GTK+ 2.0 Tree View Tutorial

Tim-Philipp Müller

GTK+ 2.0 Tree View Tutorial

by Tim-Philipp Müller

This is a tutorial on how to use the GTK (the GIMP Toolkit) GtkTreeView widget through its C interface.

Please mail all comments and suggestions to <tim at centricular dot net>

A tarball of the tutorial for off-line reading including the example source codes is available here: treeview-tutorial.tar.gz.

There is also a version in PDF format (for easier printing) and the raw docbook XML source document.

This tutorial is work-in-progress. The latest version can be found at http://scentric.net/tutorial/.

Some sections are a bit outdated (e.g. GtkTreeModelFilter has been in Gtk since 2.4), just haven't gotten around to rewrite them or update them. Sorry!

Last updated: September 29th, 2006

Table of Contents

Chapter 1. Lists and Trees: the GtkTreeView Widget

`GtkTreeView` is a widget that displays single- or multi-columned lists and trees. It replaces the old Gtk+-1.2 GtkCList and GtkCTree widgets. Even though `GtkTreeView` is slightly harder to master than its predecessors, it is so much more powerful and flexible that most application developers will not want to miss it once they have come to know it.

The purpose of this chapter is not to provide an exhaustive documentation of `GtkTreeView` - that is what the API documentation is for, which should be read alongside with this tutorial. The goal is rather to present an introduction to the most commonly-used aspects of `GtkTreeView`, and to demonstrate how the various `GtkTreeView` components and concepts work together. Furthermore, an attempt has been made to shed some light on custom tree models and custom cell renderers, which seem to be often-mentioned, but rarely explained.

Developers looking for a quick and dirty introduction that teaches them everything they need to know in less than five paragraphs will not find it here. In the author's experience, developers who do not understand how the tree view and the models work together will run into problems once they try to modify the given examples, whereas developers who have worked with other toolkits that employ the Model/View/Controller-design will find that the API reference provides all the information they need to know in more condensed form anyway. Those who disagree may jump straight to the working example code of course.

Please note that the code examples in the following sections do not necessarily demonstrate how `GtkTreeView` is used best in a particular situation. There are different ways to achieve the same result, and the examples merely show those different ways, so that developers are able to decide which one is most suitable for the task at hand.

1.1. Hello World

For the impatient, here is a small treeview 'Hello World' program (which can also be found in the examples section of the treeview-tutorial.tar.gz tarball).

```
/*
 * Compile with:
 *  gcc -o helloworld helloworld.c `pkg-config --cflags --libs gtk+-2.0`
 *
 */

#include <gtk/gtk.h>

enum
{
  COL_NAME = 0,
  COL_AGE,
  NUM_COLS
} ;


static GtkTreeModel *
create_and_fill_model (void)
{
  GtkListStore  *store;
  GtkTreeIter    iter;

  store = gtk_list_store_new (NUM_COLS, G_TYPE_STRING, G_TYPE_UINT);

  /* Append a row and fill in some data */
  gtk_list_store_append (store, &iter);
  gtk_list_store_set (store, &iter,
                      COL_NAME, "Heinz El-Mann",
                      COL_AGE, 51,
                      -1);

  /* append another row and fill in some data */
  gtk_list_store_append (store, &iter);
  gtk_list_store_set (store, &iter,
                      COL_NAME, "Jane Doe",
                      COL_AGE, 23,
                      -1);

  /* ... and a third row */
```

```
  gtk_list_store_append (store, &iter);
  gtk_list_store_set (store, &iter,
                      COL_NAME, "Joe Bungop",
                      COL_AGE, 91,
                      -1);

  return GTK_TREE_MODEL (store);
}

static GtkWidget *
create_view_and_model (void)
{
  GtkCellRenderer     *renderer;
  GtkTreeModel        *model;
  GtkWidget           *view;

  view = gtk_tree_view_new ();

  /* --- Column #1 --- */

  renderer = gtk_cell_renderer_text_new ();
  gtk_tree_view_insert_column_with_attributes (GTK_TREE_VIEW (view),
                                               -1,
                                               "Name",
                                               renderer,
                                               "text", COL_NAME,
                                               NULL);

  /* --- Column #2 --- */

  renderer = gtk_cell_renderer_text_new ();
  gtk_tree_view_insert_column_with_attributes (GTK_TREE_VIEW (view),
                                               -1,
                                               "Age",
                                               renderer,
                                               "text", COL_AGE,
                                               NULL);

  model = create_and_fill_model ();

  gtk_tree_view_set_model (GTK_TREE_VIEW (view), model);

  /* The tree view has acquired its own reference to the
   *  model, so we can drop ours. That way the model will
   *  be freed automatically when the tree view is destroyed */

  g_object_unref (model);

  return view;
}


int
main (int argc, char **argv)
{
  GtkWidget *window;
  GtkWidget *view;

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (window, "delete_event", gtk_main_quit, NULL); /* dirty */

  view = create_view_and_model ();

  gtk_container_add (GTK_CONTAINER (window), view);

  gtk_widget_show_all (window);

  gtk_main ();
```

```
  return 0;
}
```

Chapter 2. Components: Model, Renderer, Column, View

The most important concept underlying `GtkTreeView` is that of complete separation between data and how that data is displayed on the screen. This is commonly known as Model/View/Controller-design (MVC). Data of various type (strings, numbers, images, etc.) is stored in a 'model'. The 'view' is then told which data to display, where to display it, and how to display it. One of the advantages of this approach is that you can have multiple views that display the same data (a directory tree for example) in different ways, or in the same way multiple times, with only one copy of the underlying data. This avoids duplication of data and programming effort if the same data is re-used in different contexts. Also, when the data in the model is updated, all views automatically get updated as well.

So, while `GtkTreeModel` is used to store data, there are other components that determine which data is displayed in the `GtkTreeView` and how it is displayed. These components are `GtkTreeViewColumn` and `GtkCellRenderer`. A `GtkTreeView` is made up of tree view columns. These are the columns that users perceive as columns. They have a clickable column header with a column title that can be hidden, and can be resized and sorted. Tree view columns do not display any data, they are only used as a device to represent the user-side of the tree view (sorting etc.) and serve as packing widgets for the components that do the actual rendering of data onto the screen, namely the `GtkCellRenderer` family of objects (I call them 'objects' because they are not GtkWidgets). There are a number of different cell renderers that specialise in rendering certain data like strings, pixbufs, or toggle buttons. More on this later.

Cell renderers are packed into tree view columns to display data. A tree view column needs to contain at least one cell renderer, but can contain multiple cell renderers. For example, if one wanted to display a 'Filename' column where each filename has a little icon on the left indicating the file type, one would pack a `GtkCellRendererPixbuf` and a `GtkCellRendererText` into one tree view column. Packing renderers into a tree view column is similar to packing widgets into a `GtkHBox`.

Chapter 3. GtkTreeModels for Data Storage: GtkListStore and GtkTreeStore

It is important to realise what GtkTreeModel is and what it is not. GtkTreeModel is basically just an 'interface' to the data store, meaning that it is a standardised set of functions that allows a GtkTreeView widget (and the application programmer) to query certain characteristics of a data store, for example how many rows there are, which rows have children, and how many children a particular row has. It also provides functions to retrieve data from the data store, and tell the tree view what type of data is stored in the model. Every data store must implement the GtkTreeModel interface and provide these functions, which you can use by casting a store to a tree model with GTK_TREE_MODEL(store). GtkTreeModel itself only provides a way to query a data store's characteristics and to retrieve existing data, it does not provide a way to remove or add rows to the store or put data into the store. This is done using the specific store's functions.

Gtk+ comes with two built-in data stores (models): GtkListStore and GtkTreeStore. As the names imply, GtkListStore is used for simple lists of data items where items have no hierarchical parent-child relationships, and GtkTreeStore is used for tree-like data structures, where items can have parent-child relationships. A list of files in a directory would be an example of a simple list structure, whereas a directory tree is an example for a tree structure. A list is basically just a special case of a tree with none of the items having any children, so one could use a tree store to maintain a simple list of items as well. The only reason GtkListStore exists is in order to provide an easier interface that does not need to cater for child-parent relationships, and because a simple list model can be optimised for the special case where no children exist, which makes it faster and more efficient.

GtkListStore and GtkTreeStore should cater for most types of data an application developer might want to display in a GtkTreeView. However, it should be noted that GtkListStore and GtkTreeStore have been designed with flexibility in mind. If you plan to store a lot of data, or have a large number of rows, you should consider implementing your own custom model that stores and manipulates data your own way and implements the GtkTreeModel interface. This will not only be more efficient, but probably also lead to saner code in the long run, and give you more control over your data. See below for more details on how to implement custom models.

Tree model implementations like GtkListStore and GtkTreeStore will take care of the view side for you once you have configured the GtkTreeView to display what you want. If you change data in the store, the model will notify the tree view and your data display will be updated. If you add or remove rows, the model will also notify the store, and your row will appear in or disappear from the view as well.

3.1. How Data is Organised in a Store

A model (data store) has model columns and rows. While a tree view will display each row in the model as a row in the view, the model's columns are not to be confused with a view's columns. A model column represents a certain data field of an item that has a fixed data type. You need to know what kind of data you want to store when you create a list store or a tree store, as you can not add new fields later on.

For example, we might want to display a list of files. We would create a list store with two fields: a field that stores the filename (ie. a string) and a field that stores the file size (ie. an unsigned integer). The filename would be stored in column 0 of the model, and the file size would be stored in column 1 of the model. For each file we would add a row to the list store, and set the row's fields to the filename and the file size.

The GLib type system (GType) is used to indicate what type of data is stored in a model column. These are the most commonly used types:

- G_TYPE_BOOLEAN
- G_TYPE_INT, G_TYPE_UINT
- G_TYPE_LONG, G_TYPE_ULONG, G_TYPE_INT64, G_TYPE_UINT64 (these are not supported in early gtk+-2.0.x versions)
- G_TYPE_FLOAT, G_TYPE_DOUBLE
- G_TYPE_STRING - stores a string in the store (makes a copy of the original string)
- G_TYPE_POINTER - stores a pointer value (does not copy any data into the store, just stores the pointer value!)
- GDK_TYPE_PIXBUF - stores a GdkPixbuf in the store (increases the pixbuf's refcount, see below)

You do not need to understand the type system, it will usually suffice to know the above types, so you can tell a list store or tree store what kind of data you want to store. Advanced users can derive their own types from the fundamental GLib types. For simple structures you could register a new boxed type for example, but that is usually not necessary. G_TYPE_POINTER will often do as well, you will just need to take care of memory allocation and freeing yourself then.

Storing `GObject`-derived types (most `GDK_TYPE_FOO` and `GTK_TYPE_FOO`) is a special case that is dealt with further below.

Here is an example of how to create a list store:

```
GtkListStore *list_store;

list_store = gtk_list_store_new (2, G_TYPE_STRING, G_TYPE_UINT);
```

This creates a new list store with two columns. Column 0 stores a string and column 1 stores an unsigned integer for each row. At this point the model has no rows yet of course. Before we start to add rows, let's have a look at the different ways used to refer to a particular row.

3.2. Refering to Rows: GtkTreeIter, GtkTreePath, GtkTreeRowReference

There are different ways to refer to a specific row. The two you will have to deal with are `GtkTreeIter` and `GtkTreePath`.

3.2.1. GtkTreePath

A `GtkTreePath` is a comparatively straight-forward way to describe the logical position of a row in the model. As a `GtkTreeView` always displays *all* rows in a model, a tree path always describes the same row in both model and view.

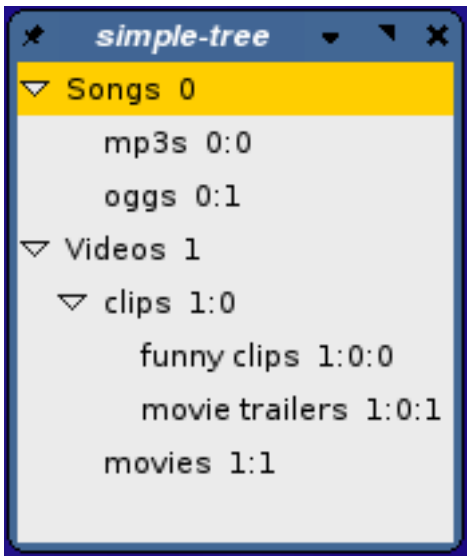

Figure 3-1. Tree Paths

The picture shows the tree path in string form next to the label. Basically, it just counts the children from the imaginary root of the tree view. An empty tree path string would specify that imaginary invisible root. Now 'Songs' is the first child (from the root) and thus its tree path is just "0". 'Videos' is the second child from the root, and its tree path is "1". 'oggs' is the second child of the first item from the root, so its tree path is "0:1". So you just count your way down from the root to the row in question, and you get your tree path.

To clarify this, a tree path of "3:9:4:1" would basically mean *in human language* (attention - this is not what it really means!) something along the lines of: go to the 3rd top-level row. Now go to the 9th child of that row. Proceed to the 4th child of the previous row. Then continue to the 1st child of that. Now you are at the row this tree path describes. This is not what it means for Gtk+ though. While humans start counting at 1, computers usually start counting at 0. So the real meaning of the tree path "3:9:4:1" is: Go to the 4th top-level row. Then go to the 10th child of that row. Pick the 5th child of that row. Then proceed to the 2nd child of the previous row. Now you are at the row this tree path describes. :)

The implication of this way of refering to rows is as follows: if you insert or delete rows in the middle or if the rows are resorted, a tree path might suddenly refer to a completely different row than it refered to before the insertion/deletion/resorting. This is important to keep in mind. (See the section on `GtkTreeRowReference`s

below for a tree path that keeps updating itself to make sure it always refers to the same row when the model changes).

This effect becomes apparent if you imagine what would happen if we were to delete the row entitled 'funny clips' from the tree in the above picture. The row 'movie trailers' would suddenly be the first and only child of 'clips', and be described by the tree path that formerly belonged to 'funny clips', ie. "1:0:0".

You can get a new `GtkTreePath` from a path in string form using `gtk_tree_path_new_from_string`, and you can convert a given `GtkTreePath` into its string notation with `gtk_tree_path_to_string`. Usually you will rarely have to handle the string notation, it is described here merely to demonstrate the concept of tree paths.

Instead of the string notation, `GtkTreePath` uses an integer array internally. You can get the depth (ie. the nesting level) of a tree path with `gtk_tree_path_get_depth`. A depth of 0 is the imaginary invisible root node of the tree view and model. A depth of 1 means that the tree path describes a top-level row. As lists are just trees without child nodes, all rows in a list always have tree paths of depth 1. `gtk_tree_path_get_indices` returns the internal integer array of a tree path. You will rarely need to operate with those either.

If you operate with tree paths, you are most likely to use a given tree path, and use functions like `gtk_tree_path_up`, `gtk_tree_path_down`, `gtk_tree_path_next`, `gtk_tree_path_prev`, `gtk_tree_path_is_ancestor`, or `gtk_tree_path_is_descendant`. Note that this way you can construct and operate on tree paths that refer to rows that do not exist in model or view! The only way to check whether a path is valid for a specific model (ie. the row described by the path exists) is to convert the path into an iter using `gtk_tree_model_get_iter`.

`GtkTreePath` is an opaque structure, with its details hidden from the compiler. If you need to make a copy of a tree path, use `gtk_tree_path_copy`.

3.2.2. GtkTreeIter

Another way to refer to a row in a list or tree is `GtkTreeIter`. A tree iter is just a structure that contains a couple of pointers that mean something to the model you are using. Tree iters are used internally by models, and they often contain a direct pointer to the internal data of the row in question. You should never look at the content of a tree iter and you must not modify it directly either.

All tree models (and therefore also `GtkListStore` and `GtkTreeStore`) must support the `GtkTreeModel` functions that operate on tree iters (e.g. get the tree iter for the first child of the row specified by a given tree iter, get the first row in the list/tree, get the n-th child of a given iter etc.). Some of these functions are:

- `gtk_tree_model_get_iter_first` - sets the given iter to the first top-level item in the list or tree
- `gtk_tree_model_iter_next` - sets the given iter to the next item at the current level in a list or tree.
- `gtk_tree_model_iter_children` - sets the first given iter to the first child of the row referenced by the second iter (not very useful for lists, mostly useful for trees).
- `gtk_tree_model_iter_n_children` - returns the number of children the row referenced by the provided iter has. If you pass `NULL` instead of a pointer to an iter structure, this function will return the number of top-level rows. You can also use this function to count the number of items in a list store.
- `gtk_tree_model_iter_nth_child` - sets the first iter to the n-th child of the row referenced by the second iter. If you pass `NULL` instead of a pointer to an iter structure as the second iter, you can get the first iter set to the n-th row of a list.
- `gtk_tree_model_iter_parent` - sets the first iter to the parent of the row referenced by the second iter (does nothing for lists, only useful for trees).

Almost all of those functions return `TRUE` if the requested operation succeeded, and return `FALSE` otherwise. There are more functions that operate on iters. Check out the `GtkTreeModel` API reference for details.

You might notice that there is no `gtk_tree_model_iter_prev`. This is unlikely to be implemented for a variety of reasons. It should be fairly simple to write a helper function that provides this functionality though once you have read this section.

Tree iters are used to retrieve data from the store, and to put data into the store. You also get a tree iter as result if you add a new row to the store using `gtk_list_store_append` or `gtk_tree_store_append`.

Tree iters are often only valid for a short time, and might become invalid if the store changes with some models. It is therefore usually a bad idea to store tree iters, unless you really know what you are doing. You can use `gtk_tree_model_get_flags` to get a model's flags, and check whether the `GTK_TREE_MODEL_ITERS_PERSIST` flag is set (in which case a tree iter will be valid as long as a row exists), yet still it is not advisable to store iter structures unless you really mean to do that. There is a better way to keep track of a row over time: `GtkTreeRowReference`

3.2.3. GtkTreeRowReference

A `GtkTreeRowReference` is basically an object that takes a tree path, and watches a model for changes. If anything changes, like rows getting inserted or removed, or rows getting re-ordered, the tree row reference object will keep the given tree path up to date, so that it always points to the same row as before. In case the given row is removed, the tree row reference will become invalid.

A new tree row reference can be created with `gtk_tree_row_reference_new`, given a model and a tree path. After that, the tree row reference will keep updating the path whenever the model changes. The current tree path of the row originally refered to when the tree row reference was created can be retrieved with `gtk_tree_row_reference_get_path`. If the row has been deleted, `NULL` will be returned instead of of a tree path. The tree path returned is a *copy*, so it will need to be freed with `gtk_tree_path_free` when it is no longer needed.

You can check whether the row referenced still exists with `gtk_tree_row_reference_valid`, and free it with when no longer needed.

For the curious: internally, the tree row reference connects to the tree model's `"row-inserted"`, `"row-deleted"`, and `"rows-reordered"` signals and updates its internal tree path whenever something happened to the model that affects the position of the referenced row.

Note that using tree row references entails a small overhead. This is hardly significant for 99.9% of all applications out there, but when you have multiple thousands of rows and/or row references, this might be something to keep in mind (because whenever rows are inserted, removed, or reordered, a signal will be sent out and processed for each row reference).

If you have read the tutorial only up to here so far, it is hard to explain really what tree row references are good for. An example where tree row references come in handy can be found further below in the section on removing multiple rows in one go.

In practice, a programmer can either use tree row references to keep track of rows over time, or store tree iters directly (if, and only if, the model has persistent iters). Both `GtkListStore` and `GtkTreeStore` have persistent iters, so storing iters is possible. However, using tree row references is definitively the Right Way(tm) to do things, even though it comes with some overhead that might impact performance in case of trees that have a very large number of rows (in that case it might be preferable to write a custom model anyway though). Especially beginners might find it easier to handle and store tree row references than iters, because tree row references are handled by pointer value, which you can easily add to a `GList` or pointer array, while it is easy to store tree iters in a wrong way.

3.2.4. Usage

Tree iters can easily be converted into tree paths using `gtk_tree_model_get_path`, and tree paths can easily be converted into tree iters using `gtk_tree_model_get_iter`. Here is an example that shows how to get the iter from the tree path that is passed to us from the tree view in the `"row-activated"` signal callback. We need the iter here to retrieve data from the store

```
/**********************************************************
 *                                                        *
 * Converting a GtkTreePath into a GtkTreeIter            *
 *                                                        *
 **********************************************************/

/**********************************************************
 *
 * onTreeViewRowActivated: a row has been double-clicked
 *
 **********************************************************/

void
onTreeViewRowActivated (GtkTreeView *view, GtkTreePath *path,
                        GtkTreeViewColumn *col, gpointer userdata)
{
GtkTreeIter   iter;
  GtkTreeModel *model;

  model = gtk_tree_view_get_model(view);

  if (gtk_tree_model_get_iter(model, &iter, path))
  {
```

```
    gchar *name;

    gtk_tree_model_get(model, &iter, COL_NAME, &name, -1);

    g_print ("The row containing the name '%s' has been double-clicked.\n", name);

    g_free(name);
  }
}
```

Tree row references reveal the current path of a row with `gtk_tree_row_reference_get_path`. There is no direct way to get a tree iter from a tree row reference, you have to retrieve the tree row reference's path first and then convert that into a tree iter.

As tree iters are only valid for a short time, they are usually allocated on the stack, as in the following example (keep in mind that `GtkTreeIter` is just a structure that contains data fields you do not need to know anything about):

```
/************************************************************
 *                                                          *
 *  Going through every row in a list store                 *
 *                                                          *
 ************************************************************/

 void
 traverse_list_store (GtkListStore *liststore)
 {
   GtkTreeIter  iter;
   gboolean     valid;

   g_return_if_fail ( liststore != NULL );

   /* Get first row in list store */
   valid = gtk_tree_model_get_iter_first(GTK_TREE_MODEL(liststore), &iter);

   while (valid)
   {
      /* ... do something with that row using the iter ...          */
      /* (Here column 0 of the list store is of type G_TYPE_STRING) */
      gtk_list_store_set(liststore, &iter, 0, "Joe", -1);

      /* Make iter point to the next row in the list store */
      valid = gtk_tree_model_iter_next(GTK_TREE_MODEL(liststore), &iter);
   }
 }
```

The code above asks the model to fill the iter structure to make it point to the first row in the list store. If there is a first row and the list store is not empty, the iter will be set, and `gtk_tree_model_get_iter_first` will return `TRUE`. If there is no first row, it will just return `FALSE`. If a first row exists, the while loop will be entered and we change some of the first row's data. Then we ask the model to make the given iter point to the next row, until there are no more rows, which is when `gtk_tree_model_iter_next` returns `FALSE`. Instead of traversing the list store we could also have used `gtk_tree_model_foreach`

3.3. Adding Rows to a Store

3.3.1. Adding Rows to a List Store

Rows are added to a list store with `gtk_list_store_append`. This will insert a new empty row at the end of the list. There are other functions, documented in the GtkListStore API reference, that give you more control about where exactly the new row is inserted, but as they work very similar to `gtk_list_store_append` and are fairly straight-forward to use, we will not deal with them here.

Here is a simple example of how to create a list store and add some (empty) rows to it:

```
GtkListStore *liststore;
GtkTreeIter   iter;
```

```
liststore = gtk_list_store_new(1, G_TYPE_STRING);

/* Append an empty row to the list store. Iter will point to the new row */
gtk_list_store_append(liststore, &iter);

/* Append an empty row to the list store. Iter will point to the new row */
gtk_list_store_append(liststore, &iter);

/* Append an empty row to the list store. Iter will point to the new row */
gtk_list_store_append(liststore, &iter);
```

This in itself is not very useful yet of course. We will add data to the rows in the next section.

3.3.2. Adding Rows to a Tree Store

Adding rows to a tree store works similar to adding rows to a list store, only that `gtk_tree_store_append` is the function to use and one more argument is required, namely the tree iter to the parent of the row to insert. If you supply NULL instead of providing the tree iter of another row, a new top-level row will be inserted. If you do provide a parent tree iter, the new empty row will be inserted after any already existing children of the parent. Again, there are other ways to insert a row into the tree store and they are documented in the GtkTreeStore API reference manual. Another short example:

```
GtkTreeStore *treestore;
GtkTreeIter   iter, child;

treestore = gtk_tree_store_new(1, G_TYPE_STRING);

/* Append an empty top-level row to the tree store.
 *  Iter will point to the new row */
gtk_tree_store_append(treestore, &iter, NULL);

/* Append another empty top-level row to the tree store.
 *  Iter will point to the new row */
gtk_tree_store_append(treestore, &iter, NULL);

/* Append a child to the row we just added.
 *  Child will point to the new row */
gtk_tree_store_append(treestore, &child, &iter);

/* Get the first row, and add a child to it as well (could have been done
 *  right away earlier of course, this is just for demonstration purposes) */
if (gtk_tree_model_get_iter_first(GTK_TREE_MODEL(treestore), &iter))
{
  /* Child will point to new row */
  gtk_tree_store_append(treestore, &child, &iter);
}
else
{
  g_error("Oops, we should have a first row in the tree store!\n");
}
```

3.3.3. Speed Issues when Adding a Lot of Rows

A common scenario is that a model needs to be filled with a lot of rows at some point, either at start-up, or when some file is opened. An equally common scenario is that this takes an awfully long time even on powerful machines once the model contains more than a couple of thousand rows, with an exponentially decreasing rate of insertion. As already pointed out above, writing a custom model might be the best thing to do in this case. Nevertheless, there are some things you can do to work around this problem and speed things up a bit even with the stock Gtk+ models:

Firstly, you should detach your list store or tree store from the tree view before doing your mass insertions, then do your insertions, and only connect your store to the tree view again when you are done with your insertions. Like this:

```
...
```

```
model = gtk_tree_view_get_model(GTK_TREE_VIEW(view));

g_object_ref(model); /* Make sure the model stays with us after the tree view unrefs it */

gtk_tree_view_set_model(GTK_TREE_VIEW(view), NULL); /* Detach model from view */

... insert a couple of thousand rows ...

gtk_tree_view_set_model(GTK_TREE_VIEW(view), model); /* Re-attach model to view */

g_object_unref(model);

...
```

Secondly, you should make sure that sorting is disabled while you are doing your mass insertions, otherwise your store might be resorted after each and every single row insertion, which is going to be everything but fast.

Thirdly, you should not keep around a lot of tree row references if you have so many rows, because with each insertion (or removal) every single tree row reference will check whether its path needs to be updated or not.

3.4. Manipulating Row Data

Adding empty rows to a data store is not terribly exciting, so let's see how we can add or change data in the store.

`gtk_list_store_set` and `gtk_tree_store_set` are used to manipulate a given row's data. There is also `gtk_list_store_set_value` and `gtk_tree_store_set_value`, but those should only be used by people familiar with GLib's GValue system.

Both `gtk_list_store_set` and `gtk_tree_store_set` take a variable number of arguments, and must be terminated with a -1 argument. The first two arguments are a pointer to the model, and the iter pointing to the row whose data we want to change. They are followed by a variable number of (column, data) argument pairs, terminated by a -1. The column refers to the model column number and is usually an enum value (to make the code more readable and to make changes easier). The data should be of the same data type as the model column.

Here is an example where we create a store that stores two strings and one integer for each row:

```
enum
{
  COL_FIRST_NAME = 0,
  COL_LAST_NAME,
  COL_YEAR_BORN,
  NUM_COLS
};

GtkListStore *liststore;
GtkTreeIter   iter;

liststore = gtk_list_store_new(NUM_COLS, G_TYPE_STRING, G_TYPE_STRING, G_TYPE_UINT);

/* Append an empty row to the list store. Iter will point to the new row */
gtk_list_store_append(liststore, &iter);

/* Fill fields with some data */
gtk_list_store_set (liststore, &iter,
                    COL_FIRST_NAME, "Joe",
                    COL_LAST_NAME, "Average",
                    COL_YEAR_BORN, (guint) 1970,
                    -1);
```

You do not need to worry about allocating and freeing memory for the data to store. The model (or more precisely: the GLib/GObject GType and GValue system) will take care of that for you. If you store a string, for example, the model will make a copy of the string and store that. If you then set the field to a new string later on, the model will automatically free the old string and again make a copy of the new string and store the copy. This applies to almost all types, be it `G_TYPE_STRING` or `GDK_TYPE_PIXBUF`.

The exception to note is `G_TYPE_POINTER`. If you allocate a chunk of data or a complex structure and store it in a `G_TYPE_POINTER` field, only the pointer *value* is stored. The model does not know anything about the size or content of the data your pointer refers to, so it could not even make a copy if it wanted to, so you need to

allocate and free the memory yourself in this case. However, if you do not want to do that yourself and want the model to take care of your custom data for you, then you need to register your own type and derive it from one of the GLib fundamental types (usually G_TYPE_BOXED). See the GObject GType reference manual for details. Making a copy of data involves memory allocation and other overhead of course, so one should consider the performance implications of using a custom GLib type over a G_TYPE_POINTER carefully before taking that approach. Again, a custom model might be the better alternative, depending on the overall amount of data to be stored (and retrieved).

3.5. Retrieving Row Data

Storing data is not very useful if it cannot be retrieved again. This is done using `gtk_tree_model_get`, which takes similar arguments as `gtk_list_store_set` or `gtk_tree_store_set` do, only that it takes (column, pointer) arguments. The pointer must point to a variable that is of the same type as the data stored in that particular model column.

Here is the previous example extended to traverse the list store and print out the data stored. As an extra, we use `gtk_tree_model_foreach` to traverse the store and retrieve the row number from the `GtkTreePath` passed to us in the foreach callback function:

```
#include <gtk/gtk.h>

enum
{
  COL_FIRST_NAME = 0,
  COL_LAST_NAME,
  COL_YEAR_BORN,
  NUM_COLS
};

gboolean
foreach_func (GtkTreeModel *model,
              GtkTreePath  *path,
              GtkTreeIter  *iter,
              gpointer      user_data)
{
  gchar *first_name, *last_name, *tree_path_str;
  guint  year_of_birth;

  /* Note: here we use 'iter' and not '&iter', because we did not allocate
   *  the iter on the stack and are already getting the pointer to a tree iter */

  gtk_tree_model_get (model, iter,
                      COL_FIRST_NAME, &first_name,
                      COL_LAST_NAME, &last_name,
                      COL_YEAR_BORN, &year_of_birth,
                      -1);

  tree_path_str = gtk_tree_path_to_string(path);

  g_print ("Row %s: %s %s, born %u\n", tree_path_str,
           first_name, last_name, year_of_birth);

  g_free(tree_path_str);

  g_free(first_name); /* gtk_tree_model_get made copies of       */
  g_free(last_name);  /* the strings for us when retrieving them */

  return FALSE; /* do not stop walking the store, call us with next row */
}

void
create_and_fill_and_dump_store (void)
{
  GtkListStore *liststore;
  GtkTreeIter   iter;

  liststore = gtk_list_store_new(NUM_COLS, G_TYPE_STRING, G_TYPE_STRING, G_TYPE_UINT);

  /* Append an empty row to the list store. Iter will point to the new row */
```

```
  gtk_list_store_append(liststore, &iter);

  /* Fill fields with some data */
  gtk_list_store_set (liststore, &iter,
                      COL_FIRST_NAME, "Joe",
                      COL_LAST_NAME, "Average",
                      COL_YEAR_BORN, (guint) 1970,
                      -1);

  /* Append another row, and fill in some data */
  gtk_list_store_append(liststore, &iter);

  gtk_list_store_set (liststore, &iter,
                      COL_FIRST_NAME, "Jane",
                      COL_LAST_NAME, "Common",
                      COL_YEAR_BORN, (guint) 1967,
                      -1);

  /* Append yet another row, and fill it */
  gtk_list_store_append(liststore, &iter);

  gtk_list_store_set (liststore, &iter,
                      COL_FIRST_NAME, "Yo",
                      COL_LAST_NAME, "Da",
                      COL_YEAR_BORN, (guint) 1873,
                      -1);

  /* Now traverse the list */

  gtk_tree_model_foreach(GTK_TREE_MODEL(liststore), foreach_func, NULL);
}

int
main (int argc, char **argv)
{
  gtk_init(&argc, &argv);

  create_and_fill_and_dump_store();

  return 0;
}
```

Note that when a new row is created, all fields of a row are set to a default NIL value appropriate for the data type in question. A field of type `G_TYPE_INT` will automatically contain the value 0 until it is set to a different value, and strings and all kind of pointer types will be `NULL` until set to something else. Those are valid contents for the model, and if you are not sure that row contents have been set to something, you need to be prepared to handle `NULL` pointers and the like in your code.

Run the above program with an additional empty row and look at the output to see this in effect.

3.5.1. Freeing Retrieved Row Data

Unless you are dealing with a model column of type `G_TYPE_POINTER`, `gtk_tree_model_get` will always make *copies* of the data retrieved.

In the case of strings, this means that you need to `g_free` the string returned when you don't need it any longer, as in the example above.

If you retrieve a `GObject` such as a `GdkPixbuf` from the store, `gtk_tree_model_get` will automatically add a reference to it, so you need to call `g_object_unref` on the retrieved object once you are done with it:

```
...

GdkPixbuf *pixbuf;

gtk_tree_model_get (model, &iter,
                    COL_PICTURE, &pixbuf,
                    NULL);
```

```
if (pixbuf != NULL)
{
  do_something_with_pixbuf (pixbuf);
  g_object_unref (pixbuf);
}

...
```

Similarly, `GBoxed`-derived types retrieved from a model need to be freed with `g_boxed_free` when done with them (don't worry if you have never heard of `GBoxed`).

If the model column is of type `G_TYPE_POINTER`, `gtk_tree_model_get` will simply copy the pointer value, but not the data (even if if it wanted to, it couldn't copy the data, because it would not know how to copy it or what to copy exactly). If you store pointers to objects or strings in a pointer column (which you should not do unless you really know what you are doing and why you are doing it), you do not need to unref or free the returned values as described above, because `gtk_tree_model_get` would not know what kind of data they are and therefore won't ref or copy them on retrieval.

3.6. Removing Rows

Rows can easily be removed with `gtk_list_store_remove` and `gtk_tree_store_remove`. The removed row will automatically be removed from the tree view as well, and all data stored will automatically be freed, with the exception of `G_TYPE_POINTER` columns (see above).

Removing a single row is fairly straight forward: you need to get the iter that identifies the row you want to remove, and then use one of the above functions. Here is a simple example that removes a row when you double-click on it (bad from a user interface point of view, but then it is just an example):

```
static void
onRowActivated (GtkTreeView        *view,
                GtkTreePath        *path,
                GtkTreeViewColumn  *col,
                gpointer            user_data)
{
  GtkTreeModel *model;
  GtkTreeIter   iter;

  g_print ("Row has been double-clicked. Removing row.\n");

  model = gtk_tree_view_get_model(view);

  if (!gtk_tree_model_get_iter(model, &iter, path))
    return; /* path describes a non-existing row - should not happen */

  gtk_list_store_remove(GTK_LIST_STORE(model), &iter);
}


void
create_treeview (void)
{
  ...
  g_signal_connect(treeview, "row-activated", G_CALLBACK(onRowActivated), NULL);
  ...
}
```

Note: `gtk_list_store_remove` and `gtk_tree_store_remove` both have slightly different semantics in Gtk+-2.0 and Gtk+-2.2 and later. In Gtk+-2.0, both functions do not return a value, while in later Gtk+ versions those functions return either `TRUE` or `FALSE` to indicate whether the iter given has been set to the next valid row (or invalidated if there is no next row). This is important to keep in mind when writing code that is supposed to work with all Gtk+-2.x versions. In that case you should just ignore the value returned (as in the call above) and check the iter with `gtk_list_store_iter_is_valid` if you need it.

If you want to remove the n-th row from a list (or the n-th child of a tree node), you have two approaches: either you first create a `GtkTreePath` that describes that row and then turn it into an iter and remove it; or you take the

iter of the parent node and use `gtk_tree_model_iter_nth_child` (which will also work for list stores if you use `NULL` as the parent iter. Of course you could also start with the iter of the first top-level row, and then step-by-step move it to the row you want, although that seems a rather awkward way of doing it.

The following code snippet will remove the n-th row of a list if it exists:

```
/*****************************************************************
 *
 *  list_store_remove_nth_row
 *
 *  Removes the nth row of a list store if it exists.
 *
 *  Returns TRUE on success or FALSE if the row does not exist.
 *
 *****************************************************************/

gboolean
list_store_remove_nth_row (GtkListStore *store, gint n)
{
  GtkTreeIter  iter;

  g_return_val_if_fail (GTK_IS_LIST_STORE(store), FALSE);

  /* NULL means the parent is the virtual root node, so the
   *  n-th top-level element is returned in iter, which is
   *  the n-th row in a list store (as a list store only has
   *  top-level elements, and no children) */
  if (gtk_tree_model_iter_nth_child(GTK_TREE_MODEL(store), &iter, NULL, n))
  {
    gtk_list_store_remove(store, &iter);
    return TRUE;
  }

  return FALSE;
}
```

3.7. Removing Multiple Rows

Removing multiple rows at once can be a bit tricky at times, and requires some thought on how to do this best. For example, it is not possible to traverse a store with `gtk_tree_model_foreach`, check in the callback function whether the given row should be removed and then just remove it by calling one of the stores' remove functions. This will not work, because the model is changed from within the foreach loop, which might suddenly invalidate formerly valid tree iters in the foreach function, and thus lead to unpredictable results.

You could traverse the store in a `while` loop of course, and call `gtk_list_store_remove` or `gtk_tree_store_remove` whenever you want to remove a row, and then just continue if the remove functions returns `TRUE` (meaning that the iter is still valid and now points to the row after the row that was removed). However, this approach will only work with Gtk+-2.2 or later and will not work if you want your programs to compile and work with Gtk+-2.0 as well, for the reasons outlined above (in Gtk+-2.0 the remove functions did not set the passed iter to the next valid row). Also, while this approach might be feasable for a list store, it gets a bit awkward for a tree store.

Here is an example for an alternative approach to removing multiple rows in one go (here we want to remove all rows from the store that contain persons that have been born after 1980, but it could just as well be all selected rows or some other criterion):

```
/*****************************************************************
 *
 *  Removing multiple rows in one go
 *
 *****************************************************************/

 ...

 gboolean
```

```
  foreach_func (GtkTreeModel *model,
                GtkTreePath  *path,
                GtkTreeIter  *iter,
                GList       **rowref_list)
  {
    guint  year_of_birth;

    g_assert ( rowref_list != NULL );

    gtk_tree_model_get (model, iter, COL_YEAR_BORN, &year_of_birth, -1);

    if ( year_of_birth > 1980 )
    {
      GtkTreeRowReference  *rowref;

      rowref = gtk_tree_row_reference_new(model, path);

      *rowref_list = g_list_append(*rowref_list, rowref);
    }

    return FALSE; /* do not stop walking the store, call us with next row */
  }

  void
  remove_people_born_after_1980 (void)
  {
     GList *rr_list = NULL;    /* list of GtkTreeRowReferences to remove */
     GList *node;

     gtk_tree_model_foreach(GTK_TREE_MODEL(store),
                            (GtkTreeModelForeachFunc) foreach_func,
                            &rr_list);

     for ( node = rr_list;  node != NULL;  node = node->next )
     {
        GtkTreePath *path;

        path = gtk_tree_row_reference_get_path((GtkTreeRowReference*)node->data);

        if (path)
        {
           GtkTreeIter  iter;

           if (gtk_tree_model_get_iter(GTK_TREE_MODEL(store), &iter, path))
           {
             gtk_list_store_remove(store, &iter);
           }

           /* FIXME/CHECK: Do we need to free the path here? */
        }
     }

     g_list_foreach(rr_list, (GFunc) gtk_tree_row_reference_free, NULL);
     g_list_free(rr_list);
  }

  ...
```

`gtk_list_store_clear` and `gtk_tree_store_clear` come in handy if you want to remove all rows.

3.8. Storing GObjects (Pixbufs etc.)

A special case are `GObject` types, like `GDK_TYPE_PIXBUF`, that get stored in a list or tree store. The store will not make a copy of the object, rather it will increase the object's refcount. The store will then unref the object again if it is no longer needed (ie. a new object is stored in the old object's place, the current value is replaced by NULL, the row is removed, or the store is destroyed).

From a developer perspective, this means that you need to `g_object_unref` an object that you have just added to the store if you want the store to automatically dispose of it when no longer needed. This is because on object creation, the object has an initial refcount of 1, which is "your" refcount, and the object will only be destroyed when it reaches a refcount of 0. Here is the life cycle of a pixbuf:

```
GtkListStore *list_store;
GtkTreeIter   iter;
GdkPixbuf    *pixbuf;
GError       *error = NULL;

list_store = gtk_list_store_new (2, GDK_TYPE_PIXBUF, G_TYPE_STRING);

pixbuf = gdk_pixbuf_new_from_file("icon.png", &error);

/* pixbuf has a refcount of 1 after creation */

if (error)
{
  g_critical ("Could not load pixbuf: %s\n", error->message);
  g_error_free(error);
  return;
}

gtk_list_store_append(list_store, &iter);

gtk_list_store_set(list_store, &iter, 0, pixbuf, 1, "foo", -1);

/* pixbuf has a refcount of 2 now, as the list store has added its own reference */

g_object_unref(pixbuf);

/* pixbuf has a refcount of 1 now that we have released our initial reference */

/* we don't want an icon in that row any longer */
gtk_list_store_set(list_store, &iter, 0, NULL, -1);

/* pixbuf has automatically been destroyed after its refcount has reached 0.
 *  The list store called g_object_unref() on the pixbuf when it replaced
 *  the object in the store with a new value (NULL). */
```

Having learned how to add, manipulate, and retrieve data from a store, the next step is to get that data displayed in a `GtkTreeView` widget.

3.9. Storing Data Structures: of Pointers, GBoxed Types, and GObject (TODO)

Unfinished chapter.

Chapter 4. Creating a Tree View

In order to display data in a tree view widget, we need to create one first, and we need to instruct it where to get the data to display from.

A new tree view is created with:

```
GtkWidget *view;

view = gtk_tree_view_new();
```

4.1. Connecting Tree View and Model

Before we proceed to the next section where we display data on the screen, we need connect our data store to the tree view, so it knows where to get the data to display from. This is achieved with `gtk_tree_view_set_model`, which will by itself do very little. However, it is a prerequisite for what we do in the following sections. `gtk_tree_view_new_with_model` is a convenience function for the previous two.

`gtk_tree_view_get_model` will return the model that is currently attached to a given tree view, which is particularly useful in callbacks where you only get passed the tree view widget (after all, we do not want to go down the road of global variables, which will inevitably lead to the Dark Side, do we?).

4.1.1. Reference counting

Tree models like `GtkListStore` and `GtkTreeStore` are `GObjects` and have a reference count of 1 after creation. The tree view will add its own reference to the model when you add the model with `gtk_tree_view_set_model`, and will unref it again when you replace the model with another model, unset the model by passing `NULL` as a model, or when the tree view is destroyed. [1]

This means that you need to take care of "your" reference yourself, otherwise the model will not be destroyed properly when you disconnect it from the tree view, and its memory will not be freed (which does not matter much if the same model is connected to the tree view from application start to end). If you plan to use the same model for a tree view for the whole duration of the application, you can get rid of "your" reference right after you have connected the model to the view - then the model will be destroyed automatically when the tree view is destroyed (which will be automatically destroyed when the window it is in is destroyed):

```
GtkListStore *liststore;
GtkWidget    *view;

view = gtk_tree_view_new();

liststore = gtk_list_store_new(1, G_TYPE_STRING);

gtk_tree_view_set_model(GTK_TREE_VIEW(view), GTK_TREE_MODEL(liststore));

g_object_unref(liststore);

/* Now the model will be destroyed when the tree view is destroyed */
```

4.2. Tree View Look and Feel

There are a couple of ways to influence the look and feel of the tree view. You can hide or show column headers with `gtk_tree_view_set_headers_visible`, and set them clickable or not with `gtk_tree_view_set_headers_clickable` (which will be done automatically for you if you enable sorting).

`gtk_tree_view_set_rules_hint` will enable or disable rules in the tree view. [2] As the function name implies, this setting is only a hint; in the end it depends on the active Gtk+ theme engine if the tree view shows ruled lines or not. Users seem to have strong feelings about rules in tree views, so it is probably a good idea to provide an option somewhere to disable rule hinting if you set it on tree views (but then, people also seem to have strong feelings about options abundance and 'sensible' default options, so whatever you do will probably upset someone at some point).

The expander column can be set with `gtk_tree_view_set_expander_column`. This is the column where child elements are indented with respect to their parents, and where rows with children have an 'expander' arrow with which a node's children can be collapsed (hidden) or expanded (shown). By default, this is the first column.

Notes

1. 'Reference counting' means that an object has a counter that can be increased or decreased (ref-ed and unref-ed). If the counter is unref-ed to 0, the object is automatically destroyed. This is useful, because other objects or application programmers only have to think about whether *they themselves* are still using that object or not, without knowing anything about others also using it. The object is simply automatically destroyed when no one is using it any more.
2. 'Rules' means that every second line of the tree view has a shaded background, which makes it easier to see which cell belongs to which row in tree views that have a lot of columns.

Chapter 5. Mapping Data to the Screen: GtkTreeViewColumn and GtkCellRenderer

As outlined above, tree view columns represent the visible columns on the screen that have a column header with a column name and can be resized or sorted. A tree view is made up of tree view columns, and you need at least one tree view column in order to display something in the tree view. Tree view columns, however, do not display anything by themselves, this is done by specialised `GtkCellRenderer` objects. Cell renderers are packed into tree view columns much like widgets are packed into `GtkHBoxes`.

Here is a diagram (courtesy of Owen Taylor) that pictures the relationship between tree view columns and cell renderers:

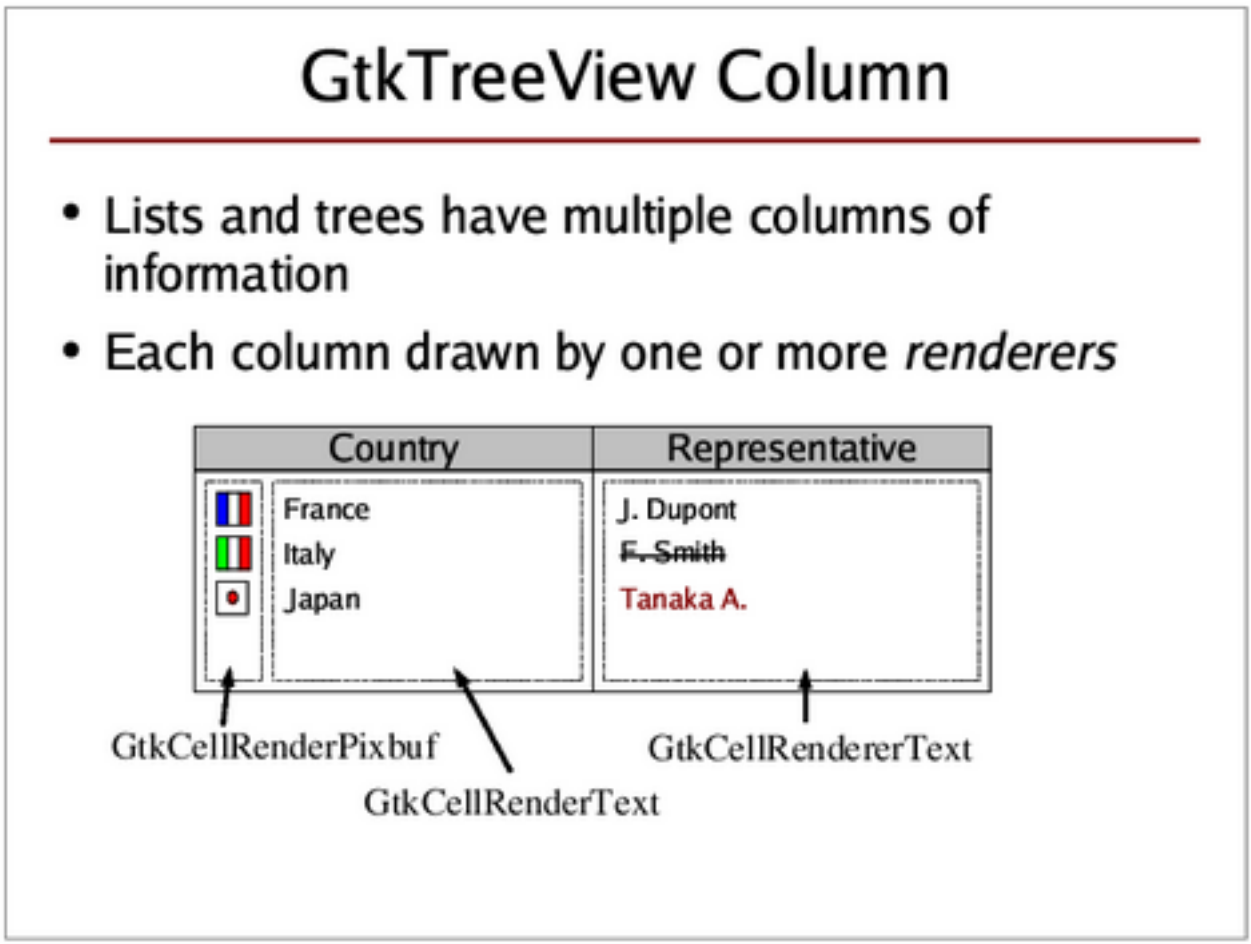

Figure 5-1. Cell Renderer Properties

In the above diagram, both 'Country' and 'Representative' are tree view columns, where the 'Country' and 'Representative' labels are the column headers. The 'Country' column contains two cell renderers, one to display the flag icons, and one to display the country name. The 'Representative' column only contains one cell renderer to display the representative's name.

5.1. Cell Renderers

Cell renderers are objects that are responsible for the actual rendering of data within a `GtkTreeViewColumn`. They are basically just GObjects (ie. not widgets) that have certain properties, and those properties determine how a single cell is drawn.

In order to draw cells in different rows with different content, a cell renderer's properties need to be set accordingly for each single row/cell to render. This is done either via attributes or cell data functions (see below). If you set up attributes, you tell Gtk which model column contains the data from which a property should be set before rendering a certain row. Then the properties of a cell renderer are set automatically according to the data in the model before each row is rendered. Alternatively, you can set up cell data functions, which are called for each

row to be rendererd, so that you can manually set the properties of the cell renderer before it is rendered. Both approaches can be used at the same time as well. Lastly, you can set a cell renderer property when you create the cell renderer. That way it will be used for all rows/cells to be rendered (unless it is changed later of course).

Different cell renderers exist for different purposes:

- `GtkCellRendererText` renders strings or numbers or boolean values as text ("Joe", "99.32", "true")
- `GtkCellRendererPixbuf` is used to display images; either user-defined images, or one of the stock icons that come with Gtk+.
- `GtkCellRendererToggle` displays a boolean value in form of a check box or as a radio button.
- `GtkCellEditable` is a special cell that implements editable cells (ie. GtkEntry or GtkSpinbutton in a treeview). This is not a cell renderer! If you want to have editable text cells, use `GtkCellRendererText` and make sure the "editable" property is set. `GtkCellEditable` is only used by implementations of editable cells and widgets that can be inside of editable cells. You are unlikely to ever need it.

Contrary to what one may think, a cell renderer does not render just one single cell, but is responsible for rendering part or whole of a tree view column for each single row. It basically starts in the first row and renders its part of the column there. Then it proceeds to the next row and renders its part of the column there again. And so on.

How does a cell renderer know what to render? A cell renderer object has certain 'properties' that are documented in the API reference (just like most other objects, and widgets). These properties determine what the cell renderer is going to render and how it is going to be rendered. Whenever the cell renderer is called upon to render a certain cell, it looks at its properties and renders the cell accordingly. This means that whenever you set a property or change a property of the cell renderer, this will affect all rows that are rendered after the change, until you change the property again.

Here is a diagram (courtesy of Owen Taylor) that tries to show what is going on when rows are rendered:

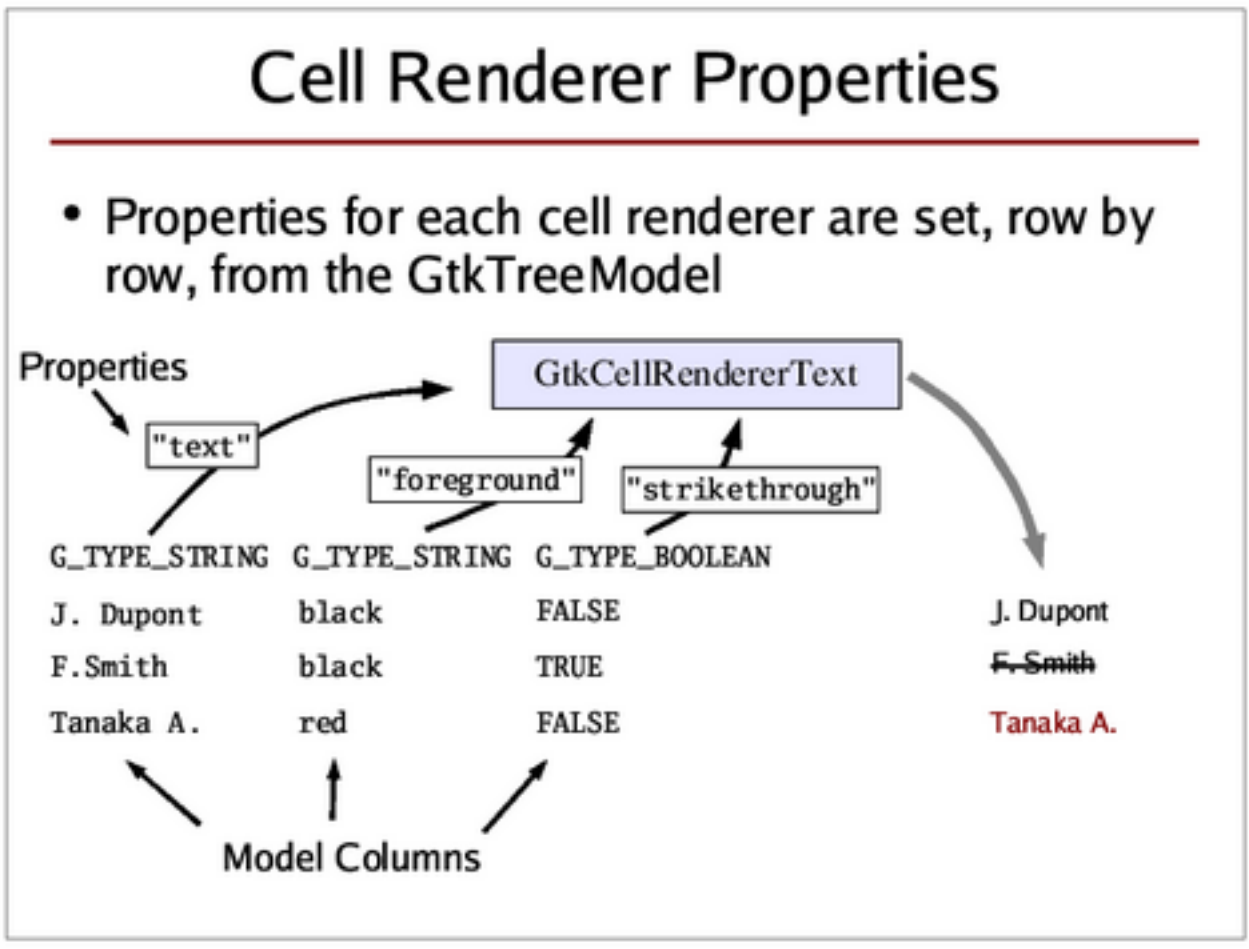

Figure 5-2. GtkTreeViewColumns and GtkCellRenderers

The above diagram shows the process when attributes are used. In the example, a text cell renderer's `"text"` property has been linked to the first model column. The `"text"` property contains the string to be rendered. The `"foreground"` property, which contains the colour of the text to be shown, has been linked to the second model column. Finally, the `"strikethrough"` property, which determines whether the text should be with a horizontal line that strikes through the text, has been connected to the third model column (of type `G_TYPE_BOOLEAN`).

With this setup, the cell renderer's properties are 'loaded' from the model before each cell is rendered.

Here is a silly and utterly useless little example that demonstrates this behaviour, and introduces some of the most commonly used properties of `GtkCellRendererText`:

```
#include <gtk/gtk.h>

enum
{
  COL_FIRST_NAME = 0,
  COL_LAST_NAME,
  NUM_COLS
} ;

static GtkTreeModel *
create_and_fill_model (void)
{
  GtkTreeStore  *treestore;
  GtkTreeIter    toplevel, child;

  treestore = gtk_tree_store_new(NUM_COLS, G_TYPE_STRING, G_TYPE_STRING);

  /* Append a top level row and leave it empty */
  gtk_tree_store_append(treestore, &toplevel, NULL);

  /* Append a second top level row, and fill it with some data */
  gtk_tree_store_append(treestore, &toplevel, NULL);
  gtk_tree_store_set(treestore, &toplevel,
                     COL_FIRST_NAME, "Joe",
                     COL_LAST_NAME, "Average",
                     -1);

  /* Append a child to the second top level row, and fill in some data */
  gtk_tree_store_append(treestore, &child, &toplevel);
  gtk_tree_store_set(treestore, &child,
                     COL_FIRST_NAME, "Jane",
                     COL_LAST_NAME, "Average",
                     -1);

  return GTK_TREE_MODEL(treestore);
}

static GtkWidget *
create_view_and_model (void)
{
  GtkTreeViewColumn   *col;
  GtkCellRenderer     *renderer;
  GtkWidget           *view;
  GtkTreeModel        *model;

  view = gtk_tree_view_new();

  /* --- Column #1 --- */

  col = gtk_tree_view_column_new();

  gtk_tree_view_column_set_title(col, "First Name");

  /* pack tree view column into tree view */
  gtk_tree_view_append_column(GTK_TREE_VIEW(view), col);

  renderer = gtk_cell_renderer_text_new();

  /* pack cell renderer into tree view column */
  gtk_tree_view_column_pack_start(col, renderer, TRUE);
```

```
  /* set 'text' property of the cell renderer */
  g_object_set(renderer, "text", "Boooo!", NULL);


  /* --- Column #2 --- */

  col = gtk_tree_view_column_new();

  gtk_tree_view_column_set_title(col, "Last Name");

  /* pack tree view column into tree view */
  gtk_tree_view_append_column(GTK_TREE_VIEW(view), col);

  renderer = gtk_cell_renderer_text_new();

  /* pack cell renderer into tree view column */
  gtk_tree_view_column_pack_start(col, renderer, TRUE);

  /* set 'cell-background' property of the cell renderer */
  g_object_set(renderer,
               "cell-background", "Orange",
               "cell-background-set", TRUE,
               NULL);

  model = create_and_fill_model();

  gtk_tree_view_set_model(GTK_TREE_VIEW(view), model);

  g_object_unref(model); /* destroy model automatically with view */

  gtk_tree_selection_set_mode(gtk_tree_view_get_selection(GTK_TREE_VIEW(view)),
                              GTK_SELECTION_NONE);

  return view;
}


int
main (int argc, char **argv)
{
  GtkWidget *window;
  GtkWidget *view;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  g_signal_connect(window, "delete_event", gtk_main_quit, NULL); /* dirty */

  view = create_view_and_model();

  gtk_container_add(GTK_CONTAINER(window), view);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;
}
```

The above code should produce something looking like this:

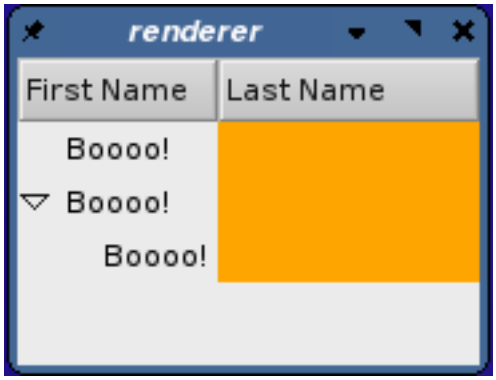

Figure 5-3. Persistent Cell Renderer Properties

It looks like the tree view display is partly correct and partly incomplete. On the one hand the tree view renders the correct number of rows (note how there is no orange on the right after row 3), and it displays the hierarchy correctly (on the left), but it does not display any of the data that we have stored in the model. This is because we have made no connection between what the cell renderers should render and the data in the model. We have simply set some cell renderer properties on start-up, and the cell renderers adhere to those set properties meticulously.

There are two different ways to connect cell renderers to data in the model: attributes and cell data functions.

5.2. Attributes

An attribute is a connection between a cell renderer property and a field/column in the model. Whenever a cell is to be rendered, a cell renderer property will be set to the values of the specified model column of the row that is to be rendered. It is very important that the column's data type is the same type that a property takes according to the API reference manual. Here is some code to look at:

```
...

col = gtk_tree_view_column_new();

renderer = gtk_cell_renderer_text_new();

gtk_tree_view_column_pack_start(col, renderer, TRUE);

gtk_tree_view_column_add_attribute(col, renderer, "text", COL_FIRST_NAME);

...
```

This means that the text cell renderer property `"text"` will be set to the string in model column `COL_FIRST_NAME` of each row to be drawn. It is important to internalise the difference between `gtk_tree_view_column_add_attribute` and `g_object_set`: `g_object_set` sets a property to a certain *value*, while `gtk_tree_view_column_add_attribute` sets a property to whatever is in the specified _model column_ at the time of rendering.

Again, when setting attributes it is very important that the data type stored in a model column is the same as the data type that a property requires as argument. Check the API reference manual to see the data type that is required for each property. When reading through the example a bit further above, you might have noticed that we set the `"cell-background"` property of a `GtkCellRendererText`, even though the API documentation does not list such a property. We can do this, because `GtkCellRendererText` is derived from `GtkCellRenderer`, which does in fact have such a property. Derived classes inherit the properties of their parents. This is the same as with widgets that you can cast into one of their ancestor classes. The API reference has an object hierarchy that shows you which classes a widget or some other object is derived from.

There are two more noteworthy things about `GtkCellRenderer` properties: one is that sometimes there are different properties which do the same, but take different arguments, such as the `"foreground"` and `"foreground-gdk"` properties of `GtkCellRendererText` (which specify the text colour). The `"foreground"` property take a colour in string form, such as "Orange" or "CornflowerBlue", whereas `"foreground-gdk"` takes a GdkColor argument. It is up to you to decide which one to use - the effect will be the same. The other thing worth mentioning is that most properties have a `"foo-set"` property taking a boolean value as argument, such as `"foreground-set"`. This is useful when you want to have a certain setting have an effect or not. If you set

the "foreground" property, but set "foreground-set" to FALSE, then your foreground color setting will be disregarded. This is useful in cell data functions (see below), or, for example, if you want set the foreground colour to a certain value at start-up, but only want this to be in effect in some columns, but not in others (in which case you could just connect the "foreground-set" property to a model column of type G_TYPE_BOOLEAN with gtk_tree_view_column_add_attribute.

Setting column attributes is the most straight-forward way to get your model data to be displayed. This is usually used whenever you want the data in the model to be displayed exactly as it is in the model.

Another way to get your model data displayed on the screen is to set up cell data functions.

5.3. Cell Data Functions

A cell data function is a function that is called for a specific cell renderer for each single row before that row is rendered. It gives you maximum control over what exactly is going to be rendered, as you can set the cell renderer's properties just like you want to have them. Remember not only to *set* a property if you want it to be active, but also to *unset* a property if it should not be active (and it might have been set in the previous row).

Cell data functions are often used if you want more fine-grained control over what is to be displayed, or if the standard way to display something is not quite like you want it to be. A case in point are floating point numbers. If you want floating point numbers to be displayed in a certain way, say with only one digit after the colon/comma, then you need to use a cell data function. Use gtk_tree_view_column_set_cell_data_func to set up a cell data function for a particular cell renderer. Here is an example:

```
enum
{
  COLUMN_NAME = 0,
  COLUMN_AGE_FLOAT,
  NUM_COLS
};

...

static void
age_cell_data_function (GtkTreeViewColumn *col,
                        GtkCellRenderer   *renderer,
                        GtkTreeModel      *model,
                        GtkTreeIter       *iter,
                        gpointer           user_data)
{
  gfloat  age;
  gchar   buf[20];

  gtk_tree_model_get(model, iter, COLUMN_AGE_FLOAT, &age, -1);

  g_snprintf(buf, sizeof(buf), "%.1f", age);

  g_object_set(renderer, "text", buf, NULL);
}

...

liststore = gtk_list_store_new(NUM_COLS, G_TYPE_STRING, G_TYPE_FLOAT);

col = gtk_tree_view_column_new();

cell = gtk_cell_renderer_text_new();

gtk_tree_view_column_pack_start(col, cell, TRUE);

gtk_tree_view_column_set_cell_data_func(col, cell, age_cell_data_func, NULL, NULL);

...
```

for each row to be rendered by this particular cell renderer, the cell data function is going to be called, which then retrieves the float from the model, and turns it into a string where the float has only one digit after the colon/comma, and renders that with the text cell renderer.

This is only a simple example, you can make cell data functions a lot more complicated if you want to. As always, there is a trade-off to keep in mind though. Your cell data function is going to be called every single time a cell in that (renderer) column is going to be rendered. Go and check how often this function is called in your program if you ever use one. If you do time-consuming operations within a cell data function, things are not going to be fast, especially if you have a lot of rows. The alternative in this case would have been to make an additional column COLUMN_AGE_FLOAT_STRING of type `G_TYPE_STRING`, and to set the float in string form whenever you set the float itself in a row, and then hook up the string column to a text cell renderer using attributes. This way the float to string conversion would only need to be done once. This is a cpu cycles / memory trade-off, and it depends on your particular case which one is more suitable. Things you should probably not do is to convert long strings into UTF8 format in a cell data function, for example.

You might notice that your cell data function is called at times even for rows that are not visible at the moment. This is because the tree view needs to know its total height, and in order to calculate this it needs to know the height of each and every single row, and it can only know that by having it measured, which is going to be slow when you have a lot of rows with different heights (if your rows all have the same height, there should not be any visible delay though).

5.4. GtkCellRendererText and Integer, Boolean and Float Types

It has been said before that, when using attributes to connect data from the model to a cell renderer property, the data in the model column specified in `gtk_tree_view_column_add_attribute` must always be of the same type as the data type that the property requires.

This is usually true, but there is an exception: if you use `gtk_tree_view_column_add_attribute` to connect a text cell renderer's `"text"` property to a model column, the model column does not need to be of `G_TYPE_STRING`, it can also be one of most other fundamental GLib types, e.g. `G_TYPE_BOOLEAN`, `G_TYPE_INT`, `G_TYPE_UINT`, `G_TYPE_LONG`, `G_TYPE_ULONG`, `G_TYPE_INT64`, `G_TYPE_UINT64`, `G_TYPE_FLOAT`, or `G_TYPE_DOUBLE`. The text cell renderer will automatically display the values of these types correctly in the tree view. For example:

```
enum
{
  COL_NAME = 0,
  COL_YEAR_BORN,
  NUM_COLS
};

liststore = gtk_list_store_new(NUM_COLS, G_TYPE_STRING, G_TYPE_UINT);

...

cell = gtk_cell_renderer_text_new();
col = gtk_tree_view_column_new();
gtk_tree_view_column_add_attribute(col, cell, "text", COL_YEAR_BORN);

...
```

Even though the `"text"` property would require a string value, we use a model column of an integer type when setting attributes. The integer will then automatically be converted into a string before the cell renderer property is set [1].

If you are using a floating point type, ie. `G_TYPE_FLOAT` or `G_TYPE_DOUBLE`, there is no way to tell the text cell renderer how many digits after the floating point (or comma) should be rendered. If you only want a certain amount of digits after the point/comma, you will need to use a cell data function.

5.5. GtkCellRendererText, UTF8, and pango markup

All text used in Gtk+-2.0 widgets needs to be in UTF8 encoding, and `GtkCellRendererText` is no exception. Text in plain ASCII is automatically valid UTF8, but as soon as you have special characters that do not exist in plain ASCII (usually characters that are not used in the English language alphabet), they need to be in UTF8 encoding. There are many different character encodings that all specify different ways to tell the computer which character is meant. Gtk+-2.0 uses UTF8, and whenever you have text that is in a different encoding, you need to convert it to UTF8 encoding first, using one of the GLib `g_convert` family of functions. If you only use text input from other Gtk+ widgets, you are on the safe side, as they will return all text in UTF8 as well.

However, if you use 'external' sources of text input, then you must convert that text from the text's encoding (or the user's locale) to UTF8, or it will not be rendered correctly (either not at all, or it will be cut off after the

first invalid character). Filenames are especially hard, because there is no indication whatsoever what character encoding a filename is in (it might have been created when the user was using a different locale, so filename encoding is basically unreliable and broken). You may want to convert to UTF8 with fallback characters in that case. You can check whether a string is valid UTF8 with `g_utf8_validate`. You should, in this author's opinion at least, put these checks into your code at crucial places wherever it is not affecting performance, especially if you are an English-speaking programmer that has little experience with non-English locales. It will make it easier for others and yourself to spot problems with non-English locales later on.

In addition to the "text" property, GtkCellRendererText also has a "markup" property that takes text with pango markup as input. Pango markup allows you to place special tags into a text string that affect the style the text is rendered (see the pango documentation). Basically you can achieve everything you can achieve with the other properties also with pango markup (only that using properties is more efficient and less messy). Pango markup has one distinct advantage though that you cannot achieve with text cell renderer properties: with pango markup, you can change the text style in the middle of the text, so you could, for example, render one part of a text string in bold print, and the rest of the text in normal. Here is an example of a string with pango markup:

```
"You can have text in <b>bold</b> or in a <span color='Orange'>different color</span>"
```

When using the `"markup"` property, you need to take into account that the `"markup"` and `"text"` properties do not seem to be mutually exclusive (I suppose this could be called a bug). In other words: whenever you set `"markup"` (and have used the `"text"` property before), set the `"text"` property to NULL, and vice versa. Example:

```
...

void
foo_cell_data_function ( ... )
{
  ...
  if (foo->is_important)
    g_object_set(renderer, "markup", "<b>important</b>", "text", NULL, NULL);
  else
    g_object_set(renderer, "markup", NULL, "text", "not important", NULL);
  ...
}

...
```

Another thing to keep in mind when using pango markup text is that you might need to escape text if you construct strings with pango markup on the fly using random input data. For example:

```
...

void
foo_cell_data_function ( ... )
{
  gchar *markuptxt;

  ...
  /* This might be problematic if artist_string or title_string
   *   contain markup characters/entities: */
  markuptxt = g_strdup_printf("<b>%s</b> - <i>%s</i>",
                              artist_string, title_string);
  ...
  g_object_set(renderer, "markup", markuptxt, "text", NULL, NULL);
  ...
  g_free(markuptxt);
}

...
```

The above example will not work if artist_string is "Simon & Garfunkel" for example, because the & character is one of the characters that is special. They need to be escaped, so that pango knows that they do not refer to any pango markup, but are just characters. In this case the string would need to be "Simon & Garfunkel" in order to make sense in between the pango markup in which it is going to be pasted. You can escape a string with `g_markup_escape` (and you will need to free the resulting newly-allocated string again with `g_free`).

It is possible to combine both pango markup and text cell renderer properties. Both will be 'added' together to render the string in question, only that the text cell renderer properties will be applied to the whole string. If you set the `"markup"` property to normal text without any pango markup, it will render as normal text just as if you

had used the "text" property. However, as opposed to the "text" property, special characters in the "markup" property text would still need to be escaped, even if you do not use pango markup in the text.

5.6. A Working Example

Here is our example from the very beginning again (with an additional column though), only that the contents of the model are rendered properly on the screen this time. Both attributes and a cell data function are used for demonstration purposes.

```
#include <gtk/gtk.h>

enum
{
  COL_FIRST_NAME = 0,
  COL_LAST_NAME,
  COL_YEAR_BORN,
  NUM_COLS
} ;

static GtkTreeModel *
create_and_fill_model (void)
{
  GtkTreeStore  *treestore;
  GtkTreeIter    toplevel, child;

  treestore = gtk_tree_store_new(NUM_COLS,
                                 G_TYPE_STRING,
                                 G_TYPE_STRING,
                                 G_TYPE_UINT);

  /* Append a top level row and leave it empty */
  gtk_tree_store_append(treestore, &toplevel, NULL);
  gtk_tree_store_set(treestore, &toplevel,
                     COL_FIRST_NAME, "Maria",
                     COL_LAST_NAME, "Incognito",
                     -1);

  /* Append a second top level row, and fill it with some data */
  gtk_tree_store_append(treestore, &toplevel, NULL);
  gtk_tree_store_set(treestore, &toplevel,
                     COL_FIRST_NAME, "Jane",
                     COL_LAST_NAME, "Average",
                     COL_YEAR_BORN, (guint) 1962,
                     -1);

  /* Append a child to the second top level row, and fill in some data */
  gtk_tree_store_append(treestore, &child, &toplevel);
  gtk_tree_store_set(treestore, &child,
                     COL_FIRST_NAME, "Janinita",
                     COL_LAST_NAME, "Average",
                     COL_YEAR_BORN, (guint) 1985,
                     -1);

  return GTK_TREE_MODEL(treestore);
}

void
age_cell_data_func (GtkTreeViewColumn *col,
                    GtkCellRenderer   *renderer,
                    GtkTreeModel      *model,
                    GtkTreeIter       *iter,
                    gpointer           user_data)
{
  guint  year_born;
  guint  year_now = 2003; /* to save code not relevant for the example */
  gchar  buf[64];

  gtk_tree_model_get(model, iter, COL_YEAR_BORN, &year_born, -1);
```

```
  if (year_born <= year_now && year_born > 0)
  {
    guint age = year_now - year_born;

    g_snprintf(buf, sizeof(buf), "%u years old", age);

    g_object_set(renderer, "foreground-set", FALSE, NULL); /* print this normal */
  }
  else
  {
    g_snprintf(buf, sizeof(buf), "age unknown");

    /* make red */
    g_object_set(renderer, "foreground", "Red", "foreground-set", TRUE, NULL);
  }

  g_object_set(renderer, "text", buf, NULL);
}


static GtkWidget *
create_view_and_model (void)
{
  GtkTreeViewColumn   *col;
  GtkCellRenderer     *renderer;
  GtkWidget           *view;
  GtkTreeModel        *model;

  view = gtk_tree_view_new();

  /* --- Column #1 --- */

  col = gtk_tree_view_column_new();

  gtk_tree_view_column_set_title(col, "First Name");

  /* pack tree view column into tree view */
  gtk_tree_view_append_column(GTK_TREE_VIEW(view), col);

  renderer = gtk_cell_renderer_text_new();

  /* pack cell renderer into tree view column */
  gtk_tree_view_column_pack_start(col, renderer, TRUE);

  /* connect 'text' property of the cell renderer to
   *  model column that contains the first name */
  gtk_tree_view_column_add_attribute(col, renderer, "text", COL_FIRST_NAME);


  /* --- Column #2 --- */

  col = gtk_tree_view_column_new();

  gtk_tree_view_column_set_title(col, "Last Name");

  /* pack tree view column into tree view */
  gtk_tree_view_append_column(GTK_TREE_VIEW(view), col);

  renderer = gtk_cell_renderer_text_new();

  /* pack cell renderer into tree view column */
  gtk_tree_view_column_pack_start(col, renderer, TRUE);

  /* connect 'text' property of the cell renderer to
   *  model column that contains the last name */
  gtk_tree_view_column_add_attribute(col, renderer, "text", COL_LAST_NAME);

  /* set 'weight' property of the cell renderer to
   *  bold print (we want all last names in bold) */
  g_object_set(renderer,
               "weight", PANGO_WEIGHT_BOLD,
```

```
                "weight-set", TRUE,
                NULL);


  /* --- Column #3 --- */

  col = gtk_tree_view_column_new();

  gtk_tree_view_column_set_title(col, "Age");

  /* pack tree view column into tree view */
  gtk_tree_view_append_column(GTK_TREE_VIEW(view), col);

  renderer = gtk_cell_renderer_text_new();

  /* pack cell renderer into tree view column */
  gtk_tree_view_column_pack_start(col, renderer, TRUE);

  /* connect a cell data function */
  gtk_tree_view_column_set_cell_data_func(col, renderer, age_cell_data_func, NULL, NULL);


  model = create_and_fill_model();

  gtk_tree_view_set_model(GTK_TREE_VIEW(view), model);

  g_object_unref(model); /* destroy model automatically with view */

  gtk_tree_selection_set_mode(gtk_tree_view_get_selection(GTK_TREE_VIEW(view)),
                              GTK_SELECTION_NONE);

  return view;
}


int
main (int argc, char **argv)
{
  GtkWidget *window;
  GtkWidget *view;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  g_signal_connect(window, "delete_event", gtk_main_quit, NULL); /* dirty */

  view = create_view_and_model();

  gtk_container_add(GTK_CONTAINER(window), view);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;
}
```

5.7. How to Make a Whole Row Bold or Coloured

This seems to be a frequently asked question, so it is worth mentioning it here. You have the two approaches mentioned above: either you use cell data functions, and check in each whether a particular row should be highlighted in a particular way (bold, coloured, whatever), and then set the renderer properties accordingly (and unset them if you want that row to look normal), or you use attributes. Cell data functions are most likely not the right choice in this case though.

If you only want every second line to have a gray background to make it easier for the user to see which data belongs to which line in wide tree views, then you do not have to bother with the stuff mentioned here. Instead

just set the rules hint on the tree view as described in the here, and everything will be done automatically, in colours that conform to the chosen theme even (unless the theme disables rule hints, that is).

Otherwise, the most suitable approach for most cases is that you add two columns to your model, one for the property itself (e.g. a column COL_ROW_COLOR of type `G_TYPE_STRING`), and one for the boolean flag of the property (e.g. a column COL_ROW_COLOR_SET of type `G_TYPE_BOOLEAN`). You would then connect these columns with the `"foreground"` and `"foreground-set"` properties of each renderer. Now, whenever you set a row's COL_ROW_COLOR field to a colour, and set that row's COL_ROW_COLOR_SET field to `TRUE`, then this column will be rendered in the colour of your choice. If you only want either the default text colour or one special other colour, you could even achieve the same thing with just one extra model column: in this case you could just set all renderer's `"foreground"` property to whatever special color you want, and only connect the COL_ROW_COLOR_SET column to all renderer's `"foreground-set"` property using attributes. This works similar with any other attribute, only that you need to adjust the data type for the property of course (e.g. `"weight"` would take a `G_TYPE_INT`, in form of a `PANGO_WEIGHT_FOO` define in this case).

As a general rule, you should not change the text colour or the background colour of a cell unless you have a really good reason for it. To quote Havoc Pennington: "Because colors in GTK+ represent a theme the user has chosen, you should never set colors purely for aesthetic reasons. If users don't like GTK+ gray, they can change it themselves to their favorite shade of orange."

5.8. How to Pack Icons into the Tree View

So far we have only put text in the tree view. While everything you need to know to display icons (in the form of `GdkPixbuf`s) has been introduced in the previous sections, a short example might help to make things clearer. The following code will pack an icon and some text into the same tree view column:

```
enum
{
  COL_ICON = 0,
  COL_TEXT,
  NUM_COLS
};

GtkListStore *
create_liststore(void)
{
  GtkListStore  *store;
  GtkTreeIter    iter;
  GdkPixbuf     *icon;
  GError        *error = NULL;

  store = gtk_list_store_new(2, GDK_TYPE_PIXBUF, G_TYPE_STRING);

  icon = gdk_pixbuf_new_from_file("icon.png", &error);
  if (error)
  {
    g_warning ("Could not load icon: %s\n", error->message);
    g_error_free(error);
    error = NULL;
  }

  gtk_list_store_append(store, &iter);
  gtk_list_store_set(store, &iter,
                     COL_ICON, icon,
                     COL_TEXT, "example",
                     -1);

  return store;
}

GtkWidget *
create_treeview(void)
{
  GtkTreeModel      *model;
  GtkTreeViewColumn *col;
  GtkCellRenderer   *renderer;
  GtkWidget         *view;
```

```
  model = GTK_TREE_MODEL(create_liststore());

  view = gtk_tree_view_new_with_model(model);

  col = gtk_tree_view_column_new();
  gtk_tree_view_column_set_title(col, "Title");

  renderer = gtk_cell_renderer_pixbuf_new();
  gtk_tree_view_column_pack_start(col, renderer, FALSE);
  gtk_tree_view_column_set_attributes(col, renderer,
                                      "pixbuf", COL_ICON,
                                      NULL);

  renderer = gtk_cell_renderer_text_new();
  gtk_tree_view_column_pack_start(col, renderer, TRUE);
  gtk_tree_view_column_set_attributes(col, renderer,
                                      "text", COL_TEXT,
                                      NULL);

  gtk_tree_view_append_column(GTK_TREE_VIEW(view), col);

  gtk_widget_show_all(view);

  return view;
}
```

Note that the tree view will not resize icons for you, but displays them in their original size. If you want to display stock icons instead of `GdkPixbuf`s loaded from file, you should have a look at the `"stock-id"` property of `GtkCellRendererPixbuf` (and your model column should be of type `G_TYPE_STRING`, as all stock IDs are just strings by which to identify the stock icon).

Notes

1. For those interested, the conversion actually takes place within `g_object_set_property`. Before a certain cell is rendered, the tree view column will call `gtk_tree_model_get_value` to set the cell renderer properties according to values stored in the tree model (if any are mapped via `gtk_tree_view_column_add_attribute` or one of the convenience functions that do the same thing), and then pass on the `GValue` retrieved to `g_object_set_property`.

Chapter 6. Selections, Double-Clicks and Context Menus

6.1. Handling Selections

One of the most basic features of a list or tree view is that rows can be selected or unselected. Selections are handled using the `GtkTreeSelection` object of a tree view. Every tree view automatically has a `GtkTreeSelection` associated with it, and you can get it using `gtk_tree_view_get_selection`. Selections are handled completely on the tree view side, which means that the model knows nothing about which rows are selected or not. There is no particular reason why selection handling could not have been implemented with functions that access the tree view widget directly, but for reasons of API cleanliness and code clarity the Gtk+ developers decided to create this special `GtkTreeSelection` object that then internally deals with the tree view widget. You will never need to create a tree selection object, it will be created for you automatically when you create a new tree view. You only need to use said `gtk_tree_view_get_selection` function to get a pointer to the selection object.

There are three ways to deal with tree view selections: either you get a list of the currently selected rows whenever you need it, for example within a context menu function, or you keep track of all select and unselect actions and keep a list of the currently selected rows around for whenever you need them; as a last resort, you can also traverse your list or tree and check each single row for whether it is selected or not (which you need to do if you want all rows that are *not* selected for example).

6.1.1. Selection Modes

You can use `gtk_tree_selection_set_mode` to influence the way that selections are handled. There are four selection modes:

- `GTK_SELECTION_NONE` - no items can be selected
- `GTK_SELECTION_SINGLE` - no more than one item can be selected
- `GTK_SELECTION_BROWSE` - exactly one item is always selected
- `GTK_SELECTION_MULTIPLE` - anything between no item and all items can be selected

6.1.2. Getting the Currently Selected Rows

You can access the currently selected rows either by traversing all selected rows using `gtk_tree_selection_selected_foreach` or get a `GList` of tree paths of the selected rows using `gtk_tree_selection_get_selected_rows`. Note that this function is only available in Gtk+-2.2 and newer, which means that you can't use it or need to reimplement it if you want your application to work with older installations.

If the selection mode you are using is either `GTK_SELECTION_SINGLE` or `GTK_SELECTION_BROWSE`, the most convenient way to get the selected row is the function `gtk_tree_selection_get_selected`, which will return `TRUE` and fill in the specified tree iter with the selected row (if a row is selected), and return `FALSE` otherwise. It is used like this:

```
...

GtkTreeSelection *selection;
GtkTreeModel     *model;
GtkTreeIter       iter;

/* This will only work in single or browse selection mode! */

selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(view));
if (gtk_tree_selection_get_selected(selection, &model, &iter))
{
  gchar *name;

  gtk_tree_model_get (model, &iter, COL_NAME, &name, -1);

  g_print ("selected row is: %s\n", name);

  g_free(name);
}
else
```

```
{
  g_print ("no row selected.\n");
}

...
```

One thing you need to be aware of is that you need to take care when removing rows from the model in a `gtk_tree_selection_selected_foreach` callback, or when looping through the list that `gtk_tree_selection_get_selected_rows` returns (because it contains paths, and when you remove rows in the middle, then the old paths will point to either a non-existing row, or to another row than the one selected). You have two ways around this problem: one way is to use the solution to removing multiple rows that has been described above, ie. to get tree row references for all selected rows and then remove the rows one by one; the other solution is to sort the list of selected tree paths so that the last rows come first in the list, so that you remove rows from the end of the list or tree. You cannot remove rows from within a foreach callback in any case, that is simply not allowed.

Here is an example of how to use `gtk_tree_selection_selected_foreach`:

```
...

gboolean
view_selected_foreach_func (GtkTreeModel  *model,
                            GtkTreePath   *path,
                            GtkTreeIter   *iter,
                            gpointer       userdata)
{
  gchar *name;

  gtk_tree_model_get (model, iter, COL_NAME, &name, -1);

  g_print ("%s is selected\n", name);
}


void
do_something_with_all_selected_rows (GtkWidget *treeview)
{
  GtkTreeSelection  *selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(treeview));

  gtk_tree_selection_selected_foreach(selection, view_selected_foreach_func, NULL);
}


void
create_view (void)
{
  GtkWidget         *view;
  GtkTreeSelection  *selection;

  ...

  view = gtk_tree_view_new();

  ...

  selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(view));

  gtk_tree_selection_set_mode(selection, GTK_SELECTION_MULTIPLE);
  ...
}

...
```

6.1.3. Using Selection Functions

You can set up a custom selection function with `gtk_tree_selection_set_select_function`. This function will then be called every time a row is going to be selected or unselected (meaning: it will be called before the selection status of that row is changed). Selection functions are commonly used for the following things:

1. ... to keep track of the currently selected items (then you maintain a list of selected items yourself). In this case, note again that your selection function is called *before* the row's selection status is changed. In other words: if the row is *going to be* selected, then the boolean path_currently_selected variable that is passed to the selection function is still `FALSE`. Also note that the selection function might not always be called when a row is removed, so you either have to unselect a row before you remove it to make sure your selection function is called and removes the row from your list, or check the validity of a row when you process the selection list you keep. You should not store tree paths in your self-maintained list of of selected rows, because whenever rows are added or removed or the model is resorted the paths might point to other rows. Use tree row references or other unique means of identifying a row instead.
2. ... to tell Gtk+ whether it is allowed to select or unselect that specific row (you should make sure though that it is otherwise obvious to a user whether a row can be selected or not, otherwise the user will be confused if she just cannot select or unselect a row). This is done by returning TRUE or FALSE in the selection function.
3. ... to take additional action whenever a row is selected or unselected.

Yet another simple example:

```
...

gboolean
view_selection_func (GtkTreeSelection *selection,
                     GtkTreeModel     *model,
                     GtkTreePath      *path,
                     gboolean          path_currently_selected,
                     gpointer          userdata)
{
  GtkTreeIter iter;

  if (gtk_tree_model_get_iter(model, &iter, path))
  {
    gchar *name;

    gtk_tree_model_get(model, &iter, COL_NAME, &name, -1);

    if (!path_currently_selected)
    {
      g_print ("%s is going to be selected.\n", name);
    }
    else
    {
      g_print ("%s is going to be unselected.\n", name);
    }

    g_free(name);
  }

  return TRUE; /* allow selection state to change */
}


void
create_view (void)
{
  GtkWidget         *view;
  GtkTreeSelection  *selection;

  ...

  view = gtk_tree_view_new();

  ...

  selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(view));
```

```
  gtk_tree_selection_set_select_function(selection, view_selection_func, NULL, NULL);
  ...
}

...
```

6.1.4. Checking Whether a Row is Selected

You can check whether a given row is selected or not using the functions `gtk_tree_selection_iter_is_selected`. or `gtk_tree_selection_path_is_selected`. If you want to know all rows that are *not* selected, for example, you could just traverse the whole list or tree, and use the above functions to check for each row whether it is selected or not.

6.1.5. Selecting and Unselecting Rows

You can select or unselect rows manually with `gtk_tree_selection_select_iter`, `gtk_tree_selection_select_path`, `gtk_tree_selection_unselect_iter`, `gtk_tree_selection_unselect_path`, `gtk_tree_selection_select_all`, and `gtk_tree_selection_unselect_all` should you ever need to do that.

6.1.6. Getting the Number of Selected Rows

Sometimes you want to know the number of rows that are currently selected (for example to set context menu entries active or inactive before you pop up a context menu). If you are using selection mode `GTK_SELECTION_SINGLE` or `GTK_SELECTION_BROWSE`, this is trivial to check with `gtk_tree_selection_get_selected`, which will return either `TRUE` or `FALSE` (meaning one selected row or no selected row).

If you are using `GTK_SELECTION_MULTIPLE` or want a more general approach that works for all selection modes, `gtk_tree_selection_count_selected_rows` will return the information you are looking for. The only caveat with this function is that it only exists in Gtk+-2.2 and newer, so you will have to reimplement it if you want users with old installations that still use Gtk+-2.0 to be able to use your program as well. Here is a way to reimplement this function:

```
static void
count_foreach_helper (GtkTreeModel *model,
                      GtkTreePath  *path,
                      GtkTreeIter  *iter,
                      gpointer      userdata)
{
  gint *p_count = (gint*) userdata;

  g_assert (p_count != NULL);

  *p_count = *p_count + 1;
}

gint
my_tree_selection_count_selected_rows (GtkTreeSelection *selection)
{
  gint count = 0;

  gtk_tree_selection_selected_foreach(selection, count_foreach_helper, &count);

  return count;
}
```

6.2. Double-Clicks on a Row

Catching double-clicks on a row is quite easy and is done by connecting to a tree view's "row-activated" signal, like this:

```
void
view_onRowActivated (GtkTreeView        *treeview,
                     GtkTreePath        *path,
                     GtkTreeViewColumn  *col,
                     gpointer            userdata)
{
  GtkTreeModel *model;
  GtkTreeIter   iter;

  g_print ("A row has been double-clicked!\n");

  model = gtk_tree_view_get_model(treeview);

  if (gtk_tree_model_get_iter(model, &iter, path))
  {
     gchar *name;

     gtk_tree_model_get(model, &iter, COLUMN_NAME, &name, -1);

     g_print ("Double-clicked row contains name %s\n", name);

     g_free(name);
  }
}

void
create_view (void)
{
  GtkWidget *view;

  view = gtk_tree_view_new();

  ...

  g_signal_connect(view, "row-activated", (GCallback) view_onRowActivated, NULL);

  ...
}
```

6.3. Context Menus on Right Click

Context menus are context-dependent menus that pop up when a user right-clicks on a list or tree and usually let the user do something with the selected items or manipulate the list or tree in other ways.

Right-clicks on a tree view are caught just like mouse button clicks are caught with any other widgets, namely by connecting to the tree view's "button_press_event" signal handler (which is a GtkWidget signal, and as GtkTreeView is derived from GtkWidget it has this signal as well). Additionally, you should also connect to the "popup-menu" signal, so users can access your context menu without a mouse. The "popup-menu" signal is emitted when the user presses Shift-F10. Also, you should make sure that all functions provided in your context menu can also be accessed by other means such as the application's main menu. See the GNOME Human Interface Guidelines (HIG) for more details. Straight from the a-snippet-of-code-says-more-than-a-thousand-words-department, some code to look at:

```
void
view_popup_menu_onDoSomething (GtkWidget *menuitem, gpointer userdata)
{
  /* we passed the view as userdata when we connected the signal */
  GtkTreeView *treeview = GTK_TREE_VIEW(userdata);
```

```
  g_print ("Do something!\n");
}


void
view_popup_menu (GtkWidget *treeview, GdkEventButton *event, gpointer userdata)
{
  GtkWidget *menu, *menuitem;

  menu = gtk_menu_new();

  menuitem = gtk_menu_item_new_with_label("Do something");

  g_signal_connect(menuitem, "activate",
                   (GCallback) view_popup_menu_onDoSomething, treeview);

  gtk_menu_shell_append(GTK_MENU_SHELL(menu), menuitem);

  gtk_widget_show_all(menu);

  /* Note: event can be NULL here when called from view_onPopupMenu;
   *  gdk_event_get_time() accepts a NULL argument */
  gtk_menu_popup(GTK_MENU(menu), NULL, NULL, NULL, NULL,
                 (event != NULL) ? event->button : 0,
                 gdk_event_get_time((GdkEvent*)event));
}


gboolean
view_onButtonPressed (GtkWidget *treeview, GdkEventButton *event, gpointer userdata)
{
  /* single click with the right mouse button? */
  if (event->type == GDK_BUTTON_PRESS  &&  event->button == 3)
  {
    g_print ("Single right click on the tree view.\n");

    /* optional: select row if no row is selected or only
     *  one other row is selected (will only do something
     *  if you set a tree selection mode as described later
     *  in the tutorial) */
    if (1)
    {
      GtkTreeSelection *selection;

      selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(treeview));

      /* Note: gtk_tree_selection_count_selected_rows() does not
       *   exist in gtk+-2.0, only in gtk+ >= v2.2 ! */
      if (gtk_tree_selection_count_selected_rows(selection)  <= 1)
      {
         GtkTreePath *path;

         /* Get tree path for row that was clicked */
         if (gtk_tree_view_get_path_at_pos(GTK_TREE_VIEW(treeview),
                                           (gint) event->x,
                                           (gint) event->y,
                                           &path, NULL, NULL, NULL))
         {
           gtk_tree_selection_unselect_all(selection);
           gtk_tree_selection_select_path(selection, path);
           gtk_tree_path_free(path);
         }
      }
    } /* end of optional bit */

    view_popup_menu(treeview, event, userdata);

    return TRUE; /* we handled this */
  }

  return FALSE; /* we did not handle this */
```

```
}


gboolean
view_onPopupMenu (GtkWidget *treeview, gpointer userdata)
{
  view_popup_menu(treeview, NULL, userdata);

  return TRUE; /* we handled this */
}


void
create_view (void)
{
  GtkWidget *view;

  view = gtk_tree_view_new();

  ...

  g_signal_connect(view, "button-press-event", (GCallback) view_onButtonPressed, NULL);
  g_signal_connect(view, "popup-menu", (GCallback) view_onPopupMenu, NULL);

  ...
}
```

Chapter 7. Sorting

Lists and trees are meant to be sorted. This is done using the `GtkTreeSortable` interface that can be implemented by tree models. 'Interface' means that you can just cast a `GtkTreeModel` into a `GtkTreeSortable` with `GTK_TREE_SORTABLE(model)` and use the documented tree sortable functions on it, just like we did before when we cast a list store to a tree model and used the `gtk_tree_model_foo` family of functions. Both `GtkListStore` and `GtkTreeStore` implement the tree sortable interface.

The most straight forward way to sort a list store or tree store is to directly use the tree sortable interface on them. This will sort the store in place, meaning that rows will actually be reordered in the store if required. This has the advantage that the position of a row in the tree view will always be the same as the position of a row in the model, in other words: a tree path refering to a row in the view will always refer to the same row in the model, so you can get a row's iter easily with `gtk_tree_model_get_iter` using a tree path supplied by the tree view. This is not only convenient, but also sufficient for most scenarios.

However, there are cases when sorting a model in place is not desirable, for example when several tree views display the same model with different sortings, or when the unsorted state of the model has some special meaning and needs to be restored at some point. This is where `GtkTreeModelSort` comes in, which is a special model that maps the unsorted rows of a child model (e.g. a list store or tree store) into a sorted state without changing the child model.

7.1. GtkTreeSortable

The tree sortable interface is fairly simple and should be easy to use. Basically you define a 'sort column ID' integer for every criterion you might want to sort by and tell the tree sortable which function should be called to compare two rows (represented by two tree iters) for every sort ID with `gtk_tree_sortable_set_sort_func`. Then you sort the model by setting the sort column ID and sort order with `gtk_tree_sortable_set_sort_column_id`, and the model will be re-sorted using the compare function you have set up. Your sort column IDs can correspond to your model columns, but they do not have to (you might want to sort according to a criterion that is not directly represented by the data in one single model column, for example). Some code to illustrate this:

```
enum
{
  COL_NAME = 0,
  COL_YEAR_BORN
};


enum
{
  SORTID_NAME = 0,
  SORTID_YEAR
};


GtkTreeModel  *liststore = NULL;


void
toolbar_onSortByYear (void)
{
  GtkTreeSortable *sortable;
  GtkSortType      order;
  gint             sortid;

  sortable = GTK_TREE_SORTABLE(liststore);

  /* If we are already sorting by year, reverse sort order,
   *  otherwise set it to year in ascending order */

  if (gtk_tree_sortable_get_sort_column_id(sortable, &sortid, &order) == TRUE
        &&  sortid == SORTID_YEAR)
  {
    GtkSortType neworder;

    neworder = (order == GTK_SORT_ASCENDING) ? GTK_SORT_DESCENDING : GTK_SORT_ASCENDING;
```

```
    gtk_tree_sortable_set_sort_column_id(sortable, SORTID_YEAR, neworder);
  }
  else
  {
    gtk_tree_sortable_set_sort_column_id(sortable, SORTID_YEAR, GTK_SORT_ASCENDING);
  }
}


/* This is not pretty. Of course you can also use a
 *  separate compare function for each sort ID value */

gint
sort_iter_compare_func (GtkTreeModel *model,
                        GtkTreeIter  *a,
                        GtkTreeIter  *b,
                        gpointer      userdata)
{
  gint sortcol = GPOINTER_TO_INT(userdata);
  gint ret = 0;

  switch (sortcol)
  {
    case SORTID_NAME:
    {
      gchar *name1, *name2;

      gtk_tree_model_get(model, a, COL_NAME, &name1, -1);
      gtk_tree_model_get(model, b, COL_NAME, &name2, -1);

      if (name1 == NULL || name2 == NULL)
      {
        if (name1 == NULL && name2 == NULL)
          break; /* both equal => ret = 0 */

        ret = (name1 == NULL) ? -1 : 1;
      }
      else
      {
        ret = g_utf8_collate(name1,name2);
      }

      g_free(name1);
      g_free(name2);
    }
    break;

    case SORTID_YEAR:
    {
      guint year1, year2;

      gtk_tree_model_get(model, a, COL_YEAR_BORN, &year1, -1);
      gtk_tree_model_get(model, b, COL_YEAR_BORN, &year2, -1);

      if (year1 != year2)
      {
        ret = (year1 > year2) ? 1 : -1;
      }
      /* else both equal => ret = 0 */
    }
    break;

    default:
      g_return_val_if_reached(0);
  }

  return ret;
}


void
```

```
create_list_and_view (void)
{
  GtkTreeSortable *sortable;

  ...

  liststore = gtk_list_store_new(2, G_TYPE_STRING, G_TYPE_UINT);

  sortable = GTK_TREE_SORTABLE(liststore);

  gtk_tree_sortable_set_sort_func(sortable, SORTID_NAME, sort_iter_compare_func,
                                  GINT_TO_POINTER(SORTID_NAME), NULL);

  gtk_tree_sortable_set_sort_func(sortable, SORTID_YEAR, sort_iter_compare_func,
                                  GINT_TO_POINTER(SORTID_YEAR), NULL);

  /* set initial sort order */
  gtk_tree_sortable_set_sort_column_id(sortable, SORTID_NAME, GTK_SORT_ASCENDING);

  ...

  view = gtk_tree_view_new_with_model(liststore);

  ...

}
```

Usually things are a bit easier if you make use of the tree view column headers for sorting, in which case you only need to assign sort column IDs and your compare functions, but do not need to set the current sort column ID or order yourself (see below).

Your tree iter compare function should return a negative value if the row specified by iter a comes before the row specified by iter b, and a positive value if row b comes before row a. It should return 0 if both rows are equal according to your sorting criterion (you might want to use a second sort criterion though to avoid 'jumping' of equal rows when the store gets resorted). Your tree iter compare function should not take the sort order into account, but assume an ascending sort order (otherwise bad things will happen).

7.2. GtkTreeModelSort

`GtkTreeModelSort` is a wrapper tree model. It takes another tree model such as a list store or a tree store as child model, and presents the child model to the 'outside' (ie. a tree view or whoever else is accessing it via the tree model interface) in a sorted state. It does that without changing the order of the rows in the child model. This is useful if you want to display the same model in different tree views with different sorting criteria for each tree view, for example, or if you need to restore the original unsorted state of your store again at some point.

`GtkTreeModelSort` implements the `GtkTreeSortable` interface, so you can treat it just as if it was your data store for sorting purposes. Here is the basic setup with a tree view:

```
...

void
create_list_and_view (void)
{
  ...

  liststore = gtk_list_store_new(2, G_TYPE_STRING, G_TYPE_UINT);

  sortmodel = gtk_tree_model_sort_new_with_model(liststore);

  gtk_tree_sortable_set_sort_func(GTK_TREE_SORTABLE(sortmodel), SORTID_NAME,
                                  sort_func, GINT_TO_POINTER(SORTID_NAME), NULL);

  gtk_tree_sortable_set_sort_func(GTK_TREE_SORTABLE(sortmodel), SORTID_YEAR,
                                  sort_func, GINT_TO_POINTER(SORTID_YEAR), NULL);

  /* set initial sort order */
  gtk_tree_sortable_set_sort_column_id(GTK_TREE_SORTABLE(sortmodel),
```

```
                                        SORTID_NAME, GTK_SORT_ASCENDING);

    ...

    view = gtk_tree_view_new_with_model(sortmodel);

    ...

}

...
```

However, when using the sort tree model, you need to be careful when you use iters and paths with the model. This is because a path pointing to a row in the view (and the sort tree model here) does probably not point to the same row in the child model which is your original list store or tree store, because the row order in the child model is probably different from the sorted order. Similarly, an iter that is valid for the sort tree model is not valid for the child model, and vice versa. You can convert paths and iters from and to the child model using `gtk_tree_model_sort_convert_child_path_to_path`, `gtk_tree_model_sort_convert_child_iter_to_iter`, `gtk_tree_model_sort_convert_path_to_child_path`, and `gtk_tree_model_sort_convert_iter_to_child_iter`. You are unlikely to need these functions frequently though, as you can still directly use `gtk_tree_model_get` on the sort tree model with a path supplied by the tree view.

For the tree view, the sort tree model is the 'real' model - it knows nothing about the sort tree model's child model at all, which means that any path or iter that you get passed from the tree view in a callback or otherwise will refer to the sort tree model, and that you need to pass a path or iter refering to the sort tree model as well if you call tree view functions.

7.3. Sorting and Tree View Column Headers

Unless you have hidden your tree view column headers or use custom tree view column header widgets, each tree view column's header can be made clickable. Clicking on a tree view column's header will then sort the list according to the data in that column. You need to do two things to make this happen: firstly, you need to tell your model which sort function to use for which sort column ID with `gtk_tree_sortable_set_sort_func`. Once you have done this, you tell each tree view column which sort column ID should be active if this column's header is clicked. This is done with `gtk_tree_view_column_set_sort_column_id`.

And that is really all you need to do to get your list or tree sorted. The tree view columns will automatically set the active sort column ID and sort order for you if you click on a column header.

7.4. Case-insensitive String Comparing

As described above in the "GtkCellRendererText, UTF8, and pango markup" section, all strings that are to be displayed in the tree view need to be encoded in UTF8 encoding. All ASCII strings are valid UTF8, but as soon as non-ASCII characters are used, things get a bit tricky and the character encoding matters.

Comparing two ASCII strings ignoring the case is trivial and can be done using `g_ascii_strcasecmp`, for example. `strcasecmp` will usually do the same, only that it is also locale-aware to some extent. The only problem is that a lot of users use locale character encodings that are not UTF8, so `strcasecmp` does not take us very far.

`g_utf8_collate` will compare two strings in UTF8 encoding, but it does not ignore the case. In order to achieve at least half-way correct linguistic case-insensitive sorting, we need to take a two-step approach. For example, we could use `g_utf8_casefold` to convert the strings to compare into a form that is independent of case, and then use `g_utf8_collate` to compare those two strings (note that the strings returned by `g_utf8_casefold` will not resemble the original string in any recognisable way; they will work fine for comparisons though). Alternatively, one could use `g_utf8_strdown` on both strings and then compare the results again with `g_utf8_collate`.

Obviously, all this is not going to be very fast, and adds up if you have a lot of rows. To speed things up, you can create a 'collation key' with `g_utf8_collate_key` and store that in your model as well. A collation key is just a string that does not mean anything to us, but can be used with `strcmp` for string comparison purposes (which is a lot faster than `g_utf8_collate`).

It should be noted that the way `g_utf8_collate` sorts is dependent on the current locale. Make sure you are not working in the 'C' locale (=default, none specified) before you are wondering about weird sorting orders. Check with 'echo $LANG' on a command line what you current locale is set to.

Check out the "Unicode Manipulation" section in the GLib API Reference for more details.

Chapter 8. Editable Cells

8.1. Editable Text Cells

With `GtkCellRendererText` you can not only display text, but you can also allow the user to edit a single cell's text right in the tree view by double-clicking on a cell.

To make this work you need to tell the cell renderer that a cell is editable, which you can do by setting the `"editable"` property of the text cell renderer in question to `TRUE`. You can either do this on a per-row basis (which allows you to set each single cell either editable or not) by connecting the `"editable"` property to a boolean type column in your tree model using attributes; or you can just do a ...

```
g_object_set(renderer, "editable", TRUE, NULL);
```

... when you create the renderer, which sets all rows in that particular renderer column to be editable.

Now that our cells are editable, we also want to be notified when a cell has been edited. This can be achieved by connecting to the cell renderer's `"edited"` signal:

```
g_signal_connect(renderer, "edited", (GCallback) cell_edited_callback, NULL);
```

This callback is then called whenever a cell has been edited. Instead of `NULL` we could have passed a pointer to the model as user data for convenience, as we probably want to store the new value in the model.

The callback for the `"edited"` signal looks like this (the API reference is a bit lacking in this particular case):

```
void        cell_edited_callback (GtkCellRendererText *cell,
                                  gchar               *path_string,
                                  gchar               *new_text,
                                  gpointer             user_data);
```

The tree path is passed to the `"edited"` signal callback in string form. You can convert this into a `GtkTreePath` with `gtk_tree_path_new_from_string`, or convert it into an iter with `gtk_tree_model_get_iter_from_string`.

Note that the cell renderer will not change the data for you in the store. After a cell has been edited, you will only receive an `"edited"` signal. If you do not change the data in the store, the old text will be rendered again as if nothing had happened.

If you have multiple (renderer) columns with editable cells, it is not necessary to have a different callback for each renderer, you can use the same callback for all renderers, and attach some data to each renderer, which you can later retrieve again in the callback to know which renderer/column has been edited. This is done like this, for example:

```
renderer = gtk_cell_renderer_text_new();
...
g_object_set_data(G_OBJECT(renderer), "my_column_num", GUINT_TO_POINTER(COLUMN_NAME));

...

renderer = gtk_cell_renderer_text_new();
...
g_object_set_data(G_OBJECT(renderer), "my_column_num", GUINT_TO_POINTER(COLUMN_YEAR_OF_BIRTH));

...
```

where COLUMN_NAME and COLUMN_YEAR_OF_BIRTH are enum values. In your callback you can then get the column number with

```
guint column_number = GPOINTER_TO_UINT(g_object_get_data(G_OBJECT(renderer), "my_column_num"));
```

You can use this mechanism to attach all kinds of custom data to any object or widget, with a string identifier to your liking.

A good example for editable cells is in gtk-demo, which is part of the Gtk+ source code tree (in gtk+-2.x.y/demos/gtk-demo).

8.1.1. Setting the cursor to a specific cell

You can move the cursor to a specific cell in a tree view with `gtk_tree_view_set_cursor` (or `gtk_tree_view_set_cursor_on_cell` if you have multiple editable cell renderers packed into one tree view column), and start editing the cell if you want to. Similarly, you can get the current row and focus column with `gtk_tree_view_get_cursor`. Use `gtk_widget_grab_focus(treeview)` will make sure that the tree view has the keyboard focus.

As the API reference points out, the tree view needs to be realised for cell editing to happen. In other words: If you want to start editing a specific cell right at program startup, you need to set up an idle timeout with `g_idle_add` that does this for you as soon as the window and everything else has been realised (return `FALSE` in the timeout to make it run only once). Alternatively you could connect to the `"realize"` signal of the treeview with `g_signal_connect_after` to achieve the same thing.

Connect to the tree view's `"cursor-changed"` and/or `"move-cursor"` signals to keep track of the current position of the cursor.

8.2. Editable Toggle and Radio Button Cells

Just like you can set a `GtkCellRendererText` editable, you can specify whether a `GtkCellRendererToggle` should change its state when clicked by setting the `"activatable"` property - either when you create the renderer (in which case all cells in that column will be clickable) or by connecting the renderer property to a model column of boolean type via attributes.

Connect to the `"toggled"` signal of the toggle cell renderer to be notified when the user clicks on a toggle button (or radio button). The user click will not change the value in the store, or the appearance of the value rendered. The toggle button will only change state when you update the value in the store. Until then it will be in an "inconsistent" state, which is also why you should read the current value of that cell from the model, and not from the cell renderer.

The callback for the `"toggled"` signal looks like this (the API reference is a bit lacking in this particular case):

```
void        cell_toggled_callback (GtkCellRendererToggle *cell,
                                   gchar                 *path_string,
                                   gpointer               user_data);
```

Just like with the `"edited"` signal of the text cell renderer, the tree path is passed to the `"toggled"` signal callback in string form. You can convert this into a `GtkTreePath` with `gtk_tree_path_new_from_string`, or convert it into an iter with `gtk_tree_model_get_iter_from_string`.

8.3. Editable Spin Button Cells

Even though `GtkSpinButton` implements the `GtkCellEditable` interface (as does `GtkEntry`), there is no easy way to get a cell renderer that uses a spin button instead of a normal entry when in editing mode.

To get this functionality, you need to either write a new cell renderer that works very similar to `GtkCellRendererText`, or you need to write a new cell renderer class that derives from the text cell renderer and changes the behaviour in editing mode.

The cleanest solution would probably be to write a 'CellRendererNumeric' that does everything that the text cell renderer does, only that it has a float type property instead of the `"text"` property, and an additional digits property. However, no one seems to have done this yet, so you need to either write one, or find another solution to get spin buttons in editing mode.

Among this tutorial's code examples there is a hackish CellRendererSpin implementation which is based on `GtkCellRendererText` and shows spin buttons in editing mode. The implementation is not very refined though, so you need to make sure it works in your particular context, and modify it as needed.

Chapter 9. Miscellaneous

This section deals with issues and questions that did not seem to fit in anywhere else. If you can think of something else that should be dealt with here, do not hesitate to send a mail to <tim at centricular dot net>.

9.1. Getting the Column Number from a Tree View Column Widget

Signal callbacks often only get passed a pointer to a `GtkTreeViewColumn` when the application programmer really just wants to know which column *number* was affected. There are two ways to find out the position of a column within the tree view. One way is to write a small helper function that looks up the column number from a given tree view column object, like this for example: [1].

```
/* Returns column number or -1 if not found or on error */

gint
get_col_number_from_tree_view_column (GtkTreeViewColumn *col)
{
  GList *cols;
  gint   num;

  g_return_val_if_fail ( col != NULL, -1 );
  g_return_val_if_fail ( col->tree_view != NULL, -1 );

  cols = gtk_tree_view_get_columns(GTK_TREE_VIEW(col->tree_view));

  num = g_list_index(cols, (gpointer) col);

  g_list_free(cols);

  return num;
}
```

Alternatively, it is possible to use `g_object_set_data` and `g_object_get_data` on the tree view column in order to identify which column it is. This also has the advantage that you can still keep track of your columns even if the columns get re-ordered within the tree view (a feature which is usually disabled though). Use like this:

```
...

enum
{
  COL_FIRSTNAME,
  COL_SURNAME,
};

...

void
some_callback (GtkWidget *treeview, ..., GtkTreeViewColumn *col, ...)
{
  guint colnum = GPOINTER_TO_UINT(g_object_get_data(G_OBJECT(col), "columnnum"));

  ...
}

void
create_view(void)
{
  ...
  col = gtk_tree_view_column_new();
  g_object_set_data(G_OBJECT(col), "columnnum", GUINT_TO_POINTER(COL_FIRSTNAME));
  ...
  col = gtk_tree_view_column_new();
  g_object_set_data(G_OBJECT(col), "columnnum", GUINT_TO_POINTER(COL_SURNAME));
  ...
}
```

`"columnnum"` is a random string in the above example - you can use whatever string you want instead, or store multiple bits of data (with different string identifiers of course). Of course you can also combine both approaches,

as they do slightly different things (the first tracks the 'physical' position of a column within the tree view, the second tracks the 'meaning' of a column to you, independent of its position within the view).

9.2. Column Expander Visibility

9.2.1. Hiding the Column Expander

Is it possible to hide the column expander completely? Yes and no. What follows, is probably a dirty hack at best and there is no guarantee that it will work with upcoming Gtk+ versions or even with all past versions (although the latter is easy enough to test of course).

What you can do to hide the column expander is to create an empty tree view column (containing empty strings, for example) and make this the first column in the tree view. Then you can hide that column with `gtk_tree_view_column_set_visible`. You will notice that the expander column will now automatically move to the formerly second, now first, visible column in the tree view. However, if you call `gtk_tree_view_set_expander_column` right after the call to `_set_visible`, then the expander will move back to the hidden column, and no expander is visible any longer.

This means of course that you will have to take care of expanding and collapsing rows yourself and use the appropriate tree view functions. While it is at last thinkable that one could implement custom expanders using custom cell rendereres or pixbuf cell renderers, this is probably a task that will keep you busy for more than five minutes. Keep those head ache tablets nearby if you attempt it anyway...

9.2.2. Forcing Column Expander Visibility

There are situations where an expander should be visible even if the row in question does not have any children yet, for instance when part of a model should only be loaded on request when a node gets expanded (e.g. to show the contents of a directory). This is not possible. An expander is only shown if a node has children.

A work-around for this problem exists however: simply attach an empty child row and set the node to collapsed state. Then listen for the tree view's `"row-expanded"` signal, and fill the contents of the already existing row with the first new row, then append new child rows. See this mailing list thread for more details.

9.3. Getting the Cell Renderer a Click Event Happened On

It seems that in many cases when people want to know the cell renderer a click event happened on, they do not really need to know the cell renderer, but rather want to modify an individual cell in a particular column. For this you do not need to know the cell renderer. Use `gtk_tree_view_get_path_at_pos` to get a tree path from the x and y coordinates of the button event that is passed to you in a `"button-press-event"` signal callback (if you use the `"row-activated"` signal to catch double-clicks you get the tree path passed directly into the callback function). Then convert that tree path into an iter using `gtk_tree_model_get_iter` and modify the data in the cell you want to modify with `gtk_list_store_set` or `gtk_tree_store_set`.

If you really do need to know the cell renderer where a button press event happened, that is a bit more tricky. Here is a suggestion on how to approach this issue (the function has not been well-tested and might not work correctly if the content rendered by one renderer in different columns varies in width; please send suggestions on how to fix or improve this function to the author):

```
static gboolean
tree_view_get_cell_from_pos(GtkTreeView *view, guint x, guint y, GtkCellRenderer **cell)
{
GtkTreeViewColumn *col = NULL;
GList             *node, *columns, *cells;
guint              colx = 0;

g_return_val_if_fail ( view != NULL, FALSE );
g_return_val_if_fail ( cell != NULL, FALSE );

/* (1) find column and column x relative to tree view coordinates */

columns = gtk_tree_view_get_columns(view);

for (node = columns;  node != NULL && col == NULL;  node = node->next)
```

```
{
GtkTreeViewColumn *checkcol = (GtkTreeViewColumn*) node->data;

if (x >= colx  &&  x < (colx + checkcol->width))
col = checkcol;
else
colx += checkcol->width;
}

g_list_free(columns);

if (col == NULL)
return FALSE; /* not found */

/* (2) find the cell renderer within the column */

cells = gtk_tree_view_column_get_cell_renderers(col);

for (node = cells;  node != NULL;  node = node->next)
{
GtkCellRenderer *checkcell = (GtkCellRenderer*) node->data;
guint            width = 0, height = 0;

/* Will this work for all packing modes? doesn't that
 *  return a random width depending on the last content
 * rendered? */
gtk_cell_renderer_get_size(checkcell, GTK_WIDGET(view), NULL, NULL, NULL, &width, NULL);

if (x >= colx && x < (colx + width))
{
*cell = checkcell;
g_list_free(cells);
return TRUE;
}

colx += width;
}

g_list_free(cells);
return FALSE; /* not found */
}

static gboolean
onButtonPress (GtkWidget *view, GdkEventButton *bevent, gpointer data)
{
GtkCellRenderer *renderer = NULL;

if (tree_view_get_cell_from_pos(GTK_TREE_VIEW(view), bevent->x, bevent->y, &renderer))
g_print ("Renderer found\n");
else
g_print ("Renderer not found!\n");
}
```

9.4. Glade and Tree Views

A frequently asked question is how you can add columns to a `GtkTreeView` in Glade. [2] The answer is basically that you don't, and that you can't. The only thing glade/libglade can do for you is to create the `GtkTreeView` for you with nothing in it. You will need to look up the tree view widget at the start of your application (after the interface has been created of course), and connect your list store or tree store to it. Then you will need to add `GtkTreeViewColumn`s and cell renderers to display the information from the model as you want it to be displayed. You will need to do all that from within your application.

An alternative approach is to derive your own special widget from `GtkTreeView` that sets up everything as you want it to, and then use the 'custom widget' function in glade. Of course this still means that you have to write all the code to fill in the columns and cell renderers and to create the model yourself.

Notes

1. This function has been inspired by this mailing list message (thanks to Ken Rastatter for the link and the topic suggestion).
2. Do *not* use Glade to generate code for you. Use Glade to create the interface. It will save the interface into a .glade file in XML format. You can then use libglade2 to construct your interface (windows etc.) from that .glade file. See this mailing list message for a short discussion about why you should avoid Glade code generation.

Chapter 10. Drag'n'Drop (DnD) **** needs revision ***

****** NEEDS REVISION

This section needs revision more than any other section. If you know anything about tree view drag'n'drop, you probably know more than the author of this text. Please give some feedback in that case.

If you want to dive into treeview drag'n'drop, you might want to check out Owen Taylor's mail on that topic. It might not be completely identical to what has actually been implemented, but it gives a great overview, and provides more information than the docs do.

In addition to the standard Gtk+ Drag and Drop mechanisms that work with any widget, there are special Drag and Drop mechanisms just for the tree view widget. You usually want to use the tree-view specific Drag-and-Drop framework.

10.1. Drag'n'Dropping Row-Unrelated Data to and from a Tree View from other Windows or Widgets

Drag'n'Dropping general information from or to a tree view widget works just like it works with any other widget and involves the standard Gtk+ Drag and Drop mechanisms. If you use this, you can receive drops to or initiate drags from anywhere in your tree view (including empty sections). This is not row- or column-specific and *is most likely not want you want*. Nevertheless, here is a small example of a tree view in which you can drag'n'drop URIs from other applications (browsers, for example), with the dropped URIs just being appended to the list (note that usually you would probably rather want to set up your whole window as a target then and not just the tree view widget):

```
#include <gtk/gtk.h>

enum
{
  COL_URI = 0,
  NUM_COLS
} ;


void
view_onDragDataReceived(GtkWidget *wgt, GdkDragContext *context, int x, int y,
                        GtkSelectionData *seldata, guint info, guint time,
                        gpointer userdata)
{
  GtkTreeModel *model;
  GtkTreeIter   iter;

  model = GTK_TREE_MODEL(userdata);

  gtk_list_store_append(GTK_LIST_STORE(model), &iter);

  gtk_list_store_set(GTK_LIST_STORE(model), &iter, COL_URI, (gchar*)seldata->data, -1);
}

static GtkWidget *
create_view_and_model (void)
{
  GtkTreeViewColumn   *col;
  GtkCellRenderer     *renderer;
  GtkListStore        *liststore;
  GtkWidget           *view;

  liststore = gtk_list_store_new(NUM_COLS, G_TYPE_STRING);

  view = gtk_tree_view_new_with_model(GTK_TREE_MODEL(liststore));

  g_object_unref(liststore); /* destroy model with view */

  col = gtk_tree_view_column_new();
  renderer = gtk_cell_renderer_text_new();

  gtk_tree_view_column_set_title(col, "URI");
  gtk_tree_view_append_column(GTK_TREE_VIEW(view), col);
```

```
  gtk_tree_view_column_pack_start(col, renderer, TRUE);
  gtk_tree_view_column_add_attribute(col, renderer, "text", COL_URI);

  gtk_tree_selection_set_mode(gtk_tree_view_get_selection(GTK_TREE_VIEW(view)),
                              GTK_SELECTION_SINGLE);

  /* Make tree view a destination for Drag'n'Drop */
  if (1)
  {
    enum
    {
      TARGET_STRING,
      TARGET_URL
    };

    static GtkTargetEntry targetentries[] =
    {
      { "STRING",        0, TARGET_STRING },
      { "text/plain",    0, TARGET_STRING },
      { "text/uri-list", 0, TARGET_URL },
    };

    gtk_drag_dest_set(view, GTK_DEST_DEFAULT_ALL, targetentries, 3,
                      GDK_ACTION_COPY|GDK_ACTION_MOVE|GDK_ACTION_LINK);

    g_signal_connect(view, "drag_data_received",
                     G_CALLBACK(view_onDragDataReceived), liststore);
  }

  return view;
}

int
main (int argc, char **argv)
{
  GtkWidget *window, *vbox, *view, *label;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  g_signal_connect(window, "delete_event", gtk_main_quit, NULL); /* dirty */
  gtk_window_set_default_size(GTK_WINDOW(window), 400, 200);

  vbox = gtk_vbox_new(FALSE, 0);
  gtk_container_add(GTK_CONTAINER(window), vbox);

  label = gtk_label_new("\nDrag and drop links from your browser into the tree view.\n");
  gtk_box_pack_start(GTK_BOX(vbox), label, FALSE, FALSE, 0);

  view = create_view_and_model();
  gtk_box_pack_start(GTK_BOX(vbox), view, TRUE, TRUE, 0);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;
}
```

If you are receiving drops into a tree view, you can connect to the view's `"drag-motion"` signal to track the mouse pointer while it is in a drag and drop operation over the tree view. This is useful for example if you want to expand a collapsed node in a tree when the mouse hovers above the node for a certain amount of time during a drag'n'drop operation. Here is an example of how to achieve this:

```
/**********************************************************************
 *
 *   onDragMotion_expand_timeout
 *
 *   Timeout used to make sure that we expand rows only
 *    after hovering about them for a certain amount
 *    of time while doing Drag'n'Drop
```

```
 *
 ************************************************************************/

gboolean
onDragMotion_expand_timeout (GtkTreePath **path)
{
        g_return_val_if_fail (  path != NULL, FALSE );
        g_return_val_if_fail ( *path != NULL, FALSE );

        gtk_tree_view_expand_row(GTK_TREE_VIEW(view), *path, FALSE);

        return FALSE; /* only call once */
}


/************************************************************************
 *
 *   view_onDragMotion: we don't want to expand unexpanded nodes
 *                      immediately when the mouse pointer passes across
 *                      them during DnD. Instead, we only want to expand
 *                      the node if the pointer has been hovering above the
 *                      node for at least 1.5 seconds or so. To achieve this,
 *                      we use a timeout that is removed whenever the row
 *                      in focus changes.
 *
 ************************************************************************/

static gboolean
view_onDragMotion (GtkWidget *widget, GdkDragContext *context, gint x,
                   gint y, guint time, gpointer data)
{
  static GtkTreePath  *lastpath;  /* NULL */
  GtkTreePath         *path = NULL;

  if (gtk_tree_view_get_path_at_pos(GTK_TREE_VIEW(widget), x, y, &path, NULL, NULL, NULL))
  {
    if (!lastpath || ((lastpath) && gtk_tree_path_compare(lastpath, path) != 0))
    {
      (void) g_source_remove_by_user_data(&lastpath);

      if (!gtk_tree_view_row_expanded(GTK_TREE_VIEW(widget), path))
      {
        /* 1500 = 1.5 secs */
        g_timeout_add(1500, (GSourceFunc) onDragMotion_expand_timeout, &lastpath);
      }
    }
  }
  else
  {
    g_source_remove_by_user_data(&lastpath);
  }

  if (lastpath)
    gtk_tree_path_free(lastpath);

  lastpath = path;

  return TRUE;
}
```

Connect to the view's `"drag-drop"` signal to be called when the drop happens. You can translate the coordinates provided into a tree path with `gtk_tree_view_get_path_at_pos`.

10.2. Dragging Rows Around Within a Tree **** TODO ***

****** TODO

Both `GtkListStore` and `GtkTreeStore` implement the `GtkTreeDragDest` and `GtkTreeDragSource` interfaces, which means that they have in-built support for row reordering. You need to call `gtk_tree_view_set_reorderable` to activate this, and then connect to the tree model's signals to catch the reorderings that take place.

*** SOMEONE NEEDS TO WRITE THIS SECTION (I have never gotten this to work in a way that does not suck, ie. where one does not have to place the row to move exact to the pixel on the target row).

10.3. Dragging Rows from One Tree to Another **** TODO ***

****** TODO (is this possible at all in Gtk+ <= 2.2?)

Chapter 11. Writing Custom Models

11.1. When is a Custom Model Useful?

A custom tree model gives you complete control over your data and how it is represented to the outside (e.g. to the tree view widget). It has the advantage that you can store, access and modify your data exactly how you need it, and you can optimise the way your data is stored and retrieved, as you can write your own functions to access your data and need not rely solely on the `gtk_tree_model_get`. A model tailored to your needs will probably also be a lot faster than the generic list and tree stores that come with gtk and that have been designed with flexibility in mind.

Another case where a custom model might come in handy is when you have all your data already stored in an external tree-like structure (for example a libxml2 XML tree) and only want to display that structure. Then you could write a custom model that maps that structure to a tree model (which is probably not quite as trivial as it sounds though).

Using a custom model you could also implement a filter model that only displays certain rows according to some filter criterion instead of displaying all rows (Gtk+-2.4 has a filter model, `GtkTreeModelFilter`, that does exactly that and much more, but you might want to implement this yourself anyway. If you need to use GtkTreeModelFilter in Gtk-2.0 or Gtk-2.2, check out the code examples of this tutorial - there is GuiTreeModelFilter, which is basically just the original GtkTreeModelFilter but has been made to work with earlier Gtk-2.x versions and has a different name space, so that it does not clash with Gtk-2.4).

However, all this comes at a cost: you are unlikely to write a useful custom model in less than a thousand lines, unless you strip all newline characters. Writing a custom model is not as difficult as it might sound though, and it may well be worth the effort, not least because it will result in much saner code if you have a lot of data to keep track of.

11.2. What Does Writing a Custom Model Involve?

Basically, all you need to do is to write a new GObject that implements the `GtkTreeModel` interface, `GtkTreeModelIface`. Intimate knowledge about the GLib GObject system is not a requirement - you just need to copy some boilerplate code and modify it a bit. The core of your custom tree model is your own implementation of a couple of `gtk_tree_model_foo` functions that reveal the structure of your data, ie. how many rows there are, how many children a row has, how many columns there are and what type of data they contain. Furthermore, you need to provide functions that convert a tree path to a tree iter and a tree iter to a tree path. Additionally, you should provide some functions to add and remove rows to your custom model, but those are only ever used by yourself anyway, so they do not fall within the scope of the tree model interface.

The functions you *need* to implement are:

- `get_flags` - tells the outside that your model has certain special characterstics, like persistent iters.
- `get_n_columns` - how many data fields per row are visible to the outside that uses gtk_tree_model_get, e.g. cell renderer attributes
- `get_column_type` - what type of data is stored in a data field (model column) that is visible to the outside
- `get_iter` - take a tree path and fill an iter structure so that you know which row it refers to
- `get_path` - take an iter and convert it into a tree path, ie. the 'physical' position within the model
- `get_value` - retrieve data from a row
- `iter_next` - take an iter structure and make it point to the next row
- `iter_children` - tell whether the row represented by a given iter has any children or not
- `iter_n_children` - tell how many children a row represented by a given iter has
- `iter_nth_child` - set a given iter structure to the n-th child of a given parent iter
- `iter_parent` - set a given iter structure to the parent of a given child iter

It is up to you to decide which of your data you make 'visible' to the outside in form of model columns and which not. You can always implement functions specific to your custom model that will return any data in any form you desire. You only *need* to make data 'visble' to the outside via the GType and GValue system if you want the tree view components to access it (e.g. when setting cell renderer attributes).

11.3. Example: A Simple Custom List Model

What follows is the outline for a simple custom list model. You can find the complete source code for this model below. The beginning of the code might look a bit scary, but you can just skip most of the GObject and GType stuff and proceed to the heart of the custom list, ie. the implementation of the tree model functions.

Our list model is represented by a simple list of records, where each row corresponds to a `CustomRecord` structure which keeps track of the data we are interested in. For now, we only want to keep track of persons' names and years of birth (usually this would not really justify a custom model, but this is still just an example). It is trivial to extend the model to deal with additional fields in the `CustomRecord` structure.

Within the model, more precisely: the `CustomList` structure, the list is stored as a pointer array, which not only provides fast access to the n-th record in the list, but also comes in handy later on when we add sorting. Apart from that, any other kind of list-specific data would go in this structure as well (the active sort column, for example, or hash tables to speed up searching for a specific row, etc.).

Each row in our list is represented by a `CustomRecord` structure. You can store whatever other data you need in that structure. How you make row data available is up to you. Either you export it via the tree model interface using the GValue system, so that you can use `gtk_tree_model_get` to retrieve your data, or you provide custom model-specific functions to retrieve data, for example `custom_list_get_name`, taking a tree iter or a tree path as argument. Of course you can also do both.

Furthermore, you will need to provide your own functions to add rows, remove rows, and set or modify row data, and you need to let the view and others know whenever something changes in your model by emitting the appropriate signals via the provided tree model functions.

Some thought should go into how exactly you fill the `GtkTreeIter` fields of the tree iters used by your model. You have three pointer fields at your disposal. These should be filled so that you can easily identify the row given the iter, and should also facilitate access to the next row and the parent row (if any). If your model advertises to have persistent iters, you need to make sure that the content of your iters is perfectly valid even if the user stores it somewhere for later use and the model gets changed or reordered. The 'stamp' field of a tree iter should be filled by a random model-instance-specific integer that was assigned to the model when it was created. This way you can catch iters that do not belong to your model. If your model does not have persistent iters, then you should change the model's stamp whenever the model changes, so that you can catch invalid iters that get passed to your functions (note: in the code below we do not check the stamp of the iters in order to save a couple of lines of code to print here).

In our specific example, we simply store a pointer to a row's `CustomRecord` structure in our model's tree iters, which is valid as long as the row exists. Additionally we store the position of a row within the list in the `CustomRecord` as well, which is not only intuitive, but is also useful later on when we resort the list.

If you want to store an integer value in an iter's fields, you should use GLib's `GINT_TO_POINTER` and `GPOINTER_TO_INT` macros for that.

Let's look at the code sections in a bit more detail:

11.3.1. custom-list.h

The header file for our custom list model defines some standard type casts and type check macros, our `CustomRecord` structure, our `CustomList` structure, and some enums for the model columns we are exporting.

The `CustomRecord` structure represents one row, while the `CustomList` structure contains all list-specific data. You can add additional fields to both structures without problems. For example, you might need a function that quickly looks up rows given the name or year of birth, for which additional hashtables or so might come in handy (which you would need to keep up to date as you insert, modify or remove rows of course).

The only function you must export is `custom_list_get_type`, as it is used by the type check and type cast macros that are also defined in the header file. Additionally, we want to export a function to create one instance of our custom model, and a function that adds some rows. You will probably add more custom model-specific functions to modify the model as you extend it to suit your needs.

11.3.2. custom-list.c

Firstly, we need some boilerplate code to register our custom model with the GObject type system. You can skip this section and proceed to the tree model implementation.

Functions of interested in this section are `custom_list_init` and `custom_list_get_type`. In `custom_list_init` we define what data type our exported model columns have, and how many columns we export. Towards the end of `custom_list_get_type` we register the `GtkTreeModel` interface with our custom model object. This is where

we can also register additional interfaces (e.g. `GtkTreeSortable` or one of the Drag'n'Drop interfaces) that we want to implement.

In `custom_list_tree_model_init` we override those tree model functions that we need to implement with our own functions. If it is beneficial for your model to know which rows are currently displayed in the tree view (for example for caching), you might want to override the `ref_node` and `unref_node` functions as well.

Let's have a look at the heart of the object type registration:

```
GType
custom_list_get_type (void)
{
  static GType custom_list_type = 0;

  if (custom_list_type)
    return custom_list_type;

  /* Some boilerplate type registration stuff */
  if (1)
  {
    static const GTypeInfo custom_list_info =
    {
      sizeof (CustomListClass),
      NULL,                                          /* base_init */
      NULL,                                          /* base_finalize */
      (GClassInitFunc) custom_list_class_init,
      NULL,                                          /* class finalize */
      NULL,                                          /* class_data */
      sizeof (CustomList),
      0,                                            /* n_preallocs */
      (GInstanceInitFunc) custom_list_init
    };

    custom_list_type = g_type_register_static (G_TYPE_OBJECT, "CustomList",
                                               &custom_list_info, (GTypeFlags)0);
  }

  /* Here we register our GtkTreeModel interface with the type system */
  if (1)
  {
    static const GInterfaceInfo tree_model_info =
    {
      (GInterfaceInitFunc) custom_list_tree_model_init,
      NULL,
      NULL
    };

    g_type_add_interface_static (custom_list_type, GTK_TYPE_TREE_MODEL, &tree_model_info);
  }

  return custom_list_type;
}
```

Here we just return the type assigned to our custom list by the type system if we have already registered it. If not, we register it and save the type. Of the three callbacks that we pass to the type system, only two are of immediate interest to us, namely `custom_list_tree_model_init` and `custom_list_init`.

In `custom_list_tree_model_init` we fill the tree model interface structure with pointers to our own functions (at least the ones we implement):

```
static void
custom_list_tree_model_init (GtkTreeModelIface *iface)
{
  /* Here we override the GtkTreeModel
   *  interface functions that we implement */
  iface->get_flags       = custom_list_get_flags;
  iface->get_n_columns   = custom_list_get_n_columns;
  iface->get_column_type = custom_list_get_column_type;
  iface->get_iter        = custom_list_get_iter;
  iface->get_path        = custom_list_get_path;
  iface->get_value       = custom_list_get_value;
```

```
  iface->iter_next       = custom_list_iter_next;
  iface->iter_children   = custom_list_iter_children;
  iface->iter_has_child  = custom_list_iter_has_child;
  iface->iter_n_children = custom_list_iter_n_children;
  iface->iter_nth_child  = custom_list_iter_nth_child;
  iface->iter_parent     = custom_list_iter_parent;
}
```

In `custom_list_init` we initialised the custom list structure to sensible default values. This function will be called whenever a new instance of our custom list is created, which we do in `custom_list_new`.

`custom_list_finalize` is called just before one of our lists is going to be destroyed. You should free all resources that you have dynamically allocated in there.

Having taken care of all the type system stuff, we now come to the heart of our custom model, namely the tree model implementation. Our tree model functions need to behave exactly as the API reference requires them to behave, including all special cases, otherwise things will not work. Here is a list of links to the API reference descriptions of the functions we are implementing:

- gtk_tree_model_get_flags
- gtk_tree_model_get_n_columns
- gtk_tree_model_get_column_type
- gtk_tree_model_get_iter
- gtk_tree_model_get_path
- gtk_tree_model_get_value
- gtk_tree_model_iter_next
- gtk_tree_model_iter_children
- gtk_tree_model_iter_has_child
- gtk_tree_model_iter_n_children
- gtk_tree_model_iter_nth_child
- gtk_tree_model_iter_parent

Almost all functions are more or less straight-forward and self-explanatory in connection with the API reference descriptions, so you should be able to jump right into the code and see how it works.

After the tree model implementation we have those functions that are specific to our custom model. `custom_list_new` will create a new custom list for us, and `custom_list_append_record` will append a new record to the end of the list. Note the call to `gtk_tree_model_row_inserted` at the end of our append function, which emits a `"row-inserted"` signal on the model and informs all interested objects (tree views, tree row references) that a new row has been inserted, and where it has been inserted.

You will need to emit tree model signals whenever something changes, e.g. rows are inserted, removed, or re-ordered, or when a row changes from a child-less row to a row which has children, or if a row's data changes. Here are the functions you need to use in those cases (we only implement row insertions here - other cases are left as an exercise for the reader):

- gtk_tree_model_row_inserted
- gtk_tree_model_row_changed (makes tree view redraw that row)
- gtk_tree_model_row_has_child_toggled
- gtk_tree_model_row_deleted
- gtk_tree_model_rows_reordered (note bug 124790)

And that is all you have to do to write a custom model.

11.4. From a List to a Tree

Writing a custom model for a tree is a bit trickier than a simple list model, but follows the same pattern. Basically you just need to extend the above model to cater for the case of children. You could do this by keeping track of the whole tree hierarchy in the `CustomList` structure, using GLib N-ary trees for example, or you could do this by keeping track of each row's children within the row's `CustomRecord` structure, keeping only a pointer to the (invisible) root record in the `CustomList` structure.

TODO: do we need anything else here?

11.5. Additional interfaces, here: the GtkTreeSortable interface

A custom model can implement additional interfaces to extend its functionality. Additional interfaces are:

- GtkTreeSortableIface
- GtkTreeDragDestIface
- GtkTreeDragSourceIface

Here, we will show how to implement additional interfaces at the example of the `GtkTreeSortable` interface, which we will implement only partially (enough to make it functional and useful though).

Three things are necessary to add another interface: we will need to register the interface with our model in `custom_list_get_type`, provide an interface init function where we set the interface to our own implementation of the interface functions, and then provide the implementation of those functions.

Firstly, we need to provide the function prototypes for our functions at the beginning of the file:

```
/* custom-list.c */

...

/* -- GtkTreeSortable interface functions -- */

static gboolean     custom_list_sortable_get_sort_column_id (GtkTreeSortable *sortable,
                                                             gint            *sort_col_id,
                                                             GtkSortType     *order);

static void         custom_list_sortable_set_sort_column_id (GtkTreeSortable *sortable,
                                                             gint             sort_col_id,
                                                             GtkSortType      order);

static void         custom_list_sortable_set_sort_func (GtkTreeSortable        *sortable,
                                                        gint                    sort_col_id,
                                                        GtkTreeIterCompareFunc  sort_func,
                                                        gpointer                user_data,
                                                        GtkDestroyNotify        destroy_func);

static void         custom_list_sortable_set_default_sort_func (GtkTreeSortable        *sortable,
                                                                GtkTreeIterCompareFunc  sort_func,
                                                                gpointer                user_data,
                                                                GtkDestroyNotify        destroy_func

static gboolean     custom_list_sortable_has_default_sort_func (GtkTreeSortable *sortable);

static void         custom_list_resort (CustomList *custom_list);

...
```

Next, let's extend our `CustomList` structure with a field for the currently active sort column ID and one for the sort order, and add an enum for the sort column IDs:

```
/* custom-list.h */

enum
{
  SORT_ID_NONE = 0,
  SORT_ID_NAME,
```

```
  SORT_ID_YEAR_BORN,
};

...

struct _CustomList
{
  GObject         parent;

  guint           num_rows;    /* number of rows that we have */
  CustomRecord  **rows;        /* a dynamically allocated array of pointers to the
                                *  CustomRecord structure for each row */

  gint            n_columns;
  GType           column_types[CUSTOM_LIST_N_COLUMNS];

  gint            sort_id;
  GtkSortType     sort_order;

  gint            stamp;       /* Random integer to check whether an iter belongs to our model */
};

...
```

Now, we make sure we initialise the new fields in `custom_list_new`, and add our new interface:

```
...

static void    custom_list_sortable_init (GtkTreeSortableIface *iface);

...

void
custom_list_init (CustomList *custom_list)
{
  ...
  custom_list->sort_id    = SORT_ID_NONE;
  custom_list->sort_order = GTK_SORT_ASCENDING;
  ...
}


GType
custom_list_get_type (void)
{
  ...
  /* Add GtkTreeSortable interface */
  if (1)
  {
    static const GInterfaceInfo tree_sortable_info =
    {
      (GInterfaceInitFunc) custom_list_sortable_init,
      NULL,
      NULL
    };

    g_type_add_interface_static (custom_list_type, GTK_TYPE_TREE_SORTABLE, &tree_sortable_info);
  }
  ...
}


static void
custom_list_sortable_init (GtkTreeSortableIface *iface)
{
  iface->get_sort_column_id    = custom_list_sortable_get_sort_column_id;
  iface->set_sort_column_id    = custom_list_sortable_set_sort_column_id;
  iface->set_sort_func         = custom_list_sortable_set_sort_func;          /* NOT SUPPORTED */
  iface->set_default_sort_func = custom_list_sortable_set_default_sort_func;  /* NOT SUPPORTED */
  iface->has_default_sort_func = custom_list_sortable_has_default_sort_func;  /* NOT SUPPORTED */
}
```

Now that we have finally taken care of the administrativa, we implement the tree sortable interface functions:

```
static gboolean
custom_list_sortable_get_sort_column_id (GtkTreeSortable *sortable,
                                         gint            *sort_col_id,
                                         GtkSortType     *order)
{
  CustomList *custom_list;

  g_return_val_if_fail ( sortable != NULL        , FALSE );
  g_return_val_if_fail ( CUSTOM_IS_LIST(sortable), FALSE );

  custom_list = CUSTOM_LIST(sortable);

  if (sort_col_id)
    *sort_col_id = custom_list->sort_id;

  if (order)
    *order =  custom_list->sort_order;

  return TRUE;
}


static void
custom_list_sortable_set_sort_column_id (GtkTreeSortable *sortable,
                                         gint             sort_col_id,
                                         GtkSortType      order)
{
  CustomList *custom_list;

  g_return_if_fail ( sortable != NULL         );
  g_return_if_fail ( CUSTOM_IS_LIST(sortable) );

  custom_list = CUSTOM_LIST(sortable);

  if (custom_list->sort_id == sort_col_id && custom_list->sort_order == order)
    return;

  custom_list->sort_id = sort_col_id;
  custom_list->sort_order  = order;

  custom_list_resort(custom_list);

  /* emit "sort-column-changed" signal to tell any tree views
   *  that the sort column has changed (so the little arrow
   *  in the column header of the sort column is drawn
   *  in the right column)                                    */

  gtk_tree_sortable_sort_column_changed(sortable);
}


static void
custom_list_sortable_set_sort_func (GtkTreeSortable        *sortable,
                                    gint                    sort_col_id,
                                    GtkTreeIterCompareFunc  sort_func,
                                    gpointer                user_data,
                                    GtkDestroyNotify        destroy_func)
{
  g_warning ("%s is not supported by the CustomList model.\n", __FUNCTION__);
}


static void
custom_list_sortable_set_default_sort_func (GtkTreeSortable        *sortable,
                                            GtkTreeIterCompareFunc  sort_func,
                                            gpointer                user_data,
```

```
                                                GtkDestroyNotify        destroy_func)
{
  g_warning ("%s is not supported by the CustomList model.\n", __FUNCTION__);
}


static gboolean
custom_list_sortable_has_default_sort_func (GtkTreeSortable *sortable)
{
  return FALSE;
}
```

Now, last but not least, the only thing missing is the function that does the actual sorting. We do not implement `set_sort_func`, `set_default_sort_func` and `set_has_default_sort_func` because we use our own internal sort function here.

The actual sorting is done using GLib's `g_qsort_with_data` function, which sorts an array using the QuickSort algorithm. Note how we notify the tree view and other objects of the new row order by emitting the "rows-reordered" signal on the tree model.

```
static gint
custom_list_compare_records (gint sort_id, CustomRecord *a, CustomRecord *b)
{
  switch(sort_id)
  {
    case SORT_ID_NONE:
      return 0;

    case SORT_ID_NAME:
    {
      if ((a->name) && (b->name))
        return g_utf8_collate(a->name, b->name);

      if (a->name == b->name)
        return 0; /* both are NULL */
      else
        return (a->name == NULL) ? -1 : 1;
    }

    case SORT_ID_YEAR_BORN:
    {
      if (a->year_born == b->year_born)
        return 0;

      return (a->year_born > b->year_born) ? 1 : -1;
    }
  }

  g_return_val_if_reached(0);
}


static gint
custom_list_qsort_compare_func (CustomRecord **a, CustomRecord **b, CustomList *custom_list)
{
  gint ret;

  g_assert ((a) && (b) && (custom_list));

  ret = custom_list_compare_records(custom_list->sort_id, *a, *b);

  /* Swap -1 and 1 if sort order is reverse */
  if (ret != 0  &&  custom_list->sort_order == GTK_SORT_DESCENDING)
    ret = (ret < 0) ? 1 : -1;

  return ret;
}
```

```
static void
custom_list_resort (CustomList *custom_list)
{
  GtkTreePath *path;
  gint        *neworder, i;

  g_return_if_fail ( custom_list != NULL );
  g_return_if_fail ( CUSTOM_IS_LIST(custom_list) );

  if (custom_list->sort_id == SORT_ID_NONE)
    return;

  if (custom_list->num_rows == 0)
    return;

  /* resort */
  g_qsort_with_data(custom_list->rows,
                    custom_list->num_rows,
                    sizeof(CustomRecord*),
                    (GCompareDataFunc) custom_list_qsort_compare_func,
                    custom_list);

  /* let other objects know about the new order */
  neworder = g_new0(gint, custom_list->num_rows);

  for (i = 0; i < custom_list->num_rows; ++i)
  {
    /* Note that the API reference might be wrong about
     * this, see bug number 124790 on bugs.gnome.org.
     * Both will work, but one will give you 'jumpy'
     * selections after row reordering. */
    /* neworder[(custom_list->rows[i])->pos] = i; */
    neworder[i] = (custom_list->rows[i])->pos;
    (custom_list->rows[i])->pos = i;
  }

  path = gtk_tree_path_new();

  gtk_tree_model_rows_reordered(GTK_TREE_MODEL(custom_list), path, NULL, neworder);

  gtk_tree_path_free(path);
  g_free(neworder);
}
```

Finally, we should make sure that the model is resorted after we have inserted a new row by adding a call to `custom_list_resort` to the end of `custom_list_append`:

```
...
void
custom_list_append_record (CustomList *custom_list, const gchar *name, guint year_born)
{
  ...

  custom_list_resort(custom_list);
}
```

And that is it. Adding two calls to `gtk_tree_view_column_set_sort_column_id` in main.c is left as yet another exercise for the reader.

If you are interested in seeing string sorting speed issues in action, you should modify main.c like this:

```
GtkWidget *
create_view_and_model (void)
{
  gint i;
  ...
  for (i=0; i < 1000; ++i)
  {
    fill_model(customlist);
```

```
    }
    ...
  }
```

Most likely, sorting 24000 rows by name will take up to several seconds now. Now, if you go back to `custom_list_compare_records` and replace the call to `g_utf8_collate` with:

```
  static gint
  custom_list_compare_records (gint sort_id, CustomRecord *a, CustomRecord *b)
  {
    ...

    if ((a->name) && (b->name))
      return strcmp(a->name_collate_key,b->name_collate_key);

    ...
  }
```

... then you should hopefully register a dramatic speed increase when sorting by name.

11.6. Working Example: Custom List Model Source Code

Here is the complete source code for the custom list model presented above. Compile with:

```
  gcc -o customlist custom-list.c main.c `pkg-config --cflags --libs gtk+-2.0`
```

- custom-list.h
- custom-list.c
- main.c

11.6.1. custom-list.h

```
#ifndef _custom_list_h_included_
#define _custom_list_h_included_

#include <gtk/gtk.h>

/* Some boilerplate GObject defines. 'klass' is used
 *   instead of 'class', because 'class' is a C++ keyword */

#define CUSTOM_TYPE_LIST            (custom_list_get_type ())
#define CUSTOM_LIST(obj)            (G_TYPE_CHECK_INSTANCE_CAST ((obj), CUSTOM_TYPE_LIST, CustomList))
#define CUSTOM_LIST_CLASS(klass)    (G_TYPE_CHECK_CLASS_CAST ((klass),  CUSTOM_TYPE_LIST, CustomListCl
#define CUSTOM_IS_LIST(obj)         (G_TYPE_CHECK_INSTANCE_TYPE ((obj), CUSTOM_TYPE_LIST))
#define CUSTOM_IS_LIST_CLASS(klass) (G_TYPE_CHECK_CLASS_TYPE ((klass),  CUSTOM_TYPE_LIST))
#define CUSTOM_LIST_GET_CLASS(obj)  (G_TYPE_INSTANCE_GET_CLASS ((obj),  CUSTOM_TYPE_LIST, CustomListCl

/* The data columns that we export via the tree model interface */

enum
{
  CUSTOM_LIST_COL_RECORD = 0,
  CUSTOM_LIST_COL_NAME,
  CUSTOM_LIST_COL_YEAR_BORN,
  CUSTOM_LIST_N_COLUMNS,
} ;


typedef struct _CustomRecord     CustomRecord;
typedef struct _CustomList       CustomList;
typedef struct _CustomListClass  CustomListClass;
```

```
/* CustomRecord: this structure represents a row */

struct _CustomRecord
{
  /* data - you can extend this */
  gchar    *name;
  gchar    *name_collate_key;
  guint     year_born;

  /* admin stuff used by the custom list model */
  guint     pos;   /* pos within the array */
};



/* CustomList: this structure contains everything we need for our
 *             model implementation. You can add extra fields to
 *             this structure, e.g. hashtables to quickly lookup
 *             rows or whatever else you might need, but it is
 *             crucial that 'parent' is the first member of the
 *             structure.                                      */

struct _CustomList
{
  GObject         parent;      /* this MUST be the first member */

  guint           num_rows;    /* number of rows that we have   */
  CustomRecord  **rows;        /* a dynamically allocated array of pointers to
                                *   the CustomRecord structure for each row    */

  /* These two fields are not absolutely necessary, but they    */
  /*   speed things up a bit in our get_value implementation    */
  gint            n_columns;
  GType           column_types[CUSTOM_LIST_N_COLUMNS];

  gint            stamp;       /* Random integer to check whether an iter belongs to our model */
};



/* CustomListClass: more boilerplate GObject stuff */

struct _CustomListClass
{
  GObjectClass parent_class;
};


GType             custom_list_get_type (void);

CustomList       *custom_list_new (void);

void              custom_list_append_record (CustomList   *custom_list,
                                             const gchar  *name,
                                             guint         year_born);

#endif /* _custom_list_h_included_ */
```

- custom-list.h
- custom-list.c
- main.c

11.6.2. custom-list.c

```
#include "custom-list.h"


/* boring declarations of local functions */

static void         custom_list_init            (CustomList      *pkg_tree);

static void         custom_list_class_init      (CustomListClass *klass);

static void         custom_list_tree_model_init (GtkTreeModelIface *iface);

static void         custom_list_finalize        (GObject           *object);

static GtkTreeModelFlags custom_list_get_flags  (GtkTreeModel      *tree_model);

static gint         custom_list_get_n_columns   (GtkTreeModel      *tree_model);

static GType        custom_list_get_column_type (GtkTreeModel      *tree_model,
                                                 gint               index);

static gboolean     custom_list_get_iter        (GtkTreeModel      *tree_model,
                                                 GtkTreeIter       *iter,
                                                 GtkTreePath       *path);

static GtkTreePath *custom_list_get_path        (GtkTreeModel      *tree_model,
                                                 GtkTreeIter       *iter);

static void         custom_list_get_value       (GtkTreeModel      *tree_model,
                                                 GtkTreeIter       *iter,
                                                 gint               column,
                                                 GValue            *value);

static gboolean     custom_list_iter_next       (GtkTreeModel      *tree_model,
                                                 GtkTreeIter       *iter);

static gboolean     custom_list_iter_children   (GtkTreeModel      *tree_model,
                                                 GtkTreeIter       *iter,
                                                 GtkTreeIter       *parent);

static gboolean     custom_list_iter_has_child  (GtkTreeModel      *tree_model,
                                                 GtkTreeIter       *iter);

static gint         custom_list_iter_n_children (GtkTreeModel      *tree_model,
                                                 GtkTreeIter       *iter);

static gboolean     custom_list_iter_nth_child  (GtkTreeModel      *tree_model,
                                                 GtkTreeIter       *iter,
                                                 GtkTreeIter       *parent,
                                                 gint               n);

static gboolean     custom_list_iter_parent     (GtkTreeModel      *tree_model,
                                                 GtkTreeIter       *iter,
                                                 GtkTreeIter       *child);


static GObjectClass *parent_class = NULL;  /* GObject stuff - nothing to worry about */


/*****************************************************************************
 *
 *  custom_list_get_type: here we register our new type and its interfaces
 *                        with the type system. If you want to implement
 *                        additional interfaces like GtkTreeSortable, you
 *                        will need to do it here.
 *
 *****************************************************************************/
```

```
GType
custom_list_get_type (void)
{
  static GType custom_list_type = 0;

  if (custom_list_type)
    return custom_list_type;

  /* Some boilerplate type registration stuff */
  if (1)
  {
    static const GTypeInfo custom_list_info =
    {
      sizeof (CustomListClass),
      NULL,                                          /* base_init */
      NULL,                                          /* base_finalize */
      (GClassInitFunc) custom_list_class_init,
      NULL,                                          /* class finalize */
      NULL,                                          /* class_data */
      sizeof (CustomList),
      0,                                            /* n_preallocs */
      (GInstanceInitFunc) custom_list_init
    };

    custom_list_type = g_type_register_static (G_TYPE_OBJECT, "CustomList",
                                               &custom_list_info, (GTypeFlags)0);
  }

  /* Here we register our GtkTreeModel interface with the type system */
  if (1)
  {
    static const GInterfaceInfo tree_model_info =
    {
      (GInterfaceInitFunc) custom_list_tree_model_init,
      NULL,
      NULL
    };

    g_type_add_interface_static (custom_list_type, GTK_TYPE_TREE_MODEL, &tree_model_info);
  }

  return custom_list_type;
}


/*****************************************************************************
 *
 *  custom_list_class_init: more boilerplate GObject/GType stuff.
 *                          Init callback for the type system,
 *                          called once when our new class is created.
 *
 *****************************************************************************/

static void
custom_list_class_init (CustomListClass *klass)
{
  GObjectClass *object_class;

  parent_class = (GObjectClass*) g_type_class_peek_parent (klass);
  object_class = (GObjectClass*) klass;

  object_class->finalize = custom_list_finalize;
}

/*****************************************************************************
 *
 *  custom_list_tree_model_init: init callback for the interface registration
 *                               in custom_list_get_type. Here we override
 *                               the GtkTreeModel interface functions that
 *                               we implement.
 *
```

```
 ******************************************************************************/

static void
custom_list_tree_model_init (GtkTreeModelIface *iface)
{
  iface->get_flags       = custom_list_get_flags;
  iface->get_n_columns   = custom_list_get_n_columns;
  iface->get_column_type = custom_list_get_column_type;
  iface->get_iter        = custom_list_get_iter;
  iface->get_path        = custom_list_get_path;
  iface->get_value       = custom_list_get_value;
  iface->iter_next       = custom_list_iter_next;
  iface->iter_children   = custom_list_iter_children;
  iface->iter_has_child  = custom_list_iter_has_child;
  iface->iter_n_children = custom_list_iter_n_children;
  iface->iter_nth_child  = custom_list_iter_nth_child;
  iface->iter_parent     = custom_list_iter_parent;
}


/******************************************************************************
 *
 *  custom_list_init: this is called everytime a new custom list object
 *                    instance is created (we do that in custom_list_new).
 *                    Initialise the list structure's fields here.
 *
 ******************************************************************************/

static void
custom_list_init (CustomList *custom_list)
{
  custom_list->n_columns       = CUSTOM_LIST_N_COLUMNS;

  custom_list->column_types[0] = G_TYPE_POINTER;  /* CUSTOM_LIST_COL_RECORD    */
  custom_list->column_types[1] = G_TYPE_STRING;   /* CUSTOM_LIST_COL_NAME      */
  custom_list->column_types[2] = G_TYPE_UINT;     /* CUSTOM_LIST_COL_YEAR_BORN */

  g_assert (CUSTOM_LIST_N_COLUMNS == 3);

  custom_list->num_rows = 0;
  custom_list->rows     = NULL;

  custom_list->stamp = g_random_int();  /* Random int to check whether an iter belongs to our model */

}


/******************************************************************************
 *
 *  custom_list_finalize: this is called just before a custom list is
 *                        destroyed. Free dynamically allocated memory here.
 *
 ******************************************************************************/

static void
custom_list_finalize (GObject *object)
{
/*  CustomList *custom_list = CUSTOM_LIST(object); */

  /* free all records and free all memory used by the list */
  #warning IMPLEMENT

  /* must chain up - finalize parent */
  (* parent_class->finalize) (object);
}


/******************************************************************************
 *
 *  custom_list_get_flags: tells the rest of the world whether our tree model
 *                         has any special characteristics. In our case,
```

```
 *                          we have a list model (instead of a tree), and each
 *                          tree iter is valid as long as the row in question
 *                          exists, as it only contains a pointer to our struct.
 *
 ******************************************************************************/

static GtkTreeModelFlags
custom_list_get_flags (GtkTreeModel *tree_model)
{
  g_return_val_if_fail (CUSTOM_IS_LIST(tree_model), (GtkTreeModelFlags)0);

  return (GTK_TREE_MODEL_LIST_ONLY | GTK_TREE_MODEL_ITERS_PERSIST);
}


/******************************************************************************
 *
 *  custom_list_get_n_columns: tells the rest of the world how many data
 *                             columns we export via the tree model interface
 *
 ******************************************************************************/

static gint
custom_list_get_n_columns (GtkTreeModel *tree_model)
{
  g_return_val_if_fail (CUSTOM_IS_LIST(tree_model), 0);

  return CUSTOM_LIST(tree_model)->n_columns;
}


/******************************************************************************
 *
 *  custom_list_get_column_type: tells the rest of the world which type of
 *                               data an exported model column contains
 *
 ******************************************************************************/

static GType
custom_list_get_column_type (GtkTreeModel *tree_model,
                             gint          index)
{
  g_return_val_if_fail (CUSTOM_IS_LIST(tree_model), G_TYPE_INVALID);
  g_return_val_if_fail (index < CUSTOM_LIST(tree_model)->n_columns && index >= 0, G_TYPE_INVALID);

  return CUSTOM_LIST(tree_model)->column_types[index];
}


/******************************************************************************
 *
 *  custom_list_get_iter: converts a tree path (physical position) into a
 *                        tree iter structure (the content of the iter
 *                        fields will only be used internally by our model).
 *                        We simply store a pointer to our CustomRecord
 *                        structure that represents that row in the tree iter.
 *
 ******************************************************************************/

static gboolean
custom_list_get_iter (GtkTreeModel *tree_model,
                      GtkTreeIter  *iter,
                      GtkTreePath  *path)
{
  CustomList    *custom_list;
  CustomRecord  *record;
  gint          *indices, n, depth;

  g_assert(CUSTOM_IS_LIST(tree_model));
  g_assert(path!=NULL);
```

```
  custom_list = CUSTOM_LIST(tree_model);

  indices = gtk_tree_path_get_indices(path);
  depth   = gtk_tree_path_get_depth(path);

  /* we do not allow children */
  g_assert(depth == 1); /* depth 1 = top level; a list only has top level nodes and no children */

  n = indices[0]; /* the n-th top level row */

  if ( n >= custom_list->num_rows || n < 0 )
    return FALSE;

  record = custom_list->rows[n];

  g_assert(record != NULL);
  g_assert(record->pos == n);

  /* We simply store a pointer to our custom record in the iter */
  iter->stamp      = custom_list->stamp;
  iter->user_data  = record;
  iter->user_data2 = NULL;   /* unused */
  iter->user_data3 = NULL;   /* unused */

  return TRUE;
}


/*****************************************************************************
 *
 *  custom_list_get_path: converts a tree iter into a tree path (ie. the
 *                        physical position of that row in the list).
 *
 *****************************************************************************/

static GtkTreePath *
custom_list_get_path (GtkTreeModel *tree_model,
                      GtkTreeIter  *iter)
{
  GtkTreePath  *path;
  CustomRecord *record;
  CustomList   *custom_list;

  g_return_val_if_fail (CUSTOM_IS_LIST(tree_model), NULL);
  g_return_val_if_fail (iter != NULL,               NULL);
  g_return_val_if_fail (iter->user_data != NULL,    NULL);

  custom_list = CUSTOM_LIST(tree_model);

  record = (CustomRecord*) iter->user_data;

  path = gtk_tree_path_new();
  gtk_tree_path_append_index(path, record->pos);

  return path;
}


/*****************************************************************************
 *
 *  custom_list_get_value: Returns a row's exported data columns
 *                         (_get_value is what gtk_tree_model_get uses)
 *
 *****************************************************************************/

static void
custom_list_get_value (GtkTreeModel *tree_model,
                       GtkTreeIter  *iter,
                       gint          column,
                       GValue       *value)
{
```

```
  CustomRecord  *record;
  CustomList    *custom_list;

  g_return_if_fail (CUSTOM_IS_LIST (tree_model));
  g_return_if_fail (iter != NULL);
  g_return_if_fail (column < CUSTOM_LIST(tree_model)->n_columns);

  g_value_init (value, CUSTOM_LIST(tree_model)->column_types[column]);

  custom_list = CUSTOM_LIST(tree_model);

  record = (CustomRecord*) iter->user_data;

  g_return_if_fail ( record != NULL );

  if(record->pos >= custom_list->num_rows)
   g_return_if_reached();

  switch(column)
  {
    case CUSTOM_LIST_COL_RECORD:
      g_value_set_pointer(value, record);
      break;

    case CUSTOM_LIST_COL_NAME:
      g_value_set_string(value, record->name);
      break;

    case CUSTOM_LIST_COL_YEAR_BORN:
      g_value_set_uint(value, record->year_born);
      break;
  }
}


/*****************************************************************************
 *
 *  custom_list_iter_next: Takes an iter structure and sets it to point
 *                         to the next row.
 *
 *****************************************************************************/

static gboolean
custom_list_iter_next (GtkTreeModel  *tree_model,
                       GtkTreeIter   *iter)
{
  CustomRecord  *record, *nextrecord;
  CustomList    *custom_list;

  g_return_val_if_fail (CUSTOM_IS_LIST (tree_model), FALSE);

  if (iter == NULL || iter->user_data == NULL)
    return FALSE;

  custom_list = CUSTOM_LIST(tree_model);

  record = (CustomRecord *) iter->user_data;

  /* Is this the last record in the list? */
  if ((record->pos + 1) >= custom_list->num_rows)
    return FALSE;

  nextrecord = custom_list->rows[(record->pos + 1)];

  g_assert ( nextrecord != NULL );
  g_assert ( nextrecord->pos == (record->pos + 1) );

  iter->stamp     = custom_list->stamp;
  iter->user_data = nextrecord;

  return TRUE;
```

```
}


/******************************************************************************
 *
 *  custom_list_iter_children: Returns TRUE or FALSE depending on whether
 *                             the row specified by 'parent' has any children.
 *                             If it has children, then 'iter' is set to
 *                             point to the first child. Special case: if
 *                             'parent' is NULL, then the first top-level
 *                             row should be returned if it exists.
 *
 ******************************************************************************/

static gboolean
custom_list_iter_children (GtkTreeModel *tree_model,
                           GtkTreeIter  *iter,
                           GtkTreeIter  *parent)
{
  CustomList  *custom_list;

  g_return_val_if_fail (parent == NULL || parent->user_data != NULL, FALSE);

  /* this is a list, nodes have no children */
  if (parent)
    return FALSE;

  /* parent == NULL is a special case; we need to return the first top-level row */

  g_return_val_if_fail (CUSTOM_IS_LIST (tree_model), FALSE);

  custom_list = CUSTOM_LIST(tree_model);

  /* No rows => no first row */
  if (custom_list->num_rows == 0)
    return FALSE;

  /* Set iter to first item in list */
  iter->stamp     = custom_list->stamp;
  iter->user_data = custom_list->rows[0];

  return TRUE;
}


/******************************************************************************
 *
 *  custom_list_iter_has_child: Returns TRUE or FALSE depending on whether
 *                              the row specified by 'iter' has any children.
 *                              We only have a list and thus no children.
 *
 ******************************************************************************/

static gboolean
custom_list_iter_has_child (GtkTreeModel *tree_model,
                            GtkTreeIter  *iter)
{
  return FALSE;
}


/******************************************************************************
 *
 *  custom_list_iter_n_children: Returns the number of children the row
 *                               specified by 'iter' has. This is usually 0,
 *                               as we only have a list and thus do not have
 *                               any children to any rows. A special case is
 *                               when 'iter' is NULL, in which case we need
 *                               to return the number of top-level nodes,
 *                               ie. the number of rows in our list.
 *
```

```
 ******************************************************************************/

static gint
custom_list_iter_n_children (GtkTreeModel *tree_model,
                             GtkTreeIter  *iter)
{
  CustomList  *custom_list;

  g_return_val_if_fail (CUSTOM_IS_LIST (tree_model), -1);
  g_return_val_if_fail (iter == NULL || iter->user_data != NULL, FALSE);

  custom_list = CUSTOM_LIST(tree_model);

  /* special case: if iter == NULL, return number of top-level rows */
  if (!iter)
    return custom_list->num_rows;

  return 0; /* otherwise, this is easy again for a list */
}


/******************************************************************************
 *
 *  custom_list_iter_nth_child: If the row specified by 'parent' has any
 *                              children, set 'iter' to the n-th child and
 *                              return TRUE if it exists, otherwise FALSE.
 *                              A special case is when 'parent' is NULL, in
 *                              which case we need to set 'iter' to the n-th
 *                              row if it exists.
 *
 ******************************************************************************/

static gboolean
custom_list_iter_nth_child (GtkTreeModel *tree_model,
                            GtkTreeIter  *iter,
                            GtkTreeIter  *parent,
                            gint          n)
{
  CustomRecord  *record;
  CustomList    *custom_list;

  g_return_val_if_fail (CUSTOM_IS_LIST (tree_model), FALSE);

  custom_list = CUSTOM_LIST(tree_model);

  /* a list has only top-level rows */
  if(parent)
    return FALSE;

  /* special case: if parent == NULL, set iter to n-th top-level row */

  if( n >= custom_list->num_rows )
    return FALSE;

  record = custom_list->rows[n];

  g_assert( record != NULL );
  g_assert( record->pos == n );

  iter->stamp = custom_list->stamp;
  iter->user_data = record;

  return TRUE;
}


/******************************************************************************
 *
 *  custom_list_iter_parent: Point 'iter' to the parent node of 'child'. As
 *                           we have a list and thus no children and no
 *                           parents of children, we can just return FALSE.
```

```
 *
 ******************************************************************************/

static gboolean
custom_list_iter_parent (GtkTreeModel *tree_model,
                         GtkTreeIter  *iter,
                         GtkTreeIter  *child)
{
  return FALSE;
}


/******************************************************************************
 *
 *  custom_list_new:  This is what you use in your own code to create a
 *                    new custom list tree model for you to use.
 *
 ******************************************************************************/

CustomList *
custom_list_new (void)
{
  CustomList *newcustomlist;

  newcustomlist = (CustomList*) g_object_new (CUSTOM_TYPE_LIST, NULL);

  g_assert( newcustomlist != NULL );

  return newcustomlist;
}


/******************************************************************************
 *
 *  custom_list_append_record:  Empty lists are boring. This function can
 *                              be used in your own code to add rows to the
 *                              list. Note how we emit the "row-inserted"
 *                              signal after we have appended the row
 *                              internally, so the tree view and other
 *                              interested objects know about the new row.
 *
 ******************************************************************************/

void
custom_list_append_record (CustomList   *custom_list,
                           const gchar  *name,
                           guint         year_born)
{
  GtkTreeIter   iter;
  GtkTreePath  *path;
  CustomRecord *newrecord;
  gulong        newsize;
  guint         pos;

  g_return_if_fail (CUSTOM_IS_LIST(custom_list));
  g_return_if_fail (name != NULL);

  pos = custom_list->num_rows;

  custom_list->num_rows++;

  newsize = custom_list->num_rows * sizeof(CustomRecord*);

  custom_list->rows = g_realloc(custom_list->rows, newsize);

  newrecord = g_new0(CustomRecord, 1);

  newrecord->name = g_strdup(name);
  newrecord->name_collate_key = g_utf8_collate_key(name,-1); /* for fast sorting, used later */
  newrecord->year_born = year_born;
```

```
  custom_list->rows[pos] = newrecord;
  newrecord->pos = pos;

  /* inform the tree view and other interested objects
   *  (e.g. tree row references) that we have inserted
   *  a new row, and where it was inserted */

  path = gtk_tree_path_new();
  gtk_tree_path_append_index(path, newrecord->pos);

  custom_list_get_iter(GTK_TREE_MODEL(custom_list), &iter, path);

  gtk_tree_model_row_inserted(GTK_TREE_MODEL(custom_list), path, &iter);

  gtk_tree_path_free(path);
}
```

- custom-list.h
- custom-list.c
- main.c

11.6.3. main.c

The following couple of lines provide a working test case that makes use of our custom list. It creates one of our custom lists, adds some records, and displays it in a tree view.

```
#include "custom-list.h"
#include <stdlib.h>

void
fill_model (CustomList *customlist)
{
  const gchar  *firstnames[] = { "Joe", "Jane", "William", "Hannibal", "Timothy", "Gargamel", NULL } ;
  const gchar  *surnames[]   = { "Grokowich", "Twitch", "Borheimer", "Bork", NULL } ;
  const gchar **fname, **sname;

  for (sname = surnames;  *sname != NULL;  sname++)
  {
    for (fname = firstnames;  *fname != NULL;  fname++)
    {
      gchar *name = g_strdup_printf ("%s %s", *fname, *sname);

      custom_list_append_record (customlist, name, 1900 + (guint) (103.0*rand()/(RAND_MAX+1900.0)));

      g_free(name);
    }
  }
}

GtkWidget *
create_view_and_model (void)
{
  GtkTreeViewColumn   *col;
  GtkCellRenderer     *renderer;
  CustomList          *customlist;
  GtkWidget           *view;

  customlist = custom_list_new();
  fill_model(customlist);

  view = gtk_tree_view_new_with_model(GTK_TREE_MODEL(customlist));

  g_object_unref(customlist); /* destroy store automatically with view */
```

```
  renderer = gtk_cell_renderer_text_new();
  col = gtk_tree_view_column_new();

  gtk_tree_view_column_pack_start (col, renderer, TRUE);
  gtk_tree_view_column_add_attribute (col, renderer, "text", CUSTOM_LIST_COL_NAME);
  gtk_tree_view_column_set_title (col, "Name");
  gtk_tree_view_append_column(GTK_TREE_VIEW(view),col);

  renderer = gtk_cell_renderer_text_new();
  col = gtk_tree_view_column_new();
  gtk_tree_view_column_pack_start (col, renderer, TRUE);
  gtk_tree_view_column_add_attribute (col, renderer, "text", CUSTOM_LIST_COL_YEAR_BORN);
  gtk_tree_view_column_set_title (col, "Year Born");
  gtk_tree_view_append_column(GTK_TREE_VIEW(view),col);

  return view;
}

int
main (int argc, char **argv)
{
  GtkWidget *window, *view, *scrollwin;

  gtk_init(&argc,&argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_default_size (GTK_WINDOW(window), 200, 400);
  g_signal_connect(window, "delete_event", gtk_main_quit, NULL);

  scrollwin = gtk_scrolled_window_new(NULL,NULL);

  view = create_view_and_model();

  gtk_container_add(GTK_CONTAINER(scrollwin), view);
  gtk_container_add(GTK_CONTAINER(window), scrollwin);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;
}
```

Chapter 12. Writing Custom Cell Renderers

The cell renderers that come with Gtk+ should be sufficient for most purposes, but there might be occasions where you want to display something in a tree view that you cannot display with the provided cell renderers, or where you want to derive from one of the provided cell renderers to extend its functionality.

You can do this by writing a new object that derives from `GtkCellRenderer` (or even one of the other cell renderers if you just want to extend an existing one).

Three things you need to do in the course of that:

- Register some new properties that your renderer needs with the type system and write your own `set_property` and `get_property` functions to set and get your new renderer's properties.
- Write your own `cell_renderer_get_size` function and override the parent object's function (usually the parent is of type `GtkCellRenderer`. Note that you should honour the standard properties for padding and cell alignment of the parent object here.
- Write your own `cell_renderer_render` function and override the parent object's function. This function does the actual rendering.

The GObject type system stuff of writing a new cell renderer is similar to what we have done above when writing a custom tree model, and is relatively straight forward in this case. Copy and paste and modify according to your own needs.

Good examples of cell renderer code to look at or even modify are `GtkCellRendererPixbuf` and `GtkCellRendererToggle` in the Gtk+ source code tree. Both cases are less than five hundred lines of code to look at and thus should be fairly easy to digest.

12.1. Working Example: a Progress Bar Cell Renderer

In the following we will write a custom cell renderer to render progress bars into a tree view (the code was "heavily inspired" by Sean Egan's progress bar cell renderer implementation in GAIM):

- custom-cell-renderer-progressbar.h
- custom-cell-renderer-progressbar.c
- main.c

12.1.1. custom-cell-renderer-progressbar.h

The header file consists of the usual GObject type cast and type check defines and our `CustomCellRendererProgress` structure. As the type of the parent indicates, we derive from `GtkCellRenderer`. The parent object must always be the first item in the structure (note also that it is not a pointer to an object, but the parent object structure itself embedded in our structure).

Our `CustomCellRendererProgress` structure is fairly uneventful and contains only a double precision float variable in which we store our new `"percentage"` property (which will determine how long the progressbar is going to be).

```
#ifndef _custom_cell_renderer_progressbar_included_
#define _custom_cell_renderer_progressbar_included_

#include <gtk/gtk.h>

/* Some boilerplate GObject type check and type cast macros.
 *  'klass' is used here instead of 'class', because 'class'
 *  is a c++ keyword */

#define CUSTOM_TYPE_CELL_RENDERER_PROGRESS             (custom_cell_renderer_progress_get_type())
#define CUSTOM_CELL_RENDERER_PROGRESS(obj)             (G_TYPE_CHECK_INSTANCE_CAST((obj),  CUSTOM_TY
#define CUSTOM_CELL_RENDERER_PROGRESS_CLASS(klass)     (G_TYPE_CHECK_CLASS_CAST ((klass),  CUSTOM_TY
#define CUSTOM_IS_CELL_PROGRESS_PROGRESS(obj)          (G_TYPE_CHECK_INSTANCE_TYPE ((obj), CUSTOM_TY
#define CUSTOM_IS_CELL_PROGRESS_PROGRESS_CLASS(klass)  (G_TYPE_CHECK_CLASS_TYPE ((klass),  CUSTOM_TY
#define CUSTOM_CELL_RENDERER_PROGRESS_GET_CLASS(obj)   (G_TYPE_INSTANCE_GET_CLASS ((obj),  CUSTOM_TY

typedef struct _CustomCellRendererProgress CustomCellRendererProgress;
typedef struct _CustomCellRendererProgressClass CustomCellRendererProgressClass;
```

```
  /* CustomCellRendererProgress: Our custom cell renderer
   *   structure. Extend according to need */

  struct _CustomCellRendererProgress
  {
    GtkCellRenderer   parent;

    gdouble           progress;
  };


  struct _CustomCellRendererProgressClass
  {
    GtkCellRendererClass  parent_class;
  };


  GType                custom_cell_renderer_progress_get_type (void);

  GtkCellRenderer     *custom_cell_renderer_progress_new (void);


  #endif /* _custom_cell_renderer_progressbar_included_ */
```

12.1.2. custom-cell-renderer-progressbar.c

The code contains everything as described above, so let's jump right into it:

```
#include "custom-cell-renderer-progressbar.h"

/* This is based mainly on GtkCellRendererProgress
 *  in GAIM, written and (c) 2002 by Sean Egan
 *  (Licensed under the GPL), which in turn is
 *  based on Gtk's GtkCellRenderer[Text|Toggle|Pixbuf]
 *  implementation by Jonathan Blandford */

/* Some boring function declarations: GObject type system stuff */

static void     custom_cell_renderer_progress_init       (CustomCellRendererProgress      *cellprogres

static void     custom_cell_renderer_progress_class_init (CustomCellRendererProgressClass *klass);

static void     custom_cell_renderer_progress_get_property  (GObject                    *object,
                                                             guint                       param_id,
                                                             GValue                     *value,
                                                             GParamSpec                 *pspec);

static void     custom_cell_renderer_progress_set_property  (GObject                    *object,
                                                             guint                       param_id,
                                                             const GValue               *value,
                                                             GParamSpec                 *pspec);

static void     custom_cell_renderer_progress_finalize (GObject *gobject);


/* These functions are the heart of our custom cell renderer: */

static void     custom_cell_renderer_progress_get_size   (GtkCellRenderer            *cell,
                                                          GtkWidget                  *widget,
                                                          GdkRectangle               *cell_area,
                                                          gint                       *x_offset,
                                                          gint                       *y_offset,
                                                          gint                       *width,
                                                          gint                       *height);

static void     custom_cell_renderer_progress_render     (GtkCellRenderer            *cell,
                                                          GdkWindow                  *window,
                                                          GtkWidget                  *widget,
```

```
                                                    GdkRectangle               *background_area,
                                                    GdkRectangle               *cell_area,
                                                    GdkRectangle               *expose_area,
                                                    guint                       flags);


enum
{
  PROP_PERCENTAGE = 1,
};

static   gpointer parent_class;


/*****************************************************************************
 *
 *  custom_cell_renderer_progress_get_type: here we register our type with
 *                                          the GObject type system if we
 *                                          haven't done so yet. Everything
 *                                          else is done in the callbacks.
 *
 *****************************************************************************/

GType
custom_cell_renderer_progress_get_type (void)
{
  static GType cell_progress_type = 0;

  if (cell_progress_type)
    return cell_progress_type;

  if (1)
  {
    static const GTypeInfo cell_progress_info =
    {
      sizeof (CustomCellRendererProgressClass),
      NULL,                                                 /* base_init */
      NULL,                                                 /* base_finalize */
      (GClassInitFunc) custom_cell_renderer_progress_class_init,
      NULL,                                                 /* class_finalize */
      NULL,                                                 /* class_data */
      sizeof (CustomCellRendererProgress),
      0,                                                    /* n_preallocs */
      (GInstanceInitFunc) custom_cell_renderer_progress_init,
    };

    /* Derive from GtkCellRenderer */
    cell_progress_type = g_type_register_static (GTK_TYPE_CELL_RENDERER,
                                                 "CustomCellRendererProgress",
                                                  &cell_progress_info,
                                                  0);
  }

  return cell_progress_type;
}


/*****************************************************************************
 *
 *  custom_cell_renderer_progress_init: set some default properties of the
 *                                      parent (GtkCellRenderer).
 *
 *****************************************************************************/

static void
custom_cell_renderer_progress_init (CustomCellRendererProgress *cellrendererprogress)
{
  GTK_CELL_RENDERER(cellrendererprogress)->mode = GTK_CELL_RENDERER_MODE_INERT;
  GTK_CELL_RENDERER(cellrendererprogress)->xpad = 2;
  GTK_CELL_RENDERER(cellrendererprogress)->ypad = 2;
}
```

```
/*****************************************************************************
 *
 *  custom_cell_renderer_progress_class_init:
 *
 *  set up our own get_property and set_property functions, and
 *  override the parent's functions that we need to implement.
 *  And make our new "percentage" property known to the type system.
 *  If you want cells that can be activated on their own (ie. not
 *  just the whole row selected) or cells that are editable, you
 *  will need to override 'activate' and 'start_editing' as well.
 *
 *****************************************************************************/

static void
custom_cell_renderer_progress_class_init (CustomCellRendererProgressClass *klass)
{
  GtkCellRendererClass *cell_class   = GTK_CELL_RENDERER_CLASS(klass);
  GObjectClass         *object_class = G_OBJECT_CLASS(klass);

  parent_class           = g_type_class_peek_parent (klass);
  object_class->finalize = custom_cell_renderer_progress_finalize;

  /* Hook up functions to set and get our
   *   custom cell renderer properties */
  object_class->get_property = custom_cell_renderer_progress_get_property;
  object_class->set_property = custom_cell_renderer_progress_set_property;

  /* Override the two crucial functions that are the heart
   *   of a cell renderer in the parent class */
  cell_class->get_size = custom_cell_renderer_progress_get_size;
  cell_class->render   = custom_cell_renderer_progress_render;

  /* Install our very own properties */
  g_object_class_install_property (object_class,
                                   PROP_PERCENTAGE,
                                   g_param_spec_double ("percentage",
                                                        "Percentage",
                                                         "The fractional progress to display",
                                                         0, 1, 0,
                                                         G_PARAM_READWRITE));
}


/*****************************************************************************
 *
 *  custom_cell_renderer_progress_finalize: free any resources here
 *
 *****************************************************************************/

static void
custom_cell_renderer_progress_finalize (GObject *object)
{
/*
  CustomCellRendererProgress *cellrendererprogress = CUSTOM_CELL_RENDERER_PROGRESS(object);
*/

  /* Free any dynamically allocated resources here */

  (* G_OBJECT_CLASS (parent_class)->finalize) (object);
}


/*****************************************************************************
 *
 *  custom_cell_renderer_progress_get_property: as it says
 *
 *****************************************************************************/

static void
```

```
custom_cell_renderer_progress_get_property (GObject    *object,
                                            guint       param_id,
                                            GValue     *value,
                                            GParamSpec *psec)
{
  CustomCellRendererProgress  *cellprogress = CUSTOM_CELL_RENDERER_PROGRESS(object);

  switch (param_id)
  {
    case PROP_PERCENTAGE:
      g_value_set_double(value, cellprogress->progress);
      break;

    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID (object, param_id, psec);
      break;
  }
}


/*****************************************************************************
 *
 *  custom_cell_renderer_progress_set_property: as it says
 *
 *****************************************************************************/

static void
custom_cell_renderer_progress_set_property (GObject      *object,
                                            guint         param_id,
                                            const GValue *value,
                                            GParamSpec   *pspec)
{
  CustomCellRendererProgress *cellprogress = CUSTOM_CELL_RENDERER_PROGRESS (object);

  switch (param_id)
  {
    case PROP_PERCENTAGE:
      cellprogress->progress = g_value_get_double(value);
      break;

    default:
      G_OBJECT_WARN_INVALID_PROPERTY_ID(object, param_id, pspec);
      break;
  }
}

/*****************************************************************************
 *
 *  custom_cell_renderer_progress_new: return a new cell renderer instance
 *
 *****************************************************************************/

GtkCellRenderer *
custom_cell_renderer_progress_new (void)
{
  return g_object_new(CUSTOM_TYPE_CELL_RENDERER_PROGRESS, NULL);
}


/*****************************************************************************
 *
 *  custom_cell_renderer_progress_get_size: crucial - calculate the size
 *                                          of our cell, taking into account
 *                                          padding and alignment properties
 *                                          of parent.
 *
 *****************************************************************************/

#define FIXED_WIDTH   100
#define FIXED_HEIGHT  10
```

```
static void
custom_cell_renderer_progress_get_size (GtkCellRenderer *cell,
                                        GtkWidget       *widget,
                                        GdkRectangle    *cell_area,
                                        gint            *x_offset,
                                        gint            *y_offset,
                                        gint            *width,
                                        gint            *height)
{
  gint calc_width;
  gint calc_height;

  calc_width  = (gint) cell->xpad * 2 + FIXED_WIDTH;
  calc_height = (gint) cell->ypad * 2 + FIXED_HEIGHT;

  if (width)
    *width = calc_width;

  if (height)
    *height = calc_height;

  if (cell_area)
  {
    if (x_offset)
    {
      *x_offset = cell->xalign * (cell_area->width - calc_width);
      *x_offset = MAX (*x_offset, 0);
    }

    if (y_offset)
    {
      *y_offset = cell->yalign * (cell_area->height - calc_height);
      *y_offset = MAX (*y_offset, 0);
    }
  }
}99


/*****************************************************************************
 *
 *  custom_cell_renderer_progress_render: crucial - do the rendering.
 *
 ****************************************************************************/

static void
custom_cell_renderer_progress_render (GtkCellRenderer *cell,
                                      GdkWindow       *window,
                                      GtkWidget       *widget,
                                      GdkRectangle    *background_area,
                                      GdkRectangle    *cell_area,
                                      GdkRectangle    *expose_area,
                                      guint            flags)
{
  CustomCellRendererProgress *cellprogress = CUSTOM_CELL_RENDERER_PROGRESS (cell);
  GtkStateType                state;
  gint                        width, height;
  gint                        x_offset, y_offset;

  custom_cell_renderer_progress_get_size (cell, widget, cell_area,
                                          &x_offset, &y_offset,
                                          &width, &height);

  if (GTK_WIDGET_HAS_FOCUS (widget))
    state = GTK_STATE_ACTIVE;
  else
    state = GTK_STATE_NORMAL;

  width  -= cell->xpad*2;
  height -= cell->ypad*2;

  gtk_paint_box (widget->style,
```

```
                  window,
                  GTK_STATE_NORMAL, GTK_SHADOW_IN,
                  NULL, widget, "trough",
                  cell_area->x + x_offset + cell->xpad,
                  cell_area->y + y_offset + cell->ypad,
                  width - 1, height - 1);

  gtk_paint_box (widget->style,
                  window,
                  state, GTK_SHADOW_OUT,
                  NULL, widget, "bar",
                  cell_area->x + x_offset + cell->xpad,
                  cell_area->y + y_offset + cell->ypad,
                  width * cellprogress->progress,
                  height - 1);
}
```

12.1.3. main.c

And here is a little test that makes use of our new `CustomCellRendererProgress`:

```
#include "custom-cell-renderer-progressbar.h"

static GtkListStore        *liststore;

static gboolean             increasing = TRUE;   /* direction of progress bar change */

enum
{
  COL_PERCENTAGE = 0,
  COL_TEXT,
  NUM_COLS
};

#define STEP  0.01

gboolean
increase_progress_timeout (GtkCellRenderer *renderer)
{
  GtkTreeIter  iter;
  gfloat       perc = 0.0;
  gchar        buf[20];

  gtk_tree_model_get_iter_first(GTK_TREE_MODEL(liststore), &iter); /* first and only row */

  gtk_tree_model_get (GTK_TREE_MODEL(liststore), &iter, COL_PERCENTAGE, &perc, -1);

  if ( perc > (1.0-STEP)  ||  (perc < STEP && perc > 0.0) )
  {
    increasing = (!increasing);
  }

  if (increasing)
    perc = perc + STEP;
  else
    perc = perc - STEP;

  g_snprintf(buf, sizeof(buf), "%u %%", (guint)(perc*100));

  gtk_list_store_set (liststore, &iter, COL_PERCENTAGE, perc, COL_TEXT, buf, -1);

  return TRUE; /* Call again */
}


GtkWidget *
create_view_and_model (void)
{
```

```
  GtkTreeViewColumn   *col;
  GtkCellRenderer     *renderer;
  GtkTreeIter          iter;
  GtkWidget           *view;

  liststore = gtk_list_store_new(NUM_COLS, G_TYPE_FLOAT, G_TYPE_STRING);
  gtk_list_store_append(liststore, &iter);
  gtk_list_store_set (liststore, &iter, COL_PERCENTAGE, 0.5, -1); /* start at 50% */

  view = gtk_tree_view_new_with_model(GTK_TREE_MODEL(liststore));

  g_object_unref(liststore); /* destroy store automatically with view */

  renderer = gtk_cell_renderer_text_new();
  col = gtk_tree_view_column_new();
  gtk_tree_view_column_pack_start (col, renderer, TRUE);
  gtk_tree_view_column_add_attribute (col, renderer, "text", COL_TEXT);
  gtk_tree_view_column_set_title (col, "Progress");
  gtk_tree_view_append_column(GTK_TREE_VIEW(view),col);

  renderer = custom_cell_renderer_progress_new();
  col = gtk_tree_view_column_new();
  gtk_tree_view_column_pack_start (col, renderer, TRUE);
  gtk_tree_view_column_add_attribute (col, renderer, "percentage", COL_PERCENTAGE);
  gtk_tree_view_column_set_title (col, "Progress");
  gtk_tree_view_append_column(GTK_TREE_VIEW(view),col);

  g_timeout_add(50, (GSourceFunc) increase_progress_timeout, NULL);

  return view;
}


int
main (int argc, char **argv)
{
  GtkWidget *window, *view;

  gtk_init(&argc,&argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_default_size (GTK_WINDOW(window), 150, 100);
  g_signal_connect(window, "delete_event", gtk_main_quit, NULL);

  view = create_view_and_model();

  gtk_container_add(GTK_CONTAINER(window), view);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;
}
```

12.2. Cell Renderers Others Have Written

Just in case you are one of those people who do not like to re-invent the wheel, here is a list of custom cell renderers other people have written:

- Progress bar cell renderer (gaim)
- Date cell renderer (mrproject) (is this one easy to re-use?)
- List/combo cell renderer (mrproject) (is this one easy to re-use?)
- Pop-up cell renderer (mrproject) (what does this do?)

- Your custom cell renderer here?!

Chapter 13. Other Resources

A short tutorial like this cannot possibly cover everything. Luckily, there is a lot more information out there. Here is a list of links that you might find useful (if you have any links that should appear here as well, please send them to `tim at centricular dot net`).

- Gtk+ API Reference Manual
- Gdk API Reference Manual
- Pango API Reference Manual
- GLib API Reference Manual
- gtk-app-devel mailing list archives - search them!
- gtk-demo - part of the Gtk+ source code (look in gtk+-2.x.y/demos/gtk-demo), especially list_store.c, tree_store.c, and stock_browser.c
- TreeView tutorial using Gtk's C++ interface (gtkmm)
- TreeView tutorial using Gtk's python interface
- Some slides from Owen Taylor's GUADEC 2003 tutorial (postscript, pdf, see pages 13-15)
- Existing applications - yes, they exist, and *you* can look at their source code. SourceForge's WebCVS browse feature is quite useful, and the same goes for GNOME as well.
- If your intention is to display external data (from a database, or in XML form) as a list or tree or table, you might also be interested GnomeDB, especially libgda and libgnomedb (e.g. the GnomeDBGrid widget). See also this PDF presentation (page 24ff).
- your link here!

Chapter 14. Copyright, License, Credits, and Revision History

14.1. Copyright and License

14.2. Credits

Thanks to Axel C. for proof-reading the first drafts, for many suggestions, and for introducing me to the tree view widget in the first place (back then when I was still convinced that porting to Gtk+-2.x was unnecessary, Gtk+-1.2 applications looked nice, and Aristotle had already said everything about politics that needs to be said).

Harring Figueiredo shed some light on how GtkListStore and GtkTreeStore deal with pixbufs.

Ken Rastatter suggested some additional topics (with complete references even).

Both Andrej Prsa and Alan B. Canon sent me a couple of suggestions, and 'taf2', Massimo Mangoni and others spotted some typos.

Many thanks to all of them, and of course also to kris and everyone else in #gtk+.

14.3. Revision History

5th June 2005

- Remove unnecessary col = gtk_tree_view_column_new() im hello world code (leftover from migration to convenience functions).

3rd February 2005

- Point out that GObjects such as GdkPixbufs retrieved with gtk_tree_model_get() need to be g_object_unref()'ed after use, as gtk_tree_model_get() adds a reference.
- Added explicit (gint) event->x double to int conversion to code snippet using gtk_tree_view_get_path_at_pos() to avoid compiler warnings.

9th September 2004

- Fixed another mistake in tree path explanation: text did not correspond picture (s/movie clips/movie trailers/); (thanks to Benjamin Brandt for spotting it).

6th August 2004

- Fixed mistake in tree path explanation (s/4th/5th/) (thanks to both Andrew Kirillov and Benjamin Brandt for spotting it).

30th April 2004

- Added Hello World

31st March 2004

- Fixed fatal typo in custom list code: g_assert() in custom_list_init() should be ==, not != (spotted by mmc).
- Added link to Owen Taylor's mail on the GtkTreeView Drag'n'Drop API.

24th January 2004

- Fixed typo in code example (remove n-th row example) (Thanks to roel for spotting it).
- Changed 'Context menus' section title

19th January 2004

- Expanded section on GtkTreeRowReferences, and on removing multiple rows.

8th January 2004

- Added tiny section on Glade and treeviews
- Added more detail to the section describing GtkTreePath, GtkTreeIter et.al.
- Reformatted document structure: instead of one single chapter with lots of sections, have multiple chapters (this tutorial is way to big to become part of the Gtk+ tutorial anyway); enumerate chapters and sections.
- Expanded the section on tree view columns and cell renderers, with help of two diagrams by Owen Taylor (from the GUADEC 2003 Gtk+ tutorial slides).

10th December 2003

- Added more information about how to remove a single row, or more specifically, the n-th row of a list store
- Added a short example about how to pack icons into the tree view.

28th October 2003

- Editable cells will work fine even if selection is set to GTK_SELECTION_NONE. Removed sentences that say otherwise.

23rd October 2003

- fix 'jumpy' selections in custom model GtkTreeSortable interface implementation. gtk_tree_model_rows_reordered() does not seem to work like the API reference implies (see bug #124790)
- added section about how to get the cell renderer a button click happened on
- added section about editable cells with spin buttons (and a CellRendererSpin implementation to the examples)

10th October 2003

- make custom model GtkTreeSortable implementation emit "sort-column-changed" signal when sortid is changed
- fixed code typo in selection function section; added a paragraph about rule hint to 'make whole row coloured or bold' section

7th October 2003

- Reformatted source code to make it fit on pages when generating ps/pdf output
- Added link to PDF and docbook XML versions.