File Search And Delete Tree

by David Selwood

Recently, I needed two standard procedures: one to delete a directory and its sub-directories and a second to search for specific files in a directory and its sub-directories. Scanning through the Delphi manuals and Help I was unable to find any procedures or components that met these criteria. I reached the same conclusion from my Delphi books, apart from one, where the algorithm left much to be desired! So I set out to build and test them myself.

Deleting A Directory Tree

A directory can contain files and other directories (which may also contain files and directories) and this can iterate to any number of directories and files. Because all this is unknown in advance, the problem lends itself to recursive programming (routines that call themselves).

I have implemented three standard procedures (since no return results are required). If a directory or file cannot be deleted then an exception will be raised. The code is in Listing 1.

`Delete_Tree` simply sets the ball rolling and is called with the directory path of the directory you want to delete. It appends a backslash (\) to the directory passed, then calls `Delete_Directories_Files`, passing the directory path along.

`Delete_Directories_Files` is the recursive section. It uses `FindFirst` and `FindNext` to search the directory passed for any files or directories. If a file or directory is found then `Found_File_Or_Directory` is called, which will delete the file or directory found. When no more files or directories can be found in the directory passed, the directory is deleted. If the directory cannot be deleted the error code is checked. Error code 16 means that the directory is in use, so we change directory to the parent of the current directory. We can then delete the target directory without error. If the error code is not 16 the exception is re-raised.

`Found_File_Or_Directory` tests the found file's attribute to see if it is a file or a directory. If it is a file then it is deleted. If the file cannot be deleted an exception is created and raised so that the calling program is made aware of the problem. Now this is the clever recursive part: if a directory is found then we simply call `Delete_Directories_Files` with the path of the sub-directory that needs to be deleted. So the process calls itself.

Searching For Files

The Delphi `FindFirst` and `FindNext` routines only search for files in the specified directory. We need a routine that recursively searches directories, returning a list of files found that match the search criteria, including directory paths.

We could return the list of files found by declaring a variable of type `TStringList` in the calling application, then passing this variable to the file search routine, to be filled with pathnames. The calling application would be responsible for freeing the memory used by the `TStringList` variable.

Whilst this would work, it's a bit messy and it would, I decided, be

➤ *Listing 1*

```
unit Deldirs;
interface
uses SysUtils,Controls,Forms,Dialogs;
procedure Delete_Tree(Dir_Path: string);
procedure Delete_Directories_Files(const Dir_Path: string);
procedure Found_File_Or_Directory(const SearchRec:
  TSearchRec; const Dir_Path: string);
implementation
{$I+}  {make all run-time I/O errors into exceptions}
procedure Delete_Tree(Dir_Path: string);
var Old_Screen_Cursor: TCursor;
begin
  Old_Screen_Cursor:= Screen.Cursor;
  Screen.Cursor:= crHourGlass;
  Application.ProcessMessages;
  try
    if Copy(Dir_Path, Length(Dir_Path),1) <> '\' then
      Dir_Path:= Dir_Path + '\';
    Delete_Directories_Files(Dir_Path);
  finally
    Screen.Cursor:= Old_Screen_Cursor;
  end;
end; { Delete_Dirs}
procedure Delete_Directories_Files(const Dir_Path: string);
var
  SearchRec: TSearchRec;
  Directory_FileMask: string;
  DelDir: string;
  NewDir: string;
begin
  Directory_FileMask:= Dir_Path + '*.*';
  {Search for files & directories in current directory}
  if FindFirst(Directory_FileMask, faAnyFile, SearchRec) = 0
  then begin
    Found_File_Or_Directory(SearchRec, Dir_Path );
    while(FindNext (SearchRec)=0) do
      Found_File_Or_Directory ( SearchRec, Dir_Path );
  end;
  FindClose (SearchRec);
  {Now remove the current directory,
   must remove backslash at end of Dir_Path}
  DelDir := LowerCase(Copy(Dir_Path, 1, Length(Dir_Path)-1));
  try
    RmDir(DelDir);
  except
    on E: EInOutError do
      {I/O Error 16: directory in use, so change directory}
      if E.ErrorCode = 16 then begin
        {Get parent directory}
        NewDir:= ExtractFilePath(DelDir);
        {Don't want the last backslash}
        NewDir:= Copy(NewDir,1,Length(NewDir)-1);
        ChDir(NewDir);
        RmDir(DelDir);
      end else
        raise;
  end; { except }
end; { Delete_Directories_Files }
procedure Found_File_Or_Directory(const SearchRec:
  TSearchRec; const Dir_Path: string);
begin
  {Make sure its not a pointer to a parent directory}
  if((SearchRec.Name<>'.') and (SearchRec.Name<>'..')) then
    {See if file}
    if (SearchRec.Attr and faDirectory = 0) then begin
      if not DeleteFile(Dir_Path + SearchRec.Name) then
        raise Exception.Create(
          'Can''t delete file: Dir_Path + SearchRec.Name');
    end else
      {Must be a directory, delete any files or directories}
      Delete_Directories_Files(Dir_Path+SearchRec.Name+'\');
end; {Found_File_Or_Directory}
end.
```

much better to have a TFileList object to encapsulate the whole business of searching for and storing a list of files. Listing 2 shows the code. TFileList has two public methods, Create and Destroy, and a public variable Files_Found of type TStringList: it is these that provide the interface. The Create method is passed the directory to start the search with, which should also include the search mask, and a boolean flag, which if True indicates that sub-directories should be searched too. Listing 3 shows an example of the class in use.

Search_Directory is the key method. It first creates a temporary sorted list of all the files in the directory passed to it that match the file mask criteria. This list of sorted files is then appended to the *unsorted* Files_Found list. We don't sort it as, for example, the following would not be sorted correctly:

```
C:\DAVID\AAA.INI
C:\DAVID\SELWOOD\FILE.INI
C:\DAVID\ZZZ.INI
```

because a sorted list sorts by ascending character position and does not take the strings' length into account.

When all the files have been checked in the current search directory, a flag is tested to see if sub-directories also need to be searched. If so, a temporary sorted list of all the directories in the current directory is created. This list is then iterated through.

For each directory identified, we call the Search_Directory procedure, so that the newly identified directory is also searched, and so on to the bottom of the directory tree: this is the recursive part of the algorithm.

Conclusion

This simple, but useful, example shows how using recursive techniques can accomplish a lot of work in surprisingly few lines of code and also the benefits of using object orientation.

David Selwood holds a degree in Computing and has recently moved from the world of Unix M-Technology (working in recursive techniques and artificial intelligence) to Delphi. He works as an analyst/programmer for BURNS Open Systems, who specialise in EDI.

➤ *Listing 3*

```
procedure Test_FileList;
var
  FileList: TFileList;
begin
  {Build a list of *.ini files in the Windows directory and its sub-directories}
  FileList:= TFileList.Create('C:\Windows\*.ini',True);
  try
    {Copy the list of files found to a ListBox}
    ListBox1.Items.Assign(FileList.Files_Found);
  finally
    {Finished with FileList object so free it}
    FileList.Free;
  end;
end;
```

➤ *Listing 2*

```
unit Filelist;
interface
uses Classes,SysUtils,Controls,Forms;
type
  TFileList = Class(TObject)
    private
      File_Mask: string;
      Sub_Directories: boolean;
      procedure Search_Directory(
        const Directory_FileMask: string);
      procedure Found_File_Dir(const SearchRec: TSearchRec;
        FileDir_List: TStringList);
    public
      Files_Found: TStringList;
      constructor Create(const Directory_FileMask: string;
        const Sub_Dirs: boolean);
      destructor Destroy; override;
  end; { TFileList }
implementation
{$I+}   {make all run-time I/O errors into exceptions}
constructor TFileList.Create(const Directory_FileMask:
  string; const Sub_Dirs: boolean);
var Old_Screen_Cursor: TCursor;
begin
  inherited Create;  {Execute the ancestor constructor}
  Old_Screen_Cursor:= Screen.Cursor;
  Screen.Cursor:=     crHourGlass;
  Application.ProcessMessages;
  try
    Files_Found:= TStringList.Create;
    Files_Found.Sorted:=     False;
    File_Mask:= ExtractFileName(Directory_FileMask);
    Sub_Directories:= Sub_Dirs;
    Search_Directory (Directory_FileMask);
  finally
    Screen.Cursor:= Old_Screen_Cursor;
  end;
end; {Create}
destructor TFileList.Destroy;
begin
  Files_Found.Free;
  inherited Destroy;   {Execute the ancestor constructor}
end; {Destroy}
procedure TFileList.Search_Directory(
  const Directory_FileMask: string);
var
  SearchRec : TSearchRec;
  Directory : string;
  FilesDirsFound :TStringList;
  c1 : integer;
begin
  Directory:= ExtractFilePath(Directory_FileMask);
  FilesDirsFound:= TStringList.Create;
  FilesDirsFound.Sorted:= True;
  FilesDirsFound.Duplicates:= dupError;
  try
    {Build list of files but not directories in current diry}
    if FindFirst(Directory_FileMask,
      faAnyFile-faDirectory, SearchRec) = 0 then begin
      Found_File_Dir(SearchRec,FilesDirsFound);
      while (FindNext(SearchRec) = 0) do
        Found_File_Dir(SearchRec,FilesDirsFound);
    end;
  FindClose(SearchRec);
  {Now copy list of sorted files found onto
   public list of unsorted files}
  for c1:= 0 to FilesDirsFound.Count-1 do
    Files_Found.Add(Directory + FilesDirsFound[c1]);
  {Now start checking the sub-directories}
  if Sub_Directories then begin
    FilesDirsFound.Clear;
    if FindFirst(Directory + '*.*',
      faDirectory, SearchRec) = 0 then begin
      Found_File_Dir(SearchRec,FilesDirsFound);
      while (FindNext(SearchRec) = 0) do
        Found_File_Dir(SearchRec,FilesDirsFound);
    end;
    FindClose(SearchRec);
    for c1:= 0 to FilesDirsFound.Count-1 do
      Search_Directory(
         Directory + FilesDirsFound[c1] + '\' + File_Mask);
  end; {if Sub_Directories}
  finally
    FilesDirsFound.Free;
  end;
end; {Search_Directory}
procedure TFileList. Found_File_Dir(const SearchRec:
  TSearchRec; FileDir_List: TStringList);
begin
  if ((SearchRec.Name<>'.') and (SearchRec.Name<>'..')) then
    FileDir_List.Add(SearchRec.Name)
end; {Found_File}
end.
```