Глава 3

Обработка сообщений

В этой главе разъясняются основные принципы построения управляемых сообщениями приложений. Рассказывается о том, как генерируются сообщения, как приложение может перехватить их с помощью таблицы сообщений и про методы обработки сообщений. Мы также обсудим присоединяемые (динамические) обработчики сообщений, а также опишем методику создания собственных классов, типов и макросов для сообщений.

3.1 Управляемые сообщениями приложения

Когда произошел первый релиз Apple Macintosh программисты были поражены насколько эта ОС отличается от всех остальных, используемых в то время. Передвигая указатель от одного окна к другому, используя полосы прокрутки, меню, текстовые элементы управления и тому подобное было сложно представить как работает код, реализующий все это. Казалось, что множество различных действий происходят параллельно, но это является лишь иллюзией. Для большинства людей Macintosh стал первым введением в мир программ, управляемых сообщениями.

Все современные GUI-приложения управляются сообщениями. Это означает, что приложение находится в цикле, ожидая прихода нового сообщения от пользователя или из другого источника (например, это происходит когда окно нуждается в перерисовке или ждет сообщений от сокетов). После прихода нового сообщения приложение его распаковывает и выполняет подходящую функцию, которая обрабатывает полученное сообщение. Хотя и кажется, что окна обновляются одновременно, но большинство графических приложений не является многопоточными, а поэтому скорость их реакции зависит от скорости выполнения других приложений. Особенно это заметно, когда что-то сильно замедляет компьютер. Тогда вы сможете увидеть, как медленно прорисовывается каждое окно, одно за другим.

Существующие мультиплатформенные библиотеки по-разному представляют обработку сообщений для программиста. Как будет показано далее, основным методом обработки сообщений в wxWidgets является использование таблиц сообщений.

3.2 Таблицы и обработчики сообщений

Обработка сообщений в wxWidgets организована более гибко, чем это возможно при использовании для этих целей виртуальных функций (в последнем случае пришлось бы объявить все возможные обработчики сообщений в базовом классе, что является не практичным и не эффективным решением).

Каждый класс который наследуется от `wxEvtHandler`, включая классы фреймов, меню и даже классы документов, может содержать таблицу сообщений, которая указывает каким образом происходит обработка сообщений. Все классы окон (являющиеся наследниками от `wxWindow`) и классы приложений являются потомками от `wxEvtHandler`.

Для создания статической таблицы сообщений (она создается в момент компиляции) вам необходимо:

1. Объявить новый класс, который явно или неявно будет являться потомком от `wxEvtHandler`.

2. Добавить в класс метод для каждого сообщения, которое будет обрабатываться.

3. Объявить таблицу сообщений в классе с помощью макроса `DECLARE_EVENT_TABLE`.

4. Реализовать таблицу сообщений в файле с реализацией класса с использованием конструкции `BEGIN_EVENT_TABLE ... END_EVENT_TABLE`.

5. Добавить соответствующие записи в таблицу сообщений (например, `EVT_BUTTON`), тем самым создавая связь между каждым сообщением и соответствующим ему методом класса.

Все функции обработки сообщений имеют одинаковую форму. Они возвращают `void`, не являются виртуальными и принимают в качестве аргумента один объект, описывающий сообщение. Если вы знакомы с MFC, то наверняка заметите отличия, так как в MFC не существует унифицированного вида для обработчика сообщений. Тип аргумента меняется в соответствии с типом обрабатываемого сообщения. Например, обработчики меню и сообщений стандартных элементов управления используют для этого класс `wxCommandEvent`. Сообщение о размере (которое приходит в случае изменения размера окна программой или пользователем) использует класс `wxSizeEvent`. Каждый тип сообщения может использовать свой класс, который можно использовать, чтобы выяснить какие именно изменения произошли в элементе управления (например, изменилось значение в элементе редактирования). В самом простом случае (таком как нажатие на клавишу) часто просто игнорирую значение в этом объекте.

Добавим в пример из прошлой главы обработку изменения размера фрейма и нажатия на кнопку OK. Тогда объявление класса, обрабатывающего сообщения, будет похоже на следующее:

```
// Объявляем наш класс для фрейма
class MyFrame : public wxFrame
{
```

```
public:
    // Констуктор
    MyFrame(const wxString& title);
    // Обработчики сообщения
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);
    void OnSize(wxSizeEvent& event);
    void OnButtonOK(wxCommandEvent& event);
private:
    // Этот класс обрабатывает сообщения
    DECLARE_EVENT_TABLE()
};
```

Код, добавляющий пункты в меню, очень похож на код из прошлой главы, а добавление кнопки «OK» будет выглядеть так:

```
wxButton* button = new wxButton(this, wxID_OK, wxT("OK"),
                                wxPoint(200, 200));
```

Далее следует код таблицы сообщений, который позволяет фрейму перехватывать сообщения от меню, кнопки и при изменении размера окна.

```
// Таблица сообщений для MyFrame
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
    EVT_MENU     (wxID_ABOUT,     MyFrame::OnAbout)
    EVT_MENU     (wxID_EXIT,      MyFrame::OnQuit)
    EVT_SIZE     (                MyFrame::OnSize)
    EVT_BUTTON   (wxID_OK,        MyFrame::OnButtonOK)
END_EVENT_TABLE()
```

Когда пользователь в меню выбирает пункт About[1] или Quit[2], происходит посылка сообщения фрейму. Таблица сообщений `MyFrame` сообщает wxWidgets, что сообщение от меню с идентификатором `wxID_ABOUT` должно быть послано в функцию `MyFrame::OnAbout`, а сообщение с идентификатором `wxID_EXIT` должно быть послано `MyFrame::OnQuit`. Другими словами происходит вызов этих функций с единственным параметром (в нашем случае объектом типа `wxCommandEvent`), когда петля сообщений обрабатывает соответствующее сообщение. Макрос `EVT_SIZE` не берет идентификатор в качестве параметра, так как это сообщение может обрабатываться только объектом, который его сгенерировал.

Запись с `EVT_BUTTON` приводит к тому, что функция `OnButtonOK` вызывается, когда нажимается кнопка с идентификатором `wxID_OK` где-нибудь в иерархии окон фрейма. Этот пример показывает, что сообщение может обрабатываться окном, не являющимся источником этого сообщения. Давайте предположим, что кнопка является дочерним окном `MyFrame`'а. Когда нажимается кнопка wxWidgets ищет в классе `wxButton` подходящий обработчик для сообщения. Если ни один не найден, то

[1] О программе (англ.)
[2] Выход (англ.)

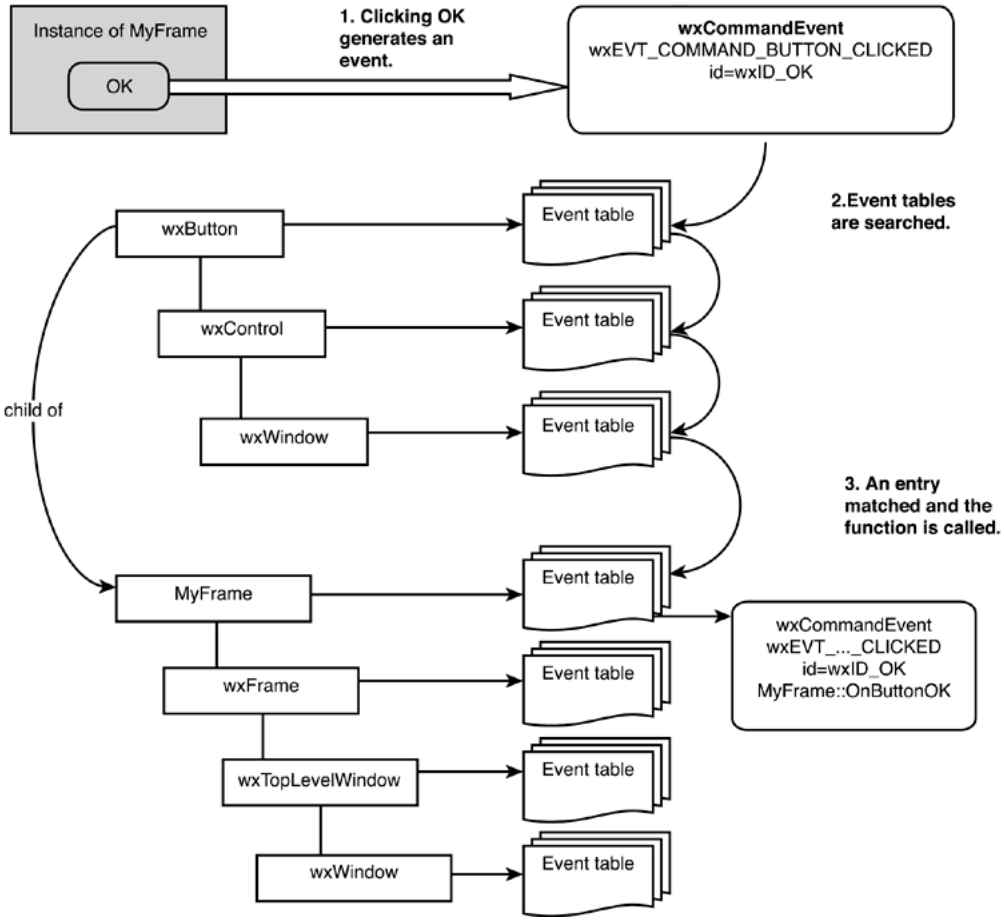

Рис. 3.1: Обработка сообщения при нажатии на кнопку

проверяется родительское окно (в нашем случае это экземпляр класса `MyFrame`. Соответствующая запись есть в его таблице сообщений, поэтому вызывается функция `MyFrame::OnButtonOK`. Поиск происходит как по иерархии оконных компонент, так и по иерархии наследования. Это означает, что программист может выбирать место обработки сообщений. Например, если вы хотите разработать диалог, который должен выполнять некоторые действия на некоторую команду (например, `wxID_OK`), но вам хочется дать возможность создания элементов управления другому программисту, использующему ваш код, то вы все еще можете определить поведение по умолчанию для элементов управления, если у них будут ожидаемые идентификаторы.

Генерация события в ответ на нажатия кнопки и дальнейший поиск подходящего обработчика в таблице событий проиллюстрировано на рис. 3.1. Показана иерархия, состоящая из двух классов: `wxButton` и `MyFrame`. Каждый класс имеет свою собственную таблицу сообщений, каждая из которых может содержать запись, обрабатывающую сообщение. Когда пользователь нажимает кнопку «OK» создается новый объект класса `wxCommandEvent`, который содержит идентификатор (`wxID_OK`) и тип сообщения (`wxEVT_COMMAND_BUTTON_CLICKED`). Далее с помощью метода `wxEvtHandler::ProcessEvent` просматривается вся таблица сообщений. Сначала просматривается таблица `wxButton`, далее `wxControl`, далее `wxWindow`. Если запись, соответствующая заданному типу сообщения и идентификатору, не найдена,

то wxWidgets начинает поиск такой таблицы в родительском окне кнопки. Она находит удовлетворяющую условиям запись:

```
EVT_BUTTON (wxID_OK, MyFrame::OnButtonOK)
```

поэтому вызывается функция `MyFrame::OnButtonOK`. Обратите внимание, что только командные сообщения (то есть те сообщения, которые явно или неявно наследуются от `wxCommandEvent`) рекурсивно передаются по цепочке к родительскому окну. Так как это обстоятельство часто вызывает недоумение для пользователей приведем список системных сообщений, которые не передаются родительским обработчикам сообщений: `wxActivateEvent`, `wxCloseEvent`, `wxEraseEvent`, `wxFocusEvent`, `wxKeyEvent`, `wxIdleEvent`, `wxInitDialogEvent`, `wxJoystickEvent`, `wxMenuEvent`, `wxMouseEvent`, `wxMoveEvent`, `wxPaintEvent`, `wxQueryLayoutInfoEvent`, `wxSizeEvent`, `wxScrollWinEvent` и `wxSysColourChangedEvent`. Эти сообщения не распространяются, так как эти сообщения имеют значение только для данного конкретного окна. Например, посылка сообщения о необходимости перерисовки дочернего окна не важна для родительского окна.

3.3 Игнорирование сообщения

Система обработки сообщений в wxWidgets реализует нечто похожее на виртуальные методы обычного C++, а это означает, что возможно переопределить поведение класса, перегружая его таблицу обработки сообщений. В большинстве случаев даже возможно изменение поведения «родных» элементов управления. Например, возможно фильтровать выбранные нажатия на клавишу, которые система посылает текстовому элементу управления с помощью создания наследника от `wxTextCtrl` и определения обработчика для нажатий клавиш, используя `EVT_KEY_DOWN`. Этого вполне достаточно, чтобы перехватить события от любой клавиши, посылаемые родному элементу управления, что делает не совсем то, что нам нужно. В этом случае обработчик сообщения может вызвать метод `wxEvent::Skip`, чтобы показать, что поиск для обработчика сообщений должен быть продолжен. Таким образом, вместо того, чтобы непосредственно вызвать метод базового класса (как вы бы сделали в случае использования виртуальных функция языка C++), вы просто должны вызвать метод `Skip` у вашего объекта сообщения.

Для примера реализуем текстовый элемент управления, который принимает только символы от «a» до «z» и от «A» до «Z»:

```
void MyTextCtrl::OnChar(wxKeyEvent& event)
{
    if ( wxIsalpha( event.KeyCode() ) )
    {
       // Клавиша допустима, поэтому обрабатываем как обычно
       event.Skip();
    }
    else
    {
       // Недопустимый символ. Мы не вызываем event.Skip(),
```

```
        // поэтому сообщение не обрабатывается нигде более
        wxBell();
    }
}
```

3.4 Подключаемые обработчики сообщений

Вместо того, чтобы наследовать новый класс от класса окна для обработки сообщений вы можете поступить проще. Создайте новый класс-наследник от `wxEvtHandler`, определите в нем соответствующую таблицу сообщений, а потом вызовете `wxWindow::PushEventHandler` для добавления этого объекта в стек обработчиков сообщений для данного окна. Теперь ваш новый обработчик сообщений перехватывает все приходящие окну сообщению, и если они в нем не обрабатываются, то ищется следующий объект в стеке и так далее. Метод `wxWindow::PopEventHandler` используется, чтобы извлечь обработчик сообщений с вершины стека. Если передать этому методу `true`, то извлеченный объект будет уничтожен.

Используя указанный прием вы можете избежать многочисленных наследований классов и теоретически использовать одинаковый обработчик для обработки сообщений от экземпляров различных классов.

Обычно значение возвращаемое `wxWindow::GetEventHandler` является указателем на само окно, однако при использовании `PushEventHandler` это естественно будет не так. Даже если вы просто хотите вручную вызвать обработчик сообщения для окна всегда используйте функцию `GetEventHandler`, чтобы получить самый верхний обработчик в стеке и используйте это значение, чтобы быть уверенным в корректной обработке вашего сообщения.

Использование `PushEventHandler` может помочь временно или навсегда изменить поведение интерфейса приложения. Например, вы можете захотеть реализовать в своем приложении с его помощью встроенный редактор для диалогов. Для этого вам необходимо перехватывать все сообщения от мыши для существующего диалога и всех его элементов управления, обработать их, а после всех изменений можно восстановить обычное поведение мыши. Технология также может быть полезна для организации интерактивных учебников, где вы проводите пользователя через серию шагов и не хотите, чтобы он отклонялся от последовательности шагов урока. Таким образом вы можете, например, перехватывать сообщения от кнопок и окон и, если всё делается верно, передавать сообщения исходным обработчикам сообщений используя `wxEvent::Skip`. Сообщения не перехватываемые вашим обработчиком сообщений передаются через таблицу сообщений окна.

3.5 Динамические обработчики сообщений

До этого мы рассматривали обработчики сообщений в терминах статических таблиц сообщений потому, что это самый распространенный метод обработки. Однако есть возможность определять связь между сообщением и его обработчиком динамически. Это может потребоваться, если вам необходимо применять различные механизмы обработки сообщений на разных этапах выполнения программы, или потому что

вы используете другой язык (такой как Python), который не поддерживает статические таблицы сообщений, или вам просто нравится писать оригинальные программы.

Динамические обработчики сообщений позволяют большую гибкость, так как позволяют оперировать отдельными сообщениями, тогда как `PushEventHandler` и `PopEventHandler` оперируют целыми таблицами сообщений. Кроме того, динамические обработчики позволяют разделать единую функцию обработки сообщений между объектами разных типов.

Для работы с динамическими таблицами сообщений применяются две функции: `wxEvtHandler::Connect` и `wxEvtHandler::Disconnect`. На практике вам достаточно редко придется вызывать `wxEvtHandler::Disconnect`, так как разъединение автоматически происходит при уничтожении объекта окна.

Рассмотрим простейший класс фрейма с двумя обработчиками сообщений:

```
// Объявляем наш класс для фрейма
class MyFrame : public wxFrame
{
public:
    // Конструктор
    MyFrame(const wxString& title);

    // Обработчики сообщений
    void OnQuit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);

private:
};
```

Обратите внимание, что мы не используем макрос `DECLARE_EVENT_TABLE`. Чтобы динамически определить обработчику сообщений мы добавим несколько строчек в функцию `OnInit` нашего класса:

```
frame->Connect( wxID_EXIT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventHandler(MyFrame::OnQuit) );

frame->Connect( wxID_ABOUT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventHandler(MyFrame::OnAbout) );
```

Мы передаем методу `Connect` идентификатор окна, идентификатор сообщения и, наконец, указатель на функцию его обработки. Обратите внимание, что имя идентификатора сообщения (`wxEVT_COMMAND_MENU_SELECTED`) отличается от имени макроса для таблицы сообщений (`EVT_MENU`). Макрос таблицы сообщений в своем коде использует данный идентификатор сообщения. Макрос `wxCommandEventHandler`, в который помещена функция необходим для корректного преобразования компилятором типа функции для использования в таблице сообщений. В общем случае, если у вас есть обработчик сообщения, который получает сообщение wxXYZEvent, то

при вызове функции `Connect` вам необходимо заключить вашу функцию в макрос wxXYZEventHandler.

Если нам понадобится удалить связь между сообщением и обработчиком сообщений, то для этого необходимо вызвать функцию `wxEvtHandler::Disconnect`:

```
frame->Disconnect( wxID_EXIT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventHandler(MyFrame::OnQuit) );

frame->Disconnect( wxID_ABOUT,
    wxEVT_COMMAND_MENU_SELECTED,
    wxCommandEventFunction(MyFrame::OnAbout) );
```

3.6 Идентификаторы окон

Идентификаторы окон являются целыми числами, которые используются для определения окна в системе сообщений. Фактически, идентификаторы не обязаны быть уникальными в пределах всего приложения — они должны быть уникальны в пределах некоторого контекста, такого как фрейм и его дочерние окна. Например, вы можете использовать идентификатор `wxID_OK` в большом числе диалогов при условии, что у вас нет нескольких окон с этим идентификатором в одном диалоге.

Если вы передадите идентификатор `wxID_ANY` конструктору для окна, то wxWidgets автоматически сгенерирует идентификатор. Эта возможность бывает полезной, когда вам не важна информация о точном значении идентификатора, например потому, что вы не собираетесь обрабатывать сообщения от этого окна или из-за того, что вы обрабатываете сообщения от всех элементов управления в одном месте. В последнем случае вы должны использовать `wxID_ANY` в таблице сообщений или при вызове `wxEvtHandler::Connect`. Сформированный таким образом идентификатор всегда отрицателен, а поэтому никогда не конфликтует с определенными пользователем идентификаторами, которые всегда должны быть положительными.

wxWidgets предоставляет некоторое число стандартных идентификаторов, перечисленных в Таблице 3.1. Используйте стандартные идентификаторы там где это только возможно! Некоторые системы имеют возможность использовать эту информацию, чтобы использовать для таких окон особую графику (как например, для кнопок OK и Cancel в GTK+) или стандартное поведение (такое как эмуляция сообщения от `wxID_CANCEL` при нажатии на клавишу Esc). В системе Mac OS X элементы меню с идентификаторами `wxID_ABOUT`, `wxID_PREFERENCES` и `wxID_EXIT` специальным образом интерпретируются и выносятся в системное меню. Некоторые компоненты wxWidgets, такие как `wxTextCtrl`, знают как обрабатывать команды меню или кнопок с идентификаторами `wxID_COPY`, `wxID_PASTE` и `wxID_UNDO`.

Таблица 3.1: Стандартные идентификаторы для окон

Идентификатор	Описание
wxID_ANY	Это значение можно передать конструктору окна и тогда wxWidgets сама выберет подходящий идентификатор

Таблица 3.1: (продолжение)

Идентификатор	Описание
wxID_LOWEST	Самое меньшее значение, которое может быть у стандартного идентификатора (4999)
wxID_HIGHEST	Самое большое значение, которое может быть у стандартного идентификатора (5999)
wxID_OPEN	Открыть файл
wxID_CLOSE	Закрыть окно
wxID_NEW	Новое окно, файл или документ
wxID_SAVE	Сохранить файл
wxID_SAVEAS	Сохранить файл под новым именем...
wxID_REVERT	Реверсировать изменения в файле на диске
wxID_EXIT	Выйти из приложения
wxID_UNDO	Откат одного изменения назад
wxID_REDO	Откат одного изменения вперед
wxID_HELP	Основная помощь (например кнопка «Помощь» в диалоге)
wxID_PRINT	Печать
wxID_PRINT_SETUP	Диалог настройки печати
wxID_PREVIEW	Предпросмотр печати
wxID_ABOUT	Показать диалог с информацией о программе
wxID_HELP_CONTENTS	Показать оглавление помощи
wxID_HELP_COMMANDS	Показать помощь по основным командам
wxID_HELP_PROCEDURES	Показать помощь по основным процедурам
wxID_HELP_CONTEXT	Не используется
wxID_CUT	Вырезать
wxID_COPY	Копировать
wxID_PASTE	Вставить
wxID_CLEAR	Очистить
wxID_FIND	Искать
wxID_DUPLICATE	Сделать дубликат
wxID_SELECTALL	Выбрать все
wxID_DELETE	Удалить (вырезать без копирования)
wxID_REPLACE	Заменить
wxID_REPLACE_ALL	Заменить все
wxID_PROPERTIES	Показать свойства выбранного объекта
wxID_VIEW_DETAILS	Посмотреть детали в списке
wxID_VIEW_LARGEICONS	Представить список в виде больших иконок
wxID_VIEW_SMALLICONS	Представить список в виде маленьких иконок
wxID_VIEW_LIST	Представить список в виде таблицы
wxID_VIEW_SORTDATE	Сортировать по дате
wxID_VIEW_SORTNAME	Сортировать по имени
wxID_VIEW_SORTSIZE	Сортировать по размеру
wxID_VIEW_SORTTYPE	Сортировать по типу
от wxID_FILE1 до wxID_FILE9	Загрузить последние обрабатываемые файлы
wxID_OK	Подтвердить выбор в диалоге
wxID_CANCEL	Отклонить выбор в диалоге
wxID_APPLY	Применить выбор в диалоге
wxID_YES	Идентификатор для кнопки «Да»

Таблица 3.1: (продолжение)

Идентификатор	Описание
wxID_NO	Идентификатор для кнопки «Нет»
wxID_STATIC	Идентификатор для статического текста или изображения
wxID_FORWARD	Передвинуться вперед
wxID_BACKWARD	Передвинуться назад
wxID_DEFAULT	Восстановить настройки по умолчанию
wxID_MORE	Увидеть больше настроек
wxID_SETUP	Посмотреть диалог с настройками
wxID_RESET	Отменить сделанные настройки
wxID_CONTEXT_HELP	Посмотреть контекстную помощь
wxID_YESTOALL	Отвечать «Да» на все вопросы
wxID_NOTOALL	Отвечать «Нет» на все вопросы
wxID_ABORT	Прервать текущую операцию
wxID_RETRY	Попробовать выполнить операцию снова
wxID_IGNORE	Игнорировать ошибку
wxID_UP	Передвинуться вверх
wxID_DOWN	Передвинуться вниз
wxID_HOME	Передвинуться в начало
wxID_REFRESH	Обновить
wxID_STOP	Завершить текущую операцию
wxID_INDEX	Показать оглавление
wxID_BOLD	Выделенное сделать жирным
wxID_ITALIC	Выделенное сделать прописным
wxID_JUSTIFY_CENTER	Выравнивание по центру
wxID_JUSTIFY_FILL	Выровнять
wxID_JUSTIFY_RIGHT	Выравнивание по правому краю
wxID_JUSTIFY_LEFT	Выравнивание по левому краю
wxID_UNDERLINE	Подчеркивание
wxID_INDENT	Отступ
wxID_UNINDENT	Удалить отступ
wxID_ZOOM_100	Показать в реальную величину
wxID_ZOOM_FIT	Изменить масштаб по размеру страницы
wxID_ZOOM_IN	Увеличить
wxID_ZOOM_OUT	Уменьшить
wxID_UNDELETE	Восстановить удаленное
wxID_REVERT_TO_SAVED	Восстановить сохраненное состояние

Для своих собственных идентификаторов вам необходимо использовать числа, которые больше `wxID_HIGHEST` или меньше `wxID_LOWEST`.

3.7 Определение собственных сообщений

Если вы хотите хотите определить свои собственный класс сообщения и макрос для него, то вам необходимо выполнить следующие действия:

1. Наследуйте ваш класс от подходящего класса, объявив информацию о динамическом типе и включив функцию `Clone`. Вы можете, если хотите, добавить в

этот класс дополнительные поля и функции для доступа к ним. Наследование вашего класса от `wxCommandEvent` позволит ему перемещаться вверх по иерархии окон, а наследование от `wxNotifyEvent` дополнительно к этому позволит пользоваться функцией `Veto`.

2. Определите `typedef` для функции-обработчика вашего события.

3. Определите таблицу типов сообщения, которые поддерживает ваш класс. Эта таблица объявляется в заголовочном файле конструкцией `BEGIN_DECLARE_EVENT_TYPES() ... END_DECLARE_EVENT_TYPES()`. Каждый тип объявляется с помощью `DECLARE_EVENT_TABLE`(имя, целое_число). Далее в файле с реализацией напишите `DEFINE_EVENT_TYPE`(имя).

4. Определите макрос для таблицы сообщений для каждого типа сообщения.

Давайте проиллюстрируем эти действия на примере. Предположим нам необходимо реализовать новый элемент управления `wxFontSelectorCtrl`, который используется для предпросмотра выбранного шрифта. При щелчке по этому элементу появляется диалог выбора шрифта. Приложению возможно понадобится перехватывать сообщение о выборе шрифта, поэтому мы пошлем собственное командное сообщение из нашего низкоуровневого обработчика сообщений от мыши.

Мы объявим новый класс сообщения `wxFontSelectorCtrlEvent`. Приложение должно быть в состоянии манипулировать этим сообщением и направлять его правильному обработчику с помощью нашего макроса `EVT_FONT_SELECTION_CHANGED(id, func)`, который использует единственный тип сообщения `wxEVT_COMMAND_FONT_SELECTION_CHANGED`. Вот почему нам необходим заголовочный файл с описанием всего необходимого для создаваемого сообщения, в дополнение к объявлению собственно самого элемента управления (здесь код последнего не приводится):

```
/*!
 * Класс сообщения о выборе шрифта
 */
class wxFontSelectorCtrlEvent : public wxNotifyEvent
{
public:
    wxFontSelectorCtrlEvent(wxEventType commandType = wxEVT_NULL,
         int id = 0): wxNotifyEvent(commandType, id) {}

    wxFontSelectorCtrlEvent(const wxFontSelectorCtrlEvent& event):
         wxNotifyEvent(event) {}

    virtual wxEvent *Clone() const
         { return new wxFontSelectorCtrlEvent(*this); }

DECLARE_DYNAMIC_CLASS(wxFontSelectorCtrlEvent);
};

typedef void (wxEvtHandler::*wxFontSelectorCtrlEventFunction)
```

```
                                        (wxFontSelectorCtrlEvent&);

/*!
 * Сообщение о выборе шрифта и макрос для его обработки
 */
BEGIN_DECLARE_EVENT_TYPES()
    DECLARE_EVENT_TYPE(wxEVT_COMMAND_FONT_SELECTION_CHANGED, 801)
END_DECLARE_EVENT_TYPES()

#define EVT_FONT_SELECTION_CHANGED(id, fn) DECLARE_EVENT_TABLE_ENTRY( \
        wxEVT_COMMAND_FONT_SELECTION_CHANGED, id, -1, \
        (wxObjectEventFunction) (wxEventFunction) \
        (wxFontSelectorCtrlEventFunction) & fn, (wxObject *) NULL ),
```

В нашем файле с реализацией мы пишем

```
DEFINE_EVENT_TYPE(wxEVT_COMMAND_FONT_SELECTION_CHANGED)
IMPLEMENT_DYNAMIC_CLASS(wxFontSelectorCtrlEvent, wxNotifyEvent)
```

Чтобы послать созданное сообщение диалог выбора шрифта может вызвать метод `ProcessEvent`, когда будет зафиксировано соответствующее изменение внутри обработчика сообщения от мыши:

```
wxFontSelectorCtrlEvent event(
                    wxEVT_COMMAND_FONT_SELECTION_CHANGED, GetId());
event.SetEventObject(this);
GetEventHandler()->ProcessEvent(event);
```

Теперь приложение может обрабатывать наше сообщение о выборе шрифта:

```
BEGIN_EVENT_TABLE(MyDialog, wxDialog)
  EVT_FONT_SELECTION_CHANGED(ID_FONTSEL, MyDialog::OnChangeFont)
END_EVENT_TABLE()

void MyDialog::OnChangeFont(wxFontSelectorCtrlEvent& event)
{
    // Подходящее действие, когда меняется шрифт
    ...
}
```

Значение идентификатора сообщения (801) не используется в последних версиях библиотеки wxWidgets и оставлено исключительно для совместимости с wxWidgets версии 2.4.

Давайте посмотрим на код макроса для обработчика сообщения:

```
#define EVT_FONT_SELECTION_CHANGED(id, fn) DECLARE_EVENT_TABLE_ENTRY( \
        wxEVT_COMMAND_FONT_SELECTION_CHANGED, id, -1, \
        (wxObjectEventFunction) (wxEventFunction) \
        (wxFontSelectorCtrlEventFunction) & fn, (wxObject *) NULL ),
```

Макрос помещает информацию в массив, который формирует таблицу сообщений, вот почему синтаксис макроса выглядит так странно. Каждая запись в таблице сообщений состоит из пяти частей, описанных ниже:

1. Тип сообщения. Один класс сообщения может обрабатывать несколько типов, но в нашем примере мы определяем только один тип сообщения, вот почему у нас всего один макрос. Этот тип должен соответствовать типу обрабатываемого сообщения.

2. Значение идентификатора, передаваемого макросу. Функция обработки сообщения вызывается только если это значение совпадает со значением в обрабатываемом сообщении.

3. Второе значение идентификатора. Используется при определении некоторого множества значений. Значение -1 говорит о том, что второго значения нет.

4. Функция обработки сообщения. Последовательность приведений, выполняемых для некоторых компиляторов. Это место, где используется `typedef` для методов.

5. Пользовательские данные, обычно `NULL`.

Полный текст примера определения собственного сообщения расположена в папке examples/chap03 и включает в себя реализацию собственного диалога выбора шрифта и удобный класс-валидатор, который вы можете использовать в своих программах. Еще одним источником для вдохновения может стать файл include/wx/event.h из вашего дистрибутива wxWidgets.

3.8 Итоги

В этой главе мы обсудили каким образом сообщения передаются через дерево наследования и иерархию окон, рассмотрели подключаемые и динамические обработчики сообщений, поговорили о идентификаторах окон и описали последовательность объявления собственных классов и макросов для сообщений. Если вы хотите получить более подробное описание механики работы приложений обратитесь к Приложению 8 «Как wxWidgets обрабатывает сообщения». В Приложении 9 «Классы и макросы для сообщений» приведены наиболее часто используемые классы и макросы для них. Вы также можете найти несколько полезных примеров в дистрибутиве wxWidgets, по большей части в папке samples/event. В следующей главе мы рассмотрим важнейшие GUI-компоненты, которые можно использовать в своих приложениях.