parse.Token <> toEOF do begin
        case Parse.Token of
          // ignore strings, comments, symbols...
          toSymbol:
          begin
            TokenStr := Parse.TokenString;
            // more than one character
            if (Length (TokenStr) > 1) and
              // doesn't end with a number
              (TokenStr [Length (TokenStr)] > 'A') and
              // not in the skip list
              (lbSkip.Items.IndexOf (TokenStr) < 0) then
            begin
              // get the listbox for the current letter
              LettList := LetterLists[Upcase(TokenStr[1])];
              // look if the token is already in the list
              // of found tokens for the current letter
              Item := LettList.Items.IndexOf (TokenStr);
              if Item < 0 then begin
                // if not, create a new list for the files
                sList := TStringList.Create;
                sList.Sorted := True;
                sList.Add (CurrFile);
                // add the new word and the string list
                LettList.Items.AddObject (TokenStr, sList);
              end else begin
                // add the new file reference
                sList := TStringList(
                  LettList.Items.Objects[Item]);
                Idx := sList.IndexOf (CurrFile);
                if Idx < 0 then
                  sList.Add (CurrFile);
              end;
            end;
          end;
        end;
        Parse.NextToken;
        Application.ProcessMessages;
      end;
    finally
      Parse.Free;
      FileText.Free;
    end;
    ProgressBar1.Position := I;
  end;
  Beep;
end;
```

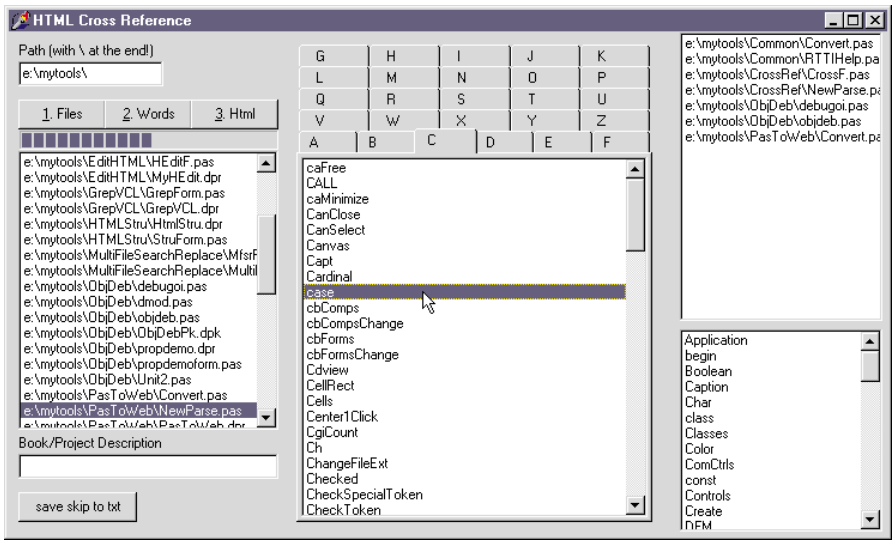
➤ *Listing 2: The code associated with the second button and used for extracting all the identifiers from the source code files*



➤ *Figure 3: Basic idea of the structure of the information in memory*

This is just one of the speed optimisations I've accomplished. The biggest impact, however, comes from sorting each of the lists, including the lists of files. Besides producing more helpful output, sorting the lists makes the `IndexOf` method faster. In fact, searching tokens in the main lists and searching for files in the secondary lists (to avoid duplicates) are the two operations which most slow down this program.

The other element you can play with is the sequence of the execution of the three tests of the `if` statement described above. I've tried to make the fastest test short, to avoid executing the others (thanks to the short-circuit evaluation of Boolean expressions). There are probably further optimisations you can perform, including using custom memory structures and search techniques, instead of using string lists, but this requires a lot of work. When I first built this program for *Mastering Delphi 3* it took many hours to execute. Simply applying some common sense techniques, as those just



➤ *Figure 4: The output of the program when the information has been stored in memory. Notice the page control with a tab sheet for each letter of the alphabet.*

```
procedure TFormCrossRef.FormCreate(Sender: TObject);
var
  Letter: Char;
  List : TListBox;
  Sheet: TTabSheet;
begin
  // create 26 list boxes, and connects them...
  for Letter := 'A' to 'Z' do begin
    Sheet := TTabSheet.Create (self);
    Sheet.PageControl := PageControl1;
    Sheet.Caption := Letter;
    List := TListBox.Create (self);
    List.Parent := Sheet;
    List.Align := alClient;
    List.Sorted := True;
    List.OnClick := LbWordsClick;
    List.OnDblClick := LbWordsDblClick;
    LetterLists [Letter] := List;
  end;
end;
```

➤ *Listing 3: When the program starts, it adds 26 sheets to a page control and a list box to each page.*

described, I reduced the time to about one tenth, which was a lot more reasonable!

As a further optimisation, consider that this program requires a lot of memory (several Mb for a large collection of files), so closing all the other applications while it runs can help speed it up.

Looking At The Data In Memory

During the generation of the lists of identifiers, and when this operation is completed, there are a few things you can do. You can click on the items of each of the list boxes with the identifier to see the list of the files in which this identifier is used (as shown in Figure 4):

```
procedure TFormCrossRef.LbWordsClick(
  Sender: TObject);
begin
  with (Sender as TListBox) do
    lbList.Items := TStringList(
      Items.Objects [ItemIndex]);
end;
```

This is helpful as a sort of preview, but also to find out if there are any identifiers which are too common, and should be removed from the generated code and added to the skip list (particularly if the generation of the words is still active). At the end you might want to save the new skip list to a file and then modify the program accordingly (the words to skip are hard coded into a list box, in fact, simply because after a little tuning they are not supposed to change).

The HTML Generation

The third button on the main form finally does the HTML generation. Contrary to what you might think this is actually the simplest part of the program, since we have already collected all the information we need. The `ButtonHtmlClick` method (see Listing 4) has an outer `for` loop, used to examine each letter:

```
for Letter := 'A' to 'Z' do
```

Inside the loop the program creates a string (called `HTML`) and fills it with the information. After adding a header and a title, the code has a `for` loop which outputs each identifier (or token) from the main list of the specific letter, and an internal `for` loop which lists the files where each identifier appears.

These file references are added as anchors, or links, and the code used for this operation probably requires some explanation. The string, added to the output for each file, has this structure:

```
'<LI><A HREF="' + reference +
  '.htm ">' + description +
  '</A>'#13#10);
```

The description of the file is simply its name, which is extracted from

➤ *Listing 4: The code use to generate the HTML cross-reference using the information already collected in memory*

```
procedure TFormCrossRef.ButtonHtmlClick(Sender: TObject);
var
  Dest: TStream;
  HTML, OutFileName: string;
  I, J: Integer;
  Letter: Char;
  sList: TStringList;
begin
  FileListBox1.Mask := '*.dpr';
  FileListBox1.FileType := [ftNormal];
  SetLength (HTML, 10000);
  // for each letter
  for Letter := 'A' to 'Z' do begin
    // select the tab sheet we are working on
    PageControl1.ActivePage :=
      LetterLists [Letter].Parent as TTabSheet;
    HTML := '';
    // add head
    HTML :=
      '<HTML><HEAD>' + #13#10 +
        '<TITLE>CrossReference</TITLE>' + #13#10 +
        '</HEAD>'#13#10 +
        '<BODY>'#13#10 +
        '<CENTER><I>' + EditBookDescription.Text +
        '</I></CENTER></H3><BR><BR>'#13#10 +
        '<H1><CENTER>Cross Reference: ' +
        Letter + '</CENTER></H1><BR><BR><HR>'#13#10;
    // for each identifier starting with the letter
    for I := 0 to LetterLists [Letter].Items.Count - 1 do
    begin
      Application.ProcessMessages;
      // add the word
      AppendStr (HTML, '<H4>' + LetterLists [Letter].
        Items[I] + '</H4>'#13#10);
      // sub-list
      AppendStr (HTML, '<UL>'#13#10);
      sList := TStringList (
        LetterLists [Letter].Items.Objects [I]);
      // add file names
      for J := 0 to sList.Count - 1 do
        AppendStr (HTML, '<LI><A HREF="' +
          ChangeFileExt (DosPathToUnixPath (
            Copy(sList[J], Length(EditPath.Text) + 1,
              1000)), '_' + Copy (ExtractFileExt(sList[J]),
              2, 3)) + '.htm ">' + Copy (sList[J],
              Length(EditPath.Text) + 1, 1000) +
              '</A>'#13#10);
      AppendStr (HTML, '</UL>'#13#10);
    end;
    // add tail
    AppendStr (HTML,
      '<BR><I><CENTER>' + 'File generated by CrossRef, '+
      'a tool by Marco Cant&ugrave;' +
      '</CENTER></I>'#13#10 + '</BODY> </HTML>');
    // create the output file
    OutFileName := EditPath.Text + Letter + '.htm';
    Dest := TFileStream.Create (OutFileName,
      fmCreate or fmOpenWrite or fmShareDenyNone);
    try
      Dest.WriteBuffer (Pointer(HTML)^, Length (HTML));
    finally
      Dest.Free;
    end;
  end; // for Letter
  Beep; // done
end;
```

the proper list (`sList[J]`). However, instead of using the full pathname the program shows the relative path from the base directory (where the file search started and where the generated HTML files will be saved). This is important because the files might be moved to a different drive or directory, maybe even a CD-ROM. Here is the code used to compute the relative path (that is, remove a number of characters corresponding to the path of the root directories from the complete file path):

```
Copy(sList[J], Length(
  EditPath.Text) + 1, 1000);
```

Building the reference is more complex. First of all, we need to use the relative path again. Then this path must be converted to a Unix style path (which uses the / character instead of the \ character), otherwise the HTML files will not work properly with Netscape Navigator and other browsers.

Finally, the actual HTML files use a special naming convention: `foo.pas` becomes `foo_pas.htm`. To make this change I use the `ChangeFileExt` function, using as second parameter the underscore plus the original extension, then I append the `.htm` string:

```
ChangeFileExt (FileName, '_' +
  Copy(ExtractFileExt(
  FileName), 2, 3)) + '.htm';
```

Now that you know the rationale of most of my choices, the source code of the `ButtonHtmlClick` method (see Listing 4) should become readable.

Once the program has created the HTML file inside a string, it saves this string to a physical file with the single low-level routine `WriteBuffer`. There are certainly higher-level techniques to save a string to a file, but using a stream and saving the data of the string directly from its memory area (that is, the address the string refers to) I think is neat.

Temporary Conclusion

This completes the description of the cross-referencing tool I've written for indexing Delphi source code files. On the companion disk you will find the complete source code of the example, ready to be compiled and used. Of course there are plenty of improvements you can make, both to the user interface (which was never meant to be used by others) and to the capabilities of the tool itself. I'm working on integrating this cross-referencer, the PasToWeb HTML generator and some extra tools into a single powerful wizard, but I'm not sure when this tool will be completed. Check my web site for updates.

Cross Referencing A Database

After I wrote this program, I realised that a similar technique could be used for cross-referencing a complex database while publishing it on a web site. Consider the tables from the `DBDEMOS` example included with Delphi: using the `customers`, `orders` and `items` tables you can build a complex structure of HTML files and publish it on the Web. The main HTML file is a table with information about each customer (see the first step of Figure 5). Each customer is linked to an HTML file with a table listing the orders (the second step of Figure 5) and each order with a further HTML file including the details (the items) of the order itself (the third step of Figure 5).

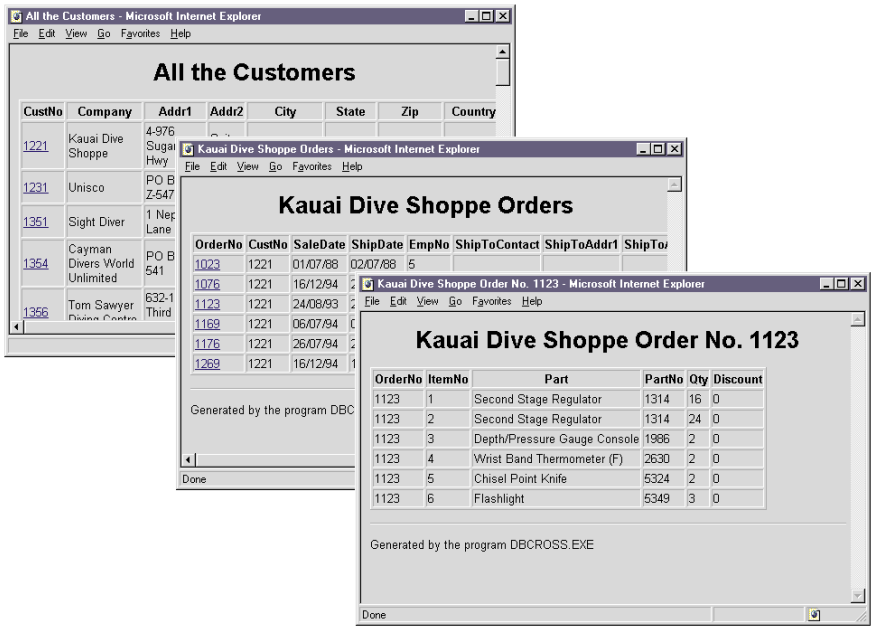
This information is extracted from three tables which have a master-detail relationship. A fourth table, `Parts`, is used to add a lookup field to the `Items` table and provide a textual description for the ordered items.

Generating The Tables

Generating the HTML files for this structure is not too difficult. Most of the HTML files generated by the DbCross program are based on HTML tables, showing an entire database table or a portion of it (in the case of detail tables). The HTML code generation is quite simple. I've inherited two classes from the `TStringList` class. The first is `THtmlStrings` and includes code for generating simple headers and footers. The second, `THtmlData`, is a further subclass of `THtmlStrings` and adds the generation of the HTML table header.

Besides this, the `THtmlData` class includes the `AddTableRow` method you can see in Listing 5. In this method, `Data` is a local field assigned to an existing data set in the constructor and `LinkStr` is a parameter indicating the initial letters of the file we should connect

➤ *Figure 5: The sequence of HTML pages you can open while browsing the published database data in the normal sequence, from the list of customers, to its orders, to the details of the order*



to the first column of the table. To make this work, of course, the first field of the table should be the one providing the link to the further details.

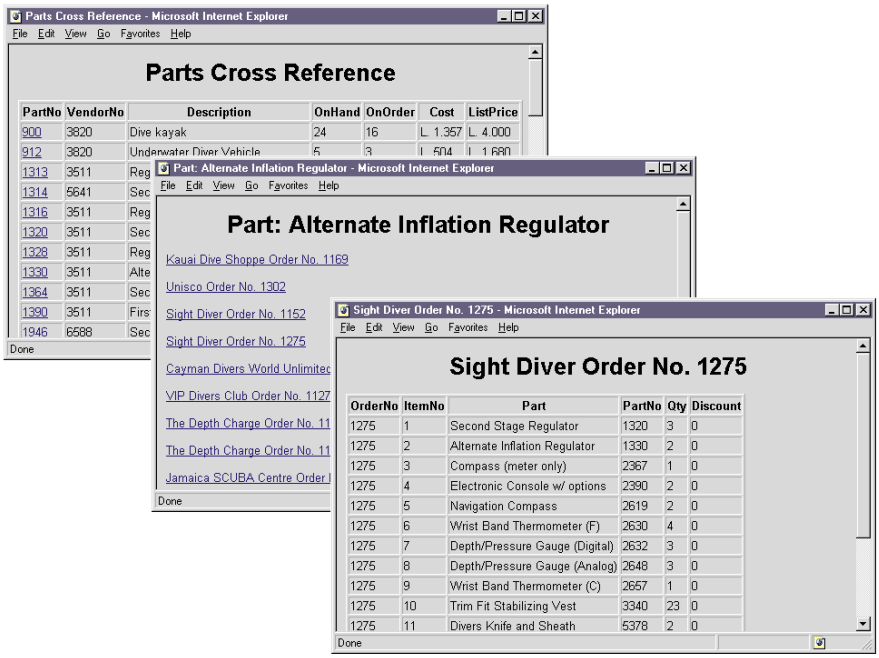
The rest of the code used to generate the base files is quite simple, so I'll leave you to study it (the code for DbCross is on the companion disk of course).

Generating The Cross-Reference

What is interesting is that the DbCross program has a second capability. While parsing the three levels of its master-detail structure, it can collect information from the `Items` table.

The aim is to be able to build, for each entry of the `Parts` table, a list of the orders where the part appears. Each of these lists is built in memory, while processing the tables, and then saved to a separate file. The code for the memory parsing is very similar to the source code cross-reference, although a little simpler.

The program keeps track of the various parts listed in each order by adding each unique part to the `ListOfLists` string list. The object portion of the items in this list is a further string list (of type `THtmlStrings`) which stores information on the orders where the particular part appears. However, instead of keeping in memory only the number of the order, the program already builds the HTML anchors for the link, providing a more detailed description (as you can see for yourself in the central step in Figure 6).



➤ *Figure 6: The sequence of HTML pages you can open while browsing the published database data using the cross reference, from the list of parts, to the orders, to the details of each order*

Listing 6 shows a portion of code inside the triple `while` loop scanning the master table, the first detail table and the third detail table. As you can see, if a part is already in the list the program extracts its object portion and converts it to the `THtmlStrings` type, to add the new element to it. Otherwise the program creates a brand new string list, then adds the same information to it.

At the end, the program also parses the `Parts` table in order to generate the main file for the cross-reference. In Figure 6 you can see the navigation of the cross reference starting with the list of parts, accessing the actual cross reference and finally looking at the details.

Publishing A Database On The Web

This program generates all the HTML files from a single button click. The operation takes about 20 seconds on my (slow) computer, and creates 321 files totalling 371Kb. Put these files on your web site and you are in the diving equipment business! Jokes aside, if you want to publish a database on the web, you can use a similar technique if the following conditions apply.

Firstly, *the data doesn't change very often*. A catalogue updated monthly or weekly is a good example. Even if you can update the site automatically every night this is still a possible technique. For real time information this is certainly not a good approach!

Secondly, *the amount of data is limited*, smaller than your available space on the Web site of course. This seems obvious, but the formatted HTML output might take much more space than the original database files. If you use CGI or ISAPI to generate the HTML from the database data on the fly you might need less disk space.

➤ *Listing 5: The method of the THtmlData string list subclass used to generate the lines of the HTML tables extracting database data*

```
procedure THtmlData.AddTableRow (LinkStr: string);
var
  I: Integer;
begin
  // new row, with table data (tag <td>)
  Add('<tr>');
  if LinkStr <> '' then
    Add('<td><a href="' + linkStr + Data.Fields[0].DisplayText + '.htm">' +
        Data.Fields[0].DisplayText + '</a></td>')
  else
    Add('<td>' + Data.Fields[0].DisplayText + '</td>');
  for I := 1 to Data.FieldCount - 1 do
    if Data.Fields[I].Visible then
      if Data.Fields[I].DisplayText <> '' then
        Add('<td>' + Data.Fields[I].DisplayText + '</td>')
      else
        Add('<td><p></td>');
  Add('</tr>');
end;
```

Lastly, *the number of ways to navigate is limited*. If there are three or four obvious paths of navigation (a main one and a two or three cross-references) you can generate all of them statically. Otherwise, the cross-referencing HTML files will be way larger than the files with the actual data and the time required to generate them might be too much.

Even if only parts of these conditions apply to your specific needs, you can consider using a mixed approach. You can have a portion of the data and some navigational files generated periodically, and have a CGI and ISAPI application on the site as well, to let user do free searches and follow other less frequent paths.

Conclusion

In this article I've applied the same cross-referencing technique to two totally different examples. The first was a source code cross-reference I used for the CD-ROMs accompanying my recent books, the second was an example of publishing database tables on the web with multiple navigation paths. There are probably many other cases where this technique (or a variation of it) might come in handy. If you bump into a good idea or application, let me know.

Marco Cantù is the author of *Mastering Delphi 3* and *Delphi Developer's Handbook*, does advanced Delphi training world wide, enjoys speaking at conferences, and can be reached on his web site (www.marcocantu.com) and by email (marco@marcocantu.com). *Also check out also his newsgroup (there's link on his website).*

➤ *Listing 6: Code used to build the database cross-reference in memory*

```
// at the beginning
ListOfLists := TStringList.Create;
// for each item of each order of each customer...
// search the part in the cross reference
Index := ListOfLists.IndexOf(TableItemsPartNo.AsString);
// if not found create a new entry
if Index < 0 then begin
  HtmlMem := THtmlStrings.Create;
  HtmlMem.AddHeader ('Part: ' +
    TableItemsPart.AsString);
  Index := ListOfLists.AddObject (
    TableItemsPartNo.AsString, HtmlMem);
end;
// add the reference to the list
THtmlStrings(ListOfLists.Objects[Index]).Add('<a href="Ord' +
  TableItemsOrderNo.AsString + '.htm">' + TableCustomersCompany.AsString +
  ' Order No. ' + TableOrders.FieldByName('OrderNo').AsString + '</a><p>');
```