The GTK+ TextView Widget

1 Introduction

GTK+ has an extremely powerful framework for multiline text editing. The `GtkTextView` widget is the primary component of this framework. However, unlike the widgets you have seen in previous lessons, the text view widget is part of a larger framework that is something like the model/view/controller software architecture. In this architecture, the visual display of the data, called the *view*, is separate from the manipulation and representation of the data, which is managed by a *model*. The `GtkTreeView` does not follow this architecture exactly, but it is a good idea to keep in mind when designing your code, that this is how things work.

The other principal component in this framework is the `GtkTextBuffer`, which is where the text data is stored. Besides this, there are ancillary objects such as text tags, text iterators, and text marks, all of which exist to support text selection, modification, and formatting. And like all other GTK+ text-related objects, the `GtkTextView` is designed to handle UTF-8 encoded characters and strings. This is especailly important in the text view widget because the fact that one character can be encoded as multiple bytes implies that there is a difference between character counts and byte counts. Character counts are referred to as offsets, while byte counts are called indexes. Certain functions work with byte indices and others work with offsets, and you must pay careful attention, otherwise, to quote the API documentation, "bad things will happen."

2 Scrolling, ScrolledWindows, and Viewports

Imagine that you had a text viewing application, that could not scroll, in other words, it had no scrollbars. Suppose we call this application `SimpleText`. If you try to open a text document that has 2000 lines in `SimpleText`, it will resize itself so that it is large enough to display the entire document. In this case, the first fifty lines would be visible on the screen, and the remaining lines would be off the screen, as would most of the `SimpleText` window. In order to read the rest of the text, you would have to grab the `SimpleText` window and "slide it" upwards, provided that there were something to grab on the window's decoration. If not, you could not read the document at all, so `SimpleText` would be limited to opening only those text documents that could fit in its visible window. This would be a rather useless application.

Obviously, the ability to scroll must be a part of any practical, text viewing application, and so we begin by discussing scrolling. In GTK+, scrolling is achieved through the use of the `GtkScrolledWindow`. The `GtkScrolledWindow` class is a subclass of `GtkBin`; it is a container that can have a single child. When a widget is added to a `GtkScrolledWindow`, the `GtkScrolledWindow` gives it scrollbars.

Giving a widget scrollbars does not give that widget the ability to scroll itself, no more than giving a pig wings gives it the ability to fly. The child widget has to have the ability to scroll itself if the scrollbars in the `GtkScrolledWindow` are going to make the child scroll. When a widget has the ability to scroll itself, it is said to have *native scrolling support.*

Recall that scrollbars are a type of `GtkRange` widget, and that range widgets are basically widgets that visualize `GtkAdjustment` objects. When a scrollbar's thumb is moved, or when it is adjusted by clicking in its trough or on its stepper arrows, it causes the internal adjustment widget to update its value. Widgets that have native scrolling support have, using the language of the API documentation, "slots" for `GtkAdjustment` objects. What this means is that the widget has two `GtkAdjustment` members, one horizontal and one vertical, and internal private functions that calculate what portions of the widget should be visible based on the values of the adjustment objects. These private functions are invoked by signal handlers as scrollbar values change, but also when other events, such as the window's being resized, take place.

The `GtkTextView` widget, along with the `GtkTreeView` and `GtkLayout`, has native scrolling support. This implies that it is enough to put the text view into a scrolled window for its scrolling to "work", using `gtk_container_add()`. For example, if `scrolled_window` is a `GtkScrolledWindow` and `text_view` is a `GtkTextView` then

```
gtk_container_add (GTK_CONTAINER (scrolled_window), text_view);
```

is all you need to do for text_view to be scrollable.

Widgets that do not have native scrolling support, such as `GtkImage` or `GtkEventBox`, must first be given that support by adding them to a `GtkViewport`. The `GtkViewport` widget acts as an adaptor class, implementing scrollability for child widgets that lack their own scrolling capabilities. The `GtkViewport` is then added to the scrolled window. There are two ways to do this:

1. create the viewport, add the widget to it, and add the viewport to the scrolled window, or
2. use the convenience function `gtk_scrolled_window_add_with_viewport()`, which cuts out a step.

As there is no reason to have access to the viewport by itself, it is usually sufficient to use this convenience function. For example, if `image` is a `GtkImage`, and we wish it to receive events on it and also have it scrollable, then we would write the following code:

```
event_box = gtk_event_box_new ();
gtk_container_add (GTK_CONTAINER (event_box), image );
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_add_with_viewport (
                        GTK_SCROLLED_WINDOW (scrolled_window), event_box );
```

The function `gtk_scrolled_window_new()` creates a scrolled window. The last two arguments specify pointers to the horizontal and vertical adjustments that the window should use. Setting these to `NULL` tells GTK+ to create new adjustments for it, which is what you usually want to do.

Although it is enough just to create the scrolled window and add the child widget to it, you should make a habit of setting the scrollbar policies of the window. You have probably noticed that some applications are designed so that, if the data being displayed is small enough that there is no need for scrollbars, the scrollbars disappear. Others always have scrollbars no matter how much data is displayable. The function

```
void      gtk_scrolled_window_set_policy    ( GtkScrolledWindow *scrolled_window,
                                              GtkPolicyType hscrollbar_policy,
                                              GtkPolicyType vscrollbar_policy);
```

can be used to set the policy. The three possible values for the horizontal and vertical policies are

`GTK_POLICY_ALWAYS`	The scrollbar is always visible.
`GTK_POLICY_AUTOMATIC`	The scrollbar will appear and disappear as necessary. For example, when all of a `GtkCList` can not be seen.
`GTK_POLICY_NEVER`	The scrollbar will never appear.

I cannot think of a reason why you would want to set the policy to never display the scrollbars. The choice is usually whether you want them to disappear when they are not needed, or not.

Scrolled windows have just a few methods besides these. Most likely you will not need to use them. You can control the shadow that the window places around its child widget using

```
void       gtk_scrolled_window_set_shadow_type ( GtkScrolledWindow *scrolled_window,
                                                 GtkShadowType type);
```

The `GtkShadowType` argument should be one of

`GTK_SHADOW_NONE`	No outline.
`GTK_SHADOW_IN`	The outline is bevelled inwards.
`GTK_SHADOW_OUT`	The outline is bevelled outwards like a button.
`GTK_SHADOW_ETCHED_IN`	The outline has a sunken 3d appearance.
`GTK_SHADOW_ETCHED_OUT`	The outline has a raised 3d appearance

You can control where the child widget should be positioned relative to the scrollbars. It has become standard practice for the horizontal scrollbar to be at the bottom, ad the vertical one to the right. If you choose to change these, you should have a very good reason. Nonetheless the method is

```
void       gtk_scrolled_window_set_placement  ( GtkScrolledWindow *scrolled_window,
                                                GtkCornerType window_placement);
```

You can read more about this in the API documentation.

There may be times when you want to share either of the embedded adjustments among two or more scrolled windows or viewports. Suppose for example that you wanted two scrolled windows to share a single pair of adjustment objects. To do this, you would need to

1. Create the first scrolled window with `NULL` adjustment pointers.
2. Get the adjustments from this first window.
3. Create the second window using the adjustments just obtained.

This is accomplished as follows (declarations omitted):

```
scrolled_window1 = gtk_scrolled_window_new (NULL, NULL);
h_adjustment = gtk_scrolled_window_get_hadjustment (
                                  GTK_SCROLLED_WINDOW (scrolled_window1));
v_adjustment = gtk_scrolled_window_get_vadjustment (
                                  GTK_SCROLLED_WINDOW (scrolled_window1));
scrolled_window2 = gtk_scrolled_window_new (h_adjustment, v_adjustment);
```

Alternatively, both scrolled windows could be created with NULL adjustments, and then the second's could be replaced using

```
void       gtk_scrolled_window_set_hadjustment ( GtkScrolledWindow *scrolled_window,
                                                 GtkAdjustment *hadjustment);
```

or the equivalent function for vertical adjustments.

Example

A complete example of a simple program that can open any image file and display it in a scrolled window is the program `scrolledwindow_demo1.c` in the `scrolledwindows` subdirectory. The program in its entirety is also displayed in Listing 1 in the appendix of these notes.

3 Overview of the GtkTextView

A `GtkTextView` widget contains a pointer to a `GtkTextBuffer`. There are two ways to create the textview – either without specifying a particular buffer (using `gtk_text_view_new()`), in which case it starts with an empty buffer, or by providing it with a pointer to an existing buffer, using `gtk_text_view_new_with_buffer()`:

```
GtkWidget *    gtk_text_view_new               ( void);
GtkWidget *    gtk_text_view_new_with_buffer   ( GtkTextBuffer *buffer);
```

The buffer can be retrieved at any time using

```
GtkTextBuffer * gtk_text_view_get_buffer       ( GtkTextView *text_view);
```

or replaced using

```
void            gtk_text_view_set_buffer       ( GtkTextView *text_view,
                                                 GtkTextBuffer *buffer);
```

Text in a buffer can be marked with *tags*, objects of type `GtkTextTag`. A tag should be thought of as a set of attributes that can be applied to a range of text. Tags can be assigned names for easy access, though this is not a requirement. For example, a tag might be named "bold-italic". Naturally, if you name a tag "bold-italic" it would be wise to make sure that the attributes that it embodies make the text bold and italic. The `GtkTextTag` object, however, is much more general than an object encapsulating physical text attributes. It can be used to make text visible or invisible, or editable or un-editable, for example. A single `GtkTextTag` can be applied to any number of text ranges in any number of buffers.

The set of attributes that can be controlled by tags is quite extensive. You can pretty much control all aspects of the appearance of the text, from its justification, the spacing between lines, the character rise, weight, scale, font-family, and so on, and the foreground and background colors and textures.

Tags are stored in a `GtkTextTagTable`. A tag table defines a set of tags that can be used together. Each buffer has one tag table associated with it; only tags from that tag table can be used with the buffer. However, multiple buffers can share a tag table.

The textview widget itself stores attributes that control the appearance of all text in the buffer, independent of the attributes defined by the tags. These attributes include indentation, justification, margin and tab settings, editability, and the spacing between paragraphs. However, these attributes will be overridden by the tags. In other words, if the textview sets the spacing above a paragraph to be 10 pixels, then all paragraphs will have 10 pixels above them, unless a tag with a different spacing value is applied to that paragraph, in which case that paragraph uses the spacing defined in the tag.

Locations within a text buffer are represented in two different ways. Text iterators, represented by the `GtkTextIter` class, are objects that represent a position *between two characters* in the text buffer. Unlike many other objects in GTK+, text iterators reside on the user stack. It never has any data stored on the heap and is therefore copiable (with shallow assignments.) However, any time that the text in the buffer is modified in a way that affects the number of characters in the buffer, all outstanding iterators become invalid. This is true even if an iterator not near the changed text and even if an insertion and susquent deletion leaves the number of characters the same. Because of this, iterators cannot be used to preserve positions as the text in the buffer is modified.

Text marks are objects that can save positions in the text. Text marks belong to the `GtkTextMark` class. Like an iterator, it is a position between characters in the text, like a cursor or an insertion point. Unlike an iterator, a text mark can float in the buffer, saving a position. In other words, if the text in the buffer changes, the mark adjusts its position accordingly.

For example, if the text on both the left and right sides of a mark is deleted, the mark stays in the position that the text occupied. To be concrete, suppose that we represent the mark by a vertical bar and that a text fragment in the buffer looks like this:

```
abcde|fghijkl
```

with the mark between the "`e`" and the "`f`" (no space in between). If the string "`defgh`" is deleted, the text becomes

```
abc|ijkl
```

If text is inserted at the mark, the mark ends up either to the left or to the right of the new text, depending on its gravity. The standard text cursor in left-to-right languages is a mark with right gravity, because it stays to the right of inserted text.

Like tags, marks can be either named or anonymous. There are special two marks built into `GtkTextBuffer`; these are named `"insert"` and `"selection_bound"` and refer to the insertion point and the boundary of the selection which is not the insertion point, respectively. If no text is selected, these two marks will be in the same position. You can manipulate what is selected and where the cursor appears by moving these marks around.

Last but not least of its capabilities, the textview widget's buffer can contain pixbufs and widgets. The `GtkTextBuffer` object has methods for loading both. Buffers that contain such non-character data have to handled carefully, because of how these are represented within the buffer. Details will follow below.

4 Textview Basics

We begin with the simplest of examples, namely loading a file into a text buffer and displaying it with almost all of the default values the textview widget has. We will add margins, because the default is for the margins to be set to zero pixels, which is just plain ugly. The application will also demonstrate how to access the entire contents of the text buffer so that they can be written to a file, only in this case, they will be written to the standard output stream.

We will also make the application a little easier to use by providing a button that opens a file selection dialog, filtering for text files only, and a button that writes the text to the output stream. The program will use a structure of type `AppState` that stores the parts of the interface that need to be accessed by the callbacks and the main program:

```
typedef struct _AppState
{
    GtkWidget *window;
    GtkWidget *text_view;
    GtkWidget *open_button;
    GtkWidget *print_button;
} AppState;
```

An instance of type `AppState` named `app_state` is declared in the main program and passed to the callbacks through the user data parameter. The code to create the textview and scrolled window and set the policies is:

```
app_state.text_view = gtk_text_view_new ();
gtk_text_view_set_left_margin (GTK_TEXT_VIEW (app_state.text_view), 10);
gtk_text_view_set_right_margin(GTK_TEXT_VIEW (app_state.text_view), 10);

scrolled_window = gtk_scrolled_window_new(NULL, NULL);
gtk_container_add (GTK_CONTAINER (scrolled_window), app_state.text_view);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                GTK_POLICY_AUTOMATIC,
                                GTK_POLICY_ALWAYS);
```

When the open button is clicked and the user selects a file, the file is loaded all at once into a UTF-8 string, which is then loaded into the text buffer. In order to do this, three steps are necessary:

1. Get the buffer contained in the textview widget using `gtk_text_view_get_buffer()`:

   ```
   buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (app_state->text_view));
   ```

2. Load the contents of the text file using GLib's `g_file_get_contents()` into a string:

   ```
   g_file_get_contents( filename, &contents, &nBytesInBuf, &error);
   ```

3. Set the text of the buffer to be that contained in the string, using `gtk_text_buffer_set_text()`:

   ```
   gtk_text_buffer_set_text(buffer, contents, -1);
   ```

The `gtk_text_buffer_set_text()` function will replace the text currently in the buffer, if any, with the new text, handling any deallocation of memory as necessary. You do not need to free the pre-existing text in any way. You do need to free the string after you call `gtk_text_buffer_set_text()`, as you no longer need a reference to it, as well as the filename that was returned by the file chooser dialog box. The last parameter of `gtk_text_buffer_set_text()` is the length of the text in bytes.

The relevant parts of the `on_open_button()` calback are as follows. Declarations are omitted.

```
void on_open_button      ( GtkWidget *button,
                           AppState *app_state)
{
    /* dialog opened, results checked, filename obtained successfully,
       then this...
    */

    /* Obtaining the buffer associated with the widget. */
    buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (app_state->text_view));

    /* Use the filename to read its contents into a gchar* string contents */
    if (! g_file_get_contents( filename, &contents, &nBytesInBuf, &error)) {
        g_printf(error->message);
        g_clear_error(&error);
        g_free(filename);
        exit(1);
    }
    /* Success, so copy contents into buffer and free the contents and
    filename strings */
    gtk_text_buffer_set_text(buffer, contents, -1);
    g_free(filename);
    g_free(contents);
    /* more stuff then return */
}
```

To retrieve the entire contents of the text buffer we can use either `gtk_text_buffer_get_text()` or `gtk_text_buffer_get_slice()`. The difference is that the `...get_text` variant ignores the non-character data,i.e., pixbufs and widgets, whereas the `...get_slice` variant puts placeholders into the returned string to preserve character counts. Their prototypes are identical. The latter is

```
gchar *      gtk_text_buffer_get_slice        ( GtkTextBuffer *buffer,
                                                 const GtkTextIter *start,
                                                 const GtkTextIter *end,
                                                 gboolean include_hidden_chars);
```

It takes a pointer to the buffer, and pointers to the two iterators that store the starting and ending bounds of the region to be copied. The last parameter should be set to `TRUE` if the returned string should include characters marked as invisible in the buffer, or `FALSE` if not.

`GtkTextBuffer` has a function for getting the start and end bounds in a buffer:

```
void          gtk_text_buffer_get_bounds        ( GtkTextBuffer *buffer,
                                                   GtkTextIter *start,
                                                   GtkTextIter *end);
```

When this returns, start and end contain iterators to the first and last positions in the buffer, suitable for passing to `gtk_text_buffer_get_slice()` to get the entire contents. Therefore, the function to print the contents of the buffer to a stream of any kind amounts to

```
{
    /* declarations omitted; app_state is a pointer to an AppState object*/
    buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW(app_state->text_view));
    if ( NULL != buffer ) {
        gtk_text_buffer_get_bounds (buffer, &start, &end);
        text = gtk_text_buffer_get_slice (buffer, &start, &end, FALSE);

        /* write text to whatever stream it needs to be written to, then */
        g_free (text);
    }
}
```

The entire program is contained in Listing 2 in the appendix.

5 Modifying Global textview Attributes

The next step is to add the ability to change the global attributes of the text using the functions of the `GtkTextView` widget itself. In general, the widget has get and set methods for each of the following properties:

Property Name	**Property Type**	**Description**
`accepts-tab`	`gboolean`	whether input tab results in tab character entered
`cursor-visible`	`gboolean`	whether the cursor is visble
`editable`	`gboolean`	whether the text can be modified by the user
`indent`	`gint`	the amount to indent paragraphs in pixels
`justification`	`GtkJustification`	left, right, or center justification, e.g., `GTK_JUSTIFY_LEFT`, ...
`left-margin`	`gint`	width of the left margin in pixels
`overwrite`	`gboolean`	whether entered text overwrites existing text
`pixels-above-lines`	`gint`	pixels of blank space above paragraphs
`pixels-below-lines`	`gint`	pixels of blank space above paragraphs
`pixels-inside-wrap`	`gint`	pixels of blank space between wrapped lines in a paragraph
`right-margin`	`gint`	width of the right margin in pixels
`tabs`	`PangoTabArray*`	custom tab positions
`wrap-mode`	`GtkWrapMode`	whether to wrap lines never, at word boundaries, or at character boundaries, e.g., `GTK_WRAP_NONE, ...`

The methods for setting properties are of the form

```
void          gtk_text_view_set_xxx            ( GtkTextView *text_view,
                                                 yyy          setting);
```

where xxx is usually the property name with '_' replacing '-', and where the type of the setting, yyy, corresponds to the type column in the list above. For example, the function that changes the pixels-above-lines property is

```
void      gtk_text_view_set_pixels_above_lines   ( GtkTextView *text_view,
                                                   int  pixels_above_lines);
```

In addition to changing these properties, you can of course use the `gtk_widget_modify()` method to change the widget's font, foreground-color, and all of the other properties that a `GtkTextView` inherits from `GtkWidget`. As an example, the following function changes several of these properties all at once, given a `GtkTextView` widget and a list of values to set in it.

```
void  set_text_view_properties( GtkTextView           *textview,
                                PangoFontDescription  *font,
                                GtkWrapMode           wrap_mode,
                                GtkJustification      justification,
                                gboolean              is_editable,
                                gboolean              is_visible_cursor,
                                gint                  left_margin,
                                gint                  right_margin
                              )
{
    gtk_widget_modify_font(GTK_WIDGET(textview), font);
    gtk_text_view_set_wrap_mode(textview, wrap_mode);
    gtk_text_view_set_justification(textview, justification);
    gtk_text_view_set_editable( textview, is_editable);
    gtk_text_view_set_cursor_visible( textview, is_visible_cursor);
    gtk_text_view_set_left_margin( textview, left_margin);
    gtk_text_view_set_right_margin( textview, right_margin);
}
```

The demo program `textview_demo2.c` in the `textviews` directory contains a few controls in the main application window that the user can use to modify the font, the editability, and the justification of the text on display. It can be generalized to modify all of the attributes, but in this case, one should put all of the input widgets into a dialog box launched with an `OPTIONS` or a `PREFERENCES` button.

Modifying the tabs property is a harder task than the others, because it requires creating a `PangoTabArray` containing the tab positions that you want to set. A `PangoTabArray` is a subclass of `GBoxed`, which is a generic wrapper mechanism for arbitrary C structures defined in the GObject library. To create a `PangoTabArray`, you can use

```
PangoTabArray *  pango_tab_array_new           ( gint initial_size,
                                                 gboolean positions_in_pixels);
```

which is given the number of tab stops that the array should contain as well as a flag indicating whether or not the stop positions are given in pixels. The alternative is that the positions are given in *Pango units*, which are the units that Pango uses internally. They are much higher resolution than pixels: one pixel is 1024 Pango units. If you use this function, you will have to add each stop one at a time using

```
void            pango_tab_array_set_tab        ( PangoTabArray *tab_array,
                                                 gint tab_index,
                                                 PangoTabAlign alignment,
                                                 gint location);
```

This function is given the Pango tab array to fill, the index within the arry of the tab stop to be set, the value `PANGO_TAB_LEFT`, and finally the location of the tab stop in whatever units were specified when the array was created. The `PangoTabAlign` enumeration has only one value at present, `PANGO_TAB_LEFT`, and this must be used.

There is a second function for creating a tab array that makes it easier to create the tab stops, because they can all be added at once:

```
PangoTabArray *  pango_tab_array_new_with_positions  (gint size,
                                                       gboolean positions_in_pixels,
                                                       PangoTabAlign first_alignment,
                                                       gint first_position,
                                                       ...);
```

This has a variable-sized parameter list. It is given the number of tab stops that should be in the array (size), whether the positions are in pixels or Pango units, and then a sequence of pairs of the form (`PANGO_TAB_LEFT`, *position of tab*).

In either case you need to specify the positions of the tabs. If just picked an arbitrary value, as the font changed, the width of the tabs would not change commensurately. Therefore, you should calculate the width of the tab based on the current font and the width of a standard character, such as a black or the letter 'm'.

In general, to calculate the number of pixels in a character string that is rendering by Pango, you need to do the following:

1. Create a Pango layout object using `gtk_widget_create_pango_layout()`, which creates a new `PangoLayout` with the appropriate font map, font description, and base direction for drawing text for this widget.
2. Set the default font description for the layout using `pango_layout_set_font_description()`.
3. Get the width in pixels of the string using `pango_layout_get_pixel_size()`.

Assuming that `tabwidth_string` is a string of the number of blanks in a tab, and that all other variables are appropriately defined, the code would be

```
PangoLayout *layout     = gtk_widget_create_pango_layout(text_view, tabwidth_string);
pango_layout_set_font_description(layout, pango_font_descr);
pango_layout_get_pixel_size(layout, &width, NULL);
PangoTabArray *tab_array = pango_tab_array_new(1, TRUE);
pango_tab_array_set_tab( tab_array, 0, PANGO_TAB_LEFT, width);
```

You would then use this tab array to set the tabs in the textview with

```
gtk_text_view_set_tabs(GTK_TEXT_VIEW(text_view), tab_array);
```

and then free the tab array using `pango_tab_array_free()`.

6 Text Buffers

One thing you should know about text buffers is that they always contain at least one line, but they may contain zero characters. However, the last line in a text buffer never ends in a line separator. All other lines in the buffer always end in a line separator. Line separators count as characters when computing character counts and character offsets. This means that an empty buffer, one with zero characters is considered to have a single line with no characters in it, and in particular, no line separator at the end.

So far, the only operations on text buffers that we have used are those that set the text on the entire buffer `gtk_text_buffer_set_text()`, and the function that gets a slice ot text between two bounds, `gtk_text_buffer_get_slice ()`. In order to do much more with text buffers, though, we need to work with iterators and text marks.

Recall from the introductory comments above that a `GtkTextIterator` represents a position between two characters, that it must be declared on the stack, and that any change to the buffere that changes the character counts invalidates all current iterators.

In general, functions that insert or delete text require iterators to indicate the text to be modified. These include

```
void          gtk_text_buffer_insert            ( GtkTextBuffer *buffer,
                                                  GtkTextIter *iter,
                                                  const gchar *text,
                                                  gint len);
```

which is given `iter`, a pointer to a text iterator, `text`, a UTF-8 text string, and its length in bytes, `len`. If `len` is -1, `text` must be `NULL`-terminated and will be inserted in its entirety. This function is convenient because, even though `iter` is invalidated when insertion occurs, the text buffer's default signal handler revalidates it to point to the end of the inserted text, so that a second call will insert text immediately after the first.

Another form of insertion is

```
void          gtk_text_buffer_insert_range      ( GtkTextBuffer *buffer,
                                                  GtkTextIter *iter,
                                                  const GtkTextIter *start,
                                                  const GtkTextIter *end);
```

which copies the text from the range specified by `start` and `end`, to the position of `iter`. To be precise, it copies text, tags, and pixbufs between `start` and `end` and inserts the copy at `iter`. This is important because it preserves images and tags. The order of `start` and `end` doesn't matter – the smaller of the two will be taken to be the start and the larger, the end.

There are several other functions for inserting text, but the next most useful is

```
void          gtk_text_buffer_insert_at_cursor  ( GtkTextBuffer *buffer,
                                                  const gchar *text,
                                                  gint len);
```

which inserts the text at the position of the cursor. The length must be specified in bytes, or if `NULL`-terminated, then set to -1. The function to delete a range of text is

```
void          gtk_text_buffer_delete             ( GtkTextBuffer *buffer,
                                                   GtkTextIter *start,
                                                   GtkTextIter *end);
```

which deletes the text between `start` and `end`. Again, the order of `start` and `end` does not matter because `gtk_text_buffer_delete()` will reorder them. After the call, the iterators will be revalidated and `start` and `end` will be point to the location where the text was deleted.

If you want to delete the currently selected text, you can use

```
gboolean        gtk_text_buffer_delete_selection   ( GtkTextBuffer *buffer,
                                                     gboolean interactive,
                                                     gboolean default_editable);
```

If the `interactive` argument is `TRUE`, it means that the deletion is being requested by the user. Text marked as uneditable cannot be deleted by the user, so setting `interactive` to `TRUE` prevents uneditable text from being deleted by the user. If this function is being called by the program, not as a result of the user's request, then you should set `interactive` to `FALSE`. You might want to call this function as a result of the program's internal actions, for example, to delete hidden text.

The `default_editable` argument specifies whether the textview's editable property is `TRUE` or `FALSE`. The best way to use this is to pass the result of a call to `gtk_text_view_get_editable()`, which returns the current state of this property. For example,

```
gtk_text_buffer_delete_selection(buffer, FALSE,
            gtk_text_view_get_editable(GTK_TEXT_VIEW(text_view)));
```

These few functions, together with the `gtk_buffer_set_text()`, `gtk_buffer_get_text()`, and `gtk_buffer_get_bounds()`, are a sufficient repertoire for most editing tasks. Although there are separate functions for retrieving an iterator that with at the first position in the text buffer, and one just after the last position, the `gtk_buffer_get_bounds()` function does this anyway, so there is little need for them.

A more general function is

```
void            gtk_text_buffer_get_iter_at_offset  ( GtkTextBuffer *buffer,
                                                      GtkTextIter *iter,
                                                      gint char_offset);
```

which positions the given iterator to a position `char_offset` chars from the start of the entire buffer. If `char_offset` is -1 or greater than the number of characters in the buffer, `iter` is initialized to the end iterator, the iterator one past the last valid character in the buffer.

It is time to put these ideas together in an example. This simple example demonstrates how to use a few editing features of a textbuffer. We modify the previous example by adding a few capabilities:

- We will add a button that lets the user add line numbers to the left of every line in the text buffer. The numbers will become part of the text of the file, not just a decoration in the margin.
- We will add a button that lets the user delete selected text. Even though the user can delete selected text in two other ways (by right-clicking, and in the pop-menu choosing "Cut" or "Delete", or by using the DELETE key on the keyboard), this will be an exercise in using some of the GtkTextBuffer's functionality.

To add line numbers to the start of a line requires that we have a means of positioning an iterator at the start of a line. There is such a function that positions an iterator at the start of a given line:

```
void            gtk_text_buffer_get_iter_at_line    ( GtkTextBuffer *buffer,
                                                      GtkTextIter *iter,
                                                      gint line_number);
```

which, given the line number and the address of our locally declared iterator, will position it to the immediate left of the first character in the line, so that an insertion at that iterator (with left gravity) will put the new text just to the left of the line start. Thus, if `str` is a `NULL`-terminated string

```
gtk_text_buffer_get_iter_at_line(buffer, &iter, 100);
gtk_text_buffer_insert( buffer, &iter, str, -1);
```

will insert `str` at the start of line 100. There is a more general function that lets us position an iterator at a particular offset (number of characters) from the start of a given line:

```
void            gtk_text_buffer_get_iter_at_line_offset
                                                    ( GtkTextBuffer *buffer,
                                                      GtkTextIter *iter,
                                                      gint line_number,
                                                      gint char_offset);
```

You can see that `gtk_text_buffer_get_iter_at_line()` is just this more general one being called with `char_offset` = 0.

To number all of the lines in the buffer requires also that we can determine how many lines are in the file. While we could do this the hard way by searching for all newline characters, `GtkTextBuffer` has a function that returns the line count:

```
gint            gtk_text_buffer_get_line_count    ( GtkTextBuffer *buffer);
```

In fact it also has a function that returns the character count:

```
gint            gtk_text_buffer_get_char_count    ( GtkTextBuffer *buffer);
```

Both of these functions are fast because the counts are cached. We can therefore number lines with the following code fragment:

```
gchar   str[20];
numlines =  gtk_text_buffer_get_line_count(buffer);

for ( k = 0; k < numlines; k++ ) {
    g_snprintf(str, 20, "%5d  ", k+1);
    gtk_text_buffer_get_iter_at_line(buffer, &iter, k);
    gtk_text_buffer_insert( buffer, &iter, str, -1);
}
```

The function `g_snprintf()` is a GLib function like `sprintf()` from the C standard I/O library but safer. It formats its arguments using the format string and writes the formatted string to the string in the first argument, but it uses the second argument, in this case 20, to prevent buffer overflow by writing no more than that many characters. Like `sprintf()` it requires that the caller allocate the memory for the resulting string. In the above code snippet, the assumption is that the string will be no larger than 20 characters, including the `NULL` character. This solves the first problem.

The second problem was already solved above. We just call

```
gtk_text_buffer_delete_selection(buffer, TRUE,
            gtk_text_view_get_editable(GTK_TEXT_VIEW(text_view)));
```

which will delete the selected text, except for any text marked as uneditable.

6.1 Text Marks and the Cursor

As we mentioned before, a `GtkTextMark` represents a position between two characters. You can create marks, named or unnamed, and soon you will see why you might want to create marks. To create a mark you use

```
GtkTextMark *  gtk_text_buffer_create_mark       ( GtkTextBuffer *buffer,
                                                   const gchar *mark_name,
                                                   const GtkTextIter *where,
                                                   gboolean left_gravity);
```

This function creates a `GtkTextMark` object at the position of the given iterator and returns a pointer to it. If you supply a name, then the mark will be known by this name and you can access it with the name. This means that you do not need the return value of the function. You could call it like this:

```
gtk_text_buffer_create_mark(buffer, “midpoint”, &iter, FALSE);
```

and a mark will be created at the position of `iter`, with right-gravity instead of left (which is what you want most of the time.) If you give it `NULL` instead of a name, it will be anonymous and you need to get the return value to access it.

Marks are allocated by the buffer itself. When you create a mark, the buffer owns the reference to it, not the caller. This is why you do not need to save the return value in a variable; you will not need to free or unref it.

Remember that there are two predefined marks in a `GtkTextBuffer`, `insert` and `selection_bound`. The insert mark is the same as the cursor position when the cursor is visible in the textview. If the user moves the cursor, the insert moves to the new position. If a range of text is selected, the selection is bounded by the insert mark and the selection_bound mark.

You can get the insert mark by calling

```
GtkTextMark *  gtk_text_buffer_get_insert        ( GtkTextBuffer *buffer);
```

which returns the mark that represents the cursor (insertion point). You could also call the more general function

```
GtkTextMark *  gtk_text_buffer_get_mark           ( GtkTextBuffer *buffer,
                                                    const gchar *name);
```

giving it the string “`insert`”. Similarly, you can get the current position of the `selection_bound` mark with

```
GtkTextMark *  gtk_text_buffer_get_selection_bound ( GtkTextBuffer *buffer);
```

or by calling

```
gtk_text_buffer_get_mark(buffer, “selection_bound”)
```

When there is no text selected, the two marks are in the same position. You could therefore test whether text is selected by getting the two marks and comparing their values. However, there is a function that does this for you in a much more efficient way:

```
gboolean       gtk_text_buffer_get_has_selection  ( GtkTextBuffer *buffer);
```

You can move these marks around as well. If you move the insert mark, of course, the cursor will change position. You can move them separately or together. There are various ways of doing this. If you want to move a mark in general, you can use

```
void            gtk_text_buffer_move_mark          ( GtkTextBuffer *buffer,
                                                     GtkTextMark *mark,
                                                     const GtkTextIter *where);
```

which is given a pointer to the mark to be moved, and the address of the iterator marking the location to which to move the mark. As an iterator cannot exist in any once place too long, being easily wiped out by a change in the buffer contents, if you need a "placeholder" in a piece of text, you can put a mark where the iterator is, knowing it will stay there.

If you gave you mark a name when you created it, then you can move it by name with

```
void            gtk_text_buffer_move_mark_by_name  ( GtkTextBuffer *buffer,
                                                     const gchar *name,
                                                     const GtkTextIter *where);
```

This does the same thing, but you do not need a pointer to the mark, just its name.

The preceding functions move the mark. If you don't want to give up the spot where it currently is, you should create a new mark at that position.

Getting back to the insert and selection_bound marks, if you need to move these in particular you can use

```
void            gtk_text_buffer_place_cursor      ( GtkTextBuffer *buffer,
                                                    const GtkTextIter *where);
```

which move both of these marks to the same place, namely the position of the given iterator. Alternatively, you can move them together to non-equal positions, which basically means you are defining a new selection range. The function that does this is naturally,

```
void            gtk_text_buffer_select_range      ( GtkTextBuffer *buffer,
                                                    const GtkTextIter *ins,
                                                    const GtkTextIter *bound);
```

It is important that you use this function, rather than moving them separately, because funny things will happen on the screen.

We will return to an example that uses text marks after we cover formatting and tags.

7 Formatting and Tags

A range of text is formatted in a buffer by applying text tags to it. There are several ways to do this. You can create the tags, either named or unnamed, and store them in the text tag table in the buffer, and then apply these tags to existing text, or you can apply the tags to text as it is inserted. There is an assortment of methods for doing each of these things.

Tags can be created in several ways. To create a tag and have it automatically placed into the buffer's `GtkTextTagTable`, use

```
    GtkTextTag *   gtk_text_buffer_create_tag        ( GtkTextBuffer *buffer,
                                                       const gchar *tag_name,
                                                       const gchar *first_property_name,
                                                       ...);
```

The second parameter is the name of the tag, and after that is a list one or more attribute/value pairs. The attribute is always a string, and its value depends upon the attribute itself. The value could be another string, a number, or a more complex structure. The `GtkTextTag` class has over sixty different configurable properties. The appendix has a full list of them with their value types.

Examples

```
    gtk_text_buffer_create_tag (buffer, "italic", "style", PANGO_STYLE_ITALIC, NULL);
    gtk_text_buffer_create_tag (buffer, "monospace", "family", "monospace", NULL);
    gtk_text_buffer_create_tag (buffer, "red_background", "background", "red", NULL);

    desc = pango_font_description_from_string ("Purisa  12");
    gtk_text_buffer_create_tag (buffer, "purisa_12", "font-desc", desc, NULL);

    gtk_text_buffer_create_tag (buffer, "readonly", "editable", FALSE, NULL);
```

You can define the properties of a tag either when it is created or after creation, by modifying the tag.

(more to come)

8 Searching in the TextView

There are two functions for searching in the text view for text, one that searches forward, and one that searches backward. The prototype for the forward-searching function is

```
    gboolean       gtk_text_iter_forward_search      ( const GtkTextIter *iter,
                                                       const gchar *str,
                                                       GtkTextSearchFlags flags,
                                                       GtkTextIter *match_start,
                                                       GtkTextIter *match_end,
                                                       const GtkTextIter *limit);
```

which searches forward for `str` starting at the position in the text given by `iter`. If a match is found, `match_start` is set to the first character of the match and `match_end` to the first character after the match. The search will not continue past `limit`. Because the search algorithm is essentially a search through an unsorted list, its running time is a linear or O(n) operation. If you have a very large file, and your application knows that the search string will not be present beyond a certain position in the text, it is foolish to allow this function to search beyond that point. Therefore, if you can position an iterator at that position, then you can limit the search by providing the address of that iterator as the last parameter.

If the `GTK_TEXT_SEARCH_VISIBLE_ONLY` flag is present, the match may have invisible text interspersed in `str`. i.e. `str` will be a possibly-noncontiguous subsequence of the matched range. Similarly, if you specify `GTK_TEXT_SEARCH_TEXT_ONLY`, the match may have pixbufs or child widgets mixed inside the matched range. If these flags are not given, the match must be exact; the special `0xFFFC` character in `str` will match embedded pixbufs or child widgets. For example, if we let `<hidden>...,\hidden>` denote text tags that hide the text in between, and if we are searching for the string "`He was a liar`" in a text buffer that has the text

```
...He was <hidden>not <\hidden>a liar....
```

then with no flags, the match will fail, but with `GTK_TEXT_SEARCH_VISIBLE_ONLY` set, the match will succeed, setting `match_start` and `match_end` as illustrated.

```
...He was <hidden>not <\hidden>a liar....
  ^                                    ^
```

The function

```
gboolean      gtk_text_iter_backward_search    ( const GtkTextIter *iter,
                                                 const gchar *str,
                                                 GtkTextSearchFlags flags,
                                                 GtkTextIter *match_start,
                                                 GtkTextIter *match_end,
                                                 const GtkTextIter *limit);
```

is the same as the forward search except that it searches backwards.

A simple function to search for an exact match in a text buffer from its start and highlight the first occurrence of the found text is

```
void search (GtkTextBuffer *buffer, const gchar *text)
{
    GtkTextIter iter;
    GtkTextIter mstart, mend;
    gboolean found;

    /* Search from the start from buffer for text. */
    gtk_text_buffer_get_start_iter (buffer, &iter);
    found = gtk_text_iter_forward_search (&iter, text, 0,
                                          &mstart, &mend, NULL);

    if (found) {
        /* If found, hilight the text. */
        gtk_text_buffer_select_range (buffer, &mstart, &mend);
    }
}
```

Each time this function is called, it will start searching at the beginning of the text buffer. If we want to provide the ability to search for subsequent matches, then the application has to remember the position of the last match. We cannot use an iterator to do this, because if any textual changes are made to the buffer between the first call to the function and a subsequent call, the iterator will be invalidated. Still worse, in the above design, the iterators are local to the function and will be destroyed when the function terminates.

To be able to continue a search where we left off, we need to place a text mark at the position of the last match. Usually applications provide different proxies for finding a first occurrence and for finding the "next" occurrence. Typically there is a FIND and a FIND AGAIN menu item or button. Text barks are stored in text buffers by name, so if we create a named text mark, we can set it to the position of the iterator, and all code that needs to retrieve it can do so by using the appropriate get-method.

(to be continued...)

9 Scrolling

(to be continued...)

10 Inserting Things Other Than Text

(to be continued...)

GtkTextTag Properties

The following table lists all properties of a `GtkTextTag` object, with the type of value that can be set against them.

Property	**Type**
accumulative-margin	`gboolean`
background	`gchar*`
background-full-height	`gboolean`
background-full-height-set	`gboolean`
background-gdk	`GdkColor*`
background-set	`gboolean`
background-stipple	`GdkPixmap*`
background-stipple-set	`gboolean`
direction	`GtkTextDirection`
editable	`gboolean`
editable-set	`gboolean`
family	`gchar*`
family-set	`gboolean`
font	`gchar*`
font-desc	`PangoFontDescription*`
foreground	`gchar*`
foreground-gdk	`GdkColor*`
foreground-set	`gboolean`
foreground-stipple	`GdkPixmap*`
foreground-stipple-set	`gboolean`
indent	`gint`
indent-set	`gboolean`
invisible	`gboolean`
invisible-set	`gboolean`
justification	`GtkJustification`
justification-set	`gboolean`
language	`gchar*`
language-set	`gboolean`
left-margin	`gint`
left-margin-set	`gboolean`
name	`gchar*`
paragraph-background	`gchar*`
paragraph-background-gdk	`GdkColor*`
paragraph-background-set	`gboolean`
pixels-above-lines	`gint`
pixels-above-lines-set	`gboolean`
pixels-below-lines	`gint`

Property	Type
pixels-below-lines-set	gboolean
pixels-inside-wrap	gint
pixels-inside-wrap-set	gboolean
right-margin	gint
right-margin-set	gboolean
rise	gint
rise-set	gboolean
scale	gdouble
scale-set	gboolean
size	gint
size-points	gdouble
size-set	gboolean
stretch	PangoStretch
stretch-set	gboolean
strikethrough	gboolean
strikethrough-set	gboolean
style	PangoStyle
style-set	gboolean
tabs	PangoTabArray*
tabs-set	gboolean
underline	PangoUnderline
underline-set	gboolean
variant	PangoVariant
variant-set	gboolean
weight	gint
weight-set	gboolean
wrap-mode	GtkWrapMode
wrap-mode-set	gboolean

Listing: scrolledwindow_demo1.c

```
Listing: scrolledwindow_demo1.c
#include <gtk/gtk.h>

#define WINWIDTH     600
#define WINHEIGHT    600

typedef struct _App
{
    GtkWidget *window;
    GtkWidget *event_box;
    GtkWidget *image;
    GtkWidget *open_button;
    GtkWidget *close_button;
} AppState;

/*****************************************************************************
                         Callback Function Prototypes
*****************************************************************************/
```

```
void on_open_button_clicked      ( GtkWidget *button,
                                   AppState *app_state);

void on_close_button_clicked     ( GtkWidget *button,
                                   AppState *app_state);

/*****************************************************************************
                               Main program
*****************************************************************************/

int  main (int argc, char *argv[])
{
    GtkWidget      *alignment;
    GtkWidget      *scrolled_window;
    GtkWidget      *vbox;
    GtkWidget      *hbox;
    AppState       app_state;

    gtk_init (&argc, &argv);

    /* Create a Window. */
    app_state.window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (app_state.window),
                          "Image Viewer");

    /* Set a decent default size for the window. */
    gtk_window_set_default_size (GTK_WINDOW (app_state.window),
                                 WINWIDTH, WINHEIGHT);
    g_signal_connect (G_OBJECT (app_state.window), "destroy",
                      G_CALLBACK (gtk_main_quit),
                      NULL);

    vbox = gtk_vbox_new (FALSE, 2);
    gtk_container_add (GTK_CONTAINER (app_state.window), vbox);

    /* Create the scrollwindow and pack it into the vbox */
    scrolled_window = gtk_scrolled_window_new (NULL, NULL);
    gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                    GTK_POLICY_AUTOMATIC,
                                    GTK_POLICY_AUTOMATIC);
    gtk_box_pack_start (GTK_BOX (vbox), scrolled_window, TRUE, TRUE, 0);

    alignment = gtk_alignment_new ( 0.5, 0.5, 0, 0 );

    gtk_scrolled_window_add_with_viewport (
                    GTK_SCROLLED_WINDOW (scrolled_window), alignment );

    app_state.event_box = gtk_event_box_new ();
    gtk_container_add ( GTK_CONTAINER (alignment ), app_state.event_box );

    hbox = gtk_hbox_new (FALSE, 2);
    gtk_box_pack_start (GTK_BOX (vbox), hbox, 0, 0, 0);

    app_state.open_button = gtk_button_new_with_label ("Open Image");
    gtk_box_pack_start (GTK_BOX (hbox), app_state.open_button, 0, 0, 0);
    g_signal_connect (G_OBJECT (app_state.open_button), "clicked",
                      G_CALLBACK (on_open_button_clicked),
                      &app_state);
    app_state.close_button = gtk_button_new_with_label ("Close Image");
    gtk_box_pack_start (GTK_BOX (hbox), app_state.close_button, 0, 0, 0);
    g_signal_connect (G_OBJECT (app_state.close_button), "clicked",
                      G_CALLBACK (on_close_button_clicked),
                      &app_state);

    gtk_widget_set_sensitive(app_state.close_button, FALSE);
    gtk_widget_set_sensitive(app_state.open_button, TRUE);

    gtk_widget_show_all (app_state.window);
    gtk_main();

    return 0;
}

/******************************************************************************
                          Callback Function Definitions
******************************************************************************/

void on_open_button_clicked     ( GtkWidget *button,
                                  AppState *app_state)
{
    GtkFileFilter   *filter;
    GtkWidget       *dialog;
    int             result;
    gchar           *filename;
    GError          *error = NULL;
    GdkPixbuf       *pixbuf;

    dialog = gtk_file_chooser_dialog_new ("Select File...",
                                          GTK_WINDOW (app_state->window),
                                          GTK_FILE_CHOOSER_ACTION_OPEN,
                                          GTK_STOCK_OK, GTK_RESPONSE_ACCEPT,
                                          GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                          NULL);
    filter = gtk_file_filter_new();
    gtk_file_filter_set_name (filter, "Image Files");
    gtk_file_filter_add_pixbuf_formats(filter);
    gtk_file_chooser_add_filter(GTK_FILE_CHOOSER(dialog), filter);

    gtk_file_chooser_set_current_folder (GTK_FILE_CHOOSER (dialog),
                                         g_get_home_dir());


    result = gtk_dialog_run (GTK_DIALOG (dialog));
    switch (result) {
    case GTK_RESPONSE_ACCEPT:
         filename = gtk_file_chooser_get_filename (GTK_FILE_CHOOSER (dialog));
         gtk_widget_destroy (dialog);
         break;
    case GTK_RESPONSE_DELETE_EVENT:
    case GTK_RESPONSE_CANCEL:
    case GTK_RESPONSE_NONE:
         gtk_widget_destroy (dialog);
         return;
    }

    if (NULL == filename) {
        GtkWidget *msg;
        msg = gtk_message_dialog_new (GTK_WINDOW (app_state->window),
                                       GTK_DIALOG_MODAL,
                                       GTK_MESSAGE_ERROR, GTK_BUTTONS_OK,
                                       "Cannot open file");
        gtk_dialog_run (GTK_DIALOG (msg));
        gtk_widget_destroy (msg);
        return;
    }

    pixbuf = gdk_pixbuf_new_from_file  ( filename, &error );
    if ( error != NULL)
    {
        g_print("%s\n", error->message);
        g_error_free(error);
        error = NULL;
        return;
    }

    app_state->image = gtk_image_new_from_pixbuf ( pixbuf );
    gtk_container_add(GTK_CONTAINER(app_state->event_box), app_state->image);

    g_object_unref(G_OBJECT(pixbuf));
    g_free(filename);

    gtk_widget_set_sensitive(app_state->close_button, TRUE);
    gtk_widget_set_sensitive(app_state->open_button, FALSE);
    gtk_widget_show_all(app_state->window);
}

void on_close_button_clicked    ( GtkWidget *button,
                                  AppState *app_state)
{
    gtk_widget_destroy(app_state->image);
    gtk_widget_set_sensitive(app_state->close_button, FALSE);
    gtk_widget_set_sensitive(app_state->open_button, TRUE);
}
```

Listing: textview_demo1.c

```
Listing 2. textview_demo1.c

#include <stdio.h>
#include <stdlib.h>
#include <gtk/gtk.h>
#include <glib.h>
#include <glib/gprintf.h>

/******************************************************************************
                         Global Constant and Type Definitions
******************************************************************************/

#define WINWIDTH      600
#define WINHEIGHT     600


typedef struct _AppState
{
    GtkWidget *window;
    GtkWidget *text_view;
    GtkWidget *open_button;
    GtkWidget *print_button;
} AppState;


/*****************************************************************************
                         Callback Function Prototypes
*****************************************************************************/

void on_open_button      ( GtkWidget *button,
                           AppState *app_state);

void on_print_button     ( GtkWidget *button,
                           AppState *app_state);


/*****************************************************************************
                                 Main program
*****************************************************************************/

int main(int argc, char *argv[])
{
    GtkWidget       *scrolled_window;
    GtkWidget       *vbox;
    GtkWidget       *hbox;
    AppState        app_state;

    gtk_init (&argc, &argv);


    /* Create the top-level window */
    app_state.window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title              (GTK_WINDOW (app_state.window),
                                       "Text Viewer");

    /* Set a decent default size for the window. */
    gtk_window_set_default_size       (GTK_WINDOW (app_state.window),
                                       WINWIDTH, WINHEIGHT);
    g_signal_connect (G_OBJECT (app_state.window), "destroy",
                      G_CALLBACK (gtk_main_quit),
                      NULL);

    vbox = gtk_vbox_new (FALSE, 2);
    gtk_container_add (GTK_CONTAINER (app_state.window), vbox);

    /* Create the textview and give it some margins. */
    app_state.text_view = gtk_text_view_new ();
    gtk_text_view_set_left_margin  (GTK_TEXT_VIEW (app_state.text_view), 10);
    gtk_text_view_set_right_margin (GTK_TEXT_VIEW (app_state.text_view), 10);


    scrolled_window = gtk_scrolled_window_new(NULL, NULL);
    gtk_container_add (GTK_CONTAINER (scrolled_window), app_state.text_view);
    gtk_scrolled_window_set_policy  (GTK_SCROLLED_WINDOW (scrolled_window),
                                     GTK_POLICY_AUTOMATIC,
                                     GTK_POLICY_ALWAYS);
    gtk_box_pack_start (GTK_BOX (vbox), scrolled_window, 1, 1, 0);

    /* Put an hbox below the textview to store the buttons */
    hbox = gtk_hbox_new (FALSE, 2);
    gtk_box_pack_start (GTK_BOX (vbox), hbox, 0, 0, 0);

    /* Create the open and print buttons, pack them into the hbox,
       and attach to their callbacks.
    */
    app_state.open_button = gtk_button_new_with_label ("Open File");
    gtk_box_pack_start (GTK_BOX (hbox), app_state.open_button, 0, 0, 0);
    g_signal_connect (G_OBJECT (app_state.open_button), "clicked",
                      G_CALLBACK (on_open_button),
                      &app_state);

    app_state.print_button = gtk_button_new_with_label ("Print to Terminal");
    gtk_box_pack_start (GTK_BOX (hbox), app_state.print_button, 0, 0, 0);
    g_signal_connect (G_OBJECT (app_state.print_button), "clicked",
                      G_CALLBACK (on_print_button),
                      &app_state);

    gtk_widget_set_sensitive(app_state.open_button, TRUE);
    gtk_widget_set_sensitive(app_state.print_button, FALSE);
    gtk_widget_show_all (app_state.window);

    gtk_main ();
    return 0;
}
/*****************************************************************************
                         Callback Function Definitions
*****************************************************************************/

/*****************************************************************************
  on_open_button
*****************************************************************************/
void on_open_button     ( GtkWidget *button,
                          AppState *app_state)
{
    GtkTextBuffer  *buffer;
    GtkFileFilter  *filter;
    GtkWidget      *dialog;
    int            result;
    gchar          *filename;
    GError         *error = NULL;
    guint          nBytesInBuf;
    gchar          *contents;

    dialog = gtk_file_chooser_dialog_new ("Select File...",
                                         GTK_WINDOW (app_state->window),
                                         GTK_FILE_CHOOSER_ACTION_OPEN,
                                         GTK_STOCK_OK, GTK_RESPONSE_ACCEPT,
                                         GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                         NULL);
    filter = gtk_file_filter_new();
    gtk_file_filter_set_name(filter, "Text Files");
    gtk_file_filter_add_mime_type(filter, "text/*");
    gtk_file_chooser_add_filter(GTK_FILE_CHOOSER(dialog), filter);
    gtk_file_chooser_set_current_folder (GTK_FILE_CHOOSER (dialog),
                                         g_get_home_dir());

    /* Run the dialog modally and get the user response */
    result = gtk_dialog_run (GTK_DIALOG (dialog));
    switch (result) {
    case GTK_RESPONSE_ACCEPT:
         filename = gtk_file_chooser_get_filename (GTK_FILE_CHOOSER (dialog));
         gtk_widget_destroy (dialog);
         break;
    case GTK_RESPONSE_DELETE_EVENT:
    case GTK_RESPONSE_CANCEL:
    case GTK_RESPONSE_NONE:
         gtk_widget_destroy (dialog);
         return;
    }

    /* This should not happen, but to be safe .. */
    if (NULL == filename) {
        GtkWidget *msg;
        msg = gtk_message_dialog_new (GTK_WINDOW (app_state->window),
                                      GTK_DIALOG_MODAL,
                                      GTK_MESSAGE_ERROR, GTK_BUTTONS_OK,
                                      "Failed to get file");
        gtk_dialog_run (GTK_DIALOG (msg));
        gtk_widget_destroy (msg);
        return;
    }

    /* Obtaining the buffer associated with the widget. */
    buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (app_state->text_view));

    /* Use the filename to read its contents into a gchar* string contents */
    if (! g_file_get_contents( filename, &contents, &nBytesInBuf, &error)) {
        g_printf(error->message);
        g_clear_error(&error);
        g_free(filename);
        exit(1);
    }
    /* Success, so copy contents into buffer and free the contents and
       filename strings
    */
    gtk_text_buffer_set_text(buffer, contents, -1);
    g_free(filename);
    g_free(contents);

    gtk_widget_set_sensitive(app_state->print_button, TRUE);
    gtk_widget_show_all(app_state->window);

}

/*****************************************************************************
  on_print_button
*****************************************************************************/
void on_print_button     ( GtkWidget *button,
                           AppState *app_state)
{
    GtkTextIter start;
    GtkTextIter end;
    gchar *text;
    GtkTextBuffer  *buffer;

    buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW(app_state->text_view));
    if ( NULL != buffer ) {
            /* Obtain iters for the start and end of points of the buffer */
          gtk_text_buffer_get_bounds (buffer, &start, &end);

            /* Get the entire buffer text. */
          text = gtk_text_buffer_get_slice (buffer, &start, &end, FALSE);

            /* Print the text */
          g_print ("%s", text);
          g_free (text);
     }

}
```