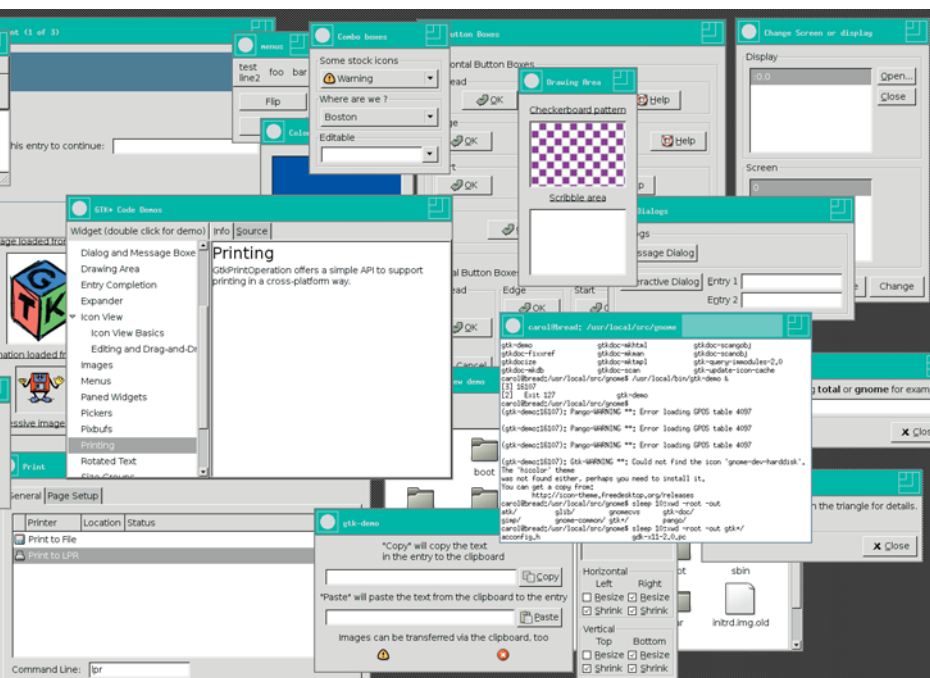




## ИЗУЧАЕМ СИГНАЛЫ

**ЧАСТЬ 3:** Наше знакомство с сигналами и событиями *GTK+* было прервано изучением увлекательной проблемы интернационализации приложений. Сегодня **Андрей Боровский** намерен вернуться к ним и показать, что их не зря называют движущей силой любой программы, основанной на *GTK+*.



Вы, конечно, заметили, что в самом первом примере программы *GTK+* мы использовали две разные функции для связывания обработчика сигнала и объекта: `g_signal_connect()` и `g_signal_connect_swapped()`. Общим у этих функций является то, что в их первом параметре передается объект-источник сигнала, для которого назначается обработчик. Разница между этими функциями заключается в порядке передачи аргументов функции-обработчику сигнала. Если обработчик сигнала был связан с объектом с помощью `g_signal_connect()`, первым параметром, передаваемым обработчику, является первый параметр `g_signal_connect()`, то есть указатель на объект-источник сигнала (информация об объекте-источнике может быть полезной, поскольку один обработчик сигналов может быть связан с несколькими объектами). Последним параметром, который получает обработчик сигнала, связанного с помощью `g_signal_connect()`, является последний параметр `g_signal_connect()`, в котором обработчику передаются произвольные дополнительные данные. Если объект и сигнал были связаны между собой с помощью `g_signal_connect_swapped()`, при вызове обработчика сигнала первым параметром, передаваемым обработчику, является последний параметр `g_signal_connect_swapped()` (что бы он там ни содержал). В то же время, первый параметр `g_signal_connect_swapped()`, который представляет

собой указатель на объект-источник сигнала, становится последним параметром, передаваемым обработчику сигнала. Зачем же нужны две разные функции связывания объекта и обработчика сигнала?

Для связывания наших собственных обработчиков сигнала мы будем использовать, в основном, функцию `g_signal_connect()`, так что заголовок функции-обработчика будет иметь вид:

```
void callback_func( GtkWidget *widget,
    ... /* дополнительные аргументы, в зависимости от типа
    сигнала */
    gpointer callback_data );
```

где `widget` – первый параметр `g_signal_connect()`, а `callback_data` – последний параметр.

Функция `g_signal_connect_swapped()` используется в двух ситуациях. Во-первых, ее применяют, когда в качестве обработчика сигнала, эмитируемого объектом, нужно назначить функцию, которая, вообще говоря, не является обработчиком сигнала. Так, например, мы поступали при вызове

```
g_signal_connect_swapped(G_OBJECT(button1), "clicked", G_
CALLBACK(gtk_widget_destroy), G_OBJECT(window));
```

Функция `gtk_widget_destroy()` не является обработчиком какого-либо сигнала (чтобы подчеркнуть это, мы явным образом приводили `gtk_widget_destroy` к типу `GtkCallback`), эта функция просто уничтожает визуальный элемент, указатель на который передан ей в качестве аргумента. Наша задача – в ответ на щелчок кнопки вызывать `gtk_widget_destroy()` для объекта `window`.

Если бы мы связывали сигнал `clicked` с функцией `gtk_widget_destroy()` с помощью функции `g_signal_connect()`, аргументом функции `gtk_widget_destroy()` стал бы объект `button1`. Однако, поскольку мы используем функцию `g_signal_connect_swapped()`, первым аргументом, переданным функции `gtk_widget_destroy()`, окажется последний аргумент `g_signal_connect_swapped()`, то есть, объект `window`. Я надеюсь, что вы понимаете, почему вместо функции `g_signal_connect_swapped()` нельзя использовать функцию `g_signal_connect()` с переставленными местами первым и последним аргументами. Ведь в этом случае сигнал окажется связанным не с тем объектом!

В примере `buttontest.c`, который вы найдете на диске, мы связываем сигнал `clicked` кнопки `button2` с функцией `g_print()`:

```
g_signal_connect_swapped(G_OBJECT(button2), "clicked", G_
CALLBACK(g_print), "Button is pressed!\n");
```

Как вы уже знаете, последний аргумент `g_signal_connect_swapped()` станет первым аргументом функции `g_print()`. В результате, каждый раз при щелчке по кнопке, на экране терминала будет распечатываться строка "Button is pressed!". Указатель на объект `button2` передается функции `g_print()` как второй аргумент. В приведенном выше фрагменте

» Месяц назад Мы изучали процесс интернационализации приложений *GTK+*.

# и события



## Передача аргументов функциям C

Современные компиляторы C следят за тем, чтобы число (и тип) аргументов, передаваемых функции при ее вызове, совпадали с числом и типом аргументов, указанным в заголовке функции, однако, если функция вызывается косвенно (как, например, функция-обработчик сигнала), в силу вступает старое правило C: обрабатываются первые **n** аргументов, на которые есть ссылки в теле функции, а остальные аргументы просто игнорируются. Например, при вызове функции `gtk_widget_destroy()` как обработчика события `clicked`, функции передается два аргумента. Первый аргумент функция использует, второй – игнорирует.

кода функция `g_print()` игнорирует все переданные ей аргументы, кроме первого, но если вы замените последний аргумент `g_signal_connect_swapped()` на `"Button is Pressed %i\n"`, то увидите, что кроме текста на экране будет распечатано число, являющееся численным представлением указателя на объект `button2`.

Еще одна ситуация, в которой мы можем обратиться к функции `g_signal_connect_swapped()`, возникает тогда, когда мы хотим, чтобы функция-обработчик, вызванная для обработки события, источником которого является один объект, думала, что обрабатывает событие, связанное с другим объектом. Если вы считаете, что обманывать обработчики событий нехорошо, то в большинстве случаев вы правы. Однако, иногда такой «обман» может оказаться полезным, как будет показано в следующем примере.

## События

Мы уже упоминали, что события *GTK+* представляют собой разновидность сигналов. В отличие от остальных сигналов *GTK+*, события тесно связаны с событиями системы X-Window. С одним из событий, а именно – `delete_event`, мы уже встречались. В общем виде заголовков функции-обработчика события выглядит так:

```
gint callback_func(GtkWidget *widget,
                  GdkEvent *event,
                  gpointer callback_data);
```

Здесь необходимо сделать некоторые уточнения. Так же как объект `widget` является корнем иерархии объектов, представляющих различные визуальные элементы и, зачастую, нам приходится приводить тип `GtkWidget *` к типу указателя на соответствующий визуальный элемент, тип `GdkEvent` является корнем иерархии объектов событий, в которой каждому событию соответствует свой объект. Объект события – это обычная структура C, которая несет о нем специфическую информацию (поскольку разные события несут разную информацию, неудивительно, что каждому событию соответствует своя структура). Например, событию `button_press_event`, возникающему при щелчке мышью, соответствует объект `GdkEventButton`. Он представляет собой структуру C, которая, помимо прочего, имеет поле `button` (содержит информацию о том, какая кнопка была нажата), и поля `x` и `y`, содержащие координаты указателя мыши в момент щелчка.

В качестве примера использования событий мы рассмотрим программу `watchwindow`, которая следит за состоянием своего главного окна (распахнута на весь экран, свернуто на панель задач, занимает

часть экрана). Ниже приводится исходный текст программы (вы найдете его также в файле `watchwindow.c` на диске).

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

gint on_window_state(GtkLabel * label, GdkEventWindowState * window_state,
                    gpointer callback_data )
{
    if (window_state->new_window_state & GDK_WINDOW_STATE_ICONIFIED) {
        g_print("Window is iconified\n");
        return 0;
    }
    if (window_state->new_window_state & GDK_WINDOW_STATE_MAXIMIZED) {
        gtk_label_set_text(label, "Window is maximized");
        return 0;
    }
    gtk_label_set_text(label, "Window is in normal state");
    return 0;
}

static gboolean delete_event(GtkWidget * widget, GdkEvent * event,
                             gpointer data)
{
    return FALSE;
}

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

int main(int argc, char ** argv)
{
    GtkWidget * window;
    GtkWidget * label;
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "WatchWindow");
    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);
    label = gtk_label_new("Starting");
    gtk_container_add(GTK_CONTAINER(window), label);
    g_signal_connect_swapped(G_OBJECT(window), "window_state_event",
                             G_CALLBACK(on_window_state), G_OBJECT(label));
    gtk_widget_show(label);
    gtk_widget_show(window);
    gtk_main();
    return 0;
}
```

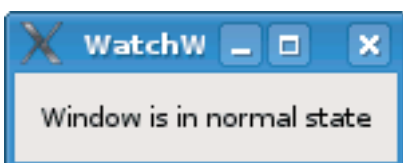
Главным визуальным элементом главного окна программы является объект `label` типа `GtkLabel`. Этот объект представляет собой простую текстовую метку, на которую выводится текст о текущем состоянии окна. Для того, чтобы главное окно могло получать информацию об изме- »

» нении своего состояния, оно должно обрабатывать сигнал-событие `window_state_event`. Мы связываем обработчик этого события, функцию `on_window_state()`, с объектом-источником события `window` с помощью функции `g_signal_connect_swapped()`. Это как раз тот случай, когда применение `g_signal_connect_swapped()` оправдано. В качестве последнего аргумента мы передаем функции `g_signal_connect_swapped()` указатель на объект `label`. Из этого следует, что функция-обработчик сигнала получит в качестве первого аргумента указатель на объект `label`, а не на объект `window`.

Перейдем теперь к функции-обработчику. Поскольку мы знаем, какое именно событие будет обрабатывать эта функция, мы можем заменить базовые типы аргументов `GtkWidget *` и `GdkEvent *` в ее заголовке на те типы, с которыми функции в действительности придется иметь дело, то есть на `GtkLabel *` и `GdkEventWindowState *`. Это позволит нам сократить исходный текст программы на пару строк за счет преобразования типов. Структура `GdkEventWindowState`, которую можно условно рассматривать как потомок объекта `GdkEvent`, несет информацию о событии `window_state_event`. Информация о состоянии окна содержится в поле `new_window_state` структуры `GdkEventWindowState`. Это поле содержит набор флагов, отражающих новое состояние кода. В начале функции обработчика мы проверяем, установлен ли в поле `new_window_state` флаг `GDK_WINDOW_STATE_ICONIFIED`. Наличие этого флага означает, что окно свернуто на панель задач. В этом случае мы с помощью функции `g_print()` распечатываем в окне терминала строку "Window is iconified" (как вы понимаете, нет смысла выводить что-либо в поле окна, которое свернуто на панель задач) и завершаем работу функции. Если флаг не установлен, мы проверяем, стоит ли флаг `GDK_WINDOW_STATE_MAXIMIZED` (окно распахнуто на весь экран). Если да, то мы выводим в поле метки `label` текст "Window is maximized" и завершаем выполнение функции-обработчика. Если же ни тот, ни другой флаги не установлены, мы задаем в качестве текста метки строку "Window is in normal state".

Порядок проверки флагов `new_window_state` имеет значение, поскольку, как это ни странно, флаги `GDK_WINDOW_STATE_ICONIFIED` и `GDK_WINDOW_STATE_MAXIMIZED` могут быть установлены одновременно. Такое происходит, когда окно сворачивается на панель задач из максимально распахнутого состояния. Поэтому сначала мы проверяем, установлен ли флаг `GDK_WINDOW_STATE_ICONIFIED` (что точно свидетельствует о том, что окно свернуто), и только если этот флаг не установлен, проверяем, установлен ли флаг, указывающий на то, что окну приданы максимальные размеры. Поле `new_window_state` может содержать и другие флаги, но они нас сейчас не интересуют. Таким образом окно нашего приложения способно получить (и сообщить нам) важную информацию о своих размерах (рис. 1).

» Рис. 1.  
Окно, которое знает  
о своем состоянии.



## Продолжаем изучать контейнеры

Ранее мы уже имели дело с объектами-контейнерами `GtkHBox` и `GtkVBox`. Прежде чем переходить к сложным средствам компоновки, нам стоит познакомиться и с другими типами контейнеров.

В серии, посвященной `Qt/KDE`, я приводил пример приложения, предназначенного для просмотра шрифтов. Не могу удержаться от того, чтобы не привести пример подобного приложения для `GTK+`, тем более что написать его совсем несложно (исходный текст программы вы найдете в файле `fontview.c`):

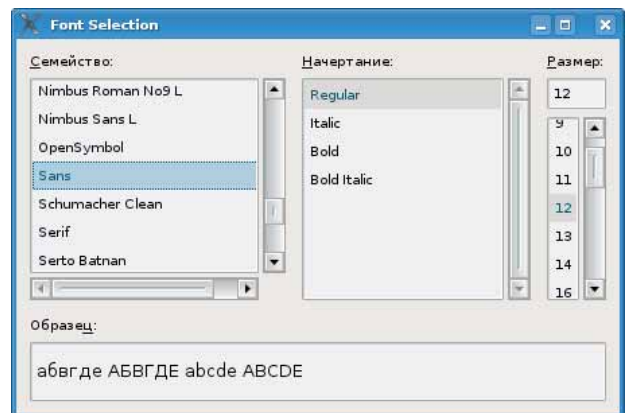
```
#include <gtk/gtk.h>

static gboolean delete_event(GtkWidget * widget, GdkEvent * event,
                             gpointer data)
{
    return FALSE;
}

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

```
int main(int argc, char ** argv)
{
    GtkWidget * window;
    GtkWidget * font_selection;
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Font Selection");
    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);
    font_selection = gtk_font_selection_new();
    gtk_container_add(GTK_CONTAINER(window), font_selection);
    gtk_widget_show(font_selection);
    gtk_widget_show(window);
    gtk_main();
    return 0;
}
```

Основным элементом главного окна этой программы является объект `GtkFontSelection`, который представляет собой полноценный визуальный элемент просмотра и выбора шрифтов (рис. 2).



» (Рис. 2) Компонент выбора шрифтов в окне приложения.

Согласно принципам `GTK+`, объект `GtkFontSelection` создается функцией `gtk_font_selection_new()`. Мы связываем объект с окном с помощью функции `gtk_container_add()` и делаем его видимым при помощи вызова `gtk_widget_show()`. На самом деле, объект `GtkFontSelection` предназначен, конечно, не для построения программ просмотра наличия шрифтов в системе. Назначение этого объекта – быть частью окон настроек свойств приложений `GTK+`, допускающих выбор шрифта для вывода различных текстовых элементов.

Сейчас для нас важно, что объект `GtkFontSelection` является прямым потомком объекта `GtkVBox`. Ничего странного тут нет, поскольку, в принципе, любой сложный визуальный элемент, содержащий несколько дочерних элементов, должен быть потомком какого-либо контейнера. Хотите узнать, какие визуальные компоненты содержит контейнер `GtkFontSelection`, и даже получить к ним доступ? Это просто. После того как объект `font_selection` создан, добавьте в текст программы строку:

```
gtk_container_foreach(GTK_CONTAINER(font_selection), child_callback,
NULL);
```

Функция `gtk_container_foreach()` вызовет функцию обратного вызова, в данном случае – `child_callback()`, для каждого непосредственного дочернего визуального элемента контейнера `font_selection`. Последний параметр `gtk_container_foreach()` предназначен для передачи произвольных данных. Функция `child_callback()` выглядит так:

```
void child_callback(GtkWidget * widget, gpointer data)
{
    g_print("%s\n", gtk_widget_get_name(widget));
    if (GTK_IS_CONTAINER(widget))
```

```
gtk_container_foreach(GTK_CONTAINER(widget), child_callback,
NULL);
}
```

Первый параметр функции — очередной дочерний визуальный элемент, для которого она вызвана, второй параметр — дополнительные данные. Мы распечатываем имя переданного нам визуального элемента, затем проверяем, является ли он контейнером. Если переданный нам виджет сам является контейнером, мы рекурсивно вызываем для него функцию `gtk_container_foreach()`. В результате выполнения программы будет распечатан список `GtkTable`, `GtkScrolledWindow`, `GtkTreeView`, `GtkScrolledWindow`, `GtkTreeView`, `GtkScrolledWindow`, `GtkTreeView`, `GtkLabel`, `GtkLabel`, `GtkLabel`, `GtkEntry`, `GtkVBox`, `GtkLabel`, `GtkHBox`, `GtkEntry`. Этот список представляет собой нечто вроде рекурсивного обхода дерева дочерних визуальных элементов контейнера в порядке 1-2-3. Функцию Отметим, что `gtk_container_foreach()` можно использовать только для перечисления внутренних элементов контейнера, то есть таких, которые не были добавлены явным образом.

Перейдем теперь к еще одному базовому типу контейнера, который должен особенно понравиться тем, кто программировал на Delphi, Borland C++ Builder или с Windows.Forms. Контейнеры `GtkHBox` и `GtkVBox` таят огромные возможности в плане компоновки элементов интерфейса. Такие приложения, как текстовый редактор, web-браузер или утилиты настройки оборудования можно написать, используя для компоновки визуальных элементов исключительно «горизонтальный» и «вертикальный» контейнеры. Полезная особенность этих типов контейнеров заключается в том, что по умолчанию они управляют расположением и размером дочерних элементов наиболее естественным образом, так что пользователю не приходится беспокоиться о дополнительных настройках. Тем не менее, свободы, предоставляемой этими контейнерами, не всегда достаточно. Если вы пишете игровое или мультимедиа-приложение, вы можете захотеть расположить визуальные элементы управления совершенно необычным образом. Для этого вам следует воспользоваться контейнером `GtkFixed`. Объект-контейнер `GtkFixed` является прямым потомком объекта `GtkContainer`, от которого происходит также объект `GtkBox`, являющийся родоначальником объектов `GtkHBox` и `GtkVBox`. Этот контейнер позволяет располагать дочерние элементы в фиксированных позициях, заданных координатами x и y относительно верхнего левого угла контейнера. Контейнер `GtkFixed` является «фиксированным» потому, что, в отличие от «горизонтального» и «вертикального» контейнеров, расположение и размеры его дочерних элементов не меняются при изменении размеров окна. Он дает вам большую свободу в расположении дочерних элементов, но требует и большей ответственности при управлении ими. Рассмотрим простую программу, использующую контейнер `GtkFixed` (исходный текст вы найдете в файле `fixedbuttons.c`):

```
#include <gtk/gtk.h>
static gboolean delete_event(GtkWidget * widget, GdkEvent * event,
gpointer data)
{
return FALSE;
}
static void destroy(GtkWidget * widget, gpointer data)
{
gtk_main_quit();
}
int main(int argc, char ** argv)
{
GtkWidget * window;
GtkWidget * fixed_container;
GtkWidget * button;
gtk_init(&argc, &argv);
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_title(GTK_WINDOW(window), "Fixed Buttons Demo");
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);
fixed_container = gtk_fixed_new ();
gtk_container_add(GTK_CONTAINER(window), fixed_container);
button = gtk_button_new_with_label("Button1");
gtk_fixed_put(GTK_FIXED(fixed_container), button, 5, 5);
gtk_widget_show(button);
button = gtk_button_new_with_label("Button2");
gtk_fixed_put(GTK_FIXED(fixed_container), button, 25, 35);
gtk_widget_show(button);
button = gtk_button_new_with_label("Button3");
gtk_fixed_put(GTK_FIXED(fixed_container), button, 45, 65);
gtk_widget_show(button);
gtk_widget_show(fixed_container);
gtk_widget_show(window);
gtk_main();
return 0;
}
```

Мы создаем объект `GtkFixed` с помощью функции `gtk_fixed_new()`. Далее мы последовательно создаем и добавляем в контейнер три кнопки. Добавление нового элемента в контейнер выполняется функцией `gtk_fixed_put()`. Первым аргументом этой функции должен быть, естественно, указатель на объект-контейнер. Вторым аргументом является указатель на добавляемый в контейнер дочерний объект, а третий и четвертый аргументы служат, соответственно, для передачи координат x и y верхнего левого угла дочернего объекта. Каждый дочерний элемент, а также сам контейнер, необходимо сделать видимым с помощью вызова `gtk_widget_show()`. Теперь кнопки в нашем приложении расположены по диагонали (рис. 3). Расположение уже добавленных в контейнер элементов можно изменить с помощью функции `gtk_fixed_move()`. Список аргументов у этой функции такой же, как и у `gtk_fixed_put()`.

У объекта `GtkFixed` есть свойство `children`, которое содержит список всех дочерних элементов. Список представляет собой объект с не очень благозвучным для русского уха названием `GList` (этот объект реализует в GTK+ классический связный список). Элементами списка `children` являются объекты типа `GtkFixedChild`. Объект `GtkFixedChild` содержит указатель на соответствующий ему дочерний объект и его координаты. Ниже показано, как с помощью списка `children` можно вывести на терминал координаты всех дочерних объектов контейнера `GtkFixed`.

```
GList * list;
GtkFixedChild * child;
...
list = GTK_FIXED(fixed_container)->children;
while (list != NULL) {
child = list->data;
g_print("x = %i, y = %i\n", child->x, child->y);
list = g_list_next(list);
}
```

Поскольку `GList` представляет собой классический связный список, макросы типа `g_list_next()` и `g_list_previous()` возвращают указатель на элемент `GList`.

На этом наше, в чем-то — шапочное, знакомство с объектами-контейнерами можно считать законченным. Однако, как вы, наверное, помните еще со времен серии о Qt/KDE, пользоваться всем этим богатством напрямую зачастую оказывается неудобно. В следующий раз мы узнаем, как конструировать пользовательские интерфейсы в режиме WYSIWYG при помощи Glade.

► Рис. 3.  
Произвольное  
расположение окон.

