Заключительные

ЧАСТЬ 9 Пришла пора ставить точку — но напоследок **Андрей Боровский** изыскал возможность рассказать о двух блестящих компонентах GNOME и работе с *GTK* из C++.

Сегодня мы завершим беглый обзор основных библиотек GNOME, начатый в предыдущей статье, примером использования libgnomeui. Эта библиотека содержит несколько дополнительных визуальных компонентов, призванных, как сказано в документации, заставить интерфейс программы GNOME сверкать по-настоящему. В нашем примере мы рассмотрим два блистательных виджета из библиотеки libgnomeui – GnomeDruid и GnomeIconSelection.

Компонент GnomeDruid предназначен для создания мастеров (хотя в русском переводе интерфейса GNOME эти компоненты именуются «помощниками», я предпочитаю называть их именно так), сопровождающих пользователя через последовательные этапы некоего процесса конфигурации или выбора параметров. Пример использования GnomeDruid, который должен быть вам уже хорошо знаком – это мастер создания нового проекта в интегрированной среде *Anjuta*.

Второй из рассматриваемых компонентов, GnomeIconSelection, представляет собой окно, в которое может быть загружен набор пиктограмм. Этот компонент применяется в тех случаях, когда необходимо предоставить пользователю возможность выбрать пиктограмму из заранее заданного набора. Оба компонента относятся к категории контейнеров (GnomeDruid является потомком GtkContainer, а GnomeIconSelection – потомком GtkVBox).

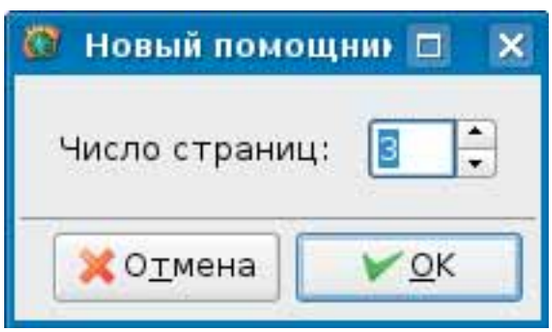


Рис. 1. Создание мастера (помощника) в среде Glade.

Создадим в среде *Anjuta* новый проект приложения GNOME с использованием *Glade* (программа, которую вы найдете на диске, называется druiddemo). В форму главного окна приложения помещаем компонент «Помощник» (то есть «мастер») со вкладки Gnome палитры компонентов *Glade*. При этом система спросит нас (**Рис. 1**), сколько страниц должен содержать создаваемый мастер (количество страниц можно будет изменить позднее, во время редактирования).

Рассчитывая количество страниц будущего мастера, следует учесть, что первая и последняя страницы обычно не содержат специальных элементов управления – они предназначены для вывода текста. Текст на первой странице обычно поясняет назначение мастера, а текст на последней странице поздравляет пользователя с успешным решением задачи и, возможно, объясняет, что следует делать дальше. Первая и последняя страницы содержат поля для вывода текста и заголовков, а также некоторые свойства, позволяющие задать дополнительные элементы оформления этих страниц. Все промежуточные страницы мастера, которые, собственно, и реализуют его функциональность, включают в себя поле для вывода заголовка страницы и контейнер GtkVBox, в котором вы можете размещать любые визуальные элементы (**Рис. 2**).

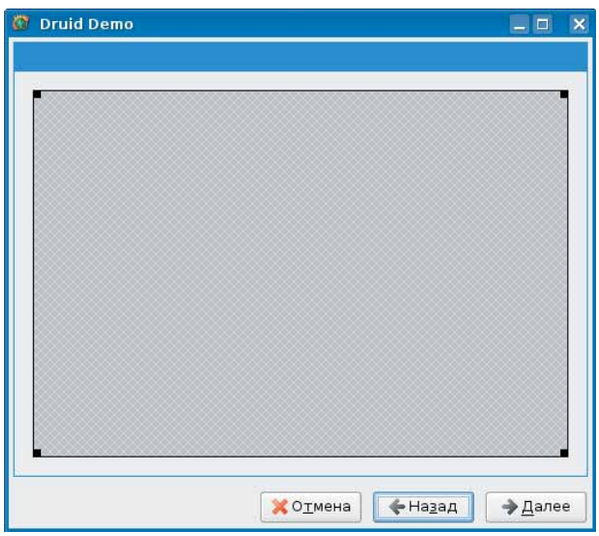


Рис. 2. Страница мастера в режиме редактирования.

Переключение между страницами мастера во время редактирования выполняется так же, как и во время выполнения, то есть с помощью кнопок Назад и Далее. Наш демонстрационный мастер содержит всего три страницы, то есть в нем будет только одна страница с элементами управления. Поместив компонент-мастер в главное окно приложения, мы можем исследовать, какие объекты были сгенерированы для него *Glade*. Прежде всего, это объект druid1 класса GnomeDruid. Первая страница мастера представлена структурой druidpagestart1 типа GnomeDruidPageEdge. Свойство Заголовок этого объекта позволяет указать заголовок начальной страницы мастера, в свойстве Текст мы вводим текст.

Помимо этих двух свойств, класс GnomeDruidPageEdge обладает и другими, позволяющими указать цвет шрифта и фона для заголовка и текста, а также отобразить эмблему мастера в правом верхнем углу страницы и фоновый узор (watermarks) – в верхней части страницы. Страница, на которой мы будем размещать элементы управления, представлена структурой druidpagestandard1 типа GnomeDruidPageStandard. На ней расположен контейнер druid-vbox1 типа GtkVBox. У структуры GnomeDruidPageStandard тот же набор свойств, что и у структуры GnomeDruidPageEdge, за исключением свойства Текст (вместо него страница отображает содержимое контейнера).

Месяц назад Мы разбирались с буфером обмена и виртуальной файловой системой GNOME.

штрихи

Последняя страница мастера, druidpagefinish1, является объектом типа GnomeDruidPageEdge, так же как и первая.

Помимо сигналов, унаследованных от своих предков, с объектами GnomeDruid связаны два специальных сигнала – help и cancel. Первый сигнал генерируется тогда, когда пользователь запрашивает справку; второй сигнал посылается приложению в результате щелчка по кнопке «Отмена», которая присутствует на каждой странице мастера. Мы назначаем обработчик сигнала cancel объекта druid1 (функция on_druid1_cancel()). Поскольку наша программа фактически состоит из одного мастера, будет вполне логично вызвать в нем функцию gtk_main_quit(). Объекты GnomeDruidPageEdge и GnomeDruidPageStandard, реализующие страницы мастера, обладают одинаковым набором дополнительных сигналов: back, next, cancel, finish и prepare.

Первые три сигнала посылаются приложению в ответ на щелчки по кнопкам Назад, Далее и Отмена соответственно. Сигнал prepare посылается приложению перед загрузкой соответствующей страницы, а сигнал finish – при щелчке по кнопке Применить, которая обычно расположена на последней странице мастера. Обратите внимание на то, что сигнал cancel есть как у объекта-мастера, так и у объекта-страницы. Зачем нужны два сигнала и как они взаимодействуют между собой? Если мы назначили обработчики обоим сигналам, то в ответ на щелчок кнопки Отмена на соответствующей странице будут, что вполне логично, вызваны оба обработчика. При этом сначала вызывается обработчик сигнала cancel объекта, реализующего страницу.

В то время как функция обработчика сигнал cancel объекта класса GnomeDruid возвращает значение void, функция обработчика cancel объекта страницы возвращает значение булевского типа. Если обработчик сигнала cancel объекта-страницы возвращает значение **TRUE**, обработчик сигнала cancel объекта класса GnomeDruid вызван не будет. Таким образом, назначая обработчик сигнала cancel объекта класса GnomeDruid и обработчик одноименного сигнала cancel для объекта страницы, мы можем запретить досрочное завершение работы мастера на соответствующей странице.

```
void on_druid1_cancel(GnomeDruid * gnomedruid, gpointer user_data)
{
  gtk_main_quit();
}

gboolean on_druidpagestandard1_cancel (GnomeDruidPage  *
gnomedruidpage,
GtkWidget * widget, gpointer user_data)
{
  return TRUE;
}
```

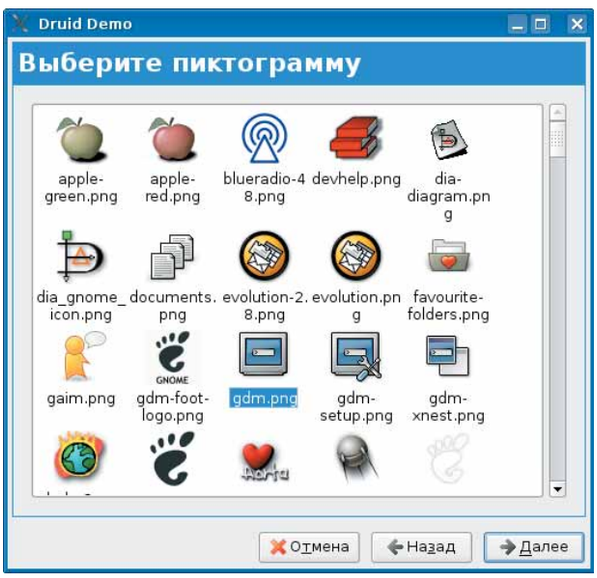
В нашем примере обработчик on_druidpagestandard1_cancel() запрещает завершение мастера на странице druidpagestandard1 (напоминаю характерную особенность интерфейса *GTK+*: когда обработчик сигнала должен запретить какие-либо дополнительные действия, он возвращает значение **TRUE**, в противном случае – значение **FALSE**). Впрочем, рассчитывать на этот метод запрета выхода из мастера особо не следует. Даже если обработчик сигнала cancel не позволит завершить работу мастера, пользователь все равно сможет сделать это, просто закрыв окно мастера с помощью кнопки «X», находящейся в его заголовке.

Сигнал finish посылается приложению тогда, когда пользователь щелкает кнопку Применить на странице GnomeDruidPageEdge. Как уже отмечалось, этот сигнал есть и у класса GnomeDruidPageStandard, однако при обычных условиях (без манипуляций с кнопками) промежуточная страница мастера этот сигнал не получит.

На странице druidpagestandard1 мы помещаем компонент GnomeIconSelection. Компонент GnomeIconSelection может быть очень красив, но в режиме редактирования мы увидим только чистое окно с полосой прокрутки. Во время выполнения программы мы должны указать компоненту источник пиктограмм для отображения и сделать отображаемые пиктограммы видимыми. Мы выполняем эти действия в обработчике сигнала prepare объекта druidpagestandard1:

```
void on_druidpagestandard1_prepare (GnomeDruidPage
*gnomedruidpage, GtkWidget * widget,
gpointer user_data)
{
 GnomeIconSelection * gis = lookup_widget(widget, “iconselection1”);
 gnome_icon_selection_add_defaults (gis);
 gnome_icon_selection_show_icons(gis);
}
```

Функция gnome_icon_selection_add_defaults() добавляет в список пиктограмм iconselection1 набор пиктограмм GNOME, заданных по умолчанию. Функция gnome_icon_selection_add_directory() позволяет добавить набор пиктограмм из произвольно заданной директории. Мы можем добавить в список GnomeIconSelection несколько наборов пиктограмм. Чтобы сделать пиктограммы видимыми, вызовем функцию gnome_icon_selection_show_icons(). В результате окно компонента заполняется симпатичными пиктограммами GNOME (**Рис. 3**).



› Рис. 3. Компонент *GnomeIconSelection* с загруженными пиктограммами.

Компонент GnomeIconSelection позволяет не только разглядывать пиктограммы, но и выбирать их. В обработчике сигнала next объекта druidpagestandard1 мы проверяем, выбрал ли пользователь пиктограмму:

```
gboolean on_druidpagestandard1_next (GnomeDruidPage *
gnomedruidpage,
```

»

```
GtkWidget * widget, gpointer user_data)
{
  GnomeIconSelection * gis = lookup_widget(widget, “iconselection1”);
  char * icon = gnome_icon_selection_get_icon(gis, TRUE);
  if (icon != NULL) {
     strncpy(selected_icon, icon, 255);
     gnome_icon_selection_clear (gis, TRUE);
     return FALSE;
  }
  return TRUE;
}
```

Функция gnome_icon_selection_get_icon() возвращает имя файла выбранной пиктограммы или NULL, если ни одна пиктограмма не выбрана. Второй параметр функции указывает, должно ли имя файла пиктограммы содержать полный путь. Строка, возвращенная функцией gnome_icon_selection_get_icon(), не выделяется специально для нас, и мы не должны пытаться ее высвободить. Если пользователь выбрал пиктограмму, мы копируем ее имя в переменную selected_icon.

Как и обработчик сигнала cancel, обработчик сигнала next позволяет разрешить или запретить последующие действия с помощью возвращаемого значения. Пока пиктограмма не выбрана, мы не разрешаем пользователю переход на следующую страницу мастера.

Обратите внимание на функцию gnome_icon_selection_clear(). Эта функция очищает список пиктограмм. Если мы не будем очищать список пиктограмм каждый раз, когда пользователь покидает страницу с компонентом GnomeIconSelection, при каждом возвращении пользователя на эту страницу в компонент будет добавляться новая копия того же самого набора пиктограмм. По этой причине мы должны вызвать функцию gnome_icon_selection_clear() также и в обработчике сигнала back.

Когда пользователь переходит на последнюю страницу мастера, у него появляется возможность щелкнуть кнопку Применить. В обработчике соответствующего сигнала finish мы назначаем выбранную пользователем пиктограмму главному окну приложения:

```
void on_druidpagefinish1_finish(GnomeDruidPage * gnomedruidpage,
    GtkWidget * widget, gpointer user_data)
{
        GtkWindow * wnd = lookup_widget(widget, “window1”);
        gtk_window_set_icon_from_file(wnd, selected_icon, NULL);
}
```

GTK+ плюс C++

Как отмечалось в самой первой статье этой серии, построение *GTK+* на основе C является определенным преимуществом, так как упрощает создание интерфейсов *GTK+* для других языков программирования. Однако вдумчивый читатель этих статей наверняка ловил себя на мысли, что объектно-ориентированная структура *GTK+* и GNOME API очень хорошо подходит для реализации на C++. Интерфейс C++ для *GTK+* и GNOME реализован в наборах библиотек *gtkmm* и *gnomemm*. Прежде чем приступать к разбору примеров *gtkmm*, убедитесь, что соответствующие пакеты установлены в вашей системе. Интерфейсы C++ для GNOME (*gnomemm*) не устанавливаются вместе с *gtkmm* по умолчанию. Их следует инсталлировать отдельно.

Рассмотрим простенькую программу, написанную с использованием *gtkmm* (файл **hellogtkmm.cpp** на диске):

```
#include <iostream>
#include <gtkmm.h>
class MainWindow : public Gtk::Window
{

        Gtk::Button m_button;
public:
        MainWindow(): m_button(“Здравствуй, gtkmm!”)
        {
                set_border_width(10);
                m_button.signal_clicked().connect(sigc::mem_
fun(*this,
                &MainWindow::on_button_clicked));
                add(m_button);
                m_button.show();
        }
        virtual ~MainWindow()
        {
        }
protected:
        virtual void on_button_clicked()
        {
                std::cout << “Button is pressed” << std::endl;
        }
};

int main(int argc, char *argv[])
{
        Gtk::Main prog(argc, argv);
        MainWindow window;
        window.set_title(“Превая программа gtkmm”);
        prog.run(window);
        return 0;
}
```

Эта программа делает то же, что и первая программа из первой статьи этой серии – создает окно с кнопкой (**Рис. 4**). Конечно, эта программа выглядит совсем не так эффектно, как предыдущая, написанная на «чистом» *GTK+*, но что уж тут поделать – изучение интерфейсов программирования развивается диалектично. Что дает нам использование интерфейса C++ для *GTK+*, написанного на C? Прежде всего – возможность более удобной группировки кода.

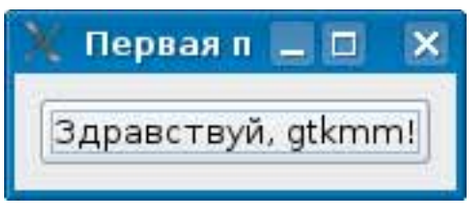


Рис. 4. Программа *hellogtkmm*.

Главное окно нашей программы реализовано в классе MainWindow, который является потомком класса Gtk::Window (пространство имен Gtk содержит классы, связанные с визуальными элементами *GTK+*). Кнопка в нашей программе реализована с помощью объекта m_button класса Gtk::Button. В конструкторе класса MainWindow мы прежде всего создаем объект m_button. В параметре конструктора Gtk::Button передается текст кнопки. Метод set_border_width(), унаследованный классом MainWindow от Gtk::Window, делает то же, что сделала бы функция gtk_container_set_border_width() для объекта GtkWindow.

Здесь мы сталкиваемся с общим правилом «перевода» C-интерфейсов *GTK+* в C++ интерфейсы *gtkmm*. Допустим, в интерфейсе *GTK+* определена функция gtk_somewidget_do_something(), оперирующая данными структуры GtkSomeWidget. В *gtkmm* этой функции соответствует метод do_something() класса Gtk::SomeWidget. По правилам интерфейса *GTK+*, первым аргументом функции gtk_somewidget_do_something() должен быть указатель на экземпляр структуры GtkSomeWidget (или ее потомка), для которого вызывается функция. Метод Gtk::SomeWidget::do_something(), если он не статический, тоже получает в качестве первого аргумента указатель на объект класса Gtk::SomeWidget (или его класса-потомка), с той разницей, что в C++ передача указателя на объект, для которого вызван метод, выполняется неявно, через переменную this, так что в заголовке метода do_something() данный указатель можно опустить.

Если структура GtkOtherWidget является потомком GtkSomeWidget

в иерархии *GTK+*, (как например, структура GtkWindow происходит от GtkContainer), то соответствующий ей класс Gtk::OtherWidget является потомком Gtk::SomeWidget и наследует все доступные методы последнего. Таким образом, метод add() (с помощью которого мы добавляем кнопку в окно) класса Gtk::Window унаследован этим классом от класса Gtk::Container, в котором он соответствует функции gtk_container_add(), а метод show() объекта Gtk::Button унаследован от класса Gtk::Widget и соответствует функции gtk_widget_show().

Пространство имен Gtk становится доступно нам благодаря включению в текст программы файла **gtkmm.h**. Этот заголовочный файл включает в себя все заголовочные файлы *gtkmm*. С одной стороны, это хорошо, так как мы можем не опасаться, что забыли включить в текст программы объявление какого-либо нужного класса. С другой стороны, при использовании **gtkmm.h** препроцессору придется обработать около мегабайта заголовочных файлов, большая часть из которых конкретной программе не нужна. Поэтому в более сложных программах рекомендуется вместо одного универсального **gtkmm.h** включать в исходные тексты специализированные файлы (**gtkmm/main.h**, **gtkmm/button.h** и т.д.).

Пропустим, пока что, остальную часть конструктора и перейдем к функции main(). Здесь настоящее объектно-ориентированное программирование проявляет себя во всей красе. Любая программа gtkmm должна начинаться с инициализации объекта класса Gtk::Main (этот объект должен быть объявлен внутри функции main(), а не глобально). У класса Gtk::Main несколько конструкторов, мы выбираем самый простой из них, тот, которому передаются переменные argc и argv. Как вы, конечно, уже догадались, в программе может быть только один объект класса Gtk::Main. Система *gtkmm* сама следит за тем, чтобы в программе не появилось второго объекта Gtk::Main, так что если вы попытаетесь создать еще один объект этого класса, вы все равно получите ссылку на тот объект, который вы создали в начале [данная конструкция часто называется «синглетон», – прим. ред.].

Далее мы создаем объект класса MainWindow. Все, что касается внутренностей главного окна приложения (как в смысле визуальных элементов, так и в смысле данных), инкапсулировано в этом объекте, так что в главной функции нам достаточно передать ссылку (не указатель!) на объект класса MainWindow методу run() объекта Gtk::Main. В результате главное окно программы будет выведено на экран и начнет обработку адресованных ему сообщений графической системы. Закрытие главного окна приведет к выходу из метода run() и завершению программы.

Обратите внимание на то, что в отличие от *GTK+*, в *gtkmm* цикл обработки сообщений завершается вместе с закрытием главного окна. В приведенном выше примере метод run() был вызван как метод объекта prog для того, чтобы сделать код более понятным. На самом деле этот метод статический. Благодаря тому, что система *gtkmm* всегда «знает», как найти созданный в программе объект Gtk::Main, мы можем обращаться к нему, используя статические методы. Например, вместо

```
prog.run(window);
```

можно написать

```
Gtk::Main::run(window);
```

Вызов window.set_title(), как вы уже знаете, эквивалентен gtk_window_set_title() в *GTK+*.

Вот мы и научились «переводить» программные конструкции с языка *GTK+* на язык *gtkmm*. Это нетрудно, поскольку почти каждой функции *GTK+* соответствует метод какого-либо класса *gtkmm*. Но что делать, если нам все же понадобится вызвать функцию *GTK+* для объекта *gtkmm*? У класса Gtk::Widget и всех его потомков есть метод gobj(), который возвращает указатель на соответствующую структуру *GTK+*. Например, метод Gtk::Button::gobj() возвращает значение типа GtkButton*, которое можно передавать функциям *GTK+*.

Перегруженная функция wrap() из пространства имен Glib выполняет обратную операцию, то есть создает на основе структуры *GTK+* объект класса *gtkmm*:

```
GtkButton  * b = GTK_BUTTON(gtk_button_new_with_label (“fine button”));
Gtk::Button * button = Glib::wrap(b);
```

Вернемся теперь к конструктору MainWindow. Как вы, наверное, заметили, мы пропустили код, связанный с назначением обработчика сигнала clicked кнопки m_button. Прежде чем мы перейдем к описанию механизмов обработки сигналов *gtkmm*, позвольте небольшое лирическое отступление. Простые идеи обработки событий, заложенные в Delphi, Borland C++Builder и C#, почему-то не в чести у разработчиков открытых наборов графических компонентов. Эти парни не любят простых путей! Вспомните систему сигналов и слотов в *Qt*, требующую применения макросов и специального препроцессора...

Разработчики *gtkmm* шагнули еще дальше и задействовали для обработки событий «тяжелую артиллерию» C++ – функторы, адаптеры и шаблоны. Все управление обработкой сигналов в *gtkmm* выполняет библиотека *libsigc*. Если вы хотите досконально разобраться в том, как создаются, живут и умирают обработчики сигналов, изучайте документацию к этой библиотеке (которая, кстати, не слишком хороша). Нижеизложенное является лишь мягким введением в вопрос.

У каждого класса *gtkmm*, реализующего визуальный элемент *GTK+*, есть набор методов, соответствующих сигналам, которые может генерировать данный визуальный элемент. Каждый из этих методов возвращает прокси-объект, созданный на основе шаблона sigc::signal, его можно использовать для назначения обработчика сигнала.

Метод signal_clicked() объекта m_button возвращает прокси-объект для назначения обработчика сигнала clicked.

Метод connect() прокси-объекта осуществляет фактическое связывание обработчика и сигнала. Аргументом метода connect() может быть объект-слот или объект-функция (называемая также «функтор»), который мы создаем с помощью адаптера sigc::mem_fun(). Этот адаптер создает объект-функцию из метода класса (первым аргументом адаптера является указатель this, вторым – адрес метода). Если бы обработчик сигнала был обычной функцией, нам следовало бы воспользоваться адаптером sigc::ptr_fun(). У метода connect() есть еще один параметр булевского типа, который позволяет указать, должен ли обработчик вызываться после стандартного обработчика (по умолчанию этому параметру присвоено значение **TRUE**).

В результате всех этих манипуляций метод on_button_clicked() становится обработчиком сигнала clicked объекта m_button. Рассмотренный механизм позволяет назначать сигналу несколько обработчиков.

В общем случае метод connect возвращает объект, описываемый страшной конструкцией sigc::signal<void,int>::iterator. Этот объект легко преобразовать к типу sigc::connection. С помощью метода disconnect() объекта класса sigc::connection мы можем удалить назначенный сигналу обработчик:

```
sigc::connection connection;
...
connection = m_button.signal_clicked().connect(sigc::mem_fun(*this,
                        &MainWindow::on_button_clicked));
...
connection.disconnect();
```

Если все это кажется вам слишком сложным, можете пойти другим путем. У класса Gtk::Button есть метод on_clicked(), объявленный в разделе protected. Вы можете перекрыть этот метод в потомке Gtk::Button (только не забудьте вызвать метод базового класса, иначе результаты могут оказаться несколько неожиданными). При таком подходе вам, конечно, придется создавать собственный класс для каждой кнопки вашего приложения [думается, что отчасти поэтому разработчики *GTK+* и *Qt* и решили не идти «путем Delphi», – прим. ред.].

Ну вот, пожалуй, и все, что можно было сказать о *gtkmm* в рамках одной статьи. Если вас интересует, как скомпилировать программу-пример командой в одну строку, то вот она, команда, использующая утилиту pkg-config:

```
g++ hellogtkmm.cpp -o hellogtkmm `pkg-config gtkmm-2.4 --cflags --libs`
```

На этом я завершаю увлекательное путешествие в мир *GTK+* и гномов и благодарю всех, кто не потерялся в пути. Встретимся вновь, изучая другие графические инструментарии. LXF