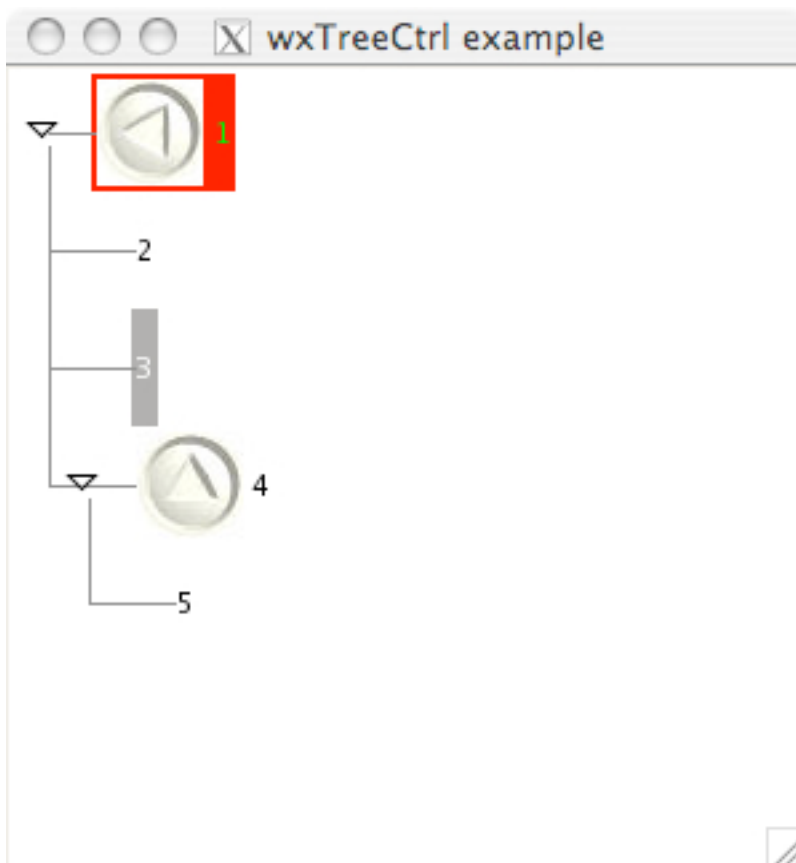


## Table of Contents

Working with the wxTreeCtrl widget in RwxWidgets ..... 1

# Working with the wxTreeCtrl widget in RwxWidgets

In this short example, we will look at how we use the wxTreeCtrl within the R bindings. The resulting GUI looks like



This image was "grabbed" on a screen with a resolution of 2560 x 1600 . It is pixelated in the PDF document.

We start by loading the RwxWidgets package.

```
library(RwxWidgets)
wxInit()
wxTR_HAS_BUTTONS = 0x0001
wxNullBitmap = getWxNullBitmap()
```

Next, we create the top-level window.

---

```
f = RFrame("wxTreeCtrl example", size = c(300, 300))
```

And now we start with the `wxTreeCtrl`. We create the top-level tree widget and then we add the different items. The top-most node is added via the function `wxTreeCtrl_AddRoot`.

```
tree = wxTreeCtrl(f, wxID_ANY)
top = tree$AddRoot("1")
```

Then we can add its children with `wxTreeCtrl_AddItem`. We specify the parent item and a label. The parent is `top`.

```
tree$AppendItem(top, "2")
tree$AppendItem(top, "3")
n = tree$AppendItem(top, "4")
```

The return value for each of these calls is a `wxTreeItemId` and this allows us to identify the items in subsequent calls. When adding the first two nodes (2 and 3), we ignored this return value. But for node "4", we assigned the value to an R variable - `n`. And now we can use this to add children to that node and not to the top-most node.

```
tree$AppendItem(n, "5")
```

In this case, the node "5" is a child of "4", not "1".

In addition to labels which are displayed in the widget, we can also associate an image with a particular node. We first load one or more images and put them into a `wxImageList`. Then we associate this image list with the `wxTreeCtrl` widget. And then, when creating the node or in a separate operation after the node is in the widget, we can specify which entry in the image list is associate with a particular node. We'll load the JPEG images in `$R_HOME/doc/html` directory of the R installation and put each of them into an image list.

```
dir = paste(Sys.getenv("R_HOME"), "doc", "html",
            sep = .Platform$file.sep)
imageNames = list.files(dir, "(left|right|up)\\.jpg$",
                        full.names = TRUE)
images = wxImageList()
sapply(imageNames,
       function(filename)
         images$Add(wxBitmap(filename, wxBITMAP_TYPE_JPEG)))
```

Now, since we have the `wxTreeItemId` handles for the top-level node and node "4" in the variables `top` and `n`, let's just work with these. Let's use the first image for the top-level node and the 3rd image for `n`. Note that the image list uses 0 for the first element, i.e. uses zero-based indexing, and we currently must use that convention in R. (This may change.)

```
tree$AssignImageList(images)
tree$SetItemImage(top, 0)
tree$SetItemImage(n, 2)
```

Make certain that you are calling these methods on the `wxTreeCtrl` and not on the `wxTreeItemId`. Also, if you forget to associate the image list with the `wxTreeCtrl`, you won't see the images. And there is a slight difference between `AssignImageList` and `SetImageList`. The former hands over (memory) management of the image list to the `wxTreeCtrl` widget. When that widget is destroyed, the image list will also be garbage collected. The `SetImageList` method leaves control with us, so we have to destroy it and are in charge of its longevity. And if we destroy it before the widget is done using it, bad things can happen.

And we can also set both the foreground and background colors for a node.

```
tree$SetItemBackgroundColour(top, "red")
```

---

```
tree$SetItemTextColour(top, "green")
```

Note that to see the colors take effect, you need to expand the root node and then select a different node. Otherwise the fact that this node is selected means that the selection colors are in effect.

Note that we cannot create multiple top-level nodes.

```
o = tree$AddItem(NULL, "6")
```

Now we will turn to handling elementary events on the wxTreeCtrl widget. Note that we use the "full" version of the event identifier wxEVT\_COMMAND\_TREE rather wxEVT\_TREE. This is because the C++ pre-processor does some tricks to turn wxEVT\_TREE into wxEVT\_COMMAND\_TREE. For now, we must use the full form in R.

```
h =  
function(ev, type) {  
  cat("expanded", tree$GetItemText(ev$GetItem()), "\n")  
  print(.Call("R_wxTreeEvent_mine", ev, tree))  
}  
tree$AddCallback(wxEVT_COMMAND_TREE_ITEM_EXPANDED, h, "expanded")
```

## Note

The wxTreeItemId was causing a little bit of a problem. It is defined as a class in treebase.h. Yet in the actual code that returns these objects, we have a pointer to a wxGenericTreeItem being returned and so these are not on the stack and we could return a reference to these in an external pointer object. However, we cannot "coerce" this type to a void \*. So we use the default copy constructor and create a new instance of wxTreeItemId on the stack. When we write the C++ code by hand to use such objects, it is quite easy to use the wxTreeItemId pointer and not the value (i.e. \*item). Since a reference to the item is the typical parameter, the C++ compiler does not complain but gets the wrong value! However, that is now sorted out (without an warning from the compiler) and so the following event handler does its job.

And the final step in this example is that we display the contents of the frame and run the event loop in a blocking manner (at present).

```
f$Show()  
eloop = wxEventLoop()  
eloop$Run()
```