Н. А. Литвиненко

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ на C++

Win32 API-приложения

- *Графический интерфейс Windows-приложения*
- *Элементы управления*
- *Создание дочерних и всплывающих окон*
- *Растровая графика*
- *Библиотеки динамической компоновки (DLL)*
- *Процессы и потоки*

УЧЕБНОЕ ПОСОБИЕ

Н. А. Литвиненко

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ на C++

Win32 API-приложения

Рекомендовано Государственным образовательным учреждением
высшего профессионального образования
«Санкт-Петербургский государственный университет
информационных технологий, механики и оптики»
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлениям подготовки «Информационные системы»,
«Информатика и вычислительная техника».

Регистрационный номер рецензии 466 от 10.09.2009 г. МГУП

Санкт-Петербург

«БХВ-Петербург»

2010

УДК 681.3.068+800C++(075.8)
ББК 32.973.26-018.1я73
Л64

Литвиненко Н. А.

Л64 Технология программирования на C++. Win32 API-приложения. — СПб.: БХВ-Петербург, 2010. — 288 с.: ил. — (Учебное пособие)

ISBN 978-5-9775-0600-7

Изложен начальный курс низкоуровневого программирования на C++ для Windows с использованием библиотеки Win32 API. Рассмотрены графический интерфейс Windows-приложения, стандартные диалоговые окна, элементы управления, растровая графика, DLL-библиотеки, процессы и потоки. Материал иллюстрирован многочисленными примерами, выполненными в Visual Studio 2010 под управлением Windows 7.

Для студентов и преподавателей технических вузов и самообразования

УДК 681.3.068+800C++(075.8)
ББК 32.973.26-018.1я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Юрий Рожко*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 29.06.10.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 23,22.
Тираж 1000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12.

ISBN 978-5-9775-0600-7

Оглавление

Введение

Данное учебное пособие продолжает курс "Технология программирования на C++. Начальный курс", изданного в 2005 году издательством "БХВ-Петербург", и предназначено для студентов технических вузов, обучающихся по специальностям "Информационные системы", "Информатика и вычислительная техника", осваивающих программирование на языке C++. При изучении данного курса требуется знание языка C++ на уровне консольных приложений. Необходимо также знание библиотеки STL (от англ. Standard Template Library).

Учебное пособие является начальным курсом низкоуровневого программирования под Windows с использованием библиотеки *Программного интерфейса приложений* (*Application Program Interface, API*), точнее, ее 32-разрядного подмножества *Win32 API*, и построено на основе курса лекций, более 5 лет читаемых студентам специальностей "Программное обеспечение вычислительной техники и автоматизированных систем" и "Информационные системы и технологии". Это накладывает определенный отпечаток на стиль изложения и объем рассмотренного материала. Курс построен на типовых задачах таким образом, что новые задачи рассматриваются по нарастающей сложности, а необходимые понятия вводятся по мере изложения. Для освоения материала необходимо выполнить все рассматриваемые примеры и решить большую часть заданий для самостоятельной работы. Все примеры протестированы в среде Visual Studio 2010 Release Candidate и операционной системе Windows 7, но работают и в Visual Studio 2005/2008 под управлением операционных систем Windows 2000, XP, Vista.

Несмотря на повсеместное внедрение NET-технологий, автор считает, что для профессионального освоения программирования под Windows необходимо начинать с "низкого" уровня, т. е. с *библиотеки Win32 API*. Следует признать, что учебников, посвященных низкоуровневому программированию для Windows, издается недостаточно. До сих пор не потеряли актуальность ставшие уже классическими курсы Ч. Педзольда, Г. Шилда, У. Мюррея [1—3], изданные в середине 90-х годов. Из литературы последнего времени нужно отметить фундаментальный труд Дж. Рихтера [4], достаточно объемную книгу Ю. А. Шупака [5], неплохой, но, к сожалению, неполный справочник Р. Д. Верма [6], а также учебное пособие В. Г. Давыдова, в котором автор осуществил попытку параллельного изложения низкоуровневого программирования в Win32 API и программирования с использованием *библиотеки MFC* (Microsoft Foundation Classes).

Книга состоит из шести глав.

- *Глава 1. "Интерфейс Windows-приложения"* — рассматривает скелет Windows-приложения, обработку сообщений, вывод текста и простейшую графику.

- *Глава 2. "Работа с файлами"* — на примере задачи построения программы для просмотра (просмотрщика) текстовых файлов рассматриваются стандартные диалоги: выбора имени файла, выбора шрифта, а также организация скроллинга.
- *Глава 3. "Окна и элементы управления"* — рассматривается техника создания дочерних и всплывающих окон, а также диалоговые окна как контейнеры для стандартных и общих элементов управления.
- *Глава 4. "Растровая графика"* — на многочисленных примерах показана методика вывода в окно растровых изображений. Рассматриваются виртуальные окна и метафайлы.
- *Глава 5. "Библиотеки динамической компоновки DLL"* — показана техника создания пользовательских динамических библиотек, их использование при явном и неявном связывании.
- *Глава 6. "Процессы и потоки"* — рассматриваются вопросы создания процессов и потоков, механизмы их синхронизации, объекты ядра и обмен данными между процессами.

При построении примеров возникает вопрос, в какой кодировке работать? Дело в том, что сейчас большинство проектов разрабатывается в Unicode-кодировке, где каждому символу выделяется 16 бит (под кириллицу выделены коды в диапазоне 0x400—0x4ff). Visual Studio позволяет работать и в традиционной для Windows однобайтной кодировке (для России кодовая страница CP-1251). Однако многие современные API-функции, даже принимающие строку в однобайтной кодировке, вынуждены преобразовывать ее в кодировку Unicode, поэтому для повышения эффективности рекомендуется изначально разрабатывать проект в Unicode.

Мы будем, по возможности, разрабатывать универсальный проект, который можно компилировать как в кодировке Unicode, так и в однобайтной Windows-кодировке. Для этого будем пользоваться определениями, размещенными в файле включений `tchar.h`, где в зависимости от выбранной кодировки происходит переопределение функций. Например, если использовать функцию `_tcslen()`, то в однобайтной кодировке ее имя преобразуется в `strlen()`, если же выбрана Unicode-кодировка (определена константа `_UNICODE`), имя преобразуется в `wcslen()`. Вот так это сделано в файле включений `tchar.h`:

```
#ifdef  _UNICODE
#define _tcslen     wcslen
#else
#define _tcslen     strlen
#endif  /* _UNICODE */
```

Большая часть API-функций также имеет две реализации: с суффиксом "`A`" для однобайтной кодировки и "`W`" — для Unicode, например `TextOutA()` или `TextOutW()`.

Глава 1

Интерфейс Windows-приложения

Стиль программирования Windows-приложений принципиально отличается от того, который сложился в операционных системах раннего поколения. В MS-DOS программа монопольно владеет всеми ресурсами системы и является инициатором взаимодействия с операционной системой. Совсем иначе дело обстоит в операционной системе Windows, которая строилась как многозадачная, и именно операционная система является инициатором обращения к программе. Все ресурсы Windows являются разделяемыми, и программа, в дальнейшем будем называть ее *приложением*, не может владеть ими монопольно. В связи с такой идеологией построения операционной системы приложение должно ждать посылки сообщения операционной системы и лишь после его получения выполнить определенные действия, затем вновь перейти в режим ожидания очередного сообщения. На рис. 1.1 схематично изображена диаграмма типичного Windows-приложения.

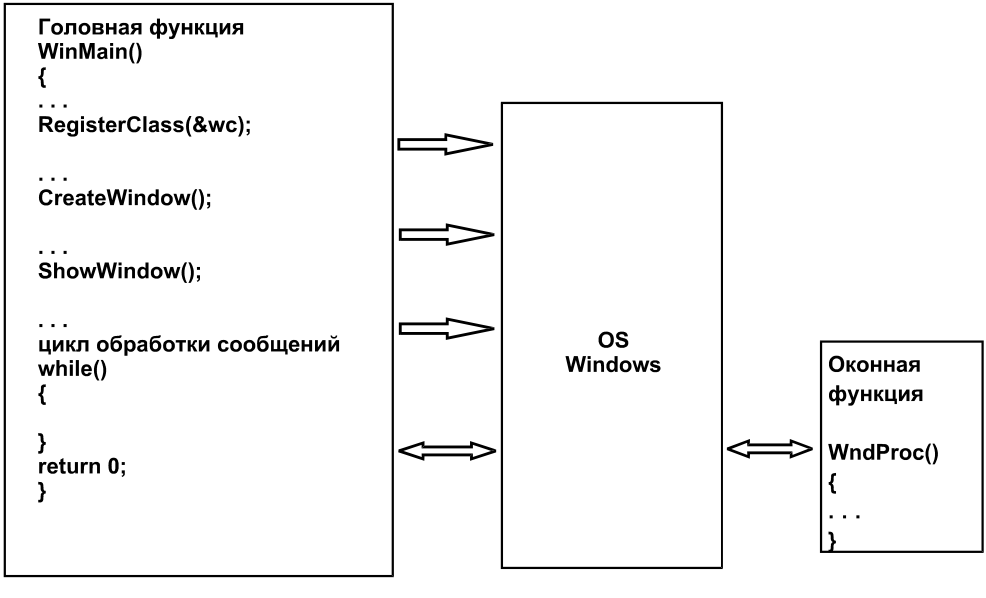

Рис. 1.1. Структура Windows-приложения

Windows генерирует множество различных сообщений, которые направляются приложению, например, щелчок кнопки мыши или нажатие клавиши на клавиатуре. Если приложение не обрабатывает какие-то сообщения, реакция на них осуществляется операционной системой стандартным способом, так что задачей программиста является обработка лишь тех сообщений, которые необходимы приложению.

Разработчиками операционной системы Windows была создана библиотека функций, при помощи которых и происходит взаимодействие приложения с операционной системой, так называемые функции *Программного интерфейса приложений* (*Application Program Interface, API*).

Подмножество этих функций, предназначенных для графического вывода на дисплей, графопостроитель и принтер, представляет собой *Интерфейс графических устройств* (*Graphics Device Interface, GDI*).

Библиотека API-функций разрабатывалась в расчете на то, что ее можно использовать для любого языка программирования, а поскольку разные языки имеют различные типы данных, то были созданы собственные Windows-типы, которые приводятся к типам данных языков программирования. Отметим только, что в Windows нет логического типа `bool`, но есть Windows-тип `BOOL`, который эквивалентен целому типу `int`. Будем рассматривать типы данных Windows по мере необходимости.

Еще одной особенностью API-функций является использование обратного, по отношению к принятому в языке C, порядка передачи параметров, как это реализовано в языке Pascal. В C для идентификации таких функций использовалось служебное слово `pascal`, в Windows введены его синонимы `CALLBACK`, `APIENTRY` или `WINAPI`. По умолчанию C-функции передают параметры, начиная с конца списка так, что первый параметр всегда находится на вершине стека. Именно это позволяет использовать в языке C функции с переменным числом параметров, что в API-функциях невозможно.

Каркас Windows-приложения

В отличие от программы, выполняемой в операционной системе MS-DOS, даже для создания простейшего приложения под Windows придется проделать намного больше работы. Чтобы иметь возможность работать с оконным интерфейсом, заготовка или каркас Windows-приложения должна выполнить некоторые стандартные действия:

1. Определить *класс окна*.
2. Зарегистрировать окно.
3. Создать окно данного класса.
4. Отобразить окно.
5. Запустить цикл обработки сообщений.

Примечание

Термин *интерфейс* здесь следует понимать как способ взаимодействия пользователя и приложения. *Класс окна* — структура, определяющая его свойства.

Рассмотрим сначала, как можно "вручную" создать минимальное Win32-приложение. Загрузив Visual Studio 2010, выполним команду **File | New | Project…** и выберем тип проекта — **Win32 Project**. В раскрывающемся списке **Location** выберем путь к рабочей папке, а в поле **Name** имя проекта (рис. 1.2). В следующем диалоговом окне, приведенном на рис. 1.3, нажимаем кнопку **Next**, а в окне опций проекта (рис. 1.4) выберем флажок **Empty project** (Пустой проект) и нажмем кнопку **Finish** — получим пустой проект, в котором нет ни одного файла.

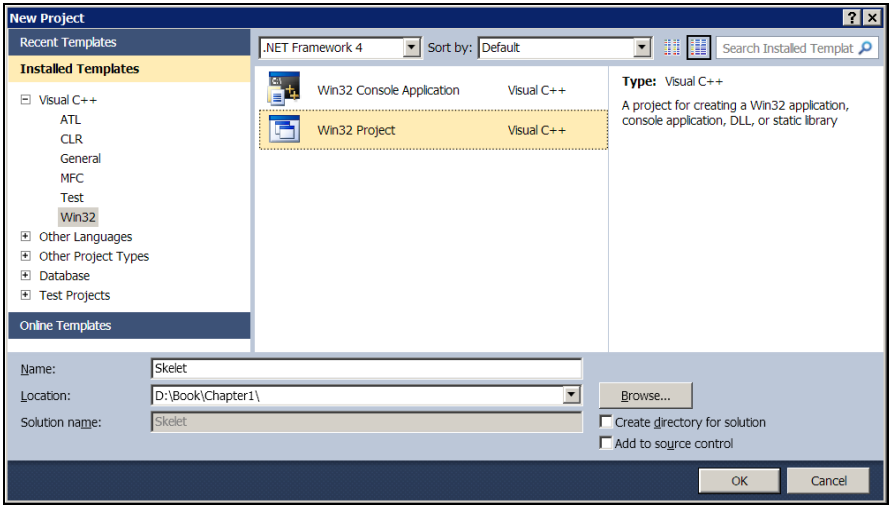

Рис. 1.2. Выбор типа проекта

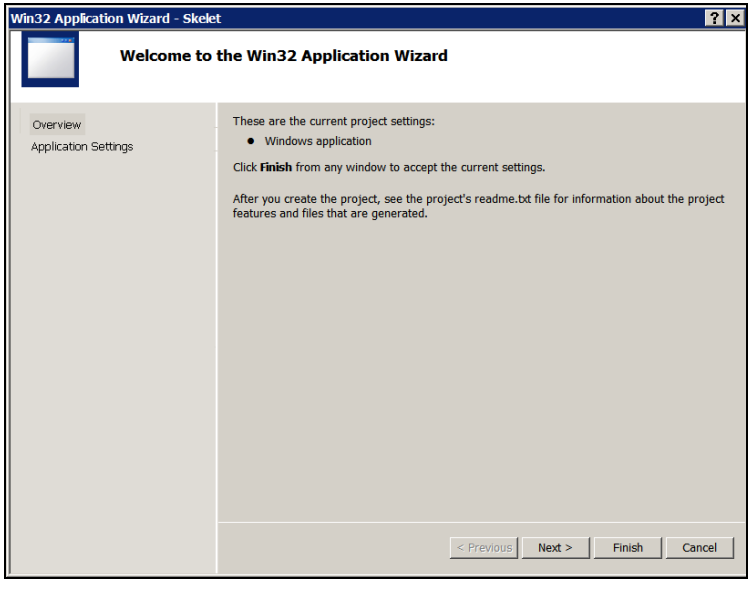

Рис. 1.3. Стартовое окно построителя приложения

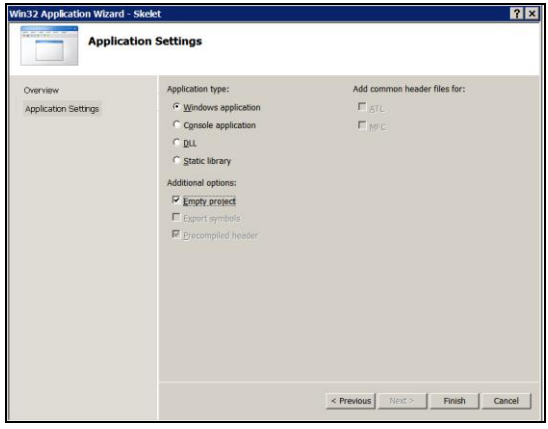

Рис. 1.4. Окно опций проекта

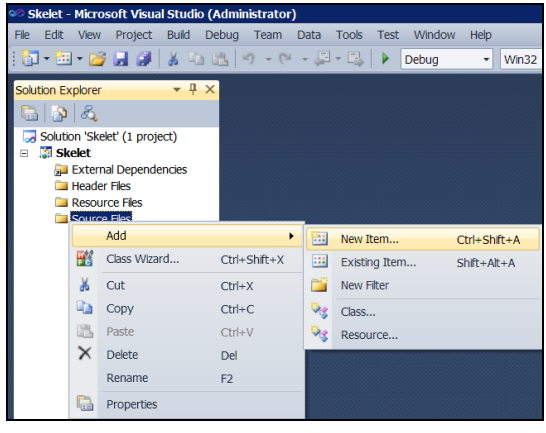

Рис. 1.5. Добавление к проекту нового объекта с помощью контекстного меню

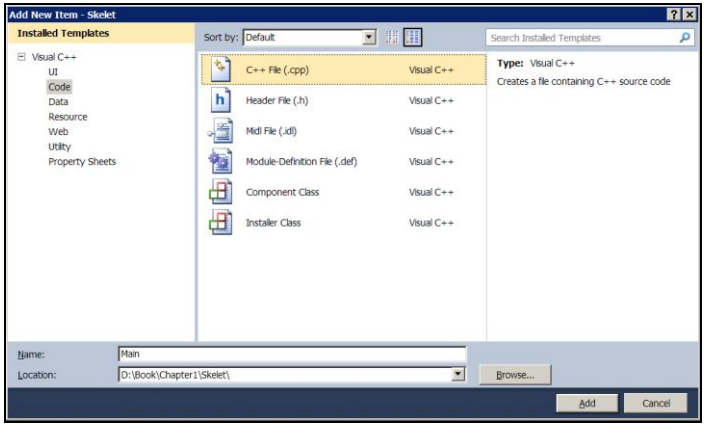

Рис. 1.6. Выбор шаблона объекта

С помощью контекстного меню (рис. 1.5) добавим файл для кода приложения, имя файла введем в ходе диалога выбора шаблона объекта на рис. 1.6. (Тот же самый диалог мы могли бы получить по команде меню **Project | Add New Item….)**

ПРИМЕЧАНИЕ

Пока мы создаем простые решения, состоящие из одного проекта, можно убрать флажок **Create directory for solution**. Это упростит структуру каталога.

С помощью листинга 1.1 рассмотрим "скелет" Windows-приложения.

Листинг 1.1. Минимальный код каркаса Windows-приложения

```
#include <windows.h>
#include <tchar.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
TCHAR WinName[] = _T("MainFrame");
int APIENTRY _tWinMain(HINSTANCE This, // Дескриптор текущего приложения
   HINSTANCE Prev,                     // В современных системах всегда 0
   LPTSTR cmd,                         // Командная строка
   int mode)                           // Режим отображения окна
{
       HWND hWnd;            // Дескриптор главного окна программы
       MSG msg;              // Структура для хранения сообщения
       WNDCLASS wc;  // Класс окна
// Определение класса окна
       wc.hInstance = This;
       wc.lpszClassName = WinName;                 // Имя класса окна
       wc.lpfnWndProc = WndProc;                          // Функция окна
       wc.style = CS_HREDRAW | CS_VREDRAW;         // Стиль окна
       wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);        // Стандартная иконка
       wc.hCursor = LoadCursor(NULL,IDC_ARROW);   // Стандартный курсор
       wc.lpszMenuName = NULL;      // Нет меню
       wc.cbClsExtra = 0;           // Нет дополнительных данных класса
       wc.cbWndExtra = 0;           // Нет дополнительных данных окна
       // Заполнение окна белым цветом
       wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
       if(!RegisterClass(&wc)) return 0;   // Регистрация класса окна
// Создание окна
       hWnd = CreateWindow(WinName, // Имя класса окна
       _T("Каркас Windows-приложения"),    // Заголовок окна
       WS_OVERLAPPEDWINDOW,         // Стиль окна
       CW_USEDEFAULT,// x
       CW_USEDEFAULT,// y   Размеры окна
       CW_USEDEFAULT,// Width
```

```
        CW_USEDEFAULT,// Height
        HWND_DESKTOP,  // Дескриптор родительского окна
        NULL,          // Нет меню
        This,          // Дескриптор приложения
        NULL);         // Дополнительной информации нет
        ShowWindow(hWnd, mode); //Показать окно
// Цикл обработки сообщений
        while(GetMessage(&msg, NULL, 0, 0))
        {
        TranslateMessage(&msg);// Функция трансляции кодов нажатой клавиши
        DispatchMessage(&msg); // Посылает сообщение функции WndProc()
        }
        return 0;
}
// Оконная функция вызывается операционной системой
// и получает сообщения из очереди для данного приложения
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                WPARAM wParam, LPARAM lParam)
{       // Обработчик сообщений
        switch(message)
        {
        case WM_DESTROY : PostQuitMessage(0);
                        break; // Завершение программы
        // Обработка сообщения по умолчанию
        default : return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

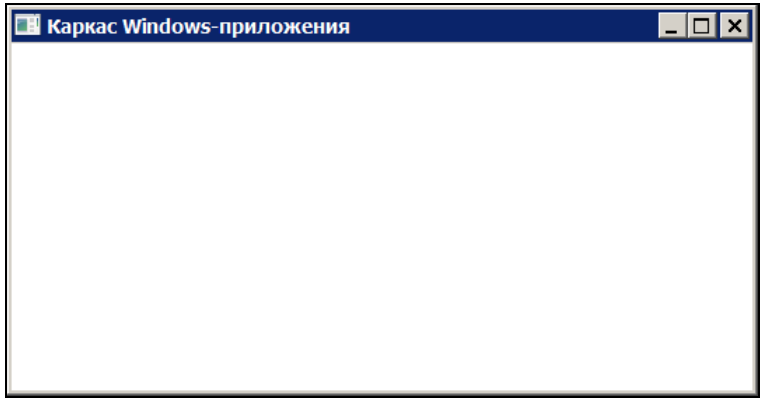

Рис. 1.7. Окно первой Windows-программы

Программа не делает ничего полезного, поэтому, запустив ее на выполнение кнопкой ▶ (**Start Debugging**), мы получим изображенное на рис. 1.7 пустое окно, имеющее заголовок и набор стандартных кнопок.

Исследование каркаса Windows-приложения

Давайте подробно рассмотрим текст нашей программы. Первая строка содержит файл включений, который обязательно присутствует во всех Windows-программах.

```
#include <windows.h>
```

Если в ранних версиях Visual Studio этот файл содержал основные определения, то сейчас он служит для вызова других файлов включений, основные из которых: `windef.h`, `winbase.h`, `wingdi.h`, `winuser.h`; а также несколько дополнительных файлов, в которых помещены определения API-функций, констант и макросов.

Дополнительно подключим:

```
#include <tchar.h>
```

В этом файле содержатся определения некоторых полезных макросов, например, макрос `_T()` служит для создания строки Unicode на этапе компиляции и определен примерно так:

```
#define _T(x)       __T(x)
#ifdef  _UNICODE
#define __T(x)      L ## x
#else
#define __T(x)      x
#endif
```

Макрос преобразуется в оператор "`L`", который является инструкцией компилятору для образования строки Unicode, если определена константа `_UNICODE`; и в "пустой оператор", если константа не определена. Константа `_UNICODE` устанавливается в зависимости от установок свойства проекта **Character Set** (рис. 1.8). Диалоговое окно свойств **Property Pages** доступно сейчас на подложке **Property Manager** панели управления **Solution Explorer**.

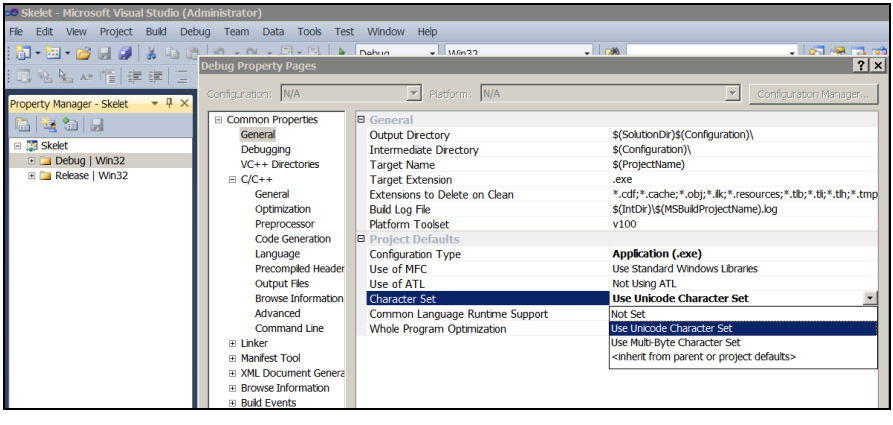

Рис. 1.8. Страница общих свойств проекта

Таким образом, этот макрос позволяет компилировать проект как в кодировке Unicode, так и в Windows-кодировке. Мы подробно рассмотрели данный макрос потому, что многие определения Windows описаны подобным образом.

Далее следует прототип оконной функции:

```
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

Оконная функция также является функцией обратного вызова, что связано с некоторыми особенностями организации вызовов операционной системы. Эта функция регистрируется в системе, а ее вызов осуществляет операционная система, когда требуется обработать сообщение. Тип возвращаемого значения функции `LRESULT` эквивалентен `long` для Win32-проекта.

На глобальном уровне описывается имя класса окна приложения в виде текстовой строки:

```
TCHAR WinName[] = _T("MainFrame");
```

Тип `TCHAR` также преобразуется в `wchar_t`, если определена константа `_UNICODE`, и в `char`, если константа не определена.

> ***Примечание***
>
> Тип `wchar_t` эквивалентен типу `short` и служит для хранения строк в кодировке Unicode, где для одного символа выделяется 16 бит.

Имя класса окна используется операционной системой для его идентификации. Имя может быть произвольным, в частности содержать кириллический текст.

Рассмотрим заголовок головной функции:

```
int APIENTRY _tWinMain(HINSTANCE This, // Дескриптор текущего приложения
       HINSTANCE Prev,          // В современных системах всегда 0
       LPTSTR cmd,       // Командная строка
       int mode)                // Режим отображения окна
```

Для Windows-приложений с Unicode она носит имя `wWinMain()`, а в 8-битной кодировке — `WinMain()`, выбор варианта определяется префиксом `_t`, что также является стандартным приемом в библиотеке API-функций. Функция имеет четыре параметра, устанавливаемых при загрузке приложения:

- `This` — дескриптор, присваиваемый операционной системой при загрузке приложения;
- `Prev` — параметр предназначен для хранения дескриптора предыдущего экземпляра приложения, уже загруженного системой. Сейчас он потерял свою актуальность и сохранен лишь для совместимости со старыми приложениями (начиная с Windows 95, параметр устанавливается в нулевое значение);
- `cmd` — указатель командной строки, но без имени запускаемой программы. Тип `LPTSTR` эквивалентен `TCHAR*`;
- `mode` — режим отображения окна.

ПРИМЕЧАНИЕ

Здесь впервые появляется Windows-тип данных — *дескриптор* (описатель), который используется для описания объектов операционной системы. Дескриптор напоминает индекс хеш-таблицы и позволяет отслеживать состояние объекта в памяти при его перемещении по инициативе операционной системы. Предусмотрено много типов дескрипторов: `HINSTANCE`, `HWND` и др., но все они являются 32-разрядными целыми числами.

Внутри головной функции описаны три переменные:

- `hWnd` — предназначена для хранения дескриптора главного окна программы;
- `msg` — это структура, в которой хранится информация о сообщении, передаваемом операционной системой окну приложения:

```
struct MSG
{
    HWND hWnd;          // Дескриптор окна
    UINT message;       // Номер сообщения
    WPARAM wParam;      // 32-разрядные целые содержат
    LPARAM lParam;      // дополнительные параметры сообщения
    DWORD time;         // Время посылки сообщения в миллисекундах
    POINT pt;           // Координаты курсора (x,y)
 };
struct POINT
{
    LONG x,y;
};
```

ПРИМЕЧАНИЕ

Тип `WPARAM` — "короткий параметр" был предназначен для передачи 16-разрядного значения в 16-разрядной операционной системе, в Win32 это такое же 32-разрядное значение, что и `LPARAM`.

- `wc` — структура, содержащая информацию по настройке окна. Требуется заполнить следующие поля:
  - `wc.hInstance = This;`

    Дескриптор текущего приложения.
  - `wc.lpszClassName = WinName;`

    Имя класса окна.
  - `wc.lpfnWndProc = WndProc;`

    Имя оконной функции для обработки сообщений.
  - `wc.style = CS_HREDRAW | CS_VREDRAW;`

    Такой стиль определяет автоматическую перерисовку окна при изменении его ширины или высоты.
  - `wc.hIcon = LoadIcon(NULL,IDI_APPLICATION);`

    Дескриптор пиктограммы (иконки) приложения. Функция `LoadIcon()` обеспечивает ее загрузку. Если первый параметр `NULL`, используется системная пиктограмма, которая выбирается по второму параметру из следующего набора:

    ◊ `IDI_APPLICATION` — стандартная иконка;

    ◊ `IDI_ASTERISK` — звездочка;

◊ IDI_EXCLAMATION — восклицательный знак;

◊ IDI_HAND — ладонь;

◊ IDI_QUESTION — вопросительный знак;

◊ IDI_WINLOGO — логотип Windows;

- `wc.hCursor = LoadCursor(NULL,IDC_ARROW);`

  Аналогичная функция `LoadCursor()` обеспечивает загрузку графического образа курсора, где нулевой первый параметр также означает использование системного курсора, вид которого можно выбрать из списка:

  ◊ IDC_ARROW — стандартный курсор;

  ◊ IDC_APPSTARTING — стандартный курсор и маленькие песочные часы;

  ◊ IDC_CROSS — перекрестие;

  ◊ IDC_IBEAM — текстовый курсор;

  ◊ IDC_NO — перечеркнутый круг;

  ◊ IDC_SIZEALL — четырехлепестковая стрелка;

  ◊ IDC_SIZENESW — двухлепестковая стрелка, северо-восток и юго-запад;

  ◊ IDC_SIZENWSE — двухлепестковая стрелка, северо-запад и юго-восток;

  ◊ IDC_SIZENS — двухлепестковая стрелка, север и юг;

  ◊ IDC_SIZEWE — двухлепестковая стрелка, запад и восток;

  ◊ IDC_UPARROW — стрелка вверх;

  ◊ IDC_WAIT — песочные часы;

- `wc.lpszMenuName = NULL;`

  Ссылка на строку главного меню, при его отсутствии NULL.

- `wc.cbClsExtra = 0;`

  Дополнительные параметры класса окна.

- `wc.cbWndExtra = 0;`

  Дополнительные параметры окна.

- `wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);`

  Дескриптор кисти, которая используется для заполнения окна. Стандартная конструкция, создает системную кисть белого цвета WHITE_BRUSH. Требуется явное преобразование типа — HBRUSH.

После того как определены основные характеристики окна, можно это окно создать при помощи API-функции `CreateWindow()`, где также нужно задать параметры:

1. WinName — имя, которое присвоено классу окна.
2. `_T("Каркас Windows-приложения")` — заголовок окна в виде строки Unicode либо С-строки.

3. `WS_OVERLAPPEDWINDOW` — макрос, определяющий стиль отображения стандартного окна, имеющего системное меню, заголовок, рамку для изменения размеров, а также кнопки минимизации, развертывания и закрытия. Это наиболее общий стиль окна, он определен так:

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED|WS_CAPTION|WS_SYSMENU|
WS_THICKFRAME|WS_MINIMIZEBOX|WS_MAXIMIZEBOX)
```

   Можно создать другой стиль, используя комбинацию стилевых макросов при помощи операции логического сложения, вот некоторые из них:

   - `WS_OVERLAPPED` — стандартное окно с рамкой;
   - `WS_CAPTION` — окно с заголовком;
   - `WS_THICKFRAME` — окно с рамкой;
   - `WS_MAXIMIZEBOX` — кнопка распахивания окна;
   - `WS_MINIMIZEBOX` — кнопка минимизации;
   - `WS_SYSMENU` — системное меню;
   - `WS_HSCROLL` — горизонтальная панель прокрутки;
   - `WS_VSCROLL` — вертикальная панель прокрутки;
   - `WS_VISIBLE` — окно отображается;
   - `WS_CHILD` — дочернее окно;
   - `WS_POPUP` — всплывающее окно;

4. Следующие два параметра определяют координаты левого верхнего угла окна `(x,y)`, еще два параметра: `Width` — ширину и `Height` — высоту окна в пикселах. Задание параметра `CW_USEDEFAULT` означает, что система сама выберет для отображения окна наиболее (с ее точки зрения) удобное место и размер.
5. Следующий параметр — указатель на структуру меню, или `NULL`, при его отсутствии.
6. Далее требуется указать дескриптор приложения, владельца окна — `This`.
7. И, наконец, указатель на дополнительную информацию, в нашем случае — `NULL`.

Окно создано, и с ним можно работать, но пока оно не отображается. Для того чтобы окно увидеть, необходимо его отобразить с помощью функции `ShowWindow(hWnd, mode)`, которая принимает два параметра: `hWnd` — дескриптор окна и `mode` — режим отображения. В нашем случае мы используем значение, полученное при открытии приложения через параметр головной функции.

Далее, заключительная часть головной функции — цикл обработки сообщений. Он задается оператором `while`, аргументом которого является функция `GetMessage(&msg, NULL, 0, 0)`. Такой цикл является обязательным для всех Windows-приложений, его цель — получение и обработка сообщений, передаваемых операционной системой. Операционная система ставит сообщения в очередь, откуда они извлекаются функцией `GetMessage()` по мере готовности приложения:

- ❒ первым параметром функции является `&msg` — указатель на структуру `MSG`, где и хранятся сообщения;

- второй параметр `hWnd` — определяет окно, для которого предназначено сообщение, если же необходимо перехватить сообщения всех окон данного приложения, он должен быть `NULL`;
- остальные два параметра определяют `[min, max]` диапазон получаемых сообщений. Чаще всего необходимо обработать все сообщения, тогда эти параметры должны быть равны 0.

> ***Примечание***
>
> Сообщения определяются их номерами, символические имена для них определены в файле включений `winuser.h`. Префикс всех системных сообщений `WM_`.

Внутри цикла расположены две функции:

```
TranslateMessage(&msg);
DispatchMessage(&msg);
```

Первая из них транслирует код нажатой клавиши в клавиатурные сообщения `WM_CHAR`. При этом в переменную `wParam` структуры `msg` помещается код нажатой клавиши в Windows-кодировке CP-1251, в младшее слово `lParam` — количество повторений этого сообщения в результате удержания клавиши в нажатом состоянии, а в старшее слово — битовая карта со значениями, приведенными в табл. 1.1.

Таблица 1.1. *Битовая карта клавиатуры,* `HIWORD(lParam)`

Бит	Значение
15	1, если клавиша отпущена, 0 — если нажата
14	1, если клавиша была нажата перед посылкой сообщения
13	1, если нажата клавиша <Alt>
12—9	Резерв
8	1, если нажата функциональная клавиша
7—0	Scan-код клавиши

Использование этой функции не обязательно и нужно только для обработки сообщений от клавиатуры.

Вторая функция, `DispatchMessage(&msg)`, обеспечивает возврат преобразованного сообщения обратно операционной системе и инициирует вызов оконной функции данного приложения для его обработки.

Данным циклом и заканчивается головная функция.

Нам осталось лишь описать оконную функцию `WndProc()`, и построение каркаса Windows-приложения будет закончено.

Основной компонент этой функции — переключатель `switch`, обеспечивающий выбор соответствующего обработчика сообщений по его номеру `message`. В нашем случае мы предусмотрели обработку лишь одного сообщения `WM_DESTROY`. Это сообщение посылается, когда пользователь завершает программу. Получив его,

оконная функция вызывает функцию `PostQuitMessage(0)`, которая завершает приложение и передает операционной системе код возврата — 0. Если говорить точнее, генерируется сообщение `WM_QUIT`, получив которое функция `GetMessage()` возвращает нулевое значение. В результате цикл обработки сообщений прекращается и происходит завершение работы приложения.

Все остальные сообщения обрабатываются по умолчанию функцией `DefWindowProc()`, имеющей такой же список параметров и аналогичное возвращаемое значение, поэтому ее вызов помещается после оператора `return`.

Стандартная заготовка Windows-приложения

Мастер Visual Studio позволяет автоматически генерировать стандартную заготовку Windows-приложения. Для этого в стартовом окне построителя Win32-приложения (см. рис. 1.3) достаточно выбрать кнопку **Finish**. Проект состоит из набора файлов, показанных на рис. 1.9.

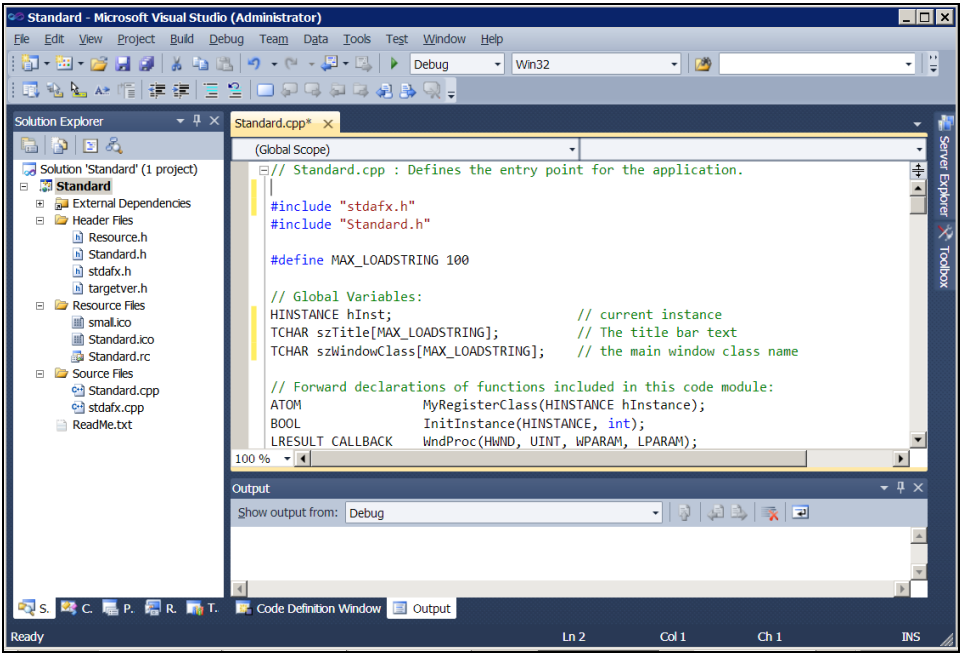

Рис. 1.9. Состав стандартной заготовки Win32-приложения

Рассмотрим подробнее представленный в листинге 1.2 проект, опуская некоторые комментарии и несущественные детали.

Листинг 1.2. Стандартная заготовка Win32-приложения

```
//stdafx.h ////////////////////////////////////////////////////
#pragma once
```

```
#include "targetver.h"
#define WIN32_LEAN_AND_MEAN  //Отключает некоторые редко используемые
//возможности компилятора, для ускорения компиляции
#include <windows.h>         //Стандартный набор файлов включений
#include <stdlib.h>          //для Win32-проекта
#include <malloc.h>
#include <memory.h>
#include <tchar.h>

//Standard.h//////////////////////////////////////////////
#pragma once
#include "resource.h"

//targetver.h/////////////////////////////////////////////
#pragma once
#include <SDKDDKVer.h>

//resource.h//////////////////////////////////////////////
#define IDS_APP_TITLE                   103
#define IDR_MAINFRAME                   128
#define IDD_STANDARD_DIALOG             102
#define IDD_ABOUTBOX                    103
#define IDM_ABOUT                       104
#define IDM_EXIT                        105
#define IDI_STANDARD                    107
#define IDI_SMALL                       108
#define IDC_STANDARD                    109
#define IDC_MYICON                      2
#ifndef IDC_STATIC
#define IDC_STATIC                      -1
#endif
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NO_MFC                     130
#define _APS_NEXT_RESOURCE_VALUE        129
#define _APS_NEXT_COMMAND_VALUE         32771
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           110
#endif
#endif

//stdafx.cpp ////////////////////////////////////////////
#include "stdafx.h"

//Standard.cpp //////////////////////////////////////////
```

```
#include "stdafx.h"
#include "Standard.h"
#define MAX_LOADSTRING 100
HINSTANCE hInst;
TCHAR szTitle[MAX_LOADSTRING];
TCHAR szWindowClass[MAX_LOADSTRING];

ATOM                MyRegisterClass(HINSTANCE hInstance);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR    lpCmdLine,
                       int       nCmdShow)
{
        UNREFERENCED_PARAMETER(hPrevInstance);
        UNREFERENCED_PARAMETER(lpCmdLine);
        MSG msg;
        HACCEL hAccelTable;
        LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
        LoadString(hInstance, IDC_STANDARD, szWindowClass, MAX_LOADSTRING);
        MyRegisterClass(hInstance);
        if (!InitInstance (hInstance, nCmdShow))
        {
                return FALSE;
        }
        hAccelTable=LoadAccelerators(hInstance,MAKEINTRESOURCE(IDC_STANDARD));
        while (GetMessage(&msg, NULL, 0, 0))
        {
                if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
                {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }
        return (int)msg.wParam;
}

ATOM MyRegisterClass(HINSTANCE hInstance)
{
        WNDCLASSEX wcex;
        wcex.cbSize    = sizeof(WNDCLASSEX);
        wcex.style             = CS_HREDRAW | CS_VREDRAW;
```

```
        wcex.lpfnWndProc      = WndProc;
        wcex.cbClsExtra       = 0;
        wcex.cbWndExtra       = 0;
        wcex.hInstance= hInstance;
        wcex.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_STANDARD));
        wcex.hCursor   = LoadCursor(NULL, IDC_ARROW);
        wcex.hbrBackground    = (HBRUSH)(COLOR_WINDOW+1);
        wcex.lpszMenuName     = MAKEINTRESOURCE(IDC_STANDARD);
        wcex.lpszClassName    = szWindowClass;
        wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));
        return RegisterClassEx(&wcex);
}

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;
   hInst = hInstance;
   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
   if (!hWnd)
   {
      return FALSE;
   }
   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);
   return TRUE;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        int wmId, wmEvent;
        PAINTSTRUCT ps;
        HDC hdc;
        switch (message)
        {
        case WM_COMMAND:
                wmId    = LOWORD(wParam); // эквивалентно (wParam & 0xffff)
                wmEvent = HIWORD(wParam); // эквивалентно (wParam >> 16)
                switch (wmId)
                {
                case IDM_ABOUT:
                DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
                break;
                case IDM_EXIT: DestroyWindow(hWnd); break;
                default: return DefWindowProc(hWnd, message, wParam, lParam);
```

```
            }
            break;
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            // Здесь добавляем код для вывода в окно
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY: PostQuitMessage(0); break;
        default: return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}

INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
        UNREFERENCED_PARAMETER(lParam);
        switch (message)
        {
        case WM_INITDIALOG: return (INT_PTR)TRUE;
        case WM_COMMAND:
            if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
            {
            EndDialog(hDlg, LOWORD(wParam));
            return (INT_PTR)TRUE;
            }
            break;
        }
        return (INT_PTR)FALSE;
}
```

Некоторые комментарии к стандартной заготовке

Мастер создает четыре файла включений к стандартной заготовке, а также два файла реализации. Это сделано в целях сокращения времени компиляции приложения. Дело в том, что при внесении изменений в проект происходит лишь частичная компиляция измененных файлов. Поэтому основные библиотеки подключаются файлом `stdafx.cpp`. В файлах включений `stdafx.h` и `targetver.h` размещены основные определения и ключи компиляции, а в файле `resource.h` — определения символических констант.

Рассмотрим подробнее файл `standard.cpp`.

Следует обратить внимание на первые две строки головной функции `_tWinMain()`:

```
UNREFERENCED_PARAMETER(hPrevInstance);
UNREFERENCED_PARAMETER(lpCmdLine);
```

Эти определения служат лишь указанием компилятору, что параметры `hPrevInstance` и `lpCmdLine` не используются и не стоит обращать на них внимания.

Текстовые строки с именем окна и класса окна, совпадающие по умолчанию с именем проекта, размещаются в ресурсе приложения и загружаются функцией `LoadString()`. Для редактирования ресурсов приложения используется редактор ресурсов, о нем мы поговорим позднее. Сейчас лишь отметим то, что, при загрузке приложения в память ресурсы загружаются после кода и могут быть извлечены при помощи соответствующего набора функций.

В функции `MyRegisterClass()` после заполнения полей структуры `WNDCLASSEX` происходит регистрация класса окна `RegisterClassEx()`.

Макрос `MAKEINTRESOURCE()`:

```
#define MAKEINTRESOURCEA(i) ((LPSTR)((ULONG_PTR)((WORD)(i))))
#define MAKEINTRESOURCEW(i) ((LPWSTR)((ULONG_PTR)((WORD)(i))))
#ifdef UNICODE
#define MAKEINTRESOURCE  MAKEINTRESOURCEW
#else
#define MAKEINTRESOURCE  MAKEINTRESOURCEA
#endif // !UNICODE
```

используется для приведения идентификатора ресурса к типу, необходимому функции `LoadIcon()`.

Обратите внимание, что здесь используются расширенные версии структуры класса окна и API-функций, на это указывает суффикс "`Ex`", они отличаются от исходных версий лишь дополнительным полем, которое заполняется в данном случае по умолчанию.

Создание окна выделено в отдельную функцию `InitInstance()`. Обратите внимание, что дескриптор приложения сохранили на глобальном уровне в переменной `hInst`. Окно создается так же, как и в листинге 1.1, но с проверкой — удачно ли оно создано? Если обращение к функции `CreateWindow()` завершилось неудачей, возвращается 0 и работа приложения будет завершена. Отображается окно функцией `ShowWindow()` с текущим режимом отображения `nCmdShow`.

Здесь появилась функция `UpdateWindow()`, которая проверяет — прорисовалось ли окно? Если окно отображено, функция ничего не делает, иначе функция ждет, пока окно не будет прорисовано. В большинстве случаев можно обойтись без этой функции.

После того как окно будет отображено, происходит загрузка таблицы клавиш-акселераторов "горячих клавиш", которые создаются в редакторе ресурсов для быстрого обращения к меню приложения. Загрузка их из ресурса приложения осуществляется функцией `LoadAccelerators()`.

В цикле обработки сообщений появилась строка:

```
if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg)),
```

которая проверяет, не является ли сообщение результатом нажатия на "горячую клавишу"? Если это так, происходит генерация сообщения `WM_COMMAND`, как и для соответствующего пункта меню, иначе обработка сообщения происходит стандартным способом.

Обратите также внимание, что прекращение работы приложения происходит с возвращаемым значением `(int)msg.wParam`. Если приложение завершается обычным образом после сообщения `WM_QUIT`, то это `0`, однако здесь появляется возможность изменить код возвращаемого значения при аварийном завершении приложения.

Рассмотрим сейчас оконную функцию `WndProc()`. Две переменные `wmId` и `wmEvent` типа `int` предназначены для хранения дополнительной информации из младшего и старшего слова `wParam`. Для извлечения их значений используются макросы `LOWORD()` и `HIWORD()`. О том, как обрабатываются сообщения при выборе пункта меню, мы поговорим позднее.

Еще две переменные:

```
PAINTSTRUCT ps;
HDC hdc;
```

необходимы для вывода в окно при обработке сообщения `WM_PAINT`. Вывод в окно мы обсудим при рассмотрении следующей задачи в листинге 1.3.

Функция `About()` нужна для обработки пункта меню "**О программе**". Оставим обсуждение этой функции до *главы 3*.

Обработка сообщений

Операционная система способна генерировать сообщения с номерами до 0x400, номера же от 0x400 и далее зарезервированы для пользовательских сообщений. В файле включений `winuser.h` размещены макроимена системных сообщений. Этот файл вызывается неявно через файл `windows.h`. Рассмотрим технику обработки наиболее распространенных сообщений Windows.

Нажатие клавиши

При нажатии любой алфавитно-цифровой клавиши на клавиатуре вырабатывается сообщение `WM_CHAR`.

> ***Примечание***
>
> На самом деле генерируются сообщения о нажатии и отпускании клавиши `WM_KEYDOWN`, `WM_KEYUP` и лишь затем `WM_CHAR`.

Чтобы обработать это сообщение, необходимо добавить в переключатель оконной функции еще одну строку альтернативы:

```
case WM_CHAR:
```

и описать необходимые действия для обработки нажатия клавиши.

Поставим самую простую задачу — при нажатии на клавишу выводить текущий символ в окне приложения, т. е. обеспечить эхо-печать в одну строку, пока не беспокоясь о выходе строки за границу окна.

Для решения этой задачи воспользуемся шаблонным классом `basic_string<>` и создадим класс `String`, который будет работать как с С-строкой, так и со строкой Unicode. Для этого в качестве типа данных используем `TCHAR`.

Примечание

В библиотеке *STL* (Standard Template Library) описаны классы `string` и `wstring`, которые являются реализацией базового класса `basic_string<>` для типов `char` и `wchar_t` соответственно. В Visual Studio 2010 этот класс размещен в файле `xstring` и принадлежит стандартному пространству имен `std`, как и вся библиотека STL.

Рассмотрим в листинге 1.3 оконную функцию задачи, удалив для простоты обработку сообщений меню.

Листинг 1.3. Программа эхо-печати

```
#include <xstring>
typedef std::basic_string<TCHAR, std::char_traits<TCHAR>,
std::allocator<TCHAR> > String;
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lPa-
ram)
{
        PAINTSTRUCT ps;
        HDC hdc;
        static String str;
        switch (message)
        {
        case WM_CHAR:
                str += (TCHAR)wParam;
                InvalidateRect(hWnd, NULL, TRUE);
                break;
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                TextOut(hdc, 0, 0, str.data(), str.size());
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY: PostQuitMessage(0); break;
        default: return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
```

Для ускорения компиляции не будем подключать всю область стандартных имен и явно укажем область видимости "`std::`" для всех переменных из этой области.

У нас имеется две возможности описания переменной `str` — либо на глобальном уровне, либо в статической области памяти оконной функции.

```
static String str;
```

Дело в том, что после ввода очередного символа оконная функция теряет управление и, если это будет автоматическая переменная, созданная на стеке, она потеряет свое значение после выхода из функции.

При обработке сообщения `WM_CHAR` извлекаем очередной символ из младшего слова `wParam` и добавляем к строке перегруженным оператором "`+=`".

Теперь нужно вывести символ в окно. Можно это сделать здесь же, но так поступать не принято. Весь вывод обычно стараются осуществлять при обработке сообщения `WM_PAINT`. Дело в том, что инициатором перерисовки окна может быть не только само приложение, но и операционная система. Поскольку Windows является системой многозадачной, то окно приложения может быть полностью или частично перекрыто окном другого приложения, или окном системной утилиты. В этом случае возникает необходимость восстановления окна либо его части. Операционная система Windows решает эту задачу, объявляя *"недействительным прямоугольником"* либо все *окно*, либо его часть. Такое объявление автоматически приводит к генерации сообщения `WM_PAINT` для этого окна.

Если мы будем осуществлять вывод в окно при обработке любого другого сообщения, то потеряем его при перерисовке окна, инициированного системой. То же самое произойдет при "сворачивании" и "распахивании" окна.

> ***Примечание***
>
> В Windows принято, что за содержимое окна несет ответственность приложение, его создавшее. Операционная система может лишь послать сообщение о необходимости перерисовки окна или его части.

Когда необходимо перерисовать окно, его объявляют недействительным. Для этого имеется функция `InvalidateRect()`:

```
BOOL WINAPI InvalidateRect(HWND hWnd, CONST RECT *lpRect, BOOL bErase),
```

которая объявляет недействительный прямоугольник `*lpRect` в окне `hWnd`.

Воспользуемся этим приемом, указывая вторым параметром `NULL`, что приведет к перерисовке всего окна. Значение `TRUE` третьего параметра является указанием перерисовать фон окна.

Теперь рассмотрим вывод строки в окно приложения в сообщении `WM_PAINT`. Для этого необходимо получить *контекст устройства*. В Windows все функции, выводящие что-либо в окно, используют в качестве параметра *дескриптор контекста устройства* `hdc`, который представляет собой структуру, описывающую свойства данного устройства вывода. В оконной функции опишем эту переменную:

```
HDC hdc;
```

В обработчике сообщения `WM_PAINT` вызовом функции `BeginPaint`() получим `hdc`:

```
HDC WINAPI BeginPaint(HWND hWnd, LPPAINTSTRUCT lpPaint);
```

Вся необходимая информация для перерисовки окна будет представлена в структуре `PAINTSTRUCT`:

```
struct PAINTSTRUCT {
    HDC         hdc;  //Контекст устройства
    BOOL        fErase;      //Если TRUE — фон окна перерисовывается
    RECT        rcPaint;     //Недействительный прямоугольник
    BOOL        fRestore;    //Резерв
```

```
    BOOL        fIncUpdate;  //Резерв
    BYTE        rgbReserved[32]; //Резерв
};
```

Теперь можно выводить текст в окно с помощью функции `TextOut()`:

```
BOOL WINAPI TextOutW(HDC hdc, int x, int y, LPCWSTR str, int len);
```

которая принимает в качестве параметров контекст устройства `hdc`, (*x*,*y*) — координаты начала вывода текста, указатель на символьный массив `str` и длину выводимой строки `len`. Среди GDI-функций нет функции, выводящей отдельный символ, поэтому мы будем выводить всю строку полностью с начальной позиции, чтобы не вычислять каждый раз позицию "нового" символа. Функция `TextOut()` не требует строкового типа, ей достаточно указателя первого символа массива, поскольку следующим параметром задается количество выводимых символов, поэтому мы можем воспользоваться методами класса `String` и получить требуемый указатель массива символов и размер строки:

```
TextOut(hdc, 0, 0, str.data(), str.size());
```

Вывод осуществляется с начала окна (`0,0`). По умолчанию система координат имеет начало в левом верхнем углу клиентской области окна (т. е. внутри рамки, ниже заголовка и строки меню), ось *x* направлена по горизонтали вправо, ось *y* — вниз (рис. 1.10). Одна логическая единица равна 1 пикселу.

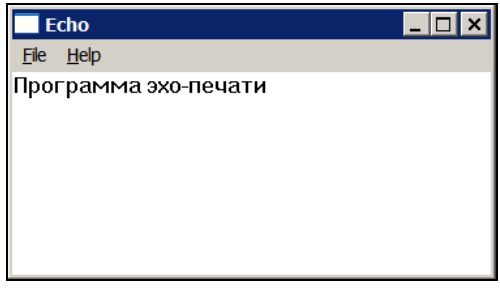

Рис. 1.10. Вид окна программы эхо-печати

Завершает обработчик сообщения функция `EndPaint()`:

```
BOOL WINAPI EndPaint(HWND hWnd, CONST PAINTSTRUCT *lpPaint),
```

которая обеспечивает освобождение контекста устройства. Необходимо учитывать, что контекст устройства является критически важным ресурсом операционной системы, и, после того как необходимость в данном контексте отпадет, его нужно уничтожить, т. е. освободить этот ресурс.

> ***Примечание***
>
> Обычно Windows-приложения работают с "*общим контекстом экрана*", который и создается в стандартной заготовке. Однако приложение может создавать и "*частные контексты экрана*", которые существуют все время жизни приложения. Для этого класс окна должен быть зарегистрирован со стилем `CS_OWNDC`. Однако это приводит к неэффективному расходованию оперативной памяти.