Anjuta: Создаем

ЧАСТЬ 7 Калькулятор, самая необходимая деталь рабочего стола, у нас уже есть – теперь настало время подумать и о текстовом редакторе. Берегись, Блокнот – **Андрей Боровский** запускает *Anjuta*!

– Разве имя должно что-то значить? – проговорила Алиса с сомнением.
– Конечно, должно, – ответил Шалтай-Болтай и фыркнул. – Возьмем, к примеру, мое имя – оно выражает мою суть!

Льюис Кэрролл, Алиса в Зазеркалье

В прошлой статье мы научились создавать программы GNOME своими руками, начиная, как говорится, с пустого файла. На этот раз мы автоматизируем процесс создания «заготовки» для нашей программы с помощью среды разработки *Anjuta* (в русском переводе, естественно, *Анюта*). Вот что написано о происхождении этого имени на сайте разработчиков *Anjuta* (**anjuta.sourceforge.net**): «[Это имя] не обозначает ничего, Наба Кумар (Naba Kumar) начал работу над проектом *Anjuta* и назвал его в честь своей девушки, так как программа посвящена именно ей». Для полного прояснения вопроса осталось узнать, что имели в виду родители, когда назвали свою дочь *Anjuta*...

Интегрированную среду *Anjuta* можно рассматривать как аналог *KDevelop* для *GTK+*. Помимо приложений *GTK+* и GNOME, *Anjuta* позволяет генерировать заготовки простых консольных программ, программ *wxWindows* (так это название пишется в настройках программы; ныне данный инструментарий называется *wxWidgets*), а также проекты «чистых» приложений X-Window, использующих для построения интерфейса только *Xlib*. В качестве языка программирования можно выбрать C или C++. Как и *KDevelop*, *Anjuta* не обладает собственными средствами визуального программирования – для этих целей используется уже знакомый нам *Glade*.

В этой статье мы детально исследуем процесс написания простенького текстового редактора GNOME средствами тандема *Anjuta* и *Glade*. Запустите программу *Anjuta*. По умолчанию, она открывает диалоговое окно (**Рис. 1**), в котором мы можем выбрать режим работы с *Anjuta*.

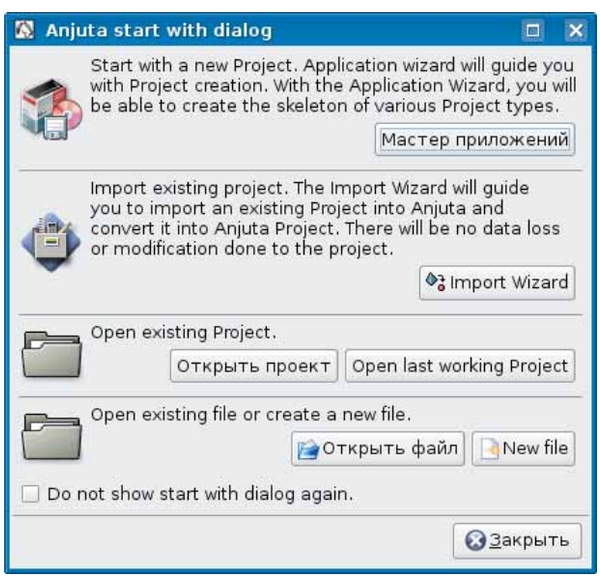


Рис. 1. Окно выбора режима работы *Anjuta*.

Щелкните по кнопке Мастер приложений. Перед нами предстает первое окно Мастера приложений *Anjuta*.

Запустить Мастер приложений можно также с помощью команды меню Файл–>Новый проект... (команда Новый проект... расположена, почему-то в меню Файл», а не в меню Проект). В Мастере приложений нажмите кнопку Далее – вы перейдете к окну, в котором предлагается указать тип проекта (**Рис. 2**). Выберите проект GNOME 2.0. В ответ на щелчок по кнопке Далее открывается окно, в котором следует ввести название нового проекта, версию, имя автора и язык разработки (здесь следует выбрать C: если указать C++, вы лишитесь функции автоматической генерации кода с помощью *Glade*).

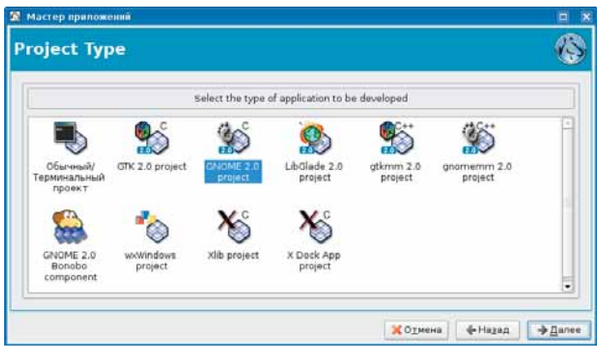


Рис. 2. Окно выбора типа проекта *Anjuta*.

Еще одно важное окно Мастера приложений (**Рис. 3**) позволяет нам определить дополнительные параметры нового проекта – нужно ли добавлять текст лицензии GPL во все файлы исходных текстов и добавлять ли код, предназначенный для работы с *gettext*. Даже если вы не собираетесь выполнять интернационализацию своего приложения в обозримом будущем, ее поддержку все равно стоит включить, просто на всякий случай. Окно дополнительных параметров проекта позволит вам активировать режим генерации исходных текстов с помощью *Glade* (в *Glade 2.x* это все еще возможно), что и следует сделать. В этом же окне Мастера приложений можно выбрать пиктограмму для нового приложения и указать, нуждается ли оно для своей работы в открытом окне консоли. На этом генерация нового проекта заканчивается.

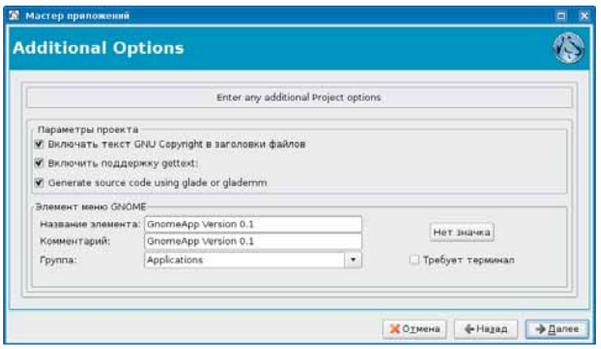
В левой части главного окна *Anjuta* расположено окно менеджера проектов. На вкладке «Проект» этого окна перечислены файлы исходных текстов приложения. Помимо файла **main.c** мы видим здесь файлы **callbacks.c**, **callbacks.h**, **interface.c**, **interface.h**, **support.c** и **support.h**. Если имена этих файлов показались вам знакомыми, интуиция вас не подвела – это файлы проекта *Glade* (мы недаром установили флажок Generate source code using Glade or Glademm в Мастере приложений). Теперь можно приступать к созданию интерфейса нашей программы.

Макет интерфейса

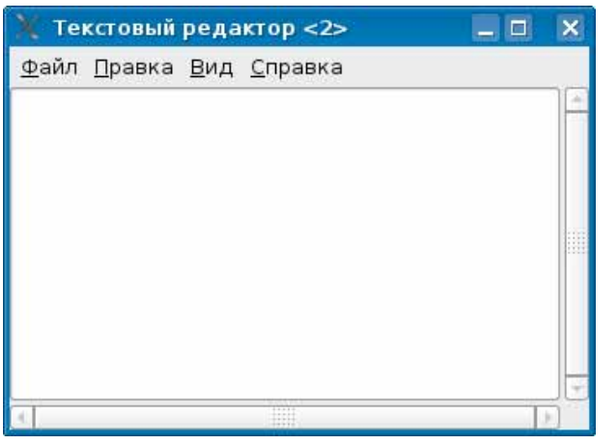
Команда меню Проект–>Редактировать графический интерфейс приложения... запускает экземпляр *Glade* для нашего проекта. При совмес-

Месяц назад Мы написали наше первое приложение GNOME – не используя при этом строку «Hello world».

текстовый редактор



› **Рис. 3. Окно дополнительных параметров проекта.**



› **Рис. 4. Форма главного окна приложения.**

тном использовании *Anjuta* и *Glade* следует учитывать, что эти программы слабо взаимодействуют между собой. Для того чтобы изменения, внесенные в проект с помощью *Glade*, стали известны *Anjuta*, нужно выполнить сохранение проекта в *Glade* (и не забывайте нажать кнопку Построить, чтобы *Glade* мог отразить в исходном коде изменения, внесенные в визуальном редакторе). Фактически, при работе с *Glade* нам почти всегда придется сохранять проект по два раза – один раз в *Glade*, другой раз – в *Anjuta*.

Получив доступ к окну визуального редактора *Glade*, мы можем воспользоваться нашими навыками проектирования интерфейсов *GTK+*. Откройте форму главного окна приложения (для этого, напомню, нужно дважды щелкнуть мышью по имени объекта-окна в главном окне *Glade*). Перейдите в редактор свойств *Glade* и назначьте свойству «Заголовок» объекта window1 строку «Текстовый редактор». Прежде чем добавлять в форму окна приложения GNOME какие-либо элементы управления, в нем следует разместить объект-контейнер. Мы воспользуемся контейнером типа «вертикальный бокс» с двумя строками. В верхнюю строку контейнера поместим визуальный элемент «Главное меню» (объект menubar1 класса GtkMenuBar). В нижнюю строку помещаем визуальный элемент «Просмотр текста» (объект textview1 класса GtkTextView). Теперь форма главного окна выглядит как настоящее окно текстового редактора (**Рис. 4**). Обратите внимание на то, что главное меню уже заполнено стандартными командами, такими как открытие и сохранение файла, копирование и вставка из буфера обмена и т.п. После добавления главного меню в форму окна приложения в файле **callbacks.c** появятся заготовки для обработчиков всех сигналов, генерируемых стандартными командами меню. Имена обработчиков сигналов команд меню создаются по шаблону on_menuitem_activate(), где menuitem – имя объекта, представляющего команду меню.

Выход здесь

Заполнение заготовки программы кодом мы начинаем, как и всегда, с кода завершения приложения. С помощью редактора свойств *Glade* назначьте обработчик сигналу destroy объекта window1 и добавьте в него вызов функции gtkmain_quit() (мы проделывали эту операцию уже много раз, так что останавливаться на ней подробно не будем). Приступим теперь к написанию обработчиков стандартных команд меню. Найдите обработчик on_quit1_activate() в файле **callbacks.c**. Он реагирует на сигнал, посылаемый командой меню **Файл | Выход**. Мы можем просто добавить в этот обработчик вызов on_window1_destroy():

```
void on_quit1_activate(GtkMenuItem *menuitem,  gpointer user_data)
{
  on_window1_destroy(NULL, NULL);
}
```

Теперь вызов команды меню **Файл | Выход** приводит к завершению работы программы. Почему мы просто не добавили в обработчик on_quit1_activate() вызов функции gtkmain_quit()? Дело в том, что приложение GNOME (как и любое оконное приложение) может быть завершено несколькими разными командами. Удобно сделать так, чтобы при завершении работы приложения всегда вызвалась одна и та же функция (мы возлагаем эту роль на функцию on_window1_destroy()). В результате, если при завершении программы нам понадобится выполнять какие-то специальные действия (например, сохранять несохраненные данные), соответствующий код нужно будет добавить только в функцию on_window1_destroy(). Вместо непосредственного вызова функции on_window1_destroy() мы могли бы вызвать функцию gtk_widget_destroy(), передав ей в качестве аргумента указатель на объект window1. В следующей статье этой серии мы подробнее обсудим методы корректного завершения работы программы-редактора (в процессе вызова функции gtk_widget_destroy() генерируется сигнал destroy для соответствующего визуального элемента).

Теперь мы можем скомпилировать программу (сделайте это с помощью команды меню *Anjuta* Сборка–>Сборка). Перед тем, как выполнять компиляцию, следует сохранить все изменения, сделанные в исходном коде. Полные исходные тексты приложения-примера к этой статье вы найдете на диске. В процессе подготовки дистрибутива из него были удалены файлы сценариев сборки (не только make-файл, но и сценарий configure). При попытке выполнить команду сборки для этого дистрибутива вам будет сообщено об отсутствии make-файла. Для того, чтобы собрать программу из пакета, в котором отсутствуют файлы сценариев сборки, нужно сначала скомандовать Сборка–>Подготовить файлы для сборки....

Скажем несколько слов о редакторе исходных текстов *Anjuta*. В случае возникновения ошибки во время сборки редактор *Anjuta* выделяет подчеркиванием строку исходного текста, вызвавшую проблемы. В редакторе также присутствует рудиментарная функция автоматического завершения кода. В процессе ввода идентификатора, являющегося параметром функции, появляется список всех идентификаторов с подходящими именами, однако контекст при этом не учитывается, так что в списке идентификаторов, начинающихся на букву m, может оказаться функция main(). Если в текст программы добавляется вызов функции, определенной в файле проекта, редактор выводит подсказку со списком ее аргументов (к сожалению, она не работает для API- »

» функций *GTK+*/GNOME). Во всем остальном редактор исходных текстов *Anjuta* похож на *KWrite*.

Объекты GtkTextView и GtkTextBuffer

Теперь, когда мы научили программу корректно завершать свою работу, давайте заполним остальные обработчики сигналов меню. Начнем с обработчика сигнала, посылаемого командой **Файл | Новый** (функция on_new1_activate()). Прежде чем описывать команды, манипулирующие текстом, необходимо разобраться, как работает объект GtkTextView, который является основой нашего текстового редактора. Объект GtkTextView и связанный с ним объект GtkTextBuffer представляют собой чрезвычайно мощные средства, позволяющие создавать полноценные текстовые редакторы, поддерживающие разметку текста, вставку изображений, гиперссылок, сложную навигацию по тексту и т.п. В нашей программе мы воспользуемся лишь незначительной частью их возможностей. Заинтересованные читатели смогут узнать о них больше из справочной системы *GTK+*.

Всю сложную механику работы с текстом выполняет класс GtkTextBuffer. Объекты GtkTextBuffer содержат текст и вспомогательную информацию, необходимую для его отображения. Функции, работающие с GtkTextBuffer, позволяют выполнять различные операции с текстом. Объекты GtkTextView играют вспомогательную роль и служат, в основном, для отображения содержимого GtkTextBuffer на экране. Когда мы создаем экземпляр объекта GtkTextView, по умолчанию создается и экземпляр GtkTextBuffer. Однако текстовый буфер может существовать и отдельно от конкретного объекта GtkTextView. Один текстовый буфер может принадлежать нескольким объектам GtkTextView. В сложной программе, предназначенной для манипулирования текстом, вы можете даже создать несколько текстовых буферов для одного визуального элемента GtkTextView (несколько буферов можно использовать, например, для хранения промежуточных результатов редактирования текста) и делать их видимыми по мере необходимости.

Таким образом, для того, чтобы работать с текстом в нашем редакторе, мы должны получить доступ к текстовому буферу объекта GtkTextView. Вот как можно сделать это в обработчике on_new1_activate():

```
void on_new1_activate  (GtkMenuItem * menuitem, gpointer user_data)
{
   GtkTextView * textview1 = lookup_widget(menuitem, “textview1”);
   GtkTextBuffer * textbuffer = gtk_text_view_get_buffer(textview1);
   gtk_text_buffer_set_text(textbuffer, “”, 0);
   filename[0] = 0;
}
```

С помощью знакомой нам функции lookup_widget() (напомню, что ее, для нашего удобства, генерирует *Glade*) мы получаем указатель на объект GtkTextView, заданный своим именем. Далее мы получаем указатель на объект GtkTextBuffer с помощью функции gtk_text_view_get_buffer(). Задача нашего обработчика – очистить текстовый буфер. Для этого мы вызываем функцию gtk_text_buffer_set_text(). Эта функция, вообще говоря, предназначена для записи в буфер текста. Ее первым аргументом является указатель на объект GtkTextBuffer, второй аргумент представляет собой строку текста в кодировке UTF-8 (единственная кодировка, которую «признает» объект GtkTextBuffer). Третьим аргументом функции gtk_text_buffer_set_text() должна быть длина строки (в байтах, а не в символах). Мы записываем в буфер строку нулевой длины, что эквивалентно его очистке. Переменная filename объявлена глобально в файле **callbacks.c** как

```
static char filename[256];
```

и содержит текущее имя файла. В обработчике on_new1_activate() мы сбрасываем имя файла (если таковое было установлено). Следует отметить, что перед очисткой текстового буфера наш простой редактор не проверяет, содержит ли он несохраненные данные. Реализация такой проверки оставляется читателю в качестве домашнего упражнения.

В обработчике on_open1_activate(), который вызывается в ответ на сигнал, сгенерированный командой меню «Файл|Открыть», мы используем диалоговое окно GtkFileChooserDialog для выбора файла точно так же, как мы делали это в LXF90. Останавливаться подробно на коде этого обработчика мы не будем. Фактическая загрузка данных из выбранного файла выполняется с помощью определенной нами функции open_file(), которая, напротив, заслуживает более внимательного рассмотрения.

```
void open_file(GtkWidget * parentwindow, char * file_name)
{
        int fd, len;
        char * buf[1024];
        GtkTextView * textview1 = lookup_widget(parentwindow,
“textview1”);
        GtkTextBuffer * textbuffer = gtk_text_view_get_
buffer(textview1);
        if ((fd = open(file_name, O_RDONLY, 0)) < 0)
        {
                GtkWidget * dialog;
                dialog = gtk_message_dialog_new(parentwindow,
GTK_DIALOG_DESTROY_WITH_PARENT,
GTK_MESSAGE_ERROR, GTK_BUTTONS_CLOSE,
“Невозможно открыть файл %s”, file_name);
                gtk_dialog_run (GTK_DIALOG (dialog));
                gtk_widget_destroy (dialog);
                return;
        }
        strncpy(filename, file_name, 256);
        gtk_text_buffer_set_text(textbuffer, “”, 0);
        while((len = read(fd, buf, 1024)) != 0)
                gtk_text_buffer_insert_at_cursor(textbuffer, buf,
len);
        close(fd);
}
```

Функция open_file() принимает два параметра. В первом передается указатель на объект, представляющий главное окно приложения. Он нужен нам, во-первых, для того, чтобы найти объект textview1, а во-вторых – чтобы вывести предупреждающее диалоговое окно в случае ошибки. Во втором параметре передается имя загружаемого файла.

В функции open_file() мы получаем указатель на текстовый буфер тем же способом, что и в функции on_new1_activate(). Далее мы пытаемся открыть файл, имя которого передано в параметре file_name, для чтения. Если открыть файл не удалось, выводится модальное диалоговое окно с информацией об ошибке.

Для вывода информирующих и предупреждающих сообщений в *GTK+* используется диалоговое окно GtkMessageDialog. Мы создаем экземпляр этого окна с помощью функции gtk_message_dialog_new(). Флаг GTK_DIALOG_DESTROY_WITH_PARENT указывает, что в случае уничтожения родительского визуального элемента диалоговое окно должно быть уничтожено вместе с ним. Флаг GTK_MESSAGE_ERROR указывает тип диалогового окна (окно с сообщением об ошибке). Для создания диалоговых окон других типов можно использовать флаги GTK_MESSAGE_INFO (нейтральное сообщение) GTK_MESSAGE_WARNING (предупреждение), GTK_MESSAGE_QUESTION (вопрос) и GTK_MESSAGE_OTHER (диалоговое окно неопределенного типа). Флаг GTK_BUTTONS_CLOSE указывает, что создаваемое диалоговое окно должно содержать одну кнопку, закрывающую окно. Если бы мы использовали

Еще раз о справке GTK+

Коль скоро речь зашла о справочной системе, нельзя не упомянуть о том, что браузер справки *Devhelp* не очень удобен. Страницы справки GTK+ обычно содержат довольно много текста, а в *Devhelp* (версии 0.1) отсутствует средство поиска текста на странице. Поскольку справка *GTK+* хранится в формате HTML, ее страницы можно читать непосредственно из каталога **/opt/gnome/share/gtk-doc/html**, однако при этом вы лишаетесь удобной возможности глобального поиска. Получить новейшую версию справки по *GTK+* можно на сайте **developer.gnome.org/doc/API**.

флаг GTK_BUTTONS_NONE, созданное нами диалоговое окно вообще не содержало бы кнопок. Флаг GTK_BUTTONS_OK_CANCEL позволяет создать окно с кнопками **OK** и **Cancel**. В справочной системе *GTK+* можно найти константы и для других комбинаций кнопок диалогового окна. Следующим параметром функции gtk_message_dialog_new() является строка форматирования текста, выводимого в диалоговом окне. Она может содержать форматирующие символы, точно так же, как и форматирующая строка функции printf(). Далее следуют аргументы, на которые ссылается форматирующая строка. Диалоговое окно становится видимым сразу же после вызова функции gtk_message_dialog_new(). По умолчанию функция gtk_message_dialog_new() создает немодальное окно. Хотя модальное окно можно создать, установив флаг GTK_DIALOG_MODAL, здесь мы поступаем иначе. Для того чтобы окно стало модальным (то есть, блокировало работу программы), мы вызываем функцию gtk_dialog_run(), которая приостанавливает выполнение программы до тех пор, пока не будет нажата одна из кнопок диалогового окна. Функция gtk_dialog_run() возвращает численное значение, соответствующее нажатой кнопке (у нашего окна только одна кнопка, так что значение, возвращенное gtk_dialog_run(), нас не интересует). После того как окно закрыто, мы уничтожаем соответствующий ему объект с помощью функции gtk_widget_destroy(). Если вы хотите вывести на экран немодальное окно и при этом получить информацию о том, какой кнопкой оно было закрыто, вы должны назначить обработчик сигнала response для объекта-окна.

После того, как мы убедились, что файл открыт, мы опустошаем текстовый буфер и начинаем заполнять его данными из файла. Функция gtk_text_buffer_insert_at_cursor() добавляет очередной блок текста в позиции курсора (по умолчанию – в конец текста). Таким образом, мы можем добавлять текст в буфер небольшими порциями (этот способ не самый изящный, но самый простой).

Функция on_save_as1_activate() обрабатывает сигнал, который создает команда меню **Файл | Сохранить как**. В этой функции мы также используем диалоговое окно GtkFileChooserDialog, только теперь оно настроено на выбор имени файла для сохранения. Обратите внимание на вызов функции gtk_file_chooser_set_do_overwrite_confirmation(). Будучи вызвана с аргументом TRUE, эта функция настраивает диалоговое окно выбора имени файла таким образом, что если файл с выбранным именем уже существует, программа спросит нас, действительно ли мы хотим перезаписать его. Сохранение данных в файле на диске выполняется с помощью функции save_file().

```
void save_file(GtkWidget * parentwindow, char * file_name)
{
        int fd, len;
        GtkTextIter start, end;
        gchar * buf;
        GtkTextView * textview1 = lookup_widget(parentwindow,
"textview1");
        GtkTextBuffer * textbuffer = gtk_text_view_get_
buffer(textview1);
        if ((fd = open(file_name, O_WRONLY|O_CREAT|O_TRUNC,
0666)) < 0)
        {
                GtkWidget * dialog;
                dialog = gtk_message_dialog_new(parentwindow,
GTK_DIALOG_DESTROY_WITH_PARENT,
GTK_MESSAGE_ERROR, GTK_BUTTONS_CLOSE,
"Невозможно открыть файл %s", file_name);
                gtk_dialog_run (GTK_DIALOG (dialog));
                gtk_widget_destroy (dialog);
                return;
        }
        gtk_text_buffer_get_start_iter(textbuffer, &start);
        gtk_text_buffer_get_end_iter(textbuffer, &end);
        buf = gtk_text_buffer_get_text(textbuffer, &start, &end,
FALSE);
        write(fd, buf, strlen(buf));
        close(fd);
        g_free(buf);
}
```

Список аргументов у этой функции такой же, как и у open_file(). Открытие файла (теперь, естественно, для записи) и проверка результата этой операции, с выводом диалогового окна в случае ошибки, выполняются практически так же, как и раньше. Самое интересное начинается потом. Мы получаем доступ к текстовому буферу объекта GtkTextView и должны извлечь из него текстовые данные для записи в файл. Получение текста из текстового буфера – совсем не такое простое дело, как может показаться на первый взгляд. Поскольку текстовый буфер GtkTextBuffer обладает чрезвычайно широкими возможностями в плане манипуляции текстом, он наделен довольно сложным интерфейсом. Интерфейс GtkTextBuffer упрощает выполнение сложных операций, но делает простые операции несколько более трудоемкими, по сравнению с аналогичными операциями в менее функциональных компонентах. Ключевую роль при обработке текста, содержащегося в буфере GtkTextBuffer, играет понятие итератора (их можно сравнить с итераторами в библиотеке шаблонов C++). Итератор текстового буфера *GTK+* представляет собой структуру типа GtkTextIter, которая указывает на некоторую позицию в тексте. Итераторы могут перемещаться по тексту в обоих направлениях в соответствии с заданным критерием. Например, с помощью итератора можно перейти к последнему символу текущей строки, найти следующее вхождение определенного символа или подстроки и т.п. Итераторы играют важную роль в поиске и навигации по тексту, но этим их функции не ограничиваются. Пара итераторов позволяет выделить фрагмент текста для дальнейших манипуляций с ним. Практически все функции текстового буфера, предназначенные для работы с фрагментами текста, используют итераторы.

Для того чтобы скопировать текст из буфера, мы определяем два итератора – start и end, которые должны указывать, соответственно, на начало и конец данных в буфере. Обратите внимание на то, что итераторы следует объявлять как переменные типа GtkTextIter, а не как указатели на этот тип, хотя функциям, работающим с итераторами, передаются именно указатели на соответствующие переменные. Функция gtk_text_buffer_get_start_iter() устанавливает переданный ей итератор в начале текста, содержащегося в буфере, а функция gtk_text_buffer_get_end_iter() – соответственно, в конце текста. Получив два итератора, указывающих на некоторый фрагмент текста (в нашем случае – на весь текст), мы можем скопировать его с помощью функции gtk_text_buffer_get_text(). Первым аргументом этой функции должен быть, как всегда, указатель на объект GtkTextBuffer. Второй и третий аргументы gtk_text_buffer_get_text() являются адресами переменных-итераторов, выделяющих фрагмент, который должен быть скопирован. Последний аргумент типа gboolean указывает, следует ли включать в копируемый текст невидимые символы (в нашем простеньком редакторе таковые отсутствуют). Функция gtk_text_buffer_get_text() возвращает строку с нулевым конечным символом, в которой содержится скопированный фрагмент текста. Для этой строки выделяется специальная область памяти, и по окончании работы с фрагментом текста строку нужно удалить с помощью функции g_free().

Если вы внимательно изучите прилагаемые к статье исходные тексты, то увидите, что мы пропустили некоторые функции нашего текстового редактора (например, работу с буфером обмена). Ему, а также другим интересным функциям *GTK+* и GNOME будет посвящена следующая статья. LXF

Через месяц Мы рассмотрим буфер обмена и другие «продвинутые» возможности GNOME.