Multiline Text Editing Widget

Vijay Kumar B. vijaykumar@bravegnu.org

1. Simple Multiline Text Widget

In this section, you will learn how to create a simple multiline text widget, using `GtkTextView`, for data entry.

The widget itself is created using

```
GtkWidget *gtk_text_view_new( void );
```

When the `GtkTextView` widget is created this way, a `GtkTextBuffer` object associated with this widget is also created. The `GtkTextBuffer` is responsible for storing the text and associated attributes, while the `GtkTextView` widget is responsible for displaying the text ie. it provides an I/O interface to buffer.

All text modification operations are related to the buffer object. The buffer associated with a `GtkTextView` widget can be obtained using

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

Some common operations performed on a simple multiline text widget are, setting the entire text and reading the entire text from the buffer. The entire text of the buffer can be set using

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

The `len` should be specified when the text contains `'\0'`. When the text does not contain `'\0'`and is terminated by a `'\0'`, `len` could be `-1`.

Getting the entire text of a buffer, could be a little more complicated than setting the entire text of the buffer. You will have to understand iterators. Iterators are objects that represent positions between two characters in a buffer. Iters in a buffer can be obtained using many different functions, but for our simple case the following functions can be used to get the iters at the start and end of the buffer.

```
void gtk_text_buffer_get_start_iter( GtkTextBuffer *buffer,
                                     GtkTextIter   *iter );

void gtk_text_buffer_get_end_iter( GtkTextBuffer *buffer,
                                   GtkTextIter   *iter );
```

Unlike other objects which are created using constructor like functions returning pointers to the objects, the `GtkTextIter` objects are created by instantiating the structure itself i.e they are allocated on the stack. So new `GtkTextIter` objects are created as follows

```
GtkTextIter start_iter;
GtkTextIter end_iter;
```

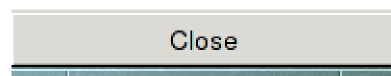
Pointers to the iterators are then passed to the above functions for initialization.

The initialized iters, can be used in the following function to retrieve the entire contents of the buffer.

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

The `include_hidden_chars` is used to specify whether are not to include text that has the *invisible* attribute set. Text attributes and how to set them will be discussed later in this tutorial. The returned string is dynamically allocated and should be freed using g_free.

Below is a sample program, that implements the simple multiline text widget. The program assigns a default text to the buffer. The text in the buffer can be modified by the user and when the close button is pressed it prints the contents of the buffer and quits.




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

void
on_window_destroy (GtkWidget *widget, gpointer data)
{
  gtk_main_quit ();
}

/* Callback for close button */
void
on_button_clicked (GtkWidget *button, GtkTextBuffer *buffer)
{
  GtkTextIter start;
  GtkTextIter end;
```

```
  gchar *text;

  /* Obtain iters for the start and end of points of the buffer */
  gtk_text_buffer_get_start_iter (buffer, &start);
  gtk_text_buffer_get_end_iter (buffer, &end);

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

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

  g_free (text);

  gtk_main_quit ();
}

int
main(int argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *vbox;
  GtkWidget *text_view;
  GtkWidget *button;
  GtkTextBuffer *buffer;

  gtk_init (&argc, &argv);

  /* Create a Window. */
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Simple Multiline Text Input");

  /* Set a decent default size for the window. */
  gtk_window_set_default_size (GTK_WINDOW (window), 200, 200);
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (on_window_destroy),
                    NULL);

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

  /* Create a multiline text widget. */
  text_view = gtk_text_view_new ();
  gtk_box_pack_start (GTK_BOX (vbox), text_view, 1, 1, 0);

  /* Obtaining the buffer associated with the widget. */
  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (text_view));
  /* Set the default buffer text. */
  gtk_text_buffer_set_text (buffer, "Hello Text View!", -1);

  /* Create a close button. */
  button = gtk_button_new_with_label ("Close");
  gtk_box_pack_start (GTK_BOX (vbox), button, 0, 0, 0);
  g_signal_connect (G_OBJECT (button), "clicked",
                    G_CALLBACK (on_button_clicked),
                    buffer);
```

```
  gtk_widget_show_all (window);

  gtk_main ();
  return 0;
}
```

1.1. More on Iterators

A point to note about text iterators - iterators are not valid indefinitely. Whenever the buffer is modified in a way that affects the number of characters in the buffer, all outstanding iterators become invalid. You will have to re-obtain iterators to use them. To preserve positions across buffer modifications the `GtkTextMark` can be used. Text marks will be discussed later in this tutorial.

1.2. UTF-8 and GTK+

GTK+ handles text in UTF-8 format. For the uninitiated, the UTF-8 is an ASCII compatible multi-byte unicode encoding. The thing to be noted is that one character can be encoded as multiple bytes. The GTK+ manual uses the term *offset* for *character counts*, and uses the term *index* for *byte counts*. The `len` argument of the `gtk_text_buffer_set_text` function is the length of the text in *bytes*.

1.3. Other Editing Functions

The following function can be used to delete text from a buffer.

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

Since this function modifies the buffer, all outstanding iterators become invalid after a call to this function. But, the iterators passed to the function are re-initialized to point to the location where the text was deleted.

The following function can be used to insert text into a buffer at a position specified by an iterator.

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

The `len` argument is similar to the `len` argument in `gtk_text_buffer_set_text` function. As with `gtk_text_buffer_delete`, the buffer is modified and hence all outstanding iterators become invalid, and `start` and `end` are re-initialized. Hence the same iterator can be used for a series of consecutive inserts.

The following function can be used to insert text at the current cursor position.

```
void gtk_text_buffer_insert_at_cursor( GtkTextBuffer *buffer,
```

```
                                      const gchar   *text,
                                      gint          len );
```

1.4. Other Functions to Obtain Iters

The following function can be used to get iterators at the beginning and end of the buffer in one go.

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

A variant of the above function can be used to obtain iterators at the beginning and end of the current selection.

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

The following functions can be used to obtain an iterator at a specified *character offset* into the buffer, at the start of the given line, at an *character offset* into a given line, or at a *byte offset* into a given line, respectively.

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

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

void gtk_text_buffer_get_iter_at_line_offset( GtkTextBuffer *buffer,
                                              GtkTextIter   *iter,
                                              gint          line_no,
                                              gint          offset );

void gtk_text_buffer_get_iter_at_line_index( GtkTextBuffer *buffer,
                                             GtkTextIter   *iter,
                                             gint          line_no,
                                             gint          index );
```

2. Formatted Text in GtkTextView

`GtkTextView` can also be used to display formatted text. This usually involves creating tags which represent a group of attributes and then applying them to a range of text.

Tag objects are associated with a buffer and are created using the following function,

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

Tags can be optionally associated with a name `tag_name`. Thus, the tag could be referred using the returned pointer or using the `tag_name`. For anonymous tags, `NULL` is passed to `tag_name`. The group of properties represented by this tag is listed as name/value pairs after the `tag_name`. The list of property/value pairs is terminated with a `NULL` pointer. `"style"`, `"weight"`, `"editable"`, `"justification"` are some common property names. The following table lists their meaning and assignable values.

Table 1. Common properties used for creating tags

Property	Meaning	Values

`"style"`	Font style as `PangoStyle`.	`PANGO_STYLE_NORMAL`
		`PANGO_STYLE_OBLIQUE`
		`PANGO_STYLE_ITALIC`

`"weight"`	Font weight as integer.	`PANGO_WEIGHT_NORMAL` `PANGO_WEIGHT_BOLD`
`"editable"`	Text modifiable by user.	`TRUE` `FALSE`

`"justification"`	Justification of text.	`GTK_JUSTIFY_LEFT`
		`GTK_JUSTIFY_RIGHT`
		`GTK_JUSTIFY_CENTER`
		`GTK_JUSTIFY_FILL`

`"foreground"`	Foreground color of text.	`"#RRGGBB"`
`"background"`	Background color of text.	`"#RRGGBB"`
`"wrap-mode"`	Text wrapping mode	`GTK_WRAP_NONE` - Don't wrap text `GTK_WRAP_CHAR` - Wrap text, breaking in between characters `GTK_WRAP_WORD` - Wrap text, breaking in between words `GTK_WRAP_WORD_CHAR` - Wrap text, breaking in words, or if that is not enough, also between characters
`"font"`	Text font specified by font description string.	`"[FAMILY-LIST] [STYLE-OPTIONS] [SIZE]"`

Here is a brief description of the font description string from the GTK+ manual.

> "`"[FAMILY-LIST] [STYLE-OPTIONS] [SIZE]"`, where FAMILY-LIST is a comma separated list of families optionally terminated by a comma, STYLE_OPTIONS is a whitespace separated list of words where each WORD describes one of style, variant, weight, or stretch, and SIZE is an decimal number (size in points). Any one of the options may be absent. If FAMILY-LIST is absent, then the family_name field of the resulting font description will be initialized to NULL. If STYLE-OPTIONS is missing, then all style options will be set to the default values. If SIZE is missing, the size in the resulting font description will be set to 0."

See the GTK+ manual (http://developer.gnome.org/doc/API/2.0/gtk/), for a complete list of properties and their

corresponding values.

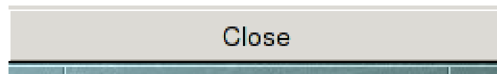
The created tag can then be applied to a range of text using the following functions,

```
void gtk_text_buffer_apply_tag( GtkTextBuffer     *buffer,
                                GtkTextTag        *tag,
                                const GtkTextIter *start,
                                const GtkTextIter *end );

void gtk_text_buffer_apply_tag_by_name( GtkTextBuffer     *buffer,
                                        const gchar       *name
                                        const GtkTextIter *start,
                                        const GtkTextIter *end );
```

The first function specifies the tag to be applied by a `tag` object and the second function specifies the tag by it's `name`. The range of text over with the tag is to applies is specified by the `start` and `end` iters.

Below is an extension of the previous example, that has a tool-bar to apply different tags to selected regions of text.

Hello Text View!




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

void
on_window_destroy (GtkWidget *widget, gpointer data)
{
  gtk_main_quit ();
}

/* Callback for buttons in the toolbar. */
void
on_format_button_clicked (GtkWidget *button, GtkTextBuffer *buffer)
{
  GtkTextIter start, end;
  gchar *tag_name;

  /* Get iters at the beginning and end of current selection. */
  gtk_text_buffer_get_selection_bounds (buffer, &start, &end);
  /* Find out what tag to apply. The key "tag" is set to the
     appropriate tag name when the button widget was created. */
  tag_name = g_object_get_data (G_OBJECT (button), "tag");
```

```
  /* Apply the tag to the selected text. */
  gtk_text_buffer_apply_tag_by_name (buffer, tag_name, &start, &end);
}

/* Callback for the close button. */
void
on_close_button_clicked (GtkWidget *button, GtkTextBuffer *buffer)
{
  GtkTextIter start;
  GtkTextIter end;

  gchar *text;

  /* Get iters at the beginning and end of the buffer. */
  gtk_text_buffer_get_bounds (buffer, &start, &end);
  /* Retrieve the text. */
  text = gtk_text_buffer_get_text (buffer, &start, &end, FALSE);
  /* Print the text. */
  g_print ("%s", text);

  g_free (text);

  gtk_main_quit ();
}

int
main(int argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *vbox;
  GtkWidget *bbox;

  GtkWidget *bold_button;
  GtkWidget *italic_button;
  GtkWidget *font_button;

  GtkWidget *text_view;
  GtkTextBuffer *buffer;

  GtkWidget *close_button;

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Formatted multiline text widget");
  gtk_window_set_default_size (GTK_WINDOW (window), 200, 200);
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (on_window_destroy),
                    NULL);

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

  /* Create a button box that will serve as a toolbar. */
  bbox = gtk_hbutton_box_new ();
  gtk_box_pack_start (GTK_BOX (vbox), bbox, 0, 0, 0);
```

```
/* Create the text view widget and set some default text. */
text_view = gtk_text_view_new ();
gtk_box_pack_start (GTK_BOX (vbox), text_view, 1, 1, 0);
buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (text_view));
gtk_text_buffer_set_text (buffer, "Hello Text View!", -1);

/* Create tags associated with the buffer. */
/* Tag with weight bold and tag name "bold" . */
gtk_text_buffer_create_tag (buffer, "bold",
                            "weight", PANGO_WEIGHT_BOLD,
                            NULL);
/* Tag with style italic and tag name "italic". */
gtk_text_buffer_create_tag (buffer, "italic",
                            "style", PANGO_STYLE_ITALIC,
                            NULL);
/* Tag with font fixed and tag name "font". */
gtk_text_buffer_create_tag (buffer, "font",
                            "font", "fixed",
                            NULL);

/* Create button for bold and add them the to the tool bar. */
bold_button = gtk_button_new_with_label ("Bold");
gtk_container_add (GTK_CONTAINER (bbox), bold_button);
/* Connect the common signal handler on_format_button_clicked. This
   signal handler is common to all the buttons. A key called "tag"
   is associated with the buttons, which speicifies the tag name
   that the button is supposed to apply. The handler reads this key
   and applies the appropriate tag. Thus only one handler is needed
   for any number of buttons in the toolbar. */
g_signal_connect (G_OBJECT (bold_button), "clicked",
                  G_CALLBACK (on_format_button_clicked),
                  buffer);
g_object_set_data (G_OBJECT (bold_button), "tag", "bold");

/* Create button for italic. */
italic_button = gtk_button_new_with_label ("Italic");
gtk_container_add (GTK_CONTAINER (bbox), italic_button);
g_signal_connect (G_OBJECT (italic_button), "clicked",
                  G_CALLBACK (on_format_button_clicked),
                  buffer);
g_object_set_data (G_OBJECT (italic_button), "tag", "italic");

/* Create button for fixed font. */
font_button = gtk_button_new_with_label ("Font Fixed");
gtk_container_add (GTK_CONTAINER (bbox), font_button);
g_signal_connect (G_OBJECT (font_button), "clicked",
                  G_CALLBACK (on_format_button_clicked),
                  buffer);
g_object_set_data (G_OBJECT (font_button), "tag", "font");

/* Create the close button. */
close_button = gtk_button_new_with_label ("Close");
gtk_box_pack_start (GTK_BOX (vbox), close_button, 0, 0, 0);
g_signal_connect (G_OBJECT (close_button), "clicked",
                  G_CALLBACK (on_close_button_clicked),
                  buffer);
```

```
  gtk_widget_show_all (window);

  gtk_main ();
  return 0;
}
```

2.1. More Functions for Applying and Removing tags

In the previous section, the `gtk_text_buffer_insert` function was introduced. A variant of this function can be used to insert text with tags applied.

```
void gtk_text_buffer_insert_with_tags( GtkTextBuffer *buffer,
                                       GtkTextIter   *iter,
                                       const gchar   *text,
                                       gint          len,
                                       GtkTextTag    *first_tag,
                                       ... );
```

The tags argument list is terminated by a `NULL` pointer. The `_by_name` suffixed variants are also available, in which the tags to be applied are specified by the tag names.

Tags applied to a range of text can be removed by using the following function

```
void gtk_text_buffer_remove_tag( GtkTextBuffer     *buffer,
                                 GtkTextTag        *tag,
                                 const GtkTextIter *start,
                                 const GtkTextIter *end );
```

This function also has the `_by_name` prefixed variant.

All tags on a range of text can be removed in one go using the following function

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

2.2. Formatting the Entire Widget

The above functions apply attributes to portions of text in a buffer. If attributes have to be applied for the entire `GtkTextView` widget, the `gtk_text_view_set_*` functions can be used. For example, the following function makes `text_view` editable/non-editable.

```
void gtk_text_view_set_editable( GtkTextView *text_view,
                                 gboolean setting );
```

See the GTK+ manual (http://developer.gnome.org/doc/API/2.0/gtk/) for a complete list of available functions.

The attributes set by these functions, on the entire widget, can be overridden by applying tags to portions of text in the buffer.

3. Cut, Copy and Paste

In this section you will learn how to do common clipboard related activities like cut, copy and paste. First, you will have to get hold of a clipboard object using

```
GtkClipboard *gtk_clipboard_get( GdkAtom selection );
```

Usually, a value of `GDK_NONE` is passed to `selection`, which gives the default clipboard. A value of `GDK_SELECTION_PRIMARY` identifies the primary X selection.

Selected text can be then copied to the clipboard using

```
void gtk_text_buffer_copy_clipboard( GtkTextBuffer *buffer,
                                     GtkClipboard *clipboard );
```

The `clipboard` is a clipboard object obtained from `gtk_clipboard_get`.

Selected text can be cut to the clipboard using

```
void gtk_text_buffer_cut_clipboard( GtkTextBuffer *buffer,
                                    GtkClipboard *clipboard,
                                    gboolean default_editable );
```

For portions of the selected text that do not have the editable tag applied, the edit-ability is assumed from `default_editable`.

Text can be pasted from the clipboard using

```
void gtk_text_buffer_paste_clipboard( GtkTextBuffer *buffer,
                                      GtkClipboard *clipboard,
                                      GtkTextIter *override_location,
                                      gboolean default_editable );
```

If `override_location` is not `NULL`, text is inserted at the iter specified by `override_location`. Else, text is inserted at the current cursor location.

4. Searching

In this section, you will learn how to search through a text buffer. Along the way you will learn about marks, as well. We will start of with implementing a basic search, and then add more features to it.

4.1. Simple Search

The following functions can be used to search for a given text within a buffer.

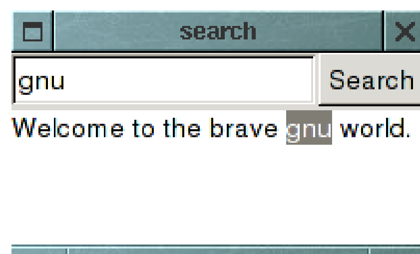
```
gboolean gtk_text_iter_forward_search( const GtkTextIter *iter,
                                       const gchar *str,
                                       GtkTextSearchFlags flags,
                                       GtkTextIter *match_start,
                                       GtkTextIter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

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

The function `gtk_text_iter_forward_search` searches for `str` starting from `iter` in the forward direction. If `match_start` and `match_end` are not NULL, the start and end iters of the first matched string are stored in them. The search is limited to the iter `limit`, if specified. The function returns `TRUE`, if a match is found. The function `gtk_text_iter_backward_search` is same as `gtk_text_iter_forward_search` but, as its name suggests, it searches in the backward direction.

The function `gtk_buffer_selection_bounds` was introduced earlier, to obtain the iters around the current selection. To set the current selection programmatically the following function can be used.

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

The function sets the selection bounds of `buffer` to `start` and `end`. The following example which demonstrates searching, uses this function to highlight matched text.




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

typedef struct _App
{
  GtkWidget *text_view;
  GtkWidget *search_entry;
} App;

/* Called when main window is destroyed. */
void
win_destroy (void)
```

```
{
  gtk_main_quit();
}

/* Called when search button is clicked. */
void
search_button_clicked (GtkWidget *search_button, App *app)
{
  const gchar *text;
  GtkTextBuffer *buffer;
  GtkTextIter iter;
  GtkTextIter mstart, mend;
  gboolean found;

  text = gtk_entry_get_text (GTK_ENTRY (app->search_entry));

  /* Get the buffer associated with the text view widget. */
  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (app->text_view));
  /* 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);
    }
}

int
main (int argc, char *argv[])
{
  GtkWidget *win;
  GtkWidget *vbox;
  GtkWidget *hbox;
  GtkWidget *search_button;
  GtkWidget *swindow;

  App app;

  gtk_init (&argc, &argv);

  /* Create a window with a search entry, search button and a text
     area. */
  win = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT (win), "destroy", win_destroy, NULL);

  vbox = gtk_vbox_new (FALSE, 2);
  gtk_container_add (GTK_CONTAINER (win), vbox);

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

  app.search_entry = gtk_entry_new ();
  gtk_box_pack_start (GTK_BOX (hbox), app.search_entry, TRUE, TRUE, 0);

  search_button = gtk_button_new_with_label ("Search");
```

```
  gtk_box_pack_start (GTK_BOX (hbox), search_button, FALSE, FALSE, 0);
  g_signal_connect (G_OBJECT (search_button), "clicked",
                    G_CALLBACK (search_button_clicked), &app);

  /* A scrolled window which automatically displays horizontal and
     vertical scrollbars when the text exceeds the text view's size. */
  swindow = gtk_scrolled_window_new (NULL, NULL);
  gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (swindow),
                                  GTK_POLICY_AUTOMATIC,
                                  GTK_POLICY_AUTOMATIC);
  gtk_box_pack_start (GTK_BOX (vbox), swindow, TRUE, TRUE, 0);

  app.text_view = gtk_text_view_new ();
  gtk_container_add (GTK_CONTAINER (swindow), app.text_view);

  gtk_widget_show_all (win);

  gtk_main();
}
```

4.2. Continuing your Search

If you had executed the above program you would have noted that, if there were more than one occurrence of the text in the buffer, pressing search will only highlight the first occurrence of the text. To provide a feature similarly to *Find Next*;, the program has to remember the location where the previous search stopped. So that you can start searching from that location. And this should happen even if the buffer were modified between the two searches.

We could store the `match_end` iter passed on `gtk_text_iter_forward_search` and use it as the starting point for the next search. But the problem is that if the buffer were modified in between, the iter would get invalidated. This takes us to marks.

4.2.1. Marks

A mark preserves a position in the buffer between modifications. This is possible because their behavior is defined when text is inserted or deleted. Quoting from gtk+ manual

> When text containing a mark is deleted, the mark remains in the position originally occupied by the deleted text. When text is inserted at a mark, a mark with left gravity will be moved to the beginning of the newly-inserted text, and a mark with right gravity will be moved to the end.

The gravity of the mark is specified while creation. The following function can be used to create a mark associated with a buffer.

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

The iter `where` specifies a position in the buffer which has to be *marked*. `left_gravity` determines how the mark moves when text is inserted at the mark. The `mark_name` is a string that can be used to identify the mark. If `mark_name` is specified, the mark can be retrieved using the following function.

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

With named tags, you do not have to carry around a pointer to the marker, which can be easily retrieved using `gtk_text_buffer_get_mark`.

A mark by itself cannot be used for buffer operations, it has to converted into an iter just before buffer operations are to be performed. The following function can be used to convert a mark into iter

```
void gtk_text_buffer_get_iter_at_mark( GtkTextBuffer *buffer,
                                       GtkTextIter *iter,
                                       GtkTextMark *mark );
```

We now know sufficient functions to implement the *Find Next* functionality. Here's the code that does that.

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

void find (GtkTextView *text_view, const gchar *text, GtkTextIter *iter)
{
  GtkTextIter mstart, mend;
  GtkTextBuffer *buffer;
  gboolean found;

  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (text_view));
  found = gtk_text_iter_forward_search (iter, text, 0, &mstart, &mend, NULL);

  if (found)
    {
      gtk_text_buffer_select_range (buffer, &mstart, &mend);
      gtk_text_buffer_create_mark (buffer, "last_pos", &mend, FALSE);
    }
}

#include <gtk/gtk.h>

typedef struct _App {
```

```
  GtkWidget *text_view;
  GtkWidget *search_entry;
} App;

void
win_destroy (void)
{
  gtk_main_quit();
}

void
next_button_clicked (GtkWidget *next_button, App *app)
{
  const gchar *text;
  GtkTextBuffer *buffer;
  GtkTextMark *last_pos;
  GtkTextIter iter;

  text = gtk_entry_get_text (GTK_ENTRY (app->search_entry));

  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (app->text_view));

  last_pos = gtk_text_buffer_get_mark (buffer, "last_pos");
  if (last_pos == NULL)
    return;

  gtk_text_buffer_get_iter_at_mark (buffer, &iter, last_pos);
  find (GTK_TEXT_VIEW (app->text_view), text, &iter);
}

void
search_button_clicked (GtkWidget *search_button, App *app)
{
  const gchar *text;
  GtkTextBuffer *buffer;
  GtkTextIter iter;

  text = gtk_entry_get_text (GTK_ENTRY (app->search_entry));

  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (app->text_view));
  gtk_text_buffer_get_start_iter (buffer, &iter);

  find (GTK_TEXT_VIEW (app->text_view), text, &iter);
}

int
main (int argc, char *argv[])
{
  GtkWidget *win;
  GtkWidget *vbox;
  GtkWidget *hbox;
  GtkWidget *search_button;
  GtkWidget *next_button;
  GtkWidget *swindow;

  App app;
```

```
  gtk_init (&argc, &argv);

  win = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT (win), "destroy", win_destroy, NULL);

  vbox = gtk_vbox_new (FALSE, 2);
  gtk_container_add (GTK_CONTAINER (win), vbox);

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

  app.search_entry = gtk_entry_new ();
  gtk_box_pack_start (GTK_BOX (hbox), app.search_entry, TRUE, TRUE, 0);

  search_button = gtk_button_new_with_label ("Search");
  gtk_box_pack_start (GTK_BOX (hbox), search_button, FALSE, FALSE, 0);
  g_signal_connect (G_OBJECT (search_button), "clicked",
                    G_CALLBACK (search_button_clicked), &app);

  next_button = gtk_button_new_with_label ("Next");
  gtk_box_pack_start (GTK_BOX (hbox), next_button, FALSE, FALSE, 0);
  g_signal_connect (G_OBJECT (next_button), "clicked",
                    G_CALLBACK (next_button_clicked), &app);

  swindow = gtk_scrolled_window_new (NULL, NULL);
  gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (swindow),
                                  GTK_POLICY_AUTOMATIC,
                                  GTK_POLICY_AUTOMATIC);
  gtk_box_pack_start (GTK_BOX (vbox), swindow, TRUE, TRUE, 0);

  app.text_view = gtk_text_view_new ();
  gtk_container_add (GTK_CONTAINER (swindow), app.text_view);

  gtk_widget_show_all (win);

  gtk_main();

  return 0;
}
```

4.3. The Scrolling Problem

There is a small problem with the above example. It does not scroll to the matched text. This can be irritating when the matched text is not in the visible portion of the buffer.

The function to scroll to a position in the buffer is

```
void gtk_text_view_scroll_mark_onscreen( GtkTextView *text_view,
                                         GtkTextMark *mark );
```

`mark` specifies the position to scroll to. Note that this is a method of the `GtkTextView` object rather than a buffer object. Since it does not change the contents of the buffer, it only changes the way a buffer is *viewed*

The example below uses `gtk_text_view_scroll_mark_onscreen` and fixes the scrolling problem.

```
void
find (GtkTextView *text_view, const gchar *text, GtkTextIter *iter)
{
  GtkTextIter mstart, mend;
  gboolean found;
  GtkTextBuffer *buffer;
  GtkTextMark *last_pos;

  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (text_view));
  found = gtk_text_iter_forward_search (iter, text, 0, &mstart, &mend, NULL);

  if (found)
    {
      gtk_text_buffer_select_range (buffer, &mstart, &mend);
      last_pos = gtk_text_buffer_create_mark (buffer, "last_pos",
                                              &mend, FALSE);
      gtk_text_view_scroll_mark_onscreen (text_view, last_pos);
    }
}
```

The `gtk_text_view_scroll_mark_onscreen` function scrolls just enough, for `mark` to be visible. But, what if you want `mark` to be centered, or to be the first line on the screen. This can be done using

```
void gtk_text_view_scroll_to_mark( GtkTextView *text_view,
                                   GtkTextMark *mark,
                                   gdouble within_margin,
                                   gboolean use_align,
                                   gdouble xalign,
                                   gdouble yalign );
```

The GTK manual explains this function -

> Scrolls `text_view` so that `mark` is on the screen in the position indicated by `xalign` and `yalign`. An alignment of 0.0 indicates left or top, 1.0 indicates right or bottom, 0.5 means center. If `use_align` is `FALSE`, the text scrolls the minimal distance to get the mark onscreen, possibly not scrolling at all. The effective screen for purposes of this function is reduced by a margin of size `within_margin`.

4.4. More on Marks

When a mark is no longer required, it can be deleted using

```
void gtk_text_buffer_delete_mark( GtkTextBuffer *buffer,
                                  GtkTextMark *mark );

void gtk_text_buffer_delete_mark_by_name( GtkTextBuffer *buffer,
                                          const gchar *name );
```

4.5. Built-in Marks

There are two marks built-in to `GtkTextBuffer` -- "insert" and "selection_bound". The "insert" mark refers to the cursor position, also called the insertion point. A selection is bounded by two marks. One is the "insert" mark and the other is "selection_bound" mark. When no text is selected the two marks are in the same position.

5. Examining and Modifying Text

Examining and modifying text is yet another common operation performed on text buffers. Examples: converting a selected portion of text into a comment while editing a program, determining and inserting the correct end tag while editing HTML, inserting a pair of HTML tags around the current word, etc. The `GtkTextIter` object provides functions to do such processing.

In this section we will develop two programs to demonstrate these functions. The first program will insert start/end `li` tags(not to be confused with text attribute tags) around the current line, when a button is clicked. The second program will insert an end tag for an unclosed start tag.

To insert tags around the current line, we first obtain an iter at the current cursor position. Then we move the iter to the beginning of the line, insert the start tag, move the iter to the end of the line, and insert the end tag.

An iter can be moved to a specified offset in the same line using

```
void gtk_text_iter_set_line_offset( GtkTextIter *iter,
                                    gint char_on_line );
```

The function moves `iter` within the line, to the character offset specified by `char_on_line`. If `char_on_line` is equal to the no. of characters in the line, the `iter` is moved to the start of the next line.

A character offset of zero, will move the iter to the beginning of the line. The iter can be moved to the end of the line using

```
gboolean gtk_text_iter_forward_to_line_end( GtkTextIter *iter );
```

Now that we know the functions required to implement the first program, here’s the code.

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

void
on_window_destroy (GtkWidget *widget, gpointer data)
{
  gtk_main_quit ();
}
```

```
/* Callback for close button */
void
on_button_clicked (GtkWidget *button, GtkTextBuffer *buffer)
{
  GtkTextIter iter;
  GtkTextIter end;
  GtkTextMark *cursor;

  gchar *text;

  /* Get the mark at cursor. */
  cursor = gtk_text_buffer_get_mark (buffer, "insert");
  /* Get the iter at cursor. */
  gtk_text_buffer_get_iter_at_mark (buffer, &iter, cursor);

  gtk_text_iter_set_line_offset (&iter, 0);
  gtk_text_buffer_insert (buffer, &iter, "<li>", -1);
  gtk_text_iter_forward_to_line_end (&iter);
  gtk_text_buffer_insert (buffer, &iter, "</li>", -1);
}

int
main(int argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *vbox;
  GtkWidget *text_view;
  GtkWidget *button;
  GtkTextBuffer *buffer;

  gtk_init (&argc, &argv);

  /* Create a Window. */
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Insert Tags");

  /* Set a decent default size for the window. */
  gtk_window_set_default_size (GTK_WINDOW (window), 200, 200);
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (on_window_destroy),
                    NULL);

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

  /* Create a multiline text widget. */
  text_view = gtk_text_view_new ();
  gtk_box_pack_start (GTK_BOX (vbox), text_view, 1, 1, 0);

  /* Obtaining the buffer associated with the widget. */
  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (text_view));
  /* Set the default buffer text. */
  gtk_text_buffer_set_text (buffer, "Item1\nItem2\nItem3", -1);

  /* Create a insert bold tags button. */
  button = gtk_button_new_with_label ("Make List Item");
  gtk_box_pack_start (GTK_BOX (vbox), button, 0, 0, 0);
```

```
  g_signal_connect (G_OBJECT (button), "clicked",
                    G_CALLBACK (on_button_clicked),
                    buffer);

  gtk_widget_show_all (window);

  gtk_main ();
  return 0;
}
```

For the second program, we will have to first get the iter at the current cursor position. We then search backwards from the cursor position, through the buffer till we hit on an unclosed tag. We then insert the corresponding end tag at the current cursor position. (Note that the procedure given does not take care of many special cases, and might not be the best way to determine an unclosed tag. But it serves our purpose of explaining text manipulation functions. Developing a perfect algorithm to determine an unclosed tag, is out of the scope of this tutorial.)

We can identify tags using the left angle bracket. So searching for start/end tags involves search for the left angle bracket. This can be done using

```
gboolean gtk_text_iter_backward_find_char( GtkTextIter *iter,
                                           GtkTextCharPredicate pred,
                                           gpointer user_data,
                                           const GtkTextIter *limit );
```

The function proceeds backwards from `iter`, and calls `pred` for each character in the buffer, with the character and `user_data` as arguments, till `pred` returns `TRUE`. (`pred` should return `TRUE` when a match is found.) If a match is found, the function moves `iter` to the matching position and returns `TRUE`. If a match is not found, the function moves `iter` to the beginning of the buffer or `limit`(if non-`NULL`) and returns `FALSE`.

For our purpose we write a predicate that returns `TRUE` when the character is a left angle bracket.

When we hit on a left angle bracket we check whether the corresponding tag is a start tag or an end tag. This is done by examining the character immediately after the left angle bracket. If it is a `'/'` it is an end tag.

To extract the character after the angle bracket we move the left angle bracket iter by one character. And then extract the character at that position. To move an iter forward by one character, the following function can be used.

```
gboolean gtk_text_iter_forward_char( GtkTextIter *iter );
```

To extract the character at an iter the following function can be used.

```
gunichar gtk_text_iter_get_char( const GtkTextIter *iter );
```

After determining the tag type we do the following,

- If the tag is an end tag, we push the tag name into a stack and then proceed to find more tags.

- If it is a start tag, we pop out it's matching tag from the stack. While poping out, if there were no more items in the stack, we have hit on an unmatched start tag! We then insert the corresponding end tag at the current cursor position.

We haven't mentioned how we extract the tag name. The tag name is extracted using two iters(start and end iter). The start iter is obtained by starting from the left angle bracket iter and searching for an alphanumeric character, in the forward direction. The end iter is obtained by starting from the start iter and searching for a non-alphanumeric character, in the forward direction. The search can be done using the `forward` variant of the `gtk_text_iter_backward_find_char`.

The code for the second example follows.

```
#include <ctype.h>
#include <string.h>

#include <gtk/gtk.h>

void
on_window_destroy (GtkWidget *widget, gpointer data)
{
  gtk_main_quit ();
}

gboolean
islangle (gunichar ch, gpointer data)
{
  if (ch == '<')
    return TRUE;
  else
    return FALSE;
}

gboolean
notalnum (gunichar ch1, gpointer data)
{
  return !isalnum (ch1);
}

/*
 * Check whether the tag at ITER is an opening tag or a closing tag.
 */
gboolean
is_closing (GtkTextIter *iter, GtkTextBuffer *buffer)
{
  GtkTextIter slash;

  slash = *iter;
  gtk_text_iter_forward_char (&slash);
  if (gtk_text_iter_get_char (&slash) == '/')
    return TRUE;
  else
    return FALSE;
}
```

```
/*
 * Returns the start/end tag at position specified by ITER.
 * Returns NULL if tag not found.
 */
char *
get_this_tag (GtkTextIter *iter, GtkTextBuffer *buffer)
{
  GtkTextIter start_tag = *iter;
  GtkTextIter end_tag;
  gboolean found;

  /* start_tag points to '<', moving to the next alphabet character
     will get the start of the tag name. */
  found = gtk_text_iter_forward_find_char (&start_tag,
                                            (GtkTextCharPredicate) isalnum,
                                            NULL, NULL);
  if (!found)
    return NULL;

  /* search for non-alnum character in the forward direction from start_tag */
  end_tag = start_tag;
  found = gtk_text_iter_forward_find_char (&end_tag,
                                            (GtkTextCharPredicate) notalnum,
                                            NULL, NULL);
  if (!found)
    return NULL;

  /* return the text between '<[/]' and non-alnum */
  return gtk_text_buffer_get_text (buffer, &start_tag, &end_tag, FALSE);
}

/*
 * Insert the closing tag specified by TAG at the current cursor
 * position.
 */
void
insert_closing_tag (GtkTextIter *iter, gchar *tag, GtkTextBuffer *buffer)
{
  char *insert;

  insert = g_strdup_printf ("</%s>", tag);
  gtk_text_buffer_insert_at_cursor(buffer, insert, strlen(insert));
  g_free (insert);
}

/* Callback for insert closing tag button */
void
on_button_clicked (GtkWidget *button, GtkTextBuffer *buffer)
{
  GtkTextIter iter;
  GQueue *stack;
  GtkTextMark *cursor;

  stack = g_queue_new ();

  /* Get the mark at cursor. */
  cursor = gtk_text_buffer_get_mark (buffer, "insert");
```

```
  /* Get the iter at cursor. */
  gtk_text_buffer_get_iter_at_mark (buffer, &iter, cursor);

  while (1)
    {
      int found;
      char *tag;
      char *tag_in_stack;

      /* Search backwards for '<'. */
      found = gtk_text_iter_backward_find_char (&iter, islangle, NULL, NULL);
      if (!found)
        break;

      tag = get_this_tag (&iter, buffer);
      if (tag == NULL)
        continue;

      if (is_closing (&iter, buffer))
        {
          /* If it is a closing tag, push it into the stack */
          g_queue_push_head(stack, tag);
        }
      else
        {
          /* If it is an opening tag, pop an item from the stack. If
             there are no items in the stack, then this tag has not
             been closed, and it is the one we should close. */
          tag_in_stack = g_queue_pop_head(stack);
          if (tag_in_stack == NULL)
            {
              insert_closing_tag (&iter, tag, buffer);
              g_free (tag);
              break;
            }
          else
            g_free (tag_in_stack);
        }
    }

  g_queue_foreach (stack, (GFunc)g_free, NULL);
  g_queue_free (stack);
}

int
main(int argc, char *argv[])
{
  GtkWidget *window;
  GtkWidget *vbox;
  GtkWidget *text_view;
  GtkWidget *button;
  GtkTextBuffer *buffer;

  gtk_init (&argc, &argv);

  /* Create a Window. */
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

```
  gtk_window_set_title (GTK_WINDOW (window), "Close Tag");

  /* Set a decent default size for the window. */
  gtk_window_set_default_size (GTK_WINDOW (window), 200, 200);
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (on_window_destroy),
                    NULL);

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

  /* Create a multiline text widget. */
  text_view = gtk_text_view_new ();
  gtk_box_pack_start (GTK_BOX (vbox), text_view, 1, 1, 0);

  /* Obtaining the buffer associated with the widget. */
  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (text_view));
  /* Set the default buffer text. */
  gtk_text_buffer_set_text (buffer,
                            "<html>\n"
                            "<head><title>Title</title></head>\n"
                            "<body>\n"
                            "<h1>Heading</h1>\n", -1);

  /* Create a close button. */
  button = gtk_button_new_with_label ("Insert Close Tag");
  gtk_box_pack_start (GTK_BOX (vbox), button, 0, 0, 0);
  g_signal_connect (G_OBJECT (button), "clicked",
                    G_CALLBACK (on_button_clicked),
                    buffer);

  gtk_widget_show_all (window);

  gtk_main ();
  return 0;
}
```

5.1. More functions to Examine and Modify Text

There a lot more functions to examine and modify text. Some of the interesting ones are listed below. You can get the complete list of available functions from the GTK+ manual.

There is a class of functions used to test for some characteristic of the text. For example to check whether the iter is at the beginning/end of word, sentence or line. The corresponding functions are

```
gboolean gtk_text_iter_starts_word( const GtkTextIter *iter );

gboolean gtk_text_iter_ends_word( const GtkTextIter *iter );

gboolean gtk_text_iter_starts_sentence( const GtkTextIter *iter );
```

```
gboolean gtk_text_iter_ends_sentence( const GtkTextIter *iter );

gboolean gtk_text_iter_starts_line( const GtkTextIter *iter );

gboolean gtk_text_iter_ends_line( const GtkTextIter *iter );
```

The family of functions based on tags and tag toggling have also not been mentioned so far. To check whether a position in the buffer starts/ends/toggles a tag the following functions can be used.

```
gboolean gtk_text_iter_begins_tag( const GtkTextIter *iter,
                                   GtkTextTag *tag );

gboolean gtk_text_iter_ends_tag( const GtkTextIter *iter,
                                 GtkTextTag *tag );

gboolean gtk_text_iter_toggles_tag( const GtkTextIter *iter,
                                    GtkTextTag *tag );
```

The return value of `toggles_tag` variant is the logical OR of `begins_tag` variant and `ends_tag` variant. If `tag` is `NULL`, the function returns `TRUE` if `iter` starts/ends/toggles *any* tag.

We can also move through a buffer based on tag toggling. To move to a position in the buffer where a particular tag toggles the following functions can be used.

```
gboolean gtk_text_iter_forward_to_tag_toggle( GtkTextIter *iter,
                                              GtkTextTag *tag );

gboolean gtk_text_iter_backward_to_tag_toggle( GtkTextIter *iter,
                                               GtkTextTag *tag );
```

If `tag` is `NULL`, toggling of *any* tag is considered.

6. Images/Widgets

A text buffer can hold images and anchor location for widgets. In this section, you will learn how to insert images and widgets. You will also learn how to retrieve an inserted image/widget.

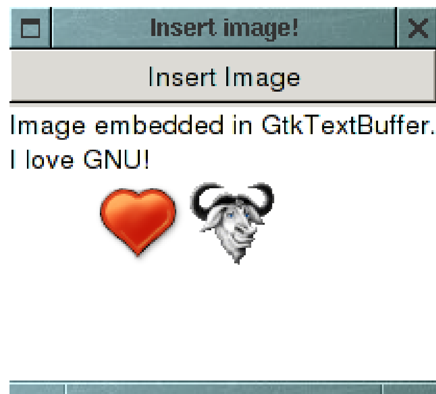
6.1. Inserting Images

An image can be inserted into a buffer using the following function

```
void gtk_text_buffer_insert_pixbuf( GtkTextBuffer *buffer,
                                    GtkTextIter *iter,
                                    GdkPixbuf *pixbuf );
```

An image represented by `pixbuf` is inserted at `iter`. The `pixbuf` can be created from an image file using `gdk_pixbuf_new_from_file`. See the gdk manual (http://developer.gnome.org/doc/API/gdk/) for more details about `GdkPixbuf` objects.

The example program given below takes in an image filename and inserts the corresponding image into a buffer.




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

typedef struct _App
{
  GtkWidget *window;
  GtkWidget *text_view;
} App;

void
on_window_destroy (GtkWidget *widget, gpointer data)
{
  gtk_main_quit ();
}

void
insert_button_clicked (GtkWidget *widget, gpointer data)
{
  App *app = (App *)data;
  GtkWidget *dialog;
  int ret;
  gchar *filename;
  GError *error = NULL;
  GdkPixbuf *pixbuf;
  GtkTextBuffer *buffer;
  GtkTextMark *cursor;
  GtkTextIter iter;

  dialog = gtk_file_chooser_dialog_new ("Image file...",
                                        GTK_WINDOW (app->window),
                                        GTK_FILE_CHOOSER_ACTION_OPEN,
                                        GTK_STOCK_OK, GTK_RESPONSE_ACCEPT,
                                        GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                        NULL);
  ret = gtk_dialog_run (GTK_DIALOG (dialog));
  switch (ret)
    {
    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:
      gtk_widget_destroy (dialog);
      /* Fall through */
    case GTK_RESPONSE_NONE:
      return;
    }

  pixbuf = gdk_pixbuf_new_from_file (filename, &error);
  g_free (filename);
  if (error)
    {
      GtkWidget *msg;
      msg = gtk_message_dialog_new (GTK_WINDOW (app->window),
                                    GTK_DIALOG_MODAL,
                                    GTK_MESSAGE_ERROR, GTK_BUTTONS_OK,
                                    error->message);
      gtk_dialog_run (GTK_DIALOG (msg));
      gtk_widget_destroy (msg);
      g_error_free (error);
      return;
    }

  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (app->text_view));
  cursor = gtk_text_buffer_get_insert (buffer);
  gtk_text_buffer_get_iter_at_mark (buffer, &iter, cursor);
  gtk_text_buffer_insert_pixbuf (buffer, &iter, pixbuf);
}

int
main(int argc, char *argv[])
{
  App app;
  GtkWidget *vbox;
  GtkWidget *insert_button;

  gtk_init (&argc, &argv);

  /* Create a Window. */
  app.window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (app.window),
                        "Insert image!");

  /* Set a decent default size for the window. */
  gtk_window_set_default_size (GTK_WINDOW (app.window), 200, 200);
  g_signal_connect (G_OBJECT (app.window), "destroy",
                    G_CALLBACK (on_window_destroy),
                    NULL);

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

  insert_button = gtk_button_new_with_label("Insert Image");
  gtk_box_pack_start (GTK_BOX (vbox), insert_button, FALSE, FALSE, 0);
  g_signal_connect (G_OBJECT (insert_button), "clicked",
                    G_CALLBACK (insert_button_clicked), &app);
```

```
  /* Create a multiline text widget. */
  app.text_view = gtk_text_view_new ();
  gtk_box_pack_start (GTK_BOX (vbox), app.text_view, 1, 1, 0);

  gtk_widget_show_all (app.window);

  gtk_main ();
  return 0;
}
```

6.2. Retrieving Images

Images in a buffer are represented by the character `0xFFFC`(Unicode object replacement character). When text containing images is retrieved from a buffer using `gtk_text_buffer_get_text` the `0xFFFC` characters representing images are dropped off in the returned text. If these characters representing images are required, use the *slice* variant - `gtk_text_buffer_get_slice`.

The image at a given position can be retrieved using

```
GdkPixbuf* gtk_text_iter_get_pixbuf( const GtkTextIter *iter );
```

6.3. Inserting Widgets

Inserting a widget, unlike inserting an image, is a two step process. The additional complexity is due to the functionality split between GtkTextView and GtkTextBuffer.

The first step is to create and insert a `GtkTextChildAnchor`. A widget is held in a buffer using a `GtkTextChildAnchor`. A child anchor according to the GTK manual is *a spot in the buffer where child widgets can be anchored*.

A child anchor can be created and inserted into a buffer using

```
GtkTextChildAnchor* gtk_text_buffer_create_child_anchor( GtkTextBuffer *buffer,
                                                         GtkTextIter *iter );
```

Where `iter` specifies the position in the buffer, where the widget is to be inserted.

The next step is to add a child widget to the text view, at the anchor location.

```
void gtk_text_view_add_child_at_anchor( GtkTextView *text_view,
                                        GtkWidget *child,
                                        GtkTextChildAnchor *anchor );
```

An anchor can hold only one widget(it could be a container widget, which in turn can contain many widgets), unless you are doing tricky things like displaying the same buffer using different `GtkTextView` objects.

The following program inserts a button widget into a text buffer, whenever the user clicks on the *Insert* button.

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

typedef struct _App
{
  GtkWidget *window;
  GtkWidget *text_view;
} App;

void
on_window_destroy (GtkWidget *widget, gpointer data)
{
  gtk_main_quit ();
}

void
insert_button_clicked (GtkWidget *widget, gpointer data)
{
  App *app = (App *)data;
  GtkTextBuffer *buffer;
  GtkTextMark *cursor;
  GtkTextIter iter;
  GtkTextChildAnchor *anchor;
  GtkWidget *button;

  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (app->text_view));
  cursor = gtk_text_buffer_get_insert (buffer);
  gtk_text_buffer_get_iter_at_mark (buffer, &iter, cursor);
  anchor = gtk_text_buffer_create_child_anchor (buffer, &iter);
  button = gtk_button_new_with_label ("New button!");
  gtk_text_view_add_child_at_anchor (GTK_TEXT_VIEW (app->text_view),
                                     button, anchor);
  gtk_widget_show (button);
}

int
main(int argc, char *argv[])
{
  App app;
  GtkWidget *vbox;
  GtkWidget *insert_button;

  gtk_init (&argc, &argv);

  /* Create a Window. */
  app.window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (app.window),
                        "Insert button!");

  /* Set a decent default size for the window. */
  gtk_window_set_default_size (GTK_WINDOW (app.window), 200, 200);
  g_signal_connect (G_OBJECT (app.window), "destroy",
```

```
                    G_CALLBACK (on_window_destroy),
                    NULL);

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

  insert_button = gtk_button_new_with_label ("Insert button");
  gtk_box_pack_start (GTK_BOX (vbox), insert_button, FALSE, FALSE, 0);
  g_signal_connect (G_OBJECT (insert_button), "clicked",
                    G_CALLBACK (insert_button_clicked), &app);


  /* Create a multiline text widget. */
  app.text_view = gtk_text_view_new ();
  gtk_box_pack_start (GTK_BOX (vbox), app.text_view, 1, 1, 0);

  gtk_widget_show_all (app.window);

  gtk_main ();
  return 0;
}
```

6.4. Retrieving Widgets

Child anchors are also represented in the buffer using the object replacement character `0xFFFC`. Retrieving a widget is also a two step process.

First, the child anchor has to be retrieved. This can be done using

```
GtkTextChildAnchor* gtk_text_iter_get_child_anchor( const GtkTextIter *iter );
```

Next, the widget(s) associated with the child anchor has to be retrieved. This can be done using

```
GList* gtk_text_child_anchor_get_widgets( GtkTextChildAnchor *anchor );
```

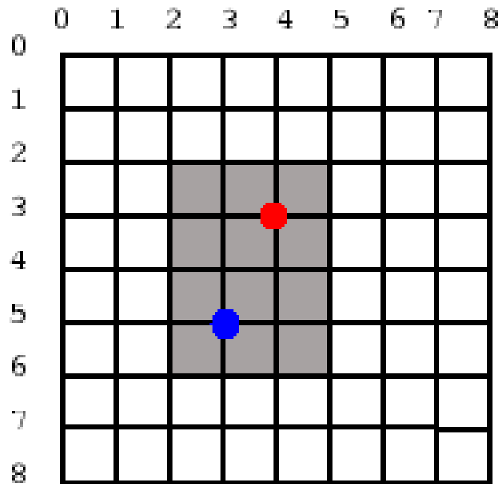
The function returns a list of widgets. As mentioned earlier, if you are not doing tricky things like multiple views for the same buffer, you will find only one widget in this list. The list has to be freed with `g_list_free`.

7. Buffer and Window Coordinates

Sometimes it is necessary to know the position of the text cursor on the screen, or the word in a buffer under the mouse cursor. For example, when you want to display the prototype of a function as a tooltip, when the user types open parenthesis. To do this, you will have to understand buffer coordinates and window coordinates.

Both the buffer and window coordinates are pixel level coordinates. The difference is that the window coordinates takes into account only the portion of the buffer displayed on the screen.

The concept would be better explained using a diagram. The large white box(with grid lines) in the following diagram depicts the text buffer. And the smaller inner grey box is the visible portion of the text buffer, displayed by the text view widget.



The buffer coordinates of the red dot(represented as (x, y)) is (4, 3). But the window coordinates of the red dot is (2, 1). This is because the window coordinates are calculated relative the visible portion of the text buffer. Similarly, the buffer coordinates of the blue dot is (3, 5) and the window coordinates is (1, 3).

7.1. Tooltips under Text Cursor

In this section, you will learn how to display tooltips under the text cursor. The procedure is as follows,

- The buffer coordinates of the text cursor is obtained.
- The buffer coordinates is converted to window coordinates (x1, y1).
- The position (x2, y2) of the text view widget on the screen is obtained.
- A window with the tooltip is displayed at (x1+x2, y1+y2).

The buffer coordinates of a particular character in a buffer can be obtained using

```
void gtk_text_view_get_iter_location( GtkTextView *text_view,
                                      const GtkTextIter *iter,
                                      GdkRectangle *location );
```

The function gets the rectangle that contains the character at `iter` and store it in `location`. The x and y members of `location` gives us the buffer coordinates.

The buffer coordinates can be converted into window coordinates using

```
void gtk_text_view_buffer_to_window_coords( GtkTextView       *text_view,
                                            GtkTextWindowType  win,
                                            gint               buffer_x,
                                            gint               buffer_y,
                                            gint              *window_x,
                                            gint              *window_y );
```

The function converts buffer coordinates (`buffer_x`, `buffer_y`), to window coordinates (`window_x`, `window_y`). You will have to pass `GTK_TEXT_WINDOW_WIDGET` for `win`, for things to work properly.(FIXME: What's the meaning of the other values of `GtkTextWidowType`?)

Now that we know the position of the character within the text view widget, we will have to find the position of the text view widget on the screen.

Each GTK widget has a corresponding `GdkWindow` associated with it. Once we know the `GdkWindow` associated with a widget, we can obtain it's X-Y coordinates using `gdk_window_get_origin`. The `GdkWindow` of the text view widget can be obtained using,

```
GdkWindow* gtk_text_view_get_window( GtkTextView *text_view,
                                     GtkTextWindowType win );
```

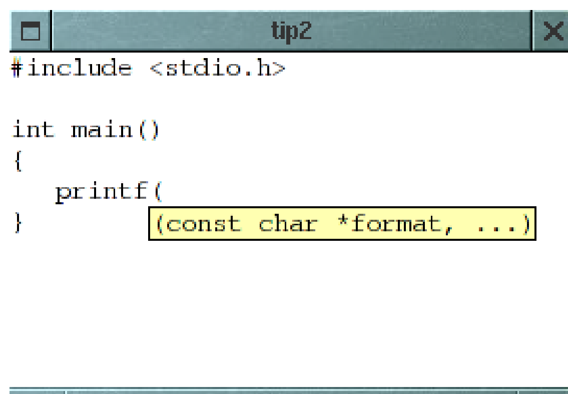
Here again you will have to pass `GTK_TEXT_WINDOW_WIDGET` for `win`.

We now know the functions required to display a tooltip under the text cursor. Before we proceed to the example, you will have to know which signal has to be trapped to do display the tooltip. Since we want the tooltip to be displayed when the user *inserts* open parenthesis, `"insert_text"` emitted by the buffer object can be used. As the signal's name suggests it is called whenever the user inserts text into the buffer. The callback prototype is

```
void (*insert_text)( GtkTextBuffer *textbuffer,
                     GtkTextIter *pos,
                     gchar *text,
                     gint length,
                     gpointer user_data )
```

The function is called with position after the inserted text `pos`, the inserted text `text` and the length of the inserted text `length`.

Below is an example program that displays a tooltip for the printf family of functions.



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

/* List of functions and their corresponding tool tips. */
static char *tips[][2] =
  {
    {"printf", "(const char *format, ...)"},
    {"fprintf", "(FILE *stream, const char *format, ...)"},
    {"sprintf", "(char *str, const char *format, ...)"},
    {"snprintf", "(char *str, size_t size, const char *format, ...)"},
    {"fputc", "(int c, FILE *stream)"},
    {"fputs", "(const char *s, FILE *stream)"},
    {"putc", "(int c, FILE *stream)"},
    {"putchar", "(int c)"},
    {"puts", "(const char *s)"},
  };

#define NUM_TIPS (sizeof (tips) / sizeof (tips[0]))

gchar *
get_tip(gchar *text)
{
  gint i;
  gboolean found;

  found = FALSE;
  for (i = 0; i < NUM_TIPS; i++)
    {
      if (strcmp (text, tips[i][0]) == 0)
        {
          found = TRUE;
          break;
        }
    }
  if (!found)
    return NULL;

  return g_strdup (tips[i][1]);
}

GtkWidget *
tip_window_new (gchar *tip)
{
  GtkWidget *win;
  GtkWidget *label;
  GtkWidget *eb;
  GdkColormap *cmap;
  GdkColor color;
  PangoFontDescription *pfd;

  win = gtk_window_new (GTK_WINDOW_POPUP);
  gtk_container_set_border_width (GTK_CONTAINER (win), 0);

  eb = gtk_event_box_new ();
  gtk_container_set_border_width (GTK_CONTAINER (eb), 1);
  gtk_container_add (GTK_CONTAINER (win), eb);

  label = gtk_label_new (tip);
```

```
  gtk_container_add (GTK_CONTAINER (eb), label);

  pfd = pango_font_description_from_string ("courier");
  gtk_widget_modify_font (label, pfd);

  cmap = gtk_widget_get_colormap (win);
  color.red = 0;
  color.green = 0;
  color.blue = 0;
  if (gdk_colormap_alloc_color (cmap, &color, FALSE, TRUE))
    gtk_widget_modify_bg (win, GTK_STATE_NORMAL, &color);
  else
    g_warning ("Color allocation failed!\n");

  cmap = gtk_widget_get_colormap (eb);
  color.red = 65535;
  color.green = 65535;
  color.blue = 45535;
  if (gdk_colormap_alloc_color (cmap, &color, FALSE, TRUE))
    gtk_widget_modify_bg (eb, GTK_STATE_NORMAL, &color);
  else
    g_warning ("Color allocation failed!\n");

  return win;
}

/* Called when main window is destroyed. */
void
win_destroy (void)
{
  gtk_main_quit();
}

void
insert_open_brace(GtkWidget **tip_win, GtkWidget *text_view, GtkTextIter *arg1)
{
  GdkWindow *win;
  GtkTextIter start;
  GdkRectangle buf_loc;
  gint x, y;
  gint win_x, win_y;
  gchar *text;
  gchar *tip_text;

  /* Get the word at cursor. */
  start = *arg1;
  if (!gtk_text_iter_backward_word_start (&start))
    return;
  text = gtk_text_iter_get_text (&start, arg1);
  g_strstrip (text);

  /* Get the corresponding tooltip. */
  tip_text = get_tip(text);
  if (tip_text == NULL)
    return;

  /* Calculate the tool tip window location. */
```

```
  gtk_text_view_get_iter_location (GTK_TEXT_VIEW (text_view), arg1,
                                   &buf_loc);
  g_printf ("Buffer: %d, %d\n", buf_loc.x, buf_loc.y);
  gtk_text_view_buffer_to_window_coords (GTK_TEXT_VIEW (text_view),
                                         GTK_TEXT_WINDOW_WIDGET,
                                         buf_loc.x, buf_loc.y,
                                         &win_x, &win_y);
  g_printf ("Window: %d, %d\n", win_x, win_y);
  win = gtk_text_view_get_window (GTK_TEXT_VIEW (text_view),
                                  GTK_TEXT_WINDOW_WIDGET);
  gdk_window_get_origin (win, &x, &y);

  /* Destroy any previous tool tip window. */
  if (*tip_win != NULL)
    gtk_widget_destroy (GTK_WIDGET (*tip_win));

  /* Create a new tool tip window and place it at the caculated
     location. */
  *tip_win = tip_window_new (tip_text);
  g_free(tip_text);
  gtk_window_move (GTK_WINDOW (*tip_win), win_x + x,
                   win_y + y + buf_loc.height);
  gtk_widget_show_all (*tip_win);
}

void
insert_close_brace (GtkWidget **tip_win)
{
  if (*tip_win != NULL)
    {
      gtk_widget_destroy (GTK_WIDGET (*tip_win));
      *tip_win = NULL;
    }
}

void
buffer_insert_text (GtkTextBuffer *textbuffer, GtkTextIter *arg1,
                    gchar *arg2, gint arg3, gpointer user_data)
{
  static GtkWidget *tip_win = NULL;

  if (strcmp (arg2, "(") == 0)
    {
      insert_open_brace(&tip_win, GTK_WIDGET (user_data), arg1);
    }

  if (strcmp (arg2, ")") == 0)
    {
      insert_close_brace(&tip_win);
    }
}

int
main (int argc, char *argv[])
{
  GtkWidget *win;
  GtkWidget *swindow;
```

```
  GtkWidget *text_view;
  GtkTextBuffer *buffer;
  PangoFontDescription *pfd;

  gtk_init (&argc, &argv);

  /* Create the window. */
  win = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT (win), "destroy", win_destroy, NULL);

  /* Create the text widget inside a scrolled window. */
  swindow = gtk_scrolled_window_new (NULL, NULL);
  gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (swindow),
                                  GTK_POLICY_AUTOMATIC,
                                  GTK_POLICY_AUTOMATIC);
  gtk_container_add (GTK_CONTAINER (win), swindow);

  text_view = gtk_text_view_new ();
  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (text_view));
  g_signal_connect (G_OBJECT (buffer), "insert_text",
                    G_CALLBACK (buffer_insert_text), text_view);
  gtk_container_add (GTK_CONTAINER (swindow), text_view);

  pfd = pango_font_description_from_string ("courier");
  gtk_widget_modify_font (text_view, pfd);

  gtk_widget_show_all (win);

  gtk_main();
}
```

7.2. More on Buffer and Window Coordinates

In the previous section we obtained the screen coordinates for a position in the buffer. What if we want to do the exact opposite, ie. what if we want to find the position in the buffer corresponding to a particular X-Y coordinate. The `GtkTextView` has functions for these as well.

Window coordinates can be converted to buffer coordinates using

```
void gtk_text_view_window_to_buffer_coords( GtkTextView *text_view,
                                            GtkTextWindowType win,
                                            gint window_x,
                                            gint window_y,
                                            gint *buffer_x,
                                            gint *buffer_y );
```

The iter at a buffer coordinate can be obtained using

```
void gtk_text_view_get_iter_at_location( GtkTextView *text_view,
```

```
GtkTextIter *iter,
gint x,
gint y );
```

8. Final Notes

8.1. GtkTextView in gtk-demo

The gtk-demo which is distributed along with the GTK+ library contains a demo on the GtkTextView and friends. It demonstrates some stuff that we have dropped here, like multiple views for a single buffer. It also has a demo on hyperlinking.

8.2. GtkSourceView

If you want to display/edit source code, you might consider using GtkSourceView and friends. They extend the GtkTextView widget, adding functionalities like

- Indentation
- Syntax Highlighting
- Bracket Matching
- Line numbering

GtkSourceView is the widget used by the well known text editor "gedit". The following example is probably the simplest implementation of the GtkSourceView widget. It creates a window with a GtkSourceView, then calls a function to load a C source file from disk and highlight its syntax.

srcview

```
#include <gtk/gtk.h>
#include <gtksourceview/gtksourceview.h>
#include <gtksourceview/gtksourcebuffer.h>
#include <gtksourceview/gtksourcelanguage.h>
#include <gtksourceview/gtksourcelanguagesmanager.h>

static gboolean open_file (GtkSourceBuffer *sBuf, const gchar *filename);

int
main( int argc, char *argv[] )
{
  static GtkWidget *window, *pScrollWin, *sView;
  PangoFontDescription *font_desc;
  GtkSourceLanguagesManager *lm;
  GtkSourceBuffer *sBuf;

  /* Create a Window. */
  gtk_init (&argc, &argv);
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);
  gtk_window_set_default_size (GTK_WINDOW(window), 660, 500);
  gtk_window_set_position (GTK_WINDOW (window), GTK_WIN_POS_CENTER);

  /* Create a Scrolled Window that will contain the GtkSourceView */
  pScrollWin = gtk_scrolled_window_new (NULL, NULL);
  gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (pScrollWin),
                                  GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);

  /* Now create a GtkSourceLanguagesManager */
  lm = gtk_source_languages_manager_new();

  /* and a GtkSourceBuffer to hold text (similar to GtkTextBuffer) */
  sBuf = GTK_SOURCE_BUFFER (gtk_source_buffer_new (NULL));
  g_object_ref (lm);
  g_object_set_data_full ( G_OBJECT (sBuf), "languages-manager",
                           lm, (GDestroyNotify) g_object_unref);
```

```
#include <gtk/gtk.h>
#include <gtksourceview/gtksourceview.h>
#include <gtksourceview/gtksourcebuffer.h>
#include <gtksourceview/gtksourcelanguage.h>
#include <gtksourceview/gtksourcelanguagesmanager.h>

static gboolean open_file (GtkSourceBuffer *sBuf, const gchar *filename);

int
main( int argc, char *argv[] )
{
  static GtkWidget *window, *pScrollWin, *sView;
  PangoFontDescription *font_desc;
  GtkSourceLanguagesManager *lm;
  GtkSourceBuffer *sBuf;

  /* Create a Window. */
  gtk_init (&argc, &argv);
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT(window),"destroy",G_CALLBACK(gtk_main_quit),NULL);
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);
  gtk_window_set_default_size (GTK_WINDOW(window), 660, 500);
```

```
  gtk_window_set_position (GTK_WINDOW (window), GTK_WIN_POS_CENTER);

  /* Create a Scrolled Window that will contain the GtkSourceView */
  pScrollWin = gtk_scrolled_window_new (NULL, NULL);
  gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (pScrollWin),
                                  GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);

  /* Now create a GtkSourceLanguagesManager */
  lm = gtk_source_languages_manager_new();

  /* and a GtkSourceBuffer to hold text (similar to GtkTextBuffer) */
  sBuf = GTK_SOURCE_BUFFER (gtk_source_buffer_new (NULL));
  g_object_ref (lm);
  g_object_set_data_full ( G_OBJECT (sBuf), "languages-manager",
                          lm, (GDestroyNotify) g_object_unref);

  /* Create the GtkSourceView and associate it with the buffer */
  sView = gtk_source_view_new_with_buffer(sBuf);
     /* Set default Font name,size */
     font_desc = pango_font_description_from_string ("mono 8");
     gtk_widget_modify_font (sView, font_desc);
     pango_font_description_free (font_desc);

  /* Attach the GtkSourceView to the scrolled Window */
  gtk_container_add (GTK_CONTAINER (pScrollWin), GTK_WIDGET (sView));
  /* And the Scrolled Window to the main Window */
  gtk_container_add (GTK_CONTAINER (window), pScrollWin);
  gtk_widget_show_all (pScrollWin);

  /* Finally load an example file to see how it works */
  open_file (sBuf, "srcview.c");

  gtk_widget_show (window);

  gtk_main();
  return 0;
}


static gboolean
open_file (GtkSourceBuffer *sBuf, const gchar *filename)
{
  GtkSourceLanguagesManager *lm;
  GtkSourceLanguage *language = NULL;
  GError *err = NULL;
  gboolean reading;
  GtkTextIter iter;
  GIOChannel *io;
  gchar *buffer;

  g_return_val_if_fail (sBuf != NULL, FALSE);
  g_return_val_if_fail (filename != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_SOURCE_BUFFER (sBuf), FALSE);

  /* get the Language for C source mimetype */
  lm = g_object_get_data (G_OBJECT (sBuf), "languages-manager");
```

```
  language = gtk_source_languages_manager_get_language_from_mime_type (lm,
"text/x-csrc");
  //g_print("Language: [%s]\n", gtk_source_language_get_name(language));

  if (language == NULL)
  {
    g_print ("No language found for mime type '%s'\n", "text/x-csrc");
    g_object_set (G_OBJECT (sBuf), "highlight", FALSE, NULL);
  }
  else
  {
   gtk_source_buffer_set_language (sBuf, language);
   g_object_set (G_OBJECT (sBuf), "highlight", TRUE, NULL);
  }

  /* Now load the file from Disk */
  io = g_io_channel_new_file (filename, "r", &err);
  if (!io)
  {
    g_print("error: %s %s\n", (err)->message, filename);
    return FALSE;
  }

  if (g_io_channel_set_encoding (io, "utf-8", &err) != G_IO_STATUS_NORMAL)
  {
   g_print("err: Failed to set encoding:\n%s\n%s", filename, (err)->message);
   return FALSE;
  }

  gtk_source_buffer_begin_not_undoable_action (sBuf);

  //gtk_text_buffer_set_text (GTK_TEXT_BUFFER (sBuf), "", 0);
  buffer = g_malloc (4096);
  reading = TRUE;
  while (reading)
  {
    gsize bytes_read;
    GIOStatus status;

    status = g_io_channel_read_chars (io, buffer, 4096, &bytes_read, &err);
    switch (status)
    {
      case G_IO_STATUS_EOF: reading = FALSE;

      case G_IO_STATUS_NORMAL:
        if (bytes_read == 0) continue;
        gtk_text_buffer_get_end_iter ( GTK_TEXT_BUFFER (sBuf), &iter);
        gtk_text_buffer_insert (GTK_TEXT_BUFFER(sBuf),&iter,buffer,bytes_read);
        break;

      case G_IO_STATUS_AGAIN: continue;

      case G_IO_STATUS_ERROR:

      default:
        g_print("err (%s): %s", filename, (err)->message);
        /* because of error in input we clear already loaded text */
```

```
        gtk_text_buffer_set_text (GTK_TEXT_BUFFER (sBuf), "", 0);

        reading = FALSE;
        break;
      }
  }
  g_free (buffer);

  gtk_source_buffer_end_not_undoable_action (sBuf);
  g_io_channel_unref (io);

  if (err)
  {
   g_error_free (err);
   return FALSE;
  }

  gtk_text_buffer_set_modified (GTK_TEXT_BUFFER (sBuf), FALSE);

  /* move cursor to the beginning */
  gtk_text_buffer_get_start_iter (GTK_TEXT_BUFFER (sBuf), &iter);
  gtk_text_buffer_place_cursor (GTK_TEXT_BUFFER (sBuf), &iter);

  g_object_set_data_full (G_OBJECT (sBuf),"filename", g_strdup (filename),
(GDestroyNotify) g_free);

  return TRUE;
}
```

To compile it, you have to specify the GtkSourceView library as an argument to `pkg-config`.

```
$ gcc -Wall -o srcview srcview.c \
> `pkg-config --cflags --libs gtk+-2.0 gtksourceview-1.0`
```

This widget offers several interesting options, here are some of them.

```
void        gtk_source_view_set_show_line_numbers
                                            (GtkSourceView *view,
                                             gboolean show);

void        gtk_source_view_set_highlight_current_line
                                            (GtkSourceView *view,
                                             gboolean show);

void        gtk_source_view_set_auto_indent (GtkSourceView *view,
                                             gboolean enable);
```

This is used to show a vertical bar on the right.

```
void        gtk_source_view_set_show_margin (GtkSourceView *view,
                                             gboolean show);
```

This specifies the column where margin will appear (number of bytes)

```
void        gtk_source_view_set_margin      (GtkSourceView *view,
                                             guint margin);
```

To use them in the example program provided above, you would write:

```
gtk_source_view_set_show_line_numbers       (GTK_SOURCE_VIEW(sView), TRUE);
gtk_source_view_set_highlight_current_line (GTK_SOURCE_VIEW(sView), TRUE);
gtk_source_view_set_auto_indent             (GTK_SOURCE_VIEW(sView), TRUE);
gtk_source_view_set_show_margin             (GTK_SOURCE_VIEW(sView), TRUE);
gtk_source_view_set_margin                  (GTK_SOURCE_VIEW(sView), 80);
```

Furthermore, the GtkSourceview handles several lang files, one for each language. These files are located in /usr/share/gtksourceview-1.0/language-specs. You can add new files there, specify or add another dir to get lang files, but this goes beyond this tutorial, which simply aims to present this great widget. For more details, see the GtkSourceView manual (http://gtksourceview.sourceforge.net/).

9. Credits

A handful of people have contributed to the development of this tutorial. They are listed below in alphabetical order.

- Bertrand-Xavier Massot contributed the description and example on GtkSourceView.
- Markus Fisher provided suggestions and feedbacks on the tutorial style.
- Vijay Kumar wrote the initial tutorial.

10. Tutorial Copyright and Permissions Notice

Copyright (C) 2005 Vijay Kumar B.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this document under the conditions for verbatim copying, provided that this copyright notice is included exactly as in the original, and that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this document into another language, under the above conditions for modified versions.

If you are intending to incorporate this document into a published work, please contact the maintainer, and we will make an effort to ensure that you have the most up to date information available.

There is no guarantee that this document lives up to its intended purpose. This is simply provided as a free resource. As such, the authors and maintainers of the information provided within can not make any guarantee that the information is even accurate.