Writing Applications with GTK+

Owen Taylor

April 28, 1998

1 Introduction

Graphical user interfaces have become almost universally familiar. However, it may be worth saying a few words about how graphical user interfaces work in Linux and X from the point of view of the programmer. The X server is responsible for only the simplest operations of drawing graphics and text on the screen and for keeping track of the user's mouse and keyboard actions. Programs communicate with the server via the Xlib library. However, programming applications in straight Xlib would be a tremendous chore. Since Xlib provides only basic drawing commands, each application would have to provide their own code to user interface elements such as buttons, or menus. (Such user interface elemenets are called *widgets*).

To avoid such a laborious job, and to provide consistancy between different applications, the normal practice is to use a toolkit - a library that builds on top of Xlib and handles the details of the user interface. The traditional choices for such a toolkit have been two libraries built upon X Intrinsics (libXt) library distributed with X, the Athena Widgets, which are distributed with X, and Motif[1]. However, Xt is complicated to learn and use, the Athena Widgets haven't looked stylish since 1985, and Motif, while somewhat more up to date, is large, slow, has an appearance disliked by many people, and, most importantly, is a proprietary product without freely available source code.

For these reasons, and others much recent development has focused on toolkits that build directly on Xlib. There are dozens of such toolkits available. Perhaps the most prominent are John Ousterhout's Tk toolkit, tightly integrated with the Tcl language, and the Qt library[2], by Troll Tech, which is written in C++.

[1] Motif is a registered trademark of The Open Group

[2] Qt is a trademark of Troll Tech AS

1.1 History

GTK+ was created as part of the GNU Image Manipulation Package (GIMP). Early revisions of the GIMP used Motif, but when a major revision was begun in 1996, Peter Mattis decided that, because of its proprietary status, Motif was not suitable, and began work a new toolkit, called the GIMP Toolkit (GTK). In early 1997, additional object oriented features were added, including inheritance and a flexible signal system for callbacks, and the toolkit was renamed from GTK to GTK+. Because the original version was never released, GTK+ is often referred to simply as GTK, and, even when written as GTK+, is typically pronounced *jee-tee-kay*, not *jee-tee-kay plus*.

1.2 Why GTK+?

GTK+ offers a number of advantages over other avaqilable toolkits. It is written entirely in C. Not only are high quality C compilers universally available, but the C-only makes it very easy to integrate with existing systems as well.

Interfaces for GTK+ exist for many different languages. Well developed bindings exist for (in addition to C), C++, guile, Perl, and Objective C. Bindings also exist or are in development for Python, Java, Eiffel, Modula-3, O'Caml, and TOM. As has long been known by users of Tcl/Tk, GUI development is much more efficient when done in a high-level "scripting" language. Although ports of Tk have been done to a number of languages other than Tcl, they have been hobbled because Tcl is heavily integrated into the Tk core, so each port has required producing a modified version of Tk, or has required interfacing to Tk through Tcl. The object-oriented design, and straight-C approach of GTK+ mean that all the above mentioned bindings use unmodified versions of GTK+.

GTK+ is licensed under the GNU Library General Public License (LGPL). his ensures the continued availability of the GTK+ source code, and modifications to it, and means allows it can be used without charge in all applications. This contrasts sharply with Motif, for which the source code is not available without paying large amounts of money, and Qt, which is free only for non-commercial use.

As you might expect for a toolkit that originated as part of a graphics program, GTK+ has a very attractive default appearance. The user interface elements are crisply defined and well spaced. The colors, fonts and background pixmaps of GTK widgets can be conveniently customized by the user via a configuration file system. If that isn't enough, work is currently in progress to add *Themeability* to GTK+Ṫhemeability, best known from its implementation the Enlightenment window manager, will allow complete customization of the user

interface without recompiling applications.

GTK+ also provides transparently provides support for a number of X extensions. With at most trivial modifications, a GTK+ program will allow the input of Chinese and Japanese text via the XIM extension, and use shared memory for images via the XShm extension. A more unusual feature of GTK+ is support for the XInput extension. Programs using the GTK+ can, again, with only minor modifications, take advantage of pressure and tilt information from a graphics tablet.

1.3 Design Principles

There are a number basic design decisions that distinguish GTK+ from other toolkits. The most basic one was mentioned above. GTK+ is written in pure C to allow the maximum portability, flexibility, and efficiency.

Another basic principle used in GTK+ is "Evenrything is a widget". In most toolkits, a button or menu item simply has a fixed text. In GTK+ buttons and menuitems are simply Container widgets. The could hold a Label widget (to display a text string), a Pixmap widget (to display a picture), and Arrow widget, a DrawingArea widget where the application draws its own graphics, and so forth.

2 General Ideas

2.1 Structure

The GTK+ package consists of three libraries. The first, `libglib`, provides two things: system-independent replacements for non-portable routines, and routines for manipulation of generic data structures. The extensive use of these data structures in the rest of GTK+ and in GTK+ applications simplifies the code, and improves performance by allowing sophisticated data structures such as hashes and caches to be conveniently used.

The second library `libgdk`, the GIMP Drawing Kit, provides a wrapper around the raw Xlib functions. This hides much of complications of X from the rest of the code, and should make porting GTK+ to other windowing systems somewhat easier. Finally, `libgtk` includes the code for the GTK+ object system, and for the widgets itself.

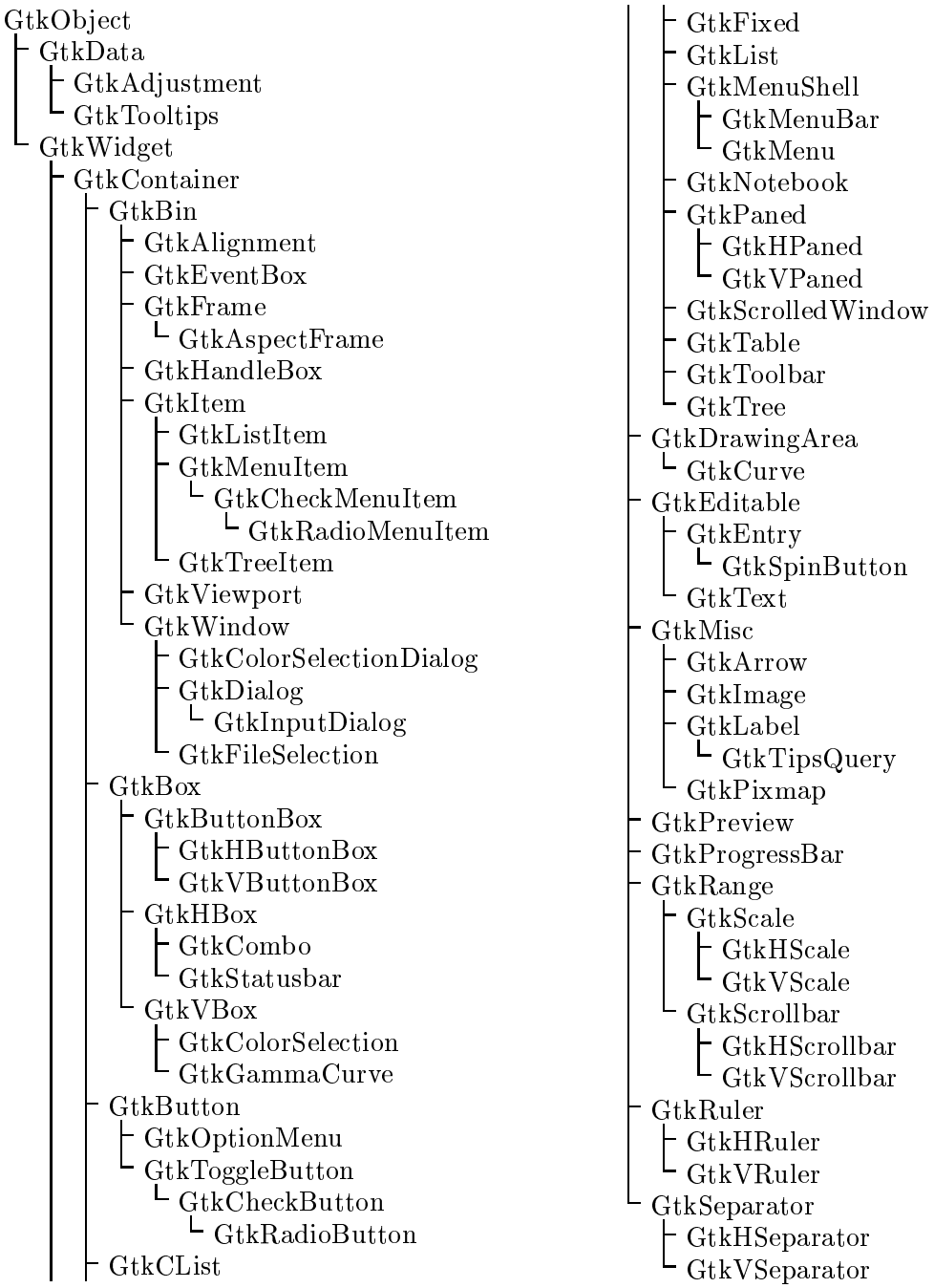

Figure 1: The GTK+ class heirarchy

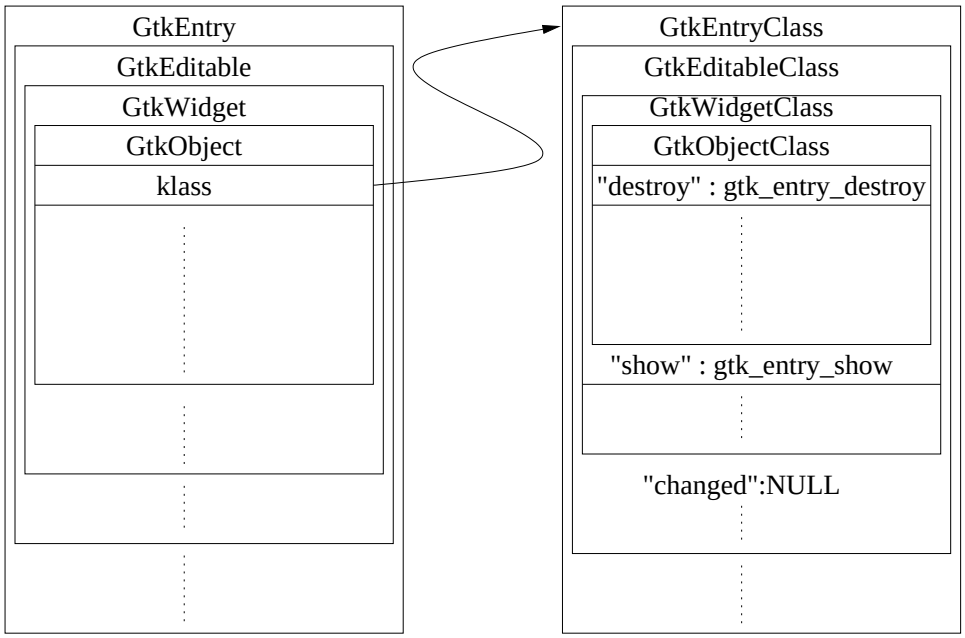

Figure 2: Memory layout of GtkEntry and GtkEntryClass structures

2.2 Inheritance

Despite being written in C, GTK+ is fully object oriented. A "method" is invoked on an object by calling a function that takes the object as its first parameter. For instance, to show a widget on the screen, one calls the function:

```
void gtk_widget_show (GtkWidget *widget);
```

As an another example, to set the text in an Entry widget one calls:

```
void *gtk_entry_set_text (GtkEntry *entry, const gchar *text);
```

A code fragment that uses hese functions looks like:

```
GtkWidget *entry = gtk_entry_new ();
gtk_entry_set_text (GTK_ENTRY (entry), "Edit Me");
gtk_widget_show (entry);
```

The interesting thing here to note is the use of the `GTK_ENTRY` macro. A `GtkWidget *` was returned by `gtk_entry_new()`, but `gtk_entry_set_text()` expects `GtkEntry *`. To get from one to the other, we use the `GTK_ENTRY()` macro. Actually, we could have written, just as well:

```
gtk_entry_set_text ((GtkEntry *)entry, "Edit Me");
```

When debugging checks in GTK+ are disabled, the `GTK_ENTRY()` macro just a convenient way to write the cast. When debugging is enabled, the macro also does some checks to make sure the pointer we are converting into a `GtkEntry *` actually points to an Entry widget. So how can the same pointer point both to a `GtkWidget` and a `GtkEntry`. Doesn't the object in memory have to to be either one or the other?

For the answer to that, see Figure 2. What is going on is that the we have nested structures. Inside the `GtkEntry` structure is a `GtkEditable` structure. Inside the `GtkEditable` structure is a `GtkWidget` structure, and inside the `GtkWidget` structure is a `GtkObject` structure. All of these structures begin at the same place in memory, so the the same pointer can be used as a `GtkEntry *`, a `GtkEditable *`, a `GtkWidget *` or a `GtkObject *`.

In this manner, GTK+ implements a heirarchy of widgets along with a few non-widget objects. (See Figure 1). At the base of the heirarchy is the GtkObject type, which provides various functionality common to all Objects, such as memory management.

There is actual a second set of structures nested in the same Russian-doll fashion. This is illustrated by our other call. `gtk_widget_show (entry)`. As you might imagine, different things are involved in showing an Entry widget on the screen, than in showing, say, a CheckButton. So how does the code in `gtk_widget_show()`, which has to handle both, know what to do? Information about such operations, is stored in a second structure nested in the same way, the *Class Structure*. There is a separate `GtkEntry` structure for each Entry widget, but only a single `GtkEntryClass` structure that all `GtkEntry` structures point to. In this `GtkEntryClass` structure is, among other things, a pointer to the function `gtk_entry_show()` which actual shows the widget. After doing some general setup, `gtk_widget_show()` calls this function which performs actions specific to showing an Entry widget.

Actually, it doesn't call that function directly, but instead uses GTK+'s signal mechanism. Which brings us to the next topic.

2.3 Signals

One of the most distinctive and flexible features of GTK+ is its signal system. Signals provide a mechanism for applications to control what happens when the user does something, by providing a *callback* - code that GTK+ calls when the event occurs. Callbacks in most C libraries are set up by simply by providing a function pointer - and this is essentially what you are doing when you call

`gtk_signal_connect`, but signals offer a number of additional benefits over a simple callback mechanism. To list a few:

- Multiple callbacks can be attached to a single signal.
- By using `gtk_signal_connect` or `gtk_signal_connect_after`, the user callback can be run before or after the default handler.
- Type information about the callback arguments is provided to language bindings, so the arguments can be converted appropriately.
- Applications can create their own signal types and attach them to existing objects.

Each type of occurrence has a separate signal type, identified by a string. Using the example of the signal that is pressed when the user clicks on a button, adding a callback to a signal looks like:

```
void
my_callback (GtkWidget *widget, gpointer data)
{
  g_print ("The button was pressed\n");
}

gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (callback), NULL);
```

The last argument, `NULL` in this example, allows arbitrary data to be passed as the second argument of the callback.

2.4 Geometry management

In keeping with the general principle that everything is a widget, layout in GTK + is done by using widgets the contain other widgets. All of these widgets inherit from the Container widget, and are known as *containers*. The most commonly used containers for layout are the HBox and VBox, which are used to group widgets into a horizontal or vertical box, and the Table widget, which allows for flexible two dimensional layout. But there are 44 other types of Containers as well. The container nature of many of these is less obvious - for instance, the Button widget, as mentioned above, is a container.

2.5 Memory Management

In a typical C program, the programmer is completely responsible for freeing memory that they have allocated. This works well if the program uses memory than frees it in a predictable manner. But imagine the difficulties in figuring out when to free the memory of a GtkObject. A pointer to a widget may be used in many places internal to GTK+ and from application code. A complex application might allow scripting in a language such as Perl, and a pointer could be kept there as well. If the memory of the object is freed before all of these pointer are removed, they will be left pointing into empty space, and a segmentation fault would be the likely consequence.

The problem of figuring out when there are no pointers to a piece of memory left is called, in general, *garbage collection*. (Because once all the pointers are gone, the object is useless - and hence "garbage" that needs to be removed) Many sophisticated schemes have been invented for doing this - most of which require extensive support from the compiler, and sometimes from the operating system.

Because, GTK+ is meant to be useable in conjunction with many different languages, and external systems, it would not be practical to implement a full garbage-collection system in GTK+. Instead, it uses a limited form of garbage-collection called *reference counting*. In reference counting, each object keeps track of the number of of pointer to that object that exist. When this count goes to zero, the object frees its memory.

When one stores a pointer to an object, one calls the function:

```
gtk_widget_ref(GtkObject *object)
```

which increments the reference count. When one no longer needs the pointer, one calls:

```
gtk_widget_unref(GtkObject *object)
```

There is one notable danger with reference counting. If a cycle exists where one object references another, and the second references the first, than neither object will ever have its reference count drop to zero, and they will never be freed. To reduce the likelyhood of this happening, GTK+ also includes a manual destruction step.

Manual destruction is a natural idea for a graphical user interface. Once an object appears on the screen, the user will typically cause it to go away through an explicit action, such as clicking a close button, or selecting 'Quit' from a menu. In response to such an action, the program will call `gtk_widget_destroy()` on the relevant widget. (For instance, on the window). This will cause all references

that the widget holds inside of GTK+ to be released, the widget to be removed from the screen, and `gtk_widget_quit()` to be called on all the widgets children. Note that there is a distinction between *destruction*, the step mentioned above and actually freeing the object, which is called *finalization*.

Most commonly, only a single reference to a GTK+ widget exists, which is held by the the container widget that holds them. (Their *parent*). Since this is removed when a widget is destroyed, the widgets reference count then drops to zero, and it is finalized when immediately after being destroyed.

2.6 The Event Loop

Programs with graphical user interfaces are by nature event driven. That is, instead of performing some computations, then exiting immediately, they wait for the user to do things, and then respond to the users actions. This is done by callbacks. The most common form of callbacks in GTK+ programs are signals. However, there are several types of callback that can be set up independent of widgets as well:

Timeouts are called at specifiedintervals. Idle functions are called when no events or timeouts need to be processed. Input callbacks are called for events on file descriptors. For instance, you might use an input callback if you were waiting for data to arrive on a socket.

3 Writing a GTK+ application

Figures 3–5 show a very short example of a GTK+ program, as implemented in C, Perl and guile. A window widget is created, a button is added to it, and a callback is set up so that when the button is clicked, "Hello World" is printed and the application quits.

Writing a substant application in GTK+ follows much the same route — One figures out how the application should look, and creates the appropriate widgets. Then one connects callbacks to the appropriate signals so the application actually does something.

4 Tips, Traps, and Techniques

Compile GTK+ with `--enable-debug` When you are developing a GTK+ program, you should use a copy of GTK+ configured with the `--enable-debug` flag. This turns on runtime checks that will help catch your programming errors when they first occur, as well as allowing the use of some very handy debugging

information that is available by setting the `GTK_DEBUG` environment variable. (See the file `docs/debugging.txt/` in the GTK+ distribution)

Make sure one window has a "destroy" handler If all of the applications windows are closed, but the event loop keeps running, then from the user's point of view, your application will appear to have hung, instead of quitting. If the user is running your application from a menu, instead of from a shell, they may well end up with 10-15 copies running at once.

To avoid this, identify a main window for your application, and make sure there is a "destroy" handler for that window that quits the main loop when the window is destroyed.

```
GtkWidget *window = gdk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_signal_connect (GTK_OBJECT (window), ``destroy'',
                    GTK_SIGNAL_FUNC (gtk_main_quit, NULL);
```

5 Future Directions for GTK+

Although GTK+ as it currently exists is suitable for writing powerful and full featured applications, work is currently beginning on a version 2 that will be even better.

Some of the things planned include, themeability, as mentioned above, more extensive internationalization, including, most likely, a switch to Unicode for the internal encoding, and the integration of the Imlib library for image loading and manipulation into the GTK+ core.

6 Further information

For further information about GTK+ and links to the language bindings mentioned above, visit `http://www.gtk.org/`. The author can be contacted at `otaylor@gtk.org`.

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

void
button_clicked (GtkWidget *button, GtkWidget *window)
{
  g_print ("Hello world\n");
  gtk_widget_destroy (window);
}

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

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_signal_connect (GTK_OBJECT (window), "destroy",
                      GTK_SIGNAL_FUNC (gtk_main_quit), NULL);

  button = gtk_button_new_with_label ("Click Me");
  gtk_container_add (GTK_CONTAINER (window), button);
  gtk_widget_show (button);

  gtk_signal_connect (GTK_OBJECT (button), "clicked",
                      GTK_SIGNAL_FUNC (button_clicked), window);

  gtk_widget_show (window);
  gtk_main();

  return 0;
}
```

Figure 3: GTK+ 'Hello World' in C

```
use Gtk;
Gtk->init;

my $window = new Gtk::Window 'toplevel';
$window->signal_connect("destroy", \&Gtk::main_quit);

my $button = new Gtk::Button "Click Me";
$window->add($button);
$button->show;

$button->signal_connect("clicked", sub {
  print "Hello\n";
  $window->destroy;
});

$window->show;
Gtk->main;
```

Figure 4: GTK+ 'Hello World' in Perl

```
(use-modules (toolkits gtk))

(let ((window (gtk-window-new 'toplevel))
      (button (gtk-button-new-with-label "Click Me")))
  (gtk-signal-connect window "destroy"
                      (lambda () (gtk-main-quit)))
  (gtk-signal-connect button "clicked"
                      (lambda ()
                        (display "Hello World")
                        (newline)
                        (gtk-widget-destroy window)))
  (gtk-container-add window button)
  (gtk-widget-show-all window))

(gtk-main)
```

Figure 5: GTK+ 'Hello World' in guile