Программирование графического интерфейса с помощью GTK+, Часть 2

В любом увлекательном деле, даже в таком, как программирование, есть своя рутина. Для меня такой рутиной всегда была интернационализация. Был бы я американским культурным империалистом, я бы вообще не обращал внимания на интернационализацию. Но я не американец и не империалист (даже в смысле культуры), а потому первая половина этой статьи будет посвящена тому, как научить программы GTK+ разговаривать на разных языках, иначе говоря, интернационализации приложений.

Для интернационализации приложений GTK+ мы воспользуемся пакетом GNU gettext, так что тем из вас, кто хорошо знаком с этим пакетом, будет достаточно беглого взгляда на приведенный ниже листинг программы, чтобы понять, что мы делаем. Для тех, кто не знаком с пакетом gettext, будут даны краткие, и никоим образом не исчерпывающие, пояснения. Более глубокое понимание работы утилит интернационализации GNU вы получите, ознакомившись со специальной документацией, которую можно найти, например, по адресу www.gnu.org/software/gettext/manual/.

Для тех, кто совсем не знаком с основами процесса перевода приложений Linux на разные языки, я кратко изложу базовые принципы. В процессе разработки программы весь текст интерфейса программы (названия кнопок и пунктов меню, текст диалоговых окон, сообщения об ошибках) пишется на одном языке, как правило на английском. При этом дальновидный программист заранее готовит интерфейс программы к переводу на другие языки (процесс подготовки программы к «многоязычию» называется интернационализацией). Программист добавляет в программу специальный код, отвечающий за работу с разными языками, а также помечает все строки интерфейса программы, которым потребуется перевод, специальными маркерами (макросами). Строка, помеченная для перевода, может выглядеть так:

```
g_print(gettext("Translate this!"));
```

Здесь gettext(), это макрос, который указывает, что необходимо перевести строку "Translate this!". Затем программист сканирует исходники программы с помощью утилиты xgettext. Утилита xgettext копирует все строки, помеченные для перевода, в специальный файл. Далее начинается процесс локализации, то есть, к адаптации программы к конкретной локали. С помощью утилиты перевода, например KBabel, на основе этого файла, полученного с помощью xgettext, создаются файлы перевода интерфейса, в которых каждой оригинальной строке интерфейса сопоставлен перевод на другой язык (такие файлы называют каталогами сообщений). Для того, чтобы интернационализированная программа могла воспользоваться каталогами сообщений, их нужно скомпилировать в специальный «машинный» формат с помощью утилиты masgfmt. Полученные в результате двоичные каталоги сообщений распространяются вместе с двоичным файлом приложения. Во время выполнения программа определяет, в какой локали она выполняется, и загружает двоичный файл, содержащий перевод сообщений интерфейса на соответствующий язык. В ходе работы программы все строки интерфейса, для которых есть перевод на соответствующий язык, заменяются строками перевода. Каким образом выполняется замена строк? Дело в том, что макросы, помечающие строки для перевода, во время работы программы выполняют замену строк. В приведенном выше примере во время выполнения программы макрос gettext() попытается найти перевод строки "Translate this!". Если перевод будет найден, аргументом функции g_print() станет строка перевода, если же перевод для данной строки найден не будет, макрос передаст функции g_print() исходную английскую строку. Возможно, эта схема показалась

вам слишком сложной, но я должен вас успокоить. Во-первых, на практике все выглядит проще, чем в описании (сейчас мы перейдем к практическому примеру, и вы сами это увидите). Во вторых, распространение файлов, содержащих перевод интерфейса отдельно от исполнимых файлов программы, представляет собой очень мощный механизм, благодаря которому у локализаторов программы появляется возможность переводить ее на другие языки, не прикасаясь к ее исходным текстам. Возможность подключить множество людей к процессу локализации вашей программы стоит того, чтобы выполнить пару лишних утилит.

Теперь я призываю расслабиться тех, кто устал от теории и вновь подключиться тех, кто ее пропустил, потому что мы переходим к практическому примеру интернационализации приложения GTK+. В качестве подспорья для интернационализации мы воспользуемся программой helloworld из предыдущей статьи. Начнем мы с того, что внесем некоторые изменения в исходный текст программы (этот исходник вы найдете в приложении к статье, в файле helloworld.c):

```
#include <gtk/gtk.h>
#include <libintl.h>
#define _(String) gettext (String)
#define gettext_noop(String) String
#define N_(String) gettext_noop (String)
#define GETTEXT_PACKAGE "helloworld"
#define LOCALEDIR "./locale"
static void button_clicked(GtkWidget * widget, gpointer data)
{
  g_print("Button pressed!\n");
}
static gboolean delete_event(GtkWidget * widget, GdkEvent * event,
gpointer data)
{
  g_print("Delete event occurred\n");
  return FALSE;
}
static void destroy(GtkWidget * widget, gpointer data)
{
  g_print("Destroy signal sent\n");
  gtk_main_quit();
}
int main(int argc, char ** argv)
{
  GtkWidget * window;
  GtkWidget * button;
  bindtextdomain (GETTEXT_PACKAGE, LOCALEDIR);
  bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
  textdomain (GETTEXT_PACKAGE);
  gtk_init(&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title(GTK_WINDOW(window), _("Hello World!"));
  gtk_container_set_border_width(GTK_CONTAINER(window), 10);
  g_signal_connect(G_OBJECT(window), "delete_event",
G_CALLBACK(delete_event), NULL);
  g_signal_connect(G_OBJECT(window), "destroy",
G_CALLBACK(destroy), NULL);
  button = gtk_button_new_with_label(_("Quit"));
```

```
  g_signal_connect(G_OBJECT(button), "clicked",
G_CALLBACK(button_clicked), NULL);
  g_signal_connect_swapped(G_OBJECT(button), "clicked",
G_CALLBACK(gtk_widget_destroy), G_OBJECT(window));
  gtk_container_add(GTK_CONTAINER(window), button);
  gtk_widget_show(button);
  gtk_widget_show(window);
  gtk_main();
  return 0;
}
```

Внимательный читатель сразу заметит, что мы добавили новый заголовочный файл - libintl.h . Этот файл содержит объявления функций, макросов и прочего, относящегося к интернационализации GNU gettext. Заголовочный файл libintl.h соответствует библиотеке libintl, которая должна быть связана с нашей программой. Эта библиотека является частью glibc, а потому, если сборка программы выполняется в Linux (или другой системе, использующей glibc), ее связывание с программой выполняется автоматически, без всяких дополнительных инструкций компоновщику. Вслед за директивами #include мы объявляем три макроса и две константы. Эти макросы представляют собой стандартную часть любого приложения, использующего интернационализацию GNU. Макрос _() является псевдонимом макроса gettext(). Сам макрос gettext() выполняет две функции, о которых говорилось в теоретическом введении, – помечает в исходном тексте программы те строки текста, которые требуют перевода, и заменяет оригинальную строку ее переводом во время выполнения программы. Мы заменяем его псевдонимом ради удобства, поскольку печатать один символ _ проще чем водить слово gettext. Забегая вперед, отметим, что с помощью макроса _() мы помечаем для перевода две строки – "Hello World!" и "Quit". Константа GETTEXT_PACKAGE указывает общее имя файлов каталогов сообщений данного приложения, из которых программа должна будет брать переводы строк. Файлов каталогов сообщений у каждого приложения много (по одному файлу для каждого языка и кодировки, на которые переведен интерфейс приложения), но все они имеют одно и то же имя (обычно соответствующее имени приложения с добавлением расширения .mo). При этом никакого конфликта не возникает, поскольку файлы, соответствующие разным языкам, хранятся в разных поддиректориях корневой директории каталогов сообщений. Если вы откроете одну из корневых директорий, в которой ресурсы локализации хранятся по умолчанию, например /usr/share/locale, то увидите в ней множество поддиректорий, имена которых совпадают с сокращенными именами различных локалей. В каждой из этих директорий вы найдете поддиректорию LC_MESSAGES. В этой директории обычно хранятся файлы локализации различных приложений для соответствующей локали. В системе есть несколько директорий, используемых по умолчанию для хранения файлов переводов. Программы GNOME ищут каталоги сообщений в поддиректориях директории /opt/gnome/share/locale. Универсальным хранилищем файлов переводов для разных приложений, использующих интернационализацию, основанную на GNU gettext, служит директория /usr/share/locale. Очевидно, что эти директории уместно использовать для хранения ресурсов тех приложений, которые глобально установлены в системе и доступны всем пользователям. Константа LOCALEDIR позволяет нам указать нестандартную корневую директорию для хранения файлов локализации нашего приложения. В качестве таковой мы указываем директорию locale, которая должна располагаться в рабочей директории программы.

В начале функции main() мы вызываем две функции, загружающие и настраивающие ресурсы локализации. Функция bindtextdomain(3) указывает системе интернационализации общее имя файлов ресурсов локализации и имя корневой директории, в которой они хранятся. Функция bind_textdomain_codeset(3) позволяет указать кодировку переведенных сообщений в файлах локализации. Во время выполнения наша программа определит имя

локали, в которой она работает, и будет искать файл с именем, заданным GETTEXT_PACKAGE, в соответствующей директории. Как видите, все английские строки, требующие перевода, представлены в программе как аргументы макроса _() (который, напомню, является синонимом макроса gettext()). Интерфейс программы helloworld готов к переводу на другие языки. Наша следующая задача – извлечь из исходного текста программы список строк, которые надлежит перевести. Это делается с помощью уже упомянутой утилиты xgettext. В окне консоли даем команду:

```
xgettext -C helloworld.c -k_
```

Ключ –C указывает программе, что она имеет дело с исходным файлом C/C++. Ключ –k позволяет нам указать вид маркера, которым помечены строки для перевода. В нашем случае маркером служит нижний дефис. Утилита xgettext не вносит никаких изменений в файл helloworld.c, но в результате ее выполнения на диске появится файл messages.po. Этот файл содержит все строки из файла helloworld.c, помеченные макросом _(). Мы сделаем копию этого файла и назовем ее ru.po. В этот файл мы добавим русский перевод строк интерфейса. Файлы переводов *.po представляют собой документы XML. Для работы с файлами переводов можно воспользоваться редактором Emacs, можно даже добавлять строки перевода «вручную» в обычном текстовом редакторе (при этом, конечно, необходимо соблюдать формат файлов *.po). Однако в вашей системе наверняка установлен гораздо более удобный инструмент, который, правда не является частью пакетов GTK+ или GNU Gettext. Речь идет о редакторе файлов перевода KBabel, входящем в состав пакета разработчика KDE.

Окно KBabel вертикально разделено на две половины. Левая половина содержит три окна. В самом верхнем окне мы найдем список всех строк, подлежащих переводу. В среднем окне мы выбираем строку для перевода, а в нижнем окне вводим сам перевод. Правая часть окна KBabel содержит некоторые вспомогательные инструменты. Выполним перевод всех строк на русский язык и сохраним изменения, внесенные в файл ru.po. Если вы просматриваете пиктограмму файла ru.po в менеджере Konqueror, то можете заметить, что цвет круга на пиктограмме файла изменился с красного на зеленый. Это значит, что все содержимое файла переведено (круг на иконке файла представляет собой диаграмму, отображающую процент переведенных строк). Полученный нами файл перевода ru.po представляет собой исходник двоичного ресурса локализации. Для того чтобы скомпилировать этот исходник в готовый к употреблению двоичный каталог сообщений, мы воспользуемся утилитой msgfmt:

```
msgfmt ru.po –o helloworld.mo
```

В результате у нас появится файл helloworld.mo, который мы сможем распространять вместе с двоичной версией нашей программы. Для того, чтобы во время выполнения программа могла использовать этот файл, его нужно помесить туда, где программа ожидает его найти. В нашем случае это директория ./locale/ru/LC_MESSAGES. Нам, кажется, осталось только скомпилировать программу helloworld. Теперь при запуске программы вы увидите надписи на русском языке (рис. 1).

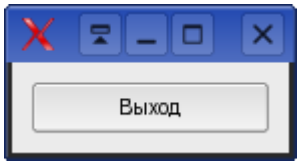

Рисунок 1. Русифицированная программа helloworld

В нашем кратком обзоре мы, разумеется, рассмотрели далеко не все аспекты интернационализации и локализации приложений с помощью системы GNU gettext, так что пренебрегать учебником GNU ни в коем случае не следует. Отметим здесь еще одну утилиту, которая может оказаться полезной в процессе локализации сложных проектов. Представьте себе, что вы локализуете свою (или чужую) программу. Вы создали файл каталога

сообщений, перевели все сообщения на ваш родной (или не родной) язык, скомпилировали двоичный каталог сообщений... Но жизнь не стоит на месте и программа, локализацией которой вы занимаетесь, продолжает развиваться. В программе появляются новые строки, требующие перевода. Было бы очень неразумно начинать весь процесс локализации сначала из-за того, что в программе появилась новая текстовая строка. Ведь у вас уже есть каталог сообщений, который содержит перевод всех остальных строк. Если вы не хотите переписывать вручную весь перевод каждый раз, когда каталог сообщений, генерируемый утилитой xgettext, изменится, вам на помощь придет утилита msgmerge. Эта утилита позволяет объединить старый, уже переведенный каталог сообщений и новый каталог, содержащий дополнения.

Две кнопки

Для того чтобы исследовать возможности GTK+ нам, конечно, понадобятся программы с более сложным интерфейсом, чем окно с одной кнопкой. Логично будет теперь рассмотреть некоторые принципы построения интерфейсов программ GTK+. В первой статье этой серии мы уже упоминали объекты-контейнеры, которые управляют размером и расположением дочерних визуальных элементов. Главное окно приложения GTK+ само представляет собой объект-контейнер. Впрочем, возможности главного окна, как объекта-контейнера, весьма ограничены. В частности, попытка добавить в окно приложения вторую кнопку наталкивается на неожиданное препятствие, - у главного окна приложения может быть только один дочерний визуальный элемент. Если мы хотим, чтобы окно приложения содержало более одного визуального элемента (естественное желание, не правда ли?), мы должны сначала разместить в окне дочерний объект-контейнер, способный управлять большим числом элементов. Объектов-контейнеров в GTK+ реализовано немало. Все они являются потомками объекта GtkContainer, реализующего абстрактный контейнер. В частности, объект-контейнер главного окна принадлежит классу GtkBin, представляющему собой контейнер, способный содержать (вы догадались!) только один дочерний элемент. Другие контейнеры позволяют управлять сразу многими дочерними визуальными элементами. Все объекты контейнеры GTK+ можно разделить на две категории. Одни контейнеры управляют расположением дочерних визуальных элементов, но сами визуальными элементами не являются. К этой категории контейнеров относится контейнер GtkHBox, с которым мы познакомимся ниже. Ко второй категории относятся контейнеры, которые по умолчанию включают определенные визуальные элементы управления. Примером контейнеров этого типа может служить контейнер GtkNotebook, который создает панель с несколькими вкладками. Мы рассмотрим пример использования простого контейнера GtkHBox, который позволяет расположить горизонтально несколько дочерних элементов. Рассмотрим текст программы buttontest, в которой мы воспользуемся не одной, а двумя кнопками.

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

static void button_clicked(GtkWidget * widget, gpointer data)
{
  gint i = * (gint *) data;
  g_print("Button #%i is pressed!\n", i);
}
static gboolean delete_event(GtkWidget * widget, GdkEvent * event,
gpointer data)
{
  g_print("Delete event occurred\n");
  return FALSE;
}
```

```
static void destroy(GtkWidget * widget, gpointer data)
{
  g_print("Destroy signal sent\n");
  gtk_main_quit();
}
int main(int argc, char ** argv)
{
  GtkWidget * window;
  GtkWidget * button1;
  GtkWidget * button2;
  GtkWidget * box;
  gint i1, i2;
  gtk_init(&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title(GTK_WINDOW(window), "Buttons");
  gtk_container_set_border_width(GTK_CONTAINER(window), 10);
  g_signal_connect(G_OBJECT(window), "delete_event",
G_CALLBACK(delete_event), NULL);
  g_signal_connect(G_OBJECT(window), "destroy",
G_CALLBACK(destroy), NULL);
  box = gtk_hbox_new(TRUE, 2);
  gtk_container_add(GTK_CONTAINER(window), box);
  button1 = gtk_button_new_with_label("Выход");
  i1 = 1;
  g_signal_connect(G_OBJECT(button1), "clicked",
G_CALLBACK(button_clicked), &i1);
  g_signal_connect_swapped(G_OBJECT(button1), "clicked",
G_CALLBACK(gtk_widget_destroy), G_OBJECT(window));
  gtk_box_pack_start(GTK_BOX(box), button1, TRUE, TRUE, 0);
  button2 = gtk_button_new_with_label("Кнопка 2");
  i2 = 2;
  g_signal_connect(G_OBJECT(button2), "clicked",
G_CALLBACK(button_clicked), &i2);
  gtk_box_pack_start(GTK_BOX(box), button2, TRUE, TRUE, 0);
  gtk_widget_show(button1);
  gtk_widget_show(button2);
  gtk_widget_show(box);
  gtk_widget_show(window);
  gtk_main();
  return 0;
}
```

В этом примере дочерним элементом главного окна программы служит контейнер box, который, в свою очередь, включает две кнопки – button1 и button2. Контейнер типа GtkHBox создается функцией-конструктором gtk_hbox_new(). У функции-конструктора два аргумента. Первый аргумент, значение типа gboolean, позволяет указать, должны ли дочерние элементы контейнера иметь одинаковые размеры. Если передать в этом аргументе значение TRUE, размеры дочерних элементов будут одинаковыми. Второй аргумент имеет тип gint и позволяет указать расстояние между дочерними элементами контейнера в пикселях. Здесь уместно еще раз обратить внимание на систему типов GTK+. В целях улучшения переносимости с одной платформы на другую, GTK+ определяет собственные аналоги простых типов данных C. Многие из этих типов представляют собой псевдонимы типов C со

схожими именами. Например, тип gint является псевдонимом типа int. Сложнее обстоит дело с типом gboolean. В языке C (в отличие от C++) нет встроенного булевого типа, и тип gboolean является псевдонимом типа gint. Константы TRUE и FALSE объявлены так, чтобы соответствовать результатам логических операций C. (FALSE = 0, TRUE = !FALSE, то есть любое ненулевое значение).

После того как мы создали контейнер, мы делаем его дочерним элементом главного окна с помощью знакомой нам функции gtk_container_add(). Далее мы создаем кнопку и назначаем ей обработчики сигналов, так же как мы делали в предыдущем примере. Наша следующая задача – добавить кнопку в контейнер box. Для этого мы воспользуемся функцией gtk_box_pack_start(). Функция gtk_box_pack_start() добавляет новые визуальные элементы в контейнер в порядке слева направо, если контейнер представляет собой «горизонтальный ящик» GtkHBox, и сверху вниз, если контейнер представляет собой «вертикальный ящик» GtkVBox. Функция gtk_box_pack_end() добавляет элементы в противоположном порядке, соответственно справа налево и снизу вверх. Функции gtk_box_pack_start() и gtk_box_pack_end() обладают одинаковым набором параметров. Первым параметром каждой функции является указатель на объект-контейнер. Вторым параметром служит указатель на добавляемый объект. Третий и четвертый параметры позволяют указать порядок добавления и распределения нового пространства, добавляемого вместе с новым элементом. Последний параметр указывает сколько дополнительных пикселей следует расположить между новы элементом и его соседями.

Хотя объект box сам по себе не создает никаких визуальных элементов, его следует сделать видимым с помощью функции gtk_widget_show(), как и все визуальные элементы программы. В результате получаем окно с двумя кнопками (рис. 2).