Gtk and C++

- Gtk is widely used graphical support library, allows use of typical gui components (windows, buttons, file selects, images, arrows, titles, rulers, menubars, menus, calendars, text boxes, etc etc)
- Assumes developer is creating (potentially) complex and persistent window/menu system, e.g. window with various panes, controls, drop down menus, etc
- Program creates the various components, connects the parts together, and starts main event loop
- Event loop waits for user to do things (interact with parts of the system) responds by calling functions based on what user did

Widgets, events, handlers

- The visual components are called widgets
- The user interacting with one (e.g. clicking a button) is called an event
- A function that responds to an event is called an event handler
- Programmer needs to write code defining the widgets and how they fit together, plus the functions for the event handlers, specifies which handler gets called for which event(s), and what parameters get passed to it

Don’t let the syntax get to ya

- The concepts are pretty straightforward, some of the terminology and syntax can be rather daunting
- Each of the widgets has very specific names
- Many of the construction functions have lengthy parameter lists
- Many of the options have very specific string constants
- A number of gtk-specific data types have to be used in place of the usuals, e.g. gboolean instead of bool
- Basically have a decent gtk reference page open beside you

Tiny example (but still not pretty)

```
#include <gtk/gtk.h>
// event handler: ends main event loop (and program) when user clicks x in upper right
static void destroy(GtkWidget *widget, gpointer data) {
   gtk_main_quit();
}
int main(int argc, char* argv[]) { // argc, argv used specially by gtk
   gtk_init(&argc, &argv); // necessary setup function
   // create window widget and set its border width (as a property example)
   GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
   gtk_container_set_border_width(GTK_CONTAINER (window), 10);
   g_signal_connect(window, "destroy", G_CALLBACK(destroy), NULL);
   gtk_widget_show(window); // makes it appear on-screen
   gtk_main(); // starts event loop running
}
```

Creating widgets, adding to containers

- Widgets have type gtk_widget* (ok, they're pointers to widgets)
- Specific types of widget are created using various functions like gtk_something_new(some options), where the options depend on the kind of widget being created (function name format varies too)
- Some kinds of widget, such as windows, are containers, and can contain other widgets
- you can add a widget (b below) to a container (w) with calls like *gtk_container_add(GTK_CONTAINER(w), b);*
- Note the typecast above, casting pointer w to a generic container pointer

Event handling functions

- Most functions we write to handle events will expect two parameters, one indicating which specific widget triggered the event, the other is a generic pointer to the data available for the handler to process, e.g.:
- static void destroy(GtkWidget *w, gpointer data)
- If you want to pass some data to the handler, you create a pointer to it and cast the type to gpointer (if you want to pass multiple data items put them in a struct and pass a pointer to that)
- Inside the handler, it must know what kind of data it’s expecting, and cast the gpointer back to that, e.g. *int x = (int*)(data);*

Connecting widget events to handlers

- Once a widget has been created (e.g. a clickable button, b), attached to a window, w, and a handler, f, has been written for an associated event (e.g.. the actual click) we need to connect the three

```
g_signal_connect(b, “clicked” G_CALLBACK(f), w)
```

- Note the casting of pointer f to a G_CALLBACK
- Event types have specific pre-defined string constants, like “clicked”

Setting widget properties

- Each type of widget has a specific collection of properties that can be set by specific functions, e.g.

```
gtk_container_set_border_width(GTK_CONTAINER(w), 10);
```

- Unfortunately it does mean digging through gtk documentation to find the right functions to set the properties, and the expected parameters

Passing parameters to handlers

- Parameter specified in the signal connect, must be a pointer cast to a gpointer, e.g.

```
int numclicks = 0; // how often has button been clicked?
g_signal_connect(somebutton, “clicked” G_CALLBACK(myhandler),
    (gpointer(&numclicks)))
```

- Inside the handler, cast it back to right type and use it, e.g.

```
static void myhandler(GtkWidget *w, gpointer data) {
    int *clicks = (int *)(data);
    (*clicks) += 1;
```

Gtk file chooser snippet

```
GtkWidget *dialog = gtk_file_chooser_dialog_new(
      "Open File", NULL,
      GTK_FILE_CHOOSER_ACTION_OPEN,
      GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
      GTK_STOCK_OPEN, GTK_RESPONSE_ACCEPT, NULL);
  if (gtk_dialog_run(GTK_DIALOG(dialog)) == GTK_RESPONSE_ACCEPT) {
     char *filename;
     filename = gtk_file_chooser_get_filename(GTK_FILE_CHOOSER(dialog));
     printf("file chosen was: %s\n", filename);
     g_free(filename);
  }
  gtk_widget_destroy(dialog);
```

Arranging widgets in containers

- Containers can be horizontal boxes, vertical boxes, or tables
- When we add things to hboxes they’re added left-to-right or right-to-left, depending on call used (for vboxes they’re added top-down or bottom-up, for tables we specify row/column to put them in)
- We use gtk_box_pack_start() and _end() to start/stop adding things to a box
- Can be nested, e.g. a vbox in a table cell somewere in an hbox, etc ... lots of parameters and options (padding, fixed widths, ...)