Глава 16

Написание локализованных приложений

©Перевод сделан Тюрюмовым А.Н.

Если вы сделаете свое приложение не только мультиплатформенным, но и многоязычным, то вы многократно повышаете его шансы на успех. В этой главе рассказывается, как сделать свое приложение многоязычным (internationalization). В литературе вы часто можете встретить сокращение от этого слова — «i18n» (как вы можете заметить в английском написании этого слова между буквами i и n как раз 18 символов).

16.1 Введение в интернационализацию

Если вы хотите выпустить свое приложение на международный рынок, то первое о чем необходимо подумать — о его переводе. Вы должны предоставить перевод для всех строк вашего приложения на каждом иностранном языке, который вы хотите поддерживать. В библиотеке wxWidgets это делается с помощью создания каталогов сообщений и загрузки их через класс `wxLocale`. Данная технология отличаться от использования таблиц со строками, с которой вы, возможно, уже знакомы. Вместо того, чтобы ссылаться на каждую строку с помощью числа-идентификатора и переключать таблицы, каталоги сообщений пытаются перевести строку, используя подходящий каталог (хотя вы можете вашу собственную систему для перевода, но необходимо помнить, что система локализации в библиотеке wxWidgets базируется на каталогах).

Без каталога сообщений программа будет использовать непереведенные строки из исходного кода для отображения текста в меню, кнопках и т.д. Если ваш собственный язык содержит символы не из кодировки ASCII (например, акценты), то вам возможно придется создать отдельный «перевод» (каталог сообщений) для своего языка, так как исходный код должен содержать только ASCII символы.

Отображение текста на различных языках включает в себя также различные кодировки символов. Это означает, что символ с одним и тем же кодом может представлять различные символы при выводе на экран. Вы должны быть уверены, что приложение может правильно установить кодировку, используемую элементами графического интерфейса и в ваших файлах с данными, а также чтобы в каждом случае использовалась правильная кодировка и о преобразовании между кодировками.

Еще одним нюансом интернационализации является представление чисел, даты и

времени. Заметим, что представление может отличаться даже в рамках одного языка. Например, число, представляемое в англичанами как 1,234.56 в Германии имеет вид 1.234,56, а в немецко-говорящей части Швейцарии представляется как 1'234.56. В США десятое ноября представляется в виде 11/10, тогда как для англичанина такая запись означает одиннадцатое октября. Как вы увидите в дальнейшем библиотека wxWidgets сможет помочь и в этом случае.

Перевод строк может быть длиннее, чем их английский вариант, а это значит, что элементы окна должны уметь адаптироваться для различных размеров. Сайзеры (см. Главу 7 «Построение окон с использованием сайзеров») являются наиболее подходящими для решения данной проблемы. Другой проблемой является то, что для западных языков чтение осуществляется слева направо, тогда как для других языков (например, для арабского) чтение идет справа налево, что означает, что все окно должно сменить свою ориентацию. В данный момент в библиотеке отсутствует специальный механизм для такой смены.

Последней группой элементов, которые необходимо адаптировать для различных языков, являются изображения и звуки. Это может понадобится, например, если вы пишите приложение, хранящее номера телефонов, и хотите что бы оно могло проговаривать номер (который является зависимым от языка), а также хотите отображать различные изображения в зависимости от языка.

16.2 Перевод

wxWidgets предоставляет готовый механизм для перевода сообщений, используя класс `wxLocale` и сама по себе полностью переведена на несколько языков. На домашней странице wxWidgets можно найти самые свежие переводы сообщений библиотеки.

Реализация интернационализации в wxWidgets очень похожа на реализацию ее в пакете GNU gettext. Библиотека использует каталоги сообщений, которые бинарно совместимы с каталогами gettext, позволяя использовать многие инструменты из пакета gettext. Однако при запуске интернациональных приложений никаких дополнительных приложений не требуется, так как wxWidgets может самостоятельно читать каталоги сообщений.

Однако во время разработки программы вам возможно потребуется установить пакет gettext для работы с каталогами сообщений (или установить poEdit, описание которого можно найти в следующем разделе). Существует два типа каталогов сообщений: исходный каталог, который представляет из себя текстовый файл с расширением .po, и бинарный каталог, который создается из исходного каталога с использованием программы msgfmt (является частью пакета gettext), имеющего расширение .mo. Во время выполнения программы нужны только бинарные файлы. Для каждого языка, который вы хотите поддерживать требуется один каталог сообщений.

16.2.1 poEdit

Вам необязательно использовать консольные программы для работы с каталогом сообщений. Вацлав Славик (Vaclav Slavik) написал замечательный графический инструмент poEdit (`http://www.poedit.org`), который является интерфейсом к пакету

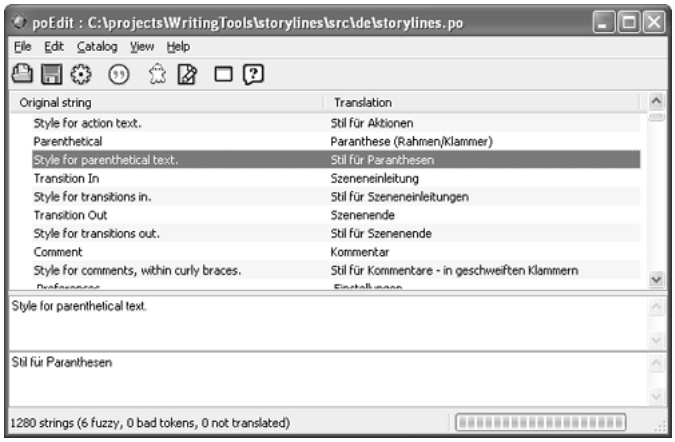

Рис. 16.1: poEdit

gettext. Программа poEdit (изображена на рис. 16.1) поможет вам получить каталог сообщений, сгенерировать .mo-файлы и эффективно управлять процессом перевода вашего приложения по мере его написания.

16.2.2 Пошаговое руководство по использованию каталогов сообщений

Вот список шагов, которые нужно сделать, чтобы создать и использовать каталоги сообщений:

1. Обернуть все текстовые строки в программе, которые необходимо перевести, в функцию `wxGetTranslation` или эквивалентный ей макрос `_()`. Строки, которые переводить не нужно должны быть обернуты в `wxT()` (или эквивалентный ему макрос `_T()`), чтобы сделать такие строки совместимыми с юникодом.

2. Извлечь все требующие перевода строки из исходного кода в .po-файл. Вам не нужно делать все это вручную, вы можете использовать для этого программу xgettext или, что проще, программу poEdit. При использовании xgettext можно использовать ключ -k, чтобы `wxGettranslation` распознавался как `_()`. В программе poEdit эта возможность включается с помощью диалога настроек каталога.

3. Перевести все извлеченные на прошлом шаге строки на другой язык, редактируя .po-файл или используя poEdit (один .po-файл на один язык). Вы также должны указать кодировку, которая используется для переведенных строк. Если вы не используете poEdit, то должны сделать это вручную, используя ваш любимый текстовый редактор. Заголовок вашего .po-файла должен выглядеть следующим образом:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
```

```
#
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 1999-02-19 16:03+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=iso8859-1\n"
"Content-Transfer-Encoding: 8bit\n"
```

Обратите внимание на вторую с конца строку в которой определяется кодировка, используемая каталогом. Все строки в каталоге должны использовать эту кодировку. Это очень важно если используется неюникодная кодировка, так как в противном случае графические элементы не смогут корректно отобразить все символы.

4. Скомпилировать .po-файл в бинарный .mo-файл, который сможет использовать программа. Это можно сделать с помощью программы poEdit, или вы возможно захотите добавить этот шаг в ваш скрипт компиляции приложения, например так:

```
msgfmt -o myapp.mo myapp.po
```

5. Установить подходящую локаль в вашей программе для использования строк для заданного языка (см. следующую раздел «Использование wxLocale»)

В системе Mac OS X вам необходимо сделать одну модификацию в файле Info.plist, который описывает содержимое «связки приложения». Этот файл (который является текстовым файлом XML в кодировке UTF-8) должен содержать запись `CFBundleDevelopmentRegion`, описывающую язык разработчика (например, English для английского) и Mac OS X опрашивает приложение на предмет присутствия некоторых каталогов с ресурсами, чтобы определить какой язык поддерживается. Например, для немецкого это должен быть каталог German.lproj. Так как приложение wxWidgets не использует эти каталоги для хранения информации о ресурсах, то приложение вместо хранения перевода в .mo-файлах должно точно сообщать о том какой язык поддерживается. Вы можете это сделать добавив запись `CFBundleLocalizations` в свой файл Info.plist. Тогда он будет выглядеть следующим образом:

```
<key>CFBundleDevelopmentRegion</key>
<string>English</string>
<key>CFBundleLocalizations</key>
<array>
       <string>en</string>
       <string>de</string>
       <string>fr</string>
</array>
```

16.2.3 Использование wxLocale

Класс `wxLocale` инкапсулирует все языко-зависимые настройки и обобщает концепцию локали языка C. Обычно вы добавляете переменную класса `wxLocale` в класс вашего приложения, например `m_locale`, и в методе `OnInit` инициализируете локаль с помощью кода, указанного ниже:

```
if (m_locale.Init(wxLANGUAGE_DEFAULT,
                  wxLOCALE_LOAD_DEFAULT | wxLOCALE_CONV_ENCODING))
{
    m_locale.AddCatalog(wxT("myapp"));
}
```

Заметим, что `wxLocale::Init` пытается найти и добавить каталог wxstd.mo, содержащий перевод стандартных сообщений библиотеки wxWidgets. Параметр `wxLANGUAGE_DEFAULT` означает, что необходимо использовать язык системы. Вы также можете обязать программу использовать какой-либо определенный язык, используя правильный код `wxLANGUAGE_xxx`.

Когда вы говорите классу `wxLocale` загрузить каталог сообщений, то каталог преобразовывается в кодировку, используемую операционной системой пользователя. Это стандартное поведение класса `wxLocale`. Вы можете отключить его, если передадите `wxLOCALE_CONV_ENCODING` в качестве второго параметра методу `wxLocale::Init`.

Где же wxWidgets ищет каталог сообщений? Для каждого каталога <DIR> во внутреннем списке библиотека просматривает каталоги:

- <DIR>/<LANG>/LC_MESSAGES
- <DIR>/<LANG>
- <DIR>

Правила формирования списка каталогов различные для каждой платформы:

- На всех платформах используется значение переменной окружения `LC_PATH`.
- В системах Unix и Mac OS X используется каталог с библиотекой wxWidgets, /share/locale, /usr/share/locale, /usr/lib/locale, /usr/locale, /share/locale, а также текущий каталог.
- В системе Windows используется каталог программы.

Вы можете добавить дополнительные каталоги, используя метод `wxLocale::AddCatalogLookupPathPrefix`. Например:

```
wxString resDir = GetAppDir() + wxFILE_SEP_PATH + wxT("resources");
m_locale.AddCatalogLookupPathPrefix(resDir);

// Если resDir это c:\MyApp\resources, то AddCatalog будет искать
// каталог для французкого языка в следующих каталогах
// (в дополнение к стандартным путям):
//
```

```
// c:\MyApp\resources\fr\LC_MESSAGES\myapp.mo
// c:\MyApp\resources\fr\myapp.mo
// c:\MyApp\resources\myapp.mo

m_locale.AddCatalog(wxT("myapp"));
```

Стандартный метод создания каталога сообщений — это создание подкаталога для каждого языка (используя каноническое имя языка в качестве имени каталога), содержащих файл <appname>.mo в каждом каталоге. Например, стандартный пример internat из дистрибутива wxWidgets содержит каталоги *fr* и *de*, представляющие локализацию для французского и немецкого языка соответственно. Аббревиатуры для языков соответствуют кодам из ISO 639.

16.3 Кодировка символов и юникод

На Земле существует так много языков, что их все невозможно уместить в 256 значений байта, который обычно используется в классическом 8-битовом представлении для символов. Поэтому чтобы иметь возможность отобразить более 256 различных символов было введено понятие кодировки символов или просто кодировки (более правильное решение — юникод будет представлено далее в этом разделе).

Поэтому символ имеющий значение 161 зависит от текущей кодировки. В кодировке ISO 8859-1 (Latin-1) это символ ¡(перевернутый восклицательный знак). В ISO 8859-2 (Latin-2) этот же код представляет знак ¥ (Aogonek).

Когда вы пишите текст в окне система обязана знать об используемой кодировке. Это называется «кодировка шрифта», хотя означает только используемую кодировку для символов. Создание шрифта без явного определения кодировки означает «использовать кодировку по-умолчанию». Это поведение подходит в большинстве случаев, так как пользователь обычно использует систему на своем родном языке.

Но если вы знаете, что какой-то символ использует другую кодировки, например, ISO 8859-2, то вы должны создать подходящий шрифт:

```
wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
              false, wxT("Arial"), wxFONTENCODING_ISO8859_2);
```

Иначе вы не сможете правильно отобразить символ в системах с иной кодировкой, например с ISO 8859-1. Обратите внимание, что возможны ситуации, когда оптимальная кодировка не доступна. В этом случае вы можете попробовать использовать альтернативную кодировку, и если таковая доступна, то преобразовать строку в эту кодировку. Следующий код показывает такую последовательность действий: строка в кодировке *enc* отображается с помощью шрифта *facename*. Использование механизма преобразований `wxCSConv` будет рассмотрено далее.

```
// Есть строка в кодировке 'enc', которую мы хотим отобразить
// шрифтом с именем 'facename'.
//
// Сначала мы должны найти какие шрифты доступны для
// отображения данной кодировки
```

```
wxString text; // Содержит текст в кодировке 'enc'
if (!wxFontMapper::Get()->IsEncodingAvailable(enc, facename))
{
   // У нас нет кодировки 'enc' доступной в данном шрифте.
   // Какая альтернативная кодировка доступна?
   wxFontEncoding alternative;
   if (wxFontMapper::Get()->GetAltForEncoding(enc, &alternative,
                                              facename, false))
   {
       // У нас есть шрифт в альтернативной кодировке, поэтому
       // мы должны преобразовать нашу строку в эту кодировку.
       wxCSConv convFrom(wxFontMapper::GetEncodingName(enc));
       wxCSConv convTo(wxFontMapper::GetEncodingName(alternative));
       text = wxString(text.wc_str(convFrom), convTo) ;

       // Создаем шрифт в альтернативной кодировке
       wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
               false, facename , alternative);
       dc.SetFont(myFont);
   }
   else
   {
      // Не можем преобразовать. Пытаемся использовать преобразование
      // с потерями в кодировку ISO 8859-1 (7-битовый ASCII)
      wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
              false, facename, wxFONTENCODING_ISO8859_1);
      dc.SetFont(myFont);
    }
}
else
{
     // Если заданная кодировка есть в шрифте, то проблем нет
     wxFont myFont(10, wxFONTFAMILY_DEFAULT, wxNORMAL, wxNORMAL,
               false, facename, enc);
     dc.SetFont(myFont);
}

// В конце пишем текст в созданном шрифте.
dc.DrawText(text, 100, 100);
```

16.3.1 Преобразование данных

В прошлом примере нам потребовалось много кода, чтобы преобразовать строку из одной кодировки в другую. Есть два пути достигнуть того же самого. Первый — использование `wxEncodingConverter` является устаревшим и не должен использоваться в новом коде. А пока ваш компилятор явно не поддерживает тип `wchar_t`, то

вам необходимо использовать преобразователи кодировок (wxCSConv, базовый класс wxMBConv).

wxEncodingConverter

Этот класс поддерживает только ограниченное подмножество кодировок, но если ваш компилятор не распознает тип wchar_t, то это единственное доступное для вас решение. Например,

```
wxEncodingConverter converter(enc, alternative, wxCONVERT_SUBSTITUTE);
text = converter.Convert(text);
```

Флаг wxCONVERT_SUBSTITUTE говорит о том, что библиотека должна попробовать использовать преобразования с потерями в случае, если не сможет точно преобразовать какие-нибудь символы. Это, например, означает, что акцентированные заглавные буквы будут заменены обычными, различные штрихи могут быть заменены символом «-» и т.п.

wxCSConv (wxMBConv)

Юникод решает множество рассмотренных ранее проблем, используя 16- или даже 32-битные большие символы (wchar_t) для хранения символов в «глобальной кодировке». Это означает, что вы не связываетесь с кодировками до тех пор, пока вам не понадобится читать или писать данные в старом 8-битном формате, который, как мы уже знаем, не содержит всей необходимой информации, чтобы определить кодировку.

Даже когда вы не компилируете wxWidgets в режиме юникода (где тип wchar_t используется для хранения символов строк), вы можете использовать эти широкие символы для осуществления преобразования когда это возможно. Вы преобразовываете из одной кодировки в строки с широкими символами, а далее преобразовываете результат в другую кодировку. Этот метод используется в классе wxString для осуществления подходящего преобразования. Но не забывайте, что в неюникодных билдах тип wxString использует 8-битовые символы для хранения символов и ничего не знает о кодировке содержимого.

Для преобразования wxString в массив широких символов вы можете использовать метод wxString::wc_str, которая получает многобайтовый конвертер в качестве единственного параметра. Данный параметр говорит в неюникодном билде какую кодировку имеет строка, но игнорируется в юникодном билде, так как строка уже использует для хранения широкие символы.

В юникодных билдах вы можете далее строить строку прямо из этих символов. Однако, в неюникодных билдах вы должны явно указать кодировку строки-назначения. Поэтому в следующем примере второй параметр convTo игнорируется в юникодных билдах.

```
text = wxString(text.wc_str(convFrom), convTo);
```

Кодировка символов предлагает больше возможностей, чем кодировка шрифта, поэтому вы можете преобразовать кодировку шрифта в кодировку символов, используя

```
wxFontMapper::GetEncodingName(fontencoding);
```

Это означает, что наша прошлая задача может быть переписана используя кодировку символов:

```
wxCSConv convFrom(wxFontMapper::GetEncodingName(enc));
wxCSConv convTo(wxFontMapper::GetEncodingName(alternative));
text = wxString(text.wc_str(convFrom) , convTo) ;
```

Существуют ситуации, когда вы можете вывести 8-битовые данные напрямую, без использования `wxString`. Данную задачу можно выполнить, используя экземпляр объекта `wxCharBuffer`, поэтому последнюю строку можно переписать, например, так

```
wxCharBuffer output = convTo.cWC2MB(text.wc_str(convFrom));
```

И если вашими входными данными является не строка, а просто 8-битовые данные (`wxCharBuffer` как в примере ниже), то вы можете написать:

```
wxCharBuffer output = convTo.cWC2MB(convFrom.cMB2WC(input));
```

В библиотеке также существуют статические объекты для преобразований. Например, доступен объект `wxConvISO8859_1`, а также указатель `wxConvCurrent` на преобразователь, который использует текущую локаль библиотеки C. Также существуют подклассы `wxMBConv`, которые оптимизированы для конкретных задач кодировки: `wxMBConvUTF7`, `wxMBConvUTF8`, `wxMBConvUTF16LE/BE` и `wxMBConvUTF32LE/BE`. Последние два являются синонимами для `wxMBConvUTF16/32`, которые используют родной для машины порядок байтов. Для получения большей информации обратитесь к разделу «wxMBConv Classes Overview» документации wxWidgets.

Преобразование без временного буфера

Как мы только что узнали классы преобразований позволяют вам легко трансформировать строку из одной кодировки в другую. Однако, результат операции для данных преобразователей возвращает новую строку, либо временный буфер. В этих функциях мы должны выполнить преобразование и сохранить результат для последующей обработки, что обычно выполняется с помощью копирования результата преобразования в отдельную строку.

Предположим, что мы хотим пересылать строки между компьютерами, например, через сокеты. В таком случае необходимо в начале обмена договорится о том какую кодировку использовать для пересылки, иначе на платформах с различными кодировками мы получим неверный вывод. Поэтому посыльный должен преобразовать строку к UTF-8, а получатель должен далее преобразовать UTF-8 в свою кодировку.

Следующий короткий пример демонстрирует как использовать комбинацию технологий, чтобы преобразовать строку в любой кодировке в UTF-8, сохранить результат в `char*` для пересылки по сети, а далее преобразовать «сырые» данные из UTF-8 в `wxString`.

```
// Преобразовать строку в UTF-8
const wxCharBuffer ConvertToUTF8(wxString anyString)
{
    return wxConvUTF8.cWC2MB( anyString.wc_str(*wxConvCurrent) ) ;
}

// Использовать сырые UTF-8 данные для построения wxString
wxString ConvertFromUTF8(const char* rawUTF8)
{
    return wxString(wxConvUTF8.cMB2WC(rawUTF8), *wxConvCurrent);
}

// Тестируем наше преобразование wxString<->UTF-8
void StringConversionTest(wxString anyString)
{
    // Преобразуем в UTF-8 и создаем ссылку на буфер
    const wxCharBuffer bUTF8 = ConvertToUTF8(anyString);

    // wxCharBuffer имеет неявный оператор преобразования для char *
    const char *cUTF8 = bUTF8 ;

    // Пересоздаем строку
    wxString stringCopy = ConvertFromUTF8(cUTF8);

    // Две строки должны быть равны
    wxASSERT(anyString == stringCopy);
}
```

16.3.2 Файлы помощи

Возможно вы захотите сделать отдельные файлы помощи для каждого языка. Тогда диалог показа помощи может выбрать имя файла помощи в соответствии с текущей локалью, возможно используя функцию `wxLocale::GetName`, чтобы получить такое имя, или просто используя макрос `_()` для преобразования имени файла в правильное значение. Например,

```
m_helpController->Initialize(_("help_english"));
```

Если вы используете `wxHtmlHelpController`, то должны убедиться, что все HTML-файлы содержат специальный тег META. Например,

```
<meta http-equiv="Content-Type" content="text/html; charset=iso8859 //2">
```

Также убедитесь, что файлы проекта (с расширением HHP) содержат дополнительную строку в секции OPTIONS:

```
Charset=iso8859-2
```

Эта дополнительная запись указывает контроллеру HTML-помощи какая кодировка используется для содержимого и таблиц с индексами.

16.4 Числа и даты

Текущая локаль используется при форматировании чисел и дат. Аналог функции `printf` в классе `wxString` заботится об этом автоматически. Например, в следующем примере

```
wxString::Format(wxT("%.1f") , myDouble);
```

метод `Format` использует правильный десятичный разделитель.

Аналогично и при представлении дат в следующем коде

```
wxDateTime t = wxDateTime::Now();
wxString str = t.Format();
```

Метод `Format` представляет текущую дату и время в правильном языке и формате. Обратитесь к документации, чтобы ознакомится с различными флагами, которые можно передавать в метод форматирования даты, но не забудьте, что возможно вам надо сделать строку даты переводимой, так как последовательность различных частей даты различается в разных языках.

Если вы хотите добавить правильный раделитель для тысяч или просто хотите знать правильный символ для десятичной точки, то вы можете запросить у объекта локали соответствующую строку, используя метод `GetInfo`:

```
wxString info = m_locale.GetInfo(wxLOCALE_THOUSANDS_SEP,
                                 wxLOCALE_CAT_NUMBER);
```

16.5 Другие ресурсы

Можно загружать другие данные (например, изображения и звуки) в зависимости от локали. При этом используется тот же принцип как и в случае текста. Например,

```
wxBitmap bitmap(_("flag.png"));
```

Этот код приводит к тому, что строка «flag.png» появится в списке строк для перевода, поэтому вам достаточно перевести строку «flag.png» в подходящее для вашей платформы имя файла (например в «de/flag.png»). Удостоверьтесь, что указанный в локализации файл доступен. Также локализованные ресурсы можно загружать из сжатого архива (обратитесь к Главе 14 «Файлы и потоки»).

16.6 Простой пример

Чтобы проиллюстрировать рассмотренные концепции на CD-ROM помещен маленький пример examples/chap16. Эта программа показывает несколько строк и флаг страны для трех языков: английского, французского и немецкого. Вы можете изменить язык из меню «File». При смене языка меняется язык меню, строки `wxStaticText` и графическое изображение флага. Для демонстрации различий в поведении макросов `_()` и `wxT()` помощь к пунктам меню в строке статуса остается на английском языке.

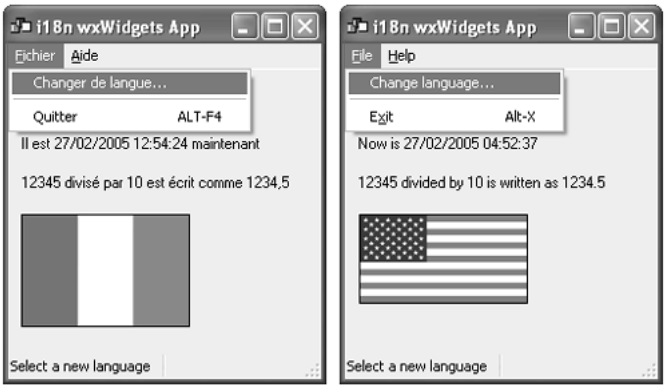

Рис. 16.2: Пример локализации

Класс приложения для данного примера содержит указатель на объект класса `wxLocale`. Метод `SelectLanguage` пересоздает в случае необходимости объект локали с требуемым языком. Далее представлено объявление и реализация класса приложения для нашего примера:

```
class MyApp : public wxApp
{
public:
    ~MyApp() ;

    // Инициализируем приложение
    virtual bool OnInit();

    // Пересоздаем m_locale, в соответствии с выбраным языком
    void SelectLanguage(int lang);

private:
    wxLocale* m_locale; // текущая локаль
};

IMPLEMENT_APP(MyApp)

bool MyApp::OnInit()
{
    wxImage::AddHandler( new wxPNGHandler );
    m_locale = NULL;
    SelectLanguage( wxLANGUAGE_DEFAULT );

    MyFrame *frame = new MyFrame(_("i18n wxWidgets App"));

    frame->Show(true);
    return true;
}
```

```
void MyApp::SelectLanguage(int lang)
{
    delete m_locale;
    m_locale = new wxLocale( lang );
    m_locale->AddCatalog( wxT("i18n") );
}

MyApp::~MyApp()
{
    delete m_locale;
}
```

В классе фрейма есть два интересных метода: `SetupStrings` и `OnChangeLanguage`. `SetupStrings` устанавливает метки и пересоздает меню, используя перевод для всех строк, кроме строк помощи к пунктам меню:

```
void MyFrame::SetupStrings()
{
    m_helloString->SetLabel(_("Welcome to International Sample"));
    m_todayString->SetLabel( wxString::Format(_("Now is %s"),
                             wxDateTime::Now().Format().c_str()) );
    m_thousandString->SetLabel( wxString::Format(
              _("12345 divided by 10 is written as %.1f"), 1234.5) );
    m_flag->SetBitmap(wxBitmap( _("flag.png") , wxBITMAP_TYPE_PNG ));

    // create a menu bar
    wxMenu *menuFile = new wxMenu;

    // the "About" item should be in the help menu
    wxMenu *helpMenu = new wxMenu;
    helpMenu->Append(wxID_ABOUT, _("&About...\tF1"),
                     wxT("Show about dialog"));

    menuFile->Append(wxID_NEW, _("Change language..."),
                     wxT("Select a new language"));
    menuFile->AppendSeparator();
    menuFile->Append(wxID_EXIT, _("E&xit\tAlt-X"),
                     wxT("Quit this program"));

    wxMenuBar *menuBar = new wxMenuBar();
    menuBar->Append(menuFile, _("&File"));
    menuBar->Append(helpMenu, _("&Help"));

    wxMenuBar* formerMenuBar = GetMenuBar();
    SetMenuBar(menuBar);
    delete formerMenuBar;
```

```
    SetStatusText(_("Welcome to wxWidgets!"));
}
```

Метод `OnChangeLanguage` вызывается, если пользователь хочет установить другой язык. Данный метод устанавливает правильный идентификатор для выбранного языка, например, такой как `wxLANGUAGE_GERMAN`. Этот идентификатор передается в `MyApp::SelectLanguage` перед тем как вызывается `SetupStrings` для того, чтобы изменить метки и изображение флага. Реализация `OnChangeLanguage` выглядит так:

```
void MyFrame::OnChangeLanguage(wxCommandEvent& event)
{
    wxArrayInt languageCodes;
    wxArrayString languageNames;

    languageCodes.Add(wxLANGUAGE_GERMAN);
    languageNames.Add(_("German"));

    languageCodes.Add(wxLANGUAGE_FRENCH);
    languageNames.Add(_("French"));

    languageCodes.Add(wxLANGUAGE_ENGLISH);
    languageNames.Add(_("English"));

    int lang = wxGetSingleChoiceIndex( _("Select language:"),
                            _("Language"), languageNames );

    if ( lang != -1 )
    {
        wxGetApp().SelectLanguage(languageCodes[lang]);
        SetupStrings();
    }
}
```

16.7 Итоги

В данной главе были рассмотрены различные пути реализации перевода в библиотеке wxWidgets, а также затронуты вопросы оформления даты и времени, валюты и т.п. Заметим, что при переводе лучше всегда работать с тем, кто знает требуемый язык или локаль, чтобы учесть все нюансы локализации.

Другой пример локализованного приложения вы можете найти в каталоге samples/internat библиотеки wxWidgets. Этот пример показывает перевод строк меню и диалогов для десяти языков.

В следующей главе мы узнаем как выполнять несколько задач одновременно с помощью многопоточного программирования.