Document :: TTreeView
Author :: Jon Jenkinson
From "The Bits..." the c++ Builder Information & Tutorial Site
http://www.cbuilder.dthomas.co.uk

No liability is accepted by the author(s) for anything which may occur whilst following this tutorial

*****This project has been built and tested under Win95, (4.00.950 not OEMR2), and using CBuilder Standard, (both patches). It has not been tested under later releases of Win95, NT nor different versions of CBuilder, and whilst no side effects should occur, follow this at your own risk.*****

TTreeView

After using Windows for any period of time, users become familiar with the TreeView, and probably come to expect it for all things hierarchical. As CBuilder programmers we have a head start, and though it isn't very obvious, the TTreeView is very easy to use.

In this tutorial we'll start by creating nodes, (doing the easy stuff:), and move through some of the more advanced features of the TTreeView showing how to link data with a node so that you don't have to *track* the nodes yourself.

We'll then save and load our tree, (and data), before adding finishing touches just to prove we can:).

This tutorial is quite long, mainly because of the explaining that goes on. It is worth following through to the end so that you can become fully familiar with one of the most flexible components CBuilder has to offer.

Colour Coding:

```
Information displayed in this colour and font is information provided by my
copy of BCB.
Information displayed in this colour and font is information you need to type.
```
Information displayed in this colour and font is information which is of general interest.

Underlying Principals

What is a TreeView. Well we all know that don't we. Using Explorer pretty well everyday we are now familiar with the way the TreeView works, do we know the terms. To explain the terms we'll

be using, we'll take two directory paths,
C:\Program Files\Borland\CBuilder\Projects
D:\Builder Projects\TreeView

Now, you can see instantly the "paths" we've used, we're used to it. In a TreeView this would be displayed as

```
C
      Program Files
            Borland
                  CBuilder
                        Projects
D
      Builder Projects
            TreeView
```

The terms we need to become familiar with are as follows.

Node Each item (*"thing"*) in our list is a node.
Root Node In our example we have two Root Nodes, "C" and "D". In Explorer the Root is usually "My Computer".

Sibling In our example, "D" is a sibling of "C", and that's it. A sibling is a node at the same ***Level*** with the same ***Parent***.

Level Our first tree has 5 levels, and our second 3. "C" & "D" are at the same level, "Program Files" & "Builder Projects" are at the same level, "Borland" and "TreeView" are at the same level. *Don't confuse Level with* ***Sibling****, which can easily be done.*

Parent The Parent term is as obvious as it seems. "Program Files" Parent is "C", and
Child "Program Files" has "Borland" as it's child. However, note that a node can have many children, but can only have one parent, (indeed, in the case of "C" & "D", a node can have no parent, and "Projects" has no children).

Path Is the link between nodes. For those who remember DOS or UNIX path's, the same is true here. We can step across paths, though the need to do this is rare, we normally traverse them.

Data A very loose term. Basically *anything* you wish to associate with a node is termed as data. This can be a simple string, another component, a dynamically created object, (as we'll do),….literally anything for which you can create a pointer.

So what does all this mean. If you remember our path's above, we can quickly visualize our tree from them, and you should be able to go the other way around. As you would in DOS, you go up and down the directory *tree*, so you go up and down our TreeView. There is one major difference though. Unlike the DOS tree, where to get to a given node you have to *travel* through other nodes, in a TreeView we can *jump* to a given node, either by ID, index, or more normally pointer.

TTreeView

Okay, now the lecture is out of the way, let's start working with a TTreeView. To do this we'll have a form, a TTreeView and a pop-up menu. We'll start by adding and deleting nodes, but then move onto the more advanced area of associating data with a node, in this case creating a

form dynamically for each node.

Once we've done that, we'll learn how to move nodes around the tree, without having to worry about moving our data link, it's already there:). Finally, we'll save and load a TreeView showing how to store the path, and bring back and recreate our data.

To start with, let's.

Step One : Create our application

Choose File|New Application,

Drop a TTreeView on the form, (Win95 Tab), and set it's Name property as FView.
Drop a TPopMenu on the form, (Standard Tab), leave all properties as default..
Drop a TOpenDialog on the form, (Dialogs Tab), Name it LoadTree
Drop a TSaveDialog on the form, (Dialogs Tab), Name it SaveTree.
Drop a TImageList on the form, (Win95Tab), leave all properties as default.
Drop a TColorDialog on the form, (Dialogs Tab), Name it FColor, and set cdFullOpen to true.

And that's all the components we'll use.

Step Two : Plug the Bits Together

Now we'll simply plug the pieces together.

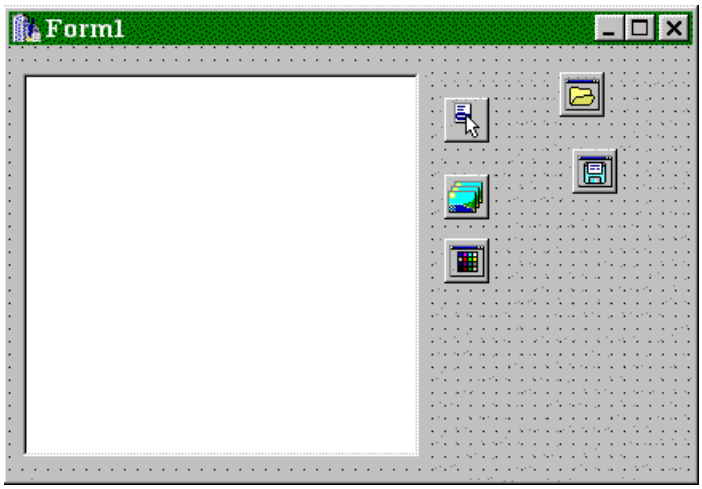
Set the following properties of FView, (our TTreeView).

Images	: use the list and select ImageList1
PopupMenu	: use the list and select PopupMenu1
SortType	: Choose stText
StateImages	: use the list and select ImageList1

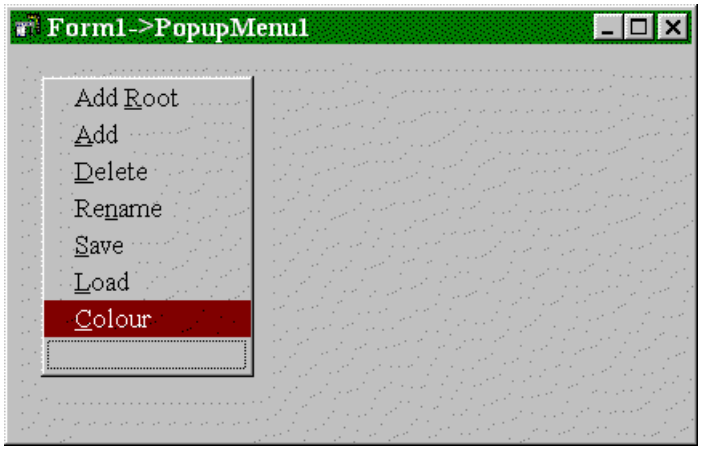
Next, double click on PopupMenu1 to bring up the menu editor. Add the following items,

Caption	Name
Add &Root	AddRootNode
&Add	AddNode
&Delete	DeleteNode
Re&name	RenameNode
&Save	SaveOurTree
&Load	LoadOurTree
&Colour	FormColour

Your form should look something like this, (resize things if you wish:),



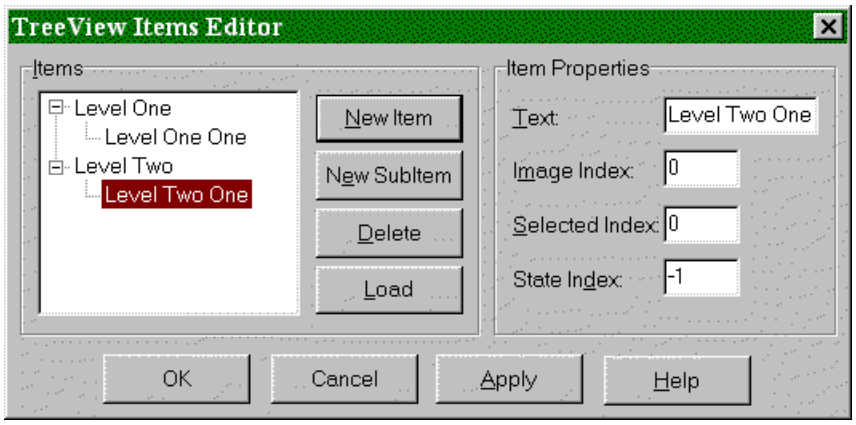
and your menu editor like this,



Step Three : Design Time Nodes

If you're certain about the nodes you'll have, you can create them all at design time. Whilst this is possible for some default nodes, I personally prefer to *hard code* such nodes during application start up. However, the TTreeView component has a node editor, which is handy to show how the things link together.

In the Object Inspector you'll see a property called Items. Click on the ellipse to bring up the Node Editor, and enter the following, (by pressing New Item, and Sub Item, play a little to get the hang of it). Leave the image index references for now, more on those later.

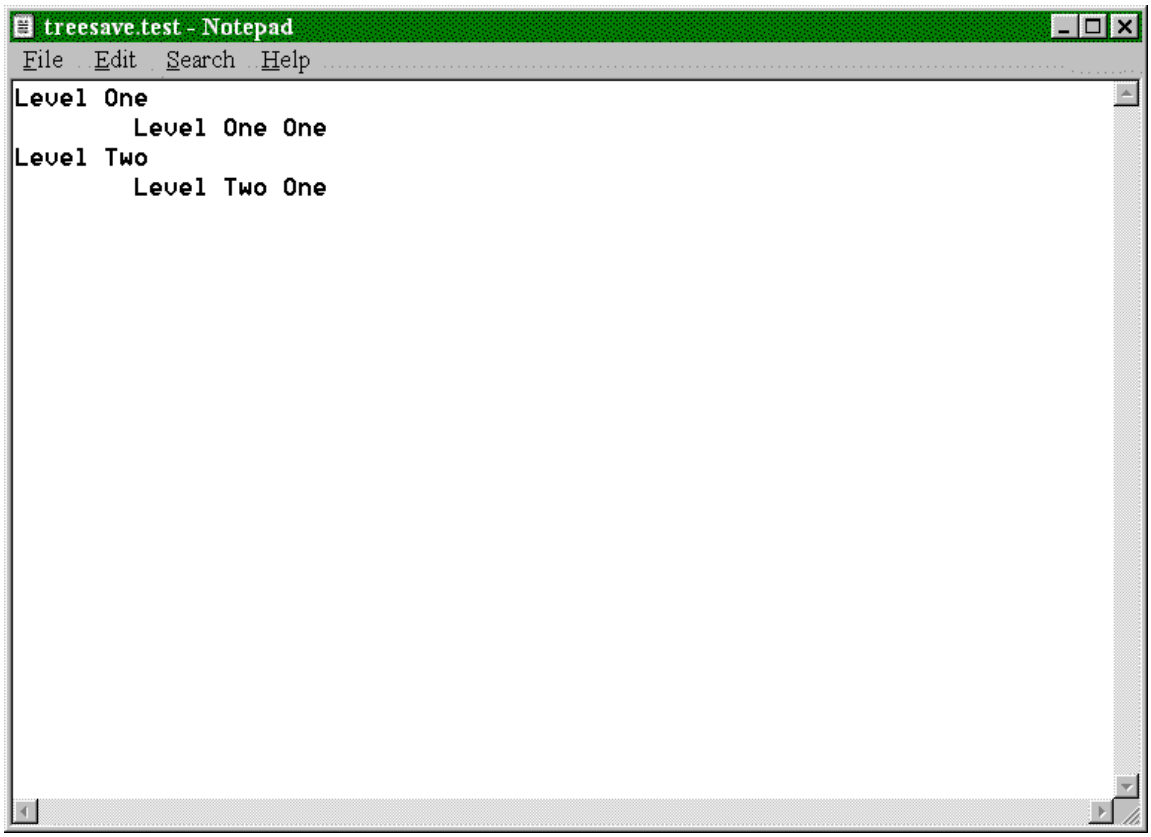


Click OK, and notice that the design time display shows the Level One and Level Two Root nodes, and adds two little "+" signs to show that there are some sub nodes. Hit F9 to Compile and run the application, and at run time click the "+" to expand the tree, (the "-" that replaces the "+" closes the tree).

When you've finished telling yourself how easy that all was, let's make it a little harder:). In the OnClick event handlers for PopupMenu1, SaveOurTree, LoadOurTree, place the following code,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::SaveOurTreeClick(TObject *Sender)
{
//save out tree out
if(SaveTree->Execute())
        {
        FView->SaveToFile(SaveTree->FileName);
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::LoadOurTreeClick(TObject *Sender)
{
//load our tree in.
if(LoadTree->Execute())
        {
        FView->LoadFromFile(LoadTree->FileName);
        }
}
//---------------------------------------------------------------------------
```

Run the application, right click on the TTreeView and select Save, giving any filename you want, and then right click and press load. You wont notice anything yet because we don't clear the TTreeView, but what I want to look at first is the file we just saved. Close the application and open the file you just created in Notepad, you'll get something like this,



Okay ? Here is our first problem. Although the above methods make it easy to Load and Save a TTreeView, if you think of a real world application of a TTreeView, you expect the node to actually point to something, (a directory in Explorer for instance), and the standard format doesn't save those links for us. We'll cover that later, when we start associating data with each node.

So we just made it harder, the Load/Save is pretty useless in a real world application, as indeed is the use of design time nodes.

Step Four : Run Time, real Nodes

Okay, now we'll take a leap of faith. Leave the above load and save routines for now, but go back to the node editor and delete all the nodes we added before, (hit the delete button for the roots and the children are deleted as well). Compile and run, and just to prove it works, right click and load the file you saved a moment ago. The TTreeView should now be restored:).

Close the application, and let's move on. A run-time node is the same as a design time node, except we have to do the logic.

Every TTreeView has a Root node that is invisible to us. If you look at our current Design time display we have a blank TTreeView. True ? Well not quite. The TTreeView creates, (at runtime at least:), an array of pointers to hold all the nodes it can expect to display, (**TTreeView->Items**). As we have no nodes at present, this array of pointers is NULL. However, in terms of logic, this NULL is our Root Node, the ultimate Parent of any nodes we may add.

In logic terms, when you're traversing a TTreeView path, (traversing nodes), you work with pointers. Consider the following,

D:\NodeOne\ChildOne

It seems quite logical to get from ChildOne to NodeOne, you check to see if the ChildNode has a parent, (which it does), and from NodeOne to D:, again is there a Parent, (yes:), but D: ? Has D: got a parent ? Well yes, it's parent is NULL, (that is non-existent!). What this means in practice is that whenever you need to ensure you're at the highest possible level of the tree, you will find that a parent is NULL.

The problem here is that when you reach the lowest level of the tree, the next Child is also NULL. If we asked ChildOne for it's children, it would say NULL. Be aware of this. I have seen code where whenever a NULL is encountered it is treated as the ultimate parent, when in fact it should be the final child.

Okay, let's start by adding some nodes by code at runtime. We have allowed two menu options for this, "Add Root" and "Add". The reason for this will become clear in a moment.

Although we haven't done it here, you could add your own, ultimate parent, as Explorer does, ("My Computer"). What this means is that we would just use the "Add" function we'll outline below, that is, everything we add has, at least, the ultimate parent at it's root.

Place the following code in the OnClick handler of PopupMenu1|Add,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::AddNodeClick(TObject *Sender)
{
TTreeNode *TempNode;

TempNode = FView->Items->AddChild(FView->Selected, "Form 1");
}
//---------------------------------------------------------------------------
```

Before I explain the above, compile and run the application, and do the following at runtime. Right-Click in the TTreeView and select Add. We get a node called Form1 at the Root:). Do the same again, and we get another node, again called Form1, again at the Root, (don't highlight anything or,). If we now highlight the first Form1, (left click on it), and select Add again, we get a "+" sign. Click on the plus and you'll see a child node. This is the reason that you need an Add Root option as well as the Add option.

I mentioned earlier the fact that you can have an overall root of your own, for example a mail program may have a root called "Mail". Then you can have a single Add routine, which requires you to have selected a node before you can add a node. In our little example above, our first and second Form1 would become children of "Mail", with our third addition a child as it is now. I leave it up to you to decide on how you wish to do it, but for completeness I've shown a separate Add Root option.

Okay, what does the code above do. Well, as you've seen it adds nodes:). The line that actually does the adding

```
TempNode = FView->Items->AddChild(FView->Selected, "Form 1");
```

works as follows. FView->Items is our overall root, (remember the bit about NULL). We then tell it to add a child node to the node pointed to by FView->Selected, and caption it "Form1". Yes ?

Okay, the AddChild function takes two parameters, a TTreeNode, and an AnsiString text. The text is quite easy, but the TTreeNode ? When we first run our application, we have no nodes,

so how can we pass it one. This is where our NULL comes in. FView->Selected points to the current node selected in the TTreeView, but as we have no nodes on first run, it points to NULL, (our overall parent:). Thus when we add a child node to NULL, we get a root node.

However, when we select an existing node, and add a child, FView->Selected points to a node that exists, so the new node is added as a child of the selected one. The problem is, that once you've selected a node, you can't unselect a node, (*okay, okay, you can, you point FView->Selected=NULL, however the code for working out if there is a node under the mouse when a user wants to deselect, is complex and beyond my chosen scope, at this time:).*. Thus, if you have a node selected you can't add another root node:(.

This is where the Add Root function comes in, add the following to it's OnClick handler,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::AddRootNodeClick(TObject *Sender)
{
TTreeNode *TempNode;

TempNode = FView->Items->Add(NULL, "Form 1");
}
//---------------------------------------------------------------------------
```

It is almost identical to the previous function, except we use Add rather than AddChild. If you compile and run this now, it will work by creating a new root node, but… You could do this by passing FView->Selected, and given the help file description, *"The node is added last in the list to which the node specified by the Node parameter belongs."* , it took me a while to work out what it meant. Simply it is this. If you have a grandchild selected, (root\child\grandchild), the use of Selected would add the new node as a sibling of the parent, that is, a sibling of child not root. Passing NULL forces it all the way back to the top. *We could do the same by passing our AddChild function a NULL, but this is easier;).*

So we now have two methods for adding a node. We also have two methods which will always name the node "Form 1", which isn't much use. We'll make a function which will take a TTreeNode parameter, give it a default caption, and then check all the siblings of that node to ensure we have a unique caption. We'll call this function, NodeJustAdded. So, add the following to the header file, under private, as we only want it to be visible to our form.

```
private:            // User declarations
      void NodeJustAdded(TTreeNode *NewNode);
      String CaptionCheck(String Name, TTreeNode *CheckNode);
public:             // User declarations
```

We've added two function definitions for a reason. The first is where we'll add images to our node later, *and a lot more later on*, the second is where we'll check the name we're trying to give our node to make sure it doesn't already exist amongst one of it's siblings. (As you've noticed so far, the TTreeView doesn't care about the caption property, but to avoid confusing your users, you should make sure that the caption displayed is unique at each sibling level).

The reason I've separated the caption check into a separate function is that the caption can be renamed in two ways. One, as we're doing here, and two by the user, (unless you make the TTreeView read-only). We're fortunate that when the user changes the caption, an event is triggered which we'll handle later, but we'll do so by calling our CaptionCheck function.

So that we can handle the node naming in the two above functions, change the two OnClick handlers we have for adding a node, (AddNodeClick, and AddRootNodeClick so they look like the following, (making the changes in red).

```
//---------------------------------------------------------------------------
```

```
void __fastcall TForm1::AddRootNodeClick(TObject *Sender)
{
TTreeNode *TempNode;

TempNode = FView->Items->Add(NULL, "");

NodeJustAdded(TempNode);
}
//---------------------------------------------------------------------------
void __fastcall TForm1::AddNodeClick(TObject *Sender)
{
TTreeNode *TempNode;

TempNode = FView->Items->AddChild(FView->Selected, "");

NodeJustAdded(TempNode);
}
//---------------------------------------------------------------------------
```

Before moving on, a little explaining. What we're doing here is creating a node as before, but doing so without creating a caption of any kind, yet. This will be done with the call to NodeJustAdded, which looks like this, (you'll have to type in all the following),

```
//---------------------------------------------------------------------------
void TForm1::NodeJustAdded(TTreeNode *NewNode)
{
NewNode->Text = CaptionCheck("New Folder", NewNode);
FView->AlphaSort();
}
//---------------------------------------------------------------------------
```

Clear ? We take the node parameter passed by our previous calls, pass it on to the CaptionCheck function before assigning the returned string to our caption. We then force the TTreeView to do an AlphaSort to ensure that our newly named node fits into the structure correctly. So what does the CaptionCheck function do…

Quite a lot, enter the following, (again, all of it).

```
String TForm1::CaptionCheck(String Name, TTreeNode *CheckNode)
{
//check the passed node's level for duplicates of the caption. If found,
repeat
//until a caption is found that doesn't exist.

String TempName(Name);
int x;
TTreeNode *TempNodePointer;
bool okay;

//cycle through the current nodes at this level and force name.
//Get the first node.
okay = false;
x=2;
while(!okay)
      {
      if(CheckNode->Parent)
            {
            TempNodePointer = CheckNode->Parent->getFirstChild();
            }
      else
            {
            TempNodePointer = FView->Items->GetFirstNode();
            }
      while(TempNodePointer)
```

```
                {
                if(TempNodePointer->Text == TempName)
                        {
                        TempName = Name;
                        TempName = TempName + " (" + (String) x + ")";
                        x++;
                        TempNodePointer = NULL;
                        okay = false;
                        }
                else
                        {
                        TempNodePointer = TempNodePointer->getNextSibling();
                        okay = true;
                        }
                }
        }

//Now TempName = the right name, so return it,
return TempName;
}
//---------------------------------------------------------------------------
```

And to explain. Remember, at this stage we want a function which will name our node for us, regardless of it's level, and also regardless of the name we pass it, we just wish to make sure it's unique to the level of the node.

```
String TempName(Name);
int x;
TTreeNode *TempNodePointer;
bool okay;
```

declares the variables and pointers we will use,

```
okay = false;
x=2;
while(!okay)
        {
```

We then set okay to false, so that our while loop will execute, and x= 2, (though this could be one. The reason for this is to mimic the Win95 Explorer style, "New Folder", "New Folder (2)".) This loop is important as you'll see, we run through it FROM THE START each time we find a node that matches our search string, to check that a node we've already checked can't be missed when we change our search string.

```
if(CheckNode->Parent)
        {
        TempNodePointer = CheckNode->Parent->getFirstChild();
        }
else
        {
        TempNodePointer = FView->Items->GetFirstNode();
        }
```

The above is very important to us, and makes the following sections work !. What this basically says is that if the node has a parent we start at the first child of our parent, but if it hasn't a parent, we start at the first node in the tree, (if it has no parent, it has to be a root node). This is done with a call to the Parent property of our CheckNode, remember if it returns NULL, we're at the top and can go no higher.

If we have a parent, we set our TempNodePointer to the first Child of our nodes Parent. Note

the call to getFirstChild(), the lowercase g is correct, a misspelling in the header files. If we have no parent, we get the first node in the tree by asking the Items list to return it. (note the uppercase G this time:).

Now TempNodePointer points to the node where we wish to check from.

```
while(TempNodePointer)
       {
```

We now enter another while loop, staying there whilst we travel along the level. It will at some point be set to NULL, and then our loop will finish.

```
if(TempNodePointer->Text == TempName)
       {
       TempName = Name;
       TempName = TempName + " (" + (String) x + ")";
       x++;
       TempNodePointer = NULL;
       okay = false;
       }
```

Here we check the caption of TempNodePointer, (at this stage equal to the first node at this level), against the name we're trying to give to our node. If the node caption exists at this level, we change the name to
name, ("New Folder"), and add a space brackets number, " (2)". We then increment x for our next run through, set the pointer to NULL so that we drop out of this while loop, but ensure that okay is false so that we start at the top of this level and check our new name.

If the node caption isn't equal to our name,

```
       else
              {
              TempNodePointer = TempNodePointer->getNextSibling();
              okay = true;
              }
       }

   }
```

we get the next node at this level, (sibling node). Note again the case of getNextSibling, and also the fact that we set okay to true.

We're currently in two loops. If we reach the above else statement, we can assume that the TempName variable doesn't exist in any of the captions checked so far, (okay = true). If getNextSibling returns NULL, (no more nodes at this level), we need to ensure that we drop from both loops. **Ensure you follow the flow of the loops, it is very very easy to cause a program lock up with TTreeNode pointers, and it isn't always obvious why your program suddenly locks up. You have been warned.**

```
//Now TempName = the right name,  so set S to it,
return TempName;
```

and finally we return the name we've created.

Right. There was a lot to take in there, but to prove it works, run the application and start adding nodes. It doesn't matter if you add Root Nodes or Child nodes, the thing *should* work as I've designed, (assuming you typed in the above correctly:).

You'll find that although the nodes are all called "New Folder", they have a number on the end to avoid duplication. In a moment we'll see this put to better use, but for now back to where we are.

Before moving on, ensure you fully understand what happens in the CaptionCheck function. If you're not happy with it, work through it in the debugger step by step to see what things are pointing to, (tip, use a watch on TempNodePointer->Text, rather than a pointer address:). The logic will become clearer as you work more with TTreeViews and TTreeNodes.

The problem so far is that calling all the nodes the same means that we have a pretty useless naming convention, and, *(although you can edit by clicking with your mouse, which would spoil the tutorial:)*, we need a way of editing the node caption. From Explorer you'll see that when you add a "New Folder", you are put into edit mode, and the node is automatically made visible. We can do the same by adding the following to our NodeJustAdded function,

Step Five : Adding Renaming

Add the code in red to our function,

```
//---------------------------------------------------------------------------
void TForm1::NodeJustAdded(TTreeNode *NewNode)
{
NewNode->Text = CaptionCheck("New Folder", NewNode);
FView->AlphaSort();

if(NewNode->Parent)
      {
      NewNode->Parent->Expand(false);
      }
NewNode->EditText();
}
//---------------------------------------------------------------------------
```

We check to see if the node we've just added has a parent, *(if it hasn't we don't need to make it visible, as it is already visible, it's a root)*, and then we tell the Parent to expand all it's children. The parameter passed tells the parent how to expand, true means that it should recurse down all it's children and grandchildren expanding them all, false just one (child) level.

Run the application again, and you'll find that each time you add a node it waits for you to name it. Add a root node, "New Folder", and press enter. Now add another root node, "New Folder (2)", but delete the
space brackets and number this time before pressing enter. Problem, all the work we've just done has failed:(. Fortunately, rectifying this is easy.

When you edit a node, (that is actually change some of the text within a caption), an event is triggered when the edit finishes, (either by return or mouse click away from the node). This event, OnEdited, is part of the TTreeView, so add the following code to the OnEdited function,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FViewEdited(TObject *Sender, TTreeNode *Node,
      AnsiString &S)
{
S = CaptionCheck(S, Node);
}
//---------------------------------------------------------------------------
```

Run the application again, and repeat the above test, (adding two root nodes), and it doesn't quite work. You'll end up with "New Folder (3)", which isn't what we want. If you add root nodes, and then edit them to "test", you will end up with "test", "test (2)", etc., so why doesn't it

work with the New Folder ?

The reason is that S is equal to the new caption, but the current caption isn't actually changed until we return S above. That is, when we have S equal to "New Folder", our CaptionCheck looks at the current tree structure and finds a node called "New Folder", and one called "New Folder (2)", not two called "New Folder". It's up to you whether this is a problem or not, it is rare that node names wont be renamed, but for completeness we'll handle it here. **Be aware, although we'll hard code "New Folder" here, if you change the name passed in NodeJustAdded, you'll have to change it here as well. This string should really be a constant defined elsewhere, but for simplicity of explanation I didn't do that here**.

To ensure that you understand how the node's Text property and S interact, we'll also handle another occurrence, the leaving of an unwanted space at the end of the edited caption, (at least we don't want it:).

To handle these two problems, enter the following code in our handler

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FViewEdited(TObject *Sender, TTreeNode *Node,
      AnsiString &S)
{
String Delimiter(" ");
String CheckFor("New Folder");

if(Node->Text.SubString(0, Node->Text.LastDelimiter(Delimiter)-1) == CheckFor)
      {
      Node->Text.Delete(Node->Text.LastDelimiter(Delimiter), Node-
>Text.Length());
      }

if(S.LastDelimiter(Delimiter) == S.Length())
      {
      S.Delete(S.LastDelimiter(Delimiter), S.Length());
      }

S = CaptionCheck(S, Node);
}
//---------------------------------------------------------------------------
```

To explain. The first if statement checks the node text that we set during the adding of the node, and strips it down to just "New Folder". *(The use of the AnsiStrings for CheckFor and Delimiter is because the Node->Text is an AnsiString, not a char *, "c style string". Again, the CheckFor variable should be set as a constant in the header file.)* We then check the S parameter to remove any spare spaces. What this means is that when we call our CaptionCheck function, we will not have any erroneous entries. Run the application and check it works.

Remember, when we check the captions, the new text has yet to replace the old text.

We'll now add the manual rename, from the menu, and by pressing F2, (the mouse already does so:).

In the RenameNode OnClick handler, the following,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::RenameNodeClick(TObject *Sender)
{
if(FView->Selected)
      {
      FView->Selected->EditText();
      }
```

```
}
//---------------------------------------------------------------------------
```
And then set up the key in the OnKeyUpEvent.
```
//---------------------------------------------------------------------------
void __fastcall TForm1::FViewKeyUp(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
switch(Key)
      {
      case VK_F2:  //catch the F2,
            RenameNodeClick(Sender);
            break;
      default:     //ignore:)
            break;
      }//end switch
}
//---------------------------------------------------------------------------
```

Now hopefully you can see the benefit of all that work with the CaptionCheck function. Now whenever we wish to rename a caption we have a total routine to stop duplicate naming.

Step Six: Deleting Nodes

Now to handle deleting of nodes. The following call,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::DeleteNodeClick(TObject *Sender)
{
if(FView->Selected)
      {
      FView->Items->Delete(FView->Selected);
      }
}
//---------------------------------------------------------------------------
```

does this for us. Run the application, add a few nodes, and children and grandchildren. If you select a node and choose delete from our pop-up menu, notice that not only is the node deleted, but **ALL** it's children will be deleted as well. **(You could also handle the renaming of the nodes that are left if you wished, but I don't think that will be a very frequent requirement).** The if is there to check that there is a node selected, otherwise you'll throw an exception by trying to delete a non-existent tree, (if there are no nodes), or delete the whole tree, (remember back to the NULL argument earlier).

If you don't wish this to be the default, add the following code which asks the user to confirm,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::DeleteNodeClick(TObject *Sender)
{
int YesNo;

if(FView->Selected)
      {
      if(FView->Selected->HasChildren)
            {
            YesNo = Application->MessageBox("This will delete all children,
      Are You Sure ?",
                  "Warning", MB_YESNO|MB_ICONWARNING);
            }
      else
            {
            YesNo = IDYES;
```

```
            }
        if(YesNo == IDYES)
            {
            FView->Items->Delete(FView->Selected);
            }
        }
}
//---------------------------------------------------------------------------
```

Whilst you only need to type the code in red, it may be easier to type the lot:). It is quite straightforward, if the node points to NULL, we do nothing. If it doesn't, and the node has no children, we set to IDYES and delete it. If it does exist, and has children, then we throw the warning box. This will return either IDYES, or IDNO, which we then use for the delete test.

Step Seven: Making Data

Okay. A refresher. So far we have a TTreeView to which we can add nodes, rename nodes, delete nodes, and as final exercise later we'll bring in drag and drop to allow us to move nodes, but what use is it. At this stage, none at all.

A node only becomes useful when it *points* to something. Win95 Explorer is only useful because when you click a node it opens a directory in a TListView, and in my own uses I usually have one pointing to a database table for display.

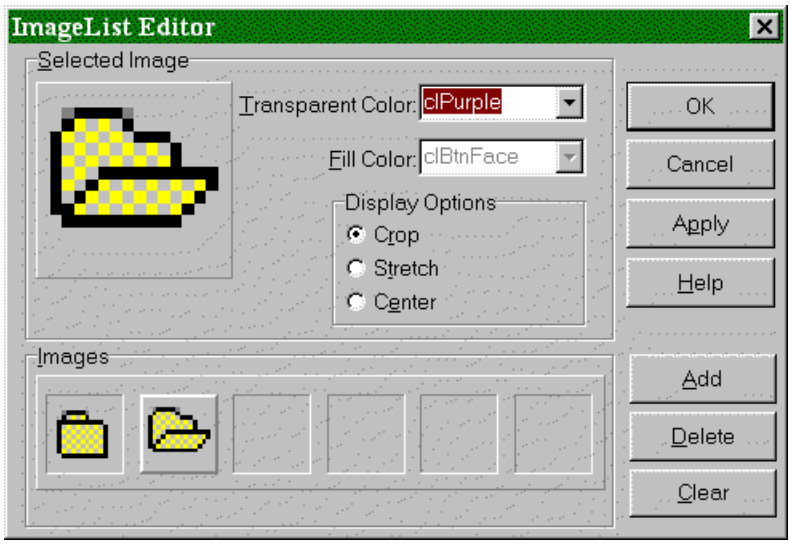
Here we'll create some *fun* data to go with our TTreeView, just to show what's possible, and in the next step we'll save this information out, and then when we load our TTreeView back, we'll re-create our *fun* data.

So what is our fun data ? Well, each node we create is going to have a Form associated with it. The TTreeView will become our method of tracing these forms, (that is we wont be able to get at them without accessing the TTreeView telling us where they are), and we'll do a small amount of manipulation on the form, before saving the data.

We're going back to our NodeJustAdded function for this, again showing why we created the separate function earlier, (it's now going to be very important:).

First let's populate our ImageList, which will let us display a graphic for our nodes. Double click on the ImageList, which brings up the image list editor,

Choose Add, and you get a file dialog. Assuming you installed CBuilder and haven't deleted them, you'll have a directory below CBuilder called "Images", which itself has a directory called "Buttons". The two bmp's we wish to add are fldrshut.bmp and fldropen.bmp, the ImageList editor should now look like this,



Click OK to close the editor, and then add the following in our NodeJustAdded function,

```
//---------------------------------------------------------------------------
void TForm1::NodeJustAdded(TTreeNode *NewNode)
{
NewNode->Text = CaptionCheck("New Folder", NewNode);
FView->AlphaSort();

if(NewNode->Parent)
      {
      NewNode->Parent->Expand(false);
      }

NewNode->ImageIndex = 0;
NewNode->SelectedIndex = 1;
NewNode->StateIndex = -1;

NewNode->EditText();
}
//---------------------------------------------------------------------------
```

If you run the application now, you'll see we get a folder open or closed depending on whether the node is selected or not. The StateIndex is to display a second glyph to the left of the one we're using. I'm sure there's a use for it, but I haven't found it yet, so -1 means it's ignored, and no space is given for it.

Okay let's have some fun. At the end of the above routine, add the following line.

```
NewNode->EditText();
NewNode->Data = new TForm(this);
}
```

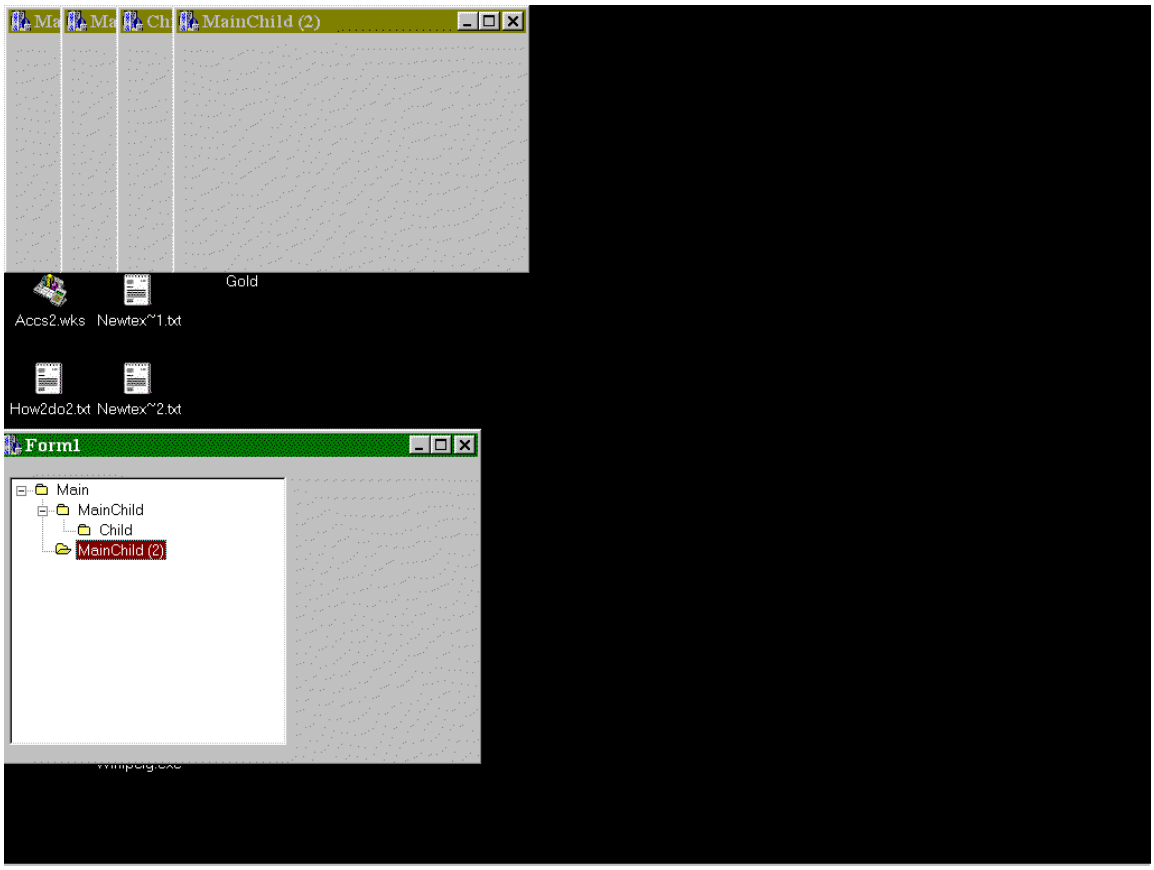
What we're doing here is dynamically creating a form, and "*tying it*" to our node. It doesn't do much, until you add the following to *decode* the pointer, which we'll do by using the F12 key, though it could be anything you like. So add the following in our OnKeyUp handler,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FViewKeyUp(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
switch(Key)
```

```
        {
        case VK_F2:
                RenameNodeClick(Sender);
                break;
        case VK_F12:
                if(FView->Selected)
                        {
                        reinterpret_cast <TForm *> (FView->Selected->Data)-
                >Visible = true;
                        reinterpret_cast <TForm *> (FView->Selected->Data)-
                >BringToFront();
                        reinterpret_cast <TForm *> (FView->Selected->Data)->Left =
                (50 *
                                                FView->Selected->AbsoluteIndex);
                        reinterpret_cast <TForm *> (FView->Selected->Data)-
>Caption =
                                                FView->Selected->Text;
                        Form1->BringToFront();
                        FView->SetFocus();
                        }
                break;
        default:
                //ignore:)
                break;
        }//end switch
}
//---------------------------------------------------------------------------
```

A small note, when you type the above in, don't break the lines:) .

Okay there's some information there. Before we dissect it, run the application, and assuming you've typed the above in correctly, add the following. Add a root node, called "Main". Add a child to it, "MainChild", another, "MainChild (2)", and then add a Child to MainChild, called "Child".. Then click on the "Main" node and press F12, a form appears. Go through each of the other nodes pressing F12 in turn and you should get a display similar to the following,



Okay, (sorry for the poor screenshot, but size:). If you now select Main again and press F12, the main form comes to the front, before you're put back into the TTreeView. Clever ?

Well there's more. Rename node "MainChild (2)" to "NewName", select it and press F12 again. You'll see the caption of the form change. Remember, the node holds the pointer to our form, so when we change something to do with the node, we can have it change something in our Data.

So how does it work. Well the assigning of a component, or anything that can be a pointer is straightforward. The frightening lines will be these,

```
if(FView->Selected)
      {
```

basically checks to make sure we're not pointing at NULL, (which would throw an access violation),

```
reinterpret_cast <TForm *> (FView->Selected->Data)->Visible = true;
reinterpret_cast <TForm *> (FView->Selected->Data)->BringToFront();
reinterpret_cast <TForm *> (FView->Selected->Data)->Left = (50 * FView-
>Selected->AbsoluteIndex);
reinterpret_cast <TForm *> (FView->Selected->Data)->Caption =    FView-
>Selected->Text;
```

is the complex stuff,

```
Form1->BringToFront();
FView->SetFocus();
}
```

which simply bring Form1 back to the front and refocuses us on the TTreeView, but that complex stuff.

The reinterpret_cast <TForm *> (FView->Selected->Data) simply gives us a pointer of type TForm. When we assign a pointer to the Data property, we are simply assigning a pointer, not a *typed* pointer. Although we create the pointer as type TForm, *(the line, **new** TForm(this) , which allows the compiler to allocate memory and create the form)*, all our TTreeView stores is a pointer, but doesn't carry any size information with it. Therefore, when we read that pointer we have to tell the compiler what type it is, (the reinterpret_cast <TForm *>), part. This tells the compiler to interpret the pointer as if it were a pointer to an object of TForm, (which in fact it is:). You will cause exceptions or at the very least memory leakage's if you try and reinterpret a pointer with the wrong type, so my advice is always use the same type to de-allocate as you used to allocate.

As the above reinterpret_cast produces a pointer of type TForm, we can then use any property of a Form, which we then do. When you close the application, note that **ALL** the forms are closed. This is because we gave them the **this** parameter when we created them, and fortunately for us, CBuilder looks after destroying them when we destroy their parent.

Although we don't need to worry about an application close, we do need to look after a node deletion, (and sub-node deletion). Rather than putting the code to delete the form in our call to delete node, we'll use the OnDeletion event of the TTreeView, which is called whenever a node is deleted. (*The reason we don't use this for checking for children is that each sub-node with children from our selected node would fire the check, users would be unhappy)*

Place the following code in the OnDeletion event handler,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FViewDeletion(TObject *Sender, TTreeNode *Node)
{
//a node has been deleted,  we need to remove the associated data.
delete reinterpret_cast <TForm *> (Node->Data);
}
//---------------------------------------------------------------------------
```

And that's it. If you remember, destroying a form will destroy all associated components. The same logic can be applied to any VCL component you choose to use as your data. *(Other methods to look out for include Release(), free, Close(), Destroy())*

Please ensure you understand the above. Although I'm now finding it quite easy, (honest if you've understood the above), it can take a little getting used to. As you'll see later when we tidy our application up, the use of the node and Data pointer is very powerful, allowing us to do many things.

I hope that's clear, but play with it. Get used to using the Data property. You should also note that you can use things which we're not used to using with pointers. The following uses an AnsiString,
to store,
FView->Selected->Data = new AnsiString("This is stored as a pointer")
and to read,
AnsiString Temp;

*Temp = *reinterpret_cast<AnsiString *> (FView->Selected->Data);*

or even a TStringList,
to store,

FView->Selected->Data = new TStringList;
and to read,
AnsiString Temp;
*Temp = reinterpret_cast <TStringList *> (FView->Selected->Data)->Strings[x];*

The second thing to note about what we've done is that we've used a simple TForm, which has very little set by default. In the real world, you would normally create a FormClass of your own, with at least some of the parameters set, or components packaged within your class.

Just be aware that you must match the reinterpret type with the allocation type.

Step Eight: Storing the data

Oh Boy!!,

And this is where the real fun starts:(As we've already seen, there is a TTreeView Save/Load from file. The problem with this is that it only loads the tree, and does nothing about our links to the data. If you guaranteed that the node's caption was unique, this wouldn't be too much of a problem, but your users wouldn't like the fact that they could only call one node "New Folder" anywhere within their tree structure, (have a look at your own use of Explorer for that:).

So we need to do some thinking and come up with our own method for storing the information. The one I use here is my own way of doing it, (Simplest first:). When the user saves the tree, we'll get the node "path", and store it to a TStringList, along with options, which we then save via the TStringList, and load in again and decode. *I'm sure there are better ways of streaming to and from a file, however, this is my way:)*.

Okay, to do this we have to look at each node, and treat it as an entity. We'll save each node in turn, and choose some information to go with it, (by way of example). So how to do it. First we need to think what we wish to save.

Each node has some common properties. A path, and that's all we need to know !. If you think about it, everything else we have is available for us when we reload it. Okay, that's not really enough, so we'll add two things for the hell of it. Firstly we'll add whether the node is expanded or not. This will actually tell us much more. If a node is a root node, it will be visible. A child's parent determines whether it is visible or not. If a child's parent is expanded, then the child is visible, etc. The second piece of information will be the form's colour, (that is the associated data's colour). At present all the forms are the same colour, but we'll add a function to change the colour and then save that when we save the tree. (Now you know what the colour dialog was for:).

Before we start, we'll look at our file format. As I mentioned I've chosen to save things via a TStringList, which means we need to delimit each object. The format I'll use is as follows;

NODESTART
path
expanded
colour

When we read the TStringList back in, each node will be delimited by the NODESTART portions, and we can then get the rest of the information we want. (Actually, I don't use the NODESTART at all when reading things back in, but it does make it easier to edit the text file manually should we wish to do so:)

Okay, let's start. So we can change the colour of each form, add the following,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FormColourClick(TObject *Sender)
{
//let's set up the colour of the form associated with this node.
if(FColor->Execute())
      {
      reinterpret_cast <TForm *> (FView->Selected->Data)->Color = FColor-
>Color;
      }
}
//---------------------------------------------------------------------------
```

which is quite straightforward, now,

Bring back our SaveOurTree function and add the code in red,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::SaveOurTreeClick(TObject *Sender)
{
TStringList *NodeList;
TTreeNode *TempNodePointer;
String Path;
//save out tree out

NodeList = new TStringList;
if(FView->Items->GetFirstNode())
      {
      if(SaveTree->Execute())
            {
            TempNodePointer = FView->Items->GetFirstNode();
            while(TempNodePointer)
                  {
                  Path = GetNodePath(TempNodePointer);
                  NodeList->Add("NODESTART");
                  NodeList->Add(Path);
                        NodeList->Add((String)(int)(TempNodePointer-
      >IsVisible));
                  NodeList->Add(reinterpret_cast <TForm *> (TempNodePointer-
            >Data)
                                                                  ->Color);
                  TempNodePointer = TempNodePointer->GetNext();
                        }
            NodeList->SaveToFile(SaveTree->FileName);
            }
      }
else
      {//there's nothing to save,  it's up to you:)
      }

delete NodeList;
}
//---------------------------------------------------------------------------
```

Right. Above you have a function that first asks if there are any nodes in the tree. If there aren't any nodes, it assumes that we don't want to save them, (nothing to do:). If there is something to save we fire up the save dialog, and assuming we get a positive response, start to build our TStringList. We move to the top of the overall tree structure, (GetFirstNode()), and then recurse through the tree finding the next node in the structure, whatever it's level, visibility etc.

This uses the GetNext() function, which is a function of each node. The functions available for locating nodes are,

TTreeView->Selected,	: returns the currently selected node
TTreeView->Items->GetFirstNode()	: returns the first node in the tree

TTreeNode->GetNext() : returns the next node in the tree
TTreeNode->GetFirstChild() : returns the first child of a node, (or null if no children)
TTreeNode->GetLastChild() : returns the last child of a node, (or null if no children)
GetNextChild(TTreeNode *) : returns the next child after the calling node
TTreeNode->GetNextSibling() : returns the next node at the same level with same parent
TTreeNode->GetNextVisible() : returns the next visible node
TTreeNode->GetPrev() : returns the previous node
GetPrevChild(TTreeNode *) : returns the previous child to the calling node
TTreeNode->GetPrevSibling() : returns the previous node at the same level, (same parent)
TTreeNode->GetPrevVisible() : returns the previous visible node.

With those functions it's pretty well easy to do anything:). Again, play with them. If you find the compiler saying the function doesn't exist, try it with a lowercase g, which seems to be a popular header file error;).

So, we use the GetNext, which from the above list returns the next node in the overall tree structure to the calling node. Note the difference between GetNext and GetNextVisible. We want to know all the nodes, whether they are visible or not, but you may wish only to save the visible one's, or only a partial tree, the power is yours.

Now into building our actual tree. We use a function we'll write in a moment, (GetNodePath), and then store the path, force a Boolean to an int, (so we can read it back in), using the IsVisible property of the TTreeNode, before finally storing the color property of the associated data form, (we do more on colour in a moment). Once we hit the bottom of the tree, (NULL), we save our tree structure.

Ensure you follow that, cause reading it in is harder;).

We need to create the following function, GetNodePath, ourselves. Enter the following into the header file, (actually two functions, saves us doing it later:).

```
private:     // User declarations
       void NodeJustAdded(TTreeNode *NewNode);
       void ShowNodeCheck(TTreeNode *Node, bool YupNah);
       String CaptionCheck(String Name, TTreeNode *CheckNode);
       String GetNodePath(TTreeNode *ThisOne);
public:            // User declarations
```

The GetNodePath looks like this,

```
//---------------------------------------------------------------------------
String TForm1::GetNodePath(TTreeNode *ThisOne)
{
//take this node and work out it's parent and the path.
// path format = path\node
TTreeNode *TempNode;
String Temp, ReturnString;

TempNode = ThisOne;
ReturnString = TempNode->Text;
Temp = "";

while(TempNode->Parent)
       {
       Temp = "\\" + ReturnString; //thisnode
       TempNode = TempNode->Parent;
       ReturnString = TempNode->Text + Temp; //parentnode
       }
```

```
return ReturnString;
}
//---------------------------------------------------------------------------
```

Now wait a minute, that seems quite simple, and indeed it is. We'll produce a path similar to that used by DOS, (e.g. parent\child\child). Note the lack of a slash at the start and end of the string, which helps us to simply read it at the end. Again we recurse the tree to do this, going up instead of down.

The simplest part is this. We set the return string to the node's text. If the node has no parent, we return the string "nodetext", in other words a root node. If the node has a parent, we add our delimiter, set the pointer to the parent, then add the parent's text to the beginning of our text, and so on.

Run the application, add some nodes and children, collapse some tree's and save the tree. Have a look at the text file produced, and then close the application, but keep the text file open, because…

Next let's look at our LoadOurTree function, and this will take some explaining:(. Here is the function,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::LoadOurTreeClick(TObject *Sender)
{
TStringList *NodeList, *PathList;
String TempNodePath, Temp;
TTreeNode *RootNode;
int x, y;
bool Found;
//load our tree in.

NodeList = new TStringList;
PathList = new TStringList;

//This may sound daft,  but we should clear our current tree,  even if
empty:).

FView->Items->Clear();
FView->Items->BeginUpdate();

if(LoadTree->Execute())
      {
      NodeList->LoadFromFile(LoadTree->FileName);
      x = 0;
      while(x < NodeList->Count-3)
            {//note,  we will look after parts of X:))  We know that a node
      should be delimited by
            //a NODESTART marker.  We could check,  but hey let's risk it:)
            Found = false;
            TempNodePath = NodeList->Strings[x+1];
            //build a local list of our path
            PathList->Clear();
            y = 0;
            while(TempNodePath.LastDelimiter("\\"))
                  {//our string has a delimiter.
                  if(TempNodePath.IsDelimiter("\\", y))
                        {//a node in our path
                        PathList->Add(TempNodePath.SubString(1, y-1));
                        Temp = TempNodePath;
                        Temp.Delete(1, y);
                        TempNodePath = Temp;
                        y = 0;
```

```
                        }
                  else
                        {
                        y++;
                        }
                  }
            //add the last "part" of the path
            PathList->Add(TempNodePath);
            //okay we now have a TStringList in which each string is part of
      our nodes path,
            //the last item in the list is OUR NODE:)
            //we have to start by finding the first part of our path.
            RootNode = FView->Items->GetFirstNode();

            for(int z=0; z < PathList->Count-1; z++)
                  {
                  Found = false;
                  while(!Found && RootNode)
                        {
                        if(RootNode->Text == PathList->Strings[z])
                              {//we've found this part of the tree
                              if(RootNode->HasChildren && (z != PathList-
                        >Count-2))
                                    {
                                    RootNode = RootNode->getFirstChild();
                                    }
                              Found = true;
                              }
                        else
                              {//get the next child.
                              RootNode = RootNode->getNextSibling();
                              }
                        }
                  }
            if(!Found)
                  {//We didn't find the a parent,  so this must be a root
node
                  RootNode = FView->Items->Add(NULL,
                                    PathList->Strings[PathList->Count-1]);
                  }
            else
                  {//RootNode should now be the parent node
                  RootNode = FView->Items->AddChild(RootNode,
                              PathList->Strings[PathList->Count-1]);
                  }
            RootNode->ImageIndex = 0;
            RootNode->SelectedIndex = 1;
            RootNode->StateIndex = -1;
            //We set the expanded property in a moment once everything is
loaded ?
            if((bool)NodeList->Strings[x+2].ToInt() && RootNode->Parent)
                  {
                  RootNode->Parent->Expand(false);
                  }
            //now add a form
            RootNode->Data = new TForm(this);
            //now set up the display of the form
            reinterpret_cast <TForm *> (RootNode->Data)->Visible = true;
            reinterpret_cast <TForm *> (RootNode->Data)->Caption = RootNode-
      >Text;
            reinterpret_cast <TForm *> (RootNode->Data)->Left = (50 *
                                                RootNode->AbsoluteIndex);
            reinterpret_cast <TForm *> (RootNode->Data)->BringToFront();
            //and finally read in our form colour and set that.
            reinterpret_cast <TForm *> (RootNode->Data)->Color =
                                          (TColor)NodeList-
                                    >Strings[x+3].ToInt();
```

```
                x = x+4;
                }
        }
else
        {//we do nothing
        }
Form1->BringToFront();
FView->Items->EndUpdate();

delete NodeList;
delete PathList;
}
//---------------------------------------------------------------------------
```

Yuck ! Now who says CBuilder doesn't involve real coding and logic then !

I'll attempt to break that up into easy to digest sections,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::LoadOurTreeClick(TObject *Sender)
{
TStringList *NodeList, *PathList;
String TempNodePath, Temp;
TTreeNode *RootNode;
int x, y;
bool Found;
//load our tree in.

NodeList = new TStringList;
PathList = new TStringList;
```

Easy this bit. Declaration of variables. We'll use two TStringList arrays, one to read in from the file and hold the information whilst we scroll through, "NodeList", and the next to help us reconstruct our individual node's path, "Path List".

The other variables are used for holding things together at some point in the following, and the only one of interest is "Found", a Boolean which will tell us when to add a node to our tree.

```
//This may sound daft,  but we should clear our current tree,  even if
empty:).

FView->Items->Clear();
FView->Items->BeginUpdate();
```

Good practice. The "BeginUpdate" function is an interesting one. We're about to add lot's of nodes to our tree, and in some instances it could be many hundreds, (Explorer again:). All we're doing here is saying to the TTreeView is that it shouldn't bother redrawing anything in the node until we've finished, (yup, we tell it later:).

```
if(LoadTree->Execute())
        {
```

If the user wishes to load,

```
        NodeList->LoadFromFile(LoadTree->FileName);
        x = 0;
```

Load the file they specify and then set x to 0 for our while loop. If you remember earlier when we wrote the save routine, we used "NODESTART" to delimit our nodes. Although I haven't done it here, the first thing we should check for is that the format we want is a file which matches our

format. This would be that NodeList->Strings[0] AND NodeList->Strings[4] = “NODESTART”, and you could step through the whole list to find out there were no errors if you wished. To save your fingers earlier, we didn’t do that, but it would be simple to implement should you choose.

```
while(x < NodeList->Count-3)
	{// note,  we will look after parts of X:))  We know that a node
	 // should be delimited by
	 // a NODESTART marker.  We could check,  but hey let's risk it:)
	Found = false;
	TempNodePath = NodeList->Strings[x+1];
	//build a local list of our path
	PathList->Clear();
	y = 0;
```

Here we start. We are going to handle the x counter for our loop. Later we increment the counter in jumps of 4, (“NODESTART” to “NODESTART”). Basically our terminate is when x is greater than the number of strings in the list - 3. When it’s Count-4, we still have a node to go. As we use x+1 to x+3 to access certain things, if x > Count-3, we’ve passed our last node, and we don’t wish to throw array index violations.

We then force “Found” to false, and select the first “path” in our file, (x = NODESTART, x+1 = path), before we clear our “PathList” string so we know it’s empty and set y to 0 for our next loop.

```
	while(TempNodePath.LastDelimiter("\\"))
		{//our string has a delimiter.
		if(TempNodePath.IsDelimiter("\\", y))
			{//a node in our path
			PathList->Add(TempNodePath.SubString(1, y-1));
			Temp = TempNodePath;
			Temp.Delete(1, y);
			TempNodePath = Temp;
			y = 0;
			}
		else
			{
			y++;
			}
		}
	//add the last "part" of the path
	PathList->Add(TempNodePath);
```

This section takes our “path”, breaks it down into individual strings. If you look at one of the paths you saved, and follow through the logic above, you’ll get the hang of it:)

```
	// okay we now have a TStringList in which each string is part of
	// our nodes path, the last item in the list is OUR NODE:)
	//we have to start by finding the first part of our path.
	RootNode = FView->Items->GetFirstNode();
```

And this is the important bit, adding the nodes. Our “PathList” now holds each individual part of the path to our node, “root node”, “parent node”, “OUR NODE”. To build our tree correctly we should start by finding if our root node exists. To do this, we need to start at the very top of the TTreeView.

```
	for(int z=0; z < PathList->Count-1; z++)
		{
		Found = false;
		while(!Found && RootNode)
			{
			if(RootNode->Text == PathList->Strings[z])
				{//we've found this part of the tree
				if(RootNode->HasChildren && (z != PathList-
```

```
                        >Count-2))
                                {
                                RootNode = RootNode->getFirstChild();
                                }
                        Found = true;
                        }
                else
                        {//get the next child.
                        RootNode = RootNode->getNextSibling();
                        }
                }
        }
```

And then we cycle through our “PathList”, and a major point to note here. This works because it relies on the order of the file we loaded. When we save our tree, we start at the top and work down. When we load our tree, we are *assuming” that the first node in our file is an overall root. Thus, when each child node is parsed through the above routine, it’s parent will already exist. If you don’t understand the above, copy the file you saved, and edit the first path so that it isn’t a root node, and you’ll see the mess it makes, it wont crash, it’ll just put nodes all over the tree.

So why does it work. Well, if the tree is empty, “RootNode” will be NULL, “Found”, false. Therefore the above while loop wont execute, and it’ll carry on to the next bit, (adding a node). If it comes through again at this point with a Child node, it will be added as a root.

If there is a root node, and it’s caption doesn’t equal the part of the path we’re trying to locate, we get the next node until we find one that matches our part of the path. If we find a node that matches our part of the path, we don’t wish to add it, (it’s already there:), so we set “Found” to true, and carry on through our “PathList” until we find a part of the path that doesn’t exist. When this occurs,
we need to add a node.

Thus, if “Found” is false,

```
        if(!Found)
                {//We didn't find the a parent,  so this must be a root
node
                RootNode = FView->Items->Add(NULL,
                                PathList->Strings[PathList->Count-1]);
                }
        else
                {//RootNode should now be the parent node
                RootNode = FView->Items->AddChild(RootNode,
                        PathList->Strings[PathList->Count-1]);
                }
```

We didn’t find the parent to our node, so we should add it as a root node, and if “Found” is true, we have found the parent node to the one we wish to add, so we add a node to it.

Please make sure you follow that, it really will enhance your understanding of the way in which a tree works with it’s nodes. If you’re unsure, follow it through in the debugger.

```
        RootNode->ImageIndex = 0;
        RootNode->SelectedIndex = 1;
        RootNode->StateIndex = -1;
```

Set up the state images for the node we’ve just added.

```
        // We set the expanded property in a moment once everything is
        // loaded ?

        if((bool)NodeList->Strings[x+2].ToInt() && RootNode->Parent)
```

```
                    {
                    RootNode->Parent->Expand(false);
                    }
```

And then expand the node if it was saved expanded. If we saved true, (anything other than 0 will be interpreted as true), and the Child has a parent, we should make sure it's visible by expanding it's parent by one level.

```
                //now add a form
                RootNode->Data = new TForm(this);
                //now set up the display of the form
                reinterpret_cast <TForm *> (RootNode->Data)->Visible = true;
                reinterpret_cast <TForm *> (RootNode->Data)->Caption =
                      RootNode->Text;
                reinterpret_cast <TForm *> (RootNode->Data)->Left = (50 *
                      RootNode->AbsoluteIndex);
                reinterpret_cast <TForm *> (RootNode->Data)->BringToFront();
                //and finally read in our form colour and set that.
                reinterpret_cast <TForm *> (RootNode->Data)->Color =
                      (TColor)NodeList->Strings[x+3].ToInt();
```

And this adds the form information we saved. Now we only saved the colour, but you could easily save the position and other data if you wished. I'll not go into the casting of the TColor, suffice to say it is needed to correct the int to the position within the TColor enum.

```
                x = x+4;
                }
        }
```

And if all that's worked, we simply increment our counter by 4.

```
else
      {//we do nothing
      }
Form1->BringToFront();
FView->Items->EndUpdate();
```

Bring the form we just created to the front, before allowing the TTreeView to update it's display.

```
delete NodeList;
delete PathList;
}
//---------------------------------------------------------------------------
```

And wasn't that easy ? Actually yes. Once you've become used to recursing through a tree, the above will make a lot more sense than it will if this is your first go. The exercise of the save and load should show you how easy creating your own file handling for the TTreeView is, and the flexibility and power it can bring to your applications.

I had a question from someone, (lost it in another Outlook mail debacle), who wished to show more than one customer account record, (and other linked forms), at one go. Easy when you had just one customer on the phone, but when another rang up you had to bring up two sets of information, and the screen became, "confused". Using a TTreeView would allow him to create a root for each customer, add nodes for each form, and when you wish to switch between customers, you just click on the root node and all their forms jump to the front.

Which brings us onto,

Step Nine : Tidy it all up.

I was going to use this section simply to add the dragging and dropping of nodes around the tree,

but as Alan D. Mills covers such things in his tutorial, “Basic Drag & Drop”, available from the site where he covers the moving of nodes within a TTreeView, (well worth reading:), so all we’ll do is tidy up our little application so it does the following. I wont cover them in detail, you should now be used to what’s happening, so mail me, (jon.jenkinson@mcmail.com), if you’re stuck with any of the following, or any other comments.

a) Shows and Brings a form to the front when you click on a node

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FViewClick(TObject *Sender)
{
ShowNodeCheck(FView->Selected, true);
//added below:)
}
```

and change our onkey up handler,

```
case VK_F12:
       if(FView->Selected)
              {
              ShowNodeCheck(FView->Selected, true);
              }
       break;
```

to use the following function

```
//---------------------------------------------------------------------------
void TForm1::ShowNodeCheck(TTreeNode *Node, bool YupNah)
{
//okay,  we only wish to process anything,  if
//the node's visibility is the not the same as the forms
//visibility.
if(YupNah)
       {
       reinterpret_cast <TForm *> (Node->Data)->Visible = true;
       }
else
       {
       reinterpret_cast <TForm *> (Node->Data)->Visible = false;
       }
reinterpret_cast <TForm *> (Node->Data)->BringToFront();
reinterpret_cast <TForm *> (Node->Data)->Left = (50 * Node->AbsoluteIndex);
reinterpret_cast <TForm *> (Node->Data)->Caption = Node->Text;
Form1->BringToFront();
FView->SetFocus();
}
//---------------------------------------------------------------------------
```

b) Updates a forms caption when the form is renamed,

Add the following line to the end of the OnEdited function.

```
S = CaptionCheck(S, Node);

reinterpret_cast <TForm *> (Node->Data)->Caption = S
}
```

a) Allows us to show/hide forms when we change the visibility of the associated node,

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FViewCollapsed(TObject *Sender, TTreeNode *Node)
{
TTreeNode *TempNode;
if(!FView->IsEditing())
       {
       TempNode = FView->Items->GetFirstNode();

       while(TempNode)
```

```
                {
                if(TempNode->Data)
                        {
                        ShowNodeCheck(TempNode, TempNode->IsVisible);
                        }
                TempNode = TempNode->GetNext();
                }
        }
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FViewExpanded(TObject *Sender, TTreeNode *Node)
{
TTreeNode *TempNode;

if(!FView->IsEditing())
        {
        TempNode = FView->Items->GetFirstNode();

        while(TempNode)
                {
                if(TempNode->Data)
                        {
                        ShowNodeCheck(TempNode, TempNode->IsVisible);
                        }
                TempNode = TempNode->GetNext();
                }
        }
}
//---------------------------------------------------------------------------
```

and the following in the NodeJustAdded function,

```
...
NewNode = FView->Items->GetFirstNode();

while(NewNode)
        {
        if(NewNode->Data)
                {
                ShowNodeCheck(NewNode, true);
                }
        NewNode = NewNode->GetNext();
        }
}//this is the end of the function
```

And there we have it. According to my edit window, 426 lines of code, (including rem statements:). You now have a skeletal program which you can use to handle multiple form occurrences at run time, as well as a good understanding of how TTreeView works.

If you have any suggestions about how to improve this tutorial, good or bad, mail me, jon.jenkinson@mcmail.com and we'll try and incorporate them in the next version.

Many thanks for your time.
Jon

A Final Note

No liability is accepted by the author(s) for anything which may occur whilst following this tutorial

Contacts :
Main Site : forgot@mcmail.com
Newspages : bitsnews@mcmail.com
Comments : comments@mcmail.com
Section Editors:
Tutorials : Kris Erickson, kson@istar.ca
Components : Alan D. Mills, millsad@tgis.co.uk
Others bits: Jon Jenkinson, jon.jenkinson@mcmail.com
Author:
****Your email if you want****

Legal Stuff

You Cannot,

Under any circumstances alter this document in any way without the express permission of the author or "The Bits...". Contact us and we'll look into your suggestions. If we agree with what you say, we will include you as a co-author of the document.

You Cannot,

*Receive **any** monies for this document without the prior consent of the author.*

You Cannot,

Take any of the ideas presented here and produce a product for distribution without the express written consent of the author. However, you may take what you learn here and produce a commercial product based on your own interpretation of your newly gained knowledge.