Teil II: Schnittstellen zum Benutzer

Kapitel 2

GUI-Programmierung mit GTK

»Mailand oder Madrid. Egal. Hauptsache Italien« – Andy Möller

Zunächst einmal die absolute Kurzfassung: Die Abkürzung GTK steht für »Gimp Toolkit«. »Gimp« wiederum ist die Abkürzung für »GNU Image Manipulation Program«, und »GNU« ist auch eine Abkürzung, diesmal für »GNU is not Unix«. Die etwas längere Fassung lautet:

Gimp war ein Projekt von Spencer Kimball und Peter Mattis an der Universität von Kalifornien in Berkeley. Es startete als Lispprogramm, hatte aber jede Menge (Stabilitäts-)Probleme, weshalb sich die Autoren entschlossen, das Programm nochmals in C zu erstellen. Da das Programm unter dem X-Window-System (kurz X11) laufen sollte, wurde ein Widgetset gesucht, welches ihren Bedürfnissen gerecht wurde. Zunächst verwendeten sie Motif, doch Motif war nicht frei, und andere fanden auch keinen Gefallen daran. Daher entschloss man sich, ein eigenes Widgetset für das Programm zu schreiben und es in eigene Bibliotheken auszulagern. Daraus resultierte das GTK und die ergänzenden Komponenten GDK und GLib. Diese fanden Gefallen in den Augen anderer Programmierer, weshalb immer mehr Programme mit dieser Bibliothek erstellt wurden, während die Entwicklung von Gimp ab der Version 0.99 nicht vorwärts kam (die Programmierer hatten nur noch wenig Zeit für das Projekt). Der endgültige Durchbruch kam aber mit der Übernahme des GTK in das GNOME-Projekt, wo es eine wichtige Komponente wurde.

Auch für Gimp zeichnet sich ein Happy End ab, da die Sourcen einen neuen Betreuer gefunden haben, der mit der Hilfe vieler anderer Programmierer Gimp mittlerweile auf Version 1.1.x gebracht hat, und auch die Presse nimmt immer häufiger Notiz.

Dieses Kapitel ist nur eine Einführung in GTK. Der Rahmen des Buches lässt es leider nicht zu, dass wir uns auch noch mit dem Gimp oder der GNOME-Programmierung beschäftigen. Es sollte aber allemal reichen, um die ersten eigenen Programme zu schreiben.

2.1 Konzept

Für die Programmierung mit GTK ist ein Verständnis des Konzepts wesentlich, das sich erheblich von der linearen Programmierung unterscheidet, die in den restlichen Kapiteln des Buches verwendet wird.

Ein GTK-Programm initialisiert zunächst einmal sich und die GTK-Bibliothek und springt dann in die GTK-Funktion `gtk_main()`, aus der es nicht mehr zurückkehrt. Die Funktion `gtk_main()` selbst schläft so lange, bis der Benutzer eine Aktion mit einem GTK-Element des Programmes durchführt. In diesem Fall wird GTK aktiv und ruft eine Funktion des Programms auf, die der Programmierer zur Verfügung gestellt hat. Diese Funktion führt die benötigten Aktionen aus und gibt die Kontrolle an `gtk_main()` zurück.

Da die vom Programmierer erstellten Funktionen nur im Falle eines Ereignisses aufgerufen werden, nennt sich diese Programmierung ereignisgesteuerte Programmierung oder auf gut Neudeutsch »event driven«.

Einer wichtigen Frage sind wir bisher aber ausgewichen: Woher weiß GTK, welche Funktion bei welchem Ereignis aufgerufen werden muss? Wenn ein Benutzer mit den verfügbaren GTK-Elementen (Widgets) arbeitet, so generiert er Signale (nicht zu verwechseln mit den Unix-Signalen wie `SIGKILL`). Für jedes dieser Signale können Sie als Programmierer eine Funktion definieren, die sich um eine Reaktion auf dieses Signal kümmert. Signale, die Sie nicht von einem so genannten Signalhandler abfangen lassen, werden von GTK bearbeitet (was auch ignorieren bedeuten kann).

Das heißt, dass Sie für jedes GTK-Element Ihres Programms eine Reihe von Signalhandlern einrichten, die auf ein bestimmtes Ereignis reagieren sollen. Diese Funktionen werden Rückruffunktionen (Callbacks) genannt. Dabei ist es durchaus erlaubt, mehr als einen Signalhandler für ein Ereignis für ein Widget zu registrieren. In diesem Fall werden die registrierten Handler in der Reihenfolge der Registrierung aufgerufen.

Neben den Signalen, die GTK selbst generiert (Knopfdruck, Checkbox (de-)aktivieren usw.) gibt es auch noch Signale, die auf Grund von Ereignissen im X-Window-System generiert werden. Dazu gehören z.B. die Ereignisse aus Tabelle 2.1.

`delete_event`	Wird vom Windowmanager an unser Programm gesendet. Wenn unsere Callbackfunktion `FALSE` zurückgibt, wird der `destroy_event` ausgeführt. `TRUE` lässt das Programm dieses Signal ignorieren.
`destroy_event`	Widget wird gelöscht (z.B. weil ein Fenster geschlossen wird).
`button_press_event`	Ein Mausknopf wurde im Widget gedrückt.
`key_press_event`	Tastendruck
`key_release_event`	Taste losgelassen
`enter_notify_event`	Das Widget hat den Fokus erhalten, weil die Maus darüber steht (kein Knopfdruck!!).
`leave_notify_event`	Das Widget hat den Fokus verloren. Das Gegenstück zu `enter_notify_event`.

Tabelle 2.1: GTK-Ereignisse

Es gibt zwei Methoden, einen Signalhandler zu registrieren:

```
gint gtk_signal_connect( GtkObject     *object,
                         gchar         *name,
                         GtkSignalFunc  func,
                         gpointer       func_data );
```

Diese Funktion erwartet einen Zeiger auf ein `GtkObject`. Der Name des Ereignisses, auf das reagiert werden soll, stellt den zweiten Parameter dar (siehe Tabelle 2.1). Der dritte Parameter ist ein Zeiger auf die Callbackfunktion, und der letzte Parameter ist ein Zeiger auf Daten, die der Callbackfunktion übergeben werden sollen (oder `NULL`).

Die entsprechende Callbackfunktion sieht wie folgt aus:

```
void callback_func( GtkWidget *widget,
                    gpointer   callback_data );
```

Dabei ist `widget` ein Zeiger auf das Widget, welches das Signal ausgelöst hat, und `callback_data` ist der Zeiger `func_data` von `gtk_signal_connect()`. In der Regel reicht eine Funktion mit dieser Anzahl an Parametern, allerdings gibt es auch Ausnahmen, wie z.B. das Signal `select_row` des CList-Widgets, welches Zeile und Spalte an die Callbackfunktion übergibt.

Die zweite Methode, eine Callbackfunktion für ein bestimmtes Signal zu registrieren, läuft über folgende GTK-Funktion:

```
gint gtk_signal_connect_object( GtkObject     *object,
                                gchar         *name,
                                GtkSignalFunc  func,
                                GtkObject     *slot_object );
```

Der einzige Unterschied zwischen `gtk_signal_connect_object()` und `gtk_signal_connect()` liegt darin, dass `gtk_signal_connect_object()` genau einen Zeiger vom Typ `GtkObject` an die Callbackfunktion übergibt, welche daher folgendes Aussehen haben sollte:

```
void callback_func( GtkObject *object );
```

2.2 Das erste Beispielprogramm

Nach der ganzen Theorie schauen wir uns einmal ein einfaches Beispielprogramm an, welches nur ein Fenster mit einem Knopf öffnet. Ein Klick auf den Knopf schließt das Fenster und beendet das Programm:

```
/* gtktest.c
 *
 * Der Versuch, ein GTK-Programm zu schreiben
 *
 * M.Rathmann und C.Wieskotten
 */

#include <gtk/gtk.h>

void GK_Hello(GtkWidget       *p_widget, gpointer data)
```

```
{
    // Diese Funktion gibt auf stdout "Hallo Welt" aus,
    // wenn der Knopf gedrückt wird
    g_print("Hallo Welt !!!\n");
}

gint GK_DelEvent(GtkWidget *p_widget,
                 GdkEvent  *p_event,
                 gpointer   data)
{
    // Wenn der Windowmanager das Fenster schließen will,
    // schickt er ein Delete-Event an unser Programm.
    // Wenn wir TRUE zurückgeben, so bleibt das Fenster
    // offen und unser Programm aktiv. Ein FALSE führt zu
    // einem Destroy-Ereignis.
       g_print("Löschereignis\n");
       return(TRUE);
}

void GK_Destroy(GtkWidget *p_widget, gpointer data)
{
     // Das Fenster wird geschlossen, jetzt ist aufräumen
     // angesagt, und gtk_main_quit() beendet gtk_main()
     g_print("Ende und Vorbei\n");
     gtk_main_quit();
}

 int main(int argc,char *argv[])
 {
    GtkWidget    *l_window;         // Zeiger auf das Fenster
    GtkWidget    *l_button;         // Zeiger auf den Knopf


    gtk_init(&argc,&argv);
    l_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_signal_connect(GTK_OBJECT(l_window), "delete_event",
                       GTK_SIGNAL_FUNC(GK_DelEvent),NULL);
    gtk_signal_connect(GTK_OBJECT(l_window), "destroy",
                       GTK_SIGNAL_FUNC(GK_Destroy),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(l_window),20);
    l_button = gtk_button_new_with_label("Hallo Welt");
    gtk_signal_connect(GTK_OBJECT(l_button),"clicked",
                       GTK_SIGNAL_FUNC(GK_Hello),NULL);
    gtk_signal_connect_object(GTK_OBJECT(l_button), "clicked",
                              GTK_SIGNAL_FUNC(gtk_widget_destroy),
                              GTK_OBJECT(l_window));
    gtk_container_add(GTK_CONTAINER(l_window),l_button);
```

```
    gtk_widget_show(l_button);
    gtk_widget_show(l_window);
    gtk_main();
    return(0);
}
```

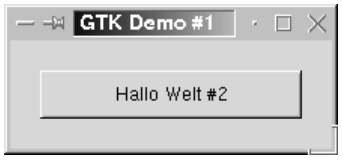

Abbildung 2.1: Erstes Beispielprogramm

Schauen wir uns Schritt für Schritt an, was dieses Programm macht. Die Kommentare wurden aus Platzgründen entfernt:

```
void GK_Hello(GtkWidget    *p_widget, gpointer data)
{
    g_print("Hallo Welt !!!\n");
}
```

Diese Funktion ist der Callback für das Drücken des Knopfes. In dieser einfachen Version gibt der Signalhandler nur den Text `Hallo Welt !!!` aus, wenn der Knopf gedrückt wird. Er bedient sich dabei der Funktion `g_print()` aus der GLib. Sie akzeptiert die gleichen Parameter und Formate wie `printf()`, `sprintf()` etc. aus der Standard-C-Blibliothek.

```
gint GK_DelEvent(GtkWidget *p_widget,
                 GdkEvent  *p_event,
                 gpointer   data)
{
    // Wenn der Windowmanager ...
    g_print("Löschereignis\n");
    return(TRUE);
}
```

Dies ist der Signalhandler für das `delete_event`. Dieser kommt z.B. vom Windowmanager, wenn der Benutzer im Fensterrahmen auf das Icon zum Fensterschließen klickt. Signalhandler für dieses Signal haben eine Besonderheit: Sie geben entweder `TRUE` oder `FALSE` zurück. Gibt die Funktion `FALSE` zurück, so wird das Fenster geschlossen, das Signal `destroy_event` ausgelöst und somit der Signalhandler für das Signal `destroy_event` aktiviert. Gibt sie jedoch `TRUE` zurück, so wird das Signal `destroy_event` nicht ausgelöst. Die Funktionalität ist sinnvoll, wenn Sie sich das Schließen eines Fensters mit einer Sicherheitsabfrage bestätigen lassen wollen.

```
void GK_Destroy(GtkWidget *p_widget, gpointer data)
{
       g_print("Ende und Vorbei\n");
       gtk_main_quit();
}
```

Wie bereits oben angedeutet, wird dieser Signalhandler aufgerufen, wenn unser Fenster geschlossen wird. Im Beispiel kann dies nur geschehen, wenn der Knopf im Fenster gedrückt wird, da `GK_DelEvent()` immer `TRUE` zurückgibt, was GTK davon abhält, das Fenster zu löschen. `gtk_main_quit()` fordert GTK auf, die Schleife von `gtk_main()` zu beenden.

```
int main(int argc,char *argv[])
 {
    GtkWidget   *l_window;      // Zeiger auf das Fenster
    GtkWidget   *l_button;      // Zeiger auf den Knopf
```

Hier startet das Hauptprogramm und definiert zwei Zeiger auf Widgets: einen für das Fenster und einen für den Knopf im Fenster.

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

Dies ist die Initialisierungsfunktion von GTK und den dazugehörigen Komponenten wie z. B. GLib. Übergeben werden die Zeiger auf `argv` und `argc`. Dieser Aufruf ist wichtig, da sonst nichts funktioniert, was GTK betrifft.

```
    l_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

Diese Zeile legt die Strukturen für ein Fenster an. Fenster sind in GTK auch nichts anderes als Widgets. Das Fenster ist zu diesem Zeitpunkt noch nicht sichtbar!

```
    gtk_signal_connect(GTK_OBJECT(l_window), "delete_event",
                       GTK_SIGNAL_FUNC(GK_DelEvent),NULL);
    gtk_signal_connect(GTK_OBJECT(l_window), "destroy",
                       GTK_SIGNAL_FUNC(GK_Destroy),NULL);
```

Mit diesen beiden Funktionen werden die Signalhandler für die Ereignisse `delete_event` und `destroy` GTK bekannt gemacht. `GTK_OBJECT()` und `GTK_SIGNAL_FUNC()` sind Makros, die einige Überprüfungen und benötigte Typumwandlungen vornehmen.

```
    gtk_container_set_border_width(GTK_CONTAINER(l_window),20);
```

Auch `GTK_CONTAINER()` ist ein Makro für Überprüfungen und Typenumwandlung. Die Funktion `gtk_container_set_border_width()` selbst setzt den Abstand zwischen Fensterrahmen und Widgets, die im Fenster eingesetzt werden, in Pixeln.

```
    l_button = gtk_button_new_with_label("Hallo Welt #2");
```

Jetzt erstellen wir die Struktur für den Knopf und geben ihm die Aufschrift `Hallo Welt #2`. Auch der Knopf ist wie das Fenster (noch) nicht zu sehen.

```
gtk_signal_connect(GTK_OBJECT(l_button),"clicked",
                   GTK_SIGNAL_FUNC(GK_Hello),NULL);
```

Damit wird der Signalhandler für das Klicken des Knopfes eingerichtet. Da unsere Callbackfunktion übergebene Daten ignoriert, setzen wir die Übergabedaten an dieser Stelle auf `NULL`, d.h., das Programm übergibt keine Daten an die Callbackfunktion.

```
gtk_signal_connect_object(GTK_OBJECT(l_button), "clicked",
                          GTK_SIGNAL_FUNC(gtk_widget_destroy),
                          GTK_OBJECT(l_window));
```

Außerdem nutzen wir diesen Knopf dazu, unser Programm zu beenden. Aus diesem Grund setzt unser Programm selbst das `destroy`-Signal ab (was bedeutet, dass dieses Signal nicht zwingend vom Windowmanager kommen muss). Da dieser Signalhandler als zweiter für `clicked` gesetzt wird, wird er auch erst als zweites (also nach `GK_Hello()`) aufgerufen. Das Programm ruft dabei den GTK-Callback `gtk_widget_destroy()` auf, der einen Parameter vom Typ `GtkWidget` erwartet. Deshalb nutzen wir `gtk_signal_connect_object()`, dessen vierter Parameter vom Typ `GtkObject` ist, während `gtk_signal_connect()` einen Parameter vom Typ `gpointer` erwartet.

```
gtk_container_add(GTK_CONTAINER(l_window),l_button);
```

Diese Funktion ordnet dem Fenster `l_window` den Knopf `l_button` zu. Damit wird dieser im Fenster auftauchen. Angezeigt wird bis jetzt immer noch nichts!

Ein Fenster kann auf diese Weise nur ein Widget aufnehmen, weshalb es Widgets gibt, die in der Lage sind, mehrere Widgets aufzunehmen und anzuzeigen. Wie das funktioniert wird etwas später in diesem Kapitel beschrieben.

```
gtk_widget_show(l_button);
gtk_widget_show(l_window);
```

Der erste Aufruf zeigt zunächst den Knopf an. Da dieser aber im noch nicht sichtbaren Fenster angezeigt werden soll, passiert zunächst immer noch nichts. Erst der zweite Aufruf mit dem Zeiger auf die Fensterstruktur führt dazu, dass sowohl Fenster als auch Knopf angezeigt werden. Die umgekehrte Reihenfolge würde zwar auch funktionieren, hätte allerdings den unschönen Nebeneffekt, dass der Benutzer sehen kann, wie erst das Fenster aufgebaut und danach darin der Knopf gezeichnet wird. Dieser Effekt macht sich aber erst bei vielen zu zeichnenden Widgets bemerkbar, so dass für dieses Beispiel die Reihenfolge wohl egal wäre.

```
gtk_main();
return(0);
```

Dieser Aufruf ist die GTK-Schleife, aus der unsere Callbacks aufgerufen werden. Diese Funktion kehrt erst dann zurück, wenn `gtk_main_quit()` aufgerufen wird, worauf das Programm `0` als Ergebnis zurückgibt.

2.3 Hilfsfunktionen

Aufruf:

```
GSList* g_slist_append(GSList      *list,
                       gpointer     data);
GSList* g_slist_prepend(GSList      *list,
                        gpointer     data);
GSList* g_slist_insert(GSList      *list,
                       gpointer     data,
                       gint         position);
GSList* g_slist_remove(GSList      *list,
                       gpointer     data);
GSList* g_slist_nth(GSList      *list,
                    guint        n);
GSList* g_slist_last(GSList      *list);
void    g_slist_free(GSList      *list);
guint   g_slist_length(GSList      *list);
```

Neben den Funktionen zur Verwaltung der grafischen Elemente gibt es auch einige Funktionen, die einige häufig benötigte Funktionalitäten zur Verfügung stellen. Dazu gehört die bereits besprochene Funktion `g_print()`. Sehr angenehm ist die Möglichkeit, mit Hilfe der GLib eine allgemeine Listenverwaltung zu implementieren. Einfach und doppelt verkettete Listen werden praktisch in jedem Programm eingesetzt, das mit größeren Datenmengen arbeitet.

Eine einfach verkettete Liste ist eine Struktur vom Typ `GSList`, die genau zwei Felder enthält. Das Feld `next` ist dabei ein Zeiger vom Typ `GSList`, welches auf den nächsten Eintrag in der Liste zeigt (oder `NULL`). `data` zeigt auf eine Struktur beliebigen Typs und somit auf die Daten, die in der Liste gespeichert werden.

Wenn Sie mit Listen arbeiten, müssen Sie die Liste mit `NULL` initialisieren, da ansonsten Probleme auftreten können. Mit der Funktion `g_slist_append()` können Sie dabei einen Eintrag `data` an das Ende der Liste `list` anhängen. `g_slist_prepend()` dagegen hängt `data` an den Listenanfang ein.

Wenn `data` an eine bestimmte Position eingehängt werden soll, so ruft man `g_slist_insert()` auf. Dabei muss sich `position` im Bereich von `0` bis `g_slist_length()-1` befinden, gibt doch `g_slist_length()` die Anzahl der Listeneinträge zurück.

Der Eintrag `data` kann aus der Liste `list` entfernt werden, indem `g_slist_remove()` aufgerufen wird.

`g_slist_nth()` gibt den `n`-ten Eintrag aus der Liste `list` zurück, während `g_slist_last()` den letzten Eintrag aus der Liste `list` zurückgibt.

Die Liste `list` kann mittels `g_slist_free()` freigegeben werden. Allerdings werden die Daten, auf die `data` zeigt, nicht automatisch freigegeben, weshalb diese vom Programm gelöscht werden müssen.

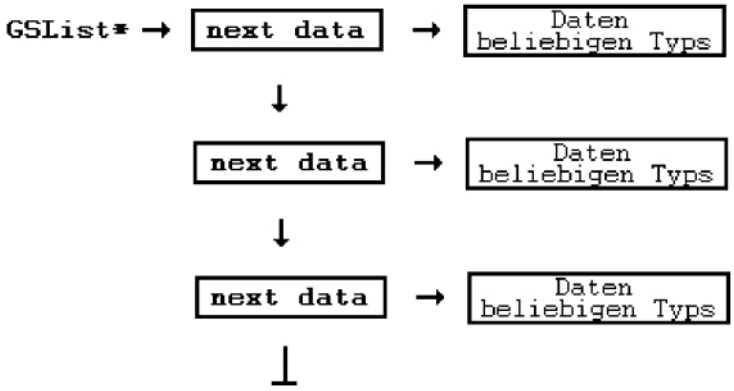

Abbildung 2.2: Einfach verkettete Liste

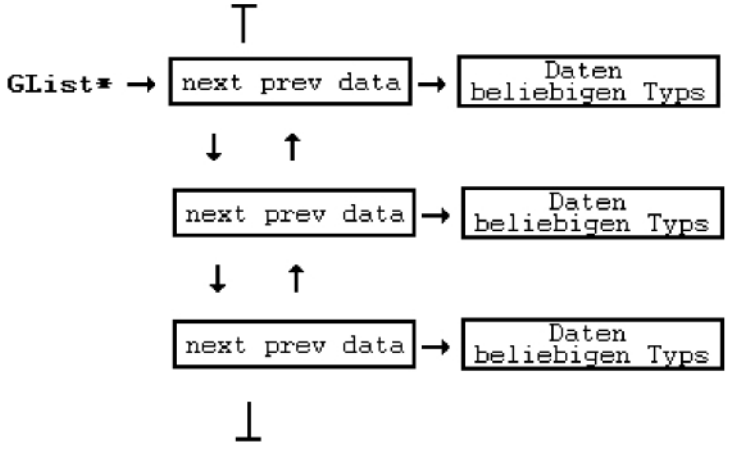

Abbildung 2.3: Doppelt verkettete Liste

Diese Funktionen gibt es auch für doppelt verkettete Listen. Der Unterschied liegt darin, dass diese Funktionen mit dem Typ `GList` arbeiten (an Stelle von `GSList`) und die Funktionsnamen auf das `s` verzichten: `g_list_append()`, `g_list_length()` usw. Der Typ `GList` hat zusätzlich zum Typ `GSList` noch einen Zeiger auf den vorherigen Eintrag in der Liste. Das entsprechende Feld in der Struktur trägt den Namen `prev`.

Dies sind zwar nur einige der Hilfsfunktionen aus der GLib, aber alle, die in den Beispielprogrammen dieses Buchs eingesetzt werden. Für weitere Informationen schauen Sie bitte in der mit Glib mitgelieferten Dokumentation oder in `<glib.h>` nach.

2.4 Widgets

Die letzten Abschnitte haben sich mit dem Konzept und einem ersten Beispielprogramm beschäftigt. In diesem Abschnitt werden einige der Widgets, die GTK anbietet, vorgestellt und eingesetzt. Die Auswahl der Widgets ist rein willkürlich und alles andere als vollständig, sollte aber ausreichend sein, um erste eigene Programme zu erstellen.

2.4.1 Packing Widgets

Aufruf:

```
GtkWidget *gtk_vbox_new(gboolean gleich,
                        gint     abstand);

GtkWidget *gtk_hbox_new(gboolean gleich,
                        gint     abstand);

void  gtk_box_pack_start(GtkBox     *box,
                         GtkWidget  *child,
                         gboolean    expand,
                         gboolean    fill,
                         guint       padding);

void  gtk_box_pack_end(GtkBox       *box,
                       GtkWidget    *child,
                       gboolean      expand,
                       gboolean      fill,
                       guint         padding);
```

Ein Container ist ein Widget, welches andere Widgets enthalten und darstellen kann. Die offensichtlichste Art eines Containers ist ein Fenster. Im Demoprogramm des letzten Abschnittes wurde dies genutzt, um einen Knopf mit der Aufschrift `Hallo Welt #2` darzustellen. Die Aufschrift selbst ist ein Label-Widget, das den Knopf zu einem Container macht.

Was die Anzahl der Widgets in einem Container betrifft, erlegt GTK Beschränkungen auf: Jeder Container darf (im Allgemeinen) nur ein Widget enthalten. Aus diesem Grunde könnte ein zweiter Knopf im obigen Programm nicht angezeigt werden. Da es aber keine Programme gibt, die in ihren Fenstern nur genau ein

Widget darstellen wollen, gibt es spezielle Container, in die mehrere Widgets gepackt werden können. Diese Widgets werden deshalb auch »Packing Widgets« oder auch »Packing Boxes« genannt.

Diese Widgets gibt es in drei Varianten:

- Vertikale Packing Boxes ordnen die in ihnen enthaltenen Widgets vertikal an.
- Horizontale Packing Boxes ordnen die in ihnen enthaltenen Widgets horizontal an.
- Tabellarische Packing Boxes erlauben die Anordnung der Widgets auf X- und Y-Achse. Diese werden im nächsten Abschnitt beschrieben.

Mit den Funktionen `gtk_vbox_new()` bzw. `gtk_hbox_new()` werden die vertikal bzw. horizontal anordnenden »Packing Boxes« angelegt. Der Parameter `gleich` legt dabei fest, ob die Widgets alle die gleiche Größe haben (`TRUE`) oder nicht (`FALSE`). Werden in einem Packwidget drei Knöpfe mit den Aufschrift »Döge«, »Wieskotten« und »Koala« angelegt, so sind alle drei Knöpfe gleich groß, wobei die Größe sich an der längsten Aufschrift orientiert. Wird `gleich` auf `FALSE` gesetzt, so nimmt jeder Knopf genau so viel Platz ein, wie er benötigt. `abstand` legt den Abstand der Widgets im Packwidget (entweder horizontal oder vertikal) in Pixeln fest.

Die Funktion `gtk_box_pack_start()` packt das übergebene Widget `child` in das Packwidget `box` und fügt dieses links (bei einem horizontalem Packwidget) bzw. oben (bei einem vertikalen Packwidget) ein. `expand` legt fest, ob das eingefügte Widget evtl. verfügbaren Platz um das Widget herum noch nutzen darf, nachdem alle Widgets dem Packingwidget hinzugefügt wurden (`TRUE`) oder nicht (`FALSE`). `expand` wird ignoriert, wenn die Packbox mit dem Parameter `gleich` gleich `TRUE` angelegt wurde.

Der Parameter `fill` bestimmt, ob das Widget seinen festgelegten Bereich komplett ausfüllen kann (`TRUE`) oder nicht (`FALSE`). Der Parameter `padding` bestimmt den Abstand in Pixeln vom Widget zu den Nachbarn. Dieser Parameter ist meistens `0`.

Die Funktion `gtk_box_pack_end()` nimmt die gleichen Parameter entgegen, fügt das Widget aber entweder rechts neben oder unter dem letzten eingefügten Widget ein.

Hinweis

Jedes Widget besteht zunächst aus einem Rechteck, in dem es dargestellt werden kann. Dieses wird Packing Box genannt. Das Widget, z.B. ein Knopf, wird in diesem Rahmen gezeichnet, ohne dass der Rahmen komplett vom Knopf ausgefüllt wird. `fill` *bezieht sich auf das Ausnutzen dieses Rahmens, während* `expand` *sich auf die Größe des Rahmens bezieht.*

2.4.2 Tabellarische Packing Widgets

Aufruf:

```
GtkWidget *gtk_table_new(guint        rows,
                         guint        columns,
                         gboolean     gleich);

void      gtk_table_attach_defaults(GtkTable       *table,
                                    GtkWidget      *widget,
                                    guint           left_attach,
                                    guint           right_attach,
                                    guint           top_attach,
                                    guint           bottom_attach);
```

Dieses Packing Widget funktioniert ähnlich wie die bereits besprochenen vertikalen und horizontalen Packwidgets. Im Gegensatz zu diesen bieten tabellarische Packwidgets (ab hier kurz als Tabelle bezeichnet) eine Kontrolle über die Positionierung der Widgets auf der X- und Y-Achse. Ein tabellarisches Packwidget wird durch `gtk_table_new()` angelegt. Es verlangt die Anzahl der Tabellenzeilen `rows` und Tabellenspalten `columns` als Parameter. `gleich` legt fest, ob die Spalten alle die gleiche Breite haben (`TRUE`) oder nicht (`FALSE`).

`gtk_table_attach_defaults()` setzt das `widget` in die Tabelle `table` ein. Die Position des Widgets fängt in der Spalte `left_attach` an und endet in der Spalte `right_attach`. Die vertikale Position des Widgets beginnt in der Zeile `top_attach` und endet in der Zeile `bottom_attach`. Zeilen und Spalten beziehen sich dabei auf die Tabelle und nicht auf Pixel. Die erste Spalte/Zeile hat dabei die Nummer `0`.

2.5 Listboxen

Aufruf:

```
GtkWidget *gtk_list_new(void);

void       gtk_list_set_selection_mode(GtkList          *list,
                                       GtkSelectionMode  mode);
GtkWidget *gtk_list_new_with_label(char *label);

void       gtk_list_clear_items(GtkList      *list,
                                gint          start,
                                gint          end);
void       gtk_list_select_item(GtkList     *list,
                                gint          item);
void       gtk_list_unselect_item(GtkList   *list,
                                  gint        item);
void       gtk_list_select_child(GtkList    *list,
                                 GtkWidget   *child);
```

```
void       gtk_list_unselect_child(GtkList   *list,
                                   GtkWidget *child);
GtkWidget* gtk_list_item_new_with_label(const gchar  *label);

void    gtk_container_add(GtkContainer    *container,
                          GtkWidget       *widget);
```

Das Anlegen einer Listbox geschieht durch den Aufruf von `gtk_list_new()` oder `gtk_list_new_with_label()`. Das so erhaltene Widget kann dann wie gewohnt in ein Packing Widget eingesetzt werden.

Eine Listbox hat zwei verschiedene Auswahlmodi: Zum einen gestattet sie die Auswahl genau eines Eintrags aus der Liste der verfügbaren Elemente. Der zweite Modus erlaubt die Auswahl mehrerer Listeneinträge aus der Liste. In GTK wird dieser Auswahlmodus durch den Aufruf von `gtk_list_set_selection_mode()` eingestellt. Dabei ist `list` der Zeiger auf ein Widget vom Typ `GtkList`. `mode` ist entweder `GTK_SELECTION_SINGLE` (genau ein Eintrag auswählbar) oder `GTK_SELECTION_MULTIPLE` (mehrere Einträge auswählbar).

Die Listbox selbst stellt nichts anderes dar als einen Container für die darin angezeigten Einträge. Ein Eintrag ist ein Widget mit einem Text und wird mit der Funktion `gtk_list_item_new_with_label()` angelegt. Mittels `gtk_container_add()` wird das Widget der Listbox hinzugefügt.

Die Tatsache, dass Widgets Signale erzeugen können, kann im Zusammenhang mit der Listbox geschickt ausgenutzt werden. Wenn ein Eintrag in der Liste selektiert wird, so wird ein Signal `select` generiert, wird der Eintrag wieder deselektiert, so wird das Signal `deselect` generiert. Es muss also nur für jedes Signal eine Callbackfunktion eingerichtet werden, so wie das weiter oben schon beschrieben wurde.

```
void GK_ItemSelect(GtkWidget *p_item,gpointer *p_data)
{
  g_print("Ergebnis (%s)\n",(char *)p_data);
}
void GK_ItemDeselect(GtkWidget *p_item,gpointer *p_data)
{
  g_print("Weg mit (%s)\n",(char *)p_data);
}

void GK_AddItem(GtkWidget *p_lbox, char *p_str)
{
  /* Diese Funktion fügt genau einen Eintrag in Listbox
     p_lbox hinzu. Der Text des Eintrages steht in p_str
  */
  GtkWidget *l_item;

  // Widget mit Text erstellen
```

```
  l_item = gtk_list_item_new_with_label(p_str);
  // Der Listbox hinzufügen
  gtk_container_add(GTK_CONTAINER(p_lbox),l_item);
  // Die Signale abfangen
  gtk_signal_connect(GTK_OBJECT(l_item),"sel-
ect",GTK_SIGNAL_FUNC(GK_ItemSelect),
                     p_str);
  gtk_signal_connect(GTK_OBJECT(l_item),"deselect",
                     GTK_SIGNAL_FUNC(GK_ItemDeselect),
                     p_str);
  gtk_widget_show(l_item); }
```

Die Funktionen `gtk_list_select_item()` bzw. `gtk_list_select_child()` erlauben das Selektieren von Einträgen in einer Listbox. Der Unterschied zwischen diesen beiden besteht darin, dass Ersteres die Nummer des Eintrags erwartet, während die zweite Funktion einen Zeiger auf das zu selektierende Widget benötigt.

Entsprechend deselektieren die Funktionen `gtk_list_unselect_item()` bzw. `gtk_list_unselect_child()` einen Listboxeintrag.

2.6 Scrolling Windows

Aufruf:

```
GtkWidget*     gtk_scrolled_window_new(GtkAdjustment     *hadjustment,
                                       GtkAdjustment     *vadjustment);
void           gtk_scrolled_window_set_policy
                                       (GtkScrolledWindow
*scrolled_window,
                                        GtkPolicyType
hscrollbar_policy,
                                        GtkPolicyType
vscrollbar_policy);
void           gtk_scrolled_window_add_with_viewport
                                       (GtkScrolledWindow
*scrolled_window,
                                        GtkWidget         *child);
```

Die Listbox aus dem letzten Kapitelabschnitt ist zwar funktionsfähig, hat allerdings den Nachteil, dass ihre Höhe abhängig von der Anzahl der Listeneinträge ist: Enthält eine Listbox keinen Eintrag, ist sie 0 Zeilen hoch, bei vier Einträgen vier Zeilen hoch usw. Dies ist natürlich nicht unbedingt gewünscht. Besser wäre es, immer eine feste Zeilenanzahl zu benutzen und bei Bedarf einen Rollbalken `vscrollbar_policy` am rechten Rand zu platzieren.

Mit den Mitteln der Listbox ist dies nicht zu erreichen. Dazu muss ein »Scrolled Window« angelegt werden. Dies ist ein Container, der ein Widget aufnimmt und, wenn dieses größer als das `scrolled Window` wird, horizontale und vertikale Rollbalken zur Verfügung stellt.

Ein solches Rollfenster wird mit der Funktion `gtk_scrolled_window_new()` angelegt. Die Parameter `hadjustment` und `vadjustment` stellen Zeiger auf Adjustments dar. Dies sind z.B. Rollbalken, mit denen der Anzeigebereich des Rollfensters angepasst (engl. adjust) werden kann. Werden diese Parameter auf `NULL` gesetzt, so legt `gtk_scrolled_window_new()` diese Adjustments selbst an.

Ob die Rollbalken des Scrolled Window immer angezeigt werden oder nur bei Bedarf, kann ein Programm durch einen Aufruf von `gtk_scrolled_window_set_policy()` bestimmen. Dabei werden der Zeiger auf das Fenster und die Einstellungen `hscrollbar_policy` und `vscrollbar_policy` für den horizontalen und den vertikalen Rollbalken festgelegt. Für beide Einstellungen gelten folgende Werte:

- `GTK_POLICY_ALWAYS`: Der Rollbalken wird immer angezeigt, egal ob er benötigt wird oder nicht.
- `GTK_POLICY_AUTOMATIC`: Der Rollbalken wird nur bei Bedarf angezeigt.

Im letzten Schritt muss der Inhalt in das Rollfenster eingefügt werden. Dazu dient die Funktion `gtk_scrolled_window_add_with_viewport()`. Übergeben wird der Zeiger auf das Rollfenster `scrolled_window` und der Zeiger `child` auf das darin darzustellende Widget.

Soll eine fertige Listbox `l_lbox` in das Rollfenster eingebaut werden und immer ein vertikaler Rollbalken verfügbar sein, sieht das so aus:

```
GtkWidget    *l_swin;        // Scroll Window

...
l_swin = gtk_scrolled_window_new(NULL,NULL);
gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(l_swin),
                               GTK_POLICY_AUTOMATIC,GTK_POLICY_ALWAYS);
gtk_container_set_border_width(GTK_CONTAINER(l_swin),10);
gtk_scrolled_window_add_with_viewport(GTK_SCROLLED_WINDOW(l_swin),l_lbox
);
gtk_widget_show(l_swin);
...
```

Natürlich können Sie beliebige Widgets in ein Scrolled Window einsetzen (z.B. Text-Widgets, Knöpfe, Labels etc), und nicht nur Listboxen.

2.7 Label

Aufruf:

```
GtkWidget* gtk_label_new(const char            *str);
void       gtk_label_set_text(GtkLabel         *label,
                              const char       *str);
void       gtk_label_get(GtkLabel              *label,
                         char                 **str);
void       gtk_label_set_line_wrap(GtkLabel    *label,
                                   gboolean     wrap);
```

Ein Label ist ein Bereich in einem Widget, welches mit beliebigem statischen Text gefüllt wird. Anders als in anderen Widgets wird für ein Label kein eigenes (Sub-)Fenster angelegt. Labels benötigen zwingend ein Parent-Widget, in dem sie direkt ausgegeben werden. So ist z. B. der Text auf einem Knopf ein Label. Labels leiten sich von der Struktur `GtkMisc` ab – wie `GtkMisc` können auch sie selbst keine Ereignisse empfangen bzw. generieren.

Mit `gtk_label_new()` wird ein Label angelegt. Übergeben wird der Text, den das Label anzeigen soll. Nachträglich kann dieser Text mittels `gtk_label_set_text()` geändert werden. Dabei wird der Zeiger auf das betroffene `label` und der neue Text `str` übergeben.

Wenn Sie den Text in einem Label ermitteln wollen, so benötigen Sie einen Zeiger vom Typ `char`, dessen Adresse der Funktion `gtk_label_get()` als zweiter Parameter übergeben wird. Der erste Parameter ist vom Typ `GtkLabel`, ein Label aber vom Typ `GtkWidget`, weshalb dieses mit der Konvertierungsfunktion `GTK_LABEL()` angepasst werden muss.

Bleibt noch die Funktion `gtk_label_set_line_wrap()`, die zunächst einen Zeiger auf eine `GtkLabel`-Struktur erwartet (`label`). Ist der zweite Parameter `wrap` gleich `TRUE`, so wird der Text am Zeilenende umbrochen, während ein `FALSE` dies verhindert. Falls der Text länger als der verfügbare Platz in der Zeile ist, wird der Text abgeschnitten.

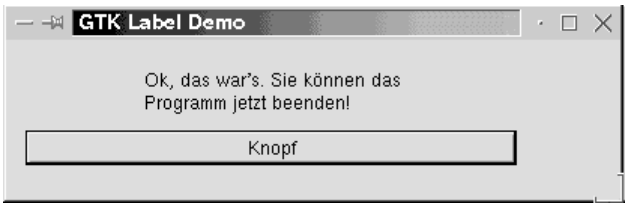

Abbildung 2.4: Beispiel: Label

Das folgende Beispiel erstellt ein Fenster, in dem ein Label und ein Knopf angelegt werden. Wenn der Knopf gedrückt wird, gibt das Programm den Text des Labels auf die Standardausgabe aus und ändert diesen ab. Das Programm kann über das Schließen-Icon im Bildrahmen beendet werden.

```
/* gtklabel.h
 *
 * Das Headerfile für das zweite Demoprogramm
 * unter GTK.
 *
 * M.Rathmann und C.Wieskotten
 */
#ifndef _C_MyGtkShow_
#define _C_MyGtkShow_

#define C_GK_MaxSLen       200         /* Maximale Länge von Strings  */
#define C_GK_MaxALen       10          /* Maximale Einträge in Arrays */

#endif


/* Gtklabel.c
 *
 * Demoprogramm für GtkLabel
 *
 */


#include <gtk/gtk.h>
#include <sys/types.h>
#include <pwd.h>
#include <string.h>
#include <stdlib.h>

#include "gtklabel.h"

GtkWidget *V_GK_Label;         // Variable für das Label
int        V_GK_Anz = 0;       // Zähler für die Anzahl der Knopfdrücke


void GK_Label(GtkWidget *p_lab,gpointer data)
{
    char *l_ch;
    char  l_buff[300];

    // Text im Label ermitteln und ausgeben
    gtk_label_get(GTK_LABEL(data),&l_ch);
```

```
    printf("Ergebnis (%s)\n",l_ch);
    // Neuen Text festlegen
    switch(V_GK_Anz)
      {
       case 0: strcpy(l_buff,"Ok, das war's. Sie können das Programm
jetzt beenden!");
           break;
       case 1: strcpy(l_buff,"Ok, das war's. Das Programm macht nichts
mehr!");
           break;
       case 2: strcpy(l_buff,"Klicken Sie bitte nicht noch mal");
           break;
       case 3: strcpy(l_buff,"Letzte Warnung");
           break;
       case 4: strcpy(l_buff,"Letzte Warnung!!!");
           break;
       case 5: gtk_main_quit();      // Max. 5 Versuche, dann ist Ende
           break;
      }
    V_GK_Anz++;
    // .. und setzen
    gtk_label_set_text(GTK_LABEL(data),l_buff);
    gtk_label_set_line_wrap(GTK_LABEL(data),TRUE);

}

gint GK_DelEvent(GtkWidget *p_widget,
                 GdkEvent  *p_event,
                 gpointer   data)
{
    // Wenn der Windowmanager das Fenster schließen will,
    // schickt er ein Delete-Event an unser Programm.
    // Wenn wir TRUE zurückgeben, so bleibt das Fenster
    // offen und unser Programm aktiv. Ein FALSE führt
    // zu einem Destroy-Ereignis
    return(FALSE);
}

void GK_Destroy(GtkWidget *p_widget, gpointer data)
{
       // Das Fenster wird geschlossen, jetzt ist Aufräumen
       // angesagt, und gtk_main_quit() beendet gtk_main()
       gtk_main_quit();
}


int main(int argc,char *argv[])
 {
```

```
    GtkWidget   *l_window;          // Zeiger auf das Fenster
    GtkWidget   *l_gbutton;         // Zeiger auf den Knopf Gruppe
    GtkWidget   *l_table;           // Packing Box: Tabelle
    GtkWidget   *l_label;           // Textausgabe



    gtk_init(&argc,&argv);
    l_table  = gtk_table_new(2,4,FALSE);
    l_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    l_label  = gtk_label_new("Wählen Sie einen Benutzer aus, über den
Sie informiert werden möchten:");
    V_GK_Label = l_label;
    gtk_widget_show(l_label);
    gtk_window_set_title(GTK_WINDOW(l_window),"GTK Label Demo");
    gtk_window_set_position(GTK_WINDOW(l_window),GTK_WIN_POS_CENTER);
    gtk_widget_set_usize(l_window,240,100);
    gtk_table_set_row_spacings(GTK_TABLE(l_table),10);
    gtk_table_set_col_spacings(GTK_TABLE(l_table),10);
    gtk_signal_connect(GTK_OBJECT(l_window), "delete_event",
                       GTK_SIGNAL_FUNC(GK_DelEvent),NULL);
    gtk_signal_connect(GTK_OBJECT(l_window), "destroy",
                       GTK_SIGNAL_FUNC(GK_Destroy),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(l_window),20);

    l_gbutton = gtk_button_new_with_label("Text");
    gtk_table_attach_defaults(GTK_TABLE(l_table),l_label,0,3,0,1);
    gtk_table_attach_defaults(GTK_TABLE(l_table),l_gbutton,1,2,1,2);
    gtk_signal_connect(GTK_OBJECT(l_gbutton),"clicked",
                       GTK_SIGNAL_FUNC(GK_Label),V_GK_Label);
    gtk_container_add(GTK_CONTAINER(l_window),l_table);
    gtk_widget_show(l_gbutton);
    gtk_widget_show(l_table);
    gtk_widget_show(l_window);
    gtk_main();
    return(0);
 }
```

Hinweis

Wenn das Programm in der Konsole den Fehler ausgibt, dass das X-Display nicht gefunden wurde, obwohl es definitiv geöffnet ist, so wurde die Umgebungsvariable `DISPLAY` *nicht gesetzt:*

```
prog@koala:/home/prog/source > gtklabel
Gtk-WARNING **: cannot open display:
prog@koala:/home/prog/source > export DISPLAY=:0.0
prog@koala:/home/prog/source >
```

2.8 Radiobuttons

Aufruf:

```
GtkWidget* gtk_radio_button_new(GSList *group);

GtkWidget* gtk_radio_button_new_with_label(GSList *group, const gchar
*label);

GSList* gtk_radio_button_group(GtkRadioButton *radio_button);

void gtk_radio_button_set_group(GtkRadioButton *radio_button, GSList
*group);
```

Ein Radiobutton ist Teil einer zusammengehörenden Gruppe von Knöpfen, aus deren Menge zu einer bestimmten Zeit nur genau ein Button ausgewählt werden kann. Wird ein anderer Knopf selektiert, so wird der zuvor aktivierte Knopf automatisch deaktiviert.

GTK verlangt für jede Radiobuttongruppe das Anlegen einer Liste vom Typ `GSList`, in der GTK dann jeden neuen Knopf der Gruppe automatisch einträgt. Aus diesem Grund ist diese Liste der erste Parameter für die Funktion `gtk_radio_button_new_with_label()`. Diese Liste *muss* beim ersten Aufruf mit `NULL` initialisiert werden. Der zweite Parameter ist der Text neben dem eigentlichen Radiobutton.

```
GSList       *l_grp;
GtkWidget    *l_radio;
...
l_radio = gtk_radio_button_new_with_label(l_grp,"Radio #1");
l_grp = gtk_radio_button_group(GTK_RADIO_BUTTON(l_radio));
...
```

Die erneute Ermittlung der Gruppenliste mittels `gtk_radio_button_group()` ist unbedingt nötig, da sich die Liste nach dem Aufruf `gtk_radio_button_new_with_label()` geändert hat und ein anderer Zeiger auf den Beginn der Liste zeigt.

Die Zuordnung eines Radiobuttons zu einer Gruppe kann nachträglich mit Hilfe der Funktion `gtk_radio_button_set_group()` vorgenommen werden. Der erste Parameter ist ein Zeiger auf die Struktur des betroffenen Radiobuttons und der zweite Parameter die Gruppenliste, in die der Knopf aufgenommen werden soll.

Fehlen noch die Signale bzw. Ereignisse, die von Radiobuttons generiert werden können. Diese sind in Tabelle 2.2 aufgeführt.

Ereignis	Grund
`toggled`	Wenn der Status eines Radiobuttons sich ändert, weil der Knopf angeklickt wurde, so wird dieses Ereignis/Signal generiert. Allerdings wird auch automatisch ein Ereignis für den Radiobutton generiert, dessen Status sich von aktiv auf inaktiv geändert hat, so dass immer zwei `toggled`-Ereignisse ausgelöst werden.
`clicked`	Dieses Signal wird generiert, wenn ein Radiobutton angeklickt wird. Auch hier wird das Deselektieren (Radiobutton wird deaktiviert) als erstes Signal gesendet, das zweite Signal gilt dem Radiobutton, der aktiviert wurde.
`pressed`	Wird gesendet, wenn der Radiobutton gedrückt wurde. Wird nur für den tatsächlich angeklickten Button generiert.
`released`	Wird gesendet, wenn der Radiobutton losgelassen wurde, und wird nur für den tatsächlich geklickten Knopf generiert.

Tabelle 2.2: Ereignisse bei Radiobuttons

Ein kurzes Beispiel:

Abbildung 2.5: Beispiel für einen Radiobutton

```
/* Gtkradio.c
 *
 * Demoprogramm für Radiobuttons
 *
 */


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

```
#include <sys/types.h>
#include <pwd.h>
#include <string.h>
#include <stdlib.h>

#include "gtkradio.h"

GtkWidget *V_GK_Label;          // Variable für das Label
int        V_GK_Anz = 0;        // Zähler für die Anzahl der Knopfdrücke


void GK_Label(GtkWidget *p_lab,gpointer *data)
{

    char *l_ch;
    char  l_buff[300];

    // Text im Label ermitteln und ausgeben
    gtk_label_get(GTK_LABEL(data),&l_ch);
    printf("Ergebnis (%s)\n",l_ch);
    // Neuen Text festlegen
    switch(V_GK_Anz)
      {
       case 0: strcpy(l_buff,"Ok, das war's. Sie können jetzt been-
den!");
           break;
       case 1: strcpy(l_buff,"Ok, das war's. Das Programm macht nichts
mehr!");
           break;
       case 2: strcpy(l_buff,"Klicken Sie bitte nicht noch mal");
           break;
       case 3: strcpy(l_buff,"Letzte Warnung");
           break;
       case 4: strcpy(l_buff,"Letzte Warnung!!!");
           break;
       case 5: gtk_main_quit();          // Max. 5 Versuche, dann ist
Ende
           break;
      }
    V_GK_Anz++;
    // ... und setzen
    gtk_label_set_text(GTK_LABEL(data),l_buff);
    gtk_label_set_line_wrap(GTK_LABEL(data),TRUE);

}

/* Die folgenden Routinen werden für die Radiobuttons aufgerufen
```

```
   wenn
    - ein Knopf (de-)aktiviert wird (GK_Toggle)
    - ein Knopf angeklickt wird     (GK_Click)
    - ein Knopf gedrückt wird       (GK_Press)
*/

void GK_Toggle(GtkWidget *p_wid, gpointer *data)
{
 g_print("Radiobutton (%s) umgestellt\n",(char *)data);
}

void GK_Click(GtkWidget *p_wid, gpointer *data)
{
 g_print("Radiobutton (%s) angeklickt\n",(char *)data);
}

void GK_Press(GtkWidget *p_wid, gpointer *data)
{
 g_print("Radiobutton (%s) gedrückt\n",(char *)data);
}

gint GK_DelEvent(GtkWidget *p_widget,
                 GdkEvent  *p_event,
                 gpointer   data)
{
 return(FALSE);
}

void GK_Destroy(GtkWidget *p_widget, gpointer data)
{
 // Das Fenster wird geschlossen, jetzt ist Aufräumen
 // angesagt, und gtk_main_quit() beendet gtk_main()
 gtk_main_quit();
}


int main(int argc,char *argv[])
 {
    GtkWidget   *l_window;      // Zeiger auf das Fenster
    GtkWidget   *l_gbutton;     // Zeiger auf den Knopf Gruppe
    GtkWidget   *l_table;       // Packing Box: Tabelle
    GtkWidget   *l_label;       // Textausgabe
    GSList      *l_grp;         // Liste der Gruppen
    GtkWidget   *l_radio;       // Widget für Radiobuttons
    GtkWidget   *l_radio1;      // Widget für Radiobuttons
```

```
gtk_init(&argc,&argv);
l_grp    = NULL;            // Gruppe ist leer
l_table  = gtk_table_new(3,4,FALSE);
l_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
l_label  = gtk_label_new("Wählen Sie ein Element aus:");
V_GK_Label = l_label;
gtk_widget_show(l_label);
gtk_window_set_title(GTK_WINDOW(l_window),"GTK Label Demo");
gtk_window_set_position(GTK_WINDOW(l_window),GTK_WIN_POS_CENTER);
gtk_widget_set_usize(l_window,340,120);
gtk_table_set_row_spacings(GTK_TABLE(l_table),10);
gtk_table_set_col_spacings(GTK_TABLE(l_table),10);
gtk_container_set_border_width(GTK_CONTAINER(l_window),20);
l_gbutton = gtk_button_new_with_label("Knopf");
// Die Radiobuttons anlegen
l_radio = gtk_radio_button_new_with_label(l_grp,"Textausgaben");
l_grp = gtk_radio_button_group(GTK_RADIO_BUTTON(l_radio));
l_radio1 = gtk_radio_button_new_with_label(l_grp,"Kein Text");
gtk_widget_show(l_radio);
gtk_widget_show(l_radio1);
// Das Layout festlegen
gtk_table_attach_defaults(GTK_TABLE(l_table),l_label,0,3,0,1);
gtk_table_attach_defaults(GTK_TABLE(l_table),l_radio,0,1,1,2);
gtk_table_attach_defaults(GTK_TABLE(l_table),l_radio1,2,3,1,2);
gtk_table_attach_defaults(GTK_TABLE(l_table),l_gbutton,1,2,2,3);
// Signalroutinen für das Fenster und
// den normalen Knopf
gtk_signal_connect(GTK_OBJECT(l_window), "delete_event",
                   GTK_SIGNAL_FUNC(GK_DelEvent),NULL);
gtk_signal_connect(GTK_OBJECT(l_window), "destroy",
                   GTK_SIGNAL_FUNC(GK_Destroy),NULL);
gtk_signal_connect(GTK_OBJECT(l_gbutton),"clicked",
                   GTK_SIGNAL_FUNC(GK_Label),V_GK_Label);
// Die Signalroutinen für die Radiobuttons aktivieren
gtk_signal_connect(GTK_OBJECT(l_radio),"toggled",
                   GTK_SIGNAL_FUNC(GK_Toggle),"Radio 1");
gtk_signal_connect(GTK_OBJECT(l_radio1),"toggled",
                   GTK_SIGNAL_FUNC(GK_Toggle),"Radio 2");
gtk_signal_connect(GTK_OBJECT(l_radio),"clicked",
                   GTK_SIGNAL_FUNC(GK_Click),"Radio 1");
gtk_signal_connect(GTK_OBJECT(l_radio1),"clicked",
                   GTK_SIGNAL_FUNC(GK_Click),"Radio 2");
gtk_signal_connect(GTK_OBJECT(l_radio),"pressed",
                   GTK_SIGNAL_FUNC(GK_Press),"Radio 1");
gtk_signal_connect(GTK_OBJECT(l_radio1),"pressed",
                   GTK_SIGNAL_FUNC(GK_Press),"Radio 2");
```

```
    gtk_container_add(GTK_CONTAINER(l_window),l_table);
    gtk_widget_show(l_gbutton);
    gtk_widget_show(l_table);
    gtk_widget_show(l_window);
    gtk_main();
    g_slist_free(l_grp);
    return(0);
}
```

2.9 Checkbuttons

Aufruf:

```
GtkWidget* gtk_check_button_new_with_label(const gchar *label);
```

Checkbuttons verhalten sich genauso wie Radiobuttons. Der einzige Unterschied besteht darin, dass Checkbuttons nicht in einer Gruppe zusammengefasst werden können, aus der dann nur ein Knopf zu einer Zeit aktiv sein kann. Zu einem gegebenen Zeitpunkt kann eine beliebige Anzahl von Checkbuttons aktiv sein.

Aus diesem Grunde wird die Gruppenliste aus dem Beispiel der Radiobuttons nicht mehr benötigt. Die Signale werden genauso generiert, wie im Abschnitt für die Radiobuttons bereits erklärt.

Angelegt wird eine Checkbox durch `gtk_check_button_new_with_label()`, wobei ein Text übergeben wird, der rechts von der Box erscheinen soll. Eine neu angelegte Checkbox ist zunächst inaktiv.

Ein ausführliches Beispiel findet sich im FTP-Archiv zu diesem Buch (das Sie unter `ftp://ftp.mut.de/public/linux/anwendungsentwicklung/` finden) im Verzeichnis `source/gtk`. An dieser Stelle verzichten wir darauf, um Platz zu sparen und weil bis auf das Anlegen der Checkboxen nichts Neues demonstriert werden könnte.

2.10 Entry Widget

Aufruf:

```
GtkWidget* gtk_entry_new_with_max_length(guint16       max);

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

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

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

void       gtk_entry_set_position(GtkEntry      *entry,
                                  gint           position);

gchar*     gtk_entry_get_text(GtkEntry      *entry);

void       gtk_entry_select_region(GtkEntry      *entry,
                                   gint           start,
                                   gint           end);

void       gtk_entry_set_editable(GtkEntry      *entry,
                                  gboolean       editable);

void       gtk_entry_set_max_length(GtkEntry      *entry,
                                    guint16        max);
```

Das Entry Widget ist eine einzeilige Texteingabebox. Sie wird durch Aufruf der Funktion `gtk_entry_new_with_max_length()` erstellt. Der Parameter bestimmt dabei, wie viele Zeichen maximal in die Textbox eingegeben werden können. Falls die Länge nachträglich geändert werden soll, so hilft der Aufruf von `gtk_entry_set_max_length()`. Falls der bereits eingetragene Text in der Box länger ist als die so neu definierte maximale Länge, so wird der Text passend abgeschnitten.

Die Eingabebox selbst kann auf verschiedene Art und Weise vom Programm mit Text versehen werden. Der Text `text` lässt sich durch den Aufruf von `gtk_entry_set_text()` in die Eingabebox `entry` eintragen. Wenn das Programm nur Text an den bereits vorhandenen anhängen soll, so wird entweder `gtk_entry_append_text()` (am Ende anhängen) oder `gtk_entry_prepend_text()` (am Anfang einfügen) aufgerufen.

Einen Zeiger auf den eingegebenen Text erhalten Sie durch den Aufruf von `gtk_entry_get_text()`. Allerdings sollten Sie den Text niemals direkt über diesen Zeiger manipulieren, sondern nur über die bereits erläuterten Funktionen.

Falls die Eingabe gesperrt werden soll, um Änderungen in der Eingabebox zu verhindern, so hilft der Aufruf von `gtk_entry_set_editable()`. Ist der Parameter `editable` gleich `FALSE`, so wird die Box gesperrt und Änderungen unterbunden, bis die Funktion mit `editable` gleich `TRUE` aufgerufen wird.

Die Position des Cursors in der Eingabebox lässt sich mit Hilfe der Funktion `gtk_entry_set_position()` manipulieren. Dabei hat das erste Zeichen die Position 0.

Teilbereiche des Texts lassen sich durch `gtk_entry_select_region()` bestimmen. Markiert wird dabei der Bereich inklusive der Position `start` bis ausschließlich der Position `end`.

Abbildung 2.6: Beispiel für ein Entry Widget

2.11 Dialogboxen

Aufruf:

```
GtkWidget* gtk_file_selection_new(const gchar      *title);

void       gtk_file_selection_set_filename(GtkFileSelection *filesel,
                                           const gchar      *filename);

gchar*     gtk_file_selection_get_filename(GtkFileSelection *filesel);

GtkWidget* gtk_dialog_new(void);

void      gtk_grab_add(GtkWidget  *widget);

void      gtk_grab_remove(GtkWidget  *widget);

GTK_WIDGET_SET_FLAGS(wid,flag)               /* Makro */
```

Dialogboxen sind Fenster, die auf eine Eingabe vom Benutzer warten. Dabei ist in der Regel die Bearbeitung von Benutzereingaben in anderen Fenstern der gleichen Applikation gesperrt (modaler Modus).

Diese Technik wird »Modalität« genannt und ist dann sinnvoll, wenn eine Abfrage beantwortet werden muss, bevor die gesamte Applikation weiterarbeiten kann. Sie sollte nicht genutzt werden, um die Eingabe innerhalb der Applikation auf jeweils ein Fenster zu fokussieren.

Angelegt wird ein Dateiauswahldialog durch Aufruf der Funktion `gtk_file_selection_new()`. Übergeben wird eine Zeichenkette `title`, die als Überschrift im Fenster der Dateiauswahl angezeigt wird. Der gewünschte Dateiname kann mit `gtk_file_selection_set_filename()` gesetzt werden. Dabei sind absolute Pfadangaben erlaubt, aber nicht zwingend erforderlich.

Wenn der OK-Knopf gedrückt wird, will das Programm natürlich auch den ausgewählten Dateinamen erfahren. Dies ist über die Funktion `gtk_file_selection_get_filename()` möglich. Übergeben wird das Widget der Dateiauswahl, zurückgegeben wird ein Zeiger auf den Dateinamen.

Hier ein kurzes Beispiel:

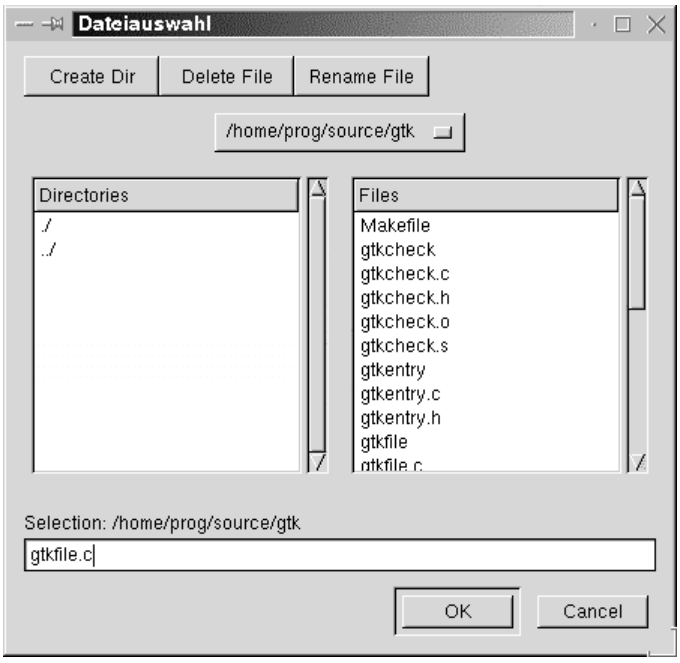

Abbildung 2.7: Beispiel für eine Dateiauswahlbox

```
/* gtkfile.c
 *
 * Eine Demo der Dateiauswahl unter GTK

 */

#include <gtk/gtk.h>

#include "gtkfile.h"   // Nichts von Interesse drin

/* Holt den selektierten Filenamen und gibt ihn auf der Konsole aus */
void GK_FileOK(GtkWidget *p_w, GtkFileSelection *p_fs)
{
    g_print ("Ausgewählt wurde: %s\n",
             gtk_file_selection_get_filename(GTK_FILE_SELECTION
```

```
(p_fs)));
}

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

int main (int argc, char *argv[])
{
    GtkWidget *l_filew;

    gtk_init (&argc, &argv);

    /* Dateiauswahl erstellen (Wdiget) */
    l_filew = gtk_file_selection_new ("Dateiauswahl");

    gtk_signal_connect (GTK_OBJECT (l_filew), "destroy",
                        (GtkSignalFunc) GK_Destroy, &l_filew);
    /* Den Signalhandler für den Ok-Knopf */
    gtk_signal_connect (GTK_OBJECT (GTK_FILE_SELECTION (l_filew)>
    ok_button),
                        "clicked", (GtkSignalFunc) GK_FileOK, l_filew );

    /* Wenn CANCEL gedrückt, dann Ende */
    gtk_signal_connect_object (GTK_OBJECT (GTK_FILE_SELECTION
                                           (l_filew)->cancel_button),
                               "clicked", (GtkSignalFunc)
gtk_widget_destroy,
                               GTK_OBJECT (l_filew));

    /* Nun die Voreinstellung für den Dateinamen */
    gtk_file_selection_set_filename (GTK_FILE_SELECTION(l_filew),
                                     "gtkfile.c");

    gtk_widget_show(l_filew);
    gtk_main ();
    return(0);
}
```

Bleiben noch die eigenen Dialogboxen. Diese werden durch `gtk_dialog_new()` erstellt, was in einem einfachen Fenster mit zwei Packingwidgets resultiert. Das eine Packingwidget (`action_area`) wird üblicherweise dazu verwendet, Knöpfe und sonstige Widgets anzuzuzeigen, in denen die Auswahl des Dialogs stattfindet. Das andere (`vbox`) zeigt die Texte bzw. die Beschreibung des Dialogs an. So könnte ein Label wie folgt in die Dialogbox eingesetzt werden:

```
GtkWidget      *l_label;
GtkWidget      *l_dialog;
GtkWidget      *l_button;
...

l_dialog = gtk_dialog_new();
/* Die Funktionen kommen später */
gtk_signal_connect(GTK_OBJECT(l_dialog),"destroy",
                   GTK_SIGNAL_FUNC(GK_Close),&l_dialog);
l_label  = gtk_label_new("Dies ist ein Labeltext");
gtk_box_pack_start(GTK_BOX(GTK_DIALOG(l_dialog)->vbox), l_label,
                   TRUE,TRUE,0);
gtk_widget_show(l_label);
/* gleich geht's weiter */
```

Ein normaler Knopf wird in die Actionarea eingehängt. Dabei kann das Programm den Knopf veranlassen, der Vorgabeknopf (`Default`) zu sein, wenn der Benutzer auf `Return` drückt. Dazu wird das Flag `GTK_CAN_DEFAULT` durch das Makro `GTK_WIDGET_SET_FLAGS()` für das Knopfwidget gesetzt. Um grafisch anzuzeigen, dass das Widget voreingestellt ist, kann durch `gtk_widget_grab_default()` ein Rahmen um das Widget gezogen werden. Der einzige Parameter dieser Funktion ist der Zeiger auf die Widgetstruktur.

```
/* weiter geht's */
l_button = gtk_button_new_with_label("OK");
GTK_WIDGET_SET_FLAGS(l_button, GTK_CAN_DEFAULT);
gtk_box_pack_start(GTK_BOX(GTK_DIALOG(l_dialog)->action_area),
                   l_button,TRUE,TRUE,0);
gtk_widget_grab_default(l_button);
gtk_widget_show(l_button);
/* Die Funktion kommt gleich noch */
gtk_signal_connect(GTK_OBJECT(l_button),"clicked",
                   GTK_SIGNAL_FUNC(GK_Button),NULL);
/* Der Rest kommt gleich */
```

Jetzt muss das Programm noch sicherstellen, dass nur die Dialogbox bearbeitet wird. Durch den Aufruf von `gtk_grab_add()` wird die Dialogbox im so genannten modalen Modus bearbeitet. Die restliche Vorgehensweise ist vergleichbar mit einem Aufruf von `gtk_main()`.

```
/* Der Rest der Funktion */
gtk_widget_show(l_dialog);
gtk_grab_add(l_dialog);
...
```

Fehlen nur noch die im Quelltext aufgerufenen Signalhandlerfunktionen. Wichtig ist bei Dialogboxen, dass der modale Modus wieder mit `gtk_grab_remove()` aufgehoben wird. Dies geschieht am besten, bevor das Dialogfenster gelöscht wird:

```
void GK_Close(GtkWidget *p_wid, gpointer data)
{
 // Hier den modalen Modus aufheben ...
 gtk_grab_remove(GTK_WIDGET(p_wid));
}

void GK_Button(GtkWidget *p_wid, gpointer data)
{ // Abbruch erzwingen
  gtk_widget_destroy(GTK_WIDGET(p_wid));
}
```

2.12 Menüs

Aufruf:

```
GtkItemFactory*     gtk_item_factory_new(GtkType
container_type,
                                         const gchar      *path,
                                         GtkAccelGroup    *accel_group);

void  gtk_item_factory_create_items(GtkItemFactory        *ifactory,
                                    guint                  n_entries,
                                    GtkItemFactoryEntry   *entries,
                                    gpointer
callback_data);

GtkAccelGroup*  gtk_accel_group_new(void);

void  gtk_accel_group_attach(GtkAccelGroup         *accel_group,
                             GtkObject             *object);
```

Als letztes Thema im Rahmen dieses GTK-Kapitels soll noch auf die Erstellung von Menüs eingegangen werden, ohne die eine vernünftige Benutzeroberfläche nur schwer zu gestalten ist. Wir wollen an dieser Stelle auf die einfachste Methode zurückgreifen, um solche Menüs anzulegen.

Die einfachste Art, Menüs anzulegen, geschieht über die so genannten ItemFactories. Dies sind vorgefüllte Strukturen mit fünf Einträgen für jeden Menüeintrag. Diese fünf Einträge lauten der Reihenfolge nach:

- Der Menüpfad:
  Jedes Menü wird angelegt wie eine Verzeichnisstruktur und mit der gleichen Notation. So legt `/Datei` ein Menü namens `Datei` an.
- Tastaturkürzel:
  Die Tastenkombination, mit der sich der Menüeintrag direkt ansprechen lässt, oder `NULL`. Gültige Einträge wären z. B. `<shift>b`, `<control>N` oder `<alt>Q`, wobei `<shift>` für die Taste `Shift` steht usw.

- Die Rückruffunktion:
  Die Funktion (ohne Parameter), die bei diesem Menüeintrag aufgerufen wird.
- Extraparameter:
  Dieser Parameter wird an die Rückruffunktion (Callback) übergeben, wenn der Menüeintrag ausgewählt wird.
- Itemtyp:
  Bestimmt die Art des zu erstellenden Menüeintrags:

`NULL`	Normaler Menüeintrag
`""`	Normaler Menüeintrag
`"<Title>"`	Erstellt einen Titeleintrag
`"<Item>"`	Normaler Menüeintrag
`"<CheckItem>"`	Erstellt einen Checkmenüeintrag. Dieser bekommt abwechselnd einen Haken (gewählt) oder kein Symbol vor dem Eintrag (nicht gewählt).
`"<ToggleItem>"`	Erstellt einen Toggleeintrag
`"<RadioItem>"`	Erstellt einen Radioeintrag. Aus einer Menge von Radiomenüeinträgen kann immer nur einer gleichzeitig ausgewählt sein.
`<path>`	Der Pfad zu den restlichen Radiomenüeinträgen, die sich gegenseitig ausschließen.
`"<Separator>"`	Trennlinie, um in längeren Menüs die Übersicht zu erhöhen.
`"<Tearoff>"`	Eine so genannte »Abreißkante«. Ein Klick auf dieses Element, und das Menü wird vom Basisfenster abgetrennt. Ein Klick auf den Separator im abgetrennten Fenster hängt das Menü wieder ein.
`"<Branch>"`	Ein Eintrag, der Untereinträge aufnehmen kann.
`"<LastBranch>"`	Legt ein Menü an, welches rechts ausgerichtet ist. Bestes Beispiel ist das Hilfsmenü, welches immer rechts auftaucht.

Tabelle 2.3: Gültige Einträge der ItemFactory

Ein einfaches Beispiel für ein Menü `Datei`, dessen einziger Eintrag `Ende` ist:

```
static GtkItemFactoryEntry V_GK_Menu[]=
{
   {"/_Datei",       NULL,      0,    0, "<Branch>"},
   {"/Datei/_Ende",  "<control>E",   GK_Ende,0}
};
```

Die Unterstriche in der Pfadangabe werden übrigens als Tastaturkürzel interpretiert und im Menü entsprechend markiert. Die Tastenkombinationen CTRL+E, ALT+D und ALT+E sind für das Beispiel also gleichbedeutend.

Zunächst einmal wird ein Programm wie gehabt ein Fenster anlegen und in dem Fenster ein vertikales Packing Widget einrichten. In diesem Packing Widget wird dann die Menüleiste eingetragen.

Für jede Menüleiste wird zunächst einmal eine Acceleratorgroup benötigt, was nichts anderes ist als eine Gruppe von Tastaturkürzeln, die im Menü definiert werden (siehe oben). Eine solche Struktur wird durch gtk_accel_group_new() initialisiert, während das Füllen selbst der Itemfactory überlassen werden soll. Wichtig ist die Initialisierung der Itemfactory, die durch die Funktion gtk_item_factory_new() erledigt wird. Als Parameter für diese Funktionen dienen der container_type, der die Art des Menüs festlegt und entweder GTK_TYPE_MENU_BAR, GTK_TYPE_MENU oder GTK_TYPE_OPTION_MENU sein kann. Für unser Beispiel beschränken wir uns auf ein »normales« Menü GTK_TYPE_MENU_BAR. Der Pfad path des Menüs sollte ein eindeutiger Namen für jede Menüleiste sein und in spitzen Klammern stehen. Der dritte Parameter accel_group ist ein Zeiger auf die Gruppe der Tastaturkürzel.

Die Menüleiste selbst wird durch gtk_item_factory_create_items() erstellt. Hier werden folgende Parameter erwartet: ifactory ist der Zeiger auf die angelegte GtkItemFactory-Struktur, die von gtk_item_factory_new() angelegt wurde. Die Anzahl der Menüeinträge wird im Parameter n_entries angegeben, damit die Funktion weiß, wie viele Menüeinträge in der Struktur abgelegt wurden. entries ist dann der Zeiger auf die gefüllte GtkItemFactoryEntry-Struktur, welche die Menüleiste definiert und bereits weiter oben ausführlich beschrieben wurde. callback_data ist entweder NULL oder ein Zeiger auf Daten, die der Callbackfunktion bei der Auswahl eines Menüeintrages übergeben werden sollen.

Somit könnte ein einfaches Beispiel mit ein wenig Funktionalität so aussehen:

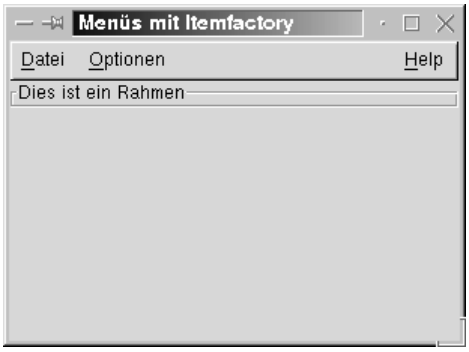

Abbildung 2.8: Beispiel: Menü

```
/* gtkmenu.c
 * Demo für Menüs in GTK
 *
 * Verbrochen nach einer Demo aus dem
 * GTK-Tutorial und erweitert um's Checkmenü und
 * den Button
 * M. Rathmann und C.Wieskotten
 */

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

GtkWidget    *V_GK_Button;
int           V_GK_Check = 0;      // Nicht gechecktes Menü

// Die Rückruffunktion Nummer 1 für's erste Menü
static void GK_hello( GtkWidget *p_w, gpointer  p_data )
{
  g_message ("Aufruf mit (%d)\n",(int) p_data);
}

static void GK_Check(GtkWidget *p_wid, gpointer data)
{
 V_GK_Check = 1 - V_GK_Check;
 if (V_GK_Check == 0)
    gtk_widget_show(V_GK_Button);
   else
    gtk_widget_hide(V_GK_Button);
}

static GtkItemFactoryEntry menu_items[] = {
{ "/_Datei",          NULL,         NULL,        0,  "<Branch>" },
{ "/Datei/Reiss",     NULL,         GK_hello,    4,  "<Tearoff>"},
{ "/Datei/_Neu",      "<control>N", GK_hello,    1,  NULL },
{ "/Datei/Ö_ffnen",   "<control>O", GK_hello,    2,  NULL },
{ "/Datei/_Speichern","<control>S", GK_hello,    3,  NULL },
{ "/Datei/Speichern _Als",  NULL,   NULL,        0,  NULL },
{ "/Datei/sep1",      NULL,         NULL,        0, "<Separator>" },
{ "/Datei/Ende",      "<control>E", gtk_main_quit, 0, NULL },
{ "/_Optionen",       NULL,         NULL,        0, "<Branch>" },
{ "/Optionen/Test",   NULL,         GK_Check,    0, "<CheckItem>" },
{ "/_Help",           NULL,         NULL,        0, "<LastBranch>" },
{ "/_Help/About",     NULL,         NULL,        0, NULL  },
};


void GK_MainMenu( GtkWidget  *p_window, GtkWidget **p_menubar )
```

```
{
  GtkItemFactory   *item_factory;
  GtkAccelGroup    *accel_group;

  gint nmenu_items = sizeof (menu_items) / sizeof (menu_items[0]);
  accel_group      = gtk_accel_group_new ();

  /*  Hier wird die Itemfactory initialisiert
      Param 1: Der Menütyp - entweder GTK_TYPE_MENU_BAR, GTK_TYPE_MENU,
               oder GTK_TYPE_OPTION_MENU.
      Param 2: Der Pfad des Menüs
      Param 3: Ein Zeiger auf einen gtk_accel_group.  Die Itemfactory
nutzt diese, um die Tastaturkürzel einzurichten
  */

  item_factory = gtk_item_factory_new(GTK_TYPE_MENU_BAR, "<main>",
accel_group);

  gtk_item_factory_create_items (item_factory, nmenu_items, menu_items,
NULL);

  /* Die Struktur für Tastaturkürzel dem Fenster zuordnen */
  gtk_accel_group_attach (accel_group, GTK_OBJECT (p_window));

  if (p_menubar)
    *p_menubar = gtk_item_factory_get_widget (item_factory, "<main>");
}

int main( int argc,
char *argv[] )
{
   GtkWidget    *l_window;       // Hauptfenster
   GtkWidget    *l_vbox;
   GtkWidget    *l_menubar;
   GtkWidget    *l_button;       // Zeiger auf den Knopf
   GtkWidget    *l_frame;        // Der Rahmen um die vbox

  gtk_init (&argc, &argv);
  l_frame = gtk_frame_new("Dies ist ein Rahmen");
  gtk_widget_show(l_frame);
  l_window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_signal_connect(GTK_OBJECT (l_window), "destroy",
                     GTK_SIGNAL_FUNC (gtk_main_quit),
                     "WM destroy");
  gtk_window_set_title (GTK_WINDOW(l_window), "Menüs mit Itemfactory");
  gtk_widget_set_usize (GTK_WIDGET(l_window), 300, 200);

  l_vbox = gtk_vbox_new (FALSE, 1);
```

```
  gtk_container_border_width (GTK_CONTAINER (l_vbox), 1);
  gtk_container_add (GTK_CONTAINER (l_window), l_vbox);
  gtk_widget_show (l_vbox);

  // Menü erstellen
  GK_MainMenu (l_window, &l_menubar);
  gtk_box_pack_start (GTK_BOX (l_vbox), l_menubar, FALSE, TRUE, 0);
  gtk_widget_show (l_menubar);

  // Den Knopf noch anlegen und erstmals anzeigen
  l_button = gtk_button_new_with_label("Ohne Funktion");
  gtk_box_pack_start(GTK_BOX(l_vbox),l_button,FALSE,TRUE,0);
  gtk_widget_show(l_button);
  V_GK_Button = l_button;

  gtk_box_pack_start(GTK_BOX(l_vbox),l_frame, FALSE, TRUE, 0);

  // Die Position des Fenster absolut festlegen und anzeigen
  gtk_widget_set_uposition(l_window,10,10);
  gtk_widget_show (l_window);
  gtk_main ();

  return(0);
}
```

Das Beispiel hat außerdem noch folgende Besonderheiten:

- Das Fenster wird mittels `gtk_widget_set_uposition()` an eine bestimmte Position auf dem Bildschirm gelegt. Die letztendliche Entscheidung darüber liegt allerdings beim Windowmanager.
- Der Knopf wird abhängig von der Auswahl des Menüeintrages unter Optionen angezeigt oder versteckt. Widgets lassen sich per `gtk_widget_hide()` verstecken, wobei das zu versteckende Widget als Parameter übergeben wird.
- Der Rahmen zeigt nur, dass der Platz vom versteckten Knopf durch das neue Layout vergeben wird.