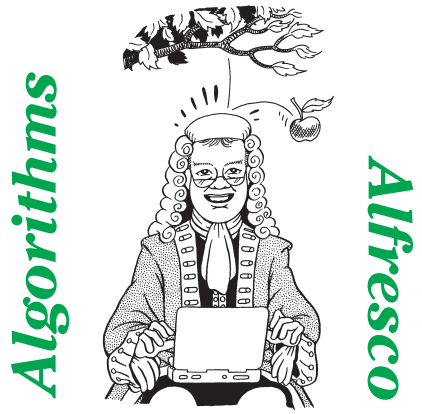
Manage The Damage

This month we chop and squeeze several files into one



by Julian Bucknall

Part of Chapter 7 in my new book, *Tomes of Delphi, Algorithms and Data Structures* (available in May), is an extensive discussion of extendible hash tables, something I briefly touched on in my article on hash tables in *The Delphi Magazine* in March 1998. The extendible hash table I introduce in the book uses three files: a directory, a file of hash buckets, and a file of data records. Whilst I was writing the code, I kept on wishing that I had a way of making them all merge into one file: three files seemed excessive somehow.

However, it wasn't as simple as it sounds. The file of data records was exactly that: an array of records contiguous on disk. The file of hash buckets could be viewed as a file of records, with each hash bucket being fixed in size, but of course they were a different size to the data records. Finally, the directory (a directory of hash buckets) was a stream of long integers, varying in number. Each of my three files had components that were completely different to each other. Even worse, each component had varying lifecycles: records could be added and deleted, hash buckets would split (and possibly merge) and the directory was the most volatile of all.

Given my final deadline for the book (the code for the extendible hash table only came together on the final day!), there was no way I could solve the riddle, so I just left it alone.

Too Much Of A Good Thing

Since then, I've been mulling it over some more. The more I thought about it the more it became clear that what I wanted was something like *OLE structured storage*, as Microsoft terms it. Structured storage is essentially a file system within a single file, sporting a COM interface. By file system, I mean that with structured storage you can create folders and put files within those folders. The whole lot, folders and files, is stored within a single file on disk. It's as if the file were a disk. (Although Microsoft calls the data structure *structured storage*, it is also sometimes known as a *compound file*, and that's how we'll be referring to it later on.)

Microsoft uses OLE structured storage for storing the various parts of a Word document, and of an Excel spreadsheet. Without it they'd have to have several different files describing a complex document (each of which could become lost); with it a document is just a single file, albeit with a complex internal structure. In my view structured storage is made a more complicated subject than it really is; Microsoft decided to call the folders *storages*, and the files *streams*. I can just about live with the word stream, but calling a folder a storage is just plain obscure (even Word's grammar checker objects to the word pair 'a storage').

The COM object API for structured storage has all the usual suspects: creating or opening a new structured storage file, creating or opening a stream, reading from or writing to a stream, creating or opening a storage (or folder). You can also iterate through the streams in a storage much as you would do a directory listing for a folder on your disk. The structure storage file comes with a root directory already created.

Something To Live For

This article, however, is not going to be about structured storage. If you want to know about it, Eric Harmon's book *Delphi COM Programming* describes it all, although a little disappointingly he doesn't provide a class encapsulation of it. Instead, as befitting a column on algorithms, we'll discuss how to build our own. We'll be designing and implementing, fanfare please, the *Algorithms Alfresco* compound file. Well, at least, the first simple iteration of it.

The first thing we have to decide, before delving into the low-level implementation, is what we want to be able to do with a compound file. My initial list was: create an empty compound file; open one; add or remove folders within the compound file (there would have to be a pre-defined root directory, obviously); create, open, read from and write to, close and delete subfiles (I'll be referring to the internal files within the compound file as *subfiles*). We should also be able to list the subfiles and folders for a given folder (akin to `FindFirst`/`FindNext`/`FindClose`).

Finally, there should also be a way of defining a *current directory*, although, in my view, that's possibly a little too restrictive. Better would be a way of opening a folder, returning a handle, and then using that handle to access the subfiles within the folder. That way, you could have as many 'current directories' as you wanted.

Although it would be nice to open and manipulates subfiles in the time-honored manner of calling separate routines, one to open a subfile, one to read a block from a subfile, etc, a simpler way would be to encapsulate them as streams, that is, class descendants of `TStream`. A stream would encapsulate accessing a subfile quite nicely. The compound file should of course be implemented as a class.

Notice that, so far, I'm ignoring the entire problem of how to implement at a low level such a beast as a compound file. We'll get to that in a moment. For now, let's concentrate on how we would want to use a compound file if we had one. Although it would be pretty amazing if we could implement an entire, efficient, error-resistant, shareable file system inside a single file within a single article, in reality we should take it slowly in simple steps. Later on, once we've written a basic compound file, we can go back and add extra functionality, consider writing a Norton Utilities for it, allow it to be shared across several processes, or whatever. So, we won't worry too much about the advanced features, and instead just consider writing a basic compound file.

I envisage the primary use of a compound file would be for programs that have a multiplicity of small configuration or other types of files. Such a program is Delphi itself: there's a bundle of files that Delphi maintains for its own use: a DSK file, a DMT file, a DRO file, etc. Another example is Netscape: for each user it maintains a dozen files. Of course, if we wanted to write a complex document type format, we could radically simplify it by storing disparate structures in different streams in different subfiles in a compound file: my original requirement, in fact.

Right, so now we have the basic functionality nailed down. Time to worry about how to design and code it.

Smalltown Boy

When I started to think about this problem in earnest, my immediate thought was that since we are to mimic a file system, let's copy some ideas from a real one. The one I chose was the only one I knew about (from way back when): the FAT system as used in DOS. It's small and simple enough to grasp in a few paragraphs.

How does this work? Well the first thing to realize is that in the DOS FAT system the disk is divided up into equal sized chunks called *clusters*. A cluster is the smallest unit of disk space we can use; they are numbered from 0 up to whatever number the last cluster on the disk will have. So, the first thing we should do is to set up our compound file as a file of equal sized blocks. The size doesn't really matter, but, since we'll assume that the subfiles we'll be storing will be smaller rather than larger, we'll choose a smallish block size of 512 bytes (`$200` bytes in hex). This is beneficial from another viewpoint: the average amount of space lost per subfile will only be 256 bytes, whereas if we used 4Kb blocks instead, say, the average wasted space per subfile would be 2Kb. We don't particularly want to go smaller, since reading or writing all the blocks for a subfile will be much slower. So, 512 byte blocks it is.

Another thing to realize is that when we format a disk we know exactly how big the disk is and therefore we know the total count of clusters on the disk. For a compound file this is a little awkward: we don't want to pre-allocate a massive amount of disk space for the compound file. What we'd really like is for it to grow when necessary and as needed. Mark that up as a possible problem.

Back to the DOS FAT system. The first cluster, cluster 0, contains various important pieces of information about the structure of the disk. We should do the same: the first block of the compound file should contain data about the compound file as a whole, for example a signature so that we can verify that a file is really a compound file, the block size, and so on.

The next few clusters of the DOS FAT system are for the FAT (generally there are two FATs; however, we'll ignore this duplication for our compound file). The FAT is the *file allocation table*. It's an array of cluster numbers, indexed by cluster number. For a disk, the FAT is fixed in size since we know the size of the disk and hence how many clusters it contains. To explain how to use the FAT it's best to use some examples.

Suppose we have a file that is spread over several clusters on a DOS FAT disk and, through a mechanism we haven't discussed yet, suppose we know the starting cluster number for the file. How do we find all the clusters in the file? We go to the FAT. The entry in the FAT for the starting cluster number is the number of the second cluster for the file. The entry in the FAT for the second cluster number is the number of the third cluster for the file, and so on. Eventually the FAT entry will read -2 (an impossible cluster number) and that signifies that the current cluster is the final one.

Suppose we want to create a file and write data to this file. We shall have to write the data to one or more unused clusters. How do we know which clusters are unused? For the FAT, another special value comes into play, -1. When we format the disk, the format program fills the FAT with -1 values to signify that all clusters are unused. When we write data to a file and need an unused cluster, we look through the FAT until we reach a -1 value: the index of it is the number of the cluster we can write to.

So, in our case, we're left with a couple of problems. We don't particularly want to create a compound file to be a fixed size (imagine if all Word documents were 1Mb in size, for example). So we'd like it to grow automatically as we stuff more and more subfiles into it. That's easy enough for a file to do, but what about our FAT look-a-like? We can't have a fixed size for this either: the FAT must grow as needed. We can, however, reserve block 1 (the second block) to be the first block of the FAT (block 0 is the header, remember). Subsequent blocks in the FAT will be found by using the FAT we've read so far. (If this gives you a headache, you have my permission to get an aspirin!) The entries in the FAT could be `longint`s, but for the intended purpose of the FAT (storage of several small files) I think we can get away with `word`-sized entries. That would limit the compound file to a maximum of 32Mb if we used 512 byte blocks: a reasonable limitation in my view.

Moving right along now: after the file allocation table, the DOS FAT system stores the first cluster of the root directory. This is the start of the actual file system, where the file and folder names are stored, where we find out which cluster a file's data starts at, and so on. The root directory (as is any directory or folder) consists of an array of directory entries, records that define a single file or folder. Each directory entry contains the name of the entity (be it file or folder), its type (is it a file, or a folder, or is it unused?), its size (the size of a file is obvious, the size of a folder is the sum of the size of its directory entries), its timestamp, its attributes (read-only, hidden, system?) and the number of its starting cluster.

And for our compound file? Pretty much the same, to be honest. We can be a little clever, though, and not limit the size of file or folder names by making the folder information streamed and not a simple array.

Don't Know What To Do

Now we've decided on the format we can start writing the compound file class. We'll take it step by step. First thing to notice is that, since everything depends on being able to read the FAT and follow block chains in there, we should read the entire FAT into memory when we

➤ *Listing 1: Create and Destroy for the compound file class.*

```
constructor TaaCompoundFile.Create(const aFileName : string;
  aMode : word);
begin
  {create the ancestor}
  inherited Create;
  {open the file stream}
  FStream := TFileStream.Create(aFileName, aMode);
  {create the in-memory FAT}
  FFAT := TaaIntList.Create;
  FFATBlocks := TaaIntList.Create;
  {allocate the header}
  GetMem(FHeader, CFBlockSize);
  {allocate the list of open folders}
  FOpenFolders := TList.Create;
  {if the stream is new (size is zero)
    write the header record}
  if (FStream.Size = 0) then
    cfPrepare
  {otherwise read the header and make sure that it's one of
    our files}
  else
    cfReadHeader;
end;

destructor TaaCompoundFile.Destroy;
var
  i : integer;
  Folder : TCFFolder;
begin
  {destroy the open folders}
  if (FOpenFolders <> nil) then begin
    for i := pred(FOpenFolders.Count) downto 0 do begin
      Folder := TCFFolder(FOpenFolders.List^[i]);
      CloseFolder(Folder);
    end;
    FOpenFolders.Free;
  end;
  {destroy the root if it was opened}
  cfSaveRootFolder;
  TCFFolder(FRoot).Free;
  {destroy the FAT}
  cfWriteFAT;
  FFATBlocks.Free;
  FFAT.Free;
  {free the header block}
  if (FHeader <> nil) then begin
    cfWriteBlock(0, FHeader^);
    FreeMem(FHeader, CFBlockSize);
  end;
  {close the stream}
  FStream.Free;
  {destroy the ancestor}
  inherited Destroy;
end;

function TaaCompoundFile.cfAddBlock(var aBlock) : integer;
begin
  Result := FCFSize div CFBlockSize;
  cfWriteBlock(Result, aBlock);
end;

procedure TaaCompoundFile.cfPrepare;
var
  Header  : PCFHeader;
  FATNode : TFATNode;
  RootDir : TCFBlock;
begin
  {initialize the header (this will block 0)}
  Header := FHeader;
  FillChar(Header^, CFBlockSize, 0);
  Header^.cfhSignature := CFSignature;
  Header^.cfhBlockSize := 512;
  Header^.cfhFATSize := 1;
  {write out the header}
  cfAddBlock(Header^);
  {initialize the first FAT node
    (most entries are "unused")}
  FillChar(FATNode, sizeof(FATNode), $FF);
  FATNode[0] := EndOfChain;
  FATNode[1] := EndOfChain;
  FATNode[2] := EndOfChain;
  {write out the first FAT node; set up the in-memory FAT}
  cfAddBlock(FATNode);
  cfReadFAT;
  {initialize the root directory}
  FillChar(RootDir, sizeof(RootDir), 0);
  {write out the root directory}
  cfAddBlock(RootDir);
end;

procedure TaaCompoundFile.cfReadFAT;
var
  i       : integer;
  Header  : PCFHeader;
  FATNode : TFATNode;
  FATInx  : integer;
begin
  {prepare the in-memory FAT}
  Header := FHeader;
  FFAT.Clear;
  FFAT.Capacity := Header^.cfhFATSize * CFFATNodeEntryCount;
  FFAT.IsSorted := false;
  FFATBlocks.Clear;
  FFAT.Capacity := Header^.cfhFATSize;
  FFAT.IsSorted := false;
  {the FAT starts at block 1}
  FATInx := 1;
  {read the FAT blocks}
  while (FATInx <> EndOfChain) do begin
    FFATBlocks.Add(FATInx);
    cfReadBlock(FATInx, FATNode);
    for i := 0 to pred(CFFATNodeEntryCount) do
      FFAT.Add(FATNode[i]);
    FATInx := FFAT[FATInx];
    Assert(FATInx <> UnusedBlock,
      'TaaCompoundFile.cfReadFAT: unused block '+
      'in FAT chain');
  end;
end;

procedure TaaCompoundFile.cfReadHeader;
var
  Header : PCFHeader;
begin
  {first test: check the stream size}
  FCFSize := FStream.Size;
  if (FCFSize < 3 * CFBlockSize) or
    (((FCFSize div CFBlockSize) * CFBlockSize) <> FCFSize)
    then
    raise Exception.Create(
      'Stream is not a compound file: wrong size');
  {second test: check the first block is a compound file
    header}
  Header := FHeader;
  cfReadBlock(0, Header^);
  if (Header^.cfhSignature <> CFSignature) or
     (Header^.cfhBlockSize <> 512) or
     (Header^.cfhFATSize <= 0) then
    raise Exception.Create(
      'Stream is not a compound file: header invalid');
  {now read the FAT}
  cfReadFAT;
end;

procedure TaaCompoundFile.cfWriteBlock(aInx : integer; var
  aBlock);
var
  Offset : integer;
begin
  Offset := aInx * CFBlockSize;
  Assert((0 <= Offset) and (Offset <= FCFSize),
    'TaaCompoundFile.cfWriteBlock: Offset to write '+
    'is out of range');
  FStream.Seek(Offset, soFromBeginning);
  FStream.WriteBuffer(aBlock, CFBlockSize);
  if (Offset = FCFSize) then
    FCFSize := Offset + CFBlockSize;
end;
```

open or create a compound file, and, if it were altered, write it out again when we free the compound file. Having the FAT in memory all the time makes the entire process more efficient. Our best bet here is to use the integer list class we used last month.

We can now move on to write the constructor and destructor. We'll probably need to revise these as we go along, but at least we can test the basic creation, opening and closing of a compound file, even if we do nothing else.

(This is a tenet of Extreme Programming, by the way: write tests as you go along, and run them at every stage. That way, you'll know immediately if something gets broken as you write extra code and you can fix it immediately whilst the code is fresh in your mind. The tests themselves may have to be altered to suit the new circumstances, but they shouldn't be deleted. The premise behind this is that, when you write some code, it is the best time to write the tests that test the code. You will also have a better idea of any boundary conditions. Can this pointer be nil? What happens if the index passed in is out of range? And you can write tests for them straight away. The worst type of testing is 'write all the code, and then try and test it.')

Listing 1 shows the constructor and destructor for the compound file class. It also shows a set of underlying methods for performing some strict basic tasks: reading a block from the compound file with error checking, writing a block, reading the entire FAT into memory and writing it out again. The latter two are a little specialized, but they show the standard process of walking a FAT chain. They are interesting for another reason. To help with reading and writing the FAT we need to store the block numbers where the FAT is to be found. When we write a FAT back we have to write it in exactly the same blocks it was found in because the FAT contains its own FAT chain.

(For other entities, it doesn't matter if the entity's data is written back to different blocks than it was read from: the FAT takes care of it. However, the FAT must be written back exactly to the same place since it carries within it the numbers of the blocks that make it up. If we were cavalier in writing the FAT back, we could end up with a FAT that cannot find its own blocks. A disaster.)

Hence I've added a list of FAT blocks to the compound file class. This list is only used at runtime to track the blocks that make up the FAT, it is not written to disk inside the compound file. As it happens, this list is invaluable when the compound file has to grow the FAT by adding another block: we can easily maintain the FAT chain for the FAT itself.

When we create a new compound file, we have to set up and write the three standard blocks of the file: the header, the first node of the FAT table, and the first block of the root folder. The `cfPrepare` method takes care of this work. When we read an existing compound file we need to check that it's one of ours (otherwise goodness knows where we'll end up) and we need to read the FAT. The `cfReadHeader` method takes care of this chore.

Don't Slip Away

So far so good: we can create a new compound file, and open and close it. It's time to consider how to add something in it. This is where the coding started to balloon in complexity. I messed around for a while, writing this bit of code and that bit of code, before I hit on the answer. In the interests of making this seem as if it came to me fully formed, I'll skip the trial-and-error.

Before we add subfiles we should worry about folders. After all, a subfile *has* to be added to a folder (which could be the root, admittedly), so if we build the infrastructure for folders we should have an easier time for subfiles.

Recall that I decided to implement the functionality that we should be able to 'open' a folder and get a handle back so that we didn't have to implement a 'current directory'. At the high level, then, we should have `AddFolder` and `OpenFolder` methods. The `AddFolder` method would take in a handle to the parent folder, a name, then create a sub-folder in the parent folder and return a handle. `OpenFolder` would take a handle to the parent and a name, and then return a handle to the open folder. Why the parent handle? Well, this would be the way we would navigate the folder structure in the compound file. We would have to have a special property to get the root handle, but from then on we could open a subfolder off the root, a subfolder off that, and so on.

The other folder methods (`CloseFolder`, `WalkFolder` and `DeleteFolder`) would all take the handle returned by `AddFolder` or `OpenFolder` to do their stuff. (`WalkFolder` is a method to walk a folder by the way, calling a separate routine for each directory entry.)

There are problems with this scheme. Suppose I open the 'Parent' folder and then the 'Child' folder off that. I then close the 'Parent' folder. Without going into details, I've just made it hard to update the information for the 'Child' folder if I need to: I no longer have a handle to the 'Parent'. (You can see why I was doing a lot of experimenting...)

My solution was to create an internal class for a folder. Instances of this class would be reference counted. In the above example, opening 'Parent' would get me a handle with count 1. Opening 'Child' would get me a handle with count 1, again, but the code would be written so that it also opened up the 'Parent' folder again (incrementing its reference count). I'd end up with a parent handle with count 2, and a child handle with count 1. That way I could store a parent handle with the child handle, and not have to worry about the parent handle being closed. I could also reuse handles. Of course I would need a list of the open folders: the `FOpenFolders` field of the class.

Listing 2 shows this internal class, `TCFFolder`. At its most basic, it encapsulates an array of

directory entries. These entries are stored in a `TList` for convenience. There are methods to get a directory entry, given a name, to add a new directory entry (this method ensures that no duplicate entries could occur), and to view the directory entries as an array.

The `TCFFolder` class also maintains a little bit of state. There's the reference count to begin with, using two methods, `IncRefCount` and `DecRefCount`, to maintain it. Notice that `DecRefCount` will automatically free the object if the reference count reaches zero. The other bit of state is a `Modified` property: whenever something

➤ *Listing 2: The internal TCFFolder class.*

```
type
  TCFFolder = class
    private
      FCount    : integer;
      FList     : TList;
      FModified : boolean;
      FName     : string;
      FParent   : TaaHandle;
      FRefCount : integer;
    protected
      function cffGetCount : integer;
      function cffGetDirEntry(aInx : integer) :
        PaaCFDirEntry;
      procedure cffClear;
    public
      constructor Create(aParent : TaaHandle; const aName :
        string);
      destructor Destroy; override;
      function AddDirEntry(const aName : string;
        aType : TaaCFDirEntryType) : PaaCFDirEntry;
      procedure RemoveDirEntry(aDE : PaaCFDirEntry);
      function GetDirEntry(const aName : string;
        aType : TaaCFDirEntryType) : PaaCFDirEntry;
      procedure LoadFromStream(aStrm : TStream);
      procedure SaveToStream(aStrm : TStream);
      procedure MarkModified;
      function DecRefCount : boolean;
      procedure IncRefCount;
      property Count : integer read cffGetCount;
      property DirEntry[aInx : integer] : PaaCFDirEntry
        read cffGetDirEntry;
      property Modified : boolean read FModified;
      property Name : string read FName;
      property Parent : TaaHandle read FParent;
  end;
constructor TCFFolder.Create(aParent : TaaHandle;
  const aName : string);
begin
  inherited Create;
  FParent := aParent;
  FName := aName;
  FList := TList.Create;
  FRefCount := 1;
end;
destructor TCFFolder.Destroy;
begin
  if (FList <> nil) then begin
    cffClear;
    FList.Free;
  end;
  inherited Destroy;
end;
function TCFFolder.AddDirEntry(const aName : string;
  aType : TaaCFDirEntryType) : PaaCFDirEntry;
begin
  Result := AllocMem(sizeof(TaaCFDirEntry));
  Result.deName := aName;
  Result.deType := aType;
  FList.Add(Result);
  MarkModified;
end;
procedure TCFFolder.cffClear;
var
  i     : integer;
  Entry : PaaCFDirEntry;
begin
  for i := 0 to pred(FList.Count) do begin
    Entry := FList.List^[i];
    Entry.deName := '';
    Dispose(Entry);
  end;
  FList.Clear;
  FCount := 0;
end;
function TCFFolder.cffGetCount : integer;
begin
  Result := FList.Count;
end;
function TCFFolder.cffGetDirEntry(aInx : integer) :
  PaaCFDirEntry;
begin
  Assert((0 <= aInx) and (aInx < Count),
    'TCFFolder.fGetDirEntry: index out of bounds');
  Result := PaaCFDirEntry(FList.List^[aInx]);
end;
function TCFFolder.DecRefCount : boolean;
begin
  dec(FRefCount);
  if (FRefCount > 0) then
    Result := false
  else begin
    Result := true;
    Free;
  end;
end;
function TCFFolder.GetDirEntry(const aName : string;
  aType : TaaCFDirEntryType) : PaaCFDirEntry;
var
  i : integer;
begin
  for i := 0 to pred(FList.Count) do begin
    Result := PaaCFDirEntry(FList.List^[i]);
    if (Result^.deType = aType) and
      (Result^.deName = aName) then
      Exit;
  end;
  Result := nil;
end;
procedure TCFFolder.IncRefCount;
begin
  inc(FRefCount);
end;
procedure TCFFolder.LoadFromStream(aStrm : TStream);
var
  i       : integer;
  Entry   : PaaCFDirEntry;
  NameLen : byte;
  CountInStrm : longint;
begin
  aStrm.Seek(0, soFromBeginning);
  cffClear;
  aStrm.ReadBuffer(CountInStrm, sizeof(longint));
  for i := 0 to pred(CountInStrm) do begin
    New(Entry);
    with Entry^ do begin
      aStrm.ReadBuffer(NameLen, sizeof(NameLen));
      SetLength(deName, NameLen);
      aStrm.ReadBuffer(deName[1], NameLen);
      aStrm.ReadBuffer(deType, sizeof(deType));
      aStrm.ReadBuffer(de1stBlock, sizeof(de1stBlock));
      aStrm.ReadBuffer(deSize, sizeof(deSize));
      aStrm.ReadBuffer(deTime, sizeof(deTime));
      aStrm.ReadBuffer(deAttr, sizeof(deAttr));
    end;
    FList.Add(Entry);
  end;
end;
procedure TCFFolder.MarkModified;
begin
  FModified := true;
end;
procedure TCFFolder.RemoveDirEntry(aDE : PaaCFDirEntry);
begin
  Dispose(aDE);
  FList.Remove(aDE);
  MarkModified;
end;
procedure TCFFolder.SaveToStream(aStrm : TStream);
var
  i       : integer;
  Entry   : PaaCFDirEntry;
  NameLen : byte;
  CountInStrm : longint;
begin
  aStrm.Seek(0, soFromBeginning);
  CountInStrm := Count;
  aStrm.WriteBuffer(CountInStrm, sizeof(longint));
  for i := 0 to pred(Count) do begin
    Entry := PaaCFDirEntry(FList.List^[i]);
    with Entry^ do begin
      NameLen := length(deName);
      aStrm.WriteBuffer(NameLen, sizeof(NameLen));
      aStrm.WriteBuffer(deName[1], NameLen);
      aStrm.WriteBuffer(deType, sizeof(deType));
      aStrm.WriteBuffer(de1stBlock, sizeof(de1stBlock));
      aStrm.WriteBuffer(deSize, sizeof(deSize));
      aStrm.WriteBuffer(deTime, sizeof(deTime));
      aStrm.WriteBuffer(deAttr, sizeof(deAttr));
    end;
  end;
end;
```

```
function TaaCompoundFile.cfGetRoot : TaaHandle;
var
  Strm : TMemoryStream;
  WorkRoot : TCFFolder;
begin
  if (FRoot = nil) then begin
    WorkRoot := TCFFolder.Create(nil, '');
    try
      if (PCFHeader(FHeader)^.cfhRootSize <> 0) then begin
        Strm := TMemoryStream.Create;
        try
          cfReadData(2, Strm, PCFHeader(
            FHeader)^.cfhRootSize);
          WorkRoot.LoadFromStream(Strm);
        finally
          Strm.Free;
        end;
      end;
    except
      WorkRoot.Free;
      raise;
    end;
    FRoot := WorkRoot;
  end;
  Result := FRoot;
end;
procedure TaaCompoundFile.cfReadData(aStartInx : integer;
  aStream : TStream; aLen : integer);
var
  Inx       : integer;
  DataBlock : TCFBlock;
  BytesToCopy : integer;
begin
  Assert(aLen <> 0,
    'TaaCompoundFile.cfReadData: length of data is zero');
  {position the stream at the start}
  aStream.Seek(0, soFromBeginning);
  {start at the first block}
  Inx := aStartInx;
  while (Inx <> EndOfChain) do begin
    Assert(aLen <> 0,
      'TaaCompoundFile.cfReadData: more data present '+
      'than length indicates');
    {read the current block}
    cfReadBlock(Inx, DataBlock);
    {write it to the stream}
    if (aLen < CFBlockSize) then
      BytesToCopy := aLen
    else
      BytesToCopy := CFBlockSize;
    aStream.WriteBuffer(DataBlock, BytesToCopy);
    dec(aLen, BytesToCopy);
    {advance along to the next block}
    Inx := FFAT[Inx];
    Assert(Inx <> UnusedBlock,
      'TaaCompoundFile.cfReadDir: unused block '+
      'in FAT chain');
  end;
  Assert(aLen = 0, 'TaaCompoundFile.cfReadData: less data '+
    'present than length indicates');
end;
```

➤ *Listing 3: Reading the root folder.*

changes in the folder (a subfile's directory entry gets added, updated, or deleted), this flag will be set. This way we can easily see whether the folder information has changed and that it needs to be updated inside the compound file.

A further layer, supplied by the `LoadFromStream` and `SaveToStream` methods, gives us a way to make the folder persistent. We'll see how the compound file calls these methods in a moment.

Now back to the compound file. We need to see how this folder class is used. Listing 3 shows the code for the `cfGetRoot` method (the accessor method for the `Root` property) as well as a special internal method for reading the entire data for a file or folder into a stream. Let's look at the method to read the root first. Since the root folder is going to be used practically all the time, it makes sense to only read it once. We see here that, should the root folder have something in it, a temporary memory stream is created and the entire root folder data is read into it. The root folder object then initializes itself from this stream. The method returns the folder instance as a bare pointer: this is the root folder handle. As for the real meat: the `cfReadData` method reads the data for a file or folder based on the start index for the entity and its length. For sanity's sake, I peppered `Assert` statements throughout this method: after all there are two ways of measuring the length of the data, the number of blocks reported by the FAT and the actual length reported by the directory entry.

```
function TaaCompoundFile.AddFolder(aParent : TaaHandle; const aName : string) :
  TaaHandle;
var
  DE     : PaaCFDirEntry;
  Folder : TCFFolder;
begin
  {check that the parent is a valid folder}
  if not cfIsValidFolder(aParent) then
    raise Exception.Create(
      'TaaCompoundFile.AddFolder: parent is not valid handle');
  {get directory entry of folder; if we succeed folder already exists-- error}
  DE := TCFFolder(aParent).GetDirEntry(aName, detFolder);
  if (DE <> nil) then
    raise Exception.Create(
      'TaaCompoundFile.AddFolder: name already exists as valid folder');
  {create the folder}
  Folder := TCFFolder.Create(aParent, aName);
  Folder.MarkModified;
  {add the folder name to the parent's directory list}
  TCFFolder(aParent).AddDirEntry(aName, detFolder);
  TCFFolder(aParent).IncRefCount;
  {add the folder to the open folders list, return the folder}
  FOpenFolders.Add(Folder);
  Result := TaaHandle(Folder);
end;
```

➤ *Listing 4: Adding a new folder.*

We can now look at the `AddFolder` method (so that we can get something into the root folder!). Listing 4 shows this method. First, we verify that the parent handle exists. Next we check to see if the parent handle has a directory entry for this particular name. If not we can go ahead and create a new instance of `TCFFolder`. We now have a lot of housekeeping to do: we need to add a directory entry to the parent folder, increment the reference count for the parent (we've just made a copy of its handle, after all), mark the parent handle as having been modified, add the new folder to the open folders list. At that point, we can return the new handle (again, a bare pointer to the `TCFFolder` instance).

Not too bad, eh? The `OpenFolder` method is not too different from the `cfGetRoot` method, except that it too, has to check for its parent being valid. Also, looking at Listing 5, you can see that we make an effort to find the folder in the list of open folders first. If we do find it there, we just increment the reference counts of both the folder and its parent, and reuse the folder handle. If not, we open the folder in

```
function TaaCompoundFile.OpenFolder(aParent : TaaHandle;
  const aName : string) : TaaHandle;
var
  DE     : PaaCFDirEntry;
  Strm   : TMemoryStream;
  Folder : TCFFolder;
  Handle : TaaHandle;
begin
  {check that the parent is a valid folder}
  if not cfIsValidFolder(aParent) then
    raise Exception.Create('TaaCompoundFile.OpenFolder: '+
      'parent is not valid handle');
  {get the directory entry of the folder; if this fails, the
    folder name doesn't exist in the parent}
  DE := TCFFolder(aParent).GetDirEntry(aName, detFolder);
  if (DE = nil) then
    raise Exception.Create('TaaCompoundFile.OpenFolder: '+
      'name is not valid folder');
  {check to see if the folder hasn't already been opened; in
    which case just increment the reference counts, return
    the open handle and exit}
  if cfIsOpenFolder(aParent, aName, Handle) then begin
    TCFFolder(aParent).IncRefCount;
    TCFFolder(Handle).IncRefCount;
    Result := Handle;
    Exit;
  end;
  {create and read the folder}
  Folder := TCFFolder.Create(aParent, aName);
  try
    if (DE^.deSize <> 0) then begin
      Strm := TMemoryStream.Create;
      try
        cfReadData(DE^.de1stBlock, Strm, DE^.deSize);
        Folder.LoadFromStream(Strm);
      finally
        Strm.Free;
      end;
    end;
  except
    Folder.Free;
    raise;
  end;
  {increment the reference count for the parent}
  TCFFolder(aParent).IncRefCount;
  {add folder to the open folders list, return the folder}
  FOpenFolders.Add(Folder);
  Result := TaaHandle(Folder);
end;

procedure TaaCompoundFile.CloseFolder(aHandle : TaaHandle);
var
  i      : integer;
  Folder : TCFFolder;
  Parent : TCFFolder;
begin
  {if the handle is not nil, nor the root...}
  if (aHandle <> nil) and (aHandle <> FRoot) then
    {find the folder in the open folders list...}
    for i := 0 to pred(FOpenFolders.Count) do begin
      Folder := TCFFolder(FOpenFolders.List^[i]);
      {if the current item is the passed handle...}
      if (aHandle = Folder) then begin
        {get the parent}
        Parent := TCFFolder(Folder.Parent);
        {decrement the reference count for the open folder}
        cfSaveFolder(Folder);
        if Folder.DecRefCount then
          FOpenFolders.Delete(i);
        {decrement the reference count for the parent}
        if (Parent <> FRoot) then begin
          cfSaveFolder(Parent);
          if Parent.DecRefCount then
            FOpenFolders.Remove(Parent);
        end;
        Exit;
      end;
    end;
end;

procedure TaaCompoundFile.cfSaveFolder(aHandle : TaaHandle);
var
  Parent : TCFFolder;
  Folder : TCFFolder;
  DE     : PaaCFDirEntry;
  Strm   : TMemoryStream;
begin
  {get the folder from the handle}
  Folder := TCFFolder(aHandle);
  {if the folder was modified...}
  if Folder.Modified then begin
    {get the parent handle}
    Parent := TCFFolder(Folder.Parent);
    {get the directory entry in the parent for this folder}
    DE := Parent.GetDirEntry(Folder.Name, detFolder);
    Assert(DE <> nil,
      'TaaCompoundFile.cfSaveFolder: parent dir entry '+
      'not found');
    {if the folder is empty...}
    if (Folder.Count = 0) then begin
      {make sure it uses no blocks}
      if (DE^.de1stBlock <> 0) then begin
        cfReleaseChain(DE^.de1stBlock, true);
        DE^.de1stBlock := 0;
      end;
      {update the parent}
      DE^.deSize := 0;
      DE^.deTime := Now;
      Parent.MarkModified;
    end
    {otherwise the folder has directory entries}
    else begin
      {if this folder has never been written, get the first
        block}
      if (DE^.de1stBlock = 0) then
        DE^.de1stBlock := cfGetEmptyBlock;
      {copy the folder data to a stream, and from thence to
        the compound file}
      Strm := TMemoryStream.Create;
      try
        {save the folder to the stream}
        Folder.SaveToStream(Strm);
        {save the stream to the compound file}
        cfWriteData(DE^.de1stBlock, Strm);
        {update the parent}
        DE^.deSize := Strm.Size;
        DE^.deTime := Now;
        Parent.MarkModified;
      finally
        Strm.Free;
      end;
    end;
  end;
end;

procedure TaaCompoundFile.cfSaveRootFolder;
var
  Folder : TCFFolder;
  Strm   : TMemoryStream;
begin
  {get the root folder}
  Folder := TCFFolder(FRoot);
  {if the folder was modified...}
  if (Folder <> nil) and Folder.Modified then begin
    {copy the folder data to a stream, and from thence to
      the compound file}
    Strm := TMemoryStream.Create;
    try
      {save the folder to the stream}
      Folder.SaveToStream(Strm);
      {save the stream to the compound file}
      cfWriteData(2, Strm);
      {update the header}
      PCFHeader(FHeader).cfhRootSize := Strm.Size;
    finally
      Strm.Free;
    end;
  end;
end;

procedure TaaCompoundFile.cfWriteData(aStartInx : integer;
  aStream : TStream);
var
  Inx       : integer;
  NewInx    : integer;
  DataBlock : TCFBlock;
  BytesToGo : integer;
  BytesTOCopy : integer;
begin
  {position the stream at the start}
  aStream.Seek(0, soFromBeginning);
  {start at the first block}
  Inx := aStartInx;
  {release all subsequent blocks}
  cfReleaseChain(aStartInx, false);
  {calculate the number of bytes to write to the first block
   (we don't have to allocate this one: it's already done)}
  BytesToGo := aStream.Size;
  if (BytesToGo > CFBlockSize) then
    BytesToCopy := CFBlockSize
  else begin
    FillChar(DataBlock, sizeof(DataBlock), $CC);
    BytesToCopy := BytesToGo;
  end;
  dec(BytesToGo, BytesToCopy);
  {copy the data over for the first block}
  aStream.ReadBuffer(DataBlock, BytesToCopy);
  cfWriteBlock(Inx, DataBlock);
  {while there is still more data to write...}
  while (BytesToGo <> 0) do begin
  {calculate the number of bytes to write to the next block}
    if (BytesToGo > CFBlockSize) then
      BytesToCopy := CFBlockSize
    else begin
      FillChar(DataBlock, sizeof(DataBlock), $CC);
      BytesToCopy := BytesToGo;
    end;
    dec(BytesToGo, BytesToCopy);
    {allocate another block from the compound file}
    NewInx := cfGetEmptyBlock;
    FFAT[Inx] := NewInx;
    Inx := NewInx;
    {copy the data over}
    aStream.ReadBuffer(DataBlock, BytesToCopy);
    cfWriteBlock(Inx, DataBlock);
  end;
end;
```

➤ *Facing page, Listing 5: Opening and closing a folder.*

the same manner as we did the root.

Closing a folder leads us to another quirk. So far we have not written the folder information back to the compound file. All changes are held in memory. Well, when we close a folder handle, it behooves us to update it inside the compound file. Listing 5 also shows us this close operation. We find the handle in the list of open folders. Once we have it, we make a note of its parent, save the folder and then decrement the folder's reference count. Why make a note of its parent prior to the decrement? Well, internally this method call might free the folder object, and if we were to then read its `Parent` property we'd hit an access violation. Anyway, if the reference count is reduced to zero and the object freed, the method will return true and we then remove the item (which no longer exists) from the list of open folders. At this point we can do exactly the same to the parent handle.

So that leaves the real work to the `cfSaveFolder` method. It is this method that is responsible for making sure that the folder gets written to the compound file before its memory might be freed. First thing it does, of course, is to make sure that the folder was altered, for if it weren't there would be nothing to save. Now the fun stuff starts. The method finds the directory entry for the folder in its parent folder (if this is not found, an assertion fails: it should be there). If the folder being updated is empty, we must release any blocks occupied by it back to the FAT. This is a simple matter: we merely mark the blocks it uses as unused, which is an operation done entirely in the FAT. We then update the directory entry and mark the parent as being modified (the directory entry we modified belongs to the parent, remember). If, on the other hand, the folder did have some entries, we create a temporary memory stream, save the folder to it and then copy the stream data to the compound file using the `cfWriteData` method. Again, we then alter the directory entry for the folder and mark the parent as modified. Notice that we may have to allocate the initial block for the folder, but not the others: that's the job of the `cfWriteData` method.

`cfWriteData` is a fairly simple workhorse routine. First it releases all the blocks occupied by the folder data, except the first (we'll reuse that one). The reason for this is code simplicity. If we didn't, we'd have to cater for the cases that the folder data occupied more or less blocks than before, which I would guess to be some fairly convoluted code. By releasing all the blocks, it makes the writing of the data simpler: apart from the first block, we shall always have to allocate new blocks to hold the data. It is also extremely likely that we'll reuse the blocks we just deallocated anyway. `cfWriteData` goes into a trivial enough loop of allocating a new block, updating the FAT table, and copying data from the stream to the new block. The only complication is the minor housekeeping code to keep track of how much data there is left to copy.

The analogous method for saving the root folder is very similar, but much slimmed down. Listing 5 has this method as well.

Alright

At this point, we have pretty well sewn up all the folder activities that the compound file can do. I'm going to leave the deletion of a folder as an Easy Exercise For The Reader, the only complication being that you don't really want to delete a folder that contains subfiles or other folders, or that is open with a reference count of greater than one.

So, finally, we reach the topic of the subfiles. Here, for the purposes of the article, I cheated a little. I decided to implement just three subfile-related methods: `ReadSubfile` will read the entire data comprising a subfile into a stream; `UpdateSubfile` will either update an existing subfile's data from a stream or create a new subfile and copy the data over; and `DeleteSubfile` will delete a subfile.

Listing 6 shows the stream class that exposes a subfile. Simple to write and to use. The constructor takes as parameters the compound file instance, a folder handle, the subfile name and a flag to say whether to create the subfile. Internally it creates a `TMemoryStream` instance and, if the subfile is an existing one, will read all of the data using `ReadSubfile`. The usual stream operations for

➤ *Listing 6: The subfile stream class.*

```
constructor TaaSubfileStream.Create(aCF : TaaCompoundFile; aFolder : TaaHandle;
  const aName : string; aCreate : boolean);
begin
  inherited Create;
  FCF := aCF;
  FFolder := aFolder;
  FName := aName;
  FStream := TMemoryStream.Create;
  if aCreate then
    aCF.DeleteSubfile(aFolder, aName)
  else
    aCF.ReadSubFile(aFolder, aName, FStream);
end;
destructor TaaSubfileStream.Destroy;
begin
  if FModified then
    FCF.UpdateSubfile(FFolder, FName, FStream);
  FStream.Free;
  inherited Destroy;
end;
function TaaSubfileStream.Read(var Buffer; Count : longint) : longint;
begin
  Result := FStream.Read(Buffer, Count);
end;
function TaaSubfileStream.Write(const Buffer; Count : longint) : longint;
begin
  Result := FStream.Write(Buffer, Count);
  FModified := true;
end;
function TaaSubfileStream.Seek(Offset : longint; Origin : Word) : longint;
begin
  Result := FStream.Seek(Offset, Origin);
end;
```

```
procedure TaaCompoundFile.DeleteSubfile(aFolder : TaaHandle;
  const aName   : string);
var
  DE : PaaCFDirEntry;
  Folder   : TCFFolder;
begin
  {check that the folder is valid}
  if not cfIsValidFolder(aFolder) then
    raise Exception.Create(
      'TaaCompoundFile.DeleteSubfile: parent is not '+
      'valid handle');
  {get the directory entry of the subfile}
  Folder := TCFFolder(aFolder);
  DE := Folder.GetDirEntry(aName, detSubfile);
  {if the directory entry exists...}
  if (DE <> nil) then begin
    {free all the blocks occupied by the subfile}
    if (DE^.de1stBlock <> 0) then
      cfReleaseChain(DE^.de1stBlock, true);
    {remove the directory entry}
    Folder.RemoveDirEntry(DE);
  end;
end;
procedure TaaCompoundFile.ReadSubfile(aFolder : TaaHandle;
  const aName   : string; aStream : TStream);
var
  DE : PaaCFDirEntry;
begin
  {check that the folder is valid}
  if not cfIsValidFolder(aFolder) then
    raise Exception.Create(
      'TaaCompoundFile.ReadSubfile: parent is not '+
      'valid handle');
  {get the directory entry of the subfile; if this fails,
   the subfile name doesn't exist in the folder}
  DE := TCFFolder(aFolder).GetDirEntry(aName, detSubfile);
  if (DE = nil) then
    raise Exception.Create(
      'TaaCompoundFile.ReadSubfile: name is not '+
      'valid subfile');
  {if there's some data, copy it to the stream}
  aStream.Seek(0, soFromBeginning);
  if (DE^.deSize <> 0) then
    cfReadData(DE^.de1stBlock, aStream, DE^.deSize);
  aStream.Size := DE^.deSize;
end;
procedure TaaCompoundFile.UpdateSubfile(aFolder : TaaHandle;
  const aName   : string; aStream : TStream);
var
  DE : PaaCFDirEntry;
  StrmSize : integer;
  Folder   : TCFFolder;
begin
  {check that the folder is valid}
  if not cfIsValidFolder(aFolder) then
    raise Exception.Create(
      'TaaCompoundFile.UpdateSubfile: parent is not '+
      'valid handle');
  {get the directory entry of the subfile}
  Folder := TCFFolder(aFolder);
  DE := Folder.GetDirEntry(aName, detSubfile);
  {if the directory entry doesn't exist, create a new one}
  if (DE = nil) then
    DE := Folder.AddDirEntry(aName, detSubfile);
  {if the stream is empty, make sure the existing blocks are
    freed}
  StrmSize := aStream.Size;
  if (StrmSize = 0) then begin
    if (DE^.de1stBlock <> 0) then begin
      cfReleaseChain(DE^.de1stBlock, true);
      DE^.de1stBlock := 0;
    end;
  end
  {otherwise there's some data to write}
  else begin
    {if this subfile has never been written, get the first
      block}
    if (DE^.de1stBlock = 0) then
      DE^.de1stBlock := cfGetEmptyBlock;
    {save the stream to the compound file}
    cfWriteData(DE^.de1stBlock, aStream);
  end;
  {update the folder}
  DE^.deSize := StrmSize;
  DE^.deTime := Now;
  Folder.MarkModified;
end;
```

➤ *Listing 7: Operations on subfiles.*

the subfile stream then act directly on the internal memory stream. When the subfile stream is closed, the destructor will make sure the subfile exists in the compound file and then copy all of the data over using UpdateSubfile.

Since the subfile stream class was so easy, the hard work must be done in the compound file methods. In reality we've seen much of the processing already: it's roughly the same code as we've already discussed for the folders. I therefore present it without comment in Listing 7.

Comment Te Dire Adieu

In this article, we've come a long way. Even so, the compound file we've dissected is very basic. There are numerous improvements that can be made. Apart from the examples I've already given, some improvements could be made in the area of subfile processing. In trying to keep things simple, I made the subfile code copy the data in and out of the compound file by use of middle-man memory streams. A better bet would be to directly access the data in the compound file with the subfile stream. Another efficiency improvement would be the search for unused blocks: at the moment it's a sequential search from the beginning (it executes in linear time), but can you devise another method that will work in constant time? Anyway, I hope you've enjoyed this foray into applied algorithms.

Julian Bucknall is looking forward to summer in this little ville. By the time you read this, TurboPower will have moved into new offices, and he'll have just about unpacked. He can be reached at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.